Fast implementation of mixed RT0 finite elements in MATLAB [*]

Theodore Weinberg
Faculty Sponsor: Bedřich Sousedík

Department of Mathematics and Statistics,
University of Maryland, Baltimore County,
1000 Hilltop Circle, Baltimore, MD 21250, USA

Abstract

We develop a fast implementation of the mixed finite element method for the Darcy's problem discretized by lowest-order Raviart-Thomas finite elements using Matlab. The implementation is based on the so-called vectorized approach applied to the computation of the finite element matrices and assembly of the global finite element matrix. The code supports both 2D and 3D domains, and the finite elements can be triangular, rectangular, tetrahedral or hexahedral. The code can also be easily modified to import user-provided meshes. We comment on our freely available code and present a performance comparison with the standard approach.

1 Introduction

Understanding and modeling flow in porous media is important in many areas including managing groundwater reserves, maintaining CO2 storage facilities, and simulating petroleum reservoirs. This has created a growing need to efficiently describe flow in porous media. Models are typically given by partial differential equations (PDEs) and the basic law at the center of most of the models is Darcy's law, named after Henry Darcy to honor his observations of fountains in Dijon in 1856. We refer, for example, to [3, 7] for a thorough discussion of modeling of porous media. The Darcy's law applied to a single phase flow in porous media will also be the core of our model problem. For the discretization we use the mixed finite element method. This is a type of finite element method which uses a couple of independent variables. In our model problem, we introduce velocity (or equivalently flux) and pressure variables. Specifically, our discrete model is built using the lowest-order Raviart-Thomas elements (RT0). These elements were proposed in [14] and have become quite popular for discretizations of mixed problems. For a concise overview of mixed and hybrid method we refer, for example, to the monographs [5, 10]. They are edge (or face) elements, and they are commonly used in discretizations of flow in porous media among other problems.

In this paper, we describe an efficient, vectorized implementation of the finite element method for the lowest order Raviart-Thomas elements (RT0). For the implementation we chose to use MATLAB, which is a popular environment for prototyping and implementing numerical methods for several reasons. Code written in MATLAB is easy to read, and the environment contains many built-in functions, particularly for matrix operations, that make coding faster and easier. However, it is also known that some standard programming techniques, such as loop structures, are computationally inefficient compared to other programming languages such as C or Fortran. To avoid relying on loops, code in MATLAB can be modified by replacing loops with array operations. This process is called vectorization, and it can significantly improve speed. Using vectorized code enables us to interact with multiple matrices simultaneously, which can allow us to work with much larger sets of data at the same time. While vectorized code is memory intensive, the increase in the speed of the code makes it worthwhile. Our code is written as an extension of the fast MATLAB implementation of the edge/face elements (RT0 and Nedelec elements for div and curl problems, respectively) provided in [1], see also [13]. The code is available at

`https://www.mathworks.com/matlabcentral/fileexchange/68926-fast-implementation-mixed-fem`

The main purpose of this contribution is to provide an efficient finite element implementation of RT0 elements to the community. The code can generate some elementary meshes, and we also provide a function to import meshes

[*]This work was supported by the U.S. National Science Foundation under grant DMS1521563 and by an Undergraduate Research Award from the UMBC Office of Undergraduate Education.

generated by NGSolve [11]. Moreover, the code can be also easily modified to import meshes provided by users. We note that implementations of the lowest-order Raviart-Thomas finite elements for mixed problems in Matlab for triangular meshes are discussed in [2]. We would also like to mention that Matlab Reservoir Simulation Toolbox (MRST) is available for advanced simulations of flow in porous media [12], but it is based predominantly on other discretizations.

The paper is organized as follows. In Section 2 we introduce the model problem, in Section 3 we describe the finite element discretization, in Section 4 we discuss the Matlab implementation, in Section 5 we present results of numerical experiments, and in Section 6 we conclude our work.

2 Model problem

Let Ω be an open, bounded polygonal or polyhedral domain in $\mathbb{R}^d$, $d = 2, 3$. We consider the model problem

$$-\nabla \cdot (\mathbb{k} \nabla p) = f, \qquad \text{in } \Omega, \tag{1}$$

where $\mathbb{k}$ is a symmetric, positive definite coefficient matrix, the right-hand side $f \in L^2(\Omega)$, and the problem is subject to sufficiently smooth boundary conditions on the boundary $\partial\Omega$. We introduce an auxiliary velocity variable

$$\vec{u} = -\mathbb{k} \nabla p,$$

and p will be called pressure. Now we may rewrite (1) as a first-order system, known as the Darcy's problem

$$\mathbb{k}^{-1} \vec{u} + \nabla p = 0, \quad \text{in } \Omega, \tag{2}$$

$$\nabla \cdot \vec{u} = f, \quad \text{in } \Omega, \tag{3}$$

satisfying on $\partial\Omega = \overline{\Gamma}_N \cup \overline{\Gamma}_E$ the boundary conditions

$$\begin{aligned} p &= g_N, \quad \text{on } \Gamma_N, \\ \vec{u} \cdot \vec{n} &= g_E, \quad \text{on } \Gamma_E, \end{aligned}$$

where $\vec{n}$ is the unit outward normal of Ω, and the functions g_N and g_E representing natural and essential boundary data, respectively, are sufficiently smooth. We note that if $\Gamma_N = \emptyset$ an additional compatibility condition is required

$$\int_\Omega f \, dx + \int_{\partial\Omega} g_E \, dx = 0,$$

and in such case the pressure p is unique up to an additive constant.

Let us define spaces

$$\begin{aligned} L^2(\Omega) &= \left\{ q : \int_\Omega q^2 \, dx < \infty \right\}, \\ \mathbf{H}(\Omega; \mathrm{div}) &= \left\{ \vec{v} : \vec{v} \in \left[L^2(\Omega) \right]^d ; \nabla \cdot \vec{v} \in L^2(\Omega) \right\}. \end{aligned}$$

In the mixed variational formulation of (2)–(3) we wish to find $(\vec{u}, p) \in (U_E, Q)$ such that

$$\int_\Omega \mathbb{k}^{-1} \vec{u} \cdot \vec{v} \, dx - \int_\Omega p \nabla \cdot \vec{v} \, dx = 0, \qquad \forall \vec{v} \in U, \tag{4}$$

$$-\int_\Omega \nabla \cdot \vec{u} q \, dx = -\int_\Omega f q \, dx, \quad \forall q \in Q, \tag{5}$$

where the pair of spaces (U, Q) is selected so that $U \subset \mathbf{H}(\Omega; \mathrm{div})$ and $Q \subset L^2(\Omega)$, and U_E is an extension of U containing velocities that satisfy the essential boundary condition, see [4, 5, 7, 10] for more details.

3 Finite element discretization

Let us consider a triangulation $\mathcal{T}_h$ of the domain Ω using triangles or quadrilaterals in 2D and tetrahedrons or hexahedrons in 3D. For simplicity, edges of elements in 2D will be called as faces, and faces in both 2D and 3D will be denoted by f_j. The discrete velocity space is defined as

$$U_h = \left\{ \vec{u} \mid \vec{u}_{|K} \in \mathrm{RT0}(K), \ K \in \mathcal{T}_h \text{ and } \vec{u} \in U \right\},$$

where RT0 is the lowest-order Raviart-Thomas space on the element K. Let the basis (shape) functions for the velocity and pressure spaces be denoted φ_i and ψ_j, respectively. The pressures are approximated by piecewise constant basis functions, and we denote by Q_h the discrete pressure space. For the velocity degrees of freedom we consider the average values of the normal components over the element faces. Specifically, the velocity degrees of freedom are defined as

$$\frac{1}{|f_j|}\int_{f_j} \varphi_i\left(x_j\right)\cdot n_j\, ds = \delta_{ij}, \tag{6}$$

where δ_{ij} is the Kronecker delta, n_j is the unit outward normal of face and x_j is its coordinate. In the calculations of the basis functions below, we use centers of the corresponding faces.

In matrix terminology, the discretization of (4)–(5) can be written as a saddle-point linear system

$$\begin{bmatrix} \mathbf{A} & \mathbf{B}^T \\ \mathbf{B} & 0 \end{bmatrix}\begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{f} \end{bmatrix}, \tag{7}$$

where

$$\begin{aligned}
\mathbf{A} &= \left[a_{ij}\right], & a_{ij} &= \int_\Omega \mathbb{k}^{-1}\,\varphi_i\cdot\varphi_j\,dx,\\
\mathbf{B} &= \left[b_{k\ell}\right], & b_{k\ell} &= -\int_\Omega \nabla\cdot\varphi_k\,\psi_\ell\,dx,\\
\mathbf{f} &= \left[f_\ell\right], & f_\ell &= -\int_\Omega f\psi_\ell\,dx
\end{aligned}$$

In this paper, we present an efficient implementation of problem (4)–(5) in Matlab leading to the setup of system (7).

3.1 The basis functions

We derive basis functions for the RT0 reference finite elements corresponding to quadrilaterals and triangles in 2D, and hexahedrons and tetrahedrons in 3D, see Figure 1 for the plots of reference elements. The basis functions and the spatial variables for the reference elements will be denoted by an additional hat symbol as $\widehat{\varphi}_i$, $\widehat{x}_i$, etc., and symbol $\widehat{f_i}$ will denote face i of a reference element. The transformation from the reference element $\widehat{K}$ to an element K will be denoted by $F_K(\widehat{x}) = B_K\widehat{x} + b_K$ with $B_K \in \mathbb{R}^{d\times d}$ and $b_K \in \mathbb{R}^d$. The mapping is illustrated by Figure 2 for a triangle in 2D. In this case the transformation is

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = B_K\begin{bmatrix} \widehat{x}_1 \\ \widehat{x}_2 \end{bmatrix} + b = \begin{bmatrix} x_1^2 - x_1^1 & x_1^3 - x_1^1 \\ x_2^2 - x_2^1 & x_2^3 - x_2^1 \end{bmatrix}\begin{bmatrix} \widehat{x}_1 \\ \widehat{x}_2 \end{bmatrix} + \begin{bmatrix} x_1^1 \\ x_2^1 \end{bmatrix},$$

and similar transformations can be easily found for the other types of elements.

Quadrilateral The basis functions are defined as

$$\widehat{\varphi}_i\left(\widehat{x}\right) = a + b\widehat{x} = \begin{pmatrix} a_1 + b_1\widehat{x}_1 \\ a_2 + b_2\widehat{x}_2 \end{pmatrix}, \qquad i = 1,2,3,4.$$

Definition (6) is used on the reference square $[-1,1]^2$ to enforce this condition for each basis function. We also note that $|\widehat{f_j}| = 2$ for every face of this element. For example, for the first face

$$\begin{aligned}
\frac{1}{\left|\widehat{f_j}\right|}\int_{\widehat{f_1}} \widehat{\varphi}_i\left(\widehat{x}_1\right)\cdot\widehat{n}_1\,ds &= \frac{1}{2}\int_{\widehat{f_1}} \begin{pmatrix} a_1 + b_1\cdot 1 \\ a_2 + b_2\cdot 0 \end{pmatrix}^T \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} ds \\
&= \frac{1}{2}\left(a_1 + b_1\right)\int_{\widehat{f_1}} ds = \left(a_1 + b_1\right)
\end{aligned}$$

and repeating the calculation for all faces and accounting for the length of each face, we get a system of equations

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}\begin{bmatrix} a_1 \\ a_2 \\ b_1 \\ b_2 \end{bmatrix} = I,$$

which is solved for each basis function with right-hand side given by columns of an identity matrix to find the coefficients of each basis function. The basis functions are

$$\widehat{\varphi}_1 = \begin{pmatrix} \frac{1}{2}\left(1 + \widehat{x}_1\right) \\ 0 \end{pmatrix}, \quad \widehat{\varphi}_2 = \begin{pmatrix} 0 \\ \frac{1}{2}\left(1 + \widehat{x}_2\right) \end{pmatrix}, \quad \widehat{\varphi}_3 = \begin{pmatrix} \frac{1}{2}\left(-1 + \widehat{x}_1\right) \\ 0 \end{pmatrix}, \quad \widehat{\varphi}_4 = \begin{pmatrix} 0 \\ \frac{1}{2}\left(-1 + \widehat{x}_2\right) \end{pmatrix}.$$

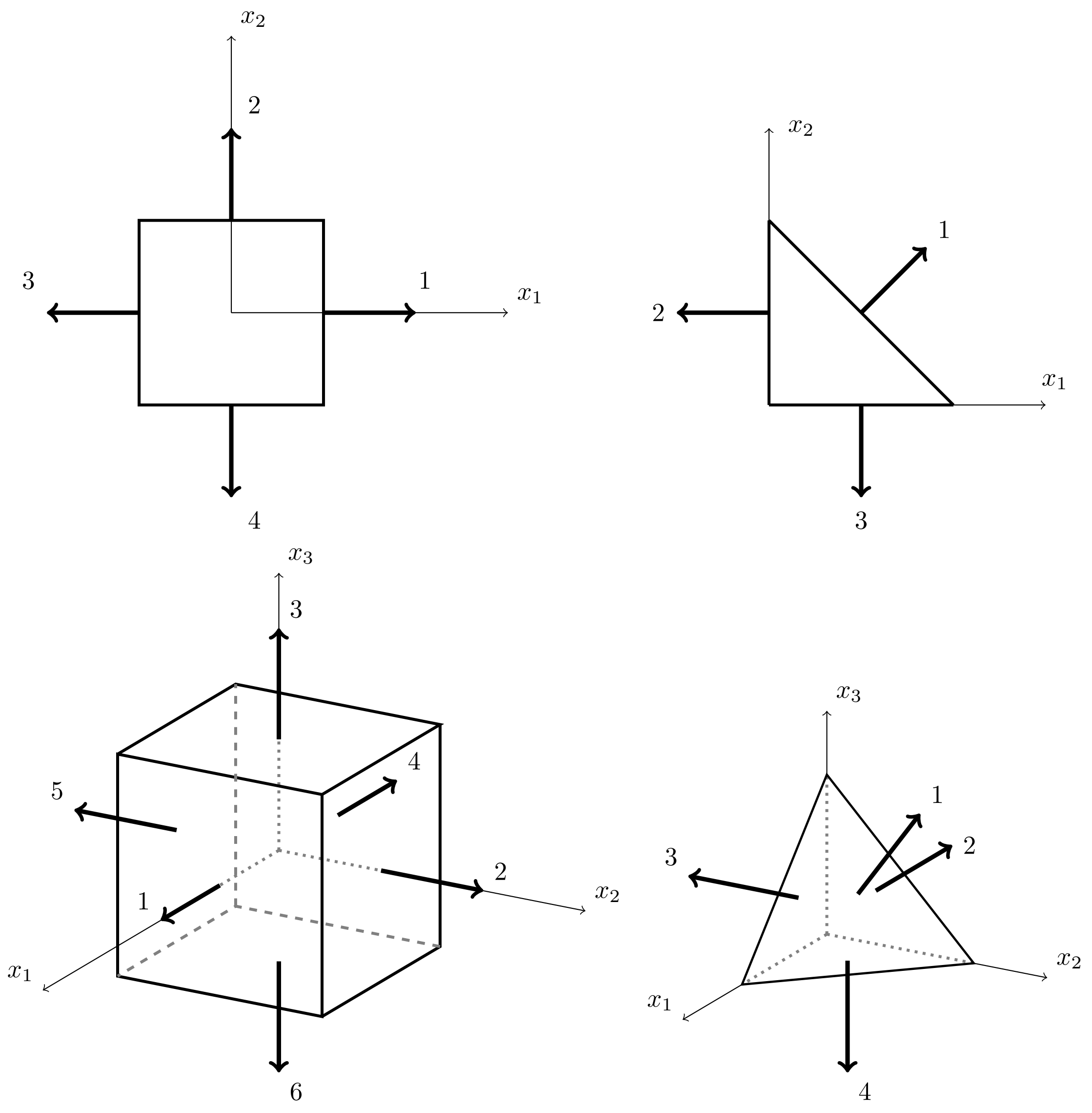

Figure 1: Reference elements.

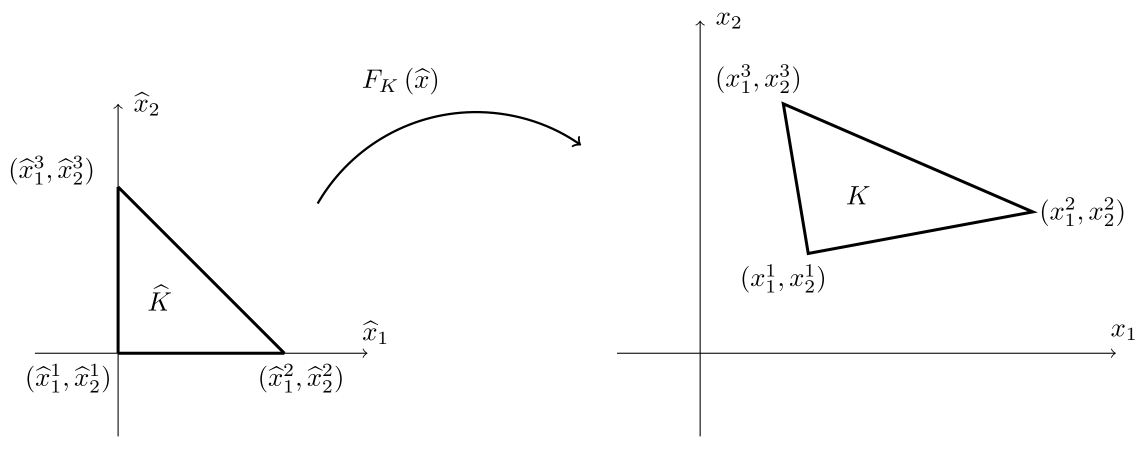

Figure 2: Finite elements $\widehat{K}$, K and a mapping $F_K(\widehat{x})$ between them.

Triangle The basis functions are defined as

$$\widehat{\varphi}_i\left(\widehat{x}\right) = a + b\widehat{x} = \begin{pmatrix} a_1 + b\widehat{x}_1 \\ a_2 + b\widehat{x}_2 \end{pmatrix}, \qquad i = 1, 2, 3.$$

Definition (6) is used as before. For example, for the first face

$$\begin{aligned}\frac{1}{\left|\widehat{f}_1\right|}\int_{\widehat{f}_1} \widehat{\varphi}_i\left(\widehat{x}_1\right)\cdot \widehat{n}_1\, ds &= \frac{1}{\left|\widehat{f}_1\right|}\int_{\widehat{f}_1} \begin{pmatrix} a_1 + b\frac{1}{2} & a_2 + b\frac{1}{2} \end{pmatrix}\cdot \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix} ds \\ &= \frac{1}{\left|\widehat{f}_1\right|\cdot\sqrt{2}}\left(a_1 + a_2 + b\right)\int_{\widehat{f}_1} 1\, ds = \frac{1}{\sqrt{2}}\left(a_1 + a_2 + b\right) = \delta_{i1}.\end{aligned}$$

After repeating this calculation for all faces, we get a system

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} a_1 \\ a_2 \\ b \end{bmatrix} = I,$$

which is solved for the coefficients of basis functions. The basis functions are

$$\widehat{\varphi}_1 = \sqrt{2}\begin{pmatrix} \widehat{x}_1 \\ \widehat{x}_2 \end{pmatrix}, \quad \widehat{\varphi}_2 = \begin{pmatrix} -1 + \widehat{x}_1 \\ \widehat{x}_2 \end{pmatrix}, \quad \widehat{\varphi}_3 = \begin{pmatrix} \widehat{x}_1 \\ -1 + \widehat{x}_2 \end{pmatrix},$$

Hexahedron The basis functions are defined as

$$\widehat{\varphi}_i\left(\widehat{x}\right) = a + b\widehat{x} = \begin{pmatrix} a_1 + b_1\widehat{x}_1 \\ a_2 + b_2\widehat{x}_2 \\ a_3 + b_3\widehat{x}_3 \end{pmatrix}, \qquad i = 1, 2, 3, 4, 5, 6.$$

Definition (6) is used on the reference cube $[-1, 1]^3$ to enforce the conditions for each basis function. For example, for the first face

$$\begin{aligned}\frac{1}{\left|\widehat{f}_1\right|}\int_{\widehat{f}_1} \widehat{\varphi}_i\left(\widehat{x}_1\right)\cdot \widehat{n}_1\, ds &= \frac{1}{\left|\widehat{f}_1\right|}\int_{\widehat{f}_1} \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix}^T \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} ds \\ &= \frac{1}{\left|\widehat{f}_1\right|}\left(a_1 + b_1\right)\int_{\widehat{f}_1} ds = \left(a_1 + b_1\right).\end{aligned}$$

Repeating the calculation for all faces,, we get a system of equations

$$\cdot\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = I,$$

which is solved with right-hand side given by columns of an identity matrix to find the coefficients of each basis function. The basis functions are

$$\widehat{\varphi}_1 = \begin{pmatrix} \frac{1}{2}\left(1 + \widehat{x}_1\right) \\ 0 \\ 0 \end{pmatrix}, \quad \widehat{\varphi}_2 = \begin{pmatrix} 0 \\ \frac{1}{2}\left(1 + \widehat{x}_2\right) \\ 0 \end{pmatrix}, \quad \widehat{\varphi}_3 = \begin{pmatrix} 0 \\ 0 \\ \frac{1}{2}\left(1 + \widehat{x}_3\right) \end{pmatrix},$$

$$\widehat{\varphi}_4 = \begin{pmatrix} \frac{1}{2}\left(-1 + \widehat{x}_1\right) \\ 0 \\ 0 \end{pmatrix}, \quad \widehat{\varphi}_5 = \begin{pmatrix} 0 \\ \frac{1}{2}\left(-1 + \widehat{x}_2\right) \\ 0 \end{pmatrix}, \quad \widehat{\varphi}_6 = \begin{pmatrix} 0 \\ 0 \\ \frac{1}{2}\left(-1 + \widehat{x}_3\right) \end{pmatrix}.$$

Tetrahedron The basis functions are defined as

$$\widehat{\varphi}_i\left(\widehat{x}\right) = a + b\widehat{x} = \begin{pmatrix} a_1 + b\widehat{x}_1 \\ a_2 + b\widehat{x}_2 \\ a_3 + b\widehat{x}_3 \end{pmatrix}, \qquad i = 1, 2, 3, 4.$$

Definition (6) is used on the reference tetrahedron to enforce the conditions for each basis function. For example, for the first face

$$\begin{aligned}\frac{1}{\left|\widehat{f}_1\right|}\int_{\widehat{f}_1} \widehat{\varphi}_i\left(\widehat{x}_1\right)\cdot\widehat{n}_1\,ds &= \int_{f_1} \begin{pmatrix} a_1 + b\frac{1}{3} & a_2 + b\frac{1}{3} & a_3 + b\frac{1}{3} \end{pmatrix}\cdot\frac{1}{\sqrt{3}}\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} ds \\ &= \frac{1}{\left|\widehat{f}_1\right|}\frac{1}{\sqrt{3}}\left(a_1 + a_2 + a_3 + b\right)\int_{\widehat{f}_1} 1\,ds = \frac{1}{\sqrt{3}}\left(a_1 + a_2 + a_3 + b\right) = \delta_{i1}.\end{aligned}$$

The remaining three faces all involve near identical calculations. For example, definition (6) applied to the second face is

$$\begin{aligned}\frac{1}{\left|\widehat{f}_2\right|}\int_{\widehat{f}_2} \widehat{\varphi}_i\left(\widehat{x}_1\right)\cdot\widehat{n}_1\,ds &= \frac{1}{\left|\widehat{f}_2\right|}\int_{\widehat{f}_2} \begin{pmatrix} a_1 & a_2 + b\frac{1}{3} & a_3 + b\frac{1}{3} \end{pmatrix}\cdot\begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} ds \\ &= \frac{1}{\left|\widehat{f}_2\right|}\left(-a_1\right)\int_{\widehat{f}_2} 1\,ds = -a_1 = \delta_{i1}.\end{aligned}$$

After repeating the calculation, we get a system

$$\begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ b \end{bmatrix} = I,$$

and the basis functions are

$$\widehat{\varphi}_1 = \sqrt{3}\begin{pmatrix} \widehat{x}_1 \\ \widehat{x}_2 \\ \widehat{x}_3 \end{pmatrix}, \quad \widehat{\varphi}_2 = \begin{pmatrix} -1 + \widehat{x}_1 \\ \widehat{x}_2 \\ \widehat{x}_3 \end{pmatrix}, \quad \widehat{\varphi}_3 = \begin{pmatrix} \widehat{x}_1 \\ -1 + \widehat{x}_2 \\ \widehat{x}_3 \end{pmatrix}, \quad \widehat{\varphi}_4 = \begin{pmatrix} \widehat{x}_1 \\ \widehat{x}_2 \\ -1 + \widehat{x}_3 \end{pmatrix}.$$

3.2 Finite element matrices

A finite element is defined by a triplet $\{K, \mathcal{P}, \mathcal{N}\}$, where K is the geometric configuration, $\mathcal{P}$ is the finite dimensional space of shape functions on K, and $\mathcal{N}$ is the set of their degrees of freedom [4, 8]. It it well known (see, for example [15]), that the mapping $\mathcal{F}$ defined by $\mathcal{F}(\varphi_i) = \varphi_i \circ F_K^{-1}$ is not an isomorphism from $\mathbf{H}(K; \mathrm{div})$ to $\mathbf{H}(\widehat{K}; \mathrm{div})$, since $\mathcal{F}$ does not preserve continuity of the normal components of the reference basis functions. Instead, the function values and the divergence are mapped using Piola transformation as

$$\varphi\big|_K(x) = \frac{1}{\det B_K} B_K \widehat{\varphi}\left(F_K^{-1}(x)\right), \qquad \mathrm{div}\,\varphi\big|_K(x) = \frac{1}{\det B_K}\widehat{\mathrm{div}}\widehat{\varphi}\left(F_K^{-1}(x)\right).$$

Next, global velocity basis functions are related to more than one element. It is necessary to decide about the orientation of global normals of element faces in order to produce global velocity basis functions whose normal components are continuous at element interfaces. While the orientation can be decided arbitrarily, we utilize the convention from [1] with more details given in the next section.

The global matrix consists of blocks $\mathbf{A}$ and $\mathbf{B}$, which are obtained by assembling contributions from the element matrices consisting of blocks $\mathbf{A}_K = [a_{ij}^K]$ and $\mathbf{B}_K = [b_j^K]$, $i, j = 1, \dots, 3$ or 4 or 6, obtained by evaluating integrals on each element K, where

$$a_{ij}^K = \frac{1}{|\det B_K|}\int_{\widehat{K}} \left(\left[\mathrm{sign}_i^K\right] B_K \widehat{\varphi}_i\left(\widehat{x}\right)\cdot\left[\mathrm{sign}_j^K\right] B_K \widehat{\varphi}_j\left(\widehat{x}\right)\right)_{\Bbbk^K} d\widehat{x}, \tag{8}$$

$$b_j^K = -\frac{1}{|\det B_K|}\int_{\widehat{K}} \left[\mathrm{sign}_j^K\right] \nabla\cdot\widehat{\varphi}_j\left(\widehat{x}\right) d\widehat{x}, \tag{9}$$

where $\left[\text{sign}_i^K\right]$ is $+1$ if the orientation of the (local) normal of face i in element K is in agreement with the orientation of the global normal and -1 otherwise. The permeability coefficients are assumed to be available as a d-dimensional vector for each element $\mathbb{k} = [\mathbb{k}_d^K]$, and specifically in element K they are used as a d-dimensional weight in the inner product. That is, the notation in (8) should be interpreted as $(a \cdot b)_\alpha = (\alpha_1 a_1 b_1 + \cdots + \alpha_d a_d b_d)$.

Finally, we note that in case of structured grids using rectangular and block finite elements, a single finite element matrix may be replicated and multiplied by the corresponding coefficients $\mathbb{k}$. Since our purpose is to allow for unstructured meshes using triangular and tetrahedral elements, we do not use this approach. However, we use only affine mappings for the rectangular and block finite elements, which then limits their use to structured grids.

4 Implementation

We discuss the Matlab implementation. Let us denote the set of finite elements by $\{\mathcal{E}_i\}$, the set of nodes by $\{\mathcal{N}_i\}$ and the set of faces by $\{\mathcal{F}_i\}$, with the symbol # denoting the size of a given set. We generate (or import) a usual nodal mesh, and the code subsequently generates a description of the face-based mesh for the RT0 discretization. The nodal mesh is described by the following fields:

`elems2nodes`	$\#\mathcal{E} \times (3 \text{ or } 4 \text{ or } 8)$	elements and their global nodes (3 or 4 in 2D, and 4 or 8 in 3D)
`nodes2coord`	$\#\mathcal{N} \times d$	global nodes and their coordinates in 2D or 3D

The face based mesh is then described by the following fields:

`elems2faces`	$\#\mathcal{E} \times (3 \text{ or } 4 \text{ or } 6)$	global elements defined by global faces (3 or 4 in 2D, 4 or 6 in 3D)
`faces2nodes`	$\#\mathcal{F} \times (2 \text{ or } 3 \text{ or } 4)$	global faces defined by global nodes (2 in 2D, and 3 or 4 in 3D)

and, additionally the code uses the following fields:

`signs`	$\#\mathcal{F} \times (3 \text{ or } 4 \text{ or } 6)$	$+1$ or -1 for every face of an element, corresponding to `elems2faces`
`coeffs`	$d \times \mathcal{E}$	coefficients for each element in each dimension
`ngdofs`	1	total number of degrees of freedom (velocity and pressure)

The orientation of the global normals of faces is assigned following a convention from [1], which is based on the orientation of a face and, for example, in 2D suggests to assign `signs(i,1)=1` if `elems2nodes(i,2)>elems2nodes(i,3)` and `-1` otherwise.

4.1 Vectorized computation of element matrices

In the non-vectorized code, we iterate in a for-loop over each element and then iterate in another for-loop through the integration points to find sequentially each local element matrix. In the the vectorized code, we instead iterate in a for-loop through the integration points and then iterate in another for-loop to calculate each entry of the local element matrices simultaneously. With the vectorized code, we also take advantage of the symmetry of the matrix and thus only need to solve for the upper right half of the matrix, which we then duplicate into the bottom left half. In both the non-vectorized and vectorized code, we determine the **A** and **B** parts of the local matrices separately. We use Gaussian quadrature rules for the integration of basis functions. Specifically, we use 4 integration points for quadrilateral elements and 8 integration points for block elements. These are found by a simple extension of the one-dimensional case. For the integration of basis functions over triangular and tetrahedral elements, we use the quadrature rules from [9] and [16], respectively. Specifically, we use 3 integration points for triangular elements and 4 integration points for tetrahedral elements.

The core of the non-vectorized version of the 2-dimensional quadrilateral code is as follows:

```
1  for e = 1:nelem % loop over elements
2      A = zeros(nelf);
3      B = zeros(1,nelf);
4      xcoord = nodes2coord(1,elems2nodes(:,e));
5      ycoord = nodes2coord(2,elems2nodes(:,e));
6      for intx = 1:nglx
7          x = point2(intx,1);
8          wtx = weight2(intx,1);
9          for inty = 1:ngly
10             y = point2(inty,2);
11             wty = weight2(inty,2);
12             [sqhape,divsqhape,dhdr,dhds] = feisoquad2D4n_RT0(x,y);
13             J = [ (xcoord(2)-xcoord(1))/2 0;
14                 0 (ycoord(4)-ycoord(1))/2 ]; %Jacobian
15             detJ = det(J);
16             sqhape1 = spdiags(signs(:,e),0,nelf,nelf) * 1/(detJ) * (sqhape*J) * ...
17                 spdiags(coeffs(:,e),0,dim,dim);
18                 sqhape2 = spdiags(signs(:,e),0,nelf,nelf) * 1/(detJ) * (sqhape*J);
19             A = A + (sqhape1*sqhape2')*wtx*wty*detJ;
20             B = B + ( signs(:,e)' .* divsqhape )*wtx*wty*detJ;
21         end
22         Kloc(:,:,e) = [ A B';
23                 B 0 ];
24     end
25 end
```

In line 1, we iterate over each element. On lines 2 and 3 we initialize the **A** and **B** parts of an element matrix. Even in non-vectorized code, pre-allocating our matrices is essential to maintaining reasonable runtimes. On lines 4-11 we prepare the coordinates and weights that will be needed. `Point2` and `weight2` are variables defined earlier in the code that provide the 2D quadrature rules. At line 12, we calculate the basis functions and the divergence. At lines 13-15, we calculate the Jacobian and the determinant. Then in lines 16-18, we determine the contributions to the **A** and **B** part of the local stiffness matrix for a particular integration point. At line 21, we add our new data to the list of local stiffness matrices.

The vectorized code relies on matrix operations to avoid the outermost for-loop iterating through each element:

```
1  Kloc = zeros(nbasis+1,nbasis+1,nelems);
2  for i=1:nip
3      for m=1:nbasis
4          for k=m:nbasis
5              Kloc(m,k,:) = squeeze(Kloc(m,k,:))' + ...
6                  w(i) .* B_K_detA'.^(-1) .* ...
7                  sum( squeeze( astam(signs(:,m), ( amsv(B_K, val(i,:,m)) .* ...
8                  reshape(coeffs,size(coeffs,1),1,size(coeffs,2)) ) ) ).* ...
9                  squeeze( astam(signs(:,k), amsv(B_K, val(i,:,k))) ) ...
10                 );
11         end
12         Kloc(m,nbasis+1,:) = squeeze(Kloc(m,nbasis+1,:)) + ...
13             w(i) .* B_K_detA.^(1) .* ...
14             (signs(:,m) .* dval(i,:,m) );
15     end
16 end
17 Kloc = copy_triu(Kloc);
```

In line 1 we pre-allocate for the set of local matrices Kloc. In lines 2-4, we set up the loop structure described above: we iterate through the integration points and the basis functions for each face. This gives us one entry of the matrix for each pair of faces. However, we only iterate through half of the pairs. This is because, for example, a pair of faces 1 and 4 is the same as a pair of faces 4 and 1, which is why the local matrices are symmetric. In lines 5-14, we calculate entries of the local matrix. The calculations are done simultaneously using matrix operations as the code is vectorized. The code inside the third loop gives the **A** part of the matrix, whereas the part after the loop gives the **B** part. Note that `B_k_detA` is the determinant of the Jacobian, and `B_k` is the determinant itself. In line 17, we call upon a command `copy_triu`, which copies the upper-left triangular part of the matrix into the lower-right triangular part.

For reference, the `astam` and `amsv` functions used in our calculation were created in [1]. Function `astam` is a simple helper function that takes a $m \times 1$ matrix and an $n \times p \times m$ matrix as inputs, and then multiplies each level on the third index by the corresponding elment in the first matrix. In this case, it is being used to multiply each element matrix by its matching sign. For $0 < i \le m$,

$$C(:,:,i) = a(i) * B(:,:,i).$$

Function `amsv` is being used to take weighted sums of each row in each layer of matrices. The second matrix must have the same number of elements as the first has columns, so that any element in that column is given that weight in the summation.

$$C(:,1,i) = A(:,:,i) * B(:).$$

4.2 Vectorized assembly of the matrix

There are several approaches to assemble the global matrix. That is, in the standard approach the contribution of each element is determined using nested for loops. In the standard method, we set up an outer loop to iterate through the elements and two inner loops to iterate through the pairs of edges or faces. It then adds the contribution of the local matrix to the global matrix. This method thus requires three for loops and each additional element requires another set of iterations through the inner two loops. This becomes quite costly as the number of elements increases.

```
K = sparse(ndofs,ndofs);
for k = 1:size(elems2faces,2)
    for j = 1:size(elems2faces,1)
        for i = 1:size(elems2faces,1)
            K(elems2faces(i,k),elems2faces(j,k)) = K(elems2faces(i,k),elems2faces(j,k)) + Kloc(i,j,k);
        end
    end
end
```

We also implemented a vectorized approach for this project. We still iterate over a list of elements, but there are two primary differences. First, we take advantage of sparse matrices. This is essential as with greater number of elements the full matrices can consume singnificant amounts of memory. It also allows the computation to determine all contributions of an element simultaneously, reducing the runtime. It is also important to remove for loops due to their inefficiency in MATLAB. By handling entire elements at once and constructing the sparse global matrix only at the end, the process of assembly becomes significantly faster as will be seen in numerical experiments.

```
temp = cell(length(elist),1);
for e = elist(:)'
    faces = elems2faces(:,e);
    faces = faces(faces>0); % omit 0's
    dofs = faces2dofs(:,faces)';
    dofs = dofs(dofs(:)>0); % stack grouped by face, omit 0's
    [k,l,m] = find(Kloc(:,:,e));
    temp{e} = [dofs(k),dofs(l),m];  % and insert it into the global and get indeces of the entries
end
temp = cell2mat(temp);
K = sparse(temp(:,1),temp(:,2),temp(:,3),ndof,ndof);
```

5 Numerical experiments

We present a comparison in terms of computational time of the standard and vectorized implementations of both computation of local finite element matrices and assembly of the global matrix. For testing we used a computer with two 8-core 2.10 GHz CPUs with 1 TB of memory running Linux openSUSE 42.3 and MATLAB version 9.2.0.538062 (R2017a). The results for a sequence of 2D quadrilateral meshes and 3D hexahedral meshes are summarized in Table 1. From the results it can be seen that for smaller meshes there is no advantage, but the speed up of the vectorized approach is quite significant for larger meshes for both 2D and 3D. Compared to the code implemented in iFEM [6] for 2D triangles, our vectorized code is somewhat slower (roughly one order of magnitude). Both codes scale near linearly, and as the sizes of problems increase our code has a consistently lower scaling factor.

As the number of elements increases the vectorized code becomes significantly faster since the number of iterations in for loops does not depend on the number of elements. Based on the amount of time the non-vectorized code took to run, it was not feasible to run for very large meshes.

Table 1: Timing [s] of the two implementations for quadrilateral meshes in 2D and hexahedral meshes in 3D: nelem is the total number of finite elements, ndof is the total number of degrees of freedom, t_e is the time needed to compute the element matrices, t_a is the time needed to assemble the global stiffness matrix, and $t_e + t_a$ is the total time.

problem setup :			standard			vectorized		
partitioning	$nelem$	$ndof$	t_e	t_a	$t_e + t_a$	t_e	t_a	$t_e + t_a$
2D								
4×4	16	56	0.1614	0.0250	0.1864	0.0879	0.0358	0.1237
8×8	64	208	0.0371	0.0151	0.0522	0.248	0.0168	0.0416
16×16	256	800	0.0816	0.0276	0.1092	0.0193	0.0181	0.0374
32×32	1024	3136	0.2842	0.1117	0.3959	0.0295	0.0435	0.073
64×64	4096	12,416	1.0820	0.9044	1.9864	0.0605	0.1439	0.2044
128×128	16,384	49,408	4.2735	11.344	15.618	0.1559	0.5199	0.6758
256×256	65,536	1.9712×10^5	17.135	197.24	214.375	0.5034	1.9920	2.4950
512×512	2.6214×10^5	7.8745×10^5	66.915	5320.7	5387.625	2.0455	7.9219	9.9674
1024×1024	1.0486×10^6	3.1478×10^6	-	-	-	5.698	34.516	40.214
2048×2048	4.1943×10^6	1.2587×10^7	-	-	-	22.642	140.89	163.532
4096×4096	1.6777×10^7	5.0340×10^7	-	-	-	90.471	566.95	657.421
8192×8192	6.7109×10^7	2.0134×10^8	-	-	-	355.58	2650.0	3005.58
16384×16384	2.6844×10^8	8.0534×10^8	-	-	-	1452.0	10,888.0	12,340.0
3D								
$4 \times 4 \times 4$	64	448	0.0197	0.0348	0.0545	0.0663	0.0387	0.1050
$8 \times 8 \times 8$	512	3584	0.3439	0.0957	0.4396	0.0399	0.0334	0.0733
$16 \times 16 \times 16$	4096	28,672	2.5233	2.4434	4.9667	0.1562	0.1569	0.3131
$32 \times 32 \times 32$	32,678	2.2938×10^5	20.073	141.210	161.283	0.9351	1.1340	2.0691
$64 \times 64 \times 64$	2.6214×10^5	1.835×10^6	157.47	19,114.0	19,271.47	7.6207	8.6154	16.2361
$128 \times 128 \times 128$	2.0972×10^6	1.468×10^7	-	-	-	62.197	71.878	134.075
$256 \times 256 \times 256$	1.6777×10^7	1.174×10^8	-	-	-	498.56	600.31	1098.87
$512 \times 512 \times 512$	1.3422×10^8	9.395×10^8	-	-	-	3898.3	5060.1	8958.4

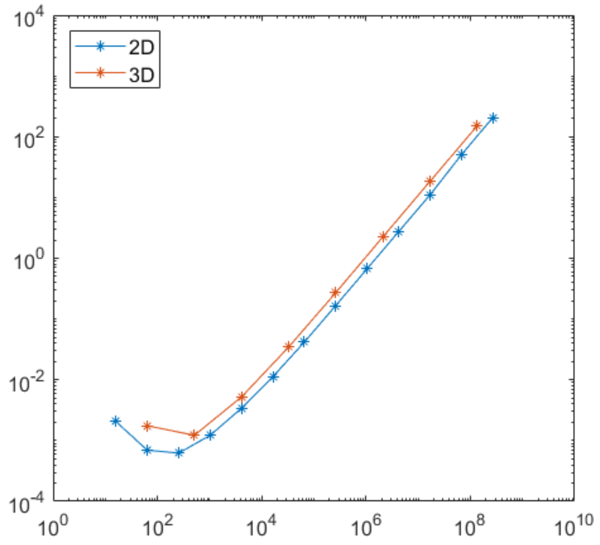

Figure 3: Number of elements versus total time ($t_e + t_a$) in minutes for quadrilateral (2D) and hexahedral (3D) finite elements. In both cases a linear scaling can be observed, based on the slopes of the lines being 1.

6 Conclusion

We presented an efficient implementation of the lowest order Raviart-Thomas finite elements in Matlab. We observed that the vectorized implementation scaled linearly with the number of elements, as can be seen by the slopes of both lines being 1 in Figure 3. The setup of the local element matrices was faster compared to the non-vectorized version by a constant factor, and the assembly for larger meshes was feasible only using the vectorized code. In the largest comparable case for 2D elements, the non-vectorized code took almost 1.5 hours, whereas the vectorized took under 10 seconds. Even just comparing the setup of the local element matrices, it took over a minute for the non-vectorized while only taking under 3 seconds for the vectorized. The results in the 3D case were similar.

References

[1] I. Anjam and J. Valdman. Fast Matlab assembly of FEM matrices in 2D and 3D: Edge elements. *Applied Mathematics and Computation*, 267:252–263, 2015.

[2] C. Bahriawati and C. Carstensen. Three Matlab implementations of the lowest-order Raviart-Thomas MFEM with a posteriori error control. *Computational Methods in Applied Mathematics*, 5(4):333–361, 2005.

[3] Jacob Bear. *Dynamics of Fluids in Porous Media*. Dover, 1988.

[4] Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Springer, New York, NY, third edition, 2008.

[5] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods*. Springer-Verlag, New York – Berlin – Heidelberg, 1991.

[6] Long Chen. Ifem. https://www.math.uci.edu/~chenlong/programming.html. Accessed: 2019-04-22.

[7] Zhangxin Chen, Guanren Huan, and Yuanle Ma. *Computational methods for multiphase flows in porous media*. SIAM, Philadelphia, PA, USA, 2006.

[8] Philippe G. Ciarlet. *The finite element method for elliptic problems*. SIAM, 2002.

[9] 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.

[10] Howard C. Elman, David J. Silvester, and Andrew J. Wathen. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Numerical Mathematics and Scientific Computation. Oxford University Press, New York, second edition, 2014.

[11] Joachim Schöberl et al. Netgen/NGSolve. https://ngsolve.org/, visited June 2017.

[12] K.-A. Lie. An introduction to reservoir simulation using MATLAB: User guide for the Matlab Reservoir Simulation Toolbox (MRST). Technical report, SINTEF ICT, December 2016. (http://www.sintef.no/projectweb/mrst/).

[13] Talal Rahman and Jan Valdman. Fast Matlab assembly of FEM matrices in 2D and 3D: Nodal elements. *Applied Mathematics and Computation*, 219(13):7151–7158, 2013.

[14] P.-A. Raviart and J. M. Thomas. Primal hybrid finite element methods for 2nd order elliptic equations. *Mathematics of Computation*, 31(138):391–413, 1977.

[15] Marie E. Rognes, Robert C. Kirby, and Anders Logg. Efficient assembly of $\boldsymbol{H}(\mathrm{div})$ and $\boldsymbol{H}(\mathrm{curl})$ conforming finite elements. *SIAM Journal on Scientific Computing*, 31(6):4130–4151, 2010.

[16] Linbo Zhang, Tao Cui, and Hui Liu. A set of symmetric quadrature rules on triangles and tetrahedra. *Journal of Computational Mathematics*, 27(1):89–96, 2009.