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 environment for solving partial differential equations using the finite element method. We present the novel and modern design principles that underpin the DOLFINx library, and describe approaches used in DOLFINx that preserve the high level of mathematical abstraction associated with FEniCS Project libraries, yet support extensibility and specialized customization. At the core of DOLFINx is a data- and function-oriented design, in contrast with the object-oriented design of more traditional libraries. We argue that this novel design approach leads to a compact and maintainable library, which is flexible in use and makes possible the creation of high performance programs in different languages.

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

1 INTRODUCTION

The finite element method emerged in the 1950s, driven by a need for accurately and effectively solving structural mechanics problems originating from aeronautical engineering and the aircraft industry [5, 24, 27, 53, 100]. As the method evolved, largely in tandem with the computer itself, the development of finite element software dates back

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, chris@ieef.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.

nearly just as far. The 1960-70s saw the advent of general-purpose finite element software [15, 41, 49, 70, 96, 98], with expanding functionality, including handling time-dependent and non-linear problems, adaptivity, enhanced user interfaces and input/output-integration, in the 1980s [65, 97].

As the application domain of finite element methods grew into nearly all areas of engineering and natural sciences, as well as the life sciences, by the 1990s a new paradigm developed. Object-oriented, general-purpose finite element libraries such as Diffpack and deal.II [6, 12, 20, 63] were designed, supporting the development and implementation of simulators for entirely new classes of problems. These libraries typically strongly and purposefully rely on class hierarchies, templates and operator overloading for representing finite element concepts, such as meshes, finite element spaces, degrees of freedom, and sparsity patterns. The main (far-from-trivial) task left to the user is to implement the assembly of element tensors, while taking advantage of predefined basis functions and numerical linear algebra algorithms.

The quest for even higher levels of abstraction, generality and automation without loss of computational efficiency led to another revolution. By combining high-level software abstractions, encapsulating as well as mimicking mathematical objects, with lower-level code generation or other forms of preprocessing, new ecosystems of finite element software formed and prospered e.g. AceGen, Feel++, NGSolve and Firedrake [2, 3, 39, 47, 60, 61, 67–69, 81, 82, 90]. In particular, the FEniCS Project [2] (FEniCS) pioneered a software pipeline consisting of a domain-specific language for defining variational forms (Unified Form Language (UFL)) [3], a finite element form compiler for generating low-level finite element code (FEniCS Form Compiler (FFC)) [68] and an automated finite element problem solving environment and library (Dynamic Object-oriented Library for FINite element computation (DOLFIN)) [67]. These new levels of abstraction enabled advanced higher-order features such as, for example, automated adjoints and derivatives of finite element models [37, 74], automated error control and adaptivity [87], shape optimization [33, 44, 89], uncertainty quantification [102], reduced order modeling [11], and hybrid finite element and neural network models [75], to mention but a few.

FEniCS has made a significant impact across fields in engineering and the applied sciences where PDEs are a prominent modeling paradigm. We can point to examples in geophysics [101], biomechanics [46], biomedicine [29], structural mechanics [43, 85], fluid mechanics [76], inverse problems [79] and optimization [32]. In addition, FEniCS has been used as a foundational tool for developing packages for new numerical approaches, e.g. fictitious-domain finite element methods [21, 34], hybrid particle-mesh methods [71] as well as for the numerical verification of results from mathematical analysis of new discretization techniques [28, 52, 95] and preconditioners [18].

Despite these successes, a number of challenges became evident with the design and continued development of the FEniCS components:

- *Age.* Parts of the FEniCS software were approaching ten or even twenty years old. Previous design decisions were based on the prevalent computer science and software engineering paradigms and technologies of the time, which have limitations that later became apparent.
- *Maintainability.* The FEniCS pipeline, and DOLFIN in particular, were becoming increasingly complex and cumbersome to maintain due to their large code base, which included both core and non-core features, and design limitations.
- *Extensibility*. Overly encapsulated data storage and a lack of fine-grained control made it difficult for developers and users to extend FEniCS non-intrusively to new problem settings and methods when the available abstractions were not sufficient. This also presented barriers to experimentation on new hardware platforms, e.g. GPUs.

- *Performance*. Parallelism was not pervasive when DOLFIN was created, and support for parallel computing was retrofitted. Although most algorithms were designed for parallel computation, a significant number were not or were not optimal, resulting in an inconsistent user experience when computing in parallel and hindering performance on the latest generation of high performance computers.

Here, we present the finite element library and problem-solving environment DOLFINx, the guiding design principles and how these manifest in the library design, and its context in the revised FEniCS ecosystem (FEniCSx). The main contributions of DOLFINx are:

- DOLFINx is a ground-up rewrite of DOLFIN using modern C++ and modern, idiomatic Python. The architecture is fundamentally data-oriented and functional, rather than relying on object hierarchies and polymorphism.
- The core features of DOLFINx are focused on automatically solving PDEs using advanced finite element discretization techniques on unstructured meshes. Our focus on achieving a relatively stable public application programming interface (API) means that DOLFINx can be used as a basis for experimental or research work without requiring intrusive extension of the core library with possibly immature and experimental technologies.
- DOLFINx integrates seamlessly with modern Python tools such as NumPy and Numba [62] for implementing finite element cell kernels, creating meshes and finite element assembly algorithms, without loss of performance over, e.g., C++ implementations of the same functionality.
- All algorithms in DOLFINx are designed with massively parallel computations in mind, and we no longer accept contributions that do not meet this design goal.

The remainder of this paper is structured as follows. We discuss the underlying design principles of the DOLFINx library in section 2, followed by a high-level overview of DOLFINx and the class of problem that it targets in section 3. In section 4, we discuss some key design features that support efficient parallel computation. In section 5, we describe how finite elements are defined in DOLFINx using the FEniCS library Basix, and how user-defined custom elements can be created. In section 6, we look at how meshes are represented and accessed in DOLFINx, with a focus in distributed parallel storage. Section 7 looks at how function spaces are defined in DOLFINx and how interpolation into arbitrary spaces can be performed. In section 8, we discuss assembly kernels, including those generated by FEniCSx Form Compiler (FFCx) and user-defined custom kernels and the creation of user assembly functions. Section 9 describes linear algebra interfaces, and looks at how DOLFINx can be used alongside a range of external linear algebra libraries without intrusion. We provide concluding remarks and pointers to further work and challenges in section 10.

2 DESIGN PRINCIPLES

The design of DOLFINx differs fundamentally from its predecessor DOLFIN in following a data-oriented and functional design, and a more modular approach that allows for lower-level and fine-grained control. This is complemented by higher-level interfaces, which characterized DOLFIN, being constructed from lower-level, user-accessible interfaces in DOLFINx. The design principles are informed by the objectives of high-performance, flexibility and usability, support for modern computer hardware (e.g. GPUs), language interoperability, and user and developer experiences of the legacy DOLFIN library. DOLFINx is distributed memory parallel by design, with a focus on scalability and consistent functionality in serial and parallel.

2.1 Data-oriented design

The design of DOLFINx follows a data-oriented approach, complemented by functional approaches, rather than a heavily object-oriented design. Data encapsulation is used where appropriate, but not dogmatically. This design

approach supports custom, low-level operations without use of object-oriented patterns such as inheritance and polymorphism. 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. Moreover, when exposing data the layout of data can also be programmatically described. A data-oriented approach lends itself to extensions to different hardware technologies, e.g. GPUs, without intrusive changes, since the underlying data can be accessed and operated on. Also, a data-oriented approach eases interoperability across languages; data is common whereas language-specific object-oriented features are not. We highlight in later sections examples of the data-oriented design.

2.2 Functional-style design

The data-oriented design is partnered with functional programming techniques in DOLFINx. Where possible, the design favors pure functions. This aligns with the mathematical structure that we aim to follow, and pure functions are simpler for users and developers to reason with in general, and have particular benefits in parallel. In many cases, modern C++ allows the use of pure functions without any performance penalty over a traditional design where functions modify their arguments. We avoid polymorphism in DOLFINx by working with, and passing, functions that act on data. Where in a traditional C++ library user-implemented functionality would typically be provided using polymorphism and implementing virtual functions, in DOLFINx a function would be passed. Both modern C++ and Python provide simple, native support for passing functions, including with captures. The resulting code is simpler, flexible, efficient and works naturally across languages. Examples of the functional design are presented in later sections.

2.3 Minimal code generation

DOLFINx leverages domain-specific code generation for specific operations where there is a compelling user benefit to do so, namely for the generation of finite element kernels for user-specified problems, and for the minimal ancillary information required to support this. In all other respects 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 than DOLFINx. Despite the success of DOLFIN and its extensive user base, ultimately an expanding reliance on code generation led to increased library complexity, reduced maintainability, slowed the rate at which new features and performance improvements could be made, and limited extensibility and customizability. The DOLFINx design overcomes these issues without loss of the most appealing features of DOLFIN.

2.4 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; programmatically described data arrays that define library objects can be accessed and used. An example of this is research into finite element assembly operations on GPUs. The necessary mesh and degree of freedom (DOF) map data can be accessed and used in computations outside of the DOLFINx library and in different languages and frameworks.

2.5 Language interoperability

The core of DOLFINx is a library with a C++20 [26] interface. It is designed, however, to support inter-language interoperability, with the data-oriented and functional design lending itself to this interoperability. The Python interface

is developed in Python and with interfaces to the C++ library (generated using pybind11 [55]). Despite the C++ core, the Python interface is idiomatic. While most users choose to use the Python interface, the C++ interface is feature complete, and with modern C++ features the syntax is similar to the Python equivalent.

Many of the underlying data structures used in DOLFINx are based on contiguous arrays, described by shape and sometimes strides, and this is exposed. This allows data to be passed in a straightforward way, without copy where possible, between languages and for the data to wrapped or owned by a suitable data structure in a given language, e.g. NumPy arrays in Python, or `std::span` or `std::mdspan` in C++23. Use of pure functions eliminates most memory management issues when passing data into the C++ library, and the object lifetime support in pybind11 supports memory management for data shared into the Python layer.

Building on the functional design, functions can be passed through the Python/C++ language barrier. Examples of use include passing a user-provided graph partitioning function for computing the parallel distribution of a mesh, a function for evaluating a mathematical expression at points for interpolating an expression in a finite element space, and finite element kernel functions. Performance limitations of a language like Python for certain operations are overcome by the provided functions performing vectorized operations using NumPy [45] or JIT compiled functions with Numba [62]. In other cases tools such as Numba [62] and the C Foreign Function Interface (CFFI) can be used to compile functions with C signatures, with function pointers passed into the C++ library.

2.6 Fine-grained control

The DOLFIN library provided mathematically expressive high-level interfaces which were well-suited to problems with well-matched abstractions. However, the high-level interfaces were not constructed/composed from lower-level, fine-grained interfaces. This could make application to problems that did not match the high-level interfaces challenging, and the implicit nature of high-level interfaces could hide performance issues in some cases. Users without an understanding of the details of the underlying implementation could write outwardly reasonable code that could be slow. Common examples include the creation and destruction of non-trivial objects within a time loop that could be re-used (e.g. sparse matrices), the pre-processing of variational forms for extraction from a JIT cache inside a loop, and increases in memory usage due to the caching of large objects that might not be re-used. 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 opaque caching objects should be explicitly controlled by the user. The approach is supported by the functional design of DOLFINx with pure functions preferred over class methods.

To preserve high-level interfaces that characterized DOLFIN and which in many cases enhanced usability and accessibility, DOLFINx composes a high-level interface from its granular lower-level interface. This preserves some attractive and easy-to-use features while permitting lower-level customizations that were difficult with DOLFIN. Moreover, high-level and explicit interfaces in DOLFINx are designed to avoid opaque performance pitfalls. Interfaces that were frequently used in ways that led to poor performance have been removed.

3 DOLFINX OVERVIEW

DOLFINx is a library for solving partial differential equations (PDEs) using the finite element method and mirrors many of the mathematical abstractions that define the finite element method. Here we provide a short synopsis of finite element concepts that are reflected in the DOLFINx design.

A large class of finite element problems can be formulated as follows. Given a mesh $\mathcal{T}$ of a domain, finite element spaces U and V defined on the mesh, and a functional $F : U \times V \to \mathbb{C}$: find $u \in U$ such that

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

As an example, for the Helmholtz equation F is defined as

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

where k is the wave number, f is a prescribed function and $\bar{v}$ denotes the complex conjugate of v. It is typical to set $U = V$, and for problems that are linear in u, such as the Helmholtz equation, to phrase the problem as: find $u \in U$ such that

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

where for the Helmholtz equation

$$a(u, v) := \int_{\mathcal{T}} \nabla u \cdot \nabla \bar{v} - k^2 u \bar{v} \, \mathrm{d}x, \tag{1}$$

is referred to as the *bilinear form*, and

$$L(v) := \int_{\mathcal{T}} f \bar{v} \, \mathrm{d}x,$$

is referred to as the *linear form*.

To provide an early sense of DOLFINx, we present in figure 1 a complete solver in Python for the Helmholtz equation on a unit cube mesh with tetrahedral cells using a degree 3 Lagrange finite element space.

3.1 Components

FEniCSx is composed of four main libraries: UFL is a domain specific language that can be used to express finite element forms; Basix is a library for constructing finite elements; FFCx generates fast element-level C kernels from UFL forms; and DOLFINx, the largest component, manages finite element meshes, assembly over meshes, linear algebra data structures and solvers, 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 are summarized in figure 2.

Figure 3 shows how the components of FEniCSx depend on each other for a typical user application code developed in Python. The user defines elements with Basix and writes their form with UFL. When assembly begins, `dolfinx.jit` passes this form to FFCx to generate code that can assemble it. Alternatively, users can develop their application code in C++: a typical workflow in this case is shown in figure 4. In our experience this has advantages, for example when running on a high performance computer (HPC) cluster. In this case, FFCx can be used to generate the assembly kernels ahead-of-time for use in a C++ code.

DOLFINx is designed such that it can be used without FFCx generated code. Element-level kernels can be programmed in C or C++ for use via the C++ or Python interfaces of DOLFINx, or programmed using the Python-based JIT compilation library Numba for high-performance kernels in a Python environment. A workflow using Numba is illustrated in figure 5. This is discussed in section 8.3.

```python
from mpi4py import MPI

import numpy as np

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 = SpatialCoordinate(msh)
k = 4 * np.pi
f = (1.0 + 1.0j) * k**2 * cos(k * x[0]) * cos(k * x[1])
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, bcs=[], petsc_options={"ksp_type": "preonly", "pc_type": "lu"})
uh = problem.solve()

# Save solution in VTX (.bp) format
with io.VTXWriter(msh.comm, "helmholtz.bp", [uh], engine="BP4") as vtx:
    vtx.write(0.0)
```

Fig. 1. DOLFINx solver for the Helmholtz problem on a unit cube with homogeneous Neumann boundary conditions.

3.2 Summary of DOLFINx features

We briefly summarize some of the main features of DOLFINx. We expand on some of these features in the following sections.

- Arbitrary degree finite elements on interval, triangle, quadrilateral, tetrahedral and hexahedral cells, including unstructured meshes without special ordering;
- Scalable, distributed meshes;
- Meshes with flat or curved cells;
- Parallel I/O;
- Code generation can be used to generate finite element kernels from forms written using UFL;
- Assembly and solvers using different floating point scalar types;
- Assembly of custom element kernels written using Numba;
- Interpolation of functions into arbitrary function spaces;
- Interpolation between function spaces built on different (non-matching) meshes, including meshes using non-affine geometry;
- Ability to non-intrusively support different linear algebra backends, e.g., NumPy, Portable, Extensible Toolkit for Scientific Computation (PETSc), Trilinos, and Eigen;
- Assembly into blocked and nested matrices, supporting the efficient and scalable [103] implementation of physics-based block preconditioners [36];

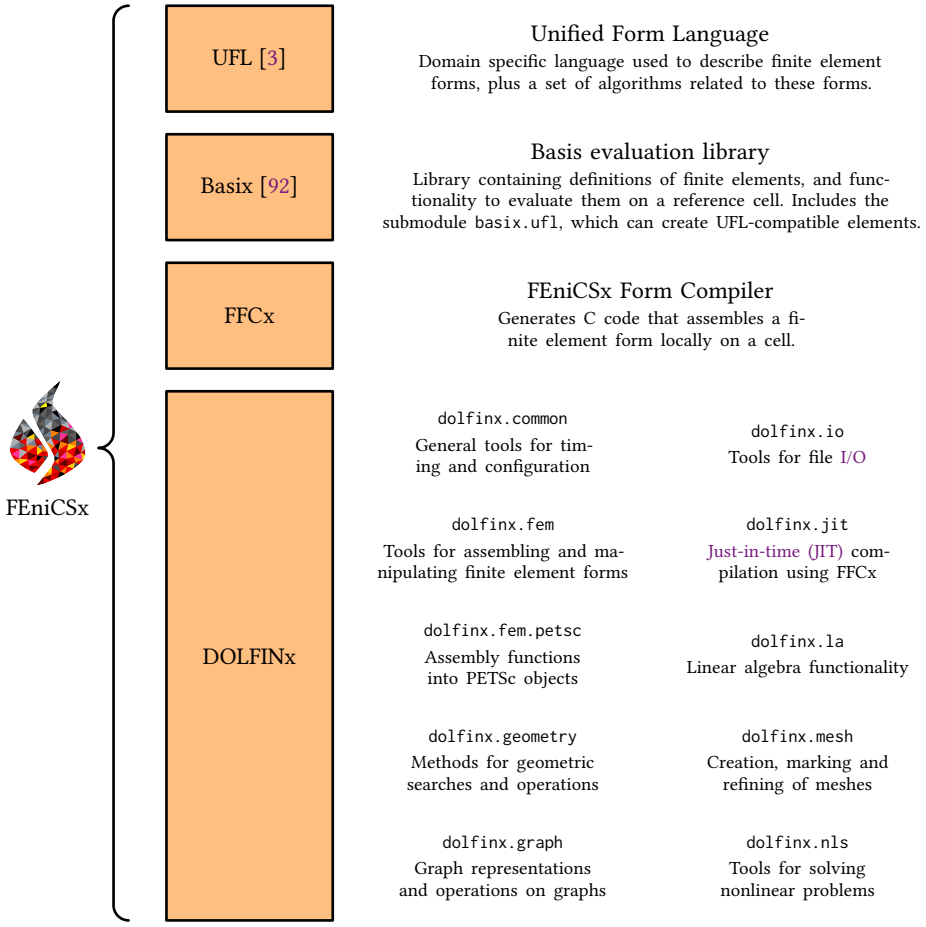

Fig. 2. The packages that make up FEniCSx, and the submodules of DOLFINx.

- User-defined finite elements.

3.3 License, availability and development

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. FEniCSx development takes place at https://github.com/FEniCS. DOLFINx is available as Debian/Ubuntu packages, via the Conda and Spack package managers and in Docker images.

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 pybind11 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.

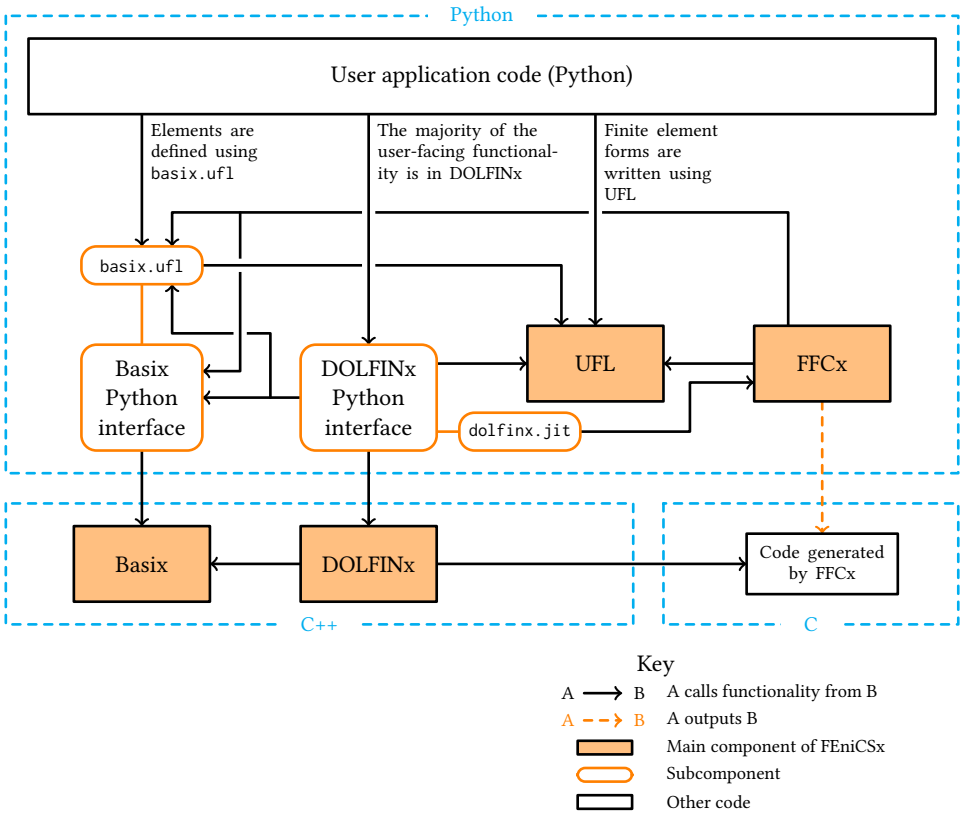

Fig. 3. The interdependence of the core components of FEniCSx (DOLFINx, FFCx, Basix, and UFL) as employed by typical user application code in Python.

4 PARALLELISM

DOLFINx is designed from the outset to support distributed memory parallelism using Message Passing Interface (MPI) and parallel efficiency has been a major consideration in the design. The library has been used to solve problems with more than 1 trillion cells. A full description of parallel design aspects of DOLFINx would require extensive coverage; we choose in this section to discuss two key building blocks for parallel designs that are used throughout DOLFINx: index maps and scatterers. We note that other libraries uses similar concepts, e.g. index sets (`IS`) and vector scatters (`VecScatter`) in PETSc.

MPI-3 neighborhood collectives are used extensively in DOLFINx, and are naturally suited to finite element computations on meshes and for linear algebra operations. Well-partitioned and well-ordered simulation data leads to small communication neighborhoods, with neighborhood sizes independent of the overall size of a problem and the number of MPI processes used in a simulation. In unstructured grid computations, there are sometimes cases where the determination of a communication neighborhood is not straightforward, for example when constructing a mesh from data in a file produced by another program. A common scenario is when a process knows which other process holds data that it requires, but the holding process is not aware of which processes require (some of) its data. In

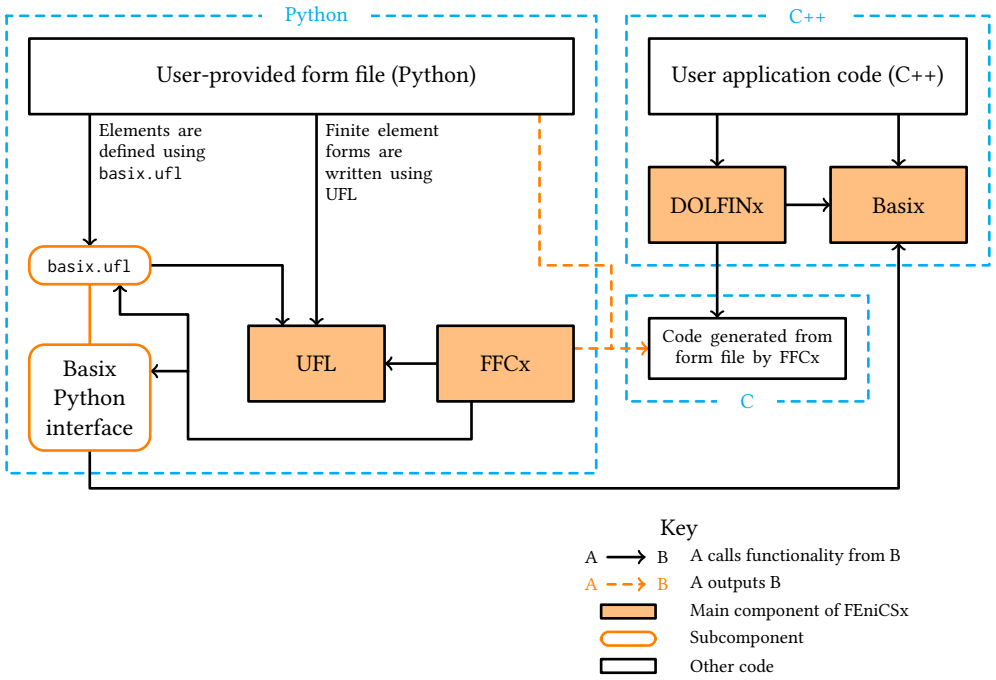

Fig. 4. The interdependence of the core components of FEniCSx (DOLFINx, FFCx, Basix, and UFL) as employed by typical user application code written in C++ and finite element formulations described by UFL in Python. The C kernel is automatically generated from a one-time execution of the Python form file code.

DOLFINx we use the non-blocking NBX consensus algorithm [50] to build neighborhood communication graphs when the communication graphs cannot be built using already known neighborhood information. With the NBX algorithm, DOLFINx does not use any MPI all-to-all communication functions.

4.1 Index maps

A typical design for distributing an object, such as a mesh, across multiple processes is to assign ownership of object entities to processes, e.g. assign ownership of each entry in a vector (distributed array) to a process. In many cases some entities will be stored on more than one process (ghost or halo regions).

The parallel layout of objects in DOLFINx is described by *index maps*, which describe how a range of indices is distributed across parallel processes. Each index is uniquely 'owned' by a single process, although other processes may be aware of the index: indices that a process is aware of but does not own are referred to as ghost indices. Ghost indices may be included because (for example) a vertex not owned by the current process is adjacent to a cell owned by the current process.

In DOLFINx, we have introduced the `IndexMap` class to describe the data layout in parallel. 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 $[i_p, i_p + n_p)$, for some $i_p, n_p \in \mathbb{N}$, $i_p, n_p \leqslant n_g$. The disjoint union of these owned indices across all processes is equal to the entire range of indices $[0, n_g)$. The global index corresponding to the first local index on process p is

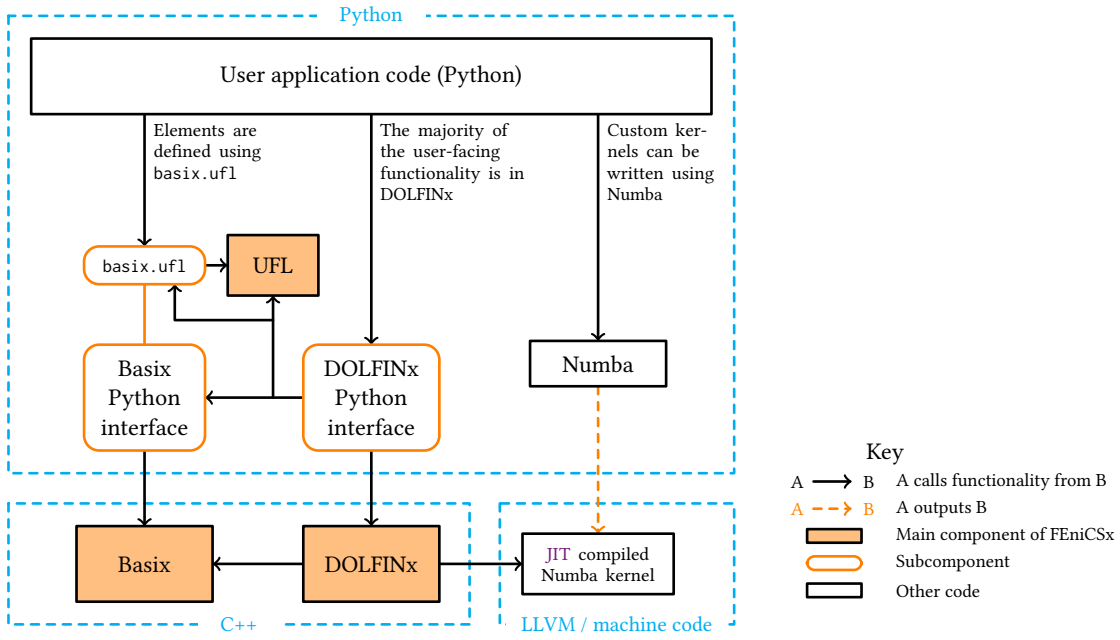

Fig. 5. A possible user workflow when using a custom assembler written in Numba.

determined by the offset i_p. The indices are distributed so that the first partition is owned by process 0, the next partition chunk by process 1, and so on, i.e. $i_p = \sum_{i=0}^{p-1} n_p$. In addition to the owned indices, an `IndexMap` stores a list of ghosts. These ghosts represent the indices that the process needs to be aware of but does not own. For each ghost, we store its global index, the rank of the owning processes and the local index of the ghost on the owning rank. 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, when required. The use of index maps in the partitioning of meshes is discussed in section 6.1.

4.2 Scatterers

Building upon the index maps, DOLFINx uses a `Scatterer` object to manage data communication between processes for distributed objects. A `Scatterer` is created from an `IndexMap` and supports communication of data associated with owned indices to processes that ghost indices (forward scatter) and communications of data associated with ghost indices to process that own the indices (reverse scatter). An `IndexMap` only stores information for the reverse communication pattern (i.e. communication edges to processes that own ghosted indices), a `Scatterer` has communication graphs for both forward and reverse communication patterns.

Communication in finite element solvers is sparse, i.e. each process exchanges data with only a small number of other processes. A DOLFINx `Scatterer` uses MPI neighborhood collectives by default for communication[1], with neighborhood communicators created by a `Scatterer` using the forward and reverse communication graphs. A `Scatterer` also supports

[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.

any required packing and unpacking operations in-and-out of communication buffers. In support of communication between GPUs, a `Scatterer` is templated over a C++ allocator, and also supports the passing of custom pack and unpack functions, allowing host-to-device transfers to be minimized.

5 FINITE ELEMENTS AND BASIX

FEniCSx normally solves problems using finite elements defined by the library Basix [92]. Basix provides a wide range of elements, including $H(\mathrm{div})$- and $H(\mathrm{curl})$-conforming elements, at arbitrary order on different cell shapes and with fine-grained control over element construction, and support for creating user-defined elements that follow the Ciarlet finite element definition. Both cases are discussed in this section.

5.1 Basix supported elements

Basix provides functionality to define a finite element, evaluate basis functions at a set of points, and apply push-forward and pull-back operations to map between reference and physical cells. Basix further provides information to DOLFINx about the layout of DOFs on each cell. Basix is written in C++ and includes a Python interface to its public API. The full list of elements implemented in Basix is shown in table 1. Basix additionally allows the user to provide their own custom element, which we discuss in section 5.2.

For Lagrange and discontinuous polynomial cubical (DPC) elements, the spacing of the points that define the element can be controlled. Typically, schemes for tensor-product elements 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 [48], those defined by centroids [17] and Isaac's method [54]. For elements that are defined using integral moments against either a Lagrange or DPC element, variants can be used to control the polynomials against which moments are taken. If no variant is input, continuous higher-degree Lagrange elements use GL or GLL points and integral moments are taken with Legendre polynomials.

5.2 User-defined finite elements

A feature of Basix is the ability to define custom finite elements through the library's Python or C++ interfaces. Using Python, custom elements can be created using Basix's UFL submodule for use with UFL in the same way as any other UFL element. We demonstrate this by example, but before doing so we introduce the Ciarlet definition of a finite element (definition 5.1), 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 element, usually a polygon or polyhedron;
- $\mathcal{P}$ is a finite dimensional polynomial space on R of dimension n;
- $\mathcal{L} := \{l_0, \ldots, l_{n-1}\}$ is a basis of the dual space $\mathcal{P}^* := \{f : \mathcal{P} \to \mathbb{R} \,|\, f \text{ is linear}\}$. Each functional $l_i \in \mathcal{L}$ is associated with a sub-entity of the cell.

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

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

We associate a local DOF with each functional l_i (or equivalently with each basis function ϕ_i). When defining a function space on a mesh, we associate a global DOF with each local DOF. Local DOFs whose functionals are associated

Element	Supported cells	Supported degrees
Lagrange	interval, triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 0$
Nédélec first kind (N1) [77]	triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 1$
Raviart–Thomas (RT) [83]	triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 1$
Nédélec second kind (N2) [78]	triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 1$
Brezzi–Douglas–Marini (BDM) [19]	triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 1$
Regge [23, 84]	triangle, tetrahedron	$\geqslant 0$
Hellan–Herrmann–Johnson (HHJ) [9]	triangle	$\geqslant 0$
Crouzeix–Raviart (CR) [30]	triangle, tetrahedron	1
discontinuous polynomial cubical (DPC) [8]	quadrilateral, hexahedron	$\geqslant 0$
serendipity [7]	interval, quadrilateral, hexahedron	$\geqslant 1$
bubble [58]	interval, triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 3$ (triangles), $\geqslant 4$ (tetrahedron), $\geqslant 2$ (other cells)
iso [16]	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, [10, 93]).

with sub-entities shared between cells will be assigned the same global DOF number, seen from neighboring cells. This yields the appropriate continuity between cells.

To demonstrate the construction of custom elements, we consider the degree 1 tiniest tensor (TNT) element [25]. The TNT element is defined by:

- $R_{\text{TNT}} := [0,1]^2$ is the unit square. In Basix, the entities of this cell are numbered as shown in figure 6.
- $\mathcal{P}_{\text{TNT}} := \text{span}\{1, x, y, xy, x^2, x^2y, y^2, xy^2\}$.
- $\mathcal{L}_{\text{TNT}} := \{l_0, \ldots, l_7\}$, 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 complete code to create this element in Basix from Python (the element could also be created in C++) is given in figure 7. The polynomial space $\mathcal{P}_{\text{TNT}}$ is defined by a matrix containing coefficients of 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 elements 0 to n 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 points and weights discretely define the functional: the functional can be applied to a function by evaluating the function at the points, multiplying by the weights, then taking the sum. For example, the points and weights for the integral functionals l_4 to l_7 will be a set of quadrature points and weights. Finally, a Python version of the element is created using the function `basix.ufl.create_custom_element`. We can immediately use any of the functionality of Basix with this element and employ it directly with UFL and the code generator FFCx, highlighting extensibility with the ability to generate code for element that are not defined in UFL or FFCx non-intrusively.

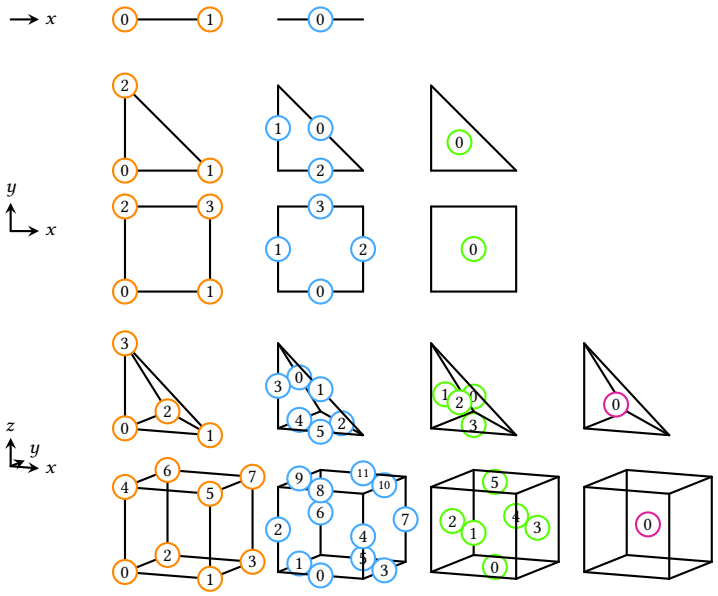

Fig. 6. The numbering of local entities on each cell type. These figures are taken from [91].

6 MESHES

DOLFINx supports distributed, unstructured meshes composed of simplex or tensor-product cells with arbitrary overlaps (support for mixed topology meshes and meshes with multiple geometric map types is under development). The representation of meshes follows a graph-centric approach, as described in [66] for non-distributed meshes (see also [59]), but the underpinning algorithms and data storage structures differ. A full discussion of the data structures, algorithms and parallel treatment would require extensive treatment; we limit our discussion to some main points.

The mesh design follows a strict separation of *topology* and *geometry*, with a `Mesh` consisting simply of the pair (i) mesh `Topology` and (ii) mesh `Geometry` defined on the topology. In its simplest form, a mesh `Topology` holds the cells of a mesh, with a cell defined by its vertices. From this, algorithms are provided that can create and number (in parallel) entities of other topological dimensions (edges and faces, which are also defined by their connected vertices). The numbering of these entities is typically required to create function spaces for high-order finite elements. It is also possible to create 'sub-meshes' (which are full `Mesh` objects) from (subsets) of mesh entities and to define finite element spaces on these sub-meshes. Examples of cases where this is helpful include defining Lagrange multiplier spaces on surfaces (internal or external), defining bounding condition functions that only exist on boundaries, and hybridized finite elements methods with spaces that are defined only on the facets ('skeleton') of a mesh. A mesh `Topology` can also store connectivities other than for an entity to its incident vertices. The connectivity fromgit entities of topological dimension d_0 to entities of d_1 can be computed and stored in a mesh `Topology`.

A mesh `Geometry` describes the geometry of the topological cells declared in a `Topology`. A mesh `Geometry` stores (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) coordinate DOFs, usually the coordinates of the mesh vertices and any 'high-order' geometry points, and (iii) a DOF map that for each cell gives the indices of the coordinate DOFs. `Geometry` is templated over the float type used to represent the mesh geometry.

```python
import numpy as np

import basix
import basix.ufl

# Coefficients defining the polynomial space in terms of orthogonal polynomials on the cell
wcoeffs = np.eye(8, 9)

geometry = basix.geometry(basix.CellType.quadrilateral)
topology = basix.topology(basix.CellType.quadrilateral)

# Points and weights use to define functionals on each sub-entity of the cell
x = [[], [], [], []]
M = [[], [], [], []]

# Associate one point evaluation with each vertex
for v in topology[0]:
    x[0].append(np.array(geometry[v]))
    M[0].append(np.ones([1, 1, 1, 1]))

# Associate an integral with each edge
pts, wts = basix.make_quadrature(basix.CellType.interval, 1)
for e in topology[1]:
    v0 = geometry[e[0]]
    v1 = geometry[e[1]]
    # Map points on the reference interval to each edge of the quadrilateral
    edge_pts = np.array([v0 + p * (v1 - v0) for p in pts])
    x[1].append(edge_pts)

    mat = np.zeros((1, 1, pts.shape[0], 1))
    mat[0, 0, :, 0] = wts
    M[1].append(mat)

# Associate 0 DOFs with the interior of the cell
x[2].append(np.zeros([0, 2]))
M[2].append(np.zeros([0, 1, 0, 1]))

tnt_degree1 = basix.ufl.custom_element(
    basix.CellType.quadrilateral, [], wcoeffs, x, M, 0,
    basix.MapType.identity, basix.SobolevSpace.H1, False, 1, 2)
```

Fig. 7. Creating a degree 1 TNT element in Basix. The polynomial space $\mathcal{P}_{\text{TNT}}$ 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 $\mathcal{L}_{\text{TNT}}$ 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.

6.1 Creating meshes

Distributed meshes can be created via a number of different interfaces at different levels of abstraction. At the lowest level, users can create a `Topology` and a `Geometry` directly, but this is rarely done in practice as all (parallel) pre-processing must be performed by the user; data must already be partitioned and distributed across process, if the input mesh data mixes topological and geometric data (which most mesh generators do) the caller must separate the two concepts and any local reordering for data locality should already be applied. An interface that users commonly call is (C++ version given):

```cpp
fem::CoordinateElement element(.....);
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, 4, 4, 1, 2, 5, 4};
mesh::Mesh mesh = mesh::create_mesh(MPI_COMM_WORLD, cells, element, x, {x.size() / 2, 2});
```

The above example creates a mesh of two bilinear quadrilateral cells. The float type for the mesh geometry is inferred from the type of the `x` array. The argument following `x` is the shape of the logically rectangular coordinate data; a quadrilateral cell mesh could be embedded in two- or three-dimensions. The input cell and coordinate data can be distributed across any of the MPI processes; it could also reside on one process or be distributed across all processes. Internally, a distributed dual graph is constructed and a partitioning of cells across ranks computed, cells are ordered for data locality, and a distributed mesh is created. The interface from Python is very similar:

```
x = np.array([[0.0, 0.0], [1.0, 0.0], [2.0, 0.0],
              [0.0, 1.0], [1.0, 1.0], [2.0, 1.0]], dtype=np.float32)
cells = np.array([[0, 1, 4, 4], [1, 2, 5, 4]], dtype=np.int64)
coordinate_element = basix.ufl.element("Lagrange", "quadrilateral", 1,
                                       shape=(x.shape[1],), gdim=x.shape[1])
msh = mesh.create_mesh(MPI.COMM_WORLD, cells, x, ufl.Mesh(coordinate_element))
```

In the above examples a default graph partitioner, e.g. PT-SCOTCH [22], is called to compute the distribution of cells across processes. However, a user may wish to provide their own cell distribution function. This is made straightforward with the function-oriented design of DOLFINx by passing a user-defined partitioning function:

```
auto part_fn = [](MPI_Comm comm, int nparts, int tdim,
                  std::span<const std::int64_t> cells, std::array<std::size_t, 2> cshape)
                  -> 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, where nodes are
                  // cells and edges are destination ranks.
                  };

mesh::Mesh mesh = mesh::create_mesh(MPI_COMM_WORLD, cells, element, x, {x.size() / 2, 2}, part_fn);
```

For convenience, DOLFINx provides partitioning functions that use PT-SCOTCH, ParMETIS [56] or KaHiP [88], with lambda captures used to specify option parameters for the partitioning libraries. From Python, users can create a cell partitioning function in Python and pass the function to the `create_mesh` function. In the same vein, users can pass re-ordering functions for user-controlled re-ordering. If the user wishes to maintain the input cell distribution across processes a non-callable object can be passed as the partition function to `create_mesh`.

The function `create_mesh` is the backbone of fully-distributed and memory-scalable mesh creation. The DOLFINx file input interfaces in essence read cell and geometry data from parallel file formats, with each read process reading a chunk of the cell topology and geometry, and then call `create_mesh`. For file formats that DOLFINx does not natively support, the user can write code to read the cell and geometry data, and if necessary apply a permutation to the cell data to conform to the DOLFINx ordering, before calling `create_mesh`. It also supports programmatic approaches to distributed mesh generation. For example, distributed meshes can be created in Python using the Gmsh [40] API without intermediate output to disk:

```
import gmsh

from dolfinx.io import gmshio
```

```
4
5  gmsh.initialize()
6
7  model = gmsh.model()
8  sphere = model.occ.addSphere(0, 0, 0, 1, tag=1)
9  model.occ.synchronize()
10 model.add_physical_group(dim=3, tags=[sphere])
11 model.mesh.generate(dim=3)
12
13 # Create a distributed DOLFINx mesh from the Gmsh model/mesh on MPI rank 0, and
14 # any any entity tag data in the Gmsh model
15 msh, celltags, facettags = gmshio.model_to_mesh(model, MPI.COMM_WORLD, rank=0)
```

In the above snippet, the DOLFINx function `model_to_mesh` manages the permutation of cell data from the Gmsh order to the DOLFINx order before calling `create_mesh`.

In addition to interfaces for creating distributed meshes from user mesh data, DOLFINx provides built-in meshes for some simple geometric shapes.

6.2 Arranging and accessing mesh data

Mesh entities of all dimensions are owned by one process and can be ghosted on other processes. Each entity has a local index in the range $[0, n)$, where n is the numbers of entities on the process (owned and ghosts), and a global index that is unique for each entity across all processes. Owned entities are numbered first, followed by ghosted entities that are owned by another process. The layout aligns with the `IndexMap` concept, described in section 4.1. A `Topology` object stores for each created entity type an `IndexMap` that describes how the entities are numbered and distributed. Index map storage is very light as the global index for owned entities is simply the local position (index) plus a process offset. 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. A `Geometry` object has an index map to describe the ownership of the geometry DOF map across processes.

Connectivities (incidence relationships) in DOLFINx follow a graph-centric approach and a natural data structure for storing connectivity data is an adjacency list. We believe that this is such a natural data structure for the DOLFINx mesh design and it conforms to our data-oriented approach that we do not attempt encapsulate the storage format behind member functions/accessors. For the (d_0, d_1) connectivity (entities of topological neighborhood d_1 that are connected to entities of topological neighborhood d_0), each node in the adjacency list corresponds to an entity (by local index) of dimension d_0 and the links (edges) are the connected (incident) entities of dimension d_1.

When executing an operation over entities of a mesh, rather than introducing mesh iterators, we execute over provided ranges of entities. By exposing connectivity data directly as adjacency lists, which are simply a data array and an array of offsets, mesh data can be operated on very efficiently and shared without overhead between libraries, including libraries in different languages. In section 8.3.2 we show an example of how this data-centric design pattern for meshes allows high-performance user assembly functions to be written in Python. The below snippet illustrates how vertex-to-cell connectivity data can be computed and then accessed as data and offset arrays for possible use in other code. An example of where we make frequent use of this approach is preparing unstructured mesh data for execution of experimental code on GPUs.

```
1 mesh::Mesh mesh = mesh::create_mesh(...);
2 int tdim = mesh.topology()->dim();
```

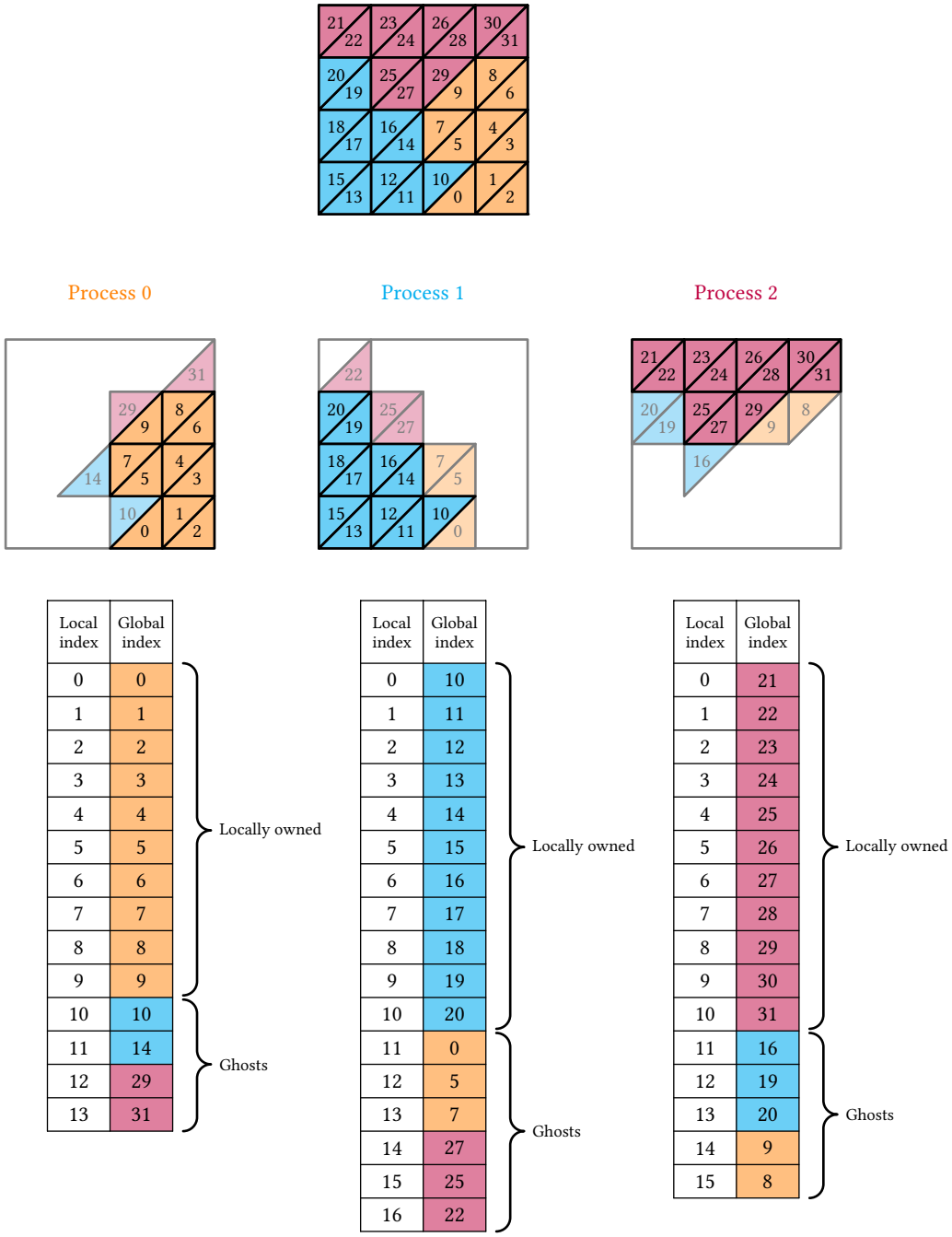

Fig. 8. The index maps for the cells of a mesh of triangles distributed over three processes. Each process owns a set of cells and a set of ghosts of cells. The ghost cells are owned by other processes that are connected to the owned cells by a facet. In each process's index map, the ghosts are stored at the end of the array. Color is added as a visual aid to indicate the process owning each index.

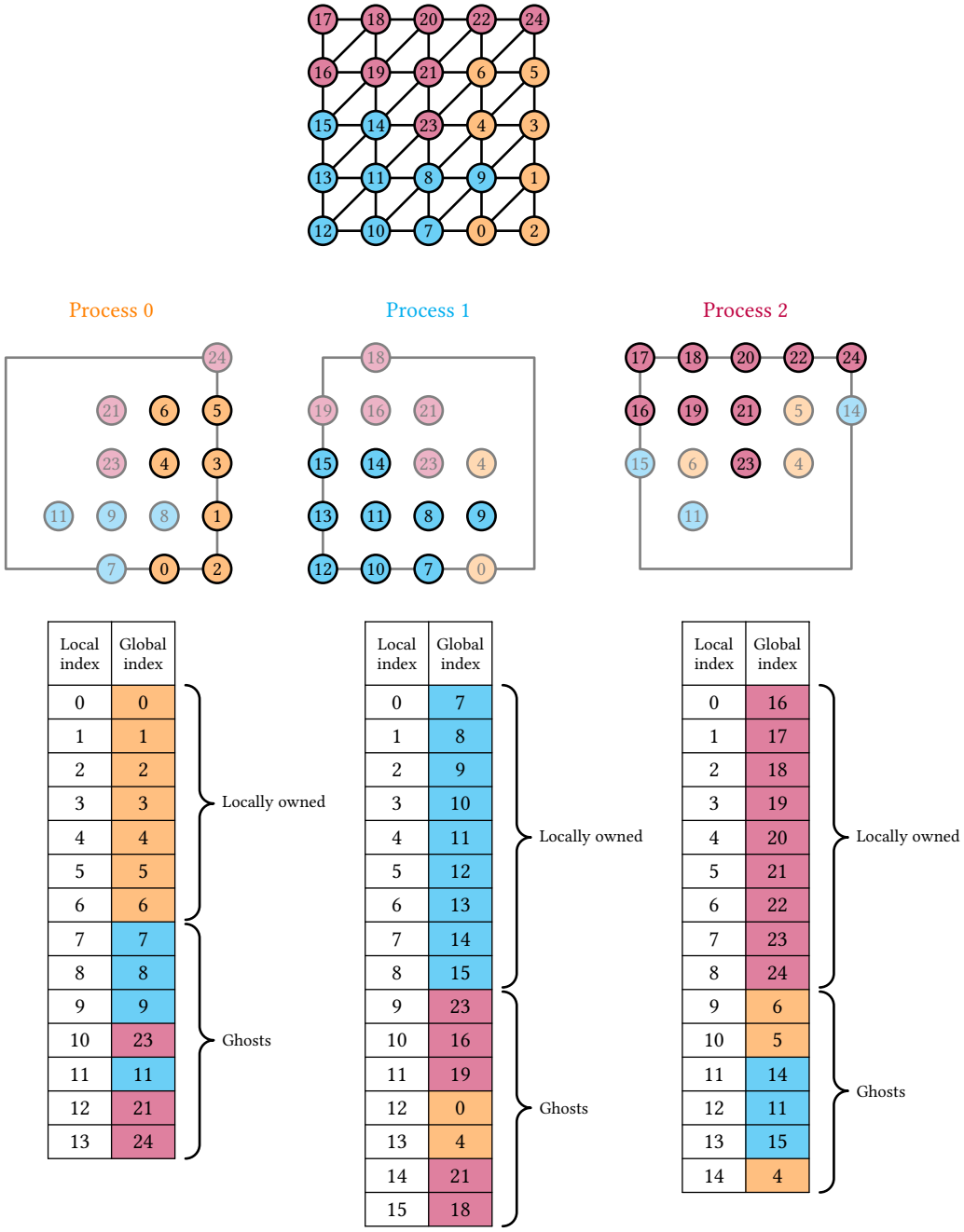

Fig. 9. The index maps for the vertices of the mesh shown in figure 8. Each process is aware of the vertices of all of the cells in its index map in figure 8. Color is added as a visual aid to indicate the process owning each index.

```
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->c_offsets();

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

From Python, the adjacency list arrays are presented as NumPy arrays. Geometry data can be accessed in the same fashion. In fact, when executing a finite element kernel over cells the only data required is (i) the range of cells to execute over, (ii) the mesh geometry DOFs (logically two-dimensional array) and (iii) the geometry DOF map (logically two-dimensional array) that for each cell points to the geometry DOFs in the geometry array.

6.3 Cell orientation encoding

A distinguishing feature of DOLFINx is support for high-order finite element spaces on general unstructured meshes without any special ordering of cells. A key feature that enables support for high-order elements on general unstructured meshes is the computation, for each cell, of local entity orientation relative to a reference orientation. A DOLFINx mesh topology algorithm encodes this information into one unsigned 32-bit integer for each cell. The need for this information is covered later in section 8.4.

6.4 Refinement

DOLFINx supports scalable refinement (local and uniform) of distributed meshes of simplex cells. It uses the algorithm presented in [80], and the algorithm uses local neighborhood communication patterns which leads to a scalable implementation.

7 FUNCTION SPACES, INTERPOLATION AND FORMS

A fundamental feature of any finite element library is its capacity to assemble matrices and vectors over finite element function space(s). In this section we consider the DOLFINx implementation of a discrete function space on a mesh, the definition of forms on function spaces and the assembly of forms over meshes.

7.1 Degree-of-freedom maps

A DOF map specifies which global DOF numbers are associated with the local DOFs on each cell. To create a DOF map, we require information about the topology of the mesh and the layout of DOFs on each cell, from which DOLFINx creates a distributed DOF map. The element-specific layout of DOFs on each cell is provided by Basix and stored in an `ElementDofLayout` in DOLFINx. An `ElementDofLayout` associates each local DOF of an element with either a vertex, an edge, a face, or the cell. Mesh topology is used to ensure that local DOFs associated with mesh entities that are shared by multiple cells are all assigned the same global DOF number. The geometry of the mesh is not required, as only the connectivity between cells has an impact on the DOF numbering. An instance of the `DofMap` class stores a vector of global DOF numbers and provides methods that return this information as an adjacency list. These data may be queried to discover global DOF numbers associated with each cell.

7.2 Function spaces and finite element functions

Mimicking the mathematical structure of a function space, DOLFINx function spaces bring together 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. for a continuous Lagrange element of degree 2, and space can be created by:

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

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

```
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, with functions spaces constructed from the nested elements. In the above example, 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.

Given a DOLFINx model of a function space, finite element functions can be created on a space. A `Function` is an object that holds a function space and the DOFs coefficients associated with the function. The below snippet demonstrates the creation of `Function` objects on a function space with different scalar types for the coefficient values.

```
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 optimal interpolation of user-provided expressions into finite element spaces. This builds on the Basix structure for defining finite elements through the definition of the dual basis.

Basix provides sufficient information to evaluate the functionals in an element's dual basis $\mathcal{L}$. When creating an element, Basix uses the implementation of the dual basis to compute a dual matrix $l_i(p_j) \in \mathbb{R}^{n\times n}$, where $\{p_0, \ldots, p_{n-1}\}$ is a basis of $\mathcal{P}$ (in Basix, we always use an orthonormal basis here). The dual matrix can then be inverted to find the coefficients that define the primal basis $\{\phi_0, \ldots, \phi_{n-1}\}$. The same information can be used to apply the functionals to a given expression to compute the coefficients of an interpolation of the expression in the finite element space.

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

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

The coefficients can be computed by applying the functionals in the dual basis $\mathcal{L}$ to the expression f. As an implementation of an element in Basix includes information for evaluating the dual basis, DOLFINx can perform interpolation by

using this information on a cell. This interpolation can be performed for any Basix element; there is no requirement for the dual space to only include point evaluation functionals.

The below snippet illustrates the interpolation of the expression $f = \sin \pi x \sin \pi y$ into a continuous Lagrange space of degree 3. It also highlights the functional design, the user provides the mathematical expression to apply to the data (coordinate data in this case). The 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.

```
msh = create_unit_square(MPI.COMM_WORLD, 12, 12, cell_type=CellType.quadrilateral)
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. The approach to interpolation used in DOLFINx is particularly powerful for elements with integral moment functionals, which are used to define a number of finite elements, including N1 [77], Nédélec second kind (N2) [78], RT [83], serendipity [7], and Brezzi–Douglas–Marini (BDM) [19] elements among others.

The following integral moment functionals are used to define a degree p N1 element on a tetrahedron:

- 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 the tangential components 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 a lowest degree N1 element, there is one functional associated with each edge that is evaluated by integrating the product of the tangential component of the input function against a constant. In this case, the element can be properly defined (in the sense of unisolvency) by associating one functional with each edge that is evaluated by finding the tangential component of the input function at the midpoint of the edge. For higher-degree elements, evaluation of the functionals is more complex but the implementation in DOLFINx is straightforward. For example, the function $\boldsymbol{g}(x, y, z) = (\sin(8x), 2^y \cos(3z), x)$ is interpolated in a degree 3 N1 space using:

```
nedelec = dolfinx.fem.FunctionSpace(msh, ("N1curl", 3))
g_h = dolfinx.fem.Function(nedelec)
g_h.interpolate(lambda x: np.array([np.sin(8*x[0]), 2**x[1]*np.cos(3*x[2]), x[0]]))
```

For higher degree N1 elements, replacing the integral functionals with a set of point evaluations leads to sub-optimal interpolation errors. We demonstrate this by defining a point evaluation-based N1 element (using Basix's custom element interface, see section 5.2). The left-hand plot in figure 10 shows the interpolation error when the expression $\boldsymbol{g}$ is interpolated using (i) the standard integral moment element (see the above snippet) and (ii) the point evaluation element, for different element degrees. The issues with the point evaluation variant is more pronounced in the right-hand plot in figure 10, where the interpolation error is shown for a degree 3 N1 element as we decrease the cell size h. The point evaluation elements converge at the sub-optimal rate of $O(h^2)$, while the integral moment version achieves the expected $O(h^3)$ rate. This sub-optimal interpolation order is observed in a number of implementations of N1 elements that use point evaluations, e.g., FInite element Automatic Tabulator (FIAT) [57, 73] and MFEM [4, 31]. Because DOLFINx can

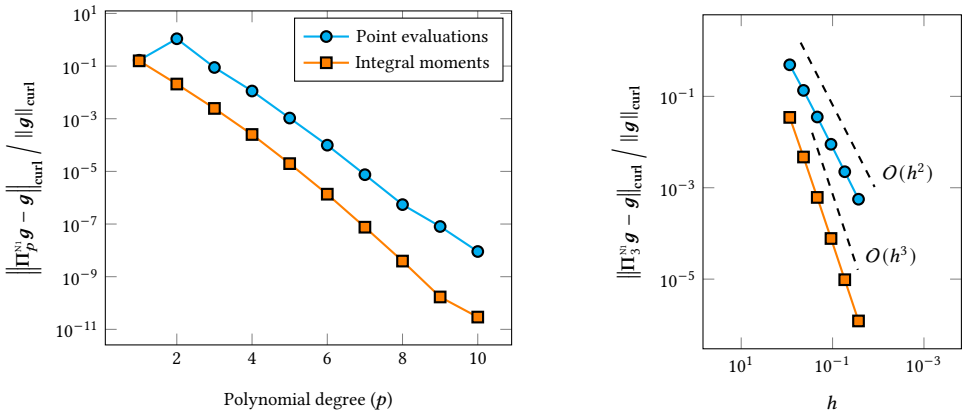

Fig. 10. The interpolation error when the function $\boldsymbol{g}(x, y, z) = (\sin(8x), 2^y \cos(3z), x)$ is interpolated into a space of N1 elements on a tetrahedron defined using either point evaluation or integral moment functionals. The left plot shows the errors as we increase p on a mesh of the 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.

interpolate functions into elements defined using any type of functional in the dual basis we avoid the disadvantages of point evaluation definitions without any added user complexity.

7.3.2 Expressions. Expressions in the DOLFINx context are symbolic UFL expressions that act on finite element functions (as defined in [3, section 3.2]) that, supported by FFCx code generation, can be computed. For example, let $\tilde{f} \in \mathcal{P}_1$ be a continuous linear Lagrange finite element function. Consider the case where we wish to compute its discrete gradient as a discontinuous vector-valued degree 0 Lagrange finite element function, $\tilde{\boldsymbol{g}} \in [\mathcal{P}_0]^2$, i.e.,

$$\tilde{\boldsymbol{g}}(\boldsymbol{x}) = \nabla \tilde{f} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \boldsymbol{\psi}_i(\boldsymbol{x}) l_i(\nabla \phi_j) f_j. \tag{3}$$

Here l_i are the m elements of the dual basis of $[\mathcal{P}_0]^2$ with associated vector-valued basis functions $\boldsymbol{\psi}_i$, ϕ_j are the n basis functions associated with $\mathcal{P}_1$ and f_j are the coefficients of $\tilde{f}$. Computing $\tilde{\boldsymbol{g}}$ (or equivalently, its coefficients g_i) involves the evaluation of the derivatives of the basis functions ϕ_i at functionals in the dual basis of $\tilde{\boldsymbol{g}}$, their contraction with the coefficients of $\tilde{f}$, and finally insertion into the global vector of coefficients $\boldsymbol{g}$. FFCx can generate a local expression kernel that performs the first two operations, and DOLFINx provides the necessary routines to perform the insertion. A common operation in simulation post-processing is to compute derived quantities, such as stress or strain from a computed displacement field. Figure 11 illustrates the code that, given a computed displacement field `uh` defines an `Expression` for evaluating the von Mises stress at the interpolation points for a discontinuous Lagrange element.

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, $H(\text{div})$- and $H(\text{curl})$-conforming finite element spaces can be visualized exactly by interpolating into a sufficiently rich discontinuous Lagrange space prior to visualization.

```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)
```

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

7.4 Forms

Finite element variational forms can be represented using UFL. UFL is now a well-established FEniCS library; we present the definition of finite element forms here briefly for completeness. UFL forms have already appeared in figure 1. The bilinear form for Helmholtz equation (1) is expressed in UFL as:

```python
V = fem.functionspace(...)
u, v = ufl.TestFunction(V), ufl.TrialFunction(V),
k = 4 * np.pi
a = ufl.inner(ufl.grad(u), ufl.grad(v)) * ufl.dx - k**2 * ufl.inner(u, v) * ufl.dx
```

As this stage, `a` is an abstract representation of the Helmholtz bilinear form, and needs to compiled by FFCx to provide a concrete representation that can be used in computations. The creation of a concrete representation of the form in DOLFINx is handled explicitly, giving the user control over when JIT compilation is triggered and the parameters that affect the generated code. For example, the below snippet shows how two concrete instantiations of a finite element form can be created from the same abstract definition, with the first using a real type and the second a complex type.

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

The main differences between an abstract UFL form and a concrete DOLFINx form are that the DOLFINx form is equipped with a finite element kernel function for evaluating the form on a cell (or a cell entity), whereas a UFL does not have an associated kernel function but can be further manipulated symbolically as a UFL object. The two concepts are very deliberately separated in DOLFINx as there is no unique kernel function implementation for a given UFL form. Explicit control over when form computation is triggered avoids mutable states that can make programs harder to reason with and introduce opaque memory resource demands, and explicit control makes it easier to manage the use of system resources, for example when triggering UFL form preprocessing and compilation.

We wish to stress that DOLFINx is designed to also be used following a more traditional approach where, in place of generated code, a user can develop element kernel functions directly. This distinguishes DOLFINx from its predecessor, DOLFIN. Some short examples on how user kernels can be used in DOLFINx are presented in the following section.

8 ASSEMBLY

Central to any finite element library is the assembly process to assemble a finite element form into a scalar, vector or matrix, depending on the form's rank. What largely defines a specific finite element problem type is the local kernel that is executed over cell entities of a particular type, most commonly cells, with the output of the kernel suitably accumulated into a scalar, vector or matrix.

Performance of a finite element solver depends heavily on the performance of the local kernel, and the local kernel is what differs most between computations for different PDEs with different element types. Local kernels are often created by users for a specific problem of interest. The DOLFIN library eased the burden of kernel creation by using a domain-specific language (UFL) and a code generator (FFC). This approach can be highly effective for a wide range of cases. However, DOLFIN could not easily support cases that fell outside of the abstractions of UFL and requiring code generation can be a significant burden when exploring the properties and performance of new numerical methods or implementations.

We demonstrate in this section how the data-oriented and functional design of DOLFINx retains the attractive features of DOLFIN and overcomes its limitation via support for a range of high performance 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 complete, performant and parallel assembly functions in Python using Numba.

8.1 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) [14]:

```
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);
```

where `T` is the data type (e.g. `float`, `double`, `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 $(c_0, \dots, c_{N_c}, d_0, \cdots, d_{N_d}, \dots)$, 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 attached to the form, holding constants (of any rank) that are constant over a subdomain. The array `coordinate_dofs` holds the physical coordinate degrees-of-freedom for a cell. The local index (relative to the cell) of the sub-entity that the kernel is executed over is pointed to by `entity_local_index`, and `quadrature_permutation` points to an integer that encodes how the sub-entity should be permuted to ensure the orientation on two neighboring cells agree. Generated or hand-coded kernels that conform to this interface can exploit the built-in assembly functions. The simplicity of the kernel interface eases possible integration into other finite element solvers, e.g. [72].

8.2 Assembling generated kernels

Any form that can be expressed in UFL can be assembled over a mesh into a tensor with DOLFINx. Given a UFL form, FFCx generates code kernels in C that are executed over cells by the assembler. From C++, the generated code can be compiled into a program. From Python, the a UFL form is JIT compiled using CFFI [86] and passed to a DOLFINx assembly function. The below snippet demonstrates the assembly of a mass operator into a DOLFINx native distributed sparse matrix using single-precision complex floating point numbers.

```
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()
```

The syntax is broadly similar to DOLFIN, given that both use UFL to express forms. Noteworthy differences are on line 4, where the form compilation is now triggered explicitly and the floating point type can be specified, and the manual reverse-scatter on line 6. Form compilation is explicit to discourage user code that introduces repeated pre-processing of a UFL form, which can have a non-negligible cost, to allow different implementations to be generated from a common UFL definition, e.g., `np.float32` and `np.complex64` versions, and to provide fine-grained control over resources by controlling when JIT compilation is triggered. 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 and makes the preservation of consistent behavior more difficult to achieve. 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:

```
1 auto V = std::make_shared<fem::FunctionSpace<float>>(.....);
2 fem::Form a = fem::create_form<float, float>(*form_mass, {V, V}, {}, {}, {}));
3 la::SparsityPattern sp = fem::create_sparsity_pattern(a);
4 sp.finalize();
5 la::MatrixCSR<float> A(sp);
6 fem::assemble_matrix(A.mat_add_values(), a, {});
7 A.scatter_rev();
```

In this case, `form_mass` is a pointer to UFCx-compliant form struct generated by FFCx and which provides an assembly kernel function.

For a wide range of examples of the DOLFINx interface and functionality for automatically assembling finite element forms, we refer to the documentation and demo programs.

8.3 Custom kernels and assemblers

From C++, DOLFINx can be used as a library with user-developed element kernel functions passed to a DOLFINx assembler, following the function-oriented approach, or users can program complete assemblers. Unlike DOLFIN, DOLFINx supports both traditional development approaches and the code generation paradigm. More challenging is supporting fast, user-developed kernels from Python. This section is dedicated to how DOLFINx enables fast custom kernels and assemblers from Python.

For custom operations, DOLFINx allows users to (1) supply a custom kernel to the built-in assembly functions, (2) use their own assembly functions with generated kernels, or (3) provide their own kernel and assembler. Within a single application, these three approaches can be combined as appropriate. From Python, both kernels and assemblers can be written using Numba [62], which just-in-time compiles Python/NumPy code to machine code using LLVM [64]. Alternatively, CFFI can be used to compile C functions that can be used in Python and in the DOLFINx core. For all cases, the functionality is enabled by the data- and function-oriented design of DOLFINx.

For the purposes of exposition, the presented examples are deliberately simple and are also acheivable within the code generation abstractions in section 8.2.

8.3.1 Kernels. We first consider the creation of an element matrix kernel using Numba. The high-level workflow for this case is shown in figure 5. Consider the assembly of element mass matrices $A := (a_{ij}) \in \mathbb{R}^{3\times3}$ on affine triangles R using degree 1 Lagrange basis functions. The local matrix entries are 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 $\hat{M} := (\hat{m}_{ij}) \in \mathbb{R}^{3\times3}$ is a mass matrix on the reference triangle $\hat{R}$. Once the matrix $\hat{\mathrm{M}}$ is computed (as `M_hat`) the Numba code in figure 12 defines a kernel that computes A on a given cell. In figure 12, `c_signature` in the decorator has type `numba.types` (details omitted) and is a Numba object that defines a C interface for the function which conforms to the kernel interface defined in section 8.1. The memory address (pointer) to the Numba compiled function can then be passed to the `Form` initializer, creating a form object that is equipped with our custom cell kernel function, illustrated by the below code extract:

```
cells = np.arange(msh.topology.index_map(msh.topology.dim).size_local, dtype=np.int32)
integrals = {dolfinx.fem.IntegralType.cell: [(-1, tabulate_A.address, cells), ]}
coefficients_A, constants_A = [], []
a = dolfinx.fem.Form(formtype([V._cpp_object, V._cpp_object],
                              integrals, coefficients_A, constants_A, False))
```

In the above example, `tabulate_tensor_A.address` (which is of type `int`) is internally cast to a `std::function` with the required kernel signature, and can be passed to the assembler. Should the custom kernel require additional data, this can be passed to the form initializer as constants or coefficients, or the kernel function can capture data from outside of its scope.

The expressive symbolic power of UFL and the code generation capabilities of FFCx can be combined with custom kernels. A user can compile UFL forms with the FFCx JIT compiler, and can call the compiled kernels from within a custom Numba kernel. This can be useful for implementing operations where one may wish to modify the standard kernel output, e.g. to apply static condensation. Figure 13 shows an example of calling a FFCx generated kernel from within a Numba kernel. The compiled kernel `ufcx_kernel` computes an element matrix (the weighted mass matrix in the full example in the supplementary material). In this case, the matrix computed by `ufcx_kernel` could be modified before being passed to the assembler.

8.3.2 Assemblers. The data-oriented approach followed by DOLFINx makes it possible to write complete and efficient assembly functions in other languages, including from Python. The data underpinning key objects, including meshes and DOF maps, can be accessed as plain data types and shared across language interfaces without copy, and parallel execution normally requires not special consideration. Figure 14 presents a complete assembly function, using Numba, for the mass matrix using degree 1 Lagrange basis on a triangle. Mesh data is passed in as non-owning NumPy array

```python
@numba.cfunc(c_signature, nopython=True)
def tabulate_A(A_, w_, c_, coords_, entity_local_index, quadrature_permutation=None):
    # 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
```

Fig. 12. A Numba function that captures the already-computed element mass matrix on the reference cell, `M_hat`, computed outside of the function body, to compute the element mass matrix on a physical cell.

```python
@numba.cfunc(c_signature, nopython=True)
def tabulate_A_wrapped(A_, w_, c_, coords_, entity_local_index, quadrature_permutation=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,
                quadrature_permutation)

    # At this point, custom manipulations could be applied to A
    A[:] = M
```

Fig. 13. A Numba kernel that calls a FFCx generated kernel (`ufcx_kernel`).

views into the DOLFINx C++ mesh data structures. `set_vals` (details omitted) is defined using Python *ctypes* to wrap the PETSc matrix insertion call at the C binary level. Consequently, Numba assemblers execute entirely using compiled code and with the Python Global Intepreter Lock (GIL) released.

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 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 an ordering that guarantees a common orientation of shared sub-entities is not possible for all hexahedral cell meshes [1, 51].

To achieve consistent sub-entity orientations in DOLFINx, we use a method of DOF permutations and transformations [94]. 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.

```python
@numba.njit
def area(x0, x1, x2) -> float:
    """Compute the area of a 2D triangle from its vertices."""
    a = (x1[0] - x2[0])**2 + (x1[1] - x2[1])**2
    b = (x0[0] - x2[0])**2 + (x0[1] - x2[1])**2
    c = (x0[0] - x1[0])**2 + (x0[1] - x1[1])**2
    return math.sqrt(2 * (a * b + a * c + b * c) - (a**2 + b**2 + c**2)) / 4.0


@numba.njit
def assemble(A, mesh, dofmap, num_cells, set_vals, mode):
    """Assemble P1 mass matrix over a mesh into the PETSc matrix A."""
    # Extract mesh topology and geometry
    v, x = mesh

    # Quadrature points and weights
    q = np.array([[0.5, 0.0], [0.5, 0.5], [0.0, 0.5]], dtype=np.double)
    weights = np.full(3, 1.0 / 3.0, dtype=np.double)

    N = np.empty(3, dtype=np.double)
    A_local = np.empty((3, 3), dtype=np.double)

    # Iterate over cells
    for cell in range(num_cells):
        cell_area = area(x[v[cell, 0]], x[v[cell, 1]], x[v[cell, 2]])

        # Loop over quadrature points
        A_local[:] = 0.0
        for j in range(q.shape[0]):
            N[0], N[1], N[2] = 1.0 - q[j, 0] - q[j, 1], q[j, 0], q[j, 1]
            for row in range(3):
                for col in range(3):
                    A_local[row, col] += weights[j] * cell_area * N[row] * N[col]

        rows = cols = dofmap[cell, :]
        set_vals(A, 3, rows.ctypes, 3, cols.ctypes, A_local.ctypes, mode)
```

Fig. 14. A Numba function that assembles a mass matrix on a mesh.

DOLFINx orients an entities in the physical mesh by identifying which of its vertices has the lowest topological index, then identifying which vertex that is connected by an edge to this lowest index vertex has the lower index. For example, a quadrilateral is oriented by first identifying the vertex which has the lowest index in the mesh topology, then looking at the two edges connected to this vertex and identifying which of the vertices at the other end of these edges has the lower index number in the mesh topology. DOF permutations and transformations are applied when the vertices used to orient a physical cell do not match those used on the reference cell. 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 [77] 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 [94].

Function name	Operation
`pre_apply_dof_transformation`	$\mathbf{MA}$
`pre_apply_transpose_dof_transformation`	$\mathbf{M}^{\mathsf{T}}\mathbf{A}$
`pre_apply_inverse_dof_transformation`	$\mathbf{M}^{-1}\mathbf{A}$
`pre_apply_inverse_transpose_dof_transformation`	$\mathbf{M}^{-\mathsf{T}}\mathbf{A}$
`post_apply_dof_transformation`	$\mathbf{AM}$
`post_apply_transpose_dof_transformation`	$\mathbf{AM}^{\mathsf{T}}$
`post_apply_inverse_dof_transformation`	$\mathbf{AM}^{-1}$
`post_apply_inverse_tranpose_dof_transformation`	$\mathbf{AM}^{-\mathsf{T}}$

Fig. 15. 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

$$f_h = \boldsymbol{c}^{\mathsf{T}}\boldsymbol{\phi} = \tilde{\boldsymbol{c}}^{\mathsf{T}}\tilde{\boldsymbol{\phi}}, \tag{5}$$

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 $\mathbf{M}$ such that

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

Inserting this into (5) leads to $\boldsymbol{c}^{\mathsf{T}}\boldsymbol{\phi} = \boldsymbol{c}^{\mathsf{T}}\mathbf{M}\tilde{\boldsymbol{\phi}} = (\mathbf{M}^{\mathsf{T}}\boldsymbol{c})^{\mathsf{T}}\tilde{\boldsymbol{\phi}} = \tilde{\boldsymbol{c}}^{\mathsf{T}}\tilde{\boldsymbol{\phi}}$, therefore vectors of DOFs transform according to

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

While $\mathbf{M}$ is often orthogonal, this not the case for all elements (see [94, 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 $\mathbf{A} = \mathbf{M}_1\tilde{\mathbf{A}}\mathbf{M}_2^{\mathsf{T}}$, where $\mathbf{M}_1$ and $\mathbf{M}_2$ are the transformations associated with the test and trial function elements, respectively.

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 `permute_dofs` applies the $\mathbf{M}$ operation to a vector, and `unpermute_dofs` applies the inverse transformation ($\mathbf{M}^{-T}$, 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 $\mathbf{M}$ in-place are listed in figure 15.

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 for permutations following [38] and for matrix products following [35].

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 $\boldsymbol{c} = \mathbf{M}^{-\mathsf{T}}\tilde{\boldsymbol{c}}$, we apply the function `pre_apply_inverse_transpose_dof_transformation` 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 custom kernels or assemblers developed in Python using Numba and which require degree-of-freedom transformations that are not just permutations and therefore must be applied during assembly, the submodule `basix.numba_helpers` provides the functions `pre_apply_dof_transformation` and `post_apply_dof_transpose_transformation` for efficient application of transformations.

9 LINEAR ALGEBRA

There exists a rich range of linear algebra libraries providing data structures and solvers, e.g. PETSc, Trilinos [99] and Eigen [42]. Our experience is that third-party linear algebra libraries are best supported *not* through wrappers, but non-intrusively and allowing users direct access to the complete interface of the 3rd-party library. Our advocated approach 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. We will show examples of how the functional and data-oriented interfaces support 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. 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 also templated over the scalar type and the internal storage container type. 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 issues 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 below code illustrates how a matrix sparsity pattern can be constructed.

```
fem::Form a(...);
la::SparsityPattern pattern = fem::create_sparsity_pattern(a);
```

```
  pattern.finalize();
  auto [nonzeros, row_offset] = pattern.graph();
```

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 a PETSc matrix factory function is also provided:

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

// ...

MatDestroy(B);
```

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, application of 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 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:

```
fem::Form a(...);
Mat A;

// Initialize PETSc matrix A . . .

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, {});
```

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 coping 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. Assembly into other library formats, e.g. Trilinos or Eigen, is straightforward and in all cases the DOLFINx core library is unaware of the matrix library interface.

10 CONCLUSIONS

We have presented an overview of the design principles and implementation of the DOLFINx finite element library. DOLFINx builds on the approach of the earlier FEniCS library DOLFIN, providing an 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 and improved performance. 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 [13]. The snippets in this paper are compatible with DOLFINx 0.7.3, Basix 0.7.0, FFCx 0.7.0 and UFL 2023.2.0.

ACKNOWLEDGMENTS

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, 5, 12

BDM Brezzi–Douglas–Marini. 13, 22

CFFI C Foreign Function Interface. 5, 26, 27
CR Crouzeix–Raviart. 13

DOF degree of freedom. 4, 12, 13, 14, 17, 20, 21, 25, 27, 28, 29, 30
DOLFIN Dynamic Object-oriented Library for FINite element computation. 2, 3, 4, 5, 24, 25, 26, 33
DPC discontinuous polynomial cubical. 12, 13

FFC FEniCS Form Compiler. 2, 25
FFCx FEniCSx Form Compiler. 3, 6, 8, 9, 10, 13, 23, 24, 26, 27, 28
FIAT FInite element Automatic Tabulator. 22

GIL Python Global Intepreter Lock. 28
GL Gauss–Legendre. 12
GLL Gauss–Lobatto–Legendre. 12

HHJ Hellan–Herrmann–Johnson. 13
HPC high performance computer. 6

I/O input/output. 6, 7, 8

JIT just-in-time. 6, 8, 11, 24, 26, 27, 33

MPI Message Passing Interface. 9

N1 Nédélec first kind. 13, 22, 23, 29
N2 Nédélec second kind. 13, 22
NumPy . 5

PDE partial differential equation. 5
PETSc Portable, Extensible Toolkit for Scientific Computation. 7, 28, 32, 33

RT Raviart–Thomas. 13, 22

TNT tiniest tensor. 13, 15

UFCx Unified Form-assembly code for FEniCSx. 25
UFL Unified Form Language. 2, 6, 7, 8, 9, 10, 12, 13, 15, 23, 24, 25, 26, 27

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] 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

[6] 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

[7] 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

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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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. 2023. *Supplementary Material. DOLFINx: The next generation FEniCS problem solving environment.* https://doi.org/10.5281/zenodo.10026723

[14] 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].

[15] 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

[16] 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

[17] 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

[18] 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.

[19] 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

[20] 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

[21] 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

[22] 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.

[23] 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

[24] 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

[25] 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

[26] 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].

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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.

[34] 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.

[35] David Eisenstat. 2014. Is there an algorithm to multiply square matrices in-place? Stack Overflow. https://stackoverflow.com/a/25451717 [Online; accessed 25-September-2023].

[36] 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

[37] 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

[38] Fich, Faith E. and Munro, J. Ian and Poblete, Patricio V. 1995. Permuting in Place. *SIAM J. Comput.* 24, 2 (1995), 266–278. https://doi.org/10.1137/S0097539792238649

[39] 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

[40] 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

[41] 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

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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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

[49] 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

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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

[55] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11 – Seamless operability between C++11 and Python. https://github.com/pybind/pybind11

[56] 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

[57] 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

[58] 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

[59] 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

[60] 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

[61] 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

[62] 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

[63] 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.

[64] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[65] 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

[66] 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

[67] 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

[68] 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

[69] 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

[70] 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

[71] 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

[72] 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

[73] 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

[74] 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

[75] 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

[76] 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

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

[78] 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

[79] 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

[80] 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

[81] 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

[82] 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

[83] 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

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

[85] 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

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

[87] 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

[88] 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

[89] 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

[90] 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

[91] Matthew W. Scroggs et al. 2023. DefElement: Reference cell numbering. https://defelement.com/reference_numbering.html. [Online; accessed 25-September-2023].

[92] 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

[93] 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

[94] 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

[95] 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

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

[97] 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

[98] 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.

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

[100] 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

[101] 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

[102] 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

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

Received