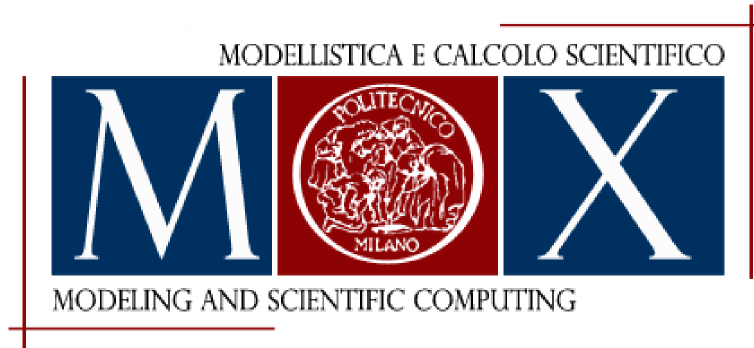


MOX–Report No. 49/2013

# Fast simulations in Matlab for Scientific Computing

Micheletti, S.

MOX, Dipartimento di Matematica "F. Brioschi"
Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)

mox@mate.polimi.it http://mox.polimi.it

# Fast simulations in Matlab for Scientific Computing

Stefano Micheletti

14 ottobre 2013

♯ MOX– Modellistica e Calcolo Scientifico
Dipartimento di Matematica "F. Brioschi"
Politecnico di Milano
Piazza L. da Vinci 32, 20133 Milano, Italy
stefano.micheletti@polimi.it

**Sommario**

We show how the numerical simulation of typical problems found in Scientific Computing can be run efficiently even under the serial Matlab environment. This is made possible by a strong employment of vectorization and sparse matrix manipulation. Numerical examples based on FEMs on 2D unstructured triangular grids assess the flexibility and efficiency of the simulation tool, both on simple elliptic problems as well as on the steady and unsteady incompressible Navier-Stokes equations. Any type of finite elements, and 1D and 2D quadrature rules can be easily accommodated within our framework. Emphasis is focused on vectorization programming and sparse matrix storage and operations, which allow one to obtain very efficient programs which run in a few minutes on a common notebook.

**Keywords:** Finite Element Method, Navier-Stokes equations, Programming languages

**AMS:** 68N15, 35Q30, 65N30, 65M60

# 1 Introduction

It is commonly believed that fast numerical simulations of partial differential equations under the Matlab environment is beyond reach. However, thanks to the employment of a suitable programming paradigm, we want to show that this objective is actually achievable, even on standard notebooks. In particular, we aim at providing the user with a simple and short open-box Matlab implementation of FEMs on 2D triangular meshes. An early attempt in this direction was proposed in [1]. However, we improve on [1] in three respects: 1) efficiency; 2) room for finite elements other than $\mathbb{P}_1$; 3) employment of arbitrary quadrature rules. In contrast to [1], we will not deal with the approximations on quadrilateral elements. Actually, once our philosophy has been understood, simple modifications of the proposed codes allow the user to also tackle such elements. Moreover, the extension to the 3D case is straightforward, once the suitable geometric data structures are available.

Our main effort has been devoted to the employment, as far as possible, of vectorized operations. As a matter of fact, the assembly of the stiffness matrix is typically based on the standard loop-over-the-elements (see, e.g., [1, 12, 13, 16]): each triangle of the mesh is processed in turn; the local matrix and right-hand side are computed and then assembled into the global matrix and load vector, respectively, exploiting the local-to-global numbering of the degrees of freedom. However, as it stands, this is the actual bottleneck of the whole procedure: the larger the number of degrees of freedom, the longer the relative time it takes to execute, until the overall computation cost is swamped with the assembly phase. To overcome this strong limitation, we move from the implementation of the stiffness matrix included in the functions `assema` and `pdeasmc` of the `pdetool` in Matlab [14].[1] There, the assembly of the stiffness matrix associated with the bilinear form $\int_\Omega c(\mathbf{x})\nabla u \cdot \nabla v \, \mathrm{d}\mathbf{x}$, deriving from an elliptic problem, is carried out. This is performed very efficiently, thanks to the employment of the command `sparse`. On the other hand, this works only for $\mathbb{P}_1$ FEM and using the midpoint quadrature rule. Inspired by this *assembly paradigm*, we have extended this philosophy to other differential problems, type of FEMs, and quadrature rules.

The layout of the paper is organized as follows. We start by introducing the data structure required for describing the mesh in Sect. 2. Here, we employ a so-called *minimum data structure*, i.e., sufficient for characterizing completely the lowest order FEM space, i.e., of piecewise linear and continuous functions. This data structure can be then enriched according to the higher order FEM at hand. To illustrate our paradigm on a model problem, the Poisson problem completed with Dirichlet boundary conditions is described in Sect. 3 along with the associated weak and discrete weak formulations, which lay the basis for the successive FEM approximation. In particular, we consider the simple case of

[1] We are referring to the release 7.9.0.529 (R2009b)

piecewise liner and continuous FEMs and we approximate the integrals involved in the load vector by the midpoint quadrature rule. Then we extend this basis approach to the case of a diffusion-reaction problem in Sect. 4. Here we consider general mixed (Dirichlet and Neumann) boundary conditions and we incorporate the use of quadrature rules for both triangles and edges. In Sect. 5, we address the discretization of the incompressible Navier-Stokes equations. In particular, we deal with some benchmark problems which aim at computing the drag and lift coefficients around a cylinder placed in a rectangular channel. These quantities involve boundary integrals around the cylinder. Instead of directly computing these integrals, we resort to a "trick" which is sometimes exploited in the finite element community. This relies on the weak form of the discretized Navier-Stokes equations and allows one to obtain more accurate results than the direct computation. The validation on the benchmarks is carried out in Sect. 6. Finally, some conclusions and future developments are gathered in Sect. 7.

## 2 A minimum data structure

Given an open and bounded polygonal domain $\Omega \subset \mathbb{R}^2$ with boundary $\partial\Omega$, we denote by $\mathcal{T}_h = \{K\}$ a conformal mesh of $\overline{\Omega} = \Omega \cup \partial\Omega$ consisting of triangles $K$'s. We want to provide the minimum data structure required for the later FEM implementation. We take inspiration from the Partial Differential Equation Toolbox `pde` in Matlab. Let us denote by `Np`, `Nt`, and `Ne` the number of vertexes, triangles and boundary edges, respectively, of the actual mesh. Then the toolbox characterizes any mesh by the three arrays `p,t` and `e`, with dimensions $2 \times$ `Np`, $4 \times$ `Nt` and $7 \times$ `Ne`, respectively. The full description of all of the entries of these arrays is not required next. Instead, let us just focus on the main ones, which constitute the minimum data structure, i.e., sufficient for the description of the piecewise linear FEMs.
The entries

1. `p(1:2,i)` provide the two coordinates of the $i$-th vertex, $\mathbf{x}_i$, of the mesh;
2. `t(1:3,k)` yield the global numbering of the three vertexes of the $k$-th triangle, in a counterclockwise fashion;
3. `e(1:2,j)` return the global numbering of the two vertexes of the $j$-th boundary edge; `e(5,j)` furnishes the labeling of the portion of the domain boundary containing the $j$-th boundary edge.

We notice that when these structures are generated by Matlab, e.g., through the command `initmesh`, there is no guarantee that the local ordering of the boundary nodes in the array `e` be counterclockwise. For later use, a post-processing of these nodes may thus be required, and we henceforth assume that this has been carried out. The labeling of the boundary portions of the domain may be useful for assigning different boundary conditions, e.g., of Dirichlet or Neumann

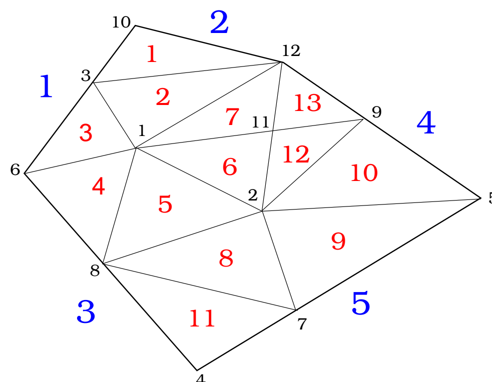


Figura 1: Example of a mesh and numbering of the geometrical entities: boundary portions (blue), vertexes (black) and elements (red).

type, etc. Figure 1 provides an example of a simple mesh comprising 5 boundary portions (blue), 12 vertexes (black), 13 elements (red), and 9 boundary edges. The reduced data structures `t` and `e` are given by

```
t =
     3    1    6    8    1    2   11    7    5    9    4    2   11
    12   12    1    1    8   11   12    2    2    2    7    9    9
    10    3    3    6    2    1    1    8    7    5    8   11   12


e =
     8      7      4      6      5      3     10     12      9
     4      5      7      8      9      6      3     10     12
     3      5      5      3      4      1      1      2      4
```

Notice that whereas the ordering of the boundary edges may not be geometrically consecutive, the local numbering of the two local nodes of each edge is yet counterclockwise. Furthermore, for later use, we designate by $\mathcal{N}_\Omega$ and $\mathcal{N}_{\partial\Omega}$ the set of indices, according to their global numbering, of the internal and boundary nodes, respectively.

## 3 The Poisson model problem

To illustrate our approach, let us start from the model Poisson problem:

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

where $\Delta$ is the Laplacian operator, and $f, g_D$ are given functions, which we suppose to be both defined in $\overline{\Omega}$. With a view to the FEM approximation to

problem (1), we recall its weak formulation. Find $u \in H_0^1(\Omega) + g_D$ such that

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

where $H_0^1(\Omega)$ is the subspace of the Sobolev space $H^1(\Omega)$, consisting of functions that together with their first weak derivatives are Lebesgue integrable, which have zero trace on the boundary ([17]), and $H_0^1(\Omega)+g_D = \{v \in H^1(\Omega) : v - g_D \in H_0^1(\Omega)\}$. The FEM approximation to (2) is obtained in a straightforward way, after introducing the FEM space $V_h^r = \{v_h \in C^0(\overline{\Omega}) : v_h|_K \in \mathbb{P}_r(K), \forall K \in \mathcal{T}_h\}$, where $\mathbb{P}_r(K)$ is the space of polynomials of maximum degree $r$ over $K$. We then let $V_{h,0}^r = V_h^r \cap H_0^1(\Omega)$. Thus the discrete formulation reads: find $u_h \in V_{h,0}^r + g_{D,h}$ such that

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

with $g_{D,h} \in V_h^r$ a suitable approximation to $g_D$. It is well known that (3) is equivalent to an algebraic linear system, say $AU = F$, where $A$ is the so-called stiffness matrix, $F$ the load vector, and $U$ collects the degrees of freedom of the FEM space ([7]). To fix ideas, we choose piecewise linear FEMs, i.e., we pick $r = 1$. In this case, the degrees of freedom are the nodal values, $\{u_h(\mathbf{x}_i)\}$, of the FEM function $u_h$ at the internal vertexes of the mesh, i.e., those not belonging to the domain boundary. Thus it is possible to expand the discrete solution as

$$u_h(\mathbf{x}) = \sum_{j \in \mathcal{N}_\Omega} u_h(\mathbf{x}_j)\phi_j(\mathbf{x}) + g_{D,h}(\mathbf{x}), \text{ with } g_{D,h}(\mathbf{x}) = \sum_{j \in \mathcal{N}_{\partial\Omega}} g_D(\mathbf{x}_j)\phi_j(\mathbf{x}),$$

where $\{\phi_j\}$ are the hat basis functions, and $g_{D,h}$ is the FEM function that interpolates $g_D$ at the boundary nodes. We recall that the hat functions satisfy the conditions

$$\text{span } \{\phi_i\}_{i=1}^{N_h} = V_{h,0}^1; \quad \phi_i(\mathbf{x}_j) = \delta_{ij} \quad i, j = 1, \ldots, N_h,$$

where $\{\mathbf{x}_i\}_{i=1}^{N_h}$ denotes the set of the internal vertexes of the mesh, $N_h$ being the number of the internal vertexes, while $\delta_{ij}$ is the Kronecker symbol. Then the entries, $A_{i,j}$, of the stiffness matrix and, $F_i$, of the load vector are given, for $i, j = 1, \ldots, N_h$, by

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

In the next example, we pick the data $f$ and $g_D$ such that the exact solution is $u = \sin(\pi(x+1)/2)\sin(\pi(y+1)/2)$ on $\Omega = (-1,1)^2$. The following algorithm gathers the Matlab implementation of the assembling and solution of the resulting algebraic system.

**Algorithm 1**

```
 1 Np = size(p,2);
 2 nD = union(e(1,:),e(2,:));
 3 % matrix and vector allocation
 4 A = sparse(Np,Np);
 5 F = sparse(Np,1);
 6 u = sparse(Np,1);
 7 % area of triangles and derivatives of the shape functions
 8 [ar,g1x,g1y,g2x,g2y,g3x,g3y] = pdetrg(p,t);
 9 % local nodes on triangles and centre of mass of triangles
10 n1 = t(1,:);
11 n2 = t(2,:);
12 n3 = t(3,:);
13 xb = (p(1,n1) + p(1,n2) + p(1,n3))/3;
14 yb = (p(2,n1) + p(2,n2) + p(2,n3))/3;
15 % source term and Dirichlet data
16 f  = @(x,y) pi^2/2*sin(pi*(x+1)/2).*sin(pi*(y+1)/2);
17 gD = @(x,y) 0*x;
18 % entries of the stiffness matrix
19 a12 = (g1x.*g2x + g1y.*g2y).*ar;
20 a23 = (g2x.*g3x + g2y.*g3y).*ar;
21 a31 = (g3x.*g1x + g3y.*g1y).*ar;
22 % assembling of the stiffness matrix
23 A = A + sparse(n1,n2,a12,Np,Np);
24 A = A + sparse(n2,n3,a23,Np,Np);
25 A = A + sparse(n3,n1,a31,Np,Np);
26 A = A + A';
27 A = A + sparse(n1,n1,-(a12+a31),Np,Np);
28 A = A + sparse(n2,n2,-(a23+a12),Np,Np);
29 A = A + sparse(n3,n3,-(a31+a23),Np,Np);
30 % assembling of the right-hand side
31 fb = f(xb,yb).*ar/3;
32 F = F + sparse(n1,1,fb,Np,1);
33 F = F + sparse(n2,1,fb,Np,1);
34 F = F + sparse(n3,1,fb,Np,1);
35 % evalutation of the Dirichlet data on the boundary nodes
36 uD = gD(p(1,nD),p(2,nD))';
37 % construction of the unknown nodes
38 unk = setdiff(1:Np,nD);
39 % elimination of the Dirichlet nodes
40 A = A(unk,:);
41 F = F(unk) - A(:,nD)*uD;
42 A = A(:,unk);
43 % solution of the linear system
44 s = A\F;
```

```
45 u(unk) = s;
46 u(nD)   = uD;
```

Some comments are in order.

1. The variable `nD` at line `2` collects the Dirichlet nodes (in this case the whole of the boundary nodes);

2. the stiffness matrix, and the load and unknown vectors are allocated at lines `4-6` via the command sparse;

3. the Matlab function `pdetrg` at line `8` returns the row vector `ar` containing the area of all the elements, and the row vectors of the partial derivatives with respect to $x$ (`g1x, g2x, g3x`) and to $y$ (`g1y, g2y, g3y`), of the local basis functions, according to the local numbering implicitly defined by the array `t`. For example, `g1x(k)` yields the (constant) derivative $\partial\phi/\partial x$ of the basis function associated with the 1st local node of triangle $k$;

4. the global numbering of the vertexes is computed at lines `10-12`, e.g., `n1` is the row vector of dimension `Nt` containing the global vertexes that are first local nodes of the triangles;

5. the coordinates of the center of mass of all the triangles are computed at lines `13-14`;

6. the data $f$ and $g_D$ are defined through anonymous functions at lines `16-17`;

7. at lines `19-21` the entries of the stiffness matrix are computed, each as a row vector of dimension `Nt`. For example, `a12(k)` provides the $1-2$ entry defined by

$$\mathtt{a12(k)} = \int_K \nabla\phi_1 \cdot \nabla\phi_2 \,\mathrm{d}\mathbf{x} = \nabla\phi_1 \cdot \nabla\phi_2 |K|,$$

   where $K$ is the triangle with numbering `k`, $|K|$ is the area of $K$, and the integral can be computed exactly since the integrand is a constant. The functions $\phi_1, \phi_2$ are the two basis functions associated with the local nodes 1 and 2 of $K$, respectively;

8. the assembling of the global stiffness matrix is performed at lines `23-29` via the command `sparse`. In particular, `sparse(n1,n2,a12,Np,Np)` builds a sparse matrix of dimension $\mathtt{Np}\times\mathtt{Np}$ and assigns the values `a12` to the entries identified by the row index `n1` and column index `n2`, i.e., $A(\mathtt{n1(k)}, \mathtt{n2(k)}) = \mathtt{a12(k)}$, for $\mathtt{k} = 1, \ldots, \mathtt{Nt}$. Notice also that the symmetric structure of the matrix, as well as that the row sum is zero are exploited at lines `26` and `27-29`, respectively;

9. In an analogous fashion, the assembling of the load vector is carried out at lines `31-34`. In particular, the first integral involved in the definition of $F$ in (4) is approximated by the midpoint quadrature rule, i.e.,

$$\mathtt{F(k)} = \int_K f\phi_i \,\mathrm{d}\mathbf{x} = \frac{1}{3} f(x_b(k), y_b(k))\,|K|,$$

   with $(x_b(k), y_b(k))$ the coordinates of the center of mass of $K$, where each basis function attains the value $1/3$, and $i$ is here understood to be a local index ranging in the set $\{1, 2, 3\}$;

10. The variables `uD` at line `36` and `unk` at line `38` define the Dirichlet boundary conditions (evaluated in correspondence with the nodes `nD`) and the unknown vector, respectively. In particular, `unk` is a column vector whose dimension is given by the cardinality of the internal vertexes. The function `setdiff` performs the set difference between two sets.

11. At lines `40-42` the elimination of the Dirichlet nodes is carried out. In practice this amounts to *i)* discarding the rows in $A$ associated with the boundary nodes, *ii)* moving the columns corresponding to the Dirichlet nodes to the right-hand side, after multiplication by `uD`;

12. the algebraic system is solved at line `44` via the built-in sparse direct solver, while the assembling of the complete solution vector, including both internal and boundary values, is accomplished at the last lines `45-46`.

In order to assess the performance of the above algorithm, we have carried out a test. Namely, we have solved (3), with the same data as above, for seven different mesh sizes, repeating each computation ten times. The overall performance is measured by the average time (over the ten runs) required to complete Algorithm 1 for each of the seven cases. The meshes have been obtained starting from an initial unstructured grid generated by the command `[p,e,t] = initmesh('squareg','hmax',0.4)` (with the function `initmesh` of the `pdetool`), which is then successively uniformly refined by the command `[p,e,t] = refinemesh('squareg',p,e,t)`. This yields meshes characterized by an average mesh size, `h`, taking the values $\{0.2 \cdot 2^i\}_{i=0}^{-6}$ (and is faster than using `initmesh` directly). To measure time, the pair of Matlab functions `tic`, `toc` have been used throughout. The computations have been carried out on a notebook with an Intel® Core$^{TM}$ 2 Duo Processor P8600 @ 2.40 GHz, and 4 GB RAM. Figure 2 shows the average time, $T$, in seconds (black $*$), as a function (in log-log scale) of the dimension, $S$, of the stiffness matrix (after eliminating the Dirichlet nodes). The slope of the estimated fit (blue line) is 1.45, showing a superlinear dependence of time on size. To get a feeling about the CPU time, notice that, for a size $S$ on the order of $4 \cdot 10^4$, which is usually considered quite large for this problem (in 2D), it takes less than 2 seconds two completely run the program. For comparison purposes, the average time, $T_l$, required by the whole

Tabella 1: Average CPU time versus matrix dimension. Mesh size $h$; matrix size $S$; CPU time $T$; CPU time $T_l$ with loop-over-the-elements; CPU time $T_s$ to solve the linear system.

| $h$ | $S$ | $T$ | $T_l$ | $T_s$ |
|---|---|---|---|---|
| 1/5 | 157 | $2.07 \cdot 10^{-03}$ | $3.07 \cdot 10^{-02}$ | $3.48 \cdot 10^{-04}$ |
| 1/10 | 665 | $5.86 \cdot 10^{-03}$ | $1.57 \cdot 10^{-01}$ | $1.58 \cdot 10^{-03}$ |
| 1/20 | 2,737 | $2.93 \cdot 10^{-02}$ | $1.95 \cdot 10^{+00}$ | $1.06 \cdot 10^{-02}$ |
| 1/40 | 11,105 | $1.81 \cdot 10^{-01}$ | $3.21 \cdot 10^{+01}$ | $4.92 \cdot 10^{-02}$ |
| 1/80 | 44,737 | $1.74 \cdot 10^{+00}$ | $6.70 \cdot 10^{+02}$ | $2.75 \cdot 10^{-01}$ |
| 1/160 | 179,585 | $2.28 \cdot 10^{+01}$ | $1.42 \cdot 10^{+04}$ | $1.62 \cdot 10^{+00}$ |
| 1/320 | 719,617 | $4.14 \cdot 10^{+02}$ | out of time | $8.86 \cdot 10^{+00}$ |

algorithm (data definition + allocation + assembling + solving) employing the loop-over-the-elements for the same problem is shown as well (red $\circ$). The code used for the assembling is provided below.

```
for k = 1:Nt
  nloc = t(1:3,k);
  Aloc = [-a12(k)-a31(k), a12(k), a31(k);
          a12(k),        -a23(k)-a12(k), a23(k);
          a31(k),         a23(k),       -a31(k)-a23(k)];
  A(nloc,nloc) = A(nloc,nloc) + Aloc;
  xb = sum(p(1,nloc))/3;
  yb = sum(p(2,nloc))/3;
  fb = f(xb,yb)*ar(k)/3;
  F(nloc) = F(nloc) + fb;
end
```

The linear fit (green line) in Figure 2 shows a slope of 1.89, which is clearly higher than the previous one. Notice that the computation on the last grid ran out of time (the estimated average time is on the order of $10^5 s \simeq 1$ day). The average computing time, $T_s$, required to solve the linear system (which is the same both with and without the loop-over-the-elements) is shown as well in Table 1 and in Figure 2 (black $\square$). The corresponding linear fit (magenta line) yields a slope of 1.21.

To sum up, these results show that, the larger the matrix size, the higher the relative time spent in the assembling. This is particularly emphasized when looping over the elements, which exhibits both a higher cost (in absolute sense) as well as a larger sensitivity to the problem size with respect to the "sparse" assembling.

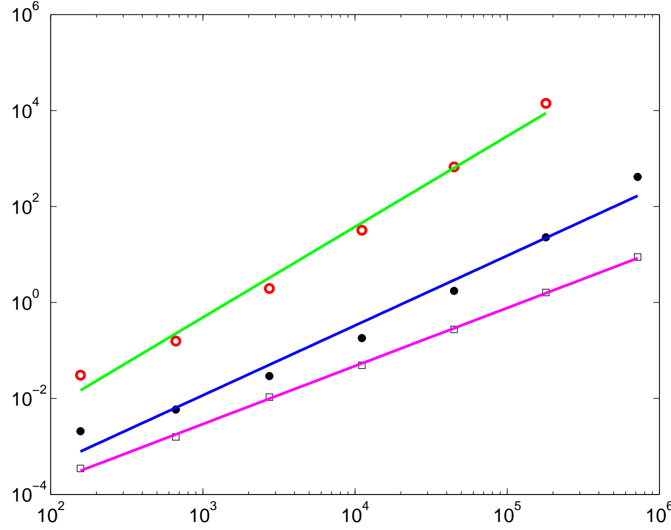


Figura 2: Average CPU time versus matrix dimension. Black ($*$): actual values; blue solid line: linear regression; red ($\circ$): actual values; green solid line: linear regression (with loop-over-the-elements); black ($\square$): actual values; magenta solid line: linear regression (linear system solve).

## 4 The diffusion-reaction problem

We now deal with the management of quadrature formulas and, for this purpose, we move to an extension of the model Poisson problem, i.e., to the diffusion-reaction problem

$$\begin{cases} -\nabla \cdot (\mu \nabla u) + \sigma\, u &= f \quad \text{in } \Omega, \\ u &= g_D \quad \text{on } \partial\Gamma_D, \\ \mu \dfrac{\partial u}{\partial n} &= g_N \quad \text{on } \partial\Gamma_N, \end{cases} \tag{5}$$

where $\overline{\Gamma}_D \cup \overline{\Gamma}_N = \partial\Omega$ with $\Gamma_D \cap \Gamma_N = \emptyset$, $g_N \in C^0(\overline{\Gamma}_N)$ is a given function, and $\partial/\partial n$ designates the outward normal derivative. The functions $\mu \in C^0(\overline{\Omega})$, with $\mu \geq \mu_0 > 0$, and $\sigma \in C^0(\overline{\Omega})$ with $\sigma \geq 0$, represent the diffusion and reaction coefficient, respectively. As in the previous section, $f \in C^0(\overline{\Omega})$ is the source term, while $g_D \in C^0(\overline{\Gamma}_D)$ is the Dirichlet datum. Although the regularity requirements for all of these functions can be relaxed, here we enforce the continuity because of the use of quadrature formulas. With a view to the FEM approximation to problem (5), we recall its weak formulation. Find $u \in H^1_{\Gamma_D}(\Omega) + g_D$ such that

$$\int_\Omega \mu\, \nabla u \cdot \nabla v \, d\mathbf{x} + \int_\Omega \sigma\, u\, v \, d\mathbf{x} = \int_\Omega f v \, d\mathbf{x} + \int_{\Gamma_N} g_N\, v \, ds \quad \forall v \in H^1_{\Gamma_D}(\Omega), \tag{6}$$

where $H^1_{\Gamma_D}(\Omega) + g_D = \{v \in H^1(\Omega) : v - g_D = 0 \text{ on } \Gamma_D\}$, with $H^1_{\Gamma_D}(\Omega) = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\}$. To obtain the FEM approximation to (6), we define the

FEM space $V^r_{h,\Gamma_D} = V^r_h \cap H^1_{\Gamma_D}(\Omega)$. Thus the discrete formulation reads: find $u_h \in V^r_{h,\Gamma_D} + g_{D,h}$ such that

$$\int_\Omega \mu\, \nabla u_h \cdot \nabla v_h \,\mathrm{d}\mathbf{x} + \int_\Omega \sigma\, u_h\, v_h \,\mathrm{d}\mathbf{x} = \int_\Omega f v_h \,\mathrm{d}\mathbf{x} + \int_{\Gamma_N} g_N\, v_h \,\mathrm{d}s \quad \forall v_h \in V^r_{h,\Gamma_D}, \tag{7}$$

with $g_{D,h} \in V^r_h$ a suitable approximation to $g_D$. As in the case of the Poisson model problem, (7) is equivalent to an algebraic linear system, say $AU = F$, where $A$ is still called stiffness matrix, $F$ the load vector, and $U$ collects the degrees of freedom of the FEM space. For example, with piecewise linear FEMs, i.e., $r = 1$, the degrees of freedom are the nodal values of the FEM function $u_h$ at the internal vertexes of the mesh and at the nodes belonging to $\Gamma_N$. Let the total number of these vertexes be $N_h$. Moreover, it holds

$$u_h(\mathbf{x}) = \sum_{j\in\mathcal{N}_\Omega\cup\mathcal{N}_{\Gamma_N}} u_h(\mathbf{x}_j)\phi_j(\mathbf{x}) + g_{D,h}(\mathbf{x}), \text{ with } g_{D,h}(\mathbf{x}) = \sum_{j\in\mathcal{N}_{\Gamma_D}} g_D(\mathbf{x}_j)\phi_j(\mathbf{x}),$$

where $\{\phi_i\}$ are the hat basis functions, $g_{D,h}$ is the FEM function that interpolates $g_D$ at the Dirichlet boundary nodes, and we have partitioned the set $\mathcal{N}_{\partial\Omega} = \mathcal{N}_{\Gamma_N} \cup \mathcal{N}_{\Gamma_D}$, with $\mathcal{N}_{\Gamma_N} \cap \mathcal{N}_{\Gamma_D} = \emptyset$, into the two sets of indices, according to their global numbering, of the Neumann, $\mathcal{N}_{\Gamma_N}$, and Dirichlet, $\mathcal{N}_{\Gamma_D}$ nodes, respectively. Then the entries, $A_{i,j}$, of the stiffness matrix and, $F_i$, of the load vector are given, for $i, j = 1, \ldots, N_h$, by

$$\begin{aligned} A_{i,j} &= \int_\Omega \mu\, \nabla\phi_j \cdot \nabla\phi_i \,\mathrm{d}\mathbf{x} + \int_\Omega \sigma\, \phi_j\, \phi_i \,\mathrm{d}\mathbf{x}, \\ F_i &= \int_\Omega f \phi_i \,\mathrm{d}\mathbf{x} + \int_{\Gamma_N} g_N\, \phi_i \,\mathrm{d}s - \sum_{j\in\mathcal{N}_{\Gamma_D}} g_D(\mathbf{x}_j) \int_\Omega \mu\, \nabla\phi_j \cdot \nabla\phi_i \,\mathrm{d}\mathbf{x}. \end{aligned} \tag{8}$$

Notice that now, for general functions $\mu, \sigma, f, g_N$, the corresponding integrals in (8) are no longer exactly computable. Thus it is necessary to resort to some quadrature formulas. These exploit the relations

$$\int_\Omega (\cdot) \,\mathrm{d}\mathbf{x} = \sum_{K\in\mathcal{T}_h} \int_K (\cdot) \,\mathrm{d}\mathbf{x} \quad \text{and} \quad \int_{\Gamma_N} (\cdot) \,\mathrm{d}s = \sum_{e\in\Gamma_N} \int_e (\cdot) \,\mathrm{d}s,$$

so that we are led to approximate integrals on triangles and edges. In particular, in the first case we employ the Dunavant-type formulas developed by J. Burkardt and downloadable from [4] (see also [5, 18]), while for the second case we use the Gauss-Legendre formulas developed by G. von Winckel and available through the Matlab Central [22]. We recall that, in general, the quadrature formulas for a dummy function $v$ on a triangle $K$, and on an edge $e$ read

$$\int_K v(\mathbf{x}) \,\mathrm{d}\mathbf{x} \simeq \sum_{i=1}^{N_q^K} w_i^K\, v(\mathbf{x}_i^K) \quad \text{and} \quad \int_e v(s) \,\mathrm{d}s \simeq \sum_{i=1}^{N_q^e} w_i^e\, v(s_i^e),$$

where $\{\mathbf{x}_i^K, w_i^K\}_{i=1}^{N_q^K}$ and $\{s_i^e, w_i^e\}_{i=1}^{N_q^e}$ define the pair of $N_q^K$ and $N_q^e$ nodes and weights, for $K$ and $e$, respectively. Proceeding according to the standard practice in the FEM community, the use of a quadrature rule, either one- or two-dimensional, is carried out by resorting to the so-called reference element. In particular, in the two-dimensional case, a reference element, say $\widehat{K}$, is chosen as the right-angled triangle with vertexes in (0,0), (1,0), and (0,1). Then any other triangle, $K \in \mathcal{T}_h$, can be defined via an affine mapping, $T_K : \widehat{K} \to K$, such that

$$\mathbf{x} = T_K(\widehat{\mathbf{x}}) = M_K \, \widehat{\mathbf{x}} + \vec{t}_K, \tag{9}$$

where $M_K \in \mathbb{R}^{2\times 2}$ is the (constant) Jacobian of the transformation, and $\vec{t}_K \in \mathbb{R}^2$ is a shift. If we denote by $\{(x_i^K, y_i^K)\}_{i=1}^3$, the coordinates of the three vertexes of $K$ (ordered counterclockwise), then it holds

$$M_K = \begin{bmatrix} x_2^K - x_1^K & x_3^K - x_1^K \\ y_2^K - y_1^K & y_3^K - y_1^K \end{bmatrix} \quad \text{and} \quad \vec{t}_K = [x_1^K, y_1^K]^T, \tag{10}$$

so that the vertexes (0,0), (1,0), and (0,1) of $\widehat{K}$ are mapped into $(x_1^K, y_1^K)$, $(x_2^K, y_2^K)$, and $(x_2^K, y_2^K)$, respectively. The employment of the reference map facilitates the implementation of the quadrature rule since the nodes and the weights can be defined once and for all on $\widehat{K}$, and mapped into the actual nodes and weights on $K$ in a simple way. Let $\{\widehat{\mathbf{x}}_i^K, \widehat{w}_i^K\}_{i=1}^{N_q^K}$ be the node-weight pair on $\widehat{K}$, such that $\sum_{i=1}^{N_q^K} \widehat{w}_i^K = 1$, then it holds

$$\mathbf{x}_i^K = T_K(\widehat{\mathbf{x}}_i^K) \quad \text{and} \quad w_i^K = \widehat{w}_i^K |K|. \tag{11}$$

Moreover, thanks to the differentiation chain rule, it is possible to write the derivative of a function, say $v$, defined on $K$ in terms of derivatives of the pullback, $\widehat{v}$, of $v$ by $T_K$, defined on $\widehat{K}$, where $\widehat{v} = v \circ T_K$, i.e., $\widehat{v}(\widehat{\mathbf{x}}) = v(T_K(\widehat{\mathbf{x}}))$, $\forall \widehat{\mathbf{x}} \in \widehat{K}$. In fact, thanks to (9), we have that $\nabla v = M_K^{-T} \widehat{\nabla} \widehat{v}$, where

$$M_K^{-T} = \frac{1}{2|K|} \begin{bmatrix} y_3^K - y_1^K & y_1^K - y_2^K \\ x_1^K - x_3^K & x_2^K - x_1^K \end{bmatrix} \tag{12}$$

designates the inverse of the transposed Jacobian in (10). Gathering all this properties, it is possible to approximate the contributions to the stiffness matrix and to the load vector. Let us start from the diffusion term:

$$\begin{aligned} \int_K \mu \, \nabla \phi_j \cdot \nabla \phi_i \, \mathrm{d}\mathbf{x} &= \int_{\widehat{K}} \widehat{\mu} \, M_K^{-T} \, \widehat{\nabla} \widehat{\phi}_j \cdot M_K^{-T} \, \widehat{\nabla} \widehat{\phi}_i \, |K| \, \widehat{\mathrm{d}\mathbf{x}} \\ \simeq |K| \sum_{l=1}^{N_q^K} &\Big( \widehat{w}_l^K \, \widehat{\mu}(\widehat{x}_l^K) \, M_K^{-T} \, \widehat{\nabla} \widehat{\phi}_j(\widehat{x}_l^K) \cdot M_K^{-T} \, \widehat{\nabla} \widehat{\phi}_i(\widehat{x}_l^K) \Big). \end{aligned} \tag{13}$$

The reaction contribution is easily dealt with as

$$\int_K \sigma\,\phi_j\,\phi_i\,\mathrm{d}\mathbf{x} = |K| \int_{\widehat{K}} \widehat{\sigma}\,\widehat{\phi}_j\,\widehat{\phi}_i\,\widehat{\mathrm{d}\mathbf{x}} \simeq |K| \sum_{l=1}^{N_q^K} \Big( \widehat{w}_l^K \widehat{\sigma}(\widehat{x}_l^K)\,\widehat{\phi}_j(\widehat{x}_l^K)\,\widehat{\phi}_i(\widehat{x}_l^K) \Big), \quad (14)$$

and in an analogous fashion the source term:

$$\int_K f\,\phi_i\,\mathrm{d}\mathbf{x} = |K| \int_{\widehat{K}} \widehat{f}\,\widehat{\phi}_i\,\widehat{\mathrm{d}\mathbf{x}} \simeq |K| \sum_{l=1}^{N_q^K} \Big( \widehat{w}_l^K\,\widehat{f}(\widehat{x}_l^K)\,\widehat{\phi}_i(\widehat{x}_l^K) \Big). \quad (15)$$

The remaining term in (8), i.e., the Neumann contribution can be treated using the one-dimensional quadrature on the generic edge $e \in \Gamma_N$ as:

$$\int_e g_N\,\phi_i\,\mathrm{d}s = |e| \int_{\widehat{e}} \widehat{g}_N\,\widehat{\phi}_i\,\widehat{\mathrm{d}s} \simeq |e| \sum_{l=1}^{N_q^e} \Big( \widehat{w}_l^e\,\widehat{g}_N(\widehat{s}_l^K)\,\widehat{\phi}_i(\widehat{s}_l^K) \Big), \quad (16)$$

where now the hatted quantities, say $\widehat{v}$, are defined through the affine map $T_e : \widehat{e} \to e$, where $\widehat{e}$ is the reference interval $[0,1]$, and thus $\widehat{v} = v \circ T_e$.

In the following example we take in (5): $\Omega = (-1,1)^2$, $\Gamma_N = \{(x,y) : x = 1\ \&\ -1 < y < 1\}$, $\Gamma_D = \partial\Omega \backslash \overline{\Gamma}_N$, $\mu = e^{x+y}$, $\sigma = 1 + e^{x+y}$, $g_N = -\frac{\pi}{2}\sin(\pi(y+1)/2)\,e^{x+y}$, $g_D = 0$, and we construct $f$ so that $u = \sin(\pi(x+1)/2)\sin(\pi(y+1)/2)$ is the exact solution. Algorithm 2 gathers the Matlab code for computing the numerical approximation $u_h$ in (7) along with the auxiliary function `basis` at the bottom.

**Algorithm 2**

```
 1 Np = size(p,2);
 2 Nt = size(t,2);
 3 e1 = find(e(5,:)==1); % top
 4 e2 = find(e(5,:)==2); % right
 5 e3 = find(e(5,:)==3); % bottom
 6 e4 = find(e(5,:)==4); % left
 7 ne1 = union(e(1,e1),e(2,e1));
 8 ne2 = union(e(1,e2),e(2,e2));
 9 ne3 = union(e(1,e3),e(2,e3));
10 ne4 = union(e(1,e4),e(2,e4));
11 % number of Neumann edges
12 Nn  = length(e2);
13 % Dirichlet nodes
14 nD  = unique([ne1,ne3,ne4]);
15 % matrix and vector allocation
16 A = sparse(Np,Np);
17 M = sparse(Np,Np);
```

```
18 F = sparse(Np,1);
19 u = sparse(Np,1);
20 % area of triangles
21 ar = pdetrg(p,t);
22 % quadrature nodes (2 x nq) and weights (1 x nq) on ref. element
23 % nq = # quadrature nodes, sum(w) = 1
24 [xy, w] = dunavant_rule (3);
25 nq = length(w);
26 % basis functions and derivatives on the ref. element (3 x nq)
27 [phi, phi_x, phi_y] = basis(xy);
28 % local nodes on triangles
29 n1 = t(1,:);
30 n2 = t(2,:);
31 n3 = t(3,:);
32 % coordinates of vertices of triangles
33 x1  = p(1,n1);
34 x2  = p(1,n2);
35 x3  = p(1,n3);
36 y1  = p(2,n1);
37 y2  = p(2,n2);
38 y3  = p(2,n3);
39 % array for the local-to-global nodes
40 loc2glob = t(1:3,:);
41 % local nodes on Neumann edges and edge length
42 nN1 = e(1,e2);
43 nN2 = e(2,e2);
44 lN  = sqrt((p(1,nN1)-p(1,nN2)).^2 + (p(2,nN1)-p(2,nN2)).^2);
45 % Gauss-Legendre quadrature (with 3 nodes) on [0,1]
46 [xgl,wgl] = lgwt(3,0,1);
47 xgl = xgl(end:-1:1);
48 % Jacobian of the transformation between ref. element and K
49 % [ x2 - x1, x3 - x1]
50 % [ y2 - y1, y3 - y1]
51 % transponse of the inverse of the Jacobian
52 % [ y3 - y1, y1 - y2]
53 % [ x1 - x3, x2 - x1]/(2*|K|)
54 iMKt11 = (y3 - y1)./(2*ar);
55 iMKt12 = (y1 - y2)./(2*ar);
56 iMKt21 = (x1 - x3)./(2*ar);
57 iMKt22 = (x2 - x1)./(2*ar);
58 % extension of iMKt (nq x Nt)
59 iMKt11q = iMKt11(ones(nq,1),:);
60 iMKt12q = iMKt12(ones(nq,1),:);
61 iMKt21q = iMKt21(ones(nq,1),:);
```

```
 62 iMKt22q = iMKt22(ones(nq,1),:);
 63 % diffusion and reaction coefficients
 64 mu    = @(x,y) exp(x + y);
 65 sigma = @(x,y) 1 + exp(x + y);
 66 % source term, Dirichlet and Neumann data
 67 f = @(x,y) (pi*exp(x + y).*(2*sin((pi*(x + y))/2) + ...
 68     pi*cos((pi*(x + y))/2) + pi*cos((pi*(x - y))/2)))/4 + ...
 69     (1 + exp(x + y)).*sin(pi*(x+1)/2).*sin(pi*(y+1)/2);
 70 gD = @(x,y)  0*x;
 71 gN = @(x,y) -pi/2*sin((pi*(y+1))/2).*exp(x+y);
 72 % assembling of lhs and rhs
 73 for i = 1:3
 74   % extension of the i-th basis function (nq x Nt)
 75   phiq   = phi  (i,:)'; phiq   = phiq  (:,ones(Nt,1));
 76   phi_xq = phi_x(i,:)'; phi_xq = phi_xq(:,ones(Nt,1));
 77   phi_yq = phi_y(i,:)'; phi_yq = phi_yq(:,ones(Nt,1));
 78   % derivative of the i-th basis function on K
 79   Phi_x = iMKt11q.*phi_xq + iMKt12q.*phi_yq;
 80   Phi_y = iMKt21q.*phi_xq + iMKt22q.*phi_yq;
 81   % quadrature nodes on K (nq x Nt)
 82   xq = (1 - xy(1,:) - xy(2,:))'*x1 + xy(1,:)'*x2 + xy(2,:)'*x3;
 83   yq = (1 - xy(1,:) - xy(2,:))'*y1 + xy(1,:)'*y2 + xy(2,:)'*y3;
 84   % rhs
 85   fi = (w*(f(xq,yq).*phiq)).*ar;
 86   F = F + sparse(loc2glob(i,:),1,fi,Np,1);
 87   for j = 1:3
 88     % extension of the j-th basis function (nq x Nt)
 89     phjq   = phi  (j,:)'; phjq   = phjq  (:,ones(Nt,1));
 90     phj_xq = phi_x(j,:)'; phj_xq = phj_xq(:,ones(Nt,1));
 91     phj_yq = phi_y(j,:)'; phj_yq = phj_yq(:,ones(Nt,1));
 92     % derivative of the j-th basis function on K
 93     Phj_x = iMKt11q.*phj_xq + iMKt12q.*phj_yq;
 94     Phj_y = iMKt21q.*phj_xq + iMKt22q.*phj_yq;
 95     % mass matrix
 96     mij = (w*(sigma(xq,yq).*phiq.*phjq)).*ar;
 97     M = M + sparse(loc2glob(i,:),loc2glob(j,:),mij,Np,Np);
 98     % stiffness matrix
 99     aij = (w*(mu(xq,yq).*(Phi_x.*Phj_x + Phi_y.*Phj_y))).*ar;
100     A = A + sparse(loc2glob(i,:),loc2glob(j,:),aij,Np,Np);
101   end
102 end
103 % global stiffness matrix
104 A = A + M;
105 % assembly of the Neumann contribution
```

```
106 phi_edge = basis_edge(xgl');
107 xe = (1 - xgl)*p(1,nN1) + xgl*p(1,nN2);
108 ye = (1 - xgl)*p(2,nN1) + xgl*p(2,nN2);
109 gNe = gN(xe,ye);
110 % extension of the basis function of first node
111 phie = phi_edge(1,:)'; phie = phie(:,ones(Nn,1));
112 fN = (wgl'*(gNe.*phie)).*lN;
113 F  = F + sparse(nN1,1,fN,Np,1);
114 % extension of the basis function of second node
115 phie = phi_edge(2,:)'; phie = phie(:,ones(Nn,1));
116 fN = (wgl'*(gNe.*phie)).*lN;
117 F  = F + sparse(nN2,1,fN,Np,1);
118 % construction of the unknown nodes
119 unk = setdiff(1:Np,nD);
120 % evalutation of the Dirichlet data on the boundary nodes
121 uD = gD(p(1,nD),p(2,nD))';
122 % elimination of the Dirichlet nodes
123 A = A(unk,:);
124 A = A(:,unk);
125 F = F(unk) - A(:,nD)*uD;
126 % solution of the linear system
127 s = A\F;
128 u(unk) = s;
129 u(nD)  = uD;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [phi, phi_x, phi_y] = basis(xy);

Phi   = @(x,y) [1 - x - y     ; x             ; y];
Phi_x = @(x,y) [-ones(size(x)); ones(size(x)) ; zeros(size(x))];
Phi_y = @(x,y) [-ones(size(x)); zeros(size(x)); ones(size(x))];
phi   = Phi  (xy(1,:),xy(2,:));
phi_x = Phi_x(xy(1,:),xy(2,:));
phi_y = Phi_y(xy(1,:),xy(2,:));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

In particular, notice that

1. The four boundary segments defining the domain $\Omega$ are numbered clockwise from 1 to 4, starting from the top segment up to the left segment (see lines `3-6`). The corresponding nodes are defined at lines `7-10` and the Dirichlet nodes at line `14` via the Matlab command `unique`. Notice that the two vertexes at the top-right and bottom-right corners of the domain are Dirichlet nodes. The mesh structures are supposed to be obtained, e.g., through the command `[p,e,t] = initmesh('squareg','hmax',h)`, where the parameter `h` controls the maximum edge size of the triangles;

2. the matrix $M$ allocated at line `17` is the mass matrix, i.e., associated only with the reaction contribution (14), while $A$ is reserved for the stiffness matrix (the diffusion contribution (13), strictly speaking). $M$ is only used temporarily; in fact the global matrix is then overwritten to $A$ at line `104`;

3. the quadrature nodes on the reference triangle are computed at line `24`; notice that the sum of the weights is 1; moreover, the values of the basis functions and of their derivatives on the reference element are computed at line `27`;

4. at line `40`, the array `loc2glob` whose dimensions are $3 \times$ `Nt` is formed. It provides the correspondence between local, i.e., $\{1, 2, 3\}$, and global numbering of the nodes. Although it coincides with the mesh array `t`, this is only a lucky coincidence due to the choice of the finite elements. The array `loc2glob` will in general set the correspondence between local and global degrees of freedom of the finite element space at hand;

5. in the same spirit, at the lines `42-43` the global numbering of the local nodes on the Neumann edges are extracted, along with the length of these edges at line `44`;

6. the nodes and weights of the Gauss-Legendre quadrature formula are computed at line `46` and the nodes are successively swapped in ascending order at line `47`;

7. the entries of the inverse of the transposed Jacobian (12) are computed in parallel for each element at the lines `54-57` as arrays of dimension `Nt`, and are then extended on the quadrature nodes as arrays $N_q^K \times$ `Nt` at lines `59-62`;

8. the data of the problem are defined at lines `64-71` as anonymous functions;

9. At lines `73-102`, the assembling of the mass and stiffness matrices, and of the load vector is implemented through two nested loops of size 3 (number of degrees of freedom of the $\mathbb{P}_1$ FEM per element). In particular, we point out the extension of the basis functions $\widehat{\phi}_i$ at the lines `75-77` and $\widehat{\phi}_j$ at `89-91`; the construction of the corresponding derivatives on $K$ at `79-80` and `93-94`; the computation of contributions due to the source (15) at `85`, the diffusion (13) at `99`, and the reaction (14) at `96`;

10. the contribution due to the Neumann boundary condition is carried out at the lines `106-117`;

11. the program then proceeds by eliminating the rows and columns of the global stiffness matrix associated with the Dirichlet nodes, and by taking into account their effect on the analogously reduced load vector at line `125`, according to (8);

12. finally, the linear system is solved at line `127` an the complete solution is recovered in the last two lines;

13. the function `basis` at the end of the algorithm computes the values of the basis functions and of their partial derivatives as functions of the local coordinates on the reference element.

## 5 The Navier-Stokes equations

We deal with some test cases that represent typical benchmark problems for the Navier-Stokes equations ([21]). The objective is to compute some physical meaningful quantity, such as the lift and drag coefficients for a cross-section of a cylinder in a channel flow, both in steady and unsteady conditions. This allows us to investigate our algorithmic paradigm on a more challenging situation. Let us consider the conservative form (with respect to the stress rate) of the Navier-Stokes equations for an incompressible fluid, completed with mixed boundary conditions:

$$\left\{\begin{array}{rcll} \rho\Big(\dfrac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\,\vec{u}\Big) - \nabla\cdot\boldsymbol{\sigma} &=& \vec{0} & \text{in } \Omega\times(0,T), \\ \nabla\cdot\vec{u} &=& 0 & \text{in } \Omega\times(0,T), \\ \boldsymbol{\sigma}\vec{n} &=& \vec{0} & \text{on } \Gamma_N\times(0,T), \\ \vec{u} &=& \vec{u}_D & \text{on } \Gamma_D\times(0,T), \\ \vec{u} &=& \vec{u}_0 & \text{on } \Omega \text{ at } t=0, \end{array}\right. \tag{17}$$

where the stress rate $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\vec{u},p) = 2\rho\nu\,\boldsymbol{\epsilon}(\vec{u}) - p\boldsymbol{I}$ depends on the velocity $\vec{u}$ and on the pressure $p$, while $\rho = 1.0$ Kg/m$^3$ is the fluid density and $\nu = 10^{-3}$ m$^2$/s is the kinematic viscosity, $\boldsymbol{\epsilon}(\vec{u}) = \frac{1}{2}\big(\nabla\vec{u} + (\nabla\vec{u})^T\big)$ represents the strain rate, and $\boldsymbol{I}$ denotes the identity tensor. The constant $T > 0$ represents the final time level while $\vec{u}_0$ is a given initial velocity profile. The operator $\nabla\cdot$ stands for the divergence (for both vectors and tensors). As is the case of the Poisson problem (5), $\Gamma_D$ and $\Gamma_N$ are two disjoint portions of the domain boundary.
The weak formulation of (17) is: given $\vec{u}(0) = \vec{u}_0$, find $(\vec{u}(t), p(t)) \in (V + \vec{u}_D) \times Q$, such that $\forall(\vec{v}, q) \in V \times Q$, and $t \in (0,T)$,

$$\begin{aligned} 0 &= \int_\Omega \rho\left(\frac{\partial\vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\,\vec{u}\right)\cdot\vec{v}\,\mathrm{d}\mathbf{x} + \int_\Omega \boldsymbol{\sigma}(\vec{u},p) : \boldsymbol{\epsilon}(\vec{v})\,\mathrm{d}\mathbf{x} - \int_\Omega q\,\nabla\cdot\vec{u}\,\mathrm{d}\mathbf{x} \\ &= \int_\Omega \rho\left(\frac{\partial\vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\,\vec{u}\right)\cdot\vec{v}\,\mathrm{d}\mathbf{x} + \int_\Omega 2\rho\nu\boldsymbol{\epsilon}(\vec{u}) : \boldsymbol{\epsilon}(\vec{v})\,\mathrm{d}\mathbf{x} - \int_\Omega p\,\nabla\cdot\vec{v}\,\mathrm{d}\mathbf{x} \\ &- \int_\Omega q\,\nabla\cdot\vec{u}\,\mathrm{d}\mathbf{x}, \end{aligned} \tag{18}$$

where we have introduced the function spaces $V = [H^1_{\Gamma_D}(\Omega)]^2$ and $Q = L^2(\Omega)$, while the inner product between tensors is denoted by ":", i.e., $\boldsymbol{\sigma} : \boldsymbol{\epsilon} = \sum_{i,j=1,2} \boldsymbol{\sigma}_{ij}\boldsymbol{\epsilon}_{ij}$.

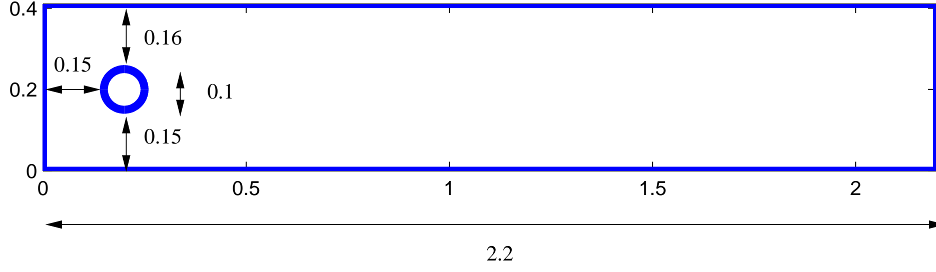


Figura 3: Domain $\Omega$ for the flow past a cylinder test case.

The benchmarks tests at hand are defined on the rectangular channel $\Omega$, with a width $H = 0.41$, drilled with a circular hole representing the cross-section of a cylinder and characterized by a slightly asymmetric configuration (see Figure 3). The boundary conditions are prescribed as follows: we take $\Gamma_D = \Gamma_{in} \cup \Gamma_{cyl} \cup \Gamma_{wall}$, where on the inflow section $\Gamma_{in} = \{(x, y) : x = 0 \,\&\, 0 < y < H\}$, $\vec{u} = [v_{in}, 0]^T$, with $v_{in} = 4\, U_m\, y\, (H - y)/H^2$ the inlet parabolic profile, and $U_m$ the peak velocity, while on the remaining rigid walls $\Gamma_{cyl} \cup \Gamma_{wall}$, the no-slip constraint $\vec{u} = \vec{0}$ holds, where $\Gamma_{cyl}, \Gamma_{wall}$ denote the cylinder and the two horizontal sides, respectively. The Neumann boundary $\Gamma_N$ coincides with the outlet $\Gamma_{out} = \{(x, y) : x = 2.2 \,\&\, 0 < y < H\}$ and the zero-traction condition applies.

The Reynolds number is defined by $Re = \overline{v}_{in} D/\nu$ and is based on the mean velocity $\overline{v}_{in}$ over $\Gamma_{in}$; the cylinder diameter is $D = 0.1$ m. The chosen physical quantities

$$J_{drag} = c_0 \int_{\Gamma_{cyl}} \boldsymbol{\sigma}(\vec{u}, p)\vec{n} \cdot \vec{1}_{\parallel}\, \mathrm{d}s \quad \text{and} \quad J_{lift} = c_0 \int_{\Gamma_{cyl}} \boldsymbol{\sigma}(\vec{u}, p)\vec{n} \cdot \vec{1}_{\perp}\, \mathrm{d}s, \tag{19}$$

represent the so-called drag and lift coefficients, where $\vec{1}_{\parallel} = [1, 0]^T, \vec{1}_{\perp} = [0, 1]^T$ are the unit vectors parallel and orthogonal, respectively to the main flow direction (the horizontal one), with $c_0 = 2/(\rho\, D\, \overline{v}_{in}{}^2)$. The goal of the benchmarks is to compute these quantities as accurate as possible in an efficient way. The results obtained with different approaches by several teams are gathered in [21]. Since the purpose of these notes is to present a computational paradigm based on Matlab, we are not going to push too hard on the computational resources. However, to show that even with a Matlab program run on a notebook featuring quite common performances it is still possible to face very challenging problems, we enhance efficiency by adopting a somewhat original approach that we describe below.

As already observed, e.g., in [9, 2], the straightforward employment of (19) to compute the drag and lift coefficients does not yield accurate results, due to the need of computing numerically first-order derivatives along the cylinder. A more stable and accurate way is instead obtained by resorting to an interior

rather than a boundary integral. In particular, if we pick any two vector fields, $\vec{w}_{drag}, \vec{w}_{lift} : \Omega \to \mathbb{R}^2$, associated with the drag and the lift, as suitable extensions into $\Omega$ of the two unit vectors $\vec{1}_{\parallel}, \vec{1}_{\perp}$, respectively, i.e., on the boundary

$$\vec{w}_{drag} = \begin{cases} \vec{1}_{\parallel} & \text{on } \Gamma_{cyl} \\ \vec{0} & \text{on } \partial\Omega\backslash\Gamma_{cyl} \end{cases} \quad \text{and} \quad \vec{w}_{lift} = \begin{cases} \vec{1}_{\perp} & \text{on } \Gamma_{cyl} \\ \vec{0} & \text{on } \partial\Omega\backslash\Gamma_{cyl}, \end{cases} \tag{20}$$

it is possible to replace (19) by the equivalent form

$$\begin{aligned} J_{drag} &= c_0 \left( \int_\Omega \rho \left( \frac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\,\vec{u} \right) \cdot \vec{w}_{drag}\,\mathrm{d}\mathbf{x} + \int_\Omega \boldsymbol{\sigma}(\vec{u},p) : \boldsymbol{\epsilon}(\vec{w}_{drag})\,\mathrm{d}\mathbf{x} \right), \\ J_{lift} &= c_0 \left( \int_\Omega \rho \left( \frac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\,\vec{u} \right) \cdot \vec{w}_{lift}\,\mathrm{d}\mathbf{x} + \int_\Omega \boldsymbol{\sigma}(\vec{u},p) : \boldsymbol{\epsilon}(\vec{w}_{lift})\,\mathrm{d}\mathbf{x} \right), \end{aligned} \tag{21}$$

where only integrals over $\Omega$ are involved. Indeed, from (21), (20) and (17), we have that

$$\begin{aligned} J_{drag} &= c_0 \int_{\Gamma_{cyl}} \boldsymbol{\sigma}(\vec{u},p)\vec{n}\cdot\vec{1}_{\parallel}\,\mathrm{d}s = c_0 \int_{\Gamma_{cyl}} \boldsymbol{\sigma}(\vec{u},p)\vec{n}\cdot\vec{w}_{drag}\,\mathrm{d}s \\ &= c_0 \int_{\partial\Omega} \boldsymbol{\sigma}(\vec{u},p)\vec{n}\cdot\vec{w}_{drag}\,\mathrm{d}s = c_0 \int_\Omega \nabla\cdot(\boldsymbol{\sigma}(\vec{u},p)\vec{w}_{drag})\,\mathrm{d}\mathbf{x} \\ &= c_0 \left( \int_\Omega (\nabla\cdot\boldsymbol{\sigma}(\vec{u},p))\cdot\vec{w}_{drag}\,\mathrm{d}\mathbf{x} + \int_\Omega \boldsymbol{\sigma}(\vec{u},p) : \boldsymbol{\epsilon}(\vec{w}_{drag})\,\mathrm{d}\mathbf{x} \right) \\ &= c_0 \left( \int_\Omega \rho \left( \frac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\,\vec{u} \right) \cdot \vec{w}_{drag}\,\mathrm{d}\mathbf{x} + \int_\Omega \boldsymbol{\sigma}(\vec{u},p) : \boldsymbol{\epsilon}(\vec{w}_{drag})\,\mathrm{d}\mathbf{x} \right), \end{aligned}$$

and analogously for $J_{lift}$. Notice that (21) can be interpreted as residuals associated with the weak formulation (18), where the test function, $\vec{w}_{drag}, \vec{w}_{lift}$, do not belong to the space $V$ (due to the nonzero value over $\Gamma_{cyl}$). In this respect, expressions (21) can be thought of as weak fluxes. These play a key role in Domain Decomposition methods ([20]).
On the continuous level, (19) and (21) are thoroughly equivalent (under standard regularity conditions for $\boldsymbol{\sigma}(u,p)$ and $\vec{w}_{drag}, \vec{w}_{lift}$). However, this regularity is definitely lost on the discrete level (as long as one does not employ flux conserving FEMs). Nonetheless, after introducing the discrete formulation of (18), we shall employ the natural discretization to (21) as an accurate evaluation of the drag and lift coefficients.

## 5.1 Discretization of the Navier-Stokes equations

The discretization of the Navier-Stokes equations with finite elements is addressed in, e.g., [6, 11, 15, 19]. The approximation of the weak formulation (18) proceeds in a standard way. As far as the spatial discretization is concerned, we employ suitable finite element spaces $V_h \subset V$ and $Q_h \subset Q$, while for the time discretization we resort to a finite different approximation. Thus, we introduce

the partition $\{t_n\}_{n=0}^{N_T}$ consisting of $N_T + 1$ time levels, such that $t_{n+1} = t_n + \Delta t$ for a given constant time step $\Delta t$ (such that $t_0 = 0$ and $t_{N_T} = T$). The choice of a uniform mesh is made just for the sake of simplicity. Then, the discretization of the time derivative $\partial\vec{u}/\partial t$ consists of the backward Euler method, BE, for the first time step (from $t_0$ to $t_1$) and of the second-order Backward Differentiation Formula, BDF2, for the remaining steps. Both approaches can be given a general format, i.e.,

$$\frac{\partial\vec{u}}{\partial t}(t_{n+1}) \simeq \frac{1}{\Delta t}\Big(a_1\,\vec{u}(t_{n+1}) - a_2\,\vec{u}(t_n) - a_3\,\vec{u}(t_{n-1})\Big), \tag{22}$$

where $a_1 = a_2 = 1, a_3 = 0$ (BE), and $a_1 = \frac{3}{2}, a_2 = 2, a_3 = -\frac{1}{2}$ (BDF2). To deal with the nonlinear convective term $(\vec{u}\cdot\nabla)\,\vec{u}$, we employ the following extrapolation

$$[(\vec{u}\cdot\nabla)\,\vec{u}](t_{n+1}) \simeq (\vec{u}^*(t_{n+1})\cdot\nabla)\,\vec{u}(t_{n+1}),\ \text{with}\ \vec{u}^*(t_{n+1}) = b_1\,\vec{u}(t_n) + b_2\,\vec{u}(t_{n-1}), \tag{23}$$

where $b_1 = 1, b_2 = 0$ (first step) and $b_1 = 2, b_2 = -1$ (successive steps). Thus we use a first-order accurate extrapolation at the first step and a second-order method for the further steps. Overall, the discretizations of the time derivative and of the nonlinear term ensure a second-order consistency error at the final time $t = T$.

As far as the choice of the finite element spaces, $V_h, Q_h$, it is well known that any choice will not do. In particular, the so-called *discrete inf-sup condition* must be satisfied in order to obtain a stable discretization ([7]). Neglecting for the moment the boundary conditions, two pairs that are inf-sup stable are provided by the so-called Taylor-Hood element: $V_h = (\mathbb{P}_2)^2$ and $Q_h = \mathbb{P}_1$, and by the *mini-element* (ME): $V_h = (\mathbb{P}_1 \oplus \mathbb{P}_b)^2$ and $Q_h = \mathbb{P}_1$, where $\mathbb{P}_b$ denotes the space of cubic polynomial bubbles, $b$, such that, for any $K \in \mathcal{T}_h$ it holds

$$b \in \mathbb{P}_3(K) \cap H_0^1(K), \quad 0 \le b \le 1, \quad b(C) = 1,$$

where $C$ is the barycenter of $K$.

We are now in a position to state the discrete formulation of the Navier-Stokes equations. We let $\vec{u}_h^n \simeq \vec{u}(t_n), p_h^n \simeq p(t_n), n = 0, 1, \ldots, N_T$. Then given $\vec{u}_h^0$, we are to find $(\vec{u}_h^{n+1}, p_h^{n+1}) \in (V_h + \vec{u}_{D,h}) \times Q_h$ such that, for $n = 0, 1, 2, \ldots, N_T - 1$

$$\begin{aligned}
& \int_\Omega \rho\left(\frac{a_1}{\Delta t}\,\vec{u}_h^{n+1} + \big((b_1\,\vec{u}_h^n + b_2\,\vec{u}_h^{n-1})\cdot\nabla\big)\,\vec{u}_h^{n+1}\right)\cdot\vec{v}_h\,\mathrm{d}\mathbf{x} \\
+ & \int_\Omega 2\rho\nu\,\boldsymbol{\epsilon}(\vec{u}_h^{n+1}) : \boldsymbol{\epsilon}(\vec{v}_h)\,\mathrm{d}\mathbf{x} - \int_\Omega p_h^{n+1}\,\nabla\cdot\vec{v}_h\,\mathrm{d}\mathbf{x} - \int_\Omega q_h\,\nabla\cdot\vec{u}_h^{n+1}\,\mathrm{d}\mathbf{x} \\
= & \int_\Omega \frac{\rho}{\Delta t}\left(a_2\,\vec{u}_h^n + a_3\,\vec{u}_h^{n-1}\right)\cdot\vec{v}_h \quad \forall(\vec{v}_h, q_h) \in V_h \times Q_h,
\end{aligned} \tag{24}$$

where $\vec{u}_h^0, \vec{u}_{D,h}$ are suitable approximations to $\vec{u}_0, \vec{u}_D$, and $V_h \subset [H^1_{\Gamma_D}(\Omega)]^2, Q_h \subset L^2(\Omega)$.

In the following test cases, we employ the discrete space ME. Thus the degrees of freedom of the velocity can be chosen as the values taken at the vertexes of the mesh belonging to $\mathcal{N}_\Omega \cup \mathcal{N}_{\Gamma_N}$ and at the centroids of the elements, while the degrees of freedom of the pressure are associated with the values attained at all the vertexes $\mathcal{N}_\Omega \cup \mathcal{N}_{\Gamma_N} \cup \mathcal{N}_{\Gamma_D}$. Let us denote the set of the indices associated with the degrees of freedom of each velocity component as $\mathcal{N}_U$, and of the pressure as $\mathcal{N}_P$, where it is understood that the first $|\mathcal{N}_\Omega \cup \mathcal{N}_{\Gamma_N}|$ indices of $\mathcal{N}_U$ correspond to the $\mathbb{P}_1$ degrees of freedom (value at vertex), $|\cdot|$ being the cardinality of a set, while the remaining `Nt` indices are associated with the $\mathbb{P}_b$ degrees of freedom (value at centroid), whereas the degrees of freedom of the pressure are all of type $\mathbb{P}_1$. Let us then denote for brevity by $N_U = |\mathcal{N}_\Omega \cup \mathcal{N}_{\Gamma_N}| + \texttt{Nt}$ and $N_P = \texttt{Np}$ the cardinalities of the two sets $\mathcal{N}_U$ and $\mathcal{N}_P$, respectively. After introducing this notation, problem (24) admits an algebraic counterpart which we now describe. Expand the discrete velocity-pressure pair at an arbitrary time level $n$ as

$$\begin{aligned}
\vec{u}_h^n(\mathbf{x}) &= \sum_{j\in\mathcal{N}_U} \vec{u}_h^n(\mathbf{x}_j)\phi_j(\mathbf{x}) + \vec{u}_{D,h}^n(\mathbf{x}), \text{ with } \vec{u}_{D,h}^n(\mathbf{x}) = \sum_{j\in\mathcal{N}_{\Gamma_D}} \vec{u}_D^n(\mathbf{x}_j)\phi_j(\mathbf{x}),\\
p_h^n(\mathbf{x}) &= \sum_{j\in\mathcal{N}_P} p_h^n(\mathbf{x}_j)\psi_j(\mathbf{x}),
\end{aligned}$$

where $\vec{u}_{D,h}^n$ is a suitable approximation to the exact data $\vec{u}_D(t_n)$ constructed only via $\mathbb{P}_1$ degrees of freedom, since the bubble functions are zero on the whole $\partial\Omega$. Other choices for $\vec{u}_{D,h}^n$ are clearly possible but this turns out to be handy in the implementation. The functions $\{\phi_j\}$ and $\{\psi_j\}$ collect the basis functions for the velocity and the pressure, respectively. For a given time level, $n$, we now collect the degrees of freedom of the velocity in the vector $\mathbf{U}^n \in \mathbb{R}^{2N_U}$, with $\mathbf{U}^n = [\mathbf{U}_1^n, \mathbf{U}_2^n]^T$, where $\mathbf{U}_i^n \in \mathbb{R}^{N_U}$ gathers the degrees of freedom associated with the $i$-th component of the velocity field, for $i = 1, 2$, and we group the degrees of freedom of the pressure in the vector $\mathbf{P}^n \in \mathbb{R}^{N_P}$.

Then the algebraic counterpart of (24) reads: given $[\mathbf{U}^0, \mathbf{P}^0]^T \in \mathbb{R}^{2N_U+N_P}$, for $n = 0, 1, 2, \ldots, N_T - 1$ find $\mathbf{W}^{n+1} = [\mathbf{U}^{n+1}, \mathbf{P}^{n+1}]^T \in \mathbb{R}^{2N_U+N_P}$ such that

$$\mathcal{A}^n \mathbf{W}^{n+1} = \mathbf{F}^{n+1}, \tag{25}$$

where $\mathcal{A}^n \in \mathbb{R}^{(2N_U+N_P)\times(2N_U+N_P)}$ is given by

$$\mathcal{A}^n = \begin{bmatrix} \dfrac{a_1}{\Delta t}M + A + N(b_1\mathbf{U}^n + b_2\mathbf{U}^{n-1}) & B^T \\ B & 0 \end{bmatrix}, \tag{26}$$

and the right-hand side $\mathbf{F}^{n+1} \in \mathbb{R}^{2N_U+N_P}$ is

$$\mathbf{F}^{n+1} = \begin{bmatrix} \mathbf{F}_u^{n+1} + \frac{1}{\Delta t}M(a_2\mathbf{U}^n + a_3\mathbf{U}^{n-1}) \\ \mathbf{F}_p^{n+1} \end{bmatrix}. \tag{27}$$

Matrix $M \in \mathbb{R}^{2N_U \times 2N_U}$ is the mass matrix of the velocity, $A \in \mathbb{R}^{2N_U \times 2N_U}$ is the stiffness matrix associated with the viscous stresses, $N(\cdot) \in \mathbb{R}^{2N_U \times 2N_U}$ is the matrix associated with the advective term, $B \in \mathbb{R}^{N_P \times 2N_U}$ collects the contributions of the "divergence" term; on the right-hand side, $\mathbf{F}_u^{n+1} \in \mathbb{R}^{2N_U}$ and $\mathbf{F}_p^{n+1} \in \mathbb{R}^{N_P}$ gather the quantities associated with the Dirichlet conditions at time level $n+1$, and all the contributions of the previous time steps. In more detail, all of these matrices have the following block structure: $M$ is the symmetric two-by-two block diagonal matrix

$$M = \begin{bmatrix} M_u & 0 \\ 0 & M_u \end{bmatrix},$$

with $M_u \in \mathbb{R}^{N_U \times N_U}$ (and zero blocks of the same size), and

$$[M_u]_{i,j} = \int_\Omega \rho\, \phi_j \phi_i \,\mathrm{d}\mathbf{x}, \quad i,j \in \mathcal{N}_U;$$

$A$ is the symmetric two-by-two block matrix

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix},$$

where the four blocks are the $\mathbb{R}^{N_U \times N_U}$ matrices, for $i,j \in \mathcal{N}_U$, given by

$$\begin{aligned}
[A_{11}]_{i,j} &= \int_\Omega 2\rho\nu\, \boldsymbol{\epsilon}([\phi_j,0]^T) : \boldsymbol{\epsilon}([\phi_i,0]^T)\,\mathrm{d}\mathbf{x} = \int_\Omega 2\rho\nu \left( \frac{\partial \phi_j}{\partial x}\frac{\partial \phi_i}{\partial x} + \frac{1}{2}\frac{\partial \phi_j}{\partial y}\frac{\partial \phi_i}{\partial y} \right) \mathrm{d}\mathbf{x} \\
[A_{12}]_{i,j} &= \int_\Omega 2\rho\nu\, \boldsymbol{\epsilon}([\phi_j,0]^T) : \boldsymbol{\epsilon}([0,\phi_i]^T)\,\mathrm{d}\mathbf{x} = \int_\Omega 2\rho\nu\, \frac{1}{2}\frac{\partial \phi_j}{\partial x}\frac{\partial \phi_i}{\partial y}\,\mathrm{d}\mathbf{x} \\
[A_{22}]_{i,j} &= \int_\Omega 2\rho\nu\, \boldsymbol{\epsilon}([0,\phi_j]^T) : \boldsymbol{\epsilon}([0,\phi_i]^T)\,\mathrm{d}\mathbf{x} = \int_\Omega 2\rho\nu \left( \frac{\partial \phi_j}{\partial y}\frac{\partial \phi_i}{\partial y} + \frac{1}{2}\frac{\partial \phi_j}{\partial x}\frac{\partial \phi_i}{\partial x} \right) \mathrm{d}\mathbf{x};
\end{aligned}$$

$N(\vec{v}_h)$, for a dummy discrete velocity field $\vec{v}_h$, is the nonsymmetric two-by-two advection matrix

$$N(\vec{v}_h) = \begin{bmatrix} N_u & 0 \\ 0 & N_u \end{bmatrix},$$

with $N_u \in \mathbb{R}^{N_U \times N_U}$ (and zero blocks of the same size), and

$$[N_u]_{i,j} = \int_\Omega \rho\, \vec{v}_h \cdot \nabla \phi_j\, \phi_i \,\mathrm{d}\mathbf{x}, \quad i,j \in \mathcal{N}_U;$$

$B$ is the one-by-two block matrix

$$B = [B_1 \quad B_2],$$

with $B_i \in \mathbb{R}^{N_P \times N_U}$, for $i = 1, 2$, and

$$[B_1]_{i,j} = -\int_\Omega \frac{\partial \phi_j}{\partial x}\psi_i \,\mathrm{d}\mathbf{x}, \quad [B_2]_{i,j} = -\int_\Omega \frac{\partial \phi_j}{\partial y}\psi_i \,\mathrm{d}\mathbf{x}, \quad i \in \mathcal{N}_P, j \in \mathcal{N}_U;$$

$\mathbf{F}_u^{n+1}$ is given by $[\mathbf{F}_{u,1}^{n+1}\ \mathbf{F}_{u,2}^{n+1}]^T$, with $\mathbf{F}_{u,i}^{n+1} \in \mathbb{R}^{N_U}$, $i = 1, 2$, and, for $i \in \mathcal{N}_U$,

$$\begin{aligned}
[\mathbf{F}_{u,1}^{n+1}]_i = & - \int_\Omega \rho \left( \frac{a_1}{\Delta t}\, \vec{u}_{D,h}^{n+1} + \left( (b_1\, \vec{u}_h^n + b_2\, \vec{u}_h^{n-1}) \cdot \nabla \right) \vec{u}_{D,h}^{n+1} \right) \cdot [\phi_i, 0]^T \, \mathrm{d}\mathbf{x} \\
& + \int_\Omega \boldsymbol{\epsilon}(\vec{u}_{D,h}^{n+1}) : \boldsymbol{\epsilon}([\phi_i, 0]^T)\, \mathrm{d}\mathbf{x} + \int_\Omega \frac{\rho}{\Delta t} \left( a_2\, \vec{u}_h^n + a_3\, \vec{u}_h^{n-1} \right) \cdot [\phi_i, 0]^T \\
[\mathbf{F}_{u,2}^{n+1}]_i = & - \int_\Omega \rho \left( \frac{a_1}{\Delta t}\, \vec{u}_{D,h}^{n+1} + \left( (b_1\, \vec{u}_h^n + b_2\, \vec{u}_h^{n-1}) \cdot \nabla \right) \vec{u}_{D,h}^{n+1} \right) \cdot [0, \phi_i]^T \, \mathrm{d}\mathbf{x} \\
& + \int_\Omega \boldsymbol{\epsilon}(\vec{u}_{D,h}^{n+1}) : \boldsymbol{\epsilon}([0, \phi_i]^T)\, \mathrm{d}\mathbf{x} + \int_\Omega \frac{\rho}{\Delta t} \left( a_2\, \vec{u}_h^n + a_3\, \vec{u}_h^{n-1} \right) \cdot [0, \phi_i]^T,
\end{aligned}$$

while $\mathbf{F}_p^{n+1} \in \mathbb{R}^{N_p}$ has components

$$[\mathbf{F}_p^{n+1}]_i = \int_\Omega \psi_i \nabla \cdot \vec{u}_{D,h}^{n+1}\, \mathrm{d}\mathbf{x}, \quad i \in \mathcal{N}_P.$$

Notice that the matrix $\mathcal{A}^n$ in (26) is time dependent due to the advection contribution only.

As for the actual computation of the drag and lift, we adopt the discrete counterpart of (21), replacing the pair $(\vec{u}, p)$ with $(\vec{u}_h, p_h)$, approximating the time derivative with the corresponding finite differences, and using as $\vec{w}_{drag}, \vec{w}_{lift}$ the harmonic finite element functions which satisfy (20) obtained as follows. We first construct the harmonic function, $Z_h \in \mathbb{P}_1 \oplus \mathbb{P}_b$, satisfying $Z_h = 1$ on $\Gamma_{cyl}$ and $Z_h = 0$ on $\partial\Omega \backslash \Gamma_{cyl}$. Then we define $\vec{w}_{drag} = [Z_h, 0]^T$ and $\vec{w}_{lift} = [0, Z_h]^T$. On the discrete level, this amounts to computing the quantity

$$J_{drag} = c_0 [\mathbf{Z}\ 0]^T (\mathcal{A}^n \mathbf{W}^{n+1} - \mathbf{F}^{n+1}) \quad J_{lift} = c_0 [0\ \mathbf{Z}]^T (\mathcal{A}^n \mathbf{W}^{n+1} - \mathbf{F}^{n+1}), \quad (28)$$

where $\mathbf{Z} \in \mathbb{R}^{2N_U}$ collects the degrees of freedom of the function $Z_h$, and the zero block has dimension $N_P$.

# 6 The benchmark test cases

We are now ready to present the numerical results on some benchmarks proposed in [21]. For simplicity, here we only focus on the drag and lift coefficients. Other quantities, such as pressure difference, Strouhal number and length of recirculation will not be addressed.

## 6.1 Test case 2D-1

This is a stationary test case, with $U_m = 0.3$, and thus $Re = 20$. The drag and lift coefficients have to be computed. Table 2 summarizes the results for this test case, showing the number of elements, `Nt`, the number of vertexes, `Np`, the size of the global stiffness matrix, $S$, the average CPU time, $T_1$, required to complete one time step, the values of the drag and lift coefficients, for three

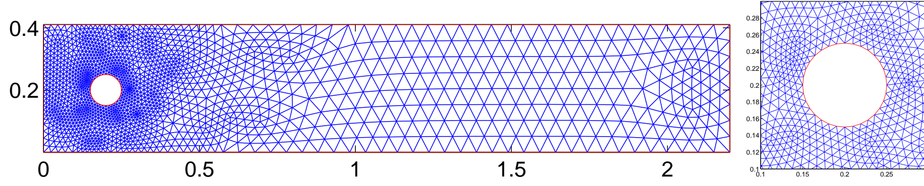

Figura 4: First mesh for the test case 2D-1 (left) and zoom in on the cylinder (right).

Tabella 2: Test case 2D-1. Number of elements `Nt`; number of nodes `Np`; matrix size $S$; average CPU time per time step $T_1$; drag coefficient $J_{drag}$; lift coefficient $J_{lift}$.

| Nt | Np | $S$ | $T_1$ | $J_{drag}$ | $J_{lift}$ |
|---|---|---|---|---|---|
| 3,392 | 1,788 | 11,794 | 0.9 | $5.5811 \cdot 10^{+00}$ | $-2.8478 \cdot 10^{-03}$ |
| 13,568 | 6,968 | 47,334 | 4.3 | $5.5799 \cdot 10^{+00}$ | $7.1270 \cdot 10^{-03}$ |
| 15,560 | 7,980 | 54,318 | 5.1 | $5.5798 \cdot 10^{+00}$ | $2.3325 \cdot 10^{-02}$ |

different computational meshes. The first mesh (see Figure. 4) exhibits some refinement around the cylinder. In particular, 32 elements are placed around the cylinder. The second mesh is obtained through a uniform refinement of the first one, i.e., each triangle is divided into four similar triangle by joining the mid-edges, and thus it has 64 elements attached to the cylinder. The third mesh is a uniform mesh characterized by a maximum mesh size of 0.014 throughout all of the domain. Around the cylinder there are only 24 elements. The number, $S$, of global unknowns is comparable to the coarsest and medium size meshes of the benchmarks reported in [21], which span from 6,562 to 30,775,296. In order to get the steady state solution, we chose $\Delta t = 10^{-2}$ and $T = 8$. The values in Table 2 for the drag coefficient are consistent with the range deduced in [21], i.e., $5.5700 \leq J_{drag} \leq 5.5900$. For this test case, the lift coefficient turns out to be the most critical quantity to compute. Actually, the values in Table 2, although not strictly in the range $1.04 \cdot 10^{-02} \leq J_{lift} \leq 1.10 \cdot 10^{-02}$ suggested in [21], are in any case of the same order as many of the values reported in [21]. Notice also that, it takes about 13 minutes (in the best case) and 1 hour (in the worst) for the program to execute a full sweep over 800 time steps.

## 6.2 Test case 2D-2

This is an unsteady test case, where $U_m = 1.5$, yielding $Re = 100$. It is required to compute the drag and lift coefficients as functions of time for one period $[t_0, t_0 + 1/f]$ (with $f = f(J_{lift})$), maximum drag coefficient, $J_{drag,M}$, maximum lift coefficient, $J_{lift,M}$. The initial data ($t = t_0$) should correspond to the flow state with $J_{lift,M}$. For the simulation, we employed $\Delta t = 10^{-2}$ and $T = 8$. Ta-

ble 3 shows the maximum value of the drag, $J_{drag,M}$, and lift, $J_{lift,M}$ coefficients over the period. The computational work per time step is the same as in the previous test case. The computed values appear to be comparable to the reference values in [21], i.e., $3.2200 \leq J_{drag,M} \leq 3.2400$ and $0.9900 \leq J_{lift,M} \leq 1.0100$.

Tabella 3: Test case 2D-2. Number of elements `Nt`; number of nodes `Np`; matrix size $S$; average CPU time per time step $T_1$; maximum drag coefficient $J_{drag,M}$; maximum lift coefficient $J_{lift,M}$.

| `Nt` | `Np` | $S$ | $T_1$ | $J_{drag,M}$ | $J_{lift,M}$ |
|---:|---:|---:|---:|---:|---:|
| 3,392 | 1,788 | 11,794 | 0.9 | $3.2289 \cdot 10^{+00}$ | $1.0088 \cdot 10^{+00}$ |
| 13,568 | 6,968 | 47,334 | 4.3 | $3.2529 \cdot 10^{+00}$ | $1.0385 \cdot 10^{+00}$ |
| 15,560 | 7,980 | 54,318 | 5.1 | $3.1065 \cdot 10^{+00}$ | $8.0528 \cdot 10^{-01}$ |

## 6.3 Test case 2D-3

Tabella 4: Test case 2D-3. Number of elements `Nt`; number of nodes `Np`; matrix size $S$; average CPU time per time step $T_1$; maximum drag coefficient $J_{drag,M}$; maximum lift coefficient $J_{lift,M}$.

| `Nt` | `Np` | $S$ | $T_1$ | $J_{drag,M}$ | $J_{lift,M}$ |
|---:|---:|---:|---:|---:|---:|
| 3,392 | 1,788 | 11,794 | 0.9 | $2.9106 \cdot 10^{+00}$ | $5.5318 \cdot 10^{-01}$ |
| 13,568 | 6,968 | 47,334 | 4.3 | $2.9370 \cdot 10^{+00}$ | $5.7065 \cdot 10^{-01}$ |
| 15,560 | 7,980 | 54,318 | 5.1 | $2.8843 \cdot 10^{+00}$ | $4.3683 \cdot 10^{-01}$ |

This is a time dependent problem, where the inflow condition is $v_{in} = 4\,U_m\,y\,(H-y)\sin(\pi t/8)/H^2$, with $U_m = 1.5$, and the time interval is $0 \leq t \leq 8$. This gives a time varying Reynolds number between $0 \leq Re(t) \leq 100$. The initial data ($t = 0$) are $\vec{u}_0 = \vec{0}, P = 0$. The following quantities should be computed: drag and lift coefficients as functions of time for $0 \leq t \leq 8$, maximum drag coefficient, $J_{drag,M}$, and maximum lift coefficient, $J_{lift,M}$. Although not explicitly mentioned in [21], the value of $\overline{v}_{in}$ to be used in the definitions (19) is the value associated with the maximum height of the inflow profile, i.e., at $t = 4$. Table 4 gathers the results for this test case. Again, $\Delta t = 10^{-2}$ and $T = 8$ were chosen. Also in this case, the computed drag and lift coefficients are consistent with the reference values in [21], i.e., $2.9300 \leq J_{drag,M} \leq 2.9700$ and $0.4700 \leq J_{lift,M} \leq 0.4900$.

The full list of the program, for the test case 2D-1, is provided in the Appendix.

## 7 Conclusions

We have developed a numerical tool for solving problems described by partial differential equations. In particular, our tool is based on a simple and short open-box Matlab implementation of FEMs on 2D triangular meshes. Our effort has been mainly devoted to making the program as efficient as possible, in order to run in few seconds or minutes (in case of a steady simulation). The employment of Matlab is by no means restrictive. Our choice was mainly motivated by the fact that it still represents a very popular interactive environment for numerical computation, visualization, and programming. However, simple modifications allow one to adapt our tool to other high-level interpreted languages, such as GNU Octave [10].

To describe our approach, we have first considered the Poisson problem, completed with both Dirichlet and Neumann boundary conditions, as one of the typical problems which appear in Scientific Computing. Then we have generalized this to the diffusion-reaction problem, where both the diffusion and reaction coefficients, as well as the boundary data may vary as a function of position, thus requiring the use of suitable quadrature rules. Finally, we have dealt with the steady and unsteady incompressible Navier-Stokes equations. In all cases, it turns out that low computing times can be really achieved even for the more challenging problems represented by the Navier-Stokes equations. This was made possible by first adopting a suitable data structure to describe the mesh, and secondly, by a thorough use of vectorization programming and sparse matrix storage and operations which is made possible within the Matlab environment. Only 2D configurations have been addressed in this paper. However, we have already extended all of this techniques to the 3D case, and impressive improvements of the computing times can be obtained also in this case. As a possible task for the future, it would be desirable to develop a tool for easily generating a data structure suitable to deal with higher-order polynomials, i.e., $\mathbb{P}_k$, with $k \geq 2$.

## Appendix A

**Algorithm 3**

```
 1 % physical and geometrical parameters
 2 rho = 1;
 3 nu  = 1e-3;
 4 mu  = rho*nu;
 5 Um  = 0.3;
 6 H   = 0.41;
 7 D   = 0.1;
 8 Ubar = 2*(4*Um*H/2*(H-H/2)/H^2)/3; % mean inflow velocity
 9 Re = Ubar*D/nu;
10
```

```
11 % data for time discretization
12 dt = 0.01;
13 Tf = 8;
14 time = linspace(0,Tf,Tf/dt+1);
15
16 % load mesh and geometry: p, e, t
17 load cylinder
18
19 % classification of the nodes
20 [ninf,nout,ndir,nunk] = find_nodes(p,e,t);
21
22 % choice of FEM space
23 % 'TH' = Taylor-Hood  (P2/P1)
24 % 'ME' = Mini-element (P1b/P1)
25 element = 'ME'
26
27 % coefficients of backward Euler
28 a1 = 1; a2 = 1; a3 = 0;
29
30 % extrapolation of convective term: b1*u^n + b2*u^{n-1}
31 % at first time step: first-order extrapolation
32 b1 = 1; b2 = 0;
33
34 % quadrature nodes (2 x nq) and weights (1 x nq) on ref. triangle
35 % sum(w) = 1
36 [ xy, w ] = dunavant_rule (5);
37 nq = length(w);
38
39 % velocity (phi) and pressure (psi) basis functions
40 % on the reference triangle: nb x nq, nb = # local basis functions
41 [phi, phi_x, phi_y, psi, psi_x, psi_y] = basis(xy,element);
42
43 % number of global dofs for velocity (dofV) and pressure (dofP)
44 % loc2globV: nbV x Nt
45 % loc2globP: nbP x Nt
46 [dofV, dofP, loc2globV, loc2globP] = dofs(p,t,element);
47
48 % area of triangles and number of vertices and elements
49 ar = pdetrg(p,t);
50 Nt = size(t,2);
51 Np = size(p,2);
52
53 nbV = size(phi,1); % # local basis functions for velocity
54 nbP = size(psi,1); % # local basis functions for pressure
```

```
55 ZZ  = sparse(dofP,dofP);
56 ZP  = sparse(dofP,1);
57
58 % initial conditions
59 U  = sparse(dofV,1);
60 U1 = sparse(dofV,1); % u^{n-1}
61 U2 = sparse(dofV,1); % u^{n-2}
62 V  = sparse(dofV,1); % v^{n-1}
63 V1 = sparse(dofV,1); % v^{n-2}
64 V2 = sparse(dofV,1);
65 P  = sparse(dofP,1);
66
67 % coordinates of local (1, 2 and 3) node of each triangle
68 n1 = t(1,:);
69 n2 = t(2,:);
70 n3 = t(3,:);
71 x1  = p(1,n1);
72 x2  = p(1,n2);
73 x3  = p(1,n3);
74 y1  = p(2,n1);
75 y2  = p(2,n2);
76 y3  = p(2,n3);
77
78 % matrix and vector allocation
79 A   = sparse(2*dofV+dofP,2*dofV+dofP);
80 F   = sparse(2*dofV+dofP,1);
81 M   = sparse(dofV,dofV);
82 A11 = sparse(dofV,dofV);
83 A12 = sparse(dofV,dofV);
84 A22 = sparse(dofV,dofV);
85 B1t = sparse(dofV,dofP);
86 B2t = sparse(dofV,dofP);
87
88 % Jacobian of the mapping from reference element to K
89 % [ x2 - x1, x3 - x1]
90 % [ y2 - y1, y3 - y1]
91 %
92 % inverse of transposed Jacobian
93 % [ y3 - y1, y1 - y2]
94 % [ x1 - x3, x2 - x1]/(2*|K|)
95 iMKt11 = (y3 - y1)./(2*ar);
96 iMKt12 = (y1 - y2)./(2*ar);
97 iMKt21 = (x1 - x3)./(2*ar);
98 iMKt22 = (x2 - x1)./(2*ar);
```

```
 99 % extension on the quadrature nodes of iMKt
100 iMKt11q = iMKt11(ones(nq,1),:);
101 iMKt12q = iMKt12(ones(nq,1),:);
102 iMKt21q = iMKt21(ones(nq,1),:);
103 iMKt22q = iMKt22(ones(nq,1),:);
104
105 % lift and drag initialization
106 cD = zeros(length(time),1);
107 cL = zeros(length(time),1);
108
109 for n = 1:(length(time)-1)
110
111   N1  = sparse(dofV,dofV);
112   N2  = sparse(dofV,dofV);
113
114   if n == 1
115
116     for i = 1:nbV
117
118       % extension of the i-th basis function: nq x Nt
119       phiq   = phi  (i,:)'; phiq   = phiq  (:,ones(Nt,1));
120       phi_xq = phi_x(i,:)'; phi_xq = phi_xq(:,ones(Nt,1));
121       phi_yq = phi_y(i,:)'; phi_yq = phi_yq(:,ones(Nt,1));
122       % derivative of i-th basis function on K
123       Phi_x = iMKt11q.*phi_xq + iMKt12q.*phi_yq;
124       Phi_y = iMKt21q.*phi_xq + iMKt22q.*phi_yq;
125
126       for j = 1:nbV
127
128         % extension of the j-th basis function: nq x Nt
129         phjq   = phi  (j,:)'; phjq   = phjq  (:,ones(Nt,1));
130         phj_xq = phi_x(j,:)'; phj_xq = phj_xq(:,ones(Nt,1));
131         phj_yq = phi_y(j,:)'; phj_yq = phj_yq(:,ones(Nt,1));
132
133         % velocity mass matrix
134         mij = rho*(w*(phiq.*phjq)).*ar/dt;
135         M = M + sparse(loc2globV(i,:),loc2globV(j,:),mij,dofV,dofV);
136
137         % derivative of j-th basis function on K
138         Phj_x = iMKt11q.*phj_xq + iMKt12q.*phj_yq;
139         Phj_y = iMKt21q.*phj_xq + iMKt22q.*phj_yq;
140
141         % viscous strains stiffness matrix
142         a11 = 2*mu*(w*(Phi_x.*Phj_x + Phi_y.*Phj_y/2)).*ar;
```

```
143        a12 = 2*mu*(w*(Phi_y.*Phj_x/2)).*ar;
144        a22 = 2*mu*(w*(Phi_y.*Phj_y + Phi_x.*Phj_x/2)).*ar;
145        A11 = A11 + sparse(loc2globV(i,:),loc2globV(j,:),a11,dofV,dofV);
146        A12 = A12 + sparse(loc2globV(i,:),loc2globV(j,:),a12,dofV,dofV);
147        A22 = A22 + sparse(loc2globV(i,:),loc2globV(j,:),a22,dofV,dofV);
148
149        % convection matrix N(u_h)
150        for l = 1:nbV
151          U1K = U1(loc2globV(l,:))';
152          U2K = U2(loc2globV(l,:))';
153          V1K = V1(loc2globV(l,:))';
154          V2K = V2(loc2globV(l,:))';
155          phlq   = phi  (l,:)'; phlq   = phlq  (:,ones(Nt,1));
156          nl1 = w*(phiq.*phlq.*Phj_x).*(b1*U1K + b2*U2K).*ar;
157          nl2 = w*(phiq.*phlq.*Phj_y).*(b1*V1K + b2*V2K).*ar;
158          N1 = N1 + sparse(loc2globV(i,:),loc2globV(j,:),nl1,dofV,dofV);
159          N2 = N2 + sparse(loc2globV(i,:),loc2globV(j,:),nl2,dofV,dofV);
160        end
161
162      end
163
164      for j = 1:nbP
165
166        psjq   = psi  (j,:)'; psjq   = psjq  (:,ones(Nt,1));
167        % ``divergence'' matrix B
168        b1t = -(w*(Phi_x.*psjq)).*ar;
169        b2t = -(w*(Phi_y.*psjq)).*ar;
170        B1t = B1t + sparse(loc2globV(i,:),loc2globP(j,:),b1t,dofV,dofP);
171        B2t = B2t + sparse(loc2globV(i,:),loc2globP(j,:),b2t,dofV,dofP);
172
173      end
174    end
175
176    % auxiliary variable for drag & lift
177    Z = compute_Z(p,e,t,ar,nbV,dofV,dofP,loc2globV,w,phi_x,phi_y);
178
179  else
180
181    for i = 1:nbV
182      % extension of the i-th basis function: nq x Nt
183      phiq   = phi  (i,:)'; phiq   = phiq  (:,ones(Nt,1));
184      phi_xq = phi_x(i,:)'; phi_xq = phi_xq(:,ones(Nt,1));
185      phi_yq = phi_y(i,:)'; phi_yq = phi_yq(:,ones(Nt,1));
186      % derivative of i-th basis function on K
```

```
187        Phi_x = iMKt11q.*phi_xq + iMKt12q.*phi_yq;
188        Phi_y = iMKt21q.*phi_xq + iMKt22q.*phi_yq;
189
190        for j = 1:nbV
191
192          % extension of the j-th basis function: nq x Nt
193          phjq   = phi  (j,:)'; phjq   = phjq  (:,ones(Nt,1));
194          phj_xq = phi_x(j,:)'; phj_xq = phj_xq(:,ones(Nt,1));
195          phj_yq = phi_y(j,:)'; phj_yq = phj_yq(:,ones(Nt,1));
196          % derivative of j-th basis function on K
197          Phj_x = iMKt11q.*phj_xq + iMKt12q.*phj_yq;
198          Phj_y = iMKt21q.*phj_xq + iMKt22q.*phj_yq;
199
200          % advection matrix N(u_h)
201          for l = 1:nbV
202            U1K = U1(loc2globV(l,:))';
203            U2K = U2(loc2globV(l,:))';
204            V1K = V1(loc2globV(l,:))';
205            V2K = V2(loc2globV(l,:))';
206            phlq   = phi  (l,:)'; phlq   = phlq  (:,ones(Nt,1));
207            nl1 = w*(phiq.*phlq.*Phj_x).*(b1*U1K + b2*U2K).*ar;
208            nl2 = w*(phiq.*phlq.*Phj_y).*(b1*V1K + b2*V2K).*ar;
209            N1 = N1 + sparse(loc2globV(i,:),loc2globV(j,:),nl1,dofV,dofV);
210            N2 = N2 + sparse(loc2globV(i,:),loc2globV(j,:),nl2,dofV,dofV);
211          end
212
213        end
214      end
215    end
216
217    N = rho*(N1 + N2);
218    % global stiffness and load vector
219    A = [a1*M+A11+N, A12       , B1t;
220         A12'      , a1*M+A22+N, B2t;
221         B1t'      , B2t'      , ZZ ];
222
223    F = [M*(a2*U1+a3*U2); M*(a2*V1+a3*V2); ZP];
224
225    % boundary conditions at inflow
226    uinf = assign_inflow(p,ninf,Um,H,time(n+1));
227
228    % global indices of unknowns
229    unk_glob = [nunk,Np+(1:Nt),dofV + [nunk,Np+(1:Nt)],2*dofV+(1:Np)];
230    Aglob = A;
```

```
231   Fglob = F;
232   A = A(unk_glob,:);
233   F = F(unk_glob);
234   F = F - A(:,ninf)*uinf;
235   A = A(:,unk_glob);
236
237   % solution of the linear system
238   numb_unkV = length(nunk) + Nt;
239   uvp  = A\F;
240   u    = uvp(1:numb_unkV);
241   v    = uvp([1:numb_unkV] + numb_unkV);
242   P    = uvp(2*numb_unkV+1:end);
243   U([nunk,(1:Nt)+Np]) = u;
244   V([nunk,(1:Nt)+Np]) = v;
245   U(ninf) = uinf;
246
247   % drag & lift computation
248   [cD(n+1), cL(n+1)] = compute_drag_lift(dofV,dofP,Aglob,Fglob,...
                                           U,V,P,Z,rho,Ubar,D);
249
250   % update velocity for next time step
251   U2 = U1;
252   U1 = U;
253   V2 = V1;
254   V1 = V;
255
256   % coefficient of BDF2
257   a1 = 3/2; a2 = 2; a3 = -1/2;
258   % coefficients of second-order extrapolation
259   b1 = 2;   b2 = -1;
260
261 end
```

The auxiliary functions required by this program are gathered below

```
function [ninf,nout,ndir,nunk] = find_nodes(p,e,t)

% ninf = inflow nodes
% nout = outflow nodes
% ndir = Dirichlet nodes = ninf + walls + cylinder
% nout = unknown nodes   = internal + outflow

Np = size(p,2);

% 1   = right
```

```
% 2   = top
% 3   = left
% 4   = bottom
% 5-8 = cylinder

e1 = find(e(5,:) == 1);
e2 = find(e(5,:) == 2);
e3 = find(e(5,:) == 3);
e4 = find(e(5,:) == 4);
e5 = find(e(5,:) == 5);
e6 = find(e(5,:) == 6);
e7 = find(e(5,:) == 7);
e8 = find(e(5,:) == 8);

n1 = union(e(1,e1),e(2,e1));
n2 = union(e(1,e2),e(2,e2));
n3 = union(e(1,e3),e(2,e3));
n4 = union(e(1,e4),e(2,e4));
n5 = union(e(1,e5),e(2,e5));
n6 = union(e(1,e6),e(2,e6));
n7 = union(e(1,e7),e(2,e7));
n8 = union(e(1,e8),e(2,e8));

ninf = n3;
nout = setdiff(n1,union(n2,n4));
ndir = unique([n2,n3,n4,n5,n6,n7,n8]);
nunk = setdiff(1:Np,ndir);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [phi, phi_x, phi_y, psi, psi_x, psi_y] = basis(xy,element);

switch element

  case 'TH'
    disp('TH element not yet implemented')

  case 'ME'

    Phi   = @(x,y) [1 - x - y     ; x             ; y ; ...
                    27*(1 - x - y)*x*y];
    Phi_x = @(x,y) [-ones(size(x)); ones(size(x)) ; zeros(size(x)); ...
                    -27*y*(2*x+y-1)];
    Phi_y = @(x,y) [-ones(size(x)); zeros(size(x)); ones(size(x)) ; ...
```

```
                    -27*x*(2*y+x-1)];
    phi   = Phi  (xy(1,:),xy(2,:));
    phi_x = Phi_x(xy(1,:),xy(2,:));
    phi_y = Phi_y(xy(1,:),xy(2,:));
    %
    psi   = phi(1:3,:);
    psi_x = phi_x(1:3,:);
    psi_y = phi_y(1:3,:);

  otherwise
    disp('Unknown method!')

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [dofV, dofP, loc2globV, loc2globP] = dofs(p,t,element);

Np = size(p,2);
Nt = size(t,2);

% Euler's theorem: Nt + Np = Nl + 1 (where Nl = # all edges)

switch element

  case 'TH'
    display('TH element not yet implemented')

  case 'ME'
    dofV = Np + Nt;
    dofP = Np;
    loc2globV = zeros(4,Nt);
    loc2globV(1:3,:) = t(1:3,:);
    loc2globV(4,:)   = Np + [1:Nt];
    loc2globP        = t(1:3,:);

  otherwise
    disp('Unknown method!')

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function uinf = assign_inflow(p,ninf,Um,H,t)
```

```
y = p(2,ninf)';
uinf = 4*Um*y.*(H-y)/H^2;

% for test case 2D-3
%uinf = 4*Um*y.*(H-y)*sin(pi*t/8)/H^2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function Z = compute_Z(p,e,t,ar,nbV,dofV,dofP,loc2globV,wq,phi_x,phi_y)

% computes the auxiliary velocity variable Z:
% -Delta Z = 0 in Omega
%        Z = 1 on cylinder
%        Z = 0 elsewhere

Np = size(p,2);
Nt = size(t,2);
nq = length(wq);

% 1   = right
% 2   = top
% 3   = left
% 4   = bottom
% 5-8 = cylinder

e1 = find(e(5,:) == 1);
e2 = find(e(5,:) == 2);
e3 = find(e(5,:) == 3);
e4 = find(e(5,:) == 4);
e5 = find(e(5,:) == 5);
e6 = find(e(5,:) == 6);
e7 = find(e(5,:) == 7);
e8 = find(e(5,:) == 8);

n1 = union(e(1,e1),e(2,e1));
n2 = union(e(1,e2),e(2,e2));
n3 = union(e(1,e3),e(2,e3));
n4 = union(e(1,e4),e(2,e4));
n5 = union(e(1,e5),e(2,e5));
n6 = union(e(1,e6),e(2,e6));
n7 = union(e(1,e7),e(2,e7));
n8 = union(e(1,e8),e(2,e8));
```

```
ndir1 = unique([n5,n6,n7,n8]);
ndir0 = unique([n1,n2,n3,n4]);
nunk  = setdiff(1:Np,[ndir0,ndir1]);

nK1 = t(1,:);
nK2 = t(2,:);
nK3 = t(3,:);
x1  = p(1,nK1);
x2  = p(1,nK2);
x3  = p(1,nK3);
y1  = p(2,nK1);
y2  = p(2,nK2);
y3  = p(2,nK3);

% Jacobian from reference element to K
% [ x2 - x1, x3 - x1]
% [ y2 - y1, y3 - y1]
%
% The transposed inverse Jacobian from reference element to K
% [ y3 - y1, y1 - y2]
% [ x1 - x3, x2 - x1]/(2*|K|)
iMKt11 = (y3 - y1)./(2*ar);
iMKt12 = (y1 - y2)./(2*ar);
iMKt21 = (x1 - x3)./(2*ar);
iMKt22 = (x2 - x1)./(2*ar);
% extension of iMKt
iMKt11q = iMKt11(ones(nq,1),:);
iMKt12q = iMKt12(ones(nq,1),:);
iMKt21q = iMKt21(ones(nq,1),:);
iMKt22q = iMKt22(ones(nq,1),:);

A = sparse(dofV,dofV);
F = sparse(dofV,1);
Z = sparse(dofV,1);

for i = 1:nbV

  % extension of the i-th basis function: nq x Nt
  phi_xq = phi_x(i,:)'; phi_xq = phi_xq(:,ones(Nt,1));
  phi_yq = phi_y(i,:)'; phi_yq = phi_yq(:,ones(Nt,1));
  % derivative of i-th basis function on K
  Phi_x = iMKt11q.*phi_xq + iMKt12q.*phi_yq;
  Phi_y = iMKt21q.*phi_xq + iMKt22q.*phi_yq;
```

```
  for j = 1:nbV

    % extension of the j-th basis function: nq x Nt
    phj_xq = phi_x(j,:)'; phj_xq = phj_xq(:,ones(Nt,1));
    phj_yq = phi_y(j,:)'; phj_yq = phj_yq(:,ones(Nt,1));

    % derivative of j-th basis function on K
    Phj_x = iMKt11q.*phj_xq + iMKt12q.*phj_yq;
    Phj_y = iMKt21q.*phj_xq + iMKt22q.*phj_yq;

    % stiffness matrix of Laplacian
    a = (wq*(Phi_x.*Phj_x + Phi_y.*Phj_y)).*ar;
    A = A + sparse(loc2globV(i,:),loc2globV(j,:),a,dofV,dofV);
  end

end

unk_glob = [nunk,Np+(1:Nt)];

A = A(unk_glob,:);
F = F(unk_glob);
F = F - A(:,ndir1)*ones(length(ndir1),1);
A = A(:,unk_glob);

s = A\F;
Z(ndir1) = 1;
Z(unk_glob) = s;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [cD, cL] = compute_drag_lift(dofV,dofP,Aglob,Fglob,U,V,P,Z,rho,Ubar,D)

W = sparse(2*dofV+dofP,1);
W(1:dofV) = Z;
drag = W'*(Aglob*[U;V;P]-Fglob);
%
W = sparse(2*dofV+dofP,1);
W((1:dofV)+dofV) = Z;
lift = W'*(Aglob*[U;V;P]-Fglob);

cD = 2*drag/(rho*Ubar^2*D);
cL = 2*lift/(rho*Ubar^2*D);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

## Riferimenti bibliografici

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

[2] W. Bangerth and R. Rannacher, *Adaptive Finite Element Methods for Differential Equations*, Lectures in Mathematics, ETH Zürich, Birkhäuser Verlag, Basel, 2003.

[3] R. Becker, *Weighted error estimators for the incompressible Navier-Stokes equations*, Rapport de recherche, 3458, INRIA, 1998.

[4] J. Burkardt
`http://people.sc.fsu.edu/~jburkardt/m_src/dunavant/dunavant_html`, 2006.

[5] D. Dunavant, *High degree efficient symmetrical Gaussian quadrature rules for the triangle*, Internat. J. Numer. Methods Engrg., 21 (1985), pp. 1129–1148.

[6] H. Elman, D. Silvester, and A. Wathen, *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics*, Oxford University Press, New-York, 2005.

[7] A. Ern and J. L. Guermond, *Theory and Practice of Finite Elements*, Text in Applied Mathematica Sciences, 159, Springer-Verlag, New-York, 2004.

[8] M. B. Giles, M. G. Larson, J. M. Levenstam, and E. Süli, *Adaptive error control for finite element approximations of the lift and drag coefficients in viscous flow*, Technical Report, NA-97/06, Oxford University Computing Laboratory, 1997.

[9] M. B. Giles and E. Süli, *Adjoint methods for PDEs: a posteriori error analysis and postprocessing by duality*, Acta Numer., 11 (2002), pp. 145–236.

[10] `www.gnu.org/software/octave`

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

[12] P. I. Kattan, *MATLAB Guide to Finite Elements: An interactive approach*, Springer-Verlag, Berlin, 2008.

[13] Y. W. Kwon, and H.C. Bang, *The Finite Element Method Using MATLAB*, 2nd Edition, CRC Press, 2000.

[14] Partial Differential Equation Toolbox$^{TM}$User's Guide, The MathWorks, Inc., `http://www.mathworks.com/access/helpdesk/help/toolbox/pde/`, 2011.

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

[16] J. Li, and Yi-T. Chen, *Computational Partial Differential Equations using MATLAB*, CRC Press, 2009.

[17] J. L. Lions and E. Magenes, *Non-Homogeneous Boundary Value Problems and Applications*, vol. I. Springer-Verlag, Berlin, 1972.

[18] J. Lyness and D. Jespersen, *Moderate degree symmetric quadrature rules for the triangle*, J. Inst. Math. Appl., 15(1) (1975), pp. 19–32.

[19] A. Quarteroni, *Numerical Models for Differential Problems*, Springer-Verlag Italia, Milan, 2009.

[20] A. Quarteroni and A. Valli, *Domain Decomposition Methods for Partial Differential Equations*, Numerical Mathematics and Scientific Computation, Clarendon Press, Oxford, 1999.

[21] M. Schäefer and S. Turek, *Benchmark computations of laminar flow around a cylinder*, in NNFM 52 "Flow Simulation on High-Performance Computers II", Vieweg, Braunschweig, 1996.

[22] G. von Winckel, *Matlab Central*, `http://www.mathworks.com/matlabcentral/fileexchange/4540-legendre-gauss-quadrature-weights-and-nodes`, 2004.

# MOX Technical Reports, last issues

**Dipartimento di Matematica "F. Brioschi",**
**Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)**

**49/2013** MICHELETTI, S.
*Fast simulations in Matlab for Scientific Computing*

**48/2013** SIMONE PALAMARA, CHRISTIAN VERGARA, ELENA FAGGIANO, FABIO NOBILE
*An effective algorithm for the generation of patient-specific Purkinje networks in computational electrocardiology*

**47/2013** CHKIFA, A.; COHEN, A.; MIGLIORATI, G.; NOBILE, F.; TEMPONE, R.
*Discrete least squares polynomial approximation with random evaluations - application to parametric and stochastic elliptic PDEs*

**46/2013** MARRON, J.S.; RAMSAY, J.O.; SANGALLI, L.M.; SRIVASTAVA, A.
*Statistics of Time Warpings and Phase Variations*

**45/2013** SANGALLI, L.M.; SECCHI, P.; VANTINI, S.
*Analysis of AneuRisk65 data: K-mean Alignment*

**44/2013** SANGALLI, L.M.; SECCHI, P.; VANTINI, S.
*AneuRisk65: a dataset of three-dimensional cerebral vascular geometries*

**43/2013** PATRIARCA, M.; SANGALLI, L.M.; SECCHI, P.; VANTINI, S.
*Analysis of Spike Train Data: an Application of K-mean Alignment*

**42/2013** BERNARDI, M.; SANGALLI, L.M.; SECCHI, P.; VANTINI, S.
*Analysis of Juggling Data: an Application of K-mean Alignment*

**41/2013** BERNARDI, M.; SANGALLI, L.M.; SECCHI, P.; VANTINI, S.
*Analysis of Proteomics data: Block K-mean Alignment*

**40/2013** PACCIARINI P.; ROZZA G.
*Stabilized reduced basis method for parametrized advection-diffusion PDEs*