The 8th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2022
5-–9 June 2022, Oslo, Norway

PSYDAC: A HIGH PERFORMANCE PYTHON LIBRARY FOR ISOGEOMETRIC ANALYSIS

Yaman Güçlü[1,*], Said Hadjout[1,2] and Ahmed Ratnani[3]

[1] Numerical Methods in Plasma Physics Division, Max Planck Institute for Plasma Physics.
Boltzmannstr. 2, 85748 Garching bei München, Germany. https://www.ipp.mpg.de.

[2] Department of Mathematics, Technical University of Munich.
Boltzmannstr. 3, 85748 Garching bei München, Germany. https://www.ma.tum.de/en.

[3] Complex Systems Engineering & Human Systems, Mohammed VI Polytechnic University.
Lot 660, Hay Moulay Rachid Ben Guerir, 43150, Morocco. https://um6p.ma.

Key words: Finite Element Method, Isogeometric Analysis, Python, High-Performance Scientific Computing, Message-Passing Interface, B-Splines

Abstract. Psydac is a Python 3 library for the solution of partial differential equations, which combines the convenience of a domain specific language with the speed of a high-performance parallel engine. Its main focus is on isogeometric analysis using tensor-product B-spline finite elements; to this end it uses an optimized sparse format called "stencil matrix", which drastically reduces memory storage compared to the popular CSR/CSC formats. It supports multi-patch mapped geometries, and finite element exterior calculus. It can distribute each domain patch across many MPI processes, with multiple OpenMP threads operating in each block.

The users of Psydac define a weak form of the model equations through SymPDE, an extension of Sympy that provides the mathematical expressions and checks their semantic validity. Simple mappings can be defined analytically, and multi-patch NURBS geometries can be imported from file. Once a finite element discretization is chosen, Psydac maps abstract concepts onto concrete objects, the basic building blocks being MPI-distributed vectors and matrices. Python code is automatically generated for the model-specific operations, namely matrix and vector assembly, and user-defined diagnostics. Finally, Psydac accelerates all computationally intensive operations using Pyccel, a transpiler which converts Python code to either C or Fortran.

We present the library design, the typical usage workflow, the user interface for a simple 2D example, and the parallel scaling results in a large 3D simulation. In addition we show a few complex applications in fluid dynamics and electromagnetism, where the accuracy of the solver is verified against manufactured and reference solutions.

1 INTRODUCTION

Finite element methods are widely used for the numerical solution of partial differential equations (PDEs). Starting from a variational form, and appropriate finite-dimensional function spaces equipped with their own bases, it systematically reduces a linear PDE to a system of linear

* Corresponding author. E-mail: yaman.guclu@ipp.mpg.de.

equations which can be solved on a computer. The generality of the method is not reflected in the implementation, as in practice one often needs to write a code for a specific problem, for which it can be fully optimized. This task takes a huge amount of effort as the software developer needs to fully understand the theory, be able to write an efficient code, and apply different optimization techniques to it.

Aiming at helping the application specialists, several frameworks seek to automate different aspects of the finite element method. Many tools are developed in C/C++ like Dolfin [13], PetIGA [9], and Sundance [14]. Some codes like FreeFem++ [11] and Feel++ [15] use domain-specific languages, while others combine a user-friendly Python API with computational kernels written in low level languages, like Fenics [3] and FireDrake [16]. Providing that kind of a tool is challenging due to two main issues: the first is the complexity of the task, as the software needs to work with different kinds of equations with different discretization techniques while providing different solvers; the second is the fact that a specialized code often outperforms a general purpose library.

In this paper we present Psydac, a new framework which aims at addressing the issues just described. Section 2 gives an overview of the library and its components. Section 3 shows some numerical examples of increasing complexity. Section 4 discusses the parallel performance of the library. Finally, Section 5 presents some conclusions.

2 LIBRARY OVERVIEW

Psydac is a Python 3 library for isogeometric analysis that combines three main features: 1) a high-level domain specific language for describing the weak formulation of a partial differential equation, 2) automatic code generation (Python and Fortran) for the model-specific computational kernels, and 3) massive runtime parallelization through MPI and OpenMP.

The dependency graph of Psydac is depicted in Figure 1. The domain specific language is provided by SymPDE, an extension of the popular computer algebra system Sympy. SymPDE provides function spaces over topological domains, differential operators, integrals, mappings, push-forward and pull-back operators, functionals, linear and bilinear forms, Sobolev norms, etc. Upon discretization, Psydac creates the finite element counterparts of the abstract objects defined in SymPDE. The discrete fields and the matrix operators are distributed across multiple processes using the message-passing interface (MPI). In addition Psydac generates an abstract syntax tree (AST) for some model-specific kernels that are computationally intensive. The Psydac AST is then converted to the AST of Pyccel, a Python to C/Fortran transpiler written in Python which provides the code generation capabilities. Pyccel generates Python code, which can be inspected by the user, and later converts it to Fortran code that is also available for inspection; finally it creates a C Python extension module that is imported from Psydac.

The workflow of a typical Psydac simulation is depicted in Figure 2. The solid arrows identify the "online" phase, where the simulation is driven by a main Python script provided by the user, and the computational domain is defined through a geometry file. Psydac writes the (possibly time-dependent) solution to a single HDF5 file. The "offline" phase is identified by dashed arrows: here the user runs a post-processing script which leverages Psydac's post-processing functionality to generate VTK files. In both phases all calculations and I/O operations are MPI-parallel. Finally the dotted arrows identify the visualization phase, where the VTK files

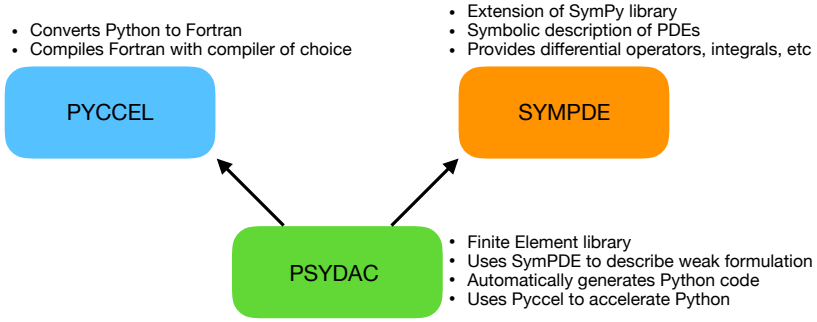

Figure 1: Dependency graph of Psydac, which requires the SymPDE and Pyccel libraries.

are read, e.g., using a parallel installation of Paraview – usually available on supercomputers.

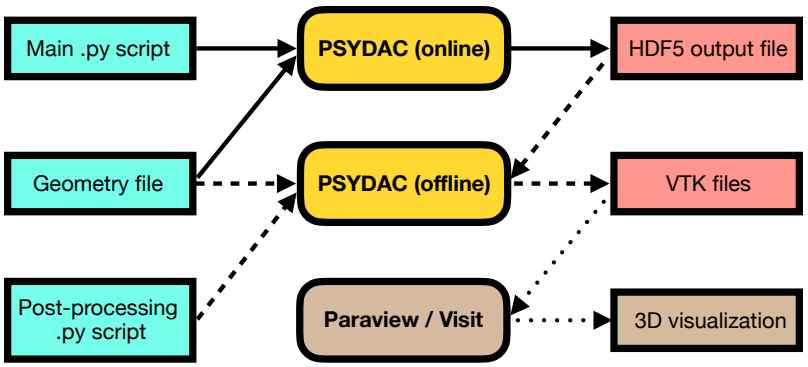

Figure 2: Workflow of a typical Psydac simulation.

3 NUMERICAL EXAMPLES

3.1 3D Poisson problem

As a first example we solve Poisson's equation on the domain $\Omega \subset \mathbb{R}^3$ with homogeneous Dirichlet conditions on the boundary $\partial\Omega$. Given $f\colon \Omega \to \mathbb{R}$, we search for $u\colon \Omega \to \mathbb{R}$ such that

$$-\nabla^2 u = f \quad \text{in } \Omega, \qquad \text{and} \quad u = 0 \quad \text{on } \partial\Omega. \tag{1}$$

If we let $f \in L^2(\Omega)$ and $u \in H^1_0(\Omega)$, the weak formulation of (1) reads

$$\text{Find } u \in H^1_0(\Omega) \text{ s.t.} \quad \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}\Omega = \int_\Omega f v \,\mathrm{d}\Omega \quad \forall v \in H^1_0(\Omega). \tag{2}$$

For our test case we take the domain $(r, \theta, \varphi) \in [1, 4] \times [0, \pi] \times [0, \pi/2]$ in the spherical coordinates. For testing purposes we choose a manufactured solution u_{ex} that vanishes on the boundary $\partial\Omega$,

$$u_{\text{ex}}(x, y, z) := xy \sin\left(\pi \frac{x^2 + y^2 + z^2 - r_{\text{in}}^2}{r_{\text{out}}^2 - r_{\text{in}}^2}\right) \cos\left(\pi \frac{x^2 + y^2 + z^2 - r_{\text{in}}^2}{r_{\text{out}}^2 - r_{\text{in}}^2}\right), \tag{3}$$

where $r_{\text{in}} = 1$, $r_{\text{out}} = 4$, and we compute the source term on the right-hand side through symbolic differentiation of the above, as $f(x, y, z) := -\nabla^2 u_{\text{ex}}(x, y, z)$.

We now present the complete code for the solution of this problem. In Python Code 1 we use Sympy and SymPDE to define the domain Ω and the exact solution u_{ex}, compute the analytical expression for the right-hand side $f := -\nabla^2 u_{\text{ex}}$, define the variational problem (2), and define the L^2 and H^1 norms of the symbolic error $u - u_{\text{ex}}$. In Python Code 2 we use Psydac to setup a finite element solver using tensor-product splines of polynomial degree p with N uniform cells along each direction, solve the resulting linear system to obtain the numerical solution u_h, and compute the L^2 and H^1 norms of the numerical error $u_h - u_{\text{ex}}$.

Table 1 contains the L^2 and H^1 norms of the numerical error for different values of p and N, together with estimates of the convergence order (computed by keeping p fixed and increasing N).

```
from sympy import pi, sin, cos

from sympde.calculus import laplace, dot, grad
from sympde.topology import Cube, SphericalMapping
from sympde.topology import ScalarFunctionSpace, elements_of
from sympde.expr     import integral, BilinearForm, LinearForm
from sympde.expr     import EssentialBC, find, Norm

# Logical domain C is a rectangular cuboid ('Cube' for brevity)
# Physical domain Omega is a quarter spherical shell
(r_in, r_out) = (1.0, 4.0)
C = Cube('C', bounds1=(r_in, r_out), bounds2=(0, pi), bounds3=(0, pi/2))
M = SphericalMapping('M')
Omega = M(C)

# Method of manufactured solutions: define exact solution u_ex
# that is = 0 at the boundaries, then compute right-hand side f
x,y,z = Omega.coordinates
r     = x**2 + y**2 + z**2
arg   = pi * (r - r_in**2) / (r_out**2 - r_in**2)
u_ex  = x * y * sin(arg)*cos(arg)
f     = -laplace(u_ex)

# Function space for trial and test functions
V = ScalarFunctionSpace('V', Omega)
u, v = elements_of(V, names='u v')

# Declare bilinear and linear forms for variational formulation
a = BilinearForm((u,v), integral(Omega, dot(grad(v), grad(u))))
l =   LinearForm(   v , integral(Omega, f * v))

# Dirichlet boundary conditions: u=0 on the domain boundary
bc = EssentialBC(u, 0, Omega.boundary)

# Variational formulation of Poisson's equation
equation = find(u, forall=v, lhs=a(u, v), rhs=l(v), bc=bc)

# Scalar error norms
l2norm = Norm(u - u_ex, Omega, kind='l2')
h1norm = Norm(u - u_ex, Omega, kind='h1')
```

Python Code 1: Numerical solution of the 3D Poisson problem (2) in a quarter spherical shell. Part I: definition of the continuum problem with SymPDE.

```python
from mpi4py import MPI

from psydac.api.discretization import discretize
from psydac.api.settings import PSYDAC_BACKENDS

# Select MPI communicator (set to None for serial code)
comm = MPI.COMM_WORLD

# Select backend for code acceleration: Pyccel + GCC compiler
backend = PSYDAC_BACKENDS['pyccel-gcc']

# Select discretization parameters
ncells = [8, 8, 8]  # N: number of cells along each direction
degree = [3, 3, 3]  # p: spline degree      "     "     "

# Create computational domain from topological domain
Omega_h = discretize(Omega, ncells=ncells, comm=comm)

# Create discrete spline space
Vh = discretize(V, Omega_h, degree=degree)

# Discretize equation and error norms
equation_h = discretize(equation, Omega_h, [Vh, Vh], backend=backend)
l2norm_h   = discretize(l2norm, Omega_h, Vh, backend=backend)
h1norm_h   = discretize(h1norm, Omega_h, Vh, backend=backend)

# Solve discrete equation to obtain numerical solution:
#  1. assemble distributed sparse matrix A and dense vector b
#  2. solve linear system A w = b
#  3. create a callable field u_h(x,y) - spline with coefficients w
u_h = equation_h.solve()

# Compute error norms from solution field
l2_error = l2norm_h.assemble(u=u_h)
h1_error = h1norm_h.assemble(u=u_h)
```

Python Code 2: Numerical solution of the 3D Poisson problem (2) on a quarter spherical shell. Part II: problem discretization and solution with Psydac.

3.2 Time-harmonic Maxwell's equations

We consider the time harmonic Maxwell equations in a 2D domain Ω with inhomogeneous essential conditions on the boundary $\partial\Omega$. We write the electric field as $\boldsymbol{E}(t, \boldsymbol{x}) = \mathfrak{Re}\left(i\omega \boldsymbol{u}(\boldsymbol{x}) e^{-i\omega t}\right)$, where $\omega \in \mathbb{R}$ is the pulsation and $\boldsymbol{u}$ is a complex field. Given a real current source $\boldsymbol{J} \colon \Omega \to \mathbb{R}^2$ and a real tangential trace $g \colon \partial\Omega \to \mathbb{R}$, we search for a real solution $\boldsymbol{u} \colon \Omega \to \mathbb{R}^2$ that satisfies

$$\begin{cases} -\omega^2 \boldsymbol{u} + \nabla \times \nabla \times \boldsymbol{u} = \boldsymbol{J} & \text{in } \Omega, \\ \boldsymbol{n} \times \boldsymbol{u} = g & \text{on } \partial\Omega, \end{cases} \tag{4}$$

where $\boldsymbol{n}$ is the outward normal unit vector to $\partial\Omega$. We notice that in 2D the cross product is scalar-valued; further, the curl of a vector is scalar-valued and the curl of a scalar is vector-valued.

We are interested in solving the problem (4) in a multi-patch domain $\Omega = \cup_k \Omega_k$ of N patches, where each of them is obtained through the mapping of a reference square patch $\hat{\Omega} := [0, 1]^2$, i.e. $\Omega_k = F_k(\hat{\Omega})$. To this end we do not construct a globally conformal approximation of the

(a) Degree $p = 2$

N	L^2 error	L^2 order	H^1 error	H^1 order
4	3.38×10^{-1}	–	2.52×10^{0}	–
8	1.02×10^{-1}	1.73	1.30×10^{0}	0.96
16	9.19×10^{-3}	3.47	3.07×10^{-1}	2.08
32	9.55×10^{-4}	3.27	7.33×10^{-2}	2.07

(b) Degree $p = 3$

N	L^2 error	L^2 order	H^1 error	H^1 order
4	4.79×10^{-1}	–	3.12×10^{0}	–
8	1.89×10^{-2}	4.66	3.25×10^{-1}	3.27
16	1.19×10^{-3}	3.99	3.98×10^{-2}	3.03
32	7.49×10^{-5}	3.99	5.11×10^{-3}	2.96

(c) Degree $p = 4$

N	L^2 error	L^2 order	H^1 error	H^1 order
4	1.23×10^{-1}	–	9.66×10^{-1}	–
8	6.61×10^{-3}	4.22	8.29×10^{-2}	3.54
16	1.28×10^{-4}	5.69	3.95×10^{-3}	4.39
32	3.23×10^{-6}	5.31	2.11×10^{-4}	4.23

(d) Degree $p = 5$

N	L^2 error	L^2 order	H^1 error	H^1 order
4	3.84×10^{-2}	–	2.75×10^{-1}	–
8	9.46×10^{-4}	5.34	1.41×10^{-2}	4.29
16	1.77×10^{-5}	5.74	4.79×10^{-4}	4.88
32	2.46×10^{-7}	6.17	1.50×10^{-5}	5.00

Table 1: Refinement study for 3D Poisson's equation (1) in a quarter spherical shell. For various spline degrees p and number of cells N along each direction, we report the L^2 and H^1 norms of the error between the numerical and analytical solutions, and compute the order of convergence.

space $H(\mathrm{curl};\Omega)$, because this would be technically cumbersome and impractical. Instead, we construct an appropriate vector-valued space over the reference patch $\hat{\Omega}$,

$$\hat{\mathbb{V}}^1 := \begin{pmatrix} \mathbb{S}_{(p-1,p)} \\ \mathbb{S}_{(p,p-1)} \end{pmatrix}, \tag{5}$$

where $\mathbb{S}_{(p_1,p_2)}$ is a scalar-valued tensor-product spline space of maximum regularity, with degrees p_1 and p_2 along the first and second logical dimensions, respectively. We then obtain a conformal approximation within each patch through a push-forward operation for 1-forms, that is

$$V_h^1(\Omega_k) := \mathcal{F}_k^1(\hat{\mathbb{V}}^1) \subset H(\mathrm{curl};\Omega_k),$$

and finally we define the global finite element space by simple juxtaposition of the local ones:

$$V_h^1 := \left\{ \boldsymbol{v} \in [L^2(\Omega)]^2 : \boldsymbol{v}|_{\Omega_k} \in V_h^1(\Omega_k) \right\} \not\subset H(\mathrm{curl};\Omega).$$

In order to connect the solution $\boldsymbol{u}_h \in V_h$ between adjacent patches, we penalize any jump in the tangential trace of $\boldsymbol{u}_h$ across the interfaces $\Gamma_{ij} := \partial\Omega_i \cap \partial\Omega_j$. Similarly we use penalization on each sub-boundary $\Gamma_k := \partial\Omega \cap \partial\Omega_k$ to impose the boundary conditions on $\boldsymbol{u}_h$. Following [7] we obtain a Nitsche weak formulation for equation (4), also known as interior penalty method [4]:

$$\text{Find} \quad \boldsymbol{u}_h \in V_h^1 \quad \text{such that} \quad a_h(\boldsymbol{u}_h, \boldsymbol{v}) = f_h(\boldsymbol{v}) \quad \forall \boldsymbol{v} \in V_h^1,$$

where

$$\begin{aligned} a_h(\boldsymbol{u}_h, \boldsymbol{v}) := & \sum_{\Omega_i} \Big((\nabla \times \boldsymbol{v}, \nabla \times \boldsymbol{u}_h)_{\Omega_i} - (\boldsymbol{v}, \omega^2 \boldsymbol{u}_h)_{\Omega_i} \Big) \\ & + \sum_{\Gamma_{ij}} \Big(-(\llbracket \boldsymbol{v} \rrbracket_T, \{\!\{ \nabla \times \boldsymbol{u}_h \}\!\})_{\Gamma_{ij}} - k(\llbracket \boldsymbol{u}_h \rrbracket_T, \{\!\{ \nabla \times \boldsymbol{v} \}\!\})_{\Gamma_{ij}} + \gamma(\llbracket \boldsymbol{u}_h \rrbracket_T, \llbracket \boldsymbol{v} \rrbracket_T)_{\Gamma_{ij}} \Big) \\ & + \sum_{\Gamma_i} \Big(-(\boldsymbol{n} \times \boldsymbol{v}, \nabla \times \boldsymbol{u}_h)_{\Gamma_i} - k(\boldsymbol{n} \times \boldsymbol{u}_h, \nabla \times \boldsymbol{v})_{\Gamma_i} + \gamma(\boldsymbol{n} \times \boldsymbol{u}_h, \boldsymbol{n} \times \boldsymbol{v})_{\Gamma_i} \Big) \end{aligned}$$

and

$$f_h(\boldsymbol{v}) := \sum_{\Omega_i} (\boldsymbol{J}, \boldsymbol{v})_{\Omega_i} + \sum_{\Gamma_i} \Big(-k(g, \nabla \times \boldsymbol{v})_{\Gamma_i} + \gamma (g, \boldsymbol{n} \times \boldsymbol{v})_{\Gamma_i} \Big).$$

The notation $(\cdot,\cdot)_\sigma$ refers to the L^2 scalar product over the domain σ, while $[\![\boldsymbol{w}]\!]_T := \boldsymbol{n}_i \times \boldsymbol{w}_i + \boldsymbol{n}_j \times \boldsymbol{w}_j$ is the jump in the tangential component of the vector field $\boldsymbol{w}$ across the interface Γ_{ij}, and $\{\!\{w\}\!\} := (w_i + w_j)/2$ is the average value of the scalar field w across the interface. The integer parameter $k \in \{-1, 0, 1\}$ allows us to switch between three different formulations: we choose $k = 1$ for the symmetric interior penalty method (SIP), $k = -1$ for the non symmetric interior penalty method (NIP), and $k = 0$ for the incomplete interior penalty method (IIP). The stabilization parameter is $\gamma := \gamma_{stab} h^{-1}$, with $\gamma_{stab} > 0$ a positive constant, and h the grid size.

As a test case we construct a pretzel-like domain using 18 patches, as shown in the left-most image of Figure 3. We differentiate the manufactured solution $\boldsymbol{u}_{\text{ex}}$ to obtain the source $\boldsymbol{J}$:

$$\boldsymbol{u}_{\text{ex}}(x,y) = \begin{pmatrix} \sin(\pi y) \\ \sin(\pi x)\cos(\pi y) \end{pmatrix}, \qquad \boldsymbol{J}(x,y) = \begin{pmatrix} \sin(\pi y)\big(-\omega^2 + \pi^2(1-\cos(\pi x))\big) \\ \sin(\pi x)\cos(\pi y)\big(-\omega^2 + \pi^2\big) \end{pmatrix}. \tag{6}$$

Using the expression above for $\boldsymbol{J}$ as input to our code, and by setting the tangential trace as $g := \boldsymbol{n} \times \boldsymbol{u}_{\text{ex}}$, we compute a numerical solution $\boldsymbol{u}_h$ and compare it to $\boldsymbol{u}_{\text{ex}}$. Figure 3 shows the amplitude $|\boldsymbol{u}_h|$ of the numerical solution, as well as the amplitude error $|\boldsymbol{u}_{\text{ex}}| - |\boldsymbol{u}_h|$, obtained with $N = 16$ and $p = 3$. Table 2 contains the L^2 norm of the numerical error for different values of p and N, together with estimates of the convergence order.

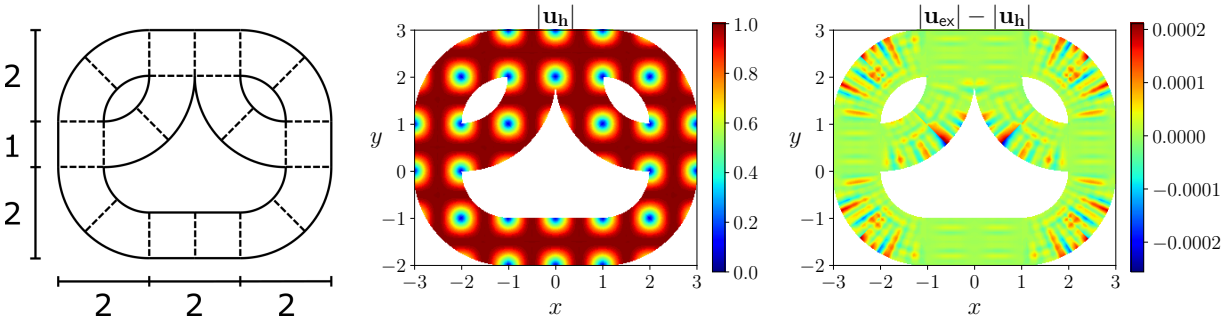

Figure 3: Time harmonic maxwell's equation on a pretzel domain with inhomogeneous essential boundary conditions (4). Multi-patch domain (left), numerical amplitude $|\boldsymbol{u}_h|$ (center) and amplitude error $|\boldsymbol{u}_{\text{ex}}| - |\boldsymbol{u}_h|$ (right) obtained with a polynomial degree $p = 3$ and $N = 16$.

3.3 Time-dependent incompressible Navier-Stokes' equations

We now consider the Navier Stokes' equations for modeling the flow of an incompressible viscous fluid of mass density $\rho = 1$ and kinematic viscosity $\nu > 0$. Given a bounded domain Ω and time interval $(0, T)$, a source term $\boldsymbol{f}$, prescribed velocity $\boldsymbol{u}_D$ on the Dirichlet boundary Γ_D and traction force $\boldsymbol{t}$ on the Neuman boundary Γ_N, and initial conditions $\boldsymbol{u}_0$ in Ω, we want to

(a) Degree $p = 3$

N	L^2 error	L^2 order
4	1.08×10^{-1}	–
8	9.59×10^{-3}	3.49
16	1.09×10^{-3}	3.13
32	1.35×10^{-4}	3.02

(b) Degree $p = 4$

N	L^2 error	L^2 order
4	3.17×10^{-2}	–
8	1.29×10^{-3}	4.62
16	6.70×10^{-5}	4.27
32	4.00×10^{-6}	4.07

(c) Degree $p = 5$

N	L^2 error	L^2 order
4	9.73×10^{-3}	–
8	2.01×10^{-4}	5.60
16	4.65×10^{-6}	5.43
32	1.33×10^{-7}	5.13

Table 2: Refinement study for the time-harmonic Maxwell equation (4) on a pretzel domain with inhomogeneous essential boundary conditions. For various polynomial degrees p and number of cells N along each direction, we report the L^2 norm of the error between the numerical and analytical solutions, and compute the order of convergence between successively refined results.

simultaneously compute the 2D vector velocity field $\boldsymbol{u}$ and the scalar pressure field p, solving

$$\begin{cases} \dfrac{\partial \boldsymbol{u}}{\partial t} - \nu \nabla^2 \boldsymbol{u} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} + \nabla p = \boldsymbol{f} & \text{in } \Omega \times (0,T) \\ \nabla \cdot \boldsymbol{u} = 0 & \text{in } \Omega \times (0,T) \\ \boldsymbol{u} = \boldsymbol{u}_D & \text{on } \Gamma_D \times (0,T) \\ -p\boldsymbol{n} + \nu(\boldsymbol{n} \cdot \nabla)\boldsymbol{u} = \boldsymbol{t} & \text{on } \Gamma_N \times (0,T) \end{cases} \tag{7}$$

where $\boldsymbol{n}$ is the unit vector normal to the boundary. If we let $\boldsymbol{f} \in L^2(\Omega)$ and choose $W := H^1(\Omega)^2 \times L^2(\Omega)$ for the trial and test spaces, we can derive the mixed variational problem

$$\text{Find } (\boldsymbol{u}, p) \in W \text{ s.t.} \quad \int_\Omega \left(\frac{\partial \boldsymbol{u}}{\partial t} \cdot \boldsymbol{v} + \nu \nabla \boldsymbol{u} : \nabla \boldsymbol{v} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} \cdot \boldsymbol{v} - q\, \nabla \cdot \boldsymbol{u} - p\, \nabla \cdot \boldsymbol{v} \right) = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} + \int_{\Gamma_n} \boldsymbol{t} \cdot \boldsymbol{v} \quad \forall (\boldsymbol{v}, q) \in W.$$

After discretization of the time variable $t_n = n\Delta t$, we apply the Crank-Nicolson scheme to obtain a non-linear equation for $\boldsymbol{u}^{n+1} \approx \boldsymbol{u}(t_{n+1})$, given the previous solution $\boldsymbol{u}^n \approx \boldsymbol{u}(t_n)$. At each time step we solve this equation iteratively, using Newton's method with $\boldsymbol{u}^n$ as first guess.

On a mapped multipatch domain we follow a strategy similar to the one for Maxwell's equations. On the reference patch $\hat{\Omega}$ we approximate the velocity and pressure functions using Taylor-Hood B-spline spaces, as suggested in [6] and [12], which can be regarded as smooth generalizations of the classical Taylor-Hood pairs of finite elements [17]:

$$\hat{\boldsymbol{V}}_h^{\mathrm{TH}} := \left(\mathcal{S}^{p_1,p_2}_{\alpha_1,\alpha_2}\right)^2 = \left(\mathcal{S}^{p_1}_{\alpha_1} \times \mathcal{S}^{p_2}_{\alpha_2}\right)^2, \qquad \hat{Q}_h^{\mathrm{TH}} := \mathcal{S}^{p_1-1,p_2-1}_{\alpha_1,\alpha_2} = \mathcal{S}^{p_1-1}_{\alpha_1} \times \mathcal{S}^{p_2-1}_{\alpha_2}.$$

On each patch $\Omega_k = F_k(\hat{\Omega})$ we construct a conformal discretization using the appropriate push-forward operators, and define the global finite element space by juxtaposition of the local ones:

$$\begin{cases} V_h^0(\Omega_k) := \mathcal{F}_k^0(\hat{\boldsymbol{V}}_h^{\mathrm{TH}}) \subset H^1(\Omega_k)^2, \\ V_h^2(\Omega_k) := \mathcal{F}_k^2(\hat{Q}_h^{\mathrm{TH}}) \subset L^2(\Omega_k), \end{cases} \quad \text{and} \quad \begin{cases} V_h^0 := \left\{ \boldsymbol{v} \in [L^2(\Omega)]^2 : \boldsymbol{v}|_{\Omega_k} \in V_h^0(\Omega_k) \right\} \not\subset H^1(\Omega)^2, \\ V_h^2 := \left\{ q \in L^2(\Omega) : q|_{\Omega_k} \in V_h^2(\Omega_k) \right\} \subset L^2(\Omega). \end{cases}$$

Our numerical solution will then be $(\boldsymbol{u}_h, p_h) \in V_h^0 \times V_h^2$. Since V_h^0 is not globally conformal, we penalize any jumps of $\boldsymbol{u}_h$ across the interfaces using Nitsche's method. We also use this method to impose Dirichlet boundary conditions. The weak formulation is detailed in [10, Eq. 7.9].

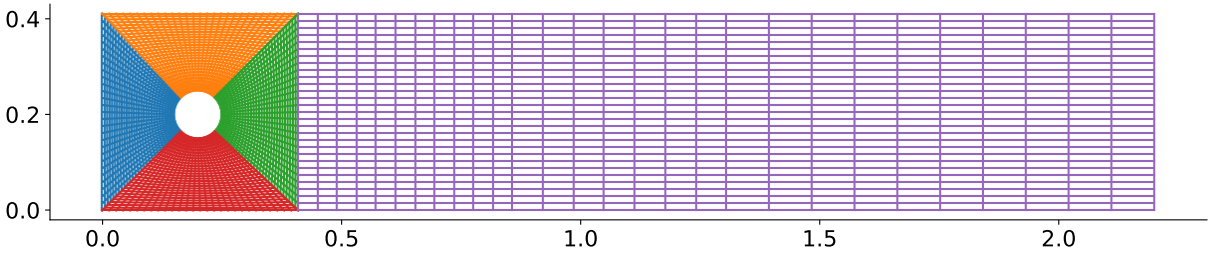

Figure 4: Multi-patch computational domain for the time-dependent Navier-Stokes problem (7). The colors identify different NURBS patches. The grid is defined by the splines' breakpoints.

As a test case we simulate the unsteady flow in a pipe with a circular obstacle [1]. The geometry is depicted in Figure 4 and is defined as $\Omega = (0, 2.2) \times (0, 0.41) \backslash B_r(0.2, 0.2)$, with the radius being $r = 0.05$. The kinematic viscosity is $\nu = 0.001$. We require no-slip boundary conditions on the lower wall $\Gamma_1 = (0, 2.2) \times \{0\}$, upper wall $\Gamma_2 = (0, 2.2) \times \{0.41\}$, and obstacle's wall $S = \partial B_r(0.2, 0.2)$: $\boldsymbol{u}|_{\Gamma_1 \cup \Gamma_2 \cup S} = \boldsymbol{0}$. On the left edge $\Gamma_3 = \{0\} \times (0, 0.41)$ we prescribe a parabolic inflow profile with time-oscillating amplitude: $\boldsymbol{u}|_{\Gamma_3} = (4Uy(0.41 - y)/0.41^2, 0)^T$ with $U(t) = 1.5 \sin(\pi t/8)$. On the right edge $\Gamma_4 = \{2.2\} \times (0, 0.41)$ we prescribe free-flow Neumann boundary conditions with $\boldsymbol{t} = \boldsymbol{0}$. As initial conditions we set $\boldsymbol{u}_0 = \boldsymbol{0}$ everywhere in Ω.

We compute the numerical solution using a polynomial degree $p = 2$ and $N = 32$ elements along both directions of each patch. The grid defined by the spline breakpoints can be seen in Figure 4. The magnitude of the velocity field $|\boldsymbol{u}_h|$ at different times is shown in Figure 5.

4 PERFORMANCE

4.1 Strong parallel scaling

We now solve the 3D Poisson equation in the unit cube, with homogeneous Dirichlet boundary conditions, and use a large computational grid distributed across many MPI processes. We run the tests on Cobra [2], a distributed memory supercomputer with 2 × 20-core Intel Xeon Gold 6148 (Skylake) processors and 192 GB of memory per node, which uses a 100 Gb/s OmniPath interconnect network. For a given problem size, indicated by the number of cells *ncells* and the degree p, we perform a strong scaling analysis by progressively increasing the number of MPI processes *nprocs* from 32 (on a single node) to 4096 (on 128 nodes, with 32 processes per node).

We compare the performance of Psydac on various problem sizes with that of PetIGA [8], a well-established framework for high-performance isogeometric analysis. For both libraries we use the same Intel compiler suite (Fortran for Psydac, and C for PetIGA) and optimization flags. In Figure 6 we show the speedup in the assembly of the stiffness matrix compared to PetIGA: with the exception of a single case, Psydac is consistently 2 to 6 times faster. Certain tests could not be run with PetIGA because of excessive memory usage (see next section). In Figure 7 we show the speedup of the matrix-vector product: here Psydac compares favorably with PetIGA as the polynomial degree is increased, and in many cases it is more than 1.5 times faster.

We think that Psydac achieves this high level of performance thanks to its dedicated data structure for sparse matrix storage, referred to as "stencil format", which provides a simple memory access pattern that is known at compile time. Further, Psydac strives to generate the

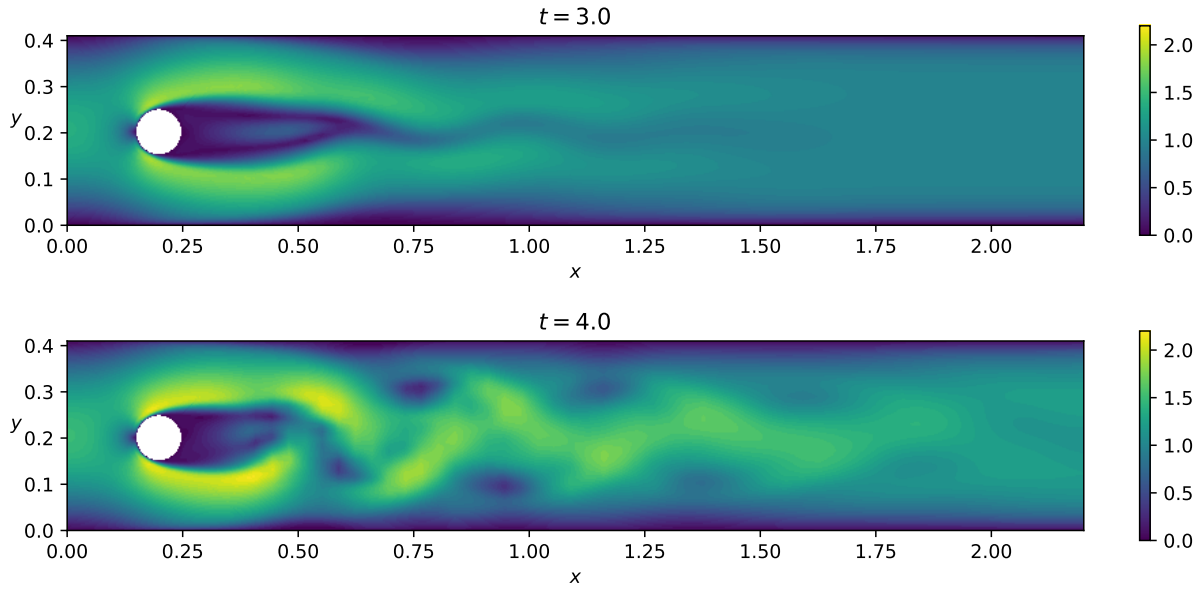

Figure 5: Incompressible 2D Navier-Stokes problem (7): unsteady flow in a pipe with a circular obstacle. We plot the magnitude of the discrete velocity field $|\boldsymbol{u}_h|$ at different time instants. The results are obtained using a polynomial degree $p = 2$ and $N = 32$ elements along both directions of each patch (see Figure 4 for the grid), and Crank-Nicolson time stepping.

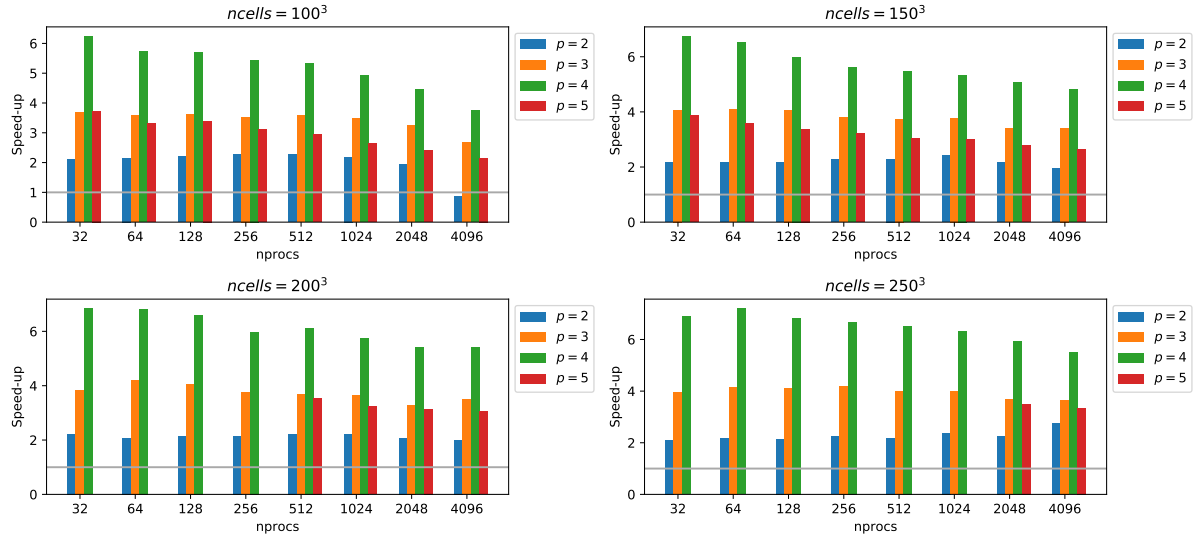

Figure 6: Stiffness matrix assembly for 3D Poisson problem. Speedup w.r.t. PetIGA (grey line).

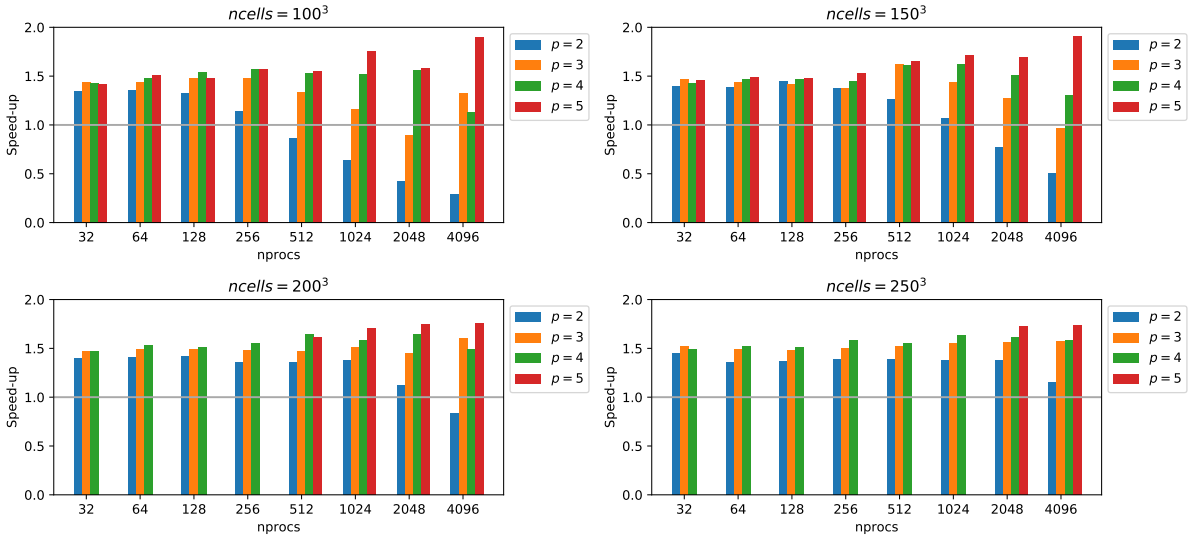

Figure 7: Matrix-vector product for 3D Poisson problem. Speedup w.r.t. PetIGA (grey line).

simplest possible Fortran code, which can be easily optimized by a good Fortran compiler.

4.2 Memory usage

In Figure 8 we show the maximum memory usage of Psydac and PetIGA per node using different numbers of cells and a fixed degree $p = 4$. For this test we run the simulation with 32 MPI processes on a single "fat" node with 768 GB RAM DDR4. Psydac consistently uses far less memory than PetIGA, with a memory reduction that exceeds a factor of 5 in some cases.

We obtain these results thanks to Psydac's own stencil format for sparse matrix storage, which takes advantage of the tensor-product structure of our spline basis functions to store only the non-zero matrix entries with no memory overhead. On the other hand PetIGA uses PETSc [5] to represent the sparse matrices in CSR format, which stores the column index of the non-zero matrix entries as well as their values: this results in a considerable memory overhead in 3D. PETSc may also require additional temporary memory during the matrix assembly process.

5 CONCLUSIONS

We have given an overview of the Psydac framework, and shown its capabilities in solving partial differential equations using isogeometric analysis. It supports multi-patch domains with analytical or NURBS mapping, and compatible finite element discretizations using tensor-product splines. This Python 3 library effectively combines three main ingredients: 1) a domain specific language for variational formulations, 2) automatic generation of Python and Fortran code, and 3) Hybrid MPI + OpenMP parallelization. We have run performance benchmarks against the well-established tool PetIGA, and shown that our sparse matrix format has a clear advantage in terms of memory footprint. The results show that Psydac is a user-friendly environment for solving a wide variety of partial differential equations in large-scale simulations.

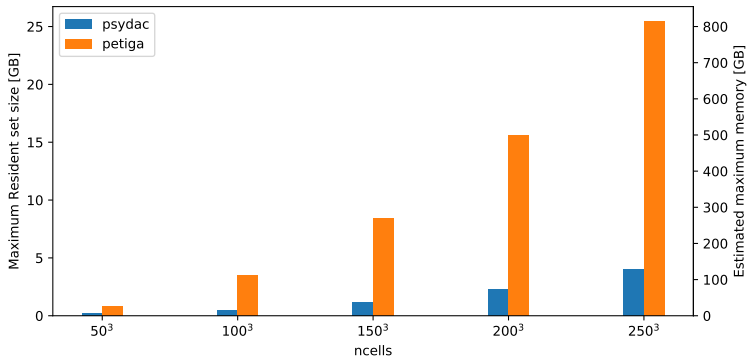

Figure 8: Maximum memory used by Psydac and PetIGA for a 3D Poisson problem.

REFERENCES

[1] Featflow web site. http://www.mathematik.tu-dortmund.de/ featflow/en/benchmarks/cfdbenchmarking/flow/dfg_benchmark3_re100.html. Accessed: 2022-06-20.

[2] The supercomputer Cobra. https://docs.mpcdf.mpg.de/doc/computing/cobra-user-guide.html#system-overview.

[3] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[4] D. N. Arnold. An interior penalty finite element method with discontinuous elements. *SIAM J. Numer. Anal.*, 19(4):742–760, 1982.

[5] S. Balay et al. PETSc Web page. `https://petsc.org/`, 2021.

[6] A. Buffa, C. de Falco, and G. Sangalli. IsoGeometric analysis: stable elements for the 2D Stokes equation. *Internat. J. Numer. Methods Fluids*, 65(11-12):1407–1422, 2011.

[7] A. Buffa and I. Perugia. Discontinuous Galerkin approximation of the Maxwell eigenproblem. *SIAM J. Numer. Anal.*, 44(5):2198–2226, 2006.

[8] N. O. Collier, L. Dalcín, and V. M. Calo. Petiga: High-performance isogeometric analysis. *CoRR*, abs/1305.4452, 2013.

[9] L. Dalcin, N. Collier, P. Vignal, A. Côrtes, and V. Calo. PetIGA: A framework for high-performance isogeometric analysis. *Comput. Methods Appl. Mech. Engrg.*, 308:151–181, 2016.

[10] J. Evans and T. Hughes. Isogeometric divergence-conforming B-splines for the steady Navier-Stokes equations. *J. Comput. Phys.*, 241:141–167, 2013.

[11] F. Hecht. New development in FreeFem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[12] B. S. Hosseini, M. Möller, and S. Turek. Isogeometric analysis of the Navier–Stokes equations with Taylor–Hood B-spline elements. *Appl. Math. Comput.*, 267:264–281, 2015.

[13] A. Logg and G. N. Wells. DOLFIN: automated finite element computing. *CoRR*, abs/1103.6248, 2011.

[14] K. R. Long. Sundance rapid prototyping tool for parallel PDE optimization. In L. T. Biegler, M. Heinkenschloss, O. Ghattas, and B. van Bloemen Waanders, editors, *Large-Scale PDE-Constrained Optimization*, pages 331–341. Springer Berlin Heidelberg, 2003.

[15] C. Prud'homme et al. Feel++: Finite element embedded library in C++, 2013–. https://github.com/feelpp/feelpp.

[16] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake. *ACM Trans. Math. Softw.*, 43(3):1–27, 2017.

[17] C. Taylor and P. Hood. A numerical solution of the Navier-Stokes equations using the finite element technique. *Comput. & Fluids*, 1(1):73–100, 1973.