Solving Poisson's equation using dataflow computing

by

Ruben van Nieuwpoort

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on
Monday December 18th, 2017 at 11:00 AM (Applied Mathematics)
Monday December 18th, 2017 at 13:30 AM (Computer Engineering)

Student number:	4424093	
Project duration:	September, 2016 – September, 2017	
Thesis committee Applied Mathematics:	Prof. dr. ir. Kees Vuik,	TU Delft
	Dr. rer nat. Matthias Möller,	TU Delft, supervisor
	Prof. dr. ir. Georgi Gaydadjiev,	TU Delft, Maxeler
	Prof. dr. ir. Hai Xiang Lin	TU Delft
Thesis committee Computer Engineering:	Dr. ir. Arjan van Genderen,	TU Delft
	Prof. dr. ir. Georgi Gaydadjiev,	TU Delft, Maxeler, supervisor
	Dr. rer nat. Matthias Möller,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Preface

Isogeometric analysis (IgA) is a concept that aims to unify computer-aided design and the finite element method (FEM), by providing a common basis for the specification of the geometric models and the numerical analysis. For the analysis of boundary value problems, one needs to assemble and solve a system of equations. For linear problems, this system is linear. and there exist many different methods exist to solve it. We can distinguish between direct methods such as the LU-decomposition, and iterative methods, like Krylov subspace methods. Some iterative methods do not require the manipulation of the coefficients of the linear system. As such, these methods can be used without ever storing the linear system in memory. These methods are known as matrix-free methods.

For IgA, the system is usually assembled by using a numerical technique called Gaussian quadrature. It is known that Gaussian quadrature is not optimal in terms of computational efficieny. When Gaussian quadrature is used, the assembly of the linear system is usually the bottleneck in the analysis. For this reason, there are many recent efforts that concentrate on finding more efficient means of numerical integration. One particularly promising development is the application of weighted quadrature for this purpose, as described in [7].

This technique is especially interesting because it is claimed that, when the linear system is assembled and stored, the bottleneck is the speed of the memory. This means that it is potentially faster to assemble the matrix ad-hoc when it is needed, than to load it from memory. In particular, iterative Krylov methods can be used for this purpose, since they only require that matrix-vector products can be computed. As such, Krylov subspace methods can be applied in a matrix-free way.

In order to benefit from such a matrix-free implementation, it is desirable to assemble the matrix in an efficient way. The process of assembly is highly parallelizable, and this can be exploited by using the concept of computing in space. In this paradigm, hardware resources on a reconfigurable chip are configured to implement logic and arithmetic operations. This has the advantage that the degree of parallelism in the assembly is potentially much higher than for implementations on central processing units (CPUs). In particular, we will use the environment and hardware for dataflow computing, which is provided by Maxeler Technologies.

The project hinges on using weighted quadrature and the implementation on a dataflow engine. The closest experts on dataflow computing were in London, England, and the experts on weighted quadrature were in Pavia, Italy. This meant that most of the time I worked in relative isolation. I did have a mailing correspondence with Mattia Tani (one of the authors of [7]) and Pavel Burovskiy (who works at Maxeler in London). Both were very helpful. However, the communication over mail is limited and sensitive to miscommunications, and most of the progress was made during the time that I visited or was visited by either Pavel or Mattia.

While the results from [7] were quickly reproduced, some testcases showed very bad convergence. Since these issues were not reported anywhere, these issues were believed to be implementation bugs. A lot of effort was spent while trying to improve the convergence. These efforts were fruitless, and it gradually became clear that this was not an implementation issue but a mathematical one. Mathematical development of weighted quadrature was not within the scope of this thesis, so this was an unfortunate situation. These issues made it necessary to settle on a dataflow implementation that is specialized for a narrow range of parameters, which limits the usefulness of the implementation.

On Tuesday June 27 2017, Mattia Tani visited Delft. He then presented the solution for the convergence problems, as well as an efficient technique for matrix-free multiplication. These were two vital ingredients for the dataflow implementation. While most of the design decisions for the dataflow implementation were already made at this point, I was still able to benefit somewhat from the more efficient matrix-free implementation. However, it would certainly be beneficial for the project if these results would have been known before the design was made.

After the meeting with Mattia, there were about two months left to incorporate the matrix-free multiplication in the design, implement the design on a dataflow engine, and write this thesis. The dataflow implementation requires a lot of esoteric work, and there always seems to be more room for improvements. Testing on a dataflow engine turned out to be a major challenge. For big problems, simulation is too slow to obtain

the convergence results in a reasonable time. Moreover, to see if a design fits, it is necessary to try to map the design to a dataflow engine. This is a time-consuming process that can take up to a day.

As a result, there was not enough time for a general, optimized implementation. The design that was made for this project still contains lots of room for improvements, and is of little practical use. However, it does show that a dataflow implementation is feasible, and can be significantly faster than a naive CPU implementation. Perhaps of more significance is the research that has been done on the application of weighted quadrature for a dataflow implementation: Novel developments are consisely described, and potential pitfals are documented. As such, it clears the way for further attempts.

This thesis is written in order to fulfill the requirements for obtaining the degree of Master of Science in Applied Mathematics and Computer Engineering. In particular, the chapters two to five are mostly written from a mathematical perspective. Chapters six and seven are written from an engineering perspective.

I would like to thank my supervisors Matthias Möller and Georgi Gaydadjiev for many fruitful discussions. Additionally, I am very grateful for the help of Mattia Tani from the University of Pavia, for the discussions about weighted quadrature, and sharing his results. I would also like to express my gratitude to Pavel Burovskiy of Maxeler Technologies, who has invested a great amount of time and was always ready to help me, with both technological and numerical issues.

Ruben van Nieuwpoort
Delft, September 2017

Contents

1 Introduction **1**
- 1.1 Scope . . . 2
- 1.2 Structure . . . 2
- 1.3 Notation . . . 3
- 1.4 List of symbols . . . 4

2 Finite elements and isogeometric analysis **7**
- 2.1 Finite elements . . . 7
  - 2.1.1 Introduction . . . 7
  - 2.1.2 Galerkin method . . . 8
  - 2.1.3 Assembly. . . . 9
- 2.2 B-spline geometry . . . 9
  - 2.2.1 Splines . . . 9
  - 2.2.2 B-splines . . . 10
  - 2.2.3 B-spline surfaces . . . 13
  - 2.2.4 Computational aspects . . . 15
- 2.3 Isogeometric analysis . . . 17
  - 2.3.1 Analysis with B-splines. . . . 17

3 State of the art in isogeometric analysis matrix assembly **23**
- 3.1 Gaussian quadrature . . . 23
- 3.2 Quadrature by interpolation . . . 24
- 3.3 Weighted quadrature . . . 25
  - 3.3.1 Extension to stiffness matrix . . . 27
  - 3.3.2 Efficient assembly . . . 29
- 3.4 Other methods . . . 32

4 Improvements on weighted quadrature **33**
- 4.1 Generalization to non-uniform open knot vectors . . . 33
  - 4.1.1 Max rule . . . 34
  - 4.1.2 Regular rule . . . 34
  - 4.1.3 Bounds on the number of quadrature points. . . . 34
  - 4.1.4 Using global interpolation . . . 35
- 4.2 Generalization to non-smooth mappings . . . 35
- 4.3 A more accurate quadrature rule . . . 38
- 4.4 Efficient matrix-free multiplication . . . 38

5 Matrix-free solution techniques **41**
- 5.1 Introduction to matrix-free methods . . . 41
- 5.2 Conjugate gradient method . . . 41
  - 5.2.1 Results . . . 42
- 5.3 Stabilized biconjugate gradient method. . . . 43
  - 5.3.1 Results . . . 44
- 5.4 Boundary conditions in matrix-free solvers . . . 45
- 5.5 Discussion . . . 46

6 Dataflow computing **49**
- 6.1 Introduction . . . 49
- 6.2 Environment . . . 51
- 6.3 Concepts . . . 52
  - 6.3.1 Kernels . . . 52

6.3.2 Streams 52
6.3.3 Managers 52
6.4 A simple example 53

7 A matrix-free dataflow implementation 55
7.1 Summary 55
7.2 Strategy for dataflow implementation 56
7.3 CPU implementation 56
7.4 Analysis of data 58
7.5 Design 59
7.5.1 BiCGSTAB 59
7.5.2 Matrix multiplication 60
7.6 Performance model 64
7.6.1 Hardware utilization 64
7.6.2 Performance 65
7.6.3 Results 66
7.7 Changes to the design 68
7.8 Build results 68

8 Conclusion 73
8.1 Future work 73
8.1.1 Improvements to the dataflow implementation 74

A Some mathematical theory 77
A.1 Multivariate calculus 77
A.2 Approximation theory 78
A.2.1 Interpolation 78
A.2.2 L^2 projection 79
A.3 Linear algebra 79

B Algorithms 81
B.1 Efficient evaluation of values and derivatives of nonzero B-spline basis functions 81
B.2 Elemental loop 82

C Code 83
C.1 Vector kernel 83
C.2 Matrix kernel 84
C.3 BiCGSTAB CPU code 85

D MAX5 hardware resources 89
D.1 System details 89
D.2 Hardware resources per arithmetic operation 89
D.2.1 27-bit fixed point 89
D.2.2 48-bit fixed point 89
D.2.3 32-bit floating point 90

Bibliography 91

1
Introduction

The finite element method (FEM) is an ubiquitous method for the analysis of boundary value problems. Specifically, it can be used to find approximations to solutions of boundary value problems on a specific domain. For industrial applications, FEM is often utilized to analyze certain properties of a geometric models. The field that deals with specifying these geometric models is called computer-aided design (CAD). The traditional method to perform analysis on the geometry defined in CAD programs, is to convert the geometric model to a analysis-suitable geometry. In general, this conversion is not exact and the analysis-suitable geometry only approximates the original geometry. For this reason, the conversion introduces an error. Additionally, the conversion is computationally expensive and needs to be done after each change to the geometry.

For these reasons, isogeometric analysis (IgA), which aims to make the representations used in CAD analysis-suitable, was introduced in [1]. In this paper, techniques were introduced which make non-uniform rational B-spline (NURBS) geometry analysis-suitable. Concretely, the geometry is made analysis-suitable by using the basis in which the geometric models are specified as a basis for analysis.

In FEM, it is necessary to assemble a linear system, whose coefficients consist of integrals computed over the domain. The domain is traditionally partitioned into so-called *elements* (and optionally, boundary segements). The integration is then performed by looping over the elements, evaluating all integrals which are nonzero on this element, and scattering the contributions to the FEM matrix . In contast to traditional FEM, the elements used in IgA are structured: they form an rectangular grid.

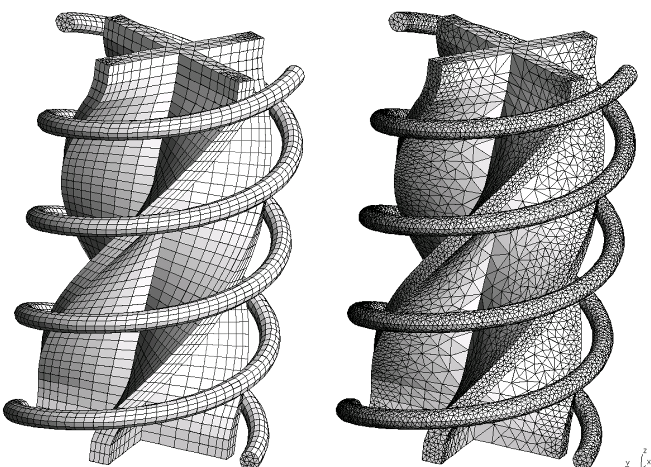


Figure 1.1: A comparison of a structured and an unstructured mesh. This image is taken from the site of the gmsh finite element software. This image was retrieved from www.geuz.org/gmsh/gallery/spirale.gif on september 5th, 2017.

Furthermore, the basis functions which are used in IgA are typically C^k-continuous for some $k \leq 1$ along elements, whereas basis functions are only C^0-continuous in classical FEM. The integrals that arise in FEM

are often approximated by using a numerical integration scheme, such as the well-known Gaussian quadrature rules.

The assembly of the linear system used to be a bottleneck for IgA. It is known that Gaussian quadrature rules, which are the most popular numerical integration scheme for IgA, are not optimal, since they do not take advantage of the higher continuity of the basis functions along the elements. So, by improving the quadrature, one might hope to solve this bottleneck. Indeed, recent work such as [7] has shown that more efficient quadrature rules exist.

In IgA, the basis functions typically have larger support than in classical FEM. The traditional approach is to use Gaussian quadrature. In this case the assembly is a bottleneck, and it is a good idea to assemble the matrix once, commit it to memory, and load the matrix whenever it is needed. However, the more efficient quadrature method proposed in [7] allows the matrix to be assembled faster than it can be loaded from memory. This opens up the possibility of a matrix-free implementation: an implementation that never stores the elements of the matrix but insteads computes the entries of the matrix as they are needed.

Of particular interest is the implementation on platforms that can take advantage of massively parallel computations. While typical CPU's are very fast, the number of operations that they can perform at once is limited. Computations that can be effectively parallelized can be sped up significantly by implementing them on a platform that allows more parallelism. Particularly, the dataflow concept is used, and the environment for dataflow computing provided by Maxeler.

1.1. Scope

Implementations on dataflow engines are typically less flexible than implementations for a CPU. The design process is very different than that for a CPU implementation. It requires one to decide which operations to perform in parallel, and which ones sequentially. Hardware resources can not be shared for operations that need to be done in parallel. Since the amount of hardware resources is limiting, the designer needs to balance the hardware resources against the degree of parallelism. It might be necessary to optimize the way in which the available resources are used to the value of a parameter to optimize the design (or make it fit on the FPGA on a dataflow engine at all).

For this project, we restrict ourselves to solving Poisson's equation for two-dimensional domains. Instead of using the more general framework of NURBS geometry, we will only use geometries defined by B-splines. This means it suffices to only use B-spline basis functions as a basis for analysis as well, which simplifies the analysis slightly.

Any additional restrictions on the dataflow implementation will be introduced and explained later in this document, in the appropriate context.

1.2. Structure

The mathematical theory behind FEM, CAD, and IgA is explained in chapter 2. Then, it is considered how weighted quadrature can be used to efficiently assemble the isogeometric FEM matrix in chapter 3. Some issues and their solutions are described in chapter 4, as well as some other developments in the theory of weighted quadrature. Some linear solvers are considered in chapter 5. Then, dataflow computing is introduced in chapter 6, and the dataflow design is described in chapter 7. Finally, the future work and conclusions are discussed in chapter 8.

1.3. Notation

Sets will be denoted with uppercase letters like V, S, U, and their closure by the same letter with a line on top: $\overline{V}, \overline{S}, \overline{U}$. The boundary of a set Ω will be denoted by $\partial\Omega$. If a set V has a finite number of elements,the number of elements will be denoted by $|V|$. The same letter might be used with different sub- or superscripts to refer to a different quantity. The notation $f(x) = O(g(x))$ will be used to indicate that there exists some $x_0 \in \mathbb{R}$ and a constant $0 < c \in \mathbb{R}$ such that $|f(x)| < c|g(x)|$ whenever $x > x_0$. The function $\delta_{i,j}$ represents the *Kronecker delta*, which is defined as

$$\delta_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

For a function $f : \Omega \to \mathbb{R}$, the support $\text{supp}(f) = \{ \mathbf{x} \in \Omega : f(\mathbf{x}) \neq 0 \}$ denotes the subset of the domain on which f assumes a nonzero value. If $f, g : \Omega \to \mathbb{R}$ satisfy $\text{supp}(f) \cap \text{supp}(g) \neq \emptyset$, we say that f and g share support. Integrals over a domain Ω will denoted with $\int_\Omega f(\xi)\, \mathrm{d}\xi$. If Ω is multi-dimensional, say $\Omega = [0,1] \times [0,1]$, this can be denoted in a number of ways: $\int_0^1 \int_0^1 f(x,y)\, \mathrm{d}x\, \mathrm{d}y = \int_\Omega f(\xi)\, \mathrm{d}\xi$. Sometimes integrals are denoted as $\int f(\xi)\, \mathrm{d}\xi$, without denoting the domain of the integral. In this case the integral is to be interpreted as an integral over the whole domain of the integrand. Similarly, the argument of a function may be omitted to simplify notation: $\int_\Omega f(\mathbf{x})\, \mathrm{d}\mathbf{x} = \int_\Omega f\, \mathrm{d}\Omega$. Summations such as $\sum_{k=0}^{N-1} a_k$ may be denoted simply as $\sum_k a_k$. This should be interpreted as a summation over all $k \in \mathbb{Z}$ for which a_k is well-defined.

If the nth derivative of a function f exist, we will say that f is C^n continuous. If the nth of f exists for every $n \in \mathbb{N}$, we say that f is C^∞-continuous or *smooth*. For the first derivative of a real-valued function f of one variable, Lagranges notation f' will be used for the first derivative, and the notation $f^{(k)}$ is used for the kth derivative. For real-valued functions f of multiple variable the notation $\frac{\partial f}{\partial x}$ is used to denote partial derivatives. The kth partial derivative with respect to x is denoted $\frac{\partial^k f}{\partial x^k}$. Further, for a function f of the variables $x_1, ..., x_n$ the gradient is defined as:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

The notation Df will be used for the $m \times n$ Jacobian matrix. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, we have:

$$Df := \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

The Jacobian matrix should be interpreted as a matrix-valued function, i.e. $Df : \mathbb{R}^n \to \mathbb{R}^{m\times n}$. We will denote $Df(\mathbf{x}) = (Df)(\mathbf{x}) \in \mathbb{R}^{m\times n}$.

Vectors will be denoted with bold-face Greek or lowercase Latin letters: $\mathbf{v}, \boldsymbol{\xi}, \boldsymbol{\Xi} \in \mathbb{R}^n$, while matrices will be represented by boldface, uppercase Latin letters: $\mathbf{M} \in \mathbb{R}^{n\times n}$. All matrices and vectors will be assumed to have only real elements. The transpose of a matrix $\mathbf{M}$ or vector $\mathbf{v}$ is denoted by a superscript $\top$, like in $\mathbf{M}^\top$, $\mathbf{v}^\top$, and the determinant of a matrix $\mathbf{M}$ is denoted by $\det(\mathbf{M})$. Vectors are interpreted as column vectors, so that the inner product of $\mathbf{u}$ and $\mathbf{v}$ can be written both as $\mathbf{u}\cdot\mathbf{v}$ and as $\mathbf{u}^\top\mathbf{v}$. The elements of a vector or matrix are denoted with a subscript indicating the index: $\mathbf{u} = (u_0, u_1, ..., u_{n-1})^\top$, $\mathbf{M}_{i,j} = \int \psi_i(\xi)\psi_j(\xi)\, \mathrm{d}\xi$. Elements of vectors are not in bold-face, so one can distinguish between a sequence of vectors $\mathbf{v}_0$, $\mathbf{v}_1$, ..., and the elements v_0, v_1 of a vector $\mathbf{v}$. The index starts at zero, which is sometimes a little awkward[1], but this simplifies implementation in programming languages which use zero-based arrays.

[1] For example, the kth element of a vector $\mathbf{v}$ is v_{k-1}, not v_k as one might expect.

1.4. List of symbols

The following symbols have a consistent meaning throughout this document. They will be introduced in the text as well, but this list can be used as a reference. The symbols are explained as they are used in this document. Specifically, it is assumed that we are dealing with the two-dimensional case.

Ω	The two-dimensional domain of the boundary value problem, which is defined as the image $\Omega = \text{Im}(\mathbf{s})$ of the mapping $\mathbf{s} : \Omega_0 \to \mathbb{R}^2$.
u	The solution $u : \Omega \to \mathbb{R}$ to the boundary value problem.
$\mathbf{s}$	The B-spline mapping $\mathbf{s} : \Omega_0 \to$ is the two-dimensional mapping that defines the domain Ω of the boundary value problem. We define $\mathbf{s}(\xi,\eta) = \sum_{i=0}^{\tilde{n}_1-1} \sum_{j=0}^{\tilde{n}_2-1} \mathbf{c_{i,j}} \tilde{M}_i(\xi)\tilde{N}_j(\eta)$
$\mathbf{c_{i,j}}$	For $i = 0,1,...,\tilde{n}_1 - 1$, $j = 0,1,...,\tilde{n}_2 - 1$, the control points $\mathbf{c_{i,j}} \in \mathbb{R}^2$, used in the B-spline mapping $\mathbf{s}$.
$\tilde{M}_0, \tilde{M}_1, ..., \tilde{M}_{\tilde{n}_1-1}$	B-spline basis functions in the first dimension, with polynomial degree $\tilde{p}_1$, and associated to the knot vector $\tilde{\mathbf{\Xi}}$, used in the B-spline mapping $\mathbf{s}$.
$\tilde{\mathbf{\Xi}}$	The knot vector $\tilde{\mathbf{\Xi}} = (\tilde{\xi}_0, \tilde{\xi}_1, ..., \tilde{\xi}_{\tilde{n}_1+\tilde{p}_1})$ defines the B-spline basis functions $\tilde{M}_0, \tilde{M}_1, ..., \tilde{M}_{\tilde{n}_1-1}$ in the first dimension.
$\tilde{N}_0, \tilde{N}_1, ..., \tilde{N}_{\tilde{n}_2-1}$	B-spline basis functions in the second dimension, with polynomial degree $\tilde{p}_2$, and associated to the knot vector $\tilde{\boldsymbol{H}}$, used in the B-spline mapping $\mathbf{s}$.
$\tilde{\boldsymbol{H}}$	The knot vector $\tilde{\boldsymbol{H}} = (\tilde{\eta}_0, \tilde{\eta}_1, ..., \tilde{\eta}_{\tilde{n}_2+\tilde{p}_2})$ defines the B-spline basis functions $\tilde{N}_0, \tilde{N}_1, ..., \tilde{N}_{\tilde{n}_2-1}$ in the second dimension.
u^h	An approximation of the solution $u : \Omega \to \mathbb{R}$ to the boundary value problem. The function $u^h : \Omega \to \mathbb{R}$ is defined as a linear combination of bivariate basis functions ψ. That is, $u^h(x,y) = \sum_{k=0}^{N} \mathbf{u}_k \psi_k(x,y)$. The vector $\mathbf{u} \in \mathbb{R}^N$ is found by solving the linear system $\mathbf{A}^* \mathbf{u} = \mathbf{b}$.
$\psi_0, \psi_1, ..., \psi_{N-1}$	The basis functions $\psi_0, \psi_1, ..., \psi_{N-1} : \Omega \to \mathbb{R}$ which are used for the finite element method, defined as $\psi_k := \phi_k \circ \mathbf{s}^{-1}$ for $k = 0, 1, ..., N-1$.
$\phi_0, \phi_1, ..., \phi_{N-1}$	The multivariate B-spline basis functions defined on the parametric domain $\Omega_0 = [0,1]^2$ as $\phi_k(\xi,\eta) := M_{k_1}(\xi) N_{k_2}(\eta)$, where $k = n_1 k_2 + k_1$.
Ω_0	The parametric space, defined as $\Omega_0 = [0,1]^2$. Integration of Ω is typically performed by a change of transformation to an integral over Ω_0, so that the regular structure can be used, and integration can be done with a square grid of quadrature points.
N	The number of degrees of freedom, or, equivalently, the number of FEM basis functions: $N = n_1 n_2$.
Q	The global, two-dimensional grid of quadrature points, which is defined by a tensor product $Q = Q^1 \times Q^2$.

n_1	The number of refined B-spline basis functions in the first dimension, which are used in the definition of the bivariate B-spline basis functions.
$M_0, M_1, ..., M_{n_1-1}$	The B-spline basis functions in the first dimension, with polynomial degree p_1, associated to the knot vector Ξ.
Ξ	The knot vector $\Xi = (\xi_0, \xi_1, ..., \xi_{n_1+p_1})$ defines the B-spline basis functions $M_0, M_1, ..., M_{n_1-1}$ in the first dimension. The knot vector Ξ is a refined version of $\tilde{\Xi}$ (see section 2.3.1).
$\xi_0^*, \xi_1^*, ..., \xi_{m^1-1}^*$	The global quadrature points in the first dimension that can be used with weighted quadrature.
$v_{k,0}^{a,b}, v_{k,1}^{a,b}, ..., v_{k,m^1-1}^{a,b}$	The weights for the quadrature points in the first dimension, that can be used with weighted quadrature and which satisfy $\sum_{s=q_j^1}^{q_j^1+m_j^1-1} M_j^{(a)}(\xi_{k,s}^*) v_{k,s}^{a,b} = \int M_j^{(a)}(\xi) M_k^{(b)}(\xi)\, \mathrm{d}\xi$.
n_2	The number of refined B-spline basis functions in the second dimension, which are used in the definition of the bivariate B-spline basis functions.
$N_0, N_1, ..., N_{n_2-1}$	The B-spline basis functions in the first dimension, with polynomial degree p_2, associated to the knot vector $\boldsymbol{H}$.
$\boldsymbol{H}$	The knot vector $\boldsymbol{H} = (\eta_0, \eta_1, ..., \eta_{n_2+p_2})$ defines the B-spline basis functions $N_0, N_1, ..., N_{n_2-1}$ in the second dimension. The knot vector $\boldsymbol{H}$ is a refined version of $\tilde{\boldsymbol{H}}$ (see section 2.3.1).
$\eta_0^*, \eta_1^*, ..., \eta_{m^2-1}^*$	The global quadrature points in the second dimension that can be used with weighted quadrature.
$w_{k,0}^{a,b}, w_{k,1}^{a,b}, ..., w_{k,m^2-1}^{a,b}$	The weights for the quadrature points in the first dimension, that can be used with weighted quadrature and which satisfy $\sum_{t=q_j^2}^{q_j^2+m_j^2-1} N_j^{(a)}(\eta_{k,t}^*) w_{k,t}^{a,b} = \int N_j^{(a)}(\eta) N_k^{(b)}(\eta)\, \mathrm{d}\eta$.

2

Finite elements and isogeometric analysis

2.1. Finite elements

2.1.1. Introduction

The finite element method is a method to solve boundary value problems. Boundary value problems pose the problem of finding a function $u : \Omega \to \mathbb{R}$ that satisfies a given differential equation on Ω and given boundary conditions on the boundary $\partial\Omega$ of the domain. We will use Poisson's problem on a two-dimensional domain with Dirichlet boundary conditions:

For a given $f \in L^2(\Omega)$, $u_{\partial\Omega} : \partial\Omega \to \mathbb{R}$, and Ω, find a function $u : \Omega \to \mathbb{R}$ such that

$$\Delta u = f \text{ on } \Omega$$

$$u = u_{\partial\Omega} \text{ on } \partial\Omega$$

The Δ represents a differential operator called the Laplace operator. For twice-differentiable functions $g : A \to \mathbb{R}$ on a domain $A \subset \mathbb{R}^d$, the Laplace operator is defined as

$$\Delta g = \sum_{i=1}^{d} \frac{\partial^2 g}{\partial x_i^2}$$

The finite element method is a numerical method. This means that the solution u is approximated by a function $u^h \approx u$. The function u^h is taken to be in some finite-dimensional *approximation space* V^h. The space V^h is spanned by a basis $\psi_0, \psi_1, ..., \psi_{N-1} : \Omega \to \mathbb{R}$. When the basis functions ψ_0, ..., ψ_{N-1} are fixed, u^h can be expressed by a vector $\mathbf{u} = (u_0, u_1, ... u_{N-1})^\top \in \mathbb{R}^N$:

$$u^h(\boldsymbol{\xi}) = \sum_{i=0}^{N-1} u_i \psi_i(\boldsymbol{\xi}) \quad \text{for } \boldsymbol{\xi} \in \Omega \tag{2.1}$$

The elements $u_0, u_1, ..., u_{N-1} \in \mathbb{R}$ are known as *degrees of freedom*.

Conceptually, the finite element method can be separated into several stages:

1. A method to convert the boundary value problem to a discrete system of symbolic equations. The most popular method is the Bubnov-Galerkin method, which is often simply called the *Galerkin method*.
2. The discretization strategy, which describes how to obtain an actual system of equations for a given symbolic system of equations. This involves picking the basis functions, and evaluating or approximating the integrals in the symbolic system. Also, one or more methods of refinement are usually provided, which can be used to increase the number of basis functions and thus the dimension of V^h, so that $u^h \in V^h$ can approximate u better.
3. The solver, which solves the system of equations. If the boundary value problem is linear, the system of equations will be linear too, and linear solvers like the conjugate gradient method, the generalized minimum-residual method, or the stabilized biconjugate gradient method are popular choices. Nonlinear systems are harder to solve, and require more advanced methods like the Newton-Rhapson method or the Picard method (see [32]). Typical nonlinear solvers iteratively linearize the system, and solve the linearized system with a linear solver.

2.1.2. Galerkin method

The Galerkin method derives a system of symbolic equations for a given boundary value problem. The Galerkin method is based on the weak formulation of the boundary value problem. First, let us define the Sobolev space $H^1(\Omega)$ of functions $f:\Omega\to\mathbb{R}$ for which the first partial derivatives are square-integrable. The weak formulation of a differential equation can be obtained by multiplying both sides by a test function $v\in V=\{\,v\in H^1(\Omega): v|_{\partial\Omega}=0\,\}$, integrating over Ω, and demanding equality regardless of the choice of $v\in V$. Applying this to $\alpha\Delta u+\beta u=f$ gives

$$\forall v\in V: \int_\Omega \alpha(\Delta u)v+\beta uv \,\mathrm{d}\Omega=\int_\Omega fv\,\mathrm{d}\Omega$$

One can then use theorem 9 from appendix A. and $v|_{\partial\Omega}=0$ to make the left-hand side symmetric:

$$\forall v\in V: \int_\Omega -\alpha(\nabla u\cdot\nabla v)+\beta uv \,\mathrm{d}\Omega=\int_\Omega fv\,\mathrm{d}\Omega$$

With this equation, the complete weak formulation becomes

For a given $f:\Omega\to\mathbb{R}$, $u_0:\partial\Omega\to\mathbb{R}$, and Ω, find a function $u\in H^1(\Omega)$ such that

$$\forall v\in V: \int_\Omega -\alpha(\nabla u\cdot\nabla v)+\beta uv \,\mathrm{d}\Omega=\int_\Omega fv\,\mathrm{d}\Omega$$

$$u=u_0 \text{ on } \partial\Omega$$

The Galerkin method replaces V by a finite-dimensional approximation V^h, spanned by basis functions $\{\psi_0,\psi_1,...,\psi_{N-1}\}$, and seeks an approximate solution $u^h\in V^h$. Ignoring the boundary conditions for now, we get the system

$$\forall i=0,1,...,N-1: \int_\Omega -\alpha(\nabla u^h\cdot\nabla\psi_i)+\beta u^h\psi_i \,\mathrm{d}\Omega=\int_\Omega f\psi_i\,\mathrm{d}\Omega$$

So $u^h\in V^h$ can be expressed as $u^h=\sum_{j=0}^{N-1}u_j\psi_j$. Using this, we get

$$\forall i=0,1,...,N-1: \int_\Omega -\alpha(\nabla(\sum_{j=0}^{N-1}u_j\psi_j)\cdot\nabla\psi_i)+\beta(\nabla(\sum_{j=0}^{N-1}u_j\psi_j)\psi_i \,\mathrm{d}\Omega=\int_\Omega f\psi_i\,\mathrm{d}\Omega$$

Bringing the summation outside the integral yields

$$\forall i=0,1,...,N-1: \sum_{j=0}^{N-1}\left(\int_\Omega -\alpha(\nabla\psi_i\cdot\nabla\psi_j)+\beta\psi_i\psi_j \,\mathrm{d}\Omega\right)u_j=\int_\Omega f\psi_i\,\mathrm{d}\Omega$$

Which can be written as a linear system $\mathbf{Au}=\mathbf{b}$ of size $N\times N$, where the *finite element matrix* $\mathbf{A}\in\mathbb{R}^{N\times N}$ is defined as

$$\mathbf{A}_{i,j}=\int_\Omega -\alpha(\nabla\psi_i\cdot\nabla\psi_j)+\beta\psi_i\psi_j\,\mathrm{d}\Omega$$

and the *right-hand side vector* $\mathbf{b}\in\mathbb{R}^N$ is defined as

$$b_i=\int_\Omega f\psi_i\,\mathrm{d}\Omega$$

Furthermore, we define the *stiffness matrix* $\mathbf{S}$ as

$$\mathbf{S}_{i,j}=\int_\Omega \nabla\psi_i\cdot\nabla\psi_j\,\mathrm{d}\Omega \tag{2.2}$$

and the *mass matrix* $\mathbf{M}$ as

$$\mathbf{M}_{i,j}=\int_\Omega \psi_i\psi_j\,\mathrm{d}\Omega \tag{2.3}$$

We can now express the finite element matrix $\mathbf{A}$ in terms of the stiffness matrix $\mathbf{S}$ and mass matrix $\mathbf{M}$ as

$$\mathbf{A}=-\alpha\mathbf{S}+\beta\mathbf{M}$$

The Dirichlet boundary conditions can be implemented by a *Dirichlet lift*. Each degree of freedom u_{i_0}, u_{i_1}, ..., $u_{i_{m-1}}$ which corresponds to a basis function which is nonzero on the boundary $\partial\Omega$ is prescribed in such a way that $\sum_{k=0}^{m-1}u_{i_k}\psi_{i_k}\approx u_{\partial\Omega}$ on $\partial\Omega$. As such, the boundary condition is approximately satisfied. To pick $u_0,u_1,...,u_{m-1}$ in such a way, one can use L^2 projection on the boundary, or make sure that u^h interpolates $u_{\partial\Omega}$ in a set of points $\mathbf{x_0}$, $\mathbf{x_1}$, ... on the boundary (see section A.2.1 and theorem 3).

2.1.3. Assembly

Suppose that the mass matrix **M** is to be assembled. One could separately evaluate the integrals $\int_\Omega \psi_i \psi_j \, \mathrm{d}\Omega$. However, the basis functions are usually defined on sections of the domain called *elements*. The domain can be partitioned into M elements $e_0, e_1, ..., e_{M-1}$ so that we have

$$\int_\Omega \psi_i \psi_j \, \mathrm{d}\Omega = \sum_{k=0}^{M-1} \int_{e_k} \psi_i \psi_j \, \mathrm{d}\Omega$$

The usual approach is to loop over the elements and evaluate $\int_{e_k} \psi_i \psi_j \, \mathrm{d}\Omega$ for the basis functions ψ_i, ψ_j which are nonzero on that element. Typically, the basis functions will have local support, and there will be a small number of nonzero basis functions on each element. For an element on which there are n nonzero basis functions $\psi_{k_0}, \psi_{k_1}, ..., \psi_{k_{n-1}}$, an *element matrix* $\mathbf{E} \in \mathbb{R}^{n \times n}$ with entries $\mathbf{E}_{i,j} = \int \psi_{k_i} \psi_{k_j} \, \mathrm{d}\Omega$ is assembled. After the element matrix is assembled, each entry $\mathbf{E}_{i,j}$ is added to $\mathbf{M}_{k_i,k_j}$ (see section B.2).

2.2. B-spline geometry

NURBS are the tool of preference for CAD modeling. NURBS are intuitive to use, and NURBS geometry is able to exactly represent conic section. The implementation of an IgA code on a dataflow engine is a proof of concept, and there is no direct need to use NURBS geometry. For this reason, only B-splines will be considered in this thesis. The theory presented can be applied to the more general case of NURBS[1].

2.2.1. Splines

Splines (or *spline functions*) are also called *piecewise polynomial*. The reason is illustrated by the following definition:

Definition 1. *A* spline *or* piecewise polynomial *function of (polynomial) degree p is a function* $f : \mathbb{R} \to \mathbb{R}$*, such that there exist* $\xi_0 < \xi_1 < ... < \xi_m$ *and polynomials* $p_0, p_1, ..., p_{m-1}$ *with degree at most p such that*

$$f(x) = \begin{cases} p_k & \text{if } x \in [\xi_k, \xi_{k+1}) \\ 0 & \text{else} \end{cases}$$

The values $\xi_0, \xi_1, ..., \xi_m$ *are called* breaks.

In order to define spline spaces, we introduce the concept of a knot vector:

Definition 2. *A* knot-vector Ξ *is a non-decreasing vector* $(\xi_0, \xi_1, ..., \xi_m)$*. The values* $\xi_0, \xi_1, ..., \xi_m$ *are called* knots*, and the number of times that a knot* ξ_k *occurs in the knot vector is called the* multiplicity μ_k *of the knot.*

We can now define a space of splines, where the continuity along the breaks is defined by the multiplicities of the knots:

Definition 3. *The* spline space $\mathbb{S}_p(\Xi)$ *of degree p for a knot vector* $\Xi = (\xi_0, \xi_1, ..., \xi_m)$ *is defined as the space of splines s of order p, with breaks* $\xi_0, \xi_1, ..., \xi_m$ *and support on a subset of* $[\xi_0, \xi_m]$*. Additionally, all splines s in the space should satisfy the continuity requirements*

$$s \text{ is } C^{p-\mu_k}\text{-continuous on } \xi_k, \text{ for } k = 0, 1, ..., m$$

To ensure that $\mathbb{S}_p(\Xi)$ is well-defined and continuous, we demand that Ξ is a knot vector of degree p.

Definition 4. *A knot vector is called a* knot vector of degree p *if all knots have a multiplicity of at most p.*

It should be noted that the requirement that s is continuous implies that $s(\xi_0) = 0$ and $\lim_{x \to \xi_m} s(x) = 0$. Sometimes, this is not desirable. To remove this restriction, the concept of an open knot vector of degree p is introduced:

Definition 5. *A knot vector is called an open knot vector of degree p when the first and last knot have multiplicity* $p+1$*, and all other knots have a multiplicity of at most p.*

[1]NURBS basis functions $N_0, N_1, ..., N_{n-1}$ can be defined based on a set of weights $w_0, w_1, ..., w_{n-1}$ and B-spline basis functions $B_0, B_1, ..., B_{n-1}$ as $N_i(\xi) = \frac{w_i B_i(\xi)}{\sum_{j=0}^{n-1} w_j B_j(\xi)}$

It should be noted that a knot vector can be an open knot vector of degree p without being a (normal) knot vector of degree p. As a standard example, we will mostly work with knot vectors where the knots are uniformly distributed on the interval $[0,1]$.

Definition 6. *A knot vector* $\Xi = (\xi_0, \xi_1, ..., \xi_m)$ *is called an* uniform knot vector *when* $\xi_k - \xi_{k-1} = c$ *for all* $k = 1,2,...,m$ *and some constant* $c > 0$. *A knot vector of the same form is called an* uniform open knot vector of degree p *when* ξ_0 *and* ξ_n *have multiplicity* $p+1$, *and* $\xi_{k+1} - \xi_k = c$ *for* $k = p+1, p+2, ..., m-p$ *and some constant* $c > 0$.

2.2.2. B-splines

B-splines[2] have a rich mathematical history (see [10] for a survey by de Boor). The recursive definition that is commonly used to introduce B-splines was presented in [11] by de Boor.

Definition 7. *For* $d \geq 1$, *a* d*-dimensional* B-spline *curve* $\mathbf{s} : \mathbb{R} \to \mathbb{R}^d$ *is defined as a linear combination of* B-spline basis functions $N_{0,p}, N_{1,p}, ..., N_{n-1,p} : \mathbb{R} \to \mathbb{R}$*:*

$$\mathbf{s}(\xi) = \sum_{i=0}^{n-1} \mathbf{c_i} N_{i,p}(\xi) \tag{2.4}$$

The image $\mathbf{s}$ *is a one-dimensional subset of* $\mathbb{R}^d$. *It is common to identify* $\mathbf{s}$ *with this subset and refer to both as a B-spline curve. The coefficients* $\mathbf{c}_0, \mathbf{c}_1, ..., \mathbf{c}_{n-1} \in \mathbb{R}^d$ *are called* control points, *and the B-spline basis functions* $N_{0,p}, N_{1,p}, ..., N_{n-1,p} : \mathbb{R} \to \mathbb{R}$ *are defined by the* Cox-de-Boor recursion formula. *The following version is from [12]:*

$$\begin{gathered} N_{i,0}(\xi) = \begin{cases} 1 & \text{if } \xi \in [\xi_i, \xi_{i+1}) \\ 0 & \text{otherwise} \end{cases} \\ N_{i,p+1}(\xi) = \alpha_{i,p}(\xi) N_{i,p}(\xi) + (1 - \alpha_{i+1,p}(\xi)) N_{i+1,p}(\xi) \\ \text{for } i = 0,1,...,n-1 \\ \text{where} \\ \alpha_{i,p}(\xi) := \begin{cases} 0 & \text{if } \xi_i = \xi_{i+p+1} \\ \frac{\xi - \xi_i}{\xi_{i+p+1} - \xi_i} & \text{otherwise} \end{cases} \end{gathered} \tag{2.5}$$

The B-spline basis functions depend on the polynomial degree p *and a* knot vector $\Xi = (\xi_0, \xi_1, ..., \xi_{n+p})$ *of degree* p. *The index* p, *denoting the polynomial degree, will often be omitted when its value if clear from the context or irrelant, so that* N_i *denotes the same basis function as* $N_{i,p}$.

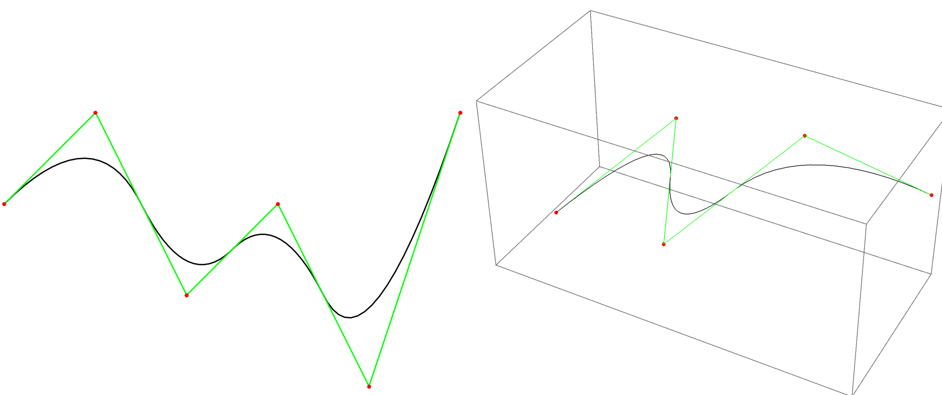


Figure 2.1: Examples of B-splines through two- and three-dimensional space.

[2] The term ‘B-splines’ is an abbreviation for ‘basis splines’. B-splines were introduced as basis functions for the spline space. However, the CAD community has adopted the term to refer to splines which are represented in terms of these basis functions. This has caught on, and it has become common to call the basis functions ‘B-spline basis functions’. This is the terminilogy that will be used in this document as well.

In figure 2.1 one can see examples of B-splines in two- and three-dimensional space. In figure 2.2, the B-spline basis functions associated to the uniform knot vector $\Xi = (0, \frac{1}{6}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{5}{6}, 1)$ for polynomials orders $p = 0, 1, 2, 3$ are shown.

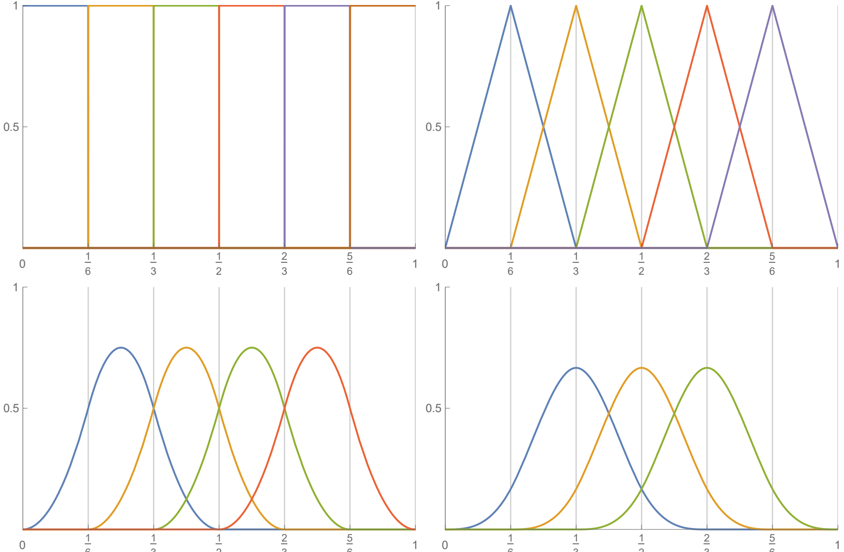


Figure 2.2: The B-spline basis functions of degree $p = 0, 1, 2, 3$ associated to the knot vector $\Xi = (0, \frac{1}{6}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{5}{6}, 1)$.

The following characterization of B-spline basis functions is due to Curry and Schoenberg in [8] and [9]:

Theorem 1 (Curry-Schoenberg)**.** *The set of B-spline basis functions $N_{p,0}, N_{p,1}, ..., N_{p,n} : \mathbb{R} \to \mathbb{R}$ associated to the knot vector $\Xi = (\xi_0, \xi_1, ..., \xi_{n+p})$ is a basis for the spline space $\mathbb{S}_p(\Xi)$. Moreover, the B-splines basis functions have* minimal support*: for any $f \in \mathbb{S}_p(\Xi)$ with $f \neq 0$ we have that there exists a basis functions N_k with supp$(N_k) \subseteq$ supp(f).*

Proof. For a proof that $N_{p,0}, N_{p,1}, ..., N_{p,n}$ is a basis of $\mathbb{S}_p(\Xi)$, see [29], chapter 1, theorem 1.8 on page 8, or [8]. For a proof that the B-spline basis functions have minimal support, see [9]. □

In CAD, it is convenient to have continuous curves that interpolate the first and last control point. This means that $\mathbf{s}(0) = \mathbf{c_0}$, and $\mathbf{s}(1) = \mathbf{c_{n-1}}$ if $\mathbf{c_{n-1}}$ is the last control point. This can be achieved by using an open knot vector to define the B-spline basis functions. The effect of taking the open knot vector $\Xi = (0, 0, 0, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1, 1, 1)$ of degree 2 is illustrated in figure 2.3.

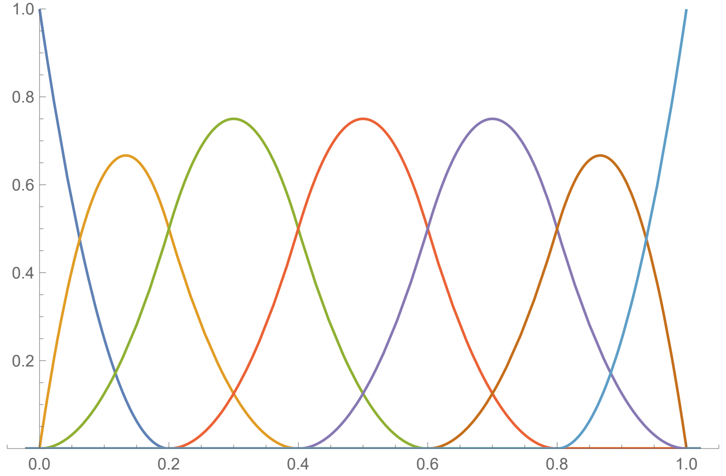


Figure 2.3: The B-spline basis functions of degree $p = 2$ associated to the open knot vector $\Xi = (0, 0, 0, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1, 1, 1)$.

It can be seen that the first and the last B-spline basis functions N_0 and N_6 are discontinuous at the first knot $\xi_0 = 0$, and the last knot $\xi_7 = 1$, since $N_0(\xi) = 0$ for $\xi < 0$, but $N_0(0) = 1$. Likewise, we have $\lim_{\xi \to 1} N_6 \xi = 1$, but $N_6(\xi) = 0$ for $\xi \geq 1$. However, since $\mathbf{s}$ is only defined on $[0,1)$, $\mathbf{s}$ is still continuous. In practice, we will extend the domain of $\mathbf{s}$ to $[0,1]$ by the following convention.

Convention 1. *Unless indicated otherwise, the knot vector $\Xi = (\xi_0, \xi_1, ..., \xi_{n+p})$ used to define B-spline basis functions of degree p, is an open knot vector that satisfies $\xi_p = 0$, $\xi_n = 1$. Moreover, the domain of B-spline curves will be taken as $[0,1]$, where the convention*

$$N_{n-1,p}(1) = 1$$

This convention ensures that B-spline curves are continuous mappings from $[0,1]$ to $\mathbb{R}^d$. In code, this is achieved by including a special check for the last basis function: If the last basis function is evaluated at the last knot of an open knot vector, it should evaluate to one (instead of zero).

Theorem 2. *Suppose that $\Xi = (\xi_0, \xi_1, ..., \xi_{n+p})$ is a (not necessarily open) knot vector of degree p. The basis functions $N_0, N_1, ..., N_{n-1}$ of order p associated to Ξ, and the B-spline curve $\mathbf{s}$ as in (2.4) with satisfy the following properties:*

1. $N_{0,p}, N_{1,p}, ..., N_{n-1,p} \in \mathbb{S}_p(\Xi)$
2. *If the derivative of a B-spline basis function $N_{i,p}$ exists at a point ξ, it is given by*

$$N'_{i,p}(\xi) = \frac{p}{\xi_{i+p} - \xi_i} N_{i,p-1} - \frac{p}{\xi_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}$$

3. *The B-spline basis functions satisfy $\sum_{i=0}^{n-1} N_i = 1$ and $\sum_{i=0}^{n-1} N'_i = 0$.*
4. *For a B-spline basis function $N_{i,p}$ we have $\overline{supp(N_{i,p})} = \overline{supp(N'_{i,p})} = [\xi_i, \xi_{i+p+1}]$. Moreover $\xi_i \in supp(N_{i,p})$ iff $p = 0$ of Ξ is an open knot vector of degree p.*
5. *$\int N_{i,p}(\xi) N_{j,p}(\xi)\, d\xi$, $\int N_{i,p}(\xi) N'_{j,p}(\xi)\, d\xi$, $\int N_{i,p}(\xi) N'_{j,p}(\xi)\, d\xi$, and $\int N'_{i,p}(\xi) N'_{j,p}(\xi)\, d\xi$ are all zero whenever $|i-j| > p$.*
6. *The B-spline basis functions satisfy $0 \leq N_{i,p} \leq 1$ and $-\frac{p}{h} \leq N'_{i,p} \leq \frac{p}{h}$, where $h := \min_{i \in \{1,2,...,n\}} \{\xi_i - \xi_{i-1} : \xi_{i-1} \neq \xi_i\}$ is the smallest nonzero difference between two consecutive knots.*
7. *On any point in the $\mathbb{R}$, there are at most $p+1$ nonzero basis functions. More specifically, there are $p+1$ nonzero basis functions on $\xi \in (\xi_p, \xi_n)$, whenever $\xi \notin \Xi$ is not a knot.*
8. *If Ξ is an open knot vector, $\mathbf{s}$ interpolates the first and last control points: $\mathbf{s}(0) = \mathbf{c}_0$, $\mathbf{s}(1) = \mathbf{c}_{n-1}$.*

Proof. A sketch of the proofs is given. The interested reader can try to prove the properties more rigorously, or look at proofs given in [11], [29] or [26]. The first property follows from theorem 1. The second property can be proved by taking the derivative of the relation (2.5). Using (2.5), it can be proved by induction on the degree p that $\sum_{i=0}^{n-1} N_{i,p}(\xi) = 1$ ($\xi = 1$ should be treated differently, since the value is defined by convention 2.2.2). By taking the derivative of both sides, it follows that $\sum_{i=0}^{n-1} N'_{i,p} = 0$, and the third property follows. The fourth property follows from (2.5) by induction (except for the case $i = n-1$, which follows from convention 2.2.2). The fifth property follows from the fourth. The sixth property follows from the second and third property. The seventh property follows from the fourth. For the eighth property, we can prove that $N_{p-j,j}(0) = 1$ for $0 \leq j \leq p$ by using (2.5) and induction on j. Likewise we have $N_{n-1,p}(1) = 1$ by convention 2.2.2. From the third property we have $\sum_{i=0}^{n-1} N_i = 1$, so that it follows that $\mathbf{s}(0) = \mathbf{c}_0$, $\mathbf{s}(1) = \mathbf{c}_{n-1}$. □

Now, the problem of finding an interpolating B-spline is considered. Suppose that we define a one-dimensional B-spline $f(\xi) = \sum_{j=0}^{n-1} f_j N_j(\xi)$ and we want to choose the control points f_0, f_1, ..., f_{n-1} such that $f(\xi_0) = y_0$, $f(\xi_1) = y_1$, ..., $f(\xi_{m-1}) = y_{m-1}$. This is the *interpolation problem*, considered in section A.2.1. We have the following theorem:

Theorem 3 (Whitney-Schoenberg)**.** *For basis functions N_{i_0}, N_{i_1}, ..., $N_{i_{n-1}}$ with $i_0 < i_1 < ... < i_{n-1}$ which are a subset of the B-spline basis functions N_0, N_1, ... associated to a not necessarily open knot vector Ξ, and interpolations points $\xi_0 < \xi_1 < ... < \xi_{m-1}$, the interpolation matrix $\mathbf{J}$ in (A.7) is nonsingular if and only if $n = m$, and*

$$N_{i_k}(\xi_k) \neq 0 \quad \textit{for } k = 0, 1, ..., n-1 \tag{2.6}$$

Proof. See [29], section 3.3, theorem 3.2 on page 19. □

Given a knot vector, it is not immediately obvious how we can choose interpolation points that satisfy the conditions (2.6) of the Whitney-Schoenberg theorem. For this purpose, one can define the *Greville abscissae.*

Definition 8. *For a knot vector $\Xi = (\xi_0, \xi_1, ..., \xi_{m-1})$ of degree p, the* Greville abscissae $x_0, x_1, ..., x_{m-p-1}$ *are defined as*

$$x_k = \frac{\xi_{k+1} + \xi_{k+2} ... + \xi_{k+p}}{p}$$

Theorem 4. *For B-spline basis functions $N_{0,p}, N_{1,p}, ..., N_{n-1,p}$ associated to an open knot vector $\Xi = (\xi_0, \xi_1, ..., \xi_{n+p})$ of degree p we have*

$$N_{k,p}(x_k) \neq 0 \quad \textit{for k = 0, 1, ..., n - 1}$$

Proof. Suppose that $\xi_k = x_k$. Then we have $\xi_k = \xi_{k+1} = ... = \xi_{k+p}$. So, the knot ξ_k has multiplicity $p+1$. If $k \neq 0, n-1$, we have that the multiplicity of ξ_{k+1} is at most p. So, we can't have $\xi_k = \xi_{k+1} = ... = \xi_{k+p}$ or $\xi_{k+1} = \xi_{k+2} = ... = \xi_{k+p+1}$ and both inequalities are strict, so we have $x_k \in (\xi_k, \xi_{k+p+1})$. By property 4, it follows that $N_k(x_k) \neq 0$.

Suppose now that $k = 0$. Then ξ_k is the first knot, and has multiplicity $p+1$, so it follows that $\xi_k = \xi_{k+1} = ... = \xi_{k+p} = 0 < \xi_{p+1}$. So it follows that $x_k = 0$, and we have $N_0(0) = 1$ by property 4. □

So, the Greville abscissae can be used as a standard choice for interpolation points for B-splines.

2.2.3. B-spline surfaces

Now that some theory on B-splines is on hands, we can consider how one can use B-splines to define surfaces, that can be used to define geometry. Earlier, we have defined B-spline curves in terms of the B-spline basis functions. In figure 2.1, B-splines through two- and three-dimensional space are shown. These curves can be obtained by increasing ξ from 0 to 1, while tracing $\mathbf{s}(\xi) \in \mathbb{R}^d$. So, the curves are a one-dimensional subset[3] of $\mathbb{R}^d$, which is the range $\{ \mathbf{s}(\xi) : \xi \in [0,1] \}$.

In CAD, one often wants to model a two-dimensional surface, or a three-dimensional volume. There are several ways to extend the notion of B-spline curves to surfaces or volumes. A simple, but effective way of doing is by defining multivariate basis functions as products of a univariate basis functions. The multivariate basis functions are said to have a *tensor product structure.* Using a tensor product structure allows one to define multivariate basis functions for an arbitrary number of dimensions. The following definition can be seen as a two-dimensional analogue for definition 7.

Definition 9. *For $d \geq 2$, a d-dimensional* B-spline *surface* $\mathbf{s} : [0,1]^2 \to \mathbb{R}^d$ *is defined as a linear combination of* bivariate basis functions $\phi_{0,0}, \phi_{1,0}, ..., \phi_{n_1-1,n_2-1} : [0,1]^2 \to \mathbb{R}$*:*

$$\mathbf{s}(\xi, \eta) = \sum_{i=0}^{n_1-1} \sum_{j=0}^{n_2-1} \mathbf{c_{i,j}} \phi_{i,j}(\xi, \eta) \tag{2.7}$$

The image of $\mathbf{s}$ *is two-dimensional subset of* $\mathbb{R}^d$*. It is common to identify* $\mathbf{s}$ *with this subset and refer to both as a* B-spline surface. *The coefficients* $\mathbf{c}_{0,0}, \mathbf{c}_{1,0}, ..., \mathbf{c}_{n_1-1,n_2-1} \in \mathbb{R}^d$ *are called* control points, *and the basis functions* $\phi_{0,0}, \phi_{1,0}, ..., \phi_{n_1-1,n_2-1} : [0,1]^2 \to \mathbb{R}$ *are defined using a* tensor product structure*:*

$$\phi_{i,j}(\xi, \eta) := M_i(\xi) N_j(\eta) \quad \textit{for } i = 0, 1, ..., n_1 - 1,\ j = 0, 1, ..., n_2 - 1 \tag{2.8}$$

where $M_0, M_1, ..., M_{n_1-1} : [0,1] \to \mathbb{R}$ and $N_0, N_1, ..., N_{n_2-1} : [0,1] \to \mathbb{R}$ are the B-spline basis functions of polynomial degree p_1, p_2, respectively, associated to the knot vectors Ξ, and $\boldsymbol{H}$, respectively.

[3]This assumes that not every control point is the same. In this case the subset would consist of a single point and thus be zero-dimensional.

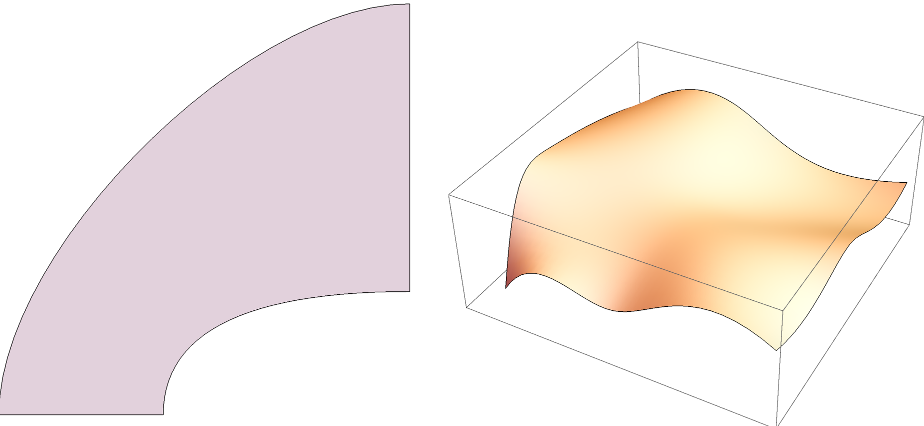


Figure 2.4: Examples of B-spline surfaces in two- and three-dimensional space.

Remark 1. *In practice, it is common to use multiple mappings to define a single geometry. Each mapping is called a* patch. *This kind of geometry is called* multi-patch geometry. *For example, if we have three patches and corresponding mapping* $\mathbf{s}_1$, $\mathbf{s}_2$, *and* $\mathbf{s}_3$, *the geometry would be given by* $Im(\mathbf{s}_1) \cup Im(\mathbf{s}_2) \cup Im(\mathbf{s}_3)$. *In this document, only single-patch geometry is used.*

From now on, we will always assume that the control points are chosen in such a way that the B-spline mapping $\mathbf{s}$ satisfies $\det(D\mathbf{s}) \neq 0$. This guarantees that the inverse mapping $\mathbf{s}^{-1}$ exists. In practice, most 'reasonable' mappings satisfy this property.

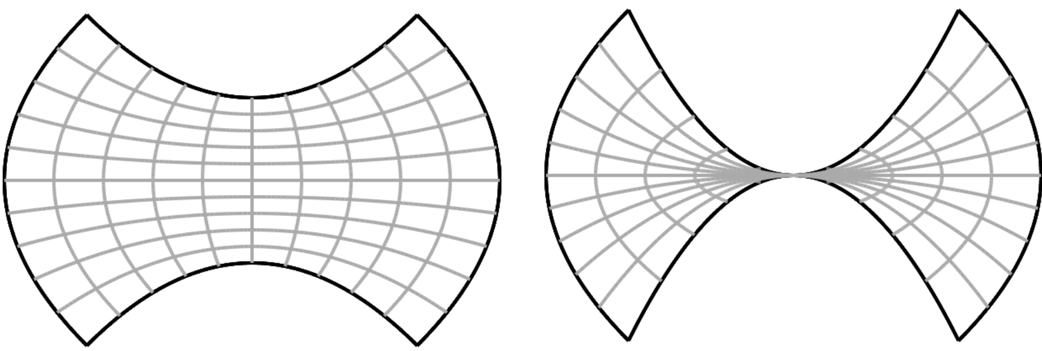


Figure 2.5: This figure shows two two-dimensional mappings. The one on the left is bijective, the one on the right is not.

Sometimes, it is desirable to enumerate the multivariate basis functions using a single index. The lexographical order will be used for this purpose: we define $N := n_1 n_2$, and $\phi_{n_1 j+i} = \phi_{i,j}$, so that the tensor product basis functions can be enumerated as $\phi_0, \phi_1, ..., \phi_{n_1 n_2 - 1}$. To be able to switch between the two notations quickly, we will use the variables $i_1 = i \pmod{n_2}$ and $i_2 = \lfloor \frac{i}{n_1} \rfloor$ to denote the indices that correspond to the univariate basis functions of ϕ_i:

$$\phi_i(\xi, \eta) = M_{i_1}(\xi) N_{i_2}(\eta)$$

As an example, suppose that we have the B-spline basis functions $M_0, M_1, ..., M_6$ and $N_0, N_1, ..., N_6$, based on the knot vectors $\Xi = \mathbf{H} = (0, 0, 0, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1, 1, 1)$ and $p_1 = p_2 = 2$. We have $n_1 = n_2 = 7$, so we have $\phi_{7j+i} = M_i(\xi) N_j(\eta)$. Figure 2.6 illustrates the relation between the bivariate basis function $\phi_{19}(\xi, \eta) = M_5(\xi) N_2(\eta)$ and the univariate basis functions M_5 and N_2.

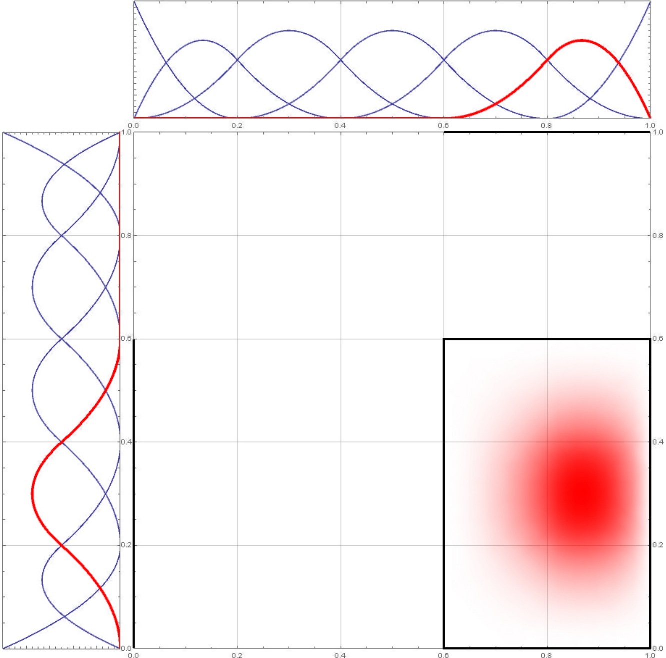


Figure 2.6: The value of the bivariate basis function $\phi_{19}(\xi,\eta) = M_5(\xi)N_2(\eta)$ is shown on the parametric space $\Omega_0 = [0,1]^2$. The parametric space is annotated with graphs of the basis functions $M_0, M_1, ..., M_6$ on top, and $N_0, N_1, ..., N_6$ on the left. The graph of M_5 and N_2 is highlighted to show the relation between the univariate basis functions and the bivariate basis function.

Many of the properties for univariate B-spline basis functions have an analogue for multivariate basis functions. We will now state some of them which will be useful later.

1. A bivariate B-spline basis function $\phi_k(\xi,\eta) = M_{k_1}(\xi)N_{k_2}(\eta)$ can only share support with other tensor product basis functions $\phi_j(\xi,\eta) = M_{j_1}(\xi)N_{j_2}(\eta)$ for which $|j_1 - k_1| \leq p_1$ and $|j_2 - k_2| \leq p_2$. Assuming that $p_1 = p_2 = p$, there are at most $(2p+1)^2$ (or, in general, $(2p+1)^d$ tensor product basis functions which share support with ϕ_k.

2. A bivariate basis function $\phi_k(\xi,\eta) = M_{k_1}(\xi)N_{k_2}(\eta)$ only has support on $[\xi_{k_1}, \xi_{k_1+p_1+1}] \times [\eta_{k_2}, \eta_{k_2+p_2+1}]$.

3. A B-spline surface is interpolatory at the corner knots: $\mathbf{s}(0,0) = \mathbf{c}_{0,0}$, $\mathbf{s}(1,0) = \mathbf{c}_{\tilde{n}_1-1,0}$, $\mathbf{s}(0,1) = \mathbf{c}_{0,\tilde{n}_2-1}$, $\mathbf{s}(1,1) = \mathbf{c}_{\tilde{n}_1-1,\tilde{n}_2-1}$.

These properties can be derived from the analogous properties for univariate B-spline basis functions.

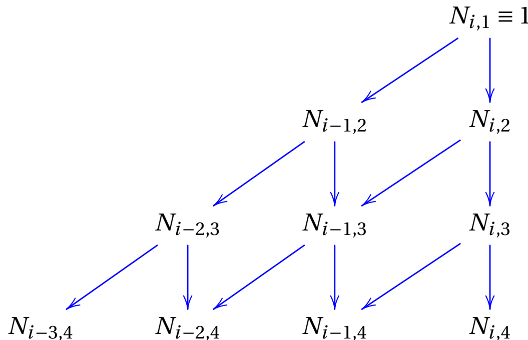
2.2.4. Computational aspects

The definition of the B-spline basis functions suggests a naive algorithm to compute the value and derivative of a B-spline basis function:

Algorithm 1 Naive evaluation of the value of a B-spline basis function

```
1: function BSPLINEBASIS(Ξ, i, p, ξ)                                  ▷ returns N_{i,p}(ξ)
2:     if p = 0 then
3:         if Ξ[i] ≤ ξ < Ξ[i+p] or Ξ[i+p] = Ξ[length(Ξ) − 1] and ξ = Ξ[i+p] then
4:             return 1
5:         else
6:             return 0
7:         end if
8:     end if
9:             return div((ξ − Ξ[i]) ∗ BSplineBasis(Ξ, i, p − 1, ξ), (Ξ[i+p] − Ξ[i]))
       +div((Ξ[i+p+1] − ξ) ∗ BSplineBasis(Ξ, i+1, p − 1, ξ), Ξ[i+p+1] − Ξ[i+1])
10: end function
11:
12: function BSPLINEBASISDERIVATIVE(Ξ, i, p, ξ)          ▷ returns the derivative N'_{i,p}(ξ) if it exists
13:     if p = 0 then
14:         return 0
15:     end if
16:     return p∗BSplineBasis(Ξ, i, p−1, ξ)/(Ξ[i+p]−Ξ[i]) - p∗BSplineBasis(Ξ, i+1, p−1, ξ)/(Ξ[i+p+1]−Ξ[i+1])
17: end function
18:
19: function DIV(numerator, denominator)
20:     if numerator = 0 then
21:         return 0
22:     end if
23:     return numerator/denominator
24: end function
```

This works, but there are more efficient algorithms available. From the definition (2.5), it can be seen that there is at most one basis function $N_{i,0}(\xi)$ with $N_{i,0}(\xi) = 1$. By starting with an array of zeroes and starting by computing the i for which $N_{i,0}(\xi) = 1$, one can compute the values $N_{i-1,1}(\xi), N_{i,1}(\xi)$, then $N_{i-2,2}(\xi), N_{i-1,2}(\xi), N_{i,2}(\xi)$, and so on, until $N_{i-p,p}(\xi), N_{i-p+1,p}(\xi), ..., N_{i,p}(\xi)$ are computed:



All computations can be done in-place by keeping just the rows in memory. So, with this algorithm, originally proposed in [11], the values of the $p+1$ basis functions can be computed:

Algorithm 2 Efficient evaluation of all values of nonzero B-spline basis functions

```
1:  function BSplineBasis(Ξ, i, p, ξ)
        ▷ returns an array with the values of all basis functions which are nonzero at this point
2:      let result be a new array of p + 1 zeroes
3:      if ξ < Ξ[0] or ξ ≥ Ξ[Ξ.length - 1] then
4:          return result
5:      end if                                              ▷ Find k such that ξ ∈ [ξ_k, ξ_{k+1})
6:      k = 0
7:      while ξ ∉ [ξ_k, ξ_{k+1} do
8:          k = k + 1
9:      end while
10:     result[0] = 1
11:     for q = 0, 1, ..., p − 1 do
12:         for j = k − q, k − q + 1, ..., k do
13:             α = (ξ − Ξ[j]) / (Ξ[j + q + 1] − Ξ[j])
14:             result[j − 1] = result[j − 1] (1 − α) * result[j]
15:             result[j] = result[j] * α
16:         end for
17:     end for
18:     return result
19: end function
```

Similar algorithms are published in [26]. The algorithm can also be adapted to calculate both all derivatives and values at once (see section B.1). This is useful for quadratures which use a global grid of quadrature points (like Gaussian quadrature, see section 3.1, and weighted quadrature, see section 3.3).

2.3. Isogeometric analysis

IgA was introduced by Hughes et al. in the seminal work [1]. Traditionally, it was necessary to convert geometry defined in a CAD program to an analysis-suitable version of the geometry. IgA aims to perform analysis directly on the geometry, without needing a conversion to another step. In other words, it aims to make the original geometry suitable for analysis, so that only one representation of the geometry is needed. IgA translates to 'analysis on the same geometry', which conveys the core idea quite nicely.

There are many different possible types of geometry that are used in CAD. Technically, IgA is an umbrella term for all techniques that use a single geometry both for design and analysis. However, the original research of Hughes et al. developed techniques to make NURBS-based geometry analysis-suitable. As such, the term 'IgA' is often used colloquially to refer to NURBS-based IgA specifically, instead of the more general isogeometric concept. More recently, there have been efforts to apply the isogeometric concept to *subdivision surfaces* (see e.g. [17]). To avoid confusion, it is usually mentioned explicitly if the geometry is not NURBS or B-spline-based (e.g. 'IgA on subdivision surfaces').

There are multiple other advantages to using IgA besides avoiding the conversion step. Since the analysis-suitable geometry is not able to represent most geometries exactly, the conversion introduces an error. For some types of analysis (shell buckling analysis, bounday layer phenomena, and analysis involving sliding contact), these errors are especially troublesome, and using IgA can yield big improvements.

2.3.1. Analysis with B-splines

An important ingredient in finite element analysis is the notion of *refinement*. Refinement is a term that is used to describe methods to increase the number of basis functions. This is useful to obtain a bigger approximation space V^h, so that the approximation $u^h \in V^h$ can approximate u better. There are different types of refinement. Here, we define *p-refinement* and *h-refinement*.

The most obvious way to introduce new basis functions is to simply insert knots. This reduces the support of the basis functions and allows for the refined basis to represent irregular functions with more precision. In this project, we will define *h-refinement* (or sometimes *knot insertion*) as adding a knot with value $\frac{\xi_k+\xi_{k+1}}{2}$ for each $\xi_k \neq \xi_{k+1}$. So, h-refinement introduces a new knot with multiplicity one in the middle of each element.

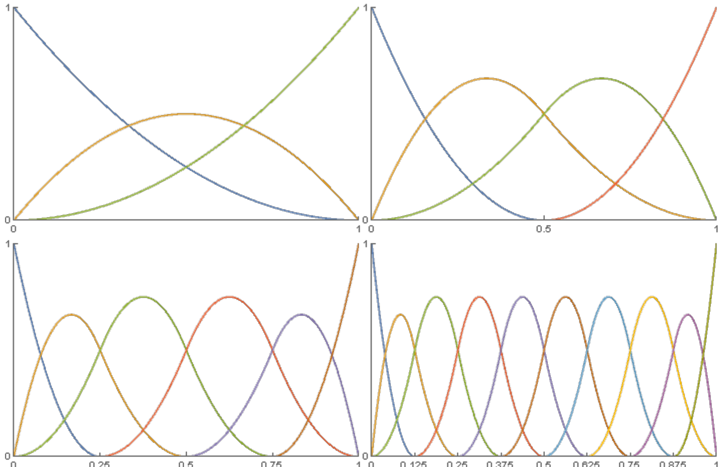


Figure 2.7: The B-spline basis functions of polynomial degree 2 are plotted for a knot vector $(0,0,0,1,1,1)$. Then, h-refinement is used to refine the knot vector to $(0,0,0,\frac{1}{2},1,1,1)$, $(0,0,0,\frac{1}{4},\frac{1}{2},\frac{3}{4},1,1,1)$, and finally $(0,0,0,\frac{1}{8},\frac{1}{4},\frac{3}{8},\frac{1}{2},\frac{5}{8},\frac{3}{4},\frac{7}{8},1,1,1)$. The corresponding basis functions are plotted.

Another way to refine the basis functions is to increase the polynomial degree. To keep the same continuity at the knots, the multiplicity of the knots should be increased. This is called *p-refinement* (or sometimes *order elevation*).

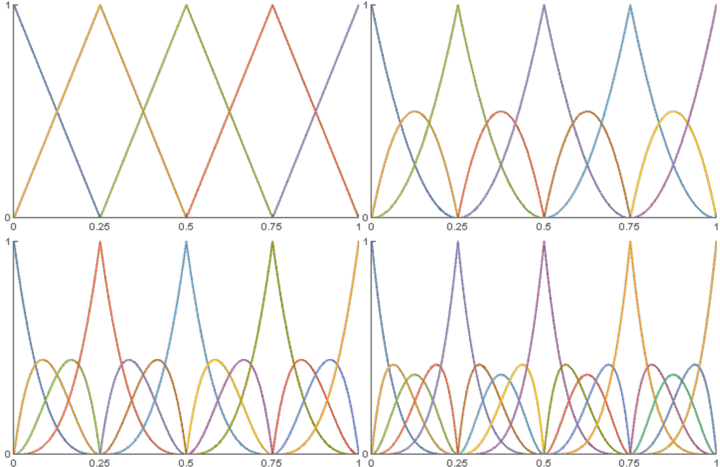


Figure 2.8: The B-spline basis functions of polynomial degree 1 are plotted for a knot vector $(0,0,\frac{1}{4},\frac{1}{2},\frac{3}{4},1,1)$. Then, p-refinement is used to refine the knot vector to $(0,0,0,\frac{1}{4},\frac{1}{4},\frac{1}{2},\frac{1}{2},\frac{3}{4},\frac{3}{4},1,1,1)$ for $p=2$, $(0,0,0,0,\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{3}{4},\frac{3}{4},\frac{3}{4},1,1,1,1)$ for $p=3$, and finally $(0,0,0,0,0,\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{3}{4},\frac{3}{4},\frac{3}{4},\frac{3}{4},1,1,1,1,1)$ for $p=4$. The corresponding basis functions are plotted.

In figures 2.7 and 2.8, the different character of h- and p-refinement can clearly been seen. Figure 2.8 also shows nicely that p-refinement preserves the continuity (in this case C^0-continuity) on the knots. In general, p-refinement reduces the error more if the solution u is smooth, while h-refinement works better if u changes rapidly. If we are working with a domain that is the range of a B-spline mapping, it is necessary to preserve the continuity at the knots. Often, the solution u is smooth. In this case, it suffices to preserve the contuinity at existing knots, and have C^{p-1} continuity at the other knots. This can be achieved by first applying p-refinement, and only then applying h-refinement.

Anticipating some developments in later chapters, we will also define a special type of refinement that lowers the continuity of the basis functions at the knots. This is obtained by keeping the same polynomial degree, but increasing the multiplicity of all internal knots of the knot vector by one. By property 1 from section 2.2.2, this will lower the continuity of the B-spline basis functions at the knots[4]: If they were C^k-continuous before, they will be C^{k-1}-continuous after the multiplicity of the knot is increased. This is not a usual refine-

[4] Or, more accurately, the continuity of all B-spline basis functions which are nonzero at a knot will be decreased at that knot.

ment strategy, but needed in some cases when weighted quadrature is used. This type of refinement will be simply referred to as *decreasing the continuity at the knots*.

It is also possible to increase the polynomial degree p without increasing the multiplicity of the inner knots. This is called *k-refinement*. Since we will want to preserve the continuity at knots, this type of refinement will not be used, and it is only mentioned for completeness.

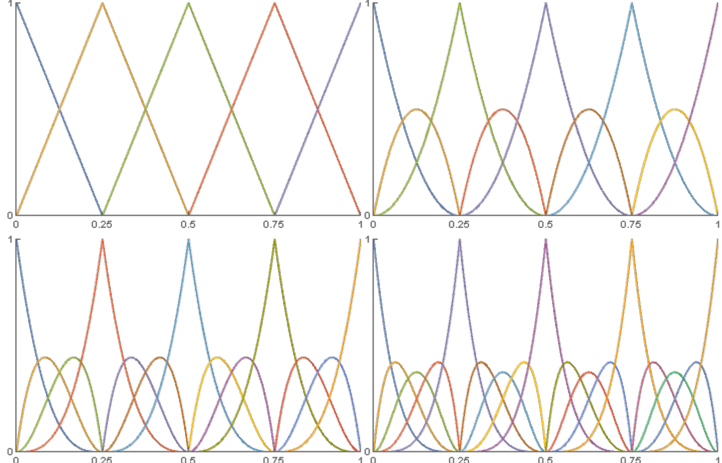
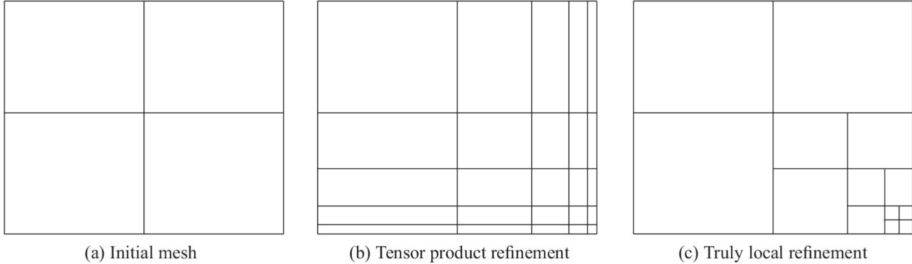


Figure 2.9: The B-spline basis functions of polynomial degree 4 are plotted for a knot vector $(0,0,0,0,0,\frac{1}{4},\frac{1}{2},\frac{3}{4},1,1,1,1,1)$. Then, the continuity is lowered at the knots to obtain the knot vectors $(0,0,0,0,0,\frac{1}{4},\frac{1}{4},\frac{1}{2},\frac{1}{2},\frac{3}{4},\frac{3}{4},1,1,1,1,1)$, $(0,0,0,0,0,\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{3}{4},\frac{3}{4},\frac{3}{4},1,1,1,1,1)$, and finally $(0,0,0,0,0,\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{3}{4},\frac{3}{4},\frac{3}{4},\frac{3}{4},1,1,1,1,1)$.

A note should be made on refinement with tensor product basis functions. Refinements can be made by refining the basis functions in one of the dimensions. Often, the complexity of geometry is not uniform, and it is desirable to do *local* refinements: refinements that only change the basis functions in certain region of the geometry. Local refinement is not possible with tensor product basis functions. If a refinement is made to one set of basis functions, the finer structure will spread across the other dimensions due to the tensor product structure. This can be seen in figure 2.10.



(a) Initial mesh (b) Tensor product refinement (c) Truly local refinement

Figure 2.10: This figure shows that when basis functions with a tensor product structure are used, local refinement is not possible

For this reason, there is interest in related techniques to represent B-spline geometries. However, not all theory for tensor product isosgeometric analysis transfers directly to these other techniques. Some work has been done to apply isogeometric analysis to alternatives to tensor product B-spline geometry, most notably to T-splines, TBH-splines, and LR-splines (see [5], [13], and [19], respectively).

Integration on B-spline geometry

Now suppose the finite element method is to be used to solve a boundary value problem on a domain Ω which is the range of a B-spline mapping $\mathbf{s} : \Omega_0 \to \Omega$.

Remark 2. *From now on, we will assume that the univariate basis functions* $\tilde{M}_0, \tilde{M}_1, \ldots, \tilde{M}_{\tilde{n}_1-1}$ *of degree* $\tilde{p}_1$ *and* $\tilde{N}_0, \tilde{N}_1, \ldots, \tilde{N}_{\tilde{n}_2-1}$ *of degree* $\tilde{p}_2$ *are the ones used in the definition of the mapping* $\mathbf{s}$*. The basis functions* M_0*,* $M_1, \ldots, M_{n_1-1}$ *of order* p_1 *and* $N_0, N_1, \ldots, N_{n_2-1}$ *of order* p_2 *are the basis functions which are used to define the bivariate basis functions* $\phi_0, \phi_1, \ldots, \phi_{N-1}$ *by means of a tensor product. These are assumed to be refined versions of the basis functions that are used in the mapping* $\mathbf{s}$*, so that* $\tilde{n}_1 \leq n_1$*,* $\tilde{p}_1 \leq p_1$*,* $\tilde{n}_2 \leq n_2$*, and* $\tilde{p}_2 \leq p_2$*.*

It is natural to use the B-spline basis functions $\phi_0, \phi_1, ..., \phi_{n-1}$ which are used in the mapping as basis functions for the finite element method. To obtain a good approximation, it might be necessary to refine the basis functions. However, these are defined on the parameteric space Ω_0, and not on the physical space Ω. A simple solution is to define the basis functions $\psi_0, \psi_1, ..., \psi_{m-1} : \Omega \to \mathbb{R}$ for the finite element method as

$$\psi = \phi \circ \mathbf{s}^{-1} \tag{2.9}$$

So, the approximation space which is used in FEM is the space spanned by the functions $\psi_0, \psi_1, ..., \psi_{N-1}$. This means that any functions in V^h can be written as a linear combination of ψ_0, ψ_1, ..., ψ_{N-1}. Specifically, we will use (2.1.1) and write the approximation $u^h \approx u$ as

$$u^h = \sum_{k=0}^{N-1} u_k \psi_k : \Omega \to \mathbb{R}$$

Substituting this in equations (2.2) and (2.1.2), using the multivariate chain rule yields the following expressions for the stiffness and mass matrix

$$\mathbf{S}_{i,j} = \int_{\Omega_0} \nabla\phi_i^{\mathrm{T}}(\xi)\mathbf{G}(\xi)\nabla\phi_j(\xi)\, \mathrm{d}\xi \tag{2.10}$$

$$\mathbf{M}_{i,j} = \int_{\Omega_0} \phi_i(\xi) g(\xi) \phi_j(\xi)\, \mathrm{d}\Omega_0 \tag{2.11}$$

where

$$\mathbf{G}(\xi) = |\det(D\mathbf{s}(\xi))|\, (D\mathbf{s}(\xi))^{-\top} (Ds(\xi))^{-1} \tag{2.12}$$

$$g = |\det D\mathbf{s}(\xi)| \tag{2.13}$$

The functions $\mathbf{G}$ and g are called the *geometric factor*. The geometric factors can be evaluated without the need to evaluate $\mathbf{s}^{-1}$ by using that $D\mathbf{s}^{-1}(\boldsymbol{\xi}) = (D\mathbf{s}(\boldsymbol{\xi}))^{-1}$. Now suppose that both Ω_0 and Ω are two-dimensional, and that s uses a tensor product structure of basis functions $\phi_i(\xi,\eta) = M_{q_i}(\xi) N_{p_i}(\eta)$. We then have

$$\phi_i(\xi,\eta) = M_{i_1}(\xi) N_{i_2}(\eta) \tag{2.14}$$

Substituting this in (2.11) yields

$$\mathbf{M}_{i,j} = \int_0^1 \int_0^1 M_{i_1}(\xi) N_{i_2}(\eta) g(\xi,\eta) M_{j_1}(\xi) N_{j_2}(\eta)\, \mathrm{d}\xi\, \mathrm{d}\eta \tag{2.15}$$

In the two-dimensional case we have $\nabla\phi_i(\xi,\eta) \in \mathbb{R}^2$ and $\mathbf{G}(\xi,\eta) \in \mathbb{R}^{2\times 2}$. Moreover, we have

$$\nabla\phi_i(\xi,\eta) = \begin{pmatrix} M'_{i_1}(\xi) N_{i_2}(\eta) \\ M_{i_1}(\xi) N'_{i_2}(\eta) \end{pmatrix}$$

$$\mathbf{G}(\xi,\eta) = \begin{pmatrix} \mathbf{G}_{0,0}(\xi,\eta) & \mathbf{G}_{1,0}(\xi,\eta) \\ \mathbf{G}_{0,1}(\xi,\eta) & \mathbf{G}_{1,1}(\xi,\eta) \end{pmatrix}$$

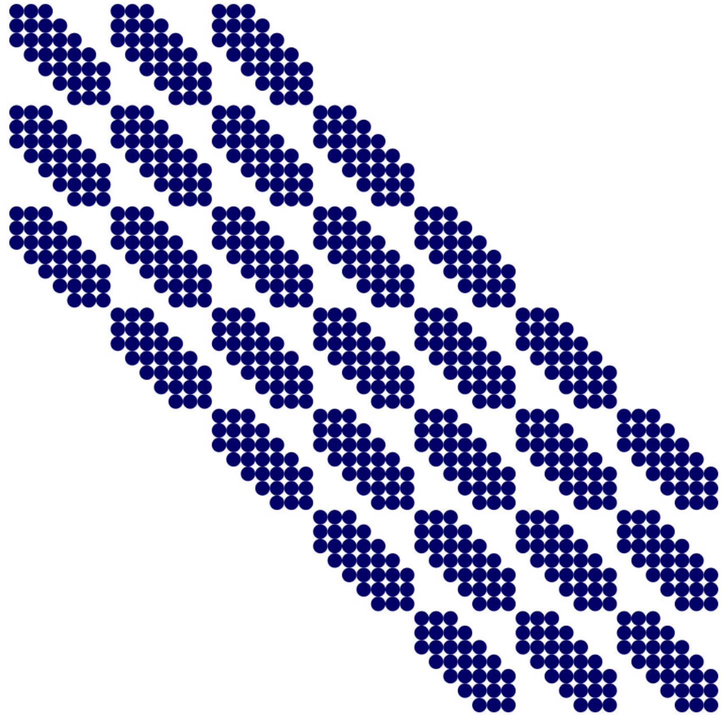
Substituting this in (2.10) gives

$$\mathbf{S}_{i,j} = \begin{array}{c} \int_0^1 \int_0^1 \mathbf{G}_{0,0}(\xi,\eta) M'_{i_1}(\xi) N_{i_2}(\eta) M'_{j_1}(\xi) N_{j_2}(\eta) \mathrm{d}\xi\, \mathrm{d}\eta \\ + \int_0^1 \int_0^1 \mathbf{G}_{0,1}(\xi,\eta) M'_{i_1}(\xi)\, N_{i_2}(\eta) M_{j_1}(\xi)\, N'_{j_2}(\eta) \mathrm{d}\xi\, \mathrm{d}\eta \\ + \int_0^1 \int_0^1 \mathbf{G}_{1,0}(\xi,\eta) M_{i_1}(\xi) N'_{i_2}(\eta) M'_{j_1}(\xi) N_{j_2}(\eta) \mathrm{d}\xi\, \mathrm{d}\eta \\ + \int_0^1 \int_0^1 \mathbf{G}_{1,1}(\xi,\eta) M_{i_1}(\xi)\, N'_{i_2}(\eta) M_{j_1}(\xi)\, N'_{j_2}(\eta) \mathrm{d}\xi\, \mathrm{d}\eta \end{array} \tag{2.16}$$

for the stiffness matrix. It can be seen that the four integrands consist of different geometric factors and combinations of derivatives, but the structure is the same in each. Moreover, the structure of the integral is also the same as for the one in the equation (2.11) for the mass matrix. Indeed the assembly of the stiffness

and mass matrix is very similar, but assembly of the stiffness matrix requires four times as much memory and computations.

Property (4) from section (2.2.2) and the ordering that we have defined on the bivariate basis functions (2.8) induce a noteworthy structure on the sparsity of both the mass and the stiffness matrix. While the exact sparsity pattern depends on the polynomial degree and knot vector of the basis functions $M_0, M_1, ...M_{n_1-1}$, $N_0, N_1, ..., N_{n_2-1}$ there is a common structure that can best be visualized. This is the two-dimensional case for polynomial degree $p = 2$, where all knot vectors have multiplicity one, and 7 basis functions are used in both dimensions:



This sparsity pattern is shared by the mass and stiffness matrix. In this two-dimensional case, there are $2p+1$ bands. Each band has width $2p+1$, so there are at most $(2p+1)^2$ nonzero elements in each row. In general, the sparsity pattern of a d-dimensional matrix can be obtained by repeating the sparsity pattern of the $(d-1)$-dimensional matrix $(2p+1)$ times, and there are at most $(2p+1)^d$ nonzero elements in each row.

The matrix can be efficiently stored by storing only the bands. For this $(2p+1)^d N$ values need to be stored, instead of the N^2 values that are required for the full matrix.

3

State of the art in isogeometric analysis matrix assembly

Quadrature rules are a simple but effictive way to approximate integrals.

Definition 10. *For a domain A and a* weight function $w : A \to \mathbb{R}$, *a* quadrature rule *or simply* quadrature *is a set of tuples* $(\xi_0^*, w_0), (\xi_1^*, w_1), ..., (\xi_{n-1}^*, w_{n-1}) \in A \times \mathbb{R}$ *such that*

$$\sum_{k=0}^{n-1} f(\xi_k^*) w_k \approx \int_A f(\xi) w(\xi)\ d\xi \tag{3.1}$$

for certain functions $f : A \to \mathbb{R}$. *If* $w \neq 1$, *the quadrature is called a* weighted quadrature. *If we have* $\sum_{k=0}^{n-1} f(\xi_k^*) w_k = \int_A f(\xi) w(\xi)\ d\xi$ *for a certain function* $f : A \to \mathbb{R}$, *the quadrature rule is said to be* exact for f. *If the quadrature is exact for every function* $f \in V$, *it is said to be exact for* V. *We will refer to* V *as the* exactness space *of the quadrature.*

When the weight function w is not mentioned, it can be assumed that $w = 1$.

Suppose that a quadrature rule is exact on a space V. In general, the quadrature is most accurate when it is applied to integrands which can be closely approximated by a function in the space V. This makes sense, since using the quadrature rule to approximate $\int f(\xi)\ \mathrm{d}\xi$ is effectively the same as approximating $\int f(\xi)\ \mathrm{d}\xi$ by $\int f^*(\xi)\ \mathrm{d}\xi$, where $f^* \in V$ interpolates f at the quadrature points.

It should be noted that if we have two-dimensional square region $I_1 \times I_2$, and there is a quadrature rule $(v_0, \xi_0^*), (v_1, \xi_1^*)..., (v_{n_1-1}, \xi_{n_1-1}^*)$ for I_1 and a quadrature rule $(w_0, \eta_0^*), w_1, \eta_1^*)..., (w_{n_2-1}, \eta_{n_2-1}^*)$ for I_2, then we can simply apply both quadrature rules[1]:

$$\sum_{i=0}^{n_1-1} \sum_{j=0}^{n_2-1} f(\xi_i^*, \eta_j^*) v_i w_j \approx \int_{I_1} \int_{I_2} f(\xi, \eta)\ \mathrm{d}\eta\ \mathrm{d}\xi$$

3.1. Gaussian quadrature

The most popular quadrature rules are *Gaussian quadrature rules.*

Theorem 5 (Gauss)**.** *For a nonempty interval I, and* $1 \le n \in \mathbb{N}$, *there exists a quadrature with n points which is exact for polynomials of degree* $p < 2n$. *This quadrature rule is called the* n-point Gaussian quadrature.

Proof. See [14], theorem 4.10 on page 58. □

It can be shown that every quadrature that is exact for the space of polynomials of degree p, needs to have at least $\lceil \frac{p-1}{2} \rceil$ quadrature points. The space of polynomials of order p and lower has dimension $p+1$. This motivates to the following definition.

[1] This method works as long as the weight function has a tensor product structure.

Definition 11. *A quadrature rule for a function space V with dimension n is called an* optimal quadrature *or a* generalized Gaussian quadrature, *if it is exact for V, and uses $\lceil \frac{n}{2} \rceil$ quadrature points.*

Consider the space $\mathbb{S}(\Xi)$, $\Xi = (0,0,0,0,\frac{1}{3},\frac{1}{3},\frac{1}{3},\frac{1}{3},\frac{2}{3},\frac{2}{3},\frac{2}{3},\frac{2}{3},1,1,1,1)$ of splines of degree 3 which are discontinuous on the knots $\frac{1}{3}$ and $\frac{2}{3}$. A function $f \in \mathbb{S}(\Xi)$. The basis functions essentially exists of three separate polynomials of degree 3, since there is no continuity along the edges. One such polynomial can be expressed as $a + bx + cx^2 + dx^3$. So, the space $\mathbb{S}(\Xi)$ has dimension 12 (one can also count the number of B-spline basis functions that are defined by Ξ). When 2-point Gaussian quadrature is used on the elements, we see that it is exact, since the splines are polynomial of order 3 on each element. Since the dimension of the space is double the number of quadrature points, Gaussian quadrature is optimal for this space.

Now consider the space $\mathbb{S}(\Xi), \Xi = (0,0,0,0,\frac{1}{3},\frac{2}{3},1,1,1,1)$ of splines of degree 3 which are C^2-continuous on the knots $\frac{1}{3}$ and $\frac{2}{3}$. There are 6 B-spline basis functions, so the dimension of the space is 6. Using 2-point Gaussian quadrature on each element is exact, but not optimal, since it needs 6 quadrature points for a space of dimension 6. Some research has been done on generating optimal quadrature rules for spline spaces (see e.g. [16], [18], [4]).

In IgA, Gaussian quadrature is often used per element. This is natural because it fits very well in the traditional framework of looping over the elements and integrating over the elements separately, and because the basis functions are typically polynomial on each element. Often, $p+1$ points are used, so that the integrals (2.11) and (2.10) in the mass and stiffness matrix can be integrated exactly when the geometric factor is constant. Of course, the geometric factor is usually not constant, so an error is introduced. In practice, this error is small enough to yield an usable numerical linear system. For a concise description of how Gaussian quadrature can be used to assembly the isogeometric FEM matrix, see [25].

3.2. Quadrature by interpolation

Suppose we have a space V with basis $v_0, v_1, ..., v_{n-1}$, and a function $f(\xi) = \sum_{j=0}^{n-1} f_j v_j(\xi) \in V$. The integral $\int_I f(\xi)\, \mathrm{d}\xi$ over some interval I can be evaluated as

$$\int_I f(\xi)\, \mathrm{d}\xi = \int_I \sum_{j=0}^{n-1} f_j v_j(\xi)\, \mathrm{d}\xi = \sum_{j=0}^{n-1} f_j \left(\int_I v_j(\xi)\, \mathrm{d}\xi \right)$$

If we define $\mathbf{f} = (f_0, f_1, ..., f_{n-1}) \in \mathbb{R}^n$ and $\mathbf{h} \in \mathbb{R}^n$ by $h_i = \int_I v_k(\xi)\, \mathrm{d}\xi$, we can write this as a dot product

$$\int_I f(\xi) \mathrm{d}\xi = \mathbf{f} \cdot \mathbf{h}$$

Now suppose $\mathbf{f}$ is chosen so that f interpolates g at $\xi_0^*, \xi_1^*, ..., \xi_{m-1}^*$. To solve for $\mathbf{f}$, we can proceed as in (A.6), and solve the linear system

$$\mathbf{Jf} = \mathbf{g}$$

where $\mathbf{J}$ is the interpolation matrix for the basis functions $v_0, v_1, ..., v_{n-1}$ and the interpolation points $\xi_0^*, \xi_1^*, ..., \xi_{m-1}^*$, and $g_i = g(\xi_i^*)$ for $i = 0, 1, ..., m-1$. It is assumed that the interpolation points are chosen such that $\mathbf{J}$ is invertible. Since $f \approx g$, we can assume that $\int_I f(\xi)\, \mathrm{d}\xi \approx \int_I g(\xi)\, \mathrm{d}\xi$ as well. So we have

$$\int_I g(\xi)\, \mathrm{d}\xi \approx \mathbf{h} \cdot \mathbf{J}^{-1} \mathbf{g}$$

Defining $\mathbf{w} := \mathbf{J}^{-\top} \mathbf{h}$ allows us to write this as

$$\int_I g(\xi)\, \mathrm{d}\xi \approx \mathbf{w} \cdot \mathbf{g} = \sum_{j=0}^{m-1} w_j g(\xi_j)$$

So we have essentially defined a quadrature rule. The weights can be computed by solving the system

$$\mathbf{J}^\top \mathbf{w} = \mathbf{h} \tag{3.2}$$

The system can be written more explicitly as

$$\forall i = 0, 1, ..., n-1 : \sum_{j=0}^{m-1} v_i(\xi_j^*) w_j = \int_I v_i(\xi)\, \mathrm{d}\xi \tag{3.3}$$

The same system arises when one simply picks a quadrature rule (ξ_0^*, w_0), (ξ_1^*, w_1), ..., (ξ_{m-1}^*, w_{m-1}), and demands that the quadrature rule is exact for all the basis functions v_0, v_1, ..., v_{n-1}. As such, these conditions are called the *exactness conditions.*

3.3. Weighted quadrature

In [7], Francesco Calabrò, Giancarlo Sangalli and Mattia Tani propose to use weighted quadrature to approximate the elements of the mass matrix:

$$\mathbf{M}_{j,k} = \int N_j(\xi) g(\xi) N_k(\xi)\, \mathrm{d}\xi \tag{3.4}$$

where $N_0, N_1, ..., N_{n-1}$ are B-spline basis functions.

In particular, this integral is then interpreted as a weighted integral $\int f(\xi) w(\xi)\, \mathrm{d}\xi$, where $f(\xi) = g(\xi)N_k(\xi)$ and $w(\xi) = N_j(\xi)$ is the weight function. The weight function is then used to define a weighted quadrature. We want to find quadrature points $\xi_0^* < \xi_1^* < ... < \xi_{m-1}^*$ and weights w_0, w_1, ..., w_{m-1} such that the quadrature can be used to approximate the integrals of the mass matrix in the following way:

$$\sum_{k=0}^{m-1} f(\xi_k) w_k \approx \int f(\xi) N_j(\xi)\, \mathrm{d}\xi \tag{3.5}$$

This is the general idea behind weighted quadrature. To evaluate all the elements of the mass matrix (3.4), it is necessary to have n different quadrature rules, obtained by setting the weight function w to any of the B-spline basis functions N_0, N_1, ..., N_{n-1}. The weights for the quadrature rule with weight function $w(\cdot) = N_j(\cdot)$ will be denoted $w_{j,0}$, $w_{j,1}$, ...

In the approach described in [7], the positions of the quadrature points are chosen beforehand. To ensure the accuracy of the quadrature, it is demanded that the quadrature rule with $w(\xi) = N_j(\xi)$ is exact for every $f \in \mathbb{S}_p(\Xi)$:

$$\forall f \in \mathbb{S}_p(\Xi) : \sum_{s=0}^{m-1} f(\xi_k) w_{j,k} = \int f(\xi) N_j(\xi)\, \mathrm{d}\xi \tag{3.6}$$

Since N_0, N_1, ..., N_{n-1} is a basis for $\mathbb{S}_p(\Xi)$, this can be formulated in an equivalent way as:

$$\forall i = 0, 1, ..., n-1 : \sum_{k=0}^{m-1} N_i(\xi_k) w_k = \int N_i(\xi) N_j(\xi)\, \mathrm{d}\xi \tag{3.7}$$

Like in quadrature by interpolation, the weights can be computed by interpreting this system as a linear system where the weights $w_{j,0}$, $w_{j,1}$, ... are the unknowns. To reduce the computational cost of evaluating the basis function and geometric factors during assembly, a global grid of quadrature points is used. That is, all quadratures use the same global grid of quadrature points $\xi_0^* < \xi_1^* < ... < \xi_{m-1}^*$. The values computed at quadrature points can then be re-used for different quadratures.

To reduce the size of the linear system (3.7), the weights of all quadrature points outside the support of N_j are set to zero. We define the set of G_j of indices of quadrature points in the support of N_j as $G_j := \{\, s : \xi_s^* \in \mathrm{supp}(N_j) \,\}$. The quadrature points in the domain of N_j will be called *active* for N_j. In this way, the exactness conditions are automatically satisfied for basis functions N_i for which $\int N_i(\xi)N_j(\xi)\, \mathrm{d}\xi = 0$. We denote the set indices of basis functions N_i for which $\int N_i(\xi)N_j(\xi)\, \mathrm{d}\xi \neq 0$ as $F_j := \{\, i : \int N_i(\xi)N_j(\xi)\, \mathrm{d}\xi \neq 0 \,\}$. Now, we can find weights that satisfy the exactness conditions (3.7) by using a reduced linear system to compute the weights:

$$\forall i \in F_j : \sum_{s \in G_j}^{m-1} N_i(\xi_s) w_s = \int N_i(\xi) N_j(\xi)\, \mathrm{d}\xi$$

In [7], it is stated without proof that the linear system has a solution when

$$|F_j| \leq |G_j| \tag{3.8}$$

for every basis function N_j. That is, for every basis function N_j, there need to be at least as many quadrature points in its support as there are basis functions that share support with N_j.

Note that we still have not defined how the weights should be picked. In [7] only the case of open knot vectors where all internal knots have multiplicity one is considered. For these knot vectors, it is proposed by a rule that we will call the *midpoint rule.* This rule states that quadrature points should be taken in the

middle of each knot span, and on the knots. The first and last knot span are an exception: there should be p quadrature points uniformly distributed knots on these knot spans (in addition to the quadrature points on the knots which are the boundaries of these knot spans). Let us consider an illustrative example.

Example
The technique is now illustrated at the hand of an example. Take $p = 2$, $\Xi = (0,0,0,\frac{1}{4},\frac{1}{2},\frac{3}{4},1,1,1)$. The basis functions look as follows:

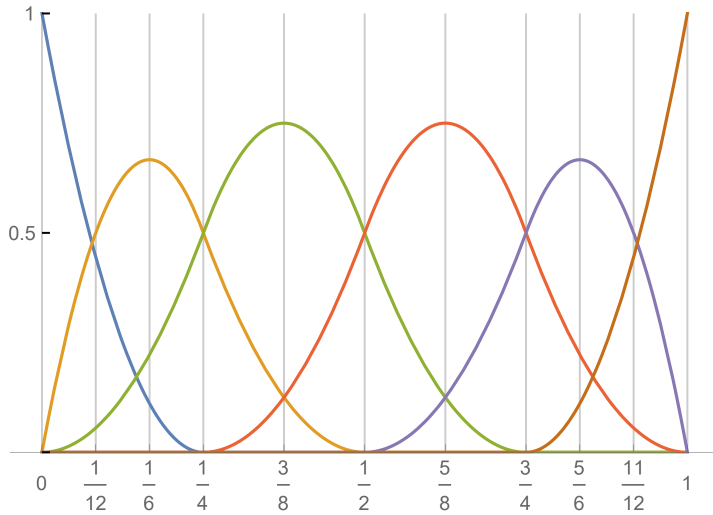


Figure 3.1: The basis functions N_0, N_1, N_2, N_3, N_4, for $p = 2$ and $\Xi = (0,0,0,\frac{1}{4},\frac{1}{2},\frac{3}{4},1,1,1)$.

Applying the described strategy of picking quadrature points, the global grid G of quadrature points is

$$(0,\frac{1}{12},\frac{1}{6},\frac{1}{4},\frac{3}{8},\frac{1}{2},\frac{5}{8},\frac{3}{4},\frac{5}{6},\frac{11}{12},1)$$

The vertical lines correspond to the position of quadrature points. The elements defined by the knot vector are $[0,\frac{1}{4})$, $[\frac{1}{4},\frac{1}{2})$, $[\frac{1}{2},\frac{3}{4})$, and $[\frac{3}{4},1)$. Using the proposed method, we define $p+1 = 3$ regularly spaced knots on the first element: $\xi_0^* = 0$, $\xi_1^* = \frac{1}{12}$, $\xi_2^* = \frac{1}{6}$. For the elements $[\frac{1}{4},\frac{1}{2})$ and $[\frac{1}{2},\frac{3}{4})$, quadrature points are taken to be in the middle and on the edges of these elements. We see that $\xi_3^* = \frac{1}{4}$, $\xi_4^* = \frac{3}{8}$, $\xi_5^* = \frac{1}{2}$, $\xi_6^* = \frac{5}{8}$, $\xi_7^* = \frac{3}{4}$. Then, there are $p+1$ quadrature points on the last element, so $\xi_8^* = \frac{5}{6}$, $\xi_9^* = \frac{11}{12}$, $\xi_1^*0 = 1$.

Now, suppose we want to compute the quadrature $(w_{1,0},\xi_0^*),(w_{1,1},\xi_1^*),\ldots,(w_{1,11},\xi_{11}^*)$ for N_1. The exactness conditions are:

$$\begin{aligned}
\textstyle\sum_{s=0}^{11} w_{1,s}N_0(\xi_s^*) &= \textstyle\int_0^1 N_0(\xi)N_1(\xi)\,\mathrm{d}\xi\\
\textstyle\sum_{s=0}^{11} w_{1,s}N_1(\xi_s^*) &= \textstyle\int_0^1 N_1(\xi)N_1(\xi)\,\mathrm{d}\xi\\
\textstyle\sum_{s=0}^{11} w_{1,s}N_2(\xi_s^*) &= \textstyle\int_0^1 N_2(\xi)N_1(\xi)\,\mathrm{d}\xi\\
\textstyle\sum_{s=0}^{11} w_{1,s}N_3(\xi_s^*) &= \textstyle\int_0^1 N_3(\xi)N_1(\xi)\,\mathrm{d}\xi\\
\textstyle\sum_{s=0}^{11} w_{1,s}N_4(\xi_s^*) &= \textstyle\int_0^1 N_4(\xi)N_1(\xi)\,\mathrm{d}\xi\\
\textstyle\sum_{s=0}^{11} w_{1,s}N_5(\xi_s^*) &= \textstyle\int_0^1 N_5(\xi)N_1(\xi)\,\mathrm{d}\xi
\end{aligned}$$

N_1 only shares support with N_0, N_1, N_2, N_3, so $F_1 := \{0,1,2,3\}$. The support of N_1 is $(0,\frac{1}{2})$. Since we have $w_{1,k} = 0$ whenever $\xi_k^* \not\in \operatorname{supp}(N_1) = (0,\frac{1}{2})$, we see that only quadrature points $\xi_1^*,\xi_2^*,\xi_3^*,\xi_4^*$ are active, so $G_1 = \{1,2,3,4\}$. The other weights are set to zero and can be ignored in the summation. Now, the condition 3.3 is satisfied for functions which do not share support with N_1, since both the right-hand side and the left-hand side will be zero. So we are left with the system:

$$\begin{pmatrix} N_0(\xi_1^*) & N_0(\xi_2^*) & N_0(\xi_3^*) & N_0(\xi_4^*)\\ N_1(\xi_1^*) & N_1(\xi_2^*) & N_1(\xi_3^*) & N_1(\xi_4^*)\\ N_2(\xi_1^*) & N_2(\xi_2^*) & N_2(\xi_3^*) & N_2(\xi_4^*)\\ N_3(\xi_1^*) & N_3(\xi_2^*) & N_3(\xi_3^*) & N_3(\xi_4^*) \end{pmatrix} \begin{pmatrix} w_{1,1}\\ w_{1,2}\\ w_{1,3}\\ w_{1,4} \end{pmatrix} = \begin{pmatrix} \int_0^1 N_0(\xi)N_1(\xi)\,\mathrm{d}\xi\\ \int_0^1 N_1(\xi)N_1(\xi)\,\mathrm{d}\xi\\ \int_0^1 N_2(\xi)N_1(\xi)\,\mathrm{d}\xi\\ \int_0^1 N_3(\xi)N_1(\xi)\,\mathrm{d}\xi \end{pmatrix}$$

Now, the B-spline basis functions in the matrix can be evaluated. Likewise, one can evaluate the integrals on the right-hand side by using the Gaussian quadrature rule with $p+1=3$ quadrature points on each element. We see that $\int_0^1 N_0(\xi)N_1(\xi)\,\mathrm{d}\xi = \frac{7}{240}$, $\int_0^1 N_1(\xi)N_1(\xi)\,\mathrm{d}\xi = \frac{1}{12}$, $\int_0^1 N_2(\xi)N_1(\xi)\,\mathrm{d}\xi = \frac{5}{96}$, and $\int_0^1 N_3(\xi)N_1(\xi)\,\mathrm{d}\xi = \frac{1}{480}$. We get the system:

$$\begin{pmatrix} \frac{4}{9} & \frac{1}{9} & 0 & 0 \\ \frac{1}{2} & \frac{2}{3} & \frac{1}{2} & \frac{1}{8} \\ \frac{1}{18} & \frac{2}{9} & \frac{1}{2} & \frac{3}{4} \\ 0 & 0 & 0 & \frac{1}{8} \end{pmatrix} \begin{pmatrix} w_{1,1} \\ w_{1,2} \\ w_{1,3} \\ w_{1,4} \end{pmatrix} = \begin{pmatrix} \frac{7}{240} \\ \frac{1}{12} \\ \frac{5}{96} \\ \frac{1}{480} \end{pmatrix}$$

Solving this system yields $w_{1,1} = \frac{9}{160}$, $w_{1,2} = \frac{3}{80}$, $w_{1,3} = \frac{9}{160}$, $w_{1,4} = \frac{1}{60}$.

In general, the linear system that is obtained this way does not need to be square, but by (3.8), it must have at least as many unknown as equations. This underdetermined system can be solved with a singular value decomposition, a QR-decomposition, or by solving the normal equations (see section A.3).

3.3.1. Extension to stiffness matrix

For assembling the stiffness matrix for a two-dimensional problem, integrals of the form $\int N_i(\xi)N_j'(\xi)\,\mathrm{d}\xi$, $\int N_i'(\xi)N_j(\xi)\,\mathrm{d}\xi$, and $\int N_i'(\xi)N_j'(\xi)\,\mathrm{d}\xi$ need to be assembled as well. To compute quadrature rules for integrals of the form $\int N_i(\xi)N_j'(\xi)\,\mathrm{d}\xi$, only the weight function $w = N_i'$ is different. Again, the quadrature points in the global grid G will be used, so only the right-hand side of the linear system is different. From now on, we will use the notation $w_{k,0}^{a,b}$, $w_{k,1}^{a,b}$, ... for weights such that:

$$\sum_{k\in G_j} N_i^{(a)}(\xi_k^*)w_{j,k}^{a,b} = \int N_i^{(a)}(\xi)N_j^{(b)}(\xi)\,\mathrm{d}\xi$$

So, the weights which were referred to as $w_{j,0}$, $w_{j,1}$, ... before, will be called $w_{j,0}^{0,0}$, $w_{j,1}^{0,0}$, ... from now on. For the weights $w_{j,0}^{0,1}$, $w_{j,1}^{0,1}$, ... we have the weight function $w = N_j'$, and we find the system

$$\forall i \in F_j : \sum_{k=0}^{m-1} N_i(\xi_k^*)w_j = \int_I N_i(\xi)N_j'(\xi)\,\mathrm{d}\xi \tag{3.9}$$

The only difference with (3.7) is the right-hand side. For the weights $w^{a,b}$ with $a=1$, which are used for the quadrature rules for integrals of the form $\int N_k'(\xi)N_j^{(b)}(\xi)\,\mathrm{d}\xi$, we have the exactness conditions

$$\forall i \in F_j : \sum_{k\in G_j} N_j^{(b)}(\xi_i)w_i = \int_I N_j^{(b)}(\xi)N_k'(\xi)\,\mathrm{d}\xi \tag{3.10}$$

The most logical way to proceed is to mimic the example and use this as a linear system. The same example as in the last section is used, but the weights $w_{1,1}^{1,0}$, $w_{1,2}^{1,0}$, ..., $w_{1,4}^{1,0}$ are computed instead of $w_{1,1}^{0,0}$, $w_{1,2}^{0,0}$, ..., $w_{1,4}^{0,0}$. We have $p=2$, $\Xi = (0,0,0,\frac{1}{4},\frac{1}{2},\frac{3}{4},1,1,1)$ and the global grid of quadrature points $(0,\frac{1}{12},\frac{1}{6},\frac{1}{4},\frac{3}{8},\frac{1}{2},\frac{5}{8},\frac{3}{4},\frac{5}{6},\frac{11}{12},1)$. The exactness conditions are now imposed on the derivatives of the basis functions N_0', N_1', ..., N_5':

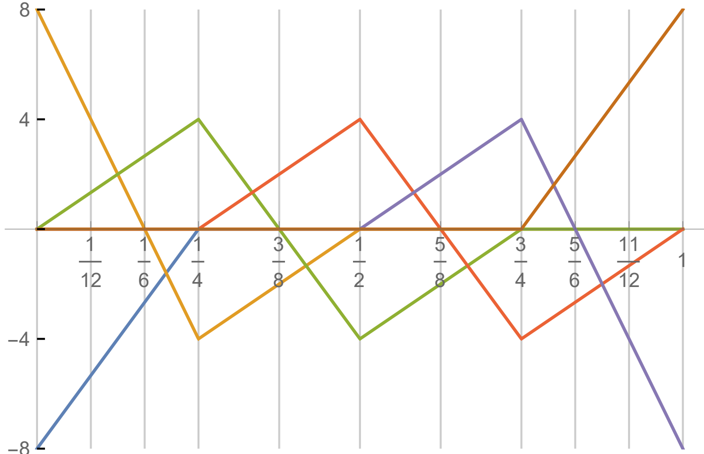


Figure 3.2: The derivatives N_0', N_1', N_2', N_3', N_4' of the basis functions, for $p = 2$ and $\Xi = (0,0,0,\frac{1}{4},\frac{1}{2},\frac{3}{4},1,1,1)$.

The linear system is

$$\begin{aligned}\sum_{s=1}^{4} w_{1,s}^{1,0} N_0'(\xi_s^*) &= \int_0^1 N_0'(\xi) N_1(\xi)\,\mathrm{d}\xi\\ \sum_{s=1}^{4} w_{1,s}^{1,0} N_1'(\xi_s^*) &= \int_0^1 N_1'(\xi) N_1(\xi)\,\mathrm{d}\xi\\ \sum_{s=1}^{4} w_{1,s}^{1,0} N_2'(\xi_s^*) &= \int_0^1 N_2'(\xi) N_1(\xi)\,\mathrm{d}\xi\\ \sum_{s=1}^{4} w_{1,s}^{1,0} N_3'(\xi_s^*) &= \int_0^1 N_3'(\xi) N_1(\xi)\,\mathrm{d}\xi\end{aligned} \tag{3.11}$$

Solving this system by using Gaussian elimination or QR-decomposition often works. However, in some cases, divisions by zero occur, and invalid weights are computed. To see why this happens, it helps to look at the system that is generated in this simple case. Using $\xi_1^* = \frac{1}{12}$, $\xi_2^* = \frac{1}{6}$, $\xi_3^* = \frac{1}{4}$, $\xi_4^* = \frac{3}{8}$, and evaluating all basis functions and integrals, we get

$$\begin{pmatrix} -\frac{16}{3} & -\frac{8}{3} & 0 & 0\\ 4 & 0 & -4 & -2\\ \frac{4}{3} & \frac{8}{3} & 4 & 0\\ 0 & 0 & 0 & 2\end{pmatrix}\begin{pmatrix} w_{1,1}^{1,0}\\ w_{1,2}^{1,0}\\ w_{1,3}^{1,0}\\ w_{1,4}^{1,0}\end{pmatrix} = \begin{pmatrix} -\frac{5}{12}\\ 0\\ \frac{3}{8}\\ \frac{1}{24}\end{pmatrix}$$

This is a singular matrix, since the rows add up to zero. The reason that solving the singular system usually works fine is because the numerical evaluation usually introduces a small error, which makes the system nonsingular. This way, the linear system can still be solved, even for methods that are not supposed to work for singular matrices. Of course, this behavior can not be relied on. One solution is to use a method that is suitable for dealing with singular linear systems. One can, for example, use a singular value decomposition to find a solution $\mathbf{x}$ for which $||\mathbf{Ax}-\mathbf{b}||_{l^2}$ is minimal. However, it is not necessary to introduce such a complicated and computationally expensive step. Instead, it might be better to avoid ending up with a singular system.

The derivatives of the basis functions $N_{0,2}, N_{1,2}, \ldots$ are linear, and in figure 3.2 it can be seen that not only the rows are linearly dependent, but the derivatives N_0', N_1', ..., N_5 of the basis functions are linearly dependent. This suggests that the exactness conditions should not be imposed on N_0', N_1', ..., N_5' but instead for a basis of this space. We will use the following theorem.

Theorem 6. *Let Ξ be an open knot vector of order p, and $N_{0,p}$, $N_{1,p}$, ..., $N_{n-1,p}$ be the B-spline basis functions that are associated to this knot vector. consider the space that is spanned by the derivates $N_{0,p}$, $N_{1,p}$, ..., $N_{n-1,p}$. This space has dimension $n-1$, so the derivates $N_{0,p}$, $N_{1,p}$, ..., $N_{n-1,p}$ themselves can not be a basis for this space. However, the B-spline basis functions $N_{1,p-1}$, $N_{2,p-1}$, ..., $N_{n-1,p-1}$ of order $p-1$ are a basis for this space.*

Proof. By property (2) from section (2.2.2), it can be seen that $\mathbb{S}_{p-1}(\Xi)$ is a basis for the space of derivatives of $\mathbb{S}_{(}\Xi)$. □

Remark 3. *Notice that the first basis function $N_{0,p-1}$ is not in the basis, since it is zero everywhere.*

Let's apply this to the example. Instead of using the derivatives $N'_{0,2}$, $N'_{1,2}$, ..., $N'_{5,2}$, we use the basis functions $N_{1,1}$, $N_{2,1}$, ..., $N_{5,1}$.

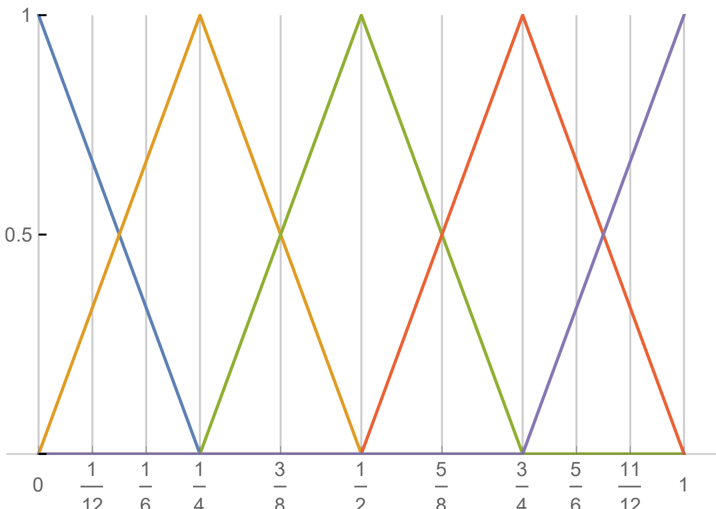


Figure 3.3: The basis functions $N_{1,1}$, $N_{2,1}$, $N_{3,1}$, $N_{4,1}$, $N_{5,1}$ for $\Xi = (0,0,0,\frac{1}{4},\frac{1}{2},\frac{3}{4},1,1,1)$.

Now, we take all the quadrature points in the domain of N_1 as active, and demand exactness for all basis functions of degree one which have support on these quadrature points. This gives the exactness conditions

$$\begin{aligned} \textstyle\sum_{s=1}^{4} w_{1,s}^{1,0} N_{0,1}(\xi_s^*) &= \textstyle\int_0^1 N_{0,1}(\xi) N_{1,2}(\xi)\, \mathrm{d}\xi \\ \textstyle\sum_{s=1}^{4} w_{1,s}^{1,0} N_{1,1}(\xi_s^*) &= \textstyle\int_0^1 N_{1,1}(\xi) N_{1,2}(\xi)\, \mathrm{d}\xi \\ \textstyle\sum_{s=1}^{4} w_{1,s}^{1,0} N_{2,1}(\xi_s^*) &= \textstyle\int_0^1 N_{2,1}(\xi) N_{1,2}(\xi)\, \mathrm{d}\xi \end{aligned}$$

Evaluating this gives the linear system

$$\begin{pmatrix} \frac{2}{3} & \frac{1}{3} & 0 & 0 \\ \frac{1}{3} & \frac{2}{3} & 1 & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} w_{1,1}^{1,0} \\ w_{1,2}^{1,0} \\ w_{1,3}^{1,0} \\ w_{1,4}^{1,0} \end{pmatrix} = \begin{pmatrix} \frac{5}{96} \\ \frac{5}{48} \\ \frac{1}{96} \end{pmatrix}$$

This linear system is underdetermined. A solution $\mathbf{x}$ with minimal l^2 norm $||\mathbf{x}||_{l^2}$ can be found by solving $\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{b}$, using a QR-decomposition or singular value decomposition (see theorem 13 and 14 in appendix A). Using one of these techniques, we can find $w_{1,1}^{1,0} = \frac{5}{96}$, $w_{1,1}^{1,0} = \frac{5}{48}$, $w_{1,3}^{1,0} = 0$, $w_{1,1}^{1,0} = \frac{1}{96}$.

3.3.2. Efficient assembly

The quadrature rule that is used is just one element of the assembly. While weighted quadrature uses a lot less quadrature points than Gaussian quadrature, it does not achieve the best performance when an elemental loop is used.

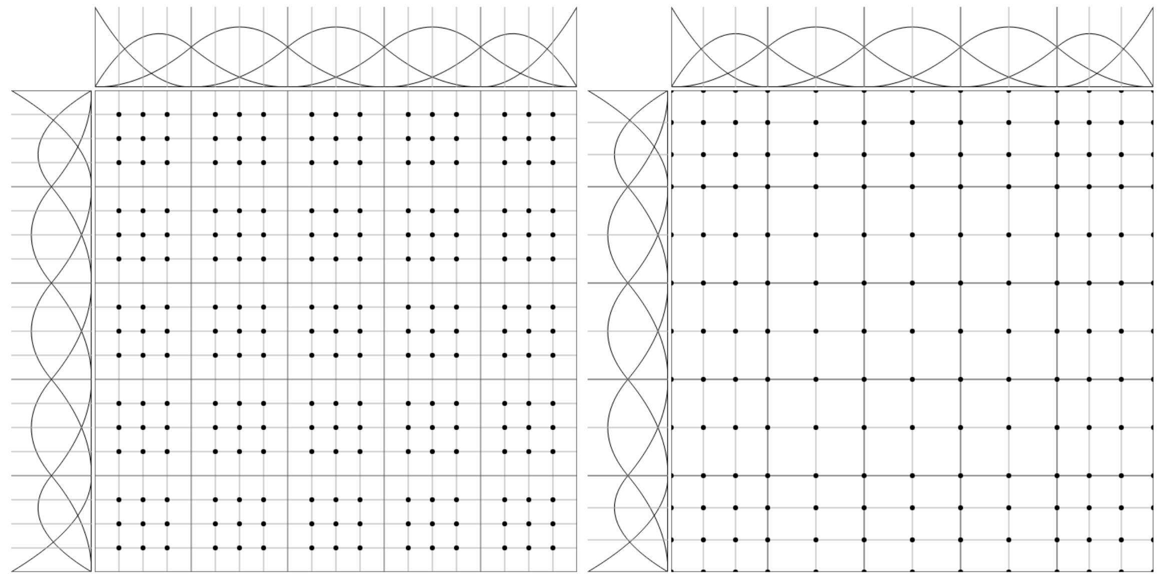


Figure 3.4: On the left, a schematic representation of the parameteric domain with the quadrature points that are used for Gauss quadrature for basis functions of degree $p = 2$. On the right, the parameteric domain with the quadrature points that are used for weighted quadrature for basis functions of degree $p = 2$.

In this section, it is described how one can use weighted quadrature to assemble matrices arising in isogeometric analysis efficiently.

Remark 4. *In this subsection, we will deviate from the notation in the last subsection. We will perform a general analysis and assume that there are on average r quadrature points per element. Further, we will assume that the polynomial degree of the B-spline basis functions is p for both dimensions, and use that the number of degrees of freedom $N = n_1 n_2$ is approximately equal to the number of elements.*

The traditional way to assembly the FEM matrix is to use a loop over the elements, as demonstrated in section B.2. As shown in 2.2.3, there are $(2p+1)^2$ nonzero basis functions on each element. So, each element contributes to $(2p+1)^2$ matrix elements. Since the number of elements is approximately equal to the number of basis functions N, the assembly of the global FEM matrix costs at least $O((2p+1)^2 N)$ operations.

If the B-spline basis functions M_0, M_1, ..., and N_0, N_1, ... are both of degree p, and two quadrature rules with r quadrature points (ξ_0^*, v_0), (ξ_1^*, v_1), ..., (ξ_{r-1}^*, v_{r-1}) and (η_0^*, w_0), (η_1^*, w_1), ..., (η_{r-1}^*, v_{r-1}) are used, one can approximate the integral in (2.15) on a single element with the summation

$$\mathbf{M}_{i,j} \approx \sum_{s=0}^{r-1} \sum_{t=0}^{r-1} v_s w_t M_{i_1}(\xi_s^*) N_{i_2}(\eta_t^*) g(\xi_s^*, \eta_t^*) M_{j_1}(\xi_s^*) N_{j_2}(\eta_t^*)$$

For simplicity, Gaussian quadrature is used here instead of weighted quadrature, but the theory applies to weighted quadrature as well. Naively computing this sum will cost $O(r^2)$ operations. By property 5, it follows that there are $(p+1)^2$ bivariate basis functions $\phi_i(\xi,\eta) = M_{i_1}(\xi) N_{i_2}(\eta)$ nonzero on each element (or $(p+1)^d$ for the d-dimensional case). So, the element matrix will be $(p+1)^2 \times (p+1)^2$. Since each entry in the element matrix will cost $O(r^2)$ operations (or r^d for the d-dimensional case), the total cost of assembling the element matrix is $O(p^4 r^2)$ (or $O(p^{2d} r^d)$ for the d-dimensional case). The cost of assembling the mass matrix this way is $O(p^4 r^2 N)$.

The idea behind *sum-factorization*, introduced in [3], is to re-structure the summations:

$$\sum_{s=0}^{r-1} v_s M_{i_1}(\xi_s^*) M_{j_1}(\xi_s^*) \underbrace{\sum_{t=0}^{r-1} w_t N_{i_2}(\eta_t^*) g(\xi_s^*, \eta_t^*) N_{j_2}(\eta_t^*)}_{I_{s,i_2,j_2}}$$

The values I_{t,i_2,j_2} should be evaluated first, and re-used in the other computations. Since $s = 0, 1, ..., r-1$, and there are $(p+1)^2$ combinations i_1, j_1, it follows that it costs $O(rp^2)$ to evaluate the intermediate values I_{t,i_2,j_2} (or $O(rp^d)$ for d dimensions). These can be re-used in the evaluation of the outer sum. So, after the

intermediate values I_{t,i_2,j_2} are calculated, it costs only $O(r)$ operations to evaluate the summation. So, the total cost of evaluating the element matrix is $O(rp^4)$ (or $O(rp^{2d})$ for d dimensions). So the mass matrix costs $O(rp^{2d}N)$ to assemble.

Row-wise assembly
As shown in the last section, it is possible to assemble an element matrix in $O(rp^{2d})$ operations. Since the number of elements is approximately equal to the number of n basis functions[2], we see that the cost of the assembly is $O(nrp^{2d})$. More generally, every approach that uses an elementwise approach has a cost of at least $O(np^{2d})$, since for each element the element matrix with $(p+1)^{2d}$ entries needs to be assembled. In this section, it is shown that it is possible to assemble the matrix in a more efficient way.

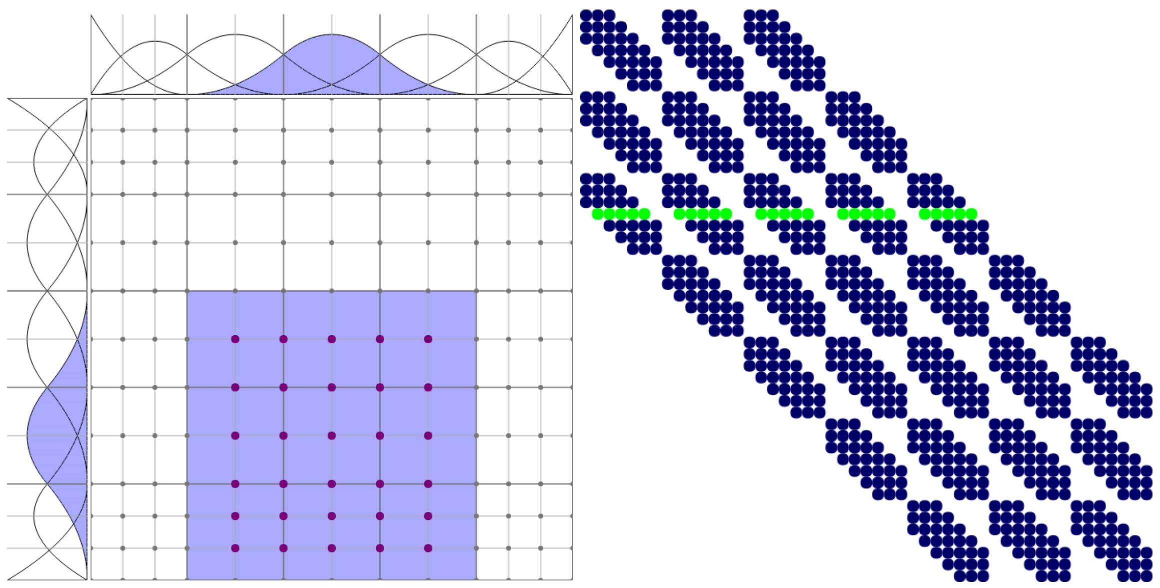


Figure 3.5: On the left, the support of $\phi_{17}(\xi)(\eta) = M_3(\xi)N_2(\eta)$ is shown on the parameteric space. On the right, the sparsity pattern of the FEM matrix shown, with the 18th row, which can be assembled by using the quadarature points in the support of ϕ_{17}, highlighted.

The idea is to assemble a single row of the matrix at once. This can be done by fixing the function ϕ_i, and then computing all $(2p+1)^d$ nonzero elements $\mathbf{M}_{i,j} = \int \phi_i(\boldsymbol{\xi})g(\boldsymbol{\xi})\psi_j(\boldsymbol{\xi})\,\mathrm{d}\boldsymbol{\xi}$ for this i. The idea is to use sum-factorization again, but now loop over all quadrature points in the support of ϕ_i. Since the support is at most $p+1$ elements, and we assume that the average element contains r quadrature points, we can write the expression to evaluate as:

$$\sum_{s=0}^{r(p+1)-1} v_s M_{i_1}(\xi_s^*) M_{j_1}(\xi_s^*) \underbrace{\sum_{t=0}^{r(p+1)-1} w_t N_{i_2}(\eta_t^*) g(\xi_s^*, \eta_t^*) N_{j_2}(\eta_t^*)}_{I_{s,i_2,j_2}}$$

Since i is fixed, and there are $p+1$ nonzero $N_{j_2}(\eta_t^*)$ at any (ξ_s^*, η_t^*), there are $(r(p+1))^d(p+1)$ values that need to be summed to calculate the intermediate values I_{s,i_2,j_2} for every ξ_s^*. In the next step, there are $p+1$ values at $r(p+1)$ points ξ_s^* which need to be summed. The first step dominates the computational complexity, and assembling a row this way costs $O(r^d p^{d+1})$. Assembling the whole mass matrix costs $O(r^d p^{d+1} N)$.

Weighted quadrature has on average $r = 2$ quadrature points per element, so it takes $O(p^{d+1}N)$ time (note that r^d is considered a constant since it is assumed that $d \leq 3$).

Application to two-dimensional case
Now, we show how weighted quadrature can be applied to a two-dimensional problem. Suppose that $\mathbf{s} : [0,1]^2 \to \Omega$ defines the domain Ω, and $M_0, M_1, ..., M_{m-1}$ and $N_0, N_1, ..., N_{n-1}$ are the univariate B-spline basis functions of degree p. By (2.15), we have:

$$\mathbf{M}_{i,j} = \int_0^1 \int_0^1 M_{i_1}(\xi) N_{i_2}(\eta) g(\xi,\eta) M_{j_1}(\xi) N_{j_2}(\eta)\,\mathrm{d}\xi\,\mathrm{d}\eta$$

[2]This holds even if the basis functions are defined using a tensor product.

Suppose we have the quadrature rules $(\xi_0^*, v_{j,0}^{a,b})$, $(\xi_1^*, v_{j,1}^{a,b})$, ... for $a, b \in \{0,1\}$, $j = 0, 1, ..., n_1 - 1$ in the first dimension, and $(\eta_0^*, w_{j,0}^{a,b})$, $(\eta_1^*, w_{j,1}^{a,b})$, ... for $a, b \in \{0,1\}$, $j = 0, 1, ..., n_2 - 1$ in the second dimension. Then we can approximate the right-hand side of 3.3.2 with weighted quadrature to obtain:

$$\mathbf{M}_{i,j} \approx \sum_{s \in F^1_{i_1}} v^{0,0}_{i_1,s} M_{j_1}(\xi^*_s) \underbrace{\sum_{t \in F^2_{i_2}} w^{0,0}_{i_2,t} g(\xi^*_s, \eta^*_t) N_{j_2}(\eta^*_t)}_{I_{s,i_2,j_2}} \tag{3.12}$$

Now suppose we have $\mathbf{G}(\xi,\eta) = \begin{pmatrix} \mathbf{G}_{0,0}(\xi,\eta) & \mathbf{G}_{0,1}(\xi,\eta) \\ \mathbf{G}_{1,0}(\xi,\eta) & \mathbf{G}_{1,1}(\xi,\eta) \end{pmatrix}$. Then we can approximate the stiffness matrix as

$$\mathbf{S}_{i,j} \approx \begin{array}{l} \sum_{s \in F^1_{i_1}} v^{1,1}_{i_1,s} M'_{j_1}(\xi^*_s) \sum_{t \in F^2_{i_2}} w^{0,0}_{i_2,t} \mathbf{G}_{0,0}(\xi^*_s, \eta^*_t) N_{j_2}(\eta^*_t) + \sum_{s \in F^1_{i_1}} v^{0,1}_{i_1,s} M_{j_1}(\xi^*_s) \sum_{t \in F^2_{i_2}} w^{1,0}_{i_2,t} \mathbf{G}_{0,1}(\xi^*_s, \eta^*_t) N'_{j_2}(\eta^*_t) \\ + \sum_{s \in F^1_{i_1}} v^{1,0}_{i_1,s} M'_{j_1}(\xi^*_s) \sum_{t \in F^2_{i_2}} w^{0,1}_{i_2,t} \mathbf{G}_{1,0}(\xi^*_s, \eta^*_t) N_{j_2}(\eta^*_t) + \sum_{s \in F^1_{i_1}} v^{0,0}_{i_1,s} M_{j_1}(\xi^*_s) \sum_{t \in F^2_{i_2}} w^{1,1}_{i_2,t} \mathbf{G}_{1,1}(\xi^*_s, \eta^*_t) N'_{j_2}(\eta^*_t) \end{array} \tag{3.13}$$

Again, sum-factorization should be used to obtain the best possible asymptotic runtime. So, the values $I^{0,0}, I^{0,1}, I^{1,0}$, and $I^{1,1}$ should be evaluated first and re-used in the other computations, so that we obtain

$$\mathbf{S}_{i,j} \approx \begin{array}{l} \sum_s v^{1,1}_{i_1,s} M'_{j_1}(\xi^*_s) I^{0,0}_{s,i_2,j_2} + \sum_s v^{0,1}_{i_1,s} M_{j_1}(\xi^*_s) I^{0,1}_{s,i_2,j_2} \\ + \sum_s v^{1,0}_{i_1,s} M'_{j_1}(\xi^*_s) I^{1,0}_{s,i_2,j_2} + \sum_s v^{0,0}_{i_1,s} M_{j_1}(\xi^*_s) I^{1,1}_{s,i_2,j_2} \end{array}$$

where

$$I^{0,0}_{s,i_2,j_2} := \sum_t w^{0,0}_{i_2,t} \mathbf{G}_{0,0}(\xi^*_s, \eta^*_t) N_{j_2}(\eta^*_t)$$

$$I^{0,1}_{s,i_2,j_2} := \sum_t w^{1,0}_{i_2,t} \mathbf{G}_{0,1}(\xi^*_s, \eta^*_t) N'_{j_2}(\eta^*_t)$$

$$I^{1,0}_{s,i_2,j_2} := \sum_t w^{0,1}_{i_2,t} \mathbf{G}_{1,0}(\xi^*_s, \eta^*_t) N_{j_2}(\eta^*_t)$$

$$I^{1,1}_{s,i_2,j_2} := \sum_t w^{1,1}_{i_2,t} \mathbf{G}_{1,1}(\xi^*_s, \eta^*_t) N'_{j_2}(\eta^*_t)$$

3.4. Other methods

In this chapter, different assembly strategies were explored. In particular, we have considered Gaussian quadrature, which has $r = p + 1$ points per element, and weighted quadrature, which has on average $r = 2$ quadrature points per element. The naive way to assemble the mass matrix costs $O(r^d p^{2d})$ per degree of freedom. Using elementwise assembly with sum-factorization costs $O(rp^{2d})$ per degree of freedom. Finally, using sum-factorization and assembling row-wise has a cost of $O(r^{d+1} p^{d+1})$. Some other state-of-the-art methods that do not use an elementwise assembly loop are now considered.

In [21], a uniform knot vector is used, so that the integrals of products of B-spline basis functions can be looked up. The geometric factor is then included in the integral by interpolating the exact integrals. This strategy takes $O(p^{2d})$ operations per degree of freedom to assemble the FEM matrix.

In [22], which relies on the work in [21], it is proposed to use a rank R tensor approximation to assemble the FEM matrix in $O(Rp^d)$ operations per degree of freedom.

4

Improvements on weighted quadrature

In this chapter, the technique described in [7] will be adapted to a more general setting. Some other mathematical improvements are also described. It should be noted that most of the results in this section were communicated by Mattia Tani, and are the work of Giancarlo Sangalli and Mattia Tani. Specifically, the results in sections 4.3, 4.4, and 4.2 are based on results reported by Mattia Tani, while the work in section 4.1 is original. Any errors in the description should be attributed to the author of this document.

While mathematical work on weighted quadrature was not within the scope of this project, the approach presented in [7] turned out to be not directly applicable for a general IgA implementation. The results in [7] are based on a smooth (that is, C^∞-continuous) mapping. The technique needs to be adapted in order to work for non-smooth mappings. To understand the problems that occur when the technique is applied to non-smooth mappings, it is useful to interpret weighted quadrature as a technique that interpolates the function on which the quadrature is applied[1]. Unfortunately, applying weighted quadrature to non-smooth mappings also requires the use of basis functions which are based on an open, non-uniform knot vector, while in [7] only uniform open knot vectors are considered.

In section 4.1, different rules for picking quadrature rules are proposed that generalize the midpoint rule to non-uniform knot vectors. In section 4.2, the problems with non-smooth mappings are demonstrated, analyzed, and solved. In section 4.3, a more accurate quadrature rule is described. Finally, in section 4.4 an efficient way to performed matrix-free multiplication by a matrix that is assembled with weighted quadrature is shown.

4.1. Generalization to non-uniform open knot vectors

In order to use weighted quadrature for non-uniform open knot vectors, it is necessary to find quadrature points that satisfy conditions (3.8). For uniform open knot vectors, the midpoint rule from [7], which is described in section 3.3, can be used. For non-uniform knot vectors, the situation is more complex. We present several generalizations of the midpoint rule that can be used for non-uniform knot vectors. Just like the midpoint rule, we will place quadrature points on the knots and evenly divided over knot spans. Since the support of the basis functions depend on the multiplicities of the knots, it is natural to let the number of quadrature points on each knot span (ξ_k, ξ_{k+1}) for which $\xi_k \neq \xi_{k+1}$ depend on the multiplicities μ_k and μ_{k+1}. Indeed, this is the approach that is taken. Moreover, we will characterize rules for picking quadrature points by a function $m : \mathbb{N}^2 \to \mathbb{N}$ that maps the multiplicities μ_k, μ_{k+1} of the knots to the number of quadrature points on the knot span (ξ_k, ξ_{k+1}).

It should be noted that there are several criteria for selecting rules for picking quadrature points. First, it is important that the quadrature points are evenly divided over the elements, in order to ensure that the quadrature rule will be accurate. Then, there are a number of ways in which the rules for picking quadrature points influence the amount of computations that need to be done:

1. For each quadrature point, the basis functions need to be evaluated, and the weights need to be computed and stored. Additionally, for each tensor product quadrature point, the geometric factor needs to be evaluated.

[1] This is a nontrivial step, since the technique is not explicitly presented as an interpolation technique.

2. For each basis function, a linear system needs to be solved in order to find the weights of the quadrature rule corresponding to that basis function. The size of this linear system depends on the number of active quadrature points.

These two considerations are the most important ones in the computing-in-time paradigm. In the computing-in-space paradigm, there are other, more important considerations:

1. In the hardware implementation, it is desirable to keep some data in registers for all active quadrature points. So, it the worst-case number of active quadrature points is high, this will make the hardware implementation more expensive.

2. If between two basis functions the number of new active quadrature points is constant, this will make the hardware implementation simpler and cheaper.

The different rules for picking quadrature points will have different trade-offs between these considerations.

4.1.1. Max rule

The max rule for picking quadrature points is defined by

$$m(\mu_1, \mu_2) = \begin{cases} p+1 & \text{if } \mu_1 + \mu_2 > p+1 \\ \max(\mu_1, \mu_2) & \text{otherwise} \end{cases}$$

This is a straightforward generalization of the midpoint rule, with an exception for basis functions which only have support on one knot span. These basis functions share support with $p+1$ basis functions. This rule will use $p+1$ quadrature points in the first knot span, while the midpoint rule uses only p. This rule can be adapted to also yield p quadrature points when $\min(\mu_k, \mu_{k+1}) = 1$ and $\max(\mu_k, \mu_{k+1}) = p+1$. If $p < 4$, this rule is guaranteed to give quadrature points that satisfy conditions (3.8). When $p \geq 4$, we can make sure that

This quadrature rule does not work for basis functions of order $p < 4$, or when enough h-refinement is used (see section 4.1.3). The max rule is the rule that is used in the final dataflow implementation.

4.1.2. Regular rule

The regular rule for picking quarature points is defined by

$$m(\mu_1, \mu_2) = \mu_1 + \mu_2 - 1$$

One should take p instead of $p+1$ quadrature points on the first and last element. This rule yields more quadrature points than the maximum rule. However, this rule has the advantage that it is possible to choose the active quadrature points in a more regular way. More specifically, when the quadrature points taken to be $\xi_0^* < \xi_1^* < ... < \xi_{m-1}^*$, the active quadrature points for basis function N_k can be taken as $\xi_{\max(0,2k-p)}$, $\xi_{\max(0,2k-p)+1}$, ..., $\xi_{\min(2k+p,m-1)}$. This way, there are only two active basis functions for N_k that are not active for N_{k-1}. This is great for a hardware implementation, since there are exactly two new active quadrature points that need to be handled. For the max rule, this number can be anywhere from 0 to $p+1$. Unfortunately there was no time to test this rule.

4.1.3. Bounds on the number of quadrature points

The regular rule yields $2(m-1)+p$ quadrature points in total, where m is the number of knots. For the number of quadrature points given by the max rule, there is no simple expression. However, max rule always gives less quadrature points than the regular rule.

We will now derive bounds on the number of active quadrature points. In general, the worst case happens when the first and last element on which a basis function has support have the maximum number of quadrature points. This is illustrated in figure 4.1. Notice that the basis function indicated in green has $4p-1=7$ quadrature points inside its support.

Figure 4.1: Basis functions of degree 2 for an open knot vector with three elements.

In this, there are $2(p-1)+1$ knots in the inner $p-1$ knot spans, and $2p$ in the outer knot spans. So there are $4p-1$ quadrature points in the support. This is an upper bound of the number of active quadrature for both the regular rule and the max rule. However, when the regular rule is used, one can choose to restrict to using at most $2p+1$ active quadrature rules.

Another possibility to reduce the worst-case number of active quadrature points, is to ensure that this specific worst case never happens. This can be done by using h-refinement. h-refinement inserts knots with multiplicity 1, which creates new knot spans. In this way, the basis functions associated to the refined knot vector have smaller support. If there are enough knots with multiplicity 1 inserted, there can be at most one knot with a higher multiplicity in the support of each basis function. Without proof, we state that if the last H refinements are all h-refinements, and

$$2^H > p+1$$

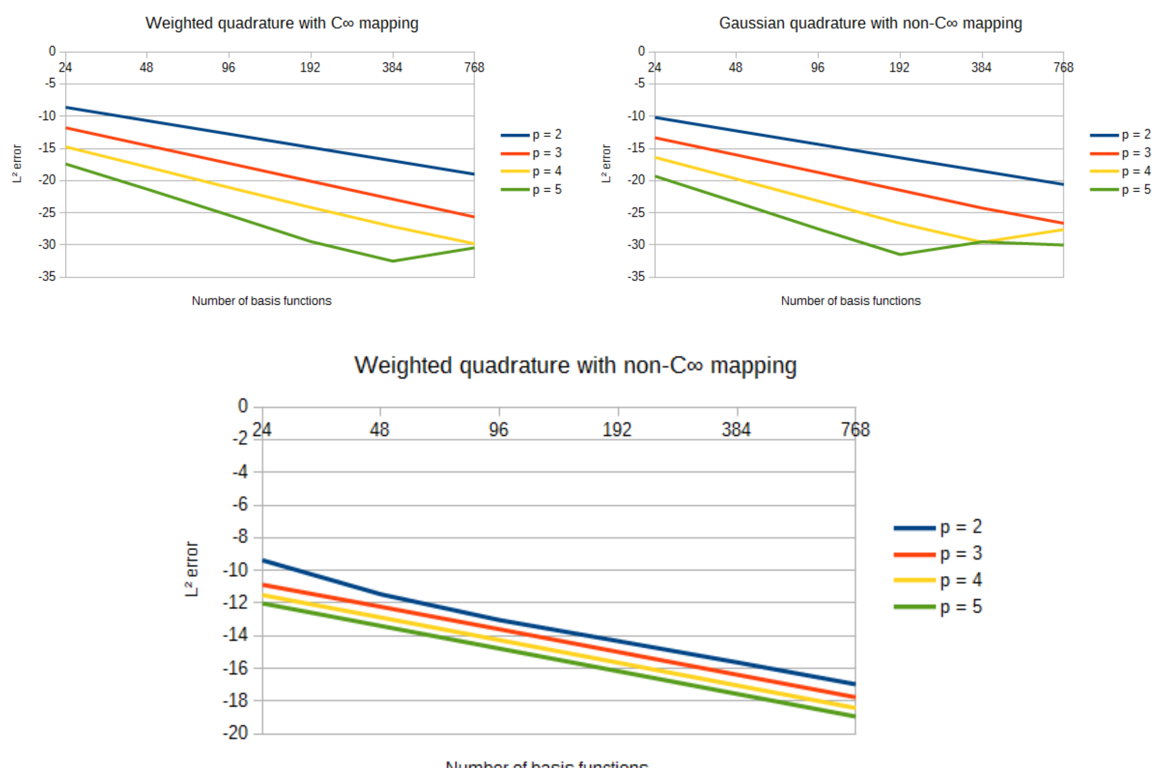
then the number of active quadrature points is at most $3p$. This approach is used in the dataflow implementation.

4.1.4. Using global interpolation

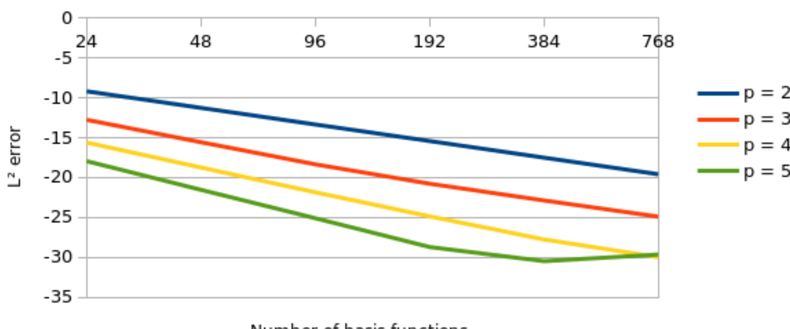
A completely different strategy is to drop the requirement that the active quadrature points should be inside the support of the weight function N_k. For example, the Greville abscissae can be used as the quadrature points. This is similar to the approach that is used in [21]. Using this approach, the number of quadrature points will be equal to the number of basis functions in the projection space, and all of the active quadrature points for the quadrature rule with weight function N_k will also be active quadrature points for the quadrature rule with weight function N_{k-1}. This is likely to be slightly less accurate, since the grid of quadrature points is not as dense. However, it is not clear if this approach has been tested with weighted quadrature, and the accuracy of the quadrature might be acceptable.

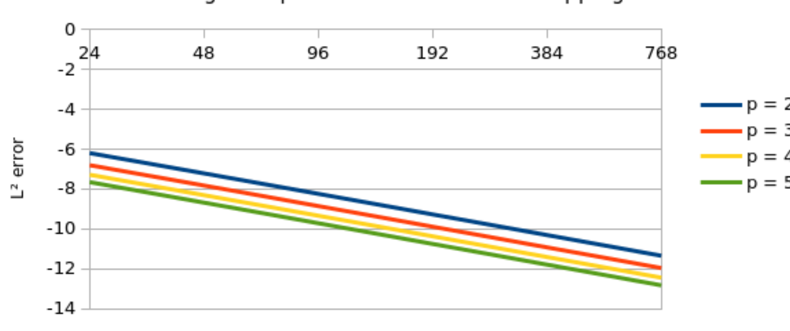
4.2. Generalization to non-smooth mappings

When weighted quadrature is applied to a mapping that is not C^∞-continuous, it can be seen that the convergence is not as good as expected. The following plots are convergence plots for one-dimensional problems.



Testing shows that the exactness conditions are satisfied. Observing the convergence graphs for the mass and stiffness matrix for a one-dimensional problem separately reveals a clue.





Obviously, the problem is in the mass matrix. Unfortunately, in two dimensions the matrix has a different structure, and the convergence is also bad for the stiffness matrix.

The author was made aware of the nature of the problem – and of its solution – by Mattia Tani. Even if the weights are picked in such a way that the exactness conditions hold, this is not a guarantee for convergence. In (3.3), the exactness is only imposed on the approximation space. Now suppose we want to evaluate

$$\int f(\xi) N_i(\xi)\, \mathrm{d}\xi$$

with a weighted quadrature rule. Using the approximation space A with weighted quadrature essentially means that we find an l^2 projection[2] f^* of f onto A. In order to have an accurate approximation $f^* \approx f$, it is necessary that the approximation space A has the same or lower continuity than the integrand f.

Suppose that the mapping **s** is smooth and one-dimensional. We can use the notation $s : [0,1] \to \mathbb{R}$ to emphasize that s is real-valued. Then we have $f(\cdot) = |s'(\cdot)| N_j(\cdot)$. Since s is C^∞-continuous, $|s'|$ is C^∞-continuous as well. The continuity of the projection space A should be equal to the continuity of the B-spline basis functions $N_0, N_1, \ldots$. So $f(\cdot) = \det(Ds(\cdot)) N_j(\cdot)$ has the same continuity as the basis functions, and the projection space is indeed not of higher continuity than the integrand f.

For the case of a one-dimensional B-spline mapping $s(\cdot) = \sum_k c_k \tilde{N}_k(\cdot)$, we have $f(\cdot) = |s'(\cdot)| N_j(\cdot)$. The B-spline basis functions $\tilde{N}_0, \tilde{N}_1, \ldots$ are of order $\tilde{p}$. The knot vector $\tilde{\Xi}$ is refined to obtain the knot vector Ξ. From Ξ, the FEM basis functions $N_0, N_1, \ldots$ of order p are obtained, so that we have $\tilde{p} \le p$. Now assume, for simplicity, that all internal knots of $\tilde{\Xi}$ have multiplicity 1. Then, s is $(\tilde{p}-1)$-continuous at the knots, and s' is $(\tilde{p}-2)$-continuous at the knots. We also assumed that the functions N_0, N_1, ... preserve the continuity of $\tilde{N}_0$, $\tilde{N}_1$, ... at the knots. So N_0, N_1, ... are used as basis functions for the projection space, the projection space is only $(\tilde{p}-1)$-continuous at the knots. So the projection space has a higher continuity than the integrand $f(\cdot) = |s'(\cdot)| N_j(\cdot)$ and the quadrature rule is not guaranteed to give an accurate approximation.

[2] More precisely, we have that f^* is the unique minimizer of $\sum_s (f(\xi_s^*) - f^*(\xi_s^*))^2$.

For the stiffness matrix, this problem does not occur. For the stiffness matrix, $N_{0,p-1}$, $N_{1,p-1}$, ... are the basis functions of the projection space. These basis functions are $(\tilde{p}-2)$-continuous, just like the integrand $f(\cdot) := |\frac{1}{s'(\cdot)}|N_j'(\cdot)$.

For the multivariate case, the problem occurs both for the mass matrix and for the stiffness matrix, since the integrals occuring in the stiffness matrix also contains integrands of the form $\int f(\xi)N_i(\xi)\,\mathrm{d}\xi$ (see 2.16).

This can be resolved by increasing the multiplicity of the knots of the knot vector for the approximation space, which reduces the continuity of the approximation space at the knots. After this, p- and h-refinement can be applied as usual. The resulting knot vector is used to obtain the basis functions of the projection space. For the knot vector on which the univariate FEM basis functions are based, it is not necessary to increase the multiplicity (but the same sequence of p- and h-refinements should be used).

4.3. A more accurate quadrature rule

This section describes a more accurate quadrature rule that is due to Giancarlo Sangalli and Mattia Tani.

In section 3.3.1, four different quadratures are used to evaluate integrals of the form

$$\begin{gathered}\int N_j(\xi)g(\xi)N_i(\xi)\,\mathrm{d}\xi\\ \int N_j(\xi)g(\xi)N_i'(\xi)\,\mathrm{d}\xi\\ \int N_j'(\xi)g(\xi)N_i(\xi)\,\mathrm{d}\xi\\ \int N_j'(\xi)g(\xi)N_i'(\xi)\,\mathrm{d}\xi\end{gathered}$$

The difference between the quadrature rules for the integrals $\int N_j(\xi)g(\xi)N_i(\xi)\,\mathrm{d}\xi$ and $\int N_j'(\xi)g(\xi)N_i(\xi)\,\mathrm{d}\xi$, is that a different projection space is used. The projection space for the quadrature rules for integrals of the form $\int N_j(\xi)g(\xi)N_i(\xi)\,\mathrm{d}\xi$ uses basis function of degree p, which are $C^{p-\mu_k}$-continuous along each knot ξ_k, while the projection space for the quadrature rules for integrals of the form $\int N_j'(\xi)g(\xi)N_i(\xi)\,\mathrm{d}\xi$ uses basis functions of degree $p-1$ which are $C^{p-\mu_k-1}$-continuous along knots ξ_k.

It is possible to use one projection space that is accurate for both integrands $N_j(\xi)g(\xi)$ and $N_j'(\xi)g(\xi)$. This can be done by taking the knot vector which is used to define the basis functions of the projection space, and increasing the multiplicity of each knot by one. Then, the basis functions of degree p can be used to define a projection space which is $C^{p-\mu_k-1}$-continuous along knots ξ_k (where μ_k denotes the multiplicity of ξ_k in the first knot vector). So the quadrature rule defined by using this projection space can be used to accurately approximate integrals of the form $\int N_j(\xi)g(\xi)N_i(\xi)\,\mathrm{d}\xi$ and $\int N_j'(\xi)g(\xi)N_i(\xi)\,\mathrm{d}\xi$. The same principle works for defining a quadrature rule which can be used for both $\int N_j(\xi)g(\xi)N_i'(\xi)\,\mathrm{d}\xi$ and $\int N_j'(\xi)g(\xi)N_i'(\xi)\,\mathrm{d}\xi$.

So, it suffices to compute just two quadrature rules: One for integrals of the form $\int f(\xi)N_i(\xi)\,\mathrm{d}\xi$ and one for integrals of the form $\int f(\xi)N_i'(\xi)\,\mathrm{d}\xi$. These quadratures are reported to be as accurate as Gaussian quadrature by Mattia Tani. However, this accuracy comes at the cost of needing more than two quadrature points per element. If the max rule is used, we obtain a minimum of three quadrature points per element. If the regular quadrature rule is used, we obtain a minimum of four quadrature points per element.

4.4. Efficient matrix-free multiplication

Suppose we want to do a matrix-free multiplication to obtain the elements $a_0, a_1, ..., a_{N-1}$ of the vector $\mathbf{a} = \mathbf{Mz}$, which is obtained by multiplying some vector $\mathbf{z} = (z_0, z_1, ..., z_{N-1})^\top$ by the mass matrix $\mathbf{M}$. By the definition of matrix multiplication, we have

$$a_i = \sum_{j=0}^{N-1} \mathbf{M}_{i,j} z_j$$

Substituting (2.1.2) gives

$$a_i = \sum_{j=0}^{N-1} \left(\int_\Omega \psi_i \psi_j \,\mathrm{d}\Omega\right) z_j$$

We can bring the summation inside the integrals, so that we have

$$a_i = \int_\Omega \psi_i \left(\sum_{j=0}^{N-1} z_j \psi_j\right) \mathrm{d}\Omega$$

Now, for the vector $\mathbf{z}$ we define the *associated spline* z as

$$z(\xi,\eta) := \sum_{j=0}^{N-1} z_j \phi_j(\xi,\eta)$$

Notice that this is analogous to (2.1.1). Using (2.3.1), we have

$$z \circ \mathbf{s}^{-1} = \sum_{j=0}^{N-1} z_j \psi_j$$

We can now rewrite the formula for a_i as

$$a_i = \int_\Omega \psi_i (z \circ \mathbf{s}^{-1})\, \mathrm{d}\Omega$$

So, elements of a matrix-vector product can be written as an integral of a basis function with another function $z \circ \mathbf{s}^{-1}$. Analogous to the derivation in section 2.3.1, we can transform this integral to one in the parametric domain:

$$a_i = \int_{\Omega_0} \psi_i\ z\ g\ \mathrm{d}\Omega$$

Now, for the two-dimensional case we can use (2.3.1) to obtain

$$a_i = \int_0^1 \int_0^1 M_{i_1}(\xi) N_{i_2}(\eta) z(\xi,\eta)\ \mathrm{d}\xi\ \mathrm{d}\eta$$

And if weighted quadrature is used to approximate this integral, we get

$$a_i \approx \sum_{s \in F^1_{i_1}} v^{0,0}_{i_1,s} \sum_{t \in F^2_{i_2}} w^{0,0}_{i_2,t} z(\xi^*_s, \eta^*_t) \tag{4.1}$$

This is a more efficient way of calculating a_i than assembling a whole row of the matrix and calculating the inner product of the row and the vector $\mathbf{z}$.

The same method can be applied to the stiffness matrix to obtain a formula for the elements $a_0, a_1, ..., a_{N-1}$ of the matrix-vector product $\mathbf{a} = \mathbf{S}\mathbf{z}$ of the vector $\mathbf{z}$ with the stiffness matrix $\mathbf{S}$. In this case, we have

$$\begin{aligned} a_i &= \int_\Omega \nabla\psi_i \cdot \nabla(z \circ \mathbf{s}^{-1})\ \mathrm{d}\Omega \\ &= \int_{\Omega_0} (\nabla\psi)^\top\ \mathbf{G}\ \nabla z\ \mathrm{d}\Omega \end{aligned}$$

Expanding this for the two-dimensional case yields

$$a_i = \begin{array}{ll} \int_0^1 \int_0^1 M'_{i_1}(\xi) N_{i_2}(\eta) \mathbf{G}_{0,0}(\xi,\eta) \frac{\partial z}{\partial \xi}(\xi,\eta) \mathrm{d}\xi\ \mathrm{d}\eta & + \int_0^1 \int_0^1 M'_{i_1}(\xi)\ N_{p_i}(\eta) \mathbf{G}_{0,1}(\xi,\eta) \frac{\partial z}{\partial \eta}(\xi,\eta) \mathrm{d}\xi\ \mathrm{d}\eta \\ + \int_0^1 \int_0^1 M_{i_1}(\xi) N'_{i_2}(\eta) \mathbf{G}_{1,0}(\xi,\eta) \frac{\partial z}{\partial \xi}(\xi,\eta) \mathrm{d}\xi\ \mathrm{d}\eta & + \int_0^1 \int_0^1 M_{i_1}(\xi)\ N'_{i_2}(\eta) \mathbf{G}_{1,1}(\xi,\eta) \frac{\partial z}{\partial \eta}(\xi,\eta) \mathrm{d}\xi\ \mathrm{d}\eta \end{array}$$

And when weighted quadrature is used, we obtain

$$a_i \approx \begin{array}{ll} \sum_s v^{1,1}_{i_1,s} \sum_t w^{0,0}_{i_2,t} \mathbf{G}_{0,0}(\xi^*_s,\eta^*_t) \frac{\partial z}{\partial \xi}(\xi^*_s,\eta^*_t) & + \sum_s v^{0,1}_{i_1,s} \sum_t w^{1,0}_{i_2,t} \mathbf{G}_{0,1}(\xi^*_s,\eta^*_t) \frac{\partial z}{\partial \eta}(\xi^*_s,\eta^*_t) \\ + \sum_s v^{1,0}_{i_1,s} \sum_t w^{0,1}_{i_2,t} \mathbf{G}_{1,0}(\xi^*_s,\eta^*_t) \frac{\partial z}{\partial \xi}(\xi^*_s,\eta^*_t) & + \sum_s v^{0,0}_{i_1,s} \sum_t w^{1,1}_{i_2,t} \mathbf{G}_{1,1}(\xi^*_s,\eta^*_t) \frac{\partial z}{\partial \eta}(\xi^*_s,\eta^*_t) \end{array}$$

In practice it will be useful to store the intermediate values $\mathbf{G}_{0,0}(\xi^*_s,\eta^*_t) \frac{\partial z}{\partial \xi}(\xi^*_s,\eta^*_t)$, $\mathbf{G}_{0,1}(\xi^*_s,\eta^*_t) \frac{\partial z}{\partial \eta}(\xi^*_s,\eta^*_t)$, $\mathbf{G}_{1,0}(\xi^*_s,\eta^*_t) \frac{\partial z}{\partial \xi}(\xi^*_s,\eta^*_t)$, $\mathbf{G}_{1,1}(\xi^*_s,\eta^*_t) \frac{\partial z}{\partial \eta}(\xi^*_s,\eta^*_t)$, since this value can be re-used. For this purpose, we define:

$$\begin{aligned} z_{0,0}(\xi^*,\eta^*) &= \mathbf{G}_{0,0}(\xi^*_s,\eta^*_t) \tfrac{\partial z}{\partial \xi}(\xi^*_s,\eta^*_t) \\ z_{0,1}(\xi^*,\eta^*) &= \mathbf{G}_{0,1}(\xi^*_s,\eta^*_t) \tfrac{\partial z}{\partial \eta}(\xi^*_s,\eta^*_t) \\ z_{1,0}(\xi^*,\eta^*) &= \mathbf{G}_{1,0}(\xi^*_s,\eta^*_t) \tfrac{\partial z}{\partial \xi}(\xi^*_s,\eta^*_t) \\ z_{1,1}(\xi^*,\eta^*) &= \mathbf{G}_{1,1}(\xi^*_s,\eta^*_t) \tfrac{\partial z}{\partial \eta}(\xi^*_s,\eta^*_t) \end{aligned} \tag{4.2}$$

So we have

$$a_i \approx \begin{array}{ccccc} & \sum_s v^{1,1}_{i_1,s} \sum_t w^{0,0}_{i_2,t} z_{0,0}(\xi^*_s,\eta^*_t) & + & \sum_s v^{0,1}_{i_1,s} \sum_t w^{1,0}_{i_2,t} z_{0,1}(\xi^*_s,\eta^*_t) \\ + & \sum_s v^{1,0}_{i_1,s} \sum_t w^{0,1}_{i_2,t} z_{1,0}(\xi^*_s,\eta^*_t) & + & \sum_s v^{0,0}_{i_1,s} \sum_t w^{1,1}_{i_2,t} z_{1,1}(\xi^*_s,\eta^*_t) \end{array} \tag{4.3}$$

5

Matrix-free solution techniques

Gaussian elimination is a simple, well-known algorithm for solving linear systems. Methods that directly compute the exact solutio $\mathbf{v}$ of the linear system $\mathbf{Mv} = \mathbf{b}$ are know as *direct solvers*. For larger linear systems, direct methods are often too slow to be of practical use. For this reason, *iterative methods* are often preferred. These methods approximate the solution $\mathbf{v}$ by a sequence of vectors $\mathbf{v_0}, \mathbf{v_1}, \ldots$ such that $\lim_{n\to\infty} \mathbf{v_n} = \mathbf{v}$.

5.1. Introduction to matrix-free methods

If a matrix $\mathbf{M}$ has a particular structure, it might be possible to calculate $\mathbf{Mb}$ without ever assembling $\mathbf{M}$ in memory. This is called a *matrix-free* multiplication. One straightforward ways to do matrix-free multiplication is to assemble only a single row at a time, and compute the ith element a_i of $\mathbf{a} = \mathbf{Mb}$ by taking the inner product of the ith row of the matrix and $\mathbf{b}$. This has the same cost as assembling the whole matrix. However, there is no need to load the assembled matrix rows from memory. If the assembly of a single row can be done faster than loading a matrix row from memory, matrix-free multiplication can be much faster. In some special cases, it is possible to perform a row-vector inner product in a more efficient way than assembling the whole row and performing the inner product.

Linear solvers that never use elements from the matrix explicitly, but only rely on matrix multiplications, can be implemented with matrix-free multiplications. Many iterative methods fall in this category. A class of particular interest is the class of so-called Krylov iterative methods.

5.2. Conjugate gradient method

The conjugate gradient method is a well-known method for solving symmetric positive-definite linear systems. It is the preferred method for symmetric positive definite systems. The conjugate gradient method was introduced by [15].

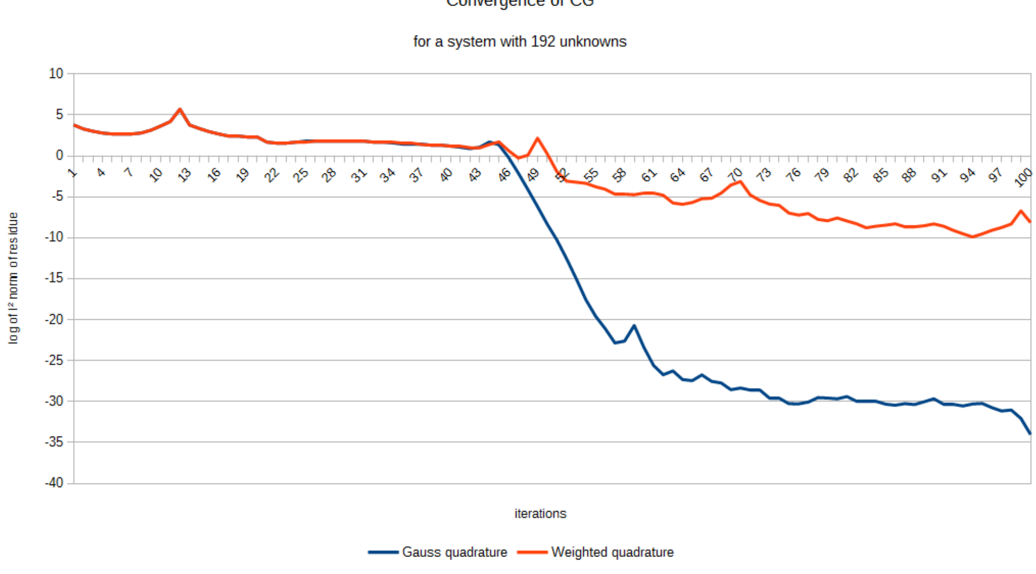
The conjugate gradient method will do at most $i_{\max}$ iterations, and will return the iterate $\mathbf{x_i}$ when the residual $r_i = \mathbf{b} - \mathbf{Ax_i}$ satisfies $||\mathbf{r_i}||_{l^2} \leq \epsilon ||r_0||_{l^2}$. The conjugate uses a starting guess $\mathbf{x_0}$. If one has a reasonable approximation or estimate $\mathbf{x_0} \approx \mathbf{x}$, this can be used. We will use no starting guess and set $\mathbf{x_0} = \mathbf{0}$. In this case we have $\mathbf{r_0} = \mathbf{b}$. The following algorithm is obtained (based on the one given in [28]):

Algorithm 3 Conjugate gradient method

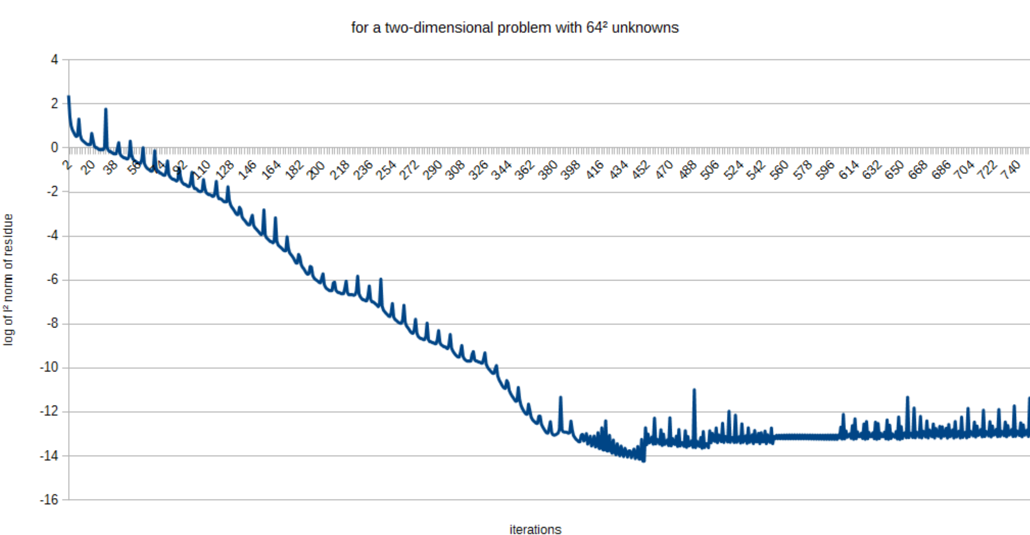
```
1: function CG(A, b, i_max, ε)        ▷ Computes solution x of Ax = b
2:     x = 0
3:     r = b
4:     δ = b^T b
5:     δ_0 = δ
6:     for i = 1 : i_max do
7:         q = Ad
8:         α = δ / (d^T q)
9:         x = x + αd
10:        r = r − αq
11:        δ_old = δ
12:        δ = r^T r
13:        if δ ≤ ε^2 δ_0 then
14:            return x
15:        end if
16:        β = δ / δ_old
17:        d = r + βd
18:    end for
19:    error("Conjugate gradient has not converged within i_max iterations")
20: end function
```

5.2.1. Results

We now compare the convergence of the symmetric linear system that is obtained with Gauss quadrature with the system that is obtained with weighted quadrature. The system is generated from the one-dimensional problem $u'' = e^x$ with 192 basis functions of polynomial degree 2.



It can be seen that the conjugate gradient method works well for the symmetric linear system that is obtained by using Gauss quadrature. For the nonsymmetric system obtained with weighted quadrature, the conjugate gradient does not converge to an accurate approximation to the solution of the linear system. For systems obtained from two-dimensional systems, the situation is similar, but the conjugate gradient converges to a more accurate approximation before the convergence stops.



It is not entirely clear why this behaviour occurs. A plausible argument is that the tensor product structure makes the matrix more symmetric, and this causes better behavior. Tests show that the number of basis functions (which is typically larger for two-dimensional problems) has no influence. Symmetrizing the matrix obtained by weighted quadrature results in good convergence of the conjugate gradient method, but the solution $\mathbf{u}$ of the linear system no longer yields an accurate approximation u^h.

It might be preferable to obtain an approximation that has a smaller error than conjugate gradient can offer. In this case, an linear method that is suitable for non-symmetric linear systems should be used, or the use of a preconditioner might be considered.

5.3. Stabilized biconjugate gradient method

The stabilized biconjugate gradient method (BiCGSTAB) was published in [31]. Like the conjugate gradient method, it is a Krylov subspace method. However, it is adapted to work for non-symmetric matrices as well. The method is related to the biconjugate gradient method, conjugate gradient squares, and generalized minimum residual method. All these methods can be used for the same purpose (the convergence varies between different methods, though). Like CG, BiCGSTAB requires a starting guess $\mathbf{x_0}$. Again, we will take $\mathbf{x_0} = \mathbf{0}$.

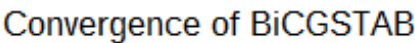
Algorithm 4 Stabilized biconjugate gradient method

```
1: function BiCGSTAB(A, b, i_max, ε)          ▷ Computes solution x of Ax = b
2:     x = 0
3:     r = b
4:     r0 = b
5:     v = 0
6:     p = 0
7:     ρ_old = 1
8:     α = 1
9:     ω0 = 1
10:    δ0 = b^T b
11:    ρ = δ0
12:    for i = 1 : i_max do
13:        β = (ρ / ρ_old)(α / ω)
14:        p = r + β(p − ωv)
15:        v = Ap
16:        α = ρ / (r0·v)
17:        h = x + αp
18:        s = r − αv
19:        t = As
20:        ω = (s·t) / (t·t)
21:        x = h + ωs
22:        r = s − ωt
23:        if r^T r ≤ ε^2 δ0 then
24:            return x
25:        end if
26:        ρ = r0 · r
27:    end for
28:    error("BiCGSTAB has not converged within i_max iterations")
29: end function
```

For a symmetric matrix, BiCGSTAB will compute the same vectors $\mathbf{x_0}, \mathbf{x_1}, \ldots$ as CG. However, BiCGSTAB needs two matrix multiplication per iteration. Since the matrix multiplication is usually the most expensive operation, the cost per iteration is approximately twice as high as for CG.

5.3.1. Results

In figure 5.1, it can be seen that solving the linear system obtained with weighted quadrature converges about as well as conjugate gradient applied to the linear system that is obtained with Gauss quadrature for the same problem. It should be noted that one iteration of BiCGSTAB performs two CG-type iterations, so a BiCGSTAB iteration is approximately twice as expensive.



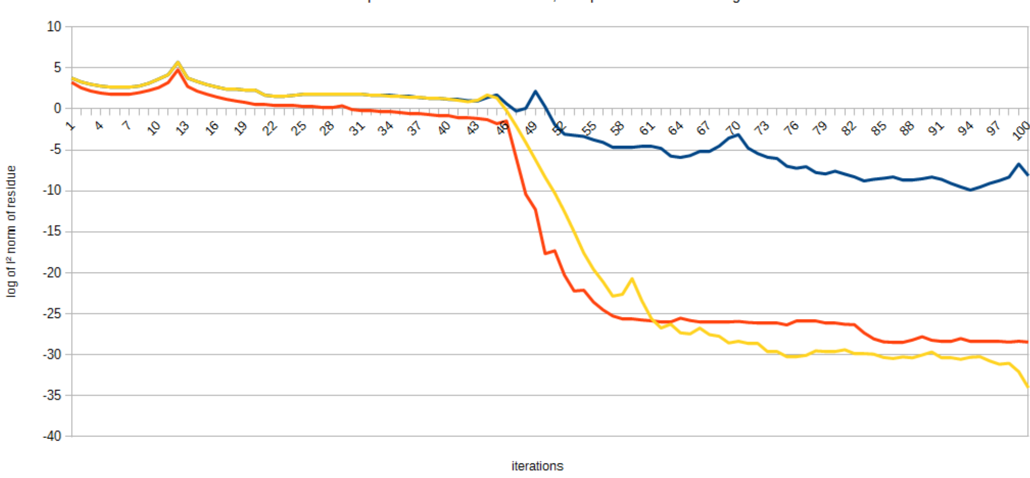


Figure 5.1: The l^2 norm of the residual is plotted as a function of the number of iterations, for CG with Gauss quadrature, CG with weighted quadrature, and BiCGSTAB with weighted quadrature.

5.4. Boundary conditions in matrix-free solvers

In this section, we will consider how to derive a linear system $\mathbf{A}'\mathbf{u} = \mathbf{b}'$ that implements the boundary conditions, as well as a linear system $\mathbf{A}^*\mathbf{u} = \mathbf{b}^*$ that implements the boundary conditions and is symmetric. The matrix-free solution of these systems is considered as well.

Implementing Dirichlet boundary conditions in linear systems arising from FEM problems, is a well-studied problem that is treated in most books on FEM. Dirichlet boundary conditions are the easiest type of boundary conditions to implement. FEM aims to compute a function $u^h : \Omega \to \mathbb{R}$ that is an approximation $u^h \approx u$ to the solution $u : \Omega \to \mathbb{R}$ of a boundary value problem. Dirichlet boundary conditions simply impose $u = u_{\partial\Omega}$ on the boundary $\partial\Omega$ of the domain Ω. The Dirichlet boundary conditions are usually imposed on the approximation u^h by prescribing the coefficients u_{i_0}, u_{i_1}, ... in the approximation $u^h = \sum_{i=0}^{N-1} u_i \psi_i$ to the solution of the boundary value problem that correspond to basis functions ψ_i that have support on the boundary.

Suppose that there are m coefficients u_{i_0}, u_{i_1}, ..., $u_{i_{m-1}}$ that correspond to basis functions with support on the boundary. We define S to be the set of the indices of these coefficients: $S = \{ i_0, i_1, ..., i_{m-1} \}$. In general, it is not possible to pick u_{i_0}, u_{i_1}, ..., $u_{i_{m-1}}$ in such a way that u^h satisfies the boundary conditions $u^h = u_{\partial\Omega}$ everywhere on the boundary $\partial\Omega$. So they are picked in such a way that $u^h \approx u_{\partial\Omega}$ on $\partial\Omega$. Assume that by some method (see sections A.2.1 and A.2.2) the prescribed values v_{i_0}, v_{i_1}, ..., $v_{i_{m-1}}$ are picked, so that the boundary conditions are implemented by imposing $u_{i_k} = v_{i_k}$ for $k = 0, 1, ..., m-1$ on the linear system.

To accurately denote the computations, the notation $\mathbf{x}_{\text{bound}}$ is used to denote the vector $(x_0, x_1, ..., x_{N-1})$ with elements

$$(\mathbf{x}_{\text{bound}})_i = \begin{cases} x_i & \text{if } i \in S \\ 0 & \text{otherwise} \end{cases}$$

The complementary vector with be denoted by $\mathbf{x}_{\text{free}}$:

$$(\mathbf{x}_{\text{free}})_i = \begin{cases} 0 & \text{if } i \in S \\ x_i & \text{otherwise} \end{cases}$$

So that $\mathbf{x} = \mathbf{x}_{\text{bound}} + \mathbf{x}_{\text{free}}$.

Usually, the boundary conditions are implemented by changing the $i+1$th equation in the linear system to $u_i = v_i$ for each prescribed degree of freedom $i \in S$. We then obtain the linear system $\mathbf{A}'\mathbf{u} = \mathbf{b}'$ where

$$\mathbf{A}'_{i,j} = \begin{cases} \delta_{i,j} & \text{if } i \in S \\ \mathbf{A}_{i,j} & \text{otherwise} \end{cases}$$

$$\mathbf{b}' = \mathbf{b}_{\text{free}} + \mathbf{v}_{\text{bound}}$$

This method requires changing the right hand side $\mathbf{b}$, as well as the FEM matrix $\mathbf{A}$, to obtain the system $\mathbf{A}'\mathbf{u} = \mathbf{b}'$. There are two problems with this

1. The linear system obtained by simply changing the $i+1$th equation to $u_i = v_i$ for each $i \in S$ is in general not symmetric.

2. For solvers that use a matrix-free multiplication that exploits the structure of $\mathbf{A}$, the matrix $\mathbf{A}'$ does not have the same structure, so the matrix-free multiplication can not be done in the same way.

The first problem is only a problem when conjugate gradient (or another method that depends on the symmetry of the system) is used. The second problem is simpler to solve, and as such, we will address it first. By reordering the equations in the linear system $\mathbf{A}'\mathbf{u} = \mathbf{b}'$ one can see that it is equivalent to $(\mathbf{A}\mathbf{u})_{\text{free}} = \mathbf{b}_{\text{free}}$ and $\mathbf{u}_{\text{bound}} = \mathbf{v}_{\text{bound}}$. So we have

$$\mathbf{A}'\mathbf{x} = (\mathbf{A}\mathbf{x})_{\text{free}} + \mathbf{x}_{\text{bound}}$$

In this way, the multiplication by the matrix $\mathbf{A}'$ can be done in a matrix-free way, by computing $\mathbf{A}\mathbf{x}_{\text{free}}$ and replacing the indices $(\mathbf{A}'\mathbf{x})_i$ that correspond to prescribed degrees of freedom $i \in S$ by the corresponding value x_i in $\mathbf{x}$. Using this method, it is possible to solve a system that implements the boundary conditions, and solve it with BiCGSTAB.

For the conjugate gradient method, however, we need a symmetric linear system[1]. We start with the system $\mathbf{A}'\mathbf{u} = \mathbf{b}'$. This linear system is asymmetric, since for $i \in S$ we have $\mathbf{A}_{i,j} = 0$ for $j \neq i$, but in general we don't have $\mathbf{A}_{j,i} = 0$ for $j \neq i$. This asymmetry can be repaired by 'sweeping' the nonzero elements $\mathbf{A}_{j,i}$ in a way similar to Gaussian elimination. This way we obtain a symmetric system $\mathbf{A}^*\mathbf{u} = \mathbf{b}^*$ where

$$\mathbf{A}^*_{i,j} = \begin{cases} \delta_{i,j} & \text{if } i \in S \text{ or } j \in S \\ \mathbf{A}_{i,j} & \text{otherwise} \end{cases}$$

$$b^*_i = \begin{cases} v_i & \text{if } i \in S \\ b_i - \sum\limits_{k \in S} \mathbf{A}_{i,k} v_k & \text{otherwise} \end{cases}$$

Now, the result of a matrix-free multiplication $\mathbf{A}^*\mathbf{x}$ can be computed by using

$$\mathbf{A}^*\mathbf{x} = (\mathbf{A}\mathbf{x}_{\text{free}})_{\text{free}} + \mathbf{x}_{\text{bound}}$$

Additionally, we need to compute the 'sweeped' right-hand side $\mathbf{b}^*$. This can be done by using

$$\mathbf{b}^* = (\mathbf{b} - \mathbf{A}\mathbf{v}_{\text{bound}})_{\text{free}} + \mathbf{v}_{\text{bound}}$$

In this way, the right-hand side vector $\mathbf{b}$ can be computed using a single matrix-free multiplication.

5.5. Discussion

While the conjugate gradient method does not converge to a good approximation in the one-dimensional case, it converges to some extent for the linear systems that correspond to two-dimensional problems. Just like in the one-dimensional case, the conjugate gradient method is only able to approximate the solution to the linear system up to a certain accuracy. However, this accuracy might be enough, depending on the purpose. In fact, the conjugate gradient method often converges faster to this accuracy than BiCGSTAB. Figure 5.2 illustrates this situation.

[1] Actually, a symmetric positive definite system is needed. We state without proof that the obtained system is positive definite as well, if the matrix $\mathbf{A}$ is symmetric positive definite to begin with.

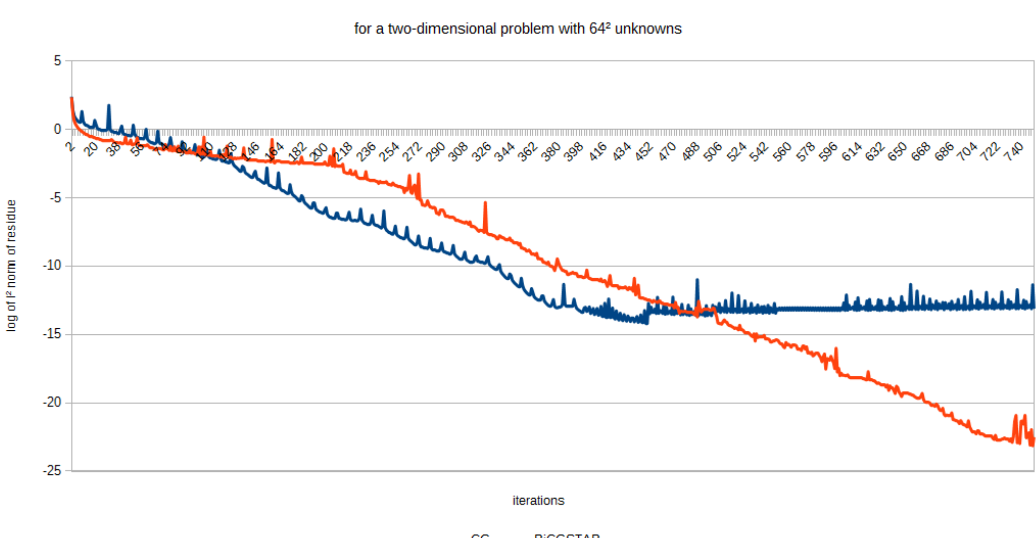


Figure 5.2: The l^2 norm of the residual is plotted as a function of the number of iterations for CG and BiCGSTAB.

It depends on the purpose if the conjugate gradient method provides the necessary accuracy. For this project, it might be necessary to obtain approximations with a higher precision than the conjugate gradient method can offer. For this reason, BiCGSTAB is preferred.

6

Dataflow computing

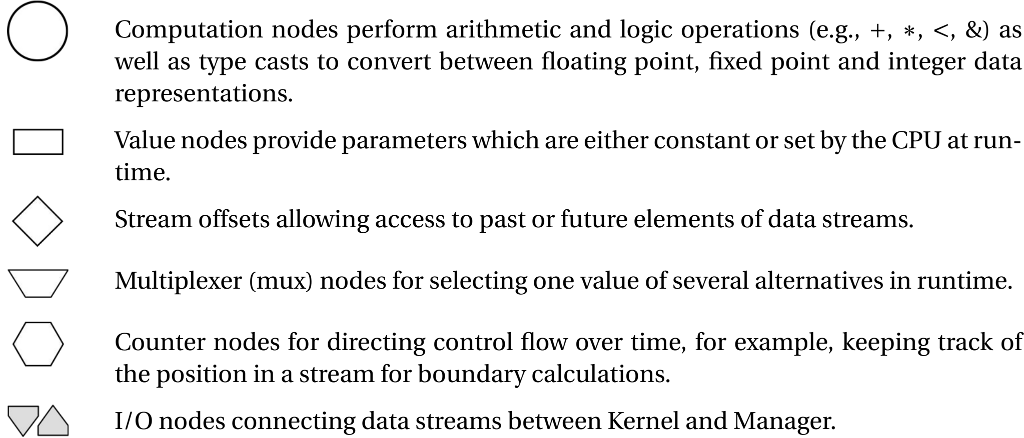
This introduction to dataflow computing is based on [24], a tutorial by Maxeler.

6.1. Introduction

In the most commonly used model of computation, a instruction sequences are executed. While modern processors can execute multiple sequences of instructions in parallel when data and code dependencies allow this, the instructions need to be executed in order, so the computations are inherently sequential. This paradigm of computing is also referred to as *computing in time* or *control-flow*. Historically, there has been a steady rise in the speed in which instructions could be executed, mainly driven by improvements in the manufactoring process in the semiconductor industry. Computations done in the control-flow paradigm directly profit from this increasing performance. However, the physical limits of the increase in speed have been reached, and sequential programs can no longer profit from an exponential increase in execution speed as in the past.

An alternative is *dataflow computing*. In dataflow computing, there is no processor. Instead there are a number of arithmetic and logic units exist which together perform a fixed function. These units are connected in a certain way, and the data 'flows' through the units, toward the final results. The units are implemented in space, so that they can perform the basic operations in parallel. So, in dataflow computing, computing is done in space instead of time. In this way, thousands of parallel operations can be done. This parallelism is the biggest strength of dataflow computing. While the execution speed of hardware is no longer rising as fast, the number of transistors that fit on a given silicon area is still rising exponentially. This means that with each generation of new technology, there is more space to compute in. So, dataflow implementations still benefit from the increase in number of transistors per unit area.

To describe dataflow graphs used in dataflow programming, Maxeler uses a small set of operations, denoted by the following symbols:



Symbol	Description
Circle	Computation nodes perform arithmetic and logic operations (e.g., +, ∗, <, &) as well as type casts to convert between floating point, fixed point and integer data representations.
Rectangle	Value nodes provide parameters which are either constant or set by the CPU at run-time.
Diamond	Stream offsets allowing access to past or future elements of data streams.
Trapezoid	Multiplexer (mux) nodes for selecting one value of several alternatives in runtime.
Hexagon	Counter nodes for directing control flow over time, for example, keeping track of the position in a stream for boundary calculations.
Pentagons	I/O nodes connecting data streams between Kernel and Manager.

Table 6.1: The symbols used in dataflow graphs and their descriptions. Adapted from [24].

Dataflow computing works especially well for programs that involve a large number of operations that do not have a data dependency between them. Take, for example, the task of computing $y_i = x_i^2 + x_i$ to a list of numbers $x_0, x_1, ..., x_{N-1}$. In the control-flow paradigm one typically writes in a high-level programming language that gets compiled to a machine-readable sequence of instructions:

```
for (int i = 0; i < N; i++)
    y[i] = x[i] * x[i] + x[i];
```

One would expect this piece of code to take $O(n)$ operations to complete.

In the dataflow paradigm, one often works from a dataflow graph. The dataflow graph that describes the computation $y = x^2 + x$ is as follows:

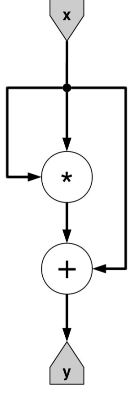
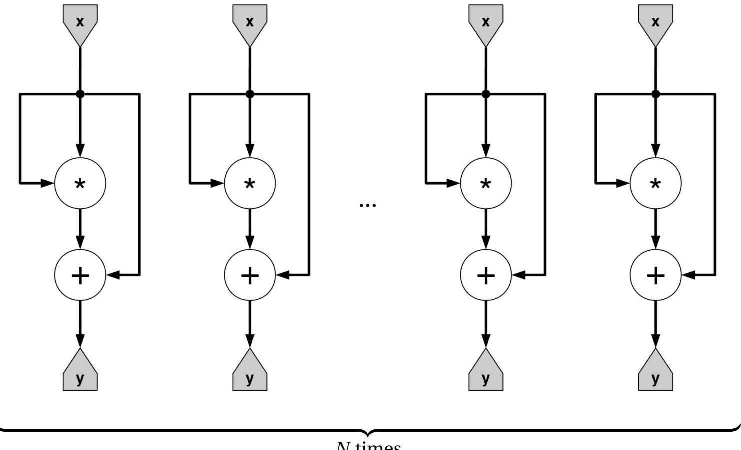


Figure 6.1: The dataflow graph that computes $y = x * x + x$.

This function can then be instantiated N times in space to get the following dataflow structure:



An implementation of this dataflow graph will be able to apply the function to N numbers $x_0, x_1, ..., x_{N-1}$ at the same time, so the time needed is only the time that is needed to perform the multiplication and addition. However, this assumes that N is fixed, there is enough space to implement N instances of the graph 6.1, and that all the numbers x_0, x_1, ..., x_{N-1} are available at the same time.

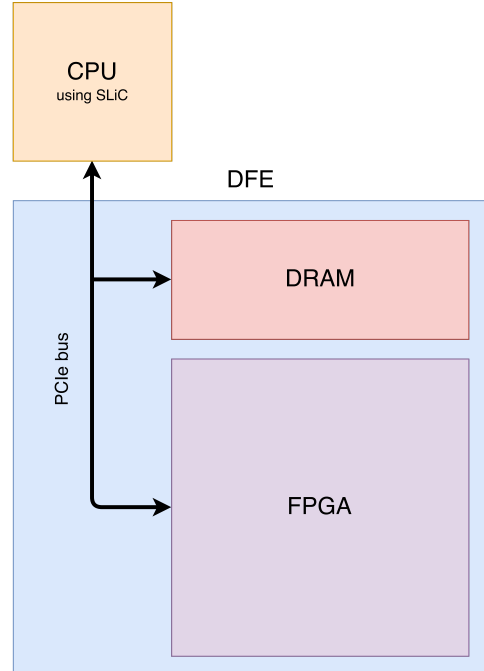
6.2. Environment

In particular, the environment provided by Maxeler Technologies was used. A short description of the environment will be given. For a more thorough introduction, see [24]. Maxeler uses custom hardware called *dataflow engines* that combine reconfigurable hardware and dynamic random-access memory. The in- and outputs of the dataflow engine, as well as the computations that the dataflow engine should perform, are specified in a high-level language called MaxJ, which is an extension of Java. The MaxJ code can be compiled by MaxCompiler to a form that can be synthesized onto the reconfigurable hardware. CPU code can then interface with the dataflow engine by using an interface that is called the simple life computer (SLiC) interface.

On the dataflow engines, a fiel-programmable gate array (FPGA) is used to implement the dataflow graph. FPGAs consists of a large array of interconnected computational units and memory blocks. The interconnections, as well as the computational units and memory blocks, can be reconfigured to implement different functions on the FPGA, and effectively implement entire dataflow graphs. FPGAs are typically programmed by providing a description of the different functions that need to be implemented in a so-called register transfer language like VHDL or Verilog. The translation from the register-transfer language to a binary *bitstream* that can be used to program the FPGA is usually done by vendor-specific, proprietary tools. It is notoriously difficult to use register-transfer language to program an FPGA to perform a non-trivial function. However, since FPGAs are able to compute in space, FPGA implementations can sometimes achieve higher speeds than traditional CPU implementations. Using a higher-level programming language that compiles to register-transfer level to program FPGAs is known as high-level synthesis. High-level synthesis aims to get rid of most of the disadvantages of writing register-transfer language, while retaining most of its advantages.

Hardware

There are different models of the dataflow engines that Maxeler produces. However, from a high-level view, they all have a similar design, and are mostly compatible. The biggest difference is that the newer models contain more and faster dynamic random-access memory (DRAM) the FPGA's have more lookup tables (LUTs) available, and are rated for higher clockspeeds. The FPGA communicates with the CPU and the DRAM via a common pheriperal component interconnect express (PCIe) bus.



Maxeler uses the term large memory (LMEM) for the DRAM, and fast memory (FMEM) for the static random-access memory (SRAM) that is incorporated in the FPGA. The speed at which an FPGA implementation can run depends both on the design and the FPGA. Optimized designs run at about one tenth of the clock speed of that of modern CPUs. To compensate for this, FPGA implementations should be highly parallel before a speedup can be expected. The expected speedup should be high enough to make the increase in engineering time worthwhile. High-level synthesis tools help managing complexity and decrease the time needed for an FPGA implementation, but the time required to implement an FPGA design is still significantly more than the time needed to make a CPU implementation.

6.3. Concepts

The programming environment provided by Maxeler introduces a few concepts, which are briefly explained here.

6.3.1. Kernels

Kernels are the most essential concept that Maxeler has introduced. Kernels correspond to a part of a dataflow graph. The computations implemented in a kernel can be described with a dataflow graph. Kernels typically have a number of streaming inputs and outputs. From the programmer's point of view, the kernel consumes its inputs on each tick. While a kernel can produce a value for each output, this is not enforced. Since kernels only tick when a value is available for each of its inputs, kernels run asynchronously with respect to each other: One kernel might tick, while the other is still waiting until all of the inputs have a value to consume. Multiple kernels can be combined on a single dataflow engine, by connecting inputs and outputs via streams.

6.3.2. Streams

Streams form the connections between kernels, CPU, and DRAM. Since all these components work asynchronously with respect to each other, it is necessary to have some kind of first-in-first-out (FIFO) queue in a stream. Most of the time, streams do their work 'behind the scenes', and the programmer need not to be aware of the details.

6.3.3. Managers

In the manager, one specifies which kernels should be instantiated on a dataflow engine, and how the kernels, the CPU, and the DRAM should be interconnected with streams to perform the desired computation.

6.4. A simple example

We will now show an example of a simple dataflow application, which is adopted from [24]. In the example, the CPU will send a stream of N numbers $x_0, x_1, ..., x_{N-1}$ to the dataflow engine. The dataflow engine then computes a running average over three consecutive numbers and outputs the averages

$$y_i = \frac{x_{i-1} + x_i + x_{i+1}}{3}$$

Two edge cases should be handled differently:

$$y_0 = \frac{x_0 + x_1}{2}$$
$$y_{N-1} = \frac{x_{N-2} + x_{N-1}}{2}$$

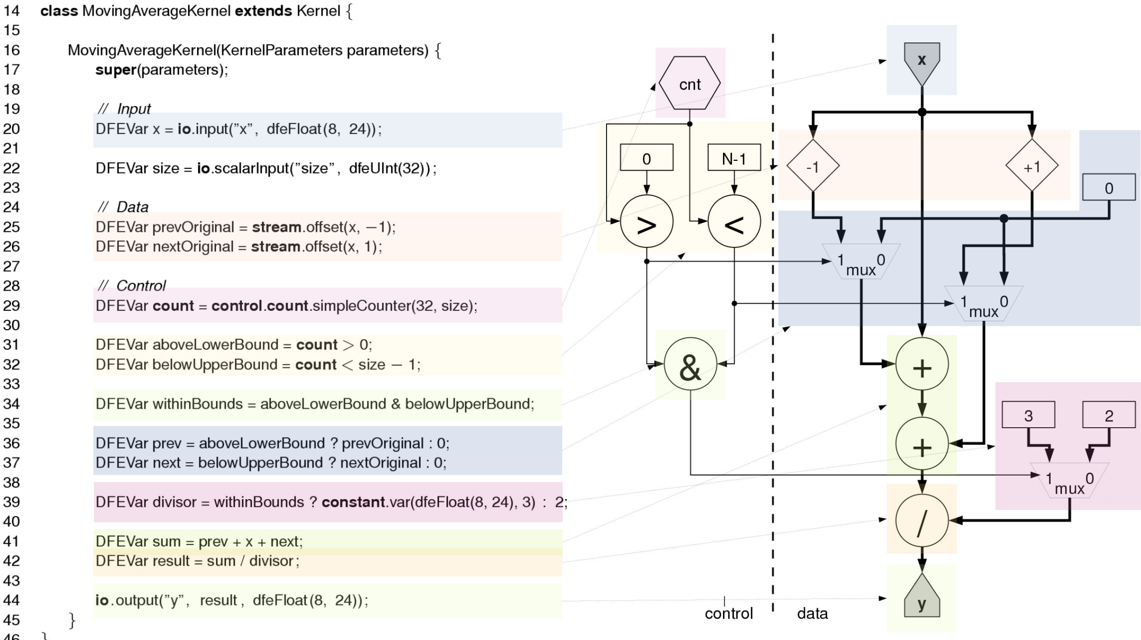


Figure 6.2: An example kernel to compute the running average. This image is taken from [24]

The part of the kernel that is on the left of the dotted line generates control signals that are needed to handle the edge cases, based on a 32-bit counter. The part of the kernel that is on the right selects either selects the previous value of x (or zero, if the counter equals zero) the current value of x, and the next value of x (or zero, if the counter equals $N-1$), and adds the three values together. Then, the sum is divided by three (or two, if the counter equals zero or $N-1$).

To actually run the kernel, it is also necessary to write a manager, which instantiates the kernel on a dataflow machine. Some CPU code which uses the SLiC interface to send data to the dataflow engine is also needed. For testing, it is not necessary to actually have a physical dataflow engine. It is far more convenient to use the simulator capabilities in MaxIDE. This way, one can test the kernel without having to place and route.

```
import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.IOType;

class AverageManager
{
    public static void main(String[] args)
    {
        EngineParameters params = new EngineParameters(args);
        Manager manager = new Manager(params);
        Kernel kernel = new MovingAverageKernel(manager.makeKernelParameters());
        manager.setKernel(kernel);
        manager.setIO(IOType.ALL_CPU);
        manager.createSLiCinterface();
        manager.build();
    }
}
```

In this example, a manager is instantiated with the parameters of the DFE. Then, a new instance of the MovingAverageKernel is instantiated and set on the DFE by the manager. The manager is then configured to allow IO from and to the CPU and to generate the SLiC interface, which can be used from the C code. Finally, the manager is built, which starts the build (either for simulation or for a real DFE, depending on the settings).

7

A matrix-free dataflow implementation

In this chapter, the implementation of a dataflow design that uses isogeometric analysis and weighted quadrature to solve Poisson's problem is described. First, a short summary is given, which captures the necessary information to understand the dataflow implementation without reading all of the preceding chapters.

7.1. Summary

The goal is to approximate the function $u : \Omega \to \mathbb{R}$, which is the solution to Poisson's problem

$$\Delta u = f \quad \text{on } \Omega$$

$$u = u_{\partial\Omega} \text{ on } \partial\Omega$$

Specifically, we use a two-dimensional domain Ω which is topologically isomorphic to a square. Moreover, Ω is defined as the image $\mathrm{Im}(\mathbf{s})$ of a mapping $\mathbf{s} : [0,1]^2 \to \mathbb{R}^2$. By using isogeometric analysis and weighted quadrature, we are able to transform this problem into a linear system

$$\mathbf{Su} = \mathbf{b}$$

so that the elements u_0, u_1, ..., u_{N-1} define an approximation $u^h(\xi,\eta) = \sum_{i=0}^{N-1} \psi_i(\xi,\eta)$. The functions ψ_0, ψ_0, ..., ψ_0 are defined as

$$\psi_k = \phi_k \circ \mathbf{s}^{-1}$$

The functions ϕ_0, ϕ_1, ..., ϕ_{N-1} are defined as a tensor product:

$$\phi_{n_1 i_2 + i_1}(\xi,\eta) = M_{i_1}(\xi) N_{i_2}(\eta)$$

where M_0, M_1, ..., M_{n_1-1} and N_0, N_1, ..., N_{n_2-1} are functions with bounded support. So, we have $N = n_1 n_2$.

To solve the linear system $\mathbf{Su} = \mathbf{b}$, the BiCGSTAB algorithm described in section 5.3 is used. To evaluate matrix-vector products, the matrix-free multiplication described in section 4.4 is used. That is, we have to compute (4.4). This expression is rather complicated, and we will illustrate the matrix-free multiplication for the mass matrix (4.4) instead, which is analogous in structure.

For a vector $\mathbf{y} = \mathbf{Mx}$, where $\mathbf{M}$ is the mass matrix, we have

$$y_i \approx \sum_{s \in G^1_{i_1}} v^{0,0}_{i_1,s} \sum_{t \in G^2_{i_2}} w^{0,0}_{i_2,t} b(\xi^*_s, \eta^*_t)$$

where

$$b(\xi,\eta) := \sum_{j=0}^{N-1} x_j M_{i_1}(\xi) N_{i_2}(\eta) g(\xi,\eta)$$

For the multiplication by the stiffness matrix we have

$$y_i \approx \begin{array}{lll} & \sum_s v^{1,1}_{i_1,s} \sum_t w^{0,0}_{i_2,t} z_{0,0}(\xi^*_s,\eta^*_t) & + \quad \sum_s v^{0,1}_{i_1,s} \sum_t w^{1,0}_{i_2,t} z_{0,1}(\xi^*_s,\eta^*_t) \\ + & \sum_s v^{1,0}_{i_1,s} \sum_t w^{0,1}_{i_2,t} z_{1,0}(\xi^*_s,\eta^*_t) & + \quad \sum_s v^{0,0}_{i_1,s} \sum_t w^{1,1}_{i_2,t} z_{1,1}(\xi^*_s,\eta^*_t) \end{array}$$

where

$$\begin{aligned}
z_{0,0}(\xi^*,\eta^*) &= \mathbf{G}_{0,0}(\xi_s^*,\eta_t^*)\tfrac{\partial z}{\partial \xi}(\xi_s^*,\eta_t^*)\\
z_{0,1}(\xi^*,\eta^*) &= \mathbf{G}_{0,1}(\xi_s^*,\eta_t^*)\tfrac{\partial z}{\partial \eta}(\xi_s^*,\eta_t^*)\\
z_{1,0}(\xi^*,\eta^*) &= \mathbf{G}_{1,0}(\xi_s^*,\eta_t^*)\tfrac{\partial z}{\partial \xi}(\xi_s^*,\eta_t^*)\\
z_{1,1}(\xi^*,\eta^*) &= \mathbf{G}_{1,1}(\xi_s^*,\eta_t^*)\tfrac{\partial z}{\partial \eta}(\xi_s^*,\eta_t^*)
\end{aligned}$$

7.2. Strategy for dataflow implementation

A dataflow design is not as easy to debug or as flexible as a CPU implementation. The dataflow design requires the configuration of hardware resources. If too many hardware resources are needed, measures need to be taken to minimize resource utilization. This can be a hard task and in the worst case it is necessary to start with a new design. To avoid ending up with a non-functional design (either due to bugs or to overutilization of resources), the following structured approach is taken.

1. A CPU implementation is constructed. This serves as a reference for both timing and numerical results. Additionally, it can be used to identify the code that is suitable for a dataflow implementation.

2. Based on the CPU implementation, the part of the code which is to be implemented on a DFE is identified, and a design plan is made.

3. A performance model is made, which estimates both the hardware utilization and the speed of the implementation. For the hardware utilization, the counts of the arithmetic operations that need to be allocated on the dataflow engine needs to be estimated. Additionally, the amount of data that needs to be stored is estimated, and it should be decided on which memory the data is to be stored (FMEM or LMEM). Based on the steps that the design takes and the memory access, the performance of the DFE implementation is estimated.

4. If the hardware utilization is too high (typically this means that the design uses more hardware resources than available on a single dataflow engine), or the performance is not satisfactory, the design is adjusted and the performance model is adapted accordingly.

5. When the estimated hardware utilization and the performance of the DFE are satisfactory, the design can be frozen. Now, a model of the DFE implementation is programmed in C. This is a CPU implementation that closely mimics the structure of the intended DFE implementation. This can help as a reference when testing the DFE implementation.

6. Now, the actual design can be described in MaxJ. The engineering principle of modular development and testing the parts in isolation should be applied, in order to avoid ending up with a large design that is untested and may contain a lot of bugs. Typically, the simulation mode is used to be able to quickly test (portions of) the design. The hardware reference can be used to quickly detect and fix problems in the design.

7. When the design is finished and yields the correct results in simulation, it can be attempted to start a hardware build. In this process, the design is converted to a configuration bitstream for the FPGA, implementing computations on the DFE. This requires vendor-specific synthesis tools to map the specification of the design to the available hardware resources. This is an NP-hard problem. Even when a heuristic is used, it can take a very long time before an acceptable result is acquired, and the synthesis can take up to a full day.

7.3. CPU implementation

The CPU is a straightforward implementation of an IgA code that solves $\alpha\Delta u + \beta u = f$ for some domain $\Omega \subset \mathbb{R}^d$ that is the image of a mapping $\mathbf{s}$. The implementation is restricted to the one- and two-dimensional case ($d = 1,2$), since there is no need to test higher-dimensional cases. The code can be easily adapted to work for three-dimensional testcases as well.

The implementation performs the following steps:

1. Define the boundary value problem that is to be solved (see section 2.1.1).

2. Define the approximation space (see section 2.1.1).

3. Compute weighted quadrature rules for all basis functions and their derivative (see chapter 3 and 4).
4. Assemble the FEM matrix **A** and the right-hand side vector **b** (see chapter 3).
5. Apply the Dirichlet boundary conditions (see section 5.2 and 5.4).
6. Solve the linear system using conjugate gradient, BiCGSTAB, or Gaussian elimination (see chapter 5).
7. The solution of the linear system now defines an approximation u^h to u. Measure the L^2 error $||u - u^h||_{L^2}$ with Gaussian quadrature.

The different steps can be divided into substeps:

1. Define the boundary value problem that is to be solved.
   (a) Choose the dimension $d \in \{1, 2\}$.
   (b) Pick $\alpha, \beta \in \mathbb{R}$ such that α and β are not both zero.
   (c) Choose the function $f \in L^2(\Omega)$.
   (d) Pick the solution $u_{\partial\Omega}$, which is used to measure the error, and to provide Dirichlet boundary conditions.
   (e) Define the domain Ω by picking a mapping $\mathbf{s} : [0, 1] \to \mathbb{R}^2$. The domain is $\Omega := \mathrm{Im}(s)$.
2. Define the approximation space. For an accurate approximation, the approximation space needs to be not more continuous than the mapping **s**. In practice, **s** is usually defined as a B-spline mapping, and appropriate continuity is ensured by only applying h- and p-refinement.
   (a) Define the knot vector $\boldsymbol{\Xi}$ of degree p_1 that define the basis functions M_0, M_1, ..., M_{n_1-1}. For $d = 1$, the FEM basis functions are $\phi_k = M_k \circ \mathbf{s}^{-1}$.
   (b) For $d = 2$, it is necessary to define another knot vector **H** of degree p_2 which defines the B-spline basis functions N_0, N_1, ..., N_{n_2-1}, and the FEM basis functions are $\psi = \phi \circ \mathbf{s}^{-1}$, where $\phi_{n_1 k_2 + k_1}(\xi, \eta) := M_{k_1}(\xi) N_{k_2}(\eta)$ for $k_1 = 0, 1, ..., n_1 - 1$, and $k_2 = 0, 1, ..., n_2 - 1$.
3. Compute weighted quadrature rules for all basis functions and their derivative.
   (a) Define the knot vector(s) that define the basis functions of the projection space. If **s** is C^∞-continuous or $\alpha = 0$, this can simply be the same as the approximation space. Otherwise, the projection space needs to be adapted to have at most the same continuity as the geometric factor on the points where **s** is not C^∞-continuous. It is sufficient to lower the continuity of the knots corresponding to these points by one.
   (b) Using the knot vector(s) defined in 3a, define a grid of global quadrature points in each dimension.
   (c) For each dimension, evaluate all nonzero basis functions of the projection space and their derivatives at both the Gaussian quadrature points and the quadrature points which are defined in 3b.
   (d) For each dimension, for each basis function and its derivative, use the computed values to generate a linear system for the weights, using the values evaluated in 3c. Solve this linear system for the weights (if necessary, use a QR-decomposition), and store them.
4. Assemble the FEM matrix **A** and the right-hand side vector **b**.
   (a) Evaluate the B-spline basis functions at the weighted quadrature points, and the mapping basis functions at the weighted quadrature points and the Gaussian quadrature points.
   (b) Using the values computed in step 4a, evaluate $f \circ \mathbf{s}$, and the geometric factor g at each Gaussian quadrature point, and evaluate the geometric factors g and **G** at each weighted quadrature point.
   (c) Using the values computed in step 4b, assemble the right-hand side vector **b** by computing and storing $b_k = \sum_{(\xi_s^*, \eta_t^*) \in \mathrm{supp}(\phi_i)} v_s w_t \phi_i(\xi_s^*, \eta_t^*) g(\xi_s^*, \eta_t^*) \approx \int_\Omega f(\mathbf{x}) \psi(\mathbf{x})\, \mathrm{d}\mathbf{x}$ for $k = 0, 1, ..., N - 1$.
   (d) Assemble the FEM matrix **A** by computing the elements of the mass matrix (see (3.3.2)) and stiffness matrix (see (3.3.2)), and use that the FEM matrix is given by $\mathbf{A}_{i,j} = -\alpha\mathbf{S} + \beta\mathbf{M}$, as derived in section 2.1.2. Store the FEM matrix in a sparse format.

5. Apply the Dirichlet boundary conditions using either L^2 projection or interpolation.
   (a) Set up the linear system for finding the coefficients by L^2 projection or interpolation.
   (b) Adapt the linear system to fix the degrees of freedom with support on the boundary.
   (c) If the conjugate gradient method is used as a solver, it is necessary to symmetrize the matrix.
6. Solve the linear system using the conjugate gradient method, the BiCGSTAB method, or Gaussian elimination. For the iterative methods, a stoppping tolerance ϵ has to be chosen.
7. The solution of the linear system $\mathbf{u} = (u_0, u_1, ..., u_{N-1})^\top$ now defines an approximation $u^h \approx u$ by $u^h = \sum_{k=0}^{N-1} u_k \psi_k$. Approximate the L^2 error $||u - u^h||_{L^2} = \sqrt{\int_\Omega (u-u)^2 \, d\Omega}$ with Gaussian quadrature.

It should be noted that the CPU implementation is not optimized for performance. It is meant to be a clear reference and to experiment with different parameters and settings. The experimental nature of weighted quadrature made this necessary: a lot of changes have been made as a reaction to the issues with weighted quadrature.

The initial code was done in MATLAB, because MATLAB natively supports many matrix operations and decompositions. However, a C(++) implementation is preferred because it resembles the structure of the dataflow engine more closely, and interfaces with DFEs more efficiently. The code also includes functions for the efficient evaluation of B-splines, matrix and vector operations, as well as a QR-decomposition, a singular value decomposition, code to prescribe values in a linear system, and code to symmetrize the matrix. This has made the codebase considerably larger than the MATLAB implementation.

7.4. Analysis of data

Remark 5. *As before, we will denote the B-spline basis functions which are used in the B-spline mapping by $\tilde{M}_0, \tilde{M}_1, ..., \tilde{M}_{\tilde{n}_1-1}$ for the first dimension, and $\tilde{N}_0, \tilde{N}_1, ..., \tilde{N}_{\tilde{n}_2-1}$ for the second dimension, so that the total number of bivariate basis functions is $\tilde{n}_1 \tilde{n}_2$. Further, we will assume that these basis functions have polynomial degree $\tilde{p}$. For the B-spline basis functions which are used in the definition of the FEM basis functions, the notation $M_0, M_1, ..., M_{n_1-1}$ will be used for the first dimension, and $N_0, N_1, ..., N_{n_2-1}$ for the second dimension. The polynomial degree of these basis function is assumed to be p. The number of basis functions is then $N = n_1 n_2$, which is also the number of degrees of freedom.*

We will now make some assumptions on the size of the data. Running an algorithm on the dataflow engine incurs some overhead. Since the two-dimensional assembly can be performed quite fast, we will assume that the number of basis functions is quite large, say $100 < n < 1000$, to justify the need for dataflow acceleration. In particular, we will assume that the number of basis functions is approximately equal to n in both dimensions: $n_1, n_2 \approx n$, so that $N = n_1 n_2 \approx n^2$. Furthermore, the number of FEM basis functions is much larger than the polynomial degrees, and the number of basis functions used in the B-spline mapping: $\tilde{n}_1, \tilde{n}_2, \tilde{p}, p \ll n$. It follows that we also have $n \ll N$.

The BiCGSTAB algorithm needs to store the elements of the vectors. Since BiCGSTAB uses 8 vectors, and each vector is of size N, there are $8N$ values that need to be stored. Dataflow engines have a couple of megabyte FMEM storage capacity. If we assume that $n_1, n_2 = 1024$ and that values are stored using 48 bits (or 6 bytes), this gives $1024^2 \cdot 8 \cdot 6 = 6$ MB. If we consider the other values that need to be stored in FMEM, this will probably not fit, so the vectors need to be stored in LMEM.

On the other hand, to assemble the matrix we need to have values at the quadrature points. There are approximately $2n$ quadrature points in each dimension, so that the two-dimensional grid of quadrature points contains $(2n)^2 = 4N$ quadrature points. For the one-dimensional quadrature points, we need to store all nonzero values of univariate basis functions, their derivatives, and four weights. So, there are $2n$ quadrature points in each dimension, and $6(p+1)$ values that need to be stored, which totals to $24(p+1)n$ values. For $n \approx 1000$, we can still store this in FMEM.

Then, we have the geometric factor $\mathbf{G}$, which is different at each tensor product quadrature point. The geometric factor does not have a nice tensor product structure, but needs a matrix inversion to be evaluated. For this, a division is necessary, which makes evaluating the geometric factor quite expensive in terms of hardware resources. Since there are $4N$ quadrature points and the geometric factor is a matrix which contains 4 elements, there are $16N$ values which need to be stored for the geometric factor. On the other hand, if we want to compute the geometric factor on-the-fly, we need the values of the mapping basis functions and

their derivative at each one-dimensional quadrature point. There are $\tilde{p}+1$ nonzero basis functions at each quadrature point, and there are $2n$ quadrature points in each dimension, which totals to $8n(\tilde{p}+1)$ values in total.

7.5. Design

We will now sketch the design of the DFE implementation.

From a high-level view, we need to implement the various steps of the BiCGSTAB algorithm, and map them onto a dataflow engine in an efficient way. Adapting a slightly more low-level view, the different vector and matrix operations need to be implemented. The most complex operation is the matrix multiplication.

7.5.1. BiCGSTAB

Consider the inner loop of BiCGSTAB in algorithm 4. There are two dot products. Since the operations after the inner product depend on the dot product, the computation of the dot products needs to finish completely before the rest of the computations can be started. Further, the matrix multiplication can not be easily pipelined with the vector operations, since the first element y_0 of a matrix-vector product $\mathbf{y} = \mathbf{M}\mathbf{x}$ can only be computed when enough elements of the vector $\mathbf{x}$ are known:

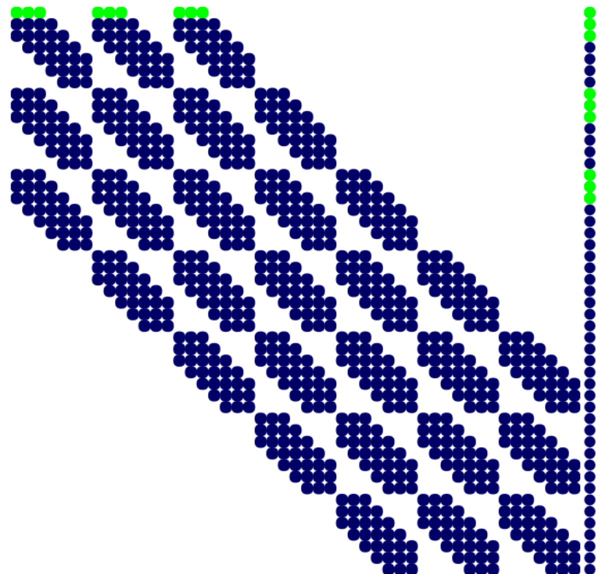
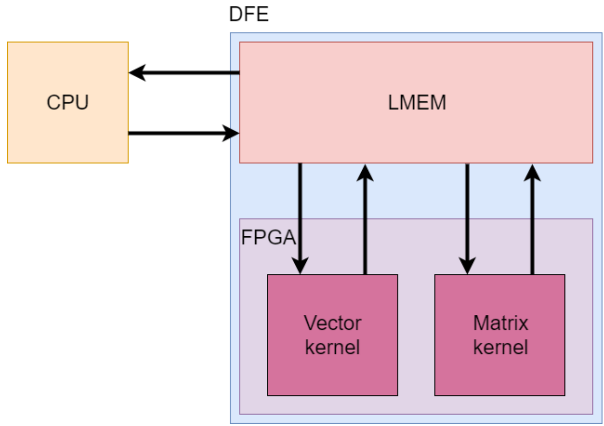


Figure 7.1: The sparsity pattern of the vector and matrix. The elements which are needed to compute the first element of the matrix-vector product are highlighted. To compute the first entry of the matrix-vector product $\mathbf{Mx}$, elements of $\mathbf{x}$ with high indices are needed, which makes it hard to pipeline the matrix multiplication with vector operations.

So, we have to split up the inner loop of BiCGSTAB in five phases that can be executed on the dataflow engine:

Phase	Operations
1	1: $\mathbf{p} = \mathbf{r} + \beta(\mathbf{p} - \omega\mathbf{v})$
2	2: $\mathbf{v} = \mathbf{A}\mathbf{p}$ 3: compute $\mathbf{r_0} \cdot \mathbf{v}$
3	4: $\mathbf{h} = \mathbf{x} + \alpha\mathbf{p}$ 5: $\mathbf{s} = \mathbf{r} - \alpha\mathbf{v}$
4	6: $\mathbf{t} = A\mathbf{s}$ 7: compute $\mathbf{s} \cdot \mathbf{t}, \mathbf{t} \cdot \mathbf{t}$
5	8: $\mathbf{x} = \mathbf{h} + \omega\mathbf{s}$ 9: $\mathbf{r} = \mathbf{s} - \omega\mathbf{t}$ 10: compute $\mathbf{r_0} \cdot \mathbf{r}, \mathbf{r} \cdot \mathbf{r}$

Part 1, 3, and 5 perform vector operations. For efficiency, they can be implemented as a single kernel which can perform the necessary operations. Part 2 and 4 can use a single kernel which performs the matrix multiplication and dot product. To compensate for the overhead of running on a dataflow engine, the design should use a fairly large number of basis functions. This means that the vectors are too big to fit in the static memory on the FPGA and should be stored in the LMEM. The kernels can now be combined in the following way:



The stream between the CPU and LMEM is needed to set vectors, and load them when the BiCGSTAB algorithm has completed. The vector kernel and matrix kernel need to load vectors, perform operations, and write back the results. So, a single arrow in the diagram might represent multiple memory streams.

7.5.2. Matrix multiplication

The original idea was to assemble one row at a time and calculate the inner product of this row and used sum-factorization to assemble a whole row of the matrix at once. It is about $p+1$ times more efficient to use the matrix-free multiplication which is derived in section 4.4. The method of assembling the matrix row-wise is not used anymore. Therefore, it will not be considered here.

The matrix multiplication computes the matrix-vector product $\mathbf{y} = \mathbf{Sz}$. From (4.4) we see that four summations have to be evaluated:

$$a_i \approx \begin{array}{lclcl} & \sum_s v^{1,1}_{i_1,s} \sum_t w^{0,0}_{i_2,t} z_{0,0}(\xi^*_s,\eta^*_t) & + & \sum_s v^{0,1}_{i_1,s} \sum_t w^{1,0}_{i_2,t} z_{0,1}(\xi^*_s,\eta^*_t) \\ + & \sum_s v^{1,0}_{i_1,s} \sum_t w^{0,1}_{i_2,t} z_{1,0}(\xi^*_s,\eta^*_t) & + & \sum_s v^{0,0}_{i_1,s} \sum_t w^{1,1}_{i_2,t} z_{1,1}(\xi^*_s,\eta^*_t) \end{array} \tag{7.1}$$

The limits of the summation are omitted to keep the formula readable, but the summation ranges from $s = q^1_{i_1}$ to $q^1_{i_1} + m^1_{i_1} - 1$ in the first dimension and from $t = q^2_{i_2}$ to $q^2_{i_1} + m^2_{i_2} - 1$ in the second dimension. By a bound from section 4.1.3, the summations are over a maximum of $3p$ quadrature points per dimension, or over $9p^2$ tensor product quadurature points. So, the weights do only need to be stored for at most $3p$ quadrature points per dimension. In addition, the values $z_{0,0}(\xi^*_s,\eta^*_t)$, $z_{0,1}(\xi^*_s,\eta^*_t)$, $z_{1,0}(\xi^*_s,\eta^*_t)$, and $z_{1,1}(\xi^*_s,\eta^*_t)$ are needed for each of the at most $9p^2$ quadrature points (ξ^*_s,η^*_t).

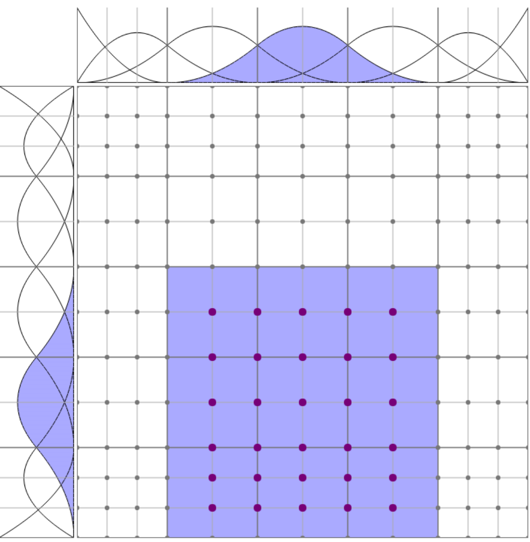


Figure 7.2: The support of $\phi_{15}(\xi)(\eta) = M_1(\xi)N_2(\eta)$ is shown on the parametric space. It can be seen that the number of quadrature points inside the support is less than $3p = 6$ in each dimension.

Now, if we assume that we succesively compute y_0, y_1, ..., y_{N-1} in this way, many values can be re-used between computations. In fact, most of the data in the window can be re-used between the computation of two sequential degrees of freedom, since two succesive basis functions ϕ_{k-1} and ϕ_k share many quadrature points:

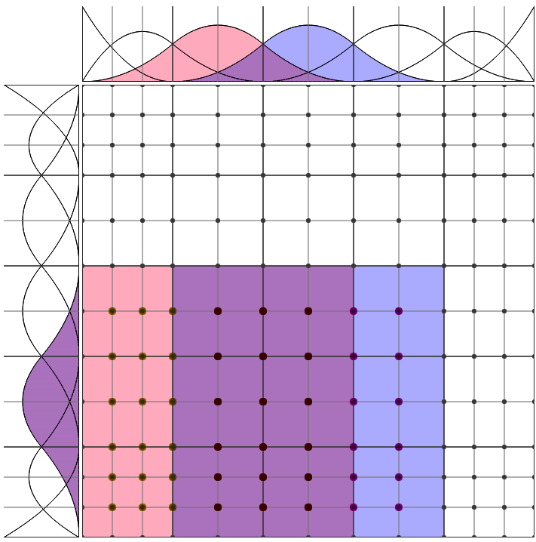


Figure 7.3: The support of $\phi_{16}(\xi,\eta) = M_2(\xi)N_2(\eta)$ is shown in red, and the support of $\phi_{17}(\xi,\eta) = M_3(\xi)N_2(\eta)$ is shown in blue. It can be seen that there is only a column of 6×2 quadrature points that is inside the support of ϕ_{17} but not inside the support of ϕ_{16}.

Since the values corresponding to these quadrature points was used in the computation of the last degree of freedom, these are still held in registers. For this reason, the values $b_{a,b}$ for $a, b = 0, 1$ can be held in $3p$ shift registers, each with size $3p$. If there are q quadrature points in the first dimension, the values in the shift register need to be shifted q places to the left to make space for the new values. Per dimension, the number of quadrature points per element is stored in an array in FMEM.

A similar situation holds for the basis functions ϕ_{k-n_1} and ϕ_k:

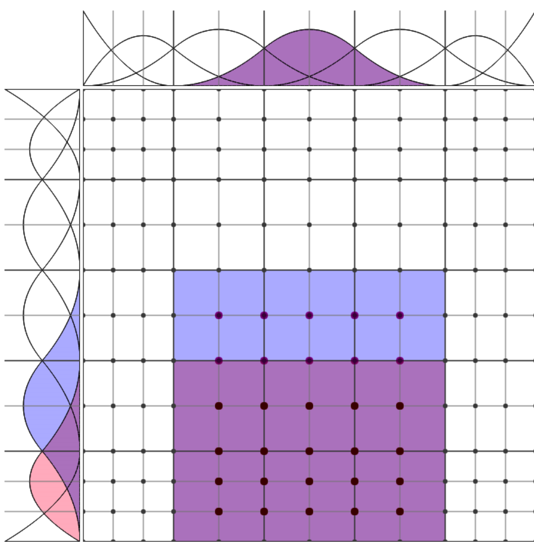


Figure 7.4: The support of $\phi_{10}(\xi,\eta) = M_3(\xi)N_1(\eta)$ is shown in red, and the support of $\phi_{17}(\xi,\eta) = M_3(\xi)N_2(\eta)$ is shown in blue.

However, the values $z_{0,0}(\xi^*,\eta^*)$, $z_{0,1}(\xi^*,\eta^*)$, $z_{1,0}(\xi^*,\eta^*)$, $z_{1,1}(\xi^*,\eta^*)$ were not used in the last computation, but in a computation n_1 cycles earlier. If these values would be held in shift registers, there would need to be n_1 of these shift registers. This is not very efficient. A better way is to allocate some FMEM and use it as a buffer for the $3p$ horizontal rows for which the quadrature points need to be stored.

So, using all values that were already computed before, the functions $z_{0,0}$ $z_{0,1}$, $z_{1,0}$, $z_{1,1}$, only need to be evaluated for the quadrature points in the top-right element. This is convenient, since for the quadrature points in the same element the same basis functions are nonzero. In order to compute $z_{0,0}$ $z_{0,1}$, $z_{1,0}$, $z_{1,1}$, we first compute $\frac{\partial z}{\partial \xi}$, $\frac{\partial z}{\partial \eta}$, and the elements $\mathbf{G}_{0,0}$, $\mathbf{G}_{0,1}$, $\mathbf{G}_{1,0}$, $\mathbf{G}_{1,1}$ of the geometric factor $\mathbf{G}$. Computing the value of $\frac{\partial z}{\partial \xi}(\xi^*,\eta^*)$, $\frac{\partial z}{\partial \eta}(\xi^*,\eta^*)$ is a matter of keeping the degrees of freedom corresponding to basis functions that are nonzero on (ξ^*,η^*) in registers, multiplying the nonzero basis functions (or their derivative) by the corresponding degrees of freedom, and adding them. Assuming that the univariate basis functions M_i, M_{i+1}, ..., M_{i+p} are nonzero on ξ^*, and the univariate basis functions N_j, N_{j+1}, ..., N_{j+p} are nonzero on η^*, we can express the idea in code as

```
float[] computepartials(float[] M, float[] dMdxi,
    float[] N, float[] dNdeta, float[][] z, int p)
{
    float dzdxi = 0, dzdeta = 0;
    for (int k2 = 0; k2 < p + 1; k2++)
        for (int k1 = 0; k1 < p + 1; k1++)
        {
            dzdxi += z[k1][k2] * dMdxi[k1] * N[k2];
            dzdeta += z[k1][k2] * M[k1] * dNdeta[k2];
        }
    return { dzdxi, dzdeta };
}
```

The computation of the entries of the Jacobian matrix $D\mathbf{s}$ can be done in a way that is similar to the evaluation of the partial derivatives. However, the Jacobian matrix is twice as expensive to evaluate, since it contains four entries, while there are only two partial derivatives. In (2.3.1), the geometric factor is defined as

$$\mathbf{G}(\xi) = |\det(D\mathbf{s}(\xi))|\,(D\mathbf{s}(\xi))^{-\top}(D\mathbf{s}(\xi))^{-1}$$

So for the evaluation of the geometric factor, it is necessary to evaluate the Jacobian matrix $D\mathbf{s}$ of the mapping $\mathbf{s}$. The Jacobian consists of 4 elements. For example, for the first element of the Jacobian matrix $D\mathbf{s}$, we have

$$(D\mathbf{s}(\xi^*,\eta^*))_{0,0} = \sum_{k=0}^{N-1} (\mathbf{c}_k)_0 \frac{\partial \phi}{\partial \xi}(\xi^*,\eta^*)$$

where the notation $(\mathbf{c}_k)_0$ is used to denote the first coordinate of the control point $\mathbf{c}_k$. Using property 4 from section 2.2.2, we see that there are only $(\tilde{p}+1)(\tilde{p}+1)$ nonzero basis functions ϕ_k. By using $\phi_k(\xi,\eta) = \tilde{M}_{k_1}(\xi)\tilde{N}_{k_2}(\eta))$, it can be seen that we have a sum as in (7.6.1) where $M = \tilde{p}+1$. So, the evaluation of the Jacobian matrix costs $4((\tilde{p}+1)^2-1)$ additions and $4(\tilde{p}+1)(\tilde{p}+2)$ multiplications.

Now define $\mathbf{A} := D\mathbf{s}(\xi,\eta)$. We then have

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})}\begin{pmatrix} \mathbf{A}_{1,1} & -\mathbf{A}_{0,1} \\ -\mathbf{A}_{1,0} & \mathbf{A}_{0,0} \end{pmatrix}$$

By expanding $\mathbf{G} = |\det(\mathbf{A})|\mathbf{A}^{-\top}\mathbf{A}^{-1}$, we get

$$\mathbf{G} = \frac{|\det(\mathbf{A})|}{\det(\mathbf{A})^2}\begin{pmatrix} \mathbf{A}_{1,1} & -\mathbf{A}_{1,0} \\ -\mathbf{A}_{0,1} & \mathbf{A}_{0,0} \end{pmatrix}\begin{pmatrix} \mathbf{A}_{1,1} & -\mathbf{A}_{0,1} \\ -\mathbf{A}_{1,0} & \mathbf{A}_{0,0} \end{pmatrix}$$

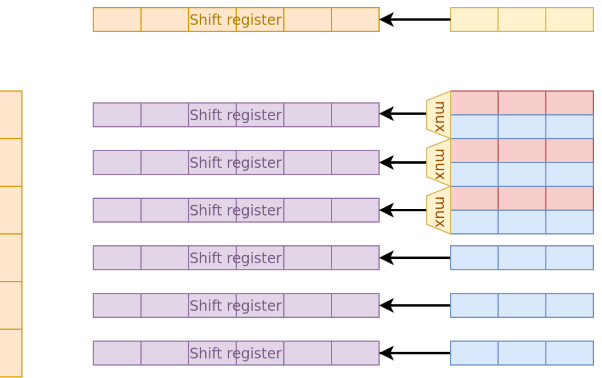
$$= \frac{1}{|\det(\mathbf{A})|}\begin{pmatrix} \mathbf{A}_{1,1}^2 + \mathbf{A}_{1,0}^2 & -(\mathbf{A}_{1,1}\mathbf{A}_{0,1} + \mathbf{A}_{1,0}\mathbf{A}_{0,0}) \\ -(\mathbf{A}_{1,1}\mathbf{A}_{0,1} + \mathbf{A}_{1,0}\mathbf{A}_{0,0}) & \mathbf{A}_{0,1}^2 + \mathbf{A}_{0,0}^2 \end{pmatrix}$$

So, in code the evaluation of the geometric factor can be done as

```
float[2][2] geometricfactor(float A[2][2])
{
    float absdetinv = 1 / abs(A[0][0] * A[1][1] - A[1][0] * A[0][1]);
    float symm = -(A[1][1] * A[0][1] + A[1][0] * A[0][0]) * absdetinv;
    return {
             { (A[1][1] * A[1][1] + A[1][0] * A[1][0]) * absdetinv, symm },
             { symm, (A[0][1] * A[0][1] + A[0][0] * A[0][0]) * absdetinv }
           };
}
```

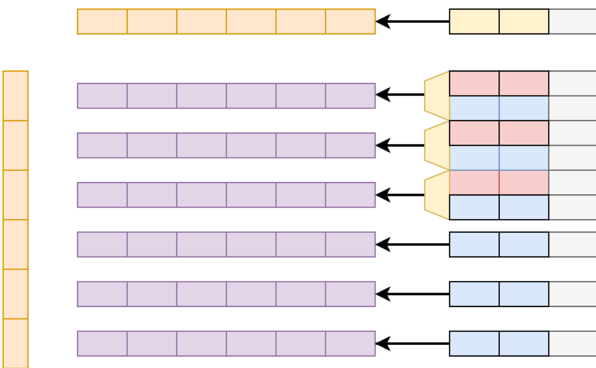
Once $\frac{\partial z}{\partial \xi}(\xi^*,\eta^*)$, $\frac{\partial z}{\partial \eta}(\xi^*,\eta^*)$ and the elements of the geometric factor are computed for each new quadrature point (ξ^*,η^*), it is trivial to compute $z_{0,0}(\xi^*,\eta^*) = \mathbf{G}_{0,0}(\xi^*,\eta^*)\frac{\partial z}{\partial \xi}(\xi^*,\eta^*)$, $z_{0,1}(\xi^*,\eta^*) = \mathbf{G}_{0,1}(\xi^*,\eta^*)\frac{\partial z}{\partial \eta}(\xi^*,\eta^*)$, $z_{1,0}(\xi^*,\eta^*) = \mathbf{G}_{1,0}(\xi^*,\eta^*)\frac{\partial z}{\partial \xi}(\xi^*,\eta^*)$, and $z_{1,1}(\xi^*,\eta^*) = \mathbf{G}_{1,1}(\xi^*,\eta^*)\frac{\partial z}{\partial \eta}(\xi^*,\eta^*)$.

Once these values are computed and the values of $z_{0,0}$, $z_{0,1}$, $z_{1,0}$, $z_{1,1}$ that were computed earlier are loaded from FMEM, the values can be placed in shift registers:

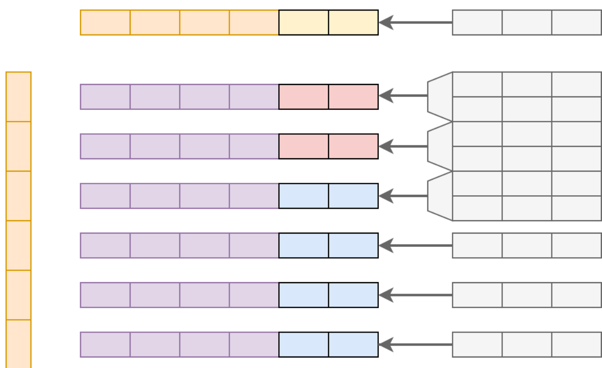


A single red, blue, or purple block represents the values $z_{0,0}(\xi^*,\eta^*)$, $z_{0,1}(\xi^*,\eta^*)$, $z_{1,0}(\xi^*,\eta^*)$, $z_{1,1}(\xi^*,\eta^*)$ for a quadrature point (ξ^*,η^*). The red blocks represent values that are computed, while the blue values represent values that were used before and were loaded from the buffer, and the purple values represent the values that were used in the last cycle, and are still present in registers. This diagram is for $p = 2$, so the grid has size $3p \times 3p = 6 \times 6$, and there are at most $p+1 = 3$ new quadrature points. The orange blocks on the side and top represent weights for a quadrature point, and the yellow blocks in the top-right corner represent the weights that correspond to the new quarature points.

Suppose, as an example, that the new element contains a grid of 2×2 quadrature points. Then, the two multiplexers on the top load the newly computed values which are represented, while the lowest multiplexer is set to load the value(s) that was loaded from the buffer into the shift registers:



The shift registers then shift the weights and the values of $z_{0,0}$, $z_{0,0}$, $z_{0,0}$, $z_{0,0}$ which were used in the last cycle two places, and shift in the new values and weights.



Finally, the values $z_{0,0}(\xi^*,\eta^*)$, $z_{0,1}(\xi^*,\eta^*)$, $z_{1,0}(\xi^*,\eta^*)$, $z_{1,1}(\xi^*,\eta^*)$ are multiplied by the weights and summed. In this way, the new degree of freedom is computed. The summations can be split over multiple cycles, but we will assume now that the four summations are all done simultaneously in a single cycle.

The next cycle, the same steps are repeated and the next degree of freedom of the matrix-vector product is computed.

7.6. Performance model

For the performance model, we estimate the hardware utilization and the performance of the dataflow implementation, to see if a dataflow implementation is feasible and worth the effort. It should be noted that the CPU implementation that was made as a reference is used as a reference for timing as well. This is not ideal, since this implementation is not optimized. In particular, an implementation that uses multiple cores and vector processing is expected to be a lot faster. Such an optimized implementation of an IgA code that uses weighted quadrature is outside the scope of this project. So, the actual speedup might be a lot lower: this is impossible to say without making such an optimized implementation,

7.6.1. Hardware utilization

To estimate the utilization of hardware resources, we count the arithmetic operations in the design. Then, we can simply multiply the number of operations of each type by the number of resources that an operation of that type takes. The final number is then compared with the number of resources on the dataflow engine. It should be noted that the estimate obtained this way is an optimistic estimate, in the sense that the actual hardware utilization will usually be higher. This is due to the hardware resources for factors that are not considered here (for example, the FIFO queues which are used in the implementation of streams, the non-arithmetic operations, and the hardware resources that are needed to pipeline the design).

For the implementation of this design, two DFEs are considered. The first is a fourth generation DFE, which is referred to as MAX4. The other one is a new DFE of the fifth generation, MAX5.

Vector operations
The hardware resources for the vector operations are relatively simple to estimate. A generic kernel that is able to compute $\mathbf{v_1} + c_1\mathbf{v_2}$, $\mathbf{v_1} + c_1\mathbf{v_2}$ and their sum is implemented. This is enough to handle all the vector operations in the BiCGSTAB algorithm. We will compute n_{vec} elements of these vectors at the same time to speed up the computations. In addition, it is needed to calculate 2 dot products. The dot products are implemented with adders. Floating-point adders have a high latency, and in practice an adder tree of 5 adders is used in the computation of the dot product. We will not elaborate on this, but simply use this figure.

By simply counting the number of operations that need to be done, we see that $5n_{\text{vec}} + 8$ additions need to be done, as well as $4n_{\text{vec}}$ multiplications.

Matrix multiplication
It is often necessary to evaluate a sum of the form

$$\sum_{i=0}^{M-1}\sum_{j=0}^{M-1} a_i b_j c_{i,j} \tag{7.2}$$

By adapting a sum-factorization-like approach, such a sum can be evaluated in the following way:

```
float accumulate(float a[M], float b[M], float c[M][M])
{
    float acc = 0;
    for (int i = 0; i < M; i++)
    {
        float acc_j = 0;
        for (int j = 0; j < M; j++)
            acc_j += c[i][j] * b[j];
        acc += a[i] * acc_j;
    }
    return acc;
}
```

This way, the evaluation can be done in $M^2 - 1$ additions[1] and $M(M+1)$ multiplications.

The evaluation of the geometric factor is done in 4 additions, 11 multiplications, and 1 division (in addition to the operations that are necessary for the evaluation of the Jacobian matrix). To evaluate the summations (7.5.2) it is still necessary to evaluate the partial derivatives of the function that is associated to the vector $\mathbf{z}$. The derivatives can be computed in a way analogous to the computation of the derivatives in the Jacobian. So, computing the partial derivatives can be done in $2((p+1)^2 - 1)$ additions and $2(p+1)(p+2)$ multiplications.

Finally, for each quadrature point (ξ^*, η^*) we combine the geometric factor and the partial derivatives of z to obtain $\mathbf{G}_{0,0}(\xi^*,\eta^*)\frac{\partial z}{\partial \xi}(\xi^*,\eta^*)$, $\mathbf{G}_{0,1}(\xi^*,\eta^*)\frac{\partial z}{\partial \eta}(\xi^*,\eta^*)$, $\mathbf{G}_{1,0}(\xi^*,\eta^*)\frac{\partial z}{\partial \xi}(\xi^*,\eta^*)$, and $\mathbf{G}_{1,1}(\xi^*,\eta^*)\frac{\partial z}{\partial \eta}(\xi^*,\eta^*)$. These are the values that are needed in (7.5.2). This is done in 4 multiplications.

Now assume that it is necessary to evaluate these values for n_{qp} quadrature points at once. Then, the evaluation of the geometric factor costs $4n_{\text{qp}}(\tilde{p}+1)^2$ additions, $4n_{\text{qp}}((\tilde{p}+1)(\tilde{p}+2)+11)$ multiplications, and n_{qp} divisions. Evaluating the values $\mathbf{G}_{0,0}(\xi^*,\eta^*)\frac{\partial z}{\partial \xi}(\xi^*,\eta^*)$, $\mathbf{G}_{0,1}(\xi^*,\eta^*)\frac{\partial z}{\partial \eta}(\xi^*,\eta^*)$, $\mathbf{G}_{1,0}(\xi^*,\eta^*)\frac{\partial z}{\partial \xi}(\xi^*,\eta^*)$, and $\mathbf{G}_{1,1}(\xi^*,\eta^*)\frac{\partial z}{\partial \eta}(\xi^*,\eta^*)$ costs $2n_{\text{qp}}((p+1)^2 - 1)$ additions and $2n_{\text{qp}}((p+1)(p+2)+2)$ multiplications (assuming the geometric factors are already computed or streamed from memory).

Finally, the computation of a single element a_i of the vector $\mathbf{a}$ requires evaluating (7.5.2). These summations are over at most $3p \times 3p$ quadrature points, so each of them can be computed in $9p^2 - 1$ additions and $3p(3p+1)$ multiplications. Since there are four of these summations which need to be added, the evaluation of (7.5.2) can be done in $4(9p^2 - 1) + 3$ additions and $12p(3p+1)$ multiplications.

7.6.2. Performance

The performance of the design is estimated by applying some simple principles. A design is said to be *compute bound* if the speed of execution is bounded by the computations that need to be done. On the other

[1] In the code, $M(M+1)$ additions are used, but there are $M+1$ additions that add some value to 0, and this addition can be eliminated in hardware (in software too, but this creates very ugly code).

hand, if the speed is limited by the speed of the memory, the design is said to be *memory bound*. In reality the situation might be more complicated, but the results of this model are usually accurate. We will simply define the memory time as the size of the memory that needs to be loaded, divided by the speed of the memory, and the compute time as the number of cycles times the duration of a cycle. We will approximate the execution time by taking the longest of these two times. If the former is the longest, we say the design is memory bound, else we say the design is compute bound. We will handle the steps with vector operations separately from the steps with a matrix multiplication.

To estimate the performance, we will assume that there is a global clock, as common in hardware design. The clock 'ticks' at a certain frequency, and the time t_{cycle} between two ticks is called a clock cycle. We will assume that every allocated unit can perform exactly one operation per tick.

We will assume that every step which does vector operations takes the same amount of time t_{vector}. The time that a matrix multiplication takes is denoted by t_{matrix}. Since a single BiCGSTAB iteration uses 3 steps with vector operations and 2 matrix multiplications, we can express the total time of a single BiCGSTAB $t_{\text{iteration}}$ as

$$t_{\text{iteration}} = 2t_{\text{matrix}} + 3t_{\text{vector}} \tag{7.3}$$

Vector operations

The vector kernel performs n_{vec} operations per cycle. So, if the kernel is compute bound, it will take $\frac{N}{n_{\text{vec}}}$ cycles to run to completion. So, in this case the execution time will be $\frac{N}{n_{\text{vec}}} t_{\text{cycle}}$.

If the execution of a step with vector operations is memory bound, the time that the step takes will be approximated by the time that the memory takes to read and/or write the results. It is assumed that we use a datatype that uses b_{datatype} bytes. The steps with vector operations use a variable number of vectors. We assume now that in each step 4 vectors are read, and two are written by the vector kernel. This is not completely accurate, since the first step with vector operations only reads three vectors and writes a single vector. It is accurate for the other two steps with vector operations, and allows us to work with a single value t_{vector} instead of three different values. Assuming that the step with vector operations is memory bound, we have $t_{\text{vector}} = 6\frac{Nb_{\text{datatype}}}{s_{\text{mem}}}$. Here, b_{datatype} is the size in bytes of the datatype that is used in the vectors, and s_{mem} is the speed of the memory.

Putting everything together, we have

$$t_{\text{vector}} = \max\left(\frac{N}{n_{\text{vec}}} t_{\text{cycle}}, 6\frac{Nb_{\text{datatype}}}{s_{\text{mem}}}\right) \tag{7.4}$$

Matrix operations

The matrix kernel is designed to compute a single degree of freedom per cycle. If the matrix multiplication is compute bound, this means that it will take $t_{\text{matrix}} = Nt_{\text{cycle}}$. For the matrix multiplication the partial derivatives of the function z associated to the vector $\mathbf{z}$ need to be evaluated. For this, $(p+1)^2$ elements of $\mathbf{z}$ are needed. Since $p+1$ 'windows' into the vector elements are necessary, the input vector is streamed $p+1$ times. Additionally, the matrix kernel takes an extra vector to compute the dot product $\mathbf{v}\cdot\mathbf{r_0}$ in step 2. The output needs to be written to memory as well. So, if the matrix multiplication is memory bound, it will take $t_{\text{matrix}} = (p+3)\frac{Nb_{\text{datatype}}}{s_{\text{memory}}}$ time.

So we find

$$t_{\text{matrix}} = \max\left(Nt_{\text{cycle}}, (p+3)\frac{Nb_{\text{datatype}}}{s_{\text{memory}}}\right) \tag{7.5}$$

7.6.3. Results

One way to handle the variable number of quadrature points is to only handle a limited number of quadrature points per cycle. This saves hardware resources, but limits the speed of the design, and is more complicated, since it is necessary to have different types of cycles (one in which a degree of freedom is computed, and one in which no degree of freedom is computed), and keeping track of all values is more complicated. Alternatively, it is possible to compute a degree of freedom in each cycles. This requires that all quadrature points in

an element are handled in a single cycle. In the worst case, there are $(p+1)^2$ quadrature points, so enough hardware resources need to be configured to handle this number of quadrature points.

The computation of the geometric factor requires a matrix inversion and is the most expensive part of the design. It is preferable to compute the geometric factor once (for example, on the CPU, or in a separate stage on the DFE), and store them in LMEM. However, this is very hard to do with a variable number of quadrature points.

At the time that the design was made, the problems with p-refinement (see section 4.2) were not solved yet, and the efficient matrix-free multiplication (see section 4.4) was not known to the author. The performance model was made for a design that uses row-wise assembly. The performance suggested that a design that uses basis functions with polynomial degree $p = 2$ would fit on the DFE without many changes. Additionally, the problems with p-refinement were not solved. So, it made sense to restrict the design to polynomial order $p = 2$.

This simplifies the design significantly. While the number of quadrature points per element is still variable, the maximum number of quadrature points per element is $(p+1)^2$. In this case, a simple design that uses a matrix multiplication that computes one degree of freedom per cycle is possible. The computation of the geometric factor can be done on-the-fly. Assuming that $p = \tilde{p} = 2$, n_{vec}, and that the matrix multiplication assembles one degree of freedom per cycle (so that we need to handle $(p+1)^2 = 9$ quadrature points in the worst case) gives 567 additions, 931 multiplications, and 9 divisions. We can use the operation counts of the LUTs, flip-flops (FFs), digital signal processing blocks (DSPs), and block random-access memories (BRAMs), and the information in appendix D to compute the hardware utilization on the MAX5 DFE. Using a 48-bit fixed point datatype we get:

Table 7.1: Hardware utilization for $p = 2$ on MAX5, using 48-bit fixed point values

		LUTs	FFs	DSPs	BRAMs
Additions	567	27216	27783		0
Multiplications	931	49343	230888	8379	0
Divisions	9	22995	45621	0	0
Data	98304 values	0	0	0	256
Buffer	49152 values	0	0	0	128
Total		99554	304292	8379	384
Total available		1182240	2364480	6840	4320
Utilization		9%	13%	123%	9%

So, the design will not fit when 48-bit fixed point values are used (unless some optimizations are done). It is obvious that the number of DSPs is the limiting factor. In the same way, it can be seen that the design will (quite easily) fit when 32-bit floating point or 27-bit fixed point is used. So, when floating point values are used, a precision between 32 and 64-bit is required, and for fixed point a precision between 27 and 45-bit is feasible[2].

The LMEM in the MAX5 DFE has a speed of 47.65 GB/s. It is assumed that a 48-bit datatype is used, 1024 basis functions are used in every dimension, and a clock speed of 200MHz is used. With these parameters we can estimate the execution time of the different steps in a BiCGSTAB iteration.

By using (7.6.2), we find that the vector steps take

$$t_{\text{vector}} \max\left(\frac{1024^2}{4} \cdot \frac{1}{200 \cdot 10^6}, 6\frac{1024^2 \cdot 6}{47.65 \cdot 1024^3}\right)$$
$$= \max(0.00131, 0.00074) = 0.00131$$

So the vector operations are compute bound, and take 1.31 ms. It should be noted that the vector computations can be done faster by increasing n_{vec}. However, the matrix multiplication is the bottleneck, so it does not help a lot to use more hardware resources for the vector operations.

Now, we can use (7.6.2) to find the time that a matrix multiplication takes:

$$t_{\text{matrix}} = \max\left(1024^2 \cdot \frac{1}{200 \cdot 10^6}, 5\frac{1024^2 \cdot 6}{47.65 \cdot 1024^3}\right)$$

[2] The multipliers on the DSP blocks on MAX5 DFE have a size of 18 by 27 bits, so using a 45-bit fixed point datatype maps more efficiently to DSP blocks and would likely fit. The same holds for floating point types with 45 mantissa bits.

$$= \max(0.00524, 0.000614) = 0.00524$$

So the matrix multiplication is compute bound as well, and takes 5.24 ms. By (7.6.2), we find that the time for a single iterations is 14.41 ms. So, if 1000 BiCGSTAB iterations are needed, this can be done in 14.41 seconds.

7.7. Changes to the design

During the testing of the design, the LMEM access on a DFE turned out to be restricted to multiples of 384 bytes. For this reason, it was necessary to instantiate extra kernels that throw away the values that are not needed.

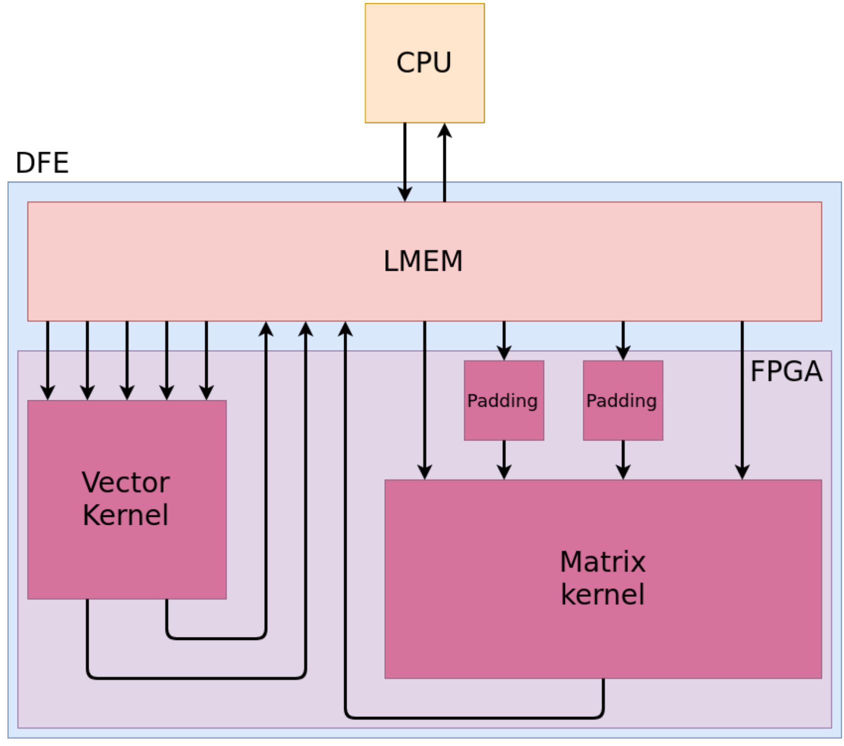


Figure 7.5: The structure of the dataflow implementation. The arrows represent streams.

Another problem was that accessing the LMEM repeatedly requires writing a memory command generator. This is a complicated and time-consuming task. For this reason, the design was simplified and now requires that the CPU starts each step. This does introduce some overhead - it is not clear how much.

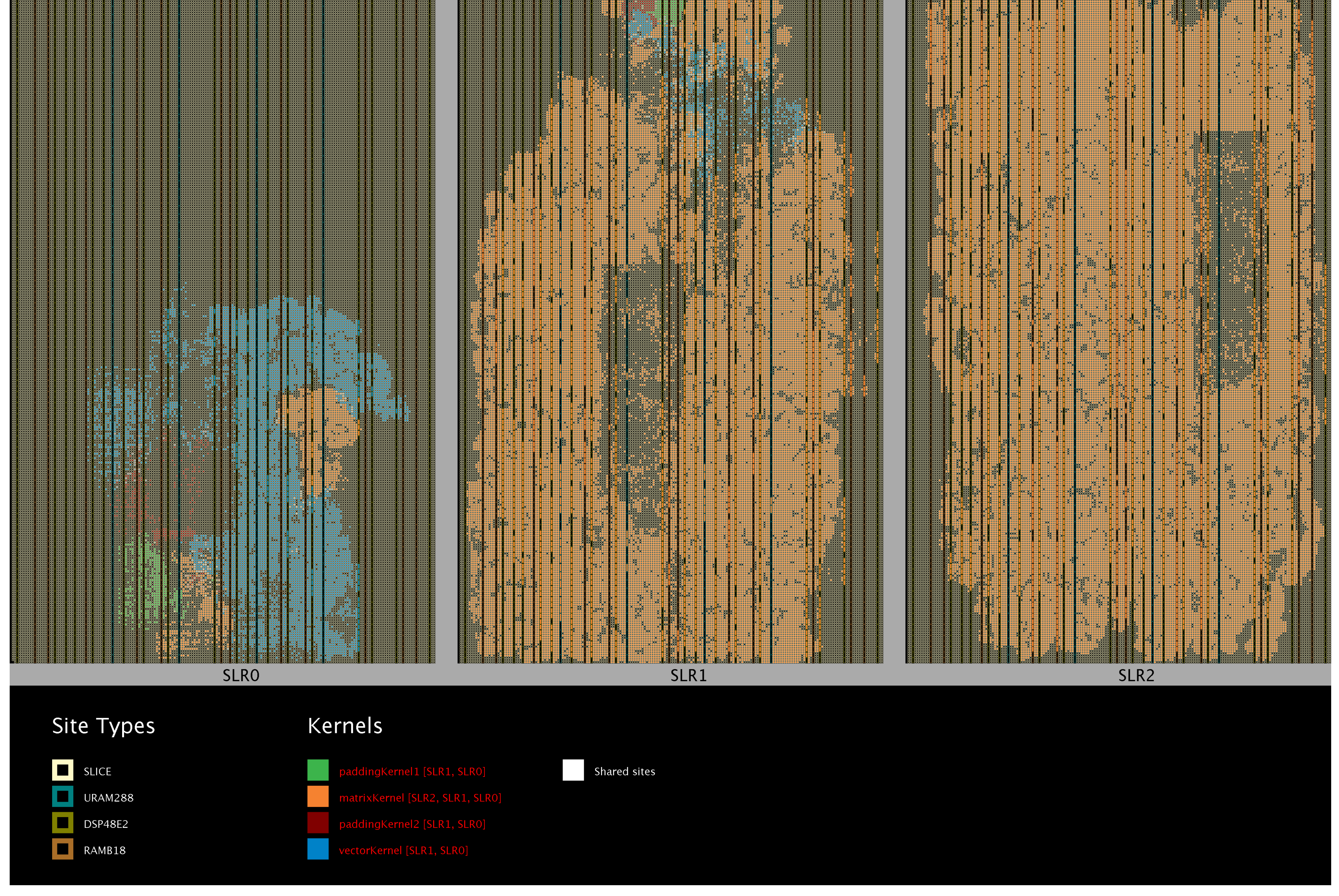
One of the factors that prevent using the design for basis functions of higher polynomial degree is the limited number of streams from and to LMEM. There can be a maximum of 15 of such streams. In figure 7.5, it can be seen that there are 14 memory streams used. For $p = 4$ and higher, there will be too few memory streams available, and it will be necessary to use some trick to optimize some of the memory streams away. One way to do this is to connect the CPU to one of the kernels instead of directly to the LMEM. The kernel can then (by setting some option from the CPU) serve as a passthrough kernel that allows the CPU to either read from or write to LMEM.

7.8. Build results

When a 45-bit floating point format is used, the design indeed fits on the MAX5 DFE:

```
PRELIMINARY RESOURCE USAGE
Logic utilization:     755147 / 1182240 (63.87%)
LUTs:                  446650 / 1182240 (37.78%)
Primary FFs:           616994 / 2364480 (26.09%)
DSP blocks:              2727 / 6840    (39.87%)
Block memory (BRAM18):   2479 / 4320    (57.38%)
```

Unfortunately the design was not able to meet the timing constraints. Pavel Burovskiy pointed out that the FPGA on the MAX5 DFE is divided into three separate silicon dies called *super logic regions* (SLRs, see [33]) for more information). The interconnections between these regions are limited, and the failure to meet the timing constraints is likely a result of the single, big kernel that implements the matrix multiplication being split over differen SLRs. Unfortunately, the author was not aware of the existence of these SLRs while working on the dataflow design. The following image, which is automatically generated by MaxIDE, shows the configuration of the hardware resources on the FPGA on the MAX5.


SLR0
SLR1
SLR2
Site Types
SLICE
URAM288
DSP48E2
RAMB18
Kernels
paddingKernel1 [SLR1, SLR0]
matrixKernel [SLR2, SLR1, SLR0]
paddingKernel2 [SLR1, SLR0]
vectorKernel [SLR1, SLR0]
Shared sites

This can be solved, but this requires a re-design into multiple kernels. Unfortunately, there was not enough time to do this. Instead, some optimizations were used, such as hardcoding the mapping, which allows to save some resources. When a 32-bit floating point format is used and pipelining is mostly disabled, the design fits on an older DFE, the MAX4:

```
FINAL RESOURCE USAGE
Logic utilization:    249336 / 262400 (95.02%)
Primary FFs:          338724 / 524800 (64.54%)
Secondary FFs:         60228 / 524800 (11.48%)
Multipliers (18x18):    1074 / 3926   (27.36%)
DSP blocks:              546 / 1963   (27.81%)
Block memory (M20K):    1892 / 2567   (73.70%)
```

The design meets timing at the very low clock speed of 20 MHz. While the single precision datatype that is used is not precise enough to converge well, the DFE implementation still works. So while the implementation is not of practical use, it does show that even an unoptimized DFE implementation can be faster than a CPU implementation.

The problem is illustrated in figure 7.6. It can be seen that, while both versions use the BiCGSTAB method, the CPU version converges faster, and to a more accurate solution, due to the higher precision that the CPU version uses.

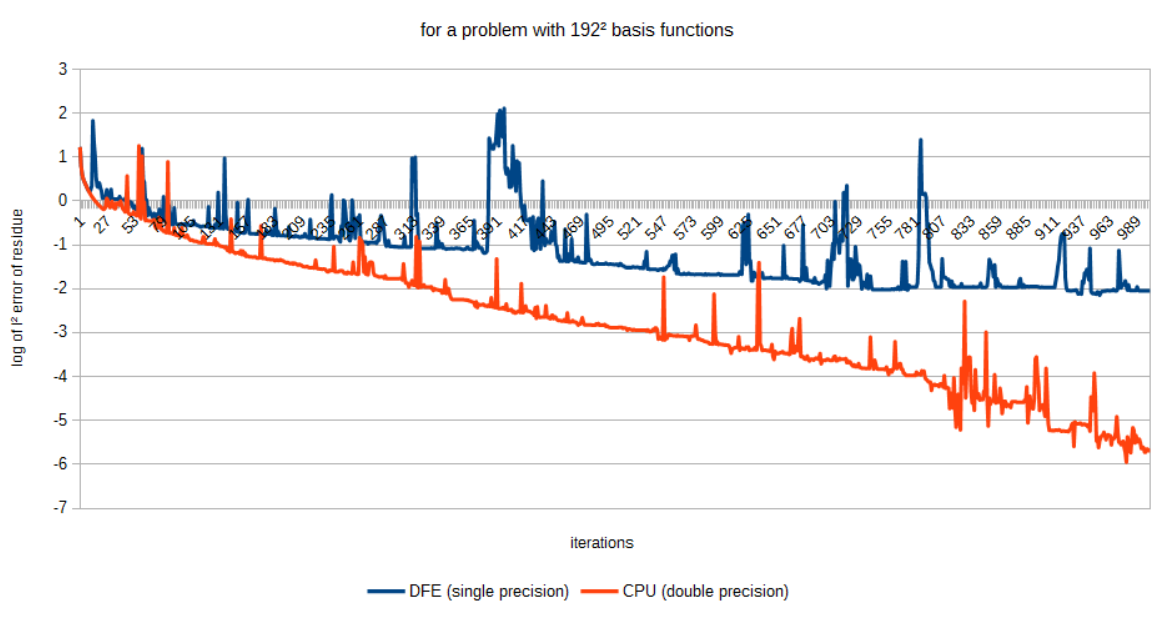


Figure 7.6: A comparison of the convergence speed of the CPU and the DFE. Both use the BiCGSTAB method, but the CPU uses higher precision.

Even with this low clock frequency, the DFE version is about five times as fast as the CPU version: 1000 BiCGSTAB iterations are done in about 10 seconds instead of the 50 seconds that the CPU version takes.

8 Conclusion

In this thesis, several contributions are made:

1. The application of weighted quadrature to IgA is described and analyzed. The technique presented in [7] is extended to non-uniform knot vectors. It is shown that the technique presented in [7] has several issues. The solutions to these issues (due to Mattia Tani and Giancarlo Sangalli) are presented. Additionally, some new developments (also due to Giancarlo Sangalli and Mattia Tani) are documented.
2. An IgA code that implements weighted and Gaussian quadrature, as well as multiple iterative and direct linear solvers has been coded from scratch.
3. The architecture for a dataflow implementation is sketched, a reference implementation on the CPU has been made, and a proof-of-concept dataflow implementation is presented.
4. Several suggestions are contributed, which should enable a more efficient and practical dataflow implementation.

The dataflow implementation is only a partial success. In order to fit the design on an older dataflow engine, it was necessary to reduce the precision. This causes the convergence of the dataflow implementation to be worse than that of the CPU version. For this reason, this specific dataflow implementation is not preferable to the CPU implementation. However, the design does show the potential of a dataflow implementation that uses weighted quadrature: Even with the unoptimized dataflow implementation that uses a low clock speed, the BiCGSTAB iterations are significantly faster on the dataflow engine than that they are on the CPU. Additionally, build results show that it should be possible to run the design in a higher precision and with a higher clock speed on the newer MAX5 DFE. The dataflow implementation is a milestone: It is the first dataflow implementation that uses IgA with weighted quadrature. Moreover, it shows the feasibility of a dataflow implementation for this use case. For this reason, the design can still be considered succesful as a proof of concept. So while a dataflow implementation has shown to be feasible, it is still not clear if the performance reasonably justifies the increase in engineering effort.

The trouble with the implementation of weighted quadrature and the lack of resources on weighted quadrature left not as much time for experimentation and engineering work as desired. Work on the mathematical development of weighted quadrature was originally outside the scope of this thesis, but it was necessary to find the cause of the problems with p-refinement. Mattia Tani, one of the authors of [7], initially reported that there should be no such issues, and the CPU implementation was believed to contain a bug for a long time. A considerable amount of time was spent trying to find the supposed bug. When the problem was acknowledged to be a mathematical issue, and the solution was presented by Mattia Tani on June 27th, there were only five days left before the final on-site stay at Maxeler in London. This has prevented a more extensive analysis of the behaviour of the convergence as a function of all the parameters.

8.1. Future work

In order to estimate the practical use of a dataflow implementation more accurately, it is necessary to have a more efficient CPU implementation. While a reference CPU implementation was made, it was beyond

the scope of this thesis to optimize this implementation. The CPU implementation does not use multiple processors, GPU acceleration, the efficient matrix-free multiplication, or x86 extensions for vector processing.

As mentioned before, there was no time to do an extensive analysis of the convergence behaviour. This extensive analysis is desirable because it might allow one to identify the range of the parameters of the problems for which dataflow engines are likely to outperform CPUs. This sounds rather straightforward, but one should not underestimate the complexity. There are a number of parameters that should be taken into account. Ignoring more advanced topics like mixed-precision computations for now, the most important parameters are:

1. The number of basis functions
2. The polynomial degree of the basis functions
3. The machine precision and number representation (fixed or floating point)
4. The rule that is used to pick the quadrature weights
5. The quadrature rule that is used
6. The linear solver that is used, and the number of iterations that is used
7. The choice of preconditioner (optional)

All of these parameters influence each other. Moreover, they influence the cost (and possibly the architecture) of the design: some parts of the design scale as $O(p^3)$ in the polynomial degree p, and some parts scale as $O(b^2)$ in the number of bits b in the machine precision. So, on the one hand there is the desire to increase the polynomial degree and the machine precision to improve the convergence. On the other hand, the design scales badly with these parameters. Of special interest is the convergence of the method when either the 'regular rule' for picking quadrature points, or global quadrature points are used (see section 4.1), since these can considerably simplify the design of a dataflow implementation.

As of yet, weighted quadrature has only been applied to B-spline geometries. There is no obvious reason that weighted quadrature would not work for NURBS geometries. So, it should be possible to generalize the technique and use weighted quadrature for NURBS geometries.

One way in which both the dataflow implementation and the CPU implementation might be improved, is by the use of a preconditioner. A preconditioner would hopefully improve the convergence behaviour. The preconditioner presented in [30] is of special interest, since it is designed to work well with weighted quadrature. The feasibility of implementing this preconditioner on a dataflow engine is not assessed in this thesis.

Since the dataflow implementation is only a proof of concept, it can be generalized in several ways. The extension to differential equations of the form $\alpha\Delta u + \beta u = f$ should not require major changes in the architecture. The same ideas that are used in the dataflow implementation of the 2D case can be used to solve 3D problems. However, the design probably needs to be significantly changed for this. It is hard to say, but a dataflow implementation might be more appropriate for 3D problems than for 2D problems. Modern CPUs are already quite fast for 2D problems on a single patch, so problems which are computationally expensive to solve usually yield a very precise solution. So, a high machine precision needs to be used, which do not map well to dataflow engines.

In practice, most geometries use multiple patches, which are coupled at the boundaries. There are also more complicated boundary conditions which might be used. These things might be challenging to do on a DFE in a matrix-free way.

8.1.1. Improvements to the dataflow implementation

For reasons mentioned earlier, the implementation was restricted to use basis functions of polynomial degree 2. In general, it is better to use basis functions of a higher degree. To make the design usable for basis functions of higher degree, and to meet the timing constraints on the MAX5 DFE, some changes to the design are required. In particular, it is necessary to:

1. Split up the design into different parts that do not need much interconnect between them
2. Reduce the area that is needed for the design

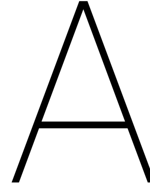
The computations in the matrix-free multiplication can be split into four summations with an identical structure. In the current design, the summations are computed in the same kernel. It should be rather straightforward to split this big kernel into four instances of the same kernel. It seems plausible that with this changes, the design would fit on the MAX5 DFE, and would meet the timing constraints.

In order to reduce the area that the design needs, multiple measures can be taken. By splitting the computations in four identical blocks, one can reduce the area that the design needs by splitting the computations over multiple clock ticks (of course, this comes at the cost of needing more cycles for a matrix multiplication).

Another promising way to reduce the design is to use a more regular pattern of quadrature points. This will help to make efficient use of the available logic and arithmetic units.

Next, the evaluation of the geometric factors is quite expensive, and it might help to precompute them, and stream them from LMEM. This does increase the size of the data that is streamed from LMEM, so one should make a performance model to see if this is likely to help for the design that is used. This measure should probably only be used when a regular grid of quadrature points is used as well. Otherwise, a variable number of data needs to be loaded and orchestrated. This is far from trivial, and is likely to be expensive, both in terms of the on-chip logic and design effort.

These measures can be applied independently, and it is expected that the area that the design uses can be reduced significantly. If the area allows, it might be beneficial to implement the preconditioner described in [30] on the DFE. It should also be mentioned that when the number of basis functions is small enough, one might be able to store the vectors in FMEM, which would allow for a significant speedup (especially when the geometric factor is computed on-the-fly, since in this case it would not be necessary to load any data from LMEM).



Some mathematical theory

A.1. Multivariate calculus

Theorem 7 (Multivariate chain rule)**.** *Suppose that $s : A \to B$ and $f : B \to C$ are differentiable functions. Then $f \circ s$ is differentiable as well. Moreover, for $\xi \in A$, we have*

$$D(f \circ s)(\xi) = Df(s(\xi))Ds(\xi) \tag{A.1}$$

Proof. See [23], section 2.5, page 153, theorem 11. □

Theorem 8 (Change of variables)**.** *Suppose that $s : \Omega_0 \to \Omega$ is a differentiable bijection. For $f : \Omega \to \mathbb{R}$ we have*

$$\int_{\Omega} f(\mathbf{x})\ d\mathbf{x} = \int_{\Omega_0} f(s(\boldsymbol{\xi}))|\det(Ds(\boldsymbol{\xi}))|\ d\boldsymbol{\xi} \tag{A.2}$$

Proof. See [23], section 6.2, page 382, theorem 2. □

Lemma 1 (Gauss' divergence theorem)**.** *Let $\Omega \subset \mathbb{R}^d$ be a bounded domain with a piecewise smooth boundary, let $\mathbf{n}$ be the outward normal, and $\mathbf{w} : \Omega \to \mathbb{R}^d$ be a continuously differentiable vector field. Then*

$$\int_{\Omega} \nabla \cdot \mathbf{w}\ d\Omega = \int_{\partial\Omega} \mathbf{w} \cdot \mathbf{n}\ d\Gamma$$

Proof. See [2], section 16.4, theorem 8, page 925. □

Lemma 2. *Let $v : \Omega \to \mathbb{R}$ and $\mathbf{w} : \Omega \to \mathbb{R}^d$ be continuously differentiable for some domain $\Omega \subset \mathbb{R}^d$. Then we have*

$$\nabla \cdot (v\mathbf{w}) = \nabla v \cdot \mathbf{w} + v(\nabla \cdot \mathbf{w})$$

Proof. See [2], section 16.2, theorem 3b, page 915. □

Theorem 9. *Given two continuously differentiable functions $u, v : \Omega \to \mathbb{R}$, we have*

$$\int_{\Omega} (\Delta u) v d\Omega = \int_{\partial\Omega} v \nabla u \cdot\ \mathbf{n} d\Gamma - \int_{\Omega} \nabla u \cdot \nabla v\ d\Omega \tag{A.3}$$

Proof. This theorem can be proved by using lemma 1 and 2. From lemma 2, we see

$$(\nabla \cdot \mathbf{w}) v = \nabla \cdot (v\mathbf{w}) - \nabla v \cdot \mathbf{w}$$

Substituting $\mathbf{w} = \nabla u$ gives

$$(\nabla \cdot \nabla u) v = \nabla \cdot (v \nabla u) - \nabla v \cdot \nabla u$$

We now have

$$\int_{\Omega} (\Delta u) v\ \mathrm{d}\Omega = \int (\nabla \cdot (\nabla u)) v\ \mathrm{d}\Omega = \int_{\Omega} \nabla \cdot (v \nabla u)\ \mathrm{d}\Omega - \int_{\Omega} \nabla v \cdot \nabla u\ \mathrm{d}\Omega \tag{A.4}$$

We can now apply lemma 1 with $\mathbf{w} = v\nabla u$ to see

$$\int_{\Omega} \nabla \cdot v\nabla u \, d\Omega = \int_{\partial\Omega} v\nabla u \cdot \mathbf{n} \, d\Gamma$$

Substituting this in (A.4) gives

$$\int_{\Omega} (\Delta u) v d\Omega = \int_{\partial\Omega} v\nabla u \cdot \mathbf{n} d\Gamma - \int_{\Omega} \nabla u \cdot \nabla v \, d\Omega$$

□

A.2. Approximation theory

Definition 12. *The L^2 norm of a function $f : \Omega$ is denoted $||f||_{L^2}$ and defined as $||f||_{L^2} = \int_{\Omega} f(\xi)^2 \, d\xi$.*

A related concept is the l^2 *norm.*

Definition 13. *The l^2 norm of a vector $v \in \mathbb{R}^n$ is denoted $||v||_{l^2}$ and defined as $||v||_{l^2} = \sqrt{\sum_{k=0}^{n-1} v_k^2}$.*

A.2.1. Interpolation

Let us consider the *interpolation problem.* Suppose that we have a space V which is spanned by basis functions $N_0, N_1, ..., N_{n-1} : A \to \mathbb{R}$, and we want to find a function $f \in V$ which interpolates the point y_i at ξ_i for $k = 0, 1, ..., m-1$. Writing this as a system yields

$$\begin{aligned} f(\xi_0) &= y_0 \\ f(\xi_1) &= y_1 \\ &\vdots \\ f(\xi_{m-1}) &= y_{m-1} \end{aligned} \tag{A.5}$$

Since $f \in V$, we can write $f(\xi) = \sum_{j=0}^{m-1} f_j N_j(\xi)$, where the coefficients f_0, f_1, ..., f_{N-1} are still unknown. Substituting this in (A.5) yields

$$\begin{aligned} \sum_{j=0}^{m-1} f_j N_j(\xi_0) &= y_0 \\ \sum_{j=0}^{m-1} f_j N_j(\xi_1) &= y_1 \\ &\vdots \\ \sum_{j=0}^{m-1} f_j N_j(\xi_{m-1}) &= y_{m-1} \end{aligned}$$

which is equivalent to a linear system

$$\mathbf{J}\mathbf{f} = \mathbf{y} \tag{A.6}$$

where

$$\mathbf{J}_{i,j} = N_j(\xi_i) \tag{A.7}$$

This matrix $\mathbf{J}$ will be called the *interpolation matrix.* So, if the interpolation matrix is square and non-singular, the interpolation problem has a unique solution for every choice of $y_0, y_1, ..., y_{m-1}$. In general, it depends on the choice of basis functions N_0, N_1, ..., N_{m-1} and nodes ξ_0, ξ_1, ..., ξ_{m-1} if the interpolation matrix $\mathbf{J}$ is singular or not. If the basis functions are B-spline basis functions, we have the Whitney-Schoenberg theorem (theorem 3).

A.2.2. L^2 projection

Suppose we have a function $g:\Omega\to\mathbb{R}$, a space V of functions on Ω, and we want to find an $f^*\in V$ such that $||f-g|_{L^2}|$ is small.

Definition 14. *The L^2 projection of a function $f:\Omega\to\mathbb{R}$ onto a space V of functions $\Omega\to\mathbb{R}$ is a function $g\in V$ for which $||f-g||_{L^2}$ is minimal. That is, there exists no $g^*\in V$ for which $||f-g||_{L^2}<||f-g||_{L^2}$.*

We have the following theorem.

Theorem 10. *If V is spanned by basis functions $N_0,N_1,...,N_{n-1}:\Omega\to\mathbb{R}$ the L^2-projection of f onto V exists and is unique.*

Proof. See [14], theorem 1.8 on page 3, and theorem 1.11 on page 4. □

The following theorem can be used to find the L^2 projection.

Theorem 11. *The L^2 projection of f onto the space V spanned by the basis functions $N_0,N_1,...,N_{n-1}$ satisfies*

$$\int_\Omega f(\xi)N_i(\xi)\,d\xi=\int_\Omega g(\xi)N_i(\xi)\,d\xi \tag{A.8}$$

Proof. See [14], page 30, theorem 3.2. □

We can substitute $f(\xi)=\sum_{j=0}^{m-1} f_jN_j(\xi)$ and interpret the resulting system

$$\sum_{j=0}^{n-1}\left(\int_\Omega N_i(\xi)N_j(\xi)\,\mathrm{d}\xi\right)f_j=\int_\Omega g(\xi)N_i(\xi)\,\mathrm{d}\xi \quad \text{for } i=0,1,...,n-1 \tag{A.9}$$

as a linear system $\mathbf{Lf}=\mathbf{g}$, where $\mathbf{L}_{i,j}=\int_\Omega N_i(\xi)N_j(\xi)\,\mathrm{d}\xi$ and $g_i=\int_\Omega g(\xi)N_i(\xi)\,\mathrm{d}\xi$. Since the l^2 projection exists and is unique, this system is nonsingular, which means that it can be used to find the vector $\mathbf{f}$ of coefficient $f_0,f_1,...,f_{n-1}$.

A.3. Linear algebra

Definition 15. *A matrix $\mathbf{M}$ is called* orthogonal *if $\mathbf{M}$ is square and $\mathbf{M}^\top\mathbf{M}=\mathbf{I}$.*

Definition 16. *For a matrix $\mathbf{M}$, the* Moore-Penrose pseudoinverse *or simply* pseudoinverse *$\mathbf{M}^+$ is defined as the matrix which satisfies*

1. $(\mathbf{M}^+)^+=\mathbf{M}$
2. $\mathbf{MM}^+\mathbf{M}=\mathbf{M}$
3. $\mathbf{M}^+\mathbf{M}$ *is symmetric*

Theorem 12. *The pseudoinverse $\mathbf{M}^+$ exists for every matrix $\mathbf{M}$ and is unique.*

Proof. See [27], theorem 1. □

Theorem 13. *For a $m\times n$ matrix $\mathbf{M}$, $\mathbf{x}=\mathbf{M}^+\mathbf{b}$ minimizes $||\mathbf{Mx}-\mathbf{b}||_{l^2}$. Moreover, over all $\mathbf{x}$ that minimize $||\mathbf{Mx}-\mathbf{b}||_{l^2}$, $\mathbf{x}=\mathbf{M}^+\mathbf{b}$ minimizes $||\mathbf{x}||_{l^2}$.*

Proof. See [6], chapter 2, page 109, corollary 3. □

Definition 17. *For a matrix $\mathbf{M}\in\mathbb{R}^{m\times n}$ with $m\geq n$, the* QR-decomposition *is a tuple $(\mathbf{Q},\mathbf{R})$, where $\mathbf{Q}\in\mathbb{R}^{n\times n}$ is orthogonal, $\mathbf{R}\in\mathbb{R}^{m\times n}$ satisfies $\mathbf{R}^{i,j}=0$ for $i>j$, and $\mathbf{QR}=\mathbf{M}$.*

Definition 18. *For a matrix $\mathbf{M}\in\mathbb{R}^{m\times n}$, the* singular value decomposition *is a triple $(\mathbf{U},\mathbf{S},\mathbf{V})$, where $\mathbf{U}\in\mathbb{R}^{n\times n}$ is orthogonal, $\mathbf{S}\in\mathbb{R}^{m\times n}$ satisfies $\mathbf{S}^{i,j}=0$ for $i\neq j$, $\mathbf{V}\in\mathbb{R}^{n\times n}$ is orthongal, and $\mathbf{USV}=\mathbf{M}$.*

Theorem 14. *Suppose we have a matrix* $\mathbf{M} \in \mathbb{R}^{m\times n}$*. Then*

$$\mathbf{M}^{+} = \mathbf{V}^{\top}\mathbf{S}^{+}\mathbf{U}^{\top}$$

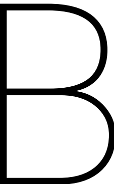
where $(\mathbf{U}, \mathbf{S}, \mathbf{V})$ *is the singular value decomposition of* $\mathbf{M}$*. Moreover, if det*$(\mathbf{M}^{\top}\mathbf{M}) \neq 0$*, we have*

$$\mathbf{M}^{+} = (\mathbf{M}^{\top}\mathbf{M})^{-1}\mathbf{M}$$

and

$$\mathbf{M}^{+} = \mathbf{Q}^{T}\mathbf{R}^{-1}$$

Proof. See [27], lemma 1.6 for a proof of the first equality. The second equality is proved in [20], proposition 3.2. □



Algorithms

B.1. Efficient evaluation of values and derivatives of nonzero B-spline basis functions

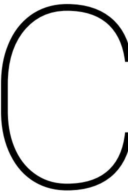
Algorithm 5 Efficient evaluation of values and derivatives of nonzero B-spline basis functions

1: **function** BSPLINEBASIS(Ξ, i, p, ξ)
▷ returns an array with the values of all basis functions which are nonzero at this point
2: let values be a new array of $p+1$ zeroes
3: let derivatives be a new array of $p+1$ zeroes
4: **if** $\xi < \Xi[0]$ or $\xi \geq \Xi[\Xi.\text{length - 1}]$ **then**
5: return values, derivatives
6: **end if** ▷ Find k such that $\xi \in [\xi_k, \xi_{k+1})$
7: k = 0
8: **while** $\xi \notin [\xi_k, \xi_{k+1}$ **do**
9: $k = k+1$
10: **end while**
11: values[0] = 1
12: **for** $q = 0, 1, ..., p-2$ **do**
13: **for** $j = k-q, k-q+1, ..., k$ **do**
14: $\alpha = (\xi - \Xi[j])$ / $(\Xi[j+q+1] - \Xi[j])$
15: values[$j-1$] = values[$j-1$] $(1-\alpha)$ * values[j]
16: values[j] = values[j] * α
17: **end for**
18: **end for**
▷ Compute derivatives
19: **for** $j = 1, 2, ..., p$ **do**
20: $\alpha = p$ / $(\Xi[j+k+p] - \Xi[j+k])$
21: derivatives[$j-1$] = derivatives[$j-1$] - α
22: derivatives[j] = α
23: **end for**
▷ Finish computing the values of the basis functions
24: **for** j = k - p + 1, k - p + 2, ..., k **do**
25: $\alpha = (\xi - \Xi[j])$ / $(\Xi[j+p] - \Xi[j])$
26: values[$j-1$] = values[$j-1$] $(1-\alpha)$ * values[j]
27: values[j] = values[j] * α
28: **end for**
29: return values, derivatives
30: **end function**

B.2. Elemental loop

Algorithm 6 Assembly of the mass matrix with an elemental loop

1: let $\psi_0, \psi_1, \ldots, \psi_{N-1}$ be the basis functions
2: let $e_0, e_1, \ldots, e_{M-1}$ be the elements
3: let glob_mat be an $N \times N$ matrix with each element equal to zero
4: **for** `k` $= 0, 1, \ldots, M-1$ **do**
5: let n be the number of basis functions with support on e_k
6: let $k_0, k_1, \ldots, k_{n-1}$ be the indices of the basis functions with support on e_k
7: let elem_mat be an $n \times n$ matrix with each element equal to zero
▷ assemble element matrix elem_mat
8: **for** $i = 0, 1, \ldots, n-1$ **do**
9: **for** $j = 0, 1, \ldots, n-1$ **do**
10: elem_mat$[i, j]$ = elem_mat$[i, j] + \int_{e_k} \psi_{k_i} \psi_{k_j} \, \mathrm{d}\Omega$
11: **end for**
12: **end for**
▷ scatter element matrix elem_mat to global matrix glob_mat
13: **for** $i = 0, 1, \ldots, n-1$ **do**
14: **for** $j = 0, 1, \ldots, n-1$ **do**
15: glob_mat$[k_i, k_j]$ = glob_mat$[k_i, k_j]$ + elem_mat$[i, j]$
16: **end for**
17: **end for**
18: **end for**



Code

C.1. Vector kernel

```
package iga;

import igaUtils.FloatingPoint;
import java.util.List;
import maxpower.kernel.io.AspectChangeIO;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVector;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVectorType;

public class VectorKernel extends Kernel
{
    public VectorKernel(KernelParameters kp)
    {
        super(kp);

        DFEType scalartype = Datatype.scalartype;
        DFEVectorType<DFEVar> scalarvectortype =
            new DFEVectorType<DFEVar>(scalartype, 4);

        // inputs
        DFEVar constant1 = io.scalarInput("constant1", scalartype);
        DFEVar constant2 = io.scalarInput("constant2", scalartype);

        AspectChangeIO lmemio = new AspectChangeIO(this, 1536);
        DFEVector<DFEVar> a = lmemio.input("a", scalarvectortype);
        DFEVector<DFEVar> b = lmemio.input("b", scalarvectortype);
        DFEVar setctob = io.scalarInput("setctob", dfeBool());
        DFEVector<DFEVar> c = setctob ? b : lmemio.input("c", scalarvectortype,
            ~setctob);
        DFEVector<DFEVar> d = lmemio.input("d", scalarvectortype);
        DFEVector<DFEVar> extra = lmemio.input("extra", scalarvectortype);

        // vector operations
        DFEVector<DFEVar> e = a + constant1 * b;
        DFEVar outputsum = io.scalarInput("outputsum", dfeBool());
```

```
        DFEVector<DFEVar> d_scaled = constant2 * d;
        DFEVector<DFEVar> f_0 = c + d_scaled;
        DFEVector<DFEVar> f = (outputsum ? (e + d_scaled) : f_0);

        // calculate dot products
        List<DFEVar> list1 = (extra * f).getElementsAsList();
        DFEVar dotproduct1 = FloatingPoint.accumulate(
            ((list1[0] + list1[1]) + (list1[2] + list1[3])));
        List<DFEVar> list2 = (f * f).getElementsAsList();
        DFEVar dotproduct2 = FloatingPoint.accumulate(
            ((list2[0] + list2[1]) + (list2[2] + list2[3])));

        DFEVar cycles = io.scalarInput("cycles", dfeUInt(64));
        DFEVar counter = control.count.simpleCounter(64);
        DFEVar flushoutput = counter.eq(cycles - 1);
        lmemio.output("e", e, ~outputsum, flushoutput);
        lmemio.output("f", f, constant.var(true), flushoutput);

        AspectChangeIO pcieio = new AspectChangeIO(this, 128);
        pcieio.output("dotproduct1", dotproduct1, flushoutput, flushoutput);
        pcieio.output("dotproduct2", dotproduct2, flushoutput, flushoutput);
    }
}
```

C.2. Matrix kernel

```
package iga;

import igaUtils.FloatingPoint;
import igaUtils.Matrix;
import maxpower.kernel.io.AspectChangeIO;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;


public class MatrixKernel extends Kernel
{
    public MatrixKernel(KernelParameters kp)
    {
        super(kp);

        DFEType scalartype = Datatype.scalartype;

        AspectChangeIO lmemio = new AspectChangeIO(this, 1536);
        DFEVar extra = lmemio.input("extra", scalartype);
        DFEVar vector_0 = lmemio.input("v0", scalartype);
        DFEVar vector_1 = io.input("v1", scalartype);
        DFEVar vector_2 = io.input("v2", scalartype);

        // multiply with matrix
        DFEVar result = Matrix.multiply(this, scalartype,
            new DFEVar[] { vector_0, vector_1, vector_2 });
        DFEVar mux = io.scalarInput("useresultindotproduct", dfeBool());

        // output result of matrix multiplication
```

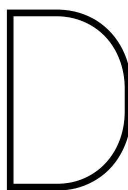
```
        DFEVar flushoutput = control.count.simpleCounter(64).eq(
            io.scalarInput("cycles", dfeUInt(64)) - 1);
        lmemio.output("result", result, constant.var(true), flushoutput);

        // calculate dot products
        DFEVar dotprodvector = mux ? vector_0 : extra;

        DFEVar add1 = dotprodvector * result;
        DFEVar add2 = result * result;
        DFEVar dotproduct1 = FloatingPoint.accumulate(add1);
        DFEVar dotproduct2 = FloatingPoint.accumulate(add2);

        // output dot products
        AspectChangeIO pcieio = new AspectChangeIO(this, 128);
        pcieio.output("dotproduct1", dotproduct1,
            flushoutput, flushoutput);
        pcieio.output("dotproduct2", dotproduct2,
            flushoutput, flushoutput);
    }
}
```

C.3. BiCGSTAB CPU code

```
// indices of vectors on the DFE
#define H      0
#define P      1
#define R      2
#define R0     3
#define S      4
#define T      5
#define V      6
#define X      7

double alpha, beta, delta, delta0, rho, rho_old, omega;

real *bicgstab_init(real *b)
{
    // DFE needs padding at the end
    b = realloc(b, vectorsize);

    double residuenorm = l2norm(b, ndofs);
    real *zerovector = calloc(vectorsize, 1);

    if (STREAMTYPE == FIXEDPOINT)
        for (int i = 0; i < ndofs; i++)
            *((int64_t*)(&(b[i]))) = (int64_t)(b[i] * MUL);

    dfe_extraindex = R0;

    dfe_setvec(P, zerovector);
    dfe_setvec(X, zerovector);
    dfe_setvec(R, b);
    dfe_setvec(R0, b);
    dfe_setvec(V, zerovector);

    alpha = 1;
```

```
    delta = residuenorm;
    delta0 = residuenorm;
    rho = residuenorm;
    rho_old = 1;
    omega = 1;

    return zerovector;
}

real *bicgstab_solve(double *b_ptr, double epsilon, int max_iterations)
{
    double dotproduct1, dotproduct2;

    real *b = (real *)b_ptr;
    if (STREAMTYPE == FLOAT)
        for (int i = 0; i < ndofs; i++)
            b[i] = (float)b_ptr[i];

    real *result = bicgstab_init(b);

    double bound = epsilon * epsilon * delta0;
    for (int i = 0; delta > bound && i < max_iterations; i++)
    {
        beta = (rho * alpha) / (rho_old * omega);

        // DFE step 1
        //     p = r + beta * (p - omega * v)
        dfe_vectorupdate(-1, P, beta, -omega * beta, R, P, -1, V, true, true,
            &dotproduct1, &dotproduct2);

        // DFE step 2
        //     v = A * p
        dfe_matrixmultiply(V, P, false, &dotproduct1, &dotproduct2);

        alpha = rho / dotproduct1;

        // DFE step 3:
        //     h = x + alpha * p
        //     s = r - alpha * v
        dfe_vectorupdate(H, S, alpha, -alpha, X, P, R, V, false, false,
            &dotproduct1, &dotproduct2);

        // DFE step 4
        //     t = A * s
        dfe_matrixmultiply(T, S, true, &dotproduct1, &dotproduct2);

        omega = dotproduct1 / dotproduct2;

        // DFE step 5
        //     x = h + omega s
        //     r = s - omega t
        dfe_vectorupdate(X, R, omega, -omega, H, S, -1, T, true, false,
            &dotproduct1, &dotproduct2);

        rho_old = rho;
        rho = dotproduct1;
```

```
        delta = dotproduct2;
    }

    dfe_readvec(result, X);
    dfe_terminate();
    return result;
}
```



MAX5 hardware resources

D.1. System details

Table D.1: MAX5 system details

Resource	Number
LUTs	1182240
FFs	2364480
DSPs	6840
BRAMs	4320
Max clock frequency	350 MHz
LMEM throughput	47.65 GB/s
LMEM capacity	48 GB

D.2. Hardware resources per arithmetic operation

D.2.1. 27-bit fixed point

Table D.2: Hardware resources for 32-bit fixed point on MAX5

Operation	LUTs	FFs	DSPs	BRAMs
Addition	27	28	0	0
Multiplication	5	45	2	0
Division	826	1716	0	0

D.2.2. 48-bit fixed point

Table D.3: Hardware resources for 48-bit fixed point on MAX5

Operation	LUTs	FFs	DSPs	BRAMs
Addition	48	49	0	0
Multiplication	53	248	9	0
Division	2555	5069	0	0

D.2.3. 32-bit floating point

Table D.4: Hardware resources for 32-bit floating point on MAX5

Operation	LUTs	FFs	DSPs	BRAMs
Addition	177	308	2	0
Multiplication	77	165	2	0
Division	742	1353	0	0

64-bit floating point

Table D.5: Hardware resources for 64-bit floating point on MAX5

Operation	LUTs	DSPs	BRAMs	
Addition	582	949	3	0
Multiplication	132	534	7	0
Division	3135	5979	0	0

Bibliography

[1] Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. *Computer Methods in Applied Mechanics and Engineering, volume = 194, number = 39, pages =.*

[2] Robert A. Adams and Christopher Essex. *Calculus: A complete course*. Pearson, 2014.

[3] P. Antolin, A. Buffa, F. Calabrò, M. Martinelli, and G. Sangalli. Efficient matrix computation for tensor-product isogeometric analysis: The use of sum factorization. *Computer Methods in Applied Mechanics and Engineering*, 285(Supplement C):817 – 828, 2015. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2014.12.013. URL `http://www.sciencedirect.com/science/article/pii/S0045782514004927`.

[4] M. Barton and V. Calo. Optimal quadrature rules for isogeometric analysis. *ArXiv e-prints*, November 2015.

[5] Y. Bazilevs, V.M. Calo, J.A. Cottrell, J.A. Evans, T.J.R. Hughes, S. Lipton, M.A. Scott, and T.W. Sederberg. Isogeometric analysis using t-splines. *Computer Methods in Applied Mechanics and Engineering*, 199 (5):229 – 263, 2010. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2009.02.036. URL `http://www.sciencedirect.com/science/article/pii/S0045782509000875`. Computational Geometry and Analysis.

[6] Adi Ben-Isreal and Thomas N.E. Grevilee. *Generalized Inverses: Theory and Applications*. Springer-Verlag New York, Inc., 2003.

[7] F. Calabrò, G. Sangalli, and M. Tani. Fast formation of isogeometric galerkin matrices by weighted quadrature. *Computer Methods in Applied Mechanics and Engineering*, 316(Supplement C):606 – 622, 2017. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2016.09.013. URL `http://www.sciencedirect.com/science/article/pii/S0045782516311495`. Special Issue on Isogeometric Analysis: Progress and Challenges.

[8] H. B. Curry and Schoenberg I. J. On spline distributions and their limits: The Pólya distribution functions. *Bulletin of the AMS*, 53, 1947.

[9] H. B. Curry and Schoenberg I. J. On Pólya frequency functions IV: The fundamental spline functions and their limits. *Journal d'Analyse Mathematique*, 1966.

[10] C. De Boor. Splines as Linear Combination of B-splines. A Survey. In G.G. Lorents, C.K. Chui, and L.L. Schumaker, editors, *Aproximation Theory II*, pages 1–47. Academic Press, 1976.

[11] Carl de Boor. On calculating with B-splines. *Journal of Approximation Theory*, 6(1):50 – 62, 1972. ISSN 0021-9045. doi: https://doi.org/10.1016/0021-9045(72)90080-9. URL `http://www.sciencedirect.com/science/article/pii/0021904572900809`.

[12] Carl de Boor. B(asic)-Spline Basics. August 1984.

[13] Carlotta Giannelli, Bert Jütler, and Hendrik Speleers. Thb-splines: The truncated basis for hierarchical splines. *Comput. Aided Geom. Des.*, 29(7):485–498, October 2012. ISSN 0167-8396. doi: 10.1016/j.cagd.2012.03.025. URL `http://dx.doi.org/10.1016/j.cagd.2012.03.025`.

[14] Wolter Groenevelt. Approximation theory lecture notes, February 2016.

[15] Magnus R. Hestenes and Eduard Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.

[16] René R. Hiemstra, Francesco Calabrò, Dominik Schillinger, and Thomas J.R. Hughes. Optimal and reduced quadrature rules for tensor product and hierarchically refined splines in isogeometric analysis. *Computer Methods in Applied Mechanics and Engineering*, 316:966 – 1004, 2017. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2016.10.049. URL http://www.sciencedirect.com/science/article/pii/S004578251631489X. Special Issue on Isogeometric Analysis: Progress and Challenges.

[17] Pieter J. Barendrecht. IsoGeometric Analysis with Subdivision Surfaces, 2013.

[18] Kjetil André Johannessen. Optimal quadrature for univariate and tensor product splines. *Computer Methods in Applied Mechanics and Engineering*, 316(Supplement C):84 – 99, 2017. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2016.04.030. URL http://www.sciencedirect.com/science/article/pii/S004578251630281X. Special Issue on Isogeometric Analysis: Progress and Challenges.

[19] Kjetil André Johannessen, Trond Kvamsdal, and Tor Dokken. Isogeometric analysis using lr b-splines. *Computer Methods in Applied Mechanics and Engineering*, 269(Supplement C):471 – 514, 2014. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2013.09.014. URL http://www.sciencedirect.com/science/article/pii/S0045782513002417.

[20] V. N. Katsikis, D. Pappas, and A. Petralias. An improved method for the computation of the Moore-Penrose inverse matrix. *ArXiv e-prints*, February 2011.

[21] Angelos Mantzaflaris and Bert Jüttler. Integration by interpolation and look-up for Galerkin-based isogeometric analysis. *Computer Methods in Applied Mechanics and Engineering*, 284(Supplement C): 373 – 400, 2015. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2014.09.014. URL http://www.sciencedirect.com/science/article/pii/S0045782514003235. Isogeometric Analysis Special Issue.

[22] Angelos Mantzaflaris, Bert Jüttler, B Khoromskij, and Ulrich Langer. Matrix Generation in Isogeometric Analysis by Low Rank Tensor Approximation. 9213, 11 2014.

[23] Jerrold E. Marsden and Anthony J. Tromba. *Vector calculus.* W. H. Freemand and Company, 2003.

[24] Maxeler. Multiscale dataflow programming, 2015.

[25] Matthias Möller. Assembly strategies in isogeometric analysis. 2015.

[26] Les Piegl and Wayne Tiller. *The NURBS Book*. Springer-Verlag New York, Inc., New York, NY, USA, 1997. ISBN 3-540-61545-8.

[27] communicated by J. A. Todd R. Penrose. A generalized inverse for matrices, 1954.

[28] Jonathan Richard Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, 1994.

[29] Tomas Sauer. Splines in Industrial Applications lecture notes, February 2007.

[30] Mattia Tani. A preconditioning strategy for linear systems arising from nonsymmetric schemes in isogeometric analysis. *Computers & Mathematics with Applications*, 2017. ISSN 0898-1221. doi: https://doi.org/10.1016/j.camwa.2017.06.013. URL http://www.sciencedirect.com/science/article/pii/S0898122117303619.

[31] H. A. van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, March 1992. ISSN 0196-5204. doi: 10.1137/0913035. URL http://dx.doi.org/10.1137/0913035.

[32] J. van Kan, A. Segal, and F. Vermolen. *Numerical Methods in Scientific Computing.* 2014.

[33] Xilinx. *Large FPGA Methodology Guide*. January 2012.