MOX-Report No. 43/2026

A validated MATLAB framework for sparse vectorized finite element assembly

Micheletti, S.

MOX, Dipartimento di Matematica
Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)

mox-dmat@polimi.it https://mox.polimi.it

A validated MATLAB framework for sparse vectorized finite element assembly

S. Micheletti

June 6, 2026

MOX - Dipartimento di Matematica, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, IT, stefano.micheletti@polimi.it

Abstract

We present a compact MATLAB framework for finite element prototyping based on sparse vectorized assembly. The purpose of the code is not to compete with large-scale production finite element libraries, but to expose, in a transparent and reproducible way, the algebraic structure that connects a variational formulation with an efficient implementation in a high-level language. The codebase reorganizes earlier MATLAB prototypes into functions with explicit inputs and outputs, examples, regression tests and validation scripts. The framework covers piecewise linear and quadratic finite elements on triangular meshes, variable-coefficient diffusion–reaction problems with quadrature, a vectorized tetrahedral implementation for the three-dimensional Poisson equation, and a Mini-element discretization of the incompressible Navier–Stokes equations. Numerical experiments illustrate both performance and reproducibility. In particular, the DFG flow-around-a-cylinder benchmark is used to validate the incompressible-flow module and to discuss the different mesh sensitivities of drag, lift and pressure-difference outputs.

Keywords: Finite Element Method; sparse assembly; vectorization; MATLAB; elliptic equations; Navier-Stokes equations; scientific software.
MSC (2020): 68N15, 35Q30, 65N30, 65M60.

1 Introduction

MATLAB remains widely used in scientific computing, numerical-analysis education and rapid prototyping. Its role, however, has changed. Modern finite element computations can rely on expressive domain-specific systems and large-scale libraries, including FreeFEM [11], automated problem-solving environments and compiled high-performance libraries. Therefore, an article on MATLAB finite element programming should no longer be motivated by the claim that

MATLAB can replace such tools in production simulations. The relevant question is different: can one design a compact, inspectable and reproducible MATLAB codebase that makes the sparse algebraic structure of finite element methods explicit while remaining efficient enough for advanced teaching and research prototyping?

This paper answers this question positively for a class of model problems in two and three space dimensions. The sparse vectorized assembly paradigm considered here was first explored in the MOX technical report [14]. The present contribution is not a mere update of that prototype. It reorganizes the original ideas into a self-contained software framework with documented data structures, functional interfaces, examples, regression tests and benchmark validations. The performance data and the validation results used as current evidence below are generated by the released codebase accompanying this manuscript. In particular, the DFG 2D-1, 2D-2 and 2D-3 cylinder benchmarks reported below have been regenerated with the current validation layer on the analytic radially refined mesh described in Section 10; earlier timing data and old prototype values are not used as current evidence.

The basic idea is simple. Local finite element matrices are not assembled by looping over elements and inserting one element contribution at a time into a global sparse matrix. Instead, elementwise geometric quantities, local-to-global indices and local matrix entries are stored as arrays. The global matrix is then created through sparse assembly from row, column and value vectors. This programming style is particularly natural in MATLAB, where vectorized array operations and sparse matrices are part of the language design. The resulting code retains a direct connection with the underlying variational formulation while avoiding the main bottleneck of elementary interpreted loops.

The contribution of the paper is threefold. First, we describe a minimal but extensible mesh and degree-of-freedom organization for triangular and tetrahedral finite elements in MATLAB. Second, we show how the same sparse vectorized assembly pattern applies to several levels of complexity: the Poisson problem, variable-coefficient diffusion–reaction equations, quadratic finite elements, tetrahedral meshes and block systems arising from incompressible flow. Third, we complement the implementation with tests and validation scripts, including small exact-solution tests, regression checks and an independent DFG/Turek–Schäfer cylinder benchmark layer for the Navier–Stokes module.

The code is intended for users who want to see and modify the algebraic core of a finite element method. Its target is therefore different from that of large production libraries. The emphasis is on transparency, reproducibility, compactness and extensibility rather than on parallel scalability. This makes the framework useful in graduate courses, in early-stage numerical experiments, and as a bridge between textbook finite element formulations and more sophisticated software environments.

The paper is organized as follows. Section 2 positions the work with respect to existing MATLAB FEM implementations and vectorized assembly literature. Section 3 summarizes the software architecture. Section 4 introduces the mesh data structure. Sections 5–7 present elliptic model problems with linear and quadratic elements. Section 8 records the extension to tetrahedral meshes. Section 9 describes the incompressible-flow module, and Section 10

discusses testing, reproducibility and the current DFG 2D-1, 2D-2 and 2D-3 benchmark results. Conclusions are drawn in Section 11.

2 Related work and positioning

Compact MATLAB finite element codes have a long tradition. The well-known work of Alberty, Carstensen and Funken [1] showed how a complete finite element implementation can be written in a few dozen lines, and remains a useful reference for educational purposes. More comprehensive MATLAB packages include iFEM, which emphasizes adaptive finite element methods, multigrid solvers and sparse-matrix based programming [3], and MILAMIN, which demonstrated that optimized MATLAB implementations can handle large two-dimensional finite element problems efficiently [5].

The vectorized assembly of finite element matrices in MATLAB and related vector languages has also been studied systematically. Rahman and Valdman [15] proposed fast assembly strategies for nodal finite elements in two and three dimensions. Cuvelier, Japhet and Scarella [4] analyzed efficient assembly in vector languages. More recently, Tchuigwa et al. [18] introduced Vectfem, a generalized MATLAB-based vectorized algorithm for matrix and force-vector assembly in linear elasticity.

The present framework is complementary to these works. It does not aim to provide the broad adaptivity infrastructure of iFEM, the large-problem specialization of MILAMIN, or the generality of recent elasticity-oriented assembly frameworks. Instead, it focuses on a compact set of finite element examples in which the same sparse vectorized pattern is visible from the mathematical formulation down to the MATLAB implementation. The inclusion of quadratic elements, three-dimensional Poisson problems and Mini-element Navier–Stokes simulations is meant to demonstrate extensibility while preserving readability.

3 Software design and reproducibility

The released codebase is organized as a small MATLAB project rather than as a collection of scripts depending on the base workspace. Core routines are placed in `src/`, examples in `examples/`, automated checks in `tests/`, validation utilities in `validation/`, and reproducible performance benchmarks in `benchmarks/`. The startup file adds the relevant folders to the MATLAB path. A minimal workflow is

```
startup
runAllTests
```

Optional examples, some of which use PDE Toolbox mesh-generation functions to reproduce the classical `p`, `e`, `t` data layout, include

```
benchmarkPoissonP1
solvePoissonP2Square
```

The core functions use plain MATLAB sparse matrices, whereas the tests are designed not to require PDE Toolbox.

The most important design choice is that assembly and solver routines are functions with explicit inputs and outputs. The linear Poisson module includes a vectorized sparse assembler and a loop-based reference implementation. The quadratic example is organized around mesh enrichment, basis evaluation and a diffusion–reaction solver. The three-dimensional module separates tetrahedral geometry from Dirichlet Poisson assembly. The incompressible-flow module separates Stokes blocks, convection blocks, degree-of-freedom numbering and the cylinder time loop.

The folder `tests/` contains small smoke and regression tests for the Poisson, quadratic, three-dimensional and Mini-element modules. The folder `validation/` contains scripts for independent numerical benchmark post-processing, including the DFG/Turek–Schäfer cylinder data, drag/lift normalization checks and mesh diagnostics. The folder `benchmarks/` contains the current-machine timing layer; it records the computing environment and writes MATLAB, CSV and LaTeX summaries of the performance experiments. The reproducible evidence reported in this paper is based exclusively on the released codebase, its automated tests, its validation scripts and the benchmark data described below.

4 Mesh and degree-of-freedom data structures

Let $\Omega \subset \mathbb{R}^2$ be a bounded polygonal domain and let $\mathcal{T}_h$ be a conforming triangulation of $\overline{\Omega}$. The two-dimensional routines use the classical PDE Toolbox-style arrays `p`, `e` and `t`. If `Np`, `Ne` and `Nt` denote the numbers of vertices, boundary edges and triangles, respectively, then `p` has size $2 \times$ `Np`, `e` stores boundary-edge information, and `t` stores element connectivity. Only a small subset of these arrays is needed by the core routines:

1. `p(:,i)` contains the coordinates of the i-th vertex;
2. `t(1:3,k)` contains the global indices of the three vertices of the k-th triangle;
3. `e(1:2,j)` contains the endpoints of the j-th boundary edge;
4. boundary labels stored in `e` are used to select Dirichlet, Neumann, inflow, outflow, wall and cylinder boundaries.

Figure 1 illustrates this convention on a small mesh. The precise ordering of the boundary edges in `e` is not required by the scalar elliptic routines. For boundary post-processing and for the incompressible-flow benchmark, however, the code explicitly classifies boundary nodes from their labels and geometry.

This format is intentionally elementary. It avoids hidden mesh objects and makes the local-to-global map visible in the code. For $\mathbb{P}_1$ scalar problems the degrees of freedom coincide with vertices. For $\mathbb{P}_2$ problems the function `p1ToP2.m` augments the vertex set with edge-midpoint

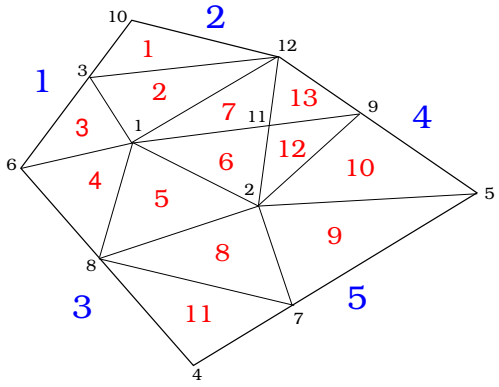

Figure 1: Example of a triangular mesh and numbering of the geometrical entities: boundary portions (blue), vertices (black) and elements (red).

degrees of freedom and builds the corresponding six-node element connectivity. For the Mini-element discretization used in Section 9, the scalar velocity space is $\mathbb{P}_1$ enriched with one bubble function per triangle, while the pressure space is $\mathbb{P}_1$. Thus the scalar velocity degrees of freedom are the vertices plus the elements, and the pressure degrees of freedom are the vertices. The helper routine `velocityPressureDofs.m` stores these maps.

The three-dimensional routines use the analogous tetrahedral connectivity: `p` is $3 \times$ `Np`, `t(1:4,k)` gives the four vertices of the k-th tetrahedron, and `e` stores boundary faces with labels. The same design principle is used in two and three dimensions: global algebraic objects are obtained by combining elementary geometry arrays, local-to-global indices and sparse assembly.

5 Sparse vectorized assembly for a P1 Poisson problem

We start from the model problem

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

The continuous weak formulation is the natural starting point for the implementation. If

$$H_0^1(\Omega) = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}, \qquad H_0^1(\Omega) + g_D = \{v \in H^1(\Omega) : v - g_D \in H_0^1(\Omega)\},$$

then the problem reads: find $u \in H_0^1(\Omega) + g_D$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\mathbf{x} = \int_\Omega f v \, \mathrm{d}\mathbf{x} \qquad \forall v \in H_0^1(\Omega). \tag{2}$$

For an integer $r \geq 1$, let

$$V_h^r = \{v_h \in C^0(\overline{\Omega}) : v_h|_K \in \mathbb{P}_r(K) \text{ for all } K \in \mathcal{T}_h\}, \qquad V_{h,0}^r = V_h^r \cap H_0^1(\Omega).$$

The discrete formulation is: find $u_h \in V_{h,0}^r + g_{D,h}$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v_h \, \mathrm{d}\mathbf{x} = \int_\Omega f v_h \, \mathrm{d}\mathbf{x} \qquad \forall v_h \in V_{h,0}^r, \tag{3}$$

where $g_{D,h} \in V_h^r$ interpolates or approximates the boundary datum. In the code path discussed here we set $r = 1$. If $\{\phi_i\}$ denotes the nodal basis associated with the free degrees of freedom, then the algebraic system is obtained from

$$A_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, \mathrm{d}\mathbf{x}, \qquad F_i = \int_\Omega f\phi_i \, \mathrm{d}\mathbf{x} - \sum_{j \in \mathcal{N}_{\partial\Omega}} g_D(\mathbf{x}_j) \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, \mathrm{d}\mathbf{x}. \tag{4}$$

This explicit weak-to-algebraic passage is useful in the software paper because the row, column and value arrays constructed by MATLAB are exactly the sparse representation of A.

On a triangle K with local basis functions $\lambda_1, \lambda_2, \lambda_3$, the local stiffness entries are

$$A_{ij}^K = \int_K \nabla\lambda_j \cdot \nabla\lambda_i \, \mathrm{d}\mathbf{x} = |K| \, \nabla\lambda_j|_K \cdot \nabla\lambda_i|_K, \qquad i, j = 1, 2, 3. \tag{5}$$

The gradients are constant on each element. The routine `triangleGeometry.m` computes the element areas and the arrays of basis-function derivatives for all triangles at once. The assembler `assemblePoissonP1.m` then forms the off-diagonal entries

$$a_{12}, \quad a_{23}, \quad a_{31}$$

as row vectors over the elements, recovers the diagonal entries from the zero row-sum property of the element stiffness matrix, and calls `sparse` with vectorized row, column and value arrays. The load vector is assembled in the same way using midpoint quadrature. In schematic form, the assembly pattern is

Code 1: Sparse vectorized $\mathbb{P}_1$ assembly pattern.

```
geom = triangleGeometry(p,t);
% elementwise arrays of local stiffness entries
A = sparse(n1,n2,a12,Np,Np) + sparse(n2,n3,a23,Np,Np) ...
+ sparse(n3,n1,a31,Np,Np);
A = A + A.';
A = A + sparse(n1,n1,diag1,Np,Np) ...
+ sparse(n2,n2,diag2,Np,Np) ...
+ sparse(n3,n3,diag3,Np,Np);
F = sparse(n1,1,elementLoad,Np,1) ...
+ sparse(n2,1,elementLoad,Np,1) ...
+ sparse(n3,1,elementLoad,Np,1);
```

The solver `solvePoissonP1Dirichlet.m` applies the Dirichlet condition by eliminating the boundary degrees of freedom. The mixed-boundary variant `solvePoissonP1Mixed.m` reuses the same volume assembly and adds boundary-edge quadrature on the Neumann part. A loop-based reference routine, `assemblePoissonP1Loop.m`, is included in the repository. It provides a

simple internal check of the vectorized assembly and a transparent comparison between element-by-element insertion and sparse triplet assembly.

The important point is not the particular Poisson equation, but the programming pattern. Once the geometric quantities and the local-to-global maps are available as arrays, the transition from the variational form to the sparse matrix is visible and compact. This is the pattern reused in the more involved examples below.

5.1 Current P1 Poisson performance benchmark

The same Poisson example provides the basic performance benchmark for the sparse assembly paradigm. The timings in this subsection were regenerated from the refactored repository by `benchmarks/runAllBenchmarks.m`; no timing data from the earlier prototype, the old manuscript or the MOX report are used. Each entry is measured after a warm-up run and summarized by the median over five repetitions; the benchmark result files retain additional dispersion statistics. The computing environment for the present draft is reported in Table 1. The Navier–Stokes runtime benchmark is available in the benchmark driver but is not included in the default timing run because it is substantially more expensive than the scalar tests.

Table 1: Benchmark computing environment used for the current-machine timing tables.

Item	Value
Date	04-Jun-2026 19:21:35
MATLAB	26.1.0.3133997 (R2026a Prerelease Update 3)
Operating system	Microsoft Windows 11 Home
Computer	PCWIN64, AMD64 architecture
CPU	AMD64 Family 25 Model 116 Stepping 1, AuthenticAMD
Computational threads	8
Detected cores	8
RAM	64 GB installed; 41.97 GB available at benchmark start
Git commit	not available

The available-memory value is the amount reported by MATLAB/Windows at the beginning of the benchmark session and should not be interpreted as the total physical memory installed on the machine.

Table 2: Current P1 Poisson assembly benchmark: vectorized sparse assembly versus loop-over-elements reference assembly.

mesh	vertices	triangles	dofs	nnz	vectorized (s)	loop (s)	speedup
square $n = 16$	289	512	289	1377	$7.551 \cdot 10^{-4}$	$6.409 \cdot 10^{-3}$	8.49
square $n = 32$	1089	2048	1089	5313	$7.862 \cdot 10^{-4}$	$1.720 \cdot 10^{-2}$	21.9
square $n = 64$	4225	8192	4225	20865	$1.991 \cdot 10^{-3}$	$1.196 \cdot 10^{-1}$	60.1
square $n = 96$	9409	18432	9409	46657	$3.572 \cdot 10^{-3}$	$5.105 \cdot 10^{-1}$	143

Table 3: Current P1 Poisson assemble-and-solve benchmark after elimination of homogeneous Dirichlet degrees of freedom.

mesh	nodes	elements	free dofs	nnz	median (s)
square $n = 16$	289	512	225	1065	$1.590 \cdot 10^{-3}$
square $n = 32$	1089	2048	961	4681	$1.280 \cdot 10^{-3}$
square $n = 64$	4225	8192	3969	19593	$5.981 \cdot 10^{-3}$
square $n = 96$	9409	18432	9025	44745	$2.526 \cdot 10^{-2}$

The current timings confirm the qualitative message of the original prototype: for the same mesh, sparse vectorized assembly is substantially faster than element-by-element insertion into a sparse matrix. The benchmark is deliberately modest in size so that it can be regenerated quickly by users and by continuous-integration checks.

6 Variable coefficients and quadrature

The next level of complexity is represented by the diffusion–reaction problem with mixed boundary conditions,

$$\begin{cases} -\nabla \cdot (\mu \nabla u) + \sigma u &= f & \text{in } \Omega, \\ u &= g_D & \text{on } \Gamma_D, \\ \mu\, \partial u/\partial n &= g_N & \text{on } \Gamma_N, \end{cases} \tag{6}$$

where $\overline{\Gamma}_D \cup \overline{\Gamma}_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \emptyset$. The weak form contains the bilinear form

$$a(u, v) = \int_\Omega \mu \nabla u \cdot \nabla v \, \mathrm{d}\mathbf{x} + \int_\Omega \sigma u v \, \mathrm{d}\mathbf{x} \tag{7}$$

and the right-hand side

$$\ell(v) = \int_\Omega f v \, \mathrm{d}\mathbf{x} + \int_{\Gamma_N} g_N v \, \mathrm{d}s. \tag{8}$$

The corresponding finite element problem is: find $u_h \in V^r_{h,\Gamma_D} + g_{D,h}$, with

$$V^r_{h,\Gamma_D} = V^r_h \cap H^1_{\Gamma_D}(\Omega), \qquad H^1_{\Gamma_D}(\Omega) = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\},$$

such that

$$a(u_h, v_h) = \ell(v_h) \qquad \forall v_h \in V^r_{h,\Gamma_D}. \tag{9}$$

For a local basis $\{\varphi_i\}_{i=1}^{n_{loc}}$, the element matrix is split into diffusion and reaction parts,

$$A^K_{ij} = \int_K \mu\, \nabla\varphi_j \cdot \nabla\varphi_i \, \mathrm{d}\mathbf{x}, \qquad M^K_{ij} = \int_K \sigma\, \varphi_j \varphi_i \, \mathrm{d}\mathbf{x}, \tag{10}$$

and the load contribution is

$$F^K_i = \int_K f \varphi_i \, \mathrm{d}\mathbf{x}, \qquad F^E_i = \int_E g_N \varphi_i \, \mathrm{d}s \quad (E \subset \Gamma_N). \tag{11}$$

Equations (9)–(11) are the formulas mirrored in the quadrature routines. For nonconstant coefficients, the local contributions are no longer reduced to element areas and constant gradients alone. The code therefore introduces quadrature rules explicitly. Symmetric rules on the reference triangle [6, 2] and one-dimensional Gauss–Legendre rules on boundary edges are provided by small helper routines.

For each triangle, physical quadrature points are generated by the affine map from the reference element. The coefficient functions μ, σ, the source term and the basis functions are evaluated in vectorized form at all quadrature points and all elements. Small fixed-size loops over local basis functions remain in some routines. This is deliberate: the loops have length three or six and do not scale with the mesh. The expensive dimension, namely the number of elements, is treated by array operations and sparse assembly.

This section is implemented in the quadratic solver described next, but the same design also applies to linear elements with variable coefficients. The separation between geometry, quadrature, basis evaluation and algebraic assembly makes the implementation easier to test and to adapt to new coefficients.

7 Quadratic finite elements

For the $\mathbb{P}_2$ diffusion–reaction example, the finite element space is formed by continuous functions that restrict to quadratic polynomials on each triangle. The local degrees of freedom are associated with the three vertices and the three edge midpoints. Starting from a $\mathbb{P}_1$ mesh, `p1ToP2.m` constructs the enlarged node array and the six-node connectivity. Edge-midpoint identification is performed globally, so neighboring triangles share the same midpoint degree of freedom.

The conversion from a linear to a quadratic mesh is a small but important preprocessing step. For each triangle with local vertices n_1, n_2, n_3, the routine first forms the three unoriented edge pairs

$$(n_2, n_3), \qquad (n_3, n_1), \qquad (n_1, n_2),$$

which correspond to the midpoint degrees of freedom of the local $\mathbb{P}_2$ ordering used in the code. The essential issue is that a midpoint is a degree of freedom attached to a geometric edge, not to a particular triangle. Thus the two occurrences of the same interior edge, possibly with opposite orientations, must be mapped to one and the same global index.

This edge-identification step can be interpreted through a pairing-function idea. Given two distinct vertex indices i and j, an unordered edge may be encoded by a symmetric integer key. One possible choice, adapted from Szudzik's pairing function [17], is

$$\pi(i, j) = \begin{cases} j^2 + i, & i < j, \\ i^2 + j, & j < i, \end{cases}$$

where the case $i = j$ never occurs for a mesh edge. Such a key is independent of the local orientation of the edge and therefore provides a simple way to recognize repeated edges. Since

the values of π are unique but not consecutive, they must then be compressed to consecutive identifiers $1, \ldots, N_E$, where N_E is the total number of distinct edges. The current implementation realizes the same idea without relying explicitly on large integer keys: it sorts the two endpoints of each local edge, builds the $3N_T \times 2$ list of all sorted edge pairs, and applies a stable unique-row map. If N_v is the number of original vertices and e_m is the resulting global edge identifier, then the associated midpoint degree of freedom is numbered as $N_v + e_m$. The new element connectivity is therefore obtained by appending to the three vertex indices the three midpoint indices in the order

$$(n_1, n_2, n_3, m_{23}, m_{31}, m_{12}).$$

The coordinate array is enlarged by adding the arithmetic midpoints of all unique edges. Boundary edges are updated in the same way: besides their two endpoints, they also store the corresponding midpoint degree of freedom and the boundary label. This makes the treatment of essential and natural boundary conditions consistent with the six-node element connectivity. The edge enumeration itself is not performance critical; it is performed once before the assembly phase and is kept explicit in order to make the construction of the quadratic degrees of freedom inspectable.

Code 2: Global edge identification used in the $\mathbb{P}_1$-to-$\mathbb{P}_2$ conversion.

```
localEdges = [t([2 3],:), t([3 1],:), t([1 2],:)];
sortedEdges = sort(localEdges,1).';
[edges,~,edgeId] = unique(sortedEdges,'rows','stable');
midpointDofs = Nv + reshape(edgeId,3,[]);
t2 = [t; midpointDofs];
p2 = [p, 0.5*(p(:,edges(:,1)) + p(:,edges(:,2)))];
```

On the reference triangle, the quadratic Lagrange basis is evaluated by `basisP2.m`. If (ξ, η) are reference coordinates and $\lambda_1 = 1 - \xi - \eta$, $\lambda_2 = \xi$, $\lambda_3 = \eta$, then the six basis functions are

$$\lambda_1(2\lambda_1 - 1), \quad \lambda_2(2\lambda_2 - 1), \quad \lambda_3(2\lambda_3 - 1), \quad 4\lambda_2\lambda_3, 4\lambda_3\lambda_1, 4\lambda_1\lambda_2. \tag{12}$$

The routine also returns reference derivatives, which are transformed to physical derivatives by the inverse transpose of the Jacobian on each triangle. The weak problem is still (9); only the trial and test space changes from V_h^1 to V_h^2. Consequently the local arrays now have size 6×6:

$$A_{ij}^K = \int_K \mu \, \nabla\varphi_j \cdot \nabla\varphi_i \, \mathrm{d}\mathbf{x}, \qquad M_{ij}^K = \int_K \sigma \, \varphi_j \varphi_i \, \mathrm{d}\mathbf{x}, \qquad i, j = 1, \ldots, 6. \tag{13}$$

The solver assembles the stiffness, mass and load terms with Dunavant quadrature. Neumann conditions are treated with a one-dimensional quadratic edge basis and Gauss–Legendre quadrature on boundary segments. Dirichlet degrees of freedom are eliminated algebraically.

The associated test checks the construction on a small mesh, and the error routine computes L^2 and H^1 indicators by quadrature. The square-domain example is the modern, function-based counterpart of the quadratic model problem used in the original prototype. The relevance of this example is twofold: it shows that the data structure extends naturally beyond vertex degrees of freedom, and it illustrates how quadrature and boundary conditions can be organized without hiding the finite element algebra.

Table 4: Current P2 reaction–diffusion assemble-and-solve benchmark. The column "nodes" reports the total number of quadratic nodes, including vertices and edge midpoints, whereas "dofs" reports the free degrees of freedom after Dirichlet elimination.

mesh	nodes	elements	dofs	nnz	median (s)
square $n = 8$	289	128	225	2229	$7.781 \cdot 10^{-3}$
square $n = 16$	1089	512	961	10293	$1.550 \cdot 10^{-2}$
square $n = 24$	2401	1152	2209	24245	$3.603 \cdot 10^{-2}$
square $n = 32$	4225	2048	3969	44085	$6.791 \cdot 10^{-2}$

8 Three-dimensional tetrahedral assembly

The three-dimensional module solves the Dirichlet Poisson problem

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^3, \qquad u = g_D \quad \text{on } \partial\Omega, \tag{14}$$

on tetrahedral meshes with $\mathbb{P}_1$ elements. The weak formulation is identical in structure to (2): find $u \in H_0^1(\Omega) + g_D$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\mathbf{x} = \int_\Omega f v \, \mathrm{d}\mathbf{x} \qquad \forall v \in H_0^1(\Omega), \tag{15}$$

and the discrete problem is obtained by replacing the triangular finite element space by continuous piecewise affine functions on tetrahedra. If K is a tetrahedron with local basis functions $\lambda_1, \dots, \lambda_4$, the local stiffness entries are again

$$A_{ij}^K = |K| \, \nabla\lambda_j|_K \cdot \nabla\lambda_i|_K, \qquad i, j = 1, \dots, 4. \tag{16}$$

The function `tetrahedronGeometry.m` computes volumes and gradients for all tetrahedra. The implementation is vectorized with respect to the mesh elements: coordinate differences, cofactors, determinants, volumes and gradients are stored as arrays over the tetrahedra. A small loop of fixed length four is used only to fill the gradients of the local basis functions.

The solver `solvePoissonP1TetraDirichlet.m` then forms the tensor of local 4×4 stiffness matrices in batch. In recent MATLAB releases this is done by page-wise multiplication:

Code 3: Elementwise vectorized tetrahedral assembly.

```
localA = pagemtimes(permute(geom.grad,[2,1,3]), geom.grad) ...
.* reshape(geom.volume,1,1,[]);
elementRows = repmat(reshape(t,4,1,[]),1,4,1);
elementCols = repmat(reshape(t,1,4,[]),4,1,1);
A = sparse(elementRows(:), elementCols(:), localA(:), numNodes, numNodes);
```

The load vector is assembled from barycentric quadrature. Dirichlet conditions are selected from boundary-face labels and imposed by elimination. This module is important for the paper because it shows that the same sparse assembly principle is not tied to two-dimensional triangles. The code does not aim at replacing specialized three-dimensional FEM packages; its role is to expose the algebraic mechanism of tetrahedral assembly in a compact MATLAB implementation.

8.1 Three-dimensional performance data

The three-dimensional example was one of the main reasons for retaining a sparse vectorized MATLAB implementation: it shows that the same assembly idea remains meaningful on tetrahedral meshes, where element loops quickly become expensive. Table 5 reports current-machine timings generated by `runAllBenchmarks.m` for structured unit-cube tetrahedral meshes with six tetrahedra per cube cell. The table is intentionally small; it is a lightweight reproducible benchmark of the released code rather than a large-scale performance claim. The larger external sphere-with-a-hole meshes used in the original three-dimensional study are reported separately in Table 6.

Table 5: Current three-dimensional P1 tetrahedral Poisson assemble-and-solve benchmark.

mesh	nodes	tetrahedra	free dofs	nnz	median (s)
cube $n = 4$	125	384	27	135	$1.127 \cdot 10^{-3}$
cube $n = 6$	343	1296	125	1237	$9.631 \cdot 10^{-4}$
cube $n = 8$	729	3072	343	2107	$1.931 \cdot 10^{-3}$
cube $n = 10$	1331	6000	729	8433	$6.446 \cdot 10^{-3}$

The structured cube tests are complemented by the original sphere-with-a-hole meshes used in the earlier prototype. These meshes provide a more realistic three-dimensional geometry and a substantially wider range of problem sizes. The computational domain is the spherical shell

$$\Omega = \{x \in \mathbb{R}^3 : \ 1 < r = \|x\| < 2\},$$

that is, the ball of radius 2 centered at the origin with the concentric spherical cavity of radius 1 removed. The model problem is the radial Dirichlet Poisson test

$$u(r) = \sin(\pi(r-1)), \qquad f(r) = -\Delta u = \pi^2 \sin(\pi(r-1)) - \frac{2\pi}{r}\cos(\pi(r-1)),$$

with homogeneous Dirichlet data on both boundary components. Equivalently, since $\sin(\pi(r-1)) = -\sin(\pi r)$ and $\cos(\pi(r-1)) = -\cos(\pi r)$, the right-hand side can be written as

$$f(r) = \frac{2\pi\cos(\pi r)}{r} - \pi^2 \sin(\pi r),$$

which is the form used in the benchmark script. Table 6 reports the current-machine timings regenerated with the released codebase. The timing column refers to the wall-clock time of the public solver, which currently measures the complete assemble-and-solve phase as a single block.

The largest original mesh contains more than three million tetrahedra and remains solvable on the present desktop machine with 64 GB of RAM. This result is not meant as a claim of competitiveness with specialized three-dimensional finite element packages; rather, it documents that the same sparse vectorized assembly pattern scales to a nontrivial tetrahedral benchmark within the intended MATLAB prototyping setting.

Table 6: Manual benchmark of the refactored three-dimensional $\mathbb{P}_1$ tetrahedral Poisson solver on the original sphere-with-a-hole meshes. The timings were regenerated with the current codebase on the machine of Table 1. Here `sh_k` abbreviates `spherewithahole_k`. The column "Est. GB" is a pre-run estimate of the memory associated with the main vectorized assembly arrays and should not be interpreted as measured peak memory usage.

Mesh	Vert.	Tets	Bdry faces	Unk.	nnz(A_r)	Total (s)	Est. GB
`sh_1`	9095	49814	5280	6451	92835	0.111028	0.145
`sh_2`	70642	398512	21120	60078	871462	2.32614	1.16
`sh_3`	550354	3188096	84480	508110	7476328	189.614	9.3

9 Mini-element Navier–Stokes module

The incompressible-flow example demonstrates that the framework can handle vector-valued, block-structured and nonlinear problems. The model equations are

$$\begin{cases} \rho\left(\partial_t \vec{u} + (\vec{u}\cdot\nabla)\vec{u}\right) - \nabla\cdot\boldsymbol{\sigma}(\vec{u},p) &= \vec{f} \quad \text{in } \Omega, \\ \nabla\cdot\vec{u} &= 0 \quad \text{in } \Omega, \end{cases} \tag{17}$$

with Cauchy stress tensor

$$\boldsymbol{\sigma}(\vec{u},p) = -p\boldsymbol{I} + 2\mu\boldsymbol{\epsilon}(\vec{u}), \qquad \boldsymbol{\epsilon}(\vec{u}) = \frac{1}{2}\left(\nabla\vec{u} + \nabla\vec{u}^{T}\right).$$

Here $\mu = \rho\nu$ is the dynamic viscosity. We impose Dirichlet data on the inflow, walls and cylinder and a do-nothing stress condition on the outflow. If

$$V = [H^1_{\Gamma_D}(\Omega)]^2, \qquad Q = L^2(\Omega),$$

then the weak formulation is: given $\vec{u}(0) = \vec{u}_0$, find $(\vec{u}(t), p(t)) \in (V + \vec{u}_D) \times Q$ such that, for all $(\vec{v}, q) \in V \times Q$,

$$\int_\Omega \rho\left(\partial_t\vec{u} + (\vec{u}\cdot\nabla)\vec{u}\right)\cdot\vec{v}\,\mathrm{d}\mathbf{x} + \int_\Omega 2\mu\,\boldsymbol{\epsilon}(\vec{u}) : \boldsymbol{\epsilon}(\vec{v})\,\mathrm{d}\mathbf{x} - \int_\Omega p\,\nabla\cdot\vec{v}\,\mathrm{d}\mathbf{x} - \int_\Omega q\,\nabla\cdot\vec{u}\,\mathrm{d}\mathbf{x} = \int_\Omega \vec{f}\cdot\vec{v}\,\mathrm{d}\mathbf{x}. \tag{18}$$

The pressure space is written as $L^2(\Omega)$ because the pressure level depends on the boundary model. In the pure velocity-Dirichlet case the pressure is understood modulo constants, or equivalently in $L^2_0(\Omega)$, whereas natural stress conditions on an outflow may fix the pressure level through the boundary condition. The implementation uses pressure pinning as an algebraic normalization to obtain nonsingular linear systems in a uniform way across test cases; pressure differences and force coefficients are unaffected by a constant pressure shift.

This weak form is the basis for both the block matrix construction and the drag/lift output functional.

The implemented spatial discretization uses the Mini-element: the velocity space is $(\mathbb{P}_1 + \text{bubble})^2$, and the pressure space is $\mathbb{P}_1$. This choice keeps the degree-of-freedom structure simple while satisfying the inf-sup requirement for incompressible flow; see, for example, standard references on finite element discretizations of viscous flow [7, 13, 10]. With time levels $t_n = n\Delta t$, the code uses backward Euler at the first step and BDF2 afterwards. Both can be written as

$$\partial_t \vec{u}(t_{n+1}) \simeq \frac{1}{\Delta t}\left(a_1 \vec{u}^{n+1} - a_2 \vec{u}^{n} - a_3 \vec{u}^{n-1}\right), \tag{19}$$

with $(a_1, a_2, a_3) = (1, 1, 0)$ for the first step and $(3/2, 2, -1/2)$ for BDF2. The nonlinear velocity in the convective field is extrapolated as

$$\vec{u}^{*} = b_1 \vec{u}^{n} + b_2 \vec{u}^{n-1}, \tag{20}$$

with $(b_1, b_2) = (1, 0)$ at the first step and $(2, -1)$ afterwards. The discrete weak problem solved at each time step is therefore: find $(\vec{u}_h^{n+1}, p_h^{n+1}) \in (V_h + \vec{u}_{D,h}) \times Q_h$ such that

$$\begin{aligned}
&\int_\Omega \rho \left(\frac{a_1}{\Delta t} \vec{u}_h^{n+1} + ((b_1 \vec{u}_h^{n} + b_2 \vec{u}_h^{n-1}) \cdot \nabla) \vec{u}_h^{n+1} \right) \cdot \vec{v}_h \, \mathrm{d}\mathbf{x} \\
&+ \int_\Omega 2\mu \, \boldsymbol{\epsilon}(\vec{u}_h^{n+1}) : \boldsymbol{\epsilon}(\vec{v}_h) \, \mathrm{d}\mathbf{x} - \int_\Omega p_h^{n+1} \nabla \cdot \vec{v}_h \, \mathrm{d}\mathbf{x} - \int_\Omega q_h \nabla \cdot \vec{u}_h^{n+1} \, \mathrm{d}\mathbf{x} \\
&= \int_\Omega \frac{\rho}{\Delta t} \left(a_2 \vec{u}_h^{n} + a_3 \vec{u}_h^{n-1} \right) \cdot \vec{v}_h \, \mathrm{d}\mathbf{x} + \int_\Omega \vec{f}^{n+1} \cdot \vec{v}_h \, \mathrm{d}\mathbf{x}
\end{aligned} \tag{21}$$

for all $(\vec{v}_h, q_h) \in V_h \times Q_h$. The mass, viscous and divergence blocks associated with (21) are built by `assembleMiniElementStokes.m`. The convection matrix is assembled separately, and one pressure degree of freedom is pinned as an algebraic normalization. Boundary classification and the inflow profile are handled by dedicated helper routines.

9.1 Block matrix interpretation

The algebraic structure assembled by the code follows directly from (21). Let $\mathbf{u}^n$ denote the vector of velocity degrees of freedom, with the two Cartesian components stored in the same order as the Mini-element velocity basis, and let $\mathbf{p}^n$ denote the vector of pressure degrees of freedom. After insertion of the Dirichlet values, the unknown free degrees of freedom at time level $n+1$ satisfy a saddle-point linear system of the form

$$\begin{bmatrix} \dfrac{\rho a_1}{\Delta t}\mathsf{M} + 2\mu\mathsf{A} + \rho\mathsf{C}(\mathbf{u}^{*}) & \mathsf{B}^T \\ \mathsf{B} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u}^{n+1} \\ \mathbf{p}^{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{r}^{n+1} \\ \mathbf{0} \end{bmatrix}. \tag{22}$$

Here M is the velocity mass matrix, A is the symmetric viscous block associated with the bilinear form $(\vec{u}, \vec{v}) \mapsto \int_\Omega \boldsymbol{\epsilon}(\vec{u}) : \boldsymbol{\epsilon}(\vec{v}) \, \mathrm{d}\mathbf{x}$, B is the discrete divergence operator, and $\mathsf{C}(\mathbf{u}^{*})$ is the convection matrix assembled with the extrapolated convective velocity $\mathbf{u}^{*} = b_1 \mathbf{u}^{n} + b_2 \mathbf{u}^{n-1}$.

The sign convention is absorbed in the definition of B, consistently with the discrete weak form in (21). The right-hand side contains the known time-history contribution,

$$\mathbf{r}^{n+1} = \frac{\rho}{\Delta t}\mathsf{M}\left(a_2\mathbf{u}^n + a_3\mathbf{u}^{n-1}\right) + \mathbf{f}^{n+1} + \text{Dirichlet contributions.} \tag{23}$$

In the implementation, the matrix in (22) is assembled from sparse blocks. The Stokes blocks M, A and B are mesh-dependent and can be assembled once for a fixed viscosity and mesh; the nonlinear block $\mathsf{C}(\mathbf{u}^*)$ is updated at each time step. The pressure degree-of-freedom pinning used in the code provides a uniform algebraic normalization of the saddle-point system. This block representation is useful both for understanding the numerical method and for reading the code: the MATLAB routines do not hide the velocity–pressure coupling, but expose it as a sparse saddle-point matrix constructed from local Mini-element contributions.

Code 4: Sparse block assembly of the linearized Mini-element Navier–Stokes system.

```
K = rho*a1/dt*M + 2*mu*A + rho*C(uStar);
S = [K, B.'; B, sparse(numPressureDofs,numPressureDofs)];
rhs = [rho/dt*M*(a2*uOld + a3*uOlder) + f + dirichletRhs; ...
zeros(numPressureDofs,1)];
% Dirichlet elimination and one pressure pinning are then applied.
```

For drag and lift, the code uses a residual-based weak probe rather than a direct boundary traction reconstruction. Let $\vec{w}_D$ and $\vec{w}_L$ be extensions of the unit drag and lift directions from the cylinder boundary to the computational domain, vanishing on $\partial\Omega \setminus \Gamma_{\text{cyl}}$. The continuous boundary quantities

$$J_D = c_0 \int_{\Gamma_{\text{cyl}}} \boldsymbol{\sigma}(\vec{u}, p)\vec{n} \cdot \vec{1}_D \, \mathrm{d}s, \qquad J_L = c_0 \int_{\Gamma_{\text{cyl}}} \boldsymbol{\sigma}(\vec{u}, p)\vec{n} \cdot \vec{1}_L \, \mathrm{d}s \tag{24}$$

can equivalently be written in residual form as

$$\begin{aligned} J_D &= c_0 \left[\int_\Omega \rho \left(\partial_t \vec{u} + (\vec{u}\cdot\nabla)\vec{u}\right) \cdot \vec{w}_D \, \mathrm{d}\mathbf{x} + \int_\Omega \boldsymbol{\sigma}(\vec{u}, p) : \boldsymbol{\epsilon}(\vec{w}_D) \, \mathrm{d}\mathbf{x} - \int_\Omega \vec{f} \cdot \vec{w}_D \, \mathrm{d}\mathbf{x} \right], \\ J_L &= c_0 \left[\int_\Omega \rho \left(\partial_t \vec{u} + (\vec{u}\cdot\nabla)\vec{u}\right) \cdot \vec{w}_L \, \mathrm{d}\mathbf{x} + \int_\Omega \boldsymbol{\sigma}(\vec{u}, p) : \boldsymbol{\epsilon}(\vec{w}_L) \, \mathrm{d}\mathbf{x} - \int_\Omega \vec{f} \cdot \vec{w}_L \, \mathrm{d}\mathbf{x} \right]. \end{aligned} \tag{25}$$

In the DFG benchmarks considered below the body force is zero, so the last term in each residual expression vanishes. The discrete implementation evaluates the analogue of (25) from the assembled residual. This keeps the output functional within the finite element algebra and avoids a separate low-order boundary traction reconstruction. The validation layer converts the sign convention to the DFG convention and checks the normalization with $\rho = 1$, cylinder diameter $L = 0.1$ and mean inflow velocity $U_{\text{mean}} = 0.2$ for the stationary 2D-1 case.

The purpose of this module is not to provide a complete CFD environment. Rather, it shows that the same open-box programming style extends from scalar elliptic problems to block systems, nonlinear assembly, time stepping and benchmark output functionals. The DFG study in Section 10 also illustrates an important practical point: derived quantities such as drag, lift and pressure differences may have different mesh sensitivities even when the algebraic solver has reached a steady regime.

10 Validation and reproducibility

The validation strategy is kept separate from the performance benchmarks reported in Tables 1–6. It has three levels. First, the repository contains automated tests for the elementary modules: linear Poisson assembly, quadratic elements, three-dimensional tetrahedral assembly, Mini-element Stokes blocks, the Navier–Stokes time loop and the DFG validation layer. These tests are small enough to be run as regression checks by `runAllTests.m`. Second, selected algorithms include independent reference implementations within the released codebase; for instance, the $\mathbb{P}_1$ Poisson matrix can be assembled both by a vectorized sparse routine and by an element loop. Third, standard benchmark quantities are used when available. The incompressible-flow module is tested on the DFG/Turek–Schäfer flow-around-a-cylinder benchmark [16, 8, 9] and on the time-dependent reference values of John [12].

The stationary DFG 2D-1 case uses the channel $[0, 2.2] \times [0, 0.41]$, a cylinder of diameter 0.1 centered at $(0.2, 0.2)$, density $\rho = 1$, viscosity $\nu = 10^{-3}$, maximum inlet velocity $U_{\max} = 0.3$, mean inlet velocity $U_{\mathrm{mean}} = 0.2$, and Reynolds number $Re = 20$. The geometry and boundary partition are shown in Figure 2. Drag and lift are reported in the DFG convention according to the normalization in (26).

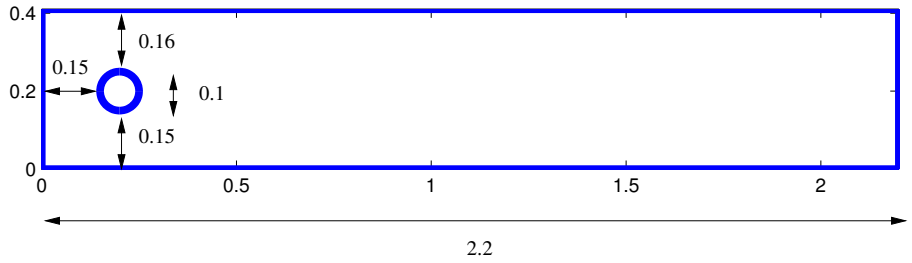

Figure 2: Computational domain for the DFG flow past a cylinder benchmark. The inlet, walls, cylinder boundary and outlet correspond respectively to the Dirichlet, no-slip and do-nothing parts used by the Mini-element Navier–Stokes module.

$$C = \frac{2F}{\rho U_{\mathrm{mean}}^2 L}, \qquad L = 0.1. \tag{26}$$

Throughout the DFG discussion the pressure difference is computed as

$$\Delta p = p(0.15, 0.2) - p(0.25, 0.2),$$

with the sign convention of the corresponding reference data. The reference values used in the tests are

$$C_D = 5.57953523384, \qquad C_L = 0.010618948146, \qquad \Delta p = 0.11752016697. \tag{27}$$

All runs in Table 7 use $\Delta t = 10^{-2}$ and final time $T = 20$. The final oscillations of drag and lift are of order 10^{-10}–10^{-9}. A separate sensitivity run with $\Delta t = 5 \cdot 10^{-3}$ produced the same printed values on the tested mesh. Thus the discrepancies reported below are spatial and post-processing effects, not time-discretization effects. Figure 3 shows the original first mesh and the near-cylinder refinement used in the earlier benchmark campaign; the current table also includes newly generated analytic meshes, generated from the exact channel-with-cylinder geometry rather than by refining a pre-existing polygonal mesh, with controlled cylinder-boundary and radial-layer resolution.

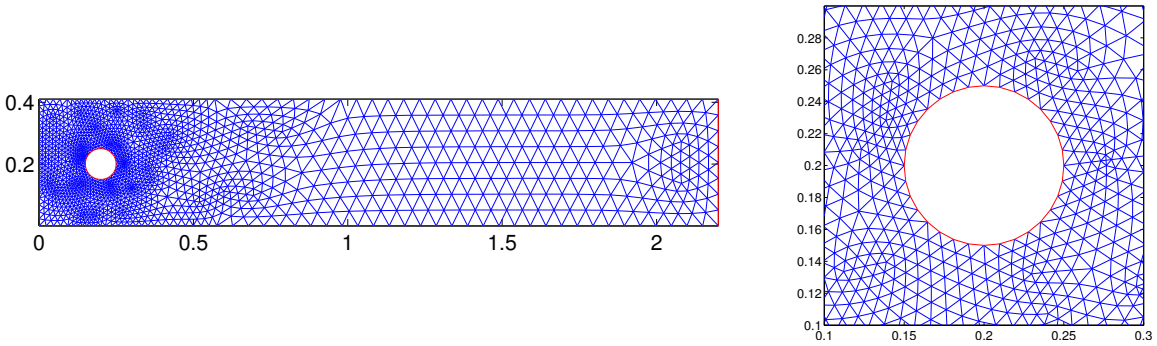

Figure 3: Original first mesh for the DFG 2D-1 test case (left) and a near-cylinder zoom (right). The figure is retained to document the benchmark geometry and to illustrate why drag, lift and pressure outputs are sensitive to near-cylinder mesh design.

Table 7: DFG 2D-1 mesh-sensitivity study for drag, lift and pressure difference. Reference values are the FeatFlow/DFG high-order spectral data for the stationary $Re = 20$ cylinder benchmark.

mesh	nodes	triangles	cyl. edges	C_D	$\lvert e_D \rvert$	C_L	$\lvert e_L \rvert$	Δp	$\lvert e_p \rvert$
`cylinder2`	7979	15558	24	5.5771113	0.0024239	0.0261992	0.0155803	0.1276032	0.0100830
`cylinder`	1788	3392	32	5.5811143	0.0015790	-0.0028174	0.0134363	0.1223680	0.0048479
`analytic-64`	1899	3526	64	5.5683691	0.0111661	0.0090819	0.0015371	0.1752332	0.0577131
`analytic-96-radial`	7270	14010	96	5.5744723	0.0050629	0.0037768	0.0068421	0.1404221	0.0229019
`analytic-64-radial`	7013	13528	64	5.5748976	0.0046377	0.0046555	0.0059634	0.1327855	0.0152653

The table should be read as both a validation and a diagnostic experiment. The drag coefficient is robust across the tested meshes and remains within about 10^{-3} relative error on the best cases. The lift coefficient is much smaller and is correspondingly more sensitive to the polygonal representation of the cylinder and to the structure of the first layers of elements around it. The non-layered analytic mesh with 64 cylinder edges gives the best lift value, while the radially layered analytic meshes improve the pressure difference but change the lift. The pressure drop is instead strongly affected by the quality of the pressure field near the sampling points $(0.15, 0.2)$ and $(0.25, 0.2)$. In the analytic radial meshes these points are mesh vertices

and the first radial layer is controlled, but the remaining discrepancy indicates that pressure-output accuracy requires a more carefully balanced near-cylinder and bulk mesh design.

10.1 Unsteady DFG benchmark cases on the analytic radial mesh

The 2D-2 case is the time-periodic $Re = 100$ benchmark with steady parabolic inflow, whereas 2D-3 uses the time-dependent inflow prescribed by the DFG benchmark on $0 \leq t \leq 8$. The unsteady DFG cases 2D-2 and 2D-3 were also rerun with the current validation layer on the analytic radially refined mesh `dfg-cylinder-analytic-64-radial`. This mesh has 7013 vertices, 13528 triangles, 64 cylinder-boundary edges and a first radial layer of size approximately $5 \cdot 10^{-3}$. Both computations used $\Delta t = 10^{-2}$. The 2D-2 run was advanced to $T = 30$, and the developed cycle was detected in the interval $[29.54, 29.87]$, giving a period of 0.33. The 2D-3 run was advanced to the prescribed final time $T = 8$. The wall-clock times of these runs are reported in Table 8; they are not legacy timings.

It is useful to recall the resolution of the reference computations. For the stationary 2D-1 case, the FeatFlow reference page attributes the quoted values to high-order spectral computations [8]. For the time-periodic 2D-2 case, the FeatFlow reference computations use a Q_2/P_1^{disc} spatial discretization, Crank–Nicolson time stepping and refinement levels up to 667264 algebraic degrees of freedom [9]. For the time-dependent 2D-3 case, the reference values of John [12] are obtained with second-order implicit time stepping and second-order isoparametric finite elements; the finest computations use about half a million spatial degrees of freedom and 6400 time steps. By contrast, the Mini-element runs reported here on the analytic radially refined mesh involve 7013 pressure degrees of freedom, 20541 scalar velocity degrees of freedom, and about $4.8 \cdot 10^4$ coupled unknowns before pressure pinning. Thus the comparison below should be read as a validation and diagnostic test for a compact MATLAB implementation, not as an attempt to reproduce the reference computations at the same resolution.

Table 8: Wall-clock times for the regenerated DFG validation runs on the analytic radially refined mesh `dfg-cylinder-analytic-64-radial`. The times were measured with MATLAB `tic/toc` on the same machine used for the performance benchmarks in Table 1.

case	Δt	final time	elapsed time (s)	elapsed time (min)
DFG 2D-1	0.01	20	1083.899	18.065
DFG 2D-2	0.01	30	1695.550	28.259
DFG 2D-3	0.01	8	451.075	7.518

The 2D-2 results are encouraging for the time-periodic case: the lift extrema and the Strouhal number are close to the reference values, while the drag extrema remain less accurate on this mesh. For 2D-3, the times of the maximum drag and lift are well reproduced, whereas the amplitudes show larger discrepancies, particularly the lift maximum. The pressure difference at final time is computed with the signed convention used in John [12]; after

Table 9: Regenerated DFG 2D-2 benchmark quantities on the analytic radially refined mesh. Reference values are the FeatFlow/DFG 2D-2 data for a developed periodic cycle at $Re = 100$.

quantity	computed	reference	absolute error	relative error
$\min C_D$	3.1009053	3.1569	0.0559947	0.0177373
$\max C_D$	3.1653639	3.2200	0.0546361	0.0169677
$\min C_L$	-1.0239235	-1.0206	0.0033235	0.0032564
$\max C_L$	0.9862967	0.9859	0.0003967	0.0004023
St	0.3030303	0.30188	0.0011503	0.0038105

Table 10: Regenerated DFG 2D-3 benchmark quantities on the analytic radially refined mesh. Reference values for the maxima and their times are those of John [12], as encoded in the validation layer.

quantity	computed	reference	absolute error	relative error
$\max C_D$	2.8886906	2.9509216	0.0622309	0.0210886
$t(\max C_D)$	3.9436978	3.93625	0.0074478	0.0018921
$\max C_L$	0.5504666	0.47795	0.0725166	0.1517242
$t(\max C_L)$	5.7080818	5.693125	0.0149568	0.0026272
$\Delta p(8)$	-0.1207493	-0.1115414	0.0092079	0.0825518

correcting the sign of the reference value, its relative error is about eight percent on this mesh. Overall, the unsteady tests confirm that the validation layer exercises the complete time loop and force post-processing, while also showing that benchmark-quality agreement of all output quantities requires mesh and post-processing choices tailored to the specific functional.

This experiment illustrates a useful role of the framework. Since the mesh, assembly, solver and post-processing routines are explicit MATLAB functions, benchmark quantities can be inspected and modified directly. The DFG campaign separates temporal convergence, geometric resolution of the cylinder boundary, near-wall radial spacing, cycle detection and local pressure-output sensitivity. This is valuable both for validation and for teaching, because it shows that different physical outputs may react very differently to the same mesh refinement strategy.

11 Conclusions

We have presented a compact MATLAB framework for finite element prototyping based on sparse vectorized assembly. The code is designed as an open-box implementation: mesh arrays, degree-of-freedom maps, quadrature rules, local-to-global indices and sparse matrix construction are visible in the source and remain close to the underlying variational formulations. This makes the framework suitable for advanced teaching and research prototyping when transparency and modifiability are more important than large-scale parallel performance.

The examples cover a progressive sequence of problems: $\mathbb{P}_1$ finite elements for Poisson

equations, variable-coefficient diffusion–reaction equations with mixed boundary conditions, $\mathbb{P}_2$ finite elements, a vectorized tetrahedral implementation of the three-dimensional Poisson problem, and a Mini-element formulation for the incompressible Navier–Stokes equations. Across these examples, the same programming principle is used: compute elementwise quantities in arrays, keep only fixed-size local loops when they improve readability, and assemble the global algebraic objects through sparse matrix operations.

The current validation and benchmark results support the intended role of the framework. Automated tests check the elementary modules, while the DFG cylinder benchmark provides an external validation and diagnostic test for the incompressible-flow module. The DFG benchmark study shows robust drag predictions in the stationary case, accurate lift extrema and Strouhal number in the regenerated 2D-2 run, and good timing of the extrema in the regenerated 2D-3 run. It also highlights the different mesh sensitivities of lift and pressure-difference outputs. Future work should include more systematic mesh-generation tools for benchmark-quality near-boundary resolution, further three-dimensional validation examples with independent reference quantities, and a broader tutorial layer.

The aim of the project is not to replace mature finite element environments, but to provide a concise, readable and reproducible MATLAB laboratory for sparse finite element assembly. In this sense, the framework complements both short educational codes and large production libraries: it is small enough to be inspected, but structured enough to support nontrivial elliptic, three-dimensional and incompressible-flow examples.

Software availability

The MATLAB code accompanying this paper is available in the public GitHub repository `https://github.com/mikeletti/vectorized-matlab-fem`. The repository contains the source code, examples, automated tests, validation scripts and benchmark drivers used to generate the results reported in this paper. The code is released under the BSD 3-Clause License.

Declaration of interests

The author declares that he has no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The present research is part of the activities of the “Dipartimento di Eccellenza 2023–2027”. The author is a member of the Gruppo Nazionale per il Calcolo Scientifico (GNCS) of the Istituto Nazionale di Alta Matematica (INdAM).

References

[1] J. Alberty, C. Carstensen, and S. A. Funken. Remarks around 50 lines of Matlab: short finite element implementation. *Numerical Algorithms*, 20(2–3):117–137, 1999.

[2] J. Burkardt. *Quadrature Rules for the Triangle*, 2014.

[3] L. Chen. iFEM: An integrated finite element methods package in MATLAB. Technical report, University of California at Irvine, 2009.

[4] F. Cuvelier, C. Japhet, and G. Scarella. An efficient way to assemble finite element matrices in vector languages. *BIT Numerical Mathematics*, 56(3):833–864, 2016.

[5] M. Dabrowski, M. Krotkiewski, and D. W. Schmid. MILAMIN: MATLAB-based finite element method solver for large problems. *Geochemistry, Geophysics, Geosystems*, 9(4):Q04030, 2008.

[6] D. A. Dunavant. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *International Journal for Numerical Methods in Engineering*, 21(6):1129–1148, 1985.

[7] H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics*. Oxford University Press, Oxford, 2005.

[8] FeatFlow Team. DFG flow around a cylinder benchmark, 2d-1 stationary case. Benchmark data page, 2026. Accessed 2026-06-04.

[9] FeatFlow Team. DFG flow around a cylinder benchmark, 2d-2 time-periodic case. Benchmark data page, 2026. Accessed 2026-06-04.

[10] P. M. Gresho and R. L. Sani. *Incompressible Flow and the Finite Element Method*, volume I–II. John Wiley & Sons, 2000.

[11] F. Hecht. New development in FreeFem++. *Journal of Numerical Mathematics*, 20(3–4):251–266, 2012.

[12] V. John. Reference values for drag and lift of a two-dimensional time-dependent flow around a cylinder. *International Journal for Numerical Methods in Fluids*, 44(7):777–788, 2004.

[13] W. Layton. *Introduction to the Numerical Analysis of Incompressible Viscous Flows*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2008.

[14] S. Micheletti. Fast simulations in Matlab for scientific computing. Technical Report 49/2013, MOX, Dipartimento di Matematica "F. Brioschi", Politecnico di Milano, Milano, Italy, 2013.

[15] T. Rahman and J. Valdman. Fast MATLAB assembly of FEM matrices in 2d and 3d: nodal elements. *Applied Mathematics and Computation*, 219(13):7151–7158, 2013.

[16] M. Schäfer, S. Turek, F. Durst, E. Krause, and R. Rannacher. Benchmark computations of laminar flow around a cylinder. In E. H. Hirschel, editor, *Flow Simulation with High-Performance Computers II*, volume 48 of *Notes on Numerical Fluid Mechanics*, pages 547–566. Vieweg+Teubner Verlag, Braunschweig/Wiesbaden, 1996.

[17] M. P. Szudzik. *An Elegant Pairing Function*. NKS 2006 Wolfram Science Conference, 2006.

[18] B. S. S. Tchuigwa, J. Krmela, J. Pokorný, V. Krmelová, and P. Jilek. Vectfem: a generalized MATLAB-based vectorized algorithm for the computation of global matrix/force for finite elements of any type and approximation order in linear elasticity. *Zeitschrift für angewandte Mathematik und Physik*, 75:150, 2024.

MOX Technical Reports, last issues

Dipartimento di Matematica
Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)

42/2026 Fumagalli, I.; Campioni, M.; Sirtori, A.; Pagani, S.; Levi, R.; Politi, L. S.; Capo, G.; Antonietti, P. F.
Patient-specific computational mechanics of functional lumbar spine units

40/2026 Marchesin, L.; Menafoglio, A.; Secchi, P.
A Convolution Process for Sea Surface Temperature Hot-Spot Identification in the Mediterranean Sea

41/2026 Sosta, L.; Ciancarelli, C; Marini, L.; Pagani, S.; Regazzoni, F.; Parolini, N.
Physics-constrained identification of graph-based thermal networks for spacecraft digital twins

38/2026 Clemente, A.; Arnone, E.; Mateu, J.; Sangalli, L.M.
Nonparametric estimators over metric graphs

39/2026 Patanè, G.; Menafoglio, A.; Krauth, A.; Fechner, P.; Dede', L.; Colosimo, B.M.; Nicolussi, F.
K-Models: a Flexible and Interpretable Method for Ordinal Clustering with Application to Antigen-Antibody Interaction Profiles

37/2026 Centofanti, E.; Ziarelli, G.; Scacchi, S.; Pavarino, L.F.
A Neural Latent Dynamics Approach for Solving Inverse Problems in Cardiac Electrophysiology

36/2026 Botti, M.; Mascotto, L.; Mosconi, M.
A nonconforming method for a generalized Darcy-Forchheimer model

35/2026 Caon, B.; Corti, M.; Bonizzoni, F.; Antonietti, P.F.
High-fidelity and Network-based Spatio-temporal Mathematical Models of Alzheimer's Disease Progression and their Validation Against PET-SUVR Imaging Data

34/2026 Mancinelli, F. M.; Torzoni, M.; Maisto, D.; Donnarumma, F.; Corigliano, A.; Pezzulo, G.; Manzoni, A.
Multi-Agent Digital Twins for strategic decision-making using Active Inference

33/2026 Franzoni, G.; Mirabella, S.; Dabek, A.; Ferro, N.; Antona, A.; Carlessi, M.; Cinquemani, S.; Matteucci, M.; Cocetta, G.; Perotto, S.
Integrating Environmental Control and Hyperspectral Imaging to Assess Light and Nutrient Effects on Lettuce Post-Harvest Quality in Vertical Farming