

Veamy: an extensible object-oriented C++ library for the virtual element method

A. Ortiz-Bernardin^{1,2} · C. Alvarez^{1,3} ·
 N. Hitschfeld-Kahler^{3,4} · A. Russo^{5,6} ·
 R. Silva-Valenzuela^{1,2} · E. Olate-Sanzana^{1,2}

Received: date / Accepted: date

Abstract This paper summarizes the development of **Veamy**, an object-oriented C++ library for the virtual element method (VEM) on general polygonal meshes, whose modular design is focused on its extensibility. The linear elastostatic and Poisson problems in two dimensions have been chosen as the starting stage for the development of this library. The theory of the VEM in which **Veamy** is based upon is presented using a notation and a terminology that resemble the language of the finite element method (FEM) in engineering analysis. Several examples are provided to demonstrate the usage of

✉ A. Ortiz-Bernardin
 E-mail: aortizb@uchile.cl

C. Alvarez
 E-mail: capalvarez.i@gmail.com

N. Hitschfeld-Kahler
 E-mail: nancy@dcc.uchile.cl

A. Russo
 E-mail: alessandro.russo@unimib.it

R. Silva-Valenzuela
 E-mail: rsilvav@outlook.com

E. Olate-Sanzana
 E-mail: olate.edgardo@gmail.com

¹Department of Mechanical Engineering, Universidad de Chile, Av. Beauchef 851, Santiago 8370456, Chile.

²Computational and Applied Mechanics Laboratory, Center for Modern Computational Engineering, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, Av. Beauchef 851, Santiago 8370456, Chile.

³Department of Computer Science, Universidad de Chile, Av. Beauchef 851, Santiago 8370456, Chile.

⁴Meshing for Applied Science Laboratory, Center for Modern Computational Engineering, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, Av. Beauchef 851, Santiago 8370456, Chile.

⁵Dipartimento di Matematica e Applicazioni, Università di Milano-Bicocca, 20153 Milano, Italy.

⁶Istituto di Matematica Applicata e Tecnologie Informatiche del CNR, via Ferrata 1, 27100 Pavia, Italy.

Veamy, and in particular, one of them features the interaction between **Veamy** and the polygonal mesh generator **PolyMesher**. A computational performance comparison between VEM and FEM is also conducted. **Veamy** is free and open source software.

Keywords virtual element method · polygonal meshes · object-oriented programming · C++

1 Introduction

When a boundary-value problem, such as the linear elastostatic problem, is solved numerically using a Galerkin weak formulation, the trial and test displacements are replaced by their discrete representations, which adopt the form of

$$\mathbf{v}^h(\mathbf{x}) = \sum_{a=1}^N \phi_a(\mathbf{x}) \mathbf{v}_a, \quad (1)$$

where $\phi_a(\mathbf{x})$ are nodal basis functions, $\mathbf{v}_a = [v_{1a} \ v_{2a}]^T$ are nodal displacements in the two-dimensional Cartesian coordinate system and N is the number of nodes that define an element. In this paper, we only consider basis functions that span the space of functions of degree 1 (affine functions). Due to the nature of some basis functions, the discrete trial and test displacement fields may represent linear fields plus some additional non-polynomial or high-order terms. Such additional terms cause inhomogeneous deformations, and when present, integration errors appear in the numerical integration of the stiffness matrix leading to stability issues that affect the convergence of the approximation method. This is the case of polygonal and polyhedral finite element methods [14, 30, 32], and meshfree Galerkin methods [4, 5, 7, 10–13, 20–23].

The virtual element method [34] (VEM) has been presented to deal with these integration issues. In short, the method consists in the construction of an algebraic (exact) representation of the stiffness matrix without the explicit evaluation of basis functions (basis functions are *virtual*). The stiffness matrix constructed in such a manner is referred to as *computable* as opposed to the *uncomputable* stiffness matrix that would require evaluation of basis functions derivatives at quadrature points inside the element. In the VEM, the stiffness matrix is decomposed into two parts: a consistent matrix that guarantees the exact reproduction of a linear displacement field and a correction matrix that provides stability. Such a decomposition is formulated in the spirit of the Lax equivalence theorem (consistency + stability \rightarrow convergence) for finite-difference schemes and is sufficient for the method to pass the patch test [6]. Recently, the virtual element framework has been used to correct integration errors in polygonal finite element methods [15, 19, 37] and in meshfree Galerkin methods [24].

Some of the advantages that VEM exhibits over the standard finite element method (FEM) are:

- Ability to deal with complicated geometries made up by elements with arbitrary number of edges not necessarily convex, having coplanar edges and collapsing nodes, while retaining the same approximation properties of the FEM.
- Possibility of formulating high-order approximations with arbitrary order of global regularity [9].
- Adaptive mesh refinement techniques are greatly facilitated since hanging nodes become automatically included as elements with coplanar edges are accepted [36].

In this paper, object-oriented programming concepts are adopted to develop a C++ library, named **Veamy**, that implements the VEM on general polygonal meshes. The current status of this library has a focus on the linear elastostatic and Poisson problems in two dimensions, but its design is geared towards its extensibility. **Veamy** uses Eigen library [16] for linear algebra, and Triangle [27] and Clipper [18] are used for the implementation of its polygonal mesh generator, **Delynoi** [1], which is based on the constrained Voronoi diagram. Despite this built-in polygonal mesh generator, **Veamy**

is capable of interacting straightforwardly with **PolyMesher** [31], a polygonal mesh generator that is widely used in the VEM and polygonal finite elements communities.

In presenting the theory of the VEM in which **Veamy** is built upon, we adopt a notation and a terminology that resemble the language of the FEM in engineering analysis. The work of Gain et al. [15] is in line with this aim and has inspired most of the notation and terminology used in this paper. We complete the excellent work of Gain et al. by providing some additional theoretical details that are not explicitly given therein.

In **Veamy**'s programming philosophy entities commonly found in the VEM and FEM literature such as mesh, degrees of freedom, elements, element stiffness matrix and element force vector, are represented by objects. In contrast to some of the well-established free and open source object-oriented FEM codes such as FreeFEM++ [17], FEniCS [2] and Feel++ [25], **Veamy** does not generate code from the variational form of a particular problem, since that kind of software design tends to hide the implementation details that are fundamental to understand the method. On the contrary, since **Veamy**'s scope is research and teaching, in its design we wanted a direct and balanced correspondence between theory and implementation. In this sense, **Veamy** is very similar in its spirit to the 50 lines MATLAB implementation of VEM [29]. However, compared to this MATLAB implementation, **Veamy** is an improvement in the following aspects:

- Its core VEM numerical implementation is entirely built on free and open source libraries.
- It offers the possibility of using a built-in polygonal mesh generator, whose implementation is also entirely built on free and open source libraries. In addition, it allows a straightforward interaction with **PolyMesher** [31], a popular and widely used MATLAB-based polygonal mesh generator.
- It is designed using the object-oriented paradigm, which allows a safer and better code design, facilitates code reuse and recycling, code maintenance, and therefore code extension.
- Its initial release implements both the two-dimensional linear elastostatic problem and the two-dimensional Poisson problem.

We are also aware that the MATLAB Reservoir Simulation Toolbox [40] provides a module that implements first and second-order virtual element methods for Poisson-type flow equations, and that this implementation was developed as part of a master thesis [41]. The toolbox also implements a module dedicated to the VEM in linear elasticity for geomechanics simulations.

Veamy is free and open source software, and to the best of our knowledge is the first object-oriented C++ implementation of the VEM.

The remainder of this paper is structured as follows. The boundary-value problem and the Galerkin weak formulation for two-dimensional linear elastostatics are presented in Section 2. Section 3 presents the theoretical aspects of the VEM for the two-dimensional linear elastostatic problem along with the construction of the VEM element stiffness matrix and the VEM element force vector. Also in this section, the VEM element stiffness matrix for the two-dimensional Poisson problem is developed by reducing the solution dimension in the linear elastostatic VEM formulation. The object-oriented implementation of **Veamy** is described and explained in Section 4. In Section 5, some guidelines for the usage of **Veamy**'s built-in polygonal mesh generator are given. Several examples that demonstrate the usage of **Veamy** and a performance comparison between VEM and FEM are presented in Section 6. The paper ends with some concluding remarks in Section 7.

2 Model problem

2.1 Linear elastostatic boundary-value problem

Consider an elastic body that occupies the open domain $\Omega \subset \mathbb{R}^2$ and is bounded by the one-dimensional surface Γ whose unit outward normal is \mathbf{n}_Γ . The boundary is assumed to admit decompositions $\Gamma = \Gamma_g \cup \Gamma_f$ and $\emptyset = \Gamma_g \cap \Gamma_f$, where Γ_g is the essential (Dirichlet) boundary and Γ_f is the natural (Neumann) boundary. The closure of the domain is $\overline{\Omega} \equiv \Omega \cup \Gamma$. Let $\mathbf{u}(\mathbf{x}) : \Omega \rightarrow \mathbb{R}^2$ be the displacement field at a point \mathbf{x} of the elastic body when the body is subjected to external tractions $\mathbf{f}(\mathbf{x}) : \Gamma_f \rightarrow \mathbb{R}^2$ and body forces $\mathbf{b}(\mathbf{x}) : \Omega \rightarrow \mathbb{R}^2$. The imposed essential (Dirichlet) boundary conditions are $\mathbf{g}(\mathbf{x}) : \Gamma_g \rightarrow \mathbb{R}^2$. The boundary-value problem for two-dimensional linear elastostatics is: find $\mathbf{u}(\mathbf{x}) : \Omega \rightarrow \mathbb{R}^2$ such that

$$\nabla \cdot \boldsymbol{\sigma} + \mathbf{b} = 0 \quad \forall \mathbf{x} \in \Omega, \quad (2a)$$

$$\mathbf{u} = \mathbf{g} \quad \forall \mathbf{x} \in \Gamma_g, \quad (2b)$$

$$\boldsymbol{\sigma} \cdot \mathbf{n}_\Gamma = \mathbf{f} \quad \forall \mathbf{x} \in \Gamma_f, \quad (2c)$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor given by

$$\boldsymbol{\sigma} = \mathbf{D} : \frac{1}{2} \left(\nabla \mathbf{u} + \nabla^\top \mathbf{u} \right), \quad (3)$$

where \mathbf{D} is a fourth-order constant tensor that depends on the material of the elastic body.

2.2 Galerkin weak formulation

When deriving the Galerkin weak form of the linear elastostatic problem using the method of weighted residuals, with \mathbf{v} being the arbitrary test (weighting) function, the following expression for the bilinear form is obtained:

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \boldsymbol{\sigma}(\mathbf{u}) : \nabla \mathbf{v} \, d\mathbf{x}. \quad (4)$$

The gradient of the displacement field can be decomposed into its symmetric ($\nabla_S \mathbf{v}$) and skew-symmetric ($\nabla_{AS} \mathbf{v}$) parts, as follows:

$$\nabla \mathbf{v} = \nabla_S \mathbf{v} + \nabla_{AS} \mathbf{v} = \boldsymbol{\varepsilon}(\mathbf{v}) + \boldsymbol{\omega}(\mathbf{v}), \quad (5)$$

where

$$\nabla_S \mathbf{v} = \boldsymbol{\varepsilon}(\mathbf{v}) = \frac{1}{2} \left(\nabla \mathbf{v} + \nabla^\top \mathbf{v} \right) \quad (6)$$

is known as the strain tensor, and

$$\nabla_{AS} \mathbf{v} = \boldsymbol{\omega}(\mathbf{v}) = \frac{1}{2} \left(\nabla \mathbf{v} - \nabla^\top \mathbf{v} \right) \quad (7)$$

is the skew-symmetric gradient tensor that represents rotations. Since the stress tensor is symmetric, its product with the skew-symmetric gradient tensor is zero, thereby simplifying the bilinear form to

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v}) \, d\mathbf{x}, \quad (8)$$

which leads to the standard form of presenting the weak formulation: find $\mathbf{u}(\mathbf{x}) \in V$ such that

$$a(\mathbf{u}, \mathbf{v}) = \ell_b(\mathbf{v}) + \ell_f(\mathbf{v}) \quad \forall \mathbf{v}(\mathbf{x}) \in W, \quad (9a)$$

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v}) \, d\mathbf{x}, \quad (9b)$$

$$\ell_b(\mathbf{v}) = \int_{\Omega} \mathbf{b} \cdot \mathbf{v} \, d\mathbf{x}, \quad \ell_f(\mathbf{v}) = \int_{\Gamma_f} \mathbf{f} \cdot \mathbf{v} \, ds, \quad (9c)$$

where V and W are the displacement trial and test spaces:

$$V := \left\{ \mathbf{u}(\mathbf{x}) : \mathbf{u} \in \mathcal{W}(\Omega) \subseteq [H^1(\Omega)]^2, \mathbf{u} = \mathbf{g} \text{ on } \Gamma_g \right\},$$

$$W := \left\{ \mathbf{v}(\mathbf{x}) : \mathbf{v} \in \mathcal{W}(\Omega) \subseteq [H^1(\Omega)]^2, \mathbf{v} = \mathbf{0} \text{ on } \Gamma_g \right\},$$

where the space $\mathcal{W}(\Omega)$ includes linear displacement fields.

The integrals that appear in the Galerkin weak form are, with few exceptions, uncomputable, which means that in practice the integrals are evaluated at quadrature points in the interior of disjoint non overlapping elements that form a partition of the domain known as a mesh. We denote by E an element having an area of $|E|$ and a boundary ∂E that is formed by edges e of length $|e|$. The partition formed by these elements is denoted by \mathcal{T}^h , where h is the maximum diameter of the elements in the partition. The set formed by the union of all the element edges in this partition is denoted by \mathcal{E}^h , and the set formed by all the element edges lying on Γ_f is denoted by \mathcal{E}_f^h . The evaluation of the Galerkin weak form integrals using quadrature points in the interior of an element is, with few exceptions, not exact (errors are introduced) and therefore the bilinear and linear forms are replaced by their approximate mesh-dependent counterparts:

$$a^h(\mathbf{u}, \mathbf{v}) = \sum_{E \in \mathcal{T}^h} a_E^h(\mathbf{u}, \mathbf{v}), \quad \text{and} \quad \ell_b^h(\mathbf{v}) = \sum_{E \in \mathcal{T}^h} \ell_{b,E}^h(\mathbf{v}) \quad \text{and} \quad \ell_f^h(\mathbf{v}) = \sum_{e \in \mathcal{E}_f^h} \ell_{f,e}^h(\mathbf{v}),$$

respectively. Furthermore, the same partition is used to approximate the trial and test displacement fields, which leads to the standard discrete bilinear and linear forms:

$$a^h(\mathbf{u}^h, \mathbf{v}^h) = \sum_{E \in \mathcal{T}^h} a_E^h(\mathbf{u}^h, \mathbf{v}^h), \quad \text{and} \quad \ell_b^h(\mathbf{v}^h) = \sum_{E \in \mathcal{T}^h} \ell_{b,E}^h(\mathbf{v}^h) \quad \text{and} \quad \ell_f^h(\mathbf{v}^h) = \sum_{e \in \mathcal{E}_f^h} \ell_{f,e}^h(\mathbf{v}^h),$$

respectively, and the corresponding discrete (element-wise) trial and test spaces as:

$$V^h := \left\{ \mathbf{u}^h(\mathbf{x}) : \mathbf{u}^h \in \mathcal{W}(E) \subseteq [H^1(E)]^2 \quad \forall E \in \mathcal{T}^h, \mathbf{u}^h = \mathbf{g} \text{ on } \Gamma_g^h \right\},$$

$$W^h := \left\{ \mathbf{v}^h(\mathbf{x}) : \mathbf{v}^h \in \mathcal{W}(E) \subseteq [H^1(E)]^2 \quad \forall E \in \mathcal{T}^h, \mathbf{v}^h = \mathbf{0} \text{ on } \Gamma_g^h \right\},$$

where Γ_g^h is the discrete essential boundary, and $V^h \subset V$ and $W^h \subset W$.

3 The virtual element method

In standard two-dimensional finite element methods, the partition \mathcal{T}^h is usually formed by triangles and quadrilaterals. Here, we consider the possibility of using elements with arbitrary number of edges, that is, polygonal elements. So in essence, the partitioning of the domain will be achieved using a polygonal mesh generator. And for these polygonal elements, we will construct their stiffness matrices and nodal force vectors using the theory of the virtual element method.

3.1 The polygonal element

Let the domain Ω be partitioned into disjoint non overlapping polygonal elements with straight edges. The number of edges and nodes of a polygonal element are denoted by N . The unit outward normal to the element boundary in the Cartesian coordinate system is denoted by $\mathbf{n} = [n_1 \ n_2]^\top$. Fig. 1 presents a schematic representation of a polygonal element for $N = 5$, where the edge e_a of length $|e_a|$ and the edge e_{a-1} of length $|e_{a-1}|$ are the element edges incident to node a , and \mathbf{n}_a and \mathbf{n}_{a-1} are the unit outward normals to these edges, respectively.

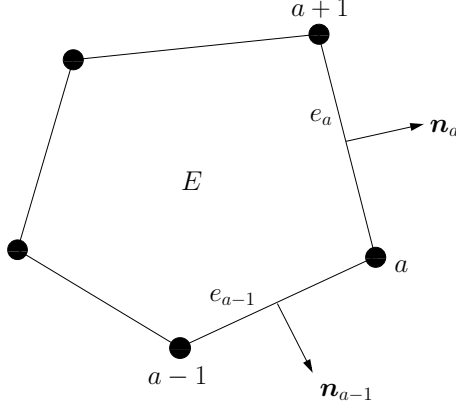


Fig. 1 Schematic representation of a polygonal element of $N = 5$ edges

3.2 Projection operators

As in finite elements, for the numerical solution to converge monotonically it is required that the displacement approximation in the polygonal element can represent rigid body modes and constant strain states. This demands that the displacement approximation in the element is at least a linear polynomial [28]. We will define projection operators that allow the extraction of the rigid body modes, the constant strain modes and the linear polynomial part of the motion. In this exposition, the displacement approximation is assumed to be continuous within the element and across inter-element boundaries (i.e. no gaps or cracks open up), which is also a necessary condition for convergence [28].

Consider the following three spaces at the element level: the space of rigid body motions (denoted by $\mathcal{R}(E)$), the space of constant strain states (denoted by $\mathcal{C}(E)$), and the space of linear displacements (denoted by $\mathcal{P}(E)$) that is able to represent rigid body motions and states of constant strains. In particular, it is required that $\mathcal{P}(E) = \mathcal{R}(E) + \mathcal{C}(E)$. Before defining these spaces, the following definition is needed: the mean value of a function h over the nodes of the polygonal element is given by

$$\bar{h} = \frac{1}{N} \sum_{j=1}^N h(\mathbf{x}_j), \quad (10)$$

where N is the number of nodes of coordinates \mathbf{x}_j that define the polygonal element. For instance, the mean value of the nodal coordinates of the polygonal element is computed using (10) and denoted by $\bar{\mathbf{x}}$. The formal definition of the spaces are given next.

The space of linear displacement is defined as

$$\mathcal{P}(E) := \left\{ \mathbf{a} + \mathbf{B}(\mathbf{x} - \bar{\mathbf{x}}) : \mathbf{a} \in \mathbb{R}^2, \mathbf{B} \in \mathbb{R}^{2 \times 2} \right\}, \quad (11)$$

where \mathbf{B} is a second-order tensor and thus can be uniquely expressed as the sum of a symmetric and a skew-symmetric tensor. Let the symmetric and skew-symmetric tensors be denoted by \mathbf{B}_S and \mathbf{B}_{AS} , respectively. Since it is required that $\mathcal{P}(E) = \mathcal{R}(E) + \mathcal{C}(E)$, we define the spaces of rigid body motions and constant strain states, respectively, as follows:

$$\mathcal{R}(E) := \left\{ \mathbf{a} + \mathbf{B}_{AS} \cdot (\mathbf{x} - \bar{\mathbf{x}}) : \mathbf{a} \in \mathbb{R}^2, \mathbf{B}_{AS} \in \mathbb{R}^{2 \times 2}, \mathbf{B}_{AS}^\top = -\mathbf{B}_{AS} \right\}, \quad (12)$$

$$\mathcal{C}(E) := \left\{ \mathbf{B}_S \cdot (\mathbf{x} - \bar{\mathbf{x}}) : \mathbf{B}_S \in \mathbb{R}^{2 \times 2}, \mathbf{B}_S^\top = \mathbf{B}_S \right\}. \quad (13)$$

Since polygonal elements can generally have more than three nodes, we have to consider that the displacement approximation in a polygonal element can be composed of a linear polynomial part and an additional non-polynomial or high-order part[†], which implies that in the element $\mathbf{u}(\mathbf{x}) \in \mathcal{W}(E) \supseteq \mathcal{P}(E)$.

To extract the components of the displacement in the three aforementioned spaces, the following projections are defined:

$$\Pi_{\mathcal{R}} : \mathcal{W}(E) \rightarrow \mathcal{R}(E), \quad \Pi_{\mathcal{R}} \mathbf{r} = \mathbf{r}, \quad \forall \mathbf{r} \in \mathcal{R}(E) \quad (14)$$

for extracting the rigid body motions,

$$\Pi_{\mathcal{C}} : \mathcal{W}(E) \rightarrow \mathcal{C}(E), \quad \Pi_{\mathcal{C}} \mathbf{c} = \mathbf{c}, \quad \forall \mathbf{c} \in \mathcal{C}(E) \quad (15)$$

for extracting the constant strain modes, and

$$\Pi_{\mathcal{P}} : \mathcal{W}(E) \rightarrow \mathcal{P}(E), \quad \Pi_{\mathcal{P}} \mathbf{p} = \mathbf{p}, \quad \forall \mathbf{p} \in \mathcal{P}(E) \quad (16)$$

for extracting the linear polynomial part. In order to guarantee that $\mathcal{P}(E) = \mathcal{R}(E) + \mathcal{C}(E)$, these operators are required to satisfy the following orthogonality conditions:

$$\Pi_{\mathcal{R}} \mathbf{c} = \mathbf{0}, \quad \forall \mathbf{c} \in \mathcal{C}(E) \quad (17)$$

$$\Pi_{\mathcal{C}} \mathbf{r} = \mathbf{0}, \quad \forall \mathbf{r} \in \mathcal{R}(E), \quad (18)$$

so that elements of \mathcal{C} have no rigid body motions and elements of \mathcal{R} have no constant strain modes, which means that $\Pi_{\mathcal{C}} \Pi_{\mathcal{R}} = \Pi_{\mathcal{R}} \Pi_{\mathcal{C}} = 0$, and thus the projection onto \mathcal{P} can be written as a direct sum of $\Pi_{\mathcal{R}}$ and $\Pi_{\mathcal{C}}$:

$$\Pi_{\mathcal{P}} = \Pi_{\mathcal{R}} + \Pi_{\mathcal{C}}. \quad (19)$$

So, any $\mathbf{u}, \mathbf{v} \in \mathcal{W}(E)$ can be decomposed into three terms, as follows:

$$\mathbf{u} = \Pi_{\mathcal{R}} \mathbf{u} + \Pi_{\mathcal{C}} \mathbf{u} + (\mathbf{u} - \Pi_{\mathcal{P}} \mathbf{u}), \quad (20a)$$

$$\mathbf{v} = \Pi_{\mathcal{R}} \mathbf{v} + \Pi_{\mathcal{C}} \mathbf{v} + (\mathbf{v} - \Pi_{\mathcal{P}} \mathbf{v}), \quad (20b)$$

[†]Just for the lowest order polygon, that is the three-node triangle, the approximation is composed solely of the linear polynomial part. For instance, the approximation in the four-node quadrilateral finite element possesses a quadratic monomial, which would be regarded as the additional high-order term.

that is, into a rigid body part, a constant strain part and the remaining non-polynomial or high-order terms.

Projection operators that satisfy the orthogonality conditions (17) and (18) can be constructed as follows: let the cell-average of the strain tensor be defined as

$$\widehat{\varepsilon}(\mathbf{v}) = \frac{1}{|E|} \int_E \varepsilon(\mathbf{v}) \, d\mathbf{x} = \frac{1}{2|E|} \int_{\partial E} (\mathbf{v} \otimes \mathbf{n} + \mathbf{n} \otimes \mathbf{v}) \, ds, \quad (21)$$

where the divergence theorem has been used to transform the volume integral into a surface integral. Similarly, the cell-average of the skew-symmetric gradient tensor is defined as

$$\widehat{\omega}(\mathbf{v}) = \frac{1}{|E|} \int_E \omega(\mathbf{v}) \, d\mathbf{x} = \frac{1}{2|E|} \int_{\partial E} (\mathbf{v} \otimes \mathbf{n} - \mathbf{n} \otimes \mathbf{v}) \, ds. \quad (22)$$

Note that by the previous definitions, $\widehat{\varepsilon}(\mathbf{v})$ and $\widehat{\omega}(\mathbf{v})$ are constant tensors in the element.

On using the preceding definitions, the projection of \mathbf{v} onto the space of rigid body motions is written as follows:

$$\Pi_{\mathcal{R}} \mathbf{v} = \widehat{\omega}(\mathbf{v}) \cdot (\mathbf{x} - \overline{\mathbf{x}}) + \overline{\mathbf{v}}, \quad (23)$$

where $\widehat{\omega}(\mathbf{v}) \cdot (\mathbf{x} - \overline{\mathbf{x}})$ and $\overline{\mathbf{v}}$ are the rotation and translation modes of \mathbf{v} , respectively. And the projection of \mathbf{v} onto the space of constant strain states is given by

$$\Pi_{\mathcal{C}} \mathbf{v} = \widehat{\varepsilon}(\mathbf{v}) \cdot (\mathbf{x} - \overline{\mathbf{x}}). \quad (24)$$

Hence, by (19) the projection of \mathbf{v} onto the space of linear displacements is written as

$$\Pi_{\mathcal{P}} \mathbf{v} = \Pi_{\mathcal{R}} \mathbf{v} + \Pi_{\mathcal{C}} \mathbf{v} = \widehat{\varepsilon}(\mathbf{v}) \cdot (\mathbf{x} - \overline{\mathbf{x}}) + \widehat{\omega}(\mathbf{v}) \cdot (\mathbf{x} - \overline{\mathbf{x}}) + \overline{\mathbf{v}}. \quad (25)$$

Note that by comparing (23) with (12), it is verified that $\mathbf{B}_{\text{AS}} = \widehat{\omega}(\mathbf{v})$. It can also be verified that $\mathbf{B}_{\text{S}} = \widehat{\varepsilon}(\mathbf{v})$.

Now, we proceed to prove that the orthogonality conditions (17) and (18) are met for the projections (23) and (24), respectively. We begin with the proof for the orthogonality condition given in (17).

Proof From (7) and (24), we have

$$\omega(\mathbf{c}) = \nabla_{\text{AS}}(\Pi_{\mathcal{C}} \mathbf{c}) = \frac{1}{2} \left(\nabla \Pi_{\mathcal{C}} \mathbf{c} - \nabla^{\top} \Pi_{\mathcal{C}} \mathbf{c} \right) = \frac{1}{2} (\widehat{\varepsilon}(\mathbf{c}) - \widehat{\varepsilon}(\mathbf{c})) = 0,$$

which by (22) implies that $\widehat{\omega}(\mathbf{c}) = 0$. In addition, $\overline{\mathbf{c}} = \overline{\Pi_{\mathcal{C}} \mathbf{c}} = \widehat{\varepsilon}(\mathbf{c}) \cdot (\overline{\mathbf{x}} - \overline{\mathbf{x}}) = 0$. Therefore, from (23) and the preceding results,

$$\Pi_{\mathcal{R}} \mathbf{c} = \widehat{\omega}(\mathbf{c}) \cdot (\mathbf{x} - \overline{\mathbf{x}}) + \overline{\mathbf{c}} = 0.$$

□

And the orthogonality condition given in (18) is proved as follows.

Proof From (23), $\nabla \Pi_{\mathcal{R}} \mathbf{r} = \widehat{\omega}(\mathbf{r})$. And since $\widehat{\omega}(\mathbf{r})$ is a skew-symmetric tensor, it follows that $\nabla^{\top} \Pi_{\mathcal{R}} \mathbf{r} = -\widehat{\omega}(\mathbf{r})$. These results are used to obtain the following strain:

$$\varepsilon(\mathbf{r}) = \nabla_{\text{S}}(\Pi_{\mathcal{R}} \mathbf{r}) = \frac{1}{2} \left(\nabla \Pi_{\mathcal{R}} \mathbf{r} + \nabla^{\top} \Pi_{\mathcal{R}} \mathbf{r} \right) = \frac{1}{2} (\widehat{\omega}(\mathbf{r}) - \widehat{\omega}(\mathbf{r})) = 0, \quad (26)$$

which shows that rigid body modes have zero strain. Therefore, from (21), (24) and (26), we obtain

$$\Pi_{\mathcal{C}} \mathbf{r} = \widehat{\varepsilon}(\mathbf{r}) \cdot (\mathbf{x} - \overline{\mathbf{x}}) = \left(\frac{1}{|E|} \int_E \varepsilon(\mathbf{r}) \, d\mathbf{x} \right) \cdot (\mathbf{x} - \overline{\mathbf{x}}) = 0.$$

□

3.3 Energy-orthogonality conditions

The crucial point in the VEM is that the displacements (20) once replaced into the bilinear form (8) effectively lead to a split of the stiffness matrix that permits to isolate the non-polynomial or high-order terms and thus take control over their behavior. The following energy-orthogonality conditions are essentials to this aim.

Consider the bilinear form (8) at the polygonal element level. The projection $\Pi_{\mathcal{C}}$ satisfies the following condition:

$$a_E(\mathbf{c}, \mathbf{v} - \Pi_{\mathcal{C}}\mathbf{v}) = 0 \quad \forall \mathbf{c} \in \mathcal{C}(E), \quad \mathbf{v} \in \mathcal{W}(E), \quad (27)$$

which means that $\mathbf{v} - \Pi_{\mathcal{C}}\mathbf{v}$ is energetically orthogonal to \mathcal{C} . The proof for this condition is as follows.

Proof From (6) and (24), we have

$$\varepsilon(\Pi_{\mathcal{C}}\mathbf{v}) = \nabla_{\mathbf{S}}(\Pi_{\mathcal{C}}\mathbf{v}) = \frac{1}{2} \left(\nabla \Pi_{\mathcal{C}}\mathbf{v} + \nabla^{\top} \Pi_{\mathcal{C}}\mathbf{v} \right) = \frac{1}{2} (\widehat{\varepsilon}(\mathbf{v}) + \widehat{\varepsilon}(\mathbf{v})) = \widehat{\varepsilon}(\mathbf{v}). \quad (28)$$

On using (8) at element level, (21), (28), and noting that $\boldsymbol{\sigma}(\mathbf{c})$ and $\widehat{\varepsilon}(\mathbf{v})$ are constant tensors, we get

$$\begin{aligned} a_E(\mathbf{c}, \mathbf{v} - \Pi_{\mathcal{C}}\mathbf{v}) &= \boldsymbol{\sigma}(\mathbf{c}) : \left[\int_E \varepsilon(\mathbf{v}) \, d\mathbf{x} - \int_E \varepsilon(\Pi_{\mathcal{C}}\mathbf{v}) \, d\mathbf{x} \right] \\ &= \boldsymbol{\sigma}(\mathbf{c}) : \left[\int_E \varepsilon(\mathbf{v}) \, d\mathbf{x} - \int_E \widehat{\varepsilon}(\mathbf{v}) \, d\mathbf{x} \right] \\ &= \boldsymbol{\sigma}(\mathbf{c}) : \left[\int_E \varepsilon(\mathbf{v}) \, d\mathbf{x} - \widehat{\varepsilon}(\mathbf{v}) \int_E d\mathbf{x} \right] \\ &= \boldsymbol{\sigma}(\mathbf{c}) : \left[\int_E \varepsilon(\mathbf{v}) \, d\mathbf{x} - \widehat{\varepsilon}(\mathbf{v}) |E| \right] \\ &= \boldsymbol{\sigma}(\mathbf{c}) : \left[\int_E \varepsilon(\mathbf{v}) \, d\mathbf{x} - \int_E \varepsilon(\mathbf{v}) \, d\mathbf{x} \right] = 0. \end{aligned}$$

□

In addition, the following energy-orthogonality condition extends from (27): the projection $\Pi_{\mathcal{P}}$ satisfies

$$a_E(\mathbf{p}, \mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v}) = 0 \quad \forall \mathbf{p} \in \mathcal{P}(E), \quad \mathbf{v} \in \mathcal{W}(E), \quad (29)$$

which means that $\mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v}$ is energetically orthogonal to \mathcal{P} . The proof is as follows.

Proof Since rigid body modes have zero strain, any term involving rigid body modes in the bilinear form is exactly zero. Hence,

$$\begin{aligned} a_E(\mathbf{p}, \mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v}) &= a_E(\mathbf{r} + \mathbf{c}, \mathbf{v} - \Pi_{\mathcal{R}}\mathbf{v} - \Pi_{\mathcal{C}}\mathbf{v}) \\ &= a_E(\mathbf{c}, \mathbf{v} - \Pi_{\mathcal{C}}\mathbf{v}) - a_E(\mathbf{c}, \Pi_{\mathcal{R}}\mathbf{v}) + a_E(\mathbf{r}, \mathbf{v} - \Pi_{\mathcal{R}}\mathbf{v} - \Pi_{\mathcal{C}}\mathbf{v}) \\ &= a_E(\mathbf{c}, \mathbf{v} - \Pi_{\mathcal{C}}\mathbf{v}) = 0, \end{aligned}$$

where (27) has been used in the last equality. □

A last observation is noted. Since $\mathbf{p} = \mathbf{r} + \mathbf{c}$ and $a_E(\mathbf{r}, \cdot) = 0$, the following energy-orthogonality condition emanates from (29):

$$a_E(\mathbf{c}, \mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v}) = 0 \quad \forall \mathbf{c} \in \mathcal{C}(E), \quad \mathbf{v} \in \mathcal{W}(E). \quad (30)$$

3.4 The VEM bilinear form

Substituting the VEM decomposition (20) into the bilinear form (8) leads to the following splitting of the bilinear form at element level:

$$\begin{aligned} a_E(\mathbf{u}, \mathbf{v}) &= a_E(\Pi_{\mathcal{R}}\mathbf{u} + \Pi_{\mathcal{C}}\mathbf{u} + (\mathbf{u} - \Pi_{\mathcal{P}}\mathbf{u}), \Pi_{\mathcal{R}}\mathbf{v} + \Pi_{\mathcal{C}}\mathbf{v} + (\mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v})) \\ &= a_E(\Pi_{\mathcal{C}}\mathbf{u}, \Pi_{\mathcal{C}}\mathbf{v}) + a_E(\mathbf{u} - \Pi_{\mathcal{P}}\mathbf{u}, \mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v}), \end{aligned} \quad (31)$$

where the symmetry of the bilinear form, the fact that $\Pi_{\mathcal{R}}\mathbf{v}$ and $\Pi_{\mathcal{R}}\mathbf{u}$ do not contribute in the bilinear form (both have zero strain), and the energy-orthogonality condition (30) have been used.

The first term on the right-hand side of (31) is the bilinear form associated with the constant strain modes that provides consistency (it leads to the *consistency* stiffness) and the second term is the bilinear form associated with the non-polynomial or high-order terms that provides stability (it leads to the *stability* stiffness). We come back to these concepts later in this section.

3.5 Projection operators revisited

The VEM projection operators (23), (24) and (25) can be “intrinsically” obtained from the energy-orthogonality condition (29). To this end, observe that $\boldsymbol{\sigma}(\mathbf{p})$ and $\nabla(\Pi_{\mathcal{P}}\mathbf{v})$ are both constant fields since $\mathbf{p}, \Pi_{\mathcal{P}}\mathbf{v} \in \mathcal{P}(E)$. Then, use the preceding observation and the bilinear form given in (4) to obtain

$$\begin{aligned} a_E(\mathbf{p}, \mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v}) &= \int_E \boldsymbol{\sigma}(\mathbf{p}) : \nabla(\mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v}) \, dx \\ &= \boldsymbol{\sigma}(\mathbf{p}) : \left[\int_E \nabla \mathbf{v} \, dx - \nabla(\Pi_{\mathcal{P}}\mathbf{v}) \int_E dx \right] \\ &= \boldsymbol{\sigma}(\mathbf{p}) : \left[\int_E \nabla \mathbf{v} \, dx - \nabla(\Pi_{\mathcal{P}}\mathbf{v})|E| \right]. \end{aligned} \quad (32)$$

And since (32) is required to be exactly zero by (29), it leads to

$$\nabla(\Pi_{\mathcal{P}}\mathbf{v}) = \frac{1}{|E|} \int_E \nabla \mathbf{v} \, dx. \quad (33)$$

On using (5), (21) and (22), Eq. (33) can be further developed as follows:

$$\nabla(\Pi_{\mathcal{P}}\mathbf{v}) = \frac{1}{|E|} \int_E \boldsymbol{\varepsilon}(\mathbf{v}) \, dx + \frac{1}{|E|} \int_E \boldsymbol{\omega}(\mathbf{v}) \, dx = \widehat{\boldsymbol{\varepsilon}}(\mathbf{v}) + \widehat{\boldsymbol{\omega}}(\mathbf{v}). \quad (34)$$

Then from (34), $\Pi_{\mathcal{P}}\mathbf{v}$ must have the following form:

$$\Pi_{\mathcal{P}}\mathbf{v} = \widehat{\boldsymbol{\varepsilon}}(\mathbf{v}) \cdot \mathbf{x} + \widehat{\boldsymbol{\omega}}(\mathbf{v}) \cdot \mathbf{x} + a_0. \quad (35)$$

And since a_0 is a constant, this means that the projection $a_E(\mathbf{p}, \mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v}) = 0$ defines $\Pi_{\mathcal{P}}\mathbf{v}$ only up to a constant. Thus, to find a_0 we need a projection operator onto constants $\Pi_0 : \mathcal{W}(E) \rightarrow \mathbb{R}^2$ such that

$$\Pi_0(\Pi_{\mathcal{P}}\mathbf{v}) = \Pi_0\mathbf{v}. \quad (36)$$

It suffices to use (10) as the projection operator onto constants [34, 36]. This gives

$$\Pi_0\mathbf{v} = \frac{1}{N} \sum_{j=1}^N \mathbf{v}(\mathbf{x}_j) = \bar{\mathbf{v}}, \quad (37)$$

which satisfies (36) as shown in the proof that follows.

Proof

$$\begin{aligned}
 \Pi_0(\Pi_{\mathcal{P}}\mathbf{v}) &= \Pi_0(\Pi_{\mathcal{R}}\mathbf{v}) + \Pi_0(\Pi_{\mathcal{C}}\mathbf{v}) \\
 &= \hat{\boldsymbol{\omega}}(\mathbf{v}) \cdot (\Pi_0\mathbf{x} - \bar{\mathbf{x}}) + \bar{\mathbf{v}} + \hat{\boldsymbol{\varepsilon}}(\mathbf{v}) \cdot (\Pi_0\mathbf{x} - \bar{\mathbf{x}}) \\
 &= \hat{\boldsymbol{\omega}}(\mathbf{v}) \cdot (\bar{\mathbf{x}} - \bar{\mathbf{x}}) + \bar{\mathbf{v}} + \hat{\boldsymbol{\varepsilon}}(\mathbf{v}) \cdot (\bar{\mathbf{x}} - \bar{\mathbf{x}}) \\
 &= \bar{\mathbf{v}} \\
 &= \Pi_0\mathbf{v},
 \end{aligned}$$

where it has been used that $\hat{\boldsymbol{\omega}}(\mathbf{v})$ and $\hat{\boldsymbol{\varepsilon}}(\mathbf{v})$ are constant tensors. \square

Applying (36) to (35) gives

$$\begin{aligned}
 \Pi_0(\Pi_{\mathcal{P}}\mathbf{v}) &= \hat{\boldsymbol{\varepsilon}}(\mathbf{v}) \cdot \Pi_0\mathbf{x} + \hat{\boldsymbol{\omega}}(\mathbf{v}) \cdot \Pi_0\mathbf{x} + \Pi_0a_0 = \Pi_0\mathbf{v} \\
 &= \hat{\boldsymbol{\varepsilon}}(\mathbf{v}) \cdot \bar{\mathbf{x}} + \hat{\boldsymbol{\omega}}(\mathbf{v}) \cdot \bar{\mathbf{x}} + a_0 = \bar{\mathbf{v}}.
 \end{aligned} \tag{38}$$

And solving for a_0 in (38) leads to

$$a_0 = \bar{\mathbf{v}} - \hat{\boldsymbol{\varepsilon}}(\mathbf{v}) \cdot \bar{\mathbf{x}} - \hat{\boldsymbol{\omega}}(\mathbf{v}) \cdot \bar{\mathbf{x}}. \tag{39}$$

Finally, substituting (39) into (35) yields the projection operator already given in (25):

$$\Pi_{\mathcal{P}}\mathbf{v} = \hat{\boldsymbol{\varepsilon}}(\mathbf{v}) \cdot (\mathbf{x} - \bar{\mathbf{x}}) + \hat{\boldsymbol{\omega}}(\mathbf{v}) \cdot (\mathbf{x} - \bar{\mathbf{x}}) + \bar{\mathbf{v}}. \tag{40}$$

3.6 Projection matrices

The projection matrices are constructed by discretizing the projection operators. For ease of derivations, we begin by writing the projections $\Pi_{\mathcal{R}}\mathbf{v}$ and $\Pi_{\mathcal{C}}\mathbf{v}$ in terms of their space basis. To this end, consider the two-dimensional Cartesian space and the skew-symmetry of $\hat{\boldsymbol{\omega}} \equiv \hat{\boldsymbol{\omega}}(\mathbf{v})^\dagger$. The projection (23) can be written as follows:

$$\begin{aligned}
 \Pi_{\mathcal{R}}\mathbf{v} &= \begin{bmatrix} \bar{v}_1 + (x_2 - \bar{x}_2)\hat{\omega}_{12} \\ \bar{v}_2 - (x_1 - \bar{x}_1)\hat{\omega}_{12} \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & (x_2 - \bar{x}_2) \\ 0 & 1 & -(x_1 - \bar{x}_1) \end{bmatrix} \begin{bmatrix} \bar{v}_1 \\ \bar{v}_2 \\ \hat{\omega}_{12} \end{bmatrix} \\
 &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \bar{v}_1 + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \bar{v}_2 + \begin{bmatrix} (x_2 - \bar{x}_2) \\ -(x_1 - \bar{x}_1) \end{bmatrix} \hat{\omega}_{12} \\
 &= \mathbf{r}_1\bar{v}_1 + \mathbf{r}_2\bar{v}_2 + \mathbf{r}_3\hat{\omega}_{12}.
 \end{aligned} \tag{41}$$

Thus, the basis for the space of rigid body modes is:

$$\mathbf{r}_1 = \begin{bmatrix} 1 & 0 \end{bmatrix}^\top, \mathbf{r}_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}^\top, \mathbf{r}_3 = \begin{bmatrix} (x_2 - \bar{x}_2) & -(x_1 - \bar{x}_1) \end{bmatrix}^\top. \tag{42}$$

[†]Note that $\hat{\omega}_{11} = \hat{\omega}_{22} = 0$ and $\hat{\omega}_{21} = -\hat{\omega}_{12}$.

Similarly, on considering the symmetry of $\hat{\varepsilon} \equiv \hat{\varepsilon}(\mathbf{v})$, the projection (24) can be written as

$$\begin{aligned}
 \Pi_C \mathbf{v} &= \begin{bmatrix} (x_1 - \bar{x}_1)\hat{\varepsilon}_{11} + (x_2 - \bar{x}_2)\hat{\varepsilon}_{12} \\ (x_1 - \bar{x}_1)\hat{\varepsilon}_{12} + (x_2 - \bar{x}_2)\hat{\varepsilon}_{22} \end{bmatrix} \\
 &= \begin{bmatrix} (x_1 - \bar{x}_1) & 0 & (x_2 - \bar{x}_2) \\ 0 & (x_2 - \bar{x}_2) & (x_1 - \bar{x}_1) \end{bmatrix} \begin{bmatrix} \hat{\varepsilon}_{11} \\ \hat{\varepsilon}_{22} \\ \hat{\varepsilon}_{12} \end{bmatrix} \\
 &= \begin{bmatrix} (x_1 - \bar{x}_1) \\ 0 \end{bmatrix} \hat{\varepsilon}_{11} + \begin{bmatrix} 0 \\ (x_2 - \bar{x}_2) \end{bmatrix} \hat{\varepsilon}_{22} + \begin{bmatrix} (x_2 - \bar{x}_2) \\ (x_1 - \bar{x}_1) \end{bmatrix} \hat{\varepsilon}_{12} \\
 &= \mathbf{c}_1 \hat{\varepsilon}_{11} + \mathbf{c}_2 \hat{\varepsilon}_{22} + \mathbf{c}_3 \hat{\varepsilon}_{12}.
 \end{aligned} \tag{43}$$

Thus, the basis for the space of constant strain states is:

$$\mathbf{c}_1 = \begin{bmatrix} (x_1 - \bar{x}_1) & 0 \end{bmatrix}^\top, \quad \mathbf{c}_2 = \begin{bmatrix} 0 & (x_2 - \bar{x}_2) \end{bmatrix}^\top, \quad \mathbf{c}_3 = \begin{bmatrix} (x_2 - \bar{x}_2) & (x_1 - \bar{x}_1) \end{bmatrix}^\top. \tag{44}$$

On each polygonal element of N edges with nodal coordinates denoted by $\mathbf{x}_a = [x_{1a} \quad x_{2a}]^\top$, the discrete trial and test displacements are given by

$$\mathbf{u}^h(\mathbf{x}) = \sum_{a=1}^N \phi_a(\mathbf{x}) \mathbf{u}_a, \quad \mathbf{v}^h(\mathbf{x}) = \sum_{b=1}^N \phi_b(\mathbf{x}) \mathbf{v}_b, \tag{45}$$

where $\phi_a(\mathbf{x})$ and $\phi_b(\mathbf{x})$ are nodal basis functions, and $\mathbf{u}_a = [u_{1a} \quad u_{2a}]^\top$ and $\mathbf{v}_b = [v_{1b} \quad v_{2b}]^\top$ are nodal displacements. The nodal basis functions are also used for the discretization of the components of the basis for the space of rigid body motions:

$$\mathbf{r}_\alpha^h(\mathbf{x}) = \sum_{a=1}^N \phi_a(\mathbf{x}) \mathbf{r}_\alpha(\mathbf{x}_a), \quad \alpha = 1, \dots, 3 \tag{46}$$

and the components of the basis for the space of constant strain modes:

$$\mathbf{c}_\beta^h(\mathbf{x}) = \sum_{a=1}^N \phi_a(\mathbf{x}) \mathbf{c}_\beta(\mathbf{x}_a), \quad \beta = 1, \dots, 3. \tag{47}$$

The discrete version of the projection to extract the rigid body motions is obtained by substituting (45) and (46) into (41), which yields

$$\begin{aligned}
 (\Pi_{\mathcal{R}} \mathbf{v}^h)_{ab} &= \begin{bmatrix} \phi_a \begin{bmatrix} 1 \\ 0 \end{bmatrix} & \phi_a \begin{bmatrix} 0 \\ 1 \end{bmatrix} & \phi_a \begin{bmatrix} (x_{2a} - \bar{x}_2) \\ -(x_{1a} - \bar{x}_1) \end{bmatrix} \end{bmatrix} \begin{bmatrix} \bar{\phi}_b v_{1b} \\ \bar{\phi}_b v_{2b} \\ q_{2b} v_{1b} - q_{1b} v_{2b} \end{bmatrix} \\
 &= \begin{bmatrix} \phi_a & 0 \\ 0 & \phi_a \end{bmatrix} \begin{bmatrix} 1 & 0 & (x_{2a} - \bar{x}_2) \\ 0 & 1 & -(x_{1a} - \bar{x}_1) \end{bmatrix} \begin{bmatrix} \bar{\phi}_b & 0 \\ 0 & \bar{\phi}_b \\ q_{2b} & -q_{1b} \end{bmatrix} \begin{bmatrix} v_{1b} \\ v_{2b} \end{bmatrix},
 \end{aligned} \tag{48}$$

where

$$q_{ia} = \frac{1}{2|E|} \int_{\partial E} \phi_a n_i \, ds, \quad i = 1, 2 \tag{49}$$

appeared because of the discretization of $\hat{\omega}_{12}$ (see (22)). The matrix form of (48) is obtained by expanding the nodal indexes, as follows:

$$\Pi_{\mathcal{R}} \mathbf{v}^h = \sum_{a=1}^N \sum_{b=1}^N (\Pi_{\mathcal{R}} \mathbf{v}^h)_{ab} = \mathbf{N} \mathbf{P}_{\mathcal{R}} \mathbf{q}, \tag{50}$$

where

$$\mathbf{N} = [(\mathbf{N})_1 \quad \cdots \quad (\mathbf{N})_a \quad \cdots \quad (\mathbf{N})_N]; \quad (\mathbf{N})_a = \begin{bmatrix} \phi_a & 0 \\ 0 & \phi_a \end{bmatrix}, \quad (51)$$

$$\mathbf{q} = [\mathbf{v}_1^\top \quad \cdots \quad \mathbf{v}_a^\top \quad \cdots \quad \mathbf{v}_N^\top]^\top; \quad \mathbf{v}_a = [v_{1a} \quad v_{2a}]^\top \quad (52)$$

and

$$\mathbf{P}_{\mathcal{R}} = \mathbf{H}_{\mathcal{R}} \mathbf{W}_{\mathcal{R}}^\top \quad (53)$$

with

$$\mathbf{H}_{\mathcal{R}} = [(\mathbf{H}_{\mathcal{R}})_1 \quad \cdots \quad (\mathbf{H}_{\mathcal{R}})_a \quad \cdots \quad (\mathbf{H}_{\mathcal{R}})_N]^\top, \quad (\mathbf{H}_{\mathcal{R}})_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ (x_{2a} - \bar{x}_2) & -(x_{1a} - \bar{x}_1) \end{bmatrix}^\top \quad (54)$$

and

$$\mathbf{W}_{\mathcal{R}} = [(\mathbf{W}_{\mathcal{R}})_1 \quad \cdots \quad (\mathbf{W}_{\mathcal{R}})_a \quad \cdots \quad (\mathbf{W}_{\mathcal{R}})_N]^\top, \quad (\mathbf{W}_{\mathcal{R}})_a = \begin{bmatrix} \bar{\phi}_a & 0 \\ 0 & \bar{\phi}_a \\ q_{2a} & -q_{1a} \end{bmatrix}^\top. \quad (55)$$

Similarly, on substituting (45) and (47) into (43) leads to the following discrete version of the projection to extract the constant strain modes:

$$\begin{aligned} (\Pi_C \mathbf{v}^h)_{ab} &= \begin{bmatrix} \phi_a \begin{bmatrix} (x_{1a} - \bar{x}_1) \\ 0 \end{bmatrix} & \phi_a \begin{bmatrix} 0 \\ (x_{2a} - \bar{x}_2) \end{bmatrix} & \phi_a \begin{bmatrix} (x_{2a} - \bar{x}_2) \\ (x_{1a} - \bar{x}_1) \end{bmatrix} \end{bmatrix} \\ &\quad \times \begin{bmatrix} 2q_{1b}v_{1b} \\ 2q_{2b}v_{2b} \\ q_{2b}v_{1b} + q_{1b}v_{2b} \end{bmatrix} \\ &= \begin{bmatrix} \phi_a & 0 \\ 0 & \phi_a \end{bmatrix} \begin{bmatrix} (x_{1a} - \bar{x}_1) & 0 & (x_{2a} - \bar{x}_2) \\ 0 & (x_{2a} - \bar{x}_2) & (x_{1a} - \bar{x}_1) \end{bmatrix} \begin{bmatrix} 2q_{1b} & 0 \\ 0 & 2q_{2b} \\ q_{2b} & q_{1b} \end{bmatrix} \\ &\quad \times \begin{bmatrix} v_{1b} \\ v_{2b} \end{bmatrix}. \end{aligned} \quad (56)$$

The matrix form of (56) is obtained by expanding the nodal indexes, as follows:

$$\Pi_C \mathbf{v}^h = \sum_{a=1}^N \sum_{b=1}^N (\Pi_C \mathbf{v}^h)_{ab} = \mathbf{N} \mathbf{P}_C \mathbf{q}, \quad (57)$$

where

$$\mathbf{P}_C = \mathbf{H}_C \mathbf{W}_C^\top \quad (58)$$

with

$$\mathbf{H}_C = [(\mathbf{H}_C)_1 \quad \cdots \quad (\mathbf{H}_C)_a \quad \cdots \quad (\mathbf{H}_C)_N]^\top, \quad (\mathbf{H}_C)_a = \begin{bmatrix} (x_{1a} - \bar{x}_1) & 0 \\ 0 & (x_{2a} - \bar{x}_2) \\ (x_{2a} - \bar{x}_2) & (x_{1a} - \bar{x}_1) \end{bmatrix}^\top \quad (59)$$

and

$$\mathbf{W}_C = [(\mathbf{W}_C)_1 \quad \cdots \quad (\mathbf{W}_C)_a \quad \cdots \quad (\mathbf{W}_C)_N]^\top, \quad (\mathbf{W}_C)_a = \begin{bmatrix} 2q_{1a} & 0 \\ 0 & 2q_{2a} \\ q_{2a} & q_{1a} \end{bmatrix}^\top. \quad (60)$$

The matrix form of the projection to extract the polynomial part of the displacement field is then $\mathbf{P}_{\mathcal{P}} = \mathbf{P}_{\mathcal{R}} + \mathbf{P}_C$.

For the development of the element *consistency* stiffness matrix in the next section, it will be useful to have the following alternative expression for the discrete projection to extract the constant strain modes:

$$\begin{aligned}
\Pi_C \mathbf{v}^h &= \mathbf{c}_1 \hat{\varepsilon}_{11} + \mathbf{c}_2 \hat{\varepsilon}_{22} + \mathbf{c}_3 \hat{\varepsilon}_{12} \\
&= \mathbf{c}_1 \sum_{b=1}^N 2q_{1b} v_{1b} + \mathbf{c}_2 \sum_{b=1}^N 2q_{2b} v_{2b} + \mathbf{c}_3 \sum_{b=1}^N (q_{2b} v_{1b} + q_{1b} v_{2b}) \\
&= \sum_{b=1}^N \begin{bmatrix} 2q_{1b} \mathbf{c}_1 + q_{2b} \mathbf{c}_3 & 2q_{2b} \mathbf{c}_2 + q_{1b} \mathbf{c}_3 \end{bmatrix} \begin{bmatrix} v_{1b} \\ v_{2b} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{c}_1 & \mathbf{c}_2 & \mathbf{c}_3 \end{bmatrix} \sum_{b=1}^N \begin{bmatrix} 2q_{1b} & 0 \\ 0 & 2q_{2b} \\ q_{2b} & q_{1b} \end{bmatrix} \begin{bmatrix} v_{1b} \\ v_{2b} \end{bmatrix} \\
&= \mathbf{c} \mathbf{W}_C^\top \mathbf{q}.
\end{aligned} \tag{61}$$

Finally, the discrete version of the projection onto constants is obtained by substituting (45) into (37), which leads to

$$\begin{aligned}
\left(\Pi_0 \mathbf{v}^h \right)_a &= \frac{1}{N} \sum_{j=1}^N \begin{bmatrix} \phi_a(\mathbf{x}_j) & 0 \\ 0 & \phi_a(\mathbf{x}_j) \end{bmatrix} \begin{bmatrix} v_{1a} \\ v_{2a} \end{bmatrix} \\
&= \begin{bmatrix} \bar{\phi}_a & 0 \\ 0 & \bar{\phi}_a \end{bmatrix} \begin{bmatrix} v_{1a} \\ v_{2a} \end{bmatrix}.
\end{aligned} \tag{62}$$

The matrix form of (62) is obtained by expanding the nodal indexes as

$$\Pi_0 \mathbf{v}^h = \sum_{a=1}^N \left(\Pi_0 \mathbf{v}^h \right)_a = \bar{\mathbf{N}} \mathbf{q}, \tag{63}$$

where

$$\bar{\mathbf{N}} = [(\bar{\mathbf{N}})_1 \quad \cdots \quad (\bar{\mathbf{N}})_a \quad \cdots \quad (\bar{\mathbf{N}})_N]; \quad (\bar{\mathbf{N}})_a = \begin{bmatrix} \bar{\phi}_a & 0 \\ 0 & \bar{\phi}_a \end{bmatrix}. \tag{64}$$

3.7 VEM element stiffness matrix

The decomposition given in (31) is used to construct the approximate mesh-dependent bilinear form $a_E^h(\mathbf{u}, \mathbf{v})$ in a way that is computable at the element level. To this end, we approximate the quantity $a_E(\mathbf{u} - \Pi_{\mathcal{P}} \mathbf{u}, \mathbf{v} - \Pi_{\mathcal{P}} \mathbf{v})$, which is uncomputable, with a computable one given by $s_E(\mathbf{u} - \Pi_{\mathcal{P}} \mathbf{u}, \mathbf{v} - \Pi_{\mathcal{P}} \mathbf{v})$ and define

$$a_E^h(\mathbf{u}, \mathbf{v}) := a_E(\Pi_C \mathbf{u}, \Pi_C \mathbf{v}) + s_E(\mathbf{u} - \Pi_{\mathcal{P}} \mathbf{u}, \mathbf{v} - \Pi_{\mathcal{P}} \mathbf{v}), \tag{65}$$

where its right-hand side, as it will be revealed in the sequel, is computed algebraically. The decomposition (65) has been proved to be endowed with the following crucial properties for establishing convergence [34, 35]:

For all h and for all E in \mathcal{T}^h

- *Consistency*: $\forall \mathbf{p} \in \mathcal{P}(E), \forall \mathbf{v}^h \in V^h$

$$a_E^h(\mathbf{p}, \mathbf{v}^h) = a_E(\mathbf{p}, \mathbf{v}^h). \tag{66}$$

- *Stability*: \exists two constants $\alpha_* > 0$ and $\alpha^* > 0$, independent of h and of E , such that

$$\forall \mathbf{v}^h \in V^h \quad \alpha_* a_E(\mathbf{v}^h, \mathbf{v}^h) \leq a_E^h(\mathbf{v}^h, \mathbf{v}^h) \leq \alpha^* a_E(\mathbf{v}^h, \mathbf{v}^h). \quad (67)$$

Of course, the computable $s_E(\mathbf{u} - \Pi_{\mathcal{P}}\mathbf{u}, \mathbf{v} - \Pi_{\mathcal{P}}\mathbf{v})$ must be chosen such that (66) and (67) hold.

The discrete version of the VEM bilinear form (65) is constructed as follows. Substitute (61) into the first term of the right-hand side of (65); use (50) and (57) to obtain $\Pi_{\mathcal{P}}\mathbf{v}^h = \Pi_{\mathcal{R}}\mathbf{v}^h + \Pi_{\mathcal{C}}\mathbf{v}^h = \mathbf{N}\mathbf{P}_{\mathcal{P}}\mathbf{q}$, where $\mathbf{P}_{\mathcal{P}} = \mathbf{H}_{\mathcal{R}}\mathbf{W}_{\mathcal{R}}^{\top} + \mathbf{H}_{\mathcal{C}}\mathbf{W}_{\mathcal{C}}^{\top}$. Also, note that $\mathbf{v}^h = \mathbf{N}\mathbf{q}$. Then, substitute the expressions for $\Pi_{\mathcal{P}}\mathbf{v}^h$ and \mathbf{v}^h into the second term of the right-hand side of (65). This yields

$$\begin{aligned} a_E^h(\mathbf{u}^h, \mathbf{v}^h) &= a_E(\mathbf{c}\mathbf{W}_{\mathcal{C}}^{\top}\mathbf{d}, \mathbf{c}\mathbf{W}_{\mathcal{C}}^{\top}\mathbf{q}) + s_E(\mathbf{N}\mathbf{d} - \mathbf{N}\mathbf{P}_{\mathcal{P}}\mathbf{d}, \mathbf{N}\mathbf{q} - \mathbf{N}\mathbf{P}_{\mathcal{P}}\mathbf{q}) \\ &= \mathbf{q}^{\top}\mathbf{W}_{\mathcal{C}} a_E(\mathbf{c}^{\top}, \mathbf{c}) \mathbf{W}_{\mathcal{C}}^{\top}\mathbf{d} + \mathbf{q}^{\top}(\mathbf{I}_{2N} - \mathbf{P}_{\mathcal{P}})^{\top} s_E(\mathbf{N}^{\top}, \mathbf{N}) (\mathbf{I}_{2N} - \mathbf{P}_{\mathcal{P}}) \mathbf{d} \\ &= \mathbf{q}^{\top} |E| \mathbf{W}_{\mathcal{C}} \mathbf{D} \mathbf{W}_{\mathcal{C}}^{\top} \mathbf{d} + \mathbf{q}^{\top} (\mathbf{I}_{2N} - \mathbf{P}_{\mathcal{P}})^{\top} \mathbf{S}_E (\mathbf{I}_{2N} - \mathbf{P}_{\mathcal{P}}) \mathbf{d}, \end{aligned} \quad (68)$$

where \mathbf{I}_{2N} is the identity $(2N \times 2N)$ matrix, \mathbf{d} is the column vector of nodal displacements, and $\mathbf{S}_E = s_E(\mathbf{N}^{\top}, \mathbf{N})$. On using Voigt notation and observing that $\boldsymbol{\varepsilon}(\mathbf{c}) = [\varepsilon_{11}(\mathbf{c}) \quad \varepsilon_{22}(\mathbf{c}) \quad \varepsilon_{12}(\mathbf{c})]^{\top} = \mathbf{I}_3$ (the identity (3×3) matrix), in (68) we have used that $a_E(\mathbf{c}^{\top}, \mathbf{c}) = \int_E \boldsymbol{\varepsilon}^{\top}(\mathbf{c}) \mathbf{D} \boldsymbol{\varepsilon}(\mathbf{c}) \, d\mathbf{x} = \mathbf{D} \int_E d\mathbf{x} = |E| \mathbf{D}$, where \mathbf{D} is the constitutive matrix for an isotropic linear elastic material given by

$$\mathbf{D} = \frac{E_Y}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & 2(1-2\nu) \end{bmatrix} \quad (69)$$

for plane strain condition, and

$$\mathbf{D} = \frac{E_Y}{(1-\nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & 2(1-\nu) \end{bmatrix} \quad (70)$$

for plane stress condition, where E_Y is the Young's modulus and ν is the Poisson's ratio.

The first term on the right-hand side of (68) is the *consistency* part of the discrete VEM bilinear form that provides patch test satisfaction when the solution is a linear displacement field (condition (66) is satisfied). The second term on the right-hand side of (68) is the *stability* part of the discrete VEM bilinear form and is dependent on the matrix $\mathbf{S}_E = s_E(\mathbf{N}^{\top}, \mathbf{N})$. This matrix must be chosen such that condition (67) holds without putting at risk condition (66) already taken care of by the consistency part. We can gather information about the desirable properties of \mathbf{S}_E by inspection. To this end, let the space of non-polynomial and high-order terms be denoted by \mathcal{H} . Hence, for $\mathbf{h} \in \mathcal{H}$ the VEM bilinear form given in (65) yields

$$a_E(\mathbf{h}, \mathbf{h}) = s_E(\mathbf{h}, \mathbf{h}). \quad (71)$$

Clearly, in (71) s_E must be positive definite on the space \mathcal{H} so that non-zero non-polynomial/high-order deformation modes are not assigned zero strain energy; s_E should also scale like the exact bilinear form a_E . Therefore, the computable matrix \mathbf{S}_E must possess the aforementioned properties. There are quite a few possibilities for this matrix (see for instance [15, 34, 35]). Herein, we adopt \mathbf{S}_E given by [15]

$$\mathbf{S}_E = \alpha_E \mathbf{I}_{2N}, \quad \alpha_E = \gamma \frac{|E| \text{trace}(\mathbf{D})}{\text{trace}(\mathbf{H}_{\mathcal{C}}^{\top} \mathbf{H}_{\mathcal{C}})}, \quad (72)$$

where α_E is the scaling parameter and γ is typically set to 1.

From (68), the final expression for the VEM element stiffness matrix can be written as the summation of the *consistency* stiffness matrix and the *stability* stiffness matrix, respectively, as follows:

$$\mathbf{K}_E = |E| \mathbf{W}_C \mathbf{D} \mathbf{W}_C^\top + (\mathbf{I}_{2N} - \mathbf{P}_P)^\top \mathbf{S}_E (\mathbf{I}_{2N} - \mathbf{P}_P), \quad (73)$$

where we recall that $\mathbf{P}_P = \mathbf{H}_R \mathbf{W}_R^\top + \mathbf{H}_C \mathbf{W}_C^\top$. Note that \mathbf{H}_R and \mathbf{H}_C , which are given in (54) and (59), respectively, are easily computed using the nodal coordinates of the element. However, in order to compute \mathbf{W}_R and \mathbf{W}_C (see their expressions in (55) and (60), respectively), we need some knowledge of the basis functions so that $\bar{\phi}_a$ and q_{ia} can be determined. Observe that $\bar{\phi}_a$ is computed using (37), which requires the knowledge of the basis functions at the element nodes. And q_{ia} is computed using (49), which requires the knowledge of the basis functions on the element edges. Therefore, everything we need to know about the basis functions is their behavior on the element boundary. In the VEM, the basis functions are assumed to be piecewise linear (edge by edge) and continuous on the element edges, which implies that $\phi_a(\mathbf{x}_b) = \delta_{ab}$, where \mathbf{x}_b are the nodal coordinates and δ_{ab} is the Kronecker delta function. These assumptions permit the computation of $\bar{\phi}_a$ simply as

$$\bar{\phi}_a = \frac{1}{N} \sum_{j=1}^N \phi_a(\mathbf{x}_j) = \frac{1}{N}, \quad (74)$$

and the exact computation of q_{ia} through a trapezoidal rule, which gives

$$q_{ia} = \frac{1}{2|E|} \int_{\partial E} \phi_a n_i \, ds = \frac{1}{4|E|} (|e_{a-1}| \{n_i\}_{a-1} + |e_a| \{n_i\}_a), \quad i = 1, 2, \quad (75)$$

where $\{n_i\}_a$ are the components of \mathbf{n}_a and $|e_a|$ is the length of the edge incident to node a , as defined in Fig. 1.

In the VEM, Eqs. (74) and (75) are used, which means that the basis functions are not evaluated explicitly — in fact, they are never computed. Thus, basis functions are said to be *virtual*. In addition, the knowledge of the basis functions in the interior of the element is not required, but the approximation in the interior of the element is still linear and is given by (25). Therefore, a more specific definition of the displacement trial space than the one already given in Section 2.2 is as follows [34, 37]:

$$V^h := \left\{ \mathbf{v}^h \in [H^1(E) \cap C^0(E)]^2 : \Delta \mathbf{v}^h = \mathbf{0} \text{ in } E, \mathbf{v}^h|_e = \mathcal{P}(e) \forall e \in \partial E \right\}, \quad (76)$$

which is known as the virtual element space.

3.8 VEM element body and traction force vectors

Since we only consider linear displacements, the body force can be approximated by a piecewise constant. Typically, this piecewise constant approximation is defined as the cell-average $\mathbf{b}^h = \frac{1}{|E|} \int_E \mathbf{b} \, d\mathbf{x} = \hat{\mathbf{b}}$. Therefore, the VEM element body force vector can be simply computed as follows [34, 35, 39]:

$$\ell_{b,E}^h(\mathbf{v}^h) = \int_E \mathbf{b}^h \cdot \Pi_0 \mathbf{v}^h \, d\mathbf{x} = |E| \hat{\mathbf{b}} \cdot \bar{\mathbf{v}}^h = \mathbf{q}^\top |E| \bar{\mathbf{N}}^\top \hat{\mathbf{b}}. \quad (77)$$

Hence, the VEM element body force vector is given by

$$\mathbf{f}_{b,E} = |E| \bar{\mathbf{N}}^\top \hat{\mathbf{b}}. \quad (78)$$

The VEM element traction force vector is similar to the VEM element body force vector but the integral is one dimension lower. Therefore, on considering the element edge as a two-node one-dimensional element, the VEM element traction force vector can be computed similarly to the VEM element body force vector, as follows:

$$\mathbf{f}_{f,e} = |e| \overline{\mathbf{N}}_T^\top \hat{\mathbf{f}}, \quad (79)$$

where

$$\overline{\mathbf{N}}_T = \begin{bmatrix} \overline{\phi}_1 & 0 & \overline{\phi}_2 & 0 \\ 0 & \overline{\phi}_1 & 0 & \overline{\phi}_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{N} & 0 & \frac{1}{N} & 0 \\ 0 & \frac{1}{N} & 0 & \frac{1}{N} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \quad (80)$$

and $\hat{\mathbf{f}} = \frac{1}{|e|} \int_e \mathbf{f} \, ds$.

3.9 L^2 -norm and H^1 -seminorm of the error

To assess the accuracy and convergence of the VEM, two global error measures are used. The relative L^2 -norm of the displacement error defined as

$$\frac{\|\mathbf{u} - \Pi_{\mathcal{P}} \mathbf{u}^h\|_{L^2(\Omega)}}{\|\mathbf{u}\|_{L^2(\Omega)}} = \sqrt{\frac{\sum_E \int_E (\mathbf{u} - \Pi_{\mathcal{P}} \mathbf{u}^h)^\top (\mathbf{u} - \Pi_{\mathcal{P}} \mathbf{u}^h) \, d\mathbf{x}}{\sum_E \int_E \mathbf{u}^\top \mathbf{u} \, d\mathbf{x}}}, \quad (81)$$

and the relative H^1 -seminorm of the displacement error given by

$$\frac{\|\mathbf{u} - \Pi_{\mathcal{P}} \mathbf{u}^h\|_{H^1(\Omega)}}{\|\mathbf{u}\|_{H^1(\Omega)}} = \sqrt{\frac{\sum_E \int_E (\varepsilon(\mathbf{u}) - \varepsilon(\Pi_{\mathcal{C}} \mathbf{u}^h))^\top \mathbf{D} (\varepsilon(\mathbf{u}) - \varepsilon(\Pi_{\mathcal{C}} \mathbf{u}^h)) \, d\mathbf{x}}{\sum_E \int_E \varepsilon(\mathbf{u})^\top \mathbf{D} \varepsilon(\mathbf{u}) \, d\mathbf{x}}}, \quad (82)$$

where the strain appears in Voigt notation, and by (28), $\varepsilon(\Pi_{\mathcal{C}} \mathbf{u}^h) = \hat{\varepsilon}(\mathbf{u}^h)$.

3.10 VEM element stiffness matrix for the Poisson problem

The VEM formulation for the Poisson problem is derived similarly to the VEM formulation for the linear elastostatic problem. However, herein we develop the VEM stiffness matrix for the Poisson problem by reducing the solution dimension in the two-dimensional linear elastostatic VEM formulation. The Poisson problem that we consider is the following: consider an open bounded domain $\Omega \subset \mathbb{R}^2$ that is bounded by the one-dimensional surface Γ whose unit outward normal is \mathbf{n}_Γ . The essential (Dirichlet) boundary is denoted by Γ_g . The closure of the domain is $\overline{\Omega} \equiv \Omega \cup \Gamma$. Let $u(\mathbf{x}) : \Omega \rightarrow \mathbb{R}$ be the field variable and $f(\mathbf{x}) : \Omega \rightarrow \mathbb{R}$ be the source term. The imposed essential (Dirichlet) boundary conditions are $g(\mathbf{x}) : \Gamma_g \rightarrow \mathbb{R}$. The boundary-value problem that governs the Poisson problem is: find $u(\mathbf{x}) : \Omega \rightarrow \mathbb{R}$ such that

$$-\nabla^2 u = f \quad \forall \mathbf{x} \in \Omega, \quad (83a)$$

$$u = g \quad \forall \mathbf{x} \in \Gamma_g. \quad (83b)$$

In the two-dimensional linear elastostatic VEM formulation, the following reductions are used: the displacement field reduces to the scalar field $u(\mathbf{x})$, the strain is simplified to $\varepsilon(u) = \nabla u$, the rotations become $\boldsymbol{\omega}(u) = \mathbf{0}$, and the constitutive matrix is replaced by the identity (2×2) matrix. Hence, the VEM projections for the Poisson problem become $\Pi_{\mathcal{R}} u = \bar{u}$ and $\Pi_{\mathcal{C}} u = \hat{\varepsilon}(u) \cdot (\mathbf{x} - \bar{\mathbf{x}})$. The matrices that result from the discretization of the projection operators are simplified to

$$\mathbf{H}_{\mathcal{R}} = [(\mathbf{H}_{\mathcal{R}})_1 \quad \cdots \quad (\mathbf{H}_{\mathcal{R}})_a \quad \cdots \quad (\mathbf{H}_{\mathcal{R}})_N]^\top, \quad (\mathbf{H}_{\mathcal{R}})_a = 1, \quad (84)$$

$$\mathbf{W}_{\mathcal{R}} = [(\mathbf{W}_{\mathcal{R}})_1 \quad \cdots \quad (\mathbf{W}_{\mathcal{R}})_a \quad \cdots \quad (\mathbf{W}_{\mathcal{R}})_N]^T, \quad (\mathbf{W}_{\mathcal{R}})_a = \frac{1}{N}, \quad (85)$$

$$\mathbf{H}_{\mathcal{C}} = [(\mathbf{H}_{\mathcal{C}})_1 \quad \cdots \quad (\mathbf{H}_{\mathcal{C}})_a \quad \cdots \quad (\mathbf{H}_{\mathcal{C}})_N]^T, \quad (\mathbf{H}_{\mathcal{C}})_a = [(x_{1a} - \bar{x}_1) \quad (x_{2a} - \bar{x}_2)], \quad (86)$$

$$\mathbf{W}_{\mathcal{C}} = [(\mathbf{W}_{\mathcal{C}})_1 \quad \cdots \quad (\mathbf{W}_{\mathcal{C}})_a \quad \cdots \quad (\mathbf{W}_{\mathcal{C}})_N]^T, \quad (\mathbf{W}_{\mathcal{C}})_a = [2q_{1a} \quad 2q_{2a}]. \quad (87)$$

On using the preceding matrices, the projection matrix is $\mathbf{P}_{\mathcal{P}} = \mathbf{H}_{\mathcal{R}} \mathbf{W}_{\mathcal{R}}^T + \mathbf{H}_{\mathcal{C}} \mathbf{W}_{\mathcal{C}}^T$ and the final expression for the VEM element stiffness matrix is written as

$$\mathbf{K}_E = |E| \mathbf{W}_{\mathcal{C}} \mathbf{W}_{\mathcal{C}}^T + (\mathbf{I}_N - \mathbf{P}_{\mathcal{P}})^T (\mathbf{I}_N - \mathbf{P}_{\mathcal{P}}), \quad (88)$$

where \mathbf{I}_N is the identity ($N \times N$) matrix and $\mathbf{S}_E = \mathbf{I}_N$ has been used in the stability stiffness as this represents a suitable choice for \mathbf{S}_E in the Poisson problem [34].

4 Object-oriented implementation of VEM in C++

In this section, we introduce **Veamy**, a library that implements the VEM for the linear elastostatic and Poisson problems in two dimensions using object-oriented programming in C++. For the purpose of comparison with the VEM, a module implementing the standard FEM is available within **Veamy** for the solution of the two-dimensional linear elastostatic problem using three-node triangular finite element meshes. In **Veamy**, entities such as elements, degrees of freedom, constraints, among others, are represented by C++ classes.

Veamy uses the following external libraries:

- Triangle [27], which is used to generate a Delaunay triangulation that is later used to construct a Voronoi-based polygonal mesh.
- Clipper [18], an open source freeware library for clipping and offsetting lines and polygons.
- Eigen [16], a C++ template library for linear algebra.

Triangle and Clipper are used in the implementation of **Delynoi** [1], a polygonal mesher that is based on the computation of the constrained Voronoi diagram. The usage of our polygonal mesher is covered in Section 5.

Veamy is free and open source software and is available to be downloaded from its project website (<http://camlab.cl/research/software/veamy/>). The source code is provided in the folder named “Veamy.” A tutorial manual that is aimed to prepare, compile, and run VEM models using **Veamy**, is provided in the folder “Veamy/docs/.” All the source code that implements the VEM is provided in the folder “Veamy/veamy/” and the subfolders therein. Everything that is used by **Veamy** but is external to it, is provided in the folder “Veamy/lib/.” The folder “Veamy/polymesher/” contains a MATLAB function that is intended to be called from **PolyMesher** [31] with the purpose of generating a file containing a **PolyMesher** mesh and boundary conditions that is readable by **Veamy**.

Several tests are located in the folder “Veamy/test/.” Some of these tests are covered in the tutorial manual and in Section 6 of this paper.

The core design of **Veamy** is presented in three UML diagrams that are intended to explain the numerical methods implemented (Fig. 2), the problem conditions inherent to the linear elastostatic and Poisson problems (Fig. 3), and the computation of the L^2 -norm and H^1 -seminorm of the errors (Fig. 4).

4.1 Numerical methods

The **Veamy** library is divided into two modules, one that implements the VEM and another one that implements the FEM. Fig. 2 summarizes the implementation of these methods. Two abstract classes are central to the **Veamy** library, **Calculator2D** and **Element**. **Calculator2D** is designed in the spirit of the controller design pattern. It receives the **ProblemDiscretization** subclasses with all their associated problem conditions, creates the required structures, applies the boundary conditions and runs the simulation. **Calculator2D**, as an abstract class, has a number of methods that all inherited classes must implement. The two most important are the one in charge of creating the elements, and the one in charge of computing the element stiffness matrix and the element (body and traction) force vector. We implement two concrete **Calculator2D** classes, called **Veamer** and **Feamer**, with the former representing the controller for the VEM and the latter for the FEM.

On the other hand, **Element** is the class that encapsulates the behavior of each element in the domain. It is in charge of keeping the degrees of freedom of the element and its associated stiffness matrix and force vector. **Element** contains methods to create and assign degrees of freedom, assemble the element stiffness matrix and the element force vector into the global ones. An **Element** has the information of its defining polygon (the three-node triangle is the lowest-order polygon) along with its degrees of freedom. **Element** has two inherited classes, **VeamyElement** and **FeamyElement**, which represent elements of the VEM and FEM, respectively. They are in charge of the computation of the element stiffness matrix and the element force vector. Algorithm 1 summarizes the implementation of the linear elastostatic VEM element stiffness matrix in the **VeamyElement** class using the notation presented in Sections 3.6 and 3.7.

Algorithm 1 Implementation of the VEM element stiffness matrix for the linear elastostatic problem in the **VeamyElement** class

```

 $H_{\mathcal{R}} = \mathbf{0}, W_{\mathcal{R}} = \mathbf{0}, H_{\mathcal{C}} = \mathbf{0}, W_{\mathcal{C}} = \mathbf{0}$ 
for each node in the polygonal element do
    | Get incident edges
    | Compute the unit outward normal vector to each incident edge
    | Compute  $(H_{\mathcal{R}})_a$  and  $(H_{\mathcal{C}})_a$ , and insert them into  $H_{\mathcal{R}}$  and  $H_{\mathcal{C}}$ , respectively
    | Compute  $(W_{\mathcal{R}})_a$  and  $(W_{\mathcal{C}})_a$ , and insert them into  $W_{\mathcal{R}}$  and  $W_{\mathcal{C}}$ , respectively
end
Compute  $I_{2N}, P_{\mathcal{R}}, P_{\mathcal{C}}, P_{\mathcal{P}}, D$ 
Compute  $S_E$ 
Output:  $K_E = |E| W_{\mathcal{C}} D W_{\mathcal{C}}^T + (I_{2N} - P_{\mathcal{P}})^T S_E (I_{2N} - P_{\mathcal{P}})$ 
    
```

The element force vector is computed with the aid of the abstract classes **BodyForceVector** and **TractionVector**. Each of them has two concrete subclasses named **VeamyBodyForceVector** and **FeamyBodyForceVector**, and **VeamyTractionVector** and **FeamyTractionVector**, respectively.

Even though we have implemented the three-node triangular finite element only as a means to comparison with VEM, we decided to define **FeamyElement** as an abstract class so that more advanced elements can be implemented if desired. Finally, each **FeamyElement** concrete implementation has a **ShapeFunction** concrete subclass, representing the shape functions that are used to interpolate the solution inside the element. For the three-node triangular finite element, we include the **Tri3ShapeFunctions** class.

One of the structures related to all **Element** classes is called **DOF**. It describes a single degree of freedom. The degree of freedom is associated with the nodal points of the mesh according to the **ProblemDiscretization** subclasses. So, in the linear elastostatic problem each nodal point has two

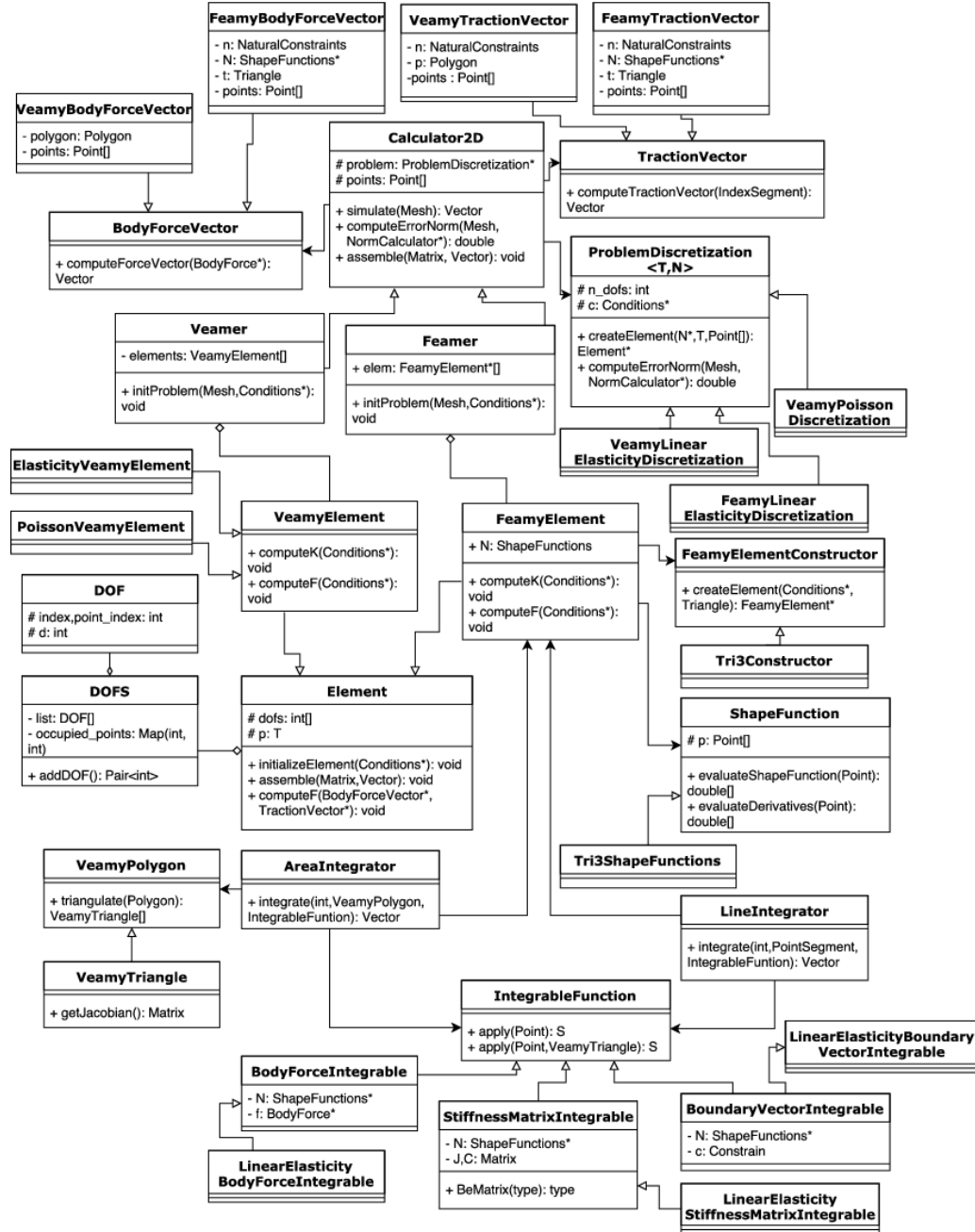


Fig. 2 UML diagram for the Veamy library. VEM and FEM modules

associated **DOF** instances and in the Poisson problem just one **DOF** instance. The **DOF** instances are kept in a list inside a container class called **DOFS**.

Although the VEM matrices are computed algebraically, the FEM matrices in general require numerical integration both inside the element (area integration) and on the edges that lie on the natural boundary (line integration). Thus, we have implemented two classes, **AreaIntegrator** and **LineIntegrator**, which contain methods that integrate a given function inside the element and on its boundary. There are several classes related to the numerical integration. **IntegrableFunction** is a template interface that has a method called **apply** that must be implemented. This method receives a sample point and must be implemented so that it returns the evaluation of a function at the sample point. We include three concrete **IntegrableFunction** implementations, one for the body force, another one for the stiffness matrix and the last one for the boundary vector.

4.2 Problem conditions

Fig. 3 presents the classes for the problem conditions used in the linear elastostatic and Poisson problems. The problem conditions are kept in a structure called **Conditions** that contains the physical properties of the material (**Material** class), the boundary conditions and the body force. **BodyForce** is a class that contains two functions pointers that represent the body force in each of the two axes of the Cartesian coordinate system. These two functions must be instantiated by the user to include a body force in the problem. By default, **Conditions** creates an instance of the **None** class, which is a subclass of **BodyForce** that represents the nonexistence of body forces. **Material** is an abstract class that keeps the elastic constants associated with the material properties (Young's modulus and Poisson's ratio) and has an abstract function that computes the material matrix; **Material** has two subclasses, **MaterialPlaneStress** and **MaterialPlaneStrain**, which return the material matrix for the plane stress and plane strain states, respectively.

To model the boundary conditions, we have created a number of classes: **Constraint** is an abstract class that represents a single constraint — a constraint can be an essential (Dirichlet) boundary condition or a natural (Neumann) boundary condition. **PointConstraint** and **SegmentConstraint** are concrete classes implementing **Constraint** and representing a constraint at a point and on a segment of the domain, respectively. **Constraints** is the class that manages all the constraints in the system and the relationship between them and the degrees of freedom; **EssentialConstraints** and **NaturalConstraints** inherit from **Constraints**. Finally, **ConstraintsContainers** is a utility class that contains **EssentialConstraints** and **NaturalConstraints** instances. **Constraint** keeps a list of domain segments subjected to a given condition, the value of this condition, and a certain direction (vertical, horizontal or both). The interface called **ConstraintValue** is the method to control the way the user inputs the constraints: to add any constraint, the user must choose between a constant value (**Constant** class) and a function (**Function** class), or implement a new class inheriting from **ConstraintValue**.

4.3 Norms of the error

As shown in Fig. 4, **Veamy** provides functionalities for computing the relative L^2 -norm and H^1 -seminorm of the error through the classes **L2NormCalculator** and **H1NormCalculator**, which inherit from the abstract class **NormCalculator**. Each **NormCalculator** instance has two instances of what we call the **NormIntegrator** classes, **VeamyIntegrator** and **FeamyIntegrator**. These are in charge of integrating the norms integrals in the VEM and FEM approaches, respectively. In these

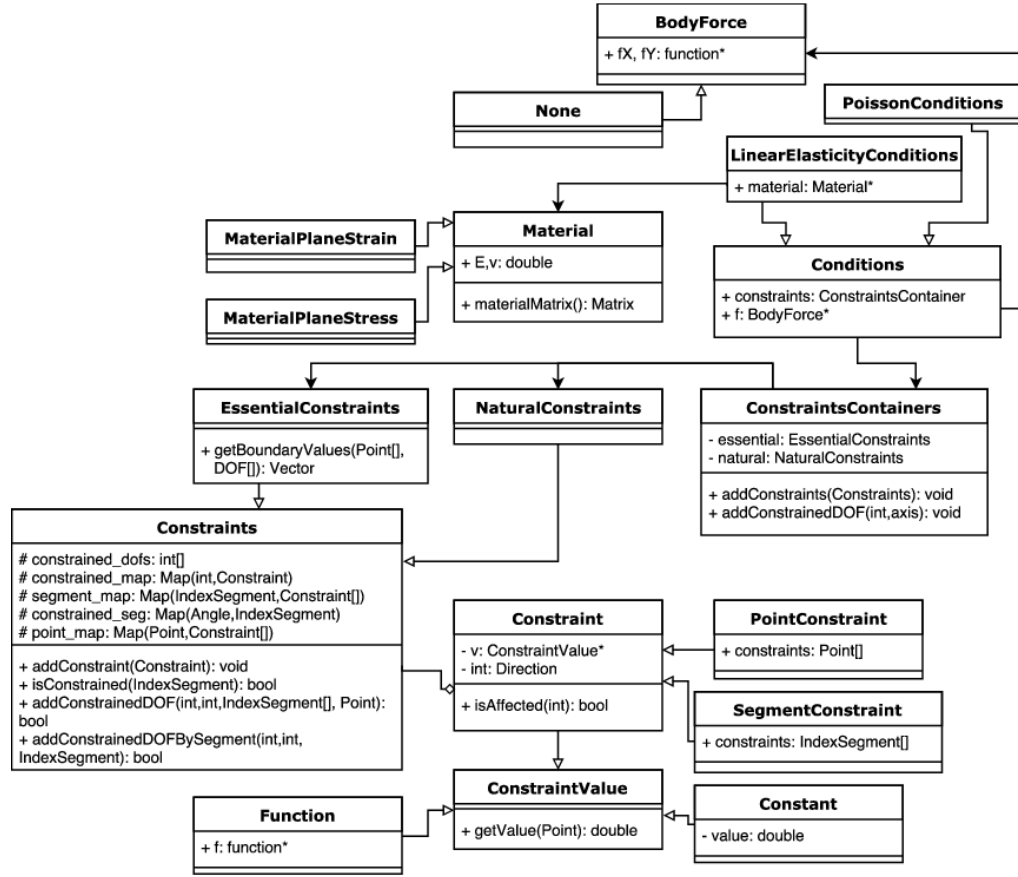


Fig. 3 UML diagram for the Veamy library. Problem conditions

NormIntegrator classes, the integrands of the norms integrals are represented by the Computable class. Depending on the integrand, we define various Computable subclasses: DisplacementComputable, DisplacementDifferenceComputable, H1Computable and its subclasses, StrainDifferenceComputable, StrainStressDifferenceComputable, StrainComputable and StrainStressComputable. Finally, DisplacementCalculator and StrainCalculator (and their subclasses) permit to obtain the numerical displacement and the numerical strain, respectively, and StrainValue and StressValue classes represent the exact value of the strains and stresses at the integration points, respectively.

4.4 Computation of nodal displacements

Each simulation is represented by a single Calculator2D instance, which is in charge of conducting the simulation through its `simulate` method until the displacement solution is obtained. The procedure

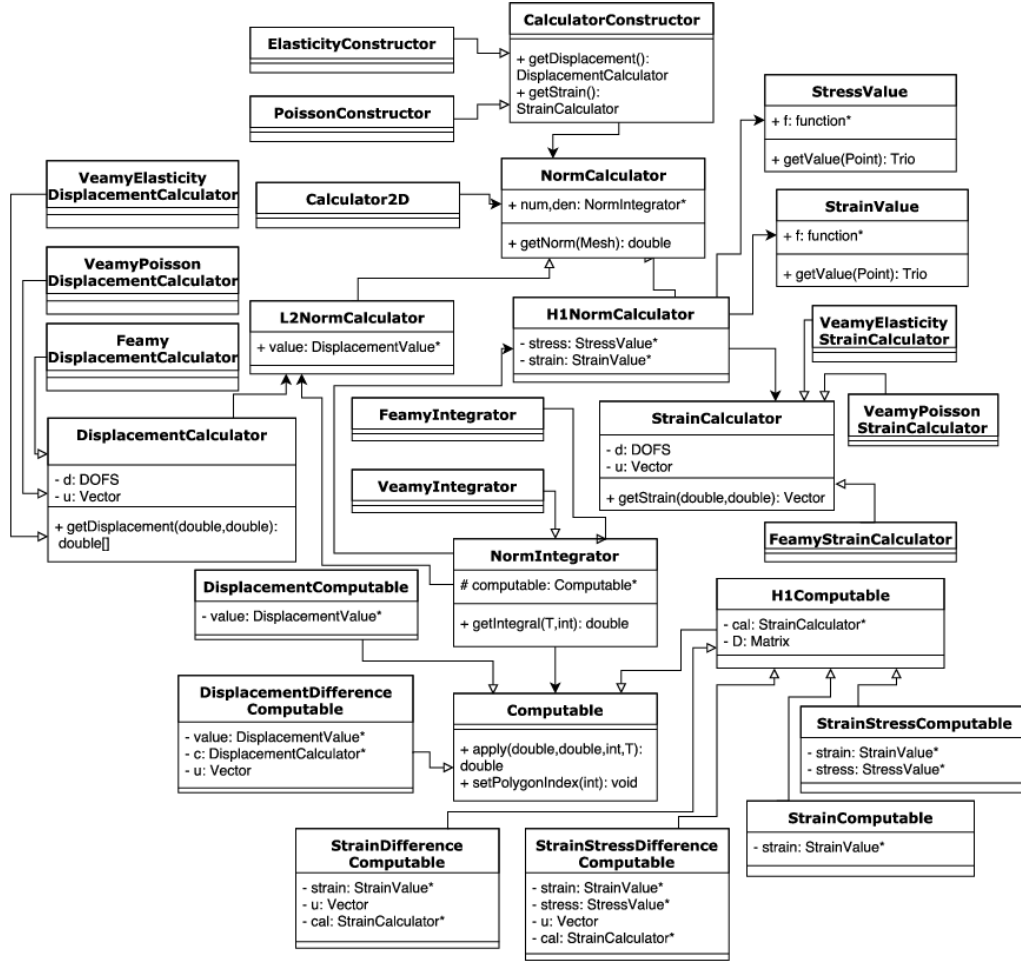


Fig. 4 UML diagram for the Veamy library. Computation of the L^2 -norm and H^1 -seminorm of the error

is similar to a finite element simulation. The implementation of the `simulate` method is summarized in Algorithm 2.

Algorithm 2 Implementation of the `simulate` method in the `Calculator2D` class

Input: Mesh

Initialization of the global stiffness matrix and the global force vector

for each element in the mesh **do**

 Compute the element stiffness matrix

 Compute the element force vector

 Assemble the element stiffness matrix and the element force vector into global ones

end

Apply natural boundary conditions to the global force vector

Impose the essential boundary conditions into the global matrix system

Solve the resulting global matrix system of linear equations

Output: Column vector containing the nodal displacements solution

The resulting matrix system of linear equations is solved using appropriate solvers available in the Eigen library [16] for linear algebra.

5 Polygonal mesh generator

In this section, we provide some guidelines for the usage of our polygonal mesh generator **Delynoi** [1].

5.1 Domain definition

The domain is defined by creating its boundary from a counterclockwise list of points. Some examples of domains created in **Delynoi** are shown in Fig. 5. We include the possibility of adding internal or intersecting holes to the domain as additional objects that are independent of the domain boundary. Some examples of domains created in **Delynoi** with one and several intersecting holes are shown in Fig. 6.

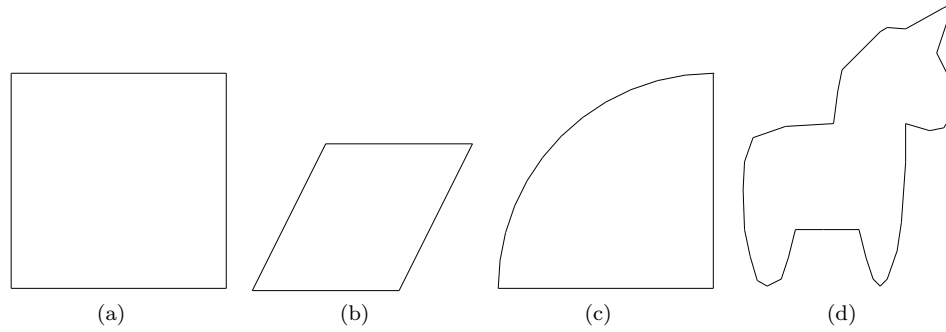


Fig. 5 Domain examples. (a) Square domain, (b) rhomboid domain, (c) quarter circle domain, (d) unicorn-shaped domain

Listing 1 shows the code to generate a square domain and a quarter circle domain. More domain definitions are given in Section 6 as part of **Veamy**'s sample usage problems.

```

1  std::vector<Point> square_points = {Point(0,0), Point(10,0), Point(10,10), Point(0,10)};
2  Region square(square_points);
3  std::vector<Point> qc_points = {Point(0,0), Point(10,0), Point(10,10)};
4  std::vector<Point> quarter = delynoi_utilities::generateArcPoints(Point(10,0), 10, 90.0, 180.0);
5  qc_points.insert(quarter_circle_points.end(), quarter.begin(), quarter.end());
6  Region quarter_circle(qc_points);

```

Listing 1 Definition of square and quarter circle domains

To add a circular hole to the center of the square domain already defined, first the required hole is created and then added to the domain as shown in Listing 2.

```

1  Hole circular = CircularHole(Point(5,5), 2);
2  square.addHole(circular);

```

Listing 2 Adding a circular hole to the center of the square domain

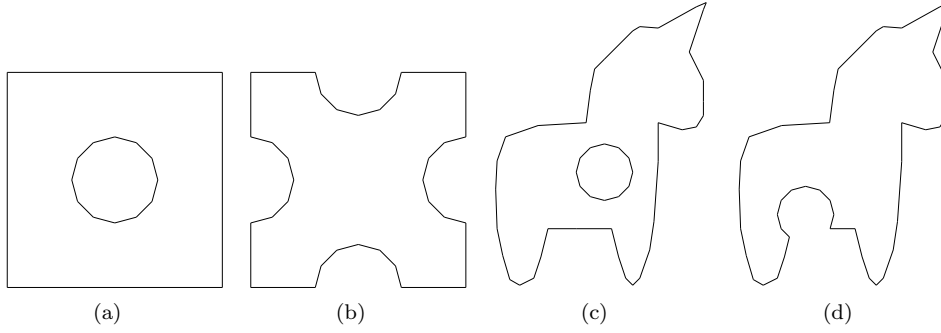


Fig. 6 Examples of domains with holes. (a) Square with an inner hole, (b) square with four intersecting holes, (c) unicorn-shaped domain with an inner hole, (d) unicorn-shaped domain with an intersecting hole

5.2 Mesh generation rules

We include a number of different rules for the generation of the seeds points for the Voronoi diagram. These rules are `constant`, `random_double`, `ConstantAlternating` and `sine`. The `constant` method generates uniformly distributed seeds points; the `random_double` method generates random seeds points; the `ConstantAlternating` method generates seeds points by displacing alternating the points along one Cartesian axis. Fig. 7 presents some examples of meshes generated on a square domain using different rules. We show how to generate constant (uniform) and random points for a given domain in Listing 3.

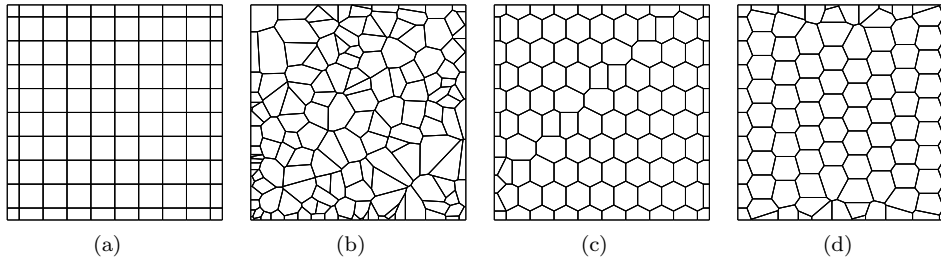


Fig. 7 Polygonal mesh generation on a square domain using different rules. (a) `constant`, (b) `random_double`, (c) `ConstantAlternating`, (d) `sine`

```

1 dom1.generateSeedPoints(PointGenerator(functions::constant(), functions::constant()), nX, nY);
2 dom2.generateSeedPoints(PointGenerator(functions::random_double(0,maxX), functions::random_double(0,maxY)), nX,
  nY);
3 // nX, nY: horizontal and vertical divisions along sides of the bounding box

```

Listing 3 Generation of constant (uniform) and random points

We also include the possibility of adding noise to the generation rules. For this, we implement a random noise function that adds a random displacement to each seed point. Fig. 8 depicts some examples of generation rules with random noise. Listing 4 presents a representative code to add random noise to the **constant** generation rule on a square domain.

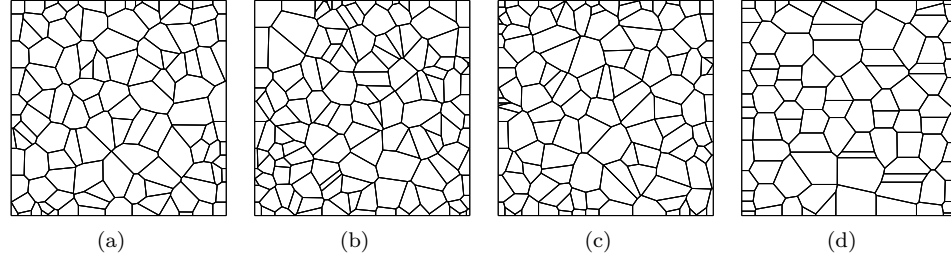


Fig. 8 Polygonal mesh generation on a square domain using different rules with random noise. (a) **constant** with noise, (b) **random_double** with noise, (c) **ConstantAlternating** with noise, (d) **sine** with noise

```

1 Functor* n = noise::random_double_noise(functions::constant(), minNoise, maxNoise);
2 square.generateSeedPoints(PointGenerator(n,n,nX, nY));
3 // nX, nY: horizontal and vertical divisions along sides of the bounding box

```

Listing 4 Generation of constant (uniform) points with random noise

5.3 Mesh generation on complicated domains

Finally, we present some examples of meshes generated on some complicated domains using **constant** and **random_double** rules. Fig. 9 shows polygonal meshes for a square domain with four intersecting holes and Fig. 10 depicts polygonal meshes for the unicorn-shaped domain without holes and with different configuration of holes.

6 Sample usage

This section illustrates the usage of **Veamy** through several examples. For each example, a main C++ file is written to setup the problem. This is the only file that needs to be written by the user in order to run a simulation in **Veamy**. All the setup files for the examples that are considered in this section are included in the folder “Veamy/test/.” To be able to run these examples, it is necessary to compile the source code. A tutorial manual that provides complete instructions on how to prepare, compile and run the examples is included in the folder “Veamy/docs/.”

6.1 Cantilever beam subjected to a parabolic end load

The VEM solution for the displacement field on a cantilever beam of unit thickness subjected to a parabolic end load P is computed using **Veamy**. Fig. 11 illustrates the geometry and boundary

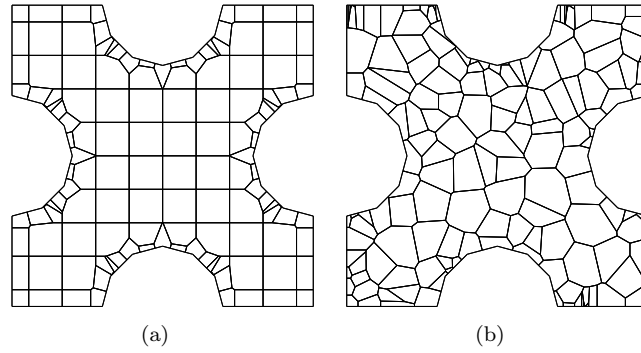


Fig. 9 Examples of polygonal meshes in complicated domains. (a) Square with four intersecting holes and **constant** generation rule, and (b) square with four intersecting holes and **random_double** generation rule

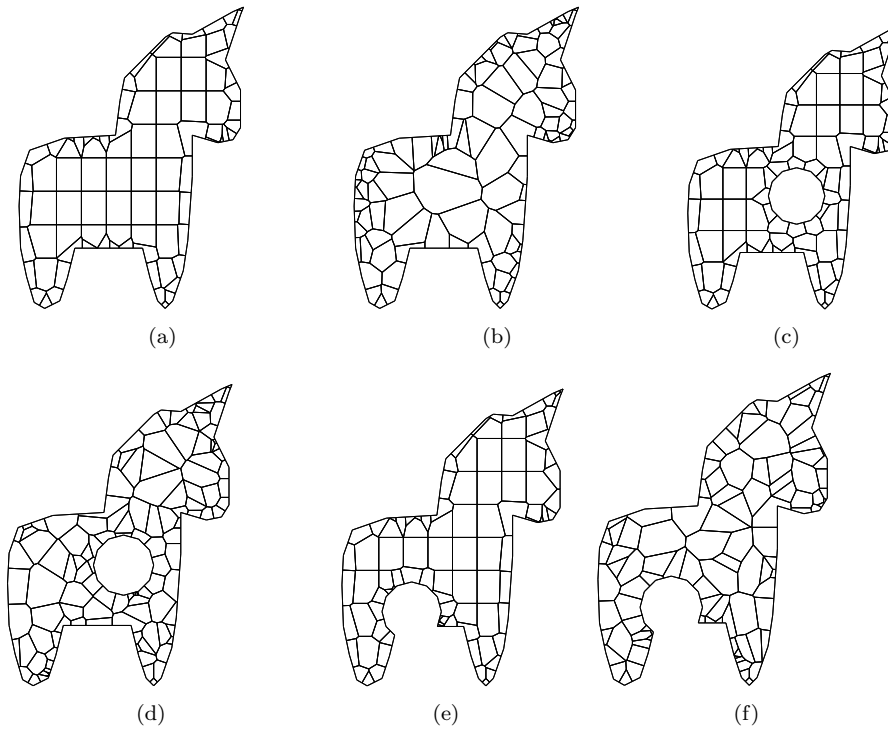


Fig. 10 Examples of polygonal meshes in complicated domains. Unicorn-shaped domain with (a) **constant** generation rule, (b) **random_double** generation rule, (c) inner hole and **constant** generation rule, (d) inner hole and **random_double** generation rule, (e) intersecting hole and **constant** generation rule, and (f) intersecting hole and **random_double** generation rule

conditions. Plane strain state is assumed. The essential boundary conditions on the clamped edge are applied according to the analytical solution given by Timoshenko and Goodier [33]:

$$u_x = -\frac{Py}{6\bar{E}_Y I} \left((6L - 3x)x + (2 + \bar{\nu})y^2 - \frac{3D^2}{2}(1 + \bar{\nu}) \right),$$

$$u_y = \frac{P}{6\bar{E}_Y I} \left(3\bar{\nu}y^2(L - x) + (3L - x)x^2 \right),$$

where $\bar{E}_Y = E_Y / (1 - \nu^2)$ with the Young's modulus set to $E_Y = 1 \times 10^7$ psi, and $\bar{\nu} = \nu / (1 - \nu)$ with the Poisson's ratio set to $\nu = 0.3$; $L = 8$ in. is the length of the beam, $D = 4$ in. is the height of the beam, and I is the second-area moment of the beam section. The total load on the traction boundary is $P = -1000$ lbf.

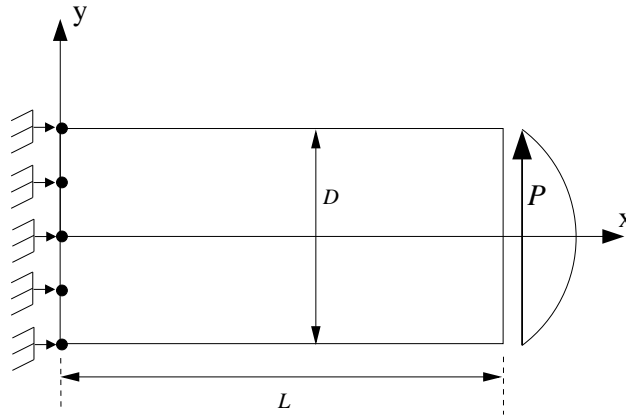


Fig. 11 Model geometry and boundary conditions for the cantilever beam problem

6.1.1 Setup file

A main C++ file is written to setup the problem. As there are different aspects to consider, we divide the setup file in several blocks and explain each of them. Herein only the main parts of this setup file are described. The complete setup instructions for this problem are provided in the file “ParabolicMain.cpp” that is located in the folder “Veamy/test/.”

Listing 5 shows the definition of the problem domain, the generation of base points for the Voronoi diagram, and the computation of the polygonal mesh.

```

1  std::vector<Point> rectangle_points = {Point(0, -2), Point(8, -2), Point(8, 2), Point(0, 2)};
2  Region rectangle(rectangle_points);
3  rectangle.generateSeedPoints(PointGenerator(functions::constantAlternating(), functions::constant()), 24, 12);
4  std::vector<Point> seeds = rectangle.getSeedPoints();
5  TriangleVoronoiGenerator meshGenerator (seeds, rectangle);
6  Mesh<Polygon> mesh = meshGenerator.getMesh();

```

Listing 5 Domain definition and mesh generation for the beam subjected to a parabolic end load

We proceed to initialize all the structures needed to represent the conditions of the problem at hand. In first place, an object of the `Material` class is created and used to initialize an object of the class `LinearElasticityConditions`. This is shown in Listing 6.

```

1 Material* material = new MaterialPlaneStrain (1e7, 0.3);
2 LinearElasticityConditions* conditions = new LinearElasticityConditions(material);

```

Listing 6 Definition of the elastic material and initialization of the problem conditions

We create a constraint that represents the essential boundary condition that is imposed on the left side of the beam, including the segment it affects, the value of the constraint and the direction (in the Cartesian coordinate system) in which the constraint is imposed. This implementation is shown in Listing 7.

```

1 double uX(double x, double y){
2     double P = -1000;
3     double Ebar = 1e7/(1 - std::pow(0.3,2));
4     double vBar = 0.3/(1 - 0.3);
5     double D = 4; double L = 8; double I = std::pow(D,3)/12;
6     return -P*y/(6*Ebar*I)*((6*L - 3*x)*x + (2+vBar)*std::pow(y,2) - 3*std::pow(D,2)/2*(1+vBar));
7 }
8
9 double uY(double x, double y){
10    double P = -1000;
11    double Ebar = 1e7/(1 - std::pow(0.3,2));
12    double vBar = 0.3/(1 - 0.3);
13    double D = 4; double L = 8; double I = std::pow(D,3)/12;
14    return P/(6*Ebar*I)*(3*vBar*std::pow(y,2)*(L-x) + (3*L-x)*std::pow(x,2));
15 }
16 Function* uXConstraint = new Function(uX);
17 Function* uYConstraint = new Function(uY);
18 PointSegment leftSide(Point(0,-2), Point(0,2));
19 SegmentConstraint const1 (leftSide, mesh.getPoints(), uXConstraint);
20 SegmentConstraint const2 (leftSide, mesh.getPoints(), uYConstraint);
21 conditions->addEssentialConstraint(const1, mesh.getPoints(), elasticity_constraints::Direction::Horizontal);
22 conditions->addEssentialConstraint(const2, mesh.getPoints(), elasticity_constraints::Direction::Vertical);

```

Listing 7 Definition of the essential boundary condition on the left side of the beam

Listing 8 presents the implementation of the natural boundary condition (the parabolic load) that is applied on the right side of the beam. The parabolic load is constructed using a function called `tangencial`.

```

1 double tangencial(double x, double y){
2     double P = -1000; double D = 4;
3     double I = std::pow(D,3)/12;
4     double value = std::pow(D,2)/4-std::pow(y,2);
5     return P/(2*I)*value;
6 }
7 Function* tangencialLoad = new Function(tangencial);
8 PointSegment rightSide(Point(8,-2), Point(8,2));
9 SegmentConstraint const3 (rightSide, mesh.getPoints(), tangencialLoad);
10 conditions->addNaturalConstraint(const3, mesh.getPoints(), elasticity_constraints::Direction::Vertical);

```

Listing 8 Definition of the natural boundary condition on the right side of the beam

The linear elastostatic problem is initialized with the problem conditions previously defined by creating an object of the class `VeamyLinearElasticityDiscretization`. And the latter along with the mesh is used to initiate a `Veamer` instance that represents the system. Finally, to obtain the nodal displacements solution the `simulate` method is invoked. These instructions are presented in Listing 9.

```

1 VeamyLinearElasticityDiscretization* problem = new VeamyLinearElasticityDiscretization(conditions);
2 Veamer v(problem);
3 v.initProblem(mesh);
4 Eigen::VectorXd displacements = v.simulate(mesh);

```

Listing 9 Initialization of the system that represents the beam subjected to a parabolic end load and start of the simulation

The output of the `simulate` method is a column vector that contains the nodal displacements solution. To print the nodal displacements solution to an output file, the `writeDisplacements` method is called after the `simulate` method. This is shown in Listing 10. The resulting text file is named as the string stored in `displacementsFileName`. The text file contains the computed displacements in the following format: nodal index, x-displacement and y-displacement. An extract of the output file generated for the beam subjected to a parabolic end load is shown in Listing 11.

```

1 v.writeDisplacements(displacementsFileName, displacements);

```

Listing 10 Printing of nodal displacements solution to an output file

1	0	9.38002e-005	-0.000100889
2	1	0.000137003	-0.000101589
3	2	9.30384e-005	-0.000115664
4	...		

Listing 11 Extract of the output file for the beam subjected to a parabolic end load

The output file contains no information about the geometry of the problem. The geometry information is kept in the `Mesh` instance created at the beginning of the example. `Mesh` includes a method to print its geometrical data to a text file with a single line of code, as shown in Listing 12.

```

1 mesh.printInFile(meshFileName);

```

Listing 12 Printing of mesh data to a text file

The text file containing the mesh information is named as the string stored in `meshFileName` and is arranged in the following format:

- First line: number of nodal points in the polygonal mesh.
- Following lines: x-coordinate y-coordinate for each nodal point in the mesh.
- One line: number of element edges in the polygonal mesh.
- Following lines: index-of-start-point index-of-end-point for each element edge in the polygonal mesh.
- One line: number of elements in the polygonal mesh.
- Following lines: number-of-element-nodes list-of-nodal-indexes centroid-x-coordinate centroid-y-coordinate for each polygon in the mesh.

6.1.2 Post processing

`Veamy` does not provide a post processing interface. The user may opt for a post processing interface of their choice. Here we visualize the displacement results using a MATLAB function written for this purpose. This MATLAB function is provided in the folder “`Veamy/lib/visualization/`” as the file “`plotPolyMeshDisplacements.m`.” In addition, a file named “`plotPolyMesh.m`” that serves for plotting

the mesh is provided in the same folder. Fig. 12 presents the polygonal mesh used and the VEM solutions.

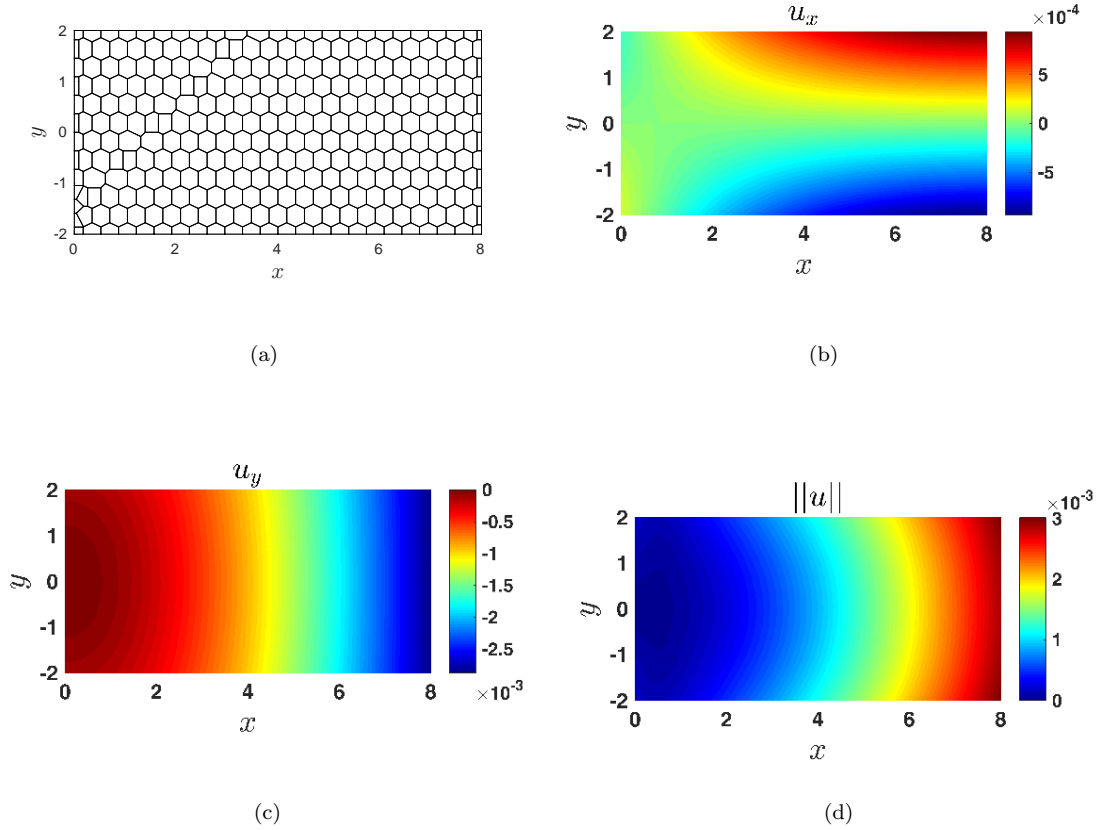


Fig. 12 Solution for the cantilever beam subjected to a parabolic end load using **Veamy**. (a) Polygonal mesh, (b) VEM horizontal displacement, (c) VEM vertical displacement, (d) norm of the VEM displacement

6.1.3 VEM performance

A performance comparison between VEM and FEM is conducted. For the FEM simulations, the **Feamy** module is used. The main C++ setup files for these tests are located in the folder “Veamy/test/” and named as “ParabolicMainVEM.cpp” for the VEM and “ParabolicMainT3.cpp” for the FEM using three-node triangles ($T3$). The performance of the two methods are compared in Fig. 13, where the H^1 -seminorm of the error and the normalized CPU time are each plotted as a function of the number of degrees of freedom (DOF). The normalized CPU time is defined as the ratio of the CPU time

of a particular model analyzed to the maximum CPU time found for any of the models analyzed. From Fig. 13 it is observed that for equal number of degrees of freedom both methods deliver similar accuracy and the computational costs are about the same as the mesh is refined.

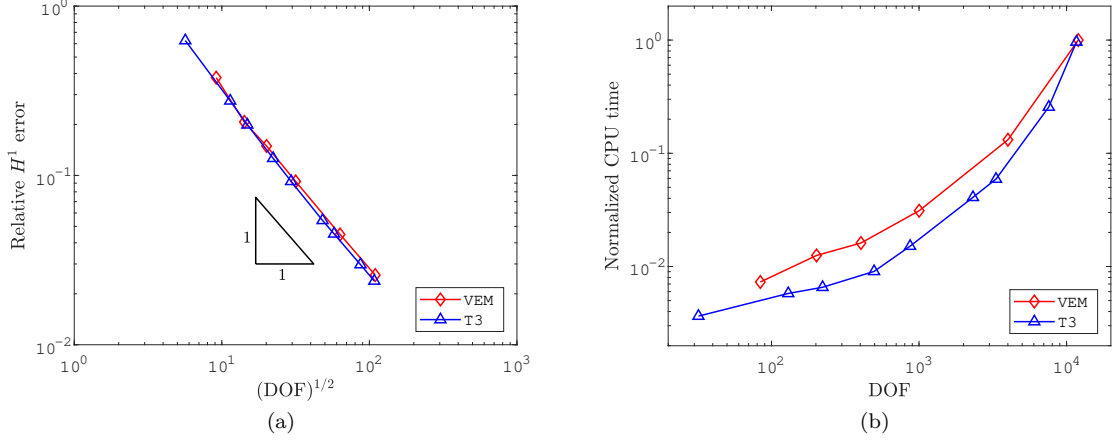


Fig. 13 Cantilever beam subjected to a parabolic end load. Performance comparison between the VEM (polygonal elements) and the FEM (three-node triangles (T3)). (a) H^1 -seminorm of the error as a function of the number of degrees of freedom and (b) normalized CPU time as a function of the number of degrees of freedom

6.2 Displacement patch test

This test consists in the solution of the linear elastostatic problem with $\mathbf{b} = \mathbf{0}$ and essential (Dirichlet) boundary conditions $\mathbf{g} = [x_1 \quad x_1 + x_2]^T$ imposed along the entire boundary of a unit square domain. Plane strain condition is assumed with the following material parameters: $E_Y = 1 \times 10^7$ psi and $\nu = 0.3$. The complete setup instructions for this problem are provided in the file “Displacement-PatchTestMain.cpp” that is located in the folder “Veamy/test/.” The polygonal mesh and the VEM results are shown in Fig. 14. The relative L^2 -norm of the error and the relative H^1 -seminorm of the error obtained for the mesh shown in Fig. 14(a) are 2.45575×10^{-16} and 1.0977×10^{-15} , respectively. Therefore, as predicted by the theory, the VEM solution coincides with the exact solution given by \mathbf{g} within machine precision.

6.3 Using a PolyMesher mesh

A polygonal mesher that is widely used in the VEM and polygonal finite elements communities is PolyMesher [31]. PolyMesher delivers a polygonal mesh with nodal displacements constraints and prescribed nodal loads. It is coded in MATLAB.

In order to conduct a simulation in Veamy using data stemming from PolyMesher, a MATLAB function named PolyMesher2Veamy, which needs to be called from PolyMesher, was especially devised

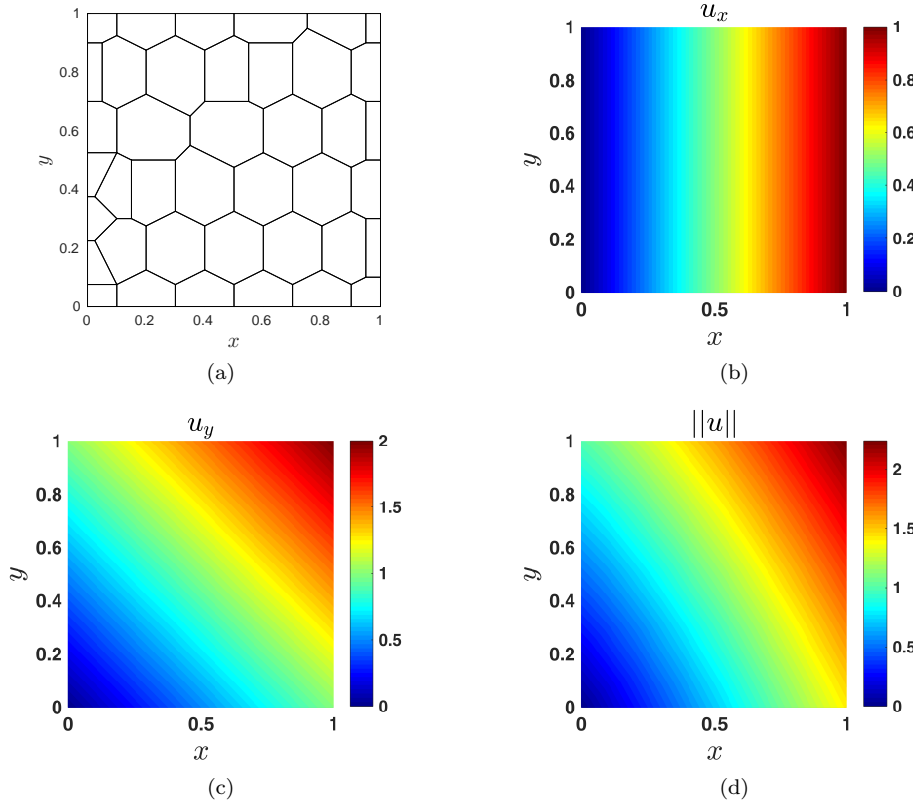


Fig. 14 Solution for the displacement patch test using **Veamy**. (a) Polygonal mesh, (b) VEM horizontal displacement, (c) VEM vertical displacement, and (d) norm of the VEM displacement. The relative L^2 -norm of the error is 2.45575×10^{-16} and the relative H^1 -seminorm of the error is 1.0977×10^{-15}

to read these data and write them to a text file in a format that is readable by **Veamy**. This MATLAB function is provided in the folder “**Veamy/polymesher/**.” Function **PolyMesher2Veamy** receives five **PolyMesher**’s data structures (**Node**, **Element**, **NElem**, **Supp**, **Load**) and writes a text file containing the mesh and boundary conditions. **Veamy** implements a function named **initProblemFromFile** that is able to read this text file and solve the problem straightforwardly.

As a demonstration of the potential that is offered to the simulation when **Veamy** interacts with **PolyMesher**, the MBB beam problem of Section 6.1 in Ref. [31] is considered. The MBB problem is shown in Fig. 15, where $L = 3$ in., $D = 1$ in. and $P = 0.5$ lbf. The following material parameters are considered: $E_Y = 1 \times 10^7$ psi, $\nu = 0.3$ and plane strain condition is assumed. The MATLAB function **PolyMesher2Veamy** is used to read the polygonal mesh and boundary conditions from **PolyMesher** and write them to the text file “**polymesher2veamy.txt**.” This text file is provided in the folder “**Veamy/test/test.files/**.” The complete setup instructions for this problem are provided in the file “**PolyMesherMain.cpp**” that is located in the folder “**Veamy/test/**.” The polygonal mesh and the VEM solutions that are obtained using **Veamy** are presented in Fig. 16.

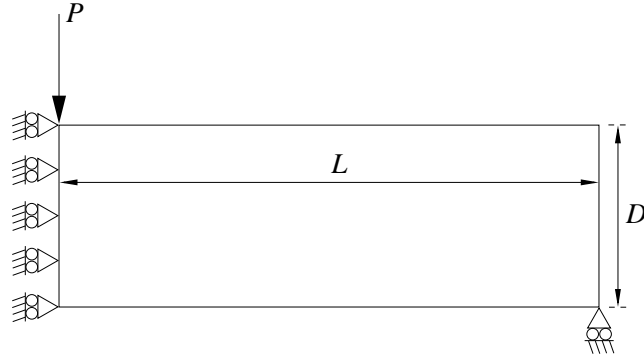


Fig. 15 MBB beam problem definition as per Section 6.1 in Ref. [31]

6.4 Perforated Cook's membrane

In this example, a perforated Cook's membrane is considered. The objective of this problem is to show more advanced domain definitions and mesh generation capabilities offered by **Veamy**. The complete setup instructions for this problem are provided in the file “CookTestMain.cpp” that is located in the folder “Veamy/test/.” The following material parameters are considered: $E_Y = 250$ MPa, $\nu = 0.3$ and plane strain condition is assumed. The model geometry, polygonal mesh and boundary conditions, and the VEM solutions are presented in Fig. 17.

6.5 A toy problem

A toy problem consisting of a unicorn loaded on its back and fixed at its feet is modeled and solved using **Veamy**. The objective of this problem is to show additional capabilities for domain definition and mesh generation that are available in **Veamy**. The complete setup instructions for this problem are provided in the file “UnicornTestMain.cpp” that is located in the folder “Veamy/test/.” The following material parameters are considered: $E_Y = 1 \times 10^4$ psi, $\nu = 0.25$ and plane strain condition is assumed. The model geometry, polygonal mesh and boundary conditions, and the VEM solutions are shown in Fig. 18.

6.6 Poisson problem with a manufactured solution

We conclude the examples by solving a Poisson problem with a source term given by $f(\mathbf{x}) = 32y(1 - y) + 32x(1 - x)$, which is the outcome of letting the solution field be $u(\mathbf{x}) = 16xy(1 - x)(1 - y)$. A unit square domain is considered and $u(\mathbf{x})$ is imposed along the entire boundary of the domain resulting in the essential (Dirichlet) boundary condition $g(\mathbf{x}) = 0$. The complete setup instructions for this problem are provided in the file “PoissonSourceTestMain.cpp” that is located in the folder “Veamy/test/.” The polygonal mesh and the VEM solution are shown in Fig. 19. The relative L^2 -norm of the error and the relative H^1 -seminorm of the error obtained for the mesh shown in Fig. 19(a) are 2.6695×10^{-3} and 6.7834×10^{-2} , respectively.

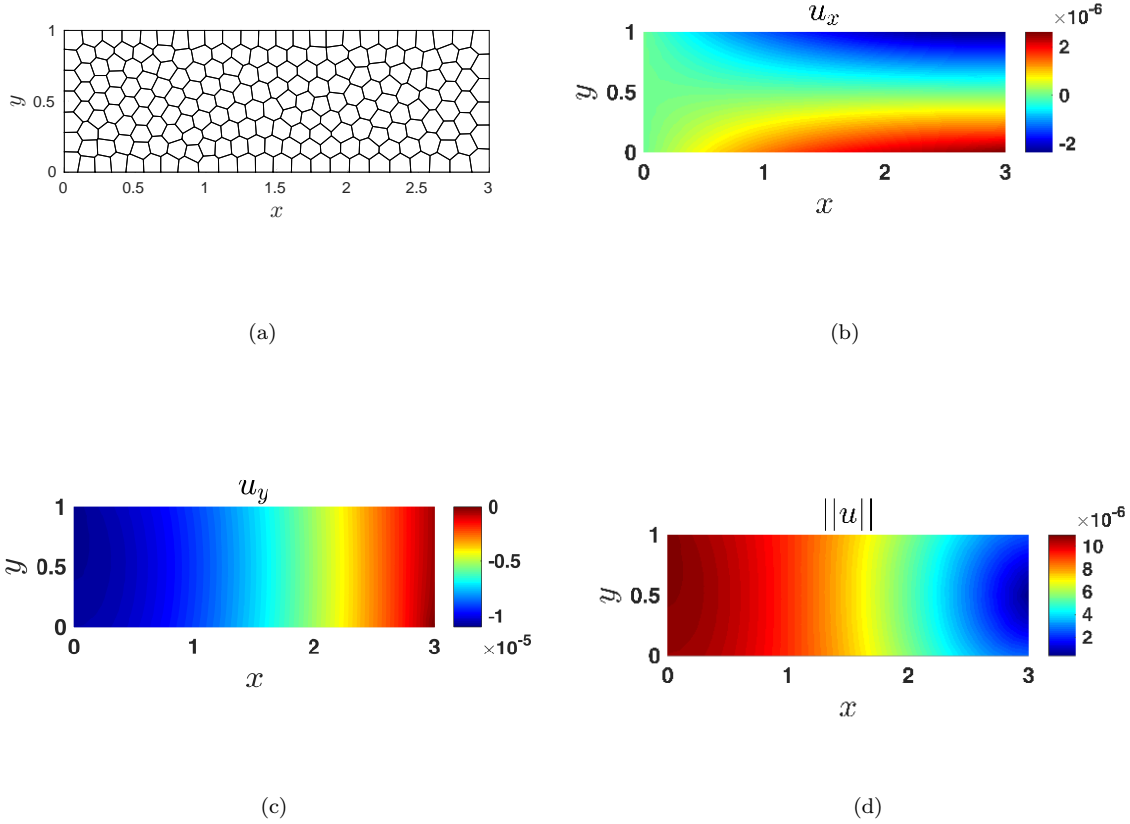


Fig. 16 Solution for the MBB beam problem using **Veamy**. (a) Polygonal mesh, (b) VEM horizontal displacement, (c) VEM vertical displacement, and (d) norm of the VEM displacement

7 Concluding remarks

In this paper, an object-oriented programming of the virtual element method for the linear elastostatic and Poisson problems in two dimensions was presented. As a result, a C++ VEM library named **Veamy** was developed. A stepwise sample usage of this library, which consisted in the solution of a cantilever beam subjected to a parabolic end load, was provided. This problem also served for the purpose of demonstrating that the VEM using polygonal elements performs similar to the FEM using three-node triangular elements. **Veamy** was also tested on a displacement patch test and a demonstration of its interaction with **PolyMesher** was featured. In just a few lines of code, **Veamy** can straightforwardly solve any linear elastostatic problem using a **PolyMesher** mesh with boundary conditions. In addition, a perforated Cook's membrane and a toy problem consisting of a unicorn loaded on its back and fixed at its feet, were presented to demonstrate more advanced capabilities available in **Veamy** for

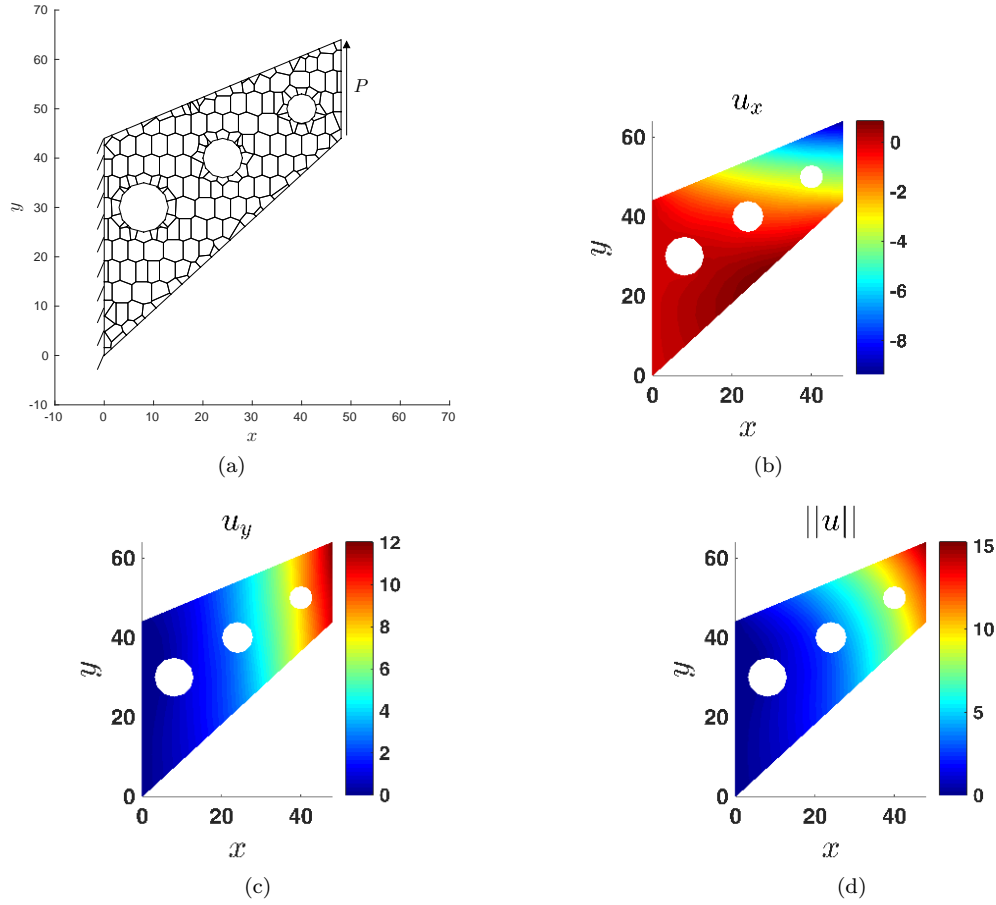


Fig. 17 Solution for the perforated Cook's membrane problem using **Veamy**. (a) Model geometry, polygonal mesh and boundary conditions, (b) VEM horizontal displacement, (c) VEM vertical displacement, and (d) norm of the VEM displacement

domain definition and polygonal mesh generation. The examples section ended with the simulation of a Poisson problem with a manufactured solution. To the best of our knowledge, **Veamy** is the first object-oriented C++ implementation of the VEM. Possible extensions of this library that are of interest include three-dimensional linear elastostatics, where an interaction with the polyhedral mesh generator Voropp [26] seems very appealing and, although recently introduced, nonlinear solid mechanics problems [3, 8, 37, 38]. We have made **Veamy** a free and open source software.

Acknowledgements AOB acknowledges the support provided by Universidad de Chile through the “Programa VID Ayuda de Viaje 2017.” The work of CA is supported by CONICYT-PCHA/Magíster Nacional/2016-22161437. NHK is grateful for the support provided by Proyecto Enlace VID 009/15.

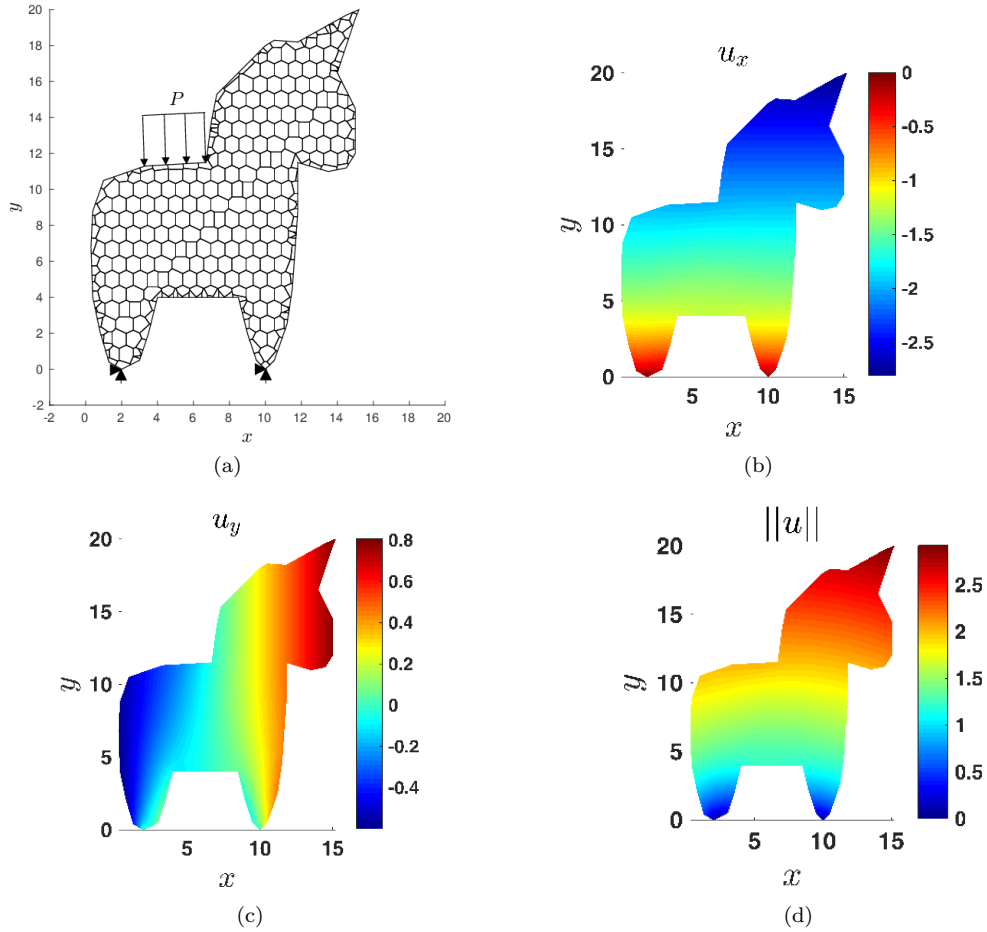


Fig. 18 Solution for the toy problem using **Veamy**. (a) Model geometry, polygonal mesh and boundary conditions, (b) VEM horizontal displacement, (c) VEM vertical displacement, and (d) norm of the VEM displacement

References

1. Delynoi v1.0. <http://camlab.cl/research/software/delynoi/>
2. Alnæs, M.S., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M.E., Wells, G.N.: The FEniCS Project Version 1.5. Arch. Numer. Softw. **3**(100), 9–23 (2015)
3. Artioli, E., Beirão da Veiga, L., Lovadina, C., Sacco, E.: Arbitrary order 2D virtual elements for polygonal meshes: part II, inelastic problem. Comput. Mech. **0**(0), 0 (2017). DOI 10.1007/s00466-017-1429-9
4. Babuška, I., Banerjee, U., Osborn, J.E., Li, Q.L.: Quadrature for meshless methods. Int. J. Numer. Meth. Engng. **76**(9), 1434–1470 (2008)

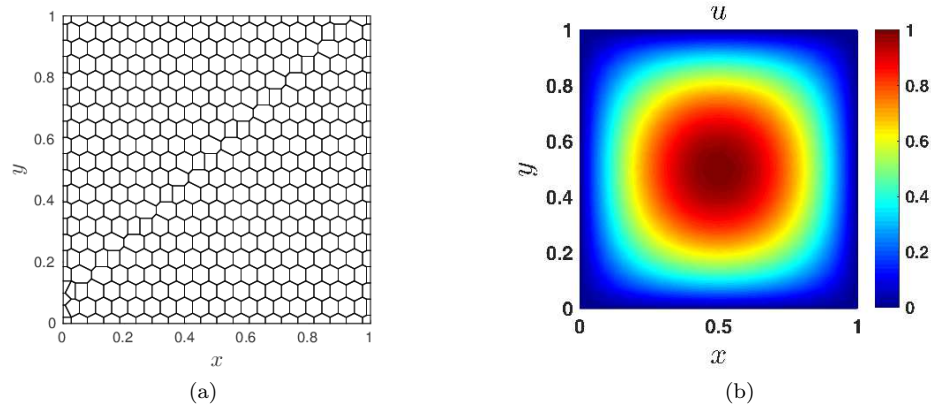


Fig. 19 Solution for the Poisson problem using **Veamy**. (a) Polygonal mesh and (b) VEM solution. The relative L^2 -norm of the error is 2.6695×10^{-3} and the relative H^1 -seminorm of the error is 6.7834×10^{-2}

5. Babuška, I., Banerjee, U., Osborn, J.E., Zhang, Q.: Effect of numerical integration on meshless methods. *Comput. Methods Appl. Mech. Engrg.* **198**(37–40), 2886–2897 (2009)
6. Cangiani, A., Manzini, G., Russo, A., Sukumar, N.: Hourglass stabilization and the virtual element method. *Int. J. Numer. Meth. Engng.* **102**(3–4), 404–436 (2015)
7. Chen, J.S., Wu, C.T., Yoon, S., You, Y.: A stabilized conforming nodal integration for Galerkin mesh-free methods. *Int. J. Numer. Meth. Engng.* **50**(2), 435–466 (2001)
8. Chi, H., Beirão da Veiga, L., Paulino, G.: Some basic formulations of the virtual element method (VEM) for finite deformations. *Comput. Methods Appl. Mech. Engrg.* **318**, 148–192 (2017)
9. Beirão da Veiga, L., Manzini, G.: A virtual element method with arbitrary regularity. *IMA J. Numer. Anal.* **34**(2), 759–781 (2014)
10. Dolbow, J., Belytschko, T.: Numerical integration of Galerkin weak form in meshfree methods. *Comput. Mech.* **23**(3), 219–230 (1999)
11. Duan, Q., Gao, X., Wang, B., Li, X., Zhang, H., Belytschko, T., Shao, Y.: Consistent element-free Galerkin method. *Int. J. Numer. Meth. Engng.* **99**(2), 79–101 (2014)
12. Duan, Q., Gao, X., Wang, B., Li, X., Zhang, H.: A four-point integration scheme with quadratic exactness for three-dimensional element-free Galerkin method based on variationally consistent formulation. *Comput. Methods Appl. Mech. Engrg.* **280**(0), 84–116 (2014)
13. Duan, Q., Li, X., Zhang, H., Belytschko, T.: Second-order accurate derivatives and integration schemes for meshfree methods. *Int. J. Numer. Meth. Engng.* **92**(4), 399–424 (2012)
14. Francis, A., Ortiz-Bernardin, A., Bordas, S., Natarajan, S.: Linear smoothed polygonal and polyhedral finite elements. *Int. J. Numer. Meth. Engng.* **109**(9), 1263–1288 (2017)
15. Gain, A.L., Talischi, C., Paulino, G.H.: On the virtual element method for three-dimensional linear elasticity problems on arbitrary polyhedral meshes. *Comput. Methods Appl. Mech. Engrg.* **282**(0), 132–160 (2014)
16. Guennebaud, G., Jacob, B., et al.: Eigen v3. <http://eigen.tuxfamily.org> (2010)
17. Hecht, F.: New development in FreeFem++. *J. Numer. Math.* **20**(3–4), 251–265 (2012)
18. Johnson, A.: Clipper - an open source freeware library for clipping and offsetting lines and polygons (version: 6.1.3). <http://www.angusj.com/delphi/clipper.php> (2014)

19. Manzini, G., Russo, A., Sukumar, N.: New perspectives on polygonal and polyhedral finite element methods. *Math. Models Methods Appl. Sci.* **24**(08), 1665–1699 (2014)
20. Ortiz, A., Puso, M.A., Sukumar, N.: Maximum-entropy meshfree method for compressible and near-incompressible elasticity. *Comput. Methods Appl. Mech. Engrg.* **199**(25–28), 1859–1871 (2010)
21. Ortiz, A., Puso, M.A., Sukumar, N.: Maximum-entropy meshfree method for incompressible media problems. *Finite Elem. Anal. Des.* **47**(6), 572–585 (2011)
22. Ortiz-Bernardin, A., Hale, J.S., Cyron, C.J.: Volume-averaged nodal projection method for nearly-incompressible elasticity using meshfree and bubble basis functions. *Comput. Methods Appl. Mech. Engrg.* **285**, 427–451 (2015)
23. Ortiz-Bernardin, A., Puso, M.A., Sukumar, N.: Improved robustness for nearly-incompressible large deformation meshfree simulations on Delaunay tessellations. *Comput. Methods Appl. Mech. Engrg.* **293**, 348–374 (2015)
24. Ortiz-Bernardin, A., Russo, A., Sukumar, N.: Consistent and stable meshfree Galerkin methods using the virtual element decomposition. *Int. J. Numer. Meth. Engng.* (2017). DOI 10.1002/nme.5519
25. PrudHomme, C., Chabannes, V., Doyeux, V., Ismail, M., Samake, A., Pena, G.: Feel++: A computational framework for Galerkin methods and advanced numerical methods. In: *ESAIM: Proceedings*, vol. 38, pp. 429–455. EDP Sciences (2012)
26. Rycroft, C.H.: Voro++: A three-dimensional Voronoi cell library in C++. *Chaos* **19** (2009)
27. Shewchuk, J.R.: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In: M.C. Lin, D. Manocha (eds.) *Applied Computational Geometry: Towards Geometric Engineering, Lecture Notes in Computer Science*, vol. 1148, pp. 203–222. Springer-Verlag (1996). From the First ACM Workshop on Applied Computational Geometry
28. Strang, G., Fix, G.: *An Analysis of the Finite Element Method*, second edn. Wellesley-Cambridge Press, MA (2008)
29. Sutton, O.J.: The virtual element method in 50 lines of MATLAB. *Numer. Algor.* **75**(4), 1141–1159 (2017)
30. Talischi, C., Paulino, G.H.: Addressing integration error for polygonal finite elements through polynomial projections: A patch test connection. *Math. Models Methods Appl. Sci.* **24**(08), 1701–1727 (2014)
31. Talischi, C., Paulino, G.H., Pereira, A., Menezes, I.F.M.: PolyMesher: a general-purpose mesh generator for polygonal elements written in Matlab. *Struct. Multidisc. Optim.* **45**(3), 309–328 (2012)
32. Talischi, C., Pereira, A., Menezes, I., Paulino, G.: Gradient correction for polygonal and polyhedral finite elements. *Int. J. Numer. Meth. Engng.* **102**(3–4), 728–747 (2015)
33. Timoshenko, S.P., Goodier, J.N.: *Theory of Elasticity*, third edn. McGraw-Hill, NY (1970)
34. Beirão da Veiga, L., Brezzi, F., Cangiani, A., Manzini, G., Marini, L.D., Russo, A.: Basic principles of virtual element methods. *Math. Models Methods Appl. Sci.* **23**(1), 199–214 (2013)
35. Beirão da Veiga, L., Brezzi, F., Marini, L.D.: Virtual elements for linear elasticity problems. *SIAM J. Numer. Anal.* **51**(2), 794–812 (2013)
36. Beirão da Veiga, L., Brezzi, F., Marini, L.D., Russo, A.: The Hitchhiker’s Guide to the Virtual Element Method. *Math. Models Methods Appl. Sci.* **24**(08), 1541–1573 (2014)
37. Beirão da Veiga, L., Lovadina, C., Mora, D.: A virtual element method for elastic and inelastic problems on polytope meshes. *Comput. Methods Appl. Mech. Engrg.* **295**, 327–346 (2015)
38. Wriggers, P., Reddy, B.D., Rust, W., Hudobivnik, B.: Efficient virtual element formulations for compressible and incompressible finite deformations. *Comput. Mech.* **60**(2), 253–268 (2017)

-
39. Artioli, E., Beirão da Veiga, L., Lovadina, C., Sacco, E.: Arbitrary order 2D virtual elements for polygonal meshes: part I, elastic problem. *Comput. Mech.* **60**(3), 355–377 (2017)
 40. Lie, K.-A.: An Introduction to Reservoir Simulation Using MATLAB: User guide for the Matlab Reservoir Simulation Toolbox (MRST). SINTEF ICT, (2016)
 41. Klemetsdal, Ø. S.: The virtual element method as a common framework for finite element and finite difference methods — Numerical and theoretical analysis. NTNU, (2016)