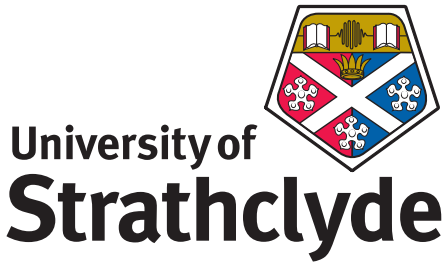


Bernstein-Bézier Techniques and Optimal Algorithms in Finite Element Analysis

Miangaly Gaëlle Andriamaro

Department of Mathematics and Statistics

University of Strathclyde

Glasgow, UK

December 2013

This thesis is submitted to the University of Strathclyde for the degree of Doctor of Philosophy in the Faculty of Science.

Acknowledgements

I would like to express the deepest appreciation to both of my supervisors, Dr Oleg Davydov and Prof. Mark Ainsworth, for their insightful advice and helpful guidance with this dissertation.

I would also like to thank my family and friends, for their encouragement and support throughout my PhD studies.

Many thanks to ORSAS and the University of Strathclyde for their financial support. I am grateful for the Technology Strategy Board for supporting my internship at Cobham CTS, and for the valuable contribution of John Simkin with the finite element simulations given in Chapter 5.

Abstract

This thesis focuses on the development of algorithms for the efficient computation of the element system matrices associated with simplicial elements of arbitrary polynomial order. The algorithms make use of properties intrinsic to Bernstein-Bézier polynomials, allowing for sum factorizations techniques to be applicable. In particular, the presented algorithms are the first to achieve optimal complexity for H^1 elements on simplicial partitions in $\mathbb{R}^d$, for $d = 1, 2, 3$. The optimal complexity result is extended to two-dimensional vector finite elements. The presented Bernstein-Bézier basis for vector finite elements presents a clear distinction between the gradient and rotational components of the vector field. Numerical results illustrate the optimal cost associated with the computation of the elemental matrices, as well as the efficiency of the Bernstein-Bézier elements. The thesis contains the documentation of `BBFEM`, a `C++` implementation of the newly developed algorithms which is available under a GNU General Public Licence. In addition, a report on the work done for a shorter Knowledge Transfer Partnership project on edge elements, with Cobham Technical Services, is also included.

Contents

Chapter 1

Introduction

The finite element method (FEM) is the most widely used tool for the discretization and numerical solution of the problems appearing in engineering design and analysis. In the finite element method, the choice of the shape functions is crucial for the stability and efficiency of the procedures involved in the computation of the approximated solution. For the general theory of FEM, we refer the reader to [31] and [33]. The solution is approximated by spaces of piecewise polynomials defined over a partition. In the conventional definition of FEM, the polynomial order is fixed at a low value, whereas the mesh is successively refined. In the p-version of FEM [23], the initial mesh is maintained, while the polynomial order of approximation is allowed to increase.

The standard finite elements consist of piecewise continuous polynomials, and hold a great variety of corresponding H^1-conforming shape functions in the literature. The most widely used are the Lagrange shape functions. They are very useful for low-order FEM. However, due to the "oscillating" aspect of these functions, they are not recommended for higher order. In [75], Szabò and Babuška defined their shape functions as integrals of Legendre polynomials, yielding sparse and well-conditioned stiffness matrices, thereby suitable for high order FEM. Dubiner [39] constructed shape functions on triangles by means of the Duffy transformation. These shape functions present the advantage of having fast integration properties. Using a similar approach, Sherwin and Karniadakis [59] extended

the construction to tetrahedra. Towards optimal complexity for setting up the stiffness matrices, Eibner and Melenk [41] adapted their shape functions to the quadrature formulas. Some authors gave a prominent place to the construction of hierarchical bases. Ainsworth and Coyle [14] introduced a hierarchical basis for tetrahedral elements, using an intrinsic orientation of edges, faces and elements in order to ensure conformity. In [32], Carnevali presented a new hierarchical basis for triangles and tetrahedra such that edge, face and region functions are orthogonal to those of degree at most $p-2$, $p-3$ and $p-4$, respectively. Adjerid *et al* [8] introduced shape functions with better conditioning than both Szabò and Babuška [75] and Carnevali [32], using a particular orthogonalization of the Szabò-Babuška basis. In [27], Bittencourt constructed shape functions using tensor products of one-dimensional shape functions. Arnold *et al* use Bernstein polynomials in their finite element construction [19].

It has been established [74, 35] that standard H^1-conforming finite elements lead to spurious solutions in electromagnetics. Indeed, on a general mesh, the continuity condition imposed by the H^1-shape functions distort the kernel in such a way that the corresponding zero eigenvalues are shifted to non-zero values which have no physical significance. When solving discrete variational problems in electromagnetics, *tangential* finite elements are the natural choice (see [28] and the references therein). In constrast to H^1, $H(\mathrm{curl})$-conformity amounts to only tangential continuity accross element interfaces. Tangential finite elements were introduced by Nédelec in [67], where he presented (Nédelec) spaces for $H(\mathrm{curl})$-conforming functions. The basis which corresponds to the lowest order Nédelec space consists of the *Whitney forms* [29]. Even if we restrict our attention to the case of two-dimensional $H(\mathrm{curl})$-conforming finite elements defined on triangular partitions, the literature presents a rich variety of generalizations of the Whitney forms to higher order. In [70], Ren and Ida use the framework of differential forms to construct their high order vector shape functions. Using the different geometries (vertex, edge, facet) of the simplex, they analyze the assigment of the degrees of freedom on each simplex, and derive a general procedure for generating $H(\mathrm{curl})$-conforming bases. From a similar geometric partition of the triangle,

Gopalakrishnan *et al* [48] design a basis obtained from some characterizations of the homogeneous part of the Nédelec space. Their basis is solely expresssed in terms of barycentric coordinates. Their construction turns out to coincide with a particular application of the later published work in [19]. In [19], Arnold *et al* generalize the classical finite element spaces to any dimension and any order of the differential forms. Using the concept of a *consistent family of extension operators*, they derive geometric decompositions of the polynomial spaces, yielding explicit local bases for them. When p-adaption is needed, *hierarchical* bases are often preferred. Motivated by the unified approach presented in [53], Hiptmair describes in [54] hierarchical bases for high order differential forms of arbitrary polynomial order. In [77], Webb gives explicit formulas for hierarchical shape functions in terms of barycentric coordinates, while maintaining the separation of gradient and rotational spaces. He then suggests a partial orthogonalization of the presented basis in order to address ill-conditioning. An alternative orthogonalization procedure is proposed in [6] for a generic hierarchical basis. In contrast to the *a posteriori* orthogonalization proposed in [77] and in [6], Graglia *et al* construct in [49] a new family of hierarchical vector bases, using the same technique as in [50], but with the generating scalar interpolatory polynomials replaced by orthogonal polynomials. In [12], Ainsworth and Coyle design hierarchical basis functions based on Legendre polynomials which are suitable for hybrid quadrilateral/triangular partitions. Their analysis includes the matrix conditioning as well as the dispersive behaviour of the presented elements. Schöberl and Zaglmayr introduce in [71] a set of hierarchical conforming basis functions based on a tensorial construction [59]. Their basis satisfy the *local complete property*, in that the subspaces associated with each geometric block form a complete sequence. Inspired by [12] and [71], Xin and Cai construct in [80] a hierarchical basis with improved matrix conditioning. The key idea is to allocate sets of shape functions to each geometry, in such a way that the shape functions associated with the same geometry are mutually orthogonal. Using an approach based on "small simplices" obtained using affine contractions of the mesh simplex, Rapetti presents in [69] a system for generating high-order Whitney shape functions.

Bernstein basis representation of polynomials on simplices is widely used in surface modeling and approximation theory [37, 44, 56, 64], where the computational methods based on this representation are usually referred to as *Bernstein-Bézier techniques*. They feature stable and efficient evaluation, differentiation and integration algorithms. In contrast to the Lagrange basis functions, Bernstein polynomials are non-negative and support shape preservation, which makes them a standard tool for computer aided design. (For example, the well known Bézier curves are based on univariate Bernstein polynomials.) Note that Bernstein bases have optimal condition numbers among all nonnegative bases on an interval, see [45]. Finally, the restriction of a Bernstein polynomial to any facet of the simplex either is identically zero or coincides with a lower dimensional Bernstein polynomial, which makes it easy to impose essential boundary conditions as well as various interelement conformity conditions. Smoothness conditions between Bernstein-Bézier triangles can be exploited to easily generate elements of higher regularity [64, 57]. These impressive properties make shape functions generated by Bernstein polynomials a promising tool for the finite element analysis, as suggested e.g. in [72, 56]. Recently, Arnold *et al* [19] have suggested to use Bernstein polynomials to construct bases for the finite element spaces for vector fields. However, Bernstein-Bézier finite elements remain relatively little known to the practitioners of FEM.

The purpose of this work is to exploit the desirable properties of the Bernstein polynomials in order to generate H^1 and $H(\mathrm{curl})$ conforming elements which produce fast and efficient numerical procedures. Taking account of the numerical quadrature cost associated with higher order elements, *optimal* complexity is obtained, in the sense that the required number of operations is of the same order as the number of entries that needs to be computed. More precisely, the obtained complexity amounts to each system matrix entry being computed with $\mathcal{O}(1)$ operations.

Naturally, the efficient evaluation and visualisation of a finite element approximation is but a single part of the overall finite element procedure: one must also

assemble the so-called *system matrix* and *load vector*, and solve the resulting system. The cost of assembling the stiffness matrix of an n^{th} order standard finite element on simplicial elements is at best generally found to be $\mathcal{O}(n^{2d+1})$ in d dimensions [59, Chapter 2, Chapter 4]. Indeed, the number of shape functions is $\mathcal{O}(n^d)$, and $\mathcal{O}(n^d)$ quadrature points are required in order to get a sufficiently accurate quadrature rule. With no a-priori information on the shape functions, this would lead to a $\mathcal{O}(n^{3d})$ cost. The ability to use sum factorizations techniques relies on a tensorial structure of the shape functions. Quadrilaterals and hexahedrons have a tensor product structure, and thus shape functions are naturally contructed with a tensor product structure. Efficient finite element procedures on quadrilaterals and hexahedrons are given in [47]. Tensorial construction of shape functions on simplices are presented in [27, 39, 60, 61]. Combining a tensor product structure of the basis functions with sum factorization techniques, Karniadakis and Sherwin [60] design algorithms which achieve the lower complexity $\mathcal{O}(n^{2d+1})$ for the construction of elemental matrices on simplicial elements. This complexity is only realized by using special choices of bases. In fact, assuming that each system matrix entry is to be computed with at least $\mathcal{O}(1)$ operations, the best possible complexity is $\mathcal{O}(n^{2d})$ for elements based on the local polynomial space $\mathbb{P}_d^n$ [41, Section A], but to-date there is no known algorithm by which this can be achieved. Eibner and Melenk do actually present an algorithm achieving the optimal order $\mathcal{O}(n^{2d})$, but this comes at the price of using a significantly larger local space (and hence more unknowns) than the space $\mathbb{P}_d^n$ consisting of polynomials of degree at most n in d dimensions, without a corresponding increase in convergence rate. Using special block structures arising in vectors and matrices representing polynomials written in their Bernstein-decomposition, Kirby develops in [62] fast and efficient algorithms based on Bernstein polynomials for constant coefficient H^1 finite elements. The presented algorithms compute the action of the elemental matrices on a vector, and thus are matrix-free. More recently, using a quadrature based on warped Gauss points, his results are extended to variable data in [63].

The thesis is organized as follows. Notations and preliminary results are

introduced in the remaining sections of this chapter. Chapter 2 then focuses on the efficient computation of the so-called *B-moments*, which is fundamental to the optimal complexity results obtained for the computation of element matrices associated with variable data. The key point consists in taking advantage of a tensor product structure which arises from the Duffy transformation, when applied to the Bernstein polynomials. Chapter 3 introduces the Bernstein shape functions associated with the H^1 finite elements, and details efficient and ready-to-implement algorithms for computing the corresponding element matrices. The associated computational costs are shown to be of optimal order in one, two and three dimensions. Chapter 3 concludes with numerical results which are consistent with the expected optimal complexity. Chapter 4 focuses on the Bernstein-Bézier shape functions associated with tangential finite elements in two dimensions. The presented basis features an explicit separation of the gradient and gradient-free shape functions, which allows for an easy projection onto the space orthogonal to the kernel of the curl operator. In addition, the optimal complexity results of Chapter 3 are extended to the vector finite elements by means of a sparse transformation into B-form. Chapter 4 includes numerical results which confirm the predicted optimal complexity, and illustrate the efficiency of the presented vector finite elements. Chapter 5 gives a brief report on the work I have done during a short Knowledge Transfer Partnership internship at Cobham Technical Services CTS Ltd, and may be read independently from the previous chapters. The internship was focused on improving the efficiency of tangential elements on non-affine meshes, as proposed in [43, 25]. Appendix A contains explicit formulas for low-order element system matrices associated with Bernstein-Bézier elements, as well as the graphs of the low-order $H(\mathrm{curl})$ shape functions illustrating the vector elements introduced in Chapter 4. Appendix B contains the documentation of the `C++` library `BBFEM` for the H^1 Bernstein finite elements in two and three dimensions, and $H(\mathrm{curl})$ finite elements in two dimensions. Examples illustrating the use of the library are also included. Appendix C discusses the shift strategy used for solving the Maxwell's generalized eigenvalue problem of Chapter 4.

1.1 Notations

Standard multi-index notations will be used throughout. Hence,

$$\mathbf{x}^{\boldsymbol{\beta}} := x_1^{\beta_1} \dots x_d^{\beta_d}, \quad \mathbf{x} \in \mathbb{R}^d,\ \boldsymbol{\beta} \in \mathbb{Z}_+^d,$$

$$\binom{\boldsymbol{\alpha}}{\boldsymbol{\beta}} := \prod_{j=1}^{d} \binom{\alpha_j}{\beta_j}, \quad \boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{Z}_+^d, \qquad \binom{n}{\boldsymbol{\beta}} := \frac{n!}{\prod_{j=1}^{d} \beta_j!}, \quad n \in \mathbb{Z}_+,\ \boldsymbol{\beta} \in \mathbb{Z}_+^d.$$

For $n \in \mathbb{Z}_+$, the space of polynomials of degree at most n in d variables will be denoted by $\mathbb{P}_d^n$. Then $\mathbb{P}_d^n$ is generated by the monomials $\mathbf{x}^{\boldsymbol{\beta}}$, with $\boldsymbol{\beta} \in \mathbb{Z}_+^d$, and satisfying $|\boldsymbol{\beta}| := \sum_{j=1}^{d} \beta_j \le n$, so that

$$\dim \mathbb{P}_d^n = \binom{n+d}{d}, \quad n \in \mathbb{Z}_+. \tag{1.1}$$

The equation $K = \mathcal{O}(n^k)$ means that $K(n)$ is equivalent to a polynomial of degree k, that is, there exists such a polynomial p satisfying $\lim K(n)/p(n) = 1$, as $n \to \infty$. Boldface symbols will only refer to vectors, with the exception of subscripts and superscripts. Also, $\mathbf{0}$ may either refer to the null vector or the null matrix, depending on the context. For simplicity, the notation $|\cdot|$ will be used for different purposes. Thus, $|S|$ will denote the cardinality of a finite set S, while $|K|$ will refer to the d-dimensional measure of a domain $K \subset \mathbb{R}^d$. Moreover, for $\rho \in \mathbb{Z}_+^\ell$, $\ell = 1, 2, \dots$, we set $|\rho| := \sum_{k=1}^{\ell} \rho_k$ as already used above. The notation $(\cdot)^{\mathrm{t}}$ refers to the transpose operator.

For n and $d \in \mathbb{Z}_+$, we use the notation

$$\mathcal{I}_d^n := \left\{ \boldsymbol{\rho} \in \mathbb{Z}_+^{d+1} :\ |\boldsymbol{\rho}| = n \right\}. \tag{1.2}$$

The symbol T is used to denote a non-degenerate d-simplex with vertices $\mathbf{v}_i$, $i = 1, \dots, d+1$, that is, $T := \langle \mathbf{v}_i,\ i = 1, \dots, d+1 \rangle \subset \mathbb{R}^d$, with $\langle \cdot \rangle$ denoting the

convex hull operator. With f and g being square-integrable, we define the inner product

$$(f,g) = (f,g)_T := \int_T f(\mathbf{x}) \cdot g(\mathbf{x}) \mathrm{d}\mathbf{x}, \quad d = 1,2,3. \tag{1.3}$$

We define the set of Bernstein-Bézier *domain points* $\mathcal{D}_d^n(T)$ associated with the simplex T by

$$\mathcal{D}_d^n(T) := \Big\{ \boldsymbol{\xi}_{\boldsymbol{\rho}} := \frac{1}{n} \sum_{j=1}^{d+1} \rho_j \mathbf{v}_j : \ \boldsymbol{\rho} \in \mathcal{I}_d^n \Big\}. \tag{1.4}$$

Given a point $\mathbf{v} \in \mathbb{R}^d$, the *barycentric coordinates* of $\mathbf{v}$ with respect to the simplex T are given by the unique $(d+1)$-tuple $\boldsymbol{\lambda}_d = \boldsymbol{\lambda} := (\lambda_1, \ldots, \lambda_{d+1})$ such that

$$\mathbf{v} = \sum_{i=1}^{d+1} \lambda_i \mathbf{v}_i, \quad \text{with } \sum_{i=1}^{d+1} \lambda_i = 1. \tag{1.5}$$

The set $\mathcal{I}_d^n$ defined in (1.2) has a one-to-one correspondence with all possible barycentric coordinates of degree n in d dimensions. Note that the cardinality of $\mathcal{I}_d^n$ is $\binom{n+d}{d}$. Occasionally, for convenience, we use the notation

$$i \in \mathcal{I}_d^1 = \{(1,0,\ldots,0),(0,1,\ldots,0),(0,\ldots,0,1)\}$$

instead of $i \in \{1,\ldots,d+1\}$ with a natural correspondence. In particular, for $d = 2$ we have $\lambda_{100} := \lambda_1$, $\lambda_{010} := \lambda_2$, $\lambda_{001} := \lambda_3$.

The Bernstein polynomials of degree n in d variables associated with T are defined by

$$B_{\boldsymbol{\eta}}^n = B_{\boldsymbol{\eta}}^{n,T} := \binom{n}{\boldsymbol{\eta}} \boldsymbol{\lambda}^{\boldsymbol{\eta}}, \quad \boldsymbol{\eta} \in \mathcal{I}_d^n, \ n \in \mathbb{Z}_+. \tag{1.6}$$

It is well known that the Bernstein polynomials are linearly independent. Observe

from (1.1) and (1.2) that the cardinality of the set

$$\left\{B^n_{\boldsymbol{\eta}} : \ \boldsymbol{\eta} \in \mathcal{I}^n_d\right\} \tag{1.7}$$

is $\binom{n+d}{d}$, and therefore it forms a basis for $\mathbb{P}^n_d$, see [37]. In particular, every polynomial $p \in \mathbb{P}^n_d$ can be uniquely written in the so-called *B-form*

$$p = \sum_{\boldsymbol{\eta} \in I^n_d} c_{\boldsymbol{\eta}} B^n_{\boldsymbol{\eta}}. \tag{1.8}$$

With $\mathbf{c}_n$ called *B-coefficient* of p, we make the convention that any formula for the B-coefficient sequence $\mathbf{c}_n$ corresponding to a given polynomial of degree n is only valid for $\boldsymbol{\alpha} \in \mathcal{I}^n_d$. That is to say, unless $\boldsymbol{\alpha} \in \mathcal{I}^n_d$, the value of the B-coefficient $c_{\boldsymbol{\alpha}}$ is zero. For the sake of brevity of various formulas, for a given $d+1$-tuple $\boldsymbol{\alpha}$, we also set $B^n_{\boldsymbol{\alpha}} = B^n_{\alpha_1,\alpha_2,\ldots,\alpha_{d+1}} := 0$, if $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_{d+1}) \notin \mathcal{I}^n_d$.

We denote by $L^2(\Omega)$ the space of square-integrable real functions on Ω, with the associated L^2-norm defined by

$$\|f\|_{L^2(\Omega)} := \Big(\int_\Omega |f(\mathbf{x})\Big|^2 \mathrm{d}\mathbf{x}\Big)^{\frac{1}{2}}, \quad f \in L^2(\Omega).$$

For any $s \geq 0$, the space $H^s(\Omega)$ is the subspace of $L^2(\Omega)$ which consists of the set $\{f \in L^2(\Omega) : \ D^{\boldsymbol{\alpha}} f \in L^2(\Omega), \ |\boldsymbol{\alpha}| \leq s\}$, where $D^{\boldsymbol{\alpha}}(\cdot) := \dfrac{\partial^{|\boldsymbol{\alpha}|}(\cdot)}{\partial^{\alpha_1} x_1 \partial^{\alpha_2} x_2 \ldots \partial^{\alpha_d} x_d}$. $H^s(\Omega)$ is endowed with the norm

$$\|f\|_{s,\Omega} := \|f\|_{H^s(\Omega)} := \Big(\sum_{|\boldsymbol{\alpha}| \leq s} \|D^{\boldsymbol{\alpha}} f\|^2_{L^2(\Omega)} \Big)^{\frac{1}{2}}.$$

In particular, $H^0(\Omega) = L^2(\Omega)$, and $H^1(\Omega)$ is the subspace of $L^2(\Omega)$ given by $\{f \in L^2(\Omega) : \ \nabla f \in L^2(\Omega)\}$, and is endowed with the norm

$$\|f\|_{1,\Omega} := \|f\|_{H^1(\Omega)} := \big(\|f\|^2_{L^2(\Omega)} + \|\nabla f\|^2_{L^2(\Omega)} \big)^{\frac{1}{2}}. \tag{1.9}$$

With a slight abuse of notation, the symbol $\|\cdot\|_{L^2(\Omega)}$ in (1.9) refers to the norms in $\big(L^2(\Omega)\big)^d$, with $d = 1, 2$, since ∇f is a vector field. In other words, with

$\mathbf{g} = (g_0, g_1)^{\mathrm{t}} \in (L^2(\Omega))^2$, the L^2-norm of $\mathbf{g}$ is given by $\|\mathbf{g}\|_{L^2(\Omega)} := \|\mathbf{g}\|_{(L^2(\Omega))^2} := \big(\|g_0\|^2_{L^2(\Omega)} + \|g_1\|^2_{L^2(\Omega)}\big)^{\frac{1}{2}}$.

For any vector-valued function $\mathbf{f} := (f_1, f_2)$, we set $\mathbf{f}^\perp := (-f_2, f_1)$, $\operatorname{curl} \mathbf{f} := \partial f_2/\partial x_1 - \partial f_1/\partial x_2$, and for a scalar function a, $\nabla a := (\partial a/\partial x_1, \partial a/\partial x_2)$. Obviously, $\mathbf{x} = (x_1, x_2)$ and $\mathbf{x}^\perp = (-x_2, x_1)$ are vector-polynomials in $(\mathbb{P}_1)^2$. Observe in particular that $\operatorname{curl} \equiv \nabla^\perp \cdot$. For a given polyhedral domain $\Omega \subset \mathbb{R}^2$ and $s \geq 0$, we define the space

$$H^s(\operatorname{curl}; \Omega) := \{\mathbf{v} \in H^s(\Omega) : \ \operatorname{curl}(v) \in H^s(\Omega)\}$$

which is endowed with the norm $\|\cdot\|_{H^s(\operatorname{curl};\Omega)}$ defined by

$$\|\mathbf{u}\|_{H^s(\operatorname{curl};\Omega)} := \big(\|\mathbf{u}\|_{s,\Omega} + \|\operatorname{curl}(\mathbf{u})\|_{s,\Omega}\big)^{\frac{1}{2}}.$$

In particular, with $s = 0$, $H^0(\operatorname{curl}; \Omega)$ consists of

$$H^0(\operatorname{curl}; \Omega) := H(\operatorname{curl}; \Omega) := \{\mathbf{v} \in L^2(\Omega) : \ \operatorname{curl}(\mathbf{v}) \in L^2(\Omega)\}, \tag{1.10}$$

and is endowed with the norm $\|\cdot\|_{H(\operatorname{curl};\Omega)}$ defined by

$$\|\mathbf{u}\|_{H(\operatorname{curl};\Omega)} := \big(\|\mathbf{u}\|^2_{L^2(\Omega)} + \|\operatorname{curl}(\mathbf{u})\|^2_{L^2(\Omega)}\big)^{\frac{1}{2}}, \quad \mathbf{u} \in H(\operatorname{curl}; \Omega). \tag{1.11}$$

Given a normed vector space $(V, \|\cdot\|_V)$, let A denote a bilinear form defined on V. A is *continuous* if there exists a constant $\nu < \infty$ such that, for any $\mathbf{u}, \mathbf{v} \in V$, it holds that $A(\mathbf{u}, \mathbf{v}) \leq \nu \|\mathbf{u}\|_V \|\mathbf{v}\|_V$. In the previous equation, ν is referred to as the *continuity constant*. Moreover, A is termed *coercive* if there exists a constant $\vartheta > 0$ such that, for any $\mathbf{u} \in V$, it holds that $A(\mathbf{u}, \mathbf{u}) \geq \vartheta \|\mathbf{u}\|^2_V$. The scalar ϑ is called the *coercivity* constant.

1.2 The Finite Element Method

In this section, we briefly review the basic concepts as well as some useful results on finite elements. For a more detailed discussion, the reader is referred to [33, 31].

1.2.1 Definitions

For a given polyhedral domain $\Omega \subset \mathbb{R}^d$, let $\Delta = \{T_j\}_{i=1}^N$ denote a *regular* triangulation of Ω, see [64, Section 4.3] and [33]. In particular, $\Omega = \cup_{j=1}^N T_j$, and the intersection of two different simplices of Δ is either empty, or composed of one common facet. Each simplex T in Δ is associated with a local finite element.

Complying with the definition given in [33], a finite element consists of a triple $(T, \mathcal{N}_n(T), \mathcal{P}_n(T))$, where:

- T is a simplex in $\mathbb{R}^d$;
- $\mathcal{P}_n(T)$ is a space of piecewise polynomials defined on T;
- $\mathcal{N}_n(T)$ consists of basis functions for the dual space $\big(\mathcal{P}_n(T)\big)'$.

In particular, every polynomial $p \in \mathcal{P}_n(T)$ is uniquely determined by its *local degrees of freedom* given by the values of the functionals $\lambda(p)$, $\lambda \in \mathcal{N}_n(T)$. In the above definition, $\mathcal{P}_n(T)$ is also called the set of local *shape functions*. The global finite element space V_n arises from the assembly of the local finite elements $(T, \mathcal{N}_n(T), \mathcal{P}_n(T))$ for all $T \in \Delta$.

Finite element discretizations rely on the so-called *variational formulation* which provides the correct mathematical framework in order to ensure the well-posedness of the problem. Typically, the variational formulation of a differential equation reads:

$$\text{Given } F \in V', \text{ find } u \in V \text{ satisfying } a(u, v) = F(v), \quad \forall v \in V, \tag{1.12}$$

where V is a closed subspace of some Hilbert space $(H, (\cdot, \cdot))$, V' denotes the

dual space of V, and $a(\cdot,\cdot)$ represents a continuous and *coercive* bilinear form on V. The Galerkin *conforming* finite-element discretization of the problem (1.12) consists in substituting the continuous space with a finite-dimensional *subspace*, so as to obtain the discrete problem of the form:

$$\begin{aligned}&\text{Given a finite-dimensional space } V_n \subset V \text{ and } F \in V',\\ &\text{find } u \in V \text{ satisfying } a(u_n, v_n) = F(v_n), \quad v \in V_n.\end{aligned}$$

The next section discusses the error bounds for conforming finite elements, when coercive variational problems are considered.

1.2.2 Approximation Properties of Conforming FEM

1.2.2.1 Standard results

The next theorem is a fundamental result behind the theory of FEM. For the details of the proof, see, for example, [31, Theorem 2.7.7] (see also [42, Proposition 2.19]):

Lax-Milgram. *Let $\big(V, (\cdot,\cdot)\big)$ denote a Hilbert space, and suppose that $a(\cdot,\cdot)$ is a continuous and coercive bilinear form on V. Then, for a continuous linear functional $F \in V'$, there exists a unique $u \in V$ which satisfies*

$$a(u, v) = F(v), \quad v \in V.$$

In addition, if V_n is a finite-dimensional subspace of V, then there exists a unique solution $u_n \in V_n$ which satisfies

$$a(u_n, v_n) = F(v_n), \quad v_n \in V_n.$$

Hence, for conforming finite elements associated with Hilbert spaces, the coercivity of the continuous problem automatically ensures the well-posedness of the

discrete variational problem. If the coercivity property holds, the next lemma provides a useful result on the approximation error produced by conforming finite elements:

Céa's Lemma. *Let $(V_n)_n$ denotes a sequence of finite-dimensional subspaces of the Hilbert space $\big(V,(\cdot,\cdot)\big)$. Assuming that the problem (1.12) satisfies the conditions of the Lax-Milgram Lemma, the Galerkin approximation error is bounded by means of*

$$\|u-u_n\|_V \leq \frac{\nu}{\vartheta}\|u-v_n\|_V, \quad \text{for any } v_n \in V_n,$$

where $u \in V$ and $u_n \in V_n$ respectively denote the exact solution to (1.12) and the solution to the corresponding Galerkin discretization. Also, ν and ϑ are the continuity constant and the coercivity constant of $a(\cdot,\cdot)$ on V.

The above lemma shows that, up to a multiplication by a constant, the Galerkin solution $u_n \in V_n$ is the best approximation of u in V with respect to the norm $\|\cdot\|_V$. This motivates the use of finite element spaces with optimal approximation properties. Assuming that the discrete variational formulations are well-posed, the next two sections provide approximation results for conforming finite elements in $H^1(\Omega)$ and $H(\mathrm{curl};\Omega)$.

1.2.2.2 Approximation Properties of Scalar Finite Elements

Let $d \in \{1,2,3\}$. For $n \in \mathbb{Z}_+$, the H^1-conforming global finite element space of order n, defined over a triangulation $\Delta = \{T_j\}_{j=1}^N$, consists of piecewise continuous polynomials of degree at most n in d variables. The next theorem gives a fundamental result [21, Lemma 4.1] on the approximation properties of H^1-conforming finite elements:

Theorem 1.2.1. *Let $s \geq 0$ and $T \in \Delta$. For any integer $n \geq 1$, there exists an operator $\pi_n : H^s(T) \to \mathbb{P}_d^n(T)$ such that, for any $f \in H^s(T)$, it holds that*

$$\|f-\pi_n f\|_{\ell,T} \leq C n^{-(s-\ell)}\|f\|_{s,T}, \quad 0 \leq \ell \leq s, \tag{1.13}$$

where the constant C is independent from f and n. In addition, the operator π_n is polynomial-preserving, in that $\pi_n(f) = f$ for any $f \in \mathbb{P}_d^n(T)$.

Using the above result, it has been shown that for a sufficiently smooth solution, increasing the degree n will improve the convergence rate [21, Theorem 4.8]. In fact, when the exact solution is analytic on the domain Ω, the rate of convergence is exponential with respect to n [22, Section 1].

1.2.2.3 Approximation Properties of Vector Finite Elements

For $n \in \mathbb{Z}_+$, the two-dimensional $H(\mathrm{curl})$-conforming Nédélec space of order n is defined by

$$\mathbb{ND}_n := (\mathbb{P}_n)^2 + \mathbf{x}^\perp \mathbb{P}_n, \qquad n \in \mathbb{Z}_+. \tag{1.14}$$

The global finite element space defined over the triangulation $\Delta = \{T_j\}_{j=1}^N$ consists of polynomials defined on Ω such that their restriction to any simplex $T_j \in \Delta$ belongs to $\mathbb{ND}_n$, and such that tangential continuity is satisfied across the element interfaces. The next result [26, Theorem 5.1] gives a similar result as in Theorem 1.2.1 in the case of high-order edge finite elements:

Theorem 1.2.2. *Let $s > 0$ and $T \in \Delta$. For any integer $n \geq 1$, there exists an operator $\pi_n^{\mathrm{curl}} : H^s(T) \cap H(\mathrm{curl}; T) \to \mathbb{ND}_n$ such that, for any $\mathbf{u} \in H^s(\mathrm{curl}; T)$, it holds that*

$$\|\mathbf{u} - \pi_n^{\mathrm{curl}} \mathbf{u}\|_{H(\mathrm{curl};T)} \leq C n^{-s} \|\mathbf{u}\|_{H^s(\mathrm{curl};T)},$$

where the constant C is independent from $\mathbf{u}$ and n. Moreover, π_n^{curl} preserves vector polynomials, in that $\pi_n^{\mathrm{curl}}(\mathbf{u}) = \mathbf{u}$ for any $\mathbf{u} \in \mathbb{ND}_n$.

Piecewise continuous polynomials and more especially Nédélec polynomial spaces have been established as suitable conforming finite element discretizations of the spaces $H^1(\Omega)$ and $H(\mathrm{curl}; \Omega)$, respectively. In this thesis, we provide

bases for theses spaces using Bernstein polynomials. In particular, the classical results that have been developed on conforming p-FEM automatically apply to our Bernstein finite elements.

Chapter 2

Bernstein-Bézier Moments

When the data is variable, the elemental quantities generally need to be approximated by means of numerical quadratures. In this case, the optimal computations associated with Bernstein-Bézier finite elements are based on the efficient evaluations of the *Bernstein-Bézier moments* (B-moments) defined by

$$\mu_{\boldsymbol{\alpha}}^n(f) := \int_T f(\mathbf{x}) B_{\boldsymbol{\alpha}}^n(\mathbf{x}) \mathrm{d}\mathbf{x}, \quad \boldsymbol{\alpha} \in \mathcal{I}_d^n, \ f \in L^2(\Omega), \tag{2.1}$$

where $T := \mathrm{conv}(\mathbf{v}_i, \ i = 1, \ldots, d+1)$ is a non-degenerate simplex in $\mathbb{R}^d$. In this section, we show that, using the change of variables defined by the Duffy transformation [40, 39], the multivariate Bernstein polynomials are mapped to tensor products of univariate Bernstein polynomials, thereby allowing for *sum factorizations* techniques [41] to be used. An alternative approach based on this chapter has been published in [11]. Both methods yield the optimal complexity $\mathcal{O}(n^{d+1})$ for the computation of the B-moments of order n in d variables. Note that only divisions and multiplications are counted as operations.

2.1 Stroud Conical Product Rule

In order to optimize the numerical cost associated with quadrature for computing integrals of the form (2.1), we want to start with a tensor product structure. To

this end, we resort to the *Duffy transformation* defined by

$$\left.\begin{aligned} \lambda_1 &= t_1, \\ \lambda_2 &= t_2(1-t_1), \\ &\vdots \\ \lambda_d &= t_d(1-t_1)(1-t_2)\dots(1-t_{d-1}) \\ \lambda_{d+1} &= (1-t_1)(1-t_2)\dots(1-t_d), \end{aligned}\right\} \tag{2.2}$$

which maps the unit cell $[0,1]^d$ to the simplex $T = \langle \mathbf{v}_i : \ i = 1, \dots, d+1 \rangle$. Thus, the Duffy transformations maps any point $\mathbf{t} \in [0,1]^d$ to $\mathbf{x}(\mathbf{t})$ given by

$$\mathbf{x}(\mathbf{t}) := \sum_{i=1}^{d+1} \lambda_i \mathbf{v}_i \in T.$$

The Duffy transformation can be used to build simplicial finite elements based on a tensorial construction [59, Section 3.2]. However, it should be pointed out that, in this work, the bases are not constructed as tensor product of polynomials. Instead, the tensor product structure arises from the application of the Duffy transformation to the (multivariate) Bernstein polynomials.

Combining the expression of $\mathbf{x}(\mathbf{t})$ with the definition of the Duffy transformation, we find that the determinant of the Jacobian is given by

$$d!|T|(1-t_1)^{d-1}(1-t_2)^{d-2}\dots(1-t_{d-1}),$$

so that

$$\int_T g(\mathbf{x})\mathrm{d}\mathbf{x} = d!|T| \int_0^1 \mathrm{d}t_1(1-t_1)^{d-1} \int_0^1 \mathrm{d}t_2(1-t_2)^{d-2} \dots \int_0^1 \mathrm{d}t_d (g \circ \mathbf{x})(\mathbf{t}). \tag{2.3}$$

The q-point *Stroud conical product rule* [73, Chapter 2] consists in approximating the right-hand side of (2.3) with some appropriate Gauss-Jacobi quadrature rules as the ones discussed in Section 2.2. The q-point Stroud conical product rule is exact for polynomials of degree at most $2q-1$. In particular, the choice $q = n+1$

in Section 3.3 yields a quadrature rule which exactly computes the integrals of polynomials of degree at most $2n+1$.

In our computations, the cell $[-1,1]^d$ is mapped to the unit cell $[0,1]^d$ by means of the change of variables given by

$$t_i(s_i) := \frac{1+s_i}{2}, \quad -1 \le s_i \le 1,\ i = 1,\ldots,d. \tag{2.4}$$

The Jacobian of this last transformation is equal to $1/2^d$, so that the right-hand side of (2.3) can be transformed into an integral over $[-1,1]^d$. We next proceed to analyze the computations in more details. For the sake of clarity, the cases $d = 1, 2, 3$ will be handled separately.

2.2 Optimal Element-Level Computations of the B-Moments

2.2.1 Binomial Coefficients

The use of the Bernstein polynomials defined in (1.6) involves multi-index binomials of the form $\binom{n}{\boldsymbol{\alpha}}$ which are products of ordinary binomial coefficients. The computation of the binomial coefficients may be unnecessarily costly, for example, if each binomial coefficient is "naively" computed by means of the representation

$$\binom{k}{p} = \frac{k(k-1)\ldots(k-\min\{p,k-p\}+1)}{(\min\{p,k-p\})!}, \quad k,p \in \mathbb{N},\ 0 \le p \le k. \tag{2.5}$$

For convenience, we recall that multi-index binomial coefficients are products of ordinary binomial coefficients which are collected in the auxiliary matrix of $\bar{\mathbf{C}}^n$ defined by

$$\bar{\mathbf{C}}_{i,j} := C_i^{i+j}, \quad 0 \le i,j \le n, \tag{2.6}$$

where $C_i^{i+j} := \binom{i+j}{i}$. The following algorithm with $m = n$ allows for the com-

putation of $\bar{\mathbf{C}}^n$ while avoiding the costly multiplications and divisions in (2.5):

Algorithm 2.1: Binomial$(\mathbf{C}, m, n)$

Input : -

Output: Binomial coefficients $\{C_p^{p+q} : \ 0 \le p \le m, \ 0 \le q \le n\}$.

```
C ≡ 0;
for p = 0 to m do
    C_{p,0} += 1;
for q = 1 to n do
    C_{0,q} += 1;
for p = 1 to m do
    for q = 1 to n do
        C_{p,q} += C_{p,q-1} + C_{p-1,q};
Return C;
```

Note that the above algorithm is nothing more than the well-known Pascal triangle method for the computation of binomial coefficients. With $m = n$, Algorithm 2.1 returns $\bar{\mathbf{C}}^n$ which has its antidiagonals given by the rows of the Pascal triangle (see Fig. 2.1). Although the entries of the array returned by `Binomial`$(\mathbf{C}, m, n)$ are all integers, for large values of m and n, it is recommended to store them as "double" numbers, in order to avoid overflow problems with integers.

$$\begin{pmatrix} 1 & \to & 1 & \to & 1 \\ \downarrow & & \downarrow\downarrow & & \downarrow\downarrow \\ 1 & \rightrightarrows & 2 & \rightrightarrows & 3 \\ \downarrow & & \downarrow\downarrow & & \downarrow\downarrow \\ 1 & \rightrightarrows & 3 & \rightrightarrows & 6 \end{pmatrix}$$

Figure 2.1: Computation of $\bar{\mathbf{C}}^2$

2.2.2 One-dimensional Setting

With $d = 1$, it follows from (2.3) and (2.4) that

$$\int_T g(x)\mathrm{d}x = \frac{|T|}{2}\int_{-1}^{1} g(x(s))\mathrm{d}s,$$

where, for $-1 \leq s \leq 1$, $x(s)$ is the point in the interval T with barycentric coordinates

$$(\frac{1+s}{2}, \frac{1-s}{2}),$$

having also used the fact that the Jacobian of the transformation (2.4) for $d = 1$ is $1/2$. Taking $g = f \cdot B^n_{\boldsymbol{\beta}}$ with $\boldsymbol{\beta} \in \mathcal{I}^n_1$ and using the definition (1.6), the q-point Gauss quadrature rule applied to the above integral gives

$$\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \frac{|T|}{2}\sum_{i=1}^{q}\omega_i f(\mathbf{x}_i)B^n_{\boldsymbol{\beta}}(\mathbf{x}_i) = \frac{|T|}{2}\binom{n}{\boldsymbol{\beta}}\sum_{i=1}^{q}\omega_i f(\mathbf{x}_i)\Big(\frac{1+\xi_i}{2}\Big)^{\beta_1}\Big(\frac{1-\xi_i}{2}\Big)^{n-\beta_1} \tag{2.7}$$

where $q = \mathcal{O}(n)$, and (ω_i, ξ_i) are the standard Gauss weights and centres for the interval $[-1, 1]$, and $\mathbf{x}_i$, $i = 1, \ldots, q$, are the corresponding centres in the interval T, with barycentric coordinates

$$\Big(\frac{1+\xi_i}{2}, \frac{1-\xi_i}{2}\Big). \tag{2.8}$$

The quadrature rule given by the right-hand side of (2.7) exactly computes $\mu^n_{\boldsymbol{\beta}}(f)$, provided that f is a polynomial of degree less or equal to ℓ, with $\ell = \max(0, 2q - 1 - n)$. As a direct consequence, with $q = n + 1$, the H^1 load vector, as discussed in Section 3.3.1, will be computed exactly as soon as f is a polynomial of degree at most n. Clearly, the combined weights of the quadrature formulae (2.7) can be pre-computed and stored in the auxiliary matrix

$$\mathbf{D} := \Big(\omega_i\Big(\frac{1+\xi_i}{2}\Big)^{\beta}\Big(\frac{1-\xi_i}{2}\Big)^{n-\beta}\Big)_{i\in\{1,\ldots,q\},\,\beta\in\{0,\ldots,n\}}. \tag{2.9}$$

Note that thanks to its product structure, $\mathbf{D}$ can be computed with $\mathcal{O}(n^2)$ operations. Indeed, for each $i = 0, \ldots, q$, the entries of the arrays

$$\sqrt{\omega_i}\Big(\frac{1-\xi_i}{2}\Big)^{n-\beta}, \quad \sqrt{\omega_i}\Big(\frac{1+\xi_i}{2}\Big)^{\beta}, \qquad \beta = 0, \ldots, n,$$

can be computed using $\mathcal{O}(n)$ operations, and then each of the $q(n+1)$ entries of $\mathbf{D}$ obtained by just one multiplication of the corresponding components of the above arrays.

Assuming that the values of f at the centres $\mathbf{x}_i$ are known, each of the $(n+1)$ quadrature centres in (2.7) requires $\mathcal{O}(n)$ operations if the matrix $\mathbf{D}$ and the binomial coefficients are pre-computed.

We summarize the findings of this section in the following algorithm for the computation of $\mu^n_{\boldsymbol{\beta}}(f)$ for $\boldsymbol{\beta} \in \mathcal{I}^n_1$ and a theorem about its cost.

Algorithm 2.2: Moment1D($\mathbf{F}, q, n$)

Input : Precomputed array $\mathbf{D}$ given by (2.9), and precomputed binomial coefficients $\{C^{p+q}_q, 0 \le p, q \le n\}$.

Output: Bernstein-Bézier moments of f obtained by means of the Stroud quadrature rule with q quadrature points.

```
F ≡ 0;
foreach β ∈ I_1^n do
    for i = 1 to q do
        F_β += D_{i,β_1} * f(x_i);
        /* x_i is the point with barycentric coordinates
        ((1+ξ_i)/2, (1−ξ_i)/2) */;
    F_β *= (|T|/2) * C^n_{β_2};
Return F;
```

Theorem 2.2.1. *Let $n, q \in \mathbb{N}$ with $q = \mathcal{O}(n)$. Given the values of a function f at the Stroud nodes $\mathbf{x}_i$, $i = 1, \ldots, q$, the* moment vector $\Big(\tilde{\mu}^n_{\boldsymbol{\beta}}(f)\Big)_{\boldsymbol{\beta}\in\mathcal{I}^n_1}$ *can be computed with $\mathcal{O}(n^2)$ operations. In addition, if f is a polynomial of degree at most ℓ, with $\ell = \max(0, 2q-1-n)$, then $\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \mu^n_{\boldsymbol{\beta}}(f)$ for any $\boldsymbol{\beta} \in \mathcal{I}^n_1$.*

2.2.3 Two-dimensional Setting

With $d = 2$, (2.3) and (2.4) give

$$\int_T g(\mathbf{x})\mathrm{d}\mathbf{x} = \frac{|T|}{4}\int_{-1}^{1}\mathrm{d}s_1(1-s_1)\int_{-1}^{1}\mathrm{d}s_2 g(\mathbf{x}(\mathbf{s}))\mathrm{d}\mathbf{s},$$

where $\mathbf{x}(\mathbf{s})$ is the point with barycentric coordinates

$$\Big(\frac{1+s_1}{2}, \frac{(1-s_1)(1+s_2)}{4}, \frac{(1-s_1)(1-s_2)}{4}\Big),$$

having also used the fact that the Jacobian of the transformation (2.4) for $d = 2$ is $1/4$.

Taking $g = f \cdot B^n_{\boldsymbol{\beta}}$ and using the definition (1.6), the above integral is approximated by means of the Stroud quadrature rule given by

$$\begin{aligned}\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \frac{|T|}{4}\binom{n}{\boldsymbol{\beta}}\sum_{i_1,i_2=1}^{q}\omega^{(1,0)}_{i_1}\omega^{(0,0)}_{i_2} f(\mathbf{x}_{i_1,i_2}) \\ \times\Big(\frac{1+\xi^{(1,0)}_{i_1}}{2}\Big)^{\beta_1}\Big(\frac{1-\xi^{(1,0)}_{i_1}}{2}\Big)^{n-\beta_1}\Big(\frac{1+\xi^{(0,0)}_{i_2}}{2}\Big)^{\beta_2}\Big(\frac{1-\xi^{(0,0)}_{i_2}}{2}\Big)^{n-\beta_1-\beta_2},\end{aligned} \tag{2.10}$$

where $(\omega^{(d,0)}_i, \xi^{(d,0)}_i)$, $i = 1, \ldots, q$, are the Gauss-Jacobi weights and centres with $(\alpha, \beta) = (d, 0)$, whereas q is chosen so as to satisfy $q = \mathcal{O}(n)$, and $\mathbf{x}_{i_1,i_2}$ are the points with barycentric coordinates

$$\Big(\frac{1+\xi^{(1,0)}_{i_1}}{2}, \frac{(1-\xi^{(1,0)}_{i_1})(1+\xi^{(0,0)}_{i_2})}{4}, \frac{(1-\xi^{(1,0)}_{i_1})(1-\xi^{(0,0)}_{i_2})}{4}\Big), \quad i_1, i_2 = 1, \ldots, q. \tag{2.11}$$

Indeed, the 2-dimensional conical product rule gives

$$\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \frac{|T|}{4}\sum_{i_1,i_2=1}^{q}\omega^{(1,0)}_{i_1}\omega^{(0,0)}_{i_2} f(\mathbf{x}_{i_1,i_2}) B^n_{\boldsymbol{\beta}}(\mathbf{x}_{i_1,i_2}),$$

which implies (2.10) because

$$B^n_{\boldsymbol{\beta}}(\mathbf{x}_{i_1,i_2}) = \binom{n}{\boldsymbol{\beta}} \Big[\frac{1+\xi^{(1,0)}_{i_1}}{2}\Big]^{\beta_1} \Big[\frac{1-\xi^{(1,0)}_{i_1}}{2}\frac{1+\xi^{(0,0)}_{i_2}}{2}\Big]^{\beta_2} \times \Big[\frac{1-\xi^{(1,0)}_{i_1}}{2}\frac{1-\xi^{(0,0)}_{i_2}}{2}\Big]^{n-\beta_1-\beta_2}.$$

The q-point quadrature rule given in (2.10) exactly computes $\mu^n_{\boldsymbol{\beta}}(f)$, provided that f is a polynomial of degree at most ℓ, with $\ell = \max(0, 2q-1-n)$.

Exploiting the product structure of (2.10) leads to the following optimal result. We use the concept of *sum factorization* explained in [41, Section A].

Theorem 2.2.2. *Let the assumptions of Theorem 2.2.1 hold. Given the values of the function f at the Stroud nodes $\mathbf{x}_{i_1,i_2}$, $i_1, i_2 = 1, \dots, q$, the moment vector $\big(\tilde{\mu}^n_{\boldsymbol{\beta}}(f)\big)_{\boldsymbol{\beta}\in\mathcal{I}^n_2}$ can be computed with $\mathcal{O}(n^3)$ operations. In addition, if f is a polynomial of degree at most ℓ with $\ell = \max(0, 2q-1-n)$, then $\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \mu^n_{\boldsymbol{\beta}}(f)$ for any $\boldsymbol{\beta} \in \mathcal{I}^n_2$.*

Proof. We proceed to estimate the computational cost corresponding to the auxilliary vector $\tilde{\boldsymbol{\mu}}(f) = \big(\tilde{\mu}^n_{\boldsymbol{\beta}}(f)\big)_{\boldsymbol{\beta}\in\mathcal{I}^n_2}$, with

$$\tilde{\mu}^n_{\boldsymbol{\beta}}(f) := \sum_{i_1,i_2=1}^{q} \omega^{(1,0)}_{i_1}\omega^{(0,0)}_{i_2} f(\mathbf{x}_{i_1,i_2}) \times \Big(\frac{1+\xi^{(1,0)}_{i_1}}{2}\Big)^{\beta_1}\Big(\frac{1-\xi^{(1,0)}_{i_1}}{2}\Big)^{n-\beta_1}\Big(\frac{1+\xi^{(0,0)}_{i_2}}{2}\Big)^{\beta_2}\Big(\frac{1-\xi^{(0,0)}_{i_2}}{2}\Big)^{n-\beta_1-\beta_2}. \quad (2.12)$$

Since obtaining the load vector $\boldsymbol{\mu}(f)$ from the auxiliary vector $\tilde{\boldsymbol{\mu}}(f)$ only involves $\mathcal{O}(n^2)$ operations, it suffices to prove that the computation of $\tilde{\boldsymbol{\mu}}(f)$ is done within a $\mathcal{O}(n^3)$ cost, so that the resulting complexity is $\mathcal{O}(n^2)+\mathcal{O}(n^3) = \mathcal{O}(n^3)$. To this end, observe from (2.10) and (2.12) that, for any $\boldsymbol{\beta} \in \mathcal{I}^n_2$,

$$\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \sum_{i_2=1}^{q} \Big[\sqrt{\omega^{(0,0)}_{i_2}}\Big(\frac{1+\xi^{(0,0)}_{i_2}}{2}\Big)^{\beta_2}\Big]\Big[\sqrt{\omega^{(0,0)}_{i_2}}\Big(\frac{1-\xi^{(0,0)}_{i_2}}{2}\Big)^{n-\beta_1-\beta_2}\Big] \times \sum_{i_1=1}^{q} \Big[\sqrt{\omega^{(1,0)}_{i_1}}\Big(\frac{1+\xi^{(1,0)}_{i_1}}{2}\Big)^{\beta_1}\Big]\Big[\sqrt{\omega^{(1,0)}_{i_1}}\Big(\frac{1-\xi^{(1,0)}_{i_1}}{2}\Big)^{n-\beta_1}\Big] f(\mathbf{x}_{i_1,i_2}),$$

that is,

$$\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \sum_{i_2=1}^{q} \Big[\sqrt{\omega^{(0,0)}_{i_2}}\Big(\frac{1+\xi^{(0,0)}_{i_2}}{2}\Big)^{\beta_2}\Big]\Big[\sqrt{\omega^{(0,0)}_{i_2}}\Big(\frac{1-\xi^{(0,0)}_{i_2}}{2}\Big)^{n-\beta_1-\beta_2}\Big] H(\beta_1, i_2), \tag{2.13}$$

where, for $\beta_1 \in \{0, \ldots, n\}$,

$$H(\beta_1, i_2) := \sum_{i_1=1}^{q} \Big[\sqrt{\omega^{(1,0)}_{i_1}}\Big(\frac{1+\xi^{(1,0)}_{i_1}}{2}\Big)^{\beta_1}\Big]\Big[\sqrt{\omega^{(1,0)}_{i_1}}\Big(\frac{1-\xi^{(1,0)}_{i_1}}{2}\Big)^{n-\beta_1}\Big] f(\mathbf{x}_{i_1,i_2}). \tag{2.14}$$

Now, assuming that the arrays

$$\Big[\sqrt{\omega^{(1,0)}_{i_1}}\Big(\frac{1+\xi^{(1,0)}_{i_1}}{2}\Big)^{\beta_1}\Big], \ \Big[\sqrt{\omega^{(1,0)}_{i_1}}\Big(\frac{1-\xi^{(1,0)}_{i_1}}{2}\Big)^{n-\beta_1}\Big], \quad i_1 = 1, \ldots, q,$$

are pre-computed for $\beta_1 \in \{0, \ldots, n\}$, the cost to set up the field H is then of order $\mathcal{O}(q^2(n+1))$. Again assuming that the arrays

$$\Big[\sqrt{\omega^{(0,0)}_{i_2}}\Big(\frac{1+\xi^{(0,0)}_{i_2}}{2}\Big)^{\beta_2}\Big], \quad \beta_2 \in \{0, \ldots, n\}, \quad i_2 = 1, \ldots, q,$$

and

$$\Big[\sqrt{\omega^{(0,0)}_{i_2}}\Big(\frac{1-\xi^{(0,0)}_{i_2}}{2}\Big)^{n-\beta_2}\Big], \quad \beta_2 \in \{0, \ldots, n\}, \quad i_2 = 1, \ldots, q,$$

are precomputed, the cost of summing over i_2 in (2.13), for all $\boldsymbol{\beta} \in \mathcal{I}^n_2$, is of order $\mathcal{O}(\frac{(n+1)(n+2)}{2}q)$. Hence, using the fact that $q = \mathcal{O}(n)$, the total cost of setting up $\tilde{\boldsymbol{\mu}}$ is

$$\mathcal{O}\big(q^2(n+1) + \frac{(n+1)(n+2)}{2}q\big) = \mathcal{O}(n^3). \tag{2.15}$$

□

Using the notations:

$$\begin{aligned}
\mathbf{D}^{(2)} &:= \left(\sqrt{\omega_{i_2}^{(0,0)}}\Big(\frac{1-\xi_{i_2}^{(0,0)}}{2}\Big)^{n-\beta}\right)_{\beta\in\{0,\dots,n\},\, i_2\in\{1,\dots,q\}},\\
\mathbf{P}^{(2)} &:= \left(\sqrt{\omega_{i_2}^{(0,0)}}\Big(\frac{1+\xi_{i_2}^{(0,0)}}{2}\Big)^{\beta}\right)_{\beta\in\{0,\dots,n\},\, i_2\in\{1,\dots,q\}},\\
\mathbf{D}^{(1)} &:= \left(\sqrt{\omega_{i_1}^{(1,0)}}\Big(\frac{1-\xi_{i_1}^{(1,0)}}{2}\Big)^{n-\beta}\right)_{\beta\in\{0,\dots,n\},\, i_1\in\{1,\dots,q\}},\\
\mathbf{P}^{(1)} &:= \left(\sqrt{\omega_{i_1}^{(1,0)}}\Big(\frac{1+\xi_{i_1}^{(1,0)}}{2}\Big)^{\beta}\right)_{\beta\in\{0,\dots,n\},\, i_1\in\{1,\dots,q\}},
\end{aligned} \tag{2.16}$$

we thus establish Algorithm 2.3 for the computation of $\mu_{\boldsymbol{\beta}}^n(f)$ for $\boldsymbol{\beta}\in\mathcal{I}_2^n$, taking into account the fact that $\binom{n}{\boldsymbol{\beta}} = C_{\beta_2}^{\beta_1+\beta_2} * C_{\beta_3}^n$.

Algorithm 2.3: Moment2D($\mathbf{F}, q, n$)

Input : Precomputed arrays $\mathbf{D}^{(k)}, \mathbf{P}^{(k)}$, $k=1,2$, defined by (2.16), and precomputed binomial coefficients $\{C_q^{p+q}, 0\le p,q\le n\}$.

Output: Bernstein-Bézier moments of f obtained by means of the Stroud conical product rule with q^2 quadrature points.

```
/* Precompute the field H */;
H ≡ 0;
for β1 = 0 to n do
    for i1 = 1 to q do
        for i2 = 1 to q do
            H(β1, i2) += D^(1)_{β1,i1} * P^(1)_{β1,i1} * f(x_{i1,i2});
F ≡ 0;
foreach β ∈ I_2^n do
    for i2 = 1 to q do
        F_β += D^(2)_{β1+β2,i2} * P^(2)_{β2,i2} * H(β1, i2);
    F_β *= (|T|/4) C_{β2}^{β1+β2} * C_{β3}^n;
Return F;
```

2.2.4 Three-dimensional Setting

With $d = 3$, it follows from (2.3) and (2.4) that

$$\int_T g(\mathbf{x})\mathrm{d}\mathbf{x} = \frac{3|T|}{32}\int_{-1}^{1}(1-s_1)^2\int_{-1}^{1}(1-s_2)\int_{-1}^{1} g(\mathbf{x}(\mathbf{s}))\mathrm{d}\mathbf{s},$$

where $\mathbf{xs}$ is the point with barycentric coordinates

$$\Big(\frac{1+s_1}{2}, \frac{(1-s_1)(1+s_2)}{4}, \frac{(1-s_1)(1-s_2)(1+s_3)}{8}, \frac{(1-s_1)(1-s_2)(1-s_3)}{8}\Big),$$

having also used the fact that, for $d = 3$, the Jacobian of the transformation (2.4) is $1/8$.

With $d = 3$, the quadrature rule is defined by

$$\begin{aligned}\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \frac{3|T|}{32}\binom{n}{\boldsymbol{\beta}}\sum_{i_1,i_2,i_3=1}^{q}\omega^{(2,0)}_{i_1}\omega^{(1,0)}_{i_2}\omega^{(0,0)}_{i_3} f(\mathbf{x}_{i_1,i_2,i_3})\Big(\frac{1+\xi^{(2,0)}_{i_1}}{2}\Big)^{\beta_1}\Big(\frac{1-\xi^{(2,0)}_{i_1}}{2}\Big)^{n-\beta_1}\\ \times\Big(\frac{1+\xi^{(1,0)}_{i_2}}{2}\Big)^{\beta_2}\Big(\frac{1-\xi^{(1,0)}_{i_2}}{2}\Big)^{n-\beta_1-\beta_2}\Big(\frac{1+\xi^{(0,0)}_{i_3}}{2}\Big)^{\beta_3}\Big(\frac{1-\xi^{(0,0)}_{i_3}}{2}\Big)^{n-\beta_1-\beta_2-\beta_3},\end{aligned} \tag{2.17}$$

where $q = \mathcal{O}(n)$, and $\mathbf{x}_{i_1,i_2,i_3}$ are the points with barycentric coordinates

$$\begin{aligned}\Big(&\frac{1+\xi^{(2,0)}_{i_1}}{2}, \frac{(1-\xi^{(2,0)}_{i_1})(1+\xi^{(1,0)}_{i_2})}{4}, \frac{(1-\xi^{(2,0)}_{i_1})(1-\xi^{(1,0)}_{i_2})(1+\xi^{(0,0)}_{i_3})}{8},\\ &\frac{(1-\xi^{(2,0)}_{i_1})(1-\xi^{(1,0)}_{i_2})(1-\xi^{(0,0)}_{i_3})}{8}\Big),\end{aligned} \tag{2.18}$$

for $i_1, i_2, i_3 = 1, \ldots, q$.

Indeed, the 3-dimensional conical product rule gives

$$\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \frac{3|T|}{32}\sum_{i_1,i_2,i_3=1}^{q}\omega^{(2,0)}_{i_1}\omega^{(1,0)}_{i_2}\omega^{(0,0)}_{i_3} f(\mathbf{x}_{i_1,i_2,i_3})B^n_{\boldsymbol{\beta}}(\mathbf{x}_{i_1,i_2,i_3}), \tag{2.19}$$

and (2.17) is obtained after collecting the factors in

$$B^n_{\boldsymbol{\beta}}(\mathbf{x}_{i_1,i_2,i_3}) = \binom{n}{\boldsymbol{\beta}} \Big(\frac{1+\xi^{(2,0)}_{i_1}}{2}\Big)^{\beta_1} \Big(\frac{1-\xi^{(2,0)}_{i_1}}{2}\frac{1+\xi^{(1,0)}_{i_2}}{2}\Big)^{\beta_2} \\ \times \Big(\frac{1-\xi^{(2,0)}_{i_1}}{2}\frac{1-\xi^{(1,0)}_{i_2}}{2}\frac{1+\xi^{(0,0)}_{i_3}}{2}\Big)^{\beta_3} \Big(\frac{1-\xi^{(2,0)}_{i_1}}{2}\frac{1-\xi^{(1,0)}_{i_2}}{2}\frac{1-\xi^{(0,0)}_{i_3}}{2}\Big)^{\beta_4}.$$

The q-point quadrature rule given in (2.17) exactly computes $\mu^n_{\boldsymbol{\beta}}(f)$, provided that f is a polynomial of degree at most ℓ, with $\ell = \max(0, 2q-1-n)$.

Exploiting the product structure of (2.17) leads to the following optimal result. We again use the concept of sum factorization.

Theorem 2.2.3. *Let the assumptions of Theorem 2.2.1 hold. Given the values of the function f at the Stroud nodes $\mathbf{x}_{i_1,i_2,i_3}$, $i_1,i_2,i_3 = 1,\dots,q$, the moment vector $\big(\tilde{\mu}^n_{\boldsymbol{\beta}}(f)\big)_{\boldsymbol{\beta}\in\mathcal{I}^n_3}$ can be computed with $\mathcal{O}(n^4)$ operations. In addition, if f is a polynomial of degree at most ℓ, with $\ell = \max(0,2q-1-n)$, then $\tilde{\mu}^n_{\boldsymbol{\beta}}(f) = \mu^n_{\boldsymbol{\beta}}(f)$ for any $\boldsymbol{\beta}\in\mathcal{I}^n_3$.*

Proof. Using a similar approach as in the case $d=2$, we proceed to evaluate the cost of computing the vector $\tilde{\boldsymbol{\mu}}(f) := \Big(\tilde{\mu}^n_{\boldsymbol{\beta}}(f)\Big)_{\boldsymbol{\beta}\in\mathcal{I}^n_3}$, with

$$\tilde{\mu}^n_{\boldsymbol{\beta}}(f) := \sum_{i_1,i_2,i_3=1}^{q} \omega^{(2,0)}_{i_1}\omega^{(1,0)}_{i_2}\omega^{(0,0)}_{i_3} f(\mathbf{x}_{i_1,i_2,i_3}) \Big(\frac{1+\xi^{(2,0)}_{i_1}}{2}\Big)^{\beta_1} \Big(\frac{1-\xi^{(2,0)}_{i_1}}{2}\Big)^{n-\beta_1} \\ \times \Big(\frac{1+\xi^{(1,0)}_{i_2}}{2}\Big)^{\beta_2} \Big(\frac{1-\xi^{(1,0)}_{i_2}}{2}\Big)^{n-\beta_1-\beta_2} \Big(\frac{1+\xi^{(0,0)}_{i_3}}{2}\Big)^{\beta_3} \Big(\frac{1-\xi^{(0,0)}_{i_3}}{2}\Big)^{n-\beta_1-\beta_2-\beta_3}.$$

Since obtaining the load vector from $\tilde{\boldsymbol{\mu}}(f)$ only involves $\mathcal{O}(n^3)$ operations, it suffices to prove that the computation of $\tilde{\boldsymbol{\mu}}(f)$ can be done with $\mathcal{O}(n^4)$ operations.

To this end, observe from the above equation that, for $\boldsymbol{\beta} \in \mathcal{I}_3^n$,

$$\begin{aligned}\tilde{\mu}_{\boldsymbol{\beta}}^n(f) = &\sum_{i_3=1}^{q} \Big[\sqrt{\omega_{i_3}^{(0,0)}}\Big(\frac{1+\xi_{i_3}^{(0,0)}}{2}\Big)^{\beta_3}\Big]\Big[\sqrt{\omega_{i_3}^{(0,0)}}\Big(\frac{1-\xi_{i_3}^{(0,0)}}{2}\Big)^{n-\beta_1-\beta_2-\beta_3}\Big]\\ &\times \sum_{i_2=1}^{q} \Big[\sqrt{\omega_{i_2}^{(1,0)}}\Big(\frac{1+\xi_{i_2}^{(1,0)}}{2}\Big)^{\beta_2}\Big]\Big[\sqrt{\omega_{i_2}^{(1,0)}}\Big(\frac{1-\xi_{i_2}^{(1,0)}}{2}\Big)^{n-\beta_1-\beta_2}\Big]\\ &\times \sum_{i_1=1}^{q} \Big[\sqrt{\omega_{i_1}^{(2,0)}}\Big(\frac{1+\xi_{i_1}^{(2,0)}}{2}\Big)^{\beta_1}\Big]\Big[\sqrt{\omega_{i_1}^{(2,0)}}\Big(\frac{1-\xi_{i_1}^{(2,0)}}{2}\Big)^{n-\beta_1}\Big] f(\mathbf{x}_{i_1,i_2,i_3}),\end{aligned}$$

that is,

$$\begin{aligned}\tilde{\mu}_{\boldsymbol{\beta}}^n(f) = &\sum_{i_3=1}^{q} \Big[\sqrt{\omega_{i_3}^{(0,0)}}\Big(\frac{1+\xi_{i_3}^{(0,0)}}{2}\Big)^{\beta_3}\Big]\Big[\sqrt{\omega_{i_3}^{(0,0)}}\Big(\frac{1-\xi_{i_3}^{(0,0)}}{2}\Big)^{n-\beta_1-\beta_2-\beta_3}\Big]\\ &\times U(\beta_1,\beta_2,i_3),\end{aligned} \tag{2.20}$$

where, for $\beta_1 = 0,\ldots,n,\ \beta_2 = 0,\ldots,n-\beta_1$,

$$\begin{aligned}U(\beta_1,\beta_2,i_3) := &\sum_{i_2=1}^{q} \Big[\sqrt{\omega_{i_2}^{(1,0)}}\Big(\frac{1+\xi_{i_2}^{(1,0)}}{2}\Big)^{\beta_2}\Big]\Big[\sqrt{\omega_{i_2}^{(1,0)}}\Big(\frac{1-\xi_{i_2}^{(1,0)}}{2}\Big)^{n-\beta_1-\beta_2}\Big]\\ &\times H(\beta_1,i_2,i_3),\end{aligned}$$

with, for $\beta_1 = 0,\ldots,n$,

$$H(\beta_1,i_2,i_3) := \sum_{i_1=1}^{q} \Big[\sqrt{\omega_{i_1}^{(2,0)}}\Big(\frac{1+\xi_{i_1}^{(2,0)}}{2}\Big)^{\beta_1}\Big]\Big[\sqrt{\omega_{i_1}^{(2,0)}}\Big(\frac{1-\xi_{i_1}^{(2,0)}}{2}\Big)^{n-\beta_1}\Big] f(\mathbf{x}_{i_1,i_2,i_3}).$$

Now, assuming that the arrays

$$\Big[\sqrt{\omega_{i_1}^{(2,0)}}\Big(\frac{1+\xi_{i_1}^{(2,0)}}{2}\Big)^{\beta}\Big],\ \Big[\sqrt{\omega_{i_1}^{(2,0)}}\Big(\frac{1-\xi_{i_1}^{(2,0)}}{2}\Big)^{n-\beta}\Big],\quad i_1 = 1,\ldots,q,\ \beta = 0,\ldots,n,$$

are precomputed, the cost of setting up the auxiliary field H is $\mathcal{O}(q^3(n+1))$. Similarly, assuming that the arrays

$$\Big[\sqrt{\omega_{i_2}^{(1,0)}}\Big(\frac{1+\xi_{i_2}^{(1,0)}}{2}\Big)^{\beta}\Big],\ \Big[\sqrt{\omega_{i_2}^{(1,0)}}\Big(\frac{1-\xi_{i_2}^{(1,0)}}{2}\Big)^{n-\beta}\Big],\quad \beta = 0,\ldots,n,\ i_2 = 0,\ldots,q,$$

the cost of setting up the auxiliary field U is $\mathcal{O}(q^2\frac{(n+1)(n+2)}{2})$. Finally, supposing that the arrays

$$\left[\sqrt{\omega_{i_3}^{(0,0)}}\Big(\frac{1+\xi_{i_3}^{(0,0)}}{2}\Big)^{\beta}\right],\ \left[\sqrt{\omega_{i_3}^{(0,0)}}\Big(\frac{1-\xi_{i_3}^{(0,0)}}{2}\Big)^{n-\beta}\right],\quad \beta=0,\ldots,n,\ i_3=1,\ldots,q,$$

are precomputed, the cost of summing (2.20) over i_3, for all $\boldsymbol{\beta}\in\mathcal{I}_3^n$, is

$$\mathcal{O}(q\frac{(n+1)(n+2)(n+3)}{6}).$$

Hence, the total cost of computing $\tilde{\boldsymbol{\mu}}(f)$ is

$$\mathcal{O}\big(q^3(n+1)+q^2\frac{(n+1)(n+2)}{2}+q\frac{(n+1)(n+2)(n+3)}{6}\big)=\mathcal{O}(n^4),$$

having also used the fact that $q=\mathcal{O}(n)$. □

Using the notations

$$\begin{aligned}\mathbf{D}^{(k)}&:=\left(\sqrt{\omega_{i_k}^{(3-k,0)}}\Big(\frac{1-\xi_{i_k}^{(3-k,0)}}{2}\Big)^{n-\beta}\right)_{\beta\in\{0,\ldots,n\},i_k\in\{1,\ldots,q\}},\\ \mathbf{P}^{(k)}&:=\left(\sqrt{\omega_{i_k}^{(3-k,0)}}\Big(\frac{1+\xi_{i_k}^{(3-k,0)}}{2}\Big)^{\beta}\right)_{\beta\in\{0,\ldots,n\},i_k\in\{1,\ldots,q\}},\end{aligned}\tag{2.21}$$

for $k=1,2,3$, we next introduce the algorithm `Moment3D` for the computation of $\mu_{\boldsymbol{\beta}}^n(f)$ for $\boldsymbol{\beta}\in\mathcal{I}_3^n$, having used the fact that $\binom{n}{\boldsymbol{\beta}}=C_{\beta_2}^{\beta_1+\beta_2}*C_{\beta_3}^{\beta_1+\beta_2+\beta_3}*C_{\beta_4}^{n}$.

Algorithm 2.4: Moment3D($\mathbf{F}, q, n$)

Input : Precomputed arrays $\mathbf{D}^{(k)}, \mathbf{P}^{(k)}$, $k = 1, 2, 3$, defined by (2.21), and precomputed binomial coefficients $\{C_q^{p+q}, 0 \leq p, q \leq n\}$.

Output: Bernstein-Bézier moments of f obtained by means of the Stroud conical product rule with q^3 quadrature points.

```
/* Precompute the field H */;
H ≡ 0;
for β1 = 0 to n do
    for i1 = 0 to q do
        ω = D^(1)_{β1,i1} * P^(1)_{β1,i1};
        for i2 = 0 to q do
            for i3 = 0 to q do
                H(β1, i2, i3) += ω * f(x_{i1,i2,i3});
/* Precompute the field U */;
U ≡ 0;
for β1 = 0 to n do
    for β2 = 0 to n − β1 do
        for i2 = 0 to q do
            ω = D^(2)_{β1+β2,i2} * P^(2)_{β2,i2};
            for i3 = 0 to q do
                U(β1, β2, i3) += ω * H(β1, i2, i3);
F ≡ 0;
foreach β ∈ I_3^n do
    for i3 = 0 to q do
        F_β += D^(3)_{β1+β2+β3,i3} * P^(3)_{β3,i3} * U(β1, β2, i3);
    F_β *= (3/32)|T| * C_{β2}^{β1+β2} * C_{β3}^{β1+β2+β3} * C_{β4}^{n};
Return F;
```

Combining Theorem 2.2.1, Theorem 2.2.2 and Theorem 2.2.3 gives the following result:

Theorem 2.2.4. *For $n \in \mathbb{N}$, let $q = \mathcal{O}(n)$ and f denote a smooth function. For $d = 1, 2, 3$, the load vector $\boldsymbol{\mu}(f) := \left(\mu_{\boldsymbol{\beta}}^n(f)\right)_{\boldsymbol{\beta} \in \mathcal{I}_d^n}$ can be computed with $\mathcal{O}(n^{d+1})$, when using the Stroud conical product rule based on q^d quadrature points.*

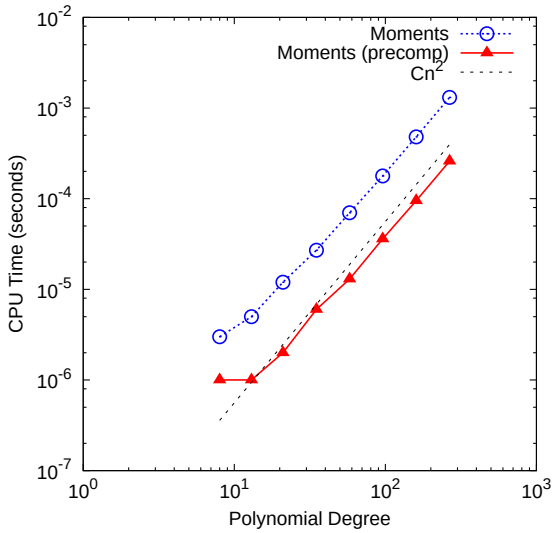
Remark 2.2.5. Observe from (2.8), (2.11) and (2.18) that the Stroud nodes have non-negative barycentric coordinates, which means that the quadrature centres involved in the Stroud conical product rule lie inside the simplex T. Since

the Bernstein polynomials $B^n_{\boldsymbol{\beta}}$, $\boldsymbol{\beta} \in \mathcal{I}^n_d$, are non-negative inside the simplex T, and since the Gauss and Gauss-Jacobi quadrature rules use positive weights, the weights involved in the quadrature rules (2.7), (2.10) and (2.17) are non-negative with respect to f, which is not necessarily the case with other shape functions. It is well-known that non-negative coefficients provide stability to the quadrature rule by preventing round-off errors [34]. Thus, positive quadrature rules are numerically stable.

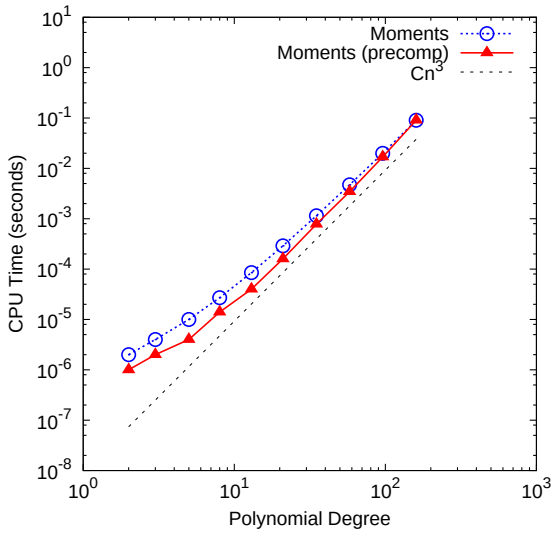
2.3 CPU Timings

Observe from (2.9), (2.16) or (2.21) that the method presented in this work for the evaluation of the B-moments relies on precomputing quadrature arrays at the Stroud nodes. The present work was used as a foundation for building alternative algorithms [11] which require no storage of precomputed arrays but still achieve the optimal complexity.

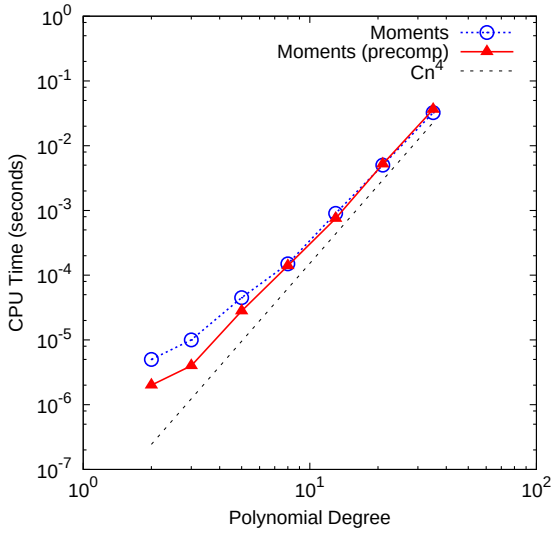
For $d = 1, 2, 3$, we now proceed to plot the CPU time involved in the computation of the B-moments against the value of n, using both the algorithms presented here and those introduced in [11]. The computations are performed on a Dell Precision T7400 workstation with Xeon 3.2GHz processor and 32Gb RAM. The obtained results are plotted on Figure 2.2. The red line corresponds to the algorithms proposed in this thesis, whereas the blue line is associated with [11, Algorithm 3]. For each value of d, the CPU time is plotted next to the curve of the function Cn^{d+1}, where C is a constant. We observe that, in all cases, the CPU timings are consistent with the predicted optimal complexity $\mathcal{O}(n^{d+1})$ for the computation of the B-moments of order n.



(a) $d = 1$



(b) $d = 2$



(c) $d = 3$

Figure 2.2: CPU timings for the computation of the B-moment vector

2.4 Conclusion

The key idea behind the optimal evaluations of the B-moments consists in combining sum factorization techniques with the conical Stroud product rule. One should note that the factorizations which are used rely on intrinsic properties of the Bernstein polynomials, in that a tensor product structure arises from the Duffy transformation, when applied to the B-moments (2.1). For the case of variable data, the optimal complexity result given in Theorem 2.2.4 will be crucial for the efficient evaluations of the elemental quantities developed in the remaining part of this work.

Chapter 3

Bernstein-Bézier Finite Elements for H^1

The purpose of this chapter is to present fast and easy to implement algorithms for assembling the load vector, mass and stiffness matrices arising from the H^1 finite element method based on the Bernstein-Bézier shape functions of any polynomial degree n in dimensions $d = 1, 2, 3$. In particular, we show that, taking the numerical quadrature into consideration, these algorithms achieve the optimal computational complexity $\mathcal{O}(n^{2d})$. The work discussed in this chapter was used as as foundation for the algorithms presented in [11]. Both approaches have been implemented in a `C++` library which is documented in Appendix B.

The general linear second order elliptic PDE is given by

$$\begin{cases} -\text{div}\,(\mathbf{A}\nabla u) + \mathbf{b}\cdot\nabla u + cu &= f \text{ in } \Omega, \\ u &= 0 \text{ on } \Gamma_D, \\ \sigma u + \mathbf{n}\cdot(\mathbf{A}\nabla u) &= g \text{ on } \Gamma_N, \\ \overline{\Gamma_D \cup \Gamma_N} &= \partial\Omega, \\ \Gamma_D \cap \Gamma_N &= \emptyset, \end{cases} \tag{3.1}$$

where $\mathbf{A}$ is a matrix-valued function which is continuous and positive definite on Ω. Multiplying the first equation by a sufficiently smooth test function v,

integrating over Ω and using the Green's formula

$$\int_\Omega -\text{div }(\mathbf{A}\nabla u)v\mathrm{d}\mathbf{x} = \int_\Omega \nabla v\cdot\mathbf{A}\cdot\nabla u\mathrm{d}\mathbf{x} - \int_{\partial\Omega} v\big(\mathbf{n}\cdot(\mathbf{A}\nabla u)\big)\mathrm{d}s$$

coupled with the boundary conditions yield

$$\int_\Omega \Big[\nabla v(\mathbf{x})\mathbf{A}(\mathbf{x})\nabla u(\mathbf{x}) + [\mathbf{b}(\mathbf{x})\cdot\nabla u(\mathbf{x})]v(\mathbf{x}) + c(\mathbf{x})u(\mathbf{x})v(\mathbf{x})\Big]\mathrm{d}\mathbf{x} + \int_{\Gamma_N}\sigma(\mathbf{x})u(\mathbf{x})v(\mathbf{x})\mathrm{d}s = \int_\Omega f(\mathbf{x})v(\mathbf{x})\mathrm{d}\mathbf{x} + \int_{\Gamma_N} g(\mathbf{x})v(\mathbf{x})\mathrm{d}s, \quad (3.2)$$

provided that all the integrals are meaningful. Defining the space $H^1_D(\Omega)$ as $H^1_D(\Omega) := \{v\in H^1(\Omega):\ v=0 \text{ on } \Gamma_D\}$, the weak formulation corresponding to (3.1) is to find $u\in H^1_D(\Omega)$ such that for all $v\in H^1_0(\Omega)$, equation (3.2) holds. The Galerkin discretization of the problem consists in replacing the continuous space $H^1_D(\Omega)$ with a finite-dimensional subspace. For simplicity, we assume that Ω is a polyhedral domain in $\mathbb{R}^d$ with $d\in\{1,2,3\}$. Assuming that Ω is triangulated into simplices, and that $\{\psi_{T,i}\}_{i=1}^N$ is a conforming set of shape functions on each simplex T, the solution $u\in H^1_D(\Omega)$, and any $v\in H^1_D(\Omega)$ are approximated by means of

$$v\approx v_{\text{FE}} := \sum_{j=1}^N \ell_j\psi_{T,j},\quad u\approx u_{\text{FE}} := \sum_{i=1}^N k_i\psi_{T,i},$$

on each triangle T of the triangulation. Inserting the above approximations into the weak form (3.2) leads to a linear system involving the load vector, mass, stiffness and convective matrices whose components are given by

$$\int_T f\psi_{T,i}\,\mathrm{d}\mathbf{x},\ \int_T c\psi_{T,i}\psi_{T,j}\,\mathrm{d}\mathbf{x},\ \int_T \nabla\psi_{T,j}\cdot\mathbf{A}\cdot\nabla\psi_{T,i}\,\mathrm{d}\mathbf{x},\ \int_T \psi_{T,j}\mathbf{b}\cdot\nabla\psi_{T,i}\mathrm{d}\mathbf{x}, \quad (3.3)$$

respectively. Observe in particular that the second integral in (3.2) involves mass matrices and load vectors of the the same form as in (3.3) on simplices of lower dimension.

The chapter is organized as follows. Section 3.1 is devoted to a description

of the Bernstein-Bézier shape functions and their well-known properties such as de Casteljau evaluation algorithm and H^1-conformity. Section 3.2 deals with the finite element assembly in the case of piecewise constant data in (3.2) and presents explicit formulae, without resort to any quadrature rules. This is because the product of two Bernstein polynomials is again a Bernstein polynomial, the components of the gradient are given by simple linear combinations of $d+1$ Bernstein polynomials, and there is a closed form expression for the integral of a Bernstein polynomial over a simplex. Section 3.2 forms the foundation for Section 3.3, where the case of spatially variable data is considered. The main result is presented in Theorem 3.5.1 which asserts that the optimal complexity is achieved. Section 3.6 concludes with some preliminary illustrative examples of the use of BB-FEM to solve some simple partial differential equations. We do not consider the important issue of the cost of solving the resulting linear system, but note that standard existing preconditioning techniques [30, 9] can be employed to assist with this task.

3.1 Bernstein-Bézier H^1 Finite Elements on a Partition

The fact that the representation (1.8) of a polynomial $p \in \mathbb{P}_d^n(T)$ is unique means that, for every $\boldsymbol{\eta} \in \mathcal{I}_d^n$, we may define a linear functional

$$\mathbb{P}_d^n \ni p \mapsto \phi_{\boldsymbol{\eta}}^T(p) = c_{\boldsymbol{\eta}},$$

where $c_{\boldsymbol{\eta}}$ is the coefficient appearing in (1.8). Hence by setting

$$\Sigma_d^n(T) := \{\phi_{\boldsymbol{\eta}}^T : \ \boldsymbol{\eta} \in \mathcal{I}_d^n\},$$

we obtain a finite element $(T, \Sigma_d^n(T), \mathbb{P}_d^n(T))$, which we refer to as the *Bernstein-Bézier finite element* (BB-FEM).

Let $\Delta = \{T_j\}_{i=1}^N$ be a regular triangulation [64, Section 4.3] of a bounded domain $\Omega \subset \mathbb{R}^d$ into d-simplices T_j in the usual sense, see e.g. [33]. Finite elements $(T_j, \mathbb{P}_d^n(T_j), \Sigma_d^n(T_j))$ on the simplices T_j, $j = 1, \ldots, N$, give rise to a finite dimensional *finite element space* $S_d^n(\Delta)$ of piecewise polynomial functions on Ω with a corresponding basis $\Sigma_d^n(\Delta)$ of the dual of $S_d^n(\Delta)$. The *global degrees of freedom* of $s \in S_d^n(\Delta)$ are the values $\phi(s)$, $\phi \in \Sigma_d^n(\Delta)$. Similar to the standard H^1-conforming finite elements, certain functionals in $\Sigma_d^n(T_j)$ and $\Sigma_d^n(T_k)$ are identified if T_j and T_k have a common interface. For this end it is convenient to use the concept of *domain points* [64, Chapter 1]. For a simplex $T = \langle \mathbf{v}_i,\ i = 1, \ldots, d+1 \rangle$, recall that the set of domain points $\mathcal{D}_d^n(T)$ is defined by (1.4), from which one can clearly see the 1-1 correspondence between the domain points $\boldsymbol{\xi}_{\boldsymbol{\eta}} \in \mathcal{D}_d^n(T)$ and the multi-indices $\boldsymbol{\eta} \in \mathcal{I}_d^n$. Thus, it follows with a harmless abuse of notation that the local degrees of freedom may be written as

$$\Sigma_d^n(T) = \{\phi_{\boldsymbol{\xi}}^T :\ \boldsymbol{\xi} \in \mathcal{D}_d^n(T)\}. \tag{3.4}$$

Similarly, the set of domain points $\mathcal{D}_d^n(\Delta)$ on the triangulation Δ is given by

$$\mathcal{D}_d^n(\Delta) := \bigcup_{j=1}^N \mathcal{D}_d^n(T_j).$$

In particular, we observe that the domain points on the interface between adjacent elements coincide.

Remark 3.1.1. Recall that the faces of T in one dimension are the endpoints of the interval T, in two dimensions vertices and edges of the triangle, and in three dimensions vertices, edges and faces of the tetrahedron.

The finite element space $S_d^n(\Delta)$ generated by the Bernstein-Bézier finite elements consists of the functions defined on Ω by the rule

$$s_{|T} := \sum_{\boldsymbol{\xi} \in \mathcal{D}_d^n(T)} c_{\boldsymbol{\xi}} B_{\boldsymbol{\xi}}^{n,T}, \quad T \in \Delta, \tag{3.5}$$

where the real numbers $c_{\boldsymbol{\xi}}$, $\boldsymbol{\xi} \in \mathcal{D}_d^n(\Delta)$, are the global degrees of freedom of

s. Respectively, the basis $\Sigma_d^n(\Delta)$ of the dual of $S_d^n(\Delta)$ is given by the linear functionals $\phi_{\boldsymbol{\xi}}$, $\boldsymbol{\xi} \in \mathcal{D}_d^n(\Delta)$, defined by

$$\phi_{\boldsymbol{\xi}}(s) = \phi_{\boldsymbol{\xi}}^T(s_{|T}), \quad \text{for any } T \in \Delta \text{ such that } \boldsymbol{\xi} \in \mathcal{D}_d^n(T).$$

Theorem 3.1.2. *The global finite element space $S_d^n(\Delta)$, resulting from the assembly of the local Bernstein-Bézier finite elements, coincides with the standard H^1-conforming finite element space of continuous piecewise polynomials of degree n on Δ. That is,*

$$S_d^n(\Delta) = \left\{ s \in C^0(\Omega) : \ s_{|T} \in \mathbb{P}_d^n, \ T \in \Delta \right\}. \tag{3.6}$$

Proof. If $s \in S_d^n(\Delta)$, then (3.5) implies that $p_T := s_{|T} \in \mathbb{P}_d^n$ for all $T \in \Delta$. To show the continuity of s, consider any pair of elements $T_j, T_k \in \Delta$ with a common interface e. Then, with $T = T_j$ or T_k, $B_{\boldsymbol{\xi}}^{n,T}$ vanishes on e if $\boldsymbol{\xi} \notin e$, and hence

$$(p_T)_{|e} = \sum_{\boldsymbol{\xi} \in \mathcal{D}_d^n(T) \cap e} c_{\boldsymbol{\xi}} B_{\boldsymbol{\xi}}^{n,T}.$$

Since $B_{\boldsymbol{\xi}}^{n,T_j}$ and $B_{\boldsymbol{\xi}}^{n,T_k}$ coincide on e for any $\boldsymbol{\xi} \in \mathcal{D}_d^n(T) \cap e$, we see that $(p_{T_j})_{|e} = (p_{T_k})_{|e}$, and hence $s_{|T_j \cup T_k}$ is continuous.

On the other hand, if s is a continuous piecewise polynomial with $s_{|T} \in \mathbb{P}_d^n$, $T \in \Delta$, then for each T there is a unique B-form representation of $s_{|T}$ in terms of the sum $s_{|T} := \sum_{\boldsymbol{\xi} \in \mathcal{D}_d^n(T)} c_{\boldsymbol{\xi}}^T B_{\boldsymbol{\xi}}^{n,T}$. It is well known [37, 64] that the continuity of s implies $c_{\boldsymbol{\xi}}^{T_j} = c_{\boldsymbol{\xi}}^{T_k}$ for any domain point $\boldsymbol{\xi} \in \mathcal{D}_d^n(T_j) \cap \mathcal{D}_d^n(T_k)$. We conclude that s belongs to $S_d^n(\Delta)$ as it satisfies (3.5) with a sequence of global degrees of freedom $c_{\boldsymbol{\xi}}$ uniquely defined by

$$c_{\boldsymbol{\xi}} = c_{\boldsymbol{\xi}}^T \quad \text{for any } T \text{ such that } \boldsymbol{\xi} \in \mathcal{D}_d^n(T).$$

□

Note that in the case of first order polynomials $n = 1$, there is no difference

with the standard piecewise linear finite element. The basis functionals in $\Sigma_d^1(\Delta)$ are the usual nodal evaluations at the vertices, and a dual basis for $S_d^1(\Delta)$ consists of the standard hat functions.

3.1.1 Evaluation and Visualisation of Bernstein-Bézier Finite Elements

The previous remarks show that we may use Bernstein-Bézier polynomials to construct a basis for the standard H^1-conforming finite element spaces of degree n on a triangulation Δ. Bernstein-Bézier representation of polynomials is extensively used in the CAGD and computer graphics. In computer graphics, the partition of unity and non-negativity properties of the Bernstein polynomials make them important tools in terms of Bézier curves, see [44, Chapter 4], [46, Chapter 11]. In CAGD, Bernstein polynomials appear in the form of control surfaces which possess important shape-preserving properties. Using degree-raising or subdivision, control surfaces can be used to render surfaces, see [64, Chapter 3]. One advantage of using Bernstein-Bézier polynomials is the availability of efficient and stable procedures for their evaluation, and for the evaluation of their derivatives.

Evaluation and visualisation both play a vital role in the post-processing and interpretation of the results of a finite element analysis. The availability of a Bernstein-Bézier representation of the finite element approximation enables one to exploit the attractive features of the B-form. In the interests of completeness, we present a short overview of techniques for the efficient and stable evaluation of a BB-FEM. We refer the reader to [64, Chapter 3], [44, Chapter 17], [36, Chapter 12] for information on aspects related to visualisation.

For $i = 1, \ldots, d+1$, the symbol $\mathbf{e}_i$ refers to the $(d+1)$-tuple belonging to $\mathcal{I}_d^1$ such that its i^{th} is one and the other components are zero. The following algorithm enables us to efficiently evaluate a polynomial written in B-form at any given point $\mathbf{v} \in \mathbb{R}^d$:

Algorithm 3.1: de Casteljau Algorithm

Input : B-form coefficients $\{c_{\boldsymbol{\xi}} : \boldsymbol{\xi} \in \mathcal{I}_d^n\}$ of polynomial $p \in \mathbb{P}_d^n$, and barycentric coordinates λ_i $(i = 1, \ldots, d+1)$ of $\mathbf{v} \in \mathbb{R}^d$.

Output: Evaluation of p at the point $\mathbf{v}$.

1 Set $c_{\boldsymbol{\eta}}^{(0)} = c_{\boldsymbol{\eta}}$ for $\boldsymbol{\eta} \in \mathcal{I}_d^n$;
2 **for** $\ell = 1$ **to** n **do**
3 **foreach** $\boldsymbol{\eta} \in \mathcal{I}_d^{n-\ell}$ **do**
4 $c_{\boldsymbol{\eta}}^{(\ell)} := \sum_{i=1}^{d+1} \lambda_i c_{\boldsymbol{\eta}+\mathbf{e}_i}^{(\ell-1)}$;
5 **Return** $c_{000}^{(n)}$;

Observe that the innermost loop contains $d+1$ multiplications which are executed $\binom{n-\ell+d}{d}$ times, for ℓ ranging from 1 to n. Hence, the operation count (multiplications and divisions only) corresponding to this algorithm is given by

$$(d+1)\sum_{\ell=1}^{n}\binom{n-\ell+d}{d}.$$

In particular, the number of operations is:

$$\begin{cases} n^2+n, & \text{if } d=1, \\ (n^3+3n^2+2n)/2, & \text{if } d=2, \\ (n^4+6n^3+11n^2+6n)/6, & \text{if } d=3. \end{cases} \tag{3.7}$$

The case $d = 2$ is proved in [64, Chapter 2].

Using sum factorization techniques, it is proved in [11, Algorithm 1] that a polynomial of order n written in its BB-form can be evaluated at the Stroud nodes of order $q = \mathcal{O}(n)$ with $\mathcal{O}(n^{d+1})$ operations.

The barycentric coordinates of a point $\mathbf{v} \in T$ are non-negative inside the triangle T. Consequently, thanks to equation (1.5) and the de Casteljau algorithm, we observe that, at each step of the algorithm, the new coefficients are computed as a convex linear combination of the previous ones [64], which leads to the numerical stability of the de Casteljau algorithm [44]. Although the de Casteljau algorithm is an established tool for the evaluation of polynomials ex-

pressed in their Bernstein representation, it is shown in [11, Section 3.2] that Bernstein polynomials can be evaluated at Stroud points using a much faster approach which costs $\mathcal{O}(n^{d+1})$ operations.

The B-form of the directional derivatives of a polynomial in B-form can also be efficiently computed, using the formulae similar to those arising in the de Casteljau algorithm [64]. The gradient will be of particular importance for our purposes and we summarize here some key properties that will prove useful later.

Lemma 3.1.3. *The gradient of the Bernstein polynomial $B^n_{\boldsymbol{\xi}}$ satisfies*

$$\nabla B^n_{\boldsymbol{\xi}} = n \sum_{i=1}^{d+1} B^{n-1}_{\boldsymbol{\xi}-\mathbf{e}_i} \nabla \lambda_i, \quad \boldsymbol{\xi} \in \mathcal{I}^n_d. \tag{3.8}$$

Hence, if the B-form coefficients of $p \in \mathbb{P}^n_d$ with respect to the d-simplex T are given by $\{c_{\boldsymbol{\xi}} : \boldsymbol{\xi} \in \mathcal{I}^n_d\}$, then

$$\nabla p = n \sum_{\boldsymbol{\xi} \in \mathcal{I}^{n-1}_d} B^{n-1}_{\boldsymbol{\xi}} \sum_{i=1}^{d+1} c_{\boldsymbol{\xi}+\mathbf{e}_i} \nabla \lambda_i. \tag{3.9}$$

Proof. Let us fix $\boldsymbol{\xi} \in \mathcal{I}^n_d$. Using the chain rule, observe that

$$\nabla B^n_{\boldsymbol{\xi}} = \binom{n}{\boldsymbol{\xi}} \nabla \Big(\prod_{i=1}^{d+1} \lambda_i^{\xi_i} \Big) = n \sum_{i=1}^{d+1} B^{n-1}_{\boldsymbol{\xi}-\mathbf{e}_i} \nabla \lambda_i.$$

Thus, if p satisfies the equality $p = \sum_{\boldsymbol{\xi} \in \mathcal{I}^n_d} c_{\boldsymbol{\xi}} B^n_{\boldsymbol{\xi}}$, then

$$\nabla p = \sum_{\boldsymbol{\xi} \in \mathcal{I}^n_d} c_{\boldsymbol{\xi}} \nabla B^n_{\boldsymbol{\xi}} = n \sum_{\boldsymbol{\xi} \in \mathcal{I}^n_d} c_{\boldsymbol{\xi}} \sum_{i=1}^{d+1} B^{n-1}_{\boldsymbol{\xi}-\mathbf{e}_i} \nabla \lambda_i,$$

and (3.9) follows by a simple change of the summation index $\boldsymbol{\xi}$. □

Remark 3.1.4. Let T be a given d-simplex and $\mathbf{x}_i$ any vertex. Let γ_i denote the $(d-1)$-simplex formed using the vertices of T with $\mathbf{x}_i$ excluded. Then, for

$i = 1, \ldots, d+1$, it holds that

$$\nabla \lambda_i = -\frac{|\gamma_i|}{d|T|}\hat{\mathbf{n}}_i,$$

where $\hat{\mathbf{n}}_i$ is the unit outwards normal on γ_i. Note that in the case $d = 1$, we take $|\gamma_i| = 1$.

3.2 Optimal FE-Assembly I: Piecewise Constant Data

In this section, we present explicit formulae for computing the load vector, the mass and the stiffness matrices in the case when $f, c, \mathbf{b}, A$ in (3.3) are constant on each simplex. Using the formula for the inner product of Bernstein polynomials as well as suitably defined shift operators which allow us to compute the stiffness matrix from the mass matrix of lower degree, we develop algorithms for the computation of the above matrices and vectors, and show that they achieve the optimal complexity $\mathcal{O}(n^{2d})$. For simplicity, we assume that the diffusion coefficient A is scalar-valued in this section, and postpone the treatment of the non-constant tensor-valued case to Section 3.3.4.

3.2.1 Load Vector

Given $f \in S^0_{0,d}(\Delta)$, that is, f being a piecewise constant polynomial on the triangulation Δ, we now want to evaluate the load vector defined by

$$\mathbf{f} = \mathbf{f}^T_{\boldsymbol{\xi}} := \mu^n_{\boldsymbol{\xi}}(f), \quad \boldsymbol{\xi} \in \mathcal{I}^n_d.$$

In particular, the load vector associated with H^1-conforming Bernstein finite elements coincides with the B-moment vector discussed in the previous chapter. Observe that, if f is constant on T, then $\mathbf{f}_{\boldsymbol{\xi}} = f_{|T}\mu^n_{\boldsymbol{\xi}}(1)$. But then, recall that [37,

Section 7]

$$\mu^n_{\boldsymbol{\xi}}(1) := \int_T B^n_{\boldsymbol{\xi}}(\mathbf{x})\mathrm{d}\mathbf{x} = \frac{|T|}{\binom{n+d}{d}}, \quad \boldsymbol{\xi} \in \mathcal{I}^n_d. \tag{3.10}$$

Therefore, we find that

$$\mathbf{f}_{\boldsymbol{\xi}} = \mu^n_{\boldsymbol{\xi}}(f) = \frac{f|_T|T|}{\binom{n+d}{d}}, \quad \boldsymbol{\xi} \in \mathcal{I}^n_d. \tag{3.11}$$

Note that the element load vector $\mathbf{f}$ corresponds to a single simplex T. Each entry of the load vector corresponds to a node belonging to T. Each node is assigned a local numbering, depending on the simplex T that is considered, and a global numbering which is independent of the simplex T. The global load vector is thus initialized to the null vector with number of entries given by the number of global nodes. The k^{th} entry of the global load vector is then obtained by summing over all the triangles the entries which have been assigned the global numbering k.

The global mass and stiffness matrices are built from the element mass and stiffness matrices in a similar way.

3.2.2 Mass Matrix

For $n \in \mathbb{Z}_+$, we want to compute the element mass matrix $\mathbf{M}^n$ given by

$$\mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \mathbf{M}^n_{\boldsymbol{\alpha},\boldsymbol{\beta}} := (cB^n_{\boldsymbol{\alpha}}, B^n_{\boldsymbol{\beta}}), \quad \boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}^n_d.$$

Exploiting the fact that the product of two Bernstein polynomials is a Bernstein polynomial, that is, $B^n_{\boldsymbol{\alpha}} B^n_{\boldsymbol{\beta}} = \binom{\boldsymbol{\alpha}+\boldsymbol{\beta}}{\boldsymbol{\alpha}} / \binom{2n}{n} B^{2n}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}$, for $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}^n_d$, and again using formula (3.10) immediately gives an explicit formula for the mass matrix corresponding to the basis (1.7), see also [64].

Theorem 3.2.1. *For $n \in \mathbb{Z}_+$,*

$$\mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \frac{c_{|T}|T|}{\binom{2n}{n}\binom{2n+d}{d}} \binom{\boldsymbol{\alpha}+\boldsymbol{\beta}}{\boldsymbol{\alpha}}, \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}_d^n. \tag{3.12}$$

The next algorithms provide a method to compute the mass matrix with the cost $\mathcal{O}(n^{2d})$. For comparison, the number of entries in $\mathbf{M}^T$ is $\binom{n+d}{d}^2 = \mathcal{O}(n^{2d})$. Hence, even if each entry were to be computed with $\mathcal{O}(1)$ operations, evaluating the mass matrix would require at least $\mathcal{O}(n^{2d})$ operations.

Algorithm 3.2: 1DMassMatConst($\mathbf{M}, n$)

Input : Precomputed binomial coefficients $\{C_p^{p+q} : \ 0 \le p \le n, 0 \le q \le n\}$.
Output: 1D element mass matrix of order n.

```
M = 0;
for α1 = n to 0 do
    for β1 = n to 0 do
        w1 = C_{α1}^{α1+β1} * c_{|T}|T| / (C_n^{2n} * (2n+1));
        w2 = w1 * C_{n-α1}^{n-α1+n-β1};
        M_{α,β} += w2;
Return M;
```

Algorithm 3.3: 2DMassMatConst($\mathbf{M}, n$)

Input : Precomputed binomial coefficients $\{C_p^{p+q} : \ 0 \le p \le n+2, 0 \le q \le n\}$.
Output: 2D element mass matrix $\mathbf{M}$ of order n.

```
M = 0;
for α1 = n to 0 do
    for β1 = n to 0 do
        w1 = C_{α1}^{α1+β1} * c_{|T}|T| / (C_n^{2n} * C_2^{2n+2});
        for α2 = n - α1 to 0 do
            for β2 = n - β1 to 0 do
                w2 = w1 * C_{α2}^{α2+β2};
                w3 = w3 * C_{n-α1-α2}^{n-α1-α2+n-β1-β2};
                M_{α,β} += w3;
Return M;
```

Algorithm 3.4: 3DMassMatConst($\mathbf{M}, n$)

Input : Precomputed binomial coefficients $\{C_p^{p+q} : \ 0 \le p \le n+3, 0 \le q \le n\}$.

Output: 3D element mass matrix of order n.

$\mathbf{D} = \mathbf{0}$;
for $\alpha_1 = n$ **to** 0 **do**
  for $\beta_1 = n$ **to** 0 **do**
    $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1} * \dfrac{c_{|T}|T|}{C_n^{2n} * C_3^{2n+3}}$;
    for $\alpha_2 = n - \alpha_1$ **to** 0 **do**
      for $\beta_2 = n - \beta_1$ **to** 0 **do**
        $w_2 = w_1 * C_{\alpha_2}^{\alpha_2+\beta_2}$;
        for $\alpha_3 = n - \alpha_1 - \alpha_2$ **to** 0 **do**
          for $\beta_3 = n - \beta_1 - \beta_2$ **to** 0 **do**
            $w_3 = w_2 * C_{\alpha_2}^{\alpha_2+\beta_2}$;
            $w_4 = w_3 * C_{n-\alpha_1-\alpha_2-\alpha_3}^{n-\alpha_1-\alpha_2-\alpha_3+n-\beta_1-\beta_2-\beta_3}$;
            $\mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}} \mathrel{+}= w_4$;
Return M;

The mass matrix can be computed by means of Algorithms 3.2, 3.3 and 3.4 for $d = 1, 2$ and 3, respectively.

We now proceed with the complexity analysis of the mass matrix algorithms. For simplicity, we focus on the case $d = 3$. The other two cases are similar. For $i = 1, 2$, every loop over the pair (α_i, β_i) contains one multiplication, and is executed $\binom{n+i}{i}^2$ times. The computational cost of the algorithm is dominated by its innermost loop which contains two multiplications, and is executed $\binom{n+3}{3}^2$ times. Hence, the total number of operations required for computing the 3D mass matrix is $(n+1)^2 + \binom{n+2}{2}^2 + 2\binom{n+3}{3}^2$. Using a similar argument, we find that the number of operations needed for computing the 1D and 2D mass matrices is respectively given by $2(n+1)^2$ and $(n+1)^2 + 2\binom{n+1}{2}^2$. In summary, the number of operations involved in the computation of the mass matrix associated with

piecewise constant data is:

$$\begin{cases} 2n^2+4n+2 & \text{for } d=1, \\ \frac{n^4}{2}+3n^3+\frac{15n^2}{2}+8n+3 & \text{for } d=2, \\ \frac{n^6}{18}+\frac{2n^5}{3}+\frac{125n^4}{36}+\frac{19n^3}{2}+\frac{539n^2}{36}+\frac{37n}{3}+4 & \text{for } d=3. \end{cases} \tag{3.13}$$

3.2.3 Stiffness Matrix

With $n \geq 1$, the stiffness matrix $\mathbf{S}$ is defined by

$$\mathbf{S}_{\boldsymbol{\alpha},\boldsymbol{\beta}} := (\mathbf{A}\nabla B^n_{\boldsymbol{\alpha}}, \nabla B^n_{\boldsymbol{\beta}}), \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}^n_d,$$

and can be obtained by using the mass matrix $\mathbf{M}$ by means of Algorithms 3.5, 3.6 and 3.7 for $d = 1, 2, 3$, respectively.

Algorithm 3.5: 1DStiffMatConst($\mathbf{S}, n$)

Input : Precomputed element mass matrix
$\mathbf{M} = \texttt{1DMassMatConst}(\mathbf{M}, n-1)$.
Output: 1D element stiffness matrix $\mathbf{S}$.

$\mathbf{S} \equiv \mathbf{0}$;
for $i = 1$ **to** 2 **do**
  for $j = 1$ **to** 2 **do**
    $s_{ij} = n^2 * \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i$;
for $\alpha_1 = n-1$ **to** 0 **do**
  for $\beta_1 = n-1$ **to** 0 **do**
    $\alpha_2 = n-1-\alpha_1$;
    $\beta_2 = n-1-\beta_1$;
    $K = \mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}}$;
    $\mathbf{S}_{(\alpha_1+1,\alpha_2),(\beta_1+1,\beta_2)} \mathrel{+}= K * s_{1,1}$;
    $\mathbf{S}_{(\alpha_1+1,\alpha_2),(\beta_1,\beta_2+1)} \mathrel{+}= K * s_{1,2}$;
    $\mathbf{S}_{(\alpha_1,\alpha_2+1),(\beta_1+1,\beta_2)} \mathrel{+}= K * s_{2,1}$;
    $\mathbf{S}_{(\alpha_1,\alpha_2+1),(\beta_1,\beta_2+1)} \mathrel{+}= K * s_{2,2}$;
Return $\mathbf{S}$;

Observe in particular that, with $d \in \{1,2,3\}$, the computation the element stiffness matrix entries takes the compact form:

$$\mathbf{S}_{\boldsymbol{\alpha}+\mathbf{e}_i,\boldsymbol{\beta}+\mathbf{e}_j} \mathrel{+}= \mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}} * s_{i,j}, \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}^{n-1}_d,\ i,j = 1,\ldots,d+1, \tag{3.14}$$

Algorithm 3.6: 2DStiffMatConst($\mathbf{S}, n$)

Input : Precomputed element mass matrix $\mathbf{M} = \texttt{2DMassMatConst}(\mathbf{M}, n-1)$.

Output: 2D element stiffness matrix $\mathbf{S}$.

$\mathbf{S} \equiv \mathbf{0}$;
for $i = 1$ **to** 3 **do**
  for $j = 1$ **to** 3 **do**
    $s_{ij} = n^2 * \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i$;
for $\alpha_1 = n-1$ **to** 0 **do**
  for $\alpha_2 = n-1-\alpha_1$ **to** 0 **do**
    for $\beta_1 = n-1$ **to** 0 **do**
      for $\beta_2 = n-1-\beta_1$ **to** 0 **do**
        $\alpha_3 = n-1-\alpha_1-\alpha_2$;
        $\beta_3 = n-1-\beta_1-\beta_2$;
        $K = \mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}}$;
        $\mathbf{S}_{(\alpha_1+1,\alpha_2,\alpha_3),(\beta_1+1,\beta_2,\beta_3)} \mathrel{+}= K * s_{1,1}$;
        $\mathbf{S}_{(\alpha_1+1,\alpha_2,\alpha_3),(\beta_1,\beta_2+1,\beta_3)} \mathrel{+}= K * s_{1,2}$;
        $\mathbf{S}_{(\alpha_1+1,\alpha_2,\alpha_3),(\beta_1,\beta_2,\beta_3+1)} \mathrel{+}= K * s_{1,3}$;
        $\mathbf{S}_{(\alpha_1,\alpha_2+1,\alpha_3),(\beta_1+1,\beta_2,\beta_3)} \mathrel{+}= K * s_{2,1}$;
        $\mathbf{S}_{(\alpha_1,\alpha_2+1,\alpha_3),(\beta_1,\beta_2+1,\beta_3)} \mathrel{+}= K * s_{2,2}$;
        $\mathbf{S}_{(\alpha_1,\alpha_2+1,\alpha_3),(\beta_1,\beta_2,\beta_3+1)} \mathrel{+}= K * s_{2,3}$;
        $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3+1),(\beta_1+1,\beta_2,\beta_3)} \mathrel{+}= K * s_{3,1}$;
        $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3+1),(\beta_1,\beta_2+1,\beta_3)} \mathrel{+}= K * s_{3,2}$;
        $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3+1),(\beta_1,\beta_2,\beta_3+1)} \mathrel{+}= K * s_{3,3}$;
Return $\mathbf{S}$;

Algorithm 3.7: 3DStiffMatConst($\mathbf{S}, n$)

Input : Precomputed element mass matrix
$\mathbf{M} = \texttt{3DMassMatConst}(\mathbf{M}, n-1)$.
Output: 3D element stiffness matrix $\mathbf{S}$.

$\mathbf{S} \equiv \mathbf{0}$;
for $i = 1$ **to** 4 **do**
  for $j = 1$ **to** 4 **do**
    $s_{ij} = n^2 * \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i$;
for $\alpha_1 = n-1$ **to** 0 **do**
  for $\alpha_2 = n-1-\alpha_1$ **to** 0 **do**
    for $\alpha_3 = n-1-\alpha_1-\alpha_2$ **to** 0 **do**
      for $\beta_1 = n-1$ **to** 0 **do**
        for $\beta_2 = n-1-\beta_1$ **to** 0 **do**
          for $\beta_3 = n-1-\beta_1-\beta_2$ **to** 0 **do**
            $\alpha_4 = n-1-\alpha_1-\alpha_2-\alpha_3$;
            $\beta_4 = n-1-\beta_1-\beta_2-\beta_3$;
            $K = \mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}}$;
            $\mathbf{S}_{(\alpha_1+1,\alpha_2,\alpha_3,\alpha_4),(\beta_1+1,\beta_2,\beta_3,\beta_4)} \mathrel{+}= K * s_{1,1}$;
            $\mathbf{S}_{(\alpha_1+1,\alpha_2,\alpha_3,\alpha_4),(\beta_1,\beta_2+1,\beta_3,\beta_4)} \mathrel{+}= K * s_{1,2}$;
            $\mathbf{S}_{(\alpha_1+1,\alpha_2,\alpha_3,\alpha_4),(\beta_1,\beta_2,\beta_3+1,\beta_4)} \mathrel{+}= K * s_{1,3}$;
            $\mathbf{S}_{(\alpha_1+1,\alpha_2,\alpha_3,\alpha_4),(\beta_1,\beta_2,\beta_3,\beta_4+1)} \mathrel{+}= K * s_{1,4}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2+1,\alpha_3,\alpha_4),(\beta_1+1,\beta_2,\beta_3,\beta_4)} \mathrel{+}= K * s_{2,1}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2+1,\alpha_3,\alpha_4),(\beta_1,\beta_2+1,\beta_3,\beta_4)} \mathrel{+}= K * s_{2,2}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2+1,\alpha_3,\alpha_4),(\beta_1,\beta_2,\beta_3+1,\beta_4)} \mathrel{+}= K * s_{2,3}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2+1,\alpha_3,\alpha_4),(\beta_1,\beta_2,\beta_3,\beta_4+1)} \mathrel{+}= K * s_{2,4}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3+1,\alpha_4),(\beta_1+1,\beta_2,\beta_3,\beta_4)} \mathrel{+}= K * s_{3,1}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3+1,\alpha_4),(\beta_1,\beta_2+1,\beta_3,\beta_4)} \mathrel{+}= K * s_{3,2}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3+1,\alpha_4),(\beta_1,\beta_2,\beta_3+1,\beta_4)} \mathrel{+}= K * s_{3,3}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3+1,\alpha_4),(\beta_1,\beta_2,\beta_3,\beta_4+1)} \mathrel{+}= K * s_{3,4}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3,\alpha_4+1),(\beta_1+1,\beta_2,\beta_3,\beta_4)} \mathrel{+}= K * s_{4,1}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3,\alpha_4+1),(\beta_1,\beta_2+1,\beta_3,\beta_4)} \mathrel{+}= K * s_{4,2}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3,\alpha_4+1),(\beta_1,\beta_2,\beta_3+1,\beta_4)} \mathrel{+}= K * s_{4,3}$;
            $\mathbf{S}_{(\alpha_1,\alpha_2,\alpha_3,\alpha_4+1),(\beta_1,\beta_2,\beta_3,\beta_4+1)} \mathrel{+}= K * s_{4,4}$;
Return $\mathbf{S}$;

with $s_{i,j} = n^2 \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i$, for $i, j = 1, \ldots, d+1$.

The compact formulation (3.14) reflects the similarity of the structure of the stiffness matrix algorithms for different values of d. This similarity allows for a unified complexity analysis for Algorithm 3.5, 3.6 and 3.7: Note that the loop over the pair (i, j) is executed $(d+1)^2$ times, and contains one matrix-vector multiplication, one inner product and one scalar product, thereby yielding a cost of $(d^2+d+1)(d+1)^2$ operations. Now considering the main loop of the algorithm, observe from (3.14) that the computations inside the β_d-loop amount to a loop over the pair (i, j) which is executed $(d+1)^2 \binom{n-1+d}{d}^2$ times, and contains one multiplication. Thus the total cost of the algorithm is

$$(d^2+d+1)(d+1)^2 + (d+1)^2 \binom{n-1+d}{d}^2 = \mathcal{O}(n^{2d}),$$

that is,

$$\begin{cases} 4n^2 + 12 & \text{for } d = 1, \\ \frac{9n^4}{4} + \frac{9n^3}{2} + \frac{9n^2}{4} + 63 & \text{for } d = 2, \\ \frac{4n^6}{9} + \frac{8n^5}{3} + \frac{52n^4}{9} + \frac{16n^3}{3} + \frac{16n^2}{9} + 208 & \text{for } d = 3. \end{cases} \tag{3.15}$$

In order to show that (3.14) provides a correct method for computing the stiffness matrix, we first introduce the following definitions:

Shift Operators

Operator E_i^d on Bernstein polynomials: For $i = 1, \ldots, d+1$ and $n \in \mathbb{N}$, we define E_i^d by

$$E_i^d(B_{\boldsymbol{\beta}}^n) := B_{\boldsymbol{\beta}-\mathbf{e}_i}^{n-1}, \quad \boldsymbol{\beta} \in \mathcal{I}_d^n. \tag{3.16}$$

Moreover, we extend E_i^d, $i = 1, \ldots, d+1$, to all polynomials in $\mathbb{P}_d^n$ by linearity, that is, for any $n \in \mathbb{N}$ and any sequence $\{c_{\boldsymbol{\beta}} : \boldsymbol{\beta} \in \mathcal{I}_d^n\}$, we set

$$E_i^d\Big(\sum_{\boldsymbol{\beta}\in\mathcal{I}_d^n} c_{\boldsymbol{\beta}} B_{\boldsymbol{\beta}}^n\Big) = \sum_{\boldsymbol{\beta}\in\mathcal{I}_d^n} c_{\boldsymbol{\beta}} E_i^d(B_{\boldsymbol{\beta}}^n), \quad i = 1, \ldots, d+1. \tag{3.17}$$

Operator $E^{i,d}$ on coefficient sequences: For $i = 1, \ldots, d+1$, $n \in \mathbb{N}$ and a given sequence $c = \{c_{\boldsymbol{\beta}} : \boldsymbol{\beta} \in \mathcal{I}_d^n\}$, we define $E^{i,d}$ by

$$(E^{i,d}c)_{\boldsymbol{\beta}} := c_{\boldsymbol{\beta}+\mathbf{e}_i}, \quad \boldsymbol{\beta} \in \mathcal{I}_d^{n-1}. \tag{3.18}$$

Remark 3.2.2. Observe from (3.16) and (3.17) that

$$E_i^d\Big(\sum_{\boldsymbol{\beta}\in\mathcal{I}_d^n} c_{\boldsymbol{\beta}} B_{\boldsymbol{\beta}}^n\Big) = \sum_{\boldsymbol{\beta}\in\mathcal{I}_d^n} c_{\boldsymbol{\beta}} B_{\boldsymbol{\beta}-\mathbf{e}_i}^{n-1} = \sum_{\boldsymbol{\beta}\in\mathcal{I}_d^{n-1}} c_{\boldsymbol{\beta}+\mathbf{e}_i} B_{\boldsymbol{\beta}}^{n-1}, \quad i = 1, \ldots, d+1,\ n \in \mathbb{N},$$

which, together with (3.18), yields

$$E_i^d\Big(\sum_{\boldsymbol{\beta}\in\mathcal{I}_d^n} c_{\boldsymbol{\beta}} B_{\boldsymbol{\beta}}^n\Big) = \sum_{\boldsymbol{\beta}\in\mathcal{I}_d^{n-1}} (E^{i,d}c)_{\boldsymbol{\beta}} B_{\boldsymbol{\beta}}^{n-1}, \quad i = 1, \ldots, d+1,\ n \in \mathbb{N}.$$

Operator $E_{i,j}^d$ on mass matrices (3.12): For $i, j = 1, \ldots, d+1$ and $n \in \mathbb{N}$, we define the operator $E_{i,j}^d$ by

$$\left(E_{i,j}^d \mathbf{M}^n\right)_{\boldsymbol{\alpha},\boldsymbol{\beta}} := \mathbf{M}^{n-1}_{\boldsymbol{\alpha}-\mathbf{e}_i,\boldsymbol{\beta}-\mathbf{e}_j}, \quad \boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}_d^n. \tag{3.19}$$

The following formula for the gradients of the Bernstein polynomials is an immediate consequence of (3.8) and (3.16):

$$\nabla B_{\boldsymbol{\beta}}^n = n \sum_{i=1}^{d+1} E_i^d(B_{\boldsymbol{\beta}}^n) \nabla\lambda_i, \quad \boldsymbol{\beta} \in \mathcal{I}_d^n. \tag{3.20}$$

This leads to the following theorem:

Theorem 3.2.3. *The stiffness matrix* $\mathbf{S}^n$ *is given by*

$$\mathbf{S} = n^2 \sum_{i,j=1}^{d+1} E_{i,j}^d(\mathbf{M}) \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i, \tag{3.21}$$

where $\mathbf{M}$ *is the mass matrix described in Theorem 3.2.1. Hence, (3.14) indeed computes the stiffness matrix.*

Proof. For any $\boldsymbol{\alpha}, \boldsymbol{\beta}$ in $\mathcal{I}_d^n$, we have by (3.20),

$$\begin{aligned}
\mathbf{S}_{\boldsymbol{\alpha},\boldsymbol{\beta}} &= n^2 \int_T \Big[\sum_{j=1}^{d+1} E_j^d(B_{\boldsymbol{\beta}}^n)(\mathbf{x}) \nabla\lambda_j \Big] \mathbf{A}(\mathbf{x}) \Big[\sum_{i=1}^{d+1} E_i^d(B_{\boldsymbol{\alpha}}^n)(\mathbf{x}) \nabla\lambda_i \Big] \mathrm{d}\mathbf{x} \\
&= n^2 \sum_{i,j=1}^{d+1} \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i \int_T E_i^d(B_{\boldsymbol{\alpha}}^n)(\mathbf{x}) E_j^d(B_{\boldsymbol{\beta}}^n)(\mathbf{x}) \mathrm{d}\mathbf{x}, \\
&= n^2 \sum_{i,j=1}^{d+1} \mathbf{M}_{\boldsymbol{\alpha}-\mathbf{e}_i, \boldsymbol{\beta}-\mathbf{e}_j}^{n-1} \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i.
\end{aligned}$$

Inserting (3.19) into the last equation gives (3.21). □

Although Algorithms 3.5, 3.6 and 3.7 achieve the optimal complexity $\mathcal{O}(n^{2d})$ for $d = 1, 2, 3$, they rely on the pre-computation of the mass matrix of lower order. As a matter of fact, it is possible to directly compute the stiffness matrix only using precomputed binomial coefficients, as shown in Algorithms 3.8, 3.9, 3.10, for $d = 1, 2, 3$, respectively.

Algorithm 3.8: 1DStiffMatConstDirect$(\mathbf{S}, n)$

Input : Precomputed binomial coeffficients $\{C_q^{p+q} : \ 0 \leq p \leq n, 0 \leq q \leq n\}$

Output: 1D element stiffness matrix.

1 $r = \dfrac{n^2}{C_{n-1}^{2n-2} * (2n-1)}$;

2 **for** $i = 1$ **to** 2 **do**

3 $\quad$ **for** $j = 1$ **to** 2 **do**

4 $\quad\quad s_{ij} = r * \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i$;

5 `//Same lines as 1DMassMatConst` $(\mathbf{S}, n-1)$ `(cf. Algorithm 3.2), with line 4 replaced with the line:`

6 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1}$;

7 `//and line 6 replaced with lines 7-13 of 1DStiffMatConst`$(\mathbf{S}, n)$ `(cf. Algorithm 3.5), where:`

8 $K \leftarrow w_2$;

9 **Return S**;

Algorithm 3.9: 2DStiffMatConstDirect$(\mathbf{S}, n)$

Input : Precomputed binomial coeffficients $\{C_q^{p+q} : \ 0 \leq p \leq n, 0 \leq q \leq n\}$

Output: 2D element stiffness matrix.

1 $r = \dfrac{n^2}{C_{n-1}^{2n-2} * C_2^{2n}}$;

2 **for** $i = 1$ **to** 3 **do**

3 $\quad$ **for** $j = 1$ **to** 3 **do**

4 $\quad\quad s_{ij} = r * \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i$;

5 `//Same lines as 2DMassMatConst` $(\mathbf{S}, n-1)$ `(cf. Algorithm 3.3), with line 4 replaced with the line:`

6 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1}$;

7 `//and line 9 replaced with lines 9-20 of 2DStiffMatConst`$(\mathbf{S}, n)$ `(cf. Algorithm 3.6), where:`

8 $K \leftarrow w_3$;

9 **Return S**;

Algorithm 3.10: 3DStiffMatConstDirect$(\mathbf{S}, n)$

Input : Precomputed binomial coeffficients
$\{C_q^{p+q} : \ 0 \leq p \leq n+1, 0 \leq q \leq n\}$
Output: 3D element stiffness matrix.

1 $r = \dfrac{n^2}{C_{n-1}^{2n-2} * C_3^{2n+1}}$;
2 **for** $i = 1$ **to** 4 **do**
3 **for** $j = 1$ **to** 4 **do**
4 $s_{ij} = r * \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i$;
5 `//Same lines as 3DMassMatConst` $(\mathbf{S}, n-1)$ `(cf. Algorithm 3.4), with line 4 replaced with the line:`
6 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1}$;
7 `//and line 12 replaced with lines 11-29 of` 3DStiffMatConst$(\mathbf{S}, n)$ `(cf. Algorithm 3.7), where:`
8 $K \leftarrow w_4$;
9 **Return S**;

Using a similar argument as the one used for the computation of the stiffness matrix using a precomputed mass matrix, it is straightforward to find that, up to the computation of the scaling constant r and two additional multiplications in the innermost loop, the number of operations required for computing the element stiffness matrix, only based on precomputed binomial coefficients, is also given by (3.15). Note however from the input of Algorithm 3.8, Algorithm 3.9 and Algorithm 3.10 that, for a given polynomial order n, the precomputed binomial matrix is at most $(n+1)$ by n, whereas the elemental mass matrix is of dimension $\binom{n+d}{d} = \mathcal{O}(n^d)$. As a result, the numerical experiments seem to indicate that memory access calls to the binomial matrix are slightly faster than those to the mass matrix. In other words, since they have the same complexity, the algorithm based on the direct computation of the stiffness matrix in terms of precomputed binomial coefficients is recommended over that based on the (precomputed) mass matrix.

3.2.4 Convective Matrix

The convective matrix $\mathbf{V}$ is defined by

$$\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \mathbf{V}^n_{\boldsymbol{\alpha},\boldsymbol{\beta}} := \int_T B^n_{\boldsymbol{\beta}}(\mathbf{x})\mathbf{b}(\mathbf{x}) \cdot \nabla B^n_{\boldsymbol{\alpha}}(\mathbf{x})\mathrm{d}\mathbf{x}, \quad \boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}^n_d, \tag{3.22}$$

and can be computed by means of Algorithms 3.11, 3.12, 3.13 for $d = 1, 2, 3$, respectively.

Algorithm 3.11: 1DConvMatConstDirect($\mathbf{V}, n$)

Input : Precomputed binomial coefficients $\{C_p^{p+q} : \ 0 \le p \le n-1, \ 0 \le q \le n\}$.
Output: 1D element convective matrix $\mathbf{V}$.

$\mathbf{V} \equiv \mathbf{0}$;
$r = \dfrac{|T|}{C_n^{2n-1} * 2}$;
for $i = 1$ **to** 2 **do**
    $v_i = r * \mathbf{b}_{|T} \cdot \nabla\lambda_i$;
for $\boldsymbol{\alpha}_1 = n-1$ **to** 0 **do**
    for $\boldsymbol{\beta}_1 = n$ **to** 0 **do**
        $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1}$;
        $\alpha_2 = n-1-\alpha_1, \ \beta_2 = n-\beta_2$;
        $w_2 = w_1 * C_{\alpha_2}^{\alpha_2+\beta_2}$;
        $\mathbf{V}_{(\alpha_1+1,\alpha_2),\boldsymbol{\beta}} \mathrel{+}= w_2 * v_1$;
        $\mathbf{V}_{(\alpha_1,\alpha_2+1),\boldsymbol{\beta}} \mathrel{+}= w_2 * v_2$;
Return $\mathbf{V}$;

Algorithm 3.12: 2DConvMatConstDirect($\mathbf{V}, n$)

Input : Precomputed binomial coefficients
$\{C_p^{p+q} :\ 0 \le p \le n-1,\ 0 \le q \le n\}$.
Output: 2D element convective matrix $\mathbf{V}$.

$\mathbf{V} \equiv \mathbf{0}$;
$r = \dfrac{n * |T|}{C_n^{2n-1} * C_2^{2n+1}}$;
for $i = 1$ **to** 3 **do**
  $v_i = r * \mathbf{b}_{|T} \cdot \nabla\lambda_i$;
for $\alpha_1 = n-1$ **to** 0 **do**
  for $\alpha_2 = n-1-\alpha_1$ **to** 0 **do**
    for $\beta_1 = n$ **to** 0 **do**
      $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1}$;
      for $\beta_2 = n-\beta_1$ **to** 0 **do**
        $w_2 = w_1 * C_{\alpha_2}^{\alpha_2+\beta_2}$;
        $\alpha_3 = n-1-\alpha_1-\alpha_2,\ \beta_3 = n-\beta_1-\beta_2$;
        $w_3 = w_2 * C_{\alpha_3}^{\alpha_3+\beta_3}$;
        $\mathbf{V}_{(\alpha_1+1,\alpha_2,\alpha_3),\boldsymbol{\beta}} \mathrel{+}= w_3 * v_1$;
        $\mathbf{V}_{(\alpha_1,\alpha_2+1,\alpha_3),\boldsymbol{\beta}} \mathrel{+}= w_3 * v_2$;
        $\mathbf{V}_{(\alpha_1,\alpha_2,\alpha_3+1),\boldsymbol{\beta}} \mathrel{+}= w_3 * v_3$;
Return $\mathbf{V}$;

Algorithm 3.13: 3DConvMatConstDirect($\mathbf{V}, n$)

Input : Precomputed binomial coefficients
$\{C_p^{p+q} : \ 0 \le p \le n-1, \ 0 \le q \le n\}$.
Output: 3D element convective matrix $\mathbf{V}$.

$\mathbf{V} \equiv \mathbf{0}$;
$r = \dfrac{n * |T|}{C_n^{2n-1} * C_3^{2n+2}}$;
for $i = 1$ **to** 4 **do**
  $v_i = r * \mathbf{b}_{|T} \cdot \nabla\lambda_i$;
for $\alpha_1 = n-1$ **to** 0 **do**
  for $\alpha_2 = n-1-\alpha_1$ **to** 0 **do**
    for $\alpha_3 = n-1-\alpha_1-\alpha_2$ **to** 0 **do**
      for $\beta_1 = n$ **to** 0 **do**
        $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1}$;
        for $\beta_2 = n-\beta_1$ **to** 0 **do**
          $w_2 = w_1 * C_{\alpha_2}^{\alpha_2+\beta_2}$;
          for $\beta_3 = n-1-\beta_1-\beta_2$ **to** 0 **do**
            $w_3 = w_2 * C_{\alpha_3}^{\alpha_3+\beta_3}$;
            $\alpha_4 = n-1-\alpha_1-\alpha_2-\alpha_3, \ \beta_4 = n-\beta_1-\beta_2-\beta_3$;
            $w_4 = w_3 * C_{\alpha_4}^{\alpha_4+\beta_4}$;
            $\mathbf{V}_{(\alpha_1+1,\alpha_2,\alpha_3,\alpha_4),\boldsymbol{\beta}} \mathrel{+}= w_3 * v_1$;
            $\mathbf{V}_{(\alpha_1,\alpha_2+1,\alpha_3,\alpha_4),\boldsymbol{\beta}} \mathrel{+}= w_3 * v_2$;
            $\mathbf{V}_{(\alpha_1,\alpha_2,\alpha_3+1,\alpha_4),\boldsymbol{\beta}} \mathrel{+}= w_3 * v_3$;
            $\mathbf{V}_{(\alpha_1,\alpha_2,\alpha_3,\alpha_4+1),\boldsymbol{\beta}} \mathrel{+}= w_3 * v_4$;
Return $\mathbf{V}$;

Observe that, for $d = 1, 2, 3$, the computation of the convective matrix entries takes the compact form:

$$\mathbf{V}_{\boldsymbol{\alpha}+\mathbf{e}_i,\boldsymbol{\beta}} \mathrel{+}= w_d * v_i, \quad \boldsymbol{\alpha} \in \mathcal{I}_d^{n-1}, \boldsymbol{\beta} \in \mathcal{I}_d^n, \ i = 1, \ldots, d+1, \tag{3.23}$$

where $v_i = (n|T|)/(C_n^{2n-1} C_d^{2n-1+d}) \mathbf{b}_{|T} \cdot \nabla\lambda_i$, for $i = 1, \ldots, d+1$.

We now proceed to the complexity analysis of the convective matrix algorithm. For simplicity, we focus on the case $d = 3$. The other two cases are similar. We start with the loop over i which is executed four times, and contains one inner product and one multiplication, which amounts to four operations. Thus, taking account of the operations required for computing the scaling constant r,

the number of operations needed in order to set up the field $(v_i)_{1\leq i\leq 4}$ is 4^2+3. The loop over β_2 is executed $\binom{n+2}{3}\binom{n+2}{2}$ times, and contains one multiplication. Similarly, the loop over β_3 is executed $\binom{n+2}{3}\binom{n+3}{3}$ times, and contains six multiplications. Thus, the total number of operations involved in the computation of the 3D convective matrix is:

$$4^2+3+\binom{n+2}{3}\binom{n+2}{2}+6\binom{n+2}{3}\binom{n+3}{3}.$$

Using a similar argument, we find that the number of operations required for computing the 1D and 2D convective matrices are respectively given by

$$\begin{aligned}&2^2+3+3n(n+1),\\&3^2+3+5\binom{n+1}{2}\binom{n+2}{2}.\end{aligned}$$

To summarize, the complexity associated with the computation of the convective matrix is:

$$\begin{cases}3n^2+3n+7 & \text{for } d=1,\\ \frac{5n^4}{4}+5n^3+\frac{25n^2}{4}+\frac{5n}{2}+12 & \text{for } d=2,\\ \frac{n^6}{6}+\frac{19n^5}{12}+\frac{17n^4}{3}+\frac{115n^3}{12}+\frac{23n^2}{3}+\frac{7n}{3}+19 & \text{for } d=3.\end{cases} \tag{3.24}$$

It is not difficult to establish that Algorithms 3.11, 3.12 and 3.13 do indeed compute the convective matrix $\mathbf{V}$, for $d=1,2,3$. Note from (3.23) that $\mathbf{V}$ is essentially given by

$$\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}}=\frac{n|T|}{\binom{2n-1}{n}\binom{2n-1+d}{d}}\sum_{i=1}^{d+1}\mathbf{b}_{|T}\cdot\nabla\lambda_i\binom{\boldsymbol{\alpha}-\mathbf{e}_i+\boldsymbol{\beta}}{\boldsymbol{\beta}},\quad \boldsymbol{\alpha},\boldsymbol{\beta}\in\mathcal{I}_d^n.$$

Theorem 3.2.4. *Let $n\in\mathbb{N}$. The convective matrix $\mathbf{V}$ is given by*

$$\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}}=\frac{n|T|}{\binom{2n-1}{n}\binom{2n-1+d}{d}}\mathbf{b}_{|T}\cdot\sum_{i=1}^{d+1}\nabla\lambda_i\binom{\boldsymbol{\alpha}-\mathbf{e}_i+\boldsymbol{\beta}}{\boldsymbol{\beta}},\quad \boldsymbol{\alpha},\boldsymbol{\beta}\in\mathcal{I}_d^n. \tag{3.25}$$

Proof. Note from (3.20), (3.22) and (1.6) that, for $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}_d^n$,

$$\begin{aligned}\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}} =& n \sum_{i=1}^{d+1} \int_T B^{n-1}_{\boldsymbol{\alpha}-\mathbf{e}_i} B^n_{\boldsymbol{\beta}} \mathbf{b} \cdot \nabla \lambda_i \\ =& \frac{n}{\binom{2n-1}{n}} \sum_{i=1}^{d+1} \binom{\boldsymbol{\alpha}-\mathbf{e}_i+\boldsymbol{\beta}}{\boldsymbol{\beta}} \mathbf{b}_{|T} \cdot \nabla \lambda_i \int_T B^{2n-1}_{\boldsymbol{\alpha}-\mathbf{e}_i+\boldsymbol{\beta}}. \end{aligned} \tag{3.26}$$

Inserting (3.10) into the last line yields

$$\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \frac{n|T|}{\binom{2n-1}{n}\binom{2n-1+d}{d}} \mathbf{b}_{|T} \cdot \sum_{i=1}^{d+1} \nabla \lambda_i \binom{\boldsymbol{\alpha}-\mathbf{e}_i+\boldsymbol{\beta}}{\boldsymbol{\beta}}, \quad \boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}_d^n. \tag{3.27}$$

□

Remark 3.2.5. The degree raising formula [64, Chapter 2] yields

$$B^{n-1}_{\boldsymbol{\alpha}-\mathbf{e}_i} = \frac{1}{n} \sum_{j=1}^{d+1} (\alpha_j - \delta_{i,j} + 1) B^n_{\boldsymbol{\alpha}-\mathbf{e}_i+\mathbf{e}_j}, \quad \boldsymbol{\alpha} \in \mathcal{I}_d^n, \ i = 1, \ldots, d+1,$$

which, inserted into the first line of (3.26), gives

$$\begin{aligned}\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}} &= \sum_{i,j=1}^{d+1} (\alpha_j - \delta_{i,j} + 1) \int_T B^n_{\boldsymbol{\alpha}-\mathbf{e}_i+\mathbf{e}_j}(\mathbf{x}) B^n_{\boldsymbol{\beta}}(\mathbf{x}) \mathbf{b}(\mathbf{x}) \cdot \nabla \lambda_i \mathrm{d}\mathbf{x} \\ &= \sum_{i=1}^{d+1} \mathbf{b}_{|T} \cdot \nabla \lambda_i \sum_{j=1}^{d+1} (\alpha_j - \delta_{i,j} + 1) \mathbf{M}_{\boldsymbol{\alpha}-\mathbf{e}_i+\mathbf{e}_j,\boldsymbol{\beta}}, \quad \boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}_d^n. \end{aligned}$$

Hence, an alternative way to compute the convective matrix $\mathbf{V}$ is by means of the weighted sum of mass matrix entries given in Remark 3.2.5. Note that the corresponding algorithms will be recommended only if they have a lower complexity than in (3.24).

The convective matrix can be computed in terms of mass matrix entries, as proposed in Algorithms 3.14, 3.15, and 3.16, for $d = 1, 2, 3$, respectively.

Algorithm 3.14: 1DConvMatConst($\mathbf{V}, n$)

Input : Precomputed element mass matrix $\mathbf{M} = \texttt{1DMassMatConst}(\mathbf{M}, n)$.
Output: 1D element convective matrix $\mathbf{V}$.

$\mathbf{V} \equiv \mathbf{0}$;
for $i = 1$ **to** 2 **do**
  $v_i = \mathbf{b}_{|T} \cdot \nabla\lambda_i$;
for $\alpha_1 = n$ **to** 0 **do**
  for $\beta_1 = n$ **to** 0 **do**
    for $i = 1$ **to** 2 **do**
      $s_i = 0$;
      for $j = 1$ **to** 2 **do**
        if $\boldsymbol{\alpha} - \mathbf{e}_i + \mathbf{e}_j \geq (0, 0)$ **then**
          $s_i \mathrel{+}= (\alpha_j - \delta_{i,j} + 1) * \mathbf{M}_{\boldsymbol{\alpha}-\mathbf{e}_i+\mathbf{e}_j,\boldsymbol{\beta}}$;
      $\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}} \mathrel{+}= v_i * s_i$;
Return $\mathbf{V}$;

Algorithm 3.15: 2DConvMatConst($\mathbf{V}, n$)

Input : Precomputed element mass matrix $\mathbf{M} = \texttt{2DMassMatConst}(\mathbf{M}, n)$.
Output: 2D element convective matrix $\mathbf{V}$.

$\mathbf{V} \equiv \mathbf{0}$;
for $i = 1$ **to** 3 **do**
  $v_i = \mathbf{b}_{|T} \cdot \nabla\lambda_i$;
for $\alpha_1 = n$ **to** 0 **do**
  for $\alpha_2 = n - \alpha_1$ **to** 0 **do**
    for $\beta_1 = n$ **to** 0 **do**
      for $\beta_2 = n - \beta_1$ **to** 0 **do**
        for $i = 1$ **to** 3 **do**
          $s_i = 0$;
          for $j = 1$ **to** 3 **do**
            if $\boldsymbol{\alpha} - \mathbf{e}_i + \mathbf{e}_j \geq (0, 0, 0)$ **then**
              $s_i \mathrel{+}= (\alpha_j - \delta_{i,j} + 1) * \mathbf{M}_{\boldsymbol{\alpha}-\mathbf{e}_i+\mathbf{e}_j,\boldsymbol{\beta}}$;
          $\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}} \mathrel{+}= v_i * s_i$;
Return $\mathbf{V}$;

Algorithm 3.16: 3DConvMatConst($\mathbf{V}, n$)

Input : Precomputed element mass matrix $\mathbf{M} = \texttt{3DMassMatConst}(\mathbf{M}, n)$.
Output: 3D element convective matrix $\mathbf{V}$.

$\mathbf{V} \equiv \mathbf{0}$;
for $i = 1$ **to** 4 **do**
  $v_i = \mathbf{b}_{|T} \cdot \nabla \lambda_i$;
for $\alpha_1 = n$ **to** 0 **do**
  for $\alpha_2 = n - \alpha_1$ **to** 0 **do**
    for $\alpha_3 = n - 1 - \alpha_1 - \alpha_2$ **to** 0 **do**
      for $\beta_1 = n$ **to** 0 **do**
        for $\beta_2 = n - \beta_1$ **to** 0 **do**
          for $\beta_3 = n - \beta_1 - \beta_2$ **to** 0 **do**
            for $i = 1$ **to** 4 **do**
              $s_i = 0$;
              for $j = 1$ **to** 4 **do**
                if $\boldsymbol{\alpha} - \mathbf{e}_i + \mathbf{e}_j \geq (0,0,0,0)$ **then**
                  $s_i \mathrel{+}= (\alpha_j - \delta_{i,j} + 1) * \mathbf{M}_{\boldsymbol{\alpha} - \mathbf{e}_i + \mathbf{e}_j, \boldsymbol{\beta}}$;
              $\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}} \mathrel{+}= v_i * s_i$;
Return $\mathbf{V}$;

We now proceed with the complexity analysis of the convective matrix algorithm, when based on the precomputed mass matrix. Once again, we focus on the case $d = 3$. The other two cases are handled in a similar way. Observe that the first loop over i is executed four times, and contains one inner product which amounts to three multiplications. Thus, the number of operations needed in order to set up the field $(v_i)_{1 \leq i \leq 4}$ is $4 \times 3 = 12$. Moving to the main part of the algorithm, note that the innermost loop over j is executed $4^2 \binom{n+3}{3}^2$, and contains at most one multiplication. Similarly, the (second) loop over i is executed $4 \binom{n+3}{3}^2$ times, and contains one multiplication. Hence, the total number of operations involved in the computation of the 3D convective matrix is at most

$$12 + 4^2 \binom{n+3}{3}^2 + 4 \binom{n+3}{3}^2.$$

Using a similar argument, we find that the complexity associated with the

computation of the 1D and 2D convective matrices is at most given by

$$2 + 2^2(n+1)^2 + 2(n+1)^2,$$
$$6 + 3^2\binom{n+2}{2}^2 + 3\binom{n+2}{2}^2,$$

respectively. To summarize, the number of operations required in order to compute the convective matrix, based on precomputed mass matrix, is at most

$$\begin{cases} 6n^2 + 12n + 8 & \text{for } d = 1, \\ 3n^4 + 18n^3 + 39n^2 + 36n + 18 & \text{for } d = 2, \\ \frac{5n^6}{9} + \frac{20n^5}{3} + \frac{290n^4}{9} + 80n^3 + \frac{965n^2}{9} + \frac{220n}{3} + 32 & \text{for } d = 3. \end{cases} \tag{3.28}$$

Though the alternative approach for computing the convective matrix presents the advantage of re-using the computed mass matrix, the comparison of (3.24) and (3.28) is in favor of the direct computation of the convective matrix, that is, only using precomputed binomial coefficients. Moreover, the presence of the conditional checking inside the innermost loop over j might also affect the computational speed. Once again, the direct approach is recommended.

Note that, in addition to optimal complexity, the formulas presented in this section are explicit, thus allowing for efficient computations of the elemental quantities, with no (quadrature) error from approximating the integrals. Section A.1 provides useful applications of the proposed algorithms for the mass and stiffness matrices.

3.3 Optimal Assembly II: Variable Data

In the previous section we have seen that in the case of piecewise constant data the load vector, mass, stiffness and convective matrices are given by the explicit formulae (3.11), (3.12), (3.21) and (3.25), respectively. The cost of their computation for a single simplex is optimal in the sense it is bounded by a constant times the number of entries, that is $\mathcal{O}(n^d)$ for the load vector and $\mathcal{O}(n^{2d})$ for the

mass, stiffness and convective matrices.

When using the shape functions of a fixed degree on all simplices of a triangulation, it is important to observe that the formulae (3.11), (3.12), (3.21) are not strongly dependent on the particular simplex T. For example the normalized mass matrix $\tilde{\mathbf{M}}$ given by

$$\tilde{\mathbf{M}}_{\boldsymbol{\alpha},\boldsymbol{\beta}} := \frac{\binom{\boldsymbol{\alpha}+\boldsymbol{\beta}}{\boldsymbol{\alpha}}}{\binom{2n}{n}\binom{2n+d}{d}}, \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}_d^n,$$

may be pre-computed according to `1DMassMatConst`, `2DMassMatConst` and `3DMassMatConst`, for $d = 1, 2, 3$, and then the mass matrix for each simplex T obtained as $\mathbf{M} = c_{|T}|T|\tilde{\mathbf{M}}$. Similarly, the matrices $E_{i,j}^d(\tilde{\mathbf{M}})$, $i, j = 1, \ldots, d+1$, may be precomputed to save time when applying (3.21) to each simplex T.

If the data is variable, that is $f, \mathbf{A}, \mathbf{b}, c$ in (3.3) are functions, then the dependence on the particular element T is much stronger than in the case of piecewise constant data. Moreover, the computation of the integrals cannot be done analytically in general and requires quadrature rules. For the quadrature accuracy not to affect the convergence of the finite element scheme, the quadrature needs to be exact for polynomials of degree $2n$, where n is the order of the finite element [20, Section 4.3] [24, Section 3]. In particular, using the q-point Stroud conical product rule [73, Chapter 2] which is exact for polynomials of degree at most $2q-1$, one can see that the choice

$$q = n + 1, \tag{3.29}$$

meets the above-mentioned requirement. In this section we present algorithms to compute the load vector, mass and stiffness matrices with optimal cost in the case of spatially varying data. For simplicity, we will not distinguish between equality and approximation in the remaining part of the chapter. That is, depending on the context, the expression $A = B$ may either reads "A is equal to B", or "B approximates A".

3.3.1 Load Vector

Recall that the load vector coincides with the *B-moment vector* $(\mu^n_{\boldsymbol{\alpha}})_{\boldsymbol{\alpha}\in\mathcal{I}^n_d}$. Note that with q as in (3.29) the load vector will be computed exactly as soon as f is a polynomial of degree at most n. The following result is a direct consequence of Theorem 2.2.4.

Theorem 3.3.1. *Let $n \in \mathbb{Z}_+$ and $d \in \{1,2,3\}$. Using the conical product rule with $q = n+1$, the element load vector can be computed with a numerical cost $\mathcal{O}(n^{d+1})$ by means of Algorithms* `Moment1D`, `Moment2D` *and* `Moment3D` *for $d = 1, 2$ and 3, respectively.*

3.3.2 Mass Matrix

Using a similar approach as for the load vector, we now want to use the conical product rule to compute the mass matrix $\mathbf{M}$ defined by

$$\mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}} := \int_T c(\mathbf{x}) B^n_{\boldsymbol{\alpha}}(\mathbf{x}) B^n_{\boldsymbol{\beta}}(\mathbf{x}) \mathrm{d}\mathbf{x} = \left(cB^n_{\boldsymbol{\alpha}}, B^n_{\boldsymbol{\beta}}\right)_T, \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}^n_d. \tag{3.30}$$

The mass matrix $\mathbf{M}$ can be obtained by means of Algorithms 3.17, 3.18, 3.19, for $d = 1, 2, 3$, respectively.

Algorithm 3.17: 1DMassMat$(\mathbf{M}, n, \{\mu^{2n}_{\boldsymbol{\beta}}(c),\ \boldsymbol{\beta} \in \mathcal{I}^{2n}_1\})$

Input : Bernstein-Bézier moments $\{\mu^{2n}_{\boldsymbol{\beta}}(c) :\ \boldsymbol{\beta} \in \mathcal{I}^{2n}_1\}$ obtained by means of the Stroud conical product rule with $q = n+1$, and precomputed binomial coefficients $\{C^{i+j}_i :\ 0 \le i,j \le n\}$.

Output: 1D element mass matrix $\mathbf{M}$.

1 `//Same lines as 1DMassMatConst`$(\mathbf{M}, n)$ `(cf. Algorithm 3.2), with line 4 replaced with the line:`

2 $w_1 = C^{\alpha_1+\beta_1}_{\alpha_1}/C^{2n}_n$;

3 `//and line 6 replaced with the line:`

4 $\mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = w_2 * \mu^{2n}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}(c)$;

5 **Return** $\mathbf{M}$;

Algorithm 3.18: 2DMassMat($\mathbf{M}, n, \{\mu_{\boldsymbol{\beta}}^{2n}(c),\ \boldsymbol{\beta} \in \mathcal{I}_2^{2n}\}$)

Input : Bernstein-Bézier moments $\{\mu_{\boldsymbol{\beta}}^{2n}(c) :\ \boldsymbol{\beta} \in \mathcal{I}_2^{2n}\}$ obtained by means of the Stroud conical product rule with $q = n + 1$, and precomputed binomial coefficients $\{C_i^{i+j} :\ 0 \le i, j \le n\}$.

Output: 2D element mass matrix $\mathbf{M}$.

1 `//Same lines as 2DMassMatConst`$(\mathbf{M}, n)$ `(cf. Algorithm 3.3), with line 4 replaced with the line:`

2 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1}/C_n^{2n}$;

3 `//and line 9 replaced with the line:`

4 $\mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = w_3 * \mu_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{2n}(c)$;

5 **Return** $\mathbf{M}$;

Algorithm 3.19: 3DMassMat($\mathbf{M}, n, \{\mu_{\boldsymbol{\beta}}^{2n}(c),\ \boldsymbol{\beta} \in \mathcal{I}_3^{2n}\}$)

Input : Bernstein-Bézier moments $\{\mu_{\boldsymbol{\beta}}^{2n}(c) :\ \boldsymbol{\beta} \in \mathcal{I}_3^{2n}\}$ obtained by means of the Stroud conical product rule with $q = n + 1$, and precomputed binomial coefficients $\{C_i^{i+j} :\ 0 \le i, j \le n\}$.

Output: 3D element mass matrix $\mathbf{M}$.

1 `//Same lines as 3DMassMatConst`$(\mathbf{M}, n)$ `(cf. Algorithm 3.4), with line 4 replaced with the line:`

2 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1}/C_n^{2n}$;

3 `//and line 12 replaced with the line:`

4 $\mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = w_4 * \mu_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{2n}(c)$;

5 **Return** $\mathbf{M}$;

We now proceed with discussing step by step the complexity of the proposed algorithms. First observe that, for $d = 1, 2, 3$, the only difference between the algorithms associated with constant and variable data lies in the computation of the B-moments of order $2n$, and one additional multiplication in the innermost loop. Thus, in addition to the cost corresponding to the B-moments (which is $\mathcal{O}(n^{d+1})$), and those given in (3.13), the number of operations involved in the computation of the mass matrix with variable coefficients is $\binom{n+d}{d}^2$, that is,

$$\begin{cases} n^2 + 2n + 1 & \text{for } d = 1, \\ \frac{n^4}{4} + 2n^3 + 6n^2 + 8n + 4 & \text{for } d = 2, \\ \frac{n^6}{36} + \frac{n^5}{3} + \frac{29n^4}{18} + 4n^3 + \frac{193n^2}{36} + \frac{11n}{3} + 1 & \text{for } d = 3. \end{cases}$$

Combining the above equations with (3.13) and Theorem 2.2.4 implies that the number of operations involved in the computation of the mass matrix is $\mathcal{O}(n^{2d})$, for $d = 1, 2, 3$.

Next, we analyze the output of Algorithms 3.17, 3.18 and 3.19. From the description of the algorithms, one can see that the mass matrix is computed as

$$\frac{\binom{\boldsymbol{\alpha}+\boldsymbol{\beta}}{\boldsymbol{\alpha}}}{\binom{2n}{n}} \left(c, B^{2n}_{\boldsymbol{\alpha}+\boldsymbol{\beta}} \right)_T.$$

But this is the right formula for $\mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}}$, as follows from (3.30) and (1.6).

We can now state the following result which is a direct consequence of the above complexity analysis.

Theorem 3.3.2. *Let $n \in \mathbb{N}$, and $d \in \{1, 2, 3\}$. Using the conical product rule with $q = n+1$, the element mass matrix of degree n can be computed with a numerical cost $\mathcal{O}(n^{2d})$ by means of Algorithms* `1DMassMat`*,* `2DMassMat`*,* `3DMassMat`*, for $d = 1, 2, 3$, respectively.*

3.3.3 Stiffness Matrix

In this section, we want to evaluate the stiffness matrix $\mathbf{S}$ defined by

$$\mathbf{S}_{\boldsymbol{\alpha},\boldsymbol{\beta}} := \int_T \nabla B^n_{\boldsymbol{\beta}}(\mathbf{x}) \cdot \mathbf{A}(\mathbf{x}) \cdot \nabla B^n_{\boldsymbol{\alpha}}(\mathbf{x}) \mathrm{d}\mathbf{x}$$

The stiffness matrix $\mathbf{S}$ can be computed using Algorithms 3.20, 3.21, 3.22, for $d = 1, 2, 3$, respectively.

Algorithm 3.20: 1DStiffMat$(\mathbf{S}, n, \{\mu^{2n-2}_{\boldsymbol{\beta}}(\mathbf{A}) : \ \boldsymbol{\beta} \in \mathcal{I}^{2n-2}_1\})$

Input : Bernstein-Bézier moments $\{\mu^{2n-2}_{\boldsymbol{\beta}}(\mathbf{A}) : \ \boldsymbol{\beta} \in \mathcal{I}^{2n-2}_1\}$ obtained by means of the Stroud conical product rule with $q = n + 1$, and precomputed binomial coefficients $\{C^{i+j}_j : \ 0 \le i, j \le n-1\}$.

Output: 1D Element stiffness matrix $\mathbf{S}$.

1 **for** $i = 1$ **to** 2 **do**

2 **for** $j = 1$ **to** 2 **do**

3 Compute $\tilde{\mu}^{(i,j)}_{\boldsymbol{\beta}} = \nabla\lambda_j \mu^{2n-2}_{\boldsymbol{\beta}}(\mathbf{A}) \nabla\lambda_i, \quad \boldsymbol{\beta} \in \mathcal{I}^{2n-2}_1$;

4 `//Same lines as 1DMassMatConst`$(\mathbf{S}, n-1)$ `(cf. Algorithm 3.2), with line 4 replaced with the line:`

5 $w_1 = C^{\alpha_1+\beta_1}_{\alpha_1} * n^2 / C^{2n-2}_{n-1}$;

6 `//and line 6 replaced with lines 7-13 of 1DStiffMatConst`$(\mathbf{S}, n)$ `(cf. Algorithm 3.5), where:`

7 $K \leftarrow w_2$ and $s_{i,j} \leftarrow \tilde{\mu}^{(i,j)}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}$, $i, j = 1, 2$;

8 **Return** $\mathbf{S}$;

Algorithm 3.21: 2DStiffMat($\mathbf{S}, n, \{\mu_{\boldsymbol{\beta}}^{2n-2}(\mathbf{A}) : \boldsymbol{\beta} \in \mathcal{I}_2^{2n-2}\}$)

Input : Bernstein-Bézier moments $\{\mu_{\boldsymbol{\beta}}^{2n-2}(\mathbf{A}) : \boldsymbol{\beta} \in \mathcal{I}_2^{2n-2}\}$ obtained by means of the Stroud conical product rule with $q = n+1$, and precomputed binomial coefficients $\{C_j^{i+j} : 0 \le i, j \le n-1\}$.

Output: 2D Element stiffness matrix $\mathbf{S}$.

1 **for** $i = 1$ **to** 3 **do**

2 **for** $j = 1$ **to** 3 **do**

3 Compute $\tilde{\mu}_{\boldsymbol{\beta}}^{(i,j)} = \nabla\lambda_j \mu_{\boldsymbol{\beta}}^{2n-2}(\mathbf{A}) \nabla\lambda_i, \quad \boldsymbol{\beta} \in \mathcal{I}_2^{2n-2}$;

4 `//Same lines as 2DMassMatConst`$(\mathbf{S}, n-1)$ `(cf. Algorithm 3.3), with line 4 replaced with the line:`

5 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1} * n^2 / C_{n-1}^{2n-2}$;

6 `//and line 9 replaced with lines 9-20 of 2DStiffMatConst`$(\mathbf{S}, n)$ `(cf. Algorithm 3.6), where:`

7 $K \leftarrow w_3$ and $s_{i,j} \leftarrow \tilde{\mu}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(i,j)}$, $i, j = 1, 2, 3$;

8 **Return S**;

Algorithm 3.22: 3DStiffMat($\mathbf{S}, n, \{\mu_{\boldsymbol{\beta}}^{2n-2}(\mathbf{A}) : \boldsymbol{\beta} \in \mathcal{I}_3^{2n-2}\}$)

Input : Bernstein-Bézier moments $\{\mu_{\boldsymbol{\beta}}^{2n-2}(\mathbf{A}) : \boldsymbol{\beta} \in \mathcal{I}_3^{2n-2}\}$ obtained by means of the Stroud conical product rule with $q = n+1$, and precomputed binomial coefficients $\{C_j^{i+j} : 0 \le i, j \le n-1\}$.

Output: 3D Element stiffness matrix $\mathbf{S}$.

1 **for** $i = 1$ **to** 4 **do**

2 **for** $j = 1$ **to** 4 **do**

3 Compute $\tilde{\mu}_{\boldsymbol{\beta}}^{(i,j)} = \nabla\lambda_j \mu_{\boldsymbol{\beta}}^{2n-2}(\mathbf{A}) \nabla\lambda_i, \quad \boldsymbol{\beta} \in \mathcal{I}_3^{2n-2}$;

4 `//Same lines as 3DMassMatConst`$(\mathbf{S}, n-1)$ `(cf. Algorithm 3.4), with line 4 replaced with the line:`

5 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1} * n^2 / C_{n-1}^{2n-2}$;

6 `//and line 12 replaced with lines 11-29 of 3DStiffMatConst`$(\mathbf{S}, n)$ `(cf. Algorithm 3.7), where:`

7 $K \leftarrow w_4$ and $s_{i,j} \leftarrow \tilde{\mu}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(i,j)}$, $i, j = 1, \ldots, 4$;

8 **Return S**;

Recall from Theorem 2.2.4 that, for $d = 1, 2, 3$, the computation of the Bernstein-Bézier moments $\{\mu_{\boldsymbol{\beta}}^{2n-2}(\mathbf{A}) : \boldsymbol{\beta} \in \mathcal{I}_d^{2n-2}\}$ requires $\mathcal{O}(n^{d+1})$ operations. Assuming that the B-moments have been computed, we now analyze the complexity of the proposed algorithms for computing the stiffness matrix. For simplicity, we focus on the case $d = 3$, that is, Algorithm 3.22. The other two cases easily follow from a similar analysis. Computing $\tilde{\mu}_{\boldsymbol{\beta}}^{(i,j)}$ for $i, j \in \{1, \ldots, 4\}$ and $\boldsymbol{\beta} \in \mathcal{I}_3^{2n-2}$ involves $3^2 + 3$ multiplications, and is executed $4^2\binom{2n+1}{3}$ times. Assuming that the scaling constant n^2/C_{n-1}^{2n-2} one line 5 is computed once and then re-used, the loop over the pair (α_1, β_1) contains one multiplication, and is executed n^2 times. Similarly, the loop over (α_2, β_2) contains one multiplication, and is executed $\binom{n+1}{2}^2$ times. Finally, the innermost loop of the algorithm contains $(3+1)^2 + 2$ multiplications, and is executed $\binom{n+2}{3}^2$ times. Thus, in addition to the cost associated with the B-moment computations, the number of operations required for computing the 3D stiffness matrix is

$$12 \times 16\binom{2n+1}{3} + n^2 + \binom{n+1}{2}^2 + 18\binom{n+2}{3}^2 + 1.$$

Similarly, we find that, assuming that the B-moments are given, the cost associated with the computation of the 1D and 2D stiffness matrices is respectively given by

$$2 \times 4(2n-1) + 6n^2 + 1, \text{ and } 6 \times 9\binom{2n}{2} + n^2 + 11\binom{n+1}{2}^2 + 1.$$

To summarize, the number of operations involved in computing the stiffness matrix is

$$\begin{cases} 6n^2 + 16n - 7 & \text{for } d = 1, \\ \frac{11n^4}{4} + \frac{33n^3}{2} + \frac{579n^2}{4} - 21n + 12 & \text{for } d = 2, \\ \frac{n^6}{2} + 3n^5 + \frac{27n^4}{4} + \frac{525n^3}{2} + \frac{13n^2}{4} - 64n + 1 & \text{for } d = 3, \end{cases}$$

plus the cost associated with the computation of the B-moments which can be precomputed with $\mathcal{O}(n^{d+1})$ operations.

Let us now consider the output of the proposed algorithms. Note from the expression of the scaling constant r that the stiffness matrix is computed as

$$\mathbf{S}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \frac{n^2}{\binom{2n-2}{n-1}} \sum_{i,j=1}^{d+1} \binom{\boldsymbol{\alpha}-\mathbf{e}_i+\boldsymbol{\beta}-\mathbf{e}_j}{\boldsymbol{\alpha}-\mathbf{e}_i} \mu^{2n-2}_{\boldsymbol{\alpha}-\mathbf{e}_i+\boldsymbol{\beta}-\mathbf{e}_j}(\nabla\lambda_j \mathbf{A} \nabla\lambda_i), \tag{3.31}$$

for $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}_d^n$. But then, observe from the definition of $\mathbf{S}$ and (3.20) that

$$\begin{aligned}\mathbf{S}_{\boldsymbol{\alpha},\boldsymbol{\beta}} =& n^2 \int_T \Big(\sum_{j=1}^{d+1} B^{n-1}_{\boldsymbol{\beta}-\mathbf{e}_j}(\mathbf{x}) \nabla\lambda_j \Big) \mathbf{A}(\mathbf{x}) \Big(\sum_{i=1}^{d+1} B^{n-1}_{\boldsymbol{\alpha}-\mathbf{e}_i}(\mathbf{x}) \nabla\lambda_i \Big) \mathrm{d}\mathbf{x} \\ =& n^2 \sum_{i,j=1}^{d+1} \int_T (\nabla\lambda_j \mathbf{A}(\mathbf{x}) \nabla\lambda_i) B^{n-1}_{\boldsymbol{\alpha}-\mathbf{e}_i}(\mathbf{x}) B^{n-1}_{\boldsymbol{\beta}-\mathbf{e}_j}(\mathbf{x}) \mathrm{d}\mathbf{x},\end{aligned}$$

which, together with the equality

$$B^n_{\boldsymbol{\alpha}} B^m_{\boldsymbol{\beta}} = \frac{\binom{\boldsymbol{\alpha}+\boldsymbol{\beta}}{\boldsymbol{\alpha}}}{\binom{n+m}{n}} B^{n+m}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}, \quad \boldsymbol{\alpha} \in \mathcal{I}_d^n,\ \boldsymbol{\beta} \in \mathcal{I}_d^m,\ n, m \in \mathbb{N}_0, \tag{3.32}$$

yields (3.31).

The next result thus holds.

Theorem 3.3.3. *Let $n \in \mathbb{N}$ and $d \in \{1, 2, 3\}$. Using the conical product rule with $q = n + 1$, the element stiffness matrix of degree n can be computed with $\mathcal{O}(n^{2d})$ operations using Algorithms* `1DStiffMat`, `2DStiffMat`, `3DStiffMat`.

3.3.4 Convective Matrix

We now want to approximate the convective matrix $\mathbf{V}$ defined by

$$\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}} := \int_T \nabla B^n_{\boldsymbol{\alpha}}(\mathbf{x}) \cdot \mathbf{b}(\mathbf{x}) B^n_{\boldsymbol{\beta}}(\mathbf{x}) \mathrm{d}\mathbf{x}, \quad \boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}_d^n. \tag{3.33}$$

The convective matrix can be computed using Algorithms 3.23, 3.24, 3.25, for $d = 1, 2, 3$, respectively.

Algorithm 3.23: 1DConvMat($\mathbf{V}, n, \{\mu_{\boldsymbol{\beta}}^{2n-1}(\mathbf{b}) : \boldsymbol{\beta} \in \mathcal{I}_1^{2n-1}\}$)

Input : Bernstein-Bézier moments $\{\mu_{\boldsymbol{\beta}}^{2n-1}(\mathbf{b}) : \boldsymbol{\beta} \in \mathcal{I}_1^{2n-1}\}$ computed by means of the Stroud conical product rule with $q = n+1$, and precomputed binomial coefficients $\{C_i^{i+j} : 0 \le i \le n,\ 0 \le j \le n-1\}$.

Output: 1D element convective matrix $\mathbf{V}$.

1 **for** $i = 1$ **to** 2 **do**

2 ⌊ Compute $\tilde{\mu}_{\boldsymbol{\beta}}^{(i)} = \nabla\lambda_i \cdot \mu_{\boldsymbol{\beta}}^{2n-1}(\mathbf{b}), \quad \boldsymbol{\beta} \in \mathcal{I}_1^{2n-1}$;

3 $\mathbf{V} \equiv \mathbf{0}$;

4 `//Lines 5-11 of 1DConvMatConstDirect`$(\mathbf{V}, n)$
`(cf. Algorithm 3.11), with line 7 replaced with the line:`

5 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1} * n / C_n^{2n-1}$;

6 `//and with:`

7 $v_i \leftarrow \tilde{\mu}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(i)},\ i = 1, 2$;

8 **Return** $\mathbf{V}$;

Algorithm 3.24: 2DConvMat($\mathbf{V}, n, \{\mu_{\boldsymbol{\beta}}^{2n-1}(\mathbf{b}) : \boldsymbol{\beta} \in \mathcal{I}_2^{2n-1}\}$)

Input : Bernstein-Bézier moments $\{\mu_{\boldsymbol{\beta}}^{2n-1}(\mathbf{b}) : \boldsymbol{\beta} \in \mathcal{I}_2^{2n-1}\}$ computed by means of the Stroud conical product rule with $q = n+1$, and precomputed binomial coefficients $\{C_i^{i+j} : 0 \le i \le n,\ 0 \le j \le n-1\}$.

Output: 2D element convective matrix $\mathbf{V}$.

1 **for** $i = 1$ **to** 3 **do**

2 ⌊ Compute $\tilde{\mu}_{\boldsymbol{\beta}}^{(i)} = \nabla\lambda_i \cdot \mu_{\boldsymbol{\beta}}^{2n-1}(\mathbf{b}), \quad \boldsymbol{\beta} \in \mathcal{I}_2^{2n-1}$;

3 $\mathbf{V} \equiv \mathbf{0}$;

4 `//Lines 5-15 of 2DConvMatConstDirect`$(\mathbf{V}, n)$
`(cf. Algorithm 3.12), with line 8 replaced with the line:`

5 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1} * n / C_n^{2n-1}$;

6 `//and with:`

7 $v_i \leftarrow \tilde{\mu}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(i)},\ i = 1, 2, 3$;

8 **Return** $\mathbf{V}$;

Algorithm 3.25: 3DConvMat($\mathbf{V}, n, \{\mu_{\boldsymbol{\beta}}^{2n-1}(\mathbf{b}) : \boldsymbol{\beta} \in \mathcal{I}_3^{2n-1}\}$)

Input : Bernstein-Bézier moments $\{\mu_{\boldsymbol{\beta}}^{2n-1}(\mathbf{b}) : \boldsymbol{\beta} \in \mathcal{I}_3^{2n-1}\}$ computed by means of the Stroud conical product rule with $q = n+1$, and precomputed binomial coefficients $\{C_i^{i+j} : 0 \le i \le n,\ 0 \le j \le n-1\}$.

Output: 3D element convective matrix $\mathbf{V}$.

1 **for** $i = 1$ **to** 4 **do**

2 Compute $\tilde{\mu}_{\boldsymbol{\beta}}^{(i)} = \nabla\lambda_i \cdot \mu_{\boldsymbol{\beta}}^{2n-1}(\mathbf{b}), \quad \boldsymbol{\beta} \in \mathcal{I}_3^{2n-1}$;

3 $\mathbf{V} \equiv \mathbf{0}$;

4 `//Lines 5-19 of 3DConvMatConstDirect`$(\mathbf{V}, n)$
`(cf. Algorithm 3.13), with line 9 replaced with the line:`

5 $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1} * n / C_n^{2n-1}$;

6 `//and with:`

7 $v_i \leftarrow \tilde{\mu}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(i)},\ i = 1, \ldots, 4$;

8 **Return** $\mathbf{V}$;

We now proceed to analyze the complexity associated with the convective matrix algorithms. For simplicity, we focus on the case $d = 3$, that is, Algorithm 3.25. The other two cases are similar. Recall from Theorem 2.2.4 that the B-moments of order $2n-1$ can be computed in $\mathcal{O}(n^4)$ operations. Assuming that the required B-moments have been computed, we study the complexity associated with the proposed algorithm. To this end, observe that the loop over i is executed four times, and contains one inner product which amounts to three multiplications. Thus, setting up the field $\tilde{\mu}_{\boldsymbol{\beta}}^{(i)}$, $1 \le i \le 4$, and $\boldsymbol{\beta} \in \mathcal{I}_3^{2n-1}$ involves twelve operations. In addition, the loop over β_1 is executed $\binom{n+2}{3}(n+1)$ times, and contains one multiplication, whereas the loop over β_2 is executed $\binom{n+2}{3}\binom{n+2}{2}$ times, and contains one multiplication. Finally, the innnermost loop is executed $\binom{n+2}{3}\binom{n+3}{3}$ times, and contains six operations. Therefore, in addition to the cost of computing the B-moments, the number of operations involved in the computation of the 3D stiffness matrix is

$$12 + \binom{n+2}{3}(n+1) + \binom{n+2}{3}\binom{n+2}{2} + 6\binom{n+2}{3}\binom{n+3}{3}.$$

Using a similar argument, we find that the cost associated with Algorithm 3.23

and Algorithm 3.24 is respectively given by

$$3+4n(n+1),\ \text{and}\ 6+\binom{n+1}{2}(n+1)+5\binom{n+1}{2}\binom{n+2}{2},$$

plus the cost involved in the B-moments computation. To summarize, in addition to the cost associated with the computation of the B-moments, the number of operations involved in computing the convective matrix is

$$\begin{cases} 4n^2+4n+2 & \text{for } d=1,\\ \frac{5n^4}{4}+\frac{11n^3}{2}+\frac{29n^2}{4}+3n+6 & \text{for } d=2,\\ \frac{n^6}{6}+\frac{19n^5}{12}+\frac{35n^4}{6}+\frac{41n^3}{4}+\frac{17n^2}{2}+\frac{8n}{3}+12 & \text{for } d=3. \end{cases}$$

Checking the output of Algorithms 3.23, 3.24 and 3.25, note that, for $d=1,2,3$, the convective matrix is computed as:

$$\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}}=\frac{n}{\binom{2n-1}{n}}\sum_{i=1}^{d+1}\binom{\boldsymbol{\alpha}-\mathbf{e}_i+\boldsymbol{\beta}}{\boldsymbol{\beta}}\int_T B^{2n-1}_{\boldsymbol{\alpha}-\mathbf{e}_i+\boldsymbol{\beta}}(\nabla\lambda_i\cdot\mathbf{b}),\quad \boldsymbol{\alpha},\boldsymbol{\beta}\in\mathcal{I}^n_d. \tag{3.34}$$

But then, observe from the definition of $\mathbf{V}$ and equation (3.20) that, for any $\boldsymbol{\alpha},\boldsymbol{\beta}\in\mathcal{I}^n_d$,

$$\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}}=n\int_T\sum_{i=1}^{d+1}B^{n-1}_{\boldsymbol{\alpha}-\mathbf{e}_i}\nabla\lambda_i\cdot\mathbf{b}B^n_{\boldsymbol{\beta}}=n\sum_{i=1}^{d+1}\int_T B^{n-1}_{\boldsymbol{\alpha}-\mathbf{e}_i}B^n_{\boldsymbol{\beta}}\nabla\lambda_i\cdot\mathbf{b},$$

which, by virtue of (3.32), coincides with (3.34).

The following theorem is a direct consequence of the above argument.

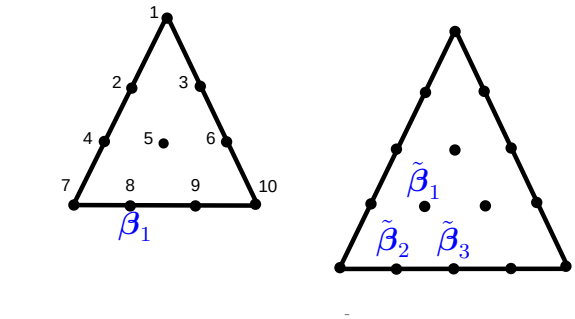
Theorem 3.3.4. *Let $n\in\mathbb{N}$ and $d\in\{1,2,3\}$. Using the conical product rule with $q=n+1$, the convective matrix of order n can be computed with $\mathcal{O}(n^{2d})$ operations using Algorithms* `1DConvMat`, `2DConvMat`, `3DConvMat`, *for $d=1,2,3$, respectively.*

3.4 Stencils

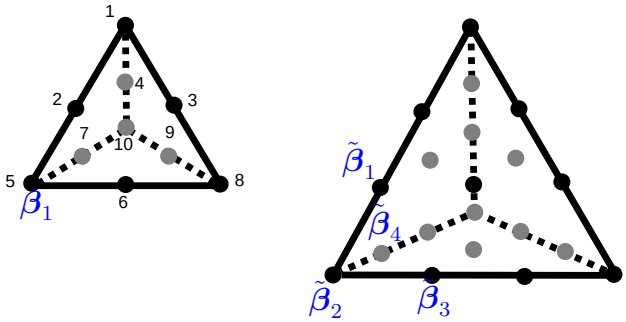
Observe on lines 16-19 of Algorithm 3.13 that, for $\boldsymbol{\beta} \in \mathcal{I}_3^{n-1} \simeq \mathcal{D}_3^{n-1}$, the computation of the 3D convective matrix involves updating the columns associated with $\boldsymbol{\beta}+\mathbf{e}_k$ with $k = 1, \ldots, 4$. Lines 14-29 of Algorithm 3.7 yield a similar statement for the stiffness matrix. One can easily check that analogous statements also hold for $d = 1, 2$. For a given domain point $\boldsymbol{\beta} \in \mathcal{I}_d^\ell$, the set $\{\tilde{\boldsymbol{\beta}}_k := \boldsymbol{\beta}+\mathbf{e}_k : \ k = 1, \ldots, d+1\}$ is termed the *stencil* associated with $\boldsymbol{\beta}$. Instead of checking the position of the entry corresponding to $\boldsymbol{\beta}+\mathbf{e}_k$, an efficient memory access call which considerably speeds up the computations, consists in using the structure of the domain point stencils. More precisely, it is possible to use the geometric architecture of the domain points in order to *systematically* determine the lexicographical position of $\boldsymbol{\beta} + \mathbf{e}_k$. The key observation is that the lexicographical position of $\boldsymbol{\beta}$ and $\tilde{\boldsymbol{\beta}}_1 := \boldsymbol{\beta} + \mathbf{e}_1$ coincide, and that the difference between domain points of order ℓ and of order $\ell + 1$ are given by the additional layer of domain points of lower dimension formed by the set $\mathcal{D}_{d-1}^{\ell+1}$ (see Figure 3.1. For the 3D case, the non-visible domain points are drawn in grey). Thus, in one, two and three dimensions, the additional layer respectively consists of one additional point, line and triangle. For $\boldsymbol{\beta} \in \mathcal{D}_d^\ell$ and $k = 2, , \ldots, d + 1$, the domain point $\tilde{\boldsymbol{\beta}}_k := \boldsymbol{\beta} + \mathbf{e}_k \in \mathcal{D}_d^{\ell+1}$ lies on the additional domain point layer. The offset between $\tilde{\boldsymbol{\beta}}_1$ and $\tilde{\boldsymbol{\beta}}_2$ is deduced from the above remarks and careful observations of the geometric structure of the domain points, leading to Algorithms `Stencil1D`, `Stencil2D` and `Stencil3D` for the efficient position evaluation of the elements of the stencil sets associated with each $\boldsymbol{\beta} \in \mathcal{I}_d^n$, assuming that the counter starts at 1. In the above-mentioned algorithms, the notation $\mathrm{lex}(\cdot)$ is used to represent the lexicographical position operator.

Algorithm 3.26: Stencil1D(n)

```
1 Initialize lex(α̃_1) = 1, lex(α̃_2) = 2;
2 for α_1 = n to 0 do
3 |   lex(α̃_1) += 1;
4 |   lex(α̃_2) += 1;
```



(a) Difference between $\mathcal{D}_2^3$ and $\mathcal{D}_2^4$



(b) Difference between $\mathcal{D}_3^2$ and $\mathcal{D}_3^3$

Figure 3.1: Domain point stencils

Algorithm 3.27: Stencil2D(n)

```
Initialize lex(α̃_1) = 1, lex(α̃_2) = 2, lex(α̃_3) = 3;
for α_1 = n to 0 do
    for α_2 = n − α_1 to 0 do
        lex(α̃_2) += 1;
        lex(α̃_3) += 1;
        lex(α̃_1) += 1;
    lex(α̃_2) += 1;
    lex(α̃_3) += 1;
```

Algorithm 3.28: Stencil3D(n)

```
1  Initialize lex(α̃_1) = 1, lex(α̃_2) = 2, lex(α̃_3) = 3, lex(α̃_4) = 4;
2  for α_1 = n to 0 do
3      for α_2 = n − α_1 to 0 do
4          for α_3 = n − α_1 − α_2 to 0 do
5              lex(α̃_1) += 1;
6              lex(α̃_2) += 1;
7              lex(α̃_3) += 1;
8              lex(α̃_4) += 1;
9          lex(α̃_3) += 1;
10         lex(α̃_4) += 1;
11     lex(α̃_2) += n − α_1 + 2;
12     lex(α̃_3) += 1;
13     lex(α̃_4) += 1;
```

In particular, for the efficient evaluation of the convective matrix, `Stencil1D`, `Stencil2D`, `Stencil3D`, as presented in Algorithms 3.26, 3.27 and 3.28, are incorporated into the loop over α_i, $i = 1, \ldots, d$, for $d = 1, 2, 3$, respectively. A similar insertion is performed for the computation of the stiffness matrix, except that the stencil structure is present in both the α_i- and β_i-loops.

3.5 Summary

We now proceed to summarize the main results of the previous sections, as following directly from (3.11), equation (3.13), equation (3.15), equation (3.24), Theorem 3.3.1, Theorem 3.3.2, Theorem 3.3.3 and Theorem 3.3.4.

Theorem 3.5.1. *In the case of piecewise constant data:*

- *the load vector entries of order n are all given by*

$$\frac{f|_T |T|}{\binom{n+d}{d}};$$

- *the mass matrix of order n is given by*

$$\mathbf{M}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \frac{c_{|T}|T|}{\binom{2n}{n}\binom{2n+d}{d}}\binom{\boldsymbol{\alpha}+\boldsymbol{\beta}}{\boldsymbol{\alpha}}, \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}_d^n,$$

  and can be computed with a $\mathcal{O}(n^{2d})$ complexity using Algorithm 3.2, 3.3 and 3.4 for $d = 1, 2,$ and 3, respectively;

- *the stiffness matrix of order n is given by*

$$\mathbf{S}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = n^2 \sum_{i,j=1}^{d+1} E_{k,\ell}(\mathbf{M}) \nabla\lambda_j \mathbf{A}_{|T} \nabla\lambda_i,$$

  and can be computed with a $\mathcal{O}(n^{2d})$ complexity using Algorithms 3.8, 3.9 and 3.10 for $d = 1, 2$ and 3, respectively ;

- *the convective matrix of order n is given by*

$$\mathbf{V}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \frac{n|T|}{\binom{2n-1}{n}\binom{2n-1+d}{d}} \mathbf{b}_{|T} \cdot \sum_{\ell=1}^{d+1} \nabla\lambda_\ell \binom{\boldsymbol{\alpha}+\boldsymbol{\beta}-\mathbf{e}_\ell}{\boldsymbol{\alpha}}, \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}_d^n,$$

  and can be computed with a $\mathcal{O}(n^{2d})$ complexity using Algorithms 3.11, 3.12, and 3.13 for $d = 1, 2$ and 3, respectively.

One should note that the above formulae are all explicit.

In the case of variable data, the integrals are approximated using the Stroud-Conical product rule with q^d quadrature points, with $q = n + 1$. It holds that:

- *the load vector of order n can be computed with a $\mathcal{O}(n^{d+1})$ complexity by means of Algorithms 2.2, 2.3 and 2.4 for $d = 1, 2$ and 3, respectively;*

- *the mass matrix of order n can be computed with a $\mathcal{O}(n^{2d})$ complexity by means of Algorithms 3.17, 3.18 and 3.19, for $d = 1, 2$ and 3, respectively;*

- *the stiffness matrix of order n can be computed with a $\mathcal{O}(n^{2d})$ complexity by means of Algorithms 3.20, 3.21, and 3.22, for $d = 1, 2, 3$, respectively;*

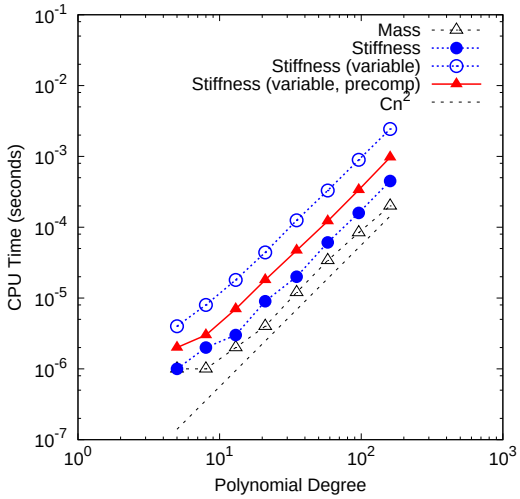
- *the convective matrix of order n can be computed with a $\mathcal{O}(n^{2d})$ complexity by means of Algorithms 3.23, 3.24 and 3.25, for $d = 1, 2, 3$, respectively.*

3.6 Numerical Examples

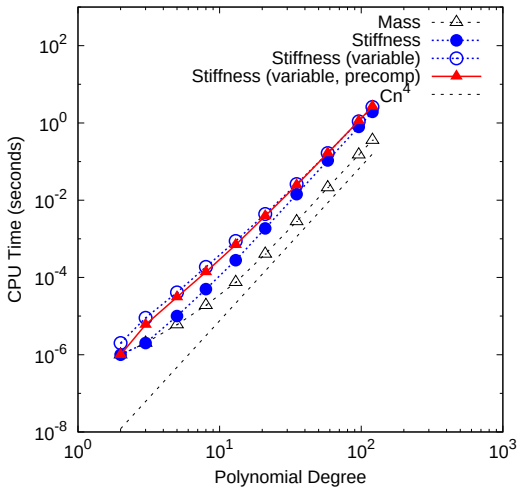
3.6.1 CPU Timings

For $d = 1, 2, 3$, we now proceed to plot the CPU time involved in the computation of the elemental mass and stiffness matrices against the value of n, using both the algorithms presented here and those introduced in [11]. We obtain Figure 3.2: "Mass" and "Stiffness" respectively refer to the CPU timings for computing the mass and stiffness matrices with constant coefficients. The CPU timings for computing the stiffness matrix with variable coefficients is represented by the plots corresponding to "Stiffness (variable)" and "Stiffness (variable, precomp)". That is, for the variable case, two approaches are represented: the one presented in this work (PRECOMP), and that developed in [11] (see Section 2.3 for more details).

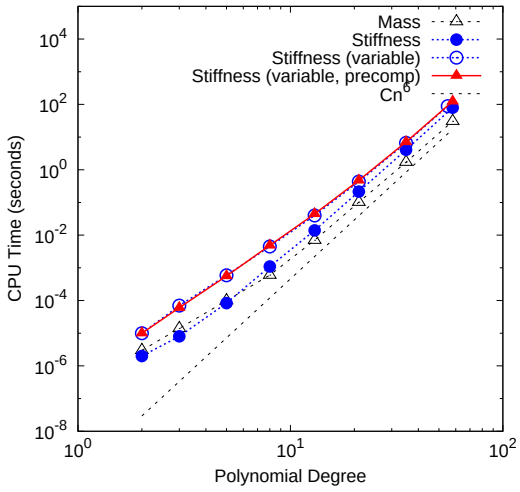
On each graph, the growth of the computational cost is compared to the curve of the function Cn^{2d}, where C is a scalar constant. We observe that, in all cases, the results confirm the predicted optimal cost $\mathcal{O}(n^{2d})$. In addition, observe that, with $d = 2, 3$, the CPU timings corresponding to the constant and variable cases are virtually the same for higher degrees, which is consistent with the fact that the additional cost occuring when the data is variable only comes from the B-moments computation which is done in only $\mathcal{O}(n^{d+1})$ operations. In contrast, with $d = 1$, the computations of the B-moments and those of the stiffness matrix are of the same order, which causes a significant difference between the constant and variable data, as observed Figure 3.2(a) for the stiffness matrix. In other words, with the proposed approach, the complexity is not dominated by the quadrature cost with $d > 1$.



(a) $d = 1$



(b) $d = 2$



(c) $d = 3$

Figure 3.2: CPU timings for the computation of the mass and stiffness matrices

3.6.2 Test Problem

We next use our basis to solve the problem

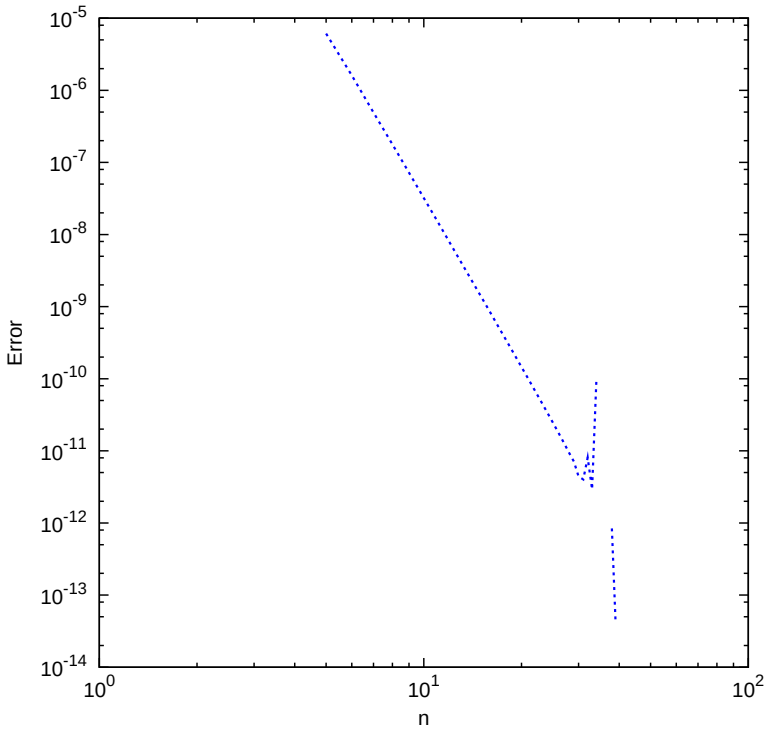
$$\begin{cases} -\triangle u &= 1 \text{ on } \Omega, \\ u &= 0 \text{ on } \partial\Omega, \end{cases} \tag{3.35}$$

on the single triangle T defined by

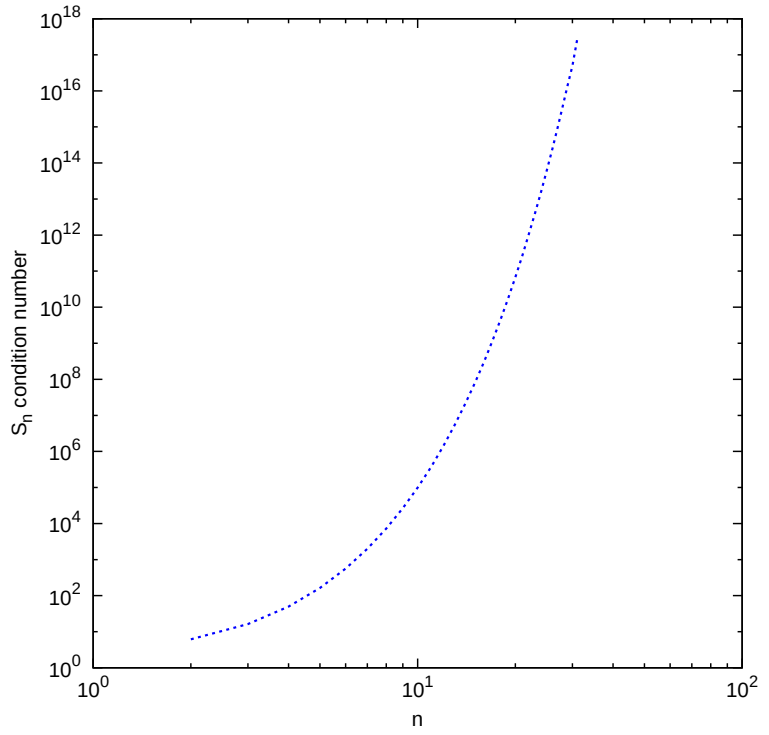
$$T := \{(x, y) \in \mathbb{R}^2 : \ 0 \leq x, y \leq 1, \ 0 \leq y \leq 1 - x\}. \tag{3.36}$$

The corresponding error in the energy norm is given in Figure 3.3(a).

Observe in particular that, for $n > 30$, the error starts oscillating, which might be due to the growth of the stiffness matrix condition number, as shown in Figure 3.3(b).



(a) Error obtained for (3.35)



(b) Growth of the 2D stiffness matrix condition number

Figure 3.3: Error and condition number for the 2D case

Chapter 4

Bernstein-Bézier Finite Elements for $H(\mathrm{curl})$ in 2D

The goal of this chapter is to generalize the results obtained in Chapter 3 to the two-dimensional space $H(\mathrm{curl})$ defined by (1.10). In two dimensions, $H(\mathrm{curl})$ is isomorphic to the $H(\mathrm{div})$ space defined by $H(\mathrm{div};\Omega) := \{\mathbf{v} \in L^2(\Omega) : \ \mathrm{div}(\mathbf{v}) \in L^2(\Omega)\}$, with $\mathrm{div}(\mathbf{v}) := \nabla \cdot \mathbf{v}$. Hence the work presented in this chapter was used as a foundation for an article on Bernstein-Bézier $H(\mathrm{div})$ finite elements in two dimensions [10].

Finite element spaces in $H(\mathrm{curl})$ are used for solving curl-curl problems of the form:

$$\begin{aligned} \mathbf{curl}\big(A\,\mathrm{curl}\,\mathbf{u}\big) + \boldsymbol{\kappa}\mathbf{u} \quad &= \mathbf{f}, \\ \mathbf{u}\cdot\boldsymbol{\tau} \quad &= 0 \text{ on } \Gamma_D, \\ A\,\mathrm{curl}\,\mathbf{u} \quad &= 0 \text{ on } \Gamma_N, \end{aligned} \tag{4.1}$$

where

$$\begin{aligned} \Gamma_D \bigcup \Gamma_N \quad &= \partial\Omega, \\ \Gamma_D \bigcap \Gamma_N \quad &= \emptyset. \end{aligned}$$

It is well-established that H^1 finite elements applied to the Galerkin discretization

of the above problem may lead to spurious solutions [74, 66].

Multiplying the first equation of (4.1) by smooth functions, and integrating by parts, the weak formulation corresponding to (4.1) reads:

$$\text{Find } \mathbf{u} \in H_0(\text{curl};\Omega) \text{ such that, for all } \mathbf{v} \in H_0(\text{curl};\Omega),$$
$$\int_\Omega A(\mathbf{x})\,\text{curl}\,\mathbf{u}(\mathbf{x})\,\text{curl}\,\mathbf{v}(\mathbf{x})\,\mathrm{d}\mathbf{x} + \int_\Omega \boldsymbol{\kappa}(\mathbf{x})\mathbf{u}(\mathbf{x})\cdot\mathbf{v}(\mathbf{x})\,\mathrm{d}\mathbf{x} = \int_\Omega \mathbf{f}(\mathbf{x})\cdot\mathbf{v}(\mathbf{x})\,\mathrm{d}\mathbf{x}, \tag{4.2}$$

where $H_0(\text{curl};\Omega)$ consists of the functions in $H(\text{curl};\Omega)$ with zero tangential component on the boundary Γ_D. Moreover, the scalar-valued function A is assumed to be continuous and strictly positive on Ω, $\boldsymbol{\kappa}$ is bounded and uniformly positive definite, and $\mathbf{f}$ is square-integrable. Under these assumptions, the bilinear form $a(\cdot,\cdot)$ and the linear form F which are defined by

$$a(\mathbf{u},\mathbf{v}) := \int_\Omega \big[A\,\text{curl}\,\mathbf{u}(\mathbf{x})\,\text{curl}\,\mathbf{v}(\mathbf{x})\mathrm{d}\mathbf{x} + \boldsymbol{\kappa}(\mathbf{x})\mathbf{u}(\mathbf{x})\cdot\mathbf{v}(\mathbf{x})\big]\mathrm{d}\mathbf{x}, \quad \mathbf{u},\mathbf{v} \in H_0(\text{curl};\Omega),$$
$$F(\mathbf{v}) := \int_\Omega \mathbf{f}(\mathbf{x})\cdot\mathbf{v}(\mathbf{x})\mathrm{d}\mathbf{x}, \quad \mathbf{v} \in H_0(\text{curl};\Omega),$$

are continuous. In addition, $a(\cdot,\cdot)$ is coercive, with coercivity constant given by

$$\vartheta = \min\Big\{\inf_{\mathbf{x}\in\Omega} A(x), \inf_{x\in\Omega} \frac{\mathbf{x}^t\boldsymbol{\kappa}(\mathbf{x})\mathbf{x}}{\|\mathbf{x}\|^2}\Big\}.$$

Thus, by virtue of the Lax-Milgram Lemma, (4.2) is well-posed. Without loss of generality, we can assume that $\boldsymbol{\kappa}$ is symmetric. Assuming that $\Omega \subset \mathbb{R}^2$ is a domain with polygonal boundary, and that $\{\boldsymbol{\phi}_{T,i}\}_{i=1}^m$ is a conforming set of shape functions on each triangle T of a triangulation of Ω, the Galerkin discretization of the problem consists in approximating any $\mathbf{v} \in H_\mathbf{0}(\text{curl};\Omega)$ and the solution $\mathbf{u}$ using the substitutions

$$\mathbf{v} \approx \mathbf{v}_{FE} = \sum_{j=1}^m \ell_j \boldsymbol{\phi}_{T,j}, \;\; \mathbf{u} \approx \mathbf{u}_{FE} = \sum_{i=1}^m k_i \boldsymbol{\phi}_{T,i},$$

on every triangle T of the triangulation. Inserting the above approximations into

(4.2) yields a linear system involving the $H(\mathrm{curl})$ load vector, mass and stiffness matrices whose components are given by

$$\int_T \mathbf{f} \cdot \boldsymbol{\phi}_{T,i}\,\mathrm{d}\mathbf{x}, \quad \int_T \boldsymbol{\kappa}\boldsymbol{\phi}_{T,i} \cdot \boldsymbol{\phi}_{T,j}\,\mathrm{d}\mathbf{x}, \quad \int_T \mathrm{curl}(\boldsymbol{\phi}_{T,i}) A\,\mathrm{curl}(\boldsymbol{\phi}_{T,j})\,\mathrm{d}\mathbf{x}, \tag{4.3}$$

respectively.

Recall that, in two dimensions, the Raviart-Thomas elements are obtained by "rotating" the Nédélec elements (see [26, Section 5]). Hence, using the notations in the above equations, the set $\{\boldsymbol{\phi}_{T,i}^{\perp}\}_{i=1}^{m}$ forms an $H(\mathrm{div})$-conforming set of shape functions on the triangle T, and the corresponding $H(\mathrm{div})$ load vector, mass and stiffness matrices are given by

$$\int_T \mathbf{f}^{\mathrm{div}} \cdot \boldsymbol{\phi}_{T,i}^{\perp}\,\mathrm{d}\mathbf{x}, \quad \int_T \boldsymbol{\kappa}^{\mathrm{div}}\boldsymbol{\phi}_{T,i}^{\perp} \cdot \boldsymbol{\phi}_{T,j}^{\perp}\,\mathrm{d}\mathbf{x}, \quad \int_T \mathrm{div}(\boldsymbol{\phi}_{T,i}^{\perp}) A^{\mathrm{div}}\,\mathrm{div}(\boldsymbol{\phi}_{T,j}^{\perp})\,\mathrm{d}\mathbf{x}, \tag{4.4}$$

where $\mathbf{f}^{\mathrm{div}}$, $\boldsymbol{\kappa}^{\mathrm{div}}$ and A^{div} respectively denote the coefficients associated with the load vector, the mass matrix and the stiffness matrix. Using a simple algebraic argument, it can be shown that

$$\left.\begin{array}{rcl} \int_T \mathbf{f}^{\mathrm{div}} \cdot \boldsymbol{\phi}_{T,i}^{\perp}\mathrm{d}\mathbf{x} & = & \int_T (-\mathbf{f}^{\mathrm{div}})^{\perp} \cdot \boldsymbol{\phi}_{T,i}\,\mathrm{d}\mathbf{x}, \\ \int_T \boldsymbol{\kappa}^{\mathrm{div}}\boldsymbol{\phi}_{T,i}^{\perp} \cdot \boldsymbol{\phi}_{T,j}^{\perp}\,\mathrm{d}\mathbf{x} & = & \int_T \tilde{\boldsymbol{\kappa}}^{\mathrm{div}}\boldsymbol{\phi}_{T,i} \cdot \boldsymbol{\phi}_{T,j}\,\mathrm{d}\mathbf{x}, \\ \int_T \mathrm{div}(\boldsymbol{\phi}_{T,i}^{\perp}) A^{\mathrm{div}}\,\mathrm{div}(\boldsymbol{\phi}_{T,j}^{\perp})\,\mathrm{d}\mathbf{x} & = & \int_T \mathrm{curl}(\boldsymbol{\phi}_{T,i}) A^{\mathrm{div}}\,\mathrm{curl}(\boldsymbol{\phi}_{T,j})\,\mathrm{d}\mathbf{x}, \end{array}\right\} \tag{4.5}$$

where $\tilde{\boldsymbol{\kappa}}^{\mathrm{div}}$ is defined as

$$\tilde{\boldsymbol{\kappa}}^{\mathrm{div}} := \begin{pmatrix} a_{22} & -a_{21} \\ -a_{12} & a_{11} \end{pmatrix} \text{ with } \boldsymbol{\kappa}^{\mathrm{div}} := (a_{ij})_{i,j=1,2}.$$

Hence, in order to compute the $H(\mathrm{div})$ elemental load vector associated with the function $\mathbf{f}^{\mathrm{div}}$, it suffices to compute the $H(\mathrm{curl})$ load vector associated with $(-\mathbf{f}^{\mathrm{div}})^{\perp}$. Similarly, the $H(\mathrm{div})$ elemental mass matrix associated with $\boldsymbol{\kappa}^{\mathrm{div}}$ can be obtained by computing the $H(\mathrm{curl})$ elemental mass matrix associated with $\tilde{\boldsymbol{\kappa}}^{\mathrm{div}}$ defined above. Using the fact that $\mathrm{div}(\boldsymbol{\phi}^{\perp}) = -\,\mathrm{curl}(\boldsymbol{\phi})$, it can be easily shown that the $H(\mathrm{div})$ elemental stiffness matrix associated with the function A

is equal to the $H(\text{curl})$ stiffness matrix associated with the same function. As a consequence, provided that the data is pre-processed as on the right-hand side of (4.5), the $H(\text{curl})$ routines presented in Appendix B can also be used for the computation of the $H(\text{div})$ elemental quantities.

The chapter is organized as follows. Section 4.1 describes the $H(\text{curl})$ Bernstein-Bézier (BB) finite element shape functions which are based on BB polynomials. The key idea behind the optimal complexity results is to write the $H(\text{curl})$ quantities in terms of linear combinations of B-moments, as discussed in Section 4.2, and then use the estimations presented in Chapter 2 for the computation of the B-moments. Section 4.3 then gives the details of the corresponding algorithms and proves that they achieve the optimal complexity $\mathcal{O}(n^4)$. Section 4.5 concludes with CPU timings which are consistent with the predicted optimal complexity results, and gives an illustrative example of the use of the presented finite element for solving a Maxwell's eigenvalue problem.

4.1 Bernstein-Bézier $H(\text{curl})$ Finite Element

Since only the case $d = 2$ is considered in this chapter, we use simplified versions of the notations used in Chapter 2. Thus, for $n \in \mathbb{Z}_0^+$, the symbol $\mathcal{I}^n$ is used instead of $\mathcal{I}_2^n$. In particular, note that $\mathcal{I}^0 = \{(0,0,0)\}$ and $\mathcal{I}^1 = \{\mathbf{e}_k : \ k = 1,2,3\}$, where $\mathbf{e}_k$ is such that the k^{th} index is one and the other entries vanish. Similarly, we use the simplified notation $\mathcal{D}^n(T) = \mathcal{D}_2^n(T)$ for the set of two-dimensional domain points with respect to the simplex T. We also denote by $\mathbb{P}_k$ the space of bivariate polynomials of total degree at most k, and $(\mathbb{P}_k)^2$ is the space of vector functions whose both components are polynomials in $\mathbb{P}_k$.

Given a triangle $T = \langle \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3 \rangle$, where the vertices are ordered counterclockwise, the notations γ_i (respectively $\boldsymbol{\tau}_i$) refer to the edge opposite to the vertex $\mathbf{v}_i$ (respectively the unit tangent vector on γ_i in the counterclockwise direction around the triangle).

The *star* of an index $\boldsymbol{\alpha} \in \mathcal{I}^n$, as represented on Figure 4.1 below, is the set

$$\begin{aligned}\texttt{Star}(\boldsymbol{\alpha}) &:= \{\boldsymbol{\alpha} + \mathbf{e}_k - \mathbf{e}_\ell : k, \ell = 1, 2, 3\} \cap \mathcal{I}^n \\ &= \big\{\boldsymbol{\alpha}, \boldsymbol{\alpha} + (-1, 1, 0), \boldsymbol{\alpha} + (-1, 0, 1), \boldsymbol{\alpha} + (0, -1, 1), \\ &\quad \boldsymbol{\alpha} + (1, -1, 0), \boldsymbol{\alpha} + (1, 0, -1), \boldsymbol{\alpha} + (0, 1, -1)\big\} \cap \mathcal{I}^n. \end{aligned} \tag{4.6}$$

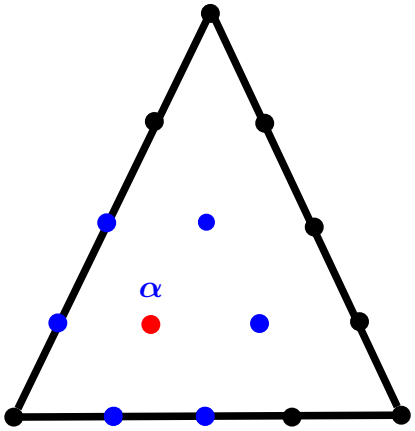


Figure 4.1: Domain Point Star

In addition, let γ_i^n (respectively $\mathring{\mathcal{I}}^n$) denote the set of domain points belonging to the edge γ_i *except the vertices* (respectively the interior domain points).

4.1.1 Bernstein-Bézier Basis for the $H(\mathrm{curl})$ Finite Element

The aim of this section is to provide a basis for the Nédélec space $\mathbb{ND}_n$ defined by (1.14) associated with a given triangle T, such that the element-wise computations of the quantities (4.3) are of optimal order.

We now recall the definition of the *Whitney functions*

$$\boldsymbol{\omega}_i := \lambda_{i+1} \nabla \lambda_{i-1} - \lambda_{i-1} \nabla \lambda_{i+1}, \qquad i = 1, 2, 3, \tag{4.7}$$

where we identify $\lambda_{\nu+3} = \lambda_\nu$. By definition, note that $\boldsymbol{\omega}_i \in \mathbb{ND}_0 = \mathbb{P}_0^2 + \mathrm{span}(\mathbf{x}^\perp)$.

It is well-known [78, 29, 67] that

$$\operatorname{span}(\{\boldsymbol{\omega}_i : \ i = 1, 2, 3\}) = \mathbb{ND}_0. \tag{4.8}$$

In addition, it is easy to check that $(\boldsymbol{\omega}_i \cdot \boldsymbol{\tau}_{i+1})|_{\gamma_i} = (\boldsymbol{\omega}_i \cdot \boldsymbol{\tau}_{i-1})|_{\gamma_i} = 0$.

Lemma 4.1.1. *The barycentric coordinates λ_i and Whitney functions $\boldsymbol{\omega}_i$, for $i = 1, 2, 3$, satisfy*

$$\nabla\lambda_i = -\frac{|\gamma_i|}{2|T|}\mathbf{n}_i, \tag{4.9}$$

$$(\nabla\lambda_i)^\perp \cdot \nabla\lambda_{i+1} = -(\nabla\lambda_i)^\perp \cdot \nabla\lambda_{i-1} = \frac{1}{2|T|}, \tag{4.10}$$

$$\operatorname{curl}(\boldsymbol{\omega}_i) = \frac{1}{|T|}, \tag{4.11}$$

$$(\boldsymbol{\omega}_i \cdot \boldsymbol{\tau}_i)|_{\gamma_i} = \nabla\lambda_{i-1} \cdot \boldsymbol{\tau}_i = -\nabla\lambda_{i+1} \cdot \boldsymbol{\tau}_i = \frac{1}{|\gamma_i|}. \tag{4.12}$$

Proof. The first identity is easy to check by definition. Since the angle between $(\nabla\lambda_i)^\perp$ and $\nabla\lambda_{i+1}$ is always acute, we have $(\nabla\lambda_i)^\perp \cdot \nabla\lambda_{i+1} = \|\nabla\lambda_i \times \nabla\lambda_{i+1}\|$, and (4.10) follows from (4.9) and the identity $|\gamma_i|\mathbf{n}_i \times |\gamma_{i+1}|\mathbf{n}_{i+1} = \pm 2|T|\vec{k}$, where $\vec{k}$ is the unit vector along the z-axis. The formula for $(\nabla\lambda_i)^\perp \cdot \nabla\lambda_{i-1}$ follows from the identity $\mathbf{a} \cdot \mathbf{b}^\perp = -\mathbf{a}^\perp \cdot \mathbf{b}$. To show (4.11), we observe that in two dimensions,

$$\operatorname{curl}(a\mathbf{f}) = -\nabla a \cdot \mathbf{f}^\perp + a \operatorname{curl}(\mathbf{f}), \tag{4.13}$$

where a is a scalar function. Taking into account that $\operatorname{curl}\nabla f = 0$, we obtain

$$\begin{aligned}\operatorname{curl}(\lambda_{i+1}\nabla\lambda_{i-1} - \lambda_{i-1}\nabla\lambda_{i+1}) &= \operatorname{curl}(\lambda_{i+1}\nabla\lambda_{i-1}) - \operatorname{curl}(\lambda_{i-1}\nabla\lambda_{i+1}) \\ &= -\nabla\lambda_{i+1} \cdot (\nabla\lambda_{i-1})^\perp + \nabla\lambda_{i-1} \cdot (\nabla\lambda_{i+1})^\perp,\end{aligned}$$

and (4.11) follows from (4.10), having also used the fact that, $(i+1)+1 = i-1$, when i is taken in the cyclic permutations of the set $(1, 2, 3)$. Finally, by (4.9), $\boldsymbol{\tau}_i = -\frac{2|T|}{|\gamma_i|}(\nabla\lambda_i)^\perp$, and hence (4.10) implies $\nabla\lambda_{i-1} \cdot \boldsymbol{\tau}_i = -\nabla\lambda_{i+1} \cdot \boldsymbol{\tau}_i = \frac{1}{|\gamma_i|}$, so that $\boldsymbol{\omega}_i \cdot \boldsymbol{\tau}_i = \frac{\lambda_{i+1}+\lambda_{i-1}}{|\gamma_i|}$. By restricting the latter to γ_i we complete the proof of (4.12). ∎

For $n \geq 0$ and $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3) \in \mathcal{I}^n$, we define $\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}} \in (\mathbb{P}_{n+1})^2$ by

$$\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}} := (n+1)B^n_{\boldsymbol{\alpha}} \begin{vmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \lambda_1 & \lambda_2 & \lambda_3 \\ \nabla\lambda_1 & \nabla\lambda_2 & \nabla\lambda_3 \end{vmatrix} = (n+1)B^n_{\boldsymbol{\alpha}} \sum_{i=1}^{3} \alpha_i \boldsymbol{\omega}_i. \tag{4.14}$$

In particular, $\boldsymbol{\sigma}^0_{000} = 0$. By (4.7) and the identity $B^n_{\boldsymbol{\alpha}}\lambda_i = \frac{\alpha_i+1}{n+1} B^{n+1}_{\boldsymbol{\alpha}+\mathbf{e}_i}$ it follows that

$$\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}} = \sum_{i=1}^{3} (\alpha_i + 1)(\alpha_{i-1}\nabla\lambda_{i+1} - \alpha_{i+1}\nabla\lambda_{i-1}) B^{n+1}_{\boldsymbol{\alpha}+\mathbf{e}_i}. \tag{4.15}$$

Since $\alpha_i B^n_{\boldsymbol{\alpha}}|_{\gamma_i} = 0$, we have

$$(\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}} \cdot \boldsymbol{\tau}_i)|_{\gamma_i} = 0, \quad i = 1, 2, 3. \tag{4.16}$$

Lemma 4.1.2. *For any* $(i, j, k) \in \mathcal{I}^n$,

$$\begin{aligned} \operatorname{curl}(\boldsymbol{\sigma}^n_{ijk}) = \frac{n+1}{|T|}[&(ij + ik + jk + n)B^n_{ijk} - \tfrac{i+1}{2}\big(jB^n_{i+1,j-1,k} + kB^n_{i+1,j,k-1}\big) \\ &- \tfrac{j+1}{2}\big(iB^n_{i-1,j+1,k} + kB^n_{i,j+1,k-1}\big) - \tfrac{k+1}{2}\big(iB^n_{i-1,j,k+1} + jB^n_{i,j-1,k+1}\big)]. \end{aligned}$$

Proof. By using (4.13) and (4.11) it is easy to show that

$$\operatorname{curl}(\boldsymbol{\sigma}^n_{ijk}) = -(n+1)\Big(\nabla B^n_{ijk}(i\boldsymbol{\omega}_1 + j\boldsymbol{\omega}_2 + k\boldsymbol{\omega}_3)^{\perp} - \frac{n}{|T|} B^n_{ijk}\Big).$$

By the chain rule,

$$\nabla B^n_{ijk} = n[B^{n-1}_{i-1,j,k}\nabla\lambda_1 + B^{n-1}_{i,j-1,k}\nabla\lambda_2 + B^{n-1}_{i,j,k-1}\nabla\lambda_3], \tag{4.17}$$

where, according to our convention, $B^m_{rst} := 0$ if $(r, s, t) \notin \mathcal{I}^m$. For example, $\nabla B^n_{n,0,0} = nB^{n-1}_{n-1,0,0}\nabla\lambda_1$, so that two terms in the right hand side of (4.17) are omitted. Inserting (4.10) and the definition of the Bernstein polynomials into

(4.17), we obtain

$$\begin{aligned}\nabla B^n_{ijk}\boldsymbol{\omega}_1^\perp &= n\big[\lambda_2 B^{n-1}_{i-1,j,k}\nabla\lambda_1(\nabla\lambda_3)^\perp - \lambda_3 B^{n-1}_{i-1,j,k}\nabla\lambda_1(\nabla\lambda_2)^\perp \\ &\qquad + \lambda_2 B^{n-1}_{i,j-1,k}\nabla\lambda_2(\nabla\lambda_3)^\perp - \lambda_3 B^{n-1}_{i,j,k-1}\nabla\lambda_3(\nabla\lambda_2)^\perp\big] \\ &= \frac{n}{2|T|}\big[\lambda_2 B^{n-1}_{i-1,j,k} + \lambda_3 B^{n-1}_{i-1,j,k} - \lambda_2 B^{n-1}_{i,j-1,k} - \lambda_3 B^{n-1}_{i,j,k-1}\big] \\ &= \frac{1}{2|T|}\big[(j+1)B^n_{i-1,j+1,k} + (k+1)B^n_{i-1,j,k+1} - jB^n_{ijk} - kB^n_{ijk}\big].\end{aligned}$$

Similarly,

$$\begin{aligned}\nabla B^n_{ijk}\boldsymbol{\omega}_2^\perp &= \frac{1}{2|T|}\big[(i+1)B^n_{i+1,j-1,k} + (k+1)B^n_{i,j-1,k+1} - iB^n_{ijk} - kB^n_{ijk}\big], \\ \nabla B^n_{ijk}\boldsymbol{\omega}_3^\perp &= \frac{1}{2|T|}\big[(i+1)B^n_{i+1,j,k-1} + (j+1)B^n_{i,j+1,k-1} - iB^n_{ijk} - jB^n_{ijk}\big].\end{aligned}$$

The lemma follows by substituting these expressions into the above formula for $\operatorname{curl}(\boldsymbol{\sigma}^n_{ijk})$. □

We are ready to describe a spanning set and a basis for $\mathbb{ND}_n$ associated with a triangle T.

Theorem 4.1.3. *A spanning set for $\mathbb{ND}_n$, $n \geq 0$, is given by*

$$\{\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \boldsymbol{\omega}_3\} \cup \big\{\nabla B^{n+1}_{\boldsymbol{\alpha}} : \ \boldsymbol{\alpha} \in \mathcal{I}^{n+1}\big\} \cup \big\{\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}} : \ \boldsymbol{\alpha} \in \mathcal{I}^n\big\}. \tag{4.18}$$

Moreover, let

$$\begin{aligned}E_n^{(i)} &:= \{\nabla B^{n+1}_{\boldsymbol{\alpha}} : \ \boldsymbol{\alpha} \in \gamma_i^{n+1}\} \cup \{\boldsymbol{\omega}_i\}, \quad i = 1,2,3, \\ I_n^\nabla &:= \{\nabla B^{n+1}_{\boldsymbol{\alpha}} : \ \boldsymbol{\alpha} \in \mathring{\mathcal{I}}^{n+1}\}, \\ I_n^\sigma &:= \big\{\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}} : \ \boldsymbol{\alpha} \in \mathcal{I}^n \setminus \{\boldsymbol{\alpha}_0\}\big\},\end{aligned}$$

where $\boldsymbol{\alpha}_0$ is an arbitrary index in $\mathcal{I}^n$. Then $\mathcal{B}_n := I_n^\sigma \cup I_n^\nabla \cup E_n^{(1)} \cup E_n^{(2)} \cup E_n^{(3)}$ is a basis for $\mathbb{ND}_n$.

Proof. The chain rules applied to the gradient operator shows that, for $\boldsymbol{\alpha} \in \mathcal{I}^{n+1}$,

$\nabla B^{n+1}_{\boldsymbol{\alpha}}$ is in $\mathbb{ND}_n$. Similarly, since the Whitney functions are in $\mathbb{ND}_0 = \mathbb{P}_0^2 + \operatorname{span}(\mathbf{x}^\perp)$, it follows from (4.14) that $\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}}$ belongs to $\mathbb{ND}_n$, for $\boldsymbol{\alpha} \in \mathcal{I}^n$. Thus, all functions listed in the first display of the theorem belong to $\mathbb{ND}_n$. Since $\mathcal{B}_n$ is included in the set defined in (4.18), it suffices to prove that $\mathcal{B}_n$ forms a basis for $\mathbb{ND}_n$. It is easy to check by definition that $\dim(\mathbb{ND}_n) = (n+1)(n+3) = \#\mathcal{B}_n$, and thus it suffices to prove that the functions in $\mathcal{B}_n$ are linearly independent.

To this end, we consider an arbitrary linear combination $\mathbf{S}$ of functions in $\mathcal{B}_n$,

$$\mathbf{S} = \mathbf{S}_{I,\sigma} + \mathbf{S}_{I,\nabla} + \mathbf{S}_{E,1} + \mathbf{S}_{E,2} + \mathbf{S}_{E,3},$$

where $\mathbf{S}_{I,\sigma} \in \operatorname{span}(I^\sigma_n)$, $\mathbf{S}_{I,\nabla} \in \operatorname{span}(I^\nabla_n)$, $\mathbf{S}_{E,i} \in \operatorname{span}(E^{(i)}_n)$, $i = 1, 2, 3$. Assuming that $\mathbf{S} = \mathbf{0}$, we will show that all coefficients of $\mathbf{S}$ are zero, which implies the desired linear independence.

It is easy to see that $(\mathbf{S} \cdot \boldsymbol{\tau}_i)|_{\gamma_i} = (\mathbf{S}_{E,i} \cdot \boldsymbol{\tau}_i)|_{\gamma_i}$, $i = 1, 2, 3$. Hence $\mathbf{S} = \mathbf{0}$ implies $(\mathbf{S}_{E,i} \cdot \boldsymbol{\tau}_i)|_{\gamma_i} = 0$. Let

$$\mathbf{S}_{E,3} = c\boldsymbol{\omega}_3 + \sum_{\boldsymbol{\alpha} \in \gamma_3^{n+1}} c_{\boldsymbol{\alpha}} \nabla B^{n+1}_{\boldsymbol{\alpha}} \qquad c, c_{\boldsymbol{\alpha}} \in \mathbb{R}.$$

By (4.12), $(\boldsymbol{\omega}_i \cdot \boldsymbol{\tau}_3)|_{\gamma_3} = \nabla\lambda_2 \cdot \boldsymbol{\tau}_3 = -\nabla\lambda_1 \cdot \boldsymbol{\tau}_3 = \frac{1}{|\gamma_3|}$. Hence

$$\begin{aligned}
\nabla B^{n+1}_{i,n+1-i,0} \cdot \boldsymbol{\tau}_3 &= (n+1)(B^n_{i-1,n+1-i,0}\nabla\lambda_1 + B^n_{i,n-i,0}\nabla\lambda_2) \cdot \boldsymbol{\tau}_3 \\
&= (n+1)(B^n_{i-1,n+1-i,0} - B^n_{i,n-i,0})\nabla\lambda_1 \cdot \boldsymbol{\tau}_3, \quad i = 1, \ldots, n,
\end{aligned}$$

and since $\sum_{i=0}^n B^n_{i,n-i,0} = 1$ on the edge γ_3, we obtain the following identity on this edge:

$$\begin{aligned}
0 = |\gamma_3|\mathbf{S}_{E,3} \cdot \boldsymbol{\tau}_3 &= c - (n+1)\sum_{i=1}^n c_{i,n+1-i,0}(B^n_{i-1,n+1-i,0} - B^n_{i,n-i,0}) \\
&= \sum_{i=0}^n (c - (n+1)c_{i+1,n-i,0} + (n+1)c_{i,n+1-i,0})B^n_{i,n-i,0},
\end{aligned}$$

where

$$c_{0,n+1,0} = c_{n+1,0,0} = 0. \tag{4.19}$$

By the linear independence of $B^n_{i,n-i,0}$, $i = 0, \ldots, n$, on γ_3, it follows that

$$c - (n+1)c_{i+1,n-i,0} + (n+1)c_{i,n+1-i,0} = 0, \quad i = 0, \ldots, n,$$

which is a linear system with respect to the $n+1$ unknows $c, c_{1,n,0}, \ldots, c_{n,1,0}$. Using (4.19), it is easy to check that the above-mentioned linear system admits only the trivial solution, thus implying that all coefficients of $\mathbf{S}_{E,3}$ are zero. Similarly, by considering the edges γ_1 and γ_2, we conclude that all coefficients of $\mathbf{S}_{E,1}$ and $\mathbf{S}_{E,2}$ are also zero. Hence

$$\mathbf{S} = \mathbf{S}_{I,\sigma} + \mathbf{S}_{I,\nabla}.$$

Since $\operatorname{curl}(\nabla f) = 0$ for any f, $\mathbf{S} = \mathbf{0}$ implies $\operatorname{curl}(\mathbf{S}_{I,\sigma}) = 0$. Let

$$\mathbf{S}_{I,\sigma} = \sum_{\boldsymbol{\alpha} \in \mathcal{I}^n} d_{\boldsymbol{\alpha}} \boldsymbol{\sigma}^n_{\boldsymbol{\alpha}}, \qquad d_{\boldsymbol{\alpha}} \in \mathbb{R}, \quad d_{\boldsymbol{\alpha}_0} = 0.$$

Then by applying Lemma 4.1.2 we obtain

$$0 = \sum_{\boldsymbol{\alpha} \in \mathcal{I}^n} d_{\boldsymbol{\alpha}} \operatorname{curl}(\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}}) = \sum_{(i,j,k) \in \mathcal{I}^n} \bar{d}_{ijk} B^n_{ijk},$$

with

$$\begin{aligned} \bar{d}_{ijk} = \frac{n+1}{|T|} \big[&(ij + ik + jk + n) d_{ijk} - \tfrac{i(j+1)}{2} d_{i-1,j+1,k} - \tfrac{i(k+1)}{2} d_{i-1,j,k+1} \\ &- \tfrac{j(i+1)}{2} d_{i+1,j-1,k} - \tfrac{j(k+1)}{2} d_{i,j-1,k+1} - \tfrac{k(i+1)}{2} d_{i+1,j,k-1} - \tfrac{k(j+1)}{2} d_{i,j+1,k-1} \big], \end{aligned}$$

which, in view of the linear independence of the Bernstein polynomials implies that

$$\bar{d}_{ijk} = 0, \qquad (i,j,k) \in \mathcal{I}^n.$$

The latter is a square linear system with respect to d_{ijk}, $(i,j,k) \in \mathcal{I}^n$. Since

the coefficients in each row of this system sum to zero, it is easy to see that the matrix of the system after the column and the row corresponding to $\boldsymbol{\alpha}_0$ are removed is irreducible and (weakly) diagonally dominant with at least one strongly diagonally dominant row. By a well known theorem of linear algebra this matrix is nonsingular, thus implying that $d_{\boldsymbol{\alpha}} = 0$ for all $\boldsymbol{\alpha} \in \mathcal{I}^n \setminus \{\boldsymbol{\alpha}_0\}$.

Hence the coefficients $g_{\boldsymbol{\alpha}}$ of $\mathbf{S}_{I,\nabla}$ satisfy

$$\mathbf{0} = \mathbf{S} = \mathbf{S}_{I,\nabla} = \sum_{\boldsymbol{\alpha} \in \mathring{\mathcal{I}}^{n+1}} g_{\boldsymbol{\alpha}} \nabla B^{n+1}_{\boldsymbol{\alpha}} = \nabla\Big(\sum_{\boldsymbol{\alpha} \in \mathring{\mathcal{I}}^{n+1}} g_{\boldsymbol{\alpha}} B^{n+1}_{\boldsymbol{\alpha}} \Big),$$

which implies that $\sum_{\boldsymbol{\alpha} \in \mathring{\mathcal{I}}^{n+1}} g_{\boldsymbol{\alpha}} B^{n+1}_{\boldsymbol{\alpha}}$ is a constant. Since this expression vanishes on the boundary of T, it follows that the constant is zero, and hence $g_{\boldsymbol{\alpha}} = 0$ for all $\boldsymbol{\alpha} \in \mathring{\mathcal{I}}^{n+1}$ by the linear independence of the Bernstein polynomials. □

Note that $\mathcal{B}_0 = \{\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \boldsymbol{\omega}_3\}$ which are the classical Whitney edge functions. In addition,

$$\mathcal{B}_1 = \{\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \boldsymbol{\omega}_3, \nabla B^2_{110}, \nabla B^2_{101}, \nabla B^2_{011}\} \cup I^{\sigma}_1,$$

where I^{σ}_1 consists of any two functions in $\{\boldsymbol{\sigma}^1_{\mathbf{e}_1}, \boldsymbol{\sigma}^1_{\mathbf{e}_2}, \boldsymbol{\sigma}^1_{\mathbf{e}_3}\}$, with $\boldsymbol{\sigma}^1_{\mathbf{e}_i} = \lambda_i \boldsymbol{\omega}_i$, $i = 1, 2, 3$. From the definition of the basis $\mathcal{B}_n$, observe that the only shape functions with non-zero tangential components consists of the edge gradients and the Whitney shape functions (see Section A.2 for the graph of the sigma and gradient basis functions for $n = 1$ and $n = 2$).

For simplicity, suppose that the shape functions described in Theorem 4.1.3 are indexed as in $\mathcal{B}_n = \{\mathbf{b}_j : j = 1, \ldots, \dim(\mathbb{ND}_n)\}$. Then, for $\mathbf{u} \in \mathbb{ND}_n$, there exists a unique coefficient sequence $\{c_j : \ j = 1, \ldots, \dim(\mathbb{ND}_n)\}$ satisfying

$$\mathbf{u} = \sum_{j=1}^{\dim(\mathbb{ND}_n)} c_j \mathbf{b}_j. \tag{4.20}$$

Introducing the set $\Sigma_n = \{\phi_j : \ j = 1, \ldots, \dim(\mathbb{ND}_n)\}$ consisting of linear func-

tionals given by

$$\mathbb{ND}_n \ni \mathbf{u} \mapsto \phi_j(\mathbf{u}) = c_j, \quad j = 1, \ldots, \dim(\mathbb{ND}_n),$$

Theorem 4.1.3 amounts to say that Σ_n *is unisolvent with respect to* $\mathbb{ND}_n$.

4.1.2 Bernstein-Bézier Vector Finite Element Spaces on a Partition

We discuss in this section the $H(\text{curl})$-conformity of the finite element. To this end, recall that a sufficient condition for a finite element to be $H(\text{curl})$ conforming is that the underlying space is in $H(\text{curl})$ and that inter-element tangential continuity holds (see Lemma 4.1.4 below). Moreover, observe that the only shape functions with non-vanishing tangential components consists of the interface gradient and the Whitney edge functions. Thus, tangential continuity needs to be checked only with the gradients and Whitney shape functions. One can check that the tangential components of the gradients match on the interface, whereas those of the Whitney functions match up to a minus sign. Therefore, the degrees of freeedom associated with the interface gradients are identified with each other, whereas, for the Whitney functions, the identification of the associated degrees of freedom takes account of the global orientation of the corresponding edge. More precisely, if the local and global orientation of a given edge are of opposite directions, then the sign of the corresponding Whitney function is reversed during the assembly process.

Let $\Delta = \{T\}$ be a regular triangulation of a polyhedral domain $\Omega \in \mathbb{R}^2$. Let $\mathbf{u} \in \mathbb{ND}_n$ and $T,\ T' \in \Delta$ such that $T \cap T' = \mathbf{e}$. Let $\boldsymbol{\tau}_T$ and $\boldsymbol{\tau}_{T'}$ respectively denote the oriented edge $\mathbf{e}$ with respect to T and T'. In particular, $\boldsymbol{\tau}_T = -\boldsymbol{\tau}_{T'}$. It then holds that $(\mathbf{u} \cdot \boldsymbol{\tau}_T)_{|T} = (\mathbf{u} \cdot \boldsymbol{\tau}_{T'})_{|T'}$ if and only if the interface gradient coefficients match and the Whitney coefficients are opposites with respect to each other on the interface. That is to say, if the values of the local gradient (respectively Whitney) degrees of freedom in the finite elements $(T, \mathbb{ND}_n(T), \Sigma_n(T))$ and

$(T', \mathbb{ND}_n(T'), \Sigma_n(T'))$ are the same (respectively opposites) on the interface, then $\mathbf{u}$ has continuous tangential components. The next lemma justifies tangential continuity as a criterion for $H(\mathrm{curl})$-conformity:

Lemma 4.1.4. *Using the above notations, let* $\mathbf{v}$ *denote a function such that, for any* $T \in \Delta$, $\mathbf{v}|_T \in H(\mathrm{curl}; T)$. *In addition, assume that* $\mathbf{v}$ *has continuous tangential components, that is, for any* $T, T' \in \Delta$ *satisfying* $T \cap T' = \mathbf{e} \neq \emptyset$, *it holds that*

$$[(\mathbf{v} \cdot \boldsymbol{\tau})|_{\mathbf{e}}]_{\mathrm{Jp}} := (\mathbf{v} \cdot \boldsymbol{\tau})|_{\mathbf{e} \cap T} - (\mathbf{v} \cdot \boldsymbol{\tau})|_{\mathbf{e} \cap T'} = 0.$$

Then $\mathbf{v}$ *is in* $H(\mathrm{curl}; \Omega)$.

Proof. See [65, Lemma 5.3]. □

The next theorem is an immediate consequence of Lemma 4.1.4:

Theorem 4.1.5. *Let* $S_n^{\mathrm{curl}}(\Delta)$ *denote the global finite element resulting from the assembly of the finite elements* $(T, \mathbb{ND}_n(T), \Sigma_n(T))$, $T \in \Delta$. *Then* $S_n^{\mathrm{curl}}(\Delta)$ *is* $H(\mathrm{curl})$*-conforming, and consists of tangentially continuous functions in the Nédélec space* $\mathbb{ND}_n(\Delta)$.

4.2 B-Moment Transformations

For simplicity, we will from now on make no distinction between equality and approximation. That is, depending on the context, the expression "$A = B$" may either reads "A is equal to B", or "B approximates A". The purpose of this section is to write all the $H(\mathrm{curl})$ quantities in terms B-moments. Combined with the routine `Moment2D`, or Algorithm 2.3 presented in Chapter 2, these transformations lead to the optimal complexity results detailed in Section 4.3.

Let $T \in \Delta$ be a triangle and let $\mathbf{f} : T \to X$ denote a smooth function, where X is a finite dimensional vector space (typically $\mathbb{R}$, $\mathbb{R}^2$ or $\mathbb{R}^{2\times 2}$). Recall that the

Bernstein-Bézier (B-) moments of degree n for the function $\mathbf{f}$ on T are given by

$$\mu_{\boldsymbol{\alpha}}^n(\mathbf{f}) = \int_T B_{\boldsymbol{\alpha}}^n(\mathbf{x})\mathbf{f}(\mathbf{x})\,\mathrm{d}\mathbf{x}, \quad \boldsymbol{\alpha} \in \mathcal{I}^n. \tag{4.21}$$

In addition, recall from Theorem 2.2.4 with the case $d = 2$ that the B-moments of order n can be computed with $\mathcal{O}(n^3)$ operations, when using the Stroud conical product rule based on $(n+1)^2$ quadrature points.

In this section we provide the transformation formulae for the element level load vector, mass and stiffness matrices (4.3) in terms of B-moments of the data when using the spanning set for $\mathbb{ND}_n$ described in Theorem 4.1.3. In an implementation appropriate rows and columns will need to be removed to obtain the matrices corresponding to the basis $\mathcal{B}_n$.

For a given matrix function $\boldsymbol{\kappa} : T \to \mathbb{R}^2$, let $(\cdot,\cdot)_{\boldsymbol{\kappa}}$ denote the inner product

$$(\mathbf{f},\mathbf{g})_{\boldsymbol{\kappa}} = \int_T \big(\boldsymbol{\kappa}(\mathbf{x})\mathbf{f}(\mathbf{x})\big)\cdot\mathbf{g}(\mathbf{x})\,\mathrm{d}\mathbf{x}.$$

In the case $\boldsymbol{\kappa}(\mathbf{x}) = \mathrm{Id} = \left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]$ we use the notation $(\cdot,\cdot)$ for simplicity.

We write the $H(\mathrm{curl})$ element load vector, mass and stiffness matrices in a block matrix form as

$$\mathbf{L} = \begin{bmatrix}\mathbf{f}^{\nabla} & \mathbf{f}^{\sigma} & \mathbf{f}^{W}\end{bmatrix}^t,$$

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}^{\nabla\nabla} & \mathbf{M}^{\nabla\sigma} & \mathbf{M}^{\nabla W} \\ (\mathbf{M}^{\nabla\sigma})^t & \mathbf{M}^{\sigma\sigma} & \mathbf{M}^{\sigma W} \\ (\mathbf{M}^{\nabla W})^t & (\mathbf{M}^{\sigma W})^t & \mathbf{M}^{WW} \end{bmatrix}, \qquad \mathbf{S} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^{\sigma\sigma} & \mathbf{S}^{\sigma W} \\ \mathbf{0} & (\mathbf{S}^{\sigma W})^t & \mathbf{S}^{WW} \end{bmatrix},$$

where

$$\begin{aligned}
\mathbf{f}_{\boldsymbol{\alpha}}^{\nabla} &= \big(\mathbf{f}, \nabla B_{\boldsymbol{\alpha}}^{n+1}\big), \quad \boldsymbol{\alpha} \in \mathcal{I}^{n+1},\\
\mathbf{f}_{\boldsymbol{\alpha}}^{\sigma} &= \big(\mathbf{f}, \boldsymbol{\sigma}_{\boldsymbol{\alpha}}^{n}\big), \quad \boldsymbol{\alpha} \in \mathcal{I}^{n},\\
\mathbf{f}_{i}^{W} &= \big(\mathbf{f}, \boldsymbol{\omega}_i\big), \quad i = 1,2,3,
\end{aligned}$$

$$
\begin{aligned}
\mathbf{M}^{\nabla\nabla}_{\boldsymbol{\alpha},\boldsymbol{\beta}} &= \big(\nabla B^{n+1}_{\boldsymbol{\alpha}}, \nabla B^{n+1}_{\boldsymbol{\beta}}\big)_{\boldsymbol{\kappa}}, \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}^{n+1},\\
\mathbf{M}^{\nabla\sigma}_{\boldsymbol{\alpha},\boldsymbol{\beta}} &= \big(\nabla B^{n+1}_{\boldsymbol{\alpha}}, \boldsymbol{\sigma}^{n}_{\boldsymbol{\beta}}\big)_{\boldsymbol{\kappa}}, \quad \boldsymbol{\alpha} \in \mathcal{I}^{n+1},\ \boldsymbol{\beta} \in \mathcal{I}^{n},\\
\mathbf{M}^{\nabla W}_{\boldsymbol{\alpha},i} &= \big(\nabla B^{n+1}_{\boldsymbol{\alpha}}, \boldsymbol{\omega}_i\big)_{\boldsymbol{\kappa}}, \quad \boldsymbol{\alpha} \in \mathcal{I}^{n+1},\ i = 1,2,3,\\
\mathbf{M}^{\sigma\sigma}_{\boldsymbol{\alpha},\boldsymbol{\beta}} &= \big(\boldsymbol{\sigma}^{n}_{\boldsymbol{\alpha}}, \boldsymbol{\sigma}^{n}_{\boldsymbol{\beta}}\big)_{\boldsymbol{\kappa}}, \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}^{n},\\
\mathbf{M}^{\sigma W}_{\boldsymbol{\alpha},i} &= \big(\boldsymbol{\sigma}^{n}_{\boldsymbol{\alpha}}, \boldsymbol{\omega}_i\big)_{\boldsymbol{\kappa}}, \quad \boldsymbol{\alpha} \in \mathcal{I}^{n},\ i = 1,2,3,\\
\mathbf{M}^{WW}_{i,j} &= \big(\boldsymbol{\omega}_i, \boldsymbol{\omega}_j\big)_{\boldsymbol{\kappa}}, \quad i,j = 1,2,3,
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{S}^{\sigma\sigma}_{\boldsymbol{\alpha},\boldsymbol{\beta}} &= \big(\operatorname{curl}(\boldsymbol{\sigma}^{n}_{\boldsymbol{\alpha}}), \operatorname{curl}(\boldsymbol{\sigma}^{n}_{\boldsymbol{\beta}})\big)_{A}, \quad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathcal{I}^{n}\\
\mathbf{S}^{\sigma W}_{\boldsymbol{\alpha},i} &= \big(\operatorname{curl}(\boldsymbol{\sigma}^{n}_{\boldsymbol{\alpha}}), \operatorname{curl}(\boldsymbol{\omega}_i)\big)_{A}, \quad \boldsymbol{\alpha} \in \mathcal{I}^{n}, i = 1,2,3,\\
\mathbf{S}^{WW}_{i,j} &= \big(\operatorname{curl}(\boldsymbol{\omega}_i), \operatorname{curl}(\boldsymbol{\omega}_j)\big)_{A}, \quad i,j = 1,2,3.
\end{aligned}
$$

Note that five blocks of $\mathbf{S}$ vanish because the curl of a gradient field is zero.

We also recall for later use the following expansions of $\boldsymbol{\sigma}^{n}_{\boldsymbol{\alpha}}$ and $\operatorname{curl}(\boldsymbol{\sigma}^{n}_{\boldsymbol{\alpha}})$ in the Bernstein basis,

$$
\boldsymbol{\sigma}^{n}_{\boldsymbol{\alpha}} = \sum_{i=1}^{3} \mathring{\mathbf{c}}^{(\boldsymbol{\alpha})}_{\boldsymbol{\alpha}+\mathbf{e}_i} B^{n+1}_{\boldsymbol{\alpha}+\mathbf{e}_i}, \qquad \boldsymbol{\alpha} \in \mathcal{I}^{n}, \tag{4.22}
$$

where

$$
\mathring{\mathbf{c}}^{(\boldsymbol{\alpha})}_{\boldsymbol{\alpha}+\mathbf{e}_k} = (\alpha_k + 1)(\alpha_{k-1}\nabla\lambda_{k+1} - \alpha_{k+1}\nabla\lambda_{k-1}), \quad k = 1,2,3, \tag{4.23}
$$

and

$$
\operatorname{curl}(\boldsymbol{\sigma}^{n}_{\boldsymbol{\alpha}}) = \sum_{\boldsymbol{\eta}\in\mathtt{Star}(\boldsymbol{\alpha})} \overline{c}^{(\boldsymbol{\alpha})}_{\boldsymbol{\eta}} B^{n}_{\boldsymbol{\eta}}, \qquad \boldsymbol{\alpha} \in \mathcal{I}^{n}, \tag{4.24}
$$

where $\mathtt{Star}(\boldsymbol{\alpha})$ is defined in (4.6), $\boldsymbol{\alpha} \in \mathcal{I}^n$, and

$$\begin{aligned} \overline{c}^{(\boldsymbol{\alpha})}_{\boldsymbol{\alpha}+\mathbf{e}_k-\mathbf{e}_\ell} &= -\frac{n+1}{2|T|}(\alpha_k+1)\alpha_\ell, \qquad k,\ell = 1,2,3, \quad k \neq \ell, \\ \overline{c}^{(\boldsymbol{\alpha})}_{\boldsymbol{\alpha}} &= \frac{n+1}{2|T|}\sum_{\substack{k,\ell=1 \\ k\neq\ell}}^{3}(\alpha_k+1)\alpha_\ell, \end{aligned} \tag{4.25}$$

Indeed, (4.23) follows from (4.15), and (4.25) follows from Lemma 4.1.2.

We can now state the main result of this section.

Theorem 4.2.1. *The entries of the $H(\mathrm{curl})$ load vector $\mathbf{L}$, mass matrix $\mathbf{M}$ and stiffness matrix $\mathbf{S}$ can be expressed in terms of B-moments of the data as follows:*

$$\mathbf{f}^{\nabla}_{\boldsymbol{\alpha}} = (n+1)\sum_{k=1}^{3}\nabla\lambda_k \cdot \mu^{n}_{\boldsymbol{\alpha}-\mathbf{e}_k}(\mathbf{f}), \tag{4.26}$$

$$\mathbf{f}^{\sigma}_{\boldsymbol{\alpha}} = \sum_{k=1}^{3}\mathring{\mathbf{c}}^{(\boldsymbol{\alpha})}_{\boldsymbol{\alpha}+\mathbf{e}_k} \cdot \mu^{n+1}_{\boldsymbol{\alpha}+\mathbf{e}_k}(\mathbf{f}), \tag{4.27}$$

$$\mathbf{f}^{W}_{i} = \sum_{r=\pm1} r\nabla\lambda_{i-r} \cdot \mu^{1}_{\mathbf{e}_{i+r}}(\mathbf{f}), \tag{4.28}$$

where $\mathring{\mathbf{c}}^{(\boldsymbol{\alpha})}_{\boldsymbol{\alpha}+\mathbf{e}_k}$ is defined in (4.23), and the terms in (4.26) for which $\boldsymbol{\alpha}-\mathbf{e}_k \notin \mathcal{I}^n$ are ignored in the summation,

$$\mathbf{M}^{\nabla\nabla}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = (n+1)^2\sum_{k,\ell=1}^{3}\frac{\binom{\boldsymbol{\alpha}-\mathbf{e}_k+\boldsymbol{\beta}-\mathbf{e}_\ell}{\boldsymbol{\alpha}-\mathbf{e}_k}}{\binom{2n}{n}}\nabla\lambda_\ell \cdot \mu^{2n}_{\boldsymbol{\alpha}-\mathbf{e}_k+\boldsymbol{\beta}-\mathbf{e}_\ell}(\boldsymbol{\kappa}) \cdot \nabla\lambda_k, \tag{4.29}$$

$$\mathbf{M}^{\nabla\sigma}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = (n+1)\sum_{k,\ell=1}^{3}\frac{\binom{\boldsymbol{\alpha}-\mathbf{e}_k+\boldsymbol{\beta}+\mathbf{e}_\ell}{\boldsymbol{\alpha}-\mathbf{e}_k}}{\binom{2n+1}{n+1}}\nabla\lambda_k \cdot \mu^{2n+1}_{\boldsymbol{\alpha}-\mathbf{e}_k+\boldsymbol{\beta}+\mathbf{e}_\ell}(\boldsymbol{\kappa}) \cdot \mathring{\mathbf{c}}^{(\boldsymbol{\beta})}_{\boldsymbol{\beta}+\mathbf{e}_\ell}, \tag{4.30}$$

$$\mathbf{M}^{\nabla W}_{\boldsymbol{\alpha},i} = \sum_{k=1}^{3}\sum_{r=\pm1} r\binom{\boldsymbol{\alpha}-\mathbf{e}_k+\mathbf{e}_{i+r}}{\boldsymbol{\alpha}-\mathbf{e}_k}\nabla\lambda_k \cdot \mu^{n+1}_{\boldsymbol{\alpha}-\mathbf{e}_k+\mathbf{e}_{i+r}}(\boldsymbol{\kappa}) \cdot \nabla\lambda_{i-r}, \tag{4.31}$$

$$\mathbf{M}^{\sigma\sigma}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \sum_{k,\ell=1}^{3} \frac{\binom{\boldsymbol{\alpha}+\mathbf{e}_k+\boldsymbol{\beta}+\mathbf{e}_\ell}{\boldsymbol{\alpha}+\mathbf{e}_k}}{\binom{2n+2}{n+1}} \mathring{\mathbf{c}}^{(\boldsymbol{\alpha})}_{\boldsymbol{\alpha}+\mathbf{e}_k} \cdot \mu^{2n+2}_{\boldsymbol{\alpha}+\mathbf{e}_k+\boldsymbol{\beta}+\mathbf{e}_\ell}(\boldsymbol{\kappa}) \cdot \mathring{\mathbf{c}}^{(\boldsymbol{\beta})}_{\boldsymbol{\beta}+\mathbf{e}_\ell}, \tag{4.32}$$

$$\mathbf{M}^{\sigma W}_{\boldsymbol{\alpha},i} = \sum_{k=1}^{3} \sum_{r=\pm1} \frac{r\binom{\boldsymbol{\alpha}+\mathbf{e}_k+\mathbf{e}_{i+r}}{\boldsymbol{\alpha}+\mathbf{e}_k}}{n+2} \mathring{\mathbf{c}}^{(\boldsymbol{\alpha})}_{\boldsymbol{\alpha}+\mathbf{e}_k} \cdot \mu^{n+2}_{\boldsymbol{\alpha}+\mathbf{e}_k+\mathbf{e}_{i+r}}(\boldsymbol{\kappa}) \cdot \nabla\lambda_{i-r}, \tag{4.33}$$

$$\mathbf{M}^{WW}_{i,j} = \sum_{r,s=\pm1} \frac{rs}{\binom{2}{\mathbf{e}_{i+r}+\mathbf{e}_{j+s}}} \nabla\lambda_{i-r} \cdot \mu^{2}_{\mathbf{e}_{i+r}+\mathbf{e}_{j+s}}(\boldsymbol{\kappa}) \cdot \nabla\lambda_{j-s}, \tag{4.34}$$

where, in (4.29), (4.30) *and* (4.31), *the terms of the form* $\mu^m_{\boldsymbol{\eta}}$ *for which* $\boldsymbol{\eta} \notin \mathcal{I}^m$ *are ignored in the summation, and*

$$\mathbf{S}^{\sigma\sigma}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \sum_{\substack{\boldsymbol{\eta}\in\mathtt{Star}(\boldsymbol{\alpha})\\ \boldsymbol{\rho}\in\mathtt{Star}(\boldsymbol{\beta})}} \frac{\binom{\boldsymbol{\eta}+\boldsymbol{\rho}}{\boldsymbol{\eta}}}{\binom{2n}{n}} \overline{c}^{(\boldsymbol{\alpha})}_{\boldsymbol{\eta}} \overline{c}^{(\boldsymbol{\beta})}_{\boldsymbol{\rho}} \mu^{2n}_{\boldsymbol{\eta}+\boldsymbol{\rho}}(A), \tag{4.35}$$

$$\mathbf{S}^{\sigma W}_{\boldsymbol{\alpha},i} = \frac{1}{|T|} \sum_{\boldsymbol{\eta}\in\mathtt{Star}(\boldsymbol{\alpha})} \overline{c}^{(\boldsymbol{\alpha})}_{\boldsymbol{\eta}} \mu^{n}_{\boldsymbol{\eta}}(A), \tag{4.36}$$

$$\mathbf{S}^{WW}_{i,j} = \frac{1}{|T|^2} \int_T A(\mathbf{x})\, d\mathbf{x} = \frac{1}{|T|^2} \mu^0_{\mathbf{0}}(A), \tag{4.37}$$

where $\overline{c}^{(\boldsymbol{\alpha})}_{\boldsymbol{\eta}}$, $\boldsymbol{\eta} \in \mathtt{Star}(\boldsymbol{\alpha})$, *is defined in* (4.25).

Proof. The chain rule applied to the gradient of the Bernstein polynomial gives $\nabla B^{n+1}_{\boldsymbol{\alpha}} = (n+1) \sum_{k=1}^{3} \nabla\lambda_k B^n_{\boldsymbol{\alpha}-\mathbf{e}_k}$. Hence,

$$\mathbf{f}^{\nabla}_{\boldsymbol{\alpha}} = \int_T \mathbf{f}(\mathbf{x}) \cdot \nabla B^{n+1}_{\boldsymbol{\alpha}}(\mathbf{x})\, \mathrm{d}\mathbf{x} = (n+1) \sum_{k=1}^{3} \nabla\lambda_k \cdot \int_T \mathbf{f}(\mathbf{x}) B^n_{\boldsymbol{\alpha}-\mathbf{e}_k}(\mathbf{x})\, \mathrm{d}\mathbf{x},$$

which yields (4.26). Similarly, (4.27) follows from (4.22), and (4.28) is obtained from the following calculation:

$$\begin{aligned} \mathbf{f}^{W}_{i} &= \int_T \mathbf{f}(\mathbf{x}) \cdot (\lambda_{i+1}(\mathbf{x})\nabla\lambda_{i-1} - \lambda_{i-1}(\mathbf{x})\nabla\lambda_{i+1})\, \mathrm{d}\mathbf{x} \\ &= \nabla\lambda_{i-1} \cdot \int_T \mathbf{f}(\mathbf{x}) B^1_{\mathbf{e}_{i+1}}(\mathbf{x})\, \mathrm{d}\mathbf{x} - \nabla\lambda_{i+1} \cdot \int_T \mathbf{f}(\mathbf{x}) B^1_{\mathbf{e}_{i-1}}\, \mathrm{d}\mathbf{x}. \end{aligned}$$

The formula (4.29) is a particular case of [11, Eq. (44)]. To obtain (4.30), we

need the well known product formula for Bernstein polynomials,

$$B^n_{\boldsymbol{\alpha}} B^m_{\boldsymbol{\beta}} = \frac{\binom{\boldsymbol{\alpha}+\boldsymbol{\beta}}{\boldsymbol{\alpha}}}{\binom{m+n}{n}} B^{m+n}_{\boldsymbol{\alpha}+\boldsymbol{\beta}}, \quad \boldsymbol{\alpha} \in \mathcal{I}^n,\ \boldsymbol{\beta} \in \mathcal{I}^m. \tag{4.38}$$

Indeed, by applying (4.22) and the gradient formula, we have

$$\begin{aligned}
\mathbf{M}^{\nabla\sigma}_{\boldsymbol{\alpha},\boldsymbol{\beta}} &= \int_T \boldsymbol{\sigma}^n_{\boldsymbol{\beta}}(\mathbf{x}) \cdot \boldsymbol{\kappa}(\mathbf{x}) \cdot \nabla B^{n+1}_{\boldsymbol{\alpha}}(\mathbf{x})\, \mathrm{d}\mathbf{x}. \\
&= (n+1) \sum_{k,\ell=1}^{3} \mathring{\mathbf{c}}^{(\boldsymbol{\beta})}_{\boldsymbol{\beta}+\mathbf{e}_\ell} \cdot \int_T \boldsymbol{\kappa}(\mathbf{x}) B^{n+1}_{\boldsymbol{\beta}+\mathbf{e}_\ell}(\mathbf{x}) B^n_{\boldsymbol{\alpha}-\mathbf{e}_k}(\mathbf{x})\, \mathrm{d}\mathbf{x} \cdot \nabla\lambda_k,
\end{aligned}$$

and (4.30) follows from (4.38). Again by the gradient formula,

$$\begin{aligned}
\mathbf{M}^{\nabla W}_{\boldsymbol{\alpha},i} &= \int_T \nabla B^{n+1}_{\boldsymbol{\alpha}}(\mathbf{x}) \cdot \boldsymbol{\kappa}(\mathbf{x}) \cdot (\lambda_{i+1}(\mathbf{x})\nabla\lambda_{i-1} - \lambda_{i-1}(\mathbf{x})\nabla\lambda_{i+1})\, \mathrm{d}\mathbf{x} \\
&= (n+1) \sum_{k=1}^{3} \nabla\lambda_k \cdot \int_T \boldsymbol{\kappa}(\mathbf{x}) B^n_{\boldsymbol{\alpha}-\mathbf{e}_k}(\mathbf{x}) \cdot \left(B^1_{\mathbf{e}_{i+1}}(\mathbf{x})\nabla\lambda_{i-1} - B^1_{\mathbf{e}_{i-1}}\nabla\lambda_{i+1}\right) \mathrm{d}\mathbf{x},
\end{aligned}$$

and by using (4.38) we obtain (4.31). Equations (4.32) and (4.33) can be shown similarly to (4.30) and (4.31). Next,

$$\begin{aligned}
\mathbf{M}^{WW}_{i,j} &= \int_T \sum_{r=\pm1} r\lambda_{i+r}(\mathbf{x})\nabla\lambda_{i-r} \cdot \boldsymbol{\kappa}(\mathbf{x}) \cdot \sum_{s=\pm1} s\lambda_{j+s}(\mathbf{x})\nabla\lambda_{j-s}\, \mathrm{d}\mathbf{x} \\
&= \sum_{r,s=\pm1} rs\nabla\lambda_{i-r} \cdot \int_T \boldsymbol{\kappa}(\mathbf{x})\lambda_{i+r}(\mathbf{x})\lambda_{j+s}(\mathbf{x})\, \mathrm{d}\mathbf{x} \cdot \nabla\lambda_{j-s},
\end{aligned}$$

which implies (4.34) since $B^2_{\mathbf{e}_k+\mathbf{e}_\ell} = \binom{2}{\mathbf{e}_k+\mathbf{e}_\ell}\lambda_k\lambda_\ell$. Finally, (4.35)–(4.37) follow immediately from (4.24), (4.38) and (4.11). □

Considering that the formulas (4.26)–(4.37) include B-moments of different degrees for the same data, for example the moments of $\boldsymbol{\kappa}$ of degree 2, $n+1$, $n+2$, $2n$, $2n+1$ and $2n+2$, it is important to investigate the cheapest method for working with B-moments of various orders. Thus, observe that the moments of a lower degree can be obtained from the moments of a higher degree by a simple *degree raising* transformation, rather than computed independently according to Theorem 2.2.4. Note that degree raising is one of standard tools in curve and

surface modelling [64, Section 2.15].

Lemma 4.2.2. *Let $\ell < n$. Then, the degree-raising formula is given by:*

$$\mu^{\ell}_{\boldsymbol{\alpha}}(\mathbf{f}) = \sum_{\boldsymbol{\eta}\in\mathcal{I}^{n-\ell}} \frac{\binom{\boldsymbol{\alpha}+\boldsymbol{\eta}}{\boldsymbol{\alpha}}}{\binom{n}{\ell}} \mu^{n}_{\boldsymbol{\alpha}+\boldsymbol{\eta}}(\mathbf{f}), \quad \boldsymbol{\alpha} \in \mathcal{I}^{\ell}. \tag{4.39}$$

Proof. Since Bernstein polynomials build a partition of unity, we have

$$\mu^{\ell}_{\boldsymbol{\alpha}}(\mathbf{f}) = \int_T B^{\ell}_{\boldsymbol{\alpha}}(\mathbf{x})\mathbf{f}(\mathbf{x})\,\mathrm{d}\mathbf{x} = \sum_{\boldsymbol{\eta}\in\mathcal{I}^{n-\ell}} \int_T B^{n-\ell}_{\boldsymbol{\eta}}(\mathbf{x}) B^{\ell}_{\boldsymbol{\alpha}}(\mathbf{x})\mathbf{f}(\mathbf{x})\,\mathrm{d}\mathbf{x},$$

and (4.39) follows by the product formula (4.38). □

In view of Lemma 4.2.2, there are three possible alternatives for handling different degrees: lower the B-moment order one step at a time using *repeated* applications of the degree-raising formula, "jump" to the desired B-moment order using only *one* application of the degree-raising formula, or directly computing the desired B-moments using `Moment2D`. The first and second approaches are respectively referred to as *degree-lowering* and *degree-jump*.

We next proceed to compare the cost involved in using degree-lowering, degree-jump or the direct computation of the B-moments:

4.2.1 Degree-Lowering for B-Moments

By (4.39), the formula for degree raising from $n-1$ to n has the form

$$\mu^{n-1}_{\boldsymbol{\alpha}}(\mathbf{f}) = \frac{1}{n}\sum_{k=1}^{3}(\alpha_k+1)\mu^{n}_{\boldsymbol{\alpha}+\mathbf{e}_k}(\mathbf{f}), \quad \boldsymbol{\alpha} \in \mathcal{I}^{n-1},$$

and allows to compute B-moments $\mu^{n-1}_{\boldsymbol{\alpha}}(\mathbf{f})$ of degree $n-1$ if the moments $\{\mu^{n}_{\boldsymbol{\alpha}}(\mathbf{f}) : \boldsymbol{\alpha} \in \mathcal{I}^{n}\}$ of degree n are known, which leads to the routine `LowerMoment` of Algorithm 4.1.

We now estimate the cost of repeated degree-lowering in the case where $X = \mathbb{R}^2$.

Algorithm 4.1: LowerMoment($\mathbf{F}$)

Input : Array $\mathbf{F}$ corresponding to degree n B-moments $\{\mu^n_{\boldsymbol{\alpha}}(\mathbf{f}) : \boldsymbol{\alpha} \in \mathcal{I}^n\}$ of $\mathbf{f} : T \to X$.

Output: Array $\mathbf{F}^{\text{out}}$ of degree $n-1$ B-moments $\{\mu^{n-1}_{\boldsymbol{\alpha}}(\mathbf{f}) : \boldsymbol{\alpha} \in \mathcal{I}^{n-1}\}$.

```
foreach α ∈ I^{n-1} do
    foreach k = 1, 2, 3 do
        F^out[α] += (α_k + 1) * F[α + e_k];
F^out /= n;
Return F^out;
```

Lemma 4.2.3. *Assume that the B-moments $\{\mu^n_{\boldsymbol{\alpha}}(\mathbf{f}) : \boldsymbol{\alpha} \in \mathcal{I}^n\}$ of degree n of a function $\mathbf{f} : T \to \mathbb{R}^2$ are known. Then its B-moments $\{\mu^\ell_{\boldsymbol{\alpha}}(\mathbf{f}) : \boldsymbol{\alpha} \in \mathcal{I}^\ell\}$ of degree $\ell < n$ can be computed by repeated applications of Algorithm 4.1 with $6\left[\binom{n+2}{3} - \binom{\ell+2}{3}\right]$ operations.*

Proof. Using `LowerMoment` to lower the degree from k to $k-1$ will cost $6\binom{k+1}{2}$ multiplications, and we need to execute this for $k = n, \ldots, \ell+1$, thus giving the overall cost of

$$6\sum_{k=\ell+1}^{n}\binom{k+1}{2} = 6\sum_{k=\ell}^{n-1}\binom{k+2}{2} = 6\sum_{k=0}^{n-1}\binom{k+2}{2} - 6\sum_{k=0}^{\ell-1}\binom{k+2}{2},$$

and the lemma follows since $\sum_{k=0}^{m}\binom{k+2}{2} = \binom{m+3}{3}$. □

4.2.2 Degree-Jump for B-Moments

Another way to approximate the moments of order ℓ from those of order n is, rather than lowering the order of the moments one step at a time, jumping directly from n to ℓ. More precisely, the degree-jump is obtained by directly implementing (4.39), as detailed in Algorithm 4.2.

In the case where $X = \mathbb{R}^2$, the following result holds:

Lemma 4.2.4. *Assume that the B-moments $\{\mu^n_{\boldsymbol{\alpha}}(\mathbf{f}) : \boldsymbol{\alpha} \in \mathcal{I}^n\}$ of degree n of a function $\mathbf{f} : T \to \mathbb{R}^2$ are known. Then its B-moments $\{\mu^\ell_{\boldsymbol{\alpha}}(f) : \boldsymbol{\alpha} \in \mathcal{I}^\ell\}$ of degree $\ell < n$ can be computed by means of Algorithm 4.2 with $(\ell+1)(n-\ell+1)(\ell n -$*

Algorithm 4.2: Degree-Jump($\mathbf{F}, n, \ell$)

Input : Array $\mathbf{F}$ corresponding to order n BB moments $\{\mu^n_{\boldsymbol{\alpha}}(\mathbf{f}) : \boldsymbol{\alpha} \in \mathcal{I}^n\}$ of $\mathbf{f} : T \to X$, and precomputed binomial coefficients $\{C_q^{p+q} : 0 \le p \le n, 0 \le q \le n\}$.

Output: Array $\mathbf{F}^{\text{out}}$ of order ℓ BB moments $\{\mu^\ell_{\boldsymbol{\alpha}}(\mathbf{f}) : \boldsymbol{\alpha} \in \mathcal{I}^\ell\}$.

$\mathbf{F}^{\text{out}} \equiv \mathbf{0}$;
for $\alpha_1 = \ell$ **to** 0 **do**
  for $\beta_1 = n - \ell$ **to** 0 **do**
    $w_1 = C_{\alpha_1}^{\alpha_1+\beta_1} / C_\ell^n$;
    for $\alpha_2 = \ell - \alpha_1$ **to** 0 **do**
      for $\beta_2 = n - \ell - \beta_1$ **to** 0 **do**
        $w_2 = w_1 * C_{\alpha_2}^{\alpha_2+\beta_2}$;
        $\alpha_3 = \ell - \alpha_1 - \alpha_2,\ \beta_3 = n - \ell - \beta_1 - \beta_2$;
        $w_3 = w_2 * C_{\alpha_3}^{\alpha_3+\beta_3}$;
        $\mathbf{F}^{\text{out}}_{\boldsymbol{\alpha}} \mathrel{+}= w_3 * \mathbf{F}[\boldsymbol{\alpha} + \boldsymbol{\beta}]$;
Return $\mathbf{F}^{\text{out}}$;

$\ell^2 + 2n + 5)$ *operations.*

Proof. From the algorithm description, one can see that Algorithm 4.2 indeed performs the operations in (4.39). We now proceed with the complexity analysis of the algorithm. To this end, observe that the loop over the pair (α_1, β_1) is executed $(\ell + 1)(n - \ell + 1)$ times, and contains one multiplication. Similarly, the loop over the pair (α_2, β_2) is executed $\binom{\ell+2}{2} \times \binom{n-\ell+2}{2}$ times, and contains one scalar-vector multiplication (which amounts to two multiplications) and two multiplications. Thus, the overall complexity for computing the BB moments of order ℓ, using Algorithm 4.2, is given by

$$\begin{aligned}(\ell+1) \times (n-\ell+1) + 4 \times \binom{\ell+2}{2} \times \binom{n-\ell+2}{2} \\ &= (\ell+1)(n-\ell+1)[1 + (\ell+2)(n-\ell+2)] \\ &= (\ell+1)(n-\ell+1)(\ell n - \ell^2 + 2n + 5). \end{aligned} \tag{4.40}$$

□

4.2.3 Direct Computation

Using a similar argument as that leading to (2.15) yields the next lemma:

Lemma 4.2.5. *For* $\mathbf{f} : T \to \mathbb{R}^2$ *and* $\ell < n$, *the B-moments* $\{\mu^{\ell}_{\boldsymbol{\alpha}}(\mathbf{f}) : \boldsymbol{\alpha} \in \mathcal{I}^{\ell}\}$ *computed with the* $q = n + 1$ *Stroud product rule, can be assembled with* $2[(n+1)^2(\ell+1) + \frac{(\ell+1)(\ell+2)}{2}(n+1)]$ *operations, using* `Moment2D`.

We now proceed to compare the cost between the alternatives for computing B-moments of different degrees. First, note from Algorithm 4.2 that, when only decreasing the B-moment degree by one, degree-jumping contains many redundancies, since most of the binomial coefficients are just equal to 1. Thus, in that case, `LowerMoment` is recommended. But it is surprising that, even for a longer jump, `LowerMoment` is also preferable. Indeed, degree-jumping from $2n$ to $n+2$ involves $n^4+6n^3+6n^2-10n-3$ operations, whereas lowering the degree one step at a time from $2n$ to $n+2$ costs $7n^3+3n^2-22n-24$ operations. Another application of Lemma 4.2.5 shows that directly computing the B-moments of order $n+2$ involves $3n^3+18n^2+33n+18$ operations, and thus is the cheapest option. Now, when decreasing the degree all the way to degree 2, degree-jumping appears to be faster than degree-lowering. Indeed, degree-jumping from degree $n+1$ to degree 2 involves $12n^2+15n$ operations, whereas degree-lowering costs $n^3+6n^2+11n-18$ operations. However, Lemma 4.2.5 shows that the direct computation of the B-moments of order 2 by means of `Moment2D` only costs $6n^2+24n+18$. operations.

In view of the above discussion, our approach to handling different B-moment degrees, for example for the matrix $\boldsymbol{\kappa}$, is as follows: Compute the moments of order $2n+2$. Use degree-lowering to get the moments of order $2n+1$, then use it again to obtain those of order $2n$. Compute the B-moments of order $n+2$ using the routine `Moment2D`. Then apply degree-lowering to get the B-moments of order $n+1$. Finally, compute the B-moments of order 2 by means of `Moment2D`.

One should also note that all the element-level computations are performed using the same number of quadrature points. Thus, one can precompute and store the value of the data at the quadrature nodes, thereby reducing the number

of function evaluations.

4.3 Optimal Order Element Level Computations

4.3.1 Evaluation of the Element Load Vector

In this section, we provide Algorithm 4.3 for computing the $H(\mathrm{curl})$ BB element load vector of order n associated with some data $\mathbf{f}$. In Algorithm 4.3, recall that the routine `Moment2D` from Chapter 3 computes the B-moments corresponding to the function $\mathbf{f}$. Hence, the $H(\mathrm{curl})$ load vector of $\mathbf{f}$ associated with a basis function in $\mathcal{B}^n$ is computed as a linear combination of B-moments, where the weights are taken as the corresponding BB coefficients. In particular, we make use of (4.26), (4.27), and Equation (4.28).

Theorem 4.3.1. *Let $n \in \mathbb{N}$ and $q = n + 2$. The element load vector of degree n can be computed using the Stroud conical product rule with q^2 quadrature points in $\mathcal{O}(n^3)$ operations using Algorithm* `HCurlLoad`.

Proof. According to Theorem 2.2.4, the B-moments of order $n + 1$ based on the Stroud conical quadrature rule with $q = n + 2$ can be approximated in $\mathcal{O}(n^3)$ operations. Besides, it follows from Lemma 4.2.3 that the B-moments of lower order can be computed in $\mathcal{O}(n^2)$ operations.

We now consider the computation of $\mathbf{f}^{\nabla}$ in Algorithm `HCurlLoad`: Note that the complexity is dominated by the innermost loop over i containing one inner product which amounts to two scalar multiplications, and is executed $3\binom{n+2}{2}$ times. Thus, given the B-moments of order $n + 1$, the overall complexity for computing $\mathbf{f}^{\nabla}$ is given by $6\binom{n+2}{2} = \mathcal{O}(n^2)$. After taking account of the complexity required for approximating the B-moments, we deduce that the complexity needed for computing $\mathbf{f}^{\nabla}$ is of order $\mathcal{O}(n^3)$.

Next, note that the complexity needed for computing $\mathbf{f}^{W}$ is dominated by the innermost loop which is executed 3^2 times, and contains seven multiplications.

Algorithm 4.3: HCurlLoad

Input : Array $\mathbf{F}$ of 2D vectors corresponding to values of a vector-valued function $\mathbf{f}$ at Stroud nodes.

Output: Blocks $\mathbf{f}^{\nabla}$, $\mathbf{f}^{\sigma}$ and $\mathbf{f}^{W}$ of the $H(\mathrm{curl})$ load vector $\mathbf{L}$.

$\mathbf{f}^{\nabla} \equiv \mathbf{0},\ \mathbf{f}^{\sigma} \equiv \mathbf{0},\ \mathbf{f}^{W} \equiv \mathbf{0}$;
$q = n + 2$;
/* Element sub-load vector $\mathbf{f}^{\sigma}$ */;
$\mathbf{f}^{\mathrm{mmt}}$=Moment2D $(\mathbf{F}, q, n+1)$ /* cf. Algorithm 2.3 */ ;
foreach $\boldsymbol{\beta} \in \mathcal{I}^n$ **do**
    for $i = 1$ *to* 3 **do**
        $\mathbf{f}^{\sigma}_{\boldsymbol{\beta}}$ += $(\beta_i + 1)(\beta_{i-1}\nabla\lambda_{i+1} - \beta_{i+1}\nabla\lambda_{i-1}) \cdot \mathbf{f}^{\mathrm{mmt}}_{\boldsymbol{\beta}+\mathbf{e}_i}$;
        /* $\mathring{\mathbf{c}}^{(\boldsymbol{\beta})}_{\boldsymbol{\beta}+\mathbf{e}_i}$ given in (4.23) */;
/* Element sub-load vector $\mathbf{f}^{\nabla}$ */;
$\mathbf{f}^{\mathrm{mmt}}$=LowerMoment $(\mathbf{f}^{\mathrm{mmt}})$ /* cf. Algorithm 4.1 */ ;
foreach $\boldsymbol{\alpha} \in \mathcal{I}^n$ **do**
    for $i = 1$ **to** 3 **do**
        $\mathbf{f}^{\nabla}_{\boldsymbol{\alpha}+\mathbf{e}_i}$ += $\nabla\lambda_i \cdot \mathbf{f}^{\mathrm{mmt}}_{\boldsymbol{\alpha}}$;
/* Element sub-load vector $\mathbf{f}^{W}$ */;
$\mathbf{f}^{\mathrm{mmt}}$= Moment2D $(\mathbf{F}, q, 1)$ /* cf. Algorithm 2.3 */ ;
for $i = 1, 2, 3$ **do**
    for $r = \pm 1$ **do**
        $\mathbf{f}^{W}_{i}$ += $r * \nabla\lambda_{i-r} \cdot \mathbf{f}^{\mathrm{mmt}}_{\mathbf{e}_{i+r}}$;
Return $\mathbf{f}^{\nabla}$, $\mathbf{f}^{\sigma}$, $\mathbf{f}^{W}$;

Thus, given the B-moments of order 2, the overall complexity for computing $\mathbf{f}^W$ is given by $7 \times 3^2 = \mathcal{O}(1)$. After taking account of the complexity needed for approximating the B-moments of order 1 based on the Stroud conical product rule with $q = n + 2$, we deduce that the complexity needed for computing $\mathbf{f}^W$ is of order $\mathcal{O}(n^2)$.

A similar argument for $\mathbf{f}^\sigma$ completes the proof. □

4.3.2 Evaluation of the Element Mass Matrix

The goal of this section is to provide efficient algorithms for the computation of the $H(\mathrm{curl})$ element mass matrix. Most of the presented algorithms refer to the following `Multinomial` routine, as presented in [11, Algorithm 5] for arbitrary dimension:

Algorithm 4.4: Multinomial$(\mathbf{D}, m, n)$

Input : Precomputed binomial coefficients $\{C_p^{p+q} : \ 0 \le p \le m, 0 \le q \le n\}$.

Output: $\mathbf{D}$ such that $\mathbf{D}_{\boldsymbol{\alpha},\boldsymbol{\beta}} = \binom{\boldsymbol{\alpha}+\boldsymbol{\beta}}{\boldsymbol{\alpha}} / \binom{m+n}{n}$, $\boldsymbol{\alpha} \in \mathcal{I}^m$, $\boldsymbol{\beta} \in \mathcal{I}^n$.

```
D = 0;
for α1 = m to 0 do
    for β1 = n to 0 do
        w1 = C_{α1}^{α1+β1} / C_n^{m+n};
        for α2 = m − α1 to 0 do
            for β2 = n − β1 to 0 do
                w2 = w1 * C_{α2}^{α2+β2};
                α3 = m − α1 − α2, β3 = n − β1 − β2;
                w3 = w2 * C_{α3}^{α3+β3};
                D_{α,β} += w3;
Return D;
```

Recall that the computation of $\mathbf{M}^{\nabla\nabla}$ is proved to be of optimal complexity $\mathcal{O}(n^4)$, if using Algorithm `StiffMat`, or Algorithm 3.21.

Now observe that Algorithm 4.5 and Algorithm 4.9 are direct consequences of (4.30) and (4.34), respectively. Besides, recall from (4.32) that the constant factor in the expression of $\mathbf{M}^{\sigma\sigma}$ is given by $1/\binom{2n+2}{n+1}$, whereas the constant fac-

tor produced by `Multinomial` $(\mathbf{M}^{\sigma\sigma}, n, n)$ is $1/\binom{2n}{n}$. Hence, in order to get the right constant for $\mathbf{M}^{\sigma\sigma}$, the computed quantities need to be factorized with $\binom{2n}{n}/\binom{2n+2}{n+1} = (n+1)/[2(2n+1)]$, as done in the line 8 of Algorithm 4.7. In addition, note that, for $m \in \mathbb{N}$, $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{I}^m$, and $k = 1, 2, 3$, it holds that

$$\binom{\boldsymbol{\alpha} + \mathbf{e}_k + \boldsymbol{\beta}}{\boldsymbol{\alpha} + \mathbf{e}_k} = \frac{\alpha_k + 1 + \beta_k}{\alpha_k + 1} \binom{\boldsymbol{\alpha} + \boldsymbol{\beta}}{\boldsymbol{\alpha}},$$

which gives, for $\ell = 1, 2, 3$,

$$\binom{\boldsymbol{\alpha} + \mathbf{e}_k + \boldsymbol{\beta} + \mathbf{e}_\ell}{\boldsymbol{\alpha} + \mathbf{e}_k} = \frac{\beta_\ell + 1 + \alpha_\ell + \delta_{k,\ell}}{\beta_\ell + 1} \times \frac{\alpha_k + 1 + \beta_k}{\alpha_k + 1} \binom{\boldsymbol{\alpha} + \boldsymbol{\beta}}{\boldsymbol{\alpha}}. \tag{4.41}$$

Inserting (4.41) into (4.32) with the substitution $\boldsymbol{\eta} = \mathbf{e}_k$ where $k = 1, 2, 3$, leads directly to Algorithm 4.7. Besides, the identity

$$\binom{\boldsymbol{\alpha} + \mathbf{e}_\ell}{\boldsymbol{\alpha}} = \alpha_\ell + 1 \tag{4.42}$$

inserted into (4.31) and (4.33) with $\ell = i + r$ yields Algorithm 4.6 and 4.8, respectively.

Algorithm 4.5: MassSubMat($\mathbf{M}^{\nabla\sigma}, \boldsymbol{\kappa}^{\text{mmt}}$)

Input : B-moments $\{\mu_{\boldsymbol{\alpha}}^{2n+1}(\boldsymbol{\kappa}) : \boldsymbol{\alpha} \in \mathcal{I}^{2n+1}\}$ computed by means of the Stroud conical product rule with $q = n + 2$, stored into array $\boldsymbol{\kappa}^{\text{mmt}} = \{\boldsymbol{\kappa}_{\boldsymbol{\alpha}}^{\text{mmt}} : \boldsymbol{\alpha} \in \mathcal{I}^{2n+1}\}$.

Output: Element mass sub-matrix $\mathbf{M}^{\nabla\sigma}$.

1 `// Same code as in Multinomial (`$\mathbf{M}^{\nabla\sigma}, n, n$`) (cf. Algorithm 4.4), with the line 10 replaced with the lines:`

2 **foreach** $\ell = 1$ **to** 3 **do**

3 $\quad \tilde{w} = (\beta_\ell + 1 + \alpha_\ell)/(\beta_\ell + 1) * w * (n + 1)/(2n + 1)$;

4 $\quad$ **Prod** $= (n + 1) * \tilde{w} \boldsymbol{\kappa}_{\boldsymbol{\alpha}+\boldsymbol{\beta}+\mathbf{e}_\ell}^{\text{mmt}} \mathring{\mathbf{c}}_{\boldsymbol{\beta}+\mathbf{e}_\ell}^{(\boldsymbol{\beta})}$;

5 $\quad$ **foreach** $k = 1$ **to** 3 **do**

6 $\quad\quad \mathbf{M}_{\boldsymbol{\alpha}+\mathbf{e}_k,\boldsymbol{\beta}}^{\nabla\sigma}$ += $\nabla\lambda_k \cdot$ **Prod** ;

7 **deleteColumn** $\mathbf{M}_{[\boldsymbol{\alpha}_0]}^{\nabla\sigma}$;

8 `/*` $\boldsymbol{\sigma}_{\boldsymbol{\alpha}_0}^n$ `is not part of the basis */`;

9 **deleteRow** $\mathbf{M}_{[(n+1,0,0)]}^{\nabla\sigma}$;

10 **deleteRow** $\mathbf{M}_{[(0,n+1,0)]}^{\nabla\sigma}$;

11 **deleteRow** $\mathbf{M}_{[(0,0,n+1)]}^{\nabla\sigma}$;

12 `/* There are no vertex gradient basis functions */`;

13 **Return** $\mathbf{M}^{\nabla\sigma}$;

Algorithm 4.6: MassSubMat($\mathbf{M}^{\nabla W}, \boldsymbol{\kappa}^{\text{mmt}}$)

Input : B-moments $\{\mu_{\boldsymbol{\alpha}}^{n+1}(\boldsymbol{\kappa}) : \boldsymbol{\alpha} \in \mathcal{I}^{n+1}\}$ computed by means of the Stroud conical product rule with $q = n + 2$, stored into array $\boldsymbol{\kappa}^{\text{mmt}} = \{\boldsymbol{\kappa}_{\boldsymbol{\alpha}}^{\text{mmt}} : \boldsymbol{\alpha} \in \mathcal{I}^{n+1}\}$.

Output: Element mass sub-matrix $\mathbf{M}^{\nabla W}$.

1 $\mathbf{M}^{\nabla W} \equiv \mathbf{0}$;

2 **foreach** $\alpha_0 = 0$ **to** n **do**

3 $\quad$ **foreach** $\alpha_1 = 0$ **to** $n - \alpha_0$ **do**

4 $\quad\quad \alpha_3 = n - \alpha_1 - \alpha_2$;

5 $\quad\quad \boldsymbol{\omega} = [\alpha_1 + 1, \alpha_2 + 1, \alpha_3 + 1]$;

6 $\quad\quad$ **for** $i = 1, 2, 3$ **do**

7 $\quad\quad\quad$ **for** $k = 1, 2, 3$ **do**

8 $\quad\quad\quad\quad$ **for** $r = \pm 1$ **do**

9 $\quad\quad\quad\quad\quad \mathbf{M}_{\boldsymbol{\alpha}+\mathbf{e}_k,i}^{\nabla W}$ += $r * \boldsymbol{\omega}_{i+r} * \nabla\lambda_k \cdot \left(\boldsymbol{\kappa}_{\boldsymbol{\alpha}+\mathbf{e}_{i+r}}^{\text{mmt}} \nabla\lambda_{i-r}\right)$;

10 **deleteRow** $\mathbf{M}_{[(n+1,0,0)]}^{\nabla W}$;

11 **deleteRow** $\mathbf{M}_{[(0,n+1,0)]}^{\nabla W}$;

12 **deleteRow** $\mathbf{M}_{[(0,0,n+1)]}^{\nabla W}$;

13 `/* There are no vertex gradient basis functions */`;

14 **Return** $\mathbf{M}^{\nabla W}$;

Algorithm 4.7: MassSubMat($\mathbf{M}^{\sigma\sigma}, \boldsymbol{\kappa}^{\rm mmt}$)

Input : B-moments $\{\mu^{2n+2}_{\boldsymbol{\alpha}}(\boldsymbol{\kappa}) : \boldsymbol{\alpha} \in \mathcal{I}^{2n+2}\}$ computed by means of the Stroud conical product rule with $q = n+2$, stored into array $\boldsymbol{\kappa}^{\rm mmt} = \{\boldsymbol{\kappa}^{\rm mmt}_{\boldsymbol{\alpha}} : \boldsymbol{\alpha} \in \mathcal{I}^{2n+2}\}$.

Output: Element mass sub-matrix $\mathbf{M}^{\sigma\sigma}$.

1 // Same code as in Multinomial ($\mathbf{M}^{\sigma\sigma}, n, n$) (cf. Algorithm 4.4), with the line 10 replaced with the lines:

2 **foreach** $k = 1$ **to** 3 **do**

3 $\quad \tilde{w}_k = (\alpha_k + 1 + \beta_k)/(\alpha_k + 1)$;

4 $\quad \mathbf{sum}_k = \mathbf{0}$;

5 $\quad$ **foreach** $\ell = 1$ **to** 3 **do**

6 $\quad\quad \tilde{w}_\ell = (\beta_\ell + 1 + \alpha_\ell + \delta_{k,\ell})/(\beta_\ell + 1)$;

7 $\quad\quad \mathbf{sum}_k \mathrel{+}= w_\ell * w_3 * \boldsymbol{\kappa}^{\rm mmt}_{\boldsymbol{\alpha}+\mathbf{e}_k+\boldsymbol{\beta}+\mathbf{e}_\ell} \mathring{\mathbf{c}}^{(\boldsymbol{\beta})}_{(\boldsymbol{\beta}+\mathbf{e}_\ell)}$;

8 $\quad \mathbf{M}^{\sigma\sigma}_{\boldsymbol{\alpha},\boldsymbol{\beta}} \mathrel{+}= \tilde{w}_k * \mathring{\mathbf{c}}^{(\boldsymbol{\alpha})}_{\boldsymbol{\alpha}+\mathbf{e}_k} \cdot \mathbf{sum}_k * (n+1)/2/(2n+1)$;

9 **deleteRow** $\mathbf{M}^{\sigma\sigma}_{[\boldsymbol{\alpha}_0]}$;

10 **deleteColumn** $\mathbf{M}^{\sigma\sigma}_{[\boldsymbol{\alpha}_0]}$;

11 /* $\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}_0}$ is not part of the basis */;

12 **Return** $\mathbf{M}^{\sigma\sigma}$;

Algorithm 4.8: MassSubMat($\mathbf{M}^{\sigma W}, \boldsymbol{\kappa}^{\rm mmt}$)

Input : B-moments $\{\mu^{n+2}_{\boldsymbol{\alpha}}(\boldsymbol{\kappa}) : \boldsymbol{\alpha} \in \mathcal{I}^{n+2}\}$ computed by means of the Stroud conical product rule with $q = n+2$, stored into array $\boldsymbol{\kappa}^{\rm mmt} = \{\boldsymbol{\kappa}^{\rm mmt}_{\boldsymbol{\alpha}} : \boldsymbol{\alpha} \in \mathcal{I}^{n+2}\}$.

Output: Element mass sub-matrix $\mathbf{M}^{\sigma W}$.

1 $\mathbf{M}^{\sigma W} \equiv \mathbf{0}$;

2 **foreach** $\alpha_0 = 0$ **to** n **do**

3 $\quad$ **foreach** $\alpha_1 = 0$ **to** $n - \alpha_0$ **do**

4 $\quad\quad \alpha_3 = n - \alpha_1 - \alpha_2$;

5 $\quad\quad \boldsymbol{\omega} = [\alpha_1 + 1, \alpha_2 + 1, \alpha_3 + 1]$;

6 $\quad\quad$ **for** $i = 1, 2, 3$ **do**

7 $\quad\quad\quad$ **for** $k = 1, 2, 3$ **do**

8 $\quad\quad\quad\quad$ **for** $r = \pm 1$ **do**

9 $\quad\quad\quad\quad\quad \mathbf{M}^{\sigma W}_{\boldsymbol{\alpha},i} \mathrel{+}= \dfrac{r}{n+2} * (\boldsymbol{\omega}_{i+r} + \delta_{k,i+r}) * \mathring{\mathbf{c}}_{\boldsymbol{\alpha}+\mathbf{e}_k} \cdot \left(\boldsymbol{\kappa}^{\rm mmt}_{\boldsymbol{\alpha}+\mathbf{e}_k+\mathbf{e}_{i+r}}(\boldsymbol{\kappa}) \nabla \lambda_{i-r}\right)$;

10 **deleteRow** $\mathbf{M}^{\sigma W}_{[\boldsymbol{\alpha}_0]}$;

11 /* $\boldsymbol{\sigma}^n_{\boldsymbol{\alpha}_0}$ is not part of the basis */;

12 **Return** $\mathbf{M}^{\sigma W}$;

Algorithm 4.9: MassSubMat($\mathbf{M}^{WW}, \boldsymbol{\kappa}^{\mathrm{mmt}}$)

Input : B-moments $\{\mu^2_{\boldsymbol{\alpha}}(\boldsymbol{\kappa}) : \ \boldsymbol{\alpha} \in \mathcal{I}^2\}$ computed by means of the Stroud conical product rule with $q = n+2$, stored into array $\boldsymbol{\kappa}^{\mathrm{mmt}} = \{\boldsymbol{\kappa}^{\mathrm{mmt}}_{\boldsymbol{\alpha}} : \ \boldsymbol{\alpha} \in \mathcal{I}^2\}$.

Output: Element mass sub-matrix $\mathbf{M}^{WW}$.

$\mathbf{M}^{WW} \equiv \mathbf{0}$;
for $i, j = 1, 2, 3$ **do**
  for $r, s = \pm 1$ **do**
    if $i + r \neq j + s$ **then**
      $\varepsilon = 1/2$;
    else
      $\varepsilon = 1$;
    $\mathbf{M}^{WW}_{i,j}$ += $r * s * \varepsilon * \nabla\lambda_{i-r} \cdot \left(\boldsymbol{\kappa}^{\mathrm{mmt}}_{\mathbf{e}_{i+r}+\mathbf{e}_{j+s}} \nabla\lambda_{j-s}\right)$;
Return $\mathbf{M}^{WW}$;

Algorithm 4.10: HCurlMassMat

Input : Array $\boldsymbol{\kappa}^{\mathrm{Stroud}}$ corresponding to values of a matrix-valued function $\boldsymbol{\kappa}$ at Stroud nodes.

Output: Blocks $\mathbf{M}^{\nabla\nabla}, \mathbf{M}^{\nabla\sigma}, \mathbf{M}^{\nabla W}, \mathbf{M}^{\sigma\sigma}, \mathbf{M}^{\sigma W}, \mathbf{M}^{WW}$ of the $H(\mathrm{curl})$ mass matrix $\mathbf{M}$.

$q = n + 2$;
$\boldsymbol{\kappa}^{\mathrm{mmt}} =$ `Moment2D` $(\boldsymbol{\kappa}^{\mathrm{Stroud}}, q, 2n+2)$ `/* cf. Algorithm 2.3 */` ;
$\mathbf{M}^{\sigma\sigma} =$ `MassSubMat(`$\mathbf{M}^{\sigma\sigma}, \boldsymbol{\kappa}^{\mathrm{mmt}}$`)` `/* cf. Algorithm 4.7 */` ;
$\boldsymbol{\kappa}^{\mathrm{mmt}} =$ `LowerMoment(`$\boldsymbol{\kappa}^{\mathrm{mmt}}$`)` `/* cf. Algorithm 4.1 */` ;
$\mathbf{M}^{\nabla\sigma} =$ `MassSubMat(`$\mathbf{M}^{\nabla\sigma}, \boldsymbol{\kappa}^{\mathrm{mmt}}$`)` `/* cf. Algorithm 4.5 */` ;
$\boldsymbol{\kappa}^{\mathrm{mmt}} =$ `LowerMoment(`$\boldsymbol{\kappa}^{\mathrm{mmt}}$`)` `/* cf. Algorithm 4.1 */` ;
$\mathbf{M}^{\nabla\nabla} =$ `StiffMat(`$\mathbf{M}^{\nabla\nabla}, n+1, \boldsymbol{\kappa}^{\mathrm{mmt}}$`)` `/* cf. Algorithm 3.21 */` ;
$\boldsymbol{\kappa}^{\mathrm{mmt}} =$ `Moment2D` $(\boldsymbol{\kappa}^{\mathrm{Stroud}}, q, n+2)$ `/* cf. Algorithm 2.3 */` ;
$\mathbf{M}^{\sigma W} =$ `MassSubMat(`$\mathbf{M}^{\sigma W}, \boldsymbol{\kappa}^{\mathrm{mmt}}$`)` `/* cf. Algorithm 4.8 */` ;
$\boldsymbol{\kappa}^{\mathrm{mmt}} =$ `LowerMoment` $(\boldsymbol{\kappa}^{\mathrm{mmt}})$ `/* cf. Algorithm 4.1 */` ;
$\mathbf{M}^{\nabla W} =$ `MassSubMat(`$\mathbf{M}^{\nabla W}, \boldsymbol{\kappa}^{\mathrm{mmt}}$`)` `/* cf. Algorithm 4.6 */` ;
$\boldsymbol{\kappa}^{\mathrm{mmt}} =$ `Moment2D` $(\boldsymbol{\kappa}^{\mathrm{Stroud}}, q, 2)$ `/* cf. Algorithm 2.3 */` ;
$\mathbf{M}^{WW} =$ `MassSubMat(`$\mathbf{M}^{WW}, \boldsymbol{\kappa}^{\mathrm{mmt}}$`)` `/* cf. Algorithm 4.9 */` ;
Return $\mathbf{M}^{\nabla\nabla}, \mathbf{M}^{\nabla\sigma}, \mathbf{M}^{\nabla W}, \mathbf{M}^{\sigma\sigma}, \mathbf{M}^{\sigma W}, \mathbf{M}^{WW}$;

Theorem 4.3.2. *Let $n \in \mathbb{N}$ and $q = n+2$. The $H(\mathrm{curl})$ element mass matrix of degree n in $\mathbb{R}^2$ can be computed and assembled using the Stroud conical quadrature rule with q^2 points in $\mathcal{O}(n^4)$ operations using Algorithm* `HCurlMassMat`.

Proof. Observe that the B-moments are computed by means of `Moment2D` only for the orders $2n+2$, $n+2$ and 2. But then, it follows from Theorem 2.2.4 that the B-moments of order $2n+2$ and $n+2$ can respectively be computed with $\mathcal{O}((2n+2)^3)$ and $\mathcal{O}((n+2)^3)$ operations. Moreover, it follows from the left-hand side of (2.15) that the B-moments of order 2 can be computed using $\mathcal{O}((n+2)^2)$ operations. For the remaining B-moment orders, Lemma 4.2.3 shows that using Algorithm `LowerMoment` for deducing the B-moments of order $\ell-1$ from those of order ℓ, with $\ell = 2n+2,\ 2n+1$ and $n+2$, involves $\mathcal{O}(n^2)$ operations. Thus, the computation of the moments can be done with $\mathcal{O}(n^3)$ complexity.

Assuming that the appropriate B-moments are stored into $\boldsymbol{\kappa}^{\mathrm{mmt}}$, we now consider the routine `MassSubMat(`$\mathbf{M}^{\nabla\sigma}, \boldsymbol{\kappa}^{\mathrm{mmt}}$`)`, that is, Algorithm 4.5. The loop over the pair (α_1, β_1) contains one division and is executed $(n+1)^2$ times. The loop over the pair (α_2, β_2) contains two multiplications and is executed $\binom{n+2}{2}^2$ times. The loop over ℓ contains two divisions, three multiplications, one matrix-vector product and one scalar-vector product, which amounts to eleven operations, and is executed $3 \times \binom{n+2}{2}^2$ times. Finally, the loop over k contains one inner product, and is executed $3^2 \times \binom{n+2}{2}^2$. Therefore, the overall complexity is given by

$$(n+1)^2 + 2\binom{n+2}{2}^2 + 11 \times 3\binom{n+2}{2}^2 + 2 \times 3^2\binom{n+2}{2}^2 = \mathcal{O}(n^4).$$

A similar argument shows that the computation of $\mathbf{M}^{\sigma\sigma}$ can be performed with $\mathcal{O}(n^4)$ operations, whereas the blocks $\mathbf{M}^{\nabla W}$ and $\mathbf{M}^{\sigma W}$ are computed with $\mathcal{O}(n^2)$ complexity. Finally, it is easy to see that the required complexity for Algorithm `MassSubMat(`$\mathbf{M}^{WW}, \boldsymbol{\kappa}^{\mathrm{mmt}}$`)` is of order $\mathcal{O}(1)$. □

Note that, though the mass matrix involves B-moments of six different degrees, the moments are explicitly computed using the routine `Moment2D` only for three

degrees. As shown in Lemma 4.2.3, whenever the B-moments of lower order are needed, it is cheaper to apply one step of `LowerMoment`, rather than calling the routine `Moment2D`.

4.3.3 Evaluation of the Element Stiffness Matrix

In this section, we provide algorithms for the computation of the $H(\mathrm{curl})$ element stiffness matrix.

Recall from (4.6) that, for a given $\boldsymbol{\alpha} \in \mathcal{I}^n$, the set $\mathtt{Star}(\boldsymbol{\alpha})$, consists of

$$\mathcal{I}^n \cap \{\boldsymbol{\alpha} - \mathbf{e}_k + \mathbf{e}_\ell : \ k, \ell = 1, 2, 3, \ k \neq \ell\} \cup \{\boldsymbol{\alpha}\}.$$

The computational details associated with the stiffness matrix are given in Algorithm 4.11.

Theorem 4.3.3. *Let $n \in \mathbb{N}$ and $q = n+2$. The $H(\mathrm{curl})$ element stiffness matrix of degree n in $\mathbb{R}^2$ can be computed using the Stroud conical quadrature rule with q^2 points in $\mathcal{O}(n^4)$ operations using* `HCurlStiffMat`. *More precisely, $\mathbf{S}^{\sigma\sigma}$, $\mathbf{S}^{\sigma W}$ and $\mathbf{S}^{WW}$ can be computed using $\mathcal{O}(n^4)$, $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$ operations, respectively.*

Proof. Using a similar argument as in the proof of Theorem 4.3.2, it is shown that the B-moments of order $2n$ and n can be computed with $\mathcal{O}(n^3)$ operations, whereas that of order 0 involves $\mathcal{O}(n^2)$ operations.

Now recall from the definition of $\mathtt{Star}(\cdot)$ that, for a given $\boldsymbol{\alpha}$, the number of indices contained in $\mathtt{Star}(\boldsymbol{\alpha})$ is (at most) seven. Considering the sub-routine for computing $\mathbf{S}^{\sigma\sigma}$, note that the cost is dominated by the innermost loop which is executed at most $7^2 \times \binom{n+2}{2}^2$, and contains two multiplications. Thus, the overall complexity for computing $\mathbf{S}^{\sigma\sigma}$ is dominated by

$$2 \times 7^2 \times \binom{n+2}{2}^2 = \mathcal{O}(n^4).$$

Similarly, observe that the complexity required by the sub-routine computing

Algorithm 4.11: HCurlStiffMat

Input : Array $\mathbf{A}^{\text{Stroud}}$ corresponding to values of a function A at Stroud nodes.

Output: Blocks $\mathbf{S}^{\sigma\sigma}$, $\mathbf{S}^{\sigma W}$, $\mathbf{S}^{WW}$ of the $H(\text{curl})$ stiffness matrix $\mathbf{S}$.

$q = n + 2$;
/* Stiffness sub-matrix $\mathbf{S}^{\sigma\sigma}$ */;
$\mathbf{A}^{\text{mmt}} = \texttt{Moment2D}\ (\mathbf{A}^{\text{Stroud}}, q, 2n)$ /* cf. Algorithm 2.3 */ ;
// Same code as in Multinomial ($\mathbf{S}^{\sigma\sigma}, n, n$) (cf. Algorithm 4.4), with the line 10 replaced with the lines:
foreach $\mathbf{s}_1 \in \texttt{Star}(\boldsymbol{\alpha})$ **do**
    $r_{\boldsymbol{\alpha},\mathbf{s}_1} = w_3 * \overline{c}_{\boldsymbol{\alpha}}^{(\mathbf{s}_1)}$;
    foreach $\mathbf{s}_2 \in \texttt{Star}(\boldsymbol{\beta})$ **do**
        $\mathbf{S}^{\sigma\sigma}_{\mathbf{s}_1,\mathbf{s}_2} \mathrel{+}= r_{\boldsymbol{\alpha},\mathbf{s}_1} * \mathbf{A}^{\text{mmt}}_{\boldsymbol{\alpha}+\boldsymbol{\beta}} * \overline{c}_{\boldsymbol{\beta}}^{(\mathbf{s}_2)}$;
deleteRow $\mathbf{S}^{\sigma\sigma}_{[\boldsymbol{\alpha}_0]}$;
deleteColumn $\mathbf{S}^{\sigma\sigma}_{[\boldsymbol{\alpha}_0]}$;
/* Stiffness sub-matrix $\mathbf{S}^{\sigma W}$ */;
$\mathbf{A}^{\text{mmt}} = \texttt{Moment2D}\ (\mathbf{A}^{\text{Stroud}}, q, n)$ /* cf. Algorithm 2.3 */ ;
$\mathbf{S}^{\sigma W} \equiv \mathbf{0}$;
for $i = 1$ **to** 3 **do**
    foreach $\boldsymbol{\alpha} \in \mathcal{I}^n$ **do**
        foreach $\mathbf{s} \in \texttt{Star}(\boldsymbol{\alpha})$ **do**
            $\mathbf{S}^{\sigma W}_{\mathbf{s},i} \mathrel{+}= \dfrac{1}{|T|} * \overline{c}_{\boldsymbol{\alpha}}^{(\mathbf{s})} * \mathbf{A}^{\text{mmt}}_{\boldsymbol{\alpha}}$;
deleteRow $\mathbf{S}^{\sigma W}_{[\boldsymbol{\alpha}_0]}$;
/* Stiffness sub-matrix $\mathbf{S}^{WW}$ */;
$\mathbf{A}^{\text{mmt}} = \texttt{Moment2D}\ (\mathbf{A}^{\text{Stroud}}, q, 0)$ /* cf. Algorithm 2.3 */ ;
for $i, j = 0$ **to** 3 **do**
    $\mathbf{S}^{WW}_{i,j} = \dfrac{1}{|T|^2} * \mathbf{A}^{\text{mmt}}_{\mathbf{0}}$;
Return $\mathbf{S}^{\sigma\sigma}$, $\mathbf{S}^{\sigma W}$, $\mathbf{S}^{WW}$;

$\mathbf{S}^{\sigma W}$ is concentrated in the innermost loop over $\mathbf{s}$, which is executed (at most) $3 \times 7 \times \binom{n+2}{2}$ times, and contains two multiplications. Hence, the complexity required for the algorithm is (at most) of order

$$2 \times 3 \times 7 \times \binom{n+2}{2} = \mathcal{O}(n^2).$$

Taking account of the complexity needed for computing the B-moments of order n, we find that $\mathbf{S}^{\sigma W}$ can be computed with $\mathcal{O}(n^3)$ operations.

Finally, it is easy to see that the complexity needed by the sub-routine corresponding to $\mathbf{S}^{WW}$ is of order $\mathcal{O}(1)$, which, together with the complexity required for computation of the B-moment of order 0, concludes the proof. □

4.4 Projection onto the Kernel of the curl Operator

This section shows that the basis described in Theorem 4.1.3 exhibits an explicit kernel splitting, in that the sigma functions are gradient-free. To this purpose, we will make use of an important result in the analysis of Maxwell's equations, *Helmoltz decomposition* [51], which states that any vector field $\mathbf{v} \in \left(L^2(\Omega)\right)^2$, there exist ϕ and φ in $H^1(\Omega)$ such that

$$\mathbf{v} = -\nabla\phi + \mathbf{curl}(\varphi). \tag{4.43}$$

Now observe that the gradient polynomials which are contained in $\mathbb{ND}_n := \mathbb{P}^2_{n+1} \oplus \mathbf{x}^{\perp}\bar{\mathbb{P}}_{n+1}$ consists of $\text{span}\{\nabla B^{n+1}_{\boldsymbol{\alpha}} : \boldsymbol{\alpha} \in \mathcal{I}^{n+1}\}$. It follows from the definition of the basis $\mathcal{B}_n$ given in Theorem 4.1.3 that the only gradient basis functions which are explicitly (but not implicitly) missing from $\mathcal{B}_n$ are those associated with vertex domain points, that is,

$$\{\nabla B^{n+1}_{(n+1)\mathbf{e}_j} : j = 1, 2, 3\}. \tag{4.44}$$

Hence, in order to prove that the sigma basis functions are gradient-free, it suffices to show that the gradient polynomials in (4.44) are orthogonal to the sigma functions (see Theorem 4.4.1).

The ability to identify the shape functions which belong to the kernel of the curl operator can be useful, when dealing with a situation, such as the Maxwell's eigenvalue problem, where the knowledge of the kernel shape functions can speed up the computations. In addition, some preconditioning techniques [52, 18] are based on the ability to identify the functions which are curl-free.

In Equation (4.43), the function φ provides the irrotational component of $\mathbf{v}$, whereas $\nabla\phi$ is termed the *gradient part* of $\mathbf{v}$. In the sequel, the functions $\mathbf{v}$ for which φ is zero in (4.43) are called *gradients*. Moreover, we denote by $\mathbf{B}_\nabla$ the matrix of the gradient operator from $\mathbb{P}_{n+1}$ to $\mathbb{ND}_n$. We next proceed to compute the projection of a function in $\mathbb{ND}_n$ onto the space orthogonal to gradients.

4.4.1 Discrete Projection

Continuous setting Given $\boldsymbol{\phi} \in \left(L^2(\Omega)\right)^2$, we seek $v \in H^1(\Omega)$ satisfying

$$(\boldsymbol{\phi} - \nabla v, \nabla u) = 0, \quad \forall u \in H^1(\Omega).$$

Discrete setting Given $\boldsymbol{\phi} \in \mathbb{ND}_n$, we seek $v \in \mathbb{P}_{n+1}$ satisfying

$$(\boldsymbol{\phi} - \nabla v, \nabla u) = 0, \quad \forall u \in \mathbb{P}_{n+1}.$$

Assuming that the Bernstein polynomials of order $n+1$ are ordered by means of $\{b_\ell : \ \ell = 1, \ldots, \dim(\mathbb{P}_{n+1})\}$, and that $v = \sum_i v_i b_i$, the above equation amounts to

$$(\boldsymbol{\phi}, \nabla b_j) = \sum_i v_i (\nabla b_i, \nabla b_j), \quad j = 1, \ldots, \dim(\mathbb{P}_{n+1}). \tag{4.45}$$

In addition, suppose that the elements of $\mathcal{B}_n$ are ordered by means of $\mathcal{B}_n = \{\mathbf{f}_k : \ k = 1, \ldots, \dim(\mathbb{ND}_n)\}$, and that $\boldsymbol{\phi} = \sum_k c_k \mathbf{f}_k$. Thus, using the fact that $\nabla b_j = \sum_\ell (\mathbf{B}_\nabla \mathbf{e}_j)_\ell \mathbf{f}_\ell$, it follows that, for any j,

$$\begin{aligned}(\boldsymbol{\phi}, \nabla b_j) = \sum_k c_k (\mathbf{f}_k, \nabla b_j) &= \sum_k c_k \sum_\ell (\mathbf{B}_\nabla \mathbf{e}_j)_\ell (\mathbf{f}_k, \mathbf{f}_\ell) \\ &= \sum_k c_k \sum_\ell (\mathbf{B}_\nabla \mathbf{e}_j)_\ell \mathbf{M}_{k,\ell} \\ &= \sum_k c_k (\mathbf{M}\mathbf{B}_\nabla \mathbf{e}_j)_k.\end{aligned}$$

Hence,

$$(\boldsymbol{\phi}, \nabla b_j) = \mathbf{C}^{\mathrm{t}} \mathbf{M} \mathbf{B}_\nabla \mathbf{e}_j = \big(\mathbf{C}^{\mathrm{t}} \mathbf{M} \mathbf{B}_\nabla\big)_{\cdot,j}, \tag{4.46}$$

where $\mathbf{C}$ is the coefficient vector of $\boldsymbol{\phi}$ with respect to the basis $\mathcal{B}_n$, and the notation $\mathbf{A}_{\cdot,j}$ refers to the j^{th} column vector of the matrix $\mathbf{A}$.

Similarly, we find that

$$\begin{aligned}(\nabla b_i, \nabla b_j) = \big(\sum_k (\mathbf{B}_\nabla \mathbf{e}_i)_k \mathbf{f}_k, \sum_\ell (\mathbf{B}_\nabla \mathbf{e}_j)_\ell \mathbf{f}_\ell\big) &= \sum_k (\mathbf{B}_\nabla \mathbf{e}_i)_k \sum_\ell (\mathbf{B}_\nabla \mathbf{e}_j)_\ell (\mathbf{f}_k, \mathbf{f}_\ell) \\ &= \sum_k (\mathbf{B}_\nabla \mathbf{e}_i)_k (\mathbf{M}\mathbf{B}_\nabla \mathbf{e}_j)_k \\ &= \mathbf{e}_i^{\mathrm{t}} \mathbf{B}_\nabla^{\mathrm{t}} \mathbf{M} \mathbf{B}_\nabla \mathbf{e}_j = (\mathbf{B}_\nabla^{\mathrm{t}} \mathbf{M} \mathbf{B}_\nabla)_{i,j}.\end{aligned} \tag{4.47}$$

Inserting (4.46) and (4.47) into (4.45) then yields

$$\big(\mathbf{C}^{\mathrm{t}} \mathbf{M} \mathbf{B}_\nabla\big)_{\cdot,j} = \sum_i v_i \big(\mathbf{B}_\nabla^{\mathrm{t}} \mathbf{M} \mathbf{B}_\nabla\big)_{i,j} = \big(\mathbf{V}^{\mathrm{t}} \mathbf{B}_\nabla^{\mathrm{t}} \mathbf{M} \mathbf{B}_\nabla\big)_{\cdot,j},$$

where $\mathbf{V}$ represents the coefficient vector of v with respect to the basis $\{b_i : \ i = 1, \ldots, \dim(\mathbb{P}_{n+1})\}$. It follows that

$$\mathbf{B}_\nabla^{\mathrm{t}} \mathbf{M} \mathbf{C} = \mathbf{B}_\nabla^{\mathrm{t}} \mathbf{M} \mathbf{B}_\nabla \mathbf{V},$$

so that

$$\mathbf{V} = \left(\mathbf{B}_\nabla^{\mathrm{t}}\mathbf{M}\mathbf{B}_\nabla\right)^{-1}\mathbf{B}_\nabla^{\mathrm{t}}\mathbf{M}\mathbf{C}.$$

In particular, the coefficient vector of ∇v with respect to the basis $\mathcal{B}_n$ is given by

$$\mathbf{B}_\nabla\mathbf{V} = \mathbf{B}_\nabla\left(\mathbf{B}_\nabla^{\mathrm{t}}\mathbf{M}\mathbf{B}_\nabla\right)^{-1}\mathbf{B}_\nabla^{\mathrm{t}}\mathbf{M}\mathbf{C}.$$

Hence, the projection from $\mathbb{ND}_n$ to the space orthogonal to $\nabla\mathbb{P}_{n+1}$ is represented by the matrix

$$\mathbf{I} - \mathbf{B}_\nabla\left(\mathbf{B}_\nabla^{\mathrm{t}}\mathbf{M}\mathbf{B}_\nabla\right)^{-1}\mathbf{B}_\nabla^{\mathrm{t}}\mathbf{M}, \tag{4.48}$$

where $\mathbf{I}$ denotes the identity matrix.

4.4.2 Gradient Matrix

We now proceed to compute $\mathbf{B}_\nabla$ which is the matrix of the gradient operator from $\mathbb{P}_{n+1}$ to $\mathbb{ND}_n$. To this end, recall that the only gradient polynomials which are missing from the definition of the basis $\mathcal{B}_n$ described in Theorem 4.1.3 are the gradients of the Bernstein polynomials associated with vertex domain points, or *vertex gradients*. Thus, in order to find the matrix $\mathbf{B}_\nabla$, it suffices to find the expression of the vertex gradients with respect to the basis $\mathcal{B}_n$, as given in the next theorem:

Theorem 4.4.1. *The gradients of the polynomials of order* 1 *are spanned by the Whitney functions, by means of:*

$$\begin{aligned}
\nabla\lambda_1 &= \boldsymbol{\omega}_2 - \boldsymbol{\omega}_3,\\
\nabla\lambda_2 &= \boldsymbol{\omega}_3 - \boldsymbol{\omega}_1,\\
\nabla\lambda_3 &= \boldsymbol{\omega}_1 - \boldsymbol{\omega}_2.
\end{aligned} \tag{4.49}$$

Moreover, with $n \geq 1$, it holds that:

$$\nabla B^{n+1}_{(n+1)\mathbf{e}_j} = - \sum_{\substack{\boldsymbol{\alpha}\in\mathcal{I}^{n+1} \\ 0<\alpha_j<n+1}} \frac{\alpha_j}{n+1} \nabla B^{n+1}_{\boldsymbol{\alpha}} + (\boldsymbol{\omega}_{j+1} - \boldsymbol{\omega}_{j-1}), \quad j = 1, 2, 3. \tag{4.50}$$

Proof. The results given in (4.49) are well-known, and can be easily checked.

We now proceed to prove (4.50). The partition of unity property of the Bernstein polynomials implies that

$$\lambda_j = \sum_{\boldsymbol{\alpha}\in\mathcal{I}^n} \lambda_j B^n_{\boldsymbol{\alpha}} = \sum_{\boldsymbol{\alpha}\in\mathcal{I}^n} \frac{\alpha_j + 1}{n+1} B^{n+1}_{\boldsymbol{\alpha}+\mathbf{e}_j}.$$

Making the change of variable $\boldsymbol{\alpha}' = \boldsymbol{\alpha} + \mathbf{e}_j$ then gives

$$\lambda_j = \sum_{\boldsymbol{\alpha}\in\mathcal{I}^{n+1}} \frac{\alpha_j}{n+1} B^{n+1}_{\boldsymbol{\alpha}} = B^{n+1}_{(n+1)\mathbf{e}_j} + \sum_{\substack{\boldsymbol{\alpha}\in\mathcal{I}^{n+1} \\ 0\leq\alpha_j<n+1}} \frac{\alpha_j}{n+1} B^{n+1}_{\boldsymbol{\alpha}}.$$

The gradient of the above equation reads

$$\begin{aligned} \nabla B^{n+1}_{(n+1)\mathbf{e}_j} &= - \sum_{\substack{\boldsymbol{\alpha}\in\mathcal{I}^{n+1} \\ 0\leq\alpha_j<n+1}} \frac{\alpha_j}{n+1} \nabla B^{n+1}_{\boldsymbol{\alpha}} + \nabla\lambda_j \\ &= - \sum_{\substack{\boldsymbol{\alpha}\in\mathcal{I}^{n+1} \\ 0<\alpha_j<n+1}} \frac{\alpha_j}{n+1} \nabla B^{n+1}_{\boldsymbol{\alpha}} + \nabla\lambda_j, \quad j = 1, 2, 3. \end{aligned} \tag{4.51}$$

Inserting (4.49) into (4.51) yields the desired result.

□

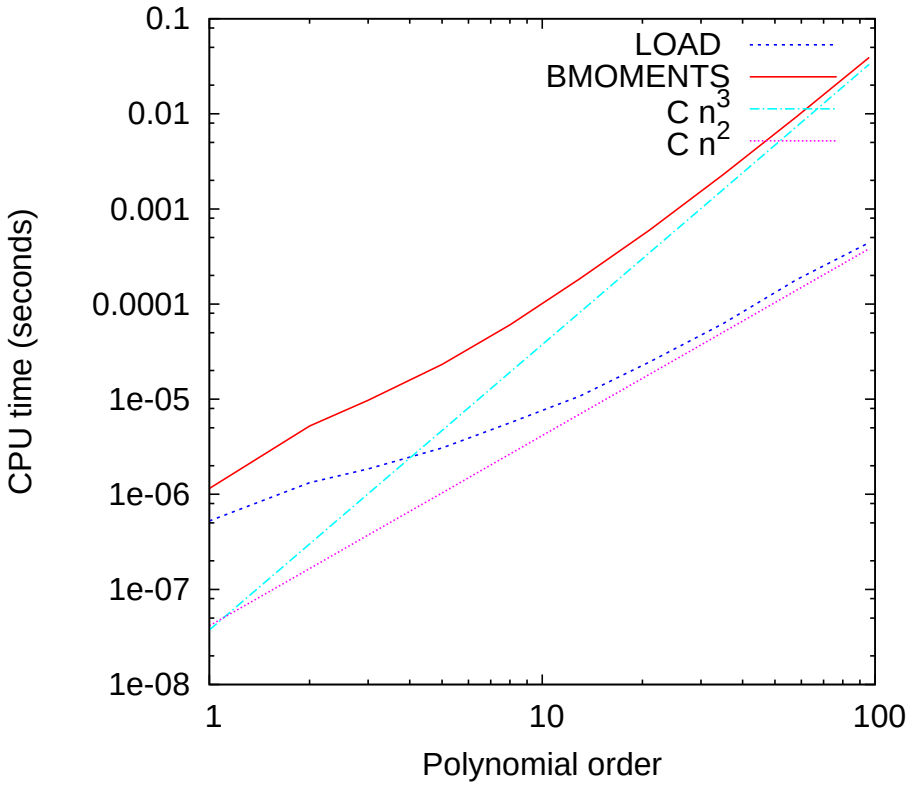
Recall that the Whitney edge functions are divergence-free, so that the projection onto the space orthogonal to gradients is only needed with higher order Nédélec spaces, that is, with $n \geq 1$. The rectangular $\dim(\mathbb{ND}_n)$ by $\dim(\mathbb{P}_{n+1})$ matrix $\mathbf{B}_\nabla$ is then such that the first $\dim(\mathbb{P}_{n+1})$ columns, which correspond to gradients, almost form the identity matrix, whereas three of these columns, which are associated with vertex gradients, can be deduced from (4.50). In addition, the rows of $\mathbf{B}_\nabla$ associated with the sigma basis functions only have zero en-

tries, which, by virtue of Helmoltz decomposition, implies that the sigma basis functions are purely solenoidal.

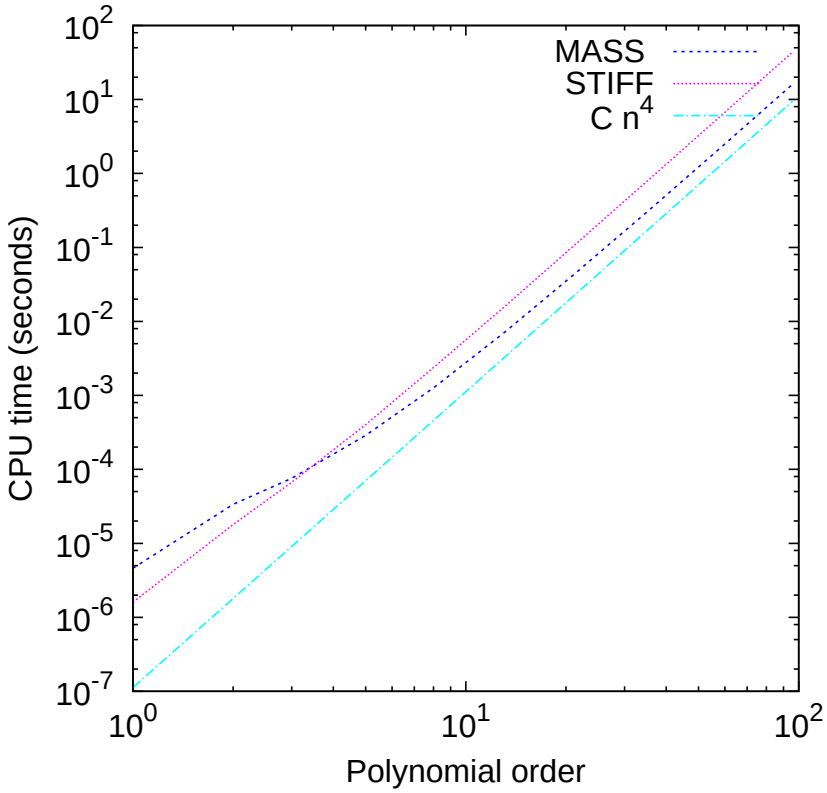
4.5 Numerical Results

4.5.1 CPU Timings

This section focuses on the CPU timings obtained after running the presented algorithms for the element load vector and the element matrices. Regarding the load vector, observe from the proof of Theorem 4.3.1 that, assuming that the appropriate B-moments are given, the computation of the load vector only requires $\mathcal{O}(n^2)$ operations. Thus, the computation of the B-moments clearly dominates the cost, and it is worth distinguishing the CPU time required for computing the load vector from that required for computing the B-moments. On Figure 4.2(a), "LOAD" and "BMOMENTS" respectively refer to the CPU timings corresponding to the element load vector and the B-moments. The associated plots are respectively compared to the curves of Cn^3 and Cn^2 with C denoting a generic constant. As illustrated on the graph, the growth of the CPU time is consistent with the predicted optimal complexity results given in Theorem 4.3.1. On Figure 4.2(b), "MASS" and "STIFF" respectively refer to the CPU timings associated with the element mass and stiffness matrices. We observe that, in both cases, the growth of the CPU time is consistent with the optimal complexity results given in Theorem 4.3.2 and Theorem 4.3.3.



(a) Element load vector



(b) Element mass and stiffness matrices

Figure 4.2: CPU timing

4.5.2 Eigenvalue Problem

Consider the curl - curl problem given by:

$$\begin{aligned} \mathbf{curl}(\operatorname{curl}\mathbf{u}) &= \lambda^2\mathbf{u} \quad \text{in } \Omega, \\ \boldsymbol{\tau}\cdot\mathbf{u} &= 0 \quad \text{on } \partial\Omega. \end{aligned}$$

The weak form corresponding to the above equations is to find $\mathbf{u} \in H_0(\text{curl})$ and $\lambda \in \mathbb{R}$ such that

$$(\operatorname{curl}\mathbf{u}, \operatorname{curl}\mathbf{v}) = \lambda^2(\mathbf{u}, \mathbf{v}), \quad \mathbf{v} \in H_0(\text{curl}). \tag{4.52}$$

In the case where $\Omega = [0,1]^2$, it is well-known [28] that the corresponding eigenfunctions and associated eigenvalues are respectively given by the sets $\{\mathbf{u}_{k,\ell} : k,\ell \in \mathbb{Z}_+\}$ and $\{\lambda_{k,\ell} : \ k,\ell \in \mathbb{Z}_+\}$ defined by

$$\mathbf{u}_{k,\ell}(x,y) := \begin{pmatrix} k\sin(k\pi x)\cos(\ell\pi x) \\ \ell\sin(\ell\pi y)\cos(k\pi x) \end{pmatrix}^{\perp} \tag{4.53}$$

and

$$\lambda^2_{k,\ell} := \pi^2(k^2+\ell^2), \quad k,\ell \in \mathbb{Z}_+. \tag{4.54}$$

Observe that the oscillations of $\mathbf{u}_{k,\ell}$ increase with k and ℓ. Thus, higher approximation order is required in order to approximate larger eigenvalues.

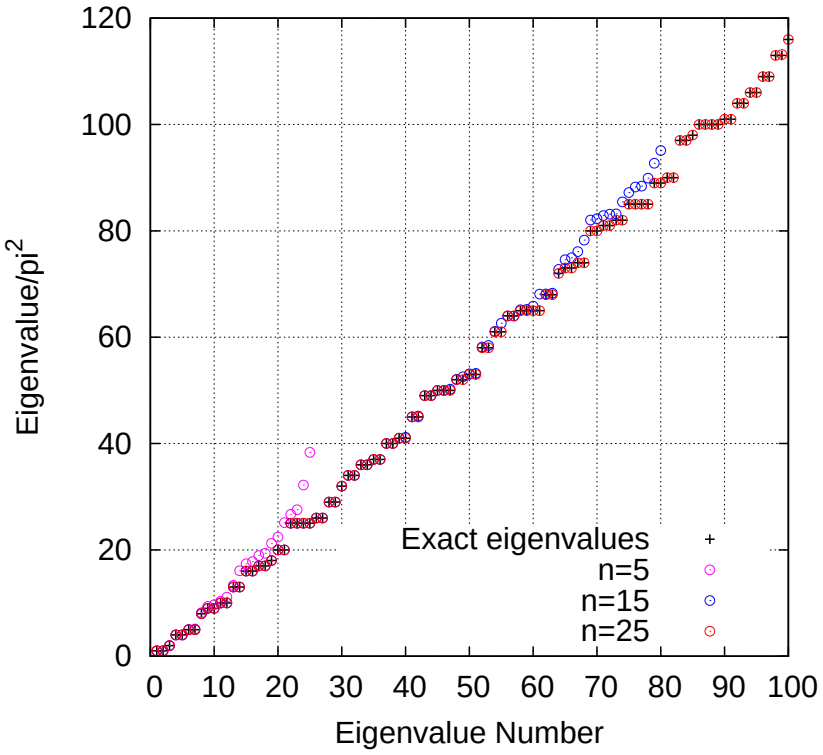
The Galerkin finite element discretization of (4.52) results in a generalized eigenvalue problem which can be solved by means of any general-purpose eigensolver. However, in order to avoid computing the multiple zero eigenvalues, we use the algorithms presented in Appendix C. It should be mentioned that the non-zero eigenpairs produced by the methods proposed in Appendix C, and those returned by numpy.linalg.eigh on Python were found to be the same, up to machine precision. Considering the mesh consisting of the two triangles separated

by one diagonal of the square, we obtain Figure 4.3. Figure 4.3(a) shows the obtained eigenvalues with $n = 5$, 15 and 25. With $n = 5$, only the first 10 eigenvalues are approximated correctly, whereas with $n = 15$, up to 54 eigenvalues are well-approximated. With $n = 25$, the first 100 eigenvalues are computed accurately. Hence, provided that the approximation order is sufficient, each eigenvalue is represented according to its correct multiplicity. For instance, with $n = 15$ or $n = 25$, the eigenvalue $\lambda^2 = 25\pi^2$ is repeated four times, which is consistent with (4.54). Moreover, Figure 4.3(b) shows the convergence rate of the discrete eigenvalues towards the exact ones. Motivated by the following error bound [13, Equation 26]

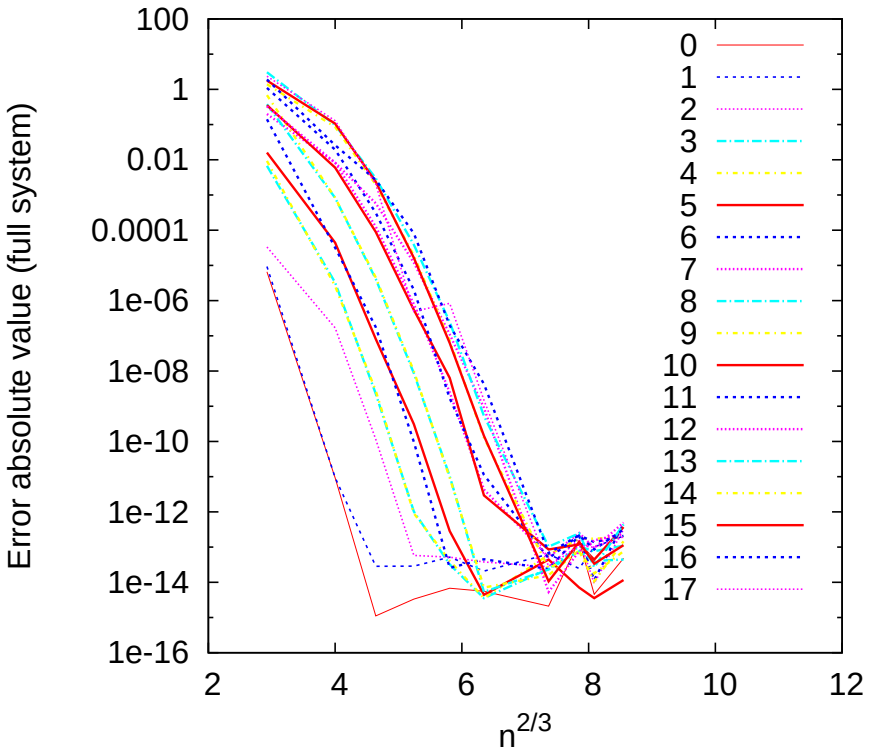
$$|\lambda - \lambda_N| \leq C \exp(-bN^{1/3}),$$

where N is the number of degrees of freedom whereas C and b are positive constants, we have chosen to represent $n^{2/3}$, and not n, on the x-axis, with n denoting the polynomial order used.

Observe in particular that the first eigenvalues are well-approximated by lower order approximation, whereas the approximation of the larger eigenvalues start to improve when the polynomial order is increased.



(a) Obtained Eigenvalues



(b) Approximation Error

Figure 4.3: Eigenvalues associated with (4.52).

Chapter 5

Enhanced Edge Elements

This chapter gives a brief report on the work I have done for a short Knowledge Transfer Partnership (KTP) project sponsored by Cobham Technical Services and the Technology Strategy Board (UK) (see [2]). Cobham Technical Services – Vector Fields Software (Kidlington, UK) (see [1]) is a leading provider in computational electromagnetics software. The KTP project involved some research and development (R&D) work on edge finite elements.

As mentioned in the previous chapters, edge elements provide the $H(\text{curl})$-conforming finite element discretization of Maxwell's equations. In a typical finite element implementation, the shape functions are defined on the *reference element*. The shape functions on the physical elements are then computed from appropriate local-to-global mappings. A mesh is termed *affine* when each element of the mesh is obtained by an affine transformation of the reference element. On such meshes, Nédélec elements have been shown to provide optimal convergence rates $\mathcal{O}(h^{n+1})$ in $H(\text{curl})$-norm [38]. However, a lower order of convergence is obtained when using more general non-affine meshes [17, 43]. In particular, for the lowest-order Nédélec space $\mathbb{ND}_0$, there is no convergence in $H(\text{curl})$-norm.

The purpose of the KTP project was to investigate this convergence degeneration, and implement a remedy suggested in [43, 25] using the `C++` programming language.

This chapter is organized as follows. Section 5.1 revisits the theoretical conditions for optimal accuracy in $H(\mathrm{curl})$-norm, including the rationale behind recently developed edge elements [43, 25] designed to preserve optimal accuracy on non-affine meshes. Section 5.2 details an analytical example illustrating the defect of standard Nédélec basis on a non-affine hexahedron, in comparison with new edge shape functions discussed in Section 5.1.2.

5.1 Completeness Condition

This section recalls the $H(\mathrm{curl})$-conforming transforms which are used in the finite element computations involving edge elements. A condition for optimal convergence rate is also given for the polynomial space defined on the reference element.

5.1.1 $H(\mathrm{curl})$-Conforming Transformations

Finite element spaces are typically defined on a reference element. The corresponding space on the physical element are generated by applying appropriate local-to-global mappings. For the sake of completeness, this section recalls the transformations used for $H(\mathrm{curl})$-conforming elements.

Assuming that the reference element $\hat{K}$ and the physical element K are related by a diffeomorphism Φ by means of

$$\mathbf{x} = \Phi(\hat{\mathbf{x}}), \quad \hat{\mathbf{x}} \in K,$$

then the vector fields are related by the $H(\mathrm{curl})$-conforming transform [17, Equation (1.3)]

$$\mathbf{A}(\mathbf{x}) = \mathrm{d}\Phi(\hat{\mathbf{x}})^{-\mathrm{tr}}\hat{\mathbf{A}}(\hat{\mathbf{x}}), \tag{5.1}$$

where $\mathrm{d}\Phi(\cdot)$ denotes the Jacobian matrix of the transformation. Hence, for a

given order r, the finite element space on the physical element K is defined as

$$P_r(K) := \{\mathbf{A} \text{ such that } \mathrm{d}\Phi^{\mathrm{tr}}\mathbf{A} \circ \Phi \in \hat{P}_r(\hat{K})\}, \tag{5.2}$$

where $\hat{P}_r(\hat{K})$ denotes the finite element space defined on the reference element.

Applying the curl operator to both sides of (5.1), and using the chain rules yields [65, Corollary 3.58]

$$\mathbf{curl}\ \mathbf{A}(\mathbf{x}) = \mathbf{P}_\Phi(\hat{\mathbf{x}})\,\hat{\mathbf{curl}}\ \hat{\mathbf{A}}(\hat{\mathbf{x}}) := |\mathrm{d}\Phi(\hat{\mathbf{x}})|^{-1}\mathrm{d}\Phi(\hat{\mathbf{x}})\,\hat{\mathbf{curl}}\ \hat{\mathbf{A}}(\hat{\mathbf{x}}), \tag{5.3}$$

where $|\mathrm{d}\Phi(\cdot)|$ represents the Jacobian of the transformation.

With h denoting the mesh size, the following condition ensures a convergence rate of order $\mathcal{O}(h^{r+1})$ in $H(\mathrm{curl})$-norm for edge elements of order r [25, Definition 2].

Completeness Condition. *Let $r \geq 0$, and $\hat{P}_r(\hat{K})$ the edge finite element space of order r on the reference element. Then $\hat{P}_r(\hat{K})$ is termed* optimal *if, for any element K, it holds that*

$$\mathbb{ND}_r \subseteq P_r(K),$$

where $\mathbb{ND}_r$ refers to the Nédélec space of order r and $P_r(K)$ is defined in (5.2).

As discussed in the following section, standard Nédélec elements need to be adjusted in order to meet the above condition on non-affine meshes.

5.1.2 Extended Edge Elements

In [43] an improved lowest-order edge basis which is complete for arbitrary trilinear hexahedra is presented. In addition to hexahedral elements, Bergot and Duruflé introduce in [25] prismatic and pyramidal elements which satisfy the completeness condition, for arbitrary polynomial order of approximation. In order to maintain the optimal accuracy of edge finite elements on non-affine meshes, both above-mentioned references involve the addition of supplementary functions to

the standard Nédélec basis. This section explains the rationale behind the design of the additional shape functions.

The key idea used in [43] is to determine which functions need to be added to the reference cube, for constants to belong to the span of the curl of the basis functions on the physical hexahedron, using the correspondence (5.3) between $\mathbf{curl}\ \mathbf{A}(\mathbf{x})$ and $\hat{\mathbf{curl}}\ \hat{\mathbf{A}}(\hat{\mathbf{x}})$, that is,

$$\mathbf{curl}\ \mathbf{A}(\mathbf{x}) = \mathbf{P}_\Phi(\hat{\mathbf{x}})\,\hat{\mathbf{curl}}\ \hat{\mathbf{A}}(\mathbf{x})$$

or, equivalently,

$$\hat{\mathbf{curl}}\ \hat{\mathbf{A}}(\hat{\mathbf{x}}) = \mathbf{P}_\Phi^{-1}(\hat{\mathbf{x}})\,\mathbf{curl}\ \mathbf{A}(\mathbf{x}) = |\mathrm{d}\Phi(\hat{\mathbf{x}})|(\mathrm{d}\Phi)^{-1}(\hat{\mathbf{x}})\,\mathbf{curl}\ \mathbf{A}(\mathbf{x}). \tag{5.4}$$

More precisely, starting with a general trilinear map $\Phi : \hat{\mathbf{K}} \to \mathbb{R}^3$, the map $\mathbf{P}_\Phi^{-1}(\hat{\mathbf{x}})$ is applied to linearly independent constant vectors, as in

$$\hat{\mathbf{curl}}\ \hat{\mathbf{A}}(\hat{\mathbf{x}}) = \mathbf{P}_\Phi^{-1}(\hat{\mathbf{x}})\mathbf{B}(\mathbf{x}),$$

where $\mathbf{B}(\mathbf{x})$ is successively set equal to $(1,0,0)^{\mathrm{tr}}$, $(0,1,0)^{\mathrm{tr}}$ and $(0,0,1)^{\mathrm{tr}}$. This gives rise to a 20-dimensional linear space, which, together with the fact that the curl of the Nédélec shape functions span a linear space of dimension 5 (see [43, p.129]), implies that 15 additional basis functions are needed for the curl of the basis functions to contain constants.

In contrast with Falk *et al*, Bergot and Duruflé [25] use the $H(\mathrm{curl})$-conforming transform, as given in (5.1), that is,

$$\mathbf{A}(\mathbf{x}) = \mathrm{d}\Phi(\hat{\mathbf{x}})^{-\mathrm{tr}}\hat{\mathbf{A}}(\hat{\mathbf{x}}),$$

with Φ defined as a generic linear isoparametric map. The additional functions that need to be included on the reference element are deduced by looping the field $\mathbf{A}$ over all Nédélec monomials on the physical element. In the lowest-order case,

this process results in additional degrees of freedom associated with the elements' faces, as displayed in Figure 5.1.

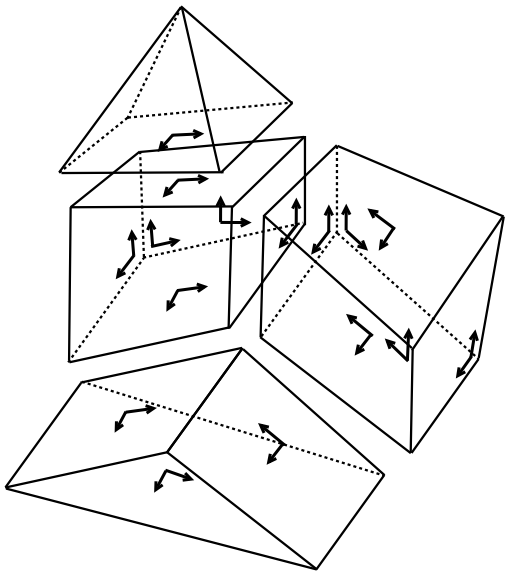


Figure 5.1: Additional Face-Based Degrees of Freedom

Note that, in addition to the face-based degrees of freedom, each hexahedron has three internal degrees of freedom.

5.2 Hexahedron: A Particular Example

This section illustrates the fact that the Nédélec edge shape functions do not satisfy the completeness condition on a simple non-affine hexahedron. In contrast, the new elements presented in [43, 25] are shown to be complete on the non-affine hexahedron. Finite element simulations involving the enhanced edge elements are also included.

5.2.1 Nédélec Basis Functions

The edge and node numbering is as in Figure 5.2.

For $i = 1, \ldots, 12$, we denote by $\mathbf{w}_i$ the Whitney edge basis function associated with the i^{th} edge. In particular, on the reference hexahedron $[-1, 1]^3$, the edge

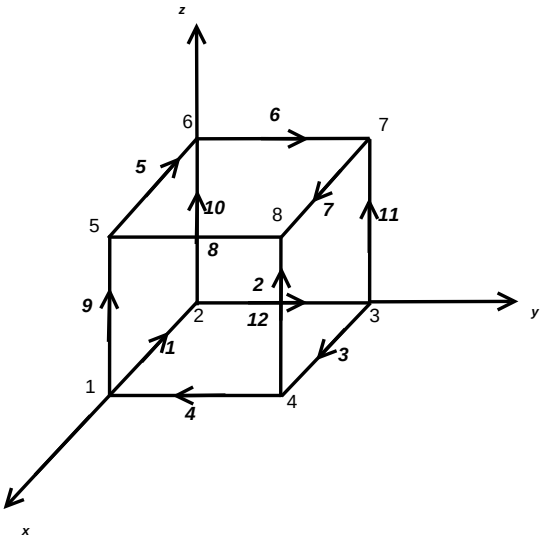


Figure 5.2: Reference Hexahedron

basis functions are defined by:

$$\hat{\mathbf{w}}_1(\hat{x},\hat{y},\hat{z}) := (-(1-\hat{y})(1-\hat{z})/8, 0, 0)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_2(\hat{x},\hat{y},\hat{z}) := (0, (1-\hat{x})(1-\hat{z})/8, 0)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_3(\hat{x},\hat{y},\hat{z}) := ((1+\hat{y})(1-\hat{z})/8, 0, 0)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_4(\hat{x},\hat{y},\hat{z}) := (0, -(1+\hat{x})(1-\hat{z})/8, 0)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_5(\hat{x},\hat{y},\hat{z}) := (-(1-\hat{y})(1+\hat{z})/8, 0, 0)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_6(\hat{x},\hat{y},\hat{z}) := (0, (1-\hat{x})(1+\hat{z})/8, 0)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_7(\hat{x},\hat{y},\hat{z}) := ((1+\hat{y})(1+\hat{z})/8, 0, 0)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_8(\hat{x},\hat{y},\hat{z}) := (0, -(1+\hat{x})(1+\hat{z})/8, 0)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_9(\hat{x},\hat{y},\hat{z}) := (0, 0, (1+\hat{x})(1-\hat{y})/8)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_{10}(\hat{x},\hat{y},\hat{z}) := (0, 0, (1-\hat{x})(1-\hat{y})/8)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_{11}(\hat{x},\hat{y},\hat{z}) := (0, 0, (1-\hat{x})(1+\hat{y})/8)^{\mathrm{tr}},$$
$$\hat{\mathbf{w}}_{12}(\hat{x},\hat{y},\hat{z}) := (0, 0, (1+\hat{x})(1+\hat{y})/8)^{\mathrm{tr}},$$

so that, on each edge $\hat{\gamma}_i$, it holds that

$$\int_{\hat{\gamma}_i} \hat{\mathbf{w}}_j \cdot \hat{\tau}_i \mathrm{d}s = \begin{cases} 1 \text{ if } i = j, \\ 0 \text{ otherwise,} \end{cases}$$

where $\hat{\tau}_i$ is the unit tangent vector along the edge $\hat{\gamma}_i$.

We consider the non-affine hexahedron K with vertices given by:

$$\begin{aligned} &\mathbf{x}_1 = (3/2, 0, 0)^{\mathrm{tr}},\ \mathbf{x}_2 = (0, 0, 0)^{\mathrm{tr}},\ \mathbf{x}_3 = (0, 1, 0)^{\mathrm{tr}},\ \mathbf{x}_4 = (1/2, 1, 0)^{\mathrm{tr}}, \\ &\mathbf{x}_5 = (3/2, 0, 1)^{\mathrm{tr}},\ \mathbf{x}_6 = (0, 0, 1)^{\mathrm{tr}},\ \mathbf{x}_7 = (0, 1, 1)^{\mathrm{tr}},\ \mathbf{x}_8 = (1/2, 1, 1)^{\mathrm{tr}}, \end{aligned} \tag{5.5}$$

as illustrated on Figure 5.3.

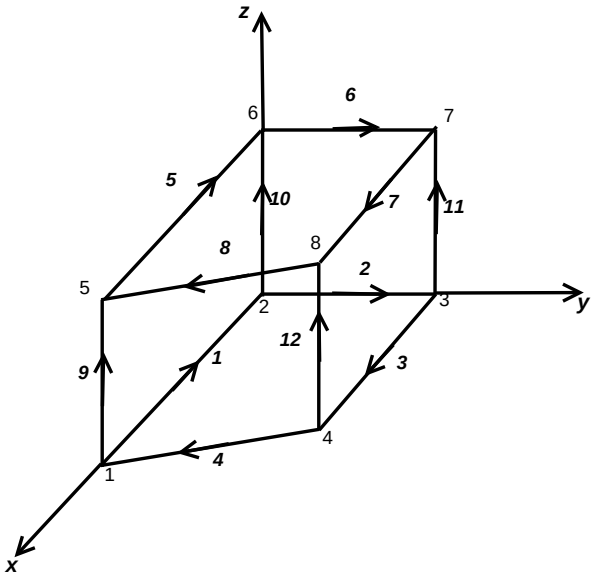


Figure 5.3: Distorted Hexahedron

Let $\mathbf{A} : K \to \mathbb{R}$ be the function defined as

$$\mathbf{A}(\mathbf{x}) := (0, x, 0)^{\mathrm{tr}}, \tag{5.6}$$

so that

$$(\mathbf{curl}\ \mathbf{A})(\mathbf{x}) := \nabla \times \mathbf{A}(\mathbf{x}) = (0, 0, 1)^{\mathrm{tr}}.$$

We now want to express **curl A** as a linear combination of the curl of the edge

basis functions. To this purpose, observe from Figure 5.3 and the expression of $\mathbf{A}(\mathbf{x})$ that the only non-zero degrees of freedom of $\mathbf{A}$ with respect to the lowest order Nédélec elements consist of its tangential components along the edges γ_4 and γ_8. Thus, in order to find the best representation of **curl A** by the curl of the edge shape shape functions, it suffices to compute the L^2-orthogonal projection of **curl A** with respect to the curl of the space $\mathrm{span}(\mathbf{w}_1, \mathbf{w}_4)$. To this purpose, it suffices to compute $\hat{\mathbf{A}}$ which is obtained from the $H(\mathrm{curl})$-conforming transform (5.1), that is,

$$\hat{\mathbf{A}}(\hat{\mathbf{x}}) := \mathrm{d}\Phi(\hat{\mathbf{x}})^{\mathrm{tr}}\mathbf{A}(\mathbf{x}) = \mathrm{d}\Phi(\hat{\mathbf{x}})^{\mathrm{tr}}\mathbf{A}(\Phi(\hat{\mathbf{x}})), \tag{5.7}$$

and find the coefficients α_1 and α_4 which solve the two-by-two system

$$\mathbf{P}\boldsymbol{\alpha} = \mathbf{f}_{\hat{\mathbf{A}}}, \tag{5.8}$$

where

$$\mathbf{P}_{1,1} := \langle \hat{\mathbf{w}}_4, \hat{\mathbf{w}}_4 \rangle_{\hat{K}} \quad \mathbf{P}_{1,2} := \langle \hat{\mathbf{w}}_4, \hat{\mathbf{w}}_8 \rangle_{\hat{K}}$$
$$\mathbf{P}_{2,1} := \langle \hat{\mathbf{w}}_8, \hat{\mathbf{w}}_4 \rangle_{\hat{K}}, \quad \mathbf{P}_{2,2} := \langle \hat{\mathbf{w}}_8, \hat{\mathbf{w}}_8 \rangle_{\hat{K}},$$

and

$$f_{\hat{\mathbf{A}},1} := \langle \hat{\mathbf{A}}, \hat{\mathbf{w}}_4 \rangle_{\hat{K}}$$
$$f_{\hat{\mathbf{A}},2} := \langle \hat{\mathbf{A}}, \hat{\mathbf{w}}_8 \rangle_{\hat{K}}$$

where the inner product $\langle \cdot, \cdot \rangle_{\hat{K}}$ is defined by means of

$$\langle \mathbf{v}, \mathbf{w} \rangle_{\hat{K}} := \int_{\hat{K}} \big(\hat{\mathbf{curl}}\ \mathbf{v}(\hat{\mathbf{x}})\big) \cdot \big(\hat{\mathbf{curl}}\ \mathbf{w}(\hat{\mathbf{x}})\big) \mathrm{d}\hat{\mathbf{x}}, \tag{5.9}$$

assuming that $\mathbf{v}$ and $\mathbf{w}$ are sufficiently regular so that the right-hand side of the above equation is well-defined.

We now proceed to compute $\hat{\mathbf{A}}(\hat{\mathbf{x}})$, as defined in (5.7). We set Φ to be the

linear isoparametric transformation [58, Section 5.5], that is,

$$\Phi(\hat{\mathbf{x}})` := \sum_{i=1}^{8} \hat{N}_i(\hat{\mathbf{x}})\mathbf{x}_i,$$

where $\hat{N}_i$ are the nodal shape functions on the reference cube $[-1,1]^3$ [5, Equation (11.2)] and $\mathbf{x}_i$ are given in (5.5), for $i =, 1, \ldots, 8$. Expanding the above expression gives

$$\Phi(\hat{x},\hat{y},\hat{z}) = (\frac{(\hat{x}+1)(2-\hat{y})}{4}, \frac{\hat{y}+1}{2}, \frac{\hat{z}+1}{2})^{\mathrm{tr}}, \tag{5.10}$$

so that

$$\mathrm{d}\Phi(\hat{x},\hat{y},\hat{z}) = \begin{pmatrix} (2-\hat{y})/4 & -(1+\hat{x})/4 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & 1/2 \end{pmatrix}, \tag{5.11}$$

and

$$|\mathrm{d}\Phi(\hat{x},\hat{y},\hat{z})| = \frac{2-\hat{y}}{16}. \tag{5.12}$$

Inserting (5.11) and (5.10) into (5.7) yields

$$\hat{\mathbf{A}}(\hat{\mathbf{x}}) = (0, (-\hat{x}-1)(\hat{y}-2)/8, 0)^{\mathrm{tr}}. \tag{5.13}$$

Using the above expression of $\hat{\mathbf{A}}$ for solving the system (5.8) gives

$$\alpha_1 = \alpha_8 = -1.$$

Hence, the representation of $\hat{\mathbf{curl}}\ \hat{\mathbf{A}}$ by the lowest order Nédélec basis is given by

$$\sum_{i=1}^{12} \alpha_i\, \hat{\mathbf{curl}}\ \hat{\mathbf{w}}_i(\hat{\mathbf{x}}) = -\hat{\mathbf{curl}}\ \hat{\mathbf{w}}_4(\hat{\mathbf{x}}) - \hat{\mathbf{curl}}\ \hat{\mathbf{w}}_8(\hat{\mathbf{x}}), \tag{5.14}$$

Inserting (5.12) into (5.3) amounts to

$$\mathbf{curl}\ \mathbf{A}(\mathbf{x}) = \frac{16}{2-\hat{y}}\mathrm{d}\Phi(\hat{\mathbf{x}})\,\hat{\mathbf{curl}}\ \hat{\mathbf{A}}(\hat{\mathbf{x}}).$$

Hence, it follows from (5.14) that an attempt to represent the constant field $(0,0,1)^{\mathrm{tr}} = (\mathbf{curl}\ \mathbf{A})(\mathbf{x})$ gives

$$\begin{aligned}
&-\frac{16}{2-\hat{y}}\mathrm{d}\Phi(\hat{\mathbf{x}})\big(\hat{\mathbf{curl}}\ \hat{\mathbf{w}}_4(\hat{\mathbf{x}}) + \hat{\mathbf{curl}}\ \hat{\mathbf{w}}_8(\hat{\mathbf{x}})\big) \\
&\qquad = -\frac{16}{2-\hat{y}}\mathrm{d}\Phi(\hat{\mathbf{x}})\Big[\begin{pmatrix} -(1+\hat{x})/8 \\ 0 \\ -(1-\hat{z})/8 \end{pmatrix} + \begin{pmatrix} (1+\hat{x})/8 \\ 0 \\ -(1+\hat{z})/8 \end{pmatrix}\Big] \\
&\qquad = \frac{4}{2-\hat{y}}\mathrm{d}\Phi(\hat{\mathbf{x}})(0,0,1)^{\mathrm{tr}},
\end{aligned}$$

which, together with (5.11), yields

$$\frac{8}{2-\hat{y}}(0,0,1)^{\mathrm{tr}}$$

instead of $(0,0,1)^{\mathrm{tr}}$, thereby showing that the curl of the lowest-order edge basis functions fail to reproduce constants.

5.2.2 Extended Edge Shape Functions

The aim of this section is to illustrate the completeness of the enhanced edge basis presented in [25, 43] with the hexahedron given in (5.5). More precisely, using the function $\mathbf{A}$ defined in (5.6), we claim that $\mathbf{curl}\ \mathbf{A} \equiv (0,0,1)^{\mathrm{tr}}$ is exactly reproduced by the curl of the new edge basis on the non-affine hexahedron. Proceeding as in the previous section, we will use the L^2-orthogonal projection of $\hat{\mathbf{curl}}\ \hat{\mathbf{A}}$, as defined in (5.13), with respect to the curl of the shape functions defined on the reference hexahedron $\hat{K} = [-1,1]^3$, and then use the mapping given in (5.3) to deduce the representation of $\mathbf{curl}\ \mathbf{A}$ by the curl of the edge basis functions defined on the physical element K.

5.2.2.1 Additional Shape Functions

In order to preserve the approximation properties of edge finite elements when applied to non-affine meshes, standard edge elements need to include additional functions [43, 25]. We recall below the formulae for the additional edge basis functions, adapted from [43, Section 6], or equivalently [25, Proposition 2], to correspond to the case $\hat{K} = [-1, 1]^3$:

$$
\begin{aligned}
\hat{\mathbf{w}}_{13}(\hat{\mathbf{x}}) &:= (0,\ (\hat{x}-1)(\hat{x}+1)(\hat{z}-1)/8,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{14}(\hat{\mathbf{x}}) &:= (0,\ -(\hat{x}-1)(\hat{x}+1)(\hat{z}+1)/8,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{15}(\hat{\mathbf{x}}) &:= (0,\ 0,\ (\hat{x}-1)(\hat{x}+1)(\hat{y}-1)/8)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{16}(\hat{\mathbf{x}}) &:= (0,\ 0,\ -(\hat{x}-1)(\hat{x}+1)(\hat{y}+1)/8)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{17}(\hat{\mathbf{x}}) &:= ((\hat{y}-1)(\hat{y}+1)(\hat{z}-1)/8,\ 0,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{18}(\hat{\mathbf{x}}) &:= (-(\hat{y}-1)(\hat{y}+1)(\hat{z}+1)/8,\ 0,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{19}(\hat{\mathbf{x}}) &:= (0,\ 0,\ (\hat{x}-1)(\hat{y}-1)(\hat{y}+1)/8)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{20}(\hat{\mathbf{x}}) &:= (0,\ 0,\ -(\hat{x}+1)(\hat{y}-1)(\hat{y}+1)/8)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{21}(\hat{\mathbf{x}}) &:= ((\hat{y}-1)(\hat{z}-1)(\hat{z}+1)/8,\ 0,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{22}(\hat{\mathbf{x}}) &:= (-(\hat{y}+1)(\hat{z}-1)(\hat{z}+1)/8,\ 0,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{23}(\hat{\mathbf{x}}) &:= (0,\ (\hat{x}-1)(\hat{z}-1)(\hat{z}+1)/8,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{24}(\hat{\mathbf{x}}) &:= (0,\ -(\hat{x}+1)(\hat{z}-1)(\hat{z}+1)/8,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{25}(\hat{\mathbf{x}}) &:= ((\hat{y}-1)(\hat{y}+1)(\hat{z}-1)(\hat{z}+1)/16,\ 0,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{26}(\hat{\mathbf{x}}) &:= (0,\ (\hat{x}-1)(\hat{x}+1)(\hat{z}-1)(\hat{z}+1)/16,\ 0)^{\mathrm{tr}},\\
\hat{\mathbf{w}}_{27}(\hat{\mathbf{x}}) &:= (0,\ 0,\ (\hat{x}-1)(\hat{x}+1)(\hat{y}-1)(\hat{y}+1)/16)^{\mathrm{tr}}.
\end{aligned}
$$

The first 12 new edge basis functions are associated with degrees of freedom of the form (see [43, Equation (6.3)])

$$
\int_{\hat{f}} (\hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}})) \cdot \hat{\mathbf{n}} \hat{p}(\hat{\mathbf{x}}) \mathrm{d}\hat{\mathbf{x}}, \quad \text{for all faces } \hat{f} \text{ with normal } \hat{\mathbf{n}}, \tag{5.15}
$$

where $p(\hat{\mathbf{x}})$ is a homogeneous linear polynomial, while the last three additional basis functions are associated with degrees of freedom of the form [43, Equation (6.4)]

$$\int_{\hat{K}} (\hat{\mathbf{curl}}\ \mathbf{u}(\hat{\mathbf{x}})) \cdot \hat{\mathbf{r}}(\hat{\mathbf{x}}) \mathrm{d}\hat{\mathbf{x}}, \tag{5.16}$$

where $\hat{\mathbf{r}}$ is taken from the set [43, Equation (3.6)]

$$\hat{\mathbf{R}} := \{\hat{\mathbf{r}}_1, \hat{\mathbf{r}}_2, \hat{\mathbf{r}}_3\} := \{(0, -\hat{z}/2, \hat{y}/2)^{\mathrm{tr}},\ (-\hat{z}/2, 0, \hat{x}/2)^{\mathrm{tr}},\ (-\hat{y}/2, \hat{x}/2, 0)^{\mathrm{tr}}\}. \tag{5.17}$$

The table below summarizes the correspondence between the new shape functions and their degrees of freedom.

Basis function	d.o.f	value
$\hat{\mathbf{w}}_{13}$	$\int_{\hat{z}=-1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{x}\mathrm{d}\hat{\mathbf{x}}$	$2/3$
$\hat{\mathbf{w}}_{14}$	$\int_{\hat{z}=1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{x}\mathrm{d}\hat{\mathbf{x}}$	$-2/3$
$\hat{\mathbf{w}}_{15}$	$\int_{\hat{y}=-1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{x}\mathrm{d}\hat{\mathbf{x}}$	$-2/3$
$\hat{\mathbf{w}}_{16}$	$\int_{\hat{y}=1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{x}\mathrm{d}\hat{\mathbf{x}}$	$2/3$
$\hat{\mathbf{w}}_{17}$	$\int_{\hat{z}=-1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{y}\mathrm{d}\hat{\mathbf{x}}$	$-2/3$
$\hat{\mathbf{w}}_{18}$	$\int_{\hat{z}=1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{y}\mathrm{d}\hat{\mathbf{x}}$	$2/3$
$\hat{\mathbf{w}}_{19}$	$\int_{\hat{x}=-1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{y}\mathrm{d}\hat{\mathbf{x}}$	$2/3$
$\hat{\mathbf{w}}_{20}$	$\int_{\hat{x}=1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{y}\mathrm{d}\hat{\mathbf{x}}$	$-2/3$
$\hat{\mathbf{w}}_{21}$	$\int_{\hat{y}=-1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{z}\mathrm{d}\hat{\mathbf{x}}$	$2/3$
$\hat{\mathbf{w}}_{22}$	$\int_{\hat{y}=1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{z}\mathrm{d}\hat{\mathbf{x}}$	$-2/3$
$\hat{\mathbf{w}}_{23}$	$\int_{\hat{x}=-1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{z}\mathrm{d}\hat{\mathbf{x}}$	$-2/3$
$\hat{\mathbf{w}}_{24}$	$\int_{\hat{x}=1} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{n}}\hat{z}\mathrm{d}\hat{\mathbf{x}}$	$2/3$
$\hat{\mathbf{w}}_{25}$	$\int_{[-1,1]^3} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{r}}_{\mathbf{1}}(\hat{\mathbf{x}})\mathrm{d}\hat{\mathbf{x}}$	$2/9$
$\hat{\mathbf{w}}_{26}$	$\int_{[-1,1]^3} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{r}}_{\mathbf{2}}(\hat{\mathbf{x}})\mathrm{d}\hat{\mathbf{x}}$	$-2/9$
$\hat{\mathbf{w}}_{27}$	$\int_{[-1,1]^3} \hat{\mathbf{curl}}\ \hat{\mathbf{u}}(\hat{\mathbf{x}}) \cdot \hat{\mathbf{r}}_{\mathbf{3}}(\hat{\mathbf{x}})\mathrm{d}\hat{\mathbf{x}}$	$2/9$

5.2.2.2 Completeness Test Revisited

We illustrate the results of [43, 25] by showing that the new edge finite element spaces, including the functions defined in the previous section, are complete on the non-affine hexahedron with vertices given by (5.5). More precisely, we want to show, that with $\mathbf{A}$ given in (5.6), that is, $\mathbf{A}(\mathbf{x}) = (0, x, 0)^{\mathrm{tr}}$, the constant field $\mathbf{curl}\ \mathbf{A}(\mathbf{x}) = (0, 0, 1)^{\mathrm{tr}}$ is correctly represented by the curl of the new basis.

We start by determining which of the new shape functions, combined with $\hat{\mathbf{w}}_4$ and $\hat{\mathbf{w}}_8$, are involved in the representation of $\mathbf{A}(\mathbf{x})$, and thus in that of $\mathbf{curl}\ \mathbf{A}(\mathbf{x})$. To this end, observe from (5.13) that

$$\hat{\mathbf{curl}}\ \hat{\mathbf{A}}(\hat{\mathbf{x}}) = (0, 0, -(\hat{y} - 2)/8)^{\mathrm{tr}},$$

so that $\hat{\mathbf{curl}}\ \hat{\mathbf{A}}(\hat{\mathbf{x}})$ is along the $\hat{z}$-direction. It follows from (5.15) and (5.16) that $\hat{\mathbf{A}}$ only has degrees of freedom on the faces $\hat{z} = -1$ and $\hat{z} = 1$, plus degrees of freedom associated with interior values of $\hat{\mathbf{curl}}\ \hat{\mathbf{A}}$. Using the degree of freedom table given in the previous section together with (5.17), we find that, in addition to $\hat{\mathbf{w}}_4$ and $\hat{\mathbf{w}}_8$, the field $\hat{\mathbf{A}}$ has degrees of freedom associated with $\hat{\mathbf{w}}_{13}$, $\hat{\mathbf{w}}_{14}$, $\hat{\mathbf{w}}_{17}$, $\hat{\mathbf{w}}_{18}$, $\hat{\mathbf{w}}_{25}$, and $\hat{\mathbf{w}}_{26}$. Hence, in order to find the representation of $\hat{\mathbf{curl}}\ \hat{\mathbf{A}}$ with respect to the curl of the new edge basis functions, it suffices to compute the L^2-orthogonal projection of $\hat{\mathbf{curl}}\ \hat{\mathbf{A}}$ onto the curl of the linear space spanned by the set $\mathcal{B}_{\hat{\mathbf{A}}}$, where $\mathcal{B}_{\hat{\mathbf{A}}}$ consists of

$$\mathcal{B}_{\hat{\mathbf{A}}} := \{\hat{\mathbf{w}}_4, \hat{\mathbf{w}}_8, \hat{\mathbf{w}}_{13}, \hat{\mathbf{w}}_{14}, \hat{\mathbf{w}}_{17}, \hat{\mathbf{w}}_{18}, \hat{\mathbf{w}}_{25}, \hat{\mathbf{w}}_{26}\}.$$

More precisely, we solve the system

$$\tilde{\mathbf{P}} = \tilde{f}_{\hat{\mathbf{A}}} \tag{5.18}$$

where $\tilde{\mathbf{P}}$ is the 8 by 8 matrix with entries

$$\tilde{\mathbf{P}}_{\hat{\mathbf{w}},\hat{\mathbf{v}}} := \langle \hat{\mathbf{w}}, \hat{\mathbf{v}} \rangle_{\hat{K}}, \quad \hat{\mathbf{w}},\ \hat{\mathbf{v}} \text{ in } \mathcal{B}_{\hat{\mathbf{A}}},$$

and $\tilde{\mathbf{A}}$ is the column vector composed of

$$\tilde{f}_{\hat{\mathbf{A}},\hat{\mathbf{w}}} := \langle \hat{\mathbf{A}}, \hat{\mathbf{w}} \rangle_{\hat{K}}, \quad \hat{\mathbf{w}} \text{ in } \mathcal{B}_{\hat{\mathbf{A}}}.$$

In the above equations, recall that $\langle \cdot, \cdot \rangle_{\hat{K}}$ denotes the inner product defined in (5.9).

Solving the linear system (5.18) yields

$$\alpha_4 = \alpha_8 = -1, \ \alpha_{17} = \alpha_{18} = -1/4, \ \alpha_{13} = \alpha_{14} = \alpha_{25} = \alpha_{26} = 0,$$

so that the representation of $\hat{\mathbf{curl}}\ \hat{\mathbf{A}}$ with respect to the curl of the new basis is given by

$$\begin{aligned}
&\hat{\mathbf{curl}}\ \big[-\hat{\mathbf{w}}_4 - \hat{\mathbf{w}}_8 - \frac{1}{4}\hat{\mathbf{w}}_{17} - \frac{1}{4}\hat{\mathbf{w}}_{18}\big] \\
&\qquad = \hat{\mathbf{curl}}\ \big[-(0, (\hat{x}+1)(\hat{z}-1)/8, 0)^{\mathrm{tr}} - (0, -(\hat{x}+1)(\hat{z}+1)/8, 0)^{\mathrm{tr}} \\
&\qquad - \frac{1}{4}((\hat{y}-1)(\hat{y}+1)(\hat{z}-1)/8, 0, 0)^{\mathrm{tr}} \\
&\qquad - \frac{1}{4}(-(\hat{y}-1)(\hat{y}+1)(\hat{z}+1)/8, 0, 0)^{\mathrm{tr}}\big],
\end{aligned}$$

which, after simplification, gives

$$\hat{\mathbf{curl}}\ (\frac{\hat{y}^2-1}{16}, \frac{\hat{x}+1}{4}, 0)^{\mathrm{tr}} = (0, 0, \frac{2-\hat{y}}{8})^{\mathrm{tr}}.$$

It then follows from (5.4) that the representation of $\mathbf{curl}\ \mathbf{A}(\mathbf{x})$ is given by

$$\mathbf{P}_\Phi(\hat{\mathbf{x}})(0, 0, \frac{2-\hat{y}}{8})^{\mathrm{tr}} = |\mathrm{d}\Phi(\hat{\mathbf{x}})|^{-1}\mathrm{d}\Phi(\hat{\mathbf{x}})(0, 0, \frac{2-\hat{y}}{8})^{\mathrm{tr}},$$

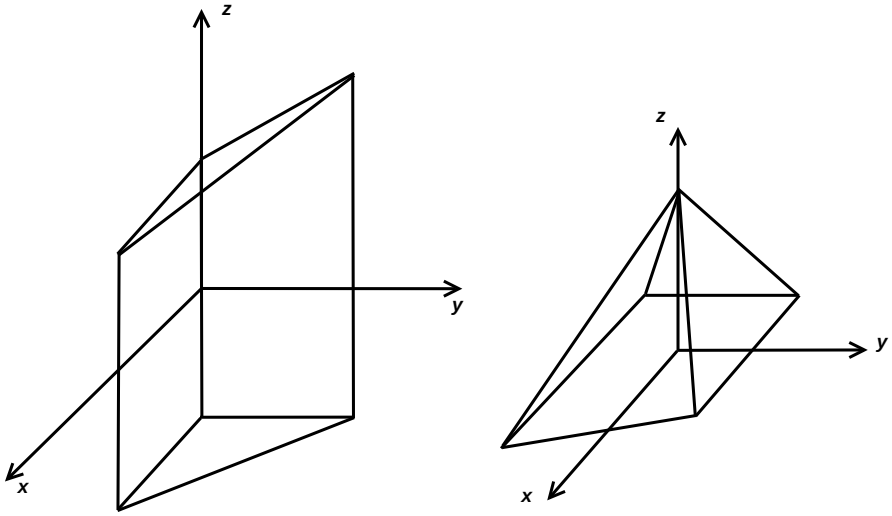
which, combined with (5.11) and (5.12), yields

$$\frac{16}{2-y}\begin{pmatrix} (2-y)/4 & -(1+x)/4 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & 1/2 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ \frac{2-\hat{y}}{8} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

thereby proving that the curl of the new edge shape functions on the physical element K indeed contains the constant vector $(0, 0, 1)^{\mathrm{tr}}$.

Although not reported, similar computations show that the unit vectors $(0, 1, 0)^{\mathrm{tr}}$ and $(1, 0, 0)^{\mathrm{tr}}$ are also generated by the curl of the new edge basis, thereby proving that the curl of the enhanced edge shape functions contain constants on the non-affine hexahedron.

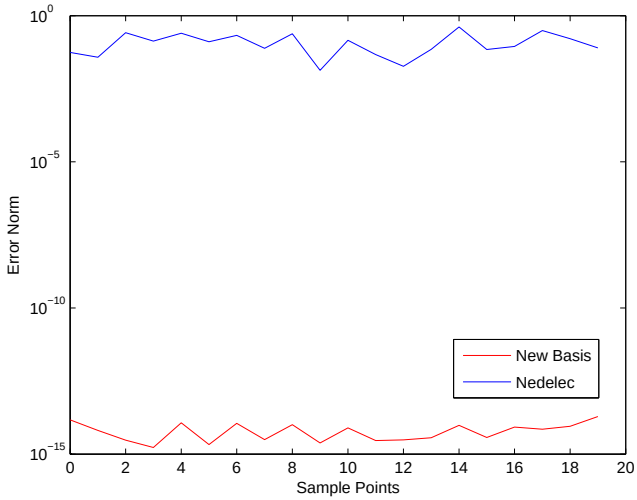
We have implemented the new shape functions presented in [43, 25] on the prism, the pyramid and the hexahedron for the lowest-order case $r = 0$. Similarly to the hexahedron given in Figure 5.3, the completeness of the new edge basis has been tested on the non-affine prism and pyramid given on Figure 5.4(a) and Figure 5.4(b) below.



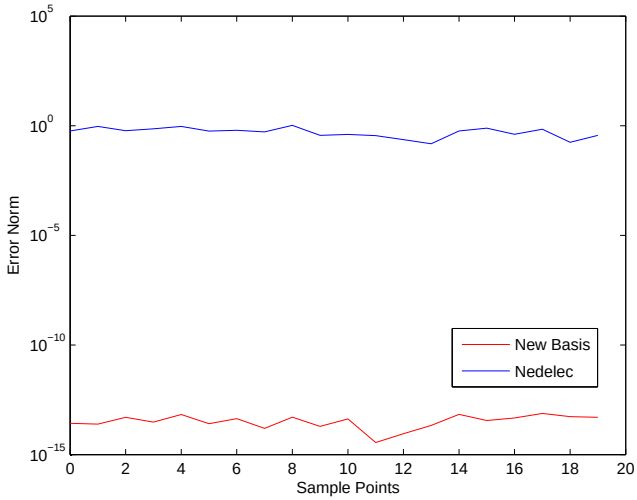
(a) Distorted Prism (b) Distorted Pyramid

Figure 5.4: Non-Affine Elements

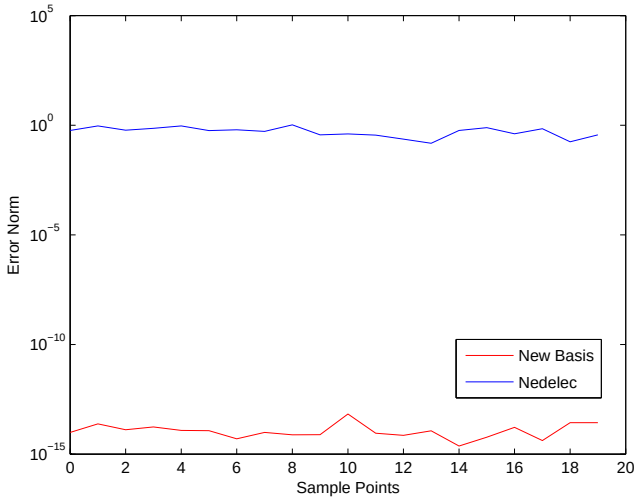
Figure 5.5 displays the error obtained when projecting constant fields onto the curl of the new edge basis, using distorted non-affine geometries, which confirms the fact that, in contrast to Nédélec shape functions, the new edge basis is able to reproduce constants on the non-affine elements.



(a) Distorted Hexahedron



(b) Distorted Prism



(c) Distorted Pyramid

Figure 5.5: Completeness Test

5.2.3 Numerical Simulations

This section includes a comparative study between the classical Nédélec elements and the enriched elements discussed in Section 5.1.2. Both hexahedral elements are used to compute the eigenvalues of a simple rectangular cavity with perfectly conducting walls. The computations were conducted by John Simkin from Cobham Technical Services [1], using Opera SOPRANO [3], with a 3.6 Ghz Intel Xeon E5-1620. The solution error is measured as the relative error in the frequency of the computed modes. For the sake of clarity, three non-trivial modes are represented in the numerical results below.

Using a uniform rectangular mesh produces the results given in Table 5.1 and Table 5.2 below, where h and N respectively denote the element size and the number of elements:

h	N	dof	time (s)	Error1	Error2	Error3
1	1000	10830	2	6.73e-6	4.07e-3	7.79e-3
0.5	8000	91260	20	4.22e-7	1.03e-3	2.04e-3
0.25	64000	748920	322	2.64e-8	2.57e-4	5.12e-4

Table 5.1: Relative Frequency Error with Enhanced Elements

h	N	dof	time (s)	Error1	Error2	Error3
1	1000	2430	1	4.12e-3	4.12e-3	1.24e-2
0.5	8000	21660	6	1.03e-3	1.03e-3	3.09e-3
0.25	64000	182520	74	2.57e-4	2.57e-4	7.71e-4

Table 5.2: Relative Frequency Error with Nédélec Elements

The variation of the error against the mesh size, as given in the above tables, is plotted on Figure 5.6.

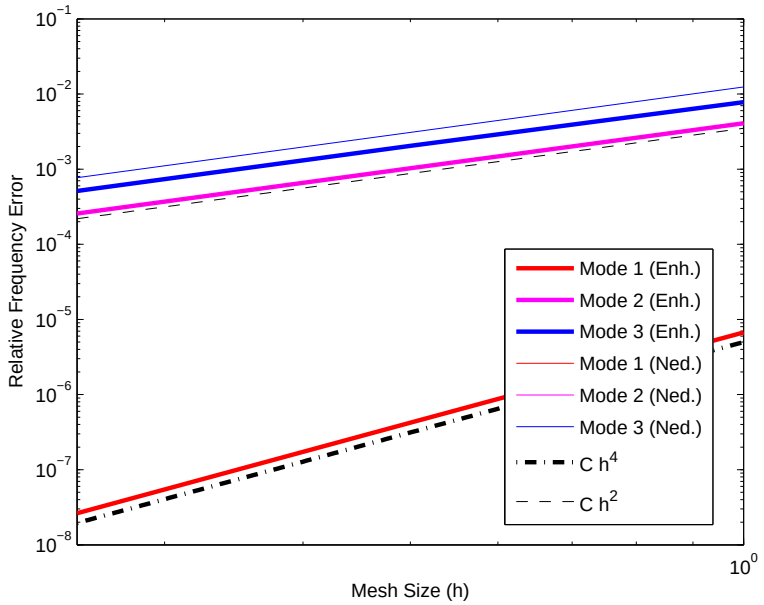


Figure 5.6: Using a Rectangular Mesh

Apart from the first mode which exhibits quadratic convergence with respect to the mesh size, Mode 2 and Mode 3 converges as $\mathcal{O}(h^2)$, when using the rectangular mesh. Hence, Nédélec elements are more efficient on affine meshes, because they involve fewer degrees of freedom.

Similar computations have been performed, using the non-affine mesh below.

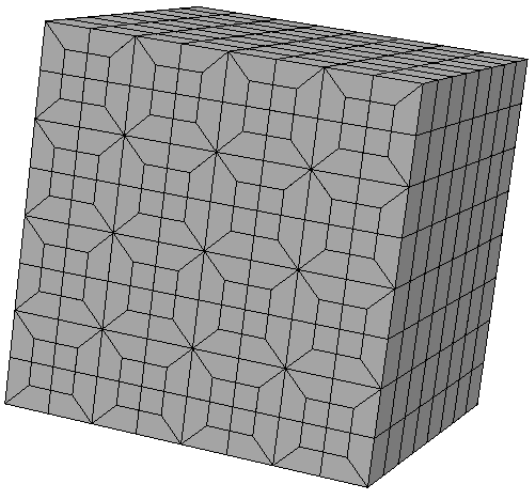


Figure 5.7: Non-Affine Ziggurat Mesh

The results obtained using the non-affine mesh given in Figure 5.7 are summarized in Table 5.3 and Table 5.4 below.

h	N	dof	time (s)	Error1	Error2	Error3
2.5	24	272	0.3	1.16e-3	1.12e-2	1.40e-2
1.25	192	2240	0.9	3.21e-4	2.79e-3	4.37e-3
0.625	1536	18176	2.74	8.23e-5	6.98e-4	2.00e-3
0.3125	12288	146432	18	2.07e-5	1.75e-4	2.75e-4
0.15625	98304	1175552	240	5.19e-6	4.37e-5	6.87e-5

Table 5.3: Relative Frequency Error with Enhanced Elements

h	N	dof	time (s)	Error1	Error2	Error3
2.5	24	186	0.2	2.41e-2	2.62e-2	6.19e-2
1.25	192	544	0.8	1.14e-2	1.28e-2	1.82e-2
0.625	1536	4480	1.29	8.18e-3	1.03e-2	1.35e-2
0.3125	12288	36352	6.5	7.38e-3	9.69e-3	1.23e-2
0.15625	98304	292864	58.1	7.18e-3	9.53e-3	1.19e-2

Table 5.4: Relative Frequency Error with Nédélec Elements

The variation of the error as a function of the mesh size is shown on Figure 5.8.

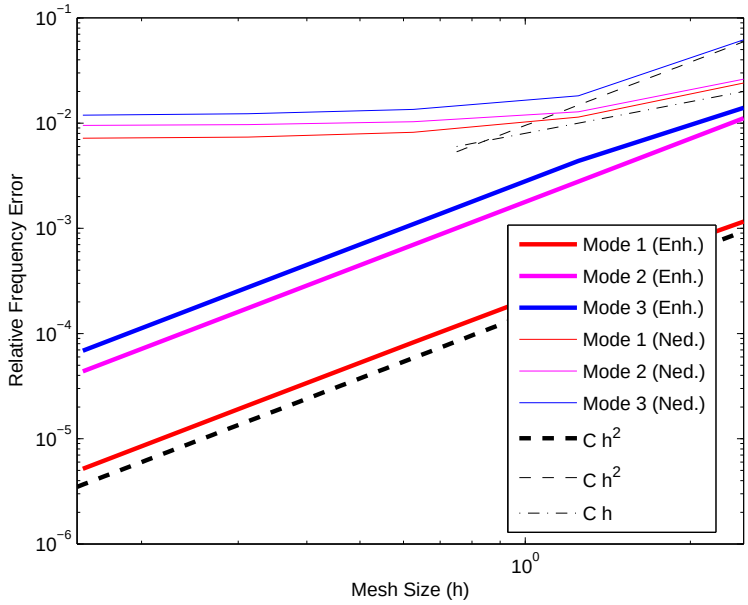


Figure 5.8: Using a Non-Affine Mesh

As predicted, the enhanced edge elements yield a $\mathcal{O}(h^2)$ convergence rate.

By contrast, refining the mesh brings no significant improvement on the accuracy produced by the Nédélec elements, as shown in Table 5.4. Comparing Table 5.3 and Table 5.4 shows that, on the non-affine mesh, the enhanced edge elements are more efficient than the Nédélec ones in terms of accuracy, number of degrees of freedom and computing time.

Chapter 6

Conclusion

This project sought to apply the desirable properties of the Bernstein-Bézier (BB) polynomials discussed in Chapter 1 to the finite element method. Although quite popular in CAGD, BB polynomials had received little attention in the finite element community. This project was set out to design finite element shape functions based on BB polynomials, establish the computational advantages of such bases, and provide ready-to-implement finite element algorithms.

The main findings consist of algorithms which enable for simplicial elemental system matrices of order n to be computed with optimal complexity $\mathcal{O}(n^{2d})$ in d dimensions, while taking account of the cost associated with numerical quadrature. The proposed algorithms include the one-, two- and three-dimensional settings for H^1-conforming finite elements, whereas two-dimensional Nédélec spaces are considered for $H(\text{curl})$-conforming elements. To the best of the author's knowledge, this optimal complexity result on simplicial elements is new.

The key approach behind the design of the algorithms consists in writing each elemental quantity in terms of the B-moments defined in (2.1), which is only possible because the product of two Bernstein polynomials is also a Bernstein polynomial of higher degree:

- in the case of piecewise constant data, this transformation, coupled with the closed expression (3.10) for the integral of a Bernstein polynomial, yields

explicit formulas which can be implemented using $\mathcal{O}(n^{2d})$ operations;

- in the case of variable data, once the elemental quantities have been written as weighted sums of B-moments, we simply make use of the efficient B-moment evaluation given in Chapter 2. As a consequence, the additional cost associated with numerical quadrature is negligible compared to the overall complexity $\mathcal{O}(n^{2d})$.

The optimal complexity is achieved in particular because the above-mentioned transformations into B-moments are sparse, and hard-coded into the computations. A fundamental property behind the sparsity of the transformation is given by the closed formula (3.8) for the gradient of a Bernstein polynomial.

For variable data, the efficient computations of the element system matrices rely on the optimal evaluation of the B-moments. A key observation behind this optimal result is that, although the Bernstein-Bézier shape functions are not based on a tensorial construction, a tensor product structure arises from the application of the Duffy transformation defined in (2.2) to the B-moments, thereby allowing for sum factorization techniques to be used.

The proposed algorithms have been implemented in the `bbfem` library which is available through a GPL licence. Numerical simulations involving benchmark test problems are consistent with the expected accuracy of the finite elements, as well as with the predicted optimal cost associated with the assembly of the BB elemental quantities.

Finally, some remarks on future work are due. Since the algorithms have been designed with high polynomial orders in mind, the running times for specific low orders can probably be improved by appropriate customization.

For reasonably large polynomial degrees, that is, up to $n = 25$, the proposed algorithms are numerically shown to produce accurate results. However, in order to fully exploit the optimal complexity of the Bernstein-Bézier elements for larger polynomial orders, future work should include the investigation of preconditioning techniques and solvers that are well suited for the resulting linear systems.

Further research may also include the extension of the two-dimensional $H(\text{curl})$ Bernstein-Bézier finite elements discussed in Chapter 4 to the three-dimensional setting.

Appendix A

Examples

This appendix provides illustrative examples regarding the finite element bases presented in this work. Section A.1 presents explicit formulas for the H^1 mass and stiffness matrices for $n = 2$ and $n = 3$, whereas Section A.2 gives the graphs of the $H(\mathrm{curl})$ shape functions introduced in Chapter 4 for the linear and quadratic polynomial orders.

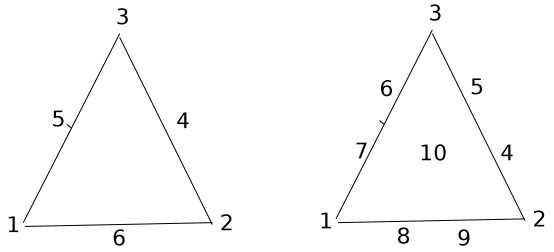
A.1 H^1 Element Mass and Stiffness Matrices

Example A.1.1. By ordering the indices in $\mathcal{I}_2^n$ as in Figure A.1 for $n = 2$ and $n = 3$ and with the help of Algorithm 3.3, we find that the two-dimensional H^1 element mass matrix is respectively given by $\mathbf{M}^{(2)}$ and $\mathbf{M}^{(3)}$, where

$$\mathbf{M}^{(2)} = \frac{|T|}{90} \begin{pmatrix} 6 & 1 & 1 & 1 & 3 & 3 \\ 1 & 6 & 1 & 3 & 1 & 3 \\ 1 & 1 & 6 & 3 & 3 & 1 \\ 1 & 3 & 3 & 4 & 2 & 2 \\ 3 & 1 & 3 & 2 & 4 & 2 \\ 3 & 3 & 1 & 2 & 2 & 4 \end{pmatrix}$$

and

$$\mathbf{M}^{(3)} = \frac{|T|}{560}\begin{pmatrix} 20 & 1 & 1 & 1 & 1 & 4 & 10 & 10 & 4 & 4 \\ 1 & 20 & 1 & 10 & 4 & 1 & 1 & 4 & 10 & 4 \\ 1 & 1 & 20 & 4 & 10 & 10 & 4 & 1 & 1 & 4 \\ 1 & 10 & 4 & 12 & 9 & 3 & 2 & 3 & 6 & 6 \\ 1 & 4 & 10 & 9 & 12 & 6 & 3 & 2 & 3 & 6 \\ 4 & 1 & 10 & 3 & 6 & 12 & 9 & 3 & 2 & 6 \\ 10 & 1 & 4 & 2 & 3 & 9 & 12 & 6 & 3 & 6 \\ 10 & 4 & 1 & 3 & 2 & 3 & 6 & 12 & 9 & 6 \\ 4 & 10 & 1 & 6 & 3 & 2 & 3 & 9 & 12 & 6 \\ 4 & 4 & 4 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \end{pmatrix}.$$



(a) $n = 2$ (b) $n = 3$

Figure A.1: Ordering used on the elements of I_2^n

Example A.1.2. Using Algorithm 3.9 with $\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, we find that the

two-dimensional H^1 element stiffness matrix $\mathbf{S}^{(2)}$, for $n = 2$, is given by

$$\mathbf{S}^{(2)} = \frac{1}{24}\left[\begin{pmatrix} 2&0&0&0&1&1\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 1&0&0&0&2&1\\ 1&0&0&0&1&2 \end{pmatrix}|\nabla\lambda_1|^2 + \begin{pmatrix} 0&1&0&1&0&2\\ 1&0&0&0&1&2\\ 0&0&0&0&0&0\\ 1&0&0&0&2&1\\ 0&1&0&2&0&1\\ 2&2&0&1&1&2 \end{pmatrix}\nabla\lambda_1\cdot\nabla\lambda_2 \right.$$
$$+ \begin{pmatrix} 0&0&1&1&2&0\\ 0&0&0&0&0&0\\ 1&0&0&0&2&1\\ 1&0&0&0&1&2\\ 2&0&2&1&2&1\\ 0&0&1&2&1&0 \end{pmatrix}\nabla\lambda_1\cdot\nabla\lambda_3 + \begin{pmatrix} 0&0&0&0&0&0\\ 0&2&0&1&0&1\\ 0&0&0&0&0&0\\ 0&1&0&2&0&1\\ 0&0&0&0&0&0\\ 0&1&0&1&0&2 \end{pmatrix}|\nabla\lambda_2|^2$$
$$\left. + \begin{pmatrix} 0&0&0&0&0&0\\ 0&0&1&2&1&0\\ 0&1&0&2&0&1\\ 0&2&2&2&1&1\\ 0&1&0&1&0&2\\ 0&0&1&1&2&0 \end{pmatrix}\nabla\lambda_2\cdot\nabla\lambda_3 + \begin{pmatrix} 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&2&1&1&0\\ 0&0&1&2&1&0\\ 0&0&1&1&2&0\\ 0&0&0&0&0&0 \end{pmatrix}|\nabla\lambda_3|^2 \right].$$

Similarly, Algorithm 3.10 with $\mathbf{A} = \begin{pmatrix} 1&0&0\\ 0&1&0\\ 0&0&1 \end{pmatrix}$ shows that the H^1 element stiffness matrix $\mathbf{S}^{(3)}$, for $n = 3$, is given by

$$\mathbf{S}^{(3)} = \frac{1}{80}\big(\mathbf{B}|\nabla\lambda_1|^2 + \mathbf{C}\nabla\lambda_1\cdot\nabla\lambda_2 + \mathbf{D}\nabla\lambda_1\cdot\nabla\lambda_3 + \mathbf{E}|\nabla\lambda_2|^2 + \mathbf{F}\nabla\lambda_2\cdot\nabla\lambda_3 + \mathbf{G}|\nabla\lambda_3|^2\big),$$

where the matrices $\mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}$ are given in Tab. A.1.

$$\mathbf{B} = \begin{pmatrix} 6 & 0 & 0 & 0 & 0 & 1 & 3 & 3 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 6 & 3 & 1 & 1 & 3 \\ 3 & 0 & 0 & 0 & 0 & 3 & 4 & 2 & 1 & 2 \\ 3 & 0 & 0 & 0 & 0 & 1 & 2 & 4 & 3 & 2 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 3 & 6 & 3 \\ 1 & 0 & 0 & 0 & 0 & 3 & 2 & 2 & 3 & 4 \end{pmatrix}, \quad \mathbf{C} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 6 & 3 & 3 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 3 & 6 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 3 & 2 & 2 & 3 & 4 \\ 1 & 0 & 0 & 0 & 0 & 6 & 3 & 1 & 1 & 3 \\ 0 & 1 & 0 & 3 & 6 & 0 & 0 & 1 & 1 & 3 \\ 0 & 1 & 0 & 2 & 3 & 0 & 0 & 3 & 2 & 4 \\ 6 & 3 & 0 & 2 & 1 & 1 & 3 & 6 & 5 & 3 \\ 3 & 6 & 0 & 3 & 1 & 1 & 2 & 5 & 6 & 3 \\ 3 & 3 & 0 & 4 & 3 & 3 & 4 & 3 & 3 & 4 \end{pmatrix}$$

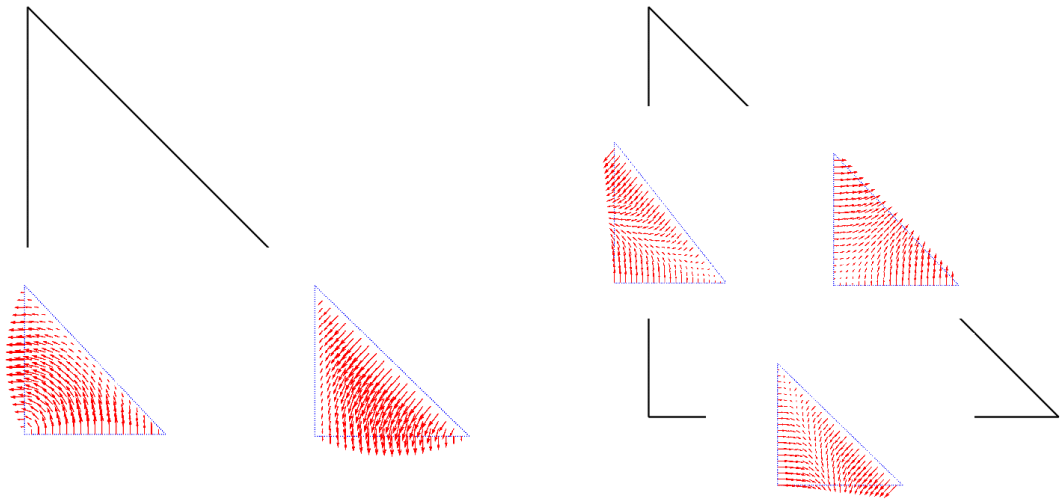
$$\mathbf{D} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 3 & 6 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 6 & 3 & 1 & 1 & 3 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 3 & 6 & 3 \\ 1 & 0 & 0 & 0 & 0 & 3 & 2 & 2 & 3 & 4 \\ 3 & 0 & 6 & 1 & 3 & 6 & 5 & 2 & 1 & 3 \\ 6 & 0 & 3 & 1 & 2 & 5 & 6 & 3 & 1 & 3 \\ 0 & 0 & 1 & 3 & 2 & 2 & 3 & 0 & 0 & 4 \\ 0 & 0 & 1 & 6 & 3 & 1 & 1 & 0 & 0 & 3 \\ 3 & 0 & 3 & 3 & 4 & 3 & 3 & 4 & 3 & 4 \end{pmatrix}, \quad \mathbf{E} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 3 & 1 & 0 & 0 & 1 & 3 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 4 & 3 & 0 & 0 & 1 & 2 & 2 \\ 0 & 1 & 0 & 3 & 6 & 0 & 0 & 1 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 6 & 3 & 3 \\ 0 & 3 & 0 & 2 & 1 & 0 & 0 & 3 & 4 & 2 \\ 0 & 1 & 0 & 2 & 3 & 0 & 0 & 3 & 2 & 4 \end{pmatrix}$$

$$\mathbf{F} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 6 & 3 & 1 & 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 3 & 6 & 0 & 0 & 1 & 1 & 3 \\ 0 & 6 & 3 & 6 & 5 & 2 & 1 & 1 & 3 & 3 \\ 0 & 3 & 6 & 5 & 6 & 3 & 1 & 1 & 2 & 3 \\ 0 & 1 & 0 & 2 & 3 & 0 & 0 & 3 & 2 & 4 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 6 & 3 & 3 \\ 0 & 0 & 1 & 1 & 1 & 3 & 6 & 0 & 0 & 3 \\ 0 & 0 & 1 & 3 & 2 & 2 & 3 & 0 & 0 & 4 \\ 0 & 3 & 3 & 3 & 3 & 4 & 3 & 3 & 4 & 4 \end{pmatrix}, \quad \mathbf{G} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 1 & 3 & 3 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 6 & 3 & 1 & 1 & 0 & 0 & 3 \\ 0 & 0 & 3 & 3 & 4 & 2 & 1 & 0 & 0 & 2 \\ 0 & 0 & 3 & 1 & 2 & 4 & 3 & 0 & 0 & 2 \\ 0 & 0 & 1 & 1 & 1 & 3 & 6 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 2 & 2 & 3 & 0 & 0 & 4 \end{pmatrix}.$$

Table A.1: The matrices in the formula for $\mathbf{S}^3$ in Example A.1.2.

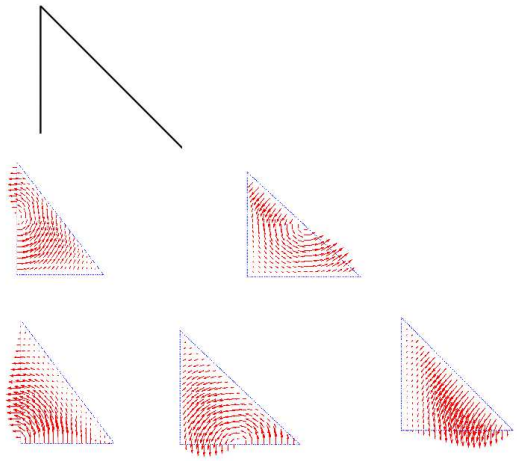
A.2 Basis Functions Graphs

We proceed in this section to plot the $H(\text{curl})$ shape functions (excluding the Whitney edge functions) associated with the standard triangle T with vertices $\mathbf{0}, (1,0)$ and $(0,1)$.

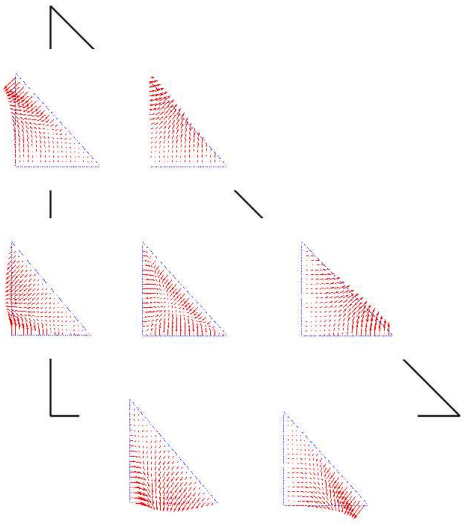


(a) Sigma basis functions (b) Gradient basis functions

Figure A.2: Basis Functions for $n = 1$



(a) Sigma basis functions



(b) Gradient basis functions

Figure A.3: Basis functions for $n = 2$

Appendix B

C++ Library Documentation

The Bernstein-Bézier algorithms presented in the previous chapters, as well as those presented in [11] have been implemented using the `C++` programming language. The code is available from [4] under a GPL license. In contrast to the algorithms given in this work, those described in [11] make no use of precomputed arrays of the form (2.9), (2.16), (2.21). For this reason, the approach considered in [11] will be referred to as the method *without precomputed arrays*. This section contains the documentation which will be useful in order to make use of the 2D and 3D source code for H^1 finite elements as well as the 2D source code for $H(\text{curl})$ finite elements. Recall that, in two dimensions, $H(\text{curl})$ is isomorphic to $H(\text{div})$, which means that the presented codes may also be used for $H(\text{div})$ computations in 2D (see Equation (4.5)). In this chapter, each routine will have its purpose, parameter(s) and output explained. For simplicity, only the routines used for 3D computations are considered for H^1 finite elements. The 2D routines are based on a similar design, and are given in [4]. For details on each routine's method and its interactions with the other routines, see [4].

The 2D and 3D source code for H^1 computations is called `bbfem.cpp`. Its purpose is to provide efficient routines for the computation of the elemental quantities for H^1 finite elements in two and three dimensions, using the algorithms presented in in Chapter 2 and Chapter 3. When the data is variable, the code makes use of Gauss-Jacobi quadrature weights and centres which are stored in the data file

`JacobiGaussNodes.h`, given in [4]. More precisely, `JacobiGaussNodes.h` contains the declarations of the `legendre`, `jacobi` and `jacobi2` arrays which respectively contain Gauss-Jacobi quadrature weights and centres with $(\alpha, \beta) = (0, 0)$, $(\alpha, \beta) = (1, 0)$ and $(\alpha, \beta) = (2, 0)$ from $q = 2$ to $q = 80$. The minimal value $q = 2$ comes from the assumption that the finite element order is at least equal to 1 for which the Stroud rule of order 2 is needed. In addition to the routines contained in `bbfem.cpp` for the computation of the B-moments, the source code for computing the $H(\text{curl})$ elemental quantities is called `bbfem2dCurl.cpp`.

This chapter is organized as follows. Section B.1 below lists the routines which are contained in `bbfem.cpp`, along with a brief description of each routine's purpose. The routines' syntax is discussed in Section B.2. Section B.3 then lists all the routines which are specific to $H(\text{curl})$ computations. The $H(\text{curl})$ routines' syntax is described in Section B.4.

B.1 H^1 Routines List

This section alphabetically lists the routines in `bbfem.cpp` for the computation of elemental quantities in two and three dimensions. Routines named with a suffix `3d` are specific to 3D computations and have 2D analogues which perform a similar task. In addition, routines which are specific to the approach developed in [11] are marked with an asterisk:

- **Area2d:** computes the area of the triangle.
- **assign_pointers_Bmom2d, assign_pointers_Mass2d, assign_pointers_Convec2d, assign_pointers_Stiff2d:** initialize the intermediate arrays used in the computation of the 2D elemental quantities.
- **assign_pointers_Bmom3d, assign_pointers_Mass3d, assign_pointers_Convec3d, assign_pointers_Stiff3d:** initialize the intermediate arrays used in the computation of the 3D elemental quantities.

- **assign_quadra2d:** assigns values to the arrays used for storing the quadrature weights and centres.
- **assign_quadra3d:** assigns values to the arrays used for storing the quadrature weights and centres.
- **bary2cart2d:** map from barycentric to Cartesian coordinates.
- **bary2cart3d:** map from barycentric to Cartesian coordinates.
- **Bmoment2d:** computes the B-moments in two dimensions.
- **Bmoment2d_const:** computes the 2D B-moments associated with constant coefficients equal to 1.
- **Bmoment2d_Index*:** computes the 2D B-moments using the method without precomputed arrays.
- **Bmoment3d:** computes the B-moments in three dimensions.
- **Bmoment3d_const:** computes the 3D B-moments associated with constant coefficients equal to 1.
- **Bmoment3d_Index*:** computes the 3D B-moments using the method without precomputed arrays.
- **computeBinomials:** computes binomial coefficients using Pascal's Triangle.
- **Convec2d:** computes the 2D convective matrix associated with variable coefficients.
- **Convec2d_const:** computes the 2D convective matrix associated with constant vector-valued coefficients.
- **Convec3d:** computes the 3D convective matrix associated with variable coefficients.

- **Convec3d_const:** computes the 3D convective matrix associated with constant vector-valued coefficients.
- **create_BinomialMat:** allocates memory for storing binomial coefficients. Precomputing binomial coefficients is a key step in order to obtain efficient routines using Bernstein polynomials.
- **create_Bmoment:** allocates memory for storing the B-moment vector entries. This routine can handle scalar-, vector- or matrix-valued B-moments.
- **create_quadraWN2d:** allocates memory for storing the quadrature weights and centres used in the computation of the 2D elemental quantities associated with variable coefficients.
- **create_quadraWN3d:** allocates memory for storing the quadrature weights and centres used in the computation of the 3D elemental quantities associated with variable coefficients.
- **create_Mat:** allocates memory to the mass, convective or stiffness matrix.
- **create_matValNodes2d:** allocates memory for storing the value of the data at the 2D Stroud nodes.
- **create_matValNodes3d:** allocates memory for storing the value of the data at the 3D Stroud nodes.
- **create_precomp2d:** allocates memory for the arrays defined in (2.16).
- **create_precomp3d:** allocates memory for the arrays defined in (2.21).
- **crossProd2:** computes the half-scaled cross-product of two vectors.
- **data_at_Nodes_Bmom2d, data_at_Nodes_Mass2d, data_at_Nodes_Convec2d, data_at_Nodes_Stiff2d:** compute the values of the data at the Stroud nodes.

- **data_at_Nodes_Bmom3d, data_at_Nodes_Mass3d, data_at_Nodes_Convec3d, data_at_Nodes_Stiff3d:** compute the values of the data at the Stroud nodes.
- **data_at_Nodes_Cval2d:** reads the value of the data at the Stroud nodes, when using an array of data values at the Stroud nodes as input.
- **data_at_Nodes_Cval3d:** reads the value of the data at the Stroud nodes, when using an array of data values at the Stroud nodes as input.
- **delete_BinomialMat:** frees the memory allocated to the array storing binomial coefficients.
- **delete_Bmoment:** frees the memory used to store B-moments.
- **delete_Mat:** frees the memory allocated to the mass, convective or stiffness matrix.
- **delete_matValNodes:** frees the memory used to store the value of the data at the Stroud nodes.
- **delete_pointers_Bmom, delete_pointers_Mass, delete_pointers_Convec, delete_pointers_Stiff:** free the memory allocated to intermediate arrays used in the computation of the elemental quantities.
- **delete_precomp:** frees the memory allocated to the arrays defined in (2.21) in 3D, and (2.16) in 2D.
- **delete_quadraWN:** frees the memory used to store the quadrature weights and centres.
- **gaussJacobiUnit2D*:** transforms the Gauss-Jacobi quadrature weights and centres on [-1;1] to those on [0;1].
- **gaussJacobiUnit3D*:** transforms the Gauss-Jacobi quadrature weights and centres on [-1;1] to those on [0;1].

- **get_Bmoments2d:** driver routine for computing the 2D B-moments.
- **get_Bmoments2d_const:** driver routine for computing the 2D B-moments associated with constant coefficients equal to 1.
- **get_Bmoments3d:** driver routine for computing the 3D B-moments.
- **get_Bmoments3d_const:** driver routine for computing the 3D B-moments associated with constant coefficients equal to 1.
- **get_convec2d:** driver routine for computing the 2D convective matrix associated with variable coefficients.
- **get_convec2d_const:** driver routine for computing the 2D convective matrix associated with constant coefficients.
- **get_convec3d:** driver routine for computing the 3D convective matrix associated with variable coefficients.
- **get_convec3d_const:** driver routine for computing the 3D convective matrix associated with constant coefficients.
- **get_mass2d:** driver routine for computing the 2D mass matrix associated with variable coefficients.
- **get_mass2d_const:** driver routine for computing the 2D mass matrix with constant coefficients equal to 1.
- **get_mass3d:** driver routine for computing the 3D mass matrix associated with variable coefficients.
- **get_mass3d_const:** driver routine for computing the 3D mass matrix with constant coefficients equal to 1.
- **get_stiffness2d:** driver routine for computing the 2D stiffness matrix associated with variable coefficients.
- **get_stiffness2d_const:** driver routine for computing the 2D stiffness matrix associated with constant coefficients.

- **get_stiffness3d:** driver routine for computing the 3D stiffness matrix associated with variable coefficients.

- **get_stiffness3d_const:** driver routine for computing the 3D stiffness matrix associated with constant coefficients.

- **init_Bmoment2d_Cval*:** contains the first step of the B-moments computation for the method without precomputed arrays, when using an array of data values at the 2D Stroud nodes as input for the coefficients. More precisely, the B-moment vector entries are initialized with the values of the data at the Stroud nodes.

- **init_Bmoment3d_Cval*:** Analogue of `init_Bmoment2d_Cval` used in 3D computations.

- **init_BmomentC_Bmom2d, init_BmomentC_Mass2d, init_BmomentC_Convec2d, init_BmomentC_Stiff2d*:** contain the first step of the B-moments computation for the method without precomputed arrays. More precisely, the B-moment vector entries are initialized with the values of the data at the Stroud nodes.

- **init_BmomentC_Bmom3d, init_BmomentC_Mass3d, init_BmomentC_Convec3d, init_BmomentC_Stiff3d*:** Analogues of the previous routines used in 3D computations.

- **init_precomp2d:** computes the arrays given in (2.16).

- **init_precomp3d:** computes the arrays given in (2.21).

- **innerProd_Coeff2d:** Given a vector-valued coefficient vectCoeff, computes the vector containing the inner products of vectCoeff with the outer normals to the triangle's edges. The innerProd_Coeff routine is needed for the computation of the convective matrix associated with constant coefficients.

- **innerProd_Coeff3d:** Given a vector-valued coefficient vectCoeff, computes the vector containing the inner products of vectCoeff with the outer normals to the tetrahedron's faces.
- **inter:** auxiliary routine which is needed in the computation of the normals to the tetrahedron's faces.
- **len_Mat2d:** returns the dimension of the 2D elemental matrices.
- **len_Mat3d:** returns the dimension of the 3D elemental matrices.
- **len_Moments2d:** returns the length of the array used to store the 2D B-moments. This depends on whether or not the method without precomputed arrays is used in order to compute the B-moments.
- **len_Moments3d:** returns the length of the array used to store the 3D B-moments. This depends on whether or not the method without precomputed arrays is used in order to compute the B-moments.
- **Mass2d:** computes the 2D mass matrix with variable coefficients.
- **Mass2d_const:** computes the 2D mass matrix associated with constant coefficients equal to 1.
- **Mass3d:** computes the 3D mass matrix with variable coefficients.
- **Mass3d_const:** computes the 3D mass matrix associated with constant coefficients equal to 1.
- **matrix_values_at_Stroud2d:** computes the values of a (symmetric) matrix-valued function at the 2D Stroud nodes of a given order.
- **matrix_values_at_Stroud3d:** computes the values of a (symmetric) matrix-valued function at the 3D Stroud nodes of a given order.
- **normals2d:** computes the outer normals to the edges of the triangle.
- **normals3d:** computes the outer normals to the faces of the tetrahedron.

- **position2d:** indexes the entries of the 2D mass, convective and stiffness matrix entries. Also indexes the B-moment vector entries, when the routines described in Chapter 3 are used for the B-moment computation.

- **position2d2*:** indexes the entries of the the 2D B-moment vector, when using the method without precomputed arrays.

- **position2d_sum:** auxiliary routine which computes the index associated with the sum of two domain points in two dimensions. This routine is needed for the 2D stiffness matrix computation.

- **position3d:** indexes the entries of the 3D mass, convective and stiffness matrix entries. Also indexes the B-moment vector entries, when the routines described in Chapter 3 are used for the B-moment computation.

- **position3d2*:** indexes the entries of the the 3D B-moment vector, when using the method without precomputed arrays.

- **position3d_sum, position3d_sum2:** auxiliary routines which compute the index associated with the sum of two domain points in three dimensions. These routines are needed for the 3D stiffness matrix computation.

- **scalarMatrix2d_Coeff:** computes the matrix containing weighted inner products of the outer normals to the triangle's edges. This matrix is needed for the computation of the 2D stiffness matrix associated with constant coefficients.

- **scalarMatrix3d_Coeff:** computes the matrix containing weighted inner products of the outer normals to the tetrahedron's faces.

- **scalarProd2d:** computes the scalar product of two vectors in two dimensions.

- **scalarProd3d:** computes the scalar product of two vectors in three dimensions.

- **scalar_values_at_Stroud2d:** computes the values of a scalar-valued function at the 2D Stroud nodes of a given order.
- **scalar_values_at_Stroud3d:** computes the values of a scalar-valued function at the 3D Stroud nodes of a given order.
- **Stiff2d:** computes the 2D stiffness matrix associated with variable coefficients.
- **Stiff2d_const:** computes the 2D stiffness matrix associated with constant matrix-valued coefficients.
- **Stiff3d:** computes the 3D stiffness matrix associated with variable coefficients.
- **Stiff3d_const:** computes the 3D stiffness matrix associated with constant matrix-valued coefficients.
- **stroud_nodes_bary2d:** computes the barycentric coordinates of the 2D Stroud nodes.
- **stroud_nodes_bary3d:** computes the barycentric coordinates of the 3D Stroud nodes.
- **subtract:** computes the difference of two 3D vectors.
- **transform_BmomentC_Convec2d:** computes the inner product of the vector-valued B-moments with the normals to the triangle's edges in 2D.
- **transform_BmomentC_Convec3d:** computes the inner product of the vector-valued B-moments with the normals to the tetrahedron's faces in 3D.
- **transform_BmomentC_Stiff2d:** multiplies the matrix-valued B-moments with the normals to the triangle's edges in 2D.
- **transform_BmomentC_Stiff3d:** multiplies the matrix-valued B-moments with the normals to the tetrahedron's faces in 3D.

- **vector_values_at_Stroud2d:** computes the values of a vector-valued function at the 2D Stroud nodes of a given order.
- **vector_values_at_Stroud3d:** computes the values of a vector-valued function at the 3D Stroud nodes of a given order.
- **Volume3d:** computes the volume of a tetrahedron.

Typically, routines of the form `create_[]` and `delete_[]` are respectively used to allocate and free the memory allocated to intermediate quantities. In addition, routines of the form `assign_[]` are used to assign values to intermediate quantities. The driver routines for computing the elemental quantities are of the form `get_[]`. Each driver routine is associated with a corresponding low-level routine which performs the same computation, with the exception that the low-level routines do not contain the initialization of auxilliary arrays. As a result, the arguments passed to low-level routines include the auxilliary arrays which are needed in order to compute the elemental quantities. The entries of the mass, convective and stiffness matrices are indexed with respect to `position3d` in 3D, and `position2d` in 2D. The above-mentioned routines are also used for indexing the B-moment vector entries, unless the method without precomputed arrays presented in [11] is used. In that case, alternative indexing methods, respectively given by `position3d2` and `position2d2` in two and three dimensions, are required. This is because the size of the array used to store the B-moments is slightly larger in the approach proposed in [11]. In the sequel, the `PRECOMP` macro indicates that the algorithms presented in Chapter 2 are used for the computation of the B-moments.

B.2 H^1 Routines Description

This section contains the execution of the driver routines computing 3D elemental quantities. The 2D routines are based on a similar design.Explanations on the use of the 3D routines listed in Section B.1 are given. For the definitions of the 2D

and 3D routines, see [4]. As mentioned before, in order to compute the elemental matrices associated with variable coefficients, only the coefficient values at the quadrature nodes is required. Hence, two data structures are possible for the input representing the coefficients: either a function definition, or an array of function values at the quadrature nodes. Thus, a flag called `functval` is used to distinguish the two approaches. By default, a function is used as input for the coefficients.

B.2.1 B-moments

This section discusses the routines used for computing the B-moments associated with variable coefficients. Since the B-moment entries associated with constant coefficients are given by a constant, the program for computing such B-moments can be considered as trivial and as such is left out. The routines presented in this section are involved with the computations of B-moments on the tetrahedron $T = \langle \mathbf{v}_i,\ i = 1, \ldots, 4 \rangle$ associated with a non-constant scalar-valued data. Depending on the value of the flag `functval`, the input for the coefficients is either given by a function called `f`, or an array called `Cval` which contains the values of the B-moment coefficients at the quadrature nodes.

With the purpose of offering a better understanding of the structure of the code, this section starts with the driver routine for computing B-moments of a given order. The routines responsible for allocating memory are then discussed in Section B.2.1.2, whereas the routines used for auxiliary computations are given in Section B.2.1.3. Section B.2.1.3 discusses the auxiliary routines which handle intermediate computations. Finally, Section B.2.1.4 displays an example of code execution for computing B-moments.

B.2.1.1 Driver Routine

This section focuses on `get_Bmoments3d` which is the driver routine used for computing the B-moments. The above-mentioned routine is declared as follows:

```
void
get_Bmoments3d(double **Bmoment, int n, double
    (*f)(double[3]), double *Cval, double v1[3], double
    v2[3],double v3[3], double v4[3], int functval);
```

In the above routine, recall that the parameter `functval` is used as a flag for setting the input used for the coefficients: With `functval`=0, the B-moments coefficients are produced by the function `f`, whereas with `functval`=1, the B-moments coefficients are produced by the array `Cval` which contains the data values at the Stroud nodes. The routine `get_Bmoments3d` computes the B-moments of order `n` on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`, associated with either `f` or `Cval`. The computed B-moments are stored into the array `Bmoment`.

B.2.1.2 Memory Allocation

This section discusses the purpose and the syntax of the B-moment routines which are used to allocate memory. These routines are listed as: `create_BinomialMat`, `create_Bmoment`, `create_quadraWN3d`, `create_matValNodes3d`, `create_precomp3d`, `delete_BinomialMat`, `delete_Bmoment`, `delete_matValNodes`, `delete_pointers_Bmom`, `delete_precomp`, `delete_quadraWN` and `len_Moments3d`.

When `PRECOMP` is switched on, the routine for computing the B-moments requires precomputed binomial coefficients. The memory used for storing such coefficients is allocated by means of `create_BinomialMat` which is declared as follows:

```
double **
create_BinomialMat(int len_binomialMat);
```

The above routine creates a square matrix of dimension `len_binomialMat`.

The routine used to allocate memory for storing the computed B-moments is declared as follows:

```
double **
```

```
create_Bmoment( int lenMoments, int nb_Array );
```

In the above routine, `lenMoments` specifies the size allocated to the B-moments, whereas `nb_Array` depends on the type of coefficients associated with the B-moments. Note that, by adjusting the value of `nb_Array` and `lenMoments`, the `create_Bmoment` routine can also be used for 2D computations.

In order to compute B-moments associated with variable data, Gauss-Jacobi quadrature rules are needed. The next routine is responsible for allocating memory for storing Gauss-Jacobi quadrature weights and centres:

```
double **
create_quadraWN3d (int len_quadra);
```

When `PRECOMP` is switched on, the routine declared below is called in order to allocate memory for storing weighted data values at the Stroud nodes:

```
double **
create_matValNodes3d(int len_matValNodes);
```

What distinguishes the algorithms described in this work from those presented in [11] is the use of precomputed arrays of the form (2.21). The latter are stored in an array called `precomp` which is created by means of the routine declared below:

```
double **
create_precomp3d(int len_precomp);
```

The next routine is used to free the memory allocated to binomial coefficients:

```
void
delete_BinomialMat(double **binomialMat, int len_binomialMat);
```

In the above routine, `binomialMat` is used to store binomial coefficients, whereas `len_binomialMat` is the size allocated to `binomialMat`.

The next routine frees the memory allocated to B-moments:

```
void
```

```
delete_Bmoment(double **Bmoment);
```

The next routine is used to free the memory allocated by `create_matValNodes3d`:

```
void
delete_matValNodes(double **matValNodes);
```

The following routine is needed in order to free the memory allocated to auxiliary arrays involved in the computation of the B-moments:

```
#ifdef PRECOMP
void
delete_pointers_Bmom(double **precomp, double **matValNodes,
    double **quadraWN);
#else
void
delete_pointers_Bmom(double **BmomentInter, double
    **quadraWN);
#endif
```

The next routine frees the memory allocated to precomputed arrays used when PRECOMP is switched on:

```
void
delete_precomp(double **precomp);
```

The following routine is used to free the memory allocated to Gauss-Jacobi quadrature weights and centres:

```
void
delete_quadraWN (double **quadraWN);
```

As mentioned before, the size allocated to the array storing the B-moments depends on whether or not the algorithms presented in [11] are used. More precisely, the following routine returns the size allocated to the array containing the B-moments:

```
1 int
```

```
2 len_Moments3d(int n, int q);
```

In the above routine, `n` is the order of the computed B-moments, whereas `q` is the order of the Stroud rule used.

B.2.1.3 Auxiliary Computations

This section describes the auxiliary routines which are needed in order to compute the B-moments computation. These routines are listed as: `assign_pointers_Bmom3d`, `assign_quadra3d`, `bary2cart3d`, `Bmoment3d`, `Bmoment3d_Index`, `computeBinomials`, `data_at_Nodes_Bmom3d`, `data_at_Nodes_Cval3d`, `gaussJacobiUnit3D`, `init_Bmoment3d_Cval`, `init_BmomentC_Bmom3d`, `init_precomp3d`, `position3d`, `position3d2`, `scalar_values_at_Stroud3d`, `stroud_nodes_bary3d` and `Volume3d`.

The routine declared below is responsible for initializing intermediate arrays used in the computation of the B-moments:

```
1 #ifdef PRECOMP
2 void
3 assign_pointers_Bmom3d (double v1[3], double v2[3], double
    v3[3], double v4[3], int n, int q, int nDash, int m, int
    nb_Array, double **matValNodes, double *Cval, double
    **quadraWN, double **precomp, double (*f) (double[3]), int
    functval);
4 #else
5 void
6 assign_pointers_Bmom3d (double v1[3], double v2[3], double
    v3[3], double v4[3], int n, int q, int nDash, int m, int
    nb_Array, double *Cval, double **quadraWN, double
    **Bmoment, double (*f) (double[3]), int functval);
7 #endif
```

In the above routine, `v1`, `v2`, `v3`, `v4` are the tetrahedron's vertices, `q` is the order of the Stroud rule, `quadraWN` stores Gauss-Jacobi quadrature weights and cen-

tres, `nb_Array` is a variable used to specify that the computed B-moments are associated with scalar-valued data, and `n` is the order of the B-moments. When `PRECOMP` is switched on, the values of the coefficients at the Stroud nodes is stored into `matValNodes`. Otherwise, they are stored into `Bmoment`. Depending on the value of the flag `functval`, either the function `f` or the array `Cval` is used as input for the B-moments coefficients.

The routine `assign_quadra3d` is used to initialize Gauss-Jacobi quadrature weights and nodes, and is declared as follows:

```
1 void
2 assign_quadra3d(int q, double **quadraWN );
```

The above routine stores the Gauss-Jacobi quadrature weights and centres of order `q` needed in the computation of the 3D elemental quantities into the array `quadraWN`.

The next routine transforms barycentric coordinates into Cartesian coordinates, and is needed when evaluating data values at Stroud nodes:

```
1 void
2 bary2cart3d(double b1, double b2, double b3, double b4,
      double v1[3], double v2[3], double v3[3], double v4[3],
      double v[3]);
```

The above routine computes the Cartesian coordinates corresponding to the barycentric coordinates (`b1`, `b2`, `b3`, `b4`).

The low-level routine for computing B-moments based on the algorithms presented in Section 2.2.4 is declared below:

```
1 double
2 Bmoment3d (int n, int q, int nb_Array, double v1[3], double
      v2[3], double v3[3], double v4[3], double **binomialMat,
      double **precomp, double **Bmoment, double **matValNodes);
```

The above routine computes B-moments of order `n` on the tetrahedron with vertices `v1`, `v2`,`v3`,`v4`, using the Stroud rule of order `q`. The array `binomialMat` is

used to store precomputed binomial coefficients, whereas `precomp` contains the precomputed arrays defined in (2.21). The coefficients values at the Stroud nodes are stored into the array `matValNodes`. The parameter `nb_Array` allows the above routine to handle scalar-, vector- or matrix-valued coefficients.

The following routine for computing B-moments is derived the algorithms presented in [11]:

```
double
Bmoment3d_Index( int n, int q, int nb_Array, double
    **Bmoment, double **BmomentInter,double **quadraWN );
```

The above routine is the analogue of the previous one, when using the algorithms without precomputed arrays. More precisely, `Bmoment3d_Index` is derived from [11, Algorithm 3] with $d = 3$.

The next routine is used for precomputing binomial coefficients, using Pascal's Triangle:

```
void
computeBinomials(double **binomialMat, int len_binomialMat);
```

In the above routine, the computed binomial coefficients are stored into the array `binomialMat`, whereas `len_binomialMat` specifies the size allocated to `binomialMat`.

The routine described below is used to initialize the entries of the coefficients at the Stroud nodes when `PRECOMP` is switched on:

```
void
data_at_Nodes_Bmom3d ( double (*f) (double[3]), double
    **matValNodes, int q, double **quadraWN, double v1[3],
    double v2[3], double v3[3], double v4[3] );
```

The routine `data_at_Nodes_Bmom3d` computes the value of the coefficient `f` at the Stroud nodes of order `q`. The computed coefficients values are then stored into the array `matValNodes`.

When the input for the B-moments coefficients is given by an array of data values at the Stroud nodes, the next routine is used instead of `data_at_Nodes_Bmom3d`:

```
void
data_at_Nodes_Cval3d(double **matValNodes, int q, double
    *Cval, int nb_Array);
```

The above routine initializes the entries of `matValNodes` by reading the coefficients values stored in `Cval`. The parameter `q` stands for the order of the Stroud rule used, whereas `nb_Array` allows `data_at_Nodes_Cval3d` to handle scalar, vector, or matrix-valued coefficients.

When `PRECOMP` is switched off, the following routine is used to compute the Gauss-Jacobi quadrature rules defined on the interval $[0, 1]$:

```
void
gaussJacobiUnit3D(int q, double **quadraWN );
```

The above routine stores the Gauss-Jacobi quadrature weights and centres of order `q` defined on the unit interval into the array `quadraWN`.

When the input for the B-moments coefficients is given by an array of data values at the Stroud nodes, the following routine is responsible for initializing the B-moments:

```
void
init_Bmoment3d_Cval (double *Cval, int q,double v1[3], double
    v2[3], double v3[3], double v4[3], double **BmomentInter,
    int nb_Array);
```

In the above routine, the coefficients values at the Stroud nodes of order `q` are stored in `Cval`, whereas `v1`, `v2`, `v3`, `v4` are the tetrahedron's vertices. The parameter `nb_Array` allows for `init_Bmoment3d_Cval` to be used with scalar-, vector-, or matrix-valued data. In particular, this routine can also be used for computing the convective and stiffness matrices associated with variable coefficients.

When a function serves as input for the coefficients, the routine described

below is used instead of `init_Bmoment3d_Cval`:

```
void
init_BmomentC_Bmom3d ( double (*f) (double [3]), int q,
    double v1[3], double v2[3], double v3[3], double v4[3],
    double **BmomentInter, double **quadraWN);
```

In the above routine, `f` is a scalar-valued function which produces the B-moments coefficients, and `quadraWN` contains Gauss-Jacobi quadrature weights and centres of order `q`. The routine `init_BmomentC_Bmom3d` stores the values of `f` at the Stroud nodes of order `q` into the array `BmomentInter`.

The next routine initializes the values assigned to the array `precomp` which is used to store the precomputed arrays defined in (2.21):

```
void
init_precomp3d(double **precomp, int n, int q, int mp, double
    **quadraWN);
```

The above routine computes the arrays defined in (2.21) into `precomp`.

When `PRECOMP` is switched on, the routines used for indexing the entries of the B-moments is displayed below:

```
int
position3d(int eta1, int eta2, int eta3);
```

Note that the above routine is also used for storing the entries of the elemental matrices, regardless of whether or not PRECOMP is used.

When `PRECOMP` is switched off, the array storing B-moments is also used to store coefficient values at the Stroud nodes, hence the need for the alternative indexing routine `position2d2` declared as:

```
int
position3d2(int i, int j, int k, int n);
```

The next routine is responsible for initializing the array containing the values of the data at the quadrature nodes:

```
1 void
2 scalar_values_at_Stroud3d(int q, double *Cval, double *B,
      double (*f)(double[3]), double v1[3], double v2[3], double
      v3[3], double v4[3] );
```

In the above routine, `B` contains the barycentric coordinates of the Stroud nodes of order `q`. The routine `scalar_values_at_Stroud3d` stores the values of the function `f` at the Stroud nodes of order `q` into the array `Cval`.

In order to evaluate B-moments associated with variable coefficients, it is necessary to evaluate the data at the Stroud nodes. The next routine returns the barycentric coordinates of the q-point Stroud nodes:

```
1 void
2 stroud_nodes_bary3d (int q, double *B);
```

The above routine stores the barycentric coordinates of the Stroud nodes of order `q` into the array `B`.

The next routine computes the volume of a tetrahedron which is needed for defining scaling constants involved in various intermediate computations:

```
1 double
2 Volume3d(double v1[3], double v2[3], double v3[3], double
      v4[3]);
```

In the above routine, `v1`, `v2`, `v3`, `v4` are the tetrahedron's vertices.

B.2.1.4 Code Execution

This section presents an example which illustrates how to make use of the routines described in the previous sections in order to compute the B-moments:

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <time.h>
4 #include <iostream>
```

```
5 #include "bbfem.h"

7 #ifndef MAX // a template
8 #define MAX(a,b) ( (a) > (b) ? (a) : (b) )
9 #endif

11 // example of coefficient function
12 double
13 f0(double v[3])
14 {
15   return sin(v[0]*v[1]*v[2]);
16 }

18 int main()
19 {
20 //  // vertices (standard tetrahedron)
21 //       double v1[3] = {0, 0, 0};
22 //       double v2[3] = {1, 0, 0};
23 //  double v3[3] = {0, 1, 0};
24 //  double v4[3] = {0, 0, 1};

26   //vertices (particular tetrahedron)
27   double v1[3] = { 1.2   , 3.4, 0};
28   double v2[3] = { -1.5 , 2. , 0};
29   double v3[3] = { 0.1   , -1., 0};
30   double v4[3] = {1. , 1., 1.};
31   int n; // order of the B-moments
32   std::cout<<"Enter a value for the polynomial order n:";
33   std::cin>>n;
34   int q=n+1; // q is for the number of quadrature points used
       in each direction

36   double **Bmoment; // pointer used for Bmoment entries
```

```
37  int lenMoments = len_Moments3d(n, q); // memory required
        for storing Bmoments depends on whether or not PRECOMP
        is used
38  int nb_Array = 1; // the Bmoments are associated with a
        scalar-valued function
39  double (*f) (double[3]) = f0; // change here to your
        routine for the B-moments coefficients
40  double *Cval; // store coefficients values at Stroud nodes,
        only used with FUNCT_VAL
41  int functval = 0; //default: using a routine (f) for
        B-moments' coefficients

43  #ifdef FUNCT_VAL
44  functval = 1;  //using the values stored in Cval for
        B-moments' coefficients
45  double *B; // store barycentric coordinates of Stroud nodes
46  B = new double [q*q*q*4];
47  stroud_nodes_bary3d (q, B);
48  int LEN = q * q * q;  // space required for 2D array with
        dimension q+1
49  Cval = new double[LEN * nb_Array];     //Cval entries are
        stored in linear memory
50  #endif

52  #ifdef FUNCT_VAL
53  scalar_values_at_Stroud3d(q, Cval, B, f, v1, v2, v3, v4);
        // storing your data in Cval
54  #endif

56  Bmoment = create_Bmoment(lenMoments, nb_Array); //allocate
        memory to Bmoments
57  get_Bmoments3d(Bmoment, n, f, Cval, v1, v2, v3, v4,
        functval); // store Bmoments into Bmoment
```

```
59   //free memory allocated memory
60   #ifdef FUNCT_VAL
61   delete[] Cval;
62   delete[] B;
63   #endif


66   // Insert your code here to make use of Bmoment. It will be
         destroyed in the next line!


68   delete_Bmoment(Bmoment);
69 }
```

Listing B.1: bmom3d.cpp

In the above example, the macro `FUNCT_VAL` is used when `Cval` serves as input for the B-moments coefficients.

B.2.2 H^1 Mass Matrix

This section describes the routines which are involved with the computation of the mass matrix associated with a tetrahedron $T =< \mathbf{v}_i,\ i = 1, \ldots, 4 >$. Depending on whether or not the data is variable, two driver routines are proposed. As in the B-moments case, a parameter `functval` serves as a flag for the type of input used for variable coefficients.

Similarly to Section B.2.1, we first start with the driver routines used for computing the elemental mass matrix. The routines responsible for allocating memory are then discussed in Section B.2.2.2. Section B.2.2.3 then focuses on the routines used for intermediate computations. Finally, an example on how to execute the mass matrix computation is given in Section B.2.2.4.

B.2.2.1 Driver Routines

This section focuses on on the driver routines used for computing the mass matrix.

The following routine computes the mass matrix associated with constant coefficients equal to 1:

```
void
get_mass3d_const(double **massMat, int n, double v1[3],
    double v2[3],
double v3[3], double v4[3]);
```

The above routine computes the mass matrix of order `n` associated with constant coefficients equal to 1 on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`. The computed mass matrix is stored into the array `massMat`.

The next routine is used to compute the mass matrix associated with variable coefficients:

```
void
get_mass3d(double **massMat, int n, double (*f)(double[3]),
    double *Cval, double v1[3], double v2[3], double v3[3],
    double v4[3], int functval);
```

Listing B.2: get_mass3d

In the above routine, recall that the parameter `functval` is used as a flag for setting the input used for the coefficients: With `functval`=0, the mass matrix coefficients are produced by the function `f`, whereas with `functval`=1, the mass matrix coefficients are produced by the array `Cval` which contains the data values at the Stroud nodes. The routine `get_mass3d` computes the mass matrix of order `n` on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`, associated with either `f` or `Cval`. The computed mass matrix is stored into the array `massMat`.

B.2.2.2 Memory Allocation

This section focuses on the mass matrix routines which are used to dynamically allocate memory. The mass matrix routines involved in allocating memory are listed as: `create_BinomialMat`, `create_Bmoment`, `create_quadraWN3d`, `create_Mat`, `create_matValNodes3d`, `create_precomp3d`, `delete_BinomialMat`, `delete_Bmoment`, `delete_Mat`, `delete_matValNodes`, `delete_pointers_Mass`, `delete_precomp`, `delete_quadraWN` and `len_Mat3d`. Note that most routines are described in Section B.2.1.2. Indeed, `create_BinomialMat`, `create_Bmoment`, `create_quadraWN3d`, `create_matValNodes3d`, `create_precomp3d`, `delete_BinomialMat`, `delete_Bmoment`, `delete_matValNodes`, `delete_precomp`, and `delete_quadraWN` are also used in the B-moments computation. Hence, only `create_Mat`, `delete_Mat`, `delete_pointers_Mass` and `len_Mat3d` are discussed in this section.

The next routine is used to allocate memory to the mass matrix:

```
double **
create_Mat(int len_Mat);
```

The above routine allocates memory to a square matrix of dimension `len_Mat`. By adjusting the value of `len_Mat`, `create_Mat` can also be used for 2D computations.

The routine `delete_Mat` frees the memory allocated by `create_Mat`, and is declared as:

```
void
delete_Mat(double **Mat);
```

The routine which frees the memory allocated to intermediate arrays involved in the computing the mass matrix is declared as follows:

```
#ifdef PRECOMP
void
delete_pointers_Mass(double **precomp, double **Bmoment,
    double **matValNodes, double **quadraWN);
```

```
#else
void
delete_pointers_Mass(double **Bmoment, double **BmomentInter,
    double **quadraWN);
#endif
```

The next routine is used to determine the size allocated to the mass matrix:

```
int
len_Mat3d(int n);
```

The above routine actually returns the dimension of $\mathbb{P}_3^n$.

B.2.2.3 Auxiliary Computations

This section describes the auxiliary routines which are involved in the mass matrix computation. These routines are listed as: `assign_pointers_Mass3d`, `assign_quadra3d`, `bary2cart3d`, `Bmoment3d`, `Bmoment3d_Index`, `computeBinomials`, `data_at_Nodes_Cval3d`, `data_at_Nodes_Mass3d`, `gaussJacobiUnit3D`, `init_Bmoment3d_Cval`, `init_BmomentC_Mass3d`, `init_precomp3d`, `Mass3d`, `Mass3d_const`, `position3d`, `position3d2`, `scalar_values_at_Stroud3d`, `stroud_nodes_bary3d` and `Volume3d`. Note that most routines are defined in Section B.2.1.3. Indeed, `assign_quadra3d`, `bary2cart3d`, `Bmoment3d`, `Bmoment3d_Index`, `computeBinomials`, `data_at_Nodes_Cval3d`, `gaussJacobiUnit3D`, `init_Bmoment3d_Cval`, `init_precomp3d`, `position3d`, `position3d2`, `scalar_values_at_Stroud3d`, `stroud_nodes_bary3d` and `Volume3d` are also used in the B-moments computation. Hence, only `assign_pointers_Mass3d`, `data_at_Nodes_Mass3d`, `init_BmomentC_Mass3d`, `Mass3d` and `Mass3d_const` are discussed in this section.

The next routine is used in order to allocate memory to auxiliary arrays involved in the computation of the mass matrix:

```
#ifdef PRECOMP
void
```

```
assign_pointers_Mass3d (double v1[3], double v2[3], double
    v3[3], double v4[3], int n, int q, int nDash, int m, int
    nb_Array, double **matValNodes, double *Cval, double
    **quadraWN,
double **precomp, double (*f) (double[3]), int functval);
#else
void
assign_pointers_Mass3d (double v1[3], double v2[3], double
    v3[3], double v4[3], int n, int q, int nDash, int m, int
    nb_Array, double *Cval, double **quadraWN, double
    **Bmoment, double (*f) (double[3]), int functval);
#endif
```

In the above routine, `v1`, `v2`, `v3`, `v4` are the tetrahedron's vertices, `q` is the order of the Stroud rule, `quadraWN` stores Gauss-Jacobi quadrature weights and centres, `nb_Array` is a variable used to specify that the computed B-moments are associated with scalar-valued data, and `n` is the order of the B-moments. Depending on the value of the flag `functval`, either the function `f` or the array `Cval` is used as input for the mass matrix coefficients.

When `PRECOMP` is switched on, the following routine is used to compute the values of the mass matrix coefficients at the Stroud nodes:

```
void
data_at_Nodes_Mass3d ( double (*f) (double[3]), double
    **matValNodes, int q, double **quadraWN, double v1[3],
    double v2[3], double v3[3], double v4[3] );
```

The above routine has the same structure as `data_at_Nodes_Bmom3d`. More precisely, `f` is a scalar-valued function which produces the mass matrix coefficients, `quadraWN` contains Gauss-Jacobi quadrature weights and centres of order `q`, and the tetrahedron's vertices are given by `v1`, `v2`, `v3`, `v4`. The computed coefficients values are stored into `matValNodes`.

When `PRECOMP` is switched off, the routine described below is used to initialize

the B-moments with the values of the mass matrix coefficients at the Stroud nodes:

```
void
init_BmomentC_Mass3d ( double (*f) (double [3]), int q,
    double v1[3], double v2[3], double v3[3], double v4[3],
    double **BmomentInter, double **quadraWN);
```

In the above routine, `f` is a scalar-valued function which produces the mass matrix coefficients, and `quadraWN` contains Gauss-Jacobi quadrature weights and centres of order `q`. The routine `init_BmomentC_Mass3d` stores the values of `f` at the Stroud nodes of order `q` into the array `BmomentInter`.

The next routine computes the mass matrix associated with constant coefficients:

```
double
Mass3d_const (int n, double v1[3], double v2[3], double
    v3[3], double v4[3], double **binomialMat, double
    **massMat);
```

The above routine computes the mass matrix of order `n` associated with constant coefficients equal to 1 on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`. The array `binomialMat` contains precomputed binomial coefficients, whereas `massMat` is used to store the computed mass matrix.

The following routine computes the mass matrix associated with variable coefficients:

```
#ifdef PRECOMP
double
Mass3d (int n, int q, double v1[3], double v2[3], double
    v3[3], double v4[3], double **binomialMat, double
    **precomp, double **Bmoment, double **massMat, double
    **matValNodes, double **quadraWN);
#else
double
```

```
Mass3d (int n, int q, double v1[3], double v2[3], double
    v3[3], double v4[3], double **binomialMat, double
    **Bmoment, double **BmomentInter, double **massMat, double
    **quadraWN);
#endif
```

The above routine implements Algorithm 3.19 when `PRECOMP` is switched on, and [11, Algorithm 6] with $d = 3$ otherwise. The main difference between the two algorithms lies in the approach for computing the B-moments associated with the mass matrix coefficients. Indeed, `PRECOMP` makes use of the precomputed arrays defined in (2.21). As a result, the arguments of the routine `Mass3d` depend on whether or not `PRECOMP` is used.

B.2.2.4 Code Execution

This section displays an example which illustrates how to make use of the routines defined in the previous sections in order to compute the mass matrix:

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <time.h>
4 #include <iostream>

6 #include "bbfem.h"

8 #ifndef MAX
9 #define MAX(a,b) ( (a) > (b) ? (a) : (b) )
10 #endif

12 // example of coefficient function
13 double
14 f0(double v[3])
15 {
16   //return 1.;
```

```
17      return sin(v[0]*v[1]*v[2]);
18  }


21  #ifdef CONSTANT
22  int main()
23  {
24  //  //vertices (standard tetrahedron)
25  //  double v1[3] = { 0, 0, 0};
26  //  double v2[3] = { 1, 0, 0};
27  //  double v3[3] = { 0, 1, 0};
28  //  double v4[3] = { 0, 0, 1};

30    //vertices (particular tetrahedron)
31    double v1[3] = { 1.2   , 3.4, 0};
32    double v2[3] = { -1.5 , 2. , 0};
33    double v3[3] = { 0.1   , -1., 0};
34    double v4[3] = {1. , 1., 1.};

36    int n; // degree of the Bernstein polynomial basis
37    std::cout<<"Enter a value for the polynomial order n:";
38    std::cin>>n;

40    double **massMat; // used for storing mass matrix entries
41    int len_Mass = len_Mat3d(n); // allocate memory for massMat
42    massMat = create_Mat(len_Mass);

44    get_mass3d_const(massMat, n, v1, v2, v3, v4); // compute
          mass matrix

46    // Insert your code here to make use of massMat. It will be
          destroyed in the next line!
```

```
48    // free allocated memory
49    delete_Mat(massMat);

51 }

53 #else // not CONSTANT

55 int main()
56 {
57 //  //vertices (standard tetrahedron)
58 //  double v1[3] = { 0, 0, 0};
59 //  double v2[3] = { 1, 0, 0};
60 //  double v3[3] = { 0, 1, 0};
61 //  double v4[3] = { 0, 0, 1};

63    //vertices (particular tetrahedron)
64    double v1[3] = { 1.2   , 3.4, 0};
65    double v2[3] = { -1.5 , 2. , 0};
66    double v3[3] = { 0.1   , -1., 0};
67    double v4[3] = {1. , 1., 1.};

69    int n; // degree of the Bernstein polynomial basis
70    std::cout<<"Enter a value for the polynomial order n:";
71    std::cin>>n;

73    double (*f) (double[3]) = f0; // change here to your
          routine for the mass matrix coefficients
74    double *Cval; // store coefficients values at Stroud nodes,
          only used with FUNCT_VAL

76    int functval = 0; //default: using a routine (f) for mass
          matrix coefficients
77    #ifdef FUNCT_VAL
```

```
78  functval = 1;  //using the values stored in Cval for
        convective matrix coefficients
79  int q = n+1;
80  int nb_Array = 1; // the mass matrix is associated with
        scalar-valued data
81  double *B; // store barycentric coordinates of Stroud nodes
82  B = new double [q*q*q*4];
83  stroud_nodes_bary3d (q, B);
84  int LEN = q * q * q;  // space required for 3D array with
        dimension q+1
85  Cval = new double[LEN * nb_Array];     //Cval entries are
        stored in LINEAR memory, and used directly
86  #endif

88  #ifdef FUNCT_VAL
89  scalar_values_at_Stroud3d(q, Cval, B, f, v1, v2, v3, v4);
        // storing your data in Cval
90  #endif

92  double **massMat; // used for storing mass matrix entries
93  int len_Mass = len_Mat3d(n); // allocate memory for massMat
94  massMat = create_Mat(len_Mass);

96  get_mass3d(massMat, n, f, Cval, v1, v2, v3, v4, functval);
        // compute mass matrix

98  // free allocated memory
99  #ifdef FUNCT_VAL
100 delete[] Cval;
101 delete[] B;
102 #endif
```

```
104    // Insert your code here to make use of massMat. It will be
           destroyed in the next line!

106    delete_Mat(massMat);

108 }
109 #endif // end not CONSTANT
```

Listing B.3: mass3d.cpp

As mentioned previously, recall that the macro `FUNCT_VAL` is used when `Cval` serves as input for the mass matrix coefficients. When `CONSTANT` is switched on, the code executes the computation of the mass matrix associated with constant coefficients. By default, the mass matrix associated with variable coefficients is computed.

B.2.3 H^1 Stiffness Matrix

This section discusses the routines which are involved with the computation of the stiffness matrix associated with a tetrahedron $T =< \mathbf{v}_i,\ i = 1, \ldots, 4 >$. Depending on whether or not the data is variable, two driver routines are proposed. Recall that a parameter `functval` serves as a flag for the type of input used for variable coefficients.

This section is organized as follows: Section B.2.3.1 first describes the driver routines used for computing the elemental stifness matrix. The routines involved in allocating memory to auxiliary arrays used in the stiffness matrix computation are then presented in Section B.2.3.2. Section B.2.3.3 then focuses on the routines used for intermediate computations. Finally, an example on how to compute the stiffness matrix using the proposed routines is given in Section B.2.3.4.

B.2.3.1 Driver Routines

This section focuses on on the driver routines used for computing the stiffness matrix.

The following routine computes the stiffness matrix associated with constant coefficients:

```
void
get_stiffness3d_const (double **stiffMat, int n, double
    v1[3], double v2[3], double v3[3], double v4[3], double
    Coeff[6]);
```

The routine `get_stiffness3d_const` computes the stiffness matrix of order `n` associated with constant matrix-valued coefficients on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`. In the above routine, `Coeff` contains the upper triangular entries of the stiffness matrix coefficients. The computed stiffness matrix is stored into the array `stiffMat`.

The next routine is used to compute the stiffness matrix associated with variable coefficients:

```
void
get_stiffness3d(double **stiffMat, int n, void (*A) (double
    [3], double [3][3]), double *Cval, double v1[3], double
    v2[3], double v3[3], double v4[3], int functval);
```

In the above routine, recall that the parameter `functval` is used as a flag for setting the input used for the coefficients: With `functval`=0, the stiffness matrix coefficients are produced by the (symmetric) matrix-valued function `A`, whereas with `functval`=1, the B-moments coefficients are produced by the array `Cval` which contains the data values at the Stroud nodes. The routine `get_Stiff3d` computes the stiffness matrix of order `n` on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`, associated with either `A` or `Cval`. The computed stiffness matrix is stored into the array `stiffMat`.

B.2.3.2 Memory Allocation

This section focuses on the stiffness matrix routines which are used to dynamically allocate memory. The stiffness matrix routines involved in allocating memory are listed as: `create_BinomialMat`, `create_Bmoment`, `create_quadraWN3d`, `create_Mat`, `create_matValNodes3d`, `create_precomp3d`, `delete_BinomialMat`, `delete_Bmoment`, `delete_Mat`, `delete_matValNodes`, `delete_pointers_Stiff`, `delete_precomp`, `delete_quadraWN` and `len_Mat3d`. Note that, with the exception of `delete_pointers_Stiff`, all the above routines are discussed in Section B.2.2.2. Hence, only `delete_pointers_Stiff` is displayed in this section.

The next routine is used to free the memory allocated to auxiliary arrays used in the stiffness matrix computation:

```
#ifdef PRECOMP
void
delete_pointers_Stiff (double **precomp, double **Bmoment,
    double **Bmomentab, double **matValNodes, double
    **quadraWN);
#else
void
delete_pointers_Stiff (double **Bmoment, double
    **BmomentInter, double **Bmomentab, double **quadraWN);
#endif
```

B.2.3.3 Auxiliary Computations

This section is focused on the auxiliary routines which are involved in the stiffness matrix computation. These routines are listed as: `assign_pointers_Stiff3d`, `assign_quadra3d`, `bary2cart3d`, `Bmoment3d`, `crossProd2`, `Bmoment3d_Index`, `computeBinomials`, `data_at_Nodes_Cval3d`, `data_at_Nodes_Stiff3d`, `gaussJacobiUnit3D`, `init_Bmoment3d_Cval`, `init_BmomentC_Stiff3d`, `init_precomp3d`, `inter`, `matrix_values_at_Stroud3d`, `normals3d`, `position3d`, `position3d2`, `position3d_sum`, `position3d_sum2`, `scalarMatrix3d_Coeff`,

`Stiff3d`, `Stiff3d_const`, `subtract`, `transform_BmomentC_Stiff3d`, `stroud_nodes_bary3d` and `Volume3d`. Note that most routines are defined in Section B.2.1.3. Indeed, `assign_quadra3d`, `bary2cart3d`, `Bmoment3d`, `Bmoment3d_Index`, `computeBinomials`, `data_at_Nodes_Cval3d`, `gaussJacobiUnit3D`, `init_Bmoment3d_Cval`, `init_precomp3d`, `position3d`, `position3d2`, `stroud_nodes_bary3d` and `Volume3d` are also involved in the computation of the B-moments. As a result, only `assign_pointers_Stiff3d`, `crossProd2`, `data_at_Nodes_Stiff3d`, `init_BmomentC_Stiff3d`, `inter`, `matrix_values_at_Stroud3d`, `normals3d`, `position3d_sum`, `position3d_sum2`, `scalarMatrix3d_Coeff`, `Stiff3d`,`Stiff3d_const`, `subtract` and `transform_BmomentC_Stiff3d` are described in this section.

The routine given below is used to compute the auxiliary arrays needed in the stiffness matrix computation:

```
#ifdef PRECOMP
void
assign_pointers_Stiff3d (double v1[3], double v2[3], double
    v3[3], double v4[3], int n, int q, int nDash, int m, int
    nb_Array, double **matValNodes, double *Cval, double
    **quadraWN, double **precomp, void (*A) (double[3],
    double[3][3]), int functval);
#else
void
assign_pointers_Stiff3d (double v1[3], double v2[3], double
    v3[3], double v4[3], int n, int q, int nDash, int m, int
    nb_Array, double *Cval, double **quadraWN, double
    **Bmoment, void (*A) (double[3], double[3][3]), int
    functval);
#endif
```

In the above routine, `v1`, `v2`, `v3`, `v4` are the tetrahedron's vertices, `q` is the order of the Stroud rule, `quadraWN` stores Gauss-Jacobi quadrature weights and centres, `nb_Array` is a variable used to specify that the computed B-moments are

associated with matrix-valued data, and `n` is the order of the B-moments. Depending on the value of the flag `functval`, either the matrix-valued function `A` or the array `Cval` is used as input for the stiffness matrix coefficients. The output of `assign_pointers_Stiff3d` is stored into `matValNodes` if `PRECOMP` is switched on, and in `Bmoment` otherwise.

The routine declared below computes the half-scaled cross-product of two vectors:

```
void
crossProd2(double w1[3], double w2[3], double Cross[3]);
```

In the above routine, the half-scaled cross-product of the vectors `w1` and `w2` is stored into `Cross`. The details of `crossProd2` are given in Listing **??**.

When `PRECOMP` is switched on, the following routine is used to compute the values of the stiffness matrix coefficients at the Stroud nodes:

```
void
data_at_Nodes_Stiff3d ( void (*A)(double[3],double[3][3]),
    double **matValNodes, int q, double **quadraWN, double
    v1[3], double v2[3], double v3[3], double v4[3] );
```

Note that the above routine has similar arguments as `data_at_Nodes_Bmom3d`. More precisely, `A` is a matrix-valued function which produces the stiffness matrix coefficients, `quadraWN` contains Gauss-Jacobi quadrature weights and centres of order `q`, and the tetrahedron's vertices are given by `v1`, `v2`, `v3`, `v4`. The computed coefficients values are stored into `matValNodes`.

When `PRECOMP` is switched off, the routine described below is used to initialize the B-moments with values of the stiffness matrix coefficients at the Stroud nodes:

```
void
init_BmomentC_Stiff3d ( void (*A) (double[3], double[3][3]),
    int q, double v1[3], double v2[3], double v3[3], double
    v4[3], double **BmomentInter, double **quadraWN);
```

In the above routine, `A` is a (symmetric) matrix-valued function which produces the stiffness matrix coefficients, and `quadraWN` contains Gauss-Jacobi quadrature weights and centres of order `q`. The routine `init_BmomentC_Mass3d` stores the values of `A` at the Stroud nodes of order `q` into the array `BmomentInter`.

The following routine compute intermediate values needed for computing the normals to the element's faces:

```
void
inter( double u[3], double v[3], double w[3], double Res[3] );
```

The routine defined below is used to compute the values of matrix-valued coefficients at the Stroud nodes:

```
void
matrix_values_at_Stroud3d(int q, double *Cval, double *B,
    void (*A) (double[3], double[3][3]), double v1[3], double
    v2[3], double v3[3], double v4[3] );
```

The above routine stores the values of the function `A` at the Stroud nodes of order `q` into the array `Cval`.

The next routine is used to compute the normals to the faces of the element:

```
void
normals3d(double v1[3], double v2[3], double v3[3], double
    v4[3], double normalMat[4][3]);
```

The above routine computes the normals to the faces of the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`. The computed normals are stored into the array `normalMat`.

The routine described below is used for computing weighted inner products of the normals to the tetrahedron's faces:

```
void
scalarMatrix3d_Coeff(double scalarMat[][4],double v1[3],
    double v2[3], double v3[3], double v4[3] , double
    normalMat[][3], double Coeff[6] );
```

The above routine computes products of the form $\mathbf{n}_i \cdot \mathbf{A} \cdot \mathbf{n}_j$, where $\mathbf{n}_\ell$ denotes the outer normals to the faces of the tetrahedron $\langle \mathtt{v1}, \mathtt{v2}, \mathtt{v3}, \mathtt{v4} \rangle$ for $1 \leq \ell \leq 4$, and $\mathbf{A}$ is the value of the constant coefficients associated with the stiffness matrix. In the above routine, the upper triangular entries of $\mathbf{A}$ are stored in the array `Coeff`, whereas the normals are stored in `normalMat`. The computed products are stored into `scalarMat`.

The routines `position3d_sum` and `position3d_sum2` are declared as follows:

```
int
position3d_sum(int eta123, int xi123);

int
position3d_sum2(int eta23, int xi23);
```

The routines `position3d_sum` and `position3d_sum2` are auxiliary indexing routines.

The following routine is used for computing the stiffness matrix associated with constant coefficients:

```
double
Stiff3d_const (int n, double v1[3], double v2[3], double
    v3[3], double v4[3], double **stiffMat, double
    **binomialMat, double scalarMat[][4], double
    normalMat[][3], double cpu_time[5], double Coeff[6]);
```

The above routine computes the stiffness matrix of order `n` associated with constant coefficients on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`. `Coeff` contains the upper triangular entries of the constant matrix-valued coefficients associated with the stiffness matrix. The computed stiffness matrix is stored into the array `stiffMat`.

The next routine is used for the computation of the stiffness matrix associated with variable coefficients:

```
#ifdef PRECOMP
double
```

```
Stiff3d (int n, int q, double v1[3], double v2[3], double
    v3[3], double v4[3], double **stiffMat, double
    **matValNodes, double **quadraWN, double **binomialMat,
    double normalMat[][3], double **precomp, double **Bmoment,
    double **Bmomentab, double cpu_time[5] );
#else
double
Stiff3d (int n, int q, double v1[3], double v2[3], double
    v3[3], double v4[3], double **stiffMat, double **quadraWN,
    double **binomialMat, double normalMat[][3], double
    **Bmoment, double **BmomentInter, double **Bmomentab,
    double cpu_time[5] );
#endif
```

`Stiff3d` computes the stiffness matrix of order `n` on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4` associated with variable coefficients. The coefficients values at the Stroud nodes of order `q` are stored in `matValNodes` if PRECOMP is switched on, and in `BmomentInter` otherwise. `Bmoment` contains the B-moments of order $2\mathtt{n}-2$ associated with the stiffness matrix coefficients. The array `Bmomentab` contains the products of the matrix-valued B-moments with the normals to the tetrahedron's faces which are stored in `normalMat`. The routine `Stiff3d` implements Algorithm 3.22 when `PRECOMP` is switched on, and [11, Algorithm 7] with $d = 3$ otherwise. The main difference between the two algorithms lies in the approach for computing the B-moments associated with the stiffness matrix coefficients. Indeed, `PRECOMP` makes use of the precomputed arrays defined in (2.21). As a consequence, the arguments of the routine `Stiff3d` depend on whether or not `PRECOMP` is used.

The routine declared below computes the difference of two 3D vectors:

```
void
subtract( double v[3], double w[3], double Sub[3]);
```

The above routine computes the difference between the vectors `v` and `w`.

The next routine is used to compute the products of matrix-valued B-moments with the normals to the tetrahedron's faces:

```
void
transform_BmomentC_Stiff3d (int n, int q, double **Bmoment,
    double **Bmomentab, double normalMat[4][3]);
```

The above routine computes the products of matrix-valued B-moments of order `n` with the normals to the tetrahedron's faces In the above routine, `q` is the order of the Stroud rule, `Bmoment` is used to store the B-moments, and `normalMat` contains the normals to the tetrahedron's faces. The computed products are stored into the array `Bmomentab`.

B.2.3.4 Code Execution

This section displays an example which illustrates how to make use of the routines defined in the previous sections in order to compute the stiffness matrix:

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <time.h>
4 #include <iostream>
5 #include "bbfem.h"

7 #ifndef MAX // a template
8 #define MAX(a,b) ( (a) > (b) ? (a) : (b) )
9 #endif

11 // example of matrix-valued coefficient matrix A
12 void
13 A0(double v[3], double matC[3][3])
14 {
15 // // identity matrix
16 //    matC[0][0]=1; matC[0][1]=0; matC[0][2]=0;
17 //    matC[1][0]=0; matC[1][1]=1; matC[1][2]=0;
```

```
18 //    matC[2][0]=0; matC[2][1]=0; matC[2][2]=1;

20   // example of non-trivial matrix
21   matC[0][0]= sin(v[0]*v[1]); matC[0][1]=0; matC[0][2]=0;
22   matC[1][0]=0; matC[1][1]=1; matC[1][2]=0;
23   matC[2][0]=0; matC[2][1]=0; matC[2][2]=exp(v[2]);
24 }


27 #ifdef CONSTANT

29 int main()
30 {
31 //  //vertices (standard tetrahedron)
32 //  double v1[3] = { 0, 0, 0};
33 //  double v2[3] = { 1, 0, 0};
34 //  double v3[3] = { 0, 1, 0};
35 //  double v4[3] = { 0, 0, 1};

37   //vertices (particular tetrahedron)
38   double v1[3] = { 1.2  , 3.4, 0};
39   double v2[3] = { -1.5 , 2. , 0};
40   double v3[3] = { 0.1  , -1., 0};
41   double v4[3] = {1. , 1., 1.};

43   int n; // degree of the Bernstein polynomial basis
44   std::cout<<"Enter a value for the polynomial order n:";
45   std::cin>>n;

47   double Coeff[6] = {1., 0., 0., 1., 0., 1.}; // upper
       triangular entries of (symmetric) matrix associated with
       stiffness matrix
```

```
49    double **stiffMat;  // store stiffness matrix entries
50    int len_Stiff = len_Mat3d(n); // allocate memory to stiffMat
51    stiffMat = create_Mat(len_Stiff);

53    get_stiffness3d_const (stiffMat, n, v1, v2, v3, v4, Coeff);
          // compute stiffness matrix

55    // Insert your code here to make use of stiffMat. It will
          be destroyed in the next line!

57    // free allocated memory;
58    delete_Mat(stiffMat);

60 }

62 #else // not CONSTANT

64 int main()
65 {
66 //  //vertices (standard tetrahedron)
67 //  double v1[3] = { 0, 0, 0};
68 //  double v2[3] = { 1, 0, 0};
69 //  double v3[3] = { 0, 1, 0};
70 //  double v4[3] = { 0, 0, 1};

72    //vertices (particular tetrahedron)
73    double v1[3] = { 1.2   , 3.4, 0};
74    double v2[3] = { -1.5 , 2. , 0};
75    double v3[3] = { 0.1   , -1., 0};
76    double v4[3] = {1. , 1., 1.};

78    int n; // degree of the Bernstein polynomial basis
79    std::cout<<"Enter a value for the polynomial order n:";
```

```
80  std::cin>>n;

82  double *Cval; // store coefficients values at Stroud
        quadrature nodes, only used with FUNCT_VAL
83  int functval = 0; //default: using a routine (A) for
        stiffness matrix coefficients
84  #ifdef FUNCT_VAL
85  functval = 1;  //using the values stored in Cval for
        stiffness matrix coefficients
86  int q = n+1;
87  int nb_Array = 6; // the stiffness matrix is associated
        with (symmetric) matrix-valued data

89  double *B; // store barycentric coordinates of Stroud nodes
90  B = new double [q*q*q*4];
91  stroud_nodes_bary3d (q, B);

93  int LEN = q * q * q;  // Cval is used to store data values
        at q^3 Stroud nodes
94  Cval = new double[LEN * nb_Array];     //Cval entries are
        stored in LINEAR memory, and used directly
95  #endif

97  void (*A) (double[3], double[3][3]) = A0; // change here to
        your routine for the stiffness matrix coefficients

99  #ifdef FUNCT_VAL
100 matrix_values_at_Stroud3d(q, Cval, B, A, v1, v2, v3, v4);
        // storing your data into Cval
101 #endif

103 double **stiffMat;  // store stiffness matrix entries
104 int len_Stiff = len_Mat3d(n); // allocate memory to stiffMat
```

```
105   stiffMat = create_Mat(len_Stiff);

107   get_stiffness3d(stiffMat, n, A, Cval, v1, v2, v3, v4,
          functval); // compute elemental stiffness matrix

109   // free allocated memory
110   #ifdef FUNCT_VAL
111   delete[] Cval;
112   delete[] B;
113   #endif

115   // Insert your code here to make use of stiffMat. It will
          be destroyed in the next line!

117   delete_Mat(stiffMat);
118 }
119 #endif // end not CONSTANT
```

Listing B.4: stiff3d.cpp

As mentioned previously, recall that the macro `FUNCT_VAL` is used when `Cval` serves as input for the stiffness matrix coefficients. When `CONSTANT` is switched on, the code executes the computation of the stiffness matrix associated with constant coefficients. By default, the stiffness matrix associated with variable coefficients is computed.

B.2.4 H^1 Convective Matrix

This section discusses the routines which are involved with the computation of the convective matrix associated with a tetrahedron $T =< \mathbf{v}_i,\ i = 1, \ldots, 4 >$. Depending on whether or not the data is variable, two driver routines are proposed. Recall that a parameter `functval` serves as a flag for the type of input used for variable coefficients.

This section is organized as follows: Section B.2.4.1 focuses on the driver routines used for computing the elemental convective matrix. The routines involved in allocating memory to auxiliary arrays used in the convective matrix computation are presented in Section B.2.4.2. The routines involved in intermediate computations are then discussed in Section B.2.4.3. Section B.2.4.4 ends with an example on how to compute the convective matrix using the proposed routines.

B.2.4.1 Driver Routines

This section focuses on on the driver routines used for computing the convective matrix.

The following routine computes the convective matrix associated with constant coefficients:

```
void
get_convec3d_const (double **convecMat, int n, double v1[3],
    double v2[3], double v3[3], double v4[3], double
    vectCoeff[3]);
```

The above routine computes the convective matrix of order `n` associated with constant coefficients equal to `vectCoeff` on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`. The computed convective matrix is stored into the array `convecMat`.

The next routine is used to compute the convective matrix associated with variable coefficients:

```
void
get_convec3d(double **convecMat, int n, void (*b) (double[3],
    double[3]), double *Cval, double v1[3], double v2[3],
    double v3[3], double v4[3], int functval);
```

Listing B.5: get_convec3d

In the above routine, recall that the parameter `functval` is used as a flag for setting the input used for the coefficients: With `functval`=0, the convective matrix coefficients are produced by the vector-valued function `b`, whereas with

`functval`=1, the B-moments coefficients are produced by the array `Cval` which contains the data values at the Stroud nodes. The routine `get_Convec3d` computes the convective matrix of order `n` on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`, associated with either `b` or `Cval`. The computed convective matrix is stored into the array `convecMat`.

B.2.4.2 Memory Allocation

This section focuses on the convective matrix routines which are used to dynamically allocate memory. The convective matrix routines involved in allocating memory are listed as: `create_BinomialMat`, `create_Bmoment`, `create_quadraWN3d`, `create_Mat`, `create_matValNodes3d`, `create_precomp3d`, `delete_BinomialMat`, `delete_Bmoment`, `delete_Mat`, `delete_matValNodes`, `delete_pointers_Convec`, `delete_precomp`, `delete_quadraWN` and `len_Mat3d`. Note that, with the exception of `delete_pointers_Convec`, all the above routines are discussed in Section B.2.2.2. Hence, only `delete_pointers_Convec` is discussed in this section.

The routine `delete_pointers_Convec` is declared as:

```
#ifdef PRECOMP
void
delete_pointers_Convec(double **precomp, double **Bmoment,
    double **Bmomentab, double **matValNodes, double
    **quadraWN);
#else
void
delete_pointers_Convec(double **Bmoment, double
    **BmomentInter, double **Bmomentab, double **quadraWN);
#endif
```

The routine `delete_pointers_Convec` frees the memory allocated to auxiliary arrays used in the convective matrix computation.

B.2.4.3 Auxiliary Computations

This section is focused on the auxiliary routines which are involved in the convective matrix computation. These routines are listed as: `assign_pointers_Convec3d`, `assign_quadra3d`, `bary2cart3d`, `Bmoment3d`, `Convec3d`, `Convec3d_const`, `crossProd2`, `Bmoment3d_Index`, `computeBinomials`, `data_at_Nodes_Cval3d`, `data_at_Nodes_Convec3d`, `gaussJacobiUnit3D`, `init_Bmoment3d_Cval`, `init_BmomentC_Convec3d`, `init_precomp3d`, `innerProd_Coeff3d`, `inter`, `vector_values_at_Stroud3d`, `normals3d`, `position3d`, `position3d2`, `position3d_sum`, `position3d_sum2`, `subtract`, `transform_BmomentC_Convec3d`, `stroud_nodes_bary3d` and `Volume3d`. Note that most routines are defined in the previous sections. Indeed, `assign_quadra3d`, `bary2cart3d`, `Bmoment3d`, `Bmoment3d_Index`, `computeBinomials`, `crossProd2`, `data_at_Nodes_Cval3d`, `gaussJacobiUnit3D`, `init_Bmoment3d_Cval`, `init_precomp3d`, `inter`, `normals3d`, `position3d`, `position3d2`, `position3d_sum`, `position3d_sum2`, `stroud_nodes_bary3d`, `subtract` and `Volume3d` are used in the stiffness matrix computation. As a result, only `assign_pointers_Convec3d`, `Convec3d`, `Convec3d_const`, `data_at_Nodes_Convec3d`, `init_BmomentC_Convec3d`, `innerProd_Coeff3d`, `transform_BmomentC_Convec3d` and `vector_values_at_Stroud3d` are defined in this section.

The routine declared below is used to compute the auxiliary arrays needed in the convective matrix computation:

```
#ifdef PRECOMP
void
assign_pointers_Convec3d (double v1[3], double v2[3], double
    v3[3], double v4[3],int n, int q, int nDash, int m, int
    nb_Array, double **matValNodes, double *Cval, double
    **quadraWN, double **precomp, void (*b) (double[3],
    double[3]), int functval );
#else
```

```
void
assign_pointers_Convec3d (double v1[3], double v2[3], double
    v3[3], double v4[3], int n, int q, int nDash, int m, int
    nb_Array, double *Cval, double **quadraWN, double
    **Bmoment, void (*b) (double[3], double[3]), int functval);
#endif
```

In the above routine, `v1`, `v2`, `v3`, `v4` are the tetrahedron's vertices, `q` is the order of the Stroud rule, `quadraWN` stores Gauss-Jacobi quadrature weights and centres, `nb_Array` is a variable used to specify that the computed B-moments are associated with vector-valued data, and `n` is the order of the B-moments. The output of `assign_pointers_Convec3d` is stored into `matValNodes` if `PRECOMP` is switched on, and in `Bmoment` otherwise. Depending on the value of the flag `functval`, either the vector-valued function `b` or the array `Cval` is used as input for the convective matrix coefficients.

The next routine computes the convective matrix associated with constant coefficients:

```
double
Convec3d_const (int n, double v1[3], double v2[3], double
    v3[3], double v4[3], double **binomialMat, double
    normalMat[][3], double innerProdMat[4], double
    **convecMat, double vectCoeff[3] );
```

The above routine implements Algorithm 3.13. More precisely, `Convec3d_const` computes the convective matrix of order `n` on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4`. The computed convective matrix is associated with constant coefficients given by `vectCoeff`, and is stored in the array `convecMat`. The array `innerProdMat` contains the scalar products of `vectCoeff` with the normals to the tetrahedron's faces stored in `normalMat`.

The routine for the computation of the convective matrix associated with variable coefficients is defined as follows:

```
double
```

```
Convec3d (int n, int q, double v1[3], double v2[3], double
    v3[3], double v4[3], double **binomialMat, double
    normalMat[][3], double **precomp, double **Bmoment, double
    **Bmomentab, double **convecMat, double **matValNodes,
    double **quadraWN);
#else
double
Convec3d (int n, int q, double v1[3], double v2[3], double
    v3[3], double v4[3], double **binomialMat, double
    normalMat[][3], double **Bmoment, double **BmomentInter,
    double **Bmomentab, double **convecMat, double **quadraWN);
#endif
```

`Convec3d` computes the convective matrix of order `n` on the tetrahedron with vertices `v1`, `v2`, `v3`, `v4` associated with variable coefficients. The coefficients values at the Stroud nodes of order `q` are stored in `matValNodes` if PRECOMP is switched on, and in `BmomentInter` otherwise. `Bmoment` contains the B-moments of order $2\mathtt{n}-1$ associated with the convective matrix coefficients. The array `Bmomentab` contains the scalar products of the vector-valued B-moments with the normals to the tetrahedron's faces. The routine `Convec3d` implements Algorithm 3.25 when `PRECOMP` is switched on, and [11, Algorithm 8] with $d = 3$ otherwise. The main difference between the two algorithms lies in the approach for computing the B-moments associated with the convective matrix coefficients. Indeed, `PRECOMP` makes use of the precomputed arrays defined in (2.21). As a consequence, the arguments of the routine `Convec3d` depend on whether or not `PRECOMP` is used.

When `PRECOMP` is switched on, the following routine is used to compute the values of the convective matrix coefficients at the Stroud nodes:

```
void
data_at_Nodes_Convec3d ( void (*b) (double[3], double[3]),
    double **matValNodes, int q, double **quadraWN, double
    v1[3], double v2[3], double v3[3], double v4[3] );
```

In the above routine, `b` is a vector-valued function which produces the convective matrix coefficients, `quadraWN` contains Gauss-Jacobi quadrature weights and centres of order `q`, and the tetrahedron's vertices are given by `v1`, `v2`, `v3`, `v4`. The computed coefficients values are stored into `matValNodes`.

When `PRECOMP` is switched off, the routine described below is used to initialize the B-moments with values of the convective matrix coefficients at the Stroud nodes:

```
void
init_BmomentC_Convec3d ( void (*b)(double[3],double[3]), int
    q, double v1[3], double v2[3], double v3[3], double v4[3],
    double **BmomentInter, double **quadraWN);
```

In the above routine, `b` is a vector-valued function which produces the convective matrix coefficients, and `quadraWN` contains Gauss-Jacobi quadrature weights and centres of order `q`. The routine `init_BmomentC_Convec3d` stores the values of `b` at the Stroud nodes of order `q` into the array `BmomentInter`.

The next routine is used to compute the inner products of the constant vector-valued coefficients with the normals to the tetrahedron's faces:

```
void
innerProd_Coeff3d (double normalMat[][3], double
    innerProdMat[4], double vectCoeff[3]);
```

The above routine is used for the computation of the convective matrix associated with constant coefficients given by `vectCoeff`. The array `normalMat` contains the normals to the tetrahedron's faces. The routine `innerProd_Coeff3d` computes the inner products of the normals with `vectCoeff`. The computed inner products are stored into `innerProdMat`.

When the data is variable, the next routine is used to compute the inner products of the vector-valued B-moments with the normals to the tetrahedron's faces:

```
void
```

```
transform_BmomentC_Convec3d(int n, int q, double **Bmoment,
    double **Bmomentab, double normalMat[4][3]);
```

In the above routine, `q` is the order of the Stroud rule, `Bmoment` is used to store the B-moments, and `normalMat` contains the normals to the tetrahedron's faces. The computed inner products are stored into the array `Bmomentab`.

The following routine is used to compute the values of the convective matrix coefficients at the Stroud nodes:

```
void
vector_values_at_Stroud3d(int q, double *Cval, double *B,
    void (*b) (double[3], double[3]), double v1[3], double
    v2[3], double v3[3], double v4[3] );
```

The above routine stores the values of the function `b` at the Stroud nodes of order `q` into the array `Cval`.

B.2.4.4 Code Execution

This section displays an example which illustrates how to make use of the routines defined in the previous sections in order to compute the convective matrix:

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <time.h>
4 #include <iostream>

6 #include "bbfem.h"

8 #ifndef MAX // a template
9 #define MAX(a,b) ( (a) > (b) ? (a) : (b) )
10 #endif



13 #ifdef CONSTANT
```

```
14 int main()
15 {
16 //  //vertices (standard tetrahedron)
17 //  double v1[3] = { 0, 0, 0};
18 //  double v2[3] = { 1, 0, 0};
19 //  double v3[3] = { 0, 1, 0};
20 //  double v4[3] = { 0, 0, 1};

22   //vertices (particular tetrahedron)
23   double v1[3] = { 1.2   , 3.4, 0};
24   double v2[3] = { -1.5 , 2. , 0};
25   double v3[3] = { 0.1   , -1., 0};
26   double v4[3] = {1. , 1., 1.};

28   int n; // degree of the Bernstein polynomial basis
29   std::cout<<"Enter a value for the polynomial order n:";
30   std::cin>>n;

32   double vectCoeff[3] = {1.,1.,1.}; // constant coefficients
         associated with the convective matrix
33   double **convecMat; // store convective matrix entries
34   int len_Convec = len_Mat3d(n);
35   convecMat = create_Mat(len_Convec); // allocate memory to
         convecMat

37   get_convec3d_const (convecMat, n, v1, v2, v3, v4,
         vectCoeff);// compute convective matrix

39   // Insert your code here to make use of convecMat. it will
         be destroyed in the next line!

41   // free allocated memory
42   delete_Mat(convecMat);
```

```
43 }


46 #else // not CONSTANT

48 // example of coefficient vector b (used with convective
       matrix)
49 void
50 b0 (double v[3], double res[3])
51 {
52   res[0] = sin(v[0]*v[1]*v[2]);
53   res[1] = 1.;
54   res[2] = exp(v[0]);
55 }

57 int main()
58 {
59 //  //vertices (standard tetrahedron)
60 //  double v1[3] = { 0, 0, 0};
61 //  double v2[3] = { 1, 0, 0};
62 //  double v3[3] = { 0, 1, 0};
63 //  double v4[3] = { 0, 0, 1};

65   //vertices (particular tetrahedron)
66   double v1[3] = { 1.2   , 3.4, 0};
67   double v2[3] = { -1.5 , 2. , 0};
68   double v3[3] = { 0.1   , -1., 0};
69   double v4[3] = {1. , 1., 1.};

71   int n; // degree of the Bernstein polynomial basis
72   std::cout<<"Enter a value for the polynomial order n:";
73   std::cin>>n;
```

```
75  void (*b) (double[3], double[3]) = b0; // change here to
        your routine for the convective matrix coefficients
76  double *Cval; // store coefficients values at Stroud
        quadrature nodes, only used with FUNCTVAL

78  int functval = 0; //default: using a routine (b) for
        convective matrix coefficients
79  #ifdef FUNCT_VAL
80  functval = 1;  //using the values stored in Cval for
        convective matrix coefficients
81  int q = n+1;
82  int nb_Array = 3; // the convective matrix is associated
        with vector-valued data
83  double *B; // store barycentric coordinates of Stroud nodes
84  B = new double [q*q*q*4];
85  stroud_nodes_bary3d (q, B); // compute Stroud nodes

87  int LEN = q * q * q;  // q^3 Stroud nodes
88  Cval = new double[LEN * nb_Array];     //Cval entries are
        stored in LINEAR memory, and used directly
89  #endif

91  #ifdef FUNCT_VAL
92  vector_values_at_Stroud3d(q, Cval, B, b, v1, v2, v3, v4);
        // storing your data into Cval
93  #endif

95  double **convecMat; // store convective matrix entries
96  int len_Convec = len_Mat3d(n);
97  convecMat = create_Mat(len_Convec); // allocate memory to
        convecMat
```

```
99    get_convec3d(convecMat, n, b, Cval, v1, v2, v3, v4,
          functval); // compute convective matrix

101   // free allocated memory
102   #ifdef FUNCT_VAL
103   delete[] Cval;
104   delete[] B;
105   #endif

107   // Insert your code here to make use of convecMat. It will
          be destroyed in the next line!

109   delete_Mat(convecMat);
110 }
111 #endif // end not CONSTANT
```

Listing B.6: convec3d.cpp

B.3 $H(\text{curl})$ Routines List

Recall from Section 4.2 that the entries of the $H(\text{curl})$ elemental quantities are reduced to linear combinations of B-moments. Hence, the routines used in `bbfem.cpp` for the computation of the B-moments are also involved in $H(\text{curl})$ computations. In addition, routines which are $H(\text{curl})$-specific are given in the source code `bbfem2dCurl.cpp`. This section alphabetically lists the routines in `bbfem2dCurl.cpp` for the computation of $H(\text{curl})$ elemental quantities in two dimensions:

- **CBar:** computes the sequences $\bar{c}^{(\boldsymbol{\alpha})}$ defined in (4.25).
- **copy_Data_at_Stroud*:** copies stored values of the data at the Stroud nodes into the array used to store B-moments. This routine is needed when handling B-moments of various orders.

- **create_cBar:** allocates memory to the sequences $\bar{c}^{(\boldsymbol{\alpha})}$ defined in (4.25).
- **create_Coeff:** allocates memory to a coefficient sequence of specified length.
- **create_cRing:** allocates memory to the sequences $\mathring{\mathbf{c}}^{(\boldsymbol{\alpha})}$ defined in (4.23).
- **CRing:** computes the sequences $\mathring{\mathbf{c}}^{(\boldsymbol{\alpha})}$ defined in (4.23).
- **data_at_Nodes_Coeff:** reads the values of the data at the Stroud nodes.
- **delete_cBar:** frees the memory allocated by `create_cBar`.
- **delete_Coeff:** frees the memory allocated by `create_Coeff`.
- **delete_cRing:** frees the memory allocated by `create_cRing`.
- **delete_pointers_Curl:** frees the memory allocated to auxilliary arrays involved in the computation of $H(\text{curl})$ elemental quantities.
- **dimCurl:** computes the dimension of the Nédélec space.
- **dim_nonGradCurl:** computes the number of gradients spanning functions discussed in Theorem 4.1.3.
- **get_load2dCurl:** driver routine for computing the $H(\text{curl})$ elemental load vector.
- **get_mass2dCurl:** driver routine for computing the $H(\text{curl})$ elemental mass matrix.
- **get_stiffness2dCurl:** driver routine for computing the $H(\text{curl})$ elemental stiffness matrix.
- **Load2d_Curl:** computes the $H(\text{curl})$ elemental load vector.
- **LowerMoment:** lowers the B-moments order. The initial array containing the B-moments is overwritten with entries of lower-order B-moments.
- **Mass2d_Curl:** computes the $H(\text{curl})$ elemental mass matrix.

- **Stiff2d_Curl:** computes the *non-zero* entries of the $H(\text{curl})$ elemental stiffness matrix. More precisely, the gradient entries of the stiffness matrix are not computed.

- **transform_BmomentC_Mass2dCurl:** multiplies the normals to the triangle's edges with matrix-valued B-moments associated with the $H(\text{curl})$ mass matrix coefficients.

For each elemental quantity, the above routines compute the entries corresponding to the spanning set described in Theorem 4.1.3. In an implementation which is not covered by the proposed code, the appropriate entries need to be removed in order to obtain the elemental quantities which correspond to the $H(\text{curl})$ finite element basis defined in Theorem 4.1.3. One should note that, in the proposed routine, the Whitney lowest order edge elements correspond to the polynomial order $\mathtt{n} = 1$.

B.4 $H(\text{curl})$ Routines Description

This section contains the execution of the driver routines computing $H(\text{curl})$ elemental quantities in two dimensions. Explanations on the use of the routines listed in Section B.3 are given. For the definitions of the $H(\text{curl})$ routines, see [4]. Similarly to the H^1 computations, a flag called `functval` is used to determine whether an array of data values or a function is used as input for the coefficients. By default, a function is used as input for the coefficients.

B.4.1 $H(\text{curl})$ Load Vector

This section describes the routines which are involved with the computation of the load vector associated with a triangle $T =< \mathbf{v}_i,\ i = 1, \ldots, 3 >$. As in the H^1 case, a parameter `functval` serves as a flag for the type of input used for the load vector coefficients.

Similarly to the routines discussed in Section B.2, we first start with the driver routine used for computing the elemental load vector. The routines responsible for allocating memory are then described in Section B.4.1.2. Section B.4.1.3 then focuses on the routines involved in auxiliary computations. Section B.4.1.4 concludes with an example on how to execute the load vector computation.

B.4.1.1 Driver Routine

This section focuses on `get_load2dCurl` which is the driver routine used for computing the load vector. The above-mentioned routine is declared as follows:

```
void
get_load2dCurl(double *loadVect, int n, void (*F)( double
    [2], double [2]), double *Cval, double v1[2], double
    v2[2], double v3[2], int functval);
```

In the above routine, recall that the parameter `functval` is used as a flag for setting the input used for the load vector coefficients: With `functval`=0, the coefficients are produced by the function `F`, whereas with `functval`=1, the coefficients are produced by the array `Cval` which contains the data values at the Stroud nodes. The routine `get_load2dCurl` computes the load vector of order `n` on the triangle with vertices `v1`, `v2`, `v3`, associated with either `F` or `Cval`. The computed load vector entries are stored into the array `loadVect`. Recall that the Whitney's lowest order edge elements correspond to the case $\mathtt{n} = 1$.

B.4.1.2 Memory Allocation

This section discusses the purpose and the syntax of the load vector routines contained in Section B.3 which are used to allocate memory. These routines are listed as: `create_cRing`, `delete_cRing`, `dimCurl`, `dim_nonGradCurl` and `delete_pointers_Curl`.

The next routine allocates memory for storing the coefficients defined in (4.23):

```
double **
```

```
create_cRing(int n);
```

In the above routine, `n` is the finite element order.

The routine declared below frees the memory allocated by `create_cRing`:

```
void
delete_cRing(double **cRing);
```

The following routine returns the number of shape functions contained in the spanning set defined in Theorem 4.1.3:

```
int
dimCurl(int n);
```

The next routine computes the number of non-gradient shape functions contained in the spanning set of Theorem 4.1.3:

```
int
dim_nonGradCurl(int n);
```

The routine declared below is used to free the memory allocated to auxiliary arrays involved in the computation of the $H(\mathrm{curl})$ load vector:

```
#ifdef PRECOMP
void
delete_pointers_Curl(double **precomp, double **Bmoment,
    double **BmomentInter, double **matValNodes, double
    **quadraWN);
#else
void
delete_pointers_Curl(double **Bmoment, double **BmomentInter,
    double **matValNodes, double **quadraWN);
#endif
```

In the above routine, `Bmoment` contains the B-moments associated with the load vector coefficients, whereas `quadraWN` contains Gauss-Jacobi quadrature weights and centres. `BmomentInter` is used to store intermediate values needed for the

B-moments computation, while `matValNodes` contains coefficients values at the Stroud nodes. In the case where `PRECOMP` is switched on, the array `precomp` is used to store the precomputed arrays given in (2.16).

B.4.1.3 Auxiliary Computations

This section describes the auxiliary routines contained in Section B.3 which are needed in the load vector computation. These routines are listed as: `CRing`, `copy_Data_at_Stroud`, `Load2d_Curl` and `LowerMoment`.

The following routine computes the coefficients defined in (4.23):

```
void
CRing(int n, double normalMat[][2], double **cRing);
```

In the above routine, `n` stands for the polynomial order and `normalMat` contains the normals to the triangle's edges. The computed coefficients are stored into the array `cRing`.

The coefficient declared below is used to initialize the B-moment entries with coefficients values at the Stroud nodes:

```
void
copy_Data_at_Stroud(int q, double v1[2], double v2[2], double
    v3[2], double **Bmoment, double **matValNodes, int
    nb_Array);
```

In the above routine, the parameter `nb_Array` determines whether the coefficients are scalar-, vector-, or matrix-valued. The array `matValNodes` contains the coefficients values at the Stroud nodes of order `q` on the triangle with vertices `v1`, `v2`, `v3`. The routine `copy_Data_at_Stroud` copies the values stored in `matValNodes` into the array `Bmoment`. Since the array `Bmoment` is overwritten during the B-moments computations, this routine is useful when handling B-moments of various orders, as it avoids having to compute data values at the Stroud nodes for each particular B-moment order.

The low-level routine for the load vector computation is declared as:

```
#ifdef PRECOMP
double
Load2d_Curl( int n,  int q, double v1[2], double v2[2],
    double v3[2],  double **binomialMat, double
    normalMat[][2], double **precomp, int mp, double
    **Bmoment, double **BmomentInter, double *loadVect, double
    **matValNodes, double **quadraWN, double **cRing, double
    cputime[3]);
#else
double
Load2d_Curl( int n, int q, double v1[2], double v2[2], double
    v3[2], double **binomialMat, double normalMat[][2], double
    **Bmoment, double **BmomentInter, double *loadVect, double
    **matValNodes, double **quadraWN, double **cRing, double
    cputime[3]);
#endif
```

The above routine computes the elemental $H(\mathrm{curl})$ load vector of order `n` on the triangle with vertices `v1`, `v2` and `v3`. Precomputed binomial coefficients are stored into `binomialMat`. The array `quadraWN` contains Gauss-Jacobi quadrature weights and centres of order `q`. The normals to the triangle's edges are stored in `normalMat`. The array `cRing` is initialized with the coefficients defined in (4.23). The values of the load vector coefficients at the Stroud nodes are stored in `matValNodes`. `Bmoment` and `BmomentInter` are involved in the computation of the B-moments associated with the load vector coefficients. The computed load vector is stored into the array `loadVect`.

`Load2d_Curl` implements Algorithm 4.3. When `PRECOMP` is switched on, the B-moments associated with the load vector coefficients are computed using the precomputed arrays defined in (2.16). Otherwise, they are computed using [11, Algorithm 6] with $d = 2$. As a result, the arguments of the routine `Load2d_Curl` depend on whether or not `PRECOMP` is active.

The next routine is used for lowering the B-moments order:

```
void
LowerMoment( int p, int q, int ell, double **Bmoment, double
    **BmomentInter, int nb_Array);
```

The above routine computes the B-moments of order `ell` from those of order `p`, with `ell < p`. The array `Bmoment` initially contains B-moments of order `p` computed by means of the `q`-point Stroud rule. Once passed to the routine `LowerMoment`, the initial entries of the array `Bmoment` are replaced with B-moments of order `ell`.

B.4.1.4 Code Execution

This section presents an example which illustrates how to make use of the routines described in the previous sections in order to compute the $H(\mathrm{curl})$ load vector:

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <iostream>

#include "bbfem.h"
#include "bbfem2dCurl.h"

// example of load vector coefficient
void
F0( double v[2], double vectF[2] )
{
   vectF[0] = sin(v[0]*v[1]);
   vectF[1] = 1 - v[0]*v[1];
}

int main()
{
```

```
19 //     //vertices (standard triangle)
20 //       double v1[2] = { 0, 0 };
21 //       double v2[2] = { 1, 0 };
22 //       double v3[2] = { 0, 1 };

24    //vertices (particular triangle)
25    double v1[2] = { 1.2   , 3.4 };
26    double v2[2] = { -1.5 , 2. };
27    double v3[2] = { 0.1   , -1. };

29    int n; // degree of the Bernstein polynomial basis
30    std::cout<<"Enter a value for the polynomial order n:";
31    std::cin>>n;

33    double *Cval; // store array of coefficients values at
          Stroud quadrature nodes
34    int functval = 0; //default: using a routine (Kappa) for
          mass matrix coefficients

36    void (*F) (double[2], double[2]) = F0; // change here to
          your routine for load vector coefficients

38    #ifdef FUNCT_VAL
39    functval = 1;  //using the values stored in Cval for load
          vector coefficients
40    int q = n+1;
41    int nb_Array = 2; // the load vector is associated with
          vector-valued data

43    double *B; // store barycentric coordinates of Stroud nodes
44    B = new double [q*q*3];
45    stroud_nodes_bary2d (q, B);
```

```
47    int LEN = q * q ;  // space required for 2D array with
          dimension q x q
48    Cval = new double[LEN * nb_Array];     //Cval entries are
          stored in LINEAR memory, and used directly
49    vector_values_at_Stroud2d(q, Cval, B, F, v1, v2, v3 ); //
          storing your data in Cval
50    #endif

52    double *loadVect;  // store load vector entries
53    int len_Load = dimCurl(n); // allocate memory to stiffMat
54    loadVect = new double [len_Load];

56    get_load2dCurl(loadVect, n, F, Cval, v1, v2, v3, functval);
          // compute load vector

58    // free allocated memory
59    #ifdef FUNCT_VAL
60    delete[] Cval;
61    delete[] B;
62    #endif

64    // Insert your code here to make use of loadVect. It will
          be destroyed in the next line!

66    delete[] loadVect;
67 }
```

Listing B.7: load2dCurl.cpp

As mentioned previously, the macro `FUNCT_VAL` is used when the array `Cval` serves as input for the load vector coefficients. In addition, the Whitney's lowest order edge elements correspond to the case `n` = 1. In Listing B.7, `stroud_nodes_bary2d` and `vector_values_at_Stroud2d` are routines defined in `bbfem.cpp` which respectively compute the barycentric coordinates of the Stroud nodes and the val-

ues of vector-valued coefficients at the Stroud nodes. In addition, recall that the above code executes the computation of the load vector entries associated with the spanning set described in Theorem 4.1.3. Hence, in an implementation the appropriate rows needs to be removed in order to correspond to the $H(\text{curl})$ finite element basis given in Theorem 4.1.3.

B.4.2 $H(\text{curl})$ Mass Matrix

This section describes the routines which are involved with the computation of the $H(\text{curl})$ mass matrix associated with a triangle $T =< \mathbf{v}_i,\ i = 1, \ldots, 3 >$.

This section is organized as follows: the driver routine used for computing the elemental mass matrix is described in Section B.4.2.1. The routines responsible for allocating memory to auxiliary arrays involved in the mass matrix computation are discussed in Section B.4.2.2. Section B.4.2.3 then focuses on the routines used in intermediate computations. Section B.4.2.4 concludes with an example on how to execute the $H(\text{curl})$ mass matrix computation.

B.4.2.1 Driver Routine

The driver routine for computing the $H(\text{curl})$ mass matrix is declared as:

```
void
get_mass2dCurl(double **massMat, int n, void (*Kappa) (double
    [2], double [2][2]), double *Cval, double v1[2], double
    v2[2], double v3[2], int functval );
```

The above routine computes the elemental mass matrix of order `n` on the triangle with vertices `v1`, `v2`, `v3`. Depending on the value of the flag `functval`, either the matrix-valued function `Kappa` or the array `Cval` is used as input for the mass matrix coefficients. The computed mass matrix is stored into the array `massMat`. In the proposed code, the Whitney's lowest order edge elements correspond to the case `n` $= 1$.

B.4.2.2 Memory Allocation

This section presents the routines mentioned in Section B.3 which are responsible for allocating memory to the intermediate arrays involved in the $H(\text{curl})$ mass matrix computation. These routines are listed as: `create_cRing`, `delete_cRing`, `delete_pointers_Curl`, `dimCurl` and `dim_nonGradCurl`. The above-mentioned routines are discussed in Section B.4.1.2.

B.4.2.3 Auxiliary Computations

This section focuses on the mass matrix routines presented in Section B.3 which are involved in intermediate computations. These routines are listed as: `copy_Data_at_Stroud`, `CRing`, `LowerMoment`, `Mass2d_Curl` and `transform_BmomentC_Mass2dCurl`. Observe that the first three routines are described in Section B.4.1.3. Hence, only `Mass2d_Curl` and `transform_BmomentC_Mass2dCurl` are discussed in this section.

The low-level routine for computing the $H(\text{curl})$ mass matrix is declared as:

```
#ifdef PRECOMP
double
Mass2d_Curl( int n, int q, double v1[2], double v2[2], double
    v3[2], double **binomialMat, double normalMat[][2], double
    **precomp, int mp, double **Bmoment, double
    **BmomentInter, double **Bmomentab, double **massMat,
    double **matValNodes, double **quadraWN, double **cRing,
    double cputime[2]);
#else
double
Mass2d_Curl( int n, int q, double v1[2], double v2[2], double
    v3[2], double **binomialMat, double normalMat[][2], double
    **Bmoment, double **BmomentInter, double **Bmomentab,
    double **massMat, double **matValNodes, double **quadraWN,
    double **cRing, double cputime[2]);
```

```
#endif
```

The above routine computes the elemental $H(\text{curl})$ mass matrix of order `n` on the triangle with vertices `v1`, `v2` and `v3`. The arguments which are common to `Mass2d_Curl` and `Load2d_Curl` discussed in Section B.4.1.3 have the same definition, with the exception that the arrays `Bmoment` and `BmomentInter` are involved in the computation of B-moments associated with the mass matrix coefficients.

The routine `Mass2d_Curl` implements Algorithm 4.10. When `PRECOMP` is switched on, the B-moments associated with the mass matrix coefficients are computed using the precomputed arrays defined in (2.16). Otherwise, they are computed using [11, Algorithm 6] with $d = 2$. Thus, the arguments of the routine `Mass2d_Curl` depend on whether or not `PRECOMP` is active.

B.4.2.4 Code Execution

This section presents an example which illustrates how to make use of the routines described in the previous sections in order to compute the $H(\text{curl})$ mass matrix:

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <time.h>
4 #include <iostream>


7 #include "bbfem.h"
8 #include "bbfem2dCurl.h"


11 //example of mass matrix coefficient:
12 void
13 Kappa0 (double v[2], double matC[2][2])
14 {
15 //   // standard coefficient matrix (identity) for stiffness
```

```
16 //    matC[0][0] = 1;
17 //    matC[0][1] = 0;
18 //    matC[1][0] = 0;
19 //    matC[1][1] = 1;

21    //non-standard coefficient
22    matC[0][0] =  10.0 + v[0];
23    matC[0][1] =  v[1] * v[0] * v[0];
24    matC[1][0] =  v[1] * v[0] * v[0];
25    matC[1][1] =  2.0 - sin(v[0]*v[1]);

27 }


30 int main()
31 {
32 //     //vertices (standard triangle)
33 //       double v1[2] = { 0, 0 };
34 //       double v2[2] = { 1, 0 };
35 //       double v3[2] = { 0, 1 };

37    //vertices (particular triangle)
38    double v1[2] = { 1.2   , 3.4 };
39    double v2[2] = { -1.5 , 2. };
40    double v3[2] = { 0.1   , -1. };

42    int n; // degree of the Bernstein polynomial basis
43    std::cout<<"Enter a value for the polynomial order n:";
44    std::cin>>n;

46    double *Cval; // store array of function values at Stroud
          quadrature nodes, needed by get_mass2dCurl
```

```
47  int functval = 0; //default: using a routine (Kappa) for
        mass matrix coefficients

49  void (*Kappa) (double[2], double[2][2]) = Kappa0; // change
        here to your routine for mass matrix coefficients

51  #ifdef FUNCT_VAL
52  functval = 1;  //using the values stored in Cval for mass
        matrix coefficients
53  int q = n+1;
54  int nb_Array = 3; // the mass matrix is associated with
        (symmetric) matrix-valued data

56  double *B; // store barycentric coordinates of Stroud nodes
57  B = new double [q*q*3];
58  stroud_nodes_bary2d (q, B);

60  int LEN = q * q ;  // space required for 2D array with
        dimension q x q
61  Cval = new double[LEN * nb_Array];     //Cval entries are
        stored in LINEAR memory, and used directly

63  matrix_values_at_Stroud2d(q, Cval, B, Kappa, v1, v2, v3 );
        // storing your data in Cval
64  #endif

66  double **massMat;  // store mass matrix entries
67  int len_Mass = dimCurl(n); // allocate memory to massMat
68  massMat = create_Mat(len_Mass);

70  get_mass2dCurl(massMat, n, Kappa, Cval, v1, v2, v3,
        functval); // compute elemental mass matrix
```

```
72    // free allocated memory
73    #ifdef FUNCT_VAL
74    delete[] Cval;
75    delete[] B;
76    #endif

78    // Insert your code here to make use of massMat. It will be
          destroyed in the next line!

80    delete_Mat(massMat);
81  }
```

Listing B.8: mass2dCurl.cpp

In the above code, `stroud_nodes_bary2d` and `matrix_values_at_Stroud2d` are routines defined in `bbfem.cpp` which respectively compute the barycentric coordinates of the Stroud nodes and the values of matrix-valued coefficients at the Stroud nodes. In addition, recall that the above code executes the computation of the mass matrix entries associated with the spanning set described in Theorem 4.1.3. Hence, in an implementation the appropriate rows needs to be removed in order to correspond to the $H(\text{curl})$ finite element basis given in Theorem 4.1.3.

B.4.3 $H(\text{curl})$ Stiffness Matrix

This section describes the routines which are involved with the computation of the $H(\text{curl})$ mass matrix associated with a triangle $T =< \mathbf{v}_i,\ i = 1, \ldots, 3 >$.

This section is organized as follows: Section B.4.3.1 focuses on the driver routine used for computing the elemental stiffness matrix. The routines responsible for allocating memory to auxiliary arrays involved in the stiffness matrix computation are discussed in Section B.4.3.2. The routines used in auxiliary computations are presented in Section B.4.3.3. Section B.4.3.4 concludes with an example on how to execute the $H(\text{curl})$ stiffness matrix computation. In the proposed code, the Whitney's lowest order edge elements correspond to the case

`n` = 1.

B.4.3.1 Driver Routine

The driver routine for computing the non-gradient stiffness matrix entries is declared as:

```
void
get_stiffness2dCurl(double **stiffMat, int n, double (*A)
    (double v[2]), double *Cval, double v1[2], double v2[2],
    double v3[2], int functval);
```

The above routine computes the non-gradient entries of the elemental stiffness matrix of order `n` on the triangle with vertices `v1`, `v2`, `v3`. Depending on the value of the flag `functval`, either the scalar-valued function `A` or the array `Cval` is used as input for the stiffness matrix coefficients. The computed stiffness matrix is stored into the array `stiffMat`. In the proposed code, the Whitney's lowest order edge elements correspond to the case `n` = 1.

B.4.3.2 Memory Allocation

This section presents the routines mentioned in Section B.3 which are responsible for allocating memory to the auxiliary arrays involved in the $H(\text{curl})$ stiffness matrix computation. These routines are listed as: `create_cBar`, `delete_cBar`, `delete_pointers_Curl` and `dim_nonGradCurl`. Observe that the last two routines are described in Section B.4.1.3. As a result, only `create_cBar` and `delete_cBar` are discussed in this section.

The routine described below is used to allocate memory to the coefficients defined in (4.25):

```
double **
create_cBar(int n);
```

In the above routine, `n` is the order of the $H(\text{curl})$ finite element. Recall that in

the presented codes, the Whitney's lowest order edge elements correspond to the case `n` = 1.

The next routine is used to free the memory allocated by `create_cBar`:

```
void
delete_cBar(double **cBar);
```

B.4.3.3 Auxiliary Computations

This section focuses on the auxiliary routines which are involved in the stiffness matrix computation. These routines are listed as: `CBar`, `copy_Data_at_Stroud` and `Stiff2d_Curl`. Observe that `copy_Data_at_Stroud` is covered by Section B.4.1.3. Hence, only the routines `CBar` and `Stiff2d_Curl` are discussed in this section.

The routine described below is used to compute the coefficients defined in (4.25)

```
void
CBar( int n, double v1[2], double v2[2], double v3[2] ,
    double **cBar);
```

In the above routine, `n` is the finite element order, and `v1`, `v2`, `v3` are the triangle's vertices. The computed coefficients are stored into the array `cBar`.

The low-level routine for computing the $H(\mathrm{curl})$ elemental stiffness matrix is declared as:

```
#ifdef PRECOMP
double
Stiff2d_Curl(int n, int q, double v1[2], double v2[2], double
    v3[2], double **binomialMat, double **precomp, int mp,
    double **Bmoment, double **BmomentInter, double **cBar,
    double **stiffMat, double **matValNodes, double
    **quadraWN, double cputime[3]);
#else
```

```
double
Stiff2d_Curl(int n, int q, double v1[2], double v2[2], double
    v3[2], double **binomialMat, double **Bmoment, double
    **BmomentInter, double **cBar, double **stiffMat, double
    **matValNodes, double **quadraWN, double cputime[3]);
#endif
```

The above routine computes the elemental $H(\mathrm{curl})$ stiffness matrix of order `n` on the triangle with vertices `v1`, `v2` and `v3`. The arguments which are common to `Stiff2d_Curl` and `Load2d_Curl` discussed in Section B.4.1.3 have the same definition, with the exception that the arrays `Bmoment` and `BmomentInter` are involved in the computation of B-moments associated with the stiffness matrix coefficients. In addition, the array `cBar` contains the coefficients defined in (4.25).

The routine `Stiff2d_Curl` implements Algorithm 4.11. When `PRECOMP` is switched on, the B-moments associated with the stiffness matrix coefficients are computed using the precomputed arrays defined in (2.16). Otherwise, they are computed using [11, Algorithm 6] with $d = 2$. As a consequence, the arguments of the routine `Stiff2d_Curl` depend on whether or not `PRECOMP` is active.

B.4.3.4 Code Execution

This section presents an example which illustrates how to make use of the routines described in the previous sections in order to compute the $H(\mathrm{curl})$ mass matrix:

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <time.h>
4 #include <iostream>

6 #include "bbfem.h"
7 #include "bbfem2dCurl.h"

9 // example of stiffness matrix coefficient
```

```
10 double
11 A0 (double v[2])
12 {
13   return 2.0 - sin(v[0]*v[1]);
14 }

16 int main()
17 {
18 //   //vertices (standard triangle)
19 //      double v1[2] = { 0, 0 };
20 //      double v2[2] = { 1, 0 };
21 //      double v3[2] = { 0, 1 };

23   //vertices (particular triangle)
24   double v1[2] = { 1.2   , 3.4 };
25   double v2[2] = { -1.5 , 2. };
26   double v3[2] = { 0.1   , -1. };

28   int n; // degree of the Bernstein polynomial basis
29   std::cout<<"Enter a value for the polynomial order n:";
30   std::cin>>n;

32   double *Cval; // store array of function values at Stroud
       quadrature nodes, needed by get_mass2dCurl
33   int functval = 0; //default: using a routine (Kappa) for
       mass matrix coefficients

35   double (*A) (double[2]) = A0; // change here to your
       routine for stiffness matrix coefficients

37   #ifdef FUNCT_VAL
38   functval = 1;  //using the values stored in Cval for mass
       matrix coefficients
```

```
39  int q = n+1;
40  int nb_Array = 1; // the stiffness matrix is associated
        with scalar-valued data

42  double *B; // store barycentric coordinates of Stroud nodes
43  B = new double [q*q*3];
44  stroud_nodes_bary2d (q, B);

46  int LEN = q * q ;  // space required for 2D array with
        dimension q x q
47  Cval = new double[LEN * nb_Array];     //Cval entries are
        stored in LINEAR memory, and used directly

49  scalar_values_at_Stroud2d(q, Cval, B, A, v1, v2, v3 ); //
        storing your data in Cval
50  #endif

52  double **stiffMat;  // store (non-zero) stiffness matrix
        entries
53  int len_Stiff = dim_nonGradCurl(n); // allocate memory to
        stiffMat
54  stiffMat = create_Mat(len_Stiff);

56  get_stiffness2dCurl(stiffMat, n, A, Cval, v1, v2, v3,
        functval); // compute non-zero stiffness matrix entries

58  // free allocated memory
59  #ifdef FUNCT_VAL
60  delete[] Cval;
61  delete[] B;
62  #endif
```

```
64    // Insert your code here to make use of stiffMat. It will
          be destroyed in the next line!

66    delete_Mat(stiffMat);
67 }
```

Listing B.9: stiff2dCurl.cpp

In the above code, `stroud_nodes_bary2d` and `scalar_values_at_Stroud2d` are routines defined in `bbfem.cpp` which respectively compute the barycentric coordinates of the Stroud nodes and the values of scalar-valued coefficients at the Stroud nodes. In addition, recall that the above code executes the computation of the stiffness matrix entries associated with the spanning set described in Theorem 4.1.3. Hence, in an implementation the appropriate rows needs to be removed in order to correspond to the $H(\text{curl})$ finite element basis given in Theorem 4.1.3.

Appendix C

Shift strategy

Applying the Galerkin finite element discretization to Maxwell's equations yields a generalized eigenvalue problem, where the pencil consists of the pair $(\mathbf{S}, \mathbf{M})$ with $\mathbf{S}$ and $\mathbf{M}$ respectively denoting the $H(\text{curl})$ stiffness and mass matrices. Having computed the element matrices by means of the algorithms presented in Chapter 4, the obtained eigenvalue problem can be solved using any general-purpose eigensolver. However, doing so results in wasteful computations of the zero eigenvalues. Hence, this section presents a systematic approach for computing directly the non-zero eigenvalues associated with the pencil $(\mathbf{S}, \mathbf{M})$.

For alternative methods regarding the numerical solutions of Maxwell's equations, the reader is referred to [7, 55, 16, 15], and the references therein.

C.1 Symmetric Eigenvalue Problem

Consider the symmetric eigenproblem consisting in finding the pairs $(\lambda, \mathbf{u})$ satisfying

$$\mathbf{S}\mathbf{u} = \lambda \mathbf{M}\mathbf{u}, \tag{C.1}$$

where the eigenvalue λ is a scalar, and $\mathbf{u}$ the associated eigenvector. In addition, the matrix $\mathbf{M}$ is assumed to be positive definite. In (C.1), the pair $(\mathbf{S}, \mathbf{M})$ is

often called a *pencil*. The next result, given in [68, Theorem 15.3.3], motivates the use of some orthogonalization (or *deflation*) procedures in the iterative scheme described later.

Theorem C.1.1. *Let* $\mathbf{S}$ *and* $\mathbf{M}$ *denote real-valued symmetric matrices of dimension* ℓ*. In addition, suppose that* $\mathbf{M}$ *is positive definite. Then, the problem (C.1) has* ℓ *real eigenvalues* $\lambda_1, \ldots, \lambda_\ell$*. For* $k = 1, \ldots, \ell$*, denote by* $\mathbf{u}_k$ *the eigenvector corresponding to* λ_k*. Then the set* $\{\mathbf{u}_k : \ k = 1, \ldots, \ell\}$ *is orthogonal with respect to the* $\mathbf{M}$*-inner product defined by*

$$\langle \mathbf{u}, \mathbf{v} \rangle_{\mathbf{M}} = \mathbf{u}^{\mathrm{t}} \mathbf{M} \mathbf{v}, \quad \mathbf{u}, \mathbf{v} \in \mathbb{R}^{\ell}.$$

For the sake of completeness, we next describe the numerical scheme used for solving Maxwell's eigenvalue problem in Chapter 4.

Based on [76], we opt for a combination of shifted inverse iteration (`INVIT`) and Rayleigh quotient iteration (`RQI`) [68, Chapter 4 and Chapter 15], denoted as `INVIT-RQI`. In order to avoid computing the zero eigenvalues associated with the above-mentioned eigenproblem, our iterative scheme also contains additional projections onto the space orthogonal to the kernel of the curl operator. For the reader's convenience, we recall in Algorithm C.1 the basic steps of `INVIT` with orthogonal projection, for finding the non-zero eigenvalues associated with (C.1), when Maxwell's eigenvalue problem is considered. The projection matrix $\mathbf{P}_\nabla$ on line 7 is given in Section 4.4.

In Algorithm C.1, $\|\cdot\|_{\mathbf{M}^{-1}}$ is the norm induced by the inner product $\langle \cdot, \cdot \rangle_{\mathbf{M}^{-1}}$, whereas "tol" is a prescribed tolerance. `RQI` differs from `INVIT` in that no LU factorization is used, and that the shift is updated at each level of iteration. `RQI` is described in more details in Algorithm C.2. Recall that, for $k \geq 2$, ρ_{k-1} denotes the *Rayleigh quotient* of the current iterate $\mathbf{u}_{k-1}$.

Without the projection step given on line 7, `INVIT` is well-known to converge *linearly* towards the eigenvalue closest to the shift θ, provided that the starting iterate $\mathbf{u}_{\text{init}}$ is not orthogonal to the corresponding eigenvector. In contrast, `RQI`

Algorithm C.1: INVIT$(\mathbf{S}, \mathbf{M}, \theta, \mathbf{u}_{\text{init}})$

Input : Matrices $\mathbf{S}$ and $\mathbf{M}$, shift θ, and starting eigenvector $\mathbf{u}_{\text{init}}$.
Output: Eigenpair $(\rho, \mathbf{u})$ such that ρ is one eigenvalue of minimal distance from the shift θ.

1 Compute the LU factorization of $\mathbf{S} - \theta\mathbf{M}$;
2 Set $\mathbf{u}_0 = \mathbf{u}_{\text{init}}$, $\rho_0 = \theta$, $k = 0$;
3 **while** $\|\mathbf{S}\mathbf{u}_k - \rho_k\mathbf{M}\mathbf{u}_k\|_{\mathbf{M}^{-1}} > \text{tol}$ **do**
4 $\quad k \mathrel{+}= 1$;
5 $\quad$ Solve $(\mathbf{S} - \theta\mathbf{M})\tilde{\mathbf{y}}_k = \mathbf{M}\mathbf{u}_{k-1}$ for $\tilde{\mathbf{y}}_k$, using the LU factorization computed on line 1 ;
6 $\quad$ `//Project onto space orthogonal to` ker(curl)`:`
7 $\quad$ Compute $\tilde{\mathbf{u}}_k = (\mathbf{I} - \mathbf{P}_\nabla)\tilde{\mathbf{y}}_k$;
8 $\quad$ Compute $\mathbf{u}_k = \tilde{\mathbf{u}}_k / \|\tilde{\mathbf{u}}_k\|_{\mathbf{M}}$;
9 $\quad$ Set $\rho_k = \mathbf{u}_k^{\mathrm{t}}\mathbf{S}\mathbf{u}_k$;
10 `//Return the final values obtained at the end of the loop`
11 **Return** $(\rho_\infty, \mathbf{u}_\infty)$;

Algorithm C.2: RQI$(\mathbf{S}, \mathbf{M}, \theta, \mathbf{u}_{\text{init}})$

Input : Matrices $\mathbf{S}$ and $\mathbf{M}$, shift θ, and starting eigenvector $\mathbf{u}_{\text{init}}$.
Output: Eigenpair $(\rho, \mathbf{u})$ such that ρ is one eigenvalue of minimal distance from the shift θ.

1 Set $\mathbf{u}_0 = \mathbf{u}_{\text{init}}$, $\rho_0 = \theta$, $k = 0$;
2 **while** $\|\mathbf{S}\mathbf{u}_k - \rho_k\mathbf{M}\mathbf{u}_k\|_{\mathbf{M}^{-1}} > \text{tol}$ **do**
3 $\quad$ Same as lines 4-9 of `INVIT` $(\mathbf{S}, \mathbf{M}, \theta, \mathbf{u}_{\text{init}})$, with the line 5 replaced with
4 $\quad$ Solve $(\mathbf{S} - \rho_{k-1}\mathbf{M})\tilde{\mathbf{y}}_k = \mathbf{M}\mathbf{u}_{k-1}$ for $\tilde{\mathbf{y}}_k$;
5 `//Return the final values obtained at the end of the loop`
6 **Return** $(\rho_\infty, \mathbf{u}_\infty)$;

has a *locally cubic* convergence [68, Section 15.9], but does not necessarily converges towards the eigenvalue which is closest to the initial shift [68, Section 4.9]. In [76], Szyld describes an elegant way of combining the properties of `INVIT` and `RQI` in order to optimize the convergence towards a desired part of the spectrum. The presented algorithm, referred to as `INVIT-RQI` in this work, is used to solve the Maxwell's eigenproblem considered at the end of Chapter 4. More precisely, starting with an initial guess $(\theta, \mathbf{u}_{\text{init}})$, the basic step of the eigenvalue algorithm that we use can be described in Algorithm C.3.

Algorithm C.3: INVIT-RQI$(\mathbf{S}, \mathbf{M}, \theta, \mathbf{u}_{\text{init}}, \eta)$

Input : Matrices $\mathbf{S}$ and $\mathbf{M}$, initial shift θ, starting eigenvector $\mathbf{u}_{\text{init}}$, and switching parameter η.

Output: Eigenpair $(\rho, \mathbf{u})$ such that ρ is one eigenvalue of minimal distance from θ.

1 Set $\mathbf{u}_0 = \mathbf{u}_{\text{init}}$, $\rho_0 = \theta$, $k = 0$;
2 `//Start with INVIT:`
3 **while** $\|\mathbf{S}\mathbf{u}_k - \theta\mathbf{M}\mathbf{u}_k\|_{\mathbf{M}^{-1}} > \eta$ **do**
4 Same as lines 4-9 of `INVIT` $(\mathbf{S}, \mathbf{M}, \theta, \mathbf{u}_{\text{init}})$;
5 `//Monitor the relative change of the Rayleigh quotient:`
6 $\delta_k = |\rho_k - \rho_{k-1}|/|\rho_k|$;
7 **if** $\delta_k < \delta$ **and** $k \geq 3$ **then**
8 **break**;
9 `//Switch to RQI using the output of the INVIT-loop as starting eigenpair:`
10 Compute $(\rho_{\texttt{RQI}}, \mathbf{u}_{\texttt{RQI}}) = \texttt{RQI}(\mathbf{S}, \mathbf{M}, \rho_{\texttt{INVIT}}, \mathbf{u}_{\texttt{INVIT}})$;
11 **Return** $(\rho_{\texttt{RQI}}, \mathbf{u}_{\texttt{RQI}})$;

If the condition

$$\|\mathbf{S}\mathbf{u}_k - \theta\mathbf{M}\mathbf{u}_k\|_{\mathbf{M}^{-1}} \leq \eta \tag{C.2}$$

is satisfied at some iteration level k, then the limit eigenvalue of `INVIT` belongs to the interval $(\theta - \eta, \theta + \eta)$ [76]. In addition, if η satifies

$$\eta \leq \min_{\lambda_i \neq \lambda_j} \frac{|\lambda_i - \lambda_j|}{4}, \tag{C.3}$$

where the minimum is taken over all the eigenvalues of the pencil $(\mathbf{S}, \mathbf{M})$, then starting with the last eigenpair produced by `INVIT` when switching to `RQI` ensures cubic convergence towards the desired eigenpair. `INVIT-RQI` also covers the case where the eigenvalue which is closest to θ does not belong to the interval $(\theta - \eta, \theta + \eta)$. In this case, (C.2) is never satisfied, and after a few `INVIT` iterations, the Rayleigh quotient starts to converge towards an eigenvalue outside $(\theta - \eta, \theta + \eta)$, so that its relative change satisfies

$$\frac{\rho_k - \rho_{k-1}}{\rho_k} \leq \delta \tag{C.4}$$

for a prescribed tolerance δ. In this case, the algorithm switches to `RQI` solely to accelerate the convergence. However, (C.4) can occur too soon, if the starting iterate happens to be very close to an actual eigenvector. In order to prevent a premature switch to `RQI`, a minimum of 3 `INVIT` iterations is enforced. For our purposes, since the problem that we consider is known to have integer eigenvalues (up to a multiplication by π^2), the choice $\eta = 1/4$ is satisfactory. For more general settings, the algorithm proposed in [76] features the extra option of switching back to `INVIT` using the latest iterate of `RQI` as starting eigenvector, in the case where the input value η is too large.

C.2 Vector Iteration

Now observe that Algorithm C.3 only computes one eigenpair such that the eigenvalue is of minimal distance from θ. In order to compute several eigenpairs, `INVIT-RQI` needs to be incorporated into a bigger loop, where the shift θ is updated according to the obtained eigenvalues. In addition, deflation procedures are applied in order to avoid convergence to already obtained eigenpairs.

This section describes the details of the "global" eigenvalue algorithm, that is, when `INVIT-RQI` given in Algorithm C.3 is used to find several eigenpairs associated with the Maxwell's eigenvalue problem in Section 4.5. In particular, details are also given regarding the shift strategy used. The method is given in

Algorithm C.4.

`EigenList` and `EigenVecList` respectively contain the list of obtained eigenvalues and the corresponding eigenvectors. Before the loop over ℓ, `EigenList` `EigenVecList` and `EigenVecSameList` are each initializated to be empty. The list `EigenVecSameList` stores the eigenvectors which belong to the same eigenvalue, and is needed for the selective deflation applied on line 8. More precisely, only the set of previously obtained eigenvectors which are associated with the latest obtained eigenvalue are considered during the deflation process. That is to say, the current iterate is not orthogonalized with respect to the full set of previously obtained eigenvectors. Thus, if the shift θ happens to be closer to a smaller eigenvalue than to the eigenvalue on the right of λ_{init}, then `INVIT-RQI` converges to the smaller eigenvalue which is already stored in `EigenList`. In that case, the shift θ is corrected by means of line 25, where r is the counter for the number of times the `INVIT-RQI` loop produced the smaller eigenvalue. In order to distinguish the different cases, the algorithm needs to compare the computed eigenvalue ρ_∞ to the previously obtained eigenvalue λ_{init}. Hence, the dummy initialization $\lambda_{\text{init}} = 0$ on line 1 covers the very first loop over ℓ.

C.3 Artificial Shift Perturbation

When inverse iteration is coupled with orthogonal deflation, for solving the eigenproblem associated with the pencil $(\mathbf{S}, \mathbf{M})$, the choice of shift given on line 21 of Algorithm C.4 might not be optimal. In fact, it is observed that slightly perturbed shifts produce better and faster results. This can be illustrated from the following simple example.

Consider the problem of finding the eigenpairs associated with $(\mathbf{S}, \mathbf{M})$, with

$$\mathbf{S} = \begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix},$$

and $\mathbf{M}$ is the identity matrix. Observe that the eigenvalues of $\mathbf{S}$ are given by

Algorithm C.4: RQI-INVIT-Extended($\mathbf{S}, \mathbf{M}, \ell$)

Input : Matrices $\mathbf{S}$ and $\mathbf{M}$, and ℓ which stands for the desired number of computed eigenpairs.

Output: First ℓ non-zero eigenpairs associated with the pencil $(\mathbf{S}, \mathbf{M})$.

```
1  EigenList = {}, EigenVecList = {}, EigenVecSameList = {}, θ = 0,
   EigenNumber = 1, λ_init = 0, r = 1 ;
2  while EigenNumber ≤ ℓ do
3      u_init = random(dim(S));
4      Same lines as in INVIT-RQI(S, M, θ, u_init), except that before the
       normalizing step:
5      u_k = ũ_k / ||ũ_k||_M
6      the following lines are inserted:
7      foreach v ∈ EigenVecSameList do
8          ũ_k ← (I − v v^t M) ũ_k;
9      //At this point, INVIT-RQI loop has produced (ρ_∞, u_∞).
10     //Case where ρ_∞ is another multiplicity of the previously
       obtained eigenvalue:
11     if EigenList == {} or |ρ_∞ − λ_init| < ϵ then
12         EigenList.append(ρ_∞), EigenVecList.append(u_∞);
13         EigenVecSameList.append(u_∞);
14         EigenNumber += 1;
15     else
16         //Case where ρ_∞ is greater than the previously obtained
           eigenvalue:
17         if ρ_∞ > λ_init + η − ϵ then
18             EigenList.append(ρ_∞), EigenVecList.append(u_∞);
19             EigenVecSameList = {u_∞}, λ_init = ρ_∞;
20             //Update shift value:
21             θ = ρ_∞;
22             EigenNumber += 1, r = 1;
23         //Case where ρ_∞ is smaller than the previously obtained
           eigenvalue:
24         else
25             θ ← θ + r * (λ_init − ρ_∞), r ← r + 1;
26 Return EigenList, EigenVecList;
```

$(3 \pm \sqrt{5})/2$, and that the corresponding eigenspaces are respectively spanned by the vectors

$$\mathbf{e}_1 = \big(1, \frac{-1+\sqrt{5}}{2}\big)^{\mathrm{T}} \text{ and } \mathbf{e}_2 = \big(1, -\frac{1+\sqrt{5}}{2}\big)^{\mathrm{T}}. \tag{C.5}$$

Starting with $\sigma = 0$, we perform 16 iterations of shifted inverse iteration, and find a good approximation of $(\lambda_1, \mathbf{e}_1)$. We then update the value of σ, and continue the inverse iteration, coupled with deflation, to find the next eigenvector: With $\sigma = \lambda_1$, the code breaks down after 3 iterations; with $\sigma = \lambda + 1\text{e-}16$, the maximal number of iterations (500) is attained, and the resulting approximation is very poor, with an error of order 1e-1 for the approximation of λ_2.

In order to undestand this phenomenon, we now proceed to "simulate" what is happening inside the implementation during the evaluation of the second eigenvalue. Since the error is very small (of order 1e-16) for λ_1, the matrix used in the code is given by $\mathbf{S} - \lambda_1\mathbf{M}$, up to machine precision. Thus, for simplicity, we will use the exact value of $\mathbf{S} - \lambda_1\mathbf{M}$ for the "simulation". One can show that $\mathbf{S} - \lambda_1\mathbf{M}$ can be factorized by means of

$$\begin{aligned}
\begin{pmatrix} \frac{-1+\sqrt{5}}{2} & -1 \\ -1 & \frac{1+\sqrt{5}}{2} \end{pmatrix} &= \begin{pmatrix} 1 & 0 \\ \frac{-2}{-1+\sqrt{5}} & 1 \end{pmatrix} \begin{pmatrix} \frac{-1+\sqrt{5}}{2} & -1 \\ 0 & 0 \end{pmatrix} \\
&\approx \begin{pmatrix} 1 & 0 \\ \frac{-2}{-1+\sqrt{5}} & 1 \end{pmatrix} \begin{pmatrix} \frac{-1+\sqrt{5}}{2} & -1 \\ 0 & \varepsilon \end{pmatrix}
\end{aligned}$$

In the above equation, the symbol "$\approx$" indicates that the right-hand side gives the numerical approximation which is produced. The constant ε is of order 1e-16. With the starting iterate $\mathbf{u}_0 = (1,1)^{\mathrm{T}}$, we want to solve $\mathbf{L}\mathbf{U}\mathbf{u}_2 = \mathbf{u}_1$ for $\mathbf{u}_2$. To this end, we first solve $\mathbf{L}\mathbf{w}_2 = \mathbf{u}_1$ for $\mathbf{w}_2$, that is,

$$\begin{pmatrix} 1 & 0 \\ \frac{-2}{-1+\sqrt{5}} & 1 \end{pmatrix} \begin{pmatrix} w_{2,1} \\ w_{2,2} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The above system gives $\mathbf{w}_2 = (1, \frac{\sqrt{5}+3}{2})^{\mathrm{T}}$. Thus, we need to solve the system

$$\begin{pmatrix} \frac{-1+\sqrt{5}}{2} & -1 \\ 0 & \varepsilon \end{pmatrix} \begin{pmatrix} u_{2,1} \\ u_{2,2} \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{\sqrt{5}+3}{2} \end{pmatrix}, \tag{C.6}$$

Now, observe from the second row that $u_{2,2} = (\sqrt{5}+3)/(2\varepsilon) = \mathcal{O}(1\text{e}{+16})$, so that

$$u_{2,2} + 1 \approx u_{2,2}. \tag{C.7}$$

But then, the first row of (C.6) yields

$$\frac{-1+\sqrt{5}}{2} u_{1,1} = 1 + u_{2,2} \approx u_{2,2},$$

which, together with (C.5), shows that the second iterate $\mathbf{u}_2$ has a significant component along the direction of the eigenvector $\mathbf{e}_1$. Thus, with $\mathbf{w}_2$ satisfying $\mathbf{L}\mathbf{w}_2 = \mathbf{u}_1$,

$$(\mathbf{U}\mathbf{u}_2 = \mathbf{w}_2) \Rightarrow (\mathbf{u}_2 \text{ is "very" parallel to } \mathbf{e}_1). \tag{C.8}$$

In fact, for any $\mathbf{w} \in \mathbb{R}^2$ which is not "too large" (which can always be attained using normalization), the solution to $\mathbf{U}\mathbf{u} = \mathbf{w}$ with $\mathbf{U}$ given on the left-hand-side of (C.6), is always a vector which is very "parallel" to $\mathbf{e}_1$. Indeed, it suffices to use the same argument as the one leading to (C.8), but with the right-hand side of (C.6) given by the normalized form of $\mathbf{w}$. As a consequence,

$$(\mathbf{L}\mathbf{U}\mathbf{u}_2 = \mathbf{u_1}) \Rightarrow (\mathbf{u}_2 \text{ is "very" parallel to } \mathbf{e}_1).$$

Hence, the inverse iteration with the shift $\theta = \lambda_1$ produces iterates $\mathbf{u}_k$ which are almost "parallel" to the eigenvector $\mathbf{e}_1$. When the deflation procedure is applied, the iterate $\mathbf{u}_k$ is replaced with its projection onto the space orthogonal to $\mathbf{e}_1$. But then, the next iterate $\mathbf{u}_{k+1}$ satisfying $\mathbf{L}\mathbf{U}\mathbf{u}_{k+1} = \mathbf{u}_k$ is again almost "parallel" to $\mathbf{e}_1$. In other words, the choice $\sigma = \lambda_1 + \mathcal{O}(1\text{e-}16)$ annihilates too much the components of the iterates along the eigenvector $\mathbf{e}_2$, making it numerically

impossible to get a good approximation of the eigenpair $(\lambda_2, \mathbf{e}_2)$.

As a particular application of the above argument, we opt for an artificial perturbation of the shift θ used in Algorithm C.4. More precisely, the shift choice on line 21 is replaced with $\theta = \rho_\infty + \varsigma$, where ς is a prescribed perturbation. In our computations, we used $\varsigma = 0.1$. The concept of artificial shift is not new, and has been mentioned, for example, in [79, p.328] for the accurate approximation of eigenvalues with multiplicities.

Bibliography

[1] Cobham Technical Services Vector Fields Software.
http://www.cobham.com/technicalservices. 124, 140

[2] Knowledge Transfer Partnership.
http://www.ktponline.org.uk. 124

[3] OPERA.
http://www.operaFEA.com. 140

[4] BBFEM, 2013.
http://www.bbfem.de. 153, 154, 164, 211

[5] Hexahedron elements, 2013.
http://www.colorado.edu/engineering/CAS/courses.d/AFEM.d/AFEM.Ch11.d/AFEM.Ch11.pdf. 132

[6] R. Abdul-Rahman and M. Kasper. Orthogonal Hierarchical Nédélec Elements. *IEEE Trans. Magn.*, 44(6):1210–1213, 2008. 4

[7] S. Adam, P. Arbenz, and R. Geus. A comparison of solvers for large eigenvalue problems originating from Maxwell's equations. *Numer. Linear Algebra Appl.*, 6(1):3–16, 1999. 231

[8] S. Adjerid, M. Aiffa, and J. E. Flaherty. Hierarchical finite element bases for triangular and tetrahedral elements. *Comput. Method. Appl. M.*, 190:2925–2941, 2001. 3

[9] M. Ainsworth. A preconditioner based on domain decomposition for *h-p* finite-element approximation on quasi-uniform meshes. *SIAM J. Numer. Anal.*, 33(4):1358–1376, 1996. 37

[10] M. Ainsworth, G. Andriamaro, and O. Davydov. A Bernstein-Bézier basis for arbitrary order Raviart-Thomas finite elements, *preprint*. http://www.strath.ac.uk/media/departments/mathematics/researchreports/2012/18AinsworthAndriamaroDavydov2012.pdf. 82

[11] M. Ainsworth, G. Andriamaro, and O. Davydov. Bernstein-Bézier finite elements of arbitrary order and optimal assembly procedures. *SIAM J. Sci. Comp.*, 33:3087–3109, 2011. 17, 32, 35, 41, 42, 78, 98, 106, 153, 154, 163, 166, 167, 170, 182, 193, 203, 215, 221, 227

[12] M. Ainsworth and J. Coyle. Hierarchic *hp*-edge element families for Maxwell's equations on hybrid quadrilateral/triangular meshes. *Comput. Methods Appl. Mech. Engrg.*, 190:6709–6733, 2001. 4

[13] M. Ainsworth and J. Coyle. Computation of Maxwell's eigenvalues on curvilinear domains using *hp*-version Nédélec elements. In F. Brezzi, A. Buffa, S. Corsaro, and A. Murli, editors, *Numerical Mathematics and Applications*, pages 219–231. Springer Milan, 2003. 122

[14] M. Ainsworth and J. Coyle. Hierarchic finite element bases on unstructured tetrahedral meshes. *Int. J. Numer. Meth. Eng.*, 58:2103–2130, 2003. 3

[15] P. Arbenz, M. Bečka, R. Geus, U. Hetmaniuk, and T. Mengotti. On a parallel multilevel preconditioned Maxwell eigensolver. *Parallel Comput.*, 32(2):157–165, Feb. 2006. 231

[16] P. Arbenz and R. Geus. Multilevel preconditioned iterative eigensolvers for Maxwell eigenvalue problems. *Appl. Numer. Math.*, 54(2):107–121, 2005. 231

[17] D. N. Arnold, D. Boffi, and R. S. Falk. Quadrilateral $H(\mathrm{div})$ finite elements. *SIAM J. Numer. Anal.*, 42:2429–2451, 2005. 124, 125

[18] D. N. Arnold, R. S. Falk, and R. Winther. Multigrid in $H(\mathrm{div})$ and $H(\mathrm{curl})$. *Numer. Math.*, 85:197–218, 2000. 115

[19] D. N. Arnold, R. S. Falk, and R. Winther. Geometric decompositions and local bases for spaces of finite element differential forms. *Comput. Methods Appl. Mech. Engrg.*, 198:1660–1672, 2009. 3, 4, 5

[20] I. Babuska and M. Suri. The p and hp Versions of the Finite Element Method, Basic Principles and Properties. *SIAM Review*, 36(4):578–632, 1994. 63

[21] I. Babuška and M. Suri. The h-p version of the finite element method with quasiuniform meshes. *RAIRO, Math. Mod. Numer. Anal.*, 21:199–238, 1987. 14, 15

[22] I. Babuška and M. Suri. The p and hp versions of the finite element method, basic principles and properties. *SIAM Review*, 36(4):578–632, 1994. 15

[23] I. Babuška, S. Szabo, and I. N. Katz. The p-version of the finite element method. *SIAM J. Numer. Anal.*, 18(3):515–545, 1981. 2

[24] U. Banerjee and M. Suri. The Effect of Numerical Quadrature in the p-Version of the Finite Element Method. *Math. Comp.*, 59(199):1–20, 1992. 63

[25] M. Bergot and M. Duruflé. High-order optimal edge elements for pyramids, prisms and hexahedra. *J Comput. Phys.*, 232:189–213, 2013. 7, 124, 125, 126, 127, 128, 133, 134, 136, 138

[26] A. Bespalov and N. Heuer. Optimal error estimation for $H(\mathrm{curl})$-conforming p-interpolation in two dimensions. *SIAM J. Numer. Anal.*, 47:3977–3989, 2009. 15, 84

[27] M. L. Bittencourt. Fully tensorial nodal and modal shape functions for triangles and tetrahedra. *Int. J. Numer. Meth. Eng.*, 63:1530–1558, 2005. 3, 6

[28] D. Boffi, P. Fernandes, L. Gastaldi, and I. Perugia. Computational models of electromagnetic resonators: Analysis of edge element approximation. *SIAM J. Numer. Anal.*, 36:1264–1290, 1999. 3, 121

[29] A. Bossavit. Whitney forms: a class of finite elements for three-dimensional computations in electromagnetism. *IEE Proc.*, 8:493–500, 1988. 3, 87

[30] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring. *Math. Comp.*, 47:103–135, 1986. 37

[31] S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*. Texts in Applied Mathematics, 2002. 2, 12, 13

[32] P. Carnevali, R. B. Morris, Y. Tsuji, and G. Taylor. New basis functions and computational procedures for p-version finite element analysis. *Int. J. Numer. Meth. Eng.*, 36:3759–3779, 1993. 3

[33] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. North-Holland Publishing Co., Amsterdam, 1978. 2, 12, 38

[34] R. Cools and A. Haegemans. Why do so many cubature formulas have so many positive weights? *BIT Numerical Mathematics*, 28:792–802, 1988. 32

[35] M. Costabel and M. Dauge. Singularities of electromagnetic fields in polyhedral domains. *Arch. Rational Mech. Anal.*, 151(3):221–276, 2000. 3

[36] T. Davis, J. Neider, and M. Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1, Second Edition*. Addison-Wesley Longman Pub (Sd), 1997. 40

[37] C. de Boor. B-form basics. In G. E. Farin, editor, *Geometric Modeling: Algorithms and New Trends*, pages 131–148. SIAM (Philadelphia), 1987. 5, 10, 39, 44

[38] L. Demkowicz, J. Kurtz, D. Pardo, M. Paszynski, W. Rachowicz, and A. Zdunek. *Computing with hp-Adaptive Finite elements, vol. II*. Chapman & Hall/CRC, 2008. 124

[39] M. Dubiner. Spectral methods on triangles and other domains. *J. Sci. Comp.*, 6:345–390, 1991. 2, 6, 17

[40] M. G. Duffy. Quadrature over a pyramid or cube of integrands with a singularity at a vertex. *SIAM J. Numer. Anal.*, 19:1260–1262, 1982. 17

[41] T. Eibner and J. M. Melenk. Fast algorithms for setting up the stiffness matrix in *hp*-fem: a comparison. In E. A. Lipitakis, editor, *Computer Mathematics and its Applications: Advances and Developments (1994-2005)*, pages 575–596, Athens, Greece, 2006. LEA Publishers. 3, 6, 17, 24

[42] A. Ern and J. L. Guermond. *Theory and Practice of Finite Elements.* Springer-Verlag, 2004. 13

[43] R. S. Falk, P. Gatto, and P. Monk. Hexahedral $H(div)$ and $H(\mathrm{curl})$ finite elements. *ESAIM*, 45:115–143, 2011. 7, 124, 125, 126, 127, 128, 133, 134, 135, 136, 138

[44] G. Farin. *Curves and surfaces for CAGD: a practical guide, Fifth Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. 5, 40, 41

[45] R. T. Farouki and T. N. T. Goodman. On the optimal stability of the Bernstein basis. *Math. Comp.*, 65(216):1553–1566, 1996. 5

[46] J. D. Foley, A. van Dan, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Company, Inc., 1996. 40

[47] K. Gerdes, J. M. Melenk, and C. Schwab. Fully discrete *hp*-finite elements: fast quadrature. *Comput. Methods Appl. Mech. Engrg*, 190:4339–4364, 2001. 6

[48] J. Gopalakrishnan, L. E. García-Castillo, and L. F. Demkowicz. Nédélec Spaces in Affine Coordinates. *Comput. Math. Appl.*, 49:1285–1294, 2005. 4

[49] R. D. Graglia, A. F. Peterson, and F. P. Andriulli. Curl-Conforming Hierarchical Vector Bases for Triangles and Tetrahedra. *IEEE Trans. Antennas Propag.*, 59(3):950–959, 2011. 4

[50] R. D. Graglia, D. R. Wilton, and A. F. Peterson. High Order Interpolatory Vector Bases for Computational Electromagnetics. *IEEE Trans. Antennas Propag.*, 45(3):329–342, 1997. 4

[51] H. Helmoltz. über Integrale der hydrodynamischen Gleichungen, welche den Wirbelbewegungen entsprechen. *J. Reine Angew. Math.*, pages 25–55, 1858. 114

[52] R. Hiptmair. Multigrid method for Maxwell's equations. *SIAM J. Numer. Anal.*, 36:204–225, 1998. 115

[53] R. Hiptmair. Canonical construction of finite elements. *Math. Comp.*, 68(228):1325–1346, 1999. 4

[54] R. Hiptmair. Higher order Whitney forms. *PIER*, 32:271–299, 2001. 4

[55] R. Hiptmair and K. Neymer. Multilevel method for mixed eigenproblems. *SIAM J. Sci. Comp.*, 23(6):2141–2164, 2001. 231

[56] J. Hoschek and D. Lasser. *Fundamentals of computer aided geometric design.* A. K. Peters, Ltd., Natick, MA, USA, 1993. 5

[57] X.-L. Hu, D.-F. Han, and M.-J. Lai. Bivariate splines of various degrees for numerical solution of partial differential equations. *SIAM J. Sci. Comp.*, 29(3):1338–1354, 2007. 5

[58] J. Jin. *The Finite Element Method in Electromagnetics.* John Wiley & Sons Inc., 2002. 132

[59] G. E. Karniadakis and S. J. Sherwin. *Spectral/hp Element Methods for CFD, Numerical Mathematics and Scientific Computation.* Oxford University Press, New York, Oxford, 1999. 2, 4, 6, 18

[60] G. E. Karniadakis and S. J. Sherwin. *Spectral/hp element methods for computational fluid dynamics.* Oxford University Press, New York, second ed., 2005. 6

[61] G. E. Karniadakis, S. J. Sherwin, and T. C. Warburton. Basis functions for triangular and quadrilateral high-order elements. *SIAM J. Sci. Comp.*, 20:1671–1695, 1999. 6

[62] R. C. Kirby. Fast simplicial finite element algorithms using Bernstein polynomials. *Numer. Math.*, 117(4):631–652, 2011. 6

[63] R. C. Kirby and K. T. Thinh. Fast simplicial quadrature-based finite element operators using Bernstein polynomials. *Numer. Math.*, 121:261–279, 2012. 6

[64] M. J. Lai and L. L. Schumaker. *Splines Functions on Triangulations*, volume 110. Encyclopedia of Mathematics, 2007. 5, 12, 38, 39, 40, 41, 42, 44, 59, 100

[65] P. Monk. *Finite Element Methods for Maxwell's Equations*. Oxford University Press Inc., 2003. 94, 126

[66] G. Mur. Edge elements, their advantages and their disadvantages. *IEEE Trans. Magn.*, 30(5):3552–3557, 1994. 83

[67] J. C. Nédélec. Mixed finite elements in $\mathbb{R}^3$. *Numer. Math.*, 35:315–341, 1980. 3, 87

[68] B. N. Parllet. *The Symmetric Eigenvalue Problem*. SIAM Classics in Applied Mathematics, 1998. 232, 234

[69] F. Rapetti. High order edge elements on simplicial meshes. *ESAIM: M2AN*, 41:1001–1020, 2007. 4

[70] Z. Ren and N. Ida. High Order Differential Form-Based Elements for the Computation of Electromagnetic Field. *IEEE Trans. Magn.*, 36(4):1472–1478, 2000. 3

[71] J. Schöberl and S. Zaglmayr. High order Nédélec elements with local complete sequence properties. *COMPEL*, 24(2):374–384, 2005. 4

[72] L. L. Schumaker. Constructive aspects of spaces of bivariate piecewise polynomials. In J. Whiteman, editor, *Mathematics of Finite Elements VI*, pages 513–520, London, 1988. Academic Press. 5

[73] A. H. Stroud. *Approximate Calculation of Multiple Integrals*. Prentice-Hall, 1971. 18, 63

[74] D. Sun, J. Manges, X. Yan, and Z. Cendes. Spurious Modes in Finite-Element Methods. *IEEE Antennas Propagat. Mag.*, 37(5):12–24, 1995. 3, 83

[75] B. A. Szabò and I. Babuška. *Finite element analysis*. Wiley Interscience: New York, 1991. 2, 3

[76] D. B. Szyld. Criteria for combining inverse and Rayleigh quotient iteration. *SIAM J. Numer. Anal.*, 25(6):1369–1375, 1988. 232, 234, 235

[77] J. P. Webb. Hierarchical Vector Basis Functions of Arbitrary Order for Triangular and Tetrahedral Finite Elements. *IEEE Trans. Antennas Propag.*, 47(8):1244–1253, 1999. 4

[78] H. Whitney. *Geometric Integration Theory*. Princeton University Press, 1957. 87

[79] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965. 240

[80] J. Xin and W. Cai. A Well-Conditioned Hierarchical Basis for Triangular $H(\text{curl})$-Conforming Elements. *Commun. Comput. Phys.*, 9(3):780–806, 2011. 4