

On a vectorized basic linear algebra package for prototyping codes in MATLAB

Alexej Moskovka^{1,5}, Talal Rahman², Jan Valdman^{3,5}, and Jon Eivind Vatne⁴

¹Department of Mathematics, Faculty of Applied Sciences,
University of West Bohemia, Technická 8, 30100 Pilsen, Czechia,
`alexmos@kma.zcu.cz`

²Faculty of Engineering and Science, Western Norway University
of Applied Sciences, Inndalsveien 28, 5063 Bergen, Norway,
`talal.rahman@hvl.no`

³Department of Computer Science, Faculty of Science, University
of South Bohemia, Branišovská 31, 37005 Č. Budějovice, Czechia,
`jvaldman@jcu.cz`

⁴Department of Economics, BI Norwegian Business School, Kong
Christian Frederiks plass 5, 5006 Bergen, Norway,
`jon.e.vatne@bi.no`

⁵The Czech Academy of Sciences, Institute of Information Theory
and Automation, Pod Vodárenskou věží 4, 18208 Prague, Czechia.

Abstract

When writing a high-performance code for numerical computation in a scripting language like MATLAB, it is crucial to have the operations in a large for-loop vectorized. If not, the code becomes too slow to be of any use, even for a moderately large problem. However, in the process of vectorizing, it often happens that the code loses its original structure and becomes less readable. This is particularly true in the case of a finite element implementation, even though finite element methods are inherently structured. A basic remedy to this is the separation of the vectorization part from the mathematics part in the code, which is easily achieved through building the code on top of the basic linear algebra subprograms that are already vectorized codes, an idea which has been used in a series of papers over the last fifteen years, developing codes that are fast and still structured and readable. We discuss the vectorized basic linear algebra package, and introduce a formalism using multi-linear algebra to explain and define formally the functions in the package, as well as MATLAB's pagetime functions. We provide examples from computations of vary-

ing complexity, including the computation of normal vectors, volumes, and finite element methods. Benchmarking shows that we also get fast computations. Using the library, we can write codes that closely follow our mathematical thinking, making it easier to write, follow, reuse, and extend the code.

Contents

1	Introduction	2
2	Background from linear algebra	4
2.1	Homomorphism spaces and tensor products	4
2.2	Indexing and page-wise operations	8
2.3	Determinants and inverses	10
2.4	A vectorization library	11
3	Prototype codes and performance comparison	13
3.1	Structures ' coords3D ' and ' vectors3D '	14
3.2	Volumes and normals evaluation	15
3.3	Volume integrals	17
3.4	' GI ' function	19
3.5	Surface integrals	21
3.6	Finite element method	23
3.6.1	Assemblies of stiffness and mass matrices in 2D and 3D .	28
3.6.2	Practical FEM computation	29
3.6.3	Related projects	33

1 Introduction

MATLAB [1] is a popular computing platform with a library of built-in functions and toolboxes provided by Mathworks Inc. (<https://mathworks.com/>), to solve scientific and engineering problems in both academia and industry. It is a scripting language that can be used to write structured and readable or understandable code. However, because the language is interpreted and not compiled, MATLAB codes containing for loops become extremely slow compared to compiled languages such as C, C++, FORTRAN, etc. It is particularly evident in a finite element calculation, since a finite element implementation is heavily based on loops over its nodes, edges, and elements; cf. [2], making any large-scale simulation with finite elements practically useless even on a supercomputer. MATLAB provides functionalities that allow basic arithmetic operations in a loop to be executed in a precompiled fashion, also known as the vectorization or array operation [1]. During the last 15 years, a number of finite element codes have been developed in MATLAB using vectorization to speed up calculations; cf. [7, 9]. Vectorized FE codes show a tremendous improvement in the performance of their time to compute compared to their non-vectorized version. However, in

a straightforward vectorized version, as the mathematics becomes interleaved with the array operations, the code quickly loses its structure and readability, making it hard to use it in a class room or in real applications to further develop or extend. To retain readability and structure, one needs to think in a whole new way, one of which is the separation of vectorization from the mathematics of the problem, an idea which was first introduced for the implementation of finite elements in the numerical simulation of the Electro-Rheological Fluid [12, 13], and documented in [19]. In this idea, the vectorization was done by extending the element-wise operations into matrix-wise or page-wise operations.

In Rahman and Valdman [20], the authors used the idea to further develop an efficient and flexible assembly procedure for the FEM stiffness and mass matrices for nodal elements in 2D and 3D. This resulted in a faster and more scalable algorithm that led to a significant speed-up of the original codes (cf. [2]). The same idea of vectorization was also used in the assembly of edge elements by Anjam and Valdman in the [3] and C^1 elements by Valdman [22]. The ability to simultaneously create FEM matrices for problems formulated in Sobolev spaces H^1 , $H(\text{div})$, $H(\text{curl})$, H^2 resulted in additional MATLAB related computations, including a posteriori functional estimates [3, 5] generalized eigenvalue problems [18] and models in the continuum mechanics of solids [8, 11]. The idea of original vectorization offers more features that were only partially explored. An example is [15] that attempted simple iterative solvers for solutions of the Laplace equation, completely avoiding the setup of stiffness and mass matrices.

The aim is to develop a mathematical framework for the library, the vectorized basic linear algebra package, to be able to formally define the functions in the library, as well understand their constructions so as to be able to write better code. We include some background material from the linear (and multilinear) algebra, where the selection of what to include is based on the library functions, MATLAB's page wise functions, and the applications we have in mind. In particular, tensors play a central role. The library contains a collection of page-wise vectorized versions of functions which are Basic Linear Algebra Subprograms (abbreviated as BLAS). The library enables us to separate the vectorization from the mathematical method or algorithm, e.g. the FE or the geometry algorithm, and hence make the final code readable and reusable. While we were preparing examples for this paper, working with the linear algebra at a higher level (i.e. above the BLAS level), the vectorization being implicit, we can make a clearer and a much more structured code. We provide several examples in detail. From geometry, these include computations of normal vectors and of volume. From FEM, our main motivation, we provide examples for different kinds of elements. In all cases, the code is strongly linked to a linear algebra formulation. In most cases, we have left this implicit, but in some cases, as a guide to the reader, we have made this explicit; see, e.g., Remarks 1 and 3.

All MATLAB codes used in this article can be downloaded at <https://www.mathworks.com/matlabcentral/fileexchange/130824>. We note that the library itself is written in MATLAB, so it is possible to read and modify if one desires.

2 Background from linear algebra

The present goal is to introduce the necessary tools of linear algebra to understand Matlab vectorization. We will make both natural and coordinate-dependent constructions. When we describe the main functions of the library in Section 2.4, we refer to the linear algebra construction underlying the functions. Later, we will give examples of how to see the connection between on the one hand the code using the library, and on the other hand the linear algebra; see, for instance, Remarks 1 and 3. We make this connection explicit only in select cases, though this reasoning permeates all our constructions.

Let $U, V, W, \dots$ be real vector spaces which we usually assume to be finite dimensional. Let $u \in U, v \in V, \dots$ denote arbitrary vectors in the given vector spaces. As a general reference for linear algebra, we suggest [21]. See e.g. [14] for a general exposition of tensor products, their connections with homomorphism spaces and much more. Some of our constructions are usually formulated in a more general setting, as in [14], but we have freely made the simplifications that are possible since we work only with vector spaces.

2.1 Homomorphism spaces and tensor products

The linear space of the transformations from U to V is

$$\text{Hom}(U, V) = \{\text{linear transformations } U \rightarrow V\}. \quad (1)$$

For any vector space U , we have an isomorphism

$$\text{Hom}(\mathbb{R}, U) \simeq U, \phi \mapsto \phi(1). \quad (2)$$

The dual space of U is

$$U^\vee = \text{Hom}(U, \mathbb{R}). \quad (3)$$

There is a natural map

$$U \rightarrow U^{\vee\vee}, u \mapsto \text{eval}_u, \text{eval}_u(\phi) = \phi(u). \quad (4)$$

This map is an isomorphism if and only if U is finite-dimensional.

The adjoint linear transformation to $\phi \in \text{Hom}(U, V)$ is the map $\phi^T \in \text{Hom}(V^\vee, U^\vee)$ given by

$$\phi^T(f)(u) = f(\phi(u)). \quad (5)$$

In words: ϕ^T sends a map f from V to $\mathbb{R}$ to its precomposition with ϕ , thus giving a map from U to $\mathbb{R}$.

When $U = V$ there is a special *identity map* $\text{Id}_U \in \text{Hom}(U, U)$ defined by $\text{Id}_U(u) = u$.

Basis

As we aim towards numerics, it is sensible to express vectors in terms of numbers. This is typically achieved by choosing a basis for each vector space we consider. A choice of basis determines several isomorphisms (again, keep in mind that our vector spaces are finite dimensional). First some notation. If the vectors $f_1, \dots, f_n$ form a basis for the vector space U , we write

$$U = \langle f_1, \dots, f_n \rangle. \quad (6)$$

Let the standard basis for $\mathbb{R}^n$ be $e_1, \dots, e_n$. Then a choice of basis for U determines an isomorphism

$$U \rightarrow \mathbb{R}^n, \sum a_i f_i \mapsto \sum a_i e_i. \quad (7)$$

It also determines a basis for the dual space $U^\vee$, namely

$$U^\vee = \langle f^1, \dots, f^n \rangle \quad \text{where} \quad f^j \left(\sum a_i f_i \right) = a_j, \quad (8)$$

and an isomorphism

$$U \rightarrow U^\vee, \sum a_i f_i \mapsto \sum a_i f^i. \quad (9)$$

Note that if we let U be an infinite dimensional vector space, U and $U^\vee$ are not isomorphic.

The space of transformations from $U = \langle f_1, \dots, f_n \rangle$ to $V = \langle g_1, \dots, g_m \rangle$ is isomorphic with the space of matrices of size $m \times n$. If T is a linear transformation, it is identified with the matrix

$$(T(f_1) | T(f_2) | \dots | T(f_n)). \quad (10)$$

Here the columns are the images of the basis vectors in U expressed in the basis for V . If we express $\sum a_i f_i$ as a column vector (a_i) , the image of this vector under T is given by the matrix product $(T(f_1) | T(f_2) | \dots | T(f_n))(a_i)$. Also, the adjoint from (5) corresponds to matrix transposition.

Tensor products

The tensor product of two vector spaces U and V is

$$U \otimes V = \{\text{linear combinations of } u \otimes v\} / \text{bilinear relations}. \quad (11)$$

We have isomorphisms

$$U \otimes V \simeq V \otimes U, u \otimes v \mapsto v \otimes u \quad (12)$$

and

$$\mathbb{R} \otimes V \simeq V, r \otimes v \mapsto rv. \quad (13)$$

Choosing bases $U = \langle f_1, \dots, f_n \rangle$ and $V = \langle g_1, \dots, g_m \rangle$ determines the basis

$$U \otimes V = \langle f_i \otimes g_j \rangle, i \in \{1, \dots, n\}, j \in \{1, \dots, m\}. \quad (14)$$

As a consequence, $U \otimes V$ is isomorphic to the space of matrices of size $n \times m$, where

$$\sum_{i,j} a_{ij} f_i \otimes g_j \mapsto (a_{ij})_{ij} \quad (15)$$

The map (12) corresponds to matrix transposition, and the map (13) to considering a matrix of size $1 \times m$ as a vector of size m , as in the MATLAB command `squeeze`.

Connections between tensors and homomorphisms

Composition of linear transformations is a bilinear operation, so that there is a natural map

$$\text{Hom}(U, V) \otimes \text{Hom}(V, W) \rightarrow \text{Hom}(U, W), \phi \otimes \psi \mapsto \psi \circ \phi. \quad (16)$$

Given two maps $\phi_i \in \text{Hom}(U_i, V_i)$, $i = 1, 2$, we can form the tensor product map $\phi_1 \otimes \phi_2 \in \text{Hom}(U_1 \otimes U_2, V_1 \otimes V_2)$ by defining

$$(\phi_1 \otimes \phi_2)(u_1 \otimes u_2) = \phi(u_1) \otimes \phi(u_2).$$

A case we will use repeatedly is when one of the two maps is the identity map.

An important connection between tensors and homomorphisms is adjunction:

$$\text{Hom}(U \otimes V, W) \simeq \text{Hom}(U, \text{Hom}(V, W)), \phi \mapsto \psi, \psi(u)(v) = \phi(u \otimes v). \quad (17)$$

Here ϕ is a function of two variables, whereas ψ is a function of one variable, whose value $\psi(u)$ is again a function of one variable. In the special case $W = \mathbb{R}$, combining (3) and (17) gives

$$(U \otimes V)^\vee \simeq \text{Hom}(U, V^\vee). \quad (18)$$

The following map gives an isomorphism (again, remember that we have assumed finite dimension):

$$U \otimes V^\vee \simeq \text{Hom}(V, U), u \otimes \phi \mapsto \psi, \psi(v) = \phi(v)u. \quad (19)$$

In particular, since any finite dimensional vector space is isomorphic to a dual space, this allows any question about tensor products to be formulated using homomorphisms, and vice versa. The matrix representations in (10) and (15) are compatible under these reformulations. In the special case $V = U$, we find that

$$U \otimes U^\vee \simeq \text{Hom}(U, U). \quad (20)$$

The element in $U \otimes U^\vee$ that corresponds to Id_U under this isomorphism is called the *trace element*. Its matrix representation is the identity matrix.

Bilinear forms

A linear map from $U \otimes U$ to $\mathbb{R}$ is called a bilinear form. It is customary to write this as $\langle -, - \rangle \in \text{Hom}(U \otimes U, \mathbb{R})$, especially if it is non-degenerate. It is seldom problematic that this notation looks similar to a vector space with a basis of two vectors. Any bilinear form induces a map $U \rightarrow U^\vee$ under the map we get from (18):

$$(U \otimes U)^\vee \simeq \text{Hom}(U, U^\vee). \quad (21)$$

Concretely, given $\langle -, - \rangle$, the element $u \in U$ is mapped to the function $\langle u, - \rangle \in U^\vee$. This gives an isomorphism $U \simeq U^\vee$ precisely when the bilinear form is non-degenerate. Even though there is much more to be said about this, we will not need anything except simple special cases. In particular, if the bilinear form is the standard inner product on $\mathbb{R}^n$, and the standard basis is used to identify $\mathbb{R}^n$ with its dual space, the isomorphism we get from (21) is the identity map.

Diagonals

For any vector space U , there is a map $\Delta \in \text{Hom}(U, U \otimes U)$ given by $u \mapsto u \otimes u$. The image of this map can be thought of as a diagonal in $U \otimes U$. If we identify $U \otimes U$ with the space of $n \times n$ matrices, the image is exactly the set of diagonal matrices. With a choice of basis $U = \langle f_1, \dots, f_n \rangle$, we can also consider this map

$$\pi \in \text{Hom}(U \otimes U, U), f_i \otimes f_i \mapsto f_i, f_i \otimes f_j \mapsto 0 \text{ if } i \neq j \quad (22)$$

In the matrix representation of a tensor product, π picks out the diagonal as a vector.

Higher tensors

Since tensor products and homomorphism spaces of vector spaces again are vector spaces, everything we have talked about can be iterated to cover more than two factors. For the applications we have in mind, we will usually need three factors, but sometimes more. We will keep in mind the straight-forward generalization, but focus on triple tensor products like

$$\mathcal{A} = U \otimes V \otimes W. \quad (23)$$

By choosing bases for each factor, we can get a basis for the triple (or higher) tensor product by triples (or higher) of basis elements. This allows for representing an element in $\mathcal{A}$ by its three-dimensional array of coefficients. In the special case where one of the spaces is one-dimensional, we can use Equation (13) to remove the one-dimensional space, and thus reduce the dimension of the array. In MATLAB, this is handled by the `squeeze`-command.

In many cases, we will consider operations that are composed from simpler maps only acting on two of the three factors, with the identity map used in

the third factor. We will also freely move between tensor formulations and homomorphism space formulations, and mix these in spaces like $U \otimes \text{Hom}(V, W)$, which is also represented by a three-dimensional array.

2.2 Indexing and page-wise operations

We will consider a number of maps, used in the implementation, in their abstract setting. A common feature in many of these examples is that we want to consider one tensor factor W as a method for indexing, and then perform ordinary matrix operations on the other factors. When using array representations, we can think of this as performing operations on layers of the array. Not much is lost if the reader wishes to think about $W = \mathbb{R}^n$, but we will sometimes need a different interpretation of W .

Copying a vector space

Consider the map $1_n \in \text{Hom}(\mathbb{R}, \mathbb{R}^n)$ that sends 1 to the vector $(1, 1, \dots, 1)^T$. For any vector space U , we get a map:

$$\text{copy} = \text{Id} \otimes 1_n : U \simeq U \otimes \mathbb{R} \rightarrow U \otimes \mathbb{R}^n. \quad (24)$$

We can think of this as giving n copies of an element in U . E.g. if $U \simeq \mathbb{R}^m$ consists of column vectors, each column vector is sent to the matrix all of whose columns are equal to the original vector.

Similarly, we can extract the part corresponding to a given index by tensoring with the map $\mathbb{R}^n \rightarrow \mathbb{R}$ that is given by the dual basis element in $\mathbb{R}^\vee$. For instance, the last part can be extracted by

$$\text{last} = \text{Id} \otimes (0 \ 0 \ \dots \ 0 \ 1) : U \otimes \mathbb{R}^n \rightarrow U \otimes \mathbb{R} \simeq U. \quad (25)$$

Page-wise matrix multiplication

We will consider vector spaces $\mathcal{A}, \mathcal{B}$ where

$$\mathcal{A} = \text{Hom}(V, U) \otimes W_1, \quad \mathcal{B} = \text{Hom}(U, X) \otimes W_2. \quad (26)$$

Using (12) and (16) we get a map

$$\mathcal{A} \otimes \mathcal{B} \rightarrow \text{Hom}(U, X) \otimes W_1 \otimes W_2 \quad (27)$$

If the case where $W = W_1 = W_2$ and have a choice of basis for W , we can then apply the map π defined in 22 to get a map

$$\text{Hom}(V, U) \otimes W \otimes \text{Hom}(U, X) \otimes W \rightarrow \text{Hom}(V, X) \otimes W. \quad (28)$$

We will think about this as a page-wise matrix multiplication: For a fixed basis element of W , the corresponding map is just ordinary matrix multiplication, which is then extended linearly. This interpretation is exactly what happens when we represent $\mathcal{A}$ and $\mathcal{B}$ as three-dimensional arrays in MATLAB and use the command `pagemtimes`.

Page-wise matrix transpose

This has two flavors, one with homomorphisms and one with tensors:

$$\text{Hom}(U, V) \otimes W \rightarrow \text{Hom}(V^\vee, U^\vee) \otimes W \quad (29)$$

$$U \otimes V \otimes W \rightarrow V \otimes U \otimes W \quad (30)$$

By choosing bases and representing these structures by three-dimensional arrays in MATLAB, this is the same as applying the command `pagetranspose`.

Page-wise scalar multiplication

As before, we consider W primarily to be an indexing space with a fixed basis. In a space $U \otimes W$, we want to perform scalar multiplication by some number in each copy of U indexed by a basis element in W . The collection of these scalars can be thought of as a vector in W , so the scalar multiplication is then really the map

$$\text{Id}_U \otimes \pi : U \otimes W \otimes W \rightarrow U \otimes W. \quad (31)$$

Here π is the map from (22). Whether the elements in U are represented as vectors, matrices or higher tensors is immaterial here.

Page-wise bilinear form evaluation

The page-wise version of the map (21) induced from a bilinear form is a map like

$$(U \otimes U)^\vee \otimes W \simeq \text{Hom}(U, U^\vee) \otimes W. \quad (32)$$

Page-wise evaluation of such a bilinear form means that we consider a map with two vector-matrix multiplications, which we can think of as

$$\text{Hom}(\mathbb{R}, U) \otimes \text{Hom}(U, U^\vee) \otimes \text{Hom}(U^\vee, \mathbb{R}) \otimes W \rightarrow \mathbb{R} \otimes W \simeq W.$$

Hadamard or element-wise operations

For any vector space U we can consider the map

$$U \otimes U^\vee \simeq U^\vee \otimes U \simeq \text{Hom}(U, \mathbb{R}) \otimes \text{Hom}(\mathbb{R}, U) \rightarrow \text{Hom}(U, U). \quad (33)$$

If we have chosen a basis for U , we can identify $U^\vee$ and U , and also $\text{Hom}(U, U)$ with $U \otimes U$. We can then compose the above map with π to get a map $U \otimes U \rightarrow U$. If two vectors are expressed in the basis as $a = (a_1, \dots, a_n)^T$ and $b = (b_1, \dots, b_n)^T$, the image of this pair of vectors is expressed as $(a_1 b_1, \dots, a_n b_n)^T$. The reason that we introduce the decomposition above is that it shows naturally the connection with matrix multiplication:

$$a \otimes b \mapsto a \otimes b^T = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \otimes (b_1 \quad \dots \quad b_n) \mapsto \begin{pmatrix} a_1 b_1 & \dots & a_1 b_n \\ \vdots & \ddots & \vdots \\ a_n b_1 & \dots & a_n b_n \end{pmatrix} \quad (34)$$

Finally, the map π extracts the vector of diagonal elements. Again, whether the elements in U are represented as vectors, matrices or higher tensors is immaterial here, and also (after composing with π) the choice of order of a and b is immaterial.

2.3 Determinants and inverses

We could present this theory in the framework of multilinear algebra and alternating forms, but for the applications we have in mind, a more down-to-earth approach will suffice. The cost is, though, that the linearity properties are less obvious.

There are several equivalent ways to define determinants. The quickest is maybe to define, for a linear transformation in $\text{Hom}_{\mathbb{C}}(U, U)$ over $\mathbb{C}$, the determinant as the product of the eigenvalues (with algebraic multiplicities). Then we know that if the space is real, the determinant is a real number, even though it can have complex eigenvalues. We will think of the determinant as a map

$$\det : \text{Hom}(U, U) \rightarrow \mathbb{R}. \quad (35)$$

With a choice of basis, this can be computed as ordinary for matrices, and therefore can also be thought of as a map from e.g. $U \otimes U$ to the real numbers.

Page-wise determinants

If we consider the space $\text{Hom}(U, U) \otimes W$, where W has a chosen basis, we can compute the determinant for each page and think of the determinant as a map

$$\det_W : \text{Hom}(U, U) \otimes W \rightarrow W. \quad (36)$$

Inverses and page-wise inverses

The subset

$$GL_U = \det^{-1}(\mathbb{R} \setminus 0) \subset \text{Hom}(U, U) \quad (37)$$

is a group under composition, the *general linear* group of U . The inverse is the ordinary inverse of a matrix if we have a choice of basis. In the space $\text{Hom}(U, U) \otimes W$, where W has a chosen basis, we can also compute inverses page by page in the subset where the determinant is non-zero in each coordinate. This subset is the inverse image $\det_W^{-1}(W^*)$ of the set $W^* \subset W$ of elements which are non-zero in each coordinate under the map (36). This yields a map (of sets that can be represented as three-dimensional arrays)

$$\text{inverse}_W : \det_W^{-1}(W^*) \rightarrow \det_W^{-1}(W^*). \quad (38)$$

In MATLAB, this map is given by the command `pageinv`.

Transforming normal vectors

Consider a normal vector n to a boundary component of a polyhedral domain, and let v be any vector in (or parallel to) that boundary component, i.e. $n \cdot v = 0$, or as a matrix product, $n^T v = 0$. Let A be an invertible matrix transforming the domain to another domain, so that the vector Av lies in the corresponding boundary component of the new domain. We want to transform n by a matrix X so that Xn is again a normal vector. So we want:

$$\begin{aligned}(Xn) \cdot (Av) &= 0 \iff \\ (Xn)^T Av &= 0 \iff \\ n^T X^T Av &= 0\end{aligned}$$

We see that setting

$$X = (A^{-1})^T \tag{39}$$

works:

$$n^T X^T Av = n^T ((A^{-1})^T)^T Av = n^T A^{-1} Av = n^T v = 0$$

To conclude, the normal vector is multiplied by $(A^{-1})^T$ in order to get the new normal vector. This can of course also be computed page-wise.

2.4 A vectorization library

The vectorized functions of the previous section are implemented in a dedicated library. The multiplication is realized by the following functions:

```
amtam(amx, ama)
avtam(avx, ama)
avtav(ava, avb)
astam(asx, ama)
```

where: '**am**' stands for 'array of matrices', '**av**' for array of vectors, '**as**' for array of scalars and '**t**' for the transpose operator. For instance, the original function '**amtam(amx, ama)**' implements the page-wise of product of an array of matrices '**amx**' and of the page-wise transpose of an array of matrices '**ama**'. It is overridden by a new function '**pagentimes(amx, 'transpose', ama, 'none')**'. For each of these functions, we have a direct link to the underlying linear algebra operations described earlier. When making this connections explicit, we will always have chosen bases for the vector spaces considered, so there is now no reason to distinguish between a space and its dual, and transposes can be omitted.

As an example, consider **astam(asx, ama)**, which implements page-wise scalar multiplication as in (31). We can think of the two inputs as scalars and matrices indexed in the same way. If there are n indices and the matrices have size $k \times m$, this means that in (31) we used $W = \mathbb{R}^n$, $U = \mathbb{R}^k \otimes \mathbb{R}^m$, **astam** is the map

$$\text{Id}_{\mathbb{R}^k \otimes \mathbb{R}^m} \otimes \pi : \mathbb{R}^k \otimes \mathbb{R}^m \otimes \mathbb{R}^n \otimes \mathbb{R}^n \rightarrow \mathbb{R}^k \otimes \mathbb{R}^m \otimes \mathbb{R}^n. \tag{40}$$

Here the first three tensor factors on the left correspond to **ama** and the fourth to **asx**. All the other functions in the library can be considered in a similar manner.

Additionally, multiplications of arrays of vectors or matrices with a single object (matrix of vector) are needed. We have the following functions:

```
amsm(ama,smx)
amsv(ama,svx)
smamt(smx,ama)
svamt(svx,ama)
```

where: '**sm**' stands for a single matrix and '**sv**' for a single vector. This is implemented as creating copies of single objects and utilizing the functions above. All these operations follow from similar linear algebra constructions, i.e. by composing (24), (28) and (30) for suitable spaces. Formally, we also need to permute factors before applying these maps, i.e. by (repeatedly) using (12). Page-wise transpose and page-wise inverse are implemented by functions:

```
aminv(ama)
amt(ama)
```

identical to MATLAB functions **pageinv**, **pagetranspose**. See also (38) and (30), respectively. A page-wise determinant, as in (36), is implemented as

```
amdets(ama)
```

Finally, a page-wise evaluation of a bilinear form, see (2.2), is implemented as

```
avtamav(ava,ama,avb)
```

Historical development

The original MATLAB library was developed in 2003 in [19] and later exploited in finite element assemblies [3, 20]. Some original library functions were optimized due to the implicit (also called arithmetic or broadcasting) expansion feature in R2016b. The additional speedup was achieved due to new page-wise functions in R2020b. For the convenience of the user, we provide three versions of libraries containing the above-described functions:

- The original library '**library_vectorization**'.
- The updated library '**library_vectorization_implicitExpansion**' contains the same functions using the MATLAB implicit expansion introduced in R2016b.
- The newest library '**library_vectorization_pageOperations**' incorporate MATLAB functions '**pagetimes**', '**pagetranspose**' and '**pageinv**' introduced in R2020b.

Application of the latest library to existing codes can shorten evaluation times.

		library		
		original	impexp	page
level	K size	K [s]	K [s]	K [s]
0	64	1.0e-02	1.1e-02	6.5e-03
1	343	3.7e-03	3.3e-03	2.4e-03
2	2197	1.5e-02	1.4e-02	9.8e-03
3	15625	1.4e-01	8.8e-02	7.3e-02
4	117649	1.0e+00	6.5e-01	5.0e-01
5	912673	1.3e+01	1.1e+01	7.3e+00

Table 1: 3D assembly of stiffness matrix K using $P1$ tetrahedral elements. Recomputed from [20].

An example that recomputes Table 1 of [20] is provided in Table 1. The utilization of the newest library provides a speedup factor of around 2. Assembly times were obtained on a MacBook Air (M1 processor, 2020) with 16 GB memory running MATLAB R2023a.

3 Prototype codes and performance comparison

We assume a three-dimensional ($dim = 3$) domain Ω approximated by its triangulation $\mathcal{T}$ into closed tetrahedral elements in the sense of Ciarlet [6]. Elements are geometrically specified by their nodes (or vertices) belonging to the set of nodes $\mathcal{N}$. Nodes are also clustered into elements' edges and faces. Imported meshes are provided by their own functions or created by a Partial Differential Equation Toolbox of MATLAB. As an example, we consider a sequence of tetrahedral meshes corresponding to the discretization of the spherical domain Ω of radius $r = 1$, see Figure 1.

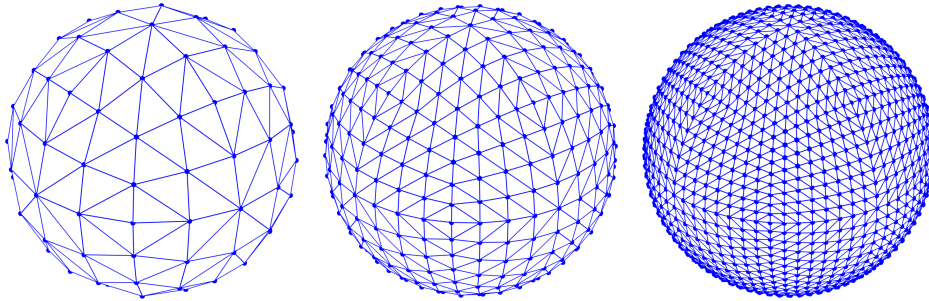


Figure 1: Example of 3D uniformly refined tetrahedral meshes (levels 1, 2, 3) of a sphere domain.

Using the script

```

paramsMesh.level = level;
paramsMesh.r = 1;
[coords,elems] = mesh_sphere(paramsMesh);

```

for a particular nonnegative integer **level**, nodes coordinates and tetrahedral elements matrices '**coords**' and '**elems**' are extracted. Numbers of nodes and elements are obtained by

```

nn = size(coords,1); % number of nodes
ne = size(elems,1);  % number of elements

```

Geometrical properties of spherical meshes are given in the second and third column of Table 2.

3.1 Structures '**coords3D**' and '**vectors3D**'

The first step is an assembly of two 3D matrices. An object

'**coords3D**' of size $dim \times d \times |\mathcal{T}|$

contains the nodes coordinates of every element. Here, dim denotes the space dimension (2 or 3), d is the number of nodes of a single element (3 or 4), and $|\mathcal{T}|$ denotes the number of elements. In case of 3D tetrahedra '**coords3D**' is of size $3 \times 4 \times |\mathcal{T}|$. Alternatively, it can also be used for storing nodes coordinates of boundary faces which are necessary for the evaluation of the surface integral of a vector field over the domain boundary. In this case,

'**coords3D**' of size $3 \times 3 \times |\mathcal{F}_b|$,

where $|\mathcal{F}_b|$ denotes the number of boundary faces. The second object

'**vectors3D**' of size $dim \times (d - 1) \times |\mathcal{T}|$

stores for every element three vectors pointing from the last local node to the first three local nodes. Similarly to '**coords3D**', it can also be used for storing vectors of boundary faces which implies this matrix would be of size $3 \times 2 \times |\mathcal{F}_b|$. Both objects are generated by the function

```

1 function [coords3D,vectors3D] = create_coords3D(coords,elems)
2 dim = size(coords,2); % spatial dimension
3 d = size(elems,2);
4 ne = size(elems,1); % number of elements
5 coords3D = zeros(dim,d,ne); % 3D matrix of coordinates
6 for j = 1:d
7     coords3D(:,j,:) = coords(elems(:,j),:);
8 end
9 if nargin==2
10     vectors3D = coords3D(:,1:end-1,:) - coords3D(:,end,:);
11 end

```

Code 1: The structures '**coords3D**' and '**vectors3D**'.

Remark 1. The assembly construction of '**coords3D**' from the matrices '**elems**' and '**coords**' does not really have a linear algebra interpretation, as the matrix '**elems**' consists of indices. On the other hand, the operation producing '**vectors3D**' from '**coords3D**' can be cast in a linear algebra formulation. First extract the last coordinate by using the map last from (25):

$$\text{Id} \otimes \text{last} \otimes \text{Id} : \mathbb{R}^{dim} \otimes \mathbb{R}^d \otimes \mathbb{R}^{|\mathcal{T}|} \rightarrow \mathbb{R}^{dim} \otimes \mathbb{R} \otimes \mathbb{R}^{|\mathcal{T}|} \quad (41)$$

Then make copies of this using the map copy from (24):

$$\text{Id} \otimes \text{copy} \otimes \text{Id} : \mathbb{R}^{dim} \otimes \mathbb{R} \otimes \mathbb{R}^{|\mathcal{T}|} \rightarrow \mathbb{R}^{dim} \otimes \mathbb{R}^d \otimes \mathbb{R}^{|\mathcal{T}|} \quad (42)$$

If we now compute

$$(\text{Id} - \text{copy} \circ \text{last})(\text{coords3D}), \quad (43)$$

we will get an element in $\mathbb{R}^{dim} \otimes \mathbb{R}^d \otimes \mathbb{R}^{|\mathcal{T}|}$ where the last index in the second factor consists only of zeroes (i.e. last coordinate minus last coordinate). Finally, we extract everything but this last coordinate by the map (for $d = 4$)

$$\text{Id} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \otimes \text{Id} \quad (44)$$

to retain only the interesting part as an element '**vectors3D**' in $\mathbb{R}^{dim} \otimes \mathbb{R}^{d-1} \otimes \mathbb{R}^{|\mathcal{T}|}$.

3.2 Volumes and normals evaluation

A reference tetrahedron is defined by four nodes

$$N_1^0 = (0, 0, 0), \quad N_2^0 = (1, 0, 0), \quad N_3^0 = (0, 1, 0), \quad N_4^0 = (0, 0, 1)$$

and shown in the left part of Figure 2 It has four faces with outer normals that can be stored as columns of a matrix

$$\mathbf{normalsRef} = \begin{pmatrix} -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 \end{pmatrix}.$$

Example 1. For a tetrahedron with nodes

$$N_1 = \frac{(7, 3, -1)}{4}, \quad N_2 = \frac{(7, -2, 4)}{4}, \quad N_3 = \frac{(10, 3, 4)}{4}, \quad N_4 = \frac{(4, 3, 4)}{4},$$

shown in the right part of Figure 2, the structure '**vectors3D**' is represented by a matrix (for a single tetrahedron)

$$\frac{1}{4} \begin{pmatrix} 3 & 3 & 6 \\ 0 & -5 & 0 \\ -5 & 0 & 0 \end{pmatrix}. \quad (45)$$

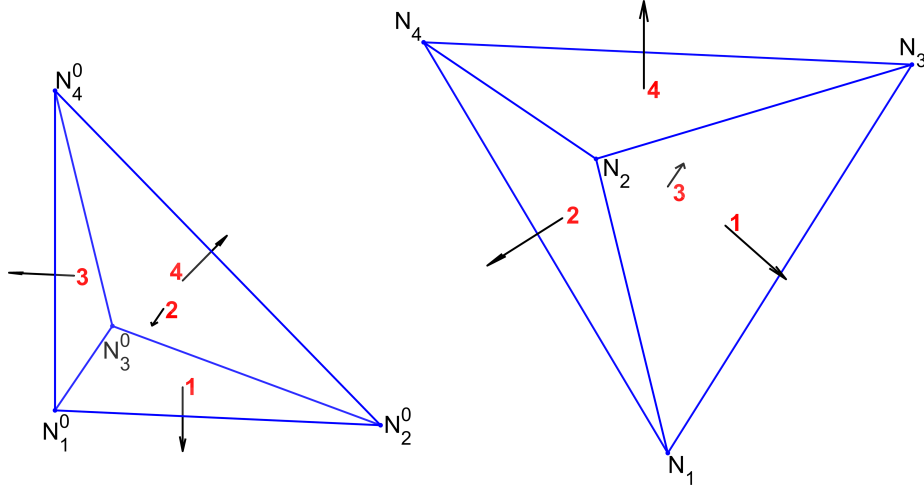


Figure 2: Normals of the reference tetrahedron (left) and normals of a single regular tetrahedron (right).

The determinant of the matrix (45) divided by 6 is equal to $-25/64$ and the absolute value of this number corresponds to the tetrahedron volume. To compute normals of the tetrahedron (cf. (39)), the formula

$$\mathbf{normals3D} = (\mathbf{vectors3D})^{-T} \cdot \mathbf{normalsRef} \quad (46)$$

is applied and yields

$$\mathbf{normals3D} = \begin{pmatrix} 0 & 0 & -2/3 & 2/3 \\ 0 & 4/5 & -2/5 & -2/5 \\ 4/5 & 0 & -2/5 & -2/5 \end{pmatrix}.$$

It should be noted that columns of **normals3D** are not normalized.

Given a general tetrahedral mesh and its corresponding structure '**vectors3D**', it is possible to obtain volumes of all tetrahedra at once by a simple script

```
dim=3; % space dimension
meass = amdet(vectors3D)/factorial(dim); % all volumes
meas = norm(meass,1); % sum of volumes
```

and all normals by

```
vectors3D_inv = aminv(vectors3D); % all inverses
normals3D = amsm(amt(vectors3D_inv), normalsRef); % all normals
```

As an example, Table 2 provides times to evaluate the volume of a unit sphere. We observe a quadratic convergence with respect to the mesh size to the exact volume $\frac{4}{3}\pi \approx 4.188790$. Note that the last column "time to evaluate" also includes the times for the mesh generation. The table can be generated by the script

`benchmark1_volumes_sphere`

Consequently, the script

`benchmark2_normals`

evaluates normals for all faces of the same sphere domain for different levels of mesh refinement. Table 3 provides the corresponding computational times.

mesh level	number of elements	number of nodes	volume	error	time to evaluate
1	384	125	3.932819	2.56e-01	1.31e-02
2	3072	729	4.123099	6.57e-02	3.12e-03
3	24576	4913	4.172259	1.65e-02	7.22e-03
4	196608	35937	4.184651	4.14e-03	3.94e-02
5	1572864	274625	4.187755	1.04e-03	3.32e-01
6	12582912	2146689	4.188531	2.59e-04	4.42e+00

Table 2: Volume evaluation of a sphere domain.

mesh level	number of elements	number of nodes	number of all faces	number of boundary faces	time to evaluate all faces
1	384	125	864	192	1.76e-02
2	3072	729	6528	768	2.99e-03
3	24576	4913	50688	3072	7.29e-03
4	196608	35937	399360	12288	7.67e-02
5	1572864	274625	3170304	49152	7.35e-01
6	12582912	2146689	25264128	196608	1.09e+01

Table 3: Normals evaluation of a sphere domain.

Finally, the script

`example_normals`

generates a 3D mesh of the pyramid, sphere, and torus together with outer normals, whose pictures are shown in Figure 3. There is an option to compute either normals to all faces (including internal element faces) or only normals corresponding to boundary faces.

3.3 Volume integrals

Provided a domain Ω and a mass density function $\rho : \Omega \rightarrow \mathbb{R}$ it is practical to compute an integral

$$m = \int_{\Omega} \rho(\mathbf{x}) \, d\mathbf{x}$$

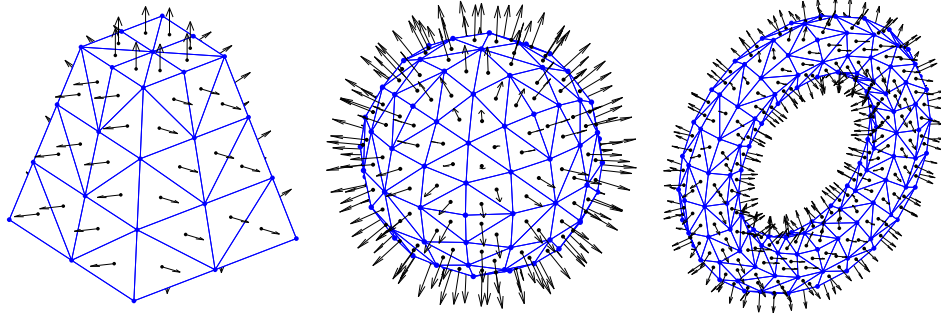


Figure 3: Examples of 3D meshes and underlying outer normals.

denoting the total mass of Ω . Given the matrices '**coords**' and '**elems**' for a specific triangulation of Ω and the mass density function, one can evaluate the total mass of the body using the code below

```

1 gqo = 3; % a quadrature order
2 [ip,w] = Gauss_points(gqo,dim); % barycentric coordinates of
   quadr. points and weights
3 coords3D = create_coords3D(coords,elems);
4 X_ip = amsm(coords3D,ip); % 3 x nip x ne
5 rho = @(X) X(1, :, :).^2 + X(2, :, :).^2; % mass density
6 mass = GI(rho(X_ip),w,volumes); % the total mass

```

Code 2: Gauss quadrature for a volume integral.

The procedure consists of the following steps:

- (line 1) the choice of (Gauss) quadrature order which in general depends on the mass density function;
- (line 2) a setup of quadrature points and their weights on a reference element;
- (line 3) a construction of a 3D matrix '**coords3D**' introduced in Sec. 3.1;
- (line 4) a construction of a 3D matrix '**X_ip**' of size $dim \times n_{ip} \times |\mathcal{T}|$ which for the k -th element stores the coordinates of all Gauss nodes (n_{ip} denotes their number);
- (line 5) definition of a mass density function $\rho(\mathbf{x}) = x_1^2 + x_2^2$;
- (line 6) Gauss integration performed by the function '**GI**' explained further in Sec. 3.4.

Remark 2. Given the code above, one can also easily evaluate the first and the second moment of the area given by

$$M_i = \int_{\Omega} x_i \rho(\mathbf{x}) \, d\mathbf{x}, \quad M_{ij} = \int_{\Omega} x_i x_j \rho(\mathbf{x}) \, d\mathbf{x}. \quad (47)$$

It can be done by modifying the last line above (e.g.) with one of the lines below:

```
M1 = GI(rho(X_ip).*X_ip(1, :, :), w, volumes);
M11 = GI(rho(X_ip).*X_ip(1, :, :).^2, w, volumes);
M12 = GI(rho(X_ip).*X_ip(1, :, :).*X_ip(2, :, :), w, volumes);
```

The moments are used for the computation of the mass center coordinates

$$x_1^c = M_1/m, \quad x_2^c = M_2/m, \quad x_3^c = M_3/m$$

or the moments of inertia to express the rotational energy of a domain. A benchmark

benchmark3_volume_integral

performs several evaluations of the moment of inertia around the x-axis

$$I_1 = \int_{\Omega} \rho(\mathbf{x})(x_2^2 + x_3^2) d\mathbf{x} = M_{22} + M_{33} \quad (48)$$

for a torus domain shown in Fig. 4. Assuming a torus domain given by

$$\begin{aligned} x_1 &= (R + r \cos v) \cos u, \\ x_2 &= r \sin v, \\ x_3 &= (R + r \cos v) \sin u \end{aligned}$$

for parameters $u, v \in (0, 2\pi)$, $r \in (0, 1/4)$ and $R = 1$ and the mass density $\rho(x, y, z) = x^2 + y^2$ one can evaluate

$$I_1 = \frac{2645}{131072} \pi^2 \approx 0.199166.$$

Table (4) provides evaluation times for different levels of mesh refinement using the third order of the Gauss quadrature.

3.4 'GI' function

The key tool of the code above is the 'GI' function which evaluates an integral of a function or a vector field using Gauss quadrature. It is given by the code below

```
1 function value = GI(fip, w, sizes, normals)
2 f_elems = reshape(amsv(fip, w), size(fip, 1), size(fip, 3));
3 if nargin==4
4     f_elems = sum(f_elems.*normals, 2); % scalar product
5 end
6 value = sum(sizes.*f_elems); % the integral value
```

Code 3: The 'GI' function.

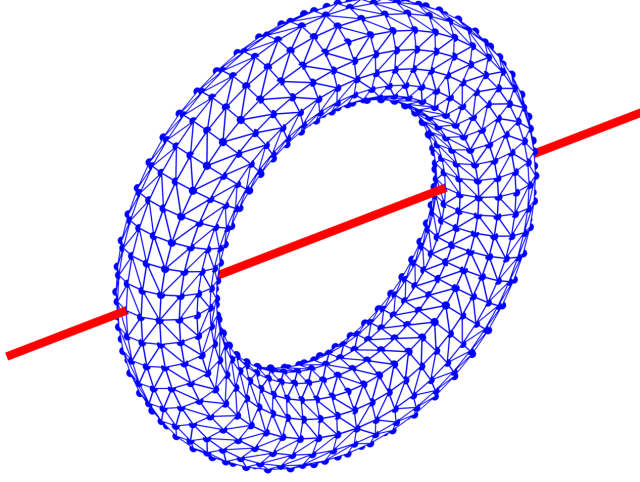


Figure 4: A torus domain rotating around x-axis.

mesh level	number of elements	number of nodes	value of I_1	error	time [s]
0	96	64	0.108588	9.06e-02	6.11e-03
1	576	216	0.169520	2.96e-02	1.33e-03
2	4992	1300	0.191582	7.58e-03	2.68e-03
3	38400	8100	0.197217	1.95e-03	1.29e-02
4	307200	57800	0.198677	4.89e-04	5.23e-02
5	2482176	439956	0.199044	1.22e-04	6.03e-01
6	19759104	3396900	0.199136	3.05e-05	1.31e+01

Table 4: Evaluation of I_1 for a torus domain using volume integration.

mesh level	number of bnd. faces	number of bnd. nodes	value of I_1	error	time [s]
0	128	64	0.108565	9.06e-02	1.23e-02
1	384	192	0.169514	2.97e-02	1.07e-02
2	1664	832	0.191581	7.58e-03	6.53e-03
3	6400	3200	0.197217	1.95e-03	6.79e-03
4	25600	12800	0.198677	4.89e-04	2.10e-02
5	103424	51712	0.199044	1.22e-04	6.17e-02
6	411648	205824	0.199136	3.05e-05	2.68e-01

Table 5: Evaluation of I_1 for a torus domain using a surface integration.

and has the following inputs:

- '**fip**' is a 3D matrix of size $d \times n_{ip} \times |\mathcal{T}|$ storing for every element the values of $f(\mathbf{x})$ in all Gauss points;
- '**w**' is a vector of Gaussian quadrature weights;
- '**sizes**' is a vector of elements' sizes (lengths for 1D, areas for 2D, volumes for 3D);
- '**normals**' is a matrix of size $|\mathcal{T}| \times dim$ storing outer normals of all elements. It must be provided for the integration of a vector field over a hyperplane.

The body of this function consists of the following steps:

- (line 2) evaluating a matrix '**f_elems**' of size $|\mathcal{T}| \times d$ which for every element and every component of $f(\mathbf{x})$ stores its averaged value over the Gauss points with respect to the corresponding weights;
- (line 4) if $f(\mathbf{x})$ is a vector field, the scalar products with the corresponding normals are calculated and summed over the components of $f(\mathbf{x})$. In this case the relation $d = dim$ holds;
- (line 6) the final value of the integral is given by the scalar product of '**f_elems**' and '**sizes**'. At this stage, '**f_elems**' is always of size $ne \times 1$.

The main advantage of this function lies in the ability to integrate both scalar and vector functions in any spatial dimension.

3.5 Surface integrals

Given a domain Ω with a piecewise smooth boundary $\partial\Omega$ and a vector field

$$F(\mathbf{x}) = (F_1(\mathbf{x}), F_2(\mathbf{x}), F_3(\mathbf{x})), \quad \mathbf{x} \in \partial\Omega,$$

one can use boundary normals of Section 3.2 to evaluate

$$\int_{\partial\Omega} F(\mathbf{x}) \cdot \vec{n} \, dS. \quad (49)$$

Since the divergence theorem reforms (49) by

$$\int_{\partial\Omega} F(\mathbf{x}) \cdot \vec{n} \, dS = \int_{\Omega} \nabla \cdot F(\mathbf{x}) \, d\mathbf{x} \quad (50)$$

it is possible to recompute all volume integrals of Sec. 3.3 by surface integrals. For a vector field (see Fig. 5)

$$F(\mathbf{x}) = \left(\frac{x_1^3(x_2^2 + x_3^2)}{3}, \frac{x_2^5}{5}, \frac{x_2^2 x_3^3}{3} \right) \quad (51)$$

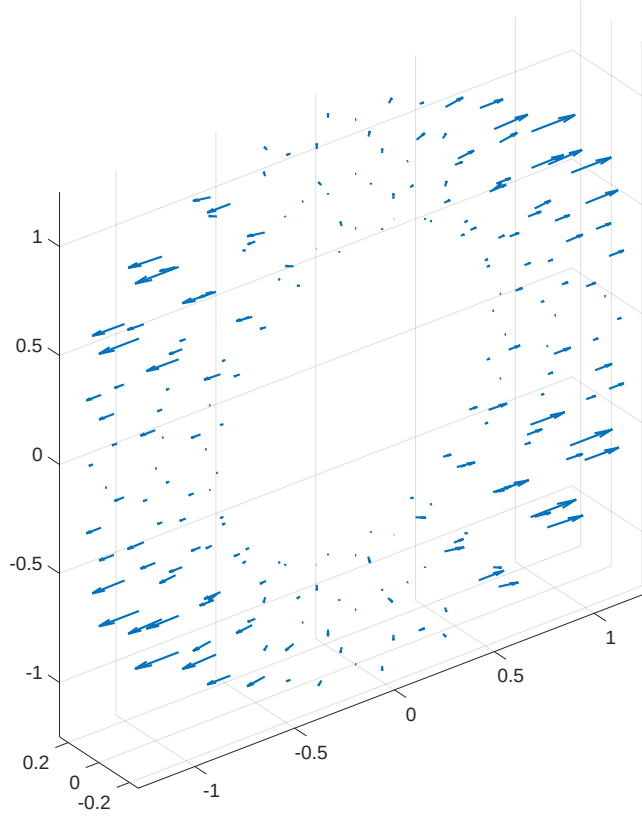


Figure 5: Vector field (51) depicted on a torus domain.

one can easily show

$$\nabla \cdot F(\mathbf{x}) = (x_1^2 + x_2^2)(x_2^2 + x_3^2) = \rho(\mathbf{x})(x_2^2 + x_3^2).$$

Therefore the integral (49) for $F(\mathbf{x})$ from (51) corresponds to the moment of inertia I_1 of (48). The code below evaluates (49):

```

1 gqo = 3; % a quadrature order
2 [ip,w] = Gauss_points(gqo,dim-1); % barycentric coordinates of
   quadr. points and weights
3 coords3D = create_coords3D(coords,facesB);
4 X_ip = amsm(coords3D,ip);
5 F=@(X) [X(1,:).^2; X(2,:).^2; X(3,:).^2]; % vector field
6 value = GI(F(X_ip),w,areasB,normals);

```

Code 4: Gauss quadrature for a surface integral.

Here, boundary faces '**facesB**' together with their areas '**areasB**' are used. The evaluation of I_1 over a surface of a torus is done in benchmark

benchmark4_surface_integral

Evaluation times are provided in Tab. 5. Clearly, the numerical values of I_1 as well as the corresponding errors are the same as in Tab. 4. However, evaluation of the surface integral is significantly faster than the corresponding volume integral.

3.6 Finite element method

The finite element method (FEM) is a numerical tool for the solution of partial differential equations [6]. We can easily apply our vectorization concepts to the class of iso-parametric elements. A reference element in local coordinates $\boldsymbol{\xi} = (\xi_1, \xi_2, \xi_3)$ is transformed to a real finite element in global coordinates $\mathbf{x} = (x_1, x_2, x_3)$ with use of shape functions $\{\varphi_i\}$. Then mapping between the global and the reference coordinate systems can be given by the following relation called the iso-parametric property,

$$x_1 = \sum_i \varphi_i(\boldsymbol{\xi}) x_{1,i}, \quad x_2 = \sum_i \varphi_i(\boldsymbol{\xi}) x_{2,i}, \quad x_3 = \sum_i \varphi_i(\boldsymbol{\xi}) x_{3,i}.$$

Here, $\mathbf{x}_i = (x_{1,i}, x_{2,i}, x_{3,i})$ is the i -th point on a general element in the global coordinate system corresponding to the shape function φ_i . The formula above is defined in three-dimensional space only but can be easily reduced to two-dimensional space as well. Consequently, the same codes are going to work both in two- and three-space dimensions unless stated otherwise.

The shape functions are defined on reference elements. We consider linear (known as P1) and quadratic (known as P2) shape functions $\{\varphi_i\}$ defined on a reference triangle or a reference tetrahedron. The reference triangle is given by three vertices $\boldsymbol{\xi}^1 = (0, 0)$, $\boldsymbol{\xi}^2 = (1, 0)$, $\boldsymbol{\xi}^3 = (0, 1)$. For the construction of quadratic shape functions, additional vertices $\boldsymbol{\xi}^4, \dots, \boldsymbol{\xi}^6$ located at the midpoints of the edges are needed. The reference tetrahedron is given by four vertices $\boldsymbol{\xi}^1 = (0, 0, 0)$, $\boldsymbol{\xi}^2 = (1, 0, 0)$, $\boldsymbol{\xi}^3 = (0, 1, 0)$, $\boldsymbol{\xi}^4 = (0, 0, 1)$ with additional vertices $\boldsymbol{\xi}^5, \dots, \boldsymbol{\xi}^{10}$ for the construction of quadratic shape functions. All shape functions $\{\varphi_i\}$ satisfy the pointwise equality $\varphi_i(\boldsymbol{\xi}^j) = \delta_i^j$, where δ denotes the Kronecker symbol. A function

```
shape = shapefun (point, etype)
```

evaluates value of all shape functions in any number of points '**points**' in the reference coordinates system. Similarly, a function

```
dshape = shapeder (point, etype)
```

evaluates values of their derivatives. An option '**etype**' specifies whether linear '**etype='P1'**' or quadratic '**etype='P2'**' shape functions are considered. Derivatives with respect to the global coordinates are easily calculated from the derivatives with respect to the reference coordinates using the Jacobian matrix. It is performed in the following function:

```

1 function [dphi, detj, jac] = phider(coords3D, ip, etype)
2
3 [dim, nlb, ne] = size(coords3D);
4 nip = size(ip, 2); % number of integration points
5
6 dshape = shaper(ip, etype); % local derivatives
7
8 dphi = zeros(dim, nlb, nip, ne); % all derivatives
9 detj = zeros(nip, ne); % all Jacobians
10 jac = zeros(dim, dim, nip, ne); % all Jacobi matrices
11
12 for i = 1:nip
13     tjac = smant(dshape(:, :, i), coords3D); % dim x dim x ne
14     [tjacinv, tjacdet] = aminv(tjac); % inverses and dets
15     dphi(:, :, i, :) = amsm(tjacinv, dshape(:, :, i));
16     detj(i, :) = abs(tjacdet);
17     jac(:, :, i, :) = tjac;
18 end

```

Code 5: The function **'phider'** evaluating derivatives of shape functions on all elements. Works both in 2D/3D.

The input is provided as the array of matrices **'coords3D'**. Additionally, the integration points' values **'ip'** are specified by positions in the local coordinate system and **'etype'** the type of element considered. The output objects are:

- **'dphi'** is a 4D array containing derivatives of all shape functions in all integration points in all elements;
- **'detj'** is a matrix storing Jacobians in all integration points on every element;
- **'jac'** is a 4D array containing Jacobi matrices in all integration points in all elements.

Remark 3. The loop in Code 5 is only over the relatively low number of integration points, and is therefore not so costly. We could in principle, however, also dispose of this loop using the linear algebra setup. Here is a brief sketch of how to do this. Start from **'dshape'**, considered as an element in $\mathbb{R}^{dim} \otimes \mathbb{R}^{nlb} \otimes \mathbb{R}^{nip}$, and **'coords3D'**, considered as an element in $\mathbb{R}^{dim} \otimes \mathbb{R}^{nlb} \otimes \mathbb{R}^{ne}$. By permuting tensor factors (the map from (30) used repeatedly) and applying page-wise matrix multiplication (the map from (27) with $W_1 = \mathbb{R}^{nip}$, $W_2 = \mathbb{R}^{ne}$), we get the page-wise construction of **'jac'**. To compute **'jacinv'** and **'detj'**, we use the page-wise inverse (38) and determinant (36) with $W = \mathbb{R}^{nip} \otimes \mathbb{R}^{ne}$. Finally, **'dphi'** is computed from **'dshape'** and **'jacinv'** by permuting tensor factors, performing page-wise matrix multiplication, and using the diagonal map (22), also in a page-wise manner. The resulting **'dphi'** is then an element in $\mathbb{R}^{dim} \otimes \mathbb{R}^{nlb} \otimes \mathbb{R}^{nip} \otimes \mathbb{R}^{ne}$.

Since the loops in the coming code listings are of a similar nature, we will not make the linear algebra connection explicit.

We explain how to assemble bilinear forms in the discretization of second-order elliptic problems. Then we typically need to construct a stiffness matrix K and a mass matrix M defined as

$$K_{ij} = \int_{\Omega} c_K(\mathbf{x}) \nabla \Phi_i \cdot \nabla \Phi_j \, d\mathbf{x}, \quad (52)$$

$$M_{ij} = \int_{\Omega} c_M(\mathbf{x}) \Phi_i \Phi_j \, d\mathbf{x}, \quad (53)$$

where ∇ denotes the gradient operator and scalar coefficient functions $c_K, c_M : \Omega \rightarrow \mathbb{R}$. This extends the functionality of [20], where no coefficients (eg. $c_K = c_M = 1$) were considered. The vectorized implementation is shown below:

```

1  function [K,K3D] = stiffness_matrixP1 (elems , coords , coeffs_fun )
2
3  dim = size (coords , 2); % problem dimension
4  ne = size (elems , 1); % number of elements
5  nn = size (coords , 1); % number of nodes
6
7  gqo = 2; [ip , w , nip] = intquad (gqo , dim);
8  coeffs = coeffs_in_ip (coords , elems , coeffs_fun , gqo); % nip x ne
9
10 coords3D = create_coords3D (coords , elems); % dim x nlb x ne
11 [dphi , detj] = phider (coords3D , ip , 'P1 ');
12
13 nlb = dim + 1; % number of local basic functions
14
15 K3D = zeros (nlb , nlb , ne);
16 for i=1:nip % loop over all IPs
17     dphi3D = squeeze (dphi (: , : , i , :)); % in one IP
18     gradTgrad_i = amtam (dphi3D , dphi3D); % bilinear form
19     % without coefficient
20     integrand_i = astam (coeffs (i , :), gradTgrad_i); % bilinear
21     % form with coefficient
22     K3D = K3D + w(i)*astam (detj (i , :), integrand_i);
23 end
24
25 Y3D = reshape (repmat (elems , 1 , nlb) , nlb , nlb , ne);
26 X3D = amt (Y3D);
27 K = sparse (X3D (:), Y3D (:), K3D (:), nn , nn);

```

Code 6: Scalar stiffness matrix for P1 elements.

Code 6 works for both triangular and tetrahedral elements. The order of integration 'gqo' can be specified in line 7. The corresponding integration (Gauss) points 'ip' and weights 'w' are then evaluated automatically for a reference element. If a coefficient function c_K provided by the handle 'coeffs_fun' is

globally constant, element-wise constant, or element-wise linear, it is enough to choose '**gqo=1**'. Then only one integration point is enough to ensure the exact integration of the stiffness matrix. In other cases, '**gqo**' should be chosen as 2 or higher. The only loop of the code starting at line 16 runs over the number of integration points '**nip**'. It sums up the local matrices evaluated at each integration point. The local matrices are assembled in all elements at once and stored in a 3D array '**K3D**' which can be extracted as the second function output. A straightforward modification of the code provides extensions to P2 elements and to mass matrices.

```

1  function [M,M3D] = mass_matrixP2(elems , coords , coeffs_fun )
2
3  dim = size(coords,2); % problem dimension
4  ne = size(elems,1);   % number of elements
5  nn = size(coords,1); % number of all nodes
6  nnP1 = max(max(elems(:,1:dim+1))); % number of nodes excluding
    midedges
7
8  gqo = 4; [ip,w,nip] = intquad(gqo,dim);
9  coeffs = coeffs_in_ip(coords(1:nnP1,:),elems(:,1:dim+1),
    coeffs_fun ,gqo); % nip x ne
10
11 sizes = sizes_of_elements(coords(1:nnP1,:),elems(:,1:dim+1));
12 detj_abs = sizes*factorial(dim); % nip x ne
13
14 nlb = (dim+1) + nchoosek(dim+1,2); % number of local basic
    functions
15
16 phiRef = shapefun(ip','P2');
17 M3D = zeros(nlb,nlb,ne);
18 for i=1:nip % loop over all IPs
19     phi2D = phiRef(:,i)*phiRef(:,i)'; % in one IP
20     phiTphi_i = repmat(phi2D,1,1,ne); % bilinear form without
        coefficient
21     integrand_i = astam(coeffs(i,:),phiTphi_i); % bilinear
        form with coefficient
22     M3D = M3D + w(i)*astam(detj_abs,integrand_i);
23 end
24
25 Y3D = reshape(repmat(elems,1,nlb)',nlb,nlb,ne);
26 X3D = amt(Y3D);
27 M = sparse(X3D(:),Y3D(:),M3D(:),nn,nn);

```

Code 7: Scalar mass matrix for P2 elements.

Assembly of mass matrix for P2 elements in Code 7 has similar structure as the Code 6. It works for both triangular and tetrahedral elements as well. In this case the minimum '**gqo=2**' is required even for a globally constant, element-wise constant, or element-wise linear coefficient function c_K . In other cases, '**gqo**' should be chosen as 4 or higher. The only loop starting at line 18 running

over the number of integration points has the same structure as in Code 6. Here bilinear forms are given by the multiplication of basis functions instead of their gradients. Therefore, '**phi2D**' in line 19 is two-dimensional, while '**dphi3D**' in line 17 of 6 is three-dimensional. Moreover, jacobians denoted by '**detj**' in line 22 are constant on every element, and therefore are one-dimensional. Altogether, we have two pairs of functions:

```
[K, K3D] = stiffness_matrixP1(elems,coords,coeffs_fun);
[M, M3D] = mass_matrixP1(elems,coords,coeffs_fun);

[K, K3D] = stiffness_matrixP2(elems,coords,coeffs_fun);
[M, M3D] = mass_matrixP2(elems,coords,coeffs_fun);
```

The performance of the above assemblies is further discussed for triangulations of a 2D unit square domain and a 3D unit cube domain shown in Figure 6.

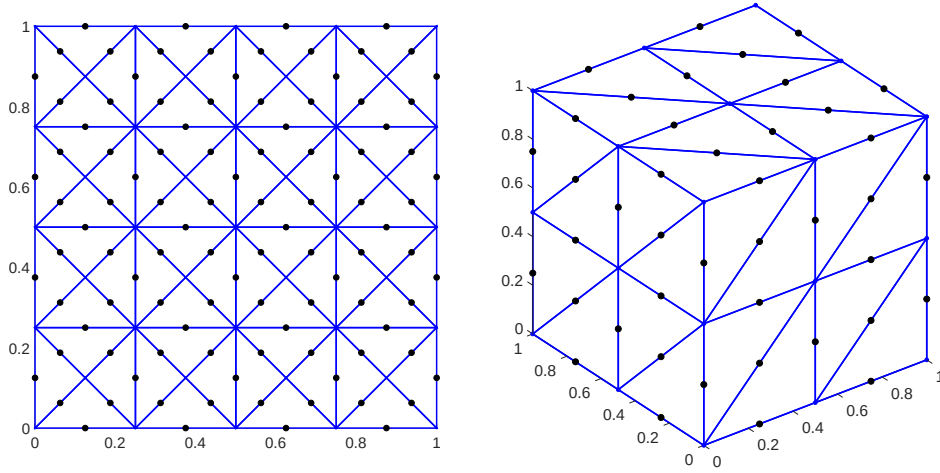


Figure 6: Domains and their triangulations. The edges midpoints (black color) are important in the construction of P2 shape functions.

Additionally, we evaluate quadratic forms

$$I_K := \int_{\Omega} c_K(\mathbf{x})(\nabla v(\mathbf{x}) \cdot \nabla v(\mathbf{x})) d\mathbf{x}, \quad I_M := \int_{\Omega} c_M(\mathbf{x})(v(\mathbf{x}))^2 d\mathbf{x} \quad (54)$$

for given coefficient functions $c_K(\mathbf{x}), c_M(\mathbf{x})$ and a given testing function $v(\mathbf{x})$. The above values are approximated by discretized quadratic forms

$$I_K \approx \tilde{v}^T K \tilde{v}, \quad I_M \approx \tilde{v}^T M \tilde{v}, \quad (55)$$

where $\tilde{v}$ is a vector of coefficients representing $v(\mathbf{x})$ in P1 or P2 basis. The error of such approximation is measured by absolute errors

$$e_K = |\tilde{v}^T K \tilde{v} - I_K|, \quad e_M = |\tilde{v}^T M \tilde{v} - I_M|. \quad (56)$$

level	K, M size	e_K	e_M	K [s]	M [s]
7	33025	7.30e-04	7.05e-05	8.64e-02	3.96e-02
8	131585	1.83e-04	1.76e-05	2.48e-01	1.20e-01
9	525313	4.56e-05	4.41e-06	9.53e-01	5.25e-01
10	2099201	1.14e-05	1.10e-06	4.82e+00	2.78e+00
11	8392705	2.85e-06	2.75e-07	2.98e+01	1.46e+01

Table 6: Assembly times of P1 stiffness and mass matrices in 2D.

level	K, M size	e_K	e_M	K [s]	M [s]
6	33025	5.87e-08	8.49e-09	1.04e-01	3.17e-02
7	131585	3.67e-09	5.31e-10	2.46e-01	1.19e-01
8	525313	2.12e-10	3.32e-11	1.10e+00	5.16e-01
9	2099201	8.10e-11	2.07e-12	4.94e+00	2.95e+00
10	8392705	1.07e-10	1.28e-13	2.96e+01	1.28e+01

Table 7: Assembly times of P2 stiffness and mass matrices in 2D.

level	K, M size	e_K	e_M	K [s]	M [s]
3	729	2.70e-01	5.04e-02	3.22e-02	1.49e-02
4	4913	6.82e-02	1.31e-02	7.76e-02	1.92e-02
5	35937	1.71e-02	3.30e-03	3.72e-01	1.18e-01
6	274625	4.28e-03	8.28e-04	3.99e+00	1.88e+00
7	2146689	1.07e-03	2.07e-04	6.66e+01	2.04e+01

Table 8: Assembly times of P1 stiffness and mass matrices in 3D.

level	K, M size	e_K	e_M	K [s]	M [s]
2	729	6.84e-03	5.75e-03	6.69e-02	1.84e-02
3	4913	4.00e-04	3.88e-04	5.86e-02	2.18e-02
4	35937	2.46e-05	2.48e-05	3.64e-01	1.04e-01
5	274625	1.53e-06	1.55e-06	4.91e+00	2.36e+00
6	2146689	9.53e-08	9.73e-08	8.76e+01	1.64e+01

Table 9: Assembly times of P2 stiffness and mass matrices in 3D.

3.6.1 Assemblies of stiffness and mass matrices in 2D and 3D

The script

```
benchmark5_assembly_2D
```

performs assemblies of K and M for nested uniform mesh refinements of the unit square domain and computes approximate values of I_K , I_M and their absolute errors e_K, e_M . The results are shown in Tables 6 and 7. As a benchmark, we take $c_K(\mathbf{x}) = c_M(\mathbf{x}) = e^{(x_1+x_2)}$, $v(\mathbf{x}) = \sin(x_1)\sin(x_2)$ corresponding to the

exact values

$$I_K = \frac{4\pi^4 (e-1)^2 (1+2\pi^2)}{(1+4\pi^2)^2} \approx 14.5610739535,$$

$$I_M = \frac{4\pi^4 (e-1)^2}{(1+4\pi^2)^2} \approx 0.7021036382.$$

The errors e_K, e_M displayed in the last two columns decrease quadratically with respect to the mesh size h for P1 elements and with the 4th order for P2 elements.

The script

`benchmark6_assembly_3D`

performs tests for the unit cube domain. Here we consider $c_K(\mathbf{x}) = c_M(\mathbf{x}) = e^{(x_1+x_2+x_3)}$, $v(\mathbf{x}) = \cos(x_1) \cos(x_2) \cos(x_3)$ and can compute that

$$I_K = \frac{6\pi^4 (e-1)^3 (1+2\pi^2)^2}{(1+4\pi^2)^3} \approx 19.2286024907,$$

$$I_M = \frac{(e-1)^3 (1+2\pi^2)^3}{(1+4\pi^2)^3} \approx 0.6823216700.$$

Tables 8 and 9 show assembly times together with the corresponding errors.

3.6.2 Practical FEM computation

We focus on solving the full diffusion-reaction boundary value problem

$$\begin{aligned} -\nabla \cdot (c_K(\mathbf{x}) \nabla u(\mathbf{x})) + c_M(\mathbf{x}) u(\mathbf{x}) &= f(\mathbf{x}) && \text{in } \Omega, \\ u(\mathbf{x}) &= u_D(\mathbf{x}) && \text{on } \Gamma_D \subset \partial\Omega, \\ \frac{\partial u}{\partial n}(\mathbf{x}) &= 0 && \text{on } \Gamma_N \subset \partial\Omega. \end{aligned} \quad (57)$$

We consider an L-shaped domain Ω given by the union of rectangles

$$(0, 0.25) \times (0, 0.25), \quad (0, 0.25) \times (0.25, 1), \quad (0.25, 1) \times (0, 0.25)$$

and shown with its triangulation on the left part of Figure 7. A non-homogeneous Dirichlet boundary condition is assumed on the upper edge Γ_D (red nodes) and a zero Neumann boundary condition on the remaining part of the domain boundary $\Gamma_N = \partial\Omega \setminus \Gamma_D$. One can show that the function

$$u(\mathbf{x}) = \cos(4\pi x_1) \cos(4\pi x_2) \quad (58)$$

is the solution of (57) for coefficient functions $c_K(\mathbf{x}) = 1 + x_1^2 - x_2$, $c_M(\mathbf{x}) = 1 - x_1 + x_2^2$, the right-hand side

$$\begin{aligned} f(\mathbf{x}) &= 8\pi x_1 \sin(4\pi x_1) \cos(4\pi x_2) + \cos(4\pi x_1) (-4\pi \sin(4\pi x_2) + \\ &\quad + (1 - x_1 + 32\pi^2(1 + x_1^2 - x_2) + x_2^2) \cos(4\pi x_2)) \end{aligned} \quad (59)$$

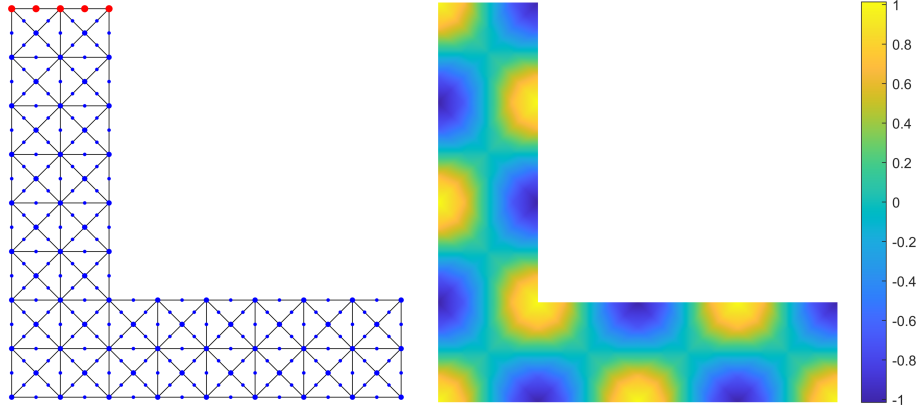


Figure 7: The computational mesh for P2 elements (left) with $|\mathcal{T}| = 112$ and $|\mathcal{N}| = 257$ (including mid-edge points) and the solution of (57) (right).

and the Dirichlet boundary condition $u_D(\mathbf{x}) = \cos(4\pi x_1)$, $x_1 \in [0, 0.25]$, $x_2 = 1$.

The script

`benchmark7_BVP_2D`

evaluates numerical solutions $u_h(\mathbf{x})$ (represented by vectors $\tilde{u}$ of coefficients in a finite element basis) of the boundary value problem (57) for different mesh refinements using P1 or P2 finite elements. An example of a P2 solution is shown on the right part of Figure 7. Practically, a linear system of equations

$$(K + M) \tilde{u} = b \quad (60)$$

is assembled in which entries of the right-hand size vector b are given by

$$b_i = \int_{\Omega} f(\mathbf{x}) \Phi_i(\mathbf{x}) \, d\mathbf{x}, \quad i \in \{1, \dots, |\mathcal{N}|\}. \quad (61)$$

Then it is solved for free entries of $\tilde{u}$, i.e., to those not corresponding to the Dirichlet boundary conditions.

Evaluation of the right-hand side vector

One can evaluate (61) by $b = M_0 \tilde{f}$, where M_0 is the mass matrix corresponding to $c_M(\mathbf{x}) = 1$ and $\tilde{f}$ is a vector of coefficients representing the approximation of $f(\mathbf{x})$ in a finite element basis. However, this approach is computationally too expensive as long as an additional global mass matrix has to be assembled. Instead, we calculate scalar products of $f(\mathbf{x})$ and local basis functions (n_{lb} denotes their number)

$$\int_{T_k} f(\mathbf{x}) \varphi_j(\mathbf{x}), \quad k \in \{1, \dots, |\mathcal{T}|\}, \quad j \in \{1, \dots, n_{lb}\}, \quad (62)$$

on each element. Consequently, any b_i from (61) is given as a sum of particular contributions from (62).

```

1 function [b,b2D] = rhs_vectorP1 (elems , coords , f_fun )
2
3 dim = size(coords,2); % problem dimension
4 ne = size(elems,1); % number of elements
5
6 gqo = 2; [ip,w,nip] = intquad(gqo,dim);
7 coeffs = coeffs_in_ip (coords , elems , f_fun , gqo); % nip x ne
8
9 sizes = sizes_of_elements (coords , elems);
10 detj_abs = sizes*factorial(dim); % nip x ne
11
12 nlb = size(elems,2);
13
14 phiRef = shapefun(ip , 'P1 ');
15 b2D = zeros(nlb,ne);
16 for i=1:nip % loop over all IPs
17     phi1D = phiRef(:,i); % in one IP
18     integrand_i = coeffs(i,:) .* phi1D;
19     b2D = b2D + w(i)*detj_abs' .* integrand_i;
20 end
21
22 elems = elems';
23 b = accumarray(elems(:),b2D(:));

```

Code 8: The right-hand side vector b .

The code 8 evaluates the right-hand side vector b from (61) for P1 elements. The only loop starting at line 16 running over the number of integration points evaluates the scalar products (62) that are stored in a matrix '**b2D**'.

Evaluation of (local) energies

An alternative to solve the boundary value problem (57) is to minimize a quadratic energy functional (see [16] for details related to efficient minimization of nonlinear functionals)

$$J(v) = \int_{\Omega} \left(\frac{1}{2} c_K(\mathbf{x}) \|\nabla v(\mathbf{x})\|^2 + \frac{1}{2} c_M(\mathbf{x}) v(\mathbf{x})^2 - f(\mathbf{x}) v(\mathbf{x}) \right) d\mathbf{x} \quad (63)$$

among all testing functions $v(\mathbf{x})$ satisfying the Dirichlet boundary condition $v(\mathbf{x}) = u_D(\mathbf{x})$ on $\Gamma_D \subset \partial\Omega$. The minimal value of the energy $J(v)$ is achieved for $v(\mathbf{x}) = u(\mathbf{x})$, where the exact solution $u(\mathbf{x})$ is given by (58) and reads

$$J(u) = J_1(u) + J_2(u) + J_3(u) \approx -14.90302171.$$

Its gradient, reactive and linear parts are

$$\begin{aligned} J_1(u) &= \frac{1}{2} \int_{\Omega} c_K(\mathbf{x}) \|\nabla u(\mathbf{x})\|^2 d\mathbf{x} = \frac{289 \pi^2}{192} \approx 14.85581079, \\ J_2(u) &= \frac{1}{2} \int_{\Omega} c_M(\mathbf{x}) u(\mathbf{x})^2 d\mathbf{x} = \frac{578 \pi^2 + 21}{12228 \pi^2} \approx 0.04721091674, \\ J_3(u) &= - \int_{\Omega} f(\mathbf{x}) v(\mathbf{x}) d\mathbf{x} = - \frac{578 (32 \pi^4 + \pi^2) + 21}{6144 \pi^2} \approx -29.80604342. \end{aligned}$$

Using the global matrices K, M and the global vector b , values of all three parts are approximated by

$$J_1(u) \approx \frac{1}{2} \tilde{u}^T K \tilde{u}, \quad J_2(u) \approx \frac{1}{2} \tilde{u}^T M \tilde{u}, \quad J_3(u) \approx -b^T \tilde{u}. \quad (64)$$

For any element T_k , $k \in \{1, \dots, |\mathcal{T}|\}$, we can also define the local (element-wise) contributions to the energy parts above by formulas

$$\begin{aligned} J_{1,k}(u) &= \frac{1}{2} \int_{T_k} c_K(\mathbf{x}) \|\nabla u(\mathbf{x})\|^2 d\mathbf{x}, \\ J_{2,k}(u) &= \frac{1}{2} \int_{T_k} c_M(\mathbf{x}) u(\mathbf{x})^2 d\mathbf{x}, \\ J_{3,k}(u) &= - \int_{T_k} f(\mathbf{x}) u(\mathbf{x}) d\mathbf{x} \end{aligned}$$

and it holds

$$J_i(u) = \sum_{k=1}^{|\mathcal{T}|} J_{i,k}(u), \quad i = 1, 2, 3.$$

Since our implementation also provides local (element-wise) contributions of $K3D$, $M3D$, $b2D$, it is possible (similarly to (64)) to evaluate the approximations of $J_{1,k}$, $J_{2,k}$, $J_{3,k}$ for all elements T_k , $k \in \{1, \dots, |\mathcal{T}|\}$, at once by

$$J_{1,k}(u) \approx \frac{1}{2} \tilde{u}_k^T K_k \tilde{u}_k, \quad J_{2,k}(u) \approx \frac{1}{2} \tilde{u}_k^T M_k \tilde{u}_k, \quad J_{3,k}(u) \approx -b_k^T \tilde{u}_k. \quad (65)$$

Here, K_k and M_k are the local stiffness and mass matrices on the k -th element, respectively. Similarly, b_k and $\tilde{u}_k$ are restrictions of b and $\tilde{u}$ on the k -th element, respectively. Fig. 8 depicts the approximations of energy components (65) for level 4 computational mesh.

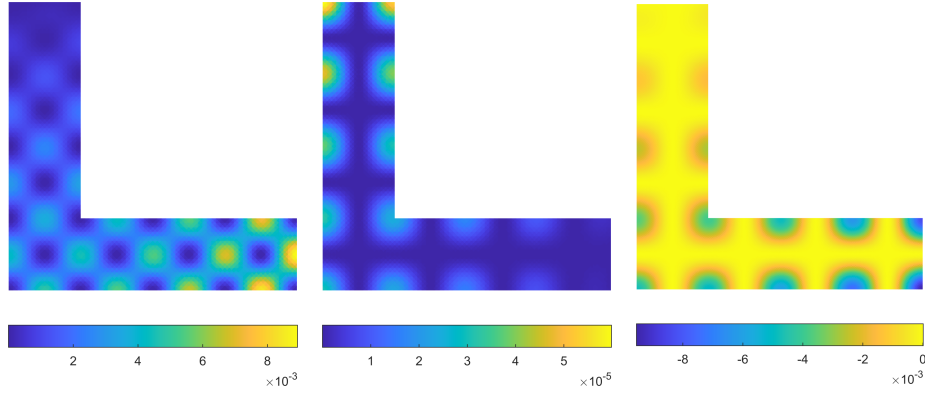


Figure 8: Approximations of the contributions of J_1 , J_2 and J_3 corresponding to the solution (58) of the problem (57) with P2 elements and the computational mesh with 7168 elements.

3.6.3 Related projects

The linear and quadratic functions described above are the simplest of the so-called Lagrange-type shape functions and are the main focus of our paper. Other available implementations that use the vectorization library include:

- edge shape functions on triangles or tetrahedra known as Raviart-Thomas and Nedgelec elements [3],
- nodal shape functions on rectangles ensuring the continuity of the first gradient along edges [22] known as Bogner-Fox-Schmitt element,
- nodal hierarchical-type functions on rectangles [17] allowing an arbitrary order of a polynomial shape function.

Another attempt to avoid the setup of sparse matrices and to work completely with a multidimensional array in terms of iterative solvers is documented in [15].

Acknowledgement

A. Moskovka and J. Valdman were supported by the project grant 23-04766S (GAČR) on Variational approaches to dynamical problems in continuum mechanics.

References

- [1] The MathWorks Inc.: MATLAB version 9.13.0 (R2022b), Natick, Massachusetts, <https://www.mathworks.com> .

- [2] J. Albery, C. Carstensen, S. A. Funken: *Remarks around 50 lines of Matlab: short finite element implementation*, Numer. Algorithms 20(2-3), 117–137, (1999).
- [3] I. Anjam, J. Valdman: *Fast MATLAB assembly of FEM matrices in 2D and 3D: edge elements*, Applied Mathematics and Computation 267, 252–263, (2015).
- [4] B. W. Bader, T. G. Kolda: *Algorithm 862: MATLAB tensor classes for fast algorithm prototyping*, ACM Transactions on Mathematical Software, 32(4), 635–653 (2006).
- [5] F. Bozorgnia, J. Valdman: *A FEM approximation of a two-phase obstacle problem and its a posteriori error estimate*, Computers & Mathematics with Applications 73, No. 3, 419–432, (2017).
- [6] P.G. Ciarlet: *The Finite Element Method for Elliptic Problems*, SIAM, Philadelphia, (2002).
- [7] F. Cuvelier, C. Japhet, G. Scarella: *An efficient way to assemble finite element matrices in vector languages*, BIT Numer. Math., 56, 833–864 (2016).
- [8] M. Friedrich, M. Kružík, J. Valdman: *Numerical approximation of von Kármán viscoelastic plates*, Discrete and Continuous Dynamical Systems, Series S 14(1): 299–319, (2021).
- [9] J. Koko: *Vectorized Matlab codes for linear two-dimensional elasticity*, Scientific Programming, 15(3), 157–172 (2007).
- [10] T. G. Kolda, B. W. Bader: *Tensor Decompositions and Applications*, SIAM Review, 51(3), 455–500 (2009).
- [11] S. Krömer, J. Valdman: *Global injectivity in second-gradient Nonlinear Elasticity and its approximation with penalty terms*, Mathematics and Mechanics of Solids 24, No. 11, 3644–3673, (2019).
- [12] W. Litvinov, T. Rahman, and R. Hoppe: *Problems of stationary flow of electro-rheological fluids in the cylindrical coordinate system*, SIAM J. Appl. Math., 65(5), 1633–1656 (2005).
- [13] W. Litvinov, T. Rahman, and R. Hoppe: *Model of an electro-rheological shock absorber and coupled problem for partial and ordinary differential equations with variable unknown domain*, Europ. J. Appl. Math., 18, 513–536 (2007).
- [14] S. MacLane: *Homology. Die Grundlehren der mathematischen Wissenschaften*, Band 114. Springer-Verlag, Berlin-New York, (1963).
- [15] L. Marcinkowski, J. Valdman: *MATLAB Implementation of Element-based Solvers*, In: I. Lirkov and S. Margenov (eds): LSSC 2019, LNCS 11958, 601–609, (2020).

- [16] A. Moskovka, J. Valdman: *Fast MATLAB evaluation of nonlinear energies using FEM in 2D and 3D: nodal elements*, Applied Mathematics and Computation 424, 127048, (2022).
- [17] A. Moskovka, J. Valdman: *MATLAB implementation of hp finite elements on rectangles using hierarchical basis functions*, PPAM 2022, Lecture Notes in Computer Science (LNCS) 13827, 287-299, (2023).
- [18] D. Pauly, J. Valdman: *Friedrichs/Poincare Type Constants for Gradient, Rotation, and Divergence: Theory and Numerical Experiments*, CAMWA 79, No. 11, 3027-3067, (2020).
- [19] T. Rahman: *SERF2D-MatLab (Ver 1.1) - Documentation*, University of Augsburg, (2003).
- [20] T. Rahman, J. Valdman: *Fast MATLAB assembly of FEM matrices in 2D and 3D: nodal elements*, Applied Mathematics and Computation 219, 7151-7158, (2013).
- [21] D.A. Simovici: *Linear Algebra Tools for Data Mining*, World Scientific, (2012).
- [22] J. Valdman: *MATLAB Implementation of C1 finite elements: Bogner-Fox-Schmit rectangle*, In: Wyrzykowski R., Deelman E., Dongarra J., Karczewski K. (eds) Parallel Processing and Applied Mathematics. PPAM 2019. Lecture Notes in Computer Science, vol 12044. Springer, Cham, 256-266, (2020).