Numerical Algorithms manuscript No.
(will be inserted by the editor)

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]

arXiv:1708.03438v4 [cs.MS] 24 Dec 2018

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, upon which `Veamy` is built, is presented using a notation and a terminology that resemble the language of the finite

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

C. Alvarez
E-mail: catalin@uchile.cl

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

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

R. Silva-Valenzuela
E-mail: rosilva@ug.uchile.cl

E. Olate-Sanzana
E-mail: edgardo.olate@ing.uchile.cl

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

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

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

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

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

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

element method (FEM) in engineering analysis. Several examples are provided to demonstrate the usage of `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 the Galerkin weak formulation of a boundary-value problem such as the linear elastostatic problem is solved numerically, the trial and test displacements are replaced by their discrete representations using basis functions. Herein, we consider basis functions that span the space of functions of degree 1 (i.e., affine functions). Due to the nature of some basis functions, the discrete trial and test displacement fields may represent linear fields plus some additional functions that are non-polynomials or high-order monomials. 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*). 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 → 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 the VEM exhibits over the standard finite element method (FEM) are:

- Ability to perform simulations using meshes formed 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 [42].

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, upon which `Veamy` is built, 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.

In `Veamy`'s programming philosophy entities commonly found in the VEM and FEM literature such as mesh, degree of freedom, element, 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-line MATLAB implementation of the 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 of the MATLAB Reservoir Simulation Toolbox [40], which provides a module for first- and second-order virtual element methods for Poisson-type flow equations that 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 model problem for two-dimensional linear elastostatics is presented in Section 2. Section 3 summarizes the theoretical framework of the VEM for the two-dimensional linear elastostatic problem. Also in this section, the VEM element stiffness matrix for the two-dimensional Poisson problem is given. 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

The Galerkin weak formulation for the linear elastostatic problem is considered for presenting the main ingredients of the VEM. 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 $\boldsymbol{n}_\Gamma$. 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 $\boldsymbol{u}(\boldsymbol{x}) : \Omega \rightarrow \mathbb{R}^2$ be the displacement field at a point $\boldsymbol{x}$ of the elastic body when the body is

subjected to external tractions $\boldsymbol{f}(\boldsymbol{x}) : \Gamma_f \to \mathbb{R}^2$ and body forces $\boldsymbol{b}(\boldsymbol{x}) : \Omega \to \mathbb{R}^2$. The imposed essential (Dirichlet) boundary conditions are $\boldsymbol{g}(\boldsymbol{x}) : \Gamma_g \to \mathbb{R}^2$. The Galerkin weak formulation, with $\boldsymbol{v}$ being the arbitrary test function, gives the following expression for the bilinear form:

$$a(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\nabla} \boldsymbol{v} \, \mathrm{d}\boldsymbol{x}, \tag{1}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor and $\boldsymbol{\nabla}$ is the gradient operator. The gradient of the displacement field can be decomposed into its symmetric ($\boldsymbol{\nabla}_{\mathrm{S}} \boldsymbol{v}$) and skew-symmetric ($\boldsymbol{\nabla}_{\mathrm{AS}} \boldsymbol{v}$) parts, as follows:

$$\boldsymbol{\nabla} \boldsymbol{v} = \boldsymbol{\nabla}_{\mathrm{S}} \boldsymbol{v} + \boldsymbol{\nabla}_{\mathrm{AS}} \boldsymbol{v} = \boldsymbol{\varepsilon}(\boldsymbol{v}) + \boldsymbol{\omega}(\boldsymbol{v}), \tag{2}$$

where

$$\boldsymbol{\nabla}_{\mathrm{S}} \boldsymbol{v} = \boldsymbol{\varepsilon}(\boldsymbol{v}) = \frac{1}{2} \left(\boldsymbol{\nabla} \boldsymbol{v} + \boldsymbol{\nabla}^{\mathsf{T}} \boldsymbol{v} \right) \tag{3}$$

is the strain tensor, and

$$\boldsymbol{\nabla}_{\mathrm{AS}} \boldsymbol{v} = \boldsymbol{\omega}(\boldsymbol{v}) = \frac{1}{2} \left(\boldsymbol{\nabla} \boldsymbol{v} - \boldsymbol{\nabla}^{\mathsf{T}} \boldsymbol{v} \right) \tag{4}$$

is the skew-symmetric gradient tensor that represents rotations. The Cauchy stress tensor is related to the strain tensor by

$$\boldsymbol{\sigma} = \boldsymbol{D} : \boldsymbol{\varepsilon}(\boldsymbol{u}), \tag{5}$$

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

Substituting (2) into (1) and noting that $\boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\omega}(\boldsymbol{v}) = 0$ because of the symmetry of the stress tensor, results in the following simplification of the bilinear form:

$$a(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\varepsilon}(\boldsymbol{v}) \, \mathrm{d}\boldsymbol{x}, \tag{6}$$

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

$$a(\boldsymbol{u}, \boldsymbol{v}) = \ell_b(\boldsymbol{v}) + \ell_f(\boldsymbol{v}) \quad \forall \boldsymbol{v}(\boldsymbol{x}) \in W, \tag{7a}$$

$$a(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\varepsilon}(\boldsymbol{v}) \, \mathrm{d}\boldsymbol{x}, \tag{7b}$$

$$\ell_b(\boldsymbol{v}) = \int_\Omega \boldsymbol{b} \cdot \boldsymbol{v} \, \mathrm{d}\boldsymbol{x}, \quad \ell_f(\boldsymbol{v}) = \int_{\Gamma_f} \boldsymbol{f} \cdot \boldsymbol{v} \, \mathrm{d}s, \tag{7c}$$

where V and W are the displacement trial and test spaces defined as follows:

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

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

In the Galerkin approximation, the domain Ω is partitioned into disjoint non overlapping elements. This partition is 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 any element 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 $\mathcal{E}^h_f$. On this partition, the trial and test displacement

fields are approximated using basis functions, and hence $\boldsymbol{u}$ and $\boldsymbol{v}$ are replaced by the approximations $\boldsymbol{u}^h$ and $\boldsymbol{v}^h$, respectively. The bilinear and linear forms are then obtained by summation of the contributions from the elements in the mesh, as follows:

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

In general, the weak form integrals are not available in closed-form expressions since functions in $\mathcal{W}(E)$, and in particular its basis, are not necessarily polynomial functions. Therefore, these integrals are evaluated using quadrature with the potential of introducing quadrature errors making them mesh-dependent. If that is the case, the convergence of the numerical solution will be affected. To reflect this, a superscript h is added to the symbols that represent the bilinear and linear forms. Thus, the Galerkin solution is sought as the solution of the global system that results from the weak formulation described by the following discrete bilinear and linear forms:

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

respectively, with the corresponding discrete global trial and test spaces defined respectively as follows:

$$V^h:=\left\{\boldsymbol{u}^h(\boldsymbol{x})\in V:\boldsymbol{u}^h|_E\in\mathcal{W}(E)\subseteq[H^1(E)]^2\ \forall E\in\mathcal{T}^h\right\},$$
$$W^h:=\left\{\boldsymbol{v}^h(\boldsymbol{x})\in W:\boldsymbol{v}^h|_E\in\mathcal{W}(E)\subseteq[H^1(E)]^2\ \forall E\in\mathcal{T}^h\right\}.$$

In the preceding discussion, we have implied that a_E^h is inexact due to its evaluation using numerical quadrature — in this case, a_E^h is said to be *uncomputable*. The situation is completely different in the VEM approach: a_E^h is not evaluated using numerical quadrature. Instead, the displacement field is computed through projection operators that are tailored to achieve an algebraic (exact) evaluation of a_E^h — in this case, a_E^h is said to be *computable*.

3 The virtual element method

In standard two-dimensional finite element methods, the partition $\mathcal{T}^h$ is usually formed by triangles and quadrilaterals. In the VEM, the partition is formed by elements with arbitrary number of edges, where triangles and quadrilaterals are particular instances. We refer to these more general elements as polygonal elements.

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 $\boldsymbol{n}=[n_1\quad n_2]^\mathsf{T}$. 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 $\boldsymbol{n}_a$ and $\boldsymbol{n}_{a-1}$ are the unit outward normals to these edges, respectively.

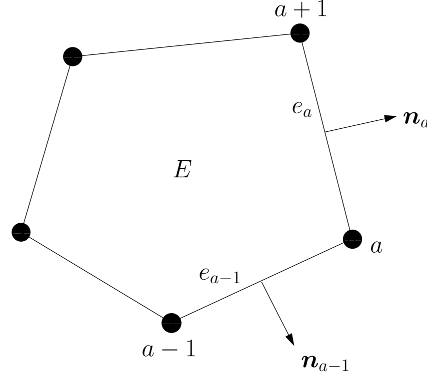


Fig. 1 Schematic representation of a polygonal element of $N = 5$ edges

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]. In the VEM, projection operators are devised to extract the rigid body modes, the constant strain states and the linear polynomial part of the motion at the element level. The spaces where these components of the motion reside are given next.

The space of linear displacements over E is defined as

$$\mathcal{P}(E) := \left\{ \boldsymbol{a} + \boldsymbol{B}(\boldsymbol{x} - \overline{\boldsymbol{x}}) : \boldsymbol{a} \in \mathbb{R}^2,\, \boldsymbol{B} \in \mathbb{R}^{2\times 2} \right\}, \tag{8}$$

where $\overline{\boldsymbol{x}}$ is defined through the mean value of a function h over the element nodes given by

$$\overline{h} = \frac{1}{N} \sum_{j=1}^{N} h(\boldsymbol{x}_j), \tag{9}$$

where N is the number of nodes of coordinates $\boldsymbol{x}_j$ that define the polygonal element*; $\boldsymbol{B}$ 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 $\boldsymbol{B}_{\mathrm{S}}$ and $\boldsymbol{B}_{\mathrm{AS}}$, respectively. The spaces of rigid body modes and constant strain states over E are defined, respectively, as follows:

$$\mathcal{R}(E) := \left\{ \boldsymbol{a} + \boldsymbol{B}_{\mathrm{AS}} \cdot (\boldsymbol{x} - \overline{\boldsymbol{x}}) : \boldsymbol{a} \in \mathbb{R}^2,\, \boldsymbol{B}_{\mathrm{AS}} \in \mathbb{R}^{2\times 2},\, \boldsymbol{B}_{\mathrm{AS}}^{\mathsf{T}} = -\boldsymbol{B}_{\mathrm{AS}} \right\}, \tag{10}$$

$$\mathcal{C}(E) := \left\{ \boldsymbol{B}_{\mathrm{S}} \cdot (\boldsymbol{x} - \overline{\boldsymbol{x}}) : \boldsymbol{B}_{\mathrm{S}} \in \mathbb{R}^{2\times 2},\, \boldsymbol{B}_{\mathrm{S}}^{\mathsf{T}} = \boldsymbol{B}_{\mathrm{S}} \right\}. \tag{11}$$

Note that the space of linear displacements is the direct sum of the spaces given in (10) and (11), that is, $\mathcal{P}(E) = \mathcal{R}(E) + \mathcal{C}(E)$.

The extraction of the components of the displacement field in the three aforementioned spaces is achieved through the following projection operators:

$$\mathit{\Pi}_{\mathcal{R}} : \mathcal{W}(E) \to \mathcal{R}(E), \quad \mathit{\Pi}_{\mathcal{R}} \boldsymbol{r} = \boldsymbol{r}, \quad \forall \boldsymbol{r} \in \mathcal{R}(E) \tag{12}$$

*Eq. (9) in fact defines any 'barred' term that appears in this paper.

for extracting the rigid body modes,

$$\Pi_{\mathcal{C}}: \mathcal{W}(E) \to \mathcal{C}(E), \quad \Pi_{\mathcal{C}} \boldsymbol{c} = \boldsymbol{c}, \quad \forall \boldsymbol{c} \in \mathcal{C}(E) \tag{13}$$

for extracting the constant strain states, and

$$\Pi_{\mathcal{P}}: \mathcal{W}(E) \to \mathcal{P}(E), \quad \Pi_{\mathcal{P}} \boldsymbol{p} = \boldsymbol{p}, \quad \forall \boldsymbol{p} \in \mathcal{P}(E) \tag{14}$$

for extracting the linear polynomial part. And since $\mathcal{P}(E) = \mathcal{R}(E) + \mathcal{C}(E)$, the projection operators satisfy the relation

$$\Pi_{\mathcal{P}} = \Pi_{\mathcal{R}} + \Pi_{\mathcal{C}}. \tag{15}$$

We know by definition that the space $\mathcal{W}(E)$ includes linear displacements. This means that $\mathcal{W}(E) \supseteq \mathcal{P}(E)$. Thus, any $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{W}(E)$ can be decomposed into three terms, as follows:

$$\boldsymbol{u} = \Pi_{\mathcal{R}} \boldsymbol{u} + \Pi_{\mathcal{C}} \boldsymbol{u} + (\boldsymbol{u} - \Pi_{\mathcal{P}} \boldsymbol{u}), \tag{16a}$$

$$\boldsymbol{v} = \Pi_{\mathcal{R}} \boldsymbol{v} + \Pi_{\mathcal{C}} \boldsymbol{v} + (\boldsymbol{v} - \Pi_{\mathcal{P}} \boldsymbol{v}), \tag{16b}$$

that is, into their rigid body modes, their constant strain states and their additional non-polynomial or high-order functions, respectively.

The explicit forms of the projection operators that are defined through (12)-(14) are given in Ref. [15] and are summarized as follows: let the cell-average of the strain tensor be defined as

$$\widehat{\boldsymbol{\varepsilon}}(\boldsymbol{v}) = \frac{1}{|E|} \int_E \boldsymbol{\varepsilon}(\boldsymbol{v}) \, \mathrm{d}\boldsymbol{x} = \frac{1}{2|E|} \int_{\partial E} (\boldsymbol{v} \otimes \boldsymbol{n} + \boldsymbol{n} \otimes \boldsymbol{v}) \, \mathrm{d}s, \tag{17}$$

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

$$\widehat{\boldsymbol{\omega}}(\boldsymbol{v}) = \frac{1}{|E|} \int_E \boldsymbol{\omega}(\boldsymbol{v}) \, \mathrm{d}\boldsymbol{x} = \frac{1}{2|E|} \int_{\partial E} (\boldsymbol{v} \otimes \boldsymbol{n} - \boldsymbol{n} \otimes \boldsymbol{v}) \, \mathrm{d}s. \tag{18}$$

Note that $\widehat{\boldsymbol{\varepsilon}}(\boldsymbol{v})$ and $\widehat{\boldsymbol{\omega}}(\boldsymbol{v})$ are constant tensors in the element.

On using the preceding definitions, the projection of $\boldsymbol{v}$ onto the space of rigid body modes is written as

$$\Pi_{\mathcal{R}} \boldsymbol{v} = \widehat{\boldsymbol{\omega}}(\boldsymbol{v}) \cdot (\boldsymbol{x} - \overline{\boldsymbol{x}}) + \overline{\boldsymbol{v}}, \tag{19}$$

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

$$\Pi_{\mathcal{C}} \boldsymbol{v} = \widehat{\boldsymbol{\varepsilon}}(\boldsymbol{v}) \cdot (\boldsymbol{x} - \overline{\boldsymbol{x}}). \tag{20}$$

Hence, by (15) the projection of $\boldsymbol{v}$ onto the space of linear displacements is written as

$$\Pi_{\mathcal{P}} \boldsymbol{v} = \Pi_{\mathcal{R}} \boldsymbol{v} + \Pi_{\mathcal{C}} \boldsymbol{v} = \widehat{\boldsymbol{\varepsilon}}(\boldsymbol{v}) \cdot (\boldsymbol{x} - \overline{\boldsymbol{x}}) + \widehat{\boldsymbol{\omega}}(\boldsymbol{v}) \cdot (\boldsymbol{x} - \overline{\boldsymbol{x}}) + \overline{\boldsymbol{v}}. \tag{21}$$

The projection operator $\Pi_{\mathcal{P}}$ satisfies some important energy-orthogonality conditions that are invoked when constructing the VEM bilinear form. The proofs can be found in Ref. [15]. The energy-orthogonality conditions are given next.

The projection $\Pi_{\mathcal{P}}$ satisfies:

$$a_E(\boldsymbol{p}, \boldsymbol{v} - \Pi_{\mathcal{P}} \boldsymbol{v}) = 0 \quad \forall \boldsymbol{p} \in \mathcal{P}(E), \quad \boldsymbol{v} \in \mathcal{W}(E), \tag{22a}$$

$$a_E(\boldsymbol{c}, \boldsymbol{v} - \Pi_{\mathcal{P}} \boldsymbol{v}) = 0 \quad \forall \boldsymbol{c} \in \mathcal{C}(E), \quad \boldsymbol{v} \in \mathcal{W}(E). \tag{22b}$$

The condition (22a) means that $\boldsymbol{v} - \Pi_{\mathcal{P}} \boldsymbol{v}$ is energetically orthogonal to $\mathcal{P}$. The condition (22b) emanates from condition (22a) after replacing $\boldsymbol{p} = \boldsymbol{r} + \boldsymbol{c}$ and using the fact that rigid body modes have zero strain, that is $a_E(\boldsymbol{r}, \cdot) = 0$.

3.3 The VEM bilinear form

Substituting the VEM decomposition (16) into the bilinear form (6) leads to the following splitting of the bilinear form at element level:

$$\begin{aligned} a_E(\boldsymbol{u},\boldsymbol{v}) &= a_E(\Pi_{\mathcal{R}}\boldsymbol{u}+\Pi_{\mathcal{C}}\boldsymbol{u}+(\boldsymbol{u}-\Pi_{\mathcal{P}}\boldsymbol{u}),\Pi_{\mathcal{R}}\boldsymbol{v}+\Pi_{\mathcal{C}}\boldsymbol{v}+(\boldsymbol{v}-\Pi_{\mathcal{P}}\boldsymbol{v})) \\ &= a_E(\Pi_{\mathcal{C}}\boldsymbol{u},\Pi_{\mathcal{C}}\boldsymbol{v})+a_E(\boldsymbol{u}-\Pi_{\mathcal{P}}\boldsymbol{u},\boldsymbol{v}-\Pi_{\mathcal{P}}\boldsymbol{v}), \end{aligned} \tag{23}$$

where the symmetry of the bilinear form, the fact that $\Pi_{\mathcal{R}}\boldsymbol{u}$ and $\Pi_{\mathcal{R}}\boldsymbol{v}$ do not contribute in the bilinear form (both have zero strain as they belong to the space of rigid body modes), and the energy-orthogonality condition (22b) have been used.

The first term on the right-hand side of (23) is the bilinear form associated with the constant strain states that provides consistency (it leads to the *consistency* stiffness) and the second term is the bilinear form associated with the additional non-polynomial or high-order functions that provides stability (it leads to the *stability* stiffness). We come back to these concepts later in this section.

3.4 Projection matrices

The projection matrices are constructed by discretizing the projection operators. We begin by writing the projections $\Pi_{\mathcal{R}}\boldsymbol{v}$ and $\Pi_{\mathcal{C}}\boldsymbol{v}$ in terms of their space basis. To this end, consider the two-dimensional Cartesian space and the skew-symmetry of $\widehat{\boldsymbol{\omega}} \equiv \widehat{\boldsymbol{\omega}}(\boldsymbol{v})$[‡]. The projection (19) can be written as follows:

$$\Pi_{\mathcal{R}}\boldsymbol{v} = \boldsymbol{r}_1\overline{v}_1+\boldsymbol{r}_2\overline{v}_2+\boldsymbol{r}_3\widehat{\omega}_{12}, \tag{24}$$

where the basis for the space of rigid body modes is:

$$\boldsymbol{r}_1 = \left[\begin{array}{cc} 1 & 0 \end{array}\right]^{\mathsf{T}},\ \boldsymbol{r}_2 = \left[\begin{array}{cc} 0 & 1 \end{array}\right]^{\mathsf{T}},\ \boldsymbol{r}_3 = \left[\begin{array}{cc} (x_2-\overline{x}_2) & -(x_1-\overline{x}_1) \end{array}\right]^{\mathsf{T}}. \tag{25}$$

Similarly, on considering the symmetry of $\widehat{\boldsymbol{\varepsilon}} \equiv \widehat{\boldsymbol{\varepsilon}}(\boldsymbol{v})$, the projection (20) can be written as

$$\Pi_{\mathcal{C}}\boldsymbol{v} = \boldsymbol{c}_1\widehat{\varepsilon}_{11}+\boldsymbol{c}_2\widehat{\varepsilon}_{22}+\boldsymbol{c}_3\widehat{\varepsilon}_{12}, \tag{26}$$

where the basis for the space of constant strain states is:

$$\boldsymbol{c}_1 = \left[\begin{array}{cc} (x_1-\overline{x}_1) & 0 \end{array}\right]^{\mathsf{T}},\ \boldsymbol{c}_2 = \left[\begin{array}{cc} 0 & (x_2-\overline{x}_2) \end{array}\right]^{\mathsf{T}},\ \boldsymbol{c}_3 = \left[\begin{array}{cc} (x_2-\overline{x}_2) & (x_1-\overline{x}_1) \end{array}\right]^{\mathsf{T}}. \tag{27}$$

On each polygonal element of N edges with nodal coordinates denoted by $\boldsymbol{x}_a = [x_{1a} \quad x_{2a}]^{\mathsf{T}}$, the trial and test displacements are locally approximated as

$$\boldsymbol{u}^h(\boldsymbol{x}) = \sum_{a=1}^{N}\phi_a(\boldsymbol{x})\boldsymbol{u}_a, \quad \boldsymbol{v}^h(\boldsymbol{x}) = \sum_{b=1}^{N}\phi_b(\boldsymbol{x})\boldsymbol{v}_b, \tag{28}$$

where $\phi_a(\boldsymbol{x})$ and $\phi_b(\boldsymbol{x})$ are assumed to be the canonical basis functions having the Kronecker delta property (i.e., Lagrange-type functions), and $\boldsymbol{u}_a = [u_{1a} \quad u_{2a}]^{\mathsf{T}}$ and $\boldsymbol{v}_b = [v_{1b} \quad v_{2b}]^{\mathsf{T}}$ are nodal

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

displacements. The canonical basis functions are also used to locally approximate the components of the basis for the space of rigid body modes:

$$\boldsymbol{r}_\alpha^h(\boldsymbol{x}) = \sum_{a=1}^{N} \phi_a(\boldsymbol{x}) \boldsymbol{r}_\alpha(\boldsymbol{x}_a), \quad \alpha = 1, \ldots, 3 \tag{29}$$

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

$$\boldsymbol{c}_\beta^h(\boldsymbol{x}) = \sum_{a=1}^{N} \phi_a(\boldsymbol{x}) \boldsymbol{c}_\beta(\boldsymbol{x}_a), \quad \beta = 1, \ldots, 3. \tag{30}$$

The discrete version of the projection to extract the rigid body modes is obtained by substituting (28) and (29) into (24), which yields

$$\Pi_{\mathcal{R}} \boldsymbol{v}^h = \boldsymbol{N} \boldsymbol{P}_{\mathcal{R}} \boldsymbol{q}, \tag{31}$$

where

$$\boldsymbol{N} = \left[(\boldsymbol{N})_1 \quad \cdots \quad (\boldsymbol{N})_a \quad \cdots \quad (\boldsymbol{N})_N \right], \; (\boldsymbol{N})_a = \begin{bmatrix} \phi_a & 0 \\ 0 & \phi_a \end{bmatrix}, \tag{32}$$

$$\boldsymbol{q} = \left[\boldsymbol{v}_1^\mathsf{T} \quad \cdots \quad \boldsymbol{v}_a^\mathsf{T} \quad \cdots \quad \boldsymbol{v}_N^\mathsf{T} \right]^\mathsf{T}, \; \boldsymbol{v}_a = \left[v_{1a} \quad v_{2a} \right]^\mathsf{T} \tag{33}$$

and

$$\boldsymbol{P}_{\mathcal{R}} = \boldsymbol{H}_{\mathcal{R}} \boldsymbol{W}_{\mathcal{R}}^\mathsf{T} \tag{34}$$

with

$$\boldsymbol{H}_{\mathcal{R}} = \left[\; (\boldsymbol{H}_{\mathcal{R}})_1 \quad \cdots \quad (\boldsymbol{H}_{\mathcal{R}})_a \quad \cdots \quad (\boldsymbol{H}_{\mathcal{R}})_N \; \right]^\mathsf{T}, \quad (\boldsymbol{H}_{\mathcal{R}})_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ (x_{2a} - \overline{x}_2) & -(x_{1a} - \overline{x}_1) \end{bmatrix}^\mathsf{T} \tag{35}$$

and

$$\boldsymbol{W}_{\mathcal{R}} = \left[\; (\boldsymbol{W}_{\mathcal{R}})_1 \quad \cdots \quad (\boldsymbol{W}_{\mathcal{R}})_a \quad \cdots \quad (\boldsymbol{W}_{\mathcal{R}})_N \; \right]^\mathsf{T}, \quad (\boldsymbol{W}_{\mathcal{R}})_a = \begin{bmatrix} \overline{\phi}_a & 0 \\ 0 & \overline{\phi}_a \\ q_{2a} & -q_{1a} \end{bmatrix}^\mathsf{T}. \tag{36}$$

In (36), q_{ia} appeared because of the discretization of $\widehat{\omega}_{12}$ (see (18)) and is given by

$$q_{ia} = \frac{1}{2|E|} \int_{\partial E} \phi_a n_i \, \mathrm{d}s, \quad i = 1, 2. \tag{37}$$

Similarly, substituting (28) and (30) into (26) leads to the following discrete version of the projection to extract the constant strain states:

$$\Pi_{\mathcal{C}} \boldsymbol{v}^h = \boldsymbol{N} \boldsymbol{P}_{\mathcal{C}} \boldsymbol{q}, \tag{38}$$

where

$$\boldsymbol{P}_{\mathcal{C}} = \boldsymbol{H}_{\mathcal{C}} \boldsymbol{W}_{\mathcal{C}}^\mathsf{T} \tag{39}$$

with

$$\boldsymbol{H}_{\mathcal{C}} = \left[\; (\boldsymbol{H}_{\mathcal{C}})_1 \quad \cdots \quad (\boldsymbol{H}_{\mathcal{C}})_a \quad \cdots \quad (\boldsymbol{H}_{\mathcal{C}})_N \; \right]^\mathsf{T}, \quad (\boldsymbol{H}_{\mathcal{C}})_a = \begin{bmatrix} (x_{1a} - \overline{x}_1) & 0 \\ 0 & (x_{2a} - \overline{x}_2) \\ (x_{2a} - \overline{x}_2) & (x_{1a} - \overline{x}_1) \end{bmatrix}^\mathsf{T} \tag{40}$$

and

$$\boldsymbol{W}_{\mathcal{C}} = [\ (\boldsymbol{W}_{\mathcal{C}})_1 \ \cdots \ (\boldsymbol{W}_{\mathcal{C}})_a \ \cdots \ (\boldsymbol{W}_{\mathcal{C}})_N\ ]^{\mathsf{T}}, \quad (\boldsymbol{W}_{\mathcal{C}})_a = \begin{bmatrix} 2q_{1a} & 0 \\ 0 & 2q_{2a} \\ q_{2a} & q_{1a} \end{bmatrix}^{\mathsf{T}}. \tag{41}$$

In (41), q_{ia} is also given by (37) but in this case it stems from the discretization of $\widehat{\varepsilon}_{ij}$ (see (17)).

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

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

$$\begin{aligned} \Pi_{\mathcal{C}} \boldsymbol{v}^h &= \boldsymbol{c}_1 \widehat{\varepsilon}_{11} + \boldsymbol{c}_2 \widehat{\varepsilon}_{22} + \boldsymbol{c}_3 \widehat{\varepsilon}_{12} \\ &= [\ \boldsymbol{c}_1 \ \ \boldsymbol{c}_2 \ \ \boldsymbol{c}_3\ ] \sum_{b=1}^{N} \begin{bmatrix} 2q_{1b} & 0 \\ 0 & 2q_{2b} \\ q_{2b} & q_{1b} \end{bmatrix} \begin{bmatrix} v_{1b} \\ v_{2b} \end{bmatrix} \\ &= \boldsymbol{c}\, \boldsymbol{W}_{\mathcal{C}}^{\mathsf{T}}\, \boldsymbol{q}. \end{aligned} \tag{42}$$

3.5 VEM element stiffness matrix

The decomposition given in (23) is used to construct the approximate mesh-dependent bilinear form $a_E^h(\boldsymbol{u}, \boldsymbol{v})$ in a way that is computable at the element level. To this end, we approximate the quantity $a_E(\boldsymbol{u} - \Pi_{\mathcal{P}}\boldsymbol{u}, \boldsymbol{v} - \Pi_{\mathcal{P}}\boldsymbol{v})$, which is uncomputable, with a computable one given by $s_E(\boldsymbol{u} - \Pi_{\mathcal{P}}\boldsymbol{u}, \boldsymbol{v} - \Pi_{\mathcal{P}}\boldsymbol{v})$ and define

$$a_E^h(\boldsymbol{u}, \boldsymbol{v}) := a_E(\Pi_{\mathcal{C}}\boldsymbol{u}, \Pi_{\mathcal{C}}\boldsymbol{v}) + s_E(\boldsymbol{u} - \Pi_{\mathcal{P}}\boldsymbol{u}, \boldsymbol{v} - \Pi_{\mathcal{P}}\boldsymbol{v}), \tag{43}$$

where its right-hand side, as it will be revealed in the sequel, is computed algebraically. The decomposition (43) 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*: $\forall \boldsymbol{p} \in \mathcal{P}(E)$ and $\forall \boldsymbol{v}^h \in V^h|_E$

$$a_E^h(\boldsymbol{p}, \boldsymbol{v}^h) = a_E(\boldsymbol{p}, \boldsymbol{v}^h). \tag{44}$$

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

$$\forall \boldsymbol{v}^h \in V^h|_E, \quad \alpha_* a_E(\boldsymbol{v}^h, \boldsymbol{v}^h) \le a_E^h(\boldsymbol{v}^h, \boldsymbol{v}^h) \le \alpha^* a_E(\boldsymbol{v}^h, \boldsymbol{v}^h). \tag{45}$$

The discrete version of the VEM element bilinear form (43) is constructed as follows. Substitute (42) into the first term of the right-hand side of (43) (note that when $\boldsymbol{u}^h$ is used instead of $\boldsymbol{v}^h$, $\boldsymbol{q}$ is replaced by the column vector of nodal displacements $\boldsymbol{d}$, which has the same structure of $\boldsymbol{q}$); use (31) and (38) to obtain $\Pi_{\mathcal{P}}\boldsymbol{v}^h = \Pi_{\mathcal{R}}\boldsymbol{v}^h + \Pi_{\mathcal{C}}\boldsymbol{v}^h = \boldsymbol{N}\boldsymbol{P}_{\mathcal{P}}\boldsymbol{q}$, where $\boldsymbol{P}_{\mathcal{P}} = \boldsymbol{H}_{\mathcal{R}}\boldsymbol{W}_{\mathcal{R}}^{\mathsf{T}} + \boldsymbol{H}_{\mathcal{C}}\boldsymbol{W}_{\mathcal{C}}^{\mathsf{T}}$. Also, note that $\boldsymbol{v}^h = \boldsymbol{N}\boldsymbol{q}$. Then, substitute the expressions for $\Pi_{\mathcal{P}}\boldsymbol{v}^h$ and $\boldsymbol{v}^h$ into the second term of the right-hand side of (43). This yields

$$\begin{aligned} a_E^h(\boldsymbol{u}^h, \boldsymbol{v}^h) &= a_E(\boldsymbol{c}\, \boldsymbol{W}_{\mathcal{C}}^{\mathsf{T}}\, \boldsymbol{d}, \boldsymbol{c}\, \boldsymbol{W}_{\mathcal{C}}^{\mathsf{T}}\, \boldsymbol{q}) + s_E(\boldsymbol{N}\boldsymbol{d} - \boldsymbol{N}\boldsymbol{P}_{\mathcal{P}}\boldsymbol{d}, \boldsymbol{N}\boldsymbol{q} - \boldsymbol{N}\boldsymbol{P}_{\mathcal{P}}\boldsymbol{q}) \\ &= \boldsymbol{q}^{\mathsf{T}} \boldsymbol{W}_{\mathcal{C}}\, a_E(\boldsymbol{c}^{\mathsf{T}}, \boldsymbol{c})\, \boldsymbol{W}_{\mathcal{C}}^{\mathsf{T}} \boldsymbol{d} + \boldsymbol{q}^{\mathsf{T}} (\boldsymbol{I}_{2N} - \boldsymbol{P}_{\mathcal{P}})^{\mathsf{T}}\, s_E(\boldsymbol{N}^{\mathsf{T}}, \boldsymbol{N})\, (\boldsymbol{I}_{2N} - \boldsymbol{P}_{\mathcal{P}})\, \boldsymbol{d} \\ &= \boldsymbol{q}^{\mathsf{T}} |E|\, \boldsymbol{W}_{\mathcal{C}}\, \boldsymbol{D}\, \boldsymbol{W}_{\mathcal{C}}^{\mathsf{T}} \boldsymbol{d} + \boldsymbol{q}^{\mathsf{T}} (\boldsymbol{I}_{2N} - \boldsymbol{P}_{\mathcal{P}})^{\mathsf{T}}\, \boldsymbol{S}_E\, (\boldsymbol{I}_{2N} - \boldsymbol{P}_{\mathcal{P}})\, \boldsymbol{d}, \end{aligned} \tag{46}$$

where $\boldsymbol{I}_{2N}$ is the identity $(2N \times 2N)$ matrix and $\boldsymbol{S}_E = s_E(\boldsymbol{N}^\mathsf{T}, \boldsymbol{N})$. Using Voigt notation and observing that $\varepsilon(\boldsymbol{c}) = [\ \varepsilon_{11}(\boldsymbol{c}) \quad \varepsilon_{22}(\boldsymbol{c}) \quad \varepsilon_{12}(\boldsymbol{c})\ ]^\mathsf{T} = \boldsymbol{I}_3$ (the identity (3×3) matrix), in (46) we have used that $a_E(\boldsymbol{c}^\mathsf{T}, \boldsymbol{c}) = \int_E \varepsilon^\mathsf{T}(\boldsymbol{c}) \boldsymbol{D} \varepsilon(\boldsymbol{c})\,\mathrm{d}\boldsymbol{x} = \boldsymbol{D} \int_E \mathrm{d}\boldsymbol{x} = |E| \boldsymbol{D}$, where $\boldsymbol{D}$ is the constitutive matrix for an isotropic linear elastic material given by

$$\boldsymbol{D} = \frac{E_Y}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & 2(1-2\nu) \end{bmatrix} \tag{47}$$

for plane strain condition, and

$$\boldsymbol{D} = \frac{E_Y}{(1-\nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & 2(1-\nu) \end{bmatrix} \tag{48}$$

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 (46) is the *consistency* part of the discrete VEM element bilinear form that provides patch test satisfaction when the solution is a linear displacement field (condition (44) is satisfied). The second term on the right-hand side of (46) is the *stability* part of the discrete VEM element bilinear form and is dependent on the matrix $\boldsymbol{S}_E = s_E(\boldsymbol{N}^\mathsf{T}, \boldsymbol{N})$. This matrix must be chosen such that condition (45) holds without putting at risk condition (44) already taken care of by the consistency part. There are quite a few possibilities for this matrix (see for instance [15, 34, 35]). Herein, we adopt $\boldsymbol{S}_E$ given by [15]

$$\boldsymbol{S}_E = \alpha_E \, \boldsymbol{I}_{2N}, \quad \alpha_E = \gamma \frac{|E| \mathrm{trace}(\boldsymbol{D})}{\mathrm{trace}(\boldsymbol{H}_\mathcal{C}^\mathsf{T} \boldsymbol{H}_\mathcal{C})}, \tag{49}$$

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

From (46), the final expression for the VEM element stiffness matrix is obtained respectively as the summation of the element *consistency* and *stability* stiffness matrices, as follows:

$$\boldsymbol{K}_E = |E| \, \boldsymbol{W}_\mathcal{C} \, \boldsymbol{D} \, \boldsymbol{W}_\mathcal{C}^\mathsf{T} + (\boldsymbol{I}_{2N} - \boldsymbol{P}_\mathcal{P})^\mathsf{T} \, \boldsymbol{S}_E \, (\boldsymbol{I}_{2N} - \boldsymbol{P}_\mathcal{P}), \tag{50}$$

where we recall that $\boldsymbol{P}_\mathcal{P} = \boldsymbol{H}_\mathcal{R} \boldsymbol{W}_\mathcal{R}^\mathsf{T} + \boldsymbol{H}_\mathcal{C} \boldsymbol{W}_\mathcal{C}^\mathsf{T}$. Note that $\boldsymbol{H}_\mathcal{R}$ and $\boldsymbol{H}_\mathcal{C}$, which are given in (35) and (40), respectively, are easily computed using the nodal coordinates of the element. However, in order to compute $\boldsymbol{W}_\mathcal{R}$ and $\boldsymbol{W}_\mathcal{C}$ (see their expressions in (36) and (41), respectively), we need some knowledge of the basis functions so that $\overline{\phi}_a$ and q_{ia} can be determined. Observe that $\overline{\phi}_a$ is computed using (9), which requires the knowledge of the basis functions at the element nodes. And q_{ia} is computed using (37), which requires the knowledge of the basis functions on the element edges. Hence, everything we need to know about the basis functions is their behavior on the element boundary.

We have already mentioned that the basis functions in the VEM are assumed to be Lagrange-type functions. This provides everything we need to know about them on the boundary of an element: basis functions are piecewise linear (edge by edge) and continuous on the element edges, and have the Kronecker delta property. Therefore, $\overline{\phi}_a$ can be computed simply as

$$\overline{\phi}_a = \frac{1}{N} \sum_{j=1}^{N} \phi_a(\boldsymbol{x}_j) = \frac{1}{N}, \tag{51}$$

and q_{ia} can be computed exactly using a trapezoidal rule, which gives

$$q_{ia} = \frac{1}{2|E|} \int_{\partial E} \phi_a n_i \,\mathrm{d}s = \frac{1}{4|E|} \left(|e_{a-1}| \{n_i\}_{a-1} + |e_a| \{n_i\}_a \right), \quad i = 1, 2, \tag{52}$$

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

The adoption of (51) and (52) in the VEM, results in an algebraic evaluation of the element stiffness matrix. This also 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, although the linear approximation of the displacement field everywhere in the element is computable through the projection (21). Therefore, a more specific discrete global trial space than the one already given in Section 2 can be built by assembling element by element the local space defined as [34, 37]

$$V^h|_E := \left\{ \boldsymbol{v}^h \in [H^1(E) \cap C^0(E)]^2 : \Delta \boldsymbol{v}^h = \boldsymbol{0} \text{ in } E,\, \boldsymbol{v}^h|_e = \mathcal{P}(e) \;\; \forall e \in \partial E \right\}. \tag{53}$$

3.6 VEM element body and traction force vectors

For linear displacements, the body force can be approximated by a piecewise constant. Typically, this piecewise constant approximation is defined as the cell-average $\boldsymbol{b}^h = \frac{1}{|E|} \int_E \boldsymbol{b} \,\mathrm{d}\boldsymbol{x} = \widehat{\boldsymbol{b}}$. Thus, the body force part of the discrete VEM element linear form can be computed as follows [34, 35, 39]:

$$\ell_{b,E}^h(\boldsymbol{v}^h) = \int_E \boldsymbol{b}^h \cdot \overline{\boldsymbol{v}}^h \,\mathrm{d}\boldsymbol{x} = |E| \widehat{\boldsymbol{b}} \cdot \overline{\boldsymbol{v}}^h = \boldsymbol{q}^\mathsf{T} |E| \,\overline{\boldsymbol{N}}^\mathsf{T} \widehat{\boldsymbol{b}}, \tag{54}$$

where

$$\overline{\boldsymbol{N}} = \left[(\overline{\boldsymbol{N}})_1 \quad \cdots \quad (\overline{\boldsymbol{N}})_a \quad \cdots \quad (\overline{\boldsymbol{N}})_N \right], \;\; (\overline{\boldsymbol{N}})_a = \begin{bmatrix} \overline{\phi}_a & 0 \\ 0 & \overline{\phi}_a \end{bmatrix}. \tag{55}$$

Hence, the VEM element body force vector is given by

$$\boldsymbol{f}_{b,E} = |E| \,\overline{\boldsymbol{N}}^\mathsf{T} \widehat{\boldsymbol{b}}. \tag{56}$$

The traction force part of the VEM element linear form is similar to the integral expression given in (54), 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 on an element edge lying on the natural (Neumann) boundary, as follows:

$$\boldsymbol{f}_{f,e} = |e| \,\overline{\boldsymbol{N}}_\Gamma^\mathsf{T} \widehat{\boldsymbol{f}}, \tag{57}$$

where

$$\overline{\boldsymbol{N}}_\Gamma = \begin{bmatrix} \overline{\phi}_1 & 0 & \overline{\phi}_2 & 0 \\ 0 & \overline{\phi}_1 & 0 & \overline{\phi}_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{N} & 0 & \frac{1}{N} & 0 \\ 0 & \frac{1}{N} & 0 & \frac{1}{N} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \tag{58}$$

and $\widehat{\boldsymbol{f}} = \frac{1}{|e|} \int_e \boldsymbol{f} \,\mathrm{d}s$.

3.7 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

$$\frac{||\boldsymbol{u} - \Pi_{\mathcal{P}}\boldsymbol{u}^h||_{L^2(\Omega)}}{||\boldsymbol{u}||_{L^2(\Omega)}} = \sqrt{\frac{\sum_E \int_E \left(\boldsymbol{u} - \Pi_{\mathcal{P}}\boldsymbol{u}^h\right)^{\mathsf{T}} \left(\boldsymbol{u} - \Pi_{\mathcal{P}}\boldsymbol{u}^h\right) \,\mathrm{d}\boldsymbol{x}}{\sum_E \int_E \boldsymbol{u}^{\mathsf{T}}\boldsymbol{u}\,\mathrm{d}\boldsymbol{x}}}, \tag{59}$$

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

$$\frac{||\boldsymbol{u} - \Pi_{\mathcal{P}}\boldsymbol{u}^h||_{H^1(\Omega)}}{||\boldsymbol{u}||_{H^1(\Omega)}} = \sqrt{\frac{\sum_E \int_E \left(\varepsilon(\boldsymbol{u}) - \varepsilon(\Pi_{\mathcal{C}}\boldsymbol{u}^h)\right)^{\mathsf{T}} \boldsymbol{D} \left(\varepsilon(\boldsymbol{u}) - \varepsilon(\Pi_{\mathcal{C}}\boldsymbol{u}^h)\right) \,\mathrm{d}\boldsymbol{x}}{\sum_E \int_E \varepsilon(\boldsymbol{u})^{\mathsf{T}}\boldsymbol{D}\varepsilon(\boldsymbol{u})\,\mathrm{d}\boldsymbol{x}}}, \tag{60}$$

where the strain appears in Voigt notation and $\varepsilon(\Pi_{\mathcal{C}}\boldsymbol{u}^h) = \boldsymbol{\nabla}_{\mathrm{S}}(\Pi_{\mathcal{C}}\boldsymbol{u}^h) = \widehat{\varepsilon}(\boldsymbol{u}^h)$ (see (3) and (20)).

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

$$\boldsymbol{H}_{\mathcal{R}} = \left[\; (\boldsymbol{H}_{\mathcal{R}})_1 \;\; \cdots \;\; (\boldsymbol{H}_{\mathcal{R}})_a \;\; \cdots \;\; (\boldsymbol{H}_{\mathcal{R}})_N \;\right]^{\mathsf{T}}, \quad (\boldsymbol{H}_{\mathcal{R}})_a = 1, \tag{61}$$

$$\boldsymbol{W}_{\mathcal{R}} = \left[\; (\boldsymbol{W}_{\mathcal{R}})_1 \;\; \cdots \;\; (\boldsymbol{W}_{\mathcal{R}})_a \;\; \cdots \;\; (\boldsymbol{W}_{\mathcal{R}})_N \;\right]^{\mathsf{T}}, \quad (\boldsymbol{W}_{\mathcal{R}})_a = \frac{1}{N}, \tag{62}$$

$$\boldsymbol{H}_{\mathcal{C}} = \left[\; (\boldsymbol{H}_{\mathcal{C}})_1 \;\; \cdots \;\; (\boldsymbol{H}_{\mathcal{C}})_a \;\; \cdots \;\; (\boldsymbol{H}_{\mathcal{C}})_N \;\right]^{\mathsf{T}}, \quad (\boldsymbol{H}_{\mathcal{C}})_a = \left[\; (x_{1a} - \overline{x}_1) \;\; (x_{2a} - \overline{x}_2) \;\right], \tag{63}$$

$$\boldsymbol{W}_{\mathcal{C}} = \left[\; (\boldsymbol{W}_{\mathcal{C}})_1 \;\; \cdots \;\; (\boldsymbol{W}_{\mathcal{C}})_a \;\; \cdots \;\; (\boldsymbol{W}_{\mathcal{C}})_N \;\right]^{\mathsf{T}}, \quad (\boldsymbol{W}_{\mathcal{C}})_a = \left[\; 2q_{1a} \;\; 2q_{2a} \;\right]. \tag{64}$$

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

$$\boldsymbol{K}_E = |E|\, \boldsymbol{W}_{\mathcal{C}}\boldsymbol{W}_{\mathcal{C}}^{\mathsf{T}} + (\boldsymbol{I}_N - \boldsymbol{P}_{\mathcal{P}})^{\mathsf{T}}(\boldsymbol{I}_N - \boldsymbol{P}_{\mathcal{P}}), \tag{65}$$

where $\boldsymbol{I}_N$ is the identity $(N \times N)$ matrix and $\boldsymbol{S}_E = \boldsymbol{I}_N$ has been used in the stability stiffness as this represents a suitable choice for $\boldsymbol{S}_E$ 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 elements. In Veamy, entities such as element, degree of freedom, constraint, among others, are represented by C++ classes.

Veamy uses the following external libraries:

- Triangle [27], a two-dimensional quality mesh generator and Delaunay triangulator.
- 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 mesh generator that is based on the constrained Voronoi diagram. The usage of our polygonal mesh generator is covered in Section 5.

Veamy is free and open source software and is available from Netlib (http://www.netlib.org/numeralgo/) as the na51 package. In addition, a website (http://camlab.cl/software/veamy/) is available, where the software is maintained. After downloading and uncompressing the software, the main directory "Veamy-2.1/" is created. This directory is organized as follows. The source code that implements the VEM is provided in the folder "veamy/" and the subfolders therein. External libraries that are used by Veamy are provided in the folder "lib/." The folder "matplots/" contains MATLAB functions that are useful for plotting meshes and the VEM solution, and for writing a PolyMesher [31] mesh and boundary conditions to a text file that is readable by Veamy. A detailed software documentation with graphical content can be found in the tutorial manual that is provided in the folder "docs/." Several tests are located in the folder "test/." Some of these tests are covered in the tutorial manual and in Section 6 of this paper. Veamy supports Linux and Mac OS machines only and compiles with g++, the GNU C++ compiler (GCC 7.3 or newer versions should be used). The installation procedure and the content that comprises the software are described in detail in the README.txt file (and also in the tutorial manual), which can be found in the main directory.

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.

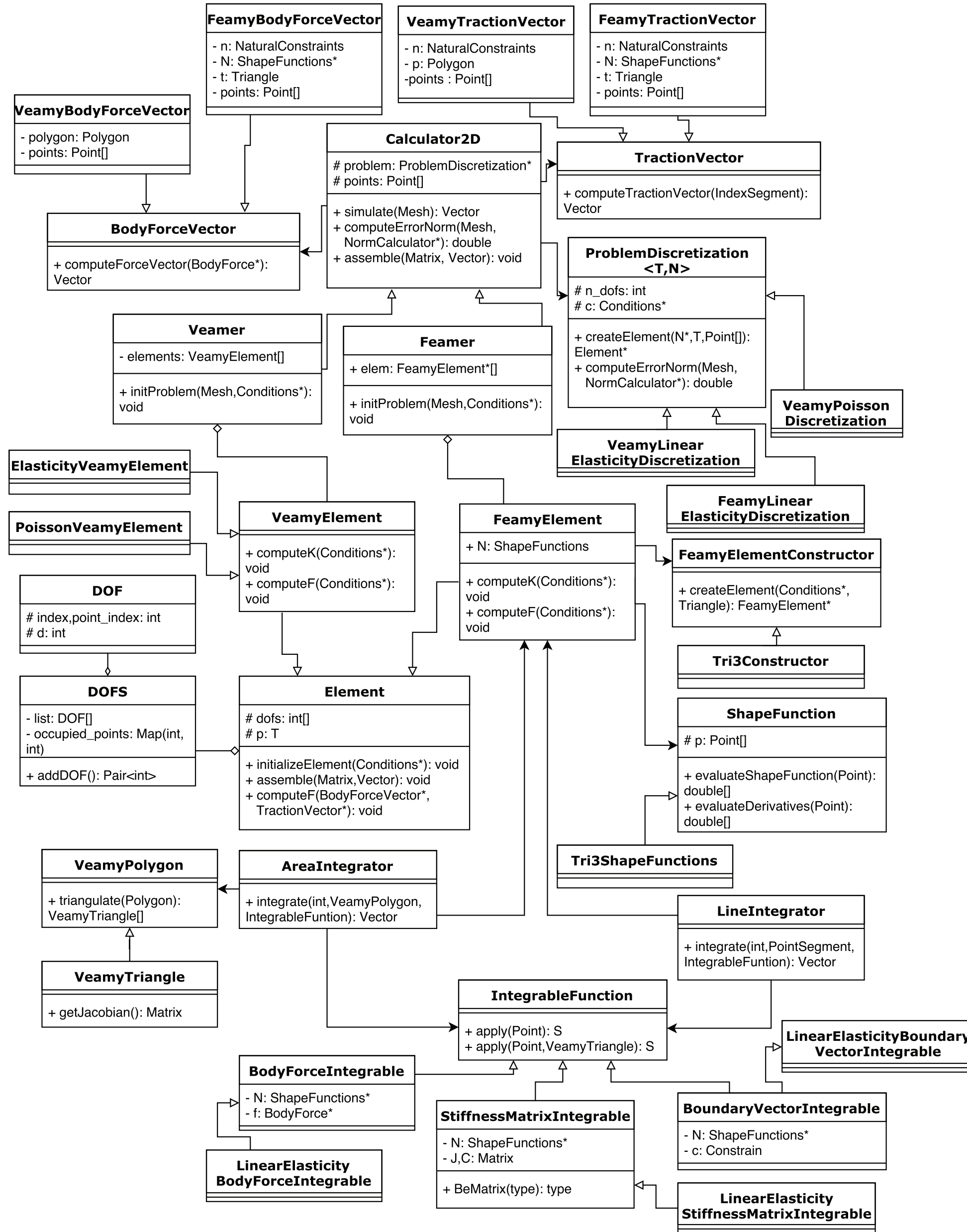


Fig. 2 UML diagram for the `Veamy` library. VEM and FEM modules

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 3.4 and 3.5.

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

$\boldsymbol{H}_{\mathcal{R}} = \mathbf{0}$, $\boldsymbol{W}_{\mathcal{R}} = \mathbf{0}$, $\boldsymbol{H}_{\mathcal{C}} = \mathbf{0}$, $\boldsymbol{W}_{\mathcal{C}} = \mathbf{0}$
for each node in the polygonal element **do**
  Get incident edges
  Compute the unit outward normal vector to each incident edge
  Compute $(\boldsymbol{H}_{\mathcal{R}})_a$ and $(\boldsymbol{H}_{\mathcal{C}})_a$, and insert them into $\boldsymbol{H}_{\mathcal{R}}$ and $\boldsymbol{H}_{\mathcal{C}}$, respectively
  Compute $(\boldsymbol{W}_{\mathcal{R}})_a$ and $(\boldsymbol{W}_{\mathcal{C}})_a$, and insert them into $\boldsymbol{W}_{\mathcal{R}}$ and $\boldsymbol{W}_{\mathcal{C}}$, respectively
end
Compute $\boldsymbol{I}_{2N}$, $\boldsymbol{P}_{\mathcal{R}}$, $\boldsymbol{P}_{\mathcal{C}}$, $\boldsymbol{P}_{\mathcal{P}}$, $\boldsymbol{D}$
Compute $\boldsymbol{S}_E$
Output: $\boldsymbol{K}_E = |E|\, \boldsymbol{W}_{\mathcal{C}}\, \boldsymbol{D}\, \boldsymbol{W}_{\mathcal{C}}^{\mathsf{T}} + (\boldsymbol{I}_{2N} - \boldsymbol{P}_{\mathcal{P}})^{\mathsf{T}}\, \boldsymbol{S}_E\, (\boldsymbol{I}_{2N} - \boldsymbol{P}_{\mathcal{P}})$

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 the 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 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.

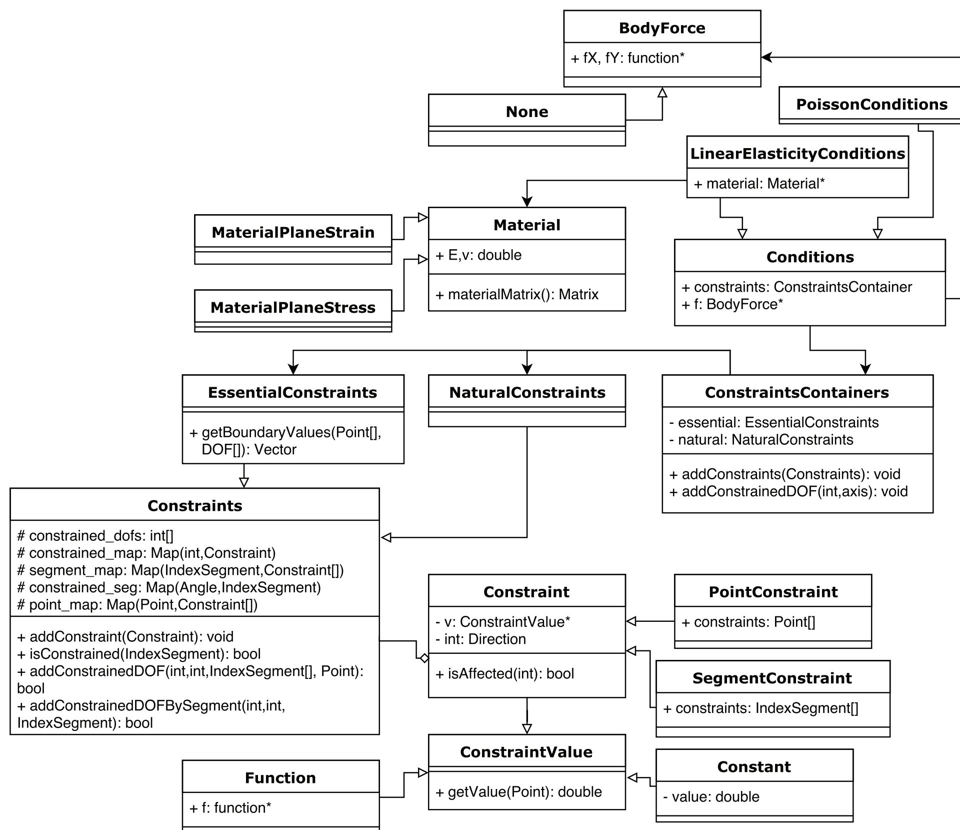


Fig. 3 UML diagram for the `Veamy` library. Problem conditions

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) bound-

ary 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`, respectively, 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 `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 quadrature 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

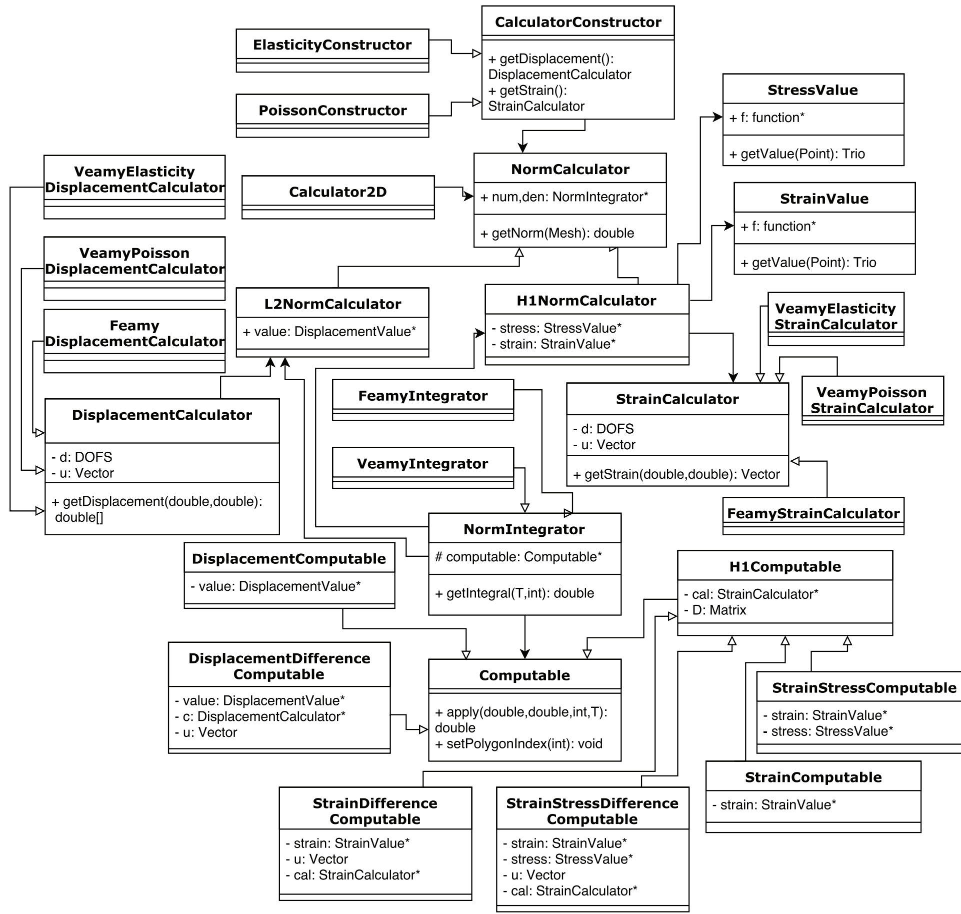


Fig. 4 UML diagram for the `Veamy` library. Computation of the L^2-norm and H^1-seminorm of the error

procedure 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.

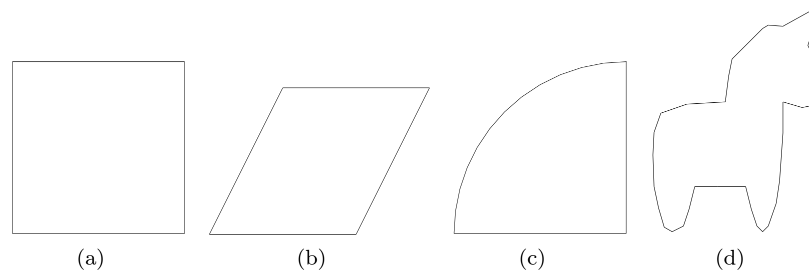


Fig. 5 Domain examples. (a) Square domain, (b) rhomboid domain, (c) quarter circle domain, (d) unicorn-shaped domain

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.

```
std::vector<Point> square_points = {Point(0,0), Point(10,0), Point(10,10), Point(0,10)};
Region square(square_points);
std::vector<Point> qc_points = {Point(0,0), Point(10,0), Point(10,10)};
std::vector<Point> quarter = delynoi_utilities::generateArcPoints(Point(10,0), 10, 90.0, 180.0);
qc_points.insert(quarter_circle_points.end(), quarter.begin(), quarter.end());
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.

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

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

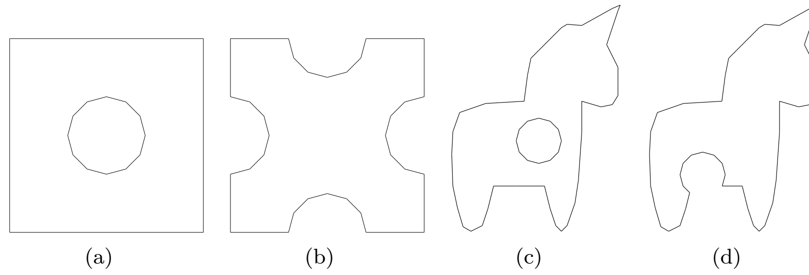


Fig. 6 Examples of domains with holes. (a) Square with an inner hole, (b) square with four intersecting holes, (c) unicorn-shaped domain with an inner hole, (d) unicorn-shaped domain with an intersecting hole

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.

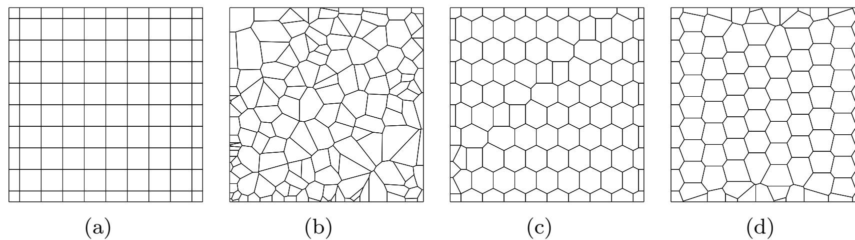


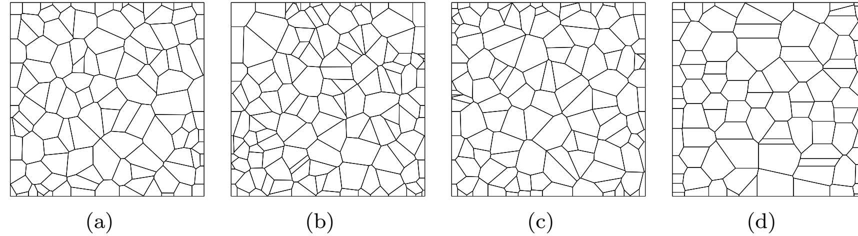
Fig. 7 Polygonal mesh generation on a square domain using different rules. (a) `constant`, (b) `random_double`, (c) `ConstantAlternating`, (d) `sine`

```
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 the code to add random noise to the constant generation rule on a square domain.



(a) (b) (c) (d)

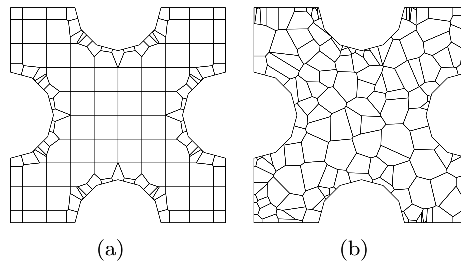
Fig. 8 Polygonal mesh generation on a square domain using different rules with random noise. (a) constant with noise, (b) random_double with noise, (c) ConstantAlternating with noise, (d) sine with noise

```
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.



(a) (b)

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

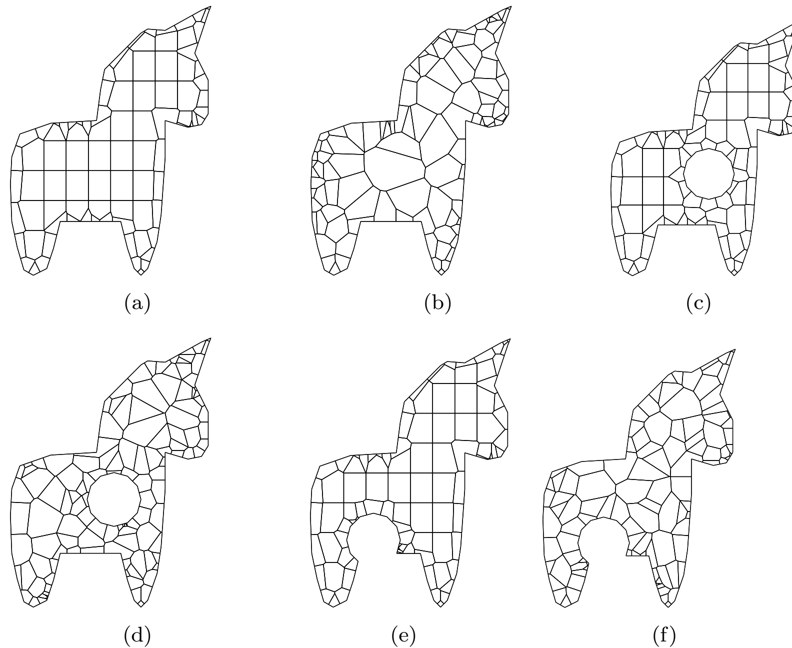


Fig. 10 Examples of polygonal meshes in complicated domains. Unicorn-shaped domain with (a) `constant` generation rule, (b) `random_double` generation rule, (c) inner hole and `constant` generation rule, (d) inner hole and `random_double` generation rule, (e) intersecting hole and `constant` generation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

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`. The setup file for each of the examples that are considered in this section is included in the folder "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 "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 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]:

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

where $\overline{E}_Y = E_Y/\left(1-\nu^2\right)$ with the Young's modulus set to $E_Y = 1\times 10^7$ psi, and $\overline{\nu} = \nu/(1-\nu)$ 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.

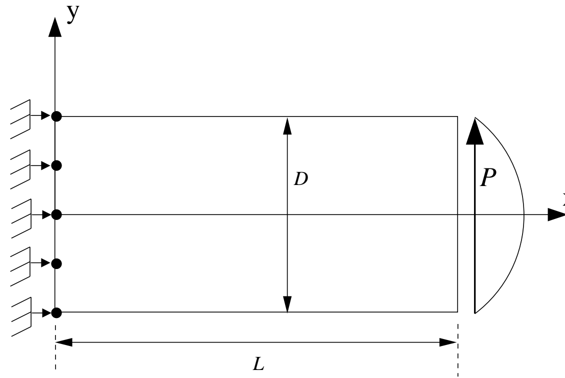


Fig. 11 Model geometry and boundary conditions for the cantilever beam problem

6.1.1 Setup file

The setup instructions for this problem are provided in the main C++ file "ParabolicMain.cpp" that is located in the folder "test/." Additionally, the structure of the setup file is explained in detail in the tutorial manual that is located in the folder "docs/." The interested reader is referred therein to learn more about this setup file.

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 displacements using a MATLAB function written for this purpose. This MATLAB function is provided in the folder "matplots/" 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.

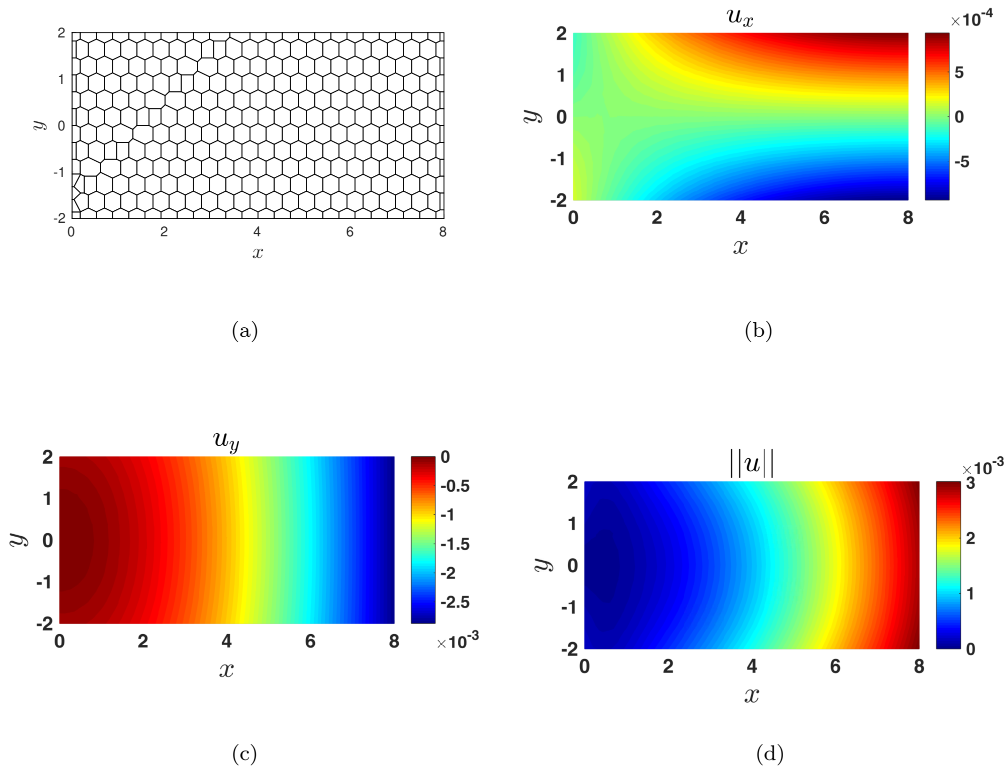


Fig. 12 Solution for the cantilever beam subjected to a parabolic end load using `Veamy`. (a) Polygonal mesh, (b) VEM horizontal displacements, (c) VEM vertical displacements, (d) norm of the VEM displacements

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 "test/" and named as "ParabolicMainVEMnorms.cpp" for the VEM and "FeamyParabolicMainNorms.cpp" for the FEM using three-node triangles ($T3$). The meshes used for these tests were written to text files, which are located in the folder "test/test_files/." `Veamy` implements a function named `createFromFile` that is used to read these mesh files. 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.

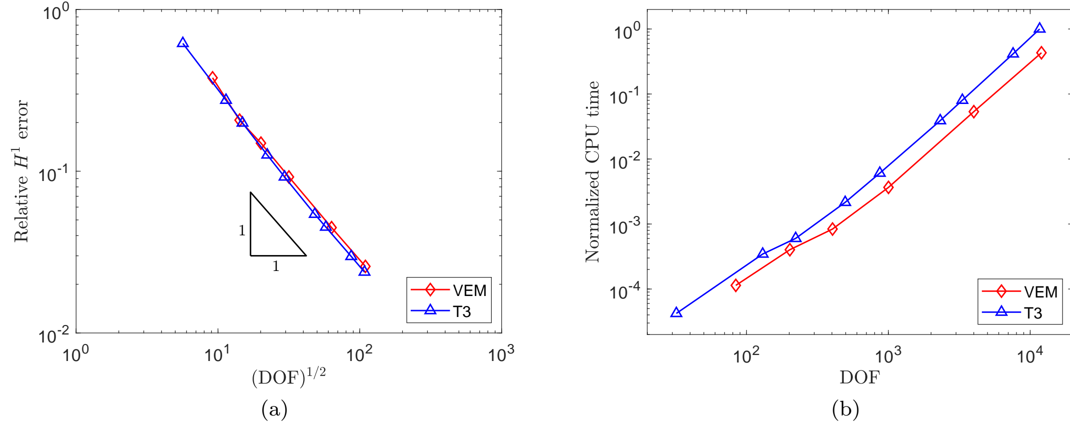


Fig. 13 Cantilever beam subjected to a parabolic end load. Performance comparison between the VEM (polygonal elements) and the FEM (three-node triangles ($T3$)). (a) H^1-seminorn of the error as a function of the number of degrees of freedom and (b) normalized CPU time as a function of the number of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

6.2 Using a `PolyMesher` mesh

In order to conduct a simulation in `Veamy` using a `PolyMesher` mesh, a MATLAB function named `PolyMesher2Veamy`, which needs to be called within `PolyMesher`, was especially devised to read this mesh and write it to a text file that is readable by `Veamy`. This MATLAB function is provided in the folder "matplots/." 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. 14, 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 file containing the translated mesh with boundary conditions is provided in the folder "test/test_files/" under the name "`polymesher2veamy.txt`." The complete setup instructions for this problem are provided in the file "PolyMesherMain.cpp" that is located in the folder "test/." The polygonal mesh and the VEM solution are presented in Fig. 15.

6.3 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 "test/." The following material parameters are considered: $E_Y = 250$ MPa,

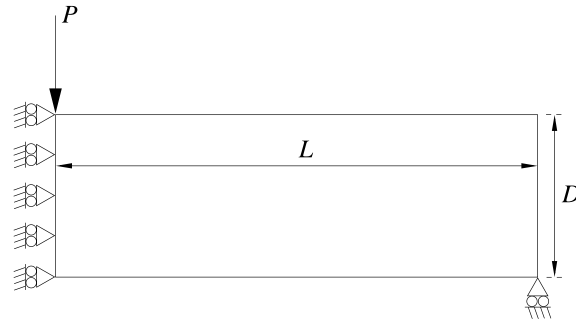


Fig. 14 MBB beam problem definition as per Section 6.1 in Ref. [31]

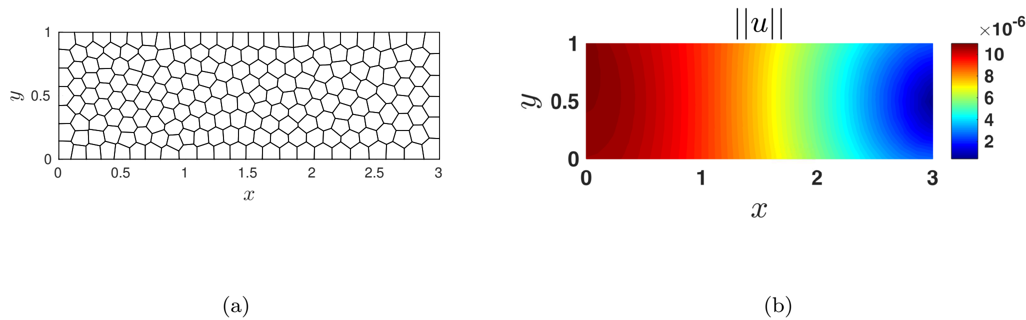


Fig. 15 Solution for the MBB beam problem using Veamy. (a) Polygonal mesh and (b) norm of the VEM displacements

$\nu = 0.3$ and plane strain condition is assumed. The model geometry, the polygonal mesh and boundary conditions, and the VEM solution are presented in Fig. 16.

6.4 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 “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, the polygonal mesh and boundary conditions, and the VEM solution are shown in Fig. 17.

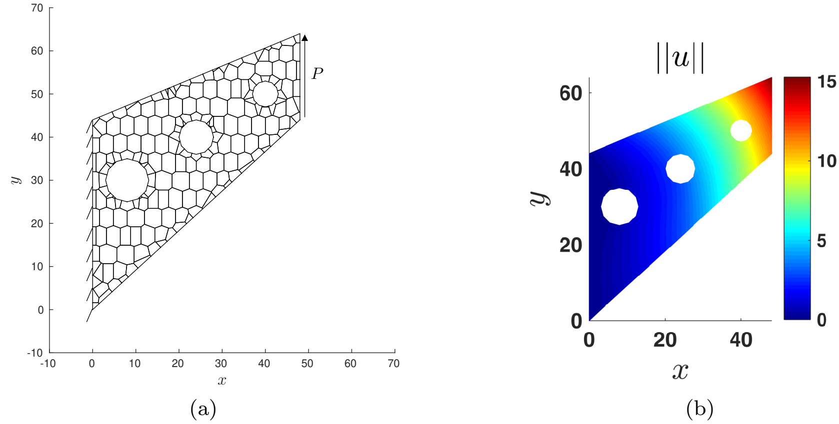


Fig. 16 Solution for the perforated Cook's membrane problem using `Veamy`. (a) Model geometry, polygonal mesh and boundary conditions, and (b) norm of the VEM displacements

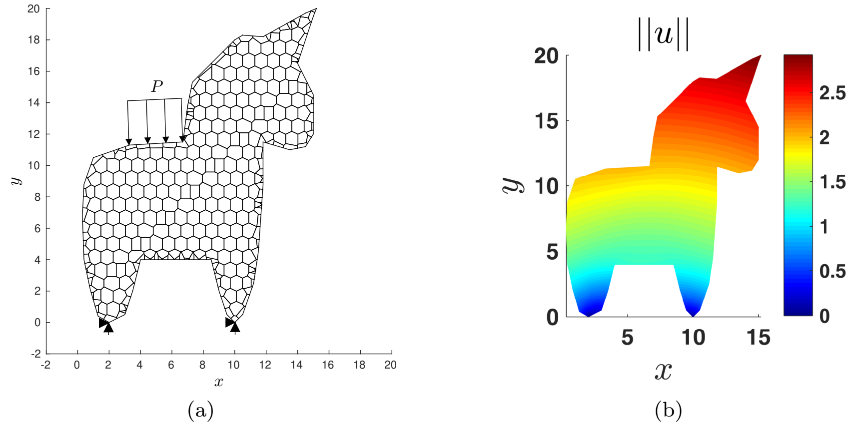


Fig. 17 Solution for the toy problem using `Veamy`. (a) Model geometry, polygonal mesh and boundary conditions, and (b) norm of the VEM displacements

6.5 Poisson problem with a manufactured solution

We conclude the examples by solving a Poisson problem with a source term given by $f(\boldsymbol{x}) = 32y(1-y)+32x(1-x)$, which is the outcome of letting the solution field be $u(\boldsymbol{x}) = 16xy(1-x)(1-y)$. A unit square domain is considered and $u(\boldsymbol{x})$ is imposed along the entire boundary of the domain resulting in the essential (Dirichlet) boundary condition $g(\boldsymbol{x}) = 0$. The complete setup instructions for this problem are provided in the file "PoissonSourceTestMain.cpp" that is located in the folder "test/." The polygonal mesh and the VEM solution are shown in Fig. 18. The relative L^2-norm of the error and the relative H^1-seminorm of the error obtained for the mesh shown in Fig. 18(a) are 2.6695×10^{-3} and 6.7834×10^{-2}, respectively.

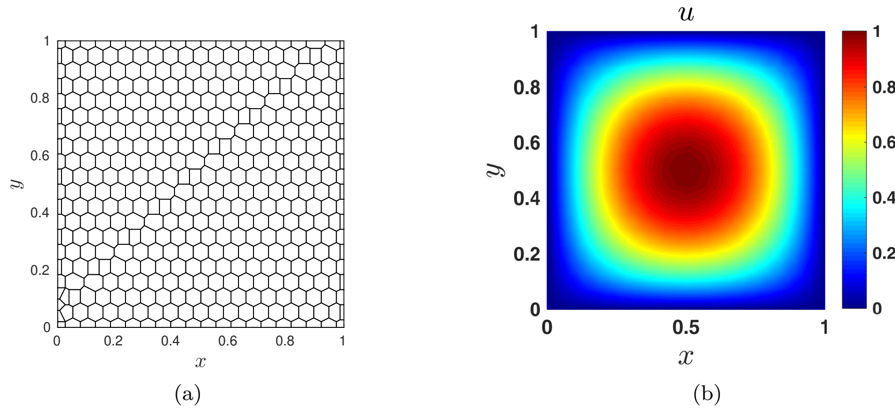


Fig. 18 Solution for the Poisson problem using `Veamy`. (a) Polygonal mesh and (b) VEM solution. The relative L^2-norm of the error is 2.6695×10^{-3} and the relative H^1-seminorm of the error is 6.7834×10^{-2}

7 Concluding remarks

In this paper, an object-oriented C++ library for the virtual element method was presented for the linear elastostatic and Poisson problems in two-dimensions. The usage of the library, named `Veamy`, was demonstrated through several examples. Possible extensions of this library that are of interest include three-dimensional linear elastostatics, where an interaction with the polyhedral mesh generator Voro++ [26] seems very appealing, and nonlinear solid mechanics [3, 8, 37, 38]. `Veamy` is free and open source software.

Acknowledgements AOB acknowledges the support provided by Universidad de Chile through the "Programa VID Ayuda de Viaje 2017" and the Chilean National Fund for Scientific and Technological Development (FONDECYT) through grant CONICYT/FONDECYT No. 1181192. The work of CA is supported by CONICYT-PCHA/Magíster Nacional/2016-22161437. NHK is grateful for the support provided by the Chilean National Fund for Scientific and Technological Development (FONDECYT) through grant CONICYT/FONDECYT No. 1181506.

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. **60**(4), 643–657 (2017)
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)
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)
42. Cangiani, A., Georgoulis, E.H., Pryer, T., Sutton, O.J.: A posteriori error estimates for the virtual element method. Numer. Math. **137**(4), 857–893 (2017)