Spring 2023

A Node Elimination Algorithm for Cubatures of High-Dimensional Polytopes

Arkadijs Slobodkins
Southern Methodist University, aslobodkins@smu.edu

A NODE ELIMINATION ALGORITHM FOR CUBATURE OF HIGH-DIMENSIONAL POLYTOPES

Approved by:

A NODE ELIMINATION ALGORITHM FOR CUBATURE OF HIGH-DIMENSIONAL POLYTOPES

A Dissertation Presented to the Graduate Faculty of the

Dedman College

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Computational and Applied Mathematics

by

Arkadijs Slobodkins

B.S., Applied Mathematics, Southern Methodist University
M.S., Computational and Applied Mathematics, Southern Methodist University

May 13, 2023

ACKNOWLEDGMENTS

First, I would like to thank my advisor Dr. Tausch. Your mathematical creativity and ability to prove theorems on the fly is astounding. Thank you for showing me how beautiful math can be and that math is more than a science and profession. You have pushed me to strive for higher ideals in and outside of research.

I am grateful to my committee members for their contributions and being on the committee. Without the original work on the Node Elimination co-authored by Dr. Gimbutas, this dissertation would be on a different, perhaps not as fascinating, research topic. I am inspired by Dr. Hagstrom, whose mathematical rigor and understanding of the subject is unmatched and with whom I had the opportunity to work with. As the author of the subject he is teaching, Dr. Xu has taught me to approach my work with maximum attention to detail and dedication.

Thank you Dr. Reynolds for introducing me to the field of high-performance computing(HPC) and your advice. In the recent years, HPC has been one of my main areas of interest and I am glad to continue my career in this field.

I am thankful to my parents for their infinite love and support. They have always believed in me and supported the decision of moving to a different country and pursuing a challenging degree.

I am thankful to my friends from Europe and the United States. Without them, this journey would not be the same.

Finally, I would like to thank everyone who is not mentioned. As a graduate student, I became acquainted with extraordinary professors and colleagues.

Slobodkins , Arkadijs B.S., Applied Mathematics, Southern Methodist University
M.S., Computational and Applied Mathematics, Methodist University

A Node Elimination Algorithm for
Cubature of High-Dimensional Polytopes

Advisor: Professor Johannes Tausch
Doctor of Philosophy degree conferred May 13, 2023
Dissertation completed April 20, 2023

Node elimination is a numerical approach for obtaining cubature rules for the approximation of multivariate integrals over domains in R^n. Beginning with a known cubature, nodes are selected for elimination, and a new, more efficient rule is constructed by iteratively solving the moment equations. In this work, a new node elimination criterion is introduced that is based on linearization of the moment equations. In addition, a penalized iterative solver is introduced that ensures positivity of weights and interiority of nodes. We aim to construct a universal algorithm for convex polytopes that produces efficient cubature rules without any user intervention or parameter tuning, which is reflected in the implementation of our package gen-quad. Strategies for constructing the initial rules for various polytopes in several space dimensions are described. Highly efficient rules in four and higher dimensions are presented. The new rules are compared to those that are obtained by combining transformed tensor products of one dimensional quadrature rules, as well as with known analytically and numerically constructed cubature rules.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

To all who find numerical integration useful or interesting. Or both.

Chapter 1

INTRODUCTION

1.1. Preliminaries

Let Ω be a domain in $\mathbb{R}^d$. The goal of the construction of cubature rules is to determine the nodes x_k and weights w_k in the cubature rule

$$\int_\Omega \phi(x)w(x)\,dx \approx \sum_{k=1}^{n} \phi(x_k)w_k, \tag{1.1}$$

such that the rule is exact for multivariate polynomials of degree up to p

$$\mathbb{P}_p^d = \operatorname{span}\left\{x^\alpha : \alpha_1 + \cdots + \alpha_d \le p\right\}. \tag{1.2}$$

Here, α is a multi index and $x^\alpha = x_1^{\alpha_1} \cdots x_d^{\alpha_d}$. The dimension of the linear space $\mathbb{P}_p^d$ is

$$M = \dim \mathbb{P}_p^d = \binom{p+d}{d}.$$

From the viewpoint of numerics the monomial basis is ill-conditioned, and so we consider a more general basis $\phi_1, \ldots, \phi_M$ of $\mathbb{P}_p^d$, for instance, orthogonal polynomials. We set

$$\Phi(x) = \begin{bmatrix} \phi_1(x) \\ \vdots \\ \phi_M(x) \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} \int_\Omega \phi_1(x)w(x)\,dx \\ \vdots \\ \int_\Omega \phi_M(x)w(x)\,dx \end{bmatrix}. \tag{1.3}$$

Further, we write for the vector of nodes and weights, $\mathbf{x} \in \mathbb{R}^{dn}$, $\mathbf{w} \in \mathbb{R}^n$, respectively, and

$$\boldsymbol{\Phi}(\mathbf{x}) = \big[\Phi(x_1), \ldots, \Phi(x_n)\big] \in \mathbb{R}^{M\times n}. \tag{1.4}$$

Exactness in $\mathbb{P}_p^d$ means that the nodes and weights must be solutions of the moment equations

$$\mathbf{f}(\mathbf{x},\mathbf{w}) = \mathbf{\Phi}(\mathbf{x})\mathbf{w} - \mathbf{b} = \mathbf{0}, \tag{1.5}$$

which is a polynomial system in $N = (d+1)n$ unknowns and M equations. The resulting system

$$\mathbf{\Phi}(\mathbf{x})\mathbf{w} = \mathbf{b} \tag{1.6}$$

has been researched extensively in the context of numerical integration [8, 51], and is known in literature as moment equations [47, 50]. We write (1.5) in a more convenient form by combining nodes and weights into one vector. Thus we let

$$z_k = [x_k, w_k] \in \mathbb{R}^{d+1} \qquad \text{and} \qquad \mathbf{z} = [z_1, \ldots, z_n]^T \in \mathbb{R}^N. \tag{1.7}$$

Since we are looking for nodes in the domain Ω that have positive weights, a feasible cubature rule is in the set

$$Z_n = \big\{\mathbf{z} \in \mathbb{R}^N : \, \mathbf{f}(\mathbf{z}) = \mathbf{0}, \; x_k \in \Omega, \; w_k \geq 0, \; 1 \leq k \leq n\big\}. \tag{1.8}$$

Collectively these conditions are called PI constraints, meaning positivity of weights and interiority of nodes.

One-dimensional integration is well studied and optimal quadrature[1] rules have been known since 19th century [52]. They are known as Gauss quadrature rules or Gaussian quadrature. Besides optimality and their analytical construction, Golub and Welsch [23] have shown that Gauss rules can be computed by finding eigenvalues of tridiagonal matrices. This appears to be strictly limited to one-dimensional case.

We call the rule optimal if it has as few number of nodes as possible for a given degree of accuracy. However, what is the minimal number of nodes required to achieve a given

[1]In one-dimensional case, numerical approximation of the integral is commonly referred as quadrature, whereas in two and higher dimensions we refer to it as cubature.

degree of accuracy in two and higher dimensions is an open research question [39, 40]. In addition, deriving cubature rules analytically becomes more challenging with the increase of dimension [19, 50, 12]. Instead, numerical approaches are sought.

One approach is to take the tensor product of one-dimensional rules(typically Gauss-Legendre rules), and compute n-dimensional tensor product to obtain rule over an n-dimensional cube. However, the resulting rules contain substantially more unknowns than equations and are suboptimal in terms of efficiency. Suboptimality increases with dimension.

Besides the exponential growth of the tensor product rules, their applicability is limited only to a narrow family of domains, such as n-dimensional cubes and simplexes. Other fundamental approaches include extensively researched Monte Carlo based methods and Smolyak rules[45]. Monte Carlo methods are often used for computing integrals in very high dimensions to overcome the curse of dimensionality of the numerical cubature [29, 12, 5]. On the other hand, methods based on Smolyak Quadrature provide an effective alternative to Monte Carlo methods[21]. However, such approach is limited to hypercubes. In this work, moment equations approach is pursued.

Although optimal number of nodes n is generally unknown, a common assumption is that optimal or near optimal cubature rules are achieved when the polynomial system in (1.5) is satisfied when the number of equations equals the number of unknowns, i.e. $N = M$. Based on this assumption, an optimality index for a cubature rule with n nodes is defined as the ratio

$$i_{opt} = \frac{n}{n_{opt}} \tag{1.9}$$

where

$$n_{\text{opt}} = \left\lceil \frac{\dim \mathbb{P}_p^d}{(d+1)} \right\rceil . \tag{1.10}$$

If $i_{opt} = 1$ is achieved, the rule is considered optimal. In some cases, it is possible that (1.5) has solution with $i_{opt} < 1$. Since this appears to be an exceptional case, this is not considered further in this thesis.

In the one-dimensional case, it is proven that Gauss-Legendre are the optimal rules with respect to the weight function $w(x) = 1$, and its efficiency index is $i_{opt} = 1$. Thus, (1.10) can also be viewed as the generalized assumption about the optimality based on dimension $d = 1$. There has been an extensive research on deriving optimal cubature rules for the case that Ω is a domain in two dimensions, squares and triangles in particular[43, 27, 28, 22, 15]. In the recent years, various approaches have been investigated to achieve efficient rules in three dimensions[31, 35, 16]. However, research of high higher-dimensional rules is much more sparse and poses significant challenges[12]. In this work, we focus on integration over three, four, five, and six-dimensional polytopes.

The main polytopes of interest are n-dimensional cubes, tetrahedra, tensor products of these domains, and a 3-dimensional pyramid. To be precise, they are defined as follows

P_3: 3-dimensional pyramid : $P3 = \left\{x \in R^3 : \ \ 0 \le x_1 \le 1, \ \ 0 \le x_2, x_3 \le x_1\right\}$,

C_d: d-dimensional cube : $C_d = \left\{x \in R^d : \ \ 0 \le x_k \le 1, \ \ k = 1...d\right\}$,

T_d: d-dimensional simplex(tetrahedron) : $T_d = \left\{x \in R^d : \ \ 0 \le x_d \ ... \le x_1 \le 1\right\}$,

$C_i \times T_j$: tensor product of C_i and T_j : $C_i \times T_j = \left\{x \in R^{i+j} : \ x = (x_1, x_2), \ x_1 \in C_i, \ x_2 \in T_j\right\}$,

$T_i \times T_j$: tensor product of T_i and T_j : $T_i \times T_j = \left\{x \in R^{i+j} : \ x = (x_1, x_2), \ x_1 \in T_i, \ x_2 \in T_j\right\}$.

1.2. Motivation

High-dimensional numerical integration is an important tool for solving numerous science and engineering problems [39]. In practice, finding integrals analytically is often challenging or intractable. For example, Finite Element method(FEM) requires computing mass and stiffness matrices, where each element of a matrix is an integral. Similarly, Boundary Element

Method(BEM) is based on computing a matrix of multi-dimensional integrals [2] [48].

In the case of the BEM, singular integrals arise, which can be treated by applying singularity removing transformations. They result into integrals over four-dimensional polytopes, such as tensor products of cubes and tetrahedra [41]. For Galerkin discretizations of fractional PDEs, similar transformations can be applied, which result in four or six dimensional polytopes [20].

Problems in FEM are often solved in two and three-dimensional space, and therefore require computing double and triple integrals [31]. Although higher-dimensional FEM is limited in use due to implementation challenges, it is a promising tool for solving space-time problems in three dimensions, which require four-dimensional integration[17]. Over the last two decades, of special interest are 4D simplex elements [33]. Moreover, these challenges are gradually improved due to progress in technology[19].

Cubes and tetrahedra are of foundational importance in numerical integration. High-dimensional tensor product domains arise in application of BEM, and therefore are directly relevant to our work. We did not initially intend to study pyramid, but it is used to show the generality of our algorithm and that it has potential to be generalized well to other domains.

1.3. Contributions

Node elimination algorithm is an iterative procedure, which was successfully applied to squares, cubes, and triangles[50]. Subsequently, more work has appeared to use variants of the node elimination algorithm to derive rules over two-dimensional polygons and tetrahedra. However, our experiments have shown that its efficiency deteriorates in higher dimensions. In addition, the applicability of the original approach quickly becomes limited due to high computational cost. Therefore, the goal of this work is to formulate a new node elimination algorithm that would generalize well in four and higher dimensions without compromising efficiency in two and three dimensions.

[2]In many cases, quadratures in FEM calculations can be carried analytically, but the case of non-constant coefficients or curvilinear elements one must resort to cubature.

Node Elimination starts with a known sub-optimal rule(e.g. tensor product rule) and then eliminates nodes using a predictor and corrector procedure. The predictor step aims to produce an accurate approximation to the cubature rules with one fewer node, whereas the corrector step is a nonlinear solver that uses initial guess from predictor to find the exact cubature using moment equations. For both of these steps, we will introduce constrained optimization techniques.

Predictor There are multiple ways to choose the criterion for eliminating a node in the predictor step. One way is to use the significance index originally introduced in [50], which will be described in Chapter 3. Instead of significance index, we use a predictor based on the linearization of the moment equations, which produces highly accurate initial guesses.

Corrector Due to increasing size of the nonlinear system in higher dimensions, finding solutions subject to constraints in (1.8) becomes increasingly difficult. To address that, a penalized Newton's method is introduced, which enforces the penalty if the Newton's update tends towards the boundary of Ω and if weights tend to zero. Such strategy is not only beneficial for solving the given nonlinear system, but it also improves the distribution of nodes and weights, allowing to successfully eliminate more nodes in the long term.

Computational cost High computational cost is addressed by recursively reusing lower dimensional cubature rules obtained by means of node elimination. A tensor product of the interval and $d-1$ dimensional domain is combined to obtain the initial guess for the domain of dimension d. For tetrahedra, pyramids, and other polytopes, the Duffy transformation is used to map the resulting tensor product to the desired domain.

Software We present and describe object oriented and parallel implementation of a numerical integration package gen-quad. Based on newly introduced features, it is capable of computing cubature rules of arbitrary degree of precision and dimension for various polytopes.

In theory, the degree and dimension in our implementation are arbitrary. In practice, however, our algorithm is limited to moderate dimensions(six or seven). To address rapidly growing problem in higher dimensions, the package is parallelized using OpenMP and C++ programming language. After describing implementation, results for multiple domains in different dimensions are compared to related publications. To conclude, possibilities for further generalization of the algorithm to even higher dimensions and broader family of polytopes are discussed.

Chapter 2

Overview

Node Elimination Algorithm is an iterative procedure for obtaining cubature rules via the process of eliminating nodes. Given a suitable initial guess of a quadrature rule, more efficient rules are obtained by eliminating a node and solving the nonlinear system (1.5), with the goal of eliminating as many nodes as possible. In brief, Node Elimination can be summarized as follows:

- Compute the starting initial guess (preferably analytical) and run iterative Node Elimination Algorithm.
- At each iteration of the node elimination, generate a set of initial guesses by eliminating a node.
- Solve the resulting nonlinear system of equations with one of the initial guesses from the set.
- Exit when optimum has been reached or no more nodes can be eliminated.

Figure(1) illustrates one iteration of the algorithm over a unit square. It is important to note that successful cubature rule must satisfy PI constraints.

A combination of the node elimination and solution of the nonlinear system forms a predictor-corrector type approach. Therefore, to maximize successful eliminations, we aim to optimize predictor-corrector algorithm as a whole, rather than each one at a time. To achieve that, similar constrained optimization techniques will be used for both the node elimination strategy(predictor) and nonlinear solver(corrector).

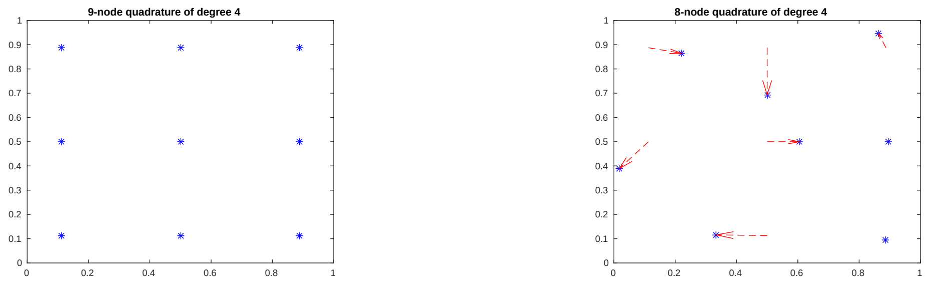

Figure 2.1. One step of the Node Elimination Algorithm over $\Omega = C_2$.

2.1. Advantages

The main advantage of the Node Elimination algorithm is its flexibility. One can start with a starting initial guess that is suboptimal in terms of efficiency, but can be easily found either numerically or analytically. In fact, the initial guess does not even have to be exact, since it can be corrected by a nonlinear solver. However, we ensure that initial guess is always analytical in a sense that polynomial basis S_p^d is integrated exactly.

Although numerous possibilities of eliminating a node complicate determination of the optimal elimination strategy, they provide freedom for selecting alternative solutions if one or multiple node eliminations did not succeed. For example, depending on the elimination algorithm, $O(n)$ or even $O(n^2)$ guesses due to eliminating any of the n nodes are generated. It will be shown that under suitable metric and effective algorithm, only a small subset of these guesses should be considered. For example, initial cubature guesses with nodes farthest away from the boundary of the domain or closest to the exact cubature rule are conducive to solving the moment equations.

Given some information about the domain of interest, a single approach is used to tackle cubature rules of different degrees, dimensions, and various domains. For example, eliminating a single node without moving other nodes can be applied to arbitrary domain. Similarly, linear inequality constraints of a polytope can be utilized to reuse the node elimination algorithm for any convex polytope. The main goal of this work is to maximize genericity in a sense that efficiency of the final cubature rule is not compromised by applying the same algorithm to multiple domains.

2.2. Challenges

One of the challenges at each iteration of the algorithm is to choose the criterion that determines which of the n nodes is eliminated. Another challenge is to solve the resulting nonlinear system in a way that would be beneficial for the subsequent iterations of the node elimination. In other words, we are interested not only in the solution of a particular nonlinear system, but also how the nodes and weights evolve as a consequence of solving one nonlinear system after another. Final success of the algorithm can vary considerably depending on the choice of the initial cubature.

On the other hand, solving the nonlinear system of moment equations under constraints is a challenging problem on its own. The corrector step is based on solving the system using Gauss-Newton method. It has the advantage of a quadratic convergence yet is less expensive than its popular alternative, Levenberg-Marquardt. However, it is only locally convergent, hence we must always start close to the solution of the system.

Intuitively, one could consider brute-force approach, i.e. solve the nonlinear system corresponding to each of the n initial guesses at the first iteration, then for all successful solutions repeat that in the subsequent iteration, and so on. However, given that nonlinear system itself becomes increasingly expensive with the increase of degree and dimension, such approach would quickly become unfeasible. Besides, it would hardly yield any useful information about the properties of the optimal cubature rules and how to obtain them in higher dimensions. Instead, displacement and relationship between nodes across multiple iterations should be investigated in more detail. In the following section, more systematic approaches to eliminating a node are considered.

2.3. Strategies

The original strategy in[50] is to eliminate a node and keep other nodes as they are. The significance index is defined as either

$$s_j = w_j \sum_{i=1}^{n} \phi_i^2(\mathbf{x}_j), \quad j = 1, ..., k,$$

or

$$s_j = \sum_{i=1}^{n} \phi_i^2(\mathbf{x}_j), \quad j = 1, ..., k.$$

Significance indices are sorted in increasing order. The idea is that the least significant index corresponds to the node that contributes to the cubature the least, and therefore is a good candidate for elimination.

One could also eliminate a node by merging two nodes

$$\hat{\mathbf{x}}_{ij} = \frac{\mathbf{x}_i + \mathbf{x}_j}{2}$$

and adding corresponding weights

$$\hat{w_{ij}} = w_i + w_j.$$

In the node merging case, the analogue of significance index is the distance between $\mathbf{x}_i$ and $\mathbf{x}_j$. The smaller the distance, the more they resemble the behavior of a single node, which improves accuracy of the initial guesses.

Both strategies have the desirable property that nodes and weights of the initial guesses satisfy constraints. In addition, they have a relatively low computational cost. However, they produce guesses that are usually farther from solution manifold than we would want, thus the corrector step may need many iterations to converge or may not converge at all. Instead, we formulate optimization procedure that is based on linearizing the solution manifold, which yields more accurate initial guesses. However, not all of the generated guesses are guaranteed to satisfy the constraints. Therefore, we have to add a penalty term. We will show that in higher dimensions adding constraints is advantageous not only for eliminating one node at a time, but also for maximizing the efficiency index of the final cubature.

2.4. Arbitrary Dimension

If a suitable guess of degree p is found, the algorithm is independent of the degree. To extend the algorithm for a particular domain to an arbitrary dimension, a suitable starting initial guess and polynomial basis must be constructed. To construct a polynomial basis, a set of multi-indices

$$S_p^d = \{\alpha : \alpha_1 + \cdots + \alpha_d \leq p\}$$

must be generated and applied to desired polynomials. It will be shown that such set is obtained by means of nested recursion. The number of basis functions corresponds to the size of the set, which is

$$Dim(\mathbb{P}_p^d) = \frac{(d+p)!}{d!p!}.$$

One way to compute a starting initial guess for cubature over a cube is to compute a tensor product of Gauss-Legendre nodes, which results in $\lceil \frac{deg+1}{2} \rceil^{dim}$ nodes and weights. Such approach is prohibitively expensive in higher dimensions, and hence a recursive procedure is used to reduce the initial number of nodes. For other domains, such as tetrahedron or a pyramid, an appropriate variant of the Duffy transformation will be used.

Chapter 3

Numerical Methods

In this chapter a predictor-corrector approach is described in detail. Based on linear inequality constraints, a generic approach is formulated to address various domains efficiently. In section 3.1, we define linear inequality constraints and how they are applied to the nodes of a cubature. Sections 3.2 and 3.3 describe predictor and corrector, respectively. In the last section, recursive procedure for improving efficiency of the initial guess is presented.

3.1. Linear Inequality Constraints

Consider a node $x \in \mathbb{R}^d$. We require that cubature rule q with n nodes is valid only if all nodes are inside of the domain Ω. Since in our work Ω is a convex polytope, it can be described by a system of linear inequalities

$$\Omega = \{ \ x \in R^d : Ax \leq b \ \ \}.$$

Therefore, for each node $x \in R^d$ of q, we can express these conditions in terms of inequality constraints

$$x_k \in \mathbb{R}^d : Ax_k \leq b, \quad k = 0, ..., n,$$

where the matrix A and the vector b define constraints for a specific polytope. In addition, we require all weights of q to be nonnegative

$$w_k \geq 0, \quad k = 0, ..., n.$$

Collectively, they are referred to as PI constraints, meaning that weights are positive and nodes are inside Ω. We combine these constraints and write

$$z_k \in \mathbb{R}^{d+1} : \hat{A} z_k \leq \hat{b}, \quad k = 0, ..., n, \tag{3.1}$$

where

$$z_k = \begin{bmatrix} w_k \\ x_k \end{bmatrix}, \quad \hat{A} = \begin{bmatrix} -1 & \underline{0}^T \\ \underline{0} & A \end{bmatrix}, \quad \hat{b} = \begin{bmatrix} 0 \\ b \end{bmatrix}.$$

Figure 3.1 illustrates the relationship between nodes and the matrix A for C_2.

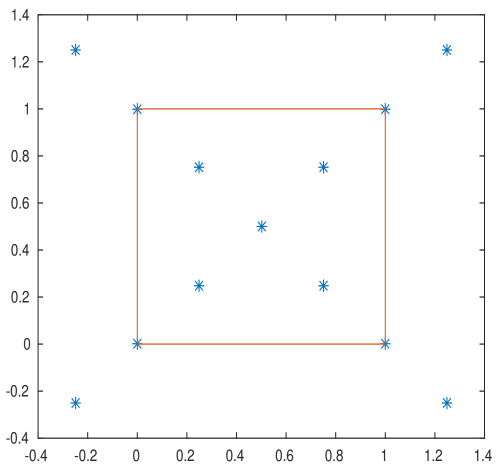

Figure 3.1. Example for $\Omega = C_2$. Nodes that are inside C_2(marked in red) satisfy inequality constraints strictly: $Ax < b$, some nodes are on the boundary: $Ax = b$, and some nodes are outside of the domain and do not satisfy constraints: $Ax > b$.

If

$$\mathbf{z} = \Big[z_1, z_2, \ \ldots, \ z_k\Big] \in \mathbf{R}^{(n+1)d},$$

then a set of valid cubature rules is

$$Z_n = \{\mathbf{z} \in R^{(n+1)d} \ : \ \phi(\mathbf{x})\mathbf{w} = \mathbf{b} \text{ and } z_k \text{ satisfy (3.1)}\}.$$

3.2. Penalized Least Squares Newton Method

As we already mentioned, the node elimination procedure is a predictor-corrector type method. In this section we describe the corrector step, which for a point $\widetilde{\mathbf{z}} \notin Z_n$, attempts to find a nearby point on the solution manifold, i.e., $\bar{\mathbf{z}} \in Z_n$.

3.2.1. Unconstrained Least Squares Newton

A common approach to solve an unconstrained nonlinear system is to use the Least Squares Newton (or sometimes Gauss-Newton) algorithm. It benefits from a rapid convergence and typically requires few iterations, but is not robust in a sense that it requires a sufficiently accurate initial guess to converge. A more robust approach is the Levenberg-Marquardt algorithm, which has a slower, but comparable rate of convergence[7]. It is especially useful if the initial guess is far from the solution. However, for our application Levenberg-Marquardt has a higher computational cost due to the underdetermined system of equations. In addition, the corrector ensures that the initial guess is sufficiently accurate and thus Least Squares Newton is a solver of choice.

The goal of the nonlinear solver is to solve $f(\mathbf{z}) = 0$ or minimize the residual $\|f(\mathbf{z})\|$. In the case of an overdetermined system, there is typically no exact solution to $f(\mathbf{x}) = 0$, whereas there are infinitely many solutions for an underdetermined system. Since our system is typically underdetermined, the solutions of $f(\mathbf{z}) = 0$ are on a lower dimensional manifold. We will exploit the availability of multiple solutions and select the one that is more likely to produce the most efficient final cubature rule. Due to double precision arithmetic, we accept the solution when the residual is less than 10^{-14}.

Each iteration of the Gauss-Newton method consists of updating the current iterate using the least squares solution of the linear system

$$J\Delta\mathbf{z} = \mathbf{f}, \tag{3.2}$$

where J is the Jacobian of $\mathbf{f}(\mathbf{w}, \mathbf{x})$

$$J(\mathbf{w}, \mathbf{x}) = \left[\begin{array}{c} \phi_1(x) \; \dots \; \phi_M(x) \; w_1\phi'(x) \; \dots \; w_m\phi'_M(x) \end{array} \right]. \tag{3.3}$$

The solution of 3.2 is given by applying the pseudo-inverse $\Delta\mathbf{z} = J^T \left(JJ^T\right)^{-1}\mathbf{f}$, which is illustrated in figure 3.2. Therefore, the update is

$$\mathbf{z}_{new} = \mathbf{z}_{cur} + \Delta\mathbf{z} = \mathbf{z}_{cur} - J^T \left(JJ^T\right)^{-1}\mathbf{f}, \tag{3.4}$$

where $\mathbf{f} := \mathbf{f}(\widetilde{\mathbf{z}}) \in \mathbb{R}^M$, $J := D\mathbf{f}(\widetilde{\mathbf{z}}) \in \mathbb{R}^{M\times N}$ is the Jacobian of $\mathbf{f}(\widetilde{\mathbf{z}})$, $\mathbf{z}$ is a vector of nodes and weights. If Newton's method converges and constraints are satisfied, $\mathbf{z}$ is a new cubature rule.

Although the unconstrained Gauss-Newton method proved to work well for eliminating nodes over two and three-dimensional domains, complications arise in higher dimensions. As the size of the system grows, satisfying inequality constraints for all nodes and weights becomes increasingly difficult. During one of the iterations of the Newton's method, one or more nodes or weights tend to violate inequality constraints. To that end, additional penalty term will be examined in section 3.2.3.

3.2.2. Damping

Since finding solutions that satisfy constraints for all nodes and weights becomes more challenging for larger problems, alternatives to unconstrained Gauss-Newton must be sought. Consider a damped version of the update

$$\alpha\Delta\mathbf{z} = \alpha J^T \left(JJ^T\right)^{-1}\mathbf{f}, \quad 0 < \alpha \leq 1. \tag{3.5}$$

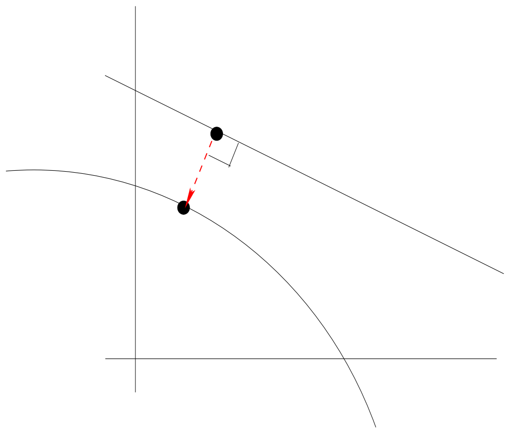

Figure 3.2. Two-dimensional illustration of the corrector step. The straight line represents the tangent space of $\mathbf{f}$ at $\widetilde{\mathbf{z}}$, whereas curved line is a manifold of feasible solutions. Least squares Newton attempts to find the solution orthogonal to the tangent space. If successful, convergence is expected to be quadratic. However, convergence and PI criteria are not guaranteed.

The idea is to choose α so that nodes stay further from the boundary. We define $dist(\mathbf{z})$ to be the distance of the constraint that is closest to the boundary

$$dist(\mathbf{z}) = \min_i \min(\mathbf{b} - A\mathbf{z}_i), \quad i = 1, \; ... \; n. \tag{3.6}$$

Based on that, the distances of the current iterate and the update are

$$dist_1 = dist(\mathbf{z}_{cur}), \tag{3.7}$$

$$dist_2 = dist(\mathbf{z}_{cur} + \Delta\mathbf{z}), \tag{3.8}$$

and α is defines as

$$\begin{cases} \alpha = \frac{dist_2}{dist_1} \quad \text{if}(dist_1 > dist_2), \\ \alpha = 1 \quad \text{otherwise.} \end{cases} \tag{3.9}$$

The interpretation is that if the subsequent iterate is further inside of the domain than the previous iterate, it is a desirable search direction, and no damping is needed. On the other hand, if it travels towards the boundary, damping is introduced. Figure 3.3 is a simplified representation of the procedure. It shows how damping is computed and applied to the node that ends up closest to the boundary, whereas it is applied to all the remaining nodes as well. Reciprocal ensures that stronger damping is applied the more update travels towards the boundary.

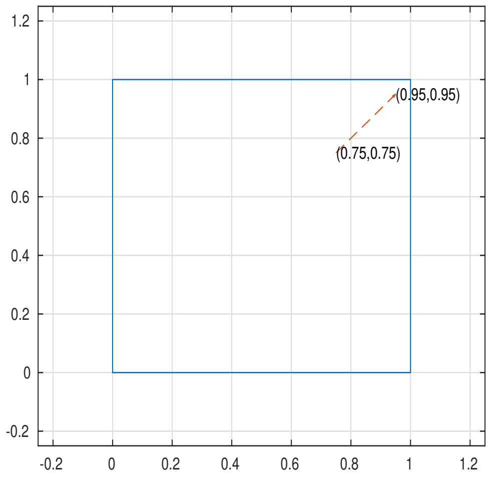

Figure 3.3. Example for $\Omega = C_2$. Infinity norm of the distance is 5 times smaller for the update, thus α is set $0.05/0.25 = 0.2$ Damping with respect to 2-norm is a valid alternative.

3.2.3. Penalty Term

The damping strategy produced more reliable results in higher dimensions, especially for more involved domains, such as $C_3 \times T_3$ or $T_3 \times T_3$ and reduced the number of nodes in the final quadrature rule in many cases. However, efficiency index continued to deteriorate considerably in higher dimensions, hence the improved version of a penalty was introduced.

With $\mathbf{z}$ defined as in (1.7), we set

$$\phi_\Omega(\mathbf{z}) = \sum_{j=1}^{n} \left[\varphi_\Omega(x_j) + \log \frac{1}{w_j} \right], \tag{3.10}$$

where $\varphi_\Omega(\cdot)$ is a function that is smooth in Ω and has a logarithmic singularity on the boundary of Ω. For instance, if Ω is a convex polytope given by the linear inequalities, then we set

$$\Omega = \left\{ x \in \mathbb{R}^d : Ax \leq b \right\} \quad \Rightarrow \quad \varphi_\Omega(x) = \sum_{\ell=1}^{\ell_A} \log \left(\frac{1}{b_\ell - a_\ell^T x} \right),$$

where a_ℓ^T are the rows of A and ℓ_A is the number of rows. For domains different from polytopes, other penalty terms have to be constructed. For instance, if Ω is a sphere, the penalty term is

$$\Omega = \left\{ x \in \mathbb{R}^d : \|x\|_2 \leq 1 \right\} \quad \Rightarrow \quad \varphi_\Omega(x) = \log \left(\frac{1}{1 - \|x\|_2} \right).$$

To derive an iterative solver that maps $\widetilde{\mathbf{z}} \notin Z_n$ to $\bar{\mathbf{z}} \in Z_n$ let $\Delta\mathbf{z} = \widetilde{\mathbf{z}} - \bar{\mathbf{z}}$, and consider the constrained optimization problem

$$\min_{\Delta\mathbf{z}} \left\{ \frac{1}{2} \|\Delta\mathbf{z}\|^2 + t\, \phi_\Omega(\widetilde{\mathbf{z}} + \Delta\mathbf{z}),\ \widetilde{\mathbf{z}} + \Delta\mathbf{z} \in Z_n \right\}. \tag{3.11}$$

Here, the parameter $t \geq 0$ controls the strength of the penalty term. We will provide more detail about how to determine t later on.

Now linearize (3.11) as follows

$$\begin{gathered} \min \frac{1}{2} \|\Delta\mathbf{z}\|^2 + t\, \mathbf{g}^T \Delta\mathbf{z} \\ \Delta\mathbf{z} : \mathbf{f}(\widetilde{\mathbf{z}}) + J\Delta\mathbf{z} = \mathbf{0}, \end{gathered} \tag{3.12}$$

where $\mathbf{f} := \mathbf{f}(\widetilde{\mathbf{z}}) \in \mathbb{R}^M$, $J := D\mathbf{f}(\widetilde{\mathbf{z}}) \in \mathbb{R}^{M \times N}$ is the Jacobian of $\mathbf{f}(\widetilde{\mathbf{z}})$ and $\mathbf{g} := \nabla\phi_\Omega(\widetilde{\mathbf{z}}) \in \mathbb{R}^N$ is the gradient of the penalty term. Using Lagrange multipliers it follows that (3.12) is equivalent to the linear system

$$\Delta\mathbf{z} + J^T\boldsymbol{\lambda} = -t\mathbf{g}$$
$$J\Delta\mathbf{z} = -\mathbf{f},$$

where $\boldsymbol{\lambda} \in \mathbb{R}^M$ is the Lagrange multiplier. Eliminating $\boldsymbol{\lambda}$ gives the solution of (3.12) in the the normal equations form

$$\begin{aligned}\Delta\mathbf{z} &= -J^T\left(JJ^T\right)^{-1}\mathbf{f} - t\left(I - J^T\left(JJ^T\right)^{-1}J\right)\mathbf{g}, \\ &= \Delta\mathbf{z}^f + t\Delta\mathbf{z}^g.\end{aligned} \tag{3.13}$$

Here, $\Delta\mathbf{z}^f = -J^T\left(JJ^T\right)^{-1}\mathbf{f}$ is the least squares solution of the underdetermined system $\mathbf{f} + J\Delta\mathbf{z} = \mathbf{0}$, i.e., the solution of (3.11) when $t = 0$. The second term $\Delta\mathbf{z}^g = -\left(I - J^T\left(JJ^T\right)^{-1}J\right)\mathbf{g}$ is the orthogonal projection of $-\mathbf{g}$ onto the nullspace of J. The form in (3.13) makes clear that the solution of (3.12) for any parameter t can be obtained by computing $\Delta\mathbf{z}^f$ and $\Delta\mathbf{z}^g$ independently of t and then forming the appropriate linear combination.

For numerical purposes it is better to compute these two vectors with the LQ factorization. If $J = LQ$ is the economy size factorization, i.e., $L \in \mathbb{R}^{M\times M}$ is lower triangular and $Q \in \mathbb{R}^{M\times N}$ has orthonormal rows, then

$$\begin{aligned}\Delta\mathbf{z}^f &= -Q^TL^{-1}\mathbf{f}, \\ \Delta\mathbf{z}^g &= -(I - Q^TQ)\mathbf{g}.\end{aligned} \tag{3.14}$$

Once $\Delta\mathbf{z}^f$ and $\Delta\mathbf{z}^g$ have been computed, the Newton update is

$$\widetilde{\mathbf{z}} \leftarrow \widetilde{\mathbf{z}} + \Delta\mathbf{z}^f + t\Delta\mathbf{z}^g. \tag{3.15}$$

We now describe how the parameter t is determined. It is determined such that the nodes of the next iterate have maximal distance from the boundary and weights are as large as possible. We focus on the case that Ω is a convex polytope given by the linear inequalities

$Ax \le b$.

Substitution of the j-th node and weight of the Newton update (5.3) into the linear constraints results into two types of inequalities

$$\begin{aligned} a_\ell^T(\widetilde{x}_j + \Delta x_j^f) - b_\ell + t a_\ell^T \Delta x_j^g &\le 0 \\ -\widetilde{w}_j - \Delta w_j^f - t\Delta w_j^g &\le 0 \end{aligned}$$

for $1 \le j \le n$, $1 \le \ell \le \ell_A$. Here $\widetilde{x}_j$, Δx_j^f and Δx_j^g are the nodal components and $\widetilde{w}_j$, Δw_j^f and Δw_j^g are the weights of the vectors $\widetilde{\mathbf{z}}$, $\Delta\mathbf{z}^f$ and $\Delta\mathbf{z}^g$, respectively. We write these conditions collectively as

$$m_k(t) = \beta_k + t\alpha_k \le 0, \quad 1 \le k \le (\ell_A + 1)n, \tag{3.16}$$

where the α_k's list all inner products $a_\ell^T \Delta x_j^g$ and all Δw_j^f's and the β_k's are defined analogously.

Algorithm 3.1 finds the value of t that minimizes the maximum over k for this family of straight lines. The algorithm starts with the maximum at $t = 0$ and follows the largest line until it intersects with another line with positive slope.

```
Algorithm 3.1    $t = 0$
  $k = \operatorname*{argmax}_j \beta_j$
  while $\alpha_k < 0$ do
      Find $k^* = \operatorname*{argmin}_k \{t_{jk},\ t_{jk} > t\}$
      $t = t_{k^*k}$
      $k = k^*$
  end while
```

If successful, the algorithm returns the t-value for which the min max of the family of straight lines is achieved. This guarantees that the next iterate has maximal distance from the boundary. Note that even though $\widetilde{\mathbf{z}}$ satisfies the constraints, the least squares solution $\widetilde{\mathbf{z}} + \Delta\mathbf{z}^f$ may not. Therefore, it is possible that the returned value of $m_k(t)$ is positive in which case at least one node does not satisfy the constraints. In the latter case one can resort to a damped Newton method, where $\Delta\mathbf{z}^f$ is multiplied by a small factor. The theoretical cost of this algorithm is $O(\ell_A^2 n^2)$. However, in practice, the while loop terminates after a small number of steps, thus its computational cost is closer to $O(\ell_A n)$ and negligible compared to the cost of computing the LQ factorization of the matrix J.

We have described algorithm 3.1 for a convex polytope. However, it can be modified for other integration domains as well. For instance, if Ω is a solid sphere then the constraints result in a family of parabolas and completely analogous algorithm can be applied to optimize the parameter t.

The constrained LS Newton method is summarized in algorithm 3.2.

Algorithm 3.2

```
while $\|f(\widetilde{\mathbf{z}})\| > TOL$ do
    Setup $\mathbf{f}$, $\mathbf{g}$ and $J$, and compute the LQ-factorization of $J$.
    Calculate $\Delta\mathbf{z}^f$ and $\Delta\mathbf{z}^g$ in (5.4).
    Determine $t$ from algorithm 3.1.
    Set $\widetilde{\mathbf{z}} \leftarrow \widetilde{\mathbf{z}} + \Delta\mathbf{z}^f + t\Delta\mathbf{z}^g$
end while
```

3.3. Node Elimination

In this section we describe a method to reduce the number of points in a cubature rule while maintaining its degree. It consists of determining points on Z_n that intersect with the coordinate planes $w_k = 0$. Obviously, if a weight vanishes in (1.1), then the corresponding node does not have to be included in the cubature rule (1.1). Thus we have found another solution of (1.5) with $n-1$ nodes that has the same degree of precision. The elimination procedure is then repeated until no further can be found.

We start with a point $\bar{\mathbf{z}}$ on Z with all positive weights and use a predictor-corrector type approach to find another point on Z_n where one of the w_k's vanishes. The predictor step consists of linearizing $\mathbf{f}$ at $\bar{\mathbf{z}}$ and then computing the nearest points on the tangent space that intersect with the one of the hyperplanes $w_k = 0$. Figure 4 illustrates the linearization for the quadrature with two weights. The k-th node is then eliminated and the point $\widetilde{\mathbf{z}}$ is mapped to Z_{n-1} using algorithm 3.2. The predictor-corrector method is restarted to eliminate further nodes. This approach guarantees that all nodes remain in the domain and all weights are non-negative.

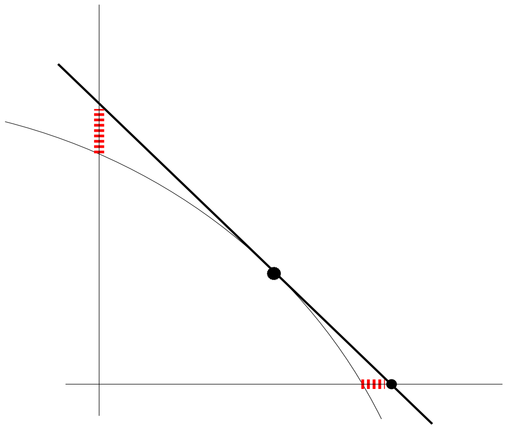

Figure 3.4. Two-dimensional illustration of the predictor step. Solution manifold is linearized by zeroing out one weight at a time.

We now describe the predictor step more detail. To obtain the tangent space of Z_n at $\bar{\mathbf{z}}$ consider the linearization of the function $\mathbf{f}$ at $\bar{\mathbf{z}}$

$$\mathbf{f}(\bar{\mathbf{z}} + \Delta\mathbf{z}) = \mathbf{f}(\bar{\mathbf{z}}) + J\Delta\mathbf{z} + \mathcal{O}(|\Delta\mathbf{z}|^2), slightly$$

where $J = D\mathbf{f}(\bar{\mathbf{z}})$. Since the first term on the right hand side vanishes, the tangent space is defined by $\mathbf{z} = \bar{\mathbf{z}} + \Delta\mathbf{z}$, where $\Delta\mathbf{z} \in (J)$. Since the matrix J is underdetermined, this nullspace is nontrivial. An orthonormal basis can be found with the full LQ-factorization of J

$$J = \begin{bmatrix} L, 0 \end{bmatrix} \begin{bmatrix} \widetilde{Q} \\ \hat{Q} \end{bmatrix}, \tag{3.17}$$

where $L \in \mathbb{R}^{M \times M}$ is lower triangular, $\widetilde{Q} \in \mathbb{R}^{M \times N}$ and $\hat{Q} \in \mathbb{R}^{N-M \times N}$. The rows of $\hat{Q}$ form an orthogonal basis of (J).

To eliminate the k-th node, consider the nearest point $\widetilde{\mathbf{z}}$ in the intersection of the tangent space with the hyperplane $w_k = 0$. If $\widetilde{\mathbf{z}} = \bar{\mathbf{z}} + \Delta\mathbf{z}$, then $\Delta\mathbf{z}$ solves the optimization problem

$$\begin{aligned} &\min \frac{1}{2}\left\|\Delta\mathbf{z}\right\|^2 + t\,\mathbf{g}^T\Delta\mathbf{z} \\ &\Delta\mathbf{z} : J\Delta\mathbf{z} = \mathbf{0} \\ &\qquad w_k + \Delta w_k = 0. \end{aligned}$$

Here $\mathbf{g} = \nabla\phi_{\Omega,k}(\bar{\mathbf{z}})$, where $\phi_{\Omega,k}(\cdot)$ is the penalty function that is obtained by excluding the node k in the summation in equation (3.10).

The unknown $\Delta\mathbf{z}$ can be expressed as a combination of the orthogonal basis of (J) obtained from the LQ factorization in (3.17). Thus there is a vector $\Delta\mathbf{y} \in \mathbb{R}^{N-M}$ such that $\Delta\mathbf{z} = \hat{Q}^T\Delta\mathbf{y}$. Substitution of this vector leads to the optimization problem

$$\begin{aligned} &\min \frac{1}{2}\left\|\Delta\mathbf{y}\right\|^2 + t\,\hat{\mathbf{g}}^T\Delta\mathbf{y} \\ &\Delta\mathbf{y} : \mathbf{m}_k^T\Delta\mathbf{y} = -\bar{w}_k. \end{aligned}$$

Here $\mathbf{m}_k$ is the column of $\hat{Q}$ corresponding to the k-th weight, and $\hat{\mathbf{g}}_k = \hat{Q}\mathbf{g}$. It can be seen that the solution $\Delta\mathbf{y}_k$ of the above optimization problem is given by

$$\Delta\mathbf{y}_k = \Delta\mathbf{y}_k^f + t\Delta\mathbf{y}_k^g, \tag{3.18}$$

where

$$\begin{aligned} \Delta\mathbf{y}_k^f &= \mathbf{m}_k\frac{\bar{w}_k}{\left\|\mathbf{m}_k\right\|^2} \\ \Delta\mathbf{y}_k^g &= \left(I - \frac{\mathbf{m}_k\mathbf{m}_k^T}{\left\|\mathbf{m}_k\right\|^2}\right)\hat{\mathbf{g}}, \end{aligned}$$

and thus the predictor is

$$\widetilde{\mathbf{z}}_k = \bar{\mathbf{z}} + \Delta\mathbf{z}_k^f + t\Delta\mathbf{z}_k^g,$$

where $\Delta\mathbf{z}^f = \hat{Q}^T \Delta\mathbf{y}_k^f$ and $\Delta\mathbf{z}^g = \hat{Q}^T \Delta\mathbf{y}_k^g$. The parameter t is selected to ensure maximal distance from the domain boundary. If the domain Ω is a polytope, then this can be achieved with algorithm 3.1.

We compute in a list predictors $\widetilde{\mathbf{z}}_k$ for all weights $k \in \{1, \dots, n\}$. The predictors that satisfy the constraints and are be close to $\bar{\mathbf{z}}$ will be mapped on Z_{n-1}. The next quadrature is the solution that has the greatest distance to the boundary. The node elimination procedure is summarized in algorithm 3.3.

Algorithm 3.3

```
Find a suitable initial cubature rule $\bar{\mathbf{z}}$.
while $N > M$ do
    Setup $J$, and compute the LQ-factorization.
    for k=1:n do
        Compute $\Delta\mathbf{z}_k^f$ and $\Delta\mathbf{z}_k^g$ in (3.3).
        Compute $t$ using using algorithm 3.1.
        Set $\Delta\mathbf{z}_k = \Delta\mathbf{z}_k^f + t\Delta\mathbf{z}_k^g$ and $\widetilde{\mathbf{z}}_k = \bar{\mathbf{z}}_k + \Delta\mathbf{z}_k$.
    end for
    Sort the $\Delta\mathbf{z}_k$'s that satisfy the constraints such that
        $\|\Delta\mathbf{z}_1\| \leq \|\Delta\mathbf{z}_2\| \leq \dots$.

    for k=1:K do
        Eliminate the k-th node from $\widetilde{\mathbf{z}}_k$.
        Use algorithm 3.2 to map $\widetilde{\mathbf{z}}_k$ to $\bar{\mathbf{z}}_k \in Z_{n-1}$
    end for
    Stop if no $\bar{\mathbf{z}}_k \in Z_{n-1}$ could be found.
    $\bar{\mathbf{z}} \leftarrow \bar{\mathbf{z}}_k$, where $\bar{\mathbf{z}}_k$'s nodes have the greatest distance to the boundary.
    $n \leftarrow n - 1$
end while
```

The main cost in this algorithm is to set up Jacobians and compute their QR factorizations.

3.4. Computational Cost

We conclude this section with some implementation details and a comparison of the computational cost with the algorithm in [50], referred here to as standard node elimination.

Most of the CPU time in our algorithm is spent in the corrector, in particular, the LQ factorization of the Jacobian J which is needed in each iteration to obtain the vectors $\Delta\mathbf{z}^g$ and $\Delta\mathbf{z}^f$ in equation (3.13). The cost of the factorization is $O(MN^2)$, while the additional matrix-vector products in (5.4) to calculate $\Delta\mathbf{z}^f$ and $\Delta\mathbf{z}^g$ are of lower order. Note that the values of n, N, M in our examples follow from the data displayed in the tables.

Standard node elimination only involves the vector $\Delta\mathbf{z}^f$, but the additional cost for $\Delta\mathbf{z}^g$ is negligible. Likewise, the determination of the parameter t in algorithm 3.1 involves only lower order operations, and is also negligible.

The leading memory cost is storing the Jacobian J which is $O(NM)$. Since standard linear algebra packages overwrite this space, no additional storage is needed for the L and Q matrices.

The predictor step involves another LQ factorization, as well as matrix-vector products described in section 3.3. To benefit from memory locality, they are expressed as a single matrix-matrix product, which is $O(N(N-M)n)$. Our predictor step is more expensive than standard node elimination, where nodes are selected for elimination by computing the Euclidean norms of the columns of the Jacobian matrix. However, the predictor is executed less frequently than the iterations of the corrector step. Moreover, the approach based on linearization provides a better initial guess, leading to fewer iterations in the corrector.

An additional factor that influences the overall CPU time is the number of predictors K that are mapped on Z_n in algorithm 3.3. In standard node elimination $K = 1$, but in our experience the efficiency of the final cubature rule is improved if K has a larger value in the final stages of node elimination. We set with $K = 3$ in our experiments. Alternatively, one could set $K = 3$ at the initial stages and end with $K = 10$. Since the predictor is dominant, the cost of one elimination step is roughly K times the standard node elimination step.

3.5. Strategies for the Initial Cubatures

So far, the discussion assumed that an initial cubature rule is known that may be sub-optimal (i.e., it has more nodes than necessary), but has the desired degree. This section describes how such an initial cubature can be obtained. Here the goal is to keep the number of nodes low such that fewer elimination steps have to be taken and the cost of the linear algebra in the initial stages of the algorithm is reduced.

The scheme is based on the fact that cubature rules for Cartesian product domains $\Omega = \Omega_1 \times \Omega_2$ can be obtained by forming tensor products of cubature rules of Ω_1 and Ω_2. Specifically, if $\{\mathbf{x}_n, w_n\}$ and $\{\mathbf{y}_m, v_m\}$ are cubature rules for Ω_1 and Ω_2 that are exact for $\mathbb{P}_p^{d_1}$ and $\mathbb{P}_p^{d_2}$, respectively, then $\{(\mathbf{x}_n, \mathbf{y}_m), w_n v_m\}$ is a cubature rule that is exact for $\mathbb{P}_p^{d_1} \times \mathbb{P}_p^{d_2}$. Since this polynomial space is larger than $\mathbb{P}_p^{d_1+d_2}$, the node elimination can be performed with the tensor product rule as an initial guess.

3.5.1. C_d

We start the discussion with the case that Ω is the d-dimensional unit cube

$$C_d = \big\{(x_1, \dots, x_d) : 0 \le x_i \le 1, i = 0, \dots, d\big\}.$$

If one were to use tensor products of degree-p Gauss Legendre rules as the initial quadrature, one would obtain $(\lfloor p/2 + 1 \rfloor)^d$ nodes. The rapid growth of this number makes this strategy unfeasible even for moderate values of p and d.

Instead, our implementation uses an iterative scheme to obtain the initial guess for the domains of interest, thereby significantly decreasing the initial number of nodes. We build the rule incrementally, by starting with $C_1 \times C_1$, and running node elimination. The resulting rule is then tensored with the C_1-rule to obtain an initial guess for C_3. The scheme is repeated until the desired dimension is reached. With this approach the number of nodes of the initial guess in the last step is greatly reduced over a d-fold tensor product of rules for C_1.

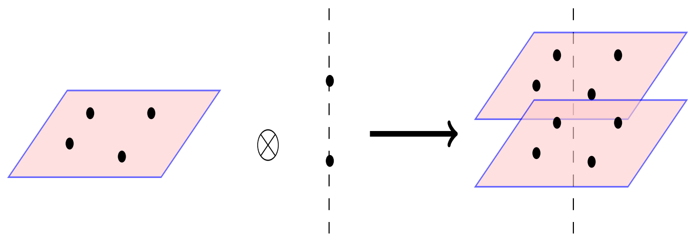

Figure 3.5. Tensor product of 2-D unit square and interval.

3.5.2. S_d

We now turn to the d-dimensional simplex

$$S_d = \{(x_1, \ldots, x_d) : 0 \le x_d \le \cdots \le x_1 \le 1\}$$

and to Cartesian products of cubes and simplexes.

The well known Duffy transformation can be used to transform a cube into a simplex. For the $d+1$-dimensional simplex it can be defined recursively as follows

$$\begin{aligned} x_1 &= \xi, \\ x_2 &= \xi\eta_1, \\ x_3 &= \xi\eta_2, \\ &\vdots \\ x_{d+1} &= \xi\eta_d \end{aligned} \tag{3.19}$$

Here $\xi \in [0,1] = C_1$ and $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_d) \in T_d$. Thus (3.19) transforms $C_1 \times T_d$ into T_{d+1} with Jacobian $J = \xi^d$. Now T_d can be transformed with another Duffy transform and repeating this leads to a transformation from C_{d+1} to T_{d+1}. However, for the construction of initial quadrature rules it is more convenient to work with one transformation at a time.

For instance, for the triangle (i.e. $d = 1$), we use Gauss-Jacobi rules for the integral over the (ξ, η) variables. Specifically, if

$$\int_0^1 f(\eta) d\eta \approx \sum_{\ell=1}^{q} f(y_\ell) v_\ell \quad \text{and} \quad \int_0^1 f(\xi)\xi d\xi \approx \sum_{k=1}^{q} f(x_k) w_\ell$$

are the quadrature rules of degree $p = 2q - 1$ for weight function $w(\eta) = 1$ and $w(\xi) = \xi$, respectively, then an integral over T_2 can be approximated as follows

$$\int_{T_2} \varphi(\mathbf{x}) d\mathbf{x} = \int_0^1 \int_0^1 \varphi(\xi, \xi\eta)\xi d\eta d\xi \approx \sum_{\substack{k=1 \\ \ell=1}}^{p} \varphi(x_k, x_k y_\ell) w_k v_\ell. \tag{3.20}$$

Note that the Jacobian ξ determines the choice of rule and affects the nodes x_k and weights w_k. If $\varphi \in \mathbb{P}_p^2$, then $(\xi, \eta) \mapsto \varphi(\xi, \xi\eta)$ is a polynomial in $\mathbb{P}_p^1 \times \mathbb{P}_p^1$ and thus the quadrature rule (3.20) is exact. This tensor product rule can now be used as the initial rule in the node elimination procedure.

Similar to the cube, the $d+1$-dimensional simplex rule is build by recursion using previously generated rules. Thus, if $\{\mathbf{x}_k, w_k\}$ is a degree-p Gauss-Jacobi rule for $\int_0^1 f(\xi)\xi^d d\xi$ and $\{\mathbf{y}_l, v_l\}$ is a degree-p rule for T_d, then

$$\int_{T_{d+1}} \varphi(\mathbf{x}) d\mathbf{x} = \int_0^1 \int_{T_d} \varphi(\xi, \xi\boldsymbol{\eta})\xi^d d\eta d\xi \approx \sum_{k,l} \varphi\big(x_k, x_k \mathbf{y}_l\big) w_k v_l$$

is a degree-p rule for T_{d+1}.

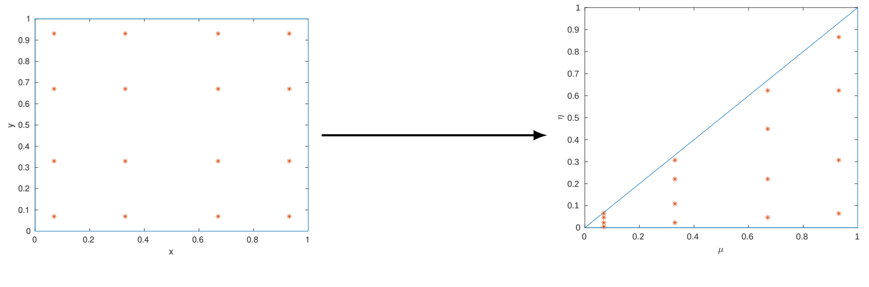

Figure 3.6. Duffy transformation from C_2 to T_2.

3.5.3. Tensor Domains

For tensor products of cubes and simplexes the analogous procedures can be applied, where some care must be a taken to use the appropriate Gauss-Jacobi rule to compensate for the Jacobian of the Duffy transformation.

$$\begin{tikzcd}
\mathbf{I} \arrow[r, bend left, "\otimes I"] & C_2 \arrow[r, "\text{NE}"] \arrow[d, dash, "\text{Duffy}"] & \mathbf{C_2} \arrow[r, bend left, "\otimes I"] & C_3 \arrow[r, "\text{NE}"] & \mathbf{C_3} \arrow[r, bend left, "\otimes I"] & C_4 \arrow[r, "\text{NE}"] & \mathbf{C_4} \\
 & T_2 \arrow[r, "\text{NE}"] & \mathbf{T_2} \arrow[r, bend left, "\otimes I"] \arrow[rrrdd, "\otimes T_2"'] & T_2C_1 \arrow[r, "\text{NE}"] \arrow[d, dash, "\text{Duffy}"] & \mathbf{T_2C_1} \arrow[r, bend left, "\otimes I"] & T_2C_2 \arrow[r, "\text{NE}"] & \mathbf{T_2C_2} \\
 & & & T_3 \arrow[r, "\text{NE}"] & \mathbf{T_3} \arrow[r, bend left, "\otimes I"] & T_3C_1 \arrow[r, "\text{NE}"] & \mathbf{T_3C_1} \\
 & & & & & T_2T_2 \arrow[r, "\text{NE}"] & T_2T_2
\end{tikzcd}$$

Figure 3.7. Construction scheme for quadrature rules for the four dimensional polytopes C_4, $T_2 \times C_2$, $T_3 \times C_1$ $T_2 \times T_2$

Figure 3.7 illustrates the procedure for various polytopes in four dimensions. It begins with Gaussian cubature on the interval (shown in blue). Then the tensor product of Gaussian quadrature on $I \otimes I$ is applied, which is an initial guess for C_2. Subsequently, depending on the domain of interest, we either proceed by running the Node Elimination algorithm, or applying the Duffy transformation to obtain T_2. The diagram below describes how to derive initial guesses for C_4, $T_2 \times C_2$, $T_3 \times C_1$ and $T_2 \times T_2$, before performing the final node elimination step, which is shown in red. The four figures are presented for four and six-dimensional cubes and simplexes to illustrate the improvements.

The number of nodes using the tensor product rule corresponds to $\lceil \frac{deg+1}{2} \rceil$ nodes. It is apparent that recursive initial guess leads to significant improvement for all degrees and

degree	5	7	9	11	13	15
t_{init}	81	256	625	1296	2401	4096
r_{init}	45	136	300	600	1064	1720

Table 3.1. Number of initial nodes using tensor product vs recursive procedure for C_4.

degree	5	7	9	11	13	15
t_{init}	81	256	625	1296	2401	4096
r_{init}	42	128	285	558	1001	1656

Table 3.2. Number of initial nodes using tensor product vs recursive procedure for T_4.

degree	5	6	7	8	9	10
t_{init}	729	4096	4096	15625	15625	46656
r_{init}	159	324	644	1095	1900	3072

Table 3.3. Number of initial nodes using tensor product vs recursive procedure for C_6.

degree	5	6	7	8	9	10
t_{init}	729	4096	4096	15625	15625	46656
r_{init}	138	340	568	1200	1745	3222

Table 3.4. Number of initial nodes using tensor product vs recursive procedure for T_6.

domains. For brevity, only odd degrees are shown for $dim = 4$. We see that improvements are greater for higher degrees. In six dimensions, results are even more exaggerated. Tensor products grow rapidly and exhibit the curse of dimensionality. Recursive strategy alleviates the rapidly growing problem. Efficient initial guesses is a direct consequence of efficient rules produced by the Node Elimination in lower dimensions.

3.5.4. Pyramid

For a pyramid, a cubature for C_3 is computed first. Then the Duffy transformation and appropriate Gauss-Jacobi rule are used to map C_3 to P_3. In particular, a transformation from C_3 to P_3 is

$$\begin{aligned} x_{1_p} &= x_{1_c} \\ x_{2_p} &= x_{1_c} x_{2_c} \\ x_{3_p} &= x_{1_c} x_{3_c}. \end{aligned} \tag{3.21}$$

Chapter 4

Preconditioning

A quadrature rule q is of degree p if it integrates $\mathbb{P}_p^d$ exactly. This results in solving the linear system (3.2) at each iteration of the Least Squares Newton. In this chapter, we will show that choosing $\mathbb{P}_p^d$ to be orthogonal is crucial for the conditioning of (3.2). Compared to the standard monomial basis, the conditioning number is reduced considerably, which allows to solve the problem with higher degrees and dimensions. Therefore, we derive and apply the orthogonal bases for all polytopes of interest and refer to it as preconditioning.

4.1. Functions, Derivatives, Integrals

Recall that in the corrector step, a polynomial basis is used to solve the nonlinear system

$$\Phi(x) = \begin{bmatrix} \phi_1(x) \\ \vdots \\ \phi_M(x) \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} \int_\Omega \phi_1(x) w(x)\, dx \\ \vdots \\ \int_\Omega \phi_M(x) w(x)\, dx \end{bmatrix}, \tag{4.1}$$

$$\mathbf{f}(\mathbf{x}, \mathbf{w}) = \mathbf{\Phi}(\mathbf{x})\mathbf{w} - \mathbf{b} = \mathbf{0}, \tag{4.2}$$

where $\mathbf{\Phi}(\mathbf{x})$ and $\mathbf{b}$ are polynomial basis functions and their integrals, respectively.

At each iteration of the Penalized Newton, we solve

$$\begin{aligned} \Delta\mathbf{z} &= -J^T \left(JJ^T\right)^{-1} \mathbf{f} - t\left(I - J^T \left(JJ^T\right)^{-1} J\right)\mathbf{g}, \\ &= \Delta\mathbf{z}^f + t\Delta\mathbf{z}^g, \end{aligned} \tag{4.3}$$

where the Jacobian is defined according to (3.3). The derivatives of the basis functions in the Jacobian can be computed either analytically or numerically using finite differences, which is what we implemented in our codes. Another alternative is to use automatic differentiation package. The integrals of $\mathbf{b}$ are computed either analytically or numerically by subdividing the domain into simplexes(tetrahedra), in case if computing integrals analytically is problematic. If orthogonal basis is chosen, computing integrals is trivial.

4.2. Multi-Indexes

In d-dimensional space, the non-symmetric basis satisfies

$$\mathbb{P}_p^d = \operatorname{span}\left\{x^\alpha : \alpha_1 + \cdots + \alpha_d \leq p\right\}.$$

We define to a set of all multi-indexes for $\mathbb{P}_p^d$

$$S_p^d = \left\{\alpha_1, \ldots, \alpha_d\right\}, \alpha_1 + \cdots + \alpha_d \leq p.$$

In order to compute monomial or orthogonal basis, a corresponding multi-index must be applied to each polynomial in the set. Multi-indexes can be computed on the fly for each dimension, or more generally, by applying nested recursion. The algorithm is a variation of recurrence algorithm in [49].

```
$S_p^1 = (\mathbf{0}, \mathbf{1}, ..., \mathbf{p})$, $p = (\mathbf{0}, \mathbf{1}, ..., \mathbf{deg})$
for d=2:dim do
    for p=0:deg do
        $S_p^d = \emptyset$
        for $\hat{\alpha} \in S_{deg-p}^{d-1}$ do
            $S_p^d$.append($[p, \hat{\alpha}]$)
        end for
    end for
end for
```

The algorithm starts with defining one-dimensional sequences of degree p, which corresponds to a sequence $(\mathbf{0}, \mathbf{1}, ..., \mathbf{p})$. Elements of the sequence are scalars. Multi-indexes in two and higher dimensions are constructed by reusing sequences of lower degree and dimension. Each element is a d-dimensional multi-index. Essentially, S_p^d is a matrix in $\mathbb{R}^{M\times d}$, such that

$$M = Dim(\mathbb{P}_p^d) = \frac{(d+p)!}{d!p!}.$$

For example, if $d = 2$, and $p = 3$, we get

$$M = Dim(\mathbb{P}_2^3) = \frac{(2+3)!}{2!3!} = 10,$$

and the corresponding matrix of multi-indexes is

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \\ 0 & 3 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 2 & 0 \\ 2 & 1 \\ 3 & 0 \end{bmatrix}.$$

Consequently, the monomial basis of S_p^d is easily computed

$$B(x,y) = \begin{bmatrix} x^0y^0 \\ x^0y^1 \\ x^0y^2 \\ x^0y^3 \\ x^1y^0 \\ x^1y^1 \\ x^1y^2 \\ x^2y^0 \\ x^2y^1 \\ x^3y^0 \end{bmatrix}.$$

4.3. Monomial Basis

Monomial basis is convenient in a sense that the basis itself and its derivatives are independent of the domain. Integrals of monomials, on the other hand, do depend on the domain but can be easily integrated by means of triangulation. For our domains of interest, integrals of monomials are computed analytically and do not require triangulation.

However, the monomial basis is known to suffer from poor conditioning[24], which would lead to instability in (4.3). The higher the degree and dimension of the problem, the worse the conditioning becomes. Therefore, orthogonal polynomials are used to precondition the linear system. They are heavily dependent on the domain and are known only for a small subset of polytopes. Finding a universal preconditioner is the most challenging aspect of applying the algorithm to an arbitrary domain. In the next section, analytical derivation of orthogonal polynomials is described.

4.4. Orthogonal Basis

In two and higher dimensions, the analytical construction of orthogonal polynomials over domains such as triangles, tetrahedra, hexagons, spheres, and other polyopes is a widely studied research area [36]. Orthogonal polynomials are frequently used for preconditioning a linear system of equations[26]. Polynomials P_n and P_m(of degree n and m) are said to be orthogonal with respect some weight function $w(x)$ over a domain Ω if

$$\int_{\Omega} P_n(x)P_m(x)w(x)dx = 0 \;\; if \;\; n \neq m.$$

If $\Omega = [-1, 1]$, Legendre polynomials $L_p(x)$ are orthogonal with respect to weight function $w(x) = 1$, whereas Jacobi polynomials $J^{\alpha,\beta}(x)$ are orthogonal for $w(x) = (1-x)^{\alpha}(1+x)^{\beta}$. In fact, Legendre polynomials are a special case of Jacobi polynomials: $L_p(x) = J_p^{0,0}(x)$. Appendix A describes these polynomials in more detail.

To solve (4.3), the orthogonal basis, its derivatives, and integrals over Ω must be computed. The derivatives are obtained either by analytical or numerical differentiation. As these polynomials become more complicated in higher dimensions, numerical differentiation is used. Due to orthogonality, integration is trivial. Scaling the only nonzero integrand results in

$$\mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

We now present construction of orthogonal bases over high-dimensional polytopes w.r.t. $w(x) = 1$.

4.4.1. Cube

Since Legendre polynomials are orthogonal over I, their tensor product

$$L_{\boldsymbol{\alpha}}(\mathbf{x}) = L_{\alpha_1}(x_1)L_{\alpha_2}(x_2) \; ... \; L_{\alpha_d}(x_d), \;\; \alpha_i \leq d$$

is orthogonal over C_d. However, the set of resulting multi-indexes includes redundant functions. Instead, it is sufficient to apply S_p^d

$$\Phi_{\boldsymbol{\alpha}}^{C_d}(\mathbf{x}) = L_{\alpha_1}(x_1) \;...\; L_{\alpha_d}(x_d), \;\boldsymbol{\alpha} \in S_p^d. \tag{4.4}$$

4.4.2. Simplex

Orthogonal polynomials over arbitrary-dimensional simplex are more involved and require a mix of Jacobi and Legendre polynomials. We will use induction to construct orthogonal polynomials over an arbitrary simplex. For the two and three-dimensional cases, one can refer to [34], and [42], respectively. We will show derivation in two dimensions first, since proof in arbitrary dimension follows a similar pattern.

Suppose $P_m = J_m^{\alpha,\beta}$. Then the orthogonal basis over the unit triangle is given by

$$\Phi_{m,n}(x_1, x_2) = P_m^{0,0}(\frac{2x_2 - x_1}{x_1})x_1^m P_n^{2m+1,0}(1 - 2x_1). \tag{4.5}$$

Proof: By definition of orthogonal polynomials, it is sufficient to show that

$$\int_{T_2} \Phi_{m,n}(x_1, x_2)\Phi_{m',n'}(x_1, x_2)dx_2dx_1 =$$
$$\int_0^1 \int_0^{x_1} \Phi_{m,n}(x_1, x_2)\Phi_{m',n'}(x_1, x_2)dx_2dx_1 =$$
$$c\;\delta_{m,m'}\delta_{n,n'}, c \neq 0.$$

In other words, orthogonality implies $\int_{T_2} \Phi_{m,n}(\mathbf{x})\Phi_{m',n'}(\mathbf{x})d\mathbf{x} = 0\; if\; m \neq m'\; or\; n \neq n'$. From (4.5),

$$\int_{T_2} \Phi_{m,n}\Phi_{m',n'} =$$
$$\int_0^1 \int_0^{x_1} P_m^{0,0}(\frac{2x_2 - x_1}{x_1})\; x_1^m\; P_n^{2m+1,0}(1 - 2x_1)P_{m'}^{0,0}(\frac{2x_2 - x_1}{x_1})\; x_1^{m'}\; P_{n'}^{2m'+1,0}(1 - 2x_1)dx_2dx_1 =$$
$$\int_0^1 \int_0^{x_1} P_m^{0,0}(\frac{2x_2 - x_1}{x_1})P_{m'}^{0,0}(\frac{2x_2 - x_1}{x_1})\; x_1^{m+m'}\; P_n^{2m+1,0}(1 - 2x_1)\;\; P_{n'}^{2m'+1,0}(1 - 2x_1)dx_2dx_1 =$$

Set $u(x_2) = \frac{2x_2 - x_1}{x_1} \rightarrow\ du = \frac{2}{x_1} dx_2,\ u(x_1) = 1,\ u(0) = -1$

$$2\int_0^1 \Big(\int_{-1}^1 P_m^{0,0}(u) P_{m'}^{0,0}(u)\ du\Big)\ P_n^{2m+1,0}(1-2x_1)\ P_{n'}^{2m'+1,0}(1-2x_1)\ x_1^{m+m'+1} dx_1 =$$
$$2\int_0^1 \delta_{m,m'}\ P_n^{2m+1,0}(1-2x_1)\ P_{n'}^{2m'+1,0}(1-2x_1)\ x_1^{m+m'+1} dx_1 =$$

Set $v(x_1) = 1 - 2x_1 \rightarrow\ dv = -2dx_1,\ v(0) = 1,\ v(1) = -1$, and $x_1 = \frac{1-v}{2}$

$$\delta_{m,m'} \int_{-1}^1\ P_n^{2m+1,0}(v)\ P_{n'}^{2m'+1,0}(v)\ x_1^{m+m'+1} \Big(\frac{1-v}{2}\Big)^{m+m'+1} dv =$$
$$\frac{1}{2^{m+m'+1}} \delta_{m,m'} \int_{-1}^1\ P_n^{2m+1,0}(v)\ P_{n'}^{2m'+1,0}(v)\ (1-v)^{m+m'+1} dv =$$
$$\frac{1}{2^{m+m'+1}} \delta_{m,m'} \delta_{n,n'},$$

since $\int_{-1}^1 P_m^{2m+1,0} P_n^{2m+1,0} (1-v)^{2m+1} = 0$ by definition of Jacobi polynomials.

Scaling $\Phi_{m,n}(\mathbf{x})$ by 2^{2m+1} yields orthonormal basis. We now present our proof in arbitrary dimension.

If $\Phi_{\boldsymbol{\alpha}}^d(\mathbf{x})$ constitutes a d-dimensional orthogonal polynomial basis, then the d+1-dimensional orthogonal basis is obtained by recurrence

$$\Phi_{\boldsymbol{\alpha}_{d+1}}^{d+1}(\mathbf{x}_{d+1}) = \Phi_{\boldsymbol{\alpha},n}^{d+1}(\xi, \mathbf{x}) = c\Phi_{\boldsymbol{\alpha}}^d(\frac{\mathbf{x}}{\xi}) P_n^{2|\boldsymbol{\alpha}|+d,0}(1-2\xi)\xi^{|\boldsymbol{\alpha}|}, \tag{4.6}$$

where $|\boldsymbol{\alpha}| = \sum_i \alpha_i$, and $d = 2$ is the base case.

Proof: First, consider integral of an arbitrary function over T_{d+1}

$$\int_{T_{d+1}} f(\xi, \mathbf{x}) d(\xi, \mathbf{x}) dT_{d+1} = \int_0^1 \int_0^\xi \int_0^{x_1}\ ...\ \int_0^{x_{d-1}} f(\xi,\ x_1,\ ...\ ,\ x_d)\ dx_d\ ...\ dx_1 d\xi =$$

Now substitute $x_j = \xi x'_j \to det(J) = \xi^d$

$$\int_0^1 \int_0^1 \int_0^{x'_1} \dots \int_0^{x'_{d-1}} f(\xi, \xi x'_1, \dots, \xi x'_d)\; dx'_n \dots\; dx'_1 \xi^d d\xi = \int_0^1 \int_{T_d} f(\xi, \xi\mathbf{x}) d\mathbf{x} \xi^d d\xi.$$

We will use this relation and strategy that resembles two-dimensional case to show orthogonality.

Second, set $\Phi^{d+1}_{\boldsymbol{\alpha},m}(\xi, \mathbf{x}) = \varphi_m(\xi)\Phi^d_{\boldsymbol{\alpha}}(\frac{\mathbf{x}}{\xi})\xi^{\boldsymbol{\alpha}}$. It remains to find φ_m that leads to orthogonality.

$$\int_{T_{d+1}} \Phi^{d+1}_{\boldsymbol{\alpha},m}(\xi, \mathbf{x})\Phi^{d+1}_{\boldsymbol{\alpha}',m'}(\xi, \mathbf{x}) dT_{d+1} = \tag{4.7}$$

$$\int_{T_{d+1}} \varphi_m(\xi)\Phi^d_{\boldsymbol{\alpha}}(\frac{\mathbf{x}}{\xi})\xi^{\boldsymbol{\alpha}}\varphi_{m'}(\xi)\Phi^d_{\boldsymbol{\alpha}'}(\frac{\mathbf{x}}{\xi})\xi^{\boldsymbol{\alpha}'} dT_{d+1} = \tag{4.8}$$

$$\int_0^1 \left(\int_{T_d} \Phi^d_{\boldsymbol{\alpha}}(\mathbf{x})\Phi^d_{\boldsymbol{\alpha}'}(\mathbf{x}) d\mathbf{x}\right)\varphi_m(\xi)\varphi_{m'}(\xi)\xi^{|\boldsymbol{\alpha}+\boldsymbol{\alpha}'|+d} d\xi = \tag{4.9}$$

By assumption, $\int_{T_d} \Phi^d_{\boldsymbol{\alpha}}(\mathbf{x})\Phi^d_{\boldsymbol{\alpha}'}(\mathbf{x}) d\mathbf{x} = \delta_{\boldsymbol{\alpha},\boldsymbol{\alpha}'}$. If $\boldsymbol{\alpha} = \boldsymbol{\alpha}'$, the integral becomes

$$\int_0^1 \varphi_m(\xi)\varphi_{m'}(\xi)\xi^{2|\boldsymbol{\alpha}|+d} d\xi = \tag{4.10}$$

$$\frac{1}{2^{2|\boldsymbol{\alpha}|+d+1}} \int_{-1}^1 \varphi_m(\frac{1-v}{2})\varphi_{m'}(\frac{1-v}{2})(1-v)^{2|\boldsymbol{\alpha}|+d} dv, \tag{4.11}$$

where $v(\xi) = 1 - 2\xi,\; v(0) = 1,\; v(1) = -1, dv = -2d\xi$.

A suitable choice for $\varphi_m(\frac{1-v}{2})$ is a Jacobi polynomial $P_m^{2|\boldsymbol{\alpha}|+d}(v)$, which in terms of $z = \frac{1-v}{2}$ and $v = 1 - 2z$ leads to

$$\varphi_m(v) = \varphi_m(1 - 2z) = P_m^{2|\boldsymbol{\alpha}|+d}(1 - 2z).$$

Then 4.10 reduces to $c\delta_m * \delta_{m'}$ and 4.7 reduces to $c\delta_m\delta_{m'}\delta_{\boldsymbol{\alpha}}\delta_{\boldsymbol{\alpha}'}$, which was to be shown.

Setting c to $\frac{1}{2^{\alpha+\frac{n+1}{2}}}$ in (4.6) makes orthogonal basis orthonormal.

4.4.3. $C_i \times T_j$

Once orthogonal basis functions for n-dimensional cube and n-dimensional simplex are obtained, construction of orthogonal basis over $C_{d_1} \times T_{d_2}$ of dimension $d_1 + d_2$ is straightforward. The resulting basis is a product of the two bases

$$\Phi_{\alpha}^{C_{d_1} \times S_{d_2}}(\mathbf{x}) = \Phi_{\alpha_1}^{C_{d_1}}(\mathbf{x}_1)\Phi_{\alpha_2}^{S_{d_2}}(\mathbf{x}_2), \tag{4.12}$$

where $\alpha = (\alpha_1, \alpha_2),\ \mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)$.

4.4.4. $S_i \times S_j$

Similarly to $C_i \times S_j$, functions for $S_{d_1} \times S_{d_2}$ of dimension $d_1 + d_2$ are based on basis functions of a simplex raised to $d_1 + d_2$ dimensional multi-indexes

$$\Phi_{\alpha}^{S_{d_1} \times S_{d_2}}(\mathbf{x}) = \Phi_{\alpha_1}^{S_{d_1}}(\mathbf{x}_1)\Phi_{\alpha_2}^{S_{d_2}}(\mathbf{x}_2). \tag{4.13}$$

4.4.5. Pyramid

An orthogonal basis for a bi-unit pyramid is presented in [9], and related work [11]. Nevertheless, we derive orthogonal basis over unit, right-angled pyramid so that it can potentially be extended to arbitrary-dimensional pyramid in the future. Currently, 3-dimensional case is sufficient. For a pyramid, our orthogonal basis is

$$\Phi_{m,n,k}(x, y, z) = P_m^{0,0}(\frac{x}{z})P_n^{0,0}(\frac{y}{z})P_k^{2,0}(z). \tag{4.14}$$

Proof: In a similar fashion to simplex, consider a transformation

$$\int_{P3} f(\hat{\mathbf{x}}) d\hat{\mathbf{x}} \equiv \int_0^1 \int_0^z \int_0^z f(\hat{x}, \hat{y}, \hat{z}) d\hat{x} d\hat{y} d\hat{z} =$$
$$\int_0^1 \int_0^1 \int_0^1 f(xz, yz, z) z^2 dx dy dz.$$

Here, $\hat{x} = xz$, $\hat{y} = yz$, and $\hat{z} = z$ is a Duffy transformation that maps unit cube to the unit pyramid. The term z^2 is determinant of the Jacobian. From 4.14, we get

$$\int_{P3} \Phi_{m,n,k}(\hat{\mathbf{x}}) \Phi_{m',n',k'}(\hat{\mathbf{x}}) d\hat{\mathbf{x}} =$$
$$\int_0^1 \int_0^z \int_0^z \Phi_{m,n,k}(\hat{x}, \hat{y}, \hat{z}) \Phi_{m',n',k'}(\hat{\mathbf{x}}) d\hat{x} d\hat{y} d\hat{z} =$$
$$\int_0^1 \int_0^1 \int_0^1 \Phi_{m,n,k}(xz, yz, z) \Phi_{m',n',k'}(xz, yz, z) z^2 dx dy dz =$$
$$\int_0^1 \int_0^1 \int_0^1 P_m^{0,0}(x) P_n^{0,0}(y) P_k^{2,0}(z) P_{m'}^{0,0}(x) P_{n'}^{0,0}(y) P_{k'}^{2,0}(z) z^2 dx dy dz =$$
$$\int_0^1 P_k^{2,0}(z) P_{k'}^{2,0}(z) z^2 dz \int_0^1 P_n^{0,0}(y) P_{n'}^{0,0}(y) dy \int_0^1 P_m^{0,0}(x) P_{m'}^{0,0}(x) dx =$$
$$\delta_{k,k'} \delta_{n,n'} \delta_{m,m'}.$$

Chapter 5

Implementation

gen-quad is a package implemented in C++11 that computes cubature rules over polytopes[44]. At the time of this writing, it supports all the domains discussed so far(cubes, simplexes, tensor product domains, pyramid). In the future, we intend to extend its functionality to a wider family of polytopes. The design targets

- Extensibility. Implementation relies on interface inheritance so that one routine is applied to cubature rules of different shapes. It also simultaneously emphasizes the design of a universal algorithm with minimal parameter tuning for each separate case. For some cases, however, special treatment might be needed. If we were to implement the algorithm for a sphere rather than a polytope, optimization related to linear inequality constraints would require an alternative.

- Reliability. Quadrature rules of any degree and dimension are expected to have a sufficiently high efficiency index. For instance, we prefer to produce quadrature rules of efficiency index of 90% for all domains, rather than 80% for some and 100% for others.

- Performance. The computational cost is dominated by operations on large dense matrices in the predictor and corrector step. Therefore, efficient linear algebra routines are essential. Another computationally demanding task is computation of orthogonal basis functions, since they are evaluated frequently to obtain nonlinear functions and their Jacobians. All expensive components are parallelized using OpenMP.

From the user viewpoint, one has to specify the degree of precision, dimension, type of the domain, and optional search width parameter, which corresponds to K in the algorithm 3.3. The optional parameter specifies how many successful cubatures are considered before selecting the one that is furthest from the boundary. gen-quad also offers configuration at compile-time, such as verbose and debug modes. Detailed description and source code is publicly available on GitHub: https://github.com/arkslobodkins/gen-quad.

5.1. Structure

5.1.1. Polymorphism

One of the primary goals is the design of a node elimination algorithm that would generalize well to a multitude of domains. It has already been demonstrated that cubes, simplexes, and tensor product domains are implemented for an arbitrary degree and dimension. Special treatment of each domain type individually is minimized, but is necessary for implementing orthogonal bases. Additional considerations might arise for irregular polytopes or domains other than polytopes, such as spheres, cylinders, etc. To that end, polymorphism based on abstract classes is used. The structure of domain types is shown in figure 5.1.

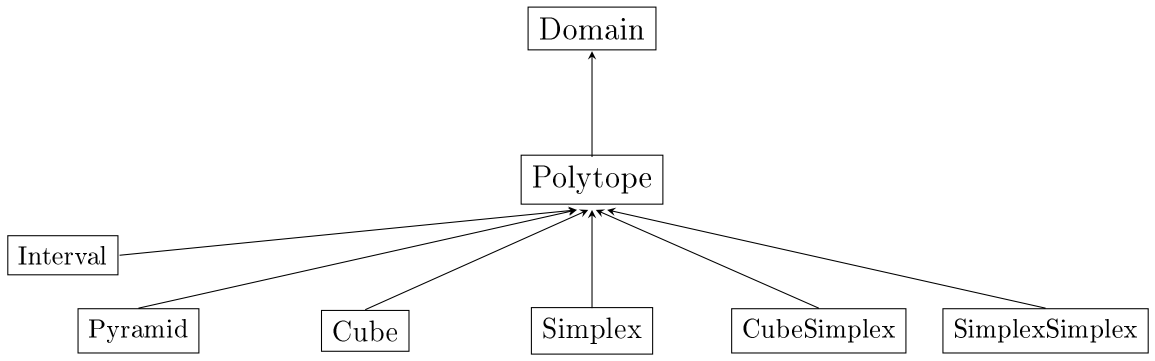

Figure 5.1. Class hierarchy for domains. Domain and Polytope are abstract classes.

Roughly, Domain is an abstract class that contains two pure abstract functions(figure 5.2.).

Polytope is a special abstract class in a sense that it contains linear inequality constraints, by means of which it implements **in_domain** method. All polytopes that are derived from

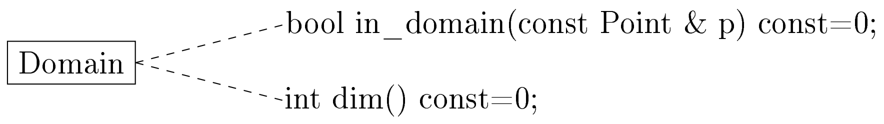

Figure 5.2. All domains are required to have a dimension and implementation of **in_domain**.

it are expected to initialize it with constraints specific to that class. Constructor accepts Matrix and Array parameters by value to benefit from move semantics[32].

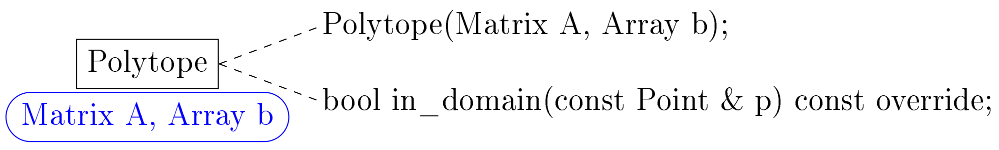
Polytope
Matrix A, Array b
Polytope(Matrix A, Array b);
bool in_domain(const Point & p) const override;

Figure 5.3. Polytope uses constraints to implement **in_domain** for all polytopes. Data members are presented in blue.

Domains that are derived from Polytope might contain additional data. For example, Cube only has one dimension parameter, whereas tensor product domains have two dimension parameters dim_1 and dim_2. In addition, most polytopes optionally implement the **reinit** function, which changes the dimension and hence associated constraints. Since interval is always one-dimensional, it does not contain **reinit**. For both quadrature and domain objects, the purpose of reinit is to provide functionality that is "out of scope" of the assignment operator. In our codes, assignment is strictly designated for objects of the same dimension. In addition, assignment operator for quadrature objects requires that they are of the same degree and have the same number of nodes. Therefore, to improve code safety, the case when dimension of the object is changed is treated separately using **reinit** function.

To implement bases, a similar hierarchy is used. However, the implementation is more involved. Each class precomputes an index table and power tables for efficiency, which are reused throughout the duration of the object. All derived classes are required to compute basis functions, derivatives, and integrals for both monomial and orthogonal bases. There is no IntervalBasis, since performing Node Elimination over the interval is redundant. Although PolytopeBasis adds no additional data compared to Basis, it is important to treat domains that are not polytopes differently if they are added in the future. Section 5.3 discusses implementation of basis classes in more detail.

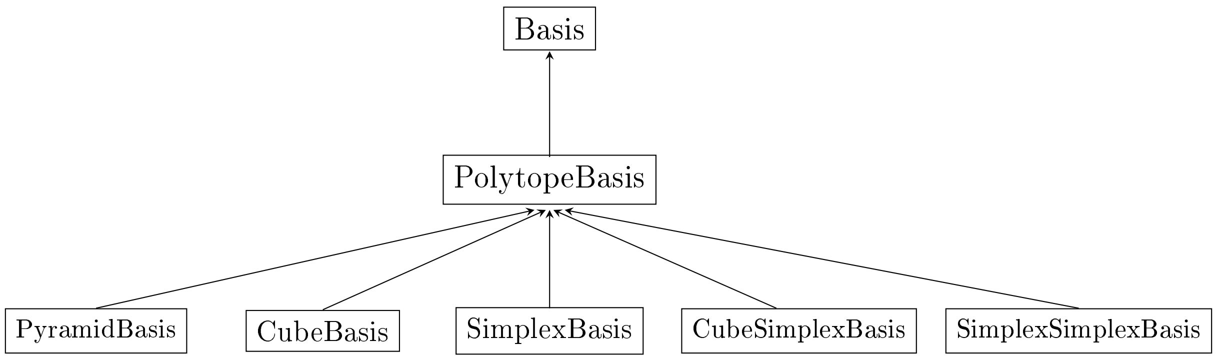

Figure 5.4. Class hierarchy for bases. Basis and PolytopeBasis are abstract classes.

All cubature objects must store nodes and weights in a suitable container. The most frequently encountered operations on cubatures are arithmetic operations on all nodes and weights, as well as individual access to each node and weight. Due to the process of eliminating nodes, occasional resizing is performed. Therefore, a dynamic array that supports element access, provides slicing, and efficient arithmetic operations fits well. That is accomplished by QuadArray class, which contains Array class, implements slicing of nodes, and stores information about the degree and dimension of the cubature.

Although inheriting from a concrete, non-abstract class such as QuadArray in many cases is not encouraged due to the problem of object slicing[25], it is beneficial here since it has a wide functionality that is reused by all derived classes. To avoid the problem of slicing, protected, rather than public inheritance is used.

Besides inheriting from QuadArray, QuadDomain has a few important responsibilities. It provides interface for returning Domain and Basis objects, so that they can be accessed and used in higher level routines. Similarly, QuadPolytope returns Polytope and PolytopeBasis on demand. Non-member functions **in_domain** and **in_constraint** determine whether constraints are satisfied. **Assign** and **clone_quad_domain** functions are provided to support polymorphic operations.

The following question arises: Does QuadDomain inherit from Domain and does QuadPolytope inherit from Polytope? If so, does QuadCube inherit from Cube? If that were the case, the class hierarchy would become complicated. From figure 5.6 it is clear that derived classes can reach their base classes through two paths, rather than one. This leads to a problem of two copies of the base class, which can be overcome by virtual inheritance. However, that would incur a cost of increased complexity and is best avoided[46]. Instead, all derived cubature classes store their domains by composition, and return them by reference when necessary.

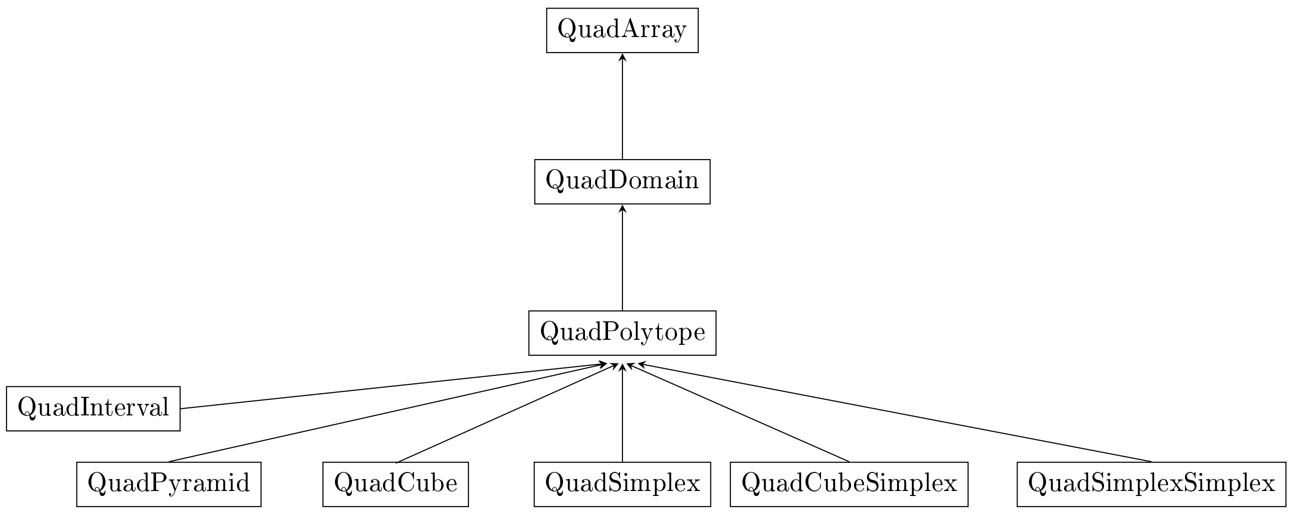

Figure 5.5. Class hierarchy for cubature objects. Derived classes contain information about their shapes and QuadArray for storing nodes and weights.

Based on these hierarchies, higher level routines are applied to abstract interfaces. For

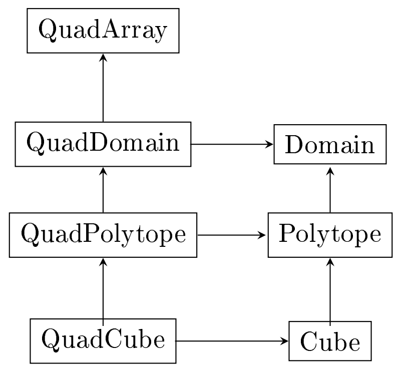

Figure 5.6. Class hierarchy for cubature objects if they were inherited from Domain. Special care is needed for incurred complexity.

example, a node elimination routine that operates on cubature rules over polytopes might have a prototype

QuadPolytope* NodeElimination(QuadPolytope* initial_guess);

Thus far, we have only implemented non-symmetric rules. If that changes in the future, separating symmetric and non-symmetric cases for polytopes would be straightforward

QuadSymPolytope* NodeElimination(QuadSymPolytope* initial_guess);

QuadNonSymPolytope* NodeElimination(QuadNonSymPolytope* initial_guess);

The only difference between QuadSymPolytope and QuadNonSymPolytope would be that they rely on symmetric and non-symmetric bases, respectively. In addition, QuadSymPolytope would provide **bool is_symmetric()=0;**.

5.1.2. Run Time vs Compile Time

Compile time programming has been gaining momentum lately. It allows to move error handling, algorithm decision making, and evaluation of routines to compile time, as opposed

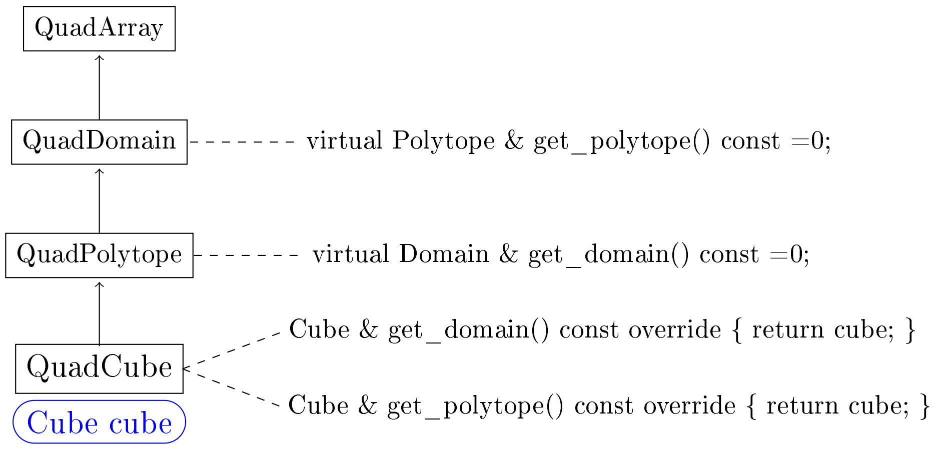
QuadArray
QuadDomain
virtual Polytope & get_polytope() const =0;
QuadPolytope
virtual Domain & get_domain() const =0;
QuadCube
Cube cube
Cube & get_domain() const override { return cube; }
Cube & get_polytope() const override { return cube; }

Figure 5.7. A better way to handle polymorphic behavior. **get_domain** is required for domains and **get_polytope** for polytopes. QuadCube stores Cube data member and returns it on demand.

to runtime[13]. In this way, one can find errors early without delaying it to runtime and in certain cases improve performance[1]. However, the compilation times tend to increase as well. As usual, whether benefits outweigh the drawbacks depends on the application. In the case of gen-quad, most computations are performed at runtime.

Given the iterative nature of the Node Elimination algorithm, parameters such as sizes of cubature arrays and sizes of matrices encountered in predictor and corrector are not known at compile time. Every time a node is eliminated, arrays and matrices shrink in size. Moreover, dimension parameters are not known at compile time, since recursive procedures reuse lower-dimensional rules at runtime. The only parameter that is fixed at compile time is the degree of precision, but even that is a subject to change since some techniques rely on reusing cubature rules of lower degrees[14] and might be employed in the future. For higher degrees and dimensions, most of arrays, tables, and matrices are too large to be stored on the stack. Instead, all containers allocated dynamically at runtime.

Originally, VLAs(variable length arrays) were used for small two and three-dimensional arrays. To insure portability and best practices, they have been replaced with dynamic arrays. Only in a few cases, such as storage for dimensions of a matrix or multi-dimensional array, fixed-size arrays are used.

5.1.3. Error Handling

Rigorous error handling is applied throughout the project. As is common, debug and release modes offer a trade-off between additional error checking and performance. Debug mode turns on many assertions, such as ensuring that containers are of non-negative length, operation on two arrays is applied only if they are of equal length, nonlinear solver returns success only if cubature rules are inside of the domain, and others. Another important feature of debug mode is range checking. Element access of arrays, tables, matrices, and other containers is checked for out of range errors. This is a considerable slowdown for computing the nonlinear function and Jacobian. In non-debug mode, these assertions are turned off, and only essential error handling is performed.

5.2. Linear Algebra

5.2.1. Libraries

A wide range of linear algebra libraries is available in C++. Some of the frequently used ones are Armadillo, Eigen, Blaze. All three of them are fully or partially based on implementations of BLAS and LAPACK libraries. When installing Armadillo, it searches for available BLAS and LAPACK on the system and provides convenient interfaces to them[10]. Eigen and Blaze, on the other hand, offer optional support that links to their routines to Intel's oneMKL, which is a highly optimized implementation targeted towards Intel architectures[3]. Otherwise, a default implementation of these routines is used, which is usually slower for large matrices[4]. Another feature of these libraries is that they rely on templates heavily, which select optimal algorithms at compile time but also increase compilation times[37].

Due to no external dependencies, header-only approach and convenient interfaces, Eigen is our library of choice. Therefore, gen-quad does not require installation of any packages or libraries. However, for larger degrees and dimensions, the presence of oneMKL library on the system is preferred due to increased performance. Makefile automatically searches if oneMKL is available on the system. If so, we provide flags necessary for Eigen to make the use of oneMKL routines.

5.2.2. Efficient Routines

Some routines cannot be directly expressed by existing library routines and require subdividing it into smaller tasks and choosing the most appropriate library routines. The most computationally demanding of those are the predictor and corrector updates. Recall that each of the initial guesses in the predictor step is computed as

$$\widetilde{\mathbf{z}}_k = \bar{\mathbf{z}} + \Delta\mathbf{z}_k^f + t\Delta\mathbf{z}_k^g,$$

where

$$\begin{aligned} \Delta\mathbf{z}_k^f &= \hat{Q}^T\Delta\mathbf{y}_k^f, \\ \Delta\mathbf{z}_k^g &= \hat{Q}^T\Delta\mathbf{y}_k^g. \end{aligned} \tag{5.1}$$

Provided that $\mathbf{m}_k$ is a kth column of $\hat{Q}^T$, $\mathbf{y}_k^f$ and $\mathbf{y}_k^g$ are

$$\begin{aligned} \Delta\mathbf{y}_k^f &= \mathbf{m}_k\frac{\bar{w}_k}{\|\mathbf{m}_k\|^2} \\ \Delta\mathbf{y}_k^g &= \left(I - \frac{\mathbf{m}_k\mathbf{m}_k^T}{\|\mathbf{m}_k\|^2}\right)\hat{\mathbf{g}}. \end{aligned}$$

It is tempting to loop over all k's and store $\widetilde{\mathbf{z}_k}$ in a larger matrix $\widetilde{\mathbf{Z}}$ one column at a time. However, that implies K matrix-vector products in (5.1) for both $\mathbf{y}_k^f$ and $\mathbf{y}_k^g$, where K is the number of nodes. That would result in inefficient utilization of hardware resources, since $O(MN)$ floating point operations are performed on $O(MN)$ data for each matrix-vector product. Instead, it is advantageous to reorder the algorithm and express is it in terms of matrix-matrix multiplication, which performs $O(MNK)$ operations and uses caching to minimize cache misses[38]. The optimized version is

$$\begin{aligned} \Delta\mathbf{z}_k^f &= \hat{Q}^T \Delta\mathbf{Y}_\mathbf{k}^\mathbf{f}, \\ \Delta\mathbf{z}_k^g &= \hat{Q}^T \Delta\mathbf{Y}_\mathbf{k}^\mathbf{g}, \end{aligned} \tag{5.2}$$

where $\mathbf{Y}_\mathbf{k}^\mathbf{f}$ and $\mathbf{Y}_\mathbf{k}^\mathbf{g}$ are matrices. To achieve that, $\mathbf{y}_k^f$ and $\mathbf{y}_k^g$ are evaluated and stored ahead of time for each K. The cost of that is storing additional two matrices in memory, which is not a limitation due to large availability of memory on modern CPU architectures. On Maneframe II [2], a typical CPU node has 256 GB of RAM, which is more than sufficient for the sizes of our matrices. The strategy in (5.2) has improved the performance between a factor of 10 and 50, depending on the size of a problem. Although not as substantial, additional performance improvements are gained by grouping $\mathbf{Y}_\mathbf{k}^\mathbf{f}$ and $\mathbf{Y}_\mathbf{k}^\mathbf{g}$ in one matrix and calling a single matrix-matrix multiplication routine

$$\begin{bmatrix} \Delta\mathbf{z}_k^f \\ \Delta\mathbf{z}_k^g \end{bmatrix} = \hat{Q}^T \begin{bmatrix} \Delta\mathbf{Y}_\mathbf{k}^\mathbf{f} \\ \Delta\mathbf{Y}_\mathbf{k}^\mathbf{g} \end{bmatrix}.$$

If $J = LQ$ is the economy size factorization, corrector update is

$$\widetilde{\mathbf{z}} \leftarrow \widetilde{\mathbf{z}} + \Delta\mathbf{z}^f + t\Delta\mathbf{z}^g, \tag{5.3}$$

where

$$\begin{aligned} \Delta\mathbf{z}^f &= -Q^T L^{-1}\mathbf{f}, \\ \Delta\mathbf{z}^g &= -(I - Q^T Q)\mathbf{g}. \end{aligned} \tag{5.4}$$

The term $Q^T L^{-1}\mathbf{f}$ is simply a solution of the nonlinear system $J^T\Delta\mathbf{z}^f = \mathbf{f}$, but we do not call a direct solver routine because it is based on QR factorization, which does not store information about matrix Q and additional work would be required for computing QR(or LQ) factorization for $\Delta\mathbf{z}^g$. To avoid redundant work, the system $J^T\Delta\mathbf{z}^f = \mathbf{f}$ is solved using a specialized routine where Q^T is reused. Further, the explicit computation of Q^T is not performed, but rather its reflectors are applied to $L^{-1}\mathbf{f}$ to obtain $Q^TL^{-1}\mathbf{f}$. Similarly, reflectors of Q are applied to **g**, which yields $Q\mathbf{g}$. Finally, by applying reflectors of Q^T to $Q\mathbf{g}$, the final expression $Q^TQ\mathbf{g}$ is computed.

5.3. Implementation of Bases

5.3.1. Performance

Classes that compute basis functions, their derivatives, and integrals were presented in figure 5.3. Since they are implemented for arbitrary degree and dimension, some compromise between performance and complexity of the code is expected. It was mentioned that each class uses a table of multi-indexes to raise monomial and orthogonal polynomials to the integer powers that form S_p^d. However, it comes at the cost of good, but suboptimal performance when compared to evaluating multi-indexes on the fly. Consider a matrix for S_2^3

$$S_p^d = \begin{bmatrix} 0 & 1 & 2 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 2 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 2 \end{bmatrix}.$$

Unlike in chapter 4, each multi-index corresponds to a column rather than a row of the matrix. Such layout is chosen since our data structures are stored in row-major order. When computing basis functions, tasks are more naturally subdivided when looping over rows. With minimal modifications of the source code syntax, **orthog_basis** routine computes orthogonal basis over a cube:

```
const Array & CubeBasis::orthog_basis(const Array & x)
{
    ArrayTable2D legendre(dim, deg+1);
        for(int d = 0; d < dim; ++d)
            Legendre(x[d], deg+1, legendre(d));
    for(int k = 0; k < num_funcs; ++k)
        functions[k] = legendre(0, index_table(0, k));

    for(int d = 1; d < dim; ++d)
        for(int k = 0; k < num_funcs; ++k)
                functions[k] *= legendre(d, index_table(d, k));
    return functions;
}
```

First, all Legendre polynomials are computed and stored in array of arrays **legendre**. After that, multi-indexes stored in **index_table** are applied to **legendre**, which access appropriate indexes of Legendre polynomials. In particular,
legendre(d, index_table(d, k)) $\equiv L(x_d)_{index\_table(d,k)}$.
From the code it is evident that accesses to **functions[k]** and **index_table(d, k)** inside the loops are contiguous, justifying the choice of storing multi-indexes in columns.

However, indexes of S_p^d themselves do not follow a contiguous pattern, and thus **legendre**(d, index_table(d, k)) is not contiguous with respect to data stored in **legendre**. This inhibits vectorization, but cache hits are high since values of index_table are close to each other.

A faster approach would be to generate multi-indexes on the fly and avoiding the access to them altogether. Consider the following excerpt of code(with minor syntactic changes for convenience):

```
if(dim == 2)
{
    int count{};
    for(int i{}; i < deg+1; ++i)
        for(int j{}; j < deg+1-i; ++j)
            functions[count++] = legendre(1, i) * legendre(0,j);
}
else if(dim == 3)
{
    int count{};
    for(int i{}; i < deg+1; ++i)
        for(int j{}; j < deg+1-i; ++j)
            for(int k{}; k < deg+1-i-j; ++k)
                functions[count++] = legendre(2, i) * legendre(1, j) * legendre(0, k);
}
else if(dim == 4)
{
    int count{};
    for(int i{}; i < deg+1; ++i)
        for(int j{}; j < deg+1-i; ++j)
            for(int k{}; k < deg+1-i-j; ++k)
                for(int l{}; l < deg+1-i-j-k; ++l)
                    functions[count++] = legendre(3, i) * legendre(2, j) *
                                         legendre(1, k) * legendre(0, l);
⋮
⋮
}
```

Here the indexes are generated on the fly inside the nested loop. However, additional loop is nested with increment of dimension. Therefore, for every dimension of interest, different nesting must be used. The resulting code is more efficient as it enables vectorization and reduces data accesses, but it comes at the cost of sacrificing dimensional independence. That was the reason for storing indexes in a matrix in the first place. Evaluation on the fly would require nesting similar loops for all orthogonal basis functions, as well as monomials, which would considerably increase the number of lines to be written. For domains other than cube, computing bases is less trivial, which would result in a highly error prone repetition of the code.

Since linear algebra routines are the major source of computational cost, the slowdown associated with a more general strategy does not have a significant impact on the runtime cost as a whole. In fact, for sufficiently large problems, speedup of at most 200 percent for basis functions was observed with the faster version, whereas they consume less than 10 percent of the runtime for most problems.

5.3.2. Derivatives

Implementation of analytical derivatives(of basis functions) for a cube is reasonably straightforward, but it is much more challenging for a simplex and tensor domains, since functions themselves are more complex. Therefore, finite differences are used to approximate their derivatives. Consequently, the nonlinear function and Jacobian are approximate as well. To ensure sufficient accuracy, fourth order finite difference is used

$$\frac{d}{dx}f(x+h) = \frac{(f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h))}{12h}. \tag{5.5}$$

The optimal parameter h is a compromise between truncation error and floating-point error due to division. In our case, $h = 5 \times 10^{-5}$ produced the most accurate approximations. The optimal parameter can be derived analytically, which is demonstrated in[18]. For the second order central finite difference, they show that optimal h$\approx 6.055 \times 10{-}5$.

5.4. Parallelization

The most expensive components of gen-quad are linear algebra routines, followed by evaluation of Jacobian and nonlinear function. The Amdahl's law states that the speedup of the program obtained from parallelizing code region p is given by

$$S(p) = \frac{1}{1 - p + \frac{p}{s}}, \tag{5.6}$$

where s is the speedup obtained from parallelizing p and $1 - p$ is the serial fraction of the program. Maximum theoretical speedup is limited to the inverse of the serial fraction of a program

$$\lim_{s\to\infty} S_{max}(p) = \frac{1}{1-p}. \tag{5.7}$$

Therefore, it is important that serial portion of the code does not contain any serious inefficiencies. Some possibilities include sorting routine in the predictor, common operations on arrays, testing whether cubature rules are inside of the domain, etc. Our containers use expression templates to avoid redundant temporaries, whereas optimized std::sort is used for sorting. Fortunately, as the problem size grows, serial portion of the code decreases compared to the parallel one, which opens more opportunities for speedup. Now we discuss potential strategies for exploiting parallelism.

5.4.1. Linear Algebra

If oneMKL library is installed on the system, Eigen calls oneMKL library for computationally intensive routines, which support multithreading. When compiled with OpenMP(-fopenmp on GCC or -qopenmp on Intel compilers), parallelism is enabled automatically. Without the oneMKL, QR factorization remains serial.

OpenMP is a convenient and widely used tool for parallization, but is limited to a single CPU node. Another possibility is to add optional calls to the GPU library, such as MAGMA. When using a GPU, one must take into account costs associated with memory transfer from a CPU. Together with the fact that oneMKL is highly efficient on 32 or 64 CPU cores, GPU would be only beneficial for very large problems. There are also linear algebra libraries available for taking advantage of multiple CPU nodes with MPI(Scalapack, SLATE). However, MPI would require a significant redesign. For the remainder of the discussion, only OpenMP parallelism on a single node is considered.

5.4.2. Function and Jacobian

The Jacobian resembles a well-parallelizable problem. Each thread is responsible for computing only a submatrix of rows and columns. Besides task subdivision, no other interaction between threads is needed. Since the code primarily uses transpose of the Jacobian rather than the Jacobian itself, the transposed version is computed directly to avoid transposition.

The illustration above is a simplification due to the nature of the nonlinear function

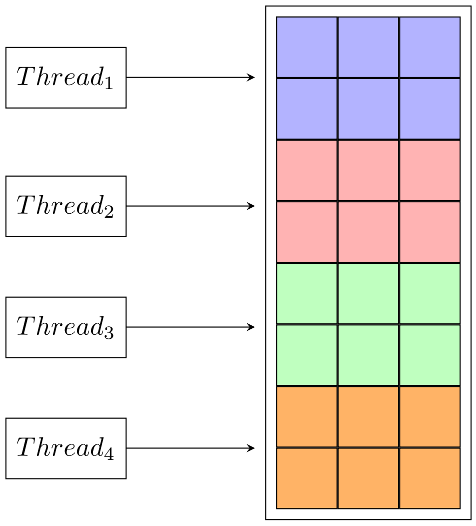

Figure 5.8. Computation of J^T is evenly partitioned among threads, simplified illustration.

$$\mathbf{f}(\mathbf{x}, \mathbf{w}) = \mathbf{\Phi}(\mathbf{x})\mathbf{w} - \mathbf{b}, \tag{5.8}$$

$$\mathbf{\Phi}(\mathbf{x}) = \left[\Phi(x_1), \dots, \Phi(x_n)\right] \in \mathbb{R}^{M\times n}, \quad \Phi(\mathbf{x}) = \begin{bmatrix} \phi_1(\mathbf{x}) \\ \vdots \\ \phi_M(\mathbf{x}) \end{bmatrix},$$

$$\mathbf{b} = \begin{bmatrix} \int_\Omega \phi_1(\mathbf{x}) w(\mathbf{x})\, d\mathbf{x} \\ \vdots \\ \int_\Omega \phi_M(\mathbf{x}) w(\mathbf{x})\, d\mathbf{x} \end{bmatrix},$$

where $\mathbb{P}_p^d$ is orthogonal basis. The analytical Jacobian then becomes

$$J(\mathbf{x}) = \begin{bmatrix} \Phi(\mathbf{x}_1) & \Phi(\mathbf{x}_2) & \dots & \Phi(\mathbf{x}_n) & w_1\Phi'(\mathbf{x}_1) & w_2\Phi'(\mathbf{x}_2) & \dots & w_n\Phi'(\mathbf{x}_n) \end{bmatrix}. \tag{5.9}$$

Since $\mathbf{x}_i \in R^d, \Phi'(\mathbf{x}_i)$ is a Jacobian itself. Then

$$J(\mathbf{x}) \in R^{M\times n(dim+1)},$$

and

$$J^T(\mathbf{x}) \in R^{n(dim+1)\times M}.$$

Equation (5.9) shows that two separate loops(at least conceptually) are needed. The first loop computes derivatives with respect to weights, which are orthogonal basis functions. The second loop computes derivatives with respect to nodes, which are derivatives of basis functions multiplied by weights. If a single loop was used, cache locality would be lost due to significant jumps across different columns(or rows in the case of a transpose). In the code below, two loops are computed in parallel using **omp schedule(static)**, partitioning the work in submatrices of equal size.

```
void EvalJacobian::eval_jacobian_omp(const QuadDomain & q, Matrix & JT)
{
    using namespace function_internal;
    int nthreads = int(ba.components.size());

    #pragma omp parallel default(none) shared(q, JT) num_threads(nthreads)
    {
        auto & bLoc = ba.components.at(omp_get_thread_num());
        #pragma omp for schedule(static)
        for(int j = 0; j < q.num_nodes(); ++j)
            JT.row(j) = ((*bLoc).*functions_fptr)(Array(q.node(j)));

        #pragma omp for schedule(static)
        for(int j = 0; j < q.num_nodes(); ++j)
        {
            auto & bder = ((*bLoc).*derivatives_fptr)(Array(q.node(j))) * q.w(j);
            int rowId = j*q.dim()+q.num_nodes();
            for(int d = 0; d < q.dim(); ++d)
                for(int k = 0; k < JT.ncols(); ++k)
                    JT(rowId+d, k) = bder[JT.ncols()*d+k];
        }
    }
}
```

Serial and parallel versions of the Jacobian were benchmarked and compared on a 36-core Xeon Phi processor, gcc 7-3 compiler with -O3 optimizations enabled. The table lists wall clock times for a Jacobian of a cubature rule over a 5-dimensional simplex of varying degrees. The number of nodes was selected as if the tensor product rule were used.

degree	7	9	11	13
size	6144×792	18750×2002	46656×4368	100842×8568
serial	0.248	1.97	14.8	60.2
parallel	0.0219	0.0763	0.449	1.88
speedup	11.3	25.8	32.9	32.0

Table 5.1. Serial and parallel setup times of the Jacobian on a 36-core processor for T_5 measured in seconds. For a sufficiently large problem, a nearly perfect speedup is observed.

For computing $\mathbf{f}$, note that (5.8) is a summation

$$\mathbf{f}(\mathbf{x}, \mathbf{w}) = \mathbf{\Phi}(\mathbf{x})\mathbf{w} - \mathbf{b} = \sum_i \Phi(\mathbf{x}_i) w_i - \mathbf{b}. \tag{5.10}$$

Computation of integrals, i.e. b is negligible compared to the summation term and is evaluated serially. On the other hand, evaluation of the sum is subdivided into partial sums. Once each thread computes its partial sum and stores it in a vector, final sum is obtained by adding partial sums. The final result is computed using **omp critical** clause to avoid race conditions. Although there is an overhead associated with a critical region, addition of vectors takes a small time compared to computing local sums.

No dynamic memory allocation takes place when computing either of the final result or partial sums. Since the nonlinear solver uses $\mathbf{f}$ and $\mathbf{J}$ repeatedly, buffers are allocated only once and reused, avoiding overhead of allocating memory. The benchmark for comparing serial and parallel versions of computing $\mathbf{f}$ was performed using identical problem, hardware, compiler, and compiler options as for $\mathbf{J}$.

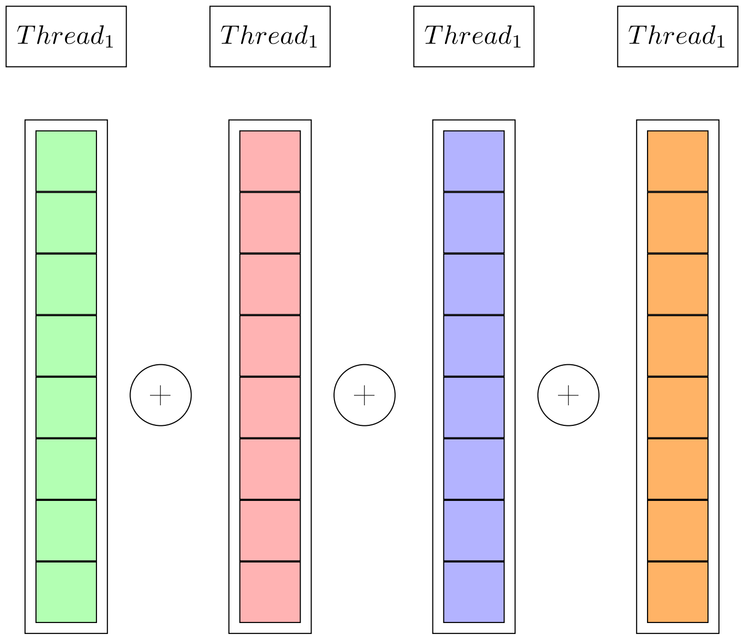

Figure 5.9. Computation of f is evenly partitioned among threads. Each thread computes the local sum and stores it in a vector. Then serial addition is performed to add contribution from each thread.

degree	7	9	11	13
serial	0.0128	0.0941	0.55	2.30
parallel	0.00833	0.00907	0.0241	0.0889
speedup	1.5	10.4	22.8	25.9

Table 5.2. Serial and parallel times of the function on a 36-core processor for T_5 measured in seconds. Evaluation of $\mathbf{f}$ is substantially cheaper than $\mathbf{J}$. Combined with the fact that computing $\mathbf{f}$ involves a small serial component, larger problem size is needed to observe good speedup.

Chapter 6

Numerical Results

This chapter presents a wide range of results and summarizes them. The efficiency index is examined in detail. The accuracy of obtained cubature rules is tested against analytically known problems.

6.1. Metric

In chapter 3 we described the Node Elimination algorithm for selecting one of the multiple solutions. The cubature rule whose "worst node" is furthest from the boundary(inside the domain) was selected. We call the parameter that determines the maximum number of successful solutions a **search width**. If search width is set to 1, the first solution that is successful is selected, without taking distance from the boundary into account. Larger parameters of **w** increase computational cost but usually lead to slightly more efficient rules. To compromise quality of the rules and computational time, we choose **search width** $= 3$ for all experiments.

Due to undeterministic nature of the Node Elimination algorithm, it is nearly impossible to choose the strategy that would yield optimal results for all instances of degree and dimension. For example, lower search width sometimes may lead to better rules. For that reason, the average efficiency index is introduced. Given a dimension and type of the domain, we define it as the average over all degrees of consideration

$$e_{ave} = \frac{\sum_i^n e_{deg_i}}{n}. \tag{6.1}$$

6.2. Two Dimensions

Although computation of cubature rules in two dimensions is well studied, we present rules over a square and triangle for completeness. To be concise, results for even degrees are excluded. We start with degree five and go up to thirty-one.

6.2.1. Square

The table 6.1 summarizes results for C_2. In most cases, we reach the efficiency of at least 95%. Another observation is that lower-degree rules are further from the optimum due to relatively few nodes.

degree	5	7	9	11	13	15	17
n_{opt}	7	12	19	26	35	46	57
n_{elim}	8	14	20	28	37	49	60
i_{opt}	0.88	0.86	0.95	0.93	0.95	0.94	0.95

degree	19	21	23	25	27	29	31
n_{opt}	70	85	100	117	136	155	176
n_{elim}	73	87	103	121	140	158	178
i_{opt}	0.96	0.98	0.97	0.97	0.97	0.98	0.99

Table 6.1. Data for the cubature rules on the C_2. $e_{ave} = 0.95$.

6.2.2. Triangle

In general, results for a triangle are slightly better than for a square and some optimal rules with $i_{opt} = 1$ are obtained(table 6.2). Without numerical experiments in higher dimensions(which will verify this assumption), it is too early to conclude that our algorithm favors simplex over a cube in arbitrary dimension.

degree	5	7	9	11	13	15	17
n_{opt}	7	12	19	26	35	46	57
n_{elim}	7	12	19	27	36	47	58
i_{opt}	1.00	1.00	1.00	0.96	0.97	0.98	0.98

degree	19	21	23	25	27	29	31
n_{opt}	70	85	100	117	136	155	176
n_{elim}	73	88	101	118	139	158	178
i_{opt}	0.96	0.97	0.99	0.99	0.98	0.98	0.99

Table 6.2. Data for the cubature rules on the T_2. $e_{ave} = 0.98$.

6.3. Three Dimensions

Although there appears to be less research for computing efficient cubature rules in three dimensions, successful results have been obtained for a cube [50], simplex [30], and pyramid [16]. We present the results for all three domains for odd degrees in the range [5, 23]. In addition, we present less frequently studied three-dimensional prism($C_1 \times S_2$).

6.3.1. Cube vs Simplex

Relationship between three-dimensional cube and simplex(tetrahedron) are similar to the square and triangle. In particular, tetrahedron always produced more efficient cubature rules than cube. The efficiency only slightly dropped for the cube compared to the square, whereas it did not drop for the tetrahedron compared to the triangle at all. This is a desirable outcome, since obtaining similar efficiency in higher dimensions is one of the primary challenges of high-dimensional integration. Results are summarized in tables 6.3 and 6.4.

6.3.2. Pyramid

Originally gen-quad was designed for n-dimensional cubes, simplexes, and their tensor product domains. However, constrained optimization techniques presented in chapter 3 can be applied to arbitrary convex polytope. To verify that these techniques in practice produce efficient rules for other domains, we have incorporated 3-dimensional pyramid to the package. For all degrees, the rules are at most three nodes away(table 6.5). Although the average efficiency index(with rounding up to two significant digits) is the same as for the tetrahedron, the rules are on average slightly more efficient.

6.3.3. Prism

Three-dimensional prism is the tensor product domain of the interval and the triangle. Similar to pyramid, the rules are at most three nodes away from the optimum(table 6.6).

6.4. Four Dimensions

We present results for T_4, C_2T_2, C_1T_3 and T_2T_2(tables 6.7, 6.8, 6.9, and 6.10). The efficiency index for a cube is satisfactory but lower compared to other four-dimensional polytopes. One possible reason is that the penalty term introduced in the Newton's method is more beneficial for integrals containing triangles or tetrahedra. For the remaining four-dimensional domains, all cubature rules have efficiency index of at least 0.95, and in many cases close to 1. Odd degrees in the range five to fifteen are presented.

6.5. Five Dimensions

As expected, the efficiency index for five-dimensional domains is lower, but is above 0.85 for the most cubature rules. Moreover, the efficiency improves for higher degrees and is close to 1 for degrees 11 and 12. This appears to be due to constrained optimization, which ensures that the node elimination produces rules that are sufficiently far inside the domain. Consequently, more nodes are eliminated in the long run. Since more node eliminations are carried for the problems with higher degrees, the effect is more pronounced. Another observation is that C_5 produced more efficient rules for even degrees, whereas other domains generally performed better for odd degrees. In addition, tensor product domains have generally higher efficiency indexes. We present degrees from four to twelve for C_5, T_5, C_3T_2, and T_3T_2(tables 6.11, 6.12, 6.13, and 6.14).

6.6. Six Dimensions

Finally, results for six-dimensional domains are presented(tables 6.15, 6.16, 6.17, and 6.18). Although the efficiency index has dropped, it is above 0.8 for most rules. In some cases, the efficiency is as high as 0.98.

Except for $T_3 \times T_3$, degrees four to ten are presented. For $T_3 \times T_3$, degree ten rule is excluded, as the estimated time to compute the rule is about three weeks. The reason is that the recursive initial guess for $T_i \times T_j$ contains more nodes. For the remaining domains, six-dimensional rules of degree ten took about two days on Maneframe's mic node with 64 cores, which reflects that the algorithm is suitable for moderate degrees up to dimension six. To obtain rules up to degree fifteen of dimension six in a reasonable amount of time, one would require changing the implementation that would use more computational resources, such as GPUs.

6.7. Runtime Comparison

In this section, runtimes for C_2. C_4, and C_6 are presented. For the other problems of the same dimension(except $T_i \times T_j$), the runtimes follow a similar pattern. The cost is somewhat higher when $\Omega = T_i \times T_j$, since the initial guess involves a tensor product of T_i and T_j, which contains more nodes than recursive initial guesses for other polytopes. Tables 6.19, 6.20, and 6.21 show the runtimes for different degrees.

All experiments were conducted on a 64-core Knights Landing processor. The results reflect rapid increase of computational cost in higher dimensions. In two dimensions, degree 40 rule took just over a minute, whereas degree 10 rule in six dimensions took almost 48 hours. However, as the degree of precision increases, the runtimes increase at a slower rate due to increased parallelism.

6.8. Quadrature Errors

In this section we compare the quality of our rules against tensor product Gauss rules in a similar fashion as we have done in [6], but for a different test function and more domains.

In particular, the results are shown for T_2, T_4, $T_2 \times T_2$, C_6, and T_6 over a test function in R^d

$$f(\mathbf{x}) = \exp(c\mathbf{x}) \equiv \prod_{i=1}^{d} \exp(cx_i). \tag{6.2}$$

The parameter $c = \frac{d}{24}$ is chosen to ensure that the exponent is greater for lower dimensions so that residual does not approach machine precision too rapidly.

6.8.1. Two Dimensions

If $\Omega = C_d$, we get

$$\int_{C_d} \exp(c\mathbf{x}) d\mathbf{x} = \frac{1}{c^d}(\exp(c) - 1)^d. \tag{6.3}$$

If $\Omega = T_d$, it can be shown by induction that

$$\int_{T_d} \exp(c\mathbf{x}) d\mathbf{x} = \frac{1}{d!\ c^d} \sum_{i=0}^{d} (-1)^i \exp(c(d-i)) \binom{d-i}{\text{dim}}. \tag{6.4}$$

The integral over $T_i \times T_j$ is a product of integrals over T_i and T_j

$$\int_{T_i \times T_j} \exp(c\mathbf{x}) d\mathbf{x} = \int_{T_i} \exp(c\mathbf{x}) d\mathbf{x} \int_{T_j} \exp(c\mathbf{x}) d\mathbf{x}. \tag{6.5}$$

Figures 6.1 and 6.2 show results for the square and triangle. The left plot shows the relative error versus the degree, and the right plot shows the error versus the number of nodes. For a square, the error of Gauss tensor rules has a smaller error, which is explained by the fact that tensor product rules integrate a larger polynomial space. As a consequence, tensor product of error vs number of points are slightly better for tensor product rules. On the other hand, additional polynomials did not reduce the error for a triangle and our rules have less nodes for the same accuracy.

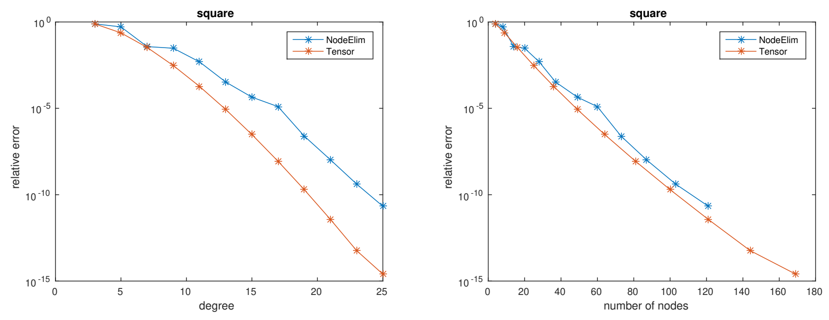

Figure 6.1. Comparison of the tensor product rules and the rules of table 6.1. $\Omega = C_2$. Left: error vs. degree. Right: error vs. number points.

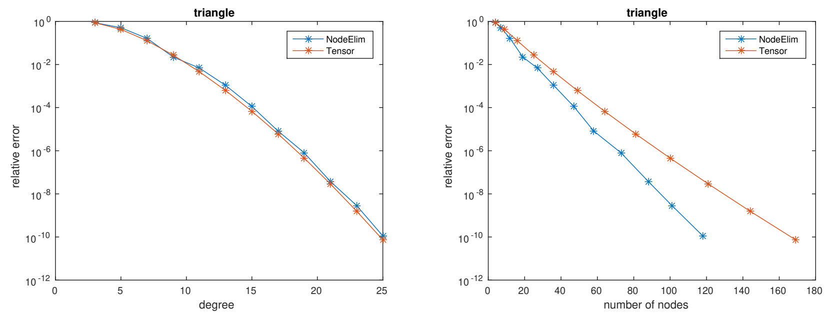

Figure 6.2. Comparison of the tensor product rules and the rules of table 6.2. $\Omega = T_2$. Left: error vs. degree. Right: error vs. number points.

6.8.2. Four Dimensions

The gain of using rules obtained by the Node Elimination increases with the dimension. Similar to T_2, the tensor rules for T_4 are only slightly more accurate, and the number of nodes required to achieve the same accuracy is substantially smaller for our rules. Since integration of T_2T_2 involves a product of two domains, the Gauss rules gain some additional accuracy, but the effect is not as pronounced as for C_2. Figures 6.3 and 6.4 illustrate savings for both domains.

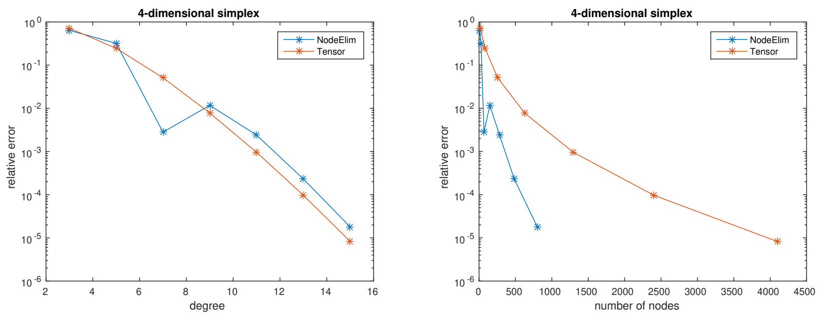

Figure 6.3. Comparison of the tensor product rules and the rules of table 6.8. $\Omega = T_4$. Left: error vs. degree. Right: error vs. number points.

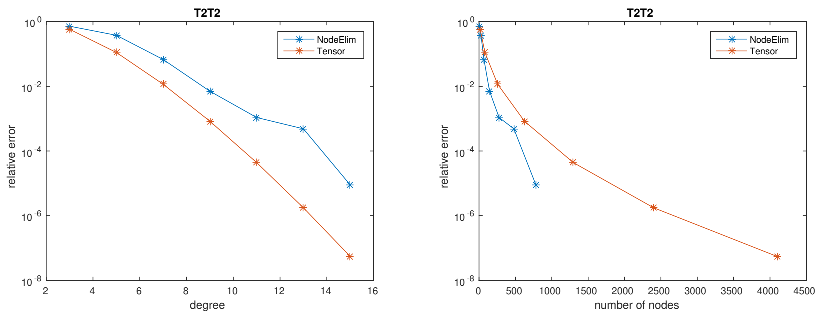

Figure 6.4. Comparison of the tensor product rules and the rules of table 6.10. $\Omega = T_2 \times T_2$. Left: error vs. degree. Right: error vs. number points.

6.8.3. Six Dimensions

Despite that our rules are considerably less accurate for the same degree when $\Omega = C_6$, they prevail in number of nodes vs relative error plot due to high efficiency index(figure 6.5). For T_6 the degree vs relative error plot in figure 6.6 shows marginal differences. The number of nodes vs relative error plot shows that our rules are considerably more efficient.

Note that with the increase of dimension, the degree of precision required to integrate with the same accuracy increases, exhibiting curse of dimensionality. If higher accuracy is required than we have shown here, higher degree rules can be obtained via the Node

Elimination algorithm.

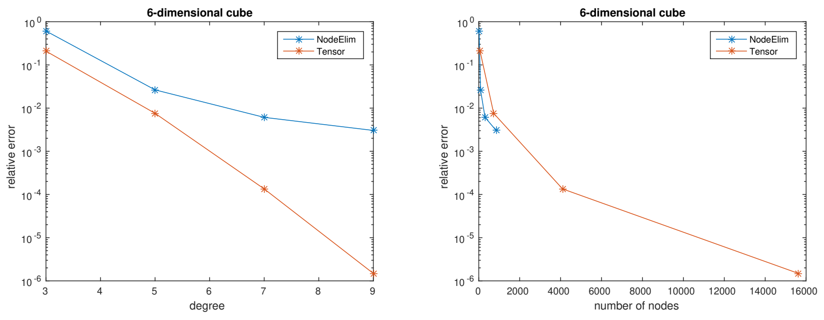

Figure 6.5. Comparison of the tensor product rules and the rules of table 6.15. $\Omega = C_6$. Left: error vs. degree. Right: error vs. number points.

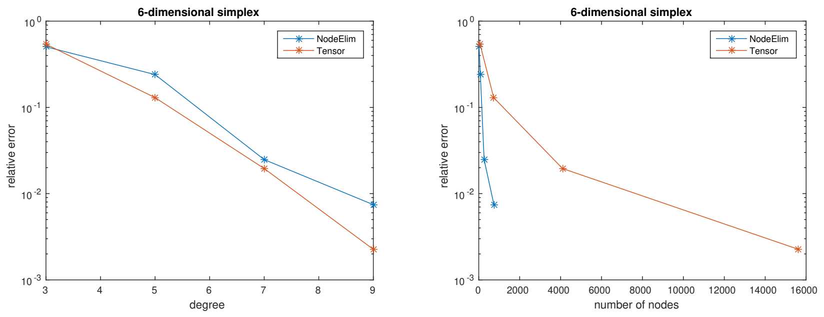

Figure 6.6. Comparison of the tensor product rules and the rules of table 6.16. $\Omega = T_6$. Left: error vs. degree. Right: error vs. number points.

degree	5	7	9	11	13
n_{opt}	14	30	55	91	140
n_{elim}	16	34	60	98	151
i_{opt}	0.88	0.88	0.92	0.93	0.93

degree	15	17	19	21	23
n_{opt}	204	285	385	506	650
n_{elim}	220	301	405	529	653
i_{opt}	0.93	0.95	0.95	0.96	1.00

Table 6.3. Data for the cubature rules on the C_3. $e_{ave} = 0.93$.

degree	5	7	9	11	13
n_{opt}	14	30	55	91	140
n_{elim}	14	32	57	94	142
i_{opt}	1.00	0.94	0.96	0.97	0.99

degree	15	17	19	21	23
n_{opt}	204	285	385	506	650
n_{elim}	207	288	388	508	653
i_{opt}	0.99	0.99	0.99	1.00	1.00

Table 6.4. Data for the cubature rules on the T_3. $e_{ave} = 0.98$.

degree	5	7	9	11	13
n_{opt}	14	30	55	91	140
n_{elim}	15	31	56	92	142
i_{opt}	0.93	0.97	0.98	0.99	0.99

degree	15	17	19	21	23
n_{opt}	204	285	385	506	650
n_{elim}	206	287	388	508	652
i_{opt}	0.99	0.99	0.99	1.00	1.00

Table 6.5. Data for the cubature rules on P_3. $e_{ave} = 0.98$.

degree	5	7	9	11	13
n_{opt}	14	30	55	91	140
n_{elim}	15	31	56	93	142
i_{opt}	0.93	0.97	0.98	0.98	0.99

degree	15	17	19	21	23
n_{opt}	204	285	385	506	650
n_{elim}	207	288	387	509	652
i_{opt}	0.99	0.99	0.99	0.99	1.00

Table 6.6. Data for the cubature rules on C_1T_2. $e_{ave} = 0.98$.

degree	5	7	9	11	13	15
n_{elim}	31	81	165	312	534	861
n_{opt}	26	66	143	273	476	776
i_{opt}	0.84	0.82	0.87	0.88	0.89	0.90

Table 6.7. Data for the cubature rules for C_4. $e_{ave} = 0.87$

degree	5	7	9	11	13	15
n_{elim}	26	68	148	281	483	800
n_{opt}	26	66	143	273	476	776
i_{opt}	1.00	0.97	0.97	0.97	0.99	0.97

Table 6.8. Data for the cubature rules for T_4. $e_{ave} = 0.98$

degree	5	7	9	11	13	15
n_{elim}	26	67	145	276	482	778
n_{opt}	26	66	143	273	476	776
i_{opt}	1.00	0.99	0.99	0.99	0.99	1.00

Table 6.9. Data for the cubature rules for C_1T_3. $e_{ave} = 0.99$

degree	5	7	9	11	13	15
n_{elim}	27	68	145	276	479	782
n_{opt}	26	66	143	273	476	776
i_{opt}	0.96	0.97	0.99	0.99	0.99	0.99

Table 6.10. Data for the cubature rules for $T_2 \times T_2$. $e_{ave} = 0.98$

degree	4	5	6	7	8
n_{elim}	23	53	81	164	220
n_{opt}	21	42	77	132	215
i_{opt}	0.91	0.79	0.95	0.80	0.98

degree	9	10	11	12
n_{elim}	378	508	843	1041
n_{opt}	334	501	728	1032
i_{opt}	0.88	0.99	0.86	0.99

Table 6.11. Data for the cubature rules for C_5. $e_{ave} = 0.90$

degree	4	5	6	7	8
n_{elim}	26	46	85	140	250
n_{opt}	21	42	77	132	215
i_{opt}	0.80	0.91	0.91	0.94	0.86

degree	9	10	11	12
n_{elim}	350	526	741	1062
n_{opt}	334	501	728	1032
i_{opt}	0.95	0.95	0.98	0.97

Table 6.12. Data for the cubature rules for $T5$. $e_{ave} = 0.92$

degree	4	5	6	7	8
n_{elim}	23	44	81	134	222
n_{opt}	21	42	77	132	215
i_{opt}	0.91	0.95	0.95	0.99	0.97

degree	9	10	11	12
n_{elim}	337	511	744	1044
n_{opt}	334	501	728	1032
i_{opt}	0.99	0.98	0.98	0.99

Table 6.13. Data for the cubature rules for $C_3 \times T_2$. $e_{ave} = 0.97$

degree	4	5	6	7	8	9	10	11	12
n_{elim}	24	44	84	136	223	340	510	737	1048
n_{opt}	21	42	77	132	215	334	501	728	1032
i_{opt}	0.88	0.95	0.92	0.97	0.96	0.98	0.98	0.99	0.98

Table 6.14. Data for the cubature rules for T_3T_2. $e_{ave} = 0.96$

degree	4	5	6	7	8	9	10
n_{elim}	32	86	140	306	443	849	1161
n_{opt}	30	66	132	246	429	715	1144
i_{opt}	0.94	0.77	0.94	0.80	0.97	0.84	0.98

Table 6.15. Data for the cubature rules for C_6. $e_{ave} = 0.89$

degree	4	5	6	7	8	9	10
n_{elim}	44	78	148	262	473	741	1239
n_{opt}	30	66	132	246	429	715	1144
i_{opt}	0.68	0.85	0.89	0.94	0.90	0.96	0.92

Table 6.16. Data for the cubature rules for T_6. $e_{ave} = 0.88$

degree	4	5	6	7	8	9	10
n_{elim}	34	69	147	259	452	732	1202
n_{opt}	30	66	132	246	429	715	1144
i_{opt}	0.88	0.96	0.90	0.95	0.95	0.98	0.95

Table 6.17. Data for the cubature rules for C_3T_3. $e_{ave} = 0.94$

degree	4	5	6	7	8	9
n_{elim}	35	70	143	252	450	727
n_{opt}	30	66	132	246	429	715
i_{opt}	0.86	0.94	0.92	0.98	0.95	0.98

Table 6.18. Data for the cubature rules for T_3T_3. $e_{ave} = 0.94$

degree	5	10	15	20	25	30	35	40
time(seconds)	0.01	0.13	0.55	1.67	3.88	14.1	34.6	83.5

Table 6.19. Runtimes for C_2.

degree	5	7	9	11	13	15
time(minutes)	0.12	0.35	2.5	18.4	79.3	260

Table 6.20. Runtimes for C_4.

degree	5	6	7	8	9	10
time(minutes)	1.8	4.5	51.8	138.2	932	2837.6

Table 6.21. Runtimes for C_6.

Chapter 7

Conclusion

We presented a new node elimination algorithm and demonstrated its success for cubature rules over various polytopes. By introducing the penalty term, the dependency of the initial guess is minimized, allowing to produce efficient results for higher degrees and dimensions without substantial loss of efficiency. In some cases, nearly optimal rules were attained in five and six dimensions, which is a strong indicator that optimal or nearly optimal cubature rules exist in higher dimensions.

The recursive initial guess and accurate predictor reduced the number of unknowns and the number of iterations in the Penalized Least Squares Newton, respectively. The reduction of the computational cost made the computation of five and six-dimensional problems more feasible in a reasonable amount of time. In addition, predictor was successful at selecting which node should be eliminated first. In most cases, the first node that was selected for elimination produced a new cubature rule.

The computational times were reduced further by the use of efficient linear algebra routines from OneMKL library and OpenMP parallelism. In four dimensions, all experiments took less than five hours on a 64-core Intel Xean Phi processors. For higher degrees in five and six dimensions, the longest experiments took about two days. To obtain further speedup, more parallelism should be exploited. Since computational cost is dominated by linear algebra routines, an efficient GPU implementation is the appropriate choice.

The results were obtained without any user intervention or parameter tuning. Further, the constrained optimization techniques apply to an arbitrary convex polytope. However, the remaining challenge is the need of an orthogonal basis or another domain-dependent preconditioner that would reduce the ill-conditioning of the monomial basis. Possible alternatives are nearly orthogonal bases or general preconditioners of a linear system. They provide a potential for further research on the topic. Another possible future direction is the exploration of symmetric rules over cubes and simplexes.

Appendix A

Orthogonal Polynomials

A.1. Legendre polynomials

Legendre polynomials are orthogonal with respect to $w(x) = 1$ over the interval [-1, 1], i.e. $\int_{-1}^{1} P_n(x)P_m(x) = 0$ if $n \neq m$.
The recursion formula for Legendre polynomials is

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ nP_n(x) &= (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x). \end{aligned}$$

Since our implementation is based on unit(rather than bi-unit) cubes and tetrahedra, polynomials are mapped from $[-1, 1]$ to $[0, 1]$. Therefore,

$$\hat{x} = \frac{x+1}{2}, x \in [-1, 1].$$

Hence, $\int_{0}^{1} P_n(\hat{x})P_m(\hat{x}) = 0$ if $n \neq m$.

A.2. Jacobi polynomials

Jacobi polynomials are orthogonal with respect to $w(x) = (1-x)^{\alpha}(1+x)^{\beta}$ over the interval [-1, 1], i.e. $\int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}P_n^{\alpha,\beta}(x)P_m^{\alpha,\beta}(x) = 0$ if $n \neq m$.

The recursion formula for Jacobi polynomials is

$$P_0^{\alpha,\beta}(x) = 1$$

$$P_1^{\alpha,\beta}(x) = (\alpha+1) + (\alpha+\beta+2)\frac{x-1}{2}$$

$$2n(n+\alpha+\beta)(2n+\alpha+\beta-2)P_n^{\alpha,\beta}(x) =$$

$$(2n+\alpha+\beta-1)[(2n+\alpha+\beta)(2n+\alpha+\beta-2)x+\alpha^2-\beta^2)]P_{n-1}^{\alpha,\beta}(x)-$$

$$2(n+\alpha-1)(n+\beta-1)(2n+\alpha+\beta)P_{n-2}^{\alpha,\beta}(x).$$

In our implementation, simplifications occur because $\alpha = 0$ for all Jacobi polynomials of interest. It is worth noting that Legendre polynomials are a special case of Jacobi polynomials, namely

$$L_n(x) = P_n^{0,0}.$$

Since our implementation is based on unit(rather than bi-unit) cubes and tetrahedra, polynomials are mapped $[-1,1] \to [0,1]$. Therefore,

$$\hat{x} = \frac{x+1}{2}, x \in [-1,1].$$

Hence,
$\int_0^1 (1-\hat{x})^\alpha (1+\hat{x})^\beta P_n^{\alpha,\beta}(\hat{x}) P_m^{\alpha,\beta}(\hat{x}) = 0$ if $n \neq m$.

Appendix B

Linear Inequality Constraints

Any convex polytope can be expressed as

$$\Omega = \left\{ x \in \mathbb{R}^d : Ax \leq b \right\}.$$

Constraints corresponding to two, three, and four-dimensional polytopes are listed. Derivation of constraints for their analogues in higher dimensions is identical.

B.1. Cube

A cube of dimension *dim* requires $2 * dim$ constraints. A point inside a two-dimensional unit cube(square) satisfies

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}.$$

Alternative representation is

$$0 \leq x_1 \leq 1$$

$$0 \leq x_2 \leq 1.$$

For a three-dimensional case,

$$\begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix},$$

or

$$0 \leq x_1 \leq 1$$
$$0 \leq x_2 \leq 1$$
$$0 \leq x_3 \leq 1.$$

B.2. Simplex

A simplex of dimension *dim* requires *dim*+1 constraints. A point inside a two-dimensional unit tetrahedron(triangle) satisfies

$$\begin{bmatrix} 1 & 0 \\ -1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Alternative representation is

$$0 \leq x_1 \leq x_2 \leq 1.$$

For a three-dimensional case,

$$\begin{bmatrix} -1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},$$

or

$$0 \le x_1 \le x_2 \le x_3 \le 1.$$

B.3. CubeSimplex

A tensor product domain of i-dimensional cube and j-dimensional simplex ($C_i \times T_j$) requires $2 * i + j + 1$ constraints, which is a combination of constraints for C_i and T_j. For $C_2 \times T_2$, we have

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \le \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

B.4. SimplexSimplex

A tensor product domain of i-dimensional simplex and j-dimensional simplex ($T_i \times T_j$) requires $i+j+2$ constraints, which is a combination of constraints for T_i and T_j. For $T_2 \times T_2$, we have

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \le \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

B.5. Pyramid

A three-dimensional pyramid requires six constraints, and in *dim* dimensions requires $2 * dim$ constraints. For the unit right-angle pyramid, equations are

$$\begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Alternatively, it is expressed as

$$0 \leq x_1 \leq 1$$

$$0 \leq x_2 \leq x_1$$

$$0 \leq x_3 \leq x_1.$$

Bibliography

[1] C++ core guidelines. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[2] Maneframe II. https://www.smu.edu/OIT/Services/ManeFrame.

[3] oneMKL. https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2023-0/intel-oneapi-math-kernel-library-onemkl.html.

[4] Eigen v3. https://eigen.tuxfamily.org/dox/TopicUsingIntelMKL.html, 2010.

[5] M. Ullrich A. Hinrichs, J. Prochno. The curse of dimensionality for numerical integration on general domains. *Journal of Complexity*, 50:25–42, 2019.

[6] J. Tausch A. Slobodkins. A Node Elimination algorithm for cubature of high-dimensional polytopes. https://arxiv.org/abs/2207.10737.

[7] LINDSEY PITTMAN ALFONSO CROEZE and WINNIE REYNOLDS. Solving non-linear Least-Squares problems with the Gauss-Newton and Levenberg-Marquardt methods. https://www.math.lsu.edu/system/files/MunozGroup1

[8] M. Oberlack B. Muller, F. Kummer. Highly accurate surface and volume integration on implicit domains by means of moment-fitting. *Internat. J. Numer. Methods Engrg.*, 96(8):512–528, 2013.

[9] G. Bedrosian. Shape functions and integration formulas for three-dimensional finite element analysis. *Internat. J. Numer. Methods Engrg.*, 35:95–108, 1992.

[10] R. Curtin C. Sanderson. Armadillo. https://arma.sourceforge.net/faq.html.

[11] J. Chan and T. Warburton. Orthogonal bases for vertex-mapped pyramids. *SIAM J. Sci.Comput.*, 38(2):1146–1170, 2016.

[12] P. Chen. Sparse quadrature for high-dimensional integration with Gaussian measure. *ESAIM: M2AN*, 52(2):631 – 657, 2018.

[13] A. Gurtovoy D. Abrahams. *C++ Template Metaprogramming.* 2004.

[14] A. H. Stroud D. D. Stancu. Quadrature formulas with simple Gaussian nodes and multiple fixed nodes. *JSTOR*, 17(84):384–394, 1963.

[15] D.A. Dunavant. Economical symmetrical quadrature rules for complete polynomials over a square domain. *Internat. J. Numer. Methods Engrg.*, 21(10):1777–1784, 1985.

[16] B. A. Yeager E. J. Kubatko and A. L. Maggi. New computationally efficient quadrature formulas for triangular prism elements. *Computers and Fluids*, 73:187–201, 2013.

[17] C.V. Frontin et. al. Foundations of space-time finite element methods: polytopes, interpolation, and integration, 2020. URL `https://arxiv.org/abs/2012.08701`.

[18] G. Chuluunbaatar et. al. On finite difference approximation of a matrix-vector product in the Jacobian-free Newton-Krylov method. *Comput. Math. Appl.*, 236:1399–1409, 2011.

[19] G. Chuluunbaatar et. al. PI-type fully symmetric quadrature rules on the 3-,..., 6-simplexes. *Comput. Math. Appl.*, 124:89–97, 2022.

[20] Marta D'Elia et. al. Numerical methods for nonlocal and fractional models. *Acta Numerica*, pages 1–124, 2020.

[21] V. Winschel F. Heissa. Likelihood approximation by numerical integration on sparse grids. *Journal of Econometrics*, pages 62–80, 2007.

[22] P.E. Vincent F.D. Witherden. An analysis of solution point coordinates for flux reconstruction schemes on triangular elements. *J. Sci. Comput.*, pages 1–26, 2013.

[23] J. Welsch G. Golub. Calculation of Gauss quadrature rules. 1969.

[24] W. Gautschi. How unstable are Vandermonde systems? *International Symposium on Asymptotic and Computational Analysis*, 124:193–210, 1990.

[25] A. Alexandrescu H. Sutter. *C++ Coding Standads.* 2004.

[26] J. Hesthaven. Integration preconditioning of pseudospectral operators. i. basic linear operators. *SIAM Journal on Numerical Analysis*, 35:1571–1593, 1998.

[27] D. Jespersen J. Lyness. Moderate degree symmetric quadrature rules for the triangle. *IMA Journal of Applied Mathematics*, 15(1):19–32, 1975.

[28] R. Cools J. Lyness. A survey of numerical cubature over triangles. *Proceedings of Symposia in Applied Mathematics*, 48:127–150, 1994.

[29] Wojciech Jarosz. Efficient Monte Carlo methods for light transport in scattering media. 2008.

[30] J. Jaśkowiec and N. Sukumar. High-order cubature rules for tetrahedra. *Internat. J. Numer. Methods Engrg.*, 121:2418âĂŞ2436, 2020.

[31] J. Jaśkowiec and N. Sukumar. High-order cubature rules for tetrahedra and pyramids. *Internat. J. Numer. Methods Engrg.*, 122:148–171, 2020.

[32] N. Josuttis. *C++ Move Semantics-The Complete Guide*. 2022.

[33] Violeta Karyofylli and Marek Behr. Simplex space-time meshes in engineering applications with moving domains, 2022. URL `https://arxiv.org/abs/2210.09831`.

[34] Tom Koornwinder. Two-variable analogues of the classical orthogonal polynomials. pages 435–495. 1975.

[35] Frank Ham Lee Shunn. Symmetric quadrature rules for tetrahedra based on a cubic close-packed lattice arrangement. *J. Comput. Appl. Math.*, 236(17):4348–4364, 2012.

[36] Orthogonal polynomials, simplices cubature formulae on balls, and spheres. *J. Comput. Appl. Math.*, 127:349–368, 2001.

[37] R. Poya. A look at the performance of expression templates in C++. https://romanpoya.medium.com/a-look-at-the-performance-of-expression-templates-in-c-eigen-vs-blaze-vs-fastor-vs-armadillo-vs-2474ed38d982.

[38] K. Goto R. van de Geijn. *BLAS (Basic Linear Algebra Subprograms), Encyclopedia of Parallel Computing.* 2011.

[39] H. Xiao S. E. Mousavi and N. Sukumar. Generalized gaussian quadrature rules on arbitrary polygons. *Internat. J. Numer. Methods Engrg.*, 82:99–113, 2010.

[40] H. Xiao S. Wandzura. Symmetric quadrature rules on a triangle. *Comput. Math. Appl.*, 45:1829–1840, 2003.

[41] S. Sauter and C. Schwab. *Boundary Element Methods.* Springer, 2011.

[42] S. J. Sherwin and G. E. Karniadakis. A new triangular and tetrahedral basis for high-order (hp) finite element methods. *Internat. J. Numer. Methods Engrg.*, 38:3775–3802, 1995.

[43] P. Silvester. Symmetric quadrature formulae for simplexes. *Mathematics of Computation*, 24(109):95–100, 1970.

[44] A. Slobodkins. gen-quad. https://github.com/arkslobodkins/gen-quad.

[45] S. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Mathematics Doklady 4*, pages 240–243, 1963.

[46] B. Stroustrup. *The C++ Programming Language, Fourth Edition.* 2013.

[47] Y. Sudhakar and W. A. Wall. Quadrature schemes for arbitrary convex/concave volumes and integration of weak form in enriched partition of unity methods. *Comput. Meth. Appl. Mech. Engrg.*, 258:39–54, 2013.

[48] J. Tausch. Adaptive quadrature rules for Galerkin BEM. *Computers Math. Appl.*, 113: 270–281, 2022.

[49] J. Tausch and A. Weckiewicz. Multidimensional fast Gauss transforms by Chebyshev expansions. *SIAM J. Sci.Comput.*, 31(5):3547–3565, 2009.

[50] H. Xiao and Z. Gimbutas. A numerical algorithm for the construction of efficient quadrature rules in two and higher dimensions.

[51] J.P. Moitinho de Almeida Y. Sudhakar and W. A. Wall. An accurate, robust, and easy-to-implement method for integration over arbitrary polyhedra: application to embedded interface methods. *J. Comput. Phys.*, 273:393–415, 2014.

[52] N. Yarvin and A. Rokhlin. Generalized Gaussian quadrature and singular value decompositions of integral operators. *SIAM J. Sci.Comput.*, 20(2):699–718, 1998.