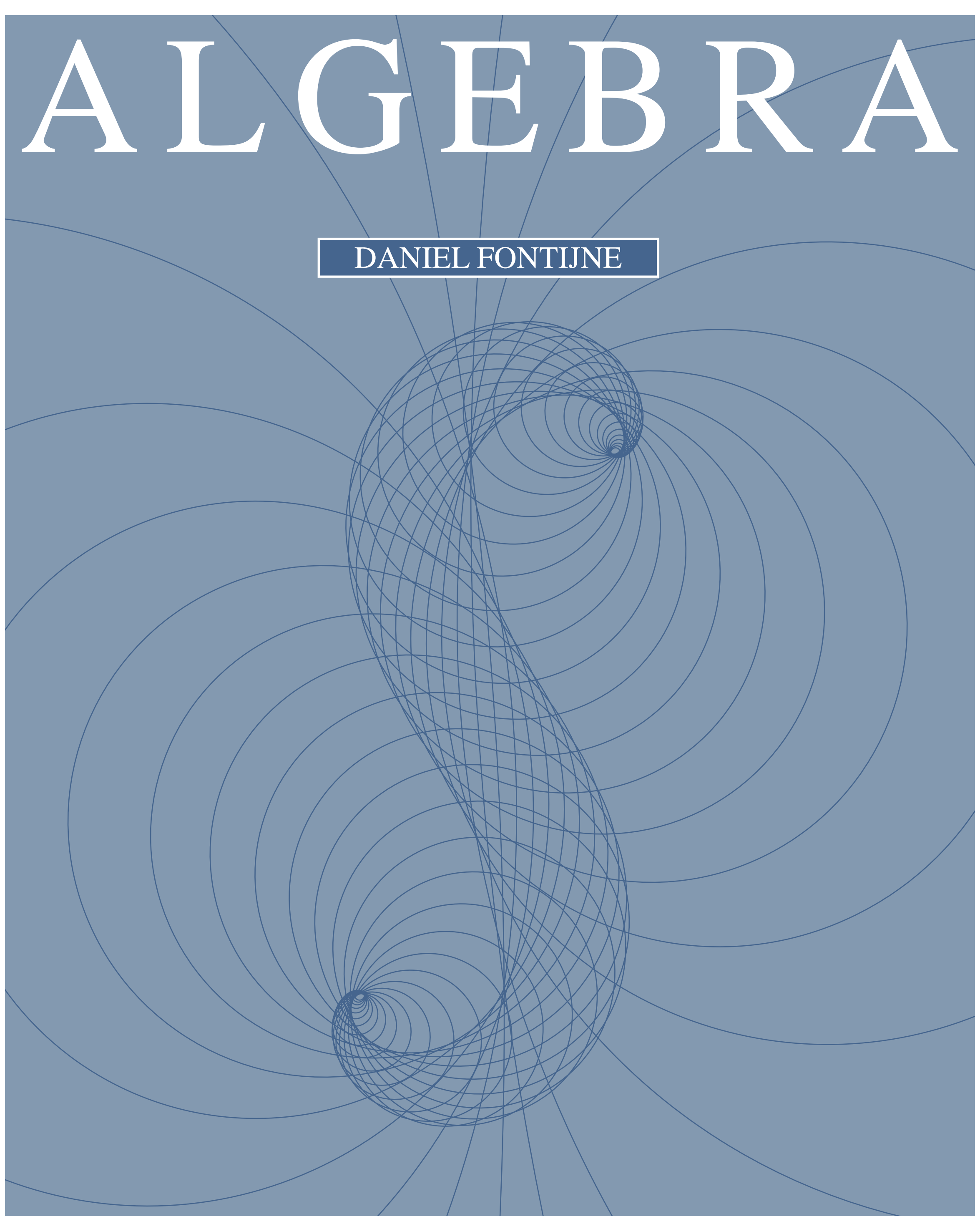
EFFICIENT IMPLEMENTATION OF

GEOMETRIC



Efficient Implementation of Geometric Algebra

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Agnietenkapel der Universiteit
op dinsdag 16 oktober 2007, te 10:00 uur.

door
Daniël Hendrikus Franciscus Dijkman
geboren te Tubbergen

Promotie commisie:

Promotor:	Prof. dr. ir. F.C.A. Groen
Co-promotor:	Dr. ir. L. Dorst
Co-promotor:	Dr. D. Grune
Overige leden:	Prof. dr. M. Boasson
	Dr. P. van Emde Boas
	Prof. dr. P. Klint
	Dr. S. Mann
	Prof. dr. M. Overmars
	Dr. H. Pijls

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

UNIVERSITEIT VAN AMSTERDAM

This work has been performed at the IAS group of the University of Amsterdam and has been supported by the Dutch organization for Scientific Research NWO (project number 612.012.006).

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Portions of this work are from the book, Geometric Algebra for Computer Science, by Leo Dorst, Daniel Fontijne, and Stephen Mann, published by Morgan Kaufmann Publishers, Copyright © 2007 by Elsevier Inc. All rights reserved.

This book has been typeset by the author using LaTeX 2_ε.
Cover design by: Daniel Fontijne.
Printed by: BOX Press.

ISBN-13: 978-90-889-10-142
© 2007 Daniel Fontijne, all rights reserved.

Contents

1 Introduction **1**
1.1 Using Linear Algebra to Encode Geometry 3
1.2 The Alternative: Geometric Algebra 5
1.3 What Makes Efficient Implementation of Geometric Algebra Non-Trivial? 6
1.4 Contributions of the Thesis 6
1.5 Overview of the thesis 7

2 Introduction to Geometric Algebra **9**
2.1 Overview of the Introduction 10
2.2 Notation and Terminology 10
2.3 Vector Spaces 11
2.4 The Outer Product 13
2.5 The Metric Products 19
2.6 The Geometric Product 24
2.7 From Geometric Product to Subspace Products 31
2.8 Basic Constructions in Geometric Algebra 33
2.9 Orthogonal Transformations 39
2.10 The Homogeneous Model 44
2.11 The Conformal Model 49
2.12 Outermorphisms 58
2.13 `meet` and `join` 60
2.14 Executive Summary 62

3 Implementation of Geometric Algebra on an Additive Basis **67**
3.1 The Basis and Operations on Basis Blades 69
3.2 Representing Multivectors and Coordinate Compression 77
3.3 Linear Operations on Multivectors 79
3.4 Non-Linear Operations on Multivectors 85
3.5 Discussion and Summary 97

4 Optimizing the Additive Implementation 99
4.1 Existing Additive Implementations 100
4.2 Issues in Efficient Implementation 101
4.3 Resolving the Issues 102
4.4 Generative Programming 104
4.5 `Gaigen 2` Implementation 105
4.6 Implementation: Specification of the Algebra 105
4.7 Implementation: Boiler Plate Code Generation 111
4.8 Implementation: DSL Functions 124
4.9 Implementation: Profiling 137
4.10 Implementation: Full System Overview 141
4.11 Discussion 141

5 Implementation of Geometric Algebra on a Multiplicative Basis 145
5.1 Introduction 146
5.2 Blades 147
5.3 Versors 155
5.4 Simplifying Versor Representations 159
5.5 Complexity Analysis 167
5.6 Implementation 169
5.7 Benchmarks 169
5.8 Comparison to the Additive Implementation 171
5.9 Discussion 171

6 Case study: Using the Algebra to Implement a Ray Tracer 173
6.1 Ray Tracing Basics 175
6.2 The Ray Tracing Algorithm 176
6.3 Representing Meshes 177
6.4 Modeling the Scene 182
6.5 Tracing the Rays 184
6.6 Shading 191
6.7 Evaluation 192

7 Benchmarks 193
7.1 Ray Tracer Benchmarks 194
7.2 Inverse Kinematics 197
7.3 Minor Benchmarks 198
7.4 Comparison of Various Implementations 199
7.5 Discussion and Conclusion 201

8 Discussion and Conclusion 203
8.1 Suitability of Geometric Algebra for Implementing Geometry on a Computer 204
8.2 `Gaigen 2` 204

8.3 Multiplicative Representation 206
8.4 Conformal model 206

Appendix A: Subspace Products from the Geometric Product 209
A.1 Outer Product from Geometric Product (for Vectors) 210
A.2 Left Contraction from Geometric Product (for Vectors) 211
A.3 The Outer Product from the Geometric Product (General) 212
A.4 The Metric Products from the Geometric Product (General) 212

Appendix B: Ray Tracer Tables 215

Appendix C: Samenvatting (Summary in Dutch) 219

Appendix D: Abstract 223

List of Figures

1.1 Path from real world problem to computer implementation 3

2.1 Influential people in the development of geometric algebra 12
2.2 Blades interpreted as subspaces 15
2.3 Venn diagram of multivector classification 17
2.4 Computing the left contraction of a vector and a 2-blade 23
2.5 Non-invertibility of the subspace products 25
2.6 The dual in 2-D space 34
2.7 The 3-D-only cross product 35
2.8 Orthogonal projection 36
2.9 Line and plane reflection 40
2.10 Rotation of a vector 41
2.11 Point and basis in the homogeneous model 45
2.12 Lines in the homogeneous model 47
2.13 Interpreting blades in the conformal model 53
2.14 Tangent blades in the conformal model 54
2.15 Classification of the blades and versors of the conformal model . . 56

3.1 Function `canonicalReorderingSign(int a, int b)` 73
3.2 Function `gp_op(BasisBlade a, BasisBlade b, boolean outer)` 74
3.3 Function `gp(BasisBlade a, BasisBlade b, double[] m)` 75
3.4 Matrices for geometric product, outer product and left contraction 81
3.5 Implementation of outer product of multivectors 84
3.6 Function `multivectorType(Multivector A, Metric M)` 89
3.7 Venn diagrams illustrating `meet`, `join` 94

4.1 Basic tool chain from source code to running application 104
4.2 The full `Gaigen 2` code generation process 106
4.3 Implementation details in a Gaigen 2 specification file 108
4.4 General multivector definition in `.gs2` file 108
4.5 Example of an active template 112
4.6 Example of a compiled active template 113
4.7 Output of an active template 114

4.8 Example of a `set()` function 118
4.9 Another example of a `set()` function 119
4.10 Specialized outermorphism class initialization 123
4.11 Partial grammar of the `Gaigen 2` DSL language 126
4.12 Some examples of DSL functions 128
4.13 The CBLP grammar 129
4.14 Example of expanding an outermorphism application 133
4.15 Some rules from the term rewrite system 135
4.16 Generated `C++` source code for the geometric product. 138
4.17 `C++` AST of code during various stages of optimization 140
4.18 The full `Gaigen 2` code generation process 141

5.1 A blade stored in factored representation 148
5.2 Benchmarks of the multiplicative and additive implementations . . 170

6.1 Teapot polygonal mesh 178
6.2 Computing the bounding sphere of a mesh 181
6.3 Constructing a bisection plane for a BSP tree 183
6.4 Casting rays for each pixel of the camera sensor 187
6.5 Computing whether a ray intersects a bounding sphere 188

List of Tables

3.1 Bitmap representation of basis blades 71
3.2 Time complexities of the linear operations in the additive implementation 85

5.1 Linear algebra operations used by the multiplicative implementation 168
5.2 Time complexity of blade algorithms in the multiplicative implementation 168
5.3 Time complexity of versor algorithms in the multiplicative implementation 168

7.1 First generation ray tracer performance benchmarks 195
7.2 Second generation ray tracer performance benchmarks 196
7.3 Source code size and compilation times for the second generation ray tracer 197
7.4 Benchmarks of `Gaigen 1` and `MV` 200

B.1 Table of ray tracer primitives 217
B.2 Table of ray tracer operations 218

Origin of Chapters

Chapters 3 through 7 and Appendix A are based on several papers, a book and a talk, as follows:

Chapter 3 is based on Chapters 18-20 of [12].

Chapter 4 is based on [18] and Chapter 22 of [12].

Chapter 5 is based on [17] (a talk).

Chapter 6 is based on [19] and Chapter 23 of [12].

Chapter 7 has results from [12, 18, 19, 28, 34].

Appendix A is almost a direct copy of Appendix C of [12], which was written by Leo Dorst.

Chapter 1

Introduction

Modern computer science is rapidly expanding into areas in which the understanding and representation of spatial data is important, such as 3-D reconstruction of a crime scene from photographs, analyzing human behavior from videos, or rendering measured motions as virtual characters in movies.

Geometry is essential to these applications. One may like to think of it as solved long ago, but the new need for integrated systems exposes the shortcomings in the traditional approach. As long as geometrical systems for graphics, CAD-CAM or robotics were stand-alone, specific techniques could be developed for those applications, meant to be used by specialized users. Now that the various systems need to be integrated and 'talk' to each other, a unifying high-level language for geometry is required to enable both the specification of geometrical operations, and the consistent exchange of geometrical information.

In any geometrical application, one has primitives (used to model the world), which combine using operations (such as spanning or intersection), and that move under certain transformations (rigid body motions, or projective imaging). Each of those elements of geometry and their relationships need to be modeled consistently in a proper algebra for geometry before we can hope to obtain a consistent mathematical formalization which can in turn lead to a clean computer implementation.

The currently used language of linear algebra falls short both in its representational power (its primitives are too low-level), and in its lack of universal geometric operations and transformation. As a consequence, present-day software for geometry contains many tricks and ad hoc extensions, such as overloading the use of vectors, or invoking quaternions. The various formalisms which are used require increasingly more involved and specialized conversions, degrading performance and flexibility and leading to implementation errors. To hide these problems, one may encapsulate algebra elements in objects (as [33] does, in a way), but this causes an extra layer of software, as illustrated in Figure 1.1.

Fortunately, there is something fundamentally better available. In recent developments, the 19th century work in *geometric algebra* has been shown to be promising as the unifying language leading to extended primitives for modeling and universal representation of geometrically significant operations. Geometric algebra removes the need for the extra layer of software (since there are much less problems to hide) while at the same time providing a much richer mathematical language for geometry. However, the current literature in geometric algebra emphasizes the mathematical beauty much more than its use for computer applications of geometry, and that is not good enough for computer science.

This thesis addresses the computational and implementational aspects of geometric algebra, and shows that its mathematical promise can be made into programming reality: geometric algebra provides a modular, structured specification language for geometry whose implementations can be automatically generated, leading to an efficiency that is competitive with the (hand-) optimized code based on the traditional linear algebra approach.

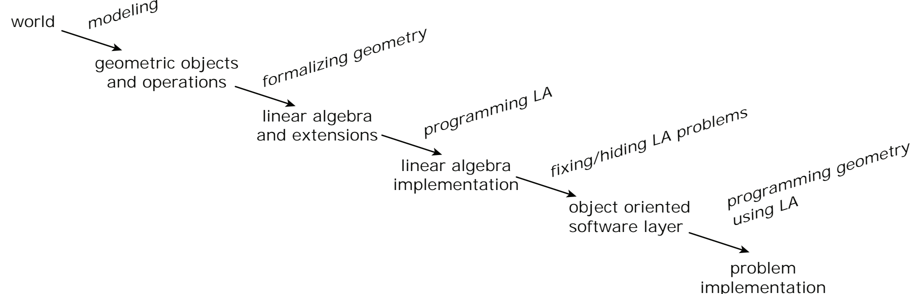


Figure 1.1: *The path from a real world geometric problem to a computer implementation using linear algebra. The world is modeled as geometric objects and operations, which are (traditionally) formalized using linear algebra. The linear algebra is implemented, and optionally an object oriented layer is added on top of this to fix the typical problems of linear algebra, as described in the main text. On top of this, the actual geometry of the problem is implemented.*
By replacing linear algebra with geometric algebra, the formalization of geometry is beautified, the (geometric algebra) implementation can be largely automated (as described in this thesis) and there is much less need for an object oriented software layer to hide problems.

1.1 Using Linear Algebra to Encode Geometry

Let us expose some of the shortcomings of the typical linear algebra approach to implementing geometry. Linear algebra formalizes n-dimensional vector spaces, metrics, and linear transformations. Using vectors, one can formalize directions in space, and matrices can be used to formalize transformations of that space. Creative use of these concepts allows one to describe and solve a lot of geometric problems.

The Limited Vocabulary of Linear Algebra

The problem with linear algebra is that its vocabulary is rather limited, which quickly leads to all kinds of kludges and extensions. We give some examples.

Vectors formalize directions, but in geometry one often needs points. This is solved by introducing an 'origin', relative to which vectors can indicate points. One of the problems with this idea is that the semantic difference between a *direction vector* and a *point vector* is not encoded in the vector type itself. In a computer program, both concepts are represented by the same vector type, and their interpretation is up to the implementer. For example, the implementer must remember that in some situations, point vectors transform differently than direction vectors.

This problem becomes larger as more and more types of objects must be represented. Planes are common objects in geometry, and in linear algebra they are usually represented using a vector perpendicular to the plane (a normal vector), plus a scalar distance to the origin. Lines through the origin can be represented using direction vectors, but in general lines additionally require a position vector relative to the origin. Another way to represent lines is as the intersection of two planes.

By now, we have introduced three geometric concepts which are represented by the same vector type: direction vectors, point vectors, and normal vectors. However, each geometric concept behaves differently in subtle ways. For example, the sum of two direction vectors is another valid direction vector, but the sum of two point vectors does not have a straightforward geometric interpretation. Such subtle differences can easily lead to (implementation) errors.

One may partially clean up the situation by using projective geometry and *Plücker coordinates*. Through Plücker coordinates, points, lines and planes each get their own type, so they do not have to be 'assembled' from vectors and scalars anymore. Unfortunately, the connection between Plücker coordinates and linear algebra is not straightforward, while linear algebra is still essential for representing and applying transformations.

Linear geometric transformations are usually represented using matrices (e.g., a rotation matrix), vectors (e.g., a translation vector) and scalars (for scaling). Projective geometry improves on this, since it allows one matrix to represent all three types of transformations. But to apply these matrices to points, lines, and planes different implementations have to be written for each case. E.g., the projective matrix that can transform the Plücker coordinates of a point cannot be applied directly to the Plücker coordinates of a plane, let alone a line.

It also turns out that matrices are not the best way to handle rotations. Matrices encode the essential geometric properties of a rotation (such as axis and angle) rather redundantly and inaccessibly. *Quaternions* [30, 38] are much better suited for handling rotations. But this is another extension that is not automatically well-connected to linear algebra, nor to Plücker coordinates.

All these loosely connected formalisms rapidly become a burden. It is hard for non-experts to understand all these formalisms, and even for experts reasoning about such problems is harder than it should be because not all elements are part of the same algebra. Computer implementations also become more complex, because all these connections have to be encoded by hand (e.g., applying a quaternion to a Plücker line). Jumping back and forth between formalisms requires custom and ad hoc conversions and thus can again easily result in errors.

Linear Algebra Clean-up Efforts

Several efforts have been made to clean up this situation, at various point in the path of Figure 1.1. Blinn studies the use of tensor diagrams to more properly integrate linear algebra and the extension described above [4], which indeed leads

to a more structured formalism. Stolfi [40] presents a detailed and enlightening discussion of projective geometry and Plücker coordinates, taking orientation of objects into consideration. This structures the existing linear algebra formalization. Goldman has studied the (ab-)use of vectors in detail leading to which expressions are allowed when vectors represent both directions and point [21], and how the weight of points in projective space may be put to use and how they may be added [22]. This helps to avoid some of the typical problems associated with linear algebra-based geometry. Mann, Litke and DeRose clean up the semantics of points, vectors, frames, transformations and so on, by encapsulating them in objects [33].

1.2 The Alternative: Geometric Algebra

Rather than trying to fix the limitations of linear algebra, one may use geometric algebra [10, 12, 25, 27] to formalize geometry. Like linear algebra, it formalizes vector spaces, but does not stop at just the vectors. In geometric algebra, vectors can be combined using the *outer product* to form higher-dimensional objects like flat surfaces (i.e., *bivectors*) and volumes (i.e., *trivectors*). These bivectors and trivectors are regular elements of the algebra, while they have no direct representation in linear algebra.

The fundamental product of geometric algebra is the *geometric product*. It allows for a simple formulation of orthogonal transformations, the most basic of which is reflection. By combining multiple reflections into *versors*, more complicated transformations such as rotations can be constructed. Using this technique, all orthogonal transformations can be directly represented by elements of the algebra.

Smart interpretation of geometric algebras extends the range of objects and transformations that can be represented. For example, in the *conformal model*, translations are orthogonal transformations, too, and primitives such as circles, spheres and rays have direct representations. *The ultimate goal is that all basic objects and transformations that are relevant to a certain geometry are represented directly by elements of some appropriate geometric algebra.*

Because of the consistent structure of geometric algebra, equations are often 'universal' and generally applicable. For example, the expression to rotate a point is no different from the expression to rotate a plane; the expression for intersecting a circle with a plane is no different from the expression to intersect a line with a sphere.

Geometric algebra has its own calculus that allows one to formalize changes (e.g., over time or space), just like vector calculus does for linear algebra, but now it works on all elements of the algebra. *Geometric algebra unifies all formalisms discussed so far, and more.* One algebra to rule them all, as it were.

1.3 What Makes Efficient Implementation of Geometric Algebra Non-Trivial?

Unfortunately, the power of geometric algebra comes at a price. A naive implementation of geometric algebra can easily be about 100 times slower than traditional (linear-algebra-based) geometry implementations. We demonstrate this in detail in Chapter 7. The core problem in implementing geometric algebra is the inherent high dimensionality of the algebra.

The elements of a geometric algebra over an n-D space live in a 2^n-dimensional space. This high dimensionality is how geometric algebra manages to represent all these different objects and transformations. As a result, the elements of a geometric algebra over a (say) 3-D space have eight coordinates. The number of coordinates grows quickly as n goes up. Elements of a 5-D algebra — which is used often in practice — have 32 coordinates. Bilinear products over these elements would multiply and add in the order of $32^2 = 1024$ combinations of coordinates. On first sight, these numbers makes it look as if practical use of geometric algebra is intractable.

Luckily, the structure of geometric algebra is such that the elements make sparse use of their coordinates. Geometrically meaningful elements use half of their coordinates in the worst case — the other coordinates are zero. In practice, the elements are even more sparse, such that an optimized implementation can approach or even exceed the performance of its traditional equivalent. This thesis shows how to achieve this.

1.4 Contributions of the Thesis

The main goal and contribution of this thesis is to show that (automatically generated) computer implementations of geometric algebra can perform at a level comparable to standard geometry implementations. By this we mean that if some piece of geometry is implemented through geometric algebra, the result should be about as efficient (in terms of computation time and memory usage) as the traditional linear algebra implementation. This should be so even without taking into account certain algorithmic enhancements that geometric algebra may allow in selected applications (such as in [28]).

This thesis describes two implementation approaches. The first approach is best suited for low-dimensional spaces ($\leq$ 10-D) and uses a 'geometry compiler' to translate high-level functions over the algebra into low-level `C++` or `Java` code. The fact that it is possible to build such a compiler shows that the extensive structure of geometric algebra is not a burden to the implementer but instead offers great opportunities for automatic code generation and analysis. Our benchmarks show that our automatically generated implementations have performance comparable to hand-optimized linear algebra implementations.

The second approach is more theoretical and best suited for high-dimensional geometric algebras ($\geq$ 10-D). It makes use of the fact that all geometrically meaningful elements can be written as a product of vectors, which significantly reduces storage and time complexity.

As a side effect of our work, we also systematically describe how to implement the basic geometric algebra operations, for both types of implementations. We have not found such a treatment elsewhere. The automatic computation of geometric products using the bitmap representation in arbitrary metrics is also novel.

We have also systematically compared the performance of geometric algebra to traditional geometry implementations in a ray tracing application. This allows for a real-world comparison of the relative elegance and performance of these approaches to implementing geometry.

1.5 Overview of the thesis

The thesis starts with a general introduction to geometric algebra in Chapter 2. Besides an introduction, this chapter also gives more motivation of why one would want to use geometric algebra, it fixes notation, and it sets the scene for (optimization) examples that are used later in the thesis. For those readers who want to fast-forward to the actual content of the thesis, we provide an 'executive summary' at the end of the chapter.

Chapter 3 is the real start of the thesis. It describes how to implement geometric algebra on an additive basis. We call this the *additive implementation*. This method represents elements of the algebra as a weighted sum of basis elements. It is best suited for low-dimensional geometric algebras. This approach is currently most common (and useful) in computer science.

Chapter 4 describes how the ideas of Chapter 3 can be implemented in such a way that we achieve an efficiency comparable to traditional geometry implementations. At the start of the chapter we list existing implementations and motivate why our new implementation was required to improve efficiency. The main idea of our implementation is to exploit the structure of geometric algebra in a way that saves memory and computation time. We implemented this in our Geometric Algebra Implementation Generator `Gaigen 2`.

Chapter 5 presents an alternative to the additive representation of Chapters 3 and 4. It shows how to implement geometric algebra using multiplicative (factored) representations of the elements of the algebra. We call this the *multiplicative implementation*. Not only does this method expose a connection between geometric algebra and linear algebra, it also makes it possible to use geometric algebra in high-dimensional spaces, where the additive implementation is intractable.

Chapter 6 details the implementation of a realistic application of geometric algebra: a ray tracer, implemented through conformal geometric algebra. It serves

as a case study of using the additive implementation of Chapters 3 and 4.

Chapter 7 presents extensive benchmarks of our additive implementation. The main set of benchmarks is based on the ray tracer. We implemented the ray tracing algorithm multiple times, each time using different geometry implementation approaches (e.g., basic 3-D linear algebra, homogeneous coordinates, conformal model) and different GA implementations (`Gaigen 1`, `Gaigen 2`, `CLU`). This allows us to compare the relative efficiency of these geometry implementations in a realistic environment. We also present many other — less extensive — benchmarks.

Chapter 8 summarizes our findings and pinpoints areas that could be improved.

Chapter 2

Introduction to Geometric Algebra

This chapter introduces geometric algebra, from the basics to the conformal model. From the viewpoint of the thesis, the goal is to enable stand-alone reading of the thesis, to fix notation and to set the scene for (optimization) examples later on. For a thorough treatment of geometric algebra we recommend [12] for computer scientists, and [10] for physicists.

Not all readers may want to read the full (40 page) introduction. For that reason, we provide a three-page 'executive summary' at the end of the chapter, in Section 2.14. Reading this summary should be enough to follow the main arguments of later chapters. It is still recommended to read the section "Notation and Terminology" right below.

2.1 Overview of the Introduction

The introduction uses a bottom-up approach: first the *outer product* and blades are treated. This is followed by the introduction of metric and the *metric products* (a generalization of the dot product from linear algebra). Together, the outer product and the metric products form an algebra of subspaces, and hence we will use the term *subspace products* to refer to them collectively. About halfway through the chapter the geometric product is introduced. The geometric product is the most fundamental product of geometric algebra. We will show how to derive the subspace products from the geometric product using grade part selection.

This is followed by treatment of the basic geometric constructions that can be made using these products, such as projection and rotation. Then we give a brief presentation of the homogeneous model, after which we quickly turn to the more powerful conformal model. Outermorphisms and the non-linear products `meet` and `join` are discussed after that; their introduction is delayed in order not to disrupt the flow of the chapter.

Along the way, we will give some notes on implementation to make the presentation a bit more tangible. However, detailed treatment of implementation is found in Chapters 3 through 5.

As we progress in this chapter, we will gradually get less formal in our treatment. For our implementation purposes, it is important to get the basics right, so we treat these rigorously. From the implementation viewpoint the homogeneous and conformal models are 'mere' applications, so we discuss them in less detail.

2.2 Notation and Terminology

2.2.1 Fonts

With some exceptions, fonts carry the following meaning throughout this thesis:

- *Scalars*: Greek (α).
- *Vectors*: lowercase bold ($\mathbf{a}$).

- *Other multivectors, blades or versors*: uppercase bold (**A**).
- *Matrices*: uppercase italic (A).
- *column or row matrices*: lowercase italic (a).

Note that we use the term "column matrix" instead of vector to refer to a $k \times 1$ matrix. Likewise for "row matrix". Sometimes we use [**A**] to denote the matrix representation of multivector **A**. This should be clear from the context.

To address matrix elements (row i, column j), we use the notation $A_{[i,j]}$. For row or column matrices this may be simplified to $a_{[i]}$, leaving out the other index. To address whole rows or columns we use the Matlab colon-notation. E.g., $A_{[:,j]}$ means column j of A.

2.2.2 Directions, Orientations, Attitudes and Weights

Because we will greatly extend the number of primitive geometric objects available as elements of computation, some definitions are in place to be able to discuss various geometric properties. These definitions use a vector **a** as the example, but the principles also apply to other elements.

- *attitude*: denotes purely the subspace occupied by the vector. **a**, −**a** and 2 **a** all have the same attitude.
- *orientation*: **a** and −**a** have a different orientation. **a** and 2 **a** have the same orientation. Orientation is always relative to some 'standard orientation'.
- *weight*: 2 **a** has twice the weight of **a**. Weight is always relative to some 'standard unit direction'. So weight can be negative, too, for a vector with negative orientation.
- *direction*: is the combination of attitude and orientation. So a direction denotes an oriented subspace, without taking the weight into account.

2.3 Vector Spaces

Like linear algebra, formal treatment of geometric algebra starts with the definition of *vector spaces*. In principle vector spaces can be defined over arbitrary fields, but for our purposes we need only the ordinary reals. By definition such a vector space $\mathbb{R}^n$ consists of elements called *vectors*, and addition and multiplication with reals (called *scalars*) such that:

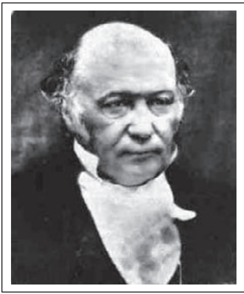
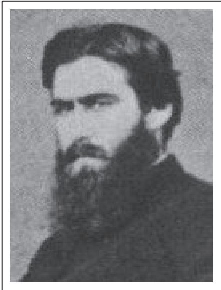
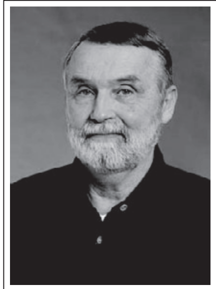
  

Figure 2.1: *Influential people in the development of Clifford algebra and geometric algebra. From left to right: Hermann Günther Grassmann (1809-1877), William Rowan Hamilton (1805-1865), William Kingdon Clifford (1845-1879), David Orlin Hestenes (1933-).*

closure under addition:	$\mathbf{x}+\mathbf{y}$	$\in$	$\mathbb{R}^n$	$\forall\, \mathbf{x},\mathbf{y} \in \mathbb{R}^n$
associativity:	$(\mathbf{x}+\mathbf{y})+\mathbf{z}$	$=$	$\mathbf{x}+(\mathbf{y}+\mathbf{z})$	$\forall\, \mathbf{x},\mathbf{y},\mathbf{z} \in \mathbb{R}^n$
identity element:	$\exists\, \mathbf{0} \in \mathbb{R}^n$	$:$	$\mathbf{x}+\mathbf{0}=\mathbf{x}$	$\forall\, \mathbf{x} \in \mathbb{R}^n$
inverse:	$\exists\, \mathbf{y} \in \mathbb{R}^n$	$:$	$\mathbf{x}+\mathbf{y}=\mathbf{0}$	$\forall\, \mathbf{x} \in \mathbb{R}^n$
commutativity:	$\mathbf{x}+\mathbf{y}$	$=$	$\mathbf{y}+\mathbf{x}$	$\forall\, \mathbf{x},\mathbf{y} \in \mathbb{R}^n$
distributivity (vector):	$\alpha\,(\mathbf{x}+\mathbf{y})$	$=$	$\alpha\,\mathbf{x}+\alpha\,\mathbf{y}$	$\forall\, \alpha \in \mathbb{R}, \mathbf{x},\mathbf{y} \in \mathbb{R}^n$
distributivity (scalar):	$(\alpha+\beta)\,\mathbf{x}$	$=$	$\alpha\,\mathbf{x}+\beta\,\mathbf{x}$	$\forall\, \alpha \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^n$
compatibility:	$(\alpha\,\beta)\,\mathbf{x}$	$=$	$\alpha\,(\beta\,\mathbf{x})$	$\forall\, \alpha \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^n$
scalar identity:	$1\,\mathbf{x}$	$=$	$\mathbf{x}$	$\forall\, \mathbf{x} \in \mathbb{R}^n$

Vectors from such a vector space are traditionally used to represent a number of geometric concepts:

1. (Weighted) directions. The weight is the length of the vector.

2. Points. A vector can be used to indicate to a point relative to some origin.

3. Translations over a vector.

4. Lines through the origin.

5. (Hyper-)planes through the origin, by their normal vector.

The problem with this versatile use of the vector type is that the semantics are all *implicit*. For each vector in an equation or computer program, the precise meaning and interpretation must be described in documentation or code comments. The vector itself does not hold any information that reveals what geometric concept it represents.

In object oriented programming, it is considered bad practice to use a single type to represent so many different concepts. As will become clear in this chapter, geometric algebra offers much more and better 'types' for the concepts listed above. The first step towards this goal is to enable the use of oriented subspaces as direct elements of computation.

2.4 The Outer Product

As stated before, in linear algebra-based geometry, a line is usually described using some composition of vectors. For example, a line can be represented as two point vectors, a point vector and a direction vector, or the intersection of two planes. What one really wants to do is represent more-than-one-dimensional-subspaces, possibly as a list of vectors that span the subspace.

It turns out that defining a product that formalizes the idea of 'an ordered list of vectors that span a subspace' leads to powerful representation of oriented subspaces. Let us use the *wedge* symbol $\wedge$ to denote the action of creating a such a list of vectors. We will call this action the *outer product*. So $\mathbf{a}\wedge\mathbf{b}$ (pronounced $\mathbf{a}$ wedge $\mathbf{b}$) represents the subspace spanned by vectors $\mathbf{a}$ and $\mathbf{b}$. $\mathbf{a}\wedge\mathbf{b}\wedge\mathbf{c}$ represents the subspace spanned by three vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$. And so on. However, if we demand that this product has certain reasonable properties, we find that it cannot retain all properties of the individual vectors that made up the list.

For example, it is reasonable to demand that it is bilinear (linear is each of its arguments). But this implies that $(2\,\mathbf{a})\wedge\mathbf{b} = \mathbf{a}\wedge(2\,\mathbf{b})$, so information about the weight of the individual vectors is lost in the product; only the overall weight is retained.

We would also like it to retain the order of the vectors. This is what we called orientation above in Section 2.2.2. For example, we define that $\mathbf{a}\wedge\mathbf{b} = -\mathbf{b}\wedge\mathbf{a}$. However, replacing $\mathbf{b}$ with $\mathbf{a}$ in this last equation gives $\mathbf{a}\wedge\mathbf{a} = -\mathbf{a}\wedge\mathbf{a}$, which implies that $\mathbf{a}\wedge\mathbf{a} = 0$.

As a consequence of linearity and orientation preservation, even more information must be lost in this product:

$$\mathbf{a}\wedge(\mathbf{a}+\mathbf{b}) = \mathbf{a}\wedge\mathbf{a}+\mathbf{a}\wedge\mathbf{b} = \mathbf{a}\wedge\mathbf{b}$$

In essence, the outer product must 'reject' the parallel part of its operands. Moreover, when the operands of the outer product are dependent, the result must also be zero. For example:

$$\mathbf{a}\wedge\mathbf{b}\wedge(\mathbf{a}+\mathbf{b}) = \mathbf{a}\wedge\mathbf{b}\wedge\mathbf{a}+\mathbf{a}\wedge\mathbf{b}\wedge\mathbf{b} = -\mathbf{a}\wedge\mathbf{a}\wedge\mathbf{b}+\mathbf{a}\wedge\mathbf{b}\wedge\mathbf{b} = 0$$

2.4.1 Definition of the Outer Product

We can formalize the ideas from the previous section as follows:

$$\begin{aligned}
\text{anti-symmetry:} &\quad \mathbf{a} \wedge \mathbf{b} = -\mathbf{b} \wedge \mathbf{a} \\
\text{scaling:} &\quad \mathbf{a} \wedge (\beta \mathbf{b}) = \beta\, (\mathbf{a} \wedge \mathbf{b}) \\
\text{distributivity:} &\quad \mathbf{a} \wedge (\mathbf{b} + \mathbf{c}) = (\mathbf{a} \wedge \mathbf{b}) + (\mathbf{a} \wedge \mathbf{c}) \\
\text{associativity:} &\quad \mathbf{a} \wedge (\mathbf{b} \wedge \mathbf{c}) = (\mathbf{a} \wedge \mathbf{b}) \wedge \mathbf{c}
\end{aligned}$$

2.4.2 Blades and Grades

An outer product of vectors is called a *blade*, and to be more specific we call the outer product of k vectors a k-blade. The value of k is called the *grade* of the blade. The term "grade" is used to avoid more general words like "dimensionality".

Conversely, a blade is any element of a geometric algebra that is factorizable under the outer product. This implies that the outer product of blades is also a blade. To bootstrap this recursive definition, we define the scalars and vectors to be blades, too. To indicate that a blade $\mathbf{A}$ is of grade k, we sometimes use the notation $\mathbf{A}_k$

$$\text{blade } \mathbf{A} \text{ is of grade } k: \qquad \mathbf{A}_k.$$

There is one exception to this rule: for vectors (typeset lowercase bold) we often use a subscript letter as an index (as in $\mathbf{a}_i$). We use this notation most often when we require a factorization of a blade.

Due to the properties of the outer product, it is impossible to create a non-zero grade $(n+1)$-blade in a geometric algebra over an n-dimensional vector space. This would require $(n+1)$ independent vectors, which do not exist in such a space. The top-grade blades $\mathbf{A}_n$ in an n-dimensional space are called *pseudoscalars*.

If we denote a grade k blade as an element of $\bigwedge^k \mathbb{R}^n$ (where $\bigwedge^1 \mathbb{R}^n = \mathbb{R}^n$ and $\bigwedge^0 \mathbb{R}^n = \mathbb{R}$), then the outer product of blades is a mapping:

$$\text{outer product} \quad \wedge : \bigwedge^{k} \mathbb{R}^n \times \bigwedge^{l} \mathbb{R}^n \rightarrow \bigwedge^{k+l} \mathbb{R}^n.$$

So the grade of an outer product is the sum the grade of operands.

2.4.3 Interpretation of Blades

In principle, blades are just elements of some algebra. To be useful for doing geometry, we need to give them a certain *interpretation*. This also simplifies reasoning about them [1].

For now, we will interpret blades as oriented subspaces through the origin. This is illustrated in Figure 2.2. The figure shows a scalar (0-blade), a vector

[1] "For geometry without algebra is dumb, and algebra without geometry is blind" [27].

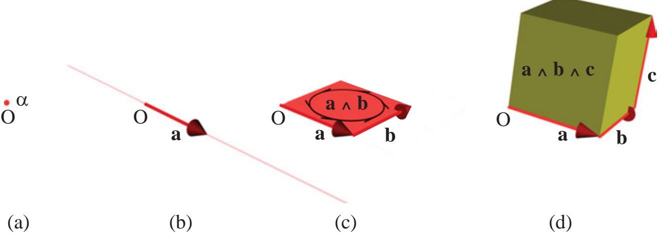


Figure 2.2: *Blades interpreted as subspaces.*

(1-blade), a 2-blade and a 3-blade, relative to the origin denoted by O. The figure also shows how each blade is spanned by vectors **a**, **b**, and **c**. The 2-blade (3-blade) is drawn as a parallelogram (parallelepiped). The surface area (volume) depicts the weight of these blades (see Section 2.7.1), but in principle blades have no specific shape. We could just as well draw them as discs (spheres) with a certain surface area (volume).

In later Sections 2.10 and 2.11 we will give different interpretations to blades, which allows us to represent more complex objects such as lines, circles and rays.

2.4.4 Intuitive Interpretation of Outer Product

When reasoning about geometric algebra equations involving the outer product, it can be helpful to think of the outer product as the 'addition operator' of subspaces, in that the outer product $\mathbf{A} \wedge \mathbf{B}$ spans the subspace that $\mathbf{A}$ and $\mathbf{B}$ span together, as long as $\mathbf{A}$ and $\mathbf{B}$ are independent. One has to be careful with this sort of intuition though, because the properties of the outer product are more subtle than just 'subspace addition'.

2.4.5 Containment of Blades

A blade $\mathbf{A}_k = \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \ldots \wedge \mathbf{a}_k$ is *contained* in another blade $\mathbf{B}$ (that is $\mathbf{A}_k \subseteq \mathbf{B}$) when $\mathbf{a}_i \wedge \mathbf{B} = 0$ for every vector factor $\mathbf{a}_i$. Once we have defined the *delta product* in Section 2.13.2, we will be able to test for containment using the equation

$$\mathbf{A} \wedge (\mathbf{A} \bigtriangleup \mathbf{B}) \neq 0.$$

This equation says that $\mathbf{A}$ should be independent of the symmetric difference of $\mathbf{A}$ and $\mathbf{B}$.

2.4.6 Reversion, Grade Involution

Two simple but useful unary operations are *reversion* and *grade involution*. Reversing a blade means to reverse the order of the factors of a blade, for instance:

$$\widetilde{\mathbf{a} \wedge \mathbf{b}} = \mathbf{b} \wedge \mathbf{a}.$$

Hence reversion only changes the orientation of the blade, which is determined by the sign. In general,

$$\widetilde{\mathbf{A}}_k = (-1)^{\frac{1}{2}k(k-1)}\, \mathbf{A}_k.$$

Note that two consecutive reversions cancel each other, and the reverse of an outer product is the reverse outer product of the reverses:

$$(\widetilde{\mathbf{A}})^{\sim} = \mathbf{A} \quad \text{and} \quad (\mathbf{A} \wedge \mathbf{B})^{\sim} = \widetilde{\mathbf{B}} \wedge \widetilde{\mathbf{A}}.$$

The other operations is grade involution. It toggles the orientation of a blade if its grade is odd:

$$\widehat{\mathbf{A}}_k = (-1)^k\, \mathbf{A}_k.$$

Two consecutive grade involutions cancel each other, and the grade involution of an outer product is outer product of the grade involutions:

$$(\widehat{\mathbf{A}})^{\wedge} = \mathbf{A} \quad \text{and} \quad (\mathbf{A} \wedge \mathbf{B})^{\wedge} = \widehat{\mathbf{A}} \wedge \widehat{\mathbf{B}}.$$

The grade involution is useful in derivations. For example when a blade swaps sides with a vector in an outer product, the grade involution can be used to keep the sign of the whole expression correct, as in $\mathbf{A} \wedge \mathbf{d} = \mathbf{d} \wedge \widehat{\mathbf{A}}$. To see why, note that when $\mathbf{A}$ has odd grade, a odd number of vector-swaps is required, thus causing the minus sign. This is precisely the minus sign that the grade involution produces. For example:

$$\begin{aligned} \mathbf{A} &= \mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}, \\ \mathbf{A} \wedge \mathbf{d} &= \mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c} \wedge \mathbf{d} = -\mathbf{d} \wedge \mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c} = -\mathbf{d} \wedge \mathbf{A} = \mathbf{d} \wedge \widehat{\mathbf{A}}. \end{aligned}$$

2.4.7 Adding Blades, Multivectors

Blades may be added (as in $\mathbf{a} \wedge \mathbf{b} + \mathbf{c} \wedge \mathbf{d}$), but the sum of two blades is not always a blade because it may not be factorizable under the outer product. When the sum is not a blade — or some other special element of the algebra — we call it a *multivector*. By allowing addition of arbitrary blades, we create the 2^n dimensional linear multivector space $\bigwedge \mathbb{R}^n$, of which all multivectors are elements.

The multivector is the general element of computation in a geometric algebra. One may define it as all elements that can be produced by adding blades. Hence a blade is a special type of multivector (one that is factorizable under the outer product). Later on, we will identify other special types of multivectors, such as *versors*, which are multivectors that are factorizable under the geometric product. Figure 2.3 shows a Venn diagram of the multivector types we ultimately identify.

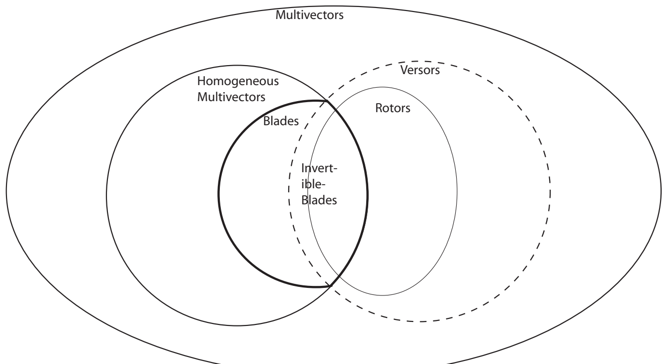


Figure 2.3: *Venn diagram of multivector classification, for the types of multivectors defined in this thesis (note that versors and rotors are introduced in Sections 2.6.5 and 2.9.3, respectively). A textual description of the figure: Every element of the algebra is a multivector. Some multivectors are homogeneous (of a single grade). Some homogeneous multivectors are blades. Some blades are invertible, and in that case, they are versors, too. Some multivectors are versors. Invertible blades are versors. Some versors are rotors, and some invertible blades are rotors.*

Homogeneous Multivectors, Bivectors, Trivectors, Quadvectors

Figure 2.3 also shows one type of multivector that we will not discuss much in this thesis: *homogeneous multivectors*. Homogeneous multivectors are multivectors of a single grade (like blades), but they do not have to be factorizable under the outer product (unlike blades). Homogeneous multivectors can be constructed by summing blades of the same grade. The terms *bivector*, *trivector* and *quadvector* are shorthands for homogeneous multivectors of grade 2, 3 and 4, respectively.

Because homogeneous multivectors usually do not have a geometric interpretation, they are not very useful for doing geometry. As an exception, bivectors are useful when dealing with transformations, see Section 2.9.3.

When is the Sum of Two Blades Another Blade?

By definition, the sum of two blades can only be another blade if both blades have the same grade. But even when both blades have the same grade, the result is a only blade when it is factorizable under the outer product. This leads to the following theorem.
Theorem. The sum of two blades $\mathbf{A}$ and $\mathbf{B}$ is another blade iff:

- they are of the same grade k, and
- they share a common factor of at least grade $k-1$.

Proof. If the blades share a common factor of their own grade k, then they are equal up to a scalar factor, and hence can be added.
If the blades share a common factor $\mathbf{C}_{k-1}$ of one grade less than their own grade, then we can split off a vector and write them as:

$$\begin{aligned}\mathbf{A} &= \mathbf{C}_{k-1} \wedge \mathbf{a}_k, \\ \mathbf{B} &= \mathbf{C}_{k-1} \wedge \mathbf{b}_k, \\ \mathbf{A}+\mathbf{B} &= \mathbf{C}_{k-1} \wedge \mathbf{a}_k + \mathbf{C}_{k-1} \wedge \mathbf{b}_k = \mathbf{C}_{k-1} \wedge (\mathbf{a}_k + \mathbf{b}_k).\end{aligned}$$

The sum of two vectors $(\mathbf{a} + \mathbf{b})$ is always a blade, and the outer product of two blades is always a blade, hence $\mathbf{C}_{k-1} \wedge (\mathbf{a}+\mathbf{b})$ is also blade.

Otherwise, the largest common factor of the blades must be $\mathbf{C}_i$ with $i < (k-1)$. In that case:

$$\begin{aligned}\mathbf{A} &= \mathbf{C}_i \wedge \mathbf{a}_{i+1} \wedge \mathbf{a}_{i+2} \wedge \ldots \wedge \mathbf{a}_k, \\ \mathbf{B} &= \mathbf{C}_i \wedge \mathbf{b}_{i+1} \wedge \mathbf{b}_{i+2} \wedge \ldots \wedge \mathbf{b}_k\end{aligned}$$

Obviously,

$$\mathbf{A}+\mathbf{B} = \mathbf{C}_i \wedge (\mathbf{a}_{i+1} \wedge \mathbf{a}_{i+2} \wedge \ldots \wedge \mathbf{a}_k + \mathbf{b}_{i+1} \wedge \mathbf{b}_{i+2} \wedge \ldots \wedge \mathbf{b}_k)$$

is not a blade, since that would require $(\mathbf{a}_{i+1} \wedge \mathbf{a}_{i+2} \wedge \ldots \wedge \mathbf{a}_k + \mathbf{b}_{i+1} \wedge \mathbf{b}_{i+2} \wedge \ldots \wedge \mathbf{b}_k)$ to be factorizable, which it clearly is not, since any of those factors would have been in the largest common factor $\mathbf{C}_i$. □

The canonical example of a non-factorizable blade-sum is $\mathbf{e}_1 \wedge \mathbf{e}_2 + \mathbf{e}_3 \wedge \mathbf{e}_4$, where the $\mathbf{e}_i$ form an orthonormal basis, as defined next.

2.4.8 A Basis for Multivectors

Multivectors can be represented inside a computer relative to a basis of blades. This forms the foundation of the implementation method discussed in Chapter 3.

First we need a basis for the vector space. Let us use a 3-D space as an example. In geometric algebra, it is common practice to label the basis vectors $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ and so on. These basis vectors should span the whole space, and for computational reasons it is preferable that the basis vectors form an orthonormal set. Relative to such a basis, any 3-D vector $\mathbf{a}$ can be written as

$$\mathbf{a} = \alpha_1 \, \mathbf{e}_1 + \alpha_2 \, \mathbf{e}_2 + \alpha_3 \, \mathbf{e}_3,$$

hence the outer product of two vectors $\mathbf{a}$ and $\mathbf{b}$ can be written as:

$$\begin{aligned}
&\mathbf{a} \wedge \mathbf{b} = \\
&\quad = (\alpha_1 \mathbf{e}_1 + \alpha_2 \mathbf{e}_2 + \alpha_3 \mathbf{e}_3) \wedge (\beta_1 \mathbf{e}_1 + \beta_2 \mathbf{e}_2 + \beta_3 \mathbf{e}_3) \\
&\quad = \alpha_1\beta_1(\mathbf{e}_1 \wedge \mathbf{e}_1) + \alpha_1\beta_2(\mathbf{e}_1 \wedge \mathbf{e}_2) + \alpha_1\beta_3(\mathbf{e}_1 \wedge \mathbf{e}_3) + \\
&\quad\quad\ \alpha_2\beta_1(\mathbf{e}_2 \wedge \mathbf{e}_1) + \alpha_2\beta_2(\mathbf{e}_2 \wedge \mathbf{e}_2) + \alpha_2\beta_3(\mathbf{e}_2 \wedge \mathbf{e}_3) + \\
&\quad\quad\ \alpha_3\beta_1(\mathbf{e}_3 \wedge \mathbf{e}_1) + \alpha_3\beta_2(\mathbf{e}_3 \wedge \mathbf{e}_2) + \alpha_3\beta_3(\mathbf{e}_3 \wedge \mathbf{e}_3) \\
&\quad = (\alpha_1\beta_2 - \alpha_2\beta_1)\, \mathbf{e}_1 \wedge \mathbf{e}_2 + (\alpha_2\beta_3 - \alpha_3\beta_2)\, \mathbf{e}_2 \wedge \mathbf{e}_3 + (\alpha_3\beta_1 - \alpha_1\beta_3)\, \mathbf{e}_3 \wedge \mathbf{e}_1.
\end{aligned}$$

This equations shows that there is a basis for bivectors, consisting of three elements $\mathbf{e}_1 \wedge \mathbf{e}_2$, $\mathbf{e}_2 \wedge \mathbf{e}_3$, $\mathbf{e}_3 \wedge \mathbf{e}_1$. We will call these elements *basis blades*. We define the scalar 1 and the basis vectors to be basis blades, too. So, for the 3-D example, the full basis is:

$$\Big\{ \underbrace{1}_{\textit{scalars}},\ \underbrace{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3}_{\textit{vector space}},\ \underbrace{\mathbf{e}_1 \wedge \mathbf{e}_2,\ \mathbf{e}_2 \wedge \mathbf{e}_3,\ \mathbf{e}_3 \wedge \mathbf{e}_1}_{\textit{bivector space}},\ \underbrace{\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3}_{\textit{trivector space}} \Big\}.$$

In general, the basis for grade k blades in n dimensional space has $\binom{n}{k}$ elements, because we need a basis blade for each *combination* of basis vectors. This leads to a total of 2^n basis blades for the whole space. A simple way to understand this number is the following. There are as many basis blades required as there are combinations of basis vectors. A basis vector is either present or not present in such a combination. This leads to 2^n elements (binary counting). We will return to this idea in Section 3.1.2, where we use it to form for the *bitmap representation of basis blades.*

Since a multivector is a sum of blades, and blades can be written as a sum of basis blades, any multivector can be decomposed on a basis of blades.

2.4.9 Grade Part Selection

It is useful to extract part of a multivector, based on grade. If $\mathbf{A}$ is the sum of homogeneous multivectors, as in

$$\mathbf{A} = \mathbf{A}_0 + \mathbf{A}_1 + \mathbf{A}_2 + \ldots + \mathbf{A}_n,$$

then the notation $\langle \mathbf{A} \rangle_i$ means to *select* or *extract* the grade i part of $\mathbf{A}$:

$$\langle \mathbf{A} \rangle_i = \mathbf{A}_i.$$

2.5 The Metric Products

The outer product can be used to span blades, and these blades can be interpreted as geometric objects. However, without metric products the blades are not very

useful since we cannot measure their relations (except for the relative orientation and weight of blades with the same attitude). Examples of blade properties that one would like to measure are norms of blades and the angle between blades.

An antagonist to the outer product may also be useful. If the outer product acts like an 'addition operator' for subspaces, then we may also need a 'subtraction operator'. This subtraction operator should remove one subspace from another. Like the outer product, the new product should take into account the orientation of the subspaces.

Exactly such a product exists in geometric algebra, and it is called the inner product. Actually, a number of variants of inner products can be defined, which is why we collectively refer to them as *the metric products*. The metric product we introduce in this section and use throughout this thesis is the *left contraction*. Other metric products are the *right contraction*, the *Hestenes inner product* and the *modified Hestenes inner product*. See also Section 3.1.7.

2.5.1 Metric Products of Vectors

Naturally, we want our metric products to be 'backwards compatible' with the regular inner (or dot) product of vectors. For that, we need first need to introduce a metric product for vectors. The mathematically preferred method is to use a *bilinear form* Q, which is a scalar-valued function of vectors. That is equivalent to defining an *inner product* $\mathbf{a} \cdot \mathbf{b}$ between two arbitrary vectors $\mathbf{a}$ and $\mathbf{b}$. Algebraically, it returns a scalar from two vectors, so it is a mapping $\mathsf{Q} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. It is also linear and symmetric. It defines a *metric* on the vector space $\mathbb{R}^n$. [2] Through Q, the inner product of vectors is defined as

$$\mathbf{a} \cdot \mathbf{b} = \mathsf{Q}[\mathbf{a}, \mathbf{b}].$$

2.5.2 Metric Matrix

In practical applications, Q is often implemented using a metric matrix M. M is a simple lookup table that defines the inner product for each combination of basis vectors. A trivial example of an M for an orthogonal 3-D Euclidean basis $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$ is:

	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_3$
$\mathbf{e}_1$	1	0	0
$\mathbf{e}_2$	0	1	0
$\mathbf{e}_3$	0	0	1

where M is of course the 3×3 matrix that contains the numbers. We define each entry of M as:

$$M_{[i,j]} = \mathsf{Q}[\mathbf{e}_i, \mathbf{e}_j].$$

[2] We might be tempted to call the combination of vector space and metric a 'metric space', but this term is reserved for cases where the inner product is positive definite.

Note that M must be symmetric, because Q is. Using M we can implement Q using matrix multiplications as:

$$\mathsf{Q}[\mathbf{a}, \mathbf{b}] = a^T\, M\, b,$$

where a and b are column matrices containing the coordinates of vectors **a** and **b** relative to the basis. If e_i is a $n \times 1$ column matrix whose i^{th} entry is 1 and all other entries 0, then applying the last equation to basis vectors shows why this works:

$$\mathsf{Q}[\mathbf{e}_i, \mathbf{e}_j] = e_i^T\, M\, e_j = e_i^T\, M_{[:,j]} = M_{[i,j]}.$$

So M is not just a specification of the metric, but can also be directly used to compute metric products of vectors.

Diagonalization of the Metric Matrix

An orthogonal basis leads to a diagonal metric matrix. This is most efficient for most applications. However, one need not choose the basis orthogonal per se. When we discuss the conformal model in Section 2.11, we will find that for Euclidean geometry it is more natural to use a non-diagonal metric matrix than it is to use a diagonal one. Moreover, the non-diagonal matrix can lead to better performance in certain applications (see the benchmark in Section 7.3.2).

Because the metric matrix is by definition symmetric, it is always possible to switch to a basis with a diagonal metric matrix. To understand why, recall the fact from linear algebra that any real matrix M can be written as the singular value decomposition $M = U \Sigma V^T$, with Σ diagonal and U and V orthogonal (see e.g., [23]). When M is symmetric , this implies that $U = V$. Thus the singular values are the eigenvalues of M, and the eigenvectors are the columns of U.

Characterization of Metric

Any metric matrix can be made diagonal, and by properly scaling the basis vectors, the diagonal of the matrix will have $+1$ and/or -1 entries only. The number of positive entries and the number of negative entries on the diagonal fully characterize the metric. We denote this by writing $\mathbb{R}^{p,q}$ for a space with p 'positive dimensions' and q 'negative dimensions'. For example, for the conformal model of 3-D Euclidean space, we use $\mathbb{R}^{4,1}$. We write $\mathbb{R}^n$ for a space with n positive and no negative dimensions.

In principle, we could also introduce null dimensions, that is dimensions with an $\mathbf{e}_i$ for which $\mathsf{Q}[\mathbf{e}_i, \mathbf{e}_i] = 0$. However, this is inconvenient in geometric algebra because of non-invertibility of the pseudoscalar of such spaces, and not useful for our applications. If required, null vectors can still be constructed in any space that has both positive and negative dimensions; see for example conformal points in Section 2.11.

2.5.3 Generalizing the Inner Product to Work on Blades

In linear algebra, the inner (or dot) product is defined for vectors only. How can the inner product be generalized to work on arbitrary blades? For example, how does one compute $\mathbf{a} \cdot (\mathbf{b} \wedge \mathbf{c})$, and what does it mean? One possible way to generalize the inner product is to compute the inner product of $\mathbf{a}$ with each of the factors $\mathbf{b}$, $\mathbf{c}$ of the 2-blade:

$$\mathbf{a} \cdot (\mathbf{b} \wedge \mathbf{c}) = (\mathbf{a} \cdot \mathbf{b})\, \mathbf{c} + (\mathbf{a} \cdot \mathbf{c})\, \mathbf{b}.$$

This works, but loses orientation (i.e., $\mathbf{a} \cdot (\mathbf{b} \wedge \mathbf{c}) = (\mathbf{b} \wedge \mathbf{c}) \cdot \mathbf{a}$). We would prefer to keep orientation. This is done by toggling the sign of each term as $\mathbf{a}$ 'passes through them':

$$\mathbf{a} \cdot (\mathbf{b} \wedge \mathbf{c}) = (\mathbf{a} \cdot \mathbf{b})\, \mathbf{c} - (\mathbf{a} \cdot \mathbf{c})\, \mathbf{b}. \tag{2.1}$$

This idea leads to the definition of the left contraction.

2.5.4 Definition of the Left Contraction

The *left contraction* $\rfloor$ is a product producing a $(k-l)$-blade from a k-blade and an l-blade. So it is a mapping:

$$\text{left contraction} \quad \rfloor : \bigwedge^{k} \mathbb{R}^n \times \bigwedge^{l} \mathbb{R}^n \to \bigwedge^{k-l} \mathbb{R}^n.$$

The defining properties of the left contraction are:

$$\begin{aligned}
\alpha \rfloor \mathbf{B} &= \alpha\, \mathbf{B} && (2.2)\\
\mathbf{B} \rfloor \alpha &= 0 \quad \text{if grade}(\mathbf{B}) > 0 && (2.3)\\
\mathbf{a} \rfloor \mathbf{b} &= \mathsf{Q}[\mathbf{a}, \mathbf{b}] && (2.4)\\
\mathbf{a} \rfloor (\mathbf{B} \wedge \mathbf{C}) &= (\mathbf{a} \rfloor \mathbf{B}) \wedge \mathbf{C} + (-1)^{\text{grade}(\mathbf{B})} \mathbf{B} \wedge (\mathbf{a} \rfloor \mathbf{C}) && (2.5)\\
(\mathbf{A} \wedge \mathbf{B}) \rfloor \mathbf{C} &= \mathbf{A} \rfloor (\mathbf{B} \rfloor \mathbf{C}) && (2.6)
\end{aligned}$$

where α is a scalar, $\mathbf{a}$ and $\mathbf{b}$ are vectors, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are blades (which could be scalars or vectors as well as higher dimensional blades).

Some explanations of this definition may be in order. Equation 2.2 is trivial. The consequence of Equation 2.3 (combined with equations 2.5 and 2.6) is that the left contraction is always 0 when the RHS operand is of lower grade than the LHS operand. Equation 2.4 just reduces the left contraction of two vectors to a scalar value under the bilinear form Q, and makes the left contraction a metric product. Equation 2.5 formalizes the idea from Equation 2.1. The final equation states that when you serially compute the left contraction of two blades with a third (i.e., $\mathbf{A} \rfloor (\mathbf{B} \rfloor \mathbf{C})$), you might as well 'add' the two blades and then perform the left contraction in a single parallel operation (i.e., $(\mathbf{A} \wedge \mathbf{B}) \rfloor \mathbf{C}$). Or — the other way around — that you may execute a contraction serially by splitting off one factor at a time.

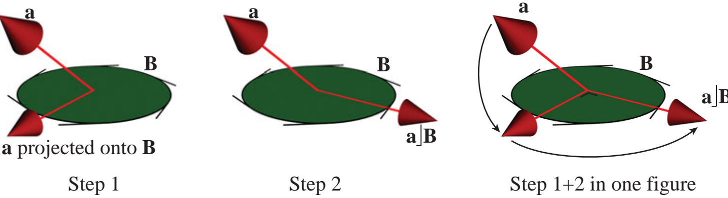


Figure 2.4: *Computing the left contraction of a vector and a 2-blade. The first two figures illustrate step 1 and step 2 as described in the main text. The rightmost figure illustrates both steps in a single figure, and shows the orthogonal angle between the projection and the left contraction.*

Containment

As should be clear from Equation 2.5 and Figure 2.4 the outcome of a left contraction of blades $\mathbf{A} \rfloor \mathbf{B}$ is always contained in $\mathbf{B}$.

2.5.5 Intuitive interpretation of the Left Contraction

Geometric Intuition

To obtain some geometric intuition about how a left contraction $\mathbf{a} \rfloor \mathbf{B}$ works, see Figure 2.4. The left contraction can be thought of as a two-step process.

1. $\mathbf{a}$ is orthogonally projected onto $\mathbf{B}$.

2. The oriented orthogonal complement with respect to $\mathbf{B}$ of the projection is computed.

In principle, this also works the for non-Euclidean metrics, although what is 'orthogonal' does not correspond to our Euclidean intuition anymore.

Computational Intuition

Computational intuition can be established through equations 2.5 and 2.6, which in essence provide the rules to recursively evaluate the left contraction. Take for example,

$$(\mathbf{a} \wedge \mathbf{b}) \rfloor (\mathbf{c} \wedge \mathbf{d}).$$

This evaluates to:

$$\begin{aligned}
(\mathbf{a} \wedge \mathbf{b}) \rfloor (\mathbf{c} \wedge \mathbf{d}) &= \mathbf{a} \rfloor (\mathbf{b} \rfloor (\mathbf{c} \wedge \mathbf{d})) \\
&= \mathbf{a} \rfloor ((\mathbf{b} \cdot \mathbf{c})\, \mathbf{d} - (\mathbf{b} \cdot \mathbf{d})\, \mathbf{c}) \\
&= (\mathbf{a} \cdot \mathbf{d})\, (\mathbf{b} \cdot \mathbf{c}) - (\mathbf{b} \cdot \mathbf{d})\, (\mathbf{a} \cdot \mathbf{c}).
\end{aligned}$$

What is happening is that each factor of $\mathbf{a} \wedge \mathbf{b}$ 'samples' each factor of $\mathbf{c} \wedge \mathbf{d}$. The extra signs are just due to re-ordering of factors, caused by the anti-symmetry of the outer product.

This method of evaluating the left contraction is not necessarily the best, computationally speaking. See Section 3.1.7 for an additive implementation and Section 5.2.9 for a multiplicative implementation.

2.5.6 Blades and Metric

The property "being a blade" depends only on factorizability under the outer product, and hence is independent of metric. As a consequence, we are allowed to use any metric when we are trying to solve problems related to bladedness. Examples of these problems are "is this a blade?" and "find a factorization of this blade".

As an example of why this matters, suppose we have a blade that we want to factorize. One approach is to project random probe vectors onto the blade (the full algorithm based on this idea is given in Section 3.4.6). Intuition suggests that any vector that has been projected onto a blade is a factor of that blade. Unless of course the result is $\mathbf{0}$ — then the vector was orthogonal to the blade. One of the problems of this idea is that the blade may have null factors, which you can never find using this projection method because projecting onto a null factor is impossible.

We can solve this problem (and many similar ones) by temporarily *LIFT*ing the blade to another algebra with a Euclidean metric. This avoids the possibility of null vectors. As long as the problem we are try to solve does not depend on metric, using a different metric is allowed. The so-called LIFT (Linear Injective Function Transformation) idea is defined more formally in [6]. We will use it in Sections 3.4.5 and 3.4.6.

2.6 The Geometric Product

In this section we introduce the *invertible geometric product* (or *Clifford product*, after its inventor). The geometric product is the fundamental product of geometric algebra from which the other linear products can be derived. We start by demonstrating the non-invertibility of the products discussed so far, which leads to the insight that a combination of these non-invertible products *is* invertible. We use this to define of the geometric product, after which we study the *versors* generated by the geometric product.

2.6.1 An Invertible Product of Vectors?

We can 'discover' the geometric product by studying the non-invertibility of the subspace products.

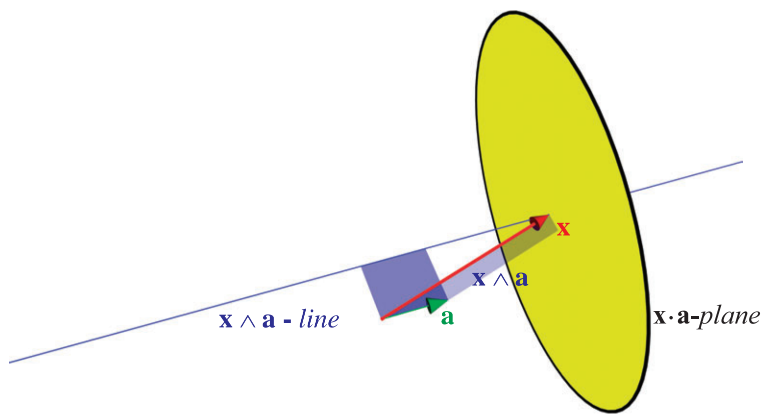


Figure 2.5: *Neither the metric products nor the outer product of vector is invertible. However, their combination* is *invertible. See text.*

The Non-Invertibility of the Outer product

The definition of the outer product implies that information is lost when computing an outer product of two k-blades with $k \geq 1$. This loss of information makes the outer product non-invertible. We already discussed this to some length in Section 2.4. Here we expose the geometry involved in this loss of information.

Suppose we are given a 2-blade $\mathbf{x} \wedge \mathbf{a}$ and the original vector $\mathbf{a}$, and are asked to retrieve the value of $\mathbf{x}$. This is impossible in general, as illustrated for the 3-D case in Figure 2.5. Because the outer product rejects the part of $\mathbf{x}$ that is parallel to $\mathbf{a}$, we get

$$\mathbf{x} \wedge \mathbf{a} = (\mathbf{x} + \alpha\, \mathbf{a}) \wedge \mathbf{a},$$

for any value of α. Adding $\alpha\, \mathbf{a}$ to $\mathbf{x}$ just reshapes the parallelogram spanned by $\mathbf{a}$ and $\mathbf{x}$. All we can say about $\mathbf{x}$ in this problem is that it is some vector $\mathbf{x}' = \mathbf{x} + \alpha\, \mathbf{a}$, i.e., $\mathbf{x}$ lies on some line, labeled as the $\mathbf{x} \wedge \mathbf{a}$-line in Figure 2.5.

The Non-Invertibility of the Metric Products

Because metric products reduce two vectors to a single scalar, information is clearly lost in the process. This is again illustrated in 3-D in Figure 2.5. The problem this time is to retrieve $\mathbf{x}$ given $\mathbf{x} \cdot \mathbf{a}$ and $\mathbf{a}$. Suppose we have a set of vectors $\mathbf{a}_i^\perp$ that span the $n-1$ dimensional space orthogonal to $\mathbf{a}$. So $\mathsf{Q}[\mathbf{a}, \mathbf{a}_i^\perp] = 0$ for all i from 1 to $n-1$. Then

$$\mathbf{x} \cdot \mathbf{a} = (\mathbf{x} + \sum_{i=1}^{n-1} \alpha_i\, \mathbf{a}_i^\perp) \cdot \mathbf{a},$$

for any choice of the $\alpha_i \in \mathbb{R}$. In other words, there is an (hyper-)plane perpendicular to $\mathbf{a}$ in which the reconstructed $\mathbf{x}$ could lie. Hence we cannot recover $\mathbf{x}$ given $\mathbf{x} \cdot \mathbf{a}$ and $\mathbf{a}$.

Combining the Outer Product and Inner Product to Form an Invertible Product

Figure 2.5 also reveals that the combination of outer product and the inner product *does* contain enough information to be invertible, since the line (solutions for $\mathbf{x}$ through the outer product) and the plane (solutions for $\mathbf{x}$ through the inner product) intersect in a unique single point, which must be $\mathbf{x}$. A straightforward way of retaining this information is defining a new product that is simply the sum of the outer and inner product:

$$\mathbf{x}\,\mathbf{a} = \mathbf{x} \cdot \mathbf{a} + \mathbf{x} \wedge \mathbf{a}. \tag{2.7}$$

This product is called the geometric product (the full definition is given below). We use the space symbol to denote it because it is so fundamental to geometric algebra. If we temporarily use the following definition for inversion of a vector

$$\mathbf{a}^{-1} = \frac{\mathbf{a}}{\mathbf{a} \cdot \mathbf{a}}, \tag{2.8}$$

we can already begin to see the invertibility:

$$\begin{aligned}
(\mathbf{x}\,\mathbf{a})\,\mathbf{a}^{-1} &= (\mathbf{x} \cdot \mathbf{a})\,\mathbf{a}^{-1} + (\mathbf{x} \wedge \mathbf{a}) \cdot \mathbf{a}^{-1} + (\mathbf{x} \wedge \mathbf{a}) \wedge \mathbf{a}^{-1} \\
&= (\mathbf{x} \cdot \mathbf{a})\,\mathbf{a}^{-1} + (\mathbf{x} \wedge \mathbf{a}) \cdot \mathbf{a}^{-1} \\
&= (\mathbf{x} \cdot \mathbf{a})\,\mathbf{a}^{-1} + \mathbf{x}\,(\mathbf{a} \cdot \mathbf{a}^{-1}) - (\mathbf{x} \cdot \mathbf{a}^{-1})\,\mathbf{a} \\
&= \mathbf{x}\,(\mathbf{a} \cdot \mathbf{a}^{-1}) \\
&= \mathbf{x}.
\end{aligned}$$

Note that in this derivation, we used some properties of the geometric product that will be defined shortly.

2.6.2 Definition of the Geometric Product

The above paragraph provided some intuition and motivation on why one would like to have the geometric product, and a 'definition' of the geometric product of vectors, in the form of Equation 2.7. We now give the full definition of the geometric product. The definition is rather abstract. You may think of the motivation behind it as "a product of oriented subspaces that is as lossless as possible, while still being bi-linear, distributive and associative". In other words, the most lossless product of oriented subspaces that one can get away with to form a reasonable algebra.

The geometric product of blades is a function

$$\text{geometric product:} \quad : \bigwedge^{k} \mathbb{R}^n \times \bigwedge^{l} \mathbb{R}^n \to \bigwedge \mathbb{R}^n$$

($\bigwedge \mathbb{R}^n$ was defined in Section 2.4.7). The geometric product involving scalars behaves just like regular scalar multiplication in $\mathbb{R}^n$, so from now on we can think of products such as

$$\alpha\,\beta \;\text{ and }\; \alpha\,\mathbf{b}, \qquad \alpha, \beta \in \mathbb{R}, \mathbf{b} \in \mathbb{R}^n$$

as using the geometric product. The other defining properties of the geometric product are:

$$\alpha\,\mathbf{B} = \mathbf{B}\,\alpha \tag{2.9}$$

$$\mathbf{a}\,\mathbf{a} = \mathsf{Q}[\mathbf{a}, \mathbf{a}] \tag{2.10}$$

$$\mathbf{A}\,(\mathbf{B} + \mathbf{C}) = \mathbf{A}\,\mathbf{B} + \mathbf{A}\,\mathbf{C} \tag{2.11}$$

$$(\mathbf{A} + \mathbf{B})\,\mathbf{C} = \mathbf{A}\,\mathbf{B} + \mathbf{B}\,\mathbf{C} \tag{2.12}$$

$$\mathbf{A}\,(\mathbf{B}\,\mathbf{C}) = (\mathbf{A}\,\mathbf{B})\,\mathbf{C} \tag{2.13}$$

$$\exists\ \mathbf{A}, \mathbf{B} \in \bigwedge \mathbb{R}^n \ : \ \mathbf{A}\,\mathbf{B} \neq \mathbf{B}\,\mathbf{A} \tag{2.14}$$

Equation 2.9 says that scalars always commute with multivectors. Equation 2.10 defines the contraction rule for vectors. Equations 2.11 and 2.12 define that the geometric product is distributive and linear. Equation 2.13 defines the geometric product to be associative. Finally, Equation 2.14 states that the geometric product is not commutative in general.

Note how minimal the definition of the geometric product is. The only rule that really 'computes' anything is the contraction (Equation 2.10). The strangest rule is non-commutativity Equation 2.14. The others rules are just reasonable properties (distributivity, associativity, scalars commute). Even through the definition of the geometric product is simple, proving that it actually is a sensible product is non-trivial. The interested reader may refer to [37].

2.6.3 Intuitive Interpretation of the Geometric Product

We will not attempt to provide intuition on why the geometric product is geometrically significant in a single section. Instead, we point forward to several sections where various aspects of the geometric product become clear.

- The subspace products can be derived from the geometric product (Section 2.7).
- Versors (geometric products of vectors) represent orthogonal transformations, and these can be applied to blades and versors using the geometric product (Section 2.9.1).

- Many of these transformations can be written as exponentials, which allows for easy interpolation, because the log space is linear (Section 2.9.3).

2.6.4 Computing with the Geometric Product

The definition of the geometric product is rather abstract and does not have an algorithmic feeling to it. It does not readily give an algorithm that will evaluate or simplify geometric products. However, the following example shows that it is possible to compute directly from the definition of the geometric product.

We will first need some results that split the geometric product of vectors into an anti-symmetric part and a symmetric part, and identify the symmetric part as the bilinear form. First we note that

$$\begin{aligned} \mathbf{a}\,\mathbf{x} &= \mathbf{a}\,\mathbf{x} + \tfrac{1}{2}\mathbf{x}\,\mathbf{a} - \tfrac{1}{2}\mathbf{x}\,\mathbf{a} \\ &= \tfrac{1}{2}(\mathbf{a}\,\mathbf{x} + \mathbf{x}\,\mathbf{a}) + \tfrac{1}{2}(\mathbf{a}\,\mathbf{x} - \mathbf{x}\,\mathbf{a}), \end{aligned}$$

where $\frac{1}{2}(\mathbf{a}\,\mathbf{x} - \mathbf{x}\,\mathbf{a})$ can be non-zero. This just shows that any geometric product has a *symmetric* part, which must be $\frac{1}{2}(\mathbf{a}\,\mathbf{x} + \mathbf{x}\,\mathbf{a})$ and an *anti-symmetric* part, which must be $\frac{1}{2}(\mathbf{a}\,\mathbf{x} - \mathbf{x}\,\mathbf{a})$. The symmetric part is the bilinear form Q:

$$\begin{aligned} \tfrac{1}{2}(\mathbf{a}\,\mathbf{x} + \mathbf{x}\,\mathbf{a}) &= \tfrac{1}{2}\big((\mathbf{a}+\mathbf{x})\,(\mathbf{a}+\mathbf{x}) - \mathbf{a}\,\mathbf{a} - \mathbf{x}\,\mathbf{x}\big) \\ &= \tfrac{1}{2}\big(\mathsf{Q}[\mathbf{a}+\mathbf{x},\mathbf{a}+\mathbf{x}] - \mathsf{Q}[\mathbf{a},\mathbf{a}] - \mathsf{Q}[\mathbf{x},\mathbf{x}]\big) \qquad (2.15) \\ &= \tfrac{1}{2}\big(\mathsf{Q}[\mathbf{a},\mathbf{a}] + \mathsf{Q}[\mathbf{a},\mathbf{x}] + \mathsf{Q}[\mathbf{x},\mathbf{a}] + \mathsf{Q}[\mathbf{x},\mathbf{x}] - \mathsf{Q}[\mathbf{a},\mathbf{a}] - \mathsf{Q}[\mathbf{x},\mathbf{x}]\big) \\ &= \mathsf{Q}[\mathbf{a},\mathbf{x}]. \end{aligned}$$

Once we have these rewrite rules, we are able to evaluate expressions to the point where it is clear that a we can write a computer implementation. As an example we try to evaluate the expression $\mathbf{a}\,\mathbf{x}\,\mathbf{a}$:

$$\begin{aligned} \mathbf{a}\,\mathbf{x}\,\mathbf{a} &= \mathsf{Q}[\mathbf{a},\mathbf{x}]\,\mathbf{a} + \tfrac{1}{2}\,(\mathbf{ax} - \mathbf{xa})\,\mathbf{a} \\ &= \mathsf{Q}[\mathbf{a},\mathbf{x}]\,\mathbf{a} + \tfrac{1}{2}\,\mathbf{a}\,\mathbf{x}\,\mathbf{a} - \tfrac{1}{2}\,\mathbf{x}\,\mathsf{Q}[\mathbf{a},\mathbf{a}]. \end{aligned}$$

By transferring $\frac{1}{2}\,\mathbf{a}\,\mathbf{x}\,\mathbf{a}$ from the RHS to the LHS we get:

$$\tfrac{1}{2}\,\mathbf{a}\,\mathbf{x}\,\mathbf{a} = \mathsf{Q}[\mathbf{a},\mathbf{x}]\,\mathbf{a} - \tfrac{1}{2}\,\mathbf{x}\,\mathsf{Q}[\mathbf{a},\mathbf{a}],$$

or

$$\mathbf{a}\,\mathbf{x}\,\mathbf{a} = 2\,\mathsf{Q}[\mathbf{a},\mathbf{x}]\,\mathbf{a} - \mathsf{Q}[\mathbf{a},\mathbf{a}]\,\mathbf{x} = 2\,(\mathbf{a}\cdot\mathbf{x})\,\mathbf{a} - (\mathbf{a}\cdot\mathbf{a})\,\mathbf{x}. \qquad (2.16)$$

The result only uses dot products and scalar multiplication, which are easily implemented. As a side note, we mention that if $\mathbf{a}$ is unit length, then $\mathbf{a}\,\mathbf{x}\,\mathbf{a}$ is the classical expression that reflects $\mathbf{x}$ in the line with direction $\mathbf{a}$, see Section 2.9.1.

2.6.5 Versors

A k-versor is an invertible multivector that is factorizable under the geometric product. That is, it can be written as the geometric product of k invertible vectors

$$\mathbf{V} = \mathbf{v}_1\,\mathbf{v}_2\,\ldots\,\mathbf{v}_k.$$

Invertibility simply means that none of the $\mathbf{v}_i$ has $\mathsf{Q}[\mathbf{v}_i, \mathbf{v}_i] = 0$. It is useful to define the following two terms:

- *even versor*: a versor with an even number of factors.
- *odd versor*: a versor with an odd number of factors.

2.6.6 Reversion and Inversion

Versors can be reversed, just like blades:

$$\widetilde{\mathbf{V}} = (\mathbf{v}_1\,\mathbf{v}_2\,\ldots\,\mathbf{v}_k)^{\sim} = \mathbf{v}_k\,\ldots\,\mathbf{v}_2\,\mathbf{v}_1.$$

The inverse $\mathbf{V}^{-1}$ of a multivector $\mathbf{V}$ is defined through:

$$\mathbf{V}\,\mathbf{V}^{-1} = \mathbf{V}^{-1}\,\mathbf{V} = 1. \tag{2.17}$$

Note that geometric algebra is not a division algebra (in a division algebra, all elements except 0 have unique inverses). All versors have inverses, by definition. But null blades are not invertible and neither are multivectors in general.

One of the nice properties of versors is that computing their inverse is simple and computationally efficient. First note that

$$\widetilde{\mathbf{V}}\,\mathbf{V} = \mathbf{V}\,\widetilde{\mathbf{V}} = (\mathbf{v}_1\,\mathbf{v}_2\,\ldots\,\mathbf{v}_k)\,(\mathbf{v}_k\,\ldots\,\mathbf{v}_2\,\mathbf{v}_1) = \mathsf{Q}[\mathbf{v}_k, \mathbf{v}_k]\,\ldots\,\mathsf{Q}[\mathbf{v}_2, \mathbf{v}_2]\,\mathsf{Q}[\mathbf{v}_1, \mathbf{v}_1].$$

It is clear that the right hand side of this equation is a non-zero scalar value. In other words, the reverse of a versor is equal to its inverse, up to a scalar factor. Hence the inverse of a versor is easily computed as:

$$\mathbf{V}^{-1} = \frac{\widetilde{\mathbf{V}}}{\mathbf{V}\,\widetilde{\mathbf{V}}}. \tag{2.18}$$

We can verify the correctness of this expression by substituting it into Equation 2.17:

$$\mathbf{V}\,\mathbf{V}^{-1} = \mathbf{V}\,\frac{\widetilde{\mathbf{V}}}{\mathbf{V}\,\widetilde{\mathbf{V}}} = \frac{\widetilde{\mathbf{V}}}{\mathbf{V}\,\widetilde{\mathbf{V}}}\,\mathbf{V} = 1.$$

Our preliminary definition of the inverse of a vector (Equation 2.8) was just a special case of this.

In general, Equation 2.18 only works for versors (and hence for invertible blades, too), but not for arbitrary multivectors. We believe that proper use of geometric algebra should not lead to the need for inverting non-versors. Nonetheless, Section 3.4.2 gives an algorithm for inverting *any* invertible multivector.

2.6.7 Norms

Various kinds of multivector norms are useful in geometric algebra. In this thesis we use two types of norms (and their squares). Mathematically speaking, a norm is any function

$$\text{norm:} \quad \|\cdot\| : \bigwedge \mathbb{R}^n \to \mathbb{R}$$

that assigns a non-negative real-valued 'length' to a multivector. It should satisfy:

$$\begin{array}{ll} (1) & \|\mathbf{X}\| \geq 0, \text{ and } \|\mathbf{X}\| = 0 \text{ only if } \mathbf{X} = 0, \\ (2) & \|\mathbf{X} + \mathbf{Y}\| \leq \|\mathbf{X}\| + \|\mathbf{Y}\|, \\ (3) & \|\alpha\, \mathbf{X}\| = |\alpha|\, \|\mathbf{X}\|. \end{array}$$

where $|\alpha|$ is the absolute value of scalar α.

Reverse Norm (Squared)

We define the *squared* reverse norm of an arbitrary multivector $\mathbf{X}$ as

$$\|\mathbf{X}\|_R^2 = \langle \mathbf{X}\, \widetilde{\mathbf{X}} \rangle_0.$$

Here $\langle \mathbf{X}\, \widetilde{\mathbf{X}} \rangle_0$ means to extract the scalar part of $\mathbf{X}\, \widetilde{\mathbf{X}}$, as defined in Section 2.4.9. The reverse norm is defined as

$$\|\mathbf{X}\|_R = \text{sign}(\|\mathbf{X}\|_R^2)\, \sqrt{|\|\mathbf{X}\|_R^2|}.$$

When a non-positive-definite metric is used, the reverse norm is not a norm in the strict mathematical sense as defined above, since $\|\mathbf{X}\|_R$ may have a negative value. However, in practice the reverse norm is useful, *especially* due to its possible negative sign. E.g., in the conformal model the sign of the reverse norm squared of a sphere indicates whether the sphere is real or imaginary. Hence we will (ab-)use the term "norm" for it throughout this thesis.

Euclidean Norm

The Euclidean norm is just the regular 2-norm over the 2^n dimensional linear space of blades. It follows from the above that it is simply:

$$\|\mathbf{X}\|_E = \sqrt{\langle \mathbf{X}\, \widetilde{\mathbf{X}} \rangle_0},$$

where the geometric product $\mathbf{X}\, \widetilde{\mathbf{X}}$ must be computed using a Euclidean metric. We also use the squared Euclidean norm, which is just:

$$\|\mathbf{X}\|_E^2 = \langle \mathbf{X}\, \widetilde{\mathbf{X}} \rangle_0,$$

again with the geometric product evaluated using a Euclidean metric.

Infinity Norm

In some algorithms (see for example Section 3.4.6), we will need the absolute largest coordinate of a multivector with respect to the basis. This is similar to the ∞-norm used in linear algebra. But because in those instances where we use the infinity norm we also need the basis blade to which the coordinate refers, there is not much use in defining this norm formally; its use in this thesis is somewhat ad hoc.

Units

Each norm induces a definition of a type of 'unit', i.e., a multivector normalized with respect to that norm. The interpretation of what this means for the Euclidean norm is straightforward, and all multivectors can be normalized using that metric. The meaning of 'being unit' with respect to the reverse norm is more subtle, as it depends on the metric. For example, points are represented by null vectors in the conformal model, so they can never be unit with respect to the reverse norm.

2.7 From Geometric Product to Subspace Products

We stated that the geometric product is the fundamental product of geometric algebra. In Appendix A we prove this by showing how the subspace products of blades can be derived from the geometric product of blades, through the following grade part selection rules:

$$\mathbf{A}_k \wedge \mathbf{B}_l = \langle \mathbf{A}_k \, \mathbf{B}_l \rangle_{l+k} \tag{2.19}$$

$$\mathbf{A}_k \rfloor \mathbf{B}_l = \langle \mathbf{A}_k \, \mathbf{B}_l \rangle_{l-k} \tag{2.20}$$

$$\mathbf{A}_k \lfloor \mathbf{B}_l = \langle \mathbf{A}_k \, \mathbf{B}_l \rangle_{k-l} \tag{2.21}$$

$$\mathbf{A}_k * \mathbf{B}_l = \langle \mathbf{A}_k \, \mathbf{B}_l \rangle_{0} \tag{2.22}$$

The last two equations define the right contraction and the scalar product, respectively.

It is possible to prove that these rules have the same properties as original product definitions, although this is non-trivial. The proofs are in Appendix A, where we show is that selected grade parts of geometric product have the properties required by the product definitions of Sections 2.4.1 and 2.5.4 and hence are isomorphic to those products (which means that they are the same, as far as mathematicians care). So from here on, we will use the grade part selection rules listed above as *the* definitions of the subspace products.

The rules formulated above apply only to products of blades, but we can easily extend them to arbitrary multivectors. Every multivector $\mathbf{A}$ can be decomposed

to a sum of blades $\mathbf{A} = \sum_{i=1} \mathbf{A}^i$ (we use superscript indices to avoid confusion with grade part selection). As an example, for the outer product of multivectors $\mathbf{A}$ and $\mathbf{B}$, we get:

$$\mathbf{A} \wedge \mathbf{B} = \left(\sum_{i=1} \mathbf{A}^i \right) \wedge \left(\sum_{j=1} \mathbf{B}^j \right) = \sum_{i=1} \sum_{j=1} \left(\mathbf{A}^i \wedge \mathbf{B}^j \right).$$

The outer product on the right hand side can readily be evaluated using Equation 2.19.

2.7.1 Relationships Between the Products

This section contains some interesting facts that can be discussed now that the relationships between the three main products is clear.

The Weight of an Outer Product

Suppose we have two unit length vectors $\mathbf{a}$ and $\mathbf{b}$, not parallel to each other. Then we can compute the unit 2-blade

$$\mathbf{I}_2 = \frac{\mathbf{a} \wedge \mathbf{b}}{\|\mathbf{a} \wedge \mathbf{b}\|_R}.$$

What is the relation in magnitude between $\mathbf{I}_2$ and $\mathbf{a} \wedge \mathbf{b}$, in other words, what is $\|\mathbf{a} \wedge \mathbf{b}\|_R$? First we need to find the value of $\mathbf{a} \cdot \mathbf{b}$, which is (by definition of the cosine)

$$\mathbf{a} \cdot \mathbf{b} = \cos\phi,$$

where ϕ is the angle between $\mathbf{a}$ and $\mathbf{b}$. We can then derive:

$$\begin{aligned}
\|\mathbf{a} \wedge \mathbf{b}\|_R^2 &= (\mathbf{a} \wedge \mathbf{b})\,(\mathbf{b} \wedge \mathbf{a}) \\
&= (-\mathbf{a} \cdot \mathbf{b} + \mathbf{a}\,\mathbf{b})\,(\mathbf{b} \wedge \mathbf{a}) \\
&= -(\mathbf{a} \cdot \mathbf{b})\,(\mathbf{b} \wedge \mathbf{a}) + \mathbf{a}\,\mathbf{b}\,(\mathbf{b} \wedge \mathbf{a}) \\
&= -(\mathbf{a} \cdot \mathbf{b})\,(\mathbf{b} \wedge \mathbf{a}) + \mathbf{a}\,(\mathbf{b} \cdot (\mathbf{b} \wedge \mathbf{a}) + \mathbf{b} \wedge (\mathbf{b} \wedge \mathbf{a})) \\
&= -(\mathbf{a} \cdot \mathbf{b})\,(\mathbf{b} \wedge \mathbf{a}) + \mathbf{a}\,((\mathbf{b} \cdot \mathbf{b})\,\mathbf{a} - (\mathbf{b} \cdot \mathbf{a})\,\mathbf{b}) \\
&= -(\mathbf{a} \cdot \mathbf{b})\,(\mathbf{b} \wedge \mathbf{a}) + ((\mathbf{b} \cdot \mathbf{b})\,(\mathbf{a} \wedge \mathbf{a}) + (\mathbf{b} \cdot \mathbf{b})\,(\mathbf{a} \cdot \mathbf{a}) - \\
&\quad (\mathbf{b} \cdot \mathbf{a})\,(\mathbf{a} \wedge \mathbf{b}) - (\mathbf{b} \cdot \mathbf{a})\,(\mathbf{a} \cdot \mathbf{b})) \\
&= (\mathbf{b} \cdot \mathbf{b})\,(\mathbf{a} \cdot \mathbf{a}) - (\mathbf{b} \cdot \mathbf{a})\,(\mathbf{b} \cdot \mathbf{a}) \\
&= 1 - (\mathbf{b} \cdot \mathbf{a})\,(\mathbf{b} \cdot \mathbf{a}) \\
&= 1 - \cos^2\phi \\
&= \sin^2\phi.
\end{aligned}$$

Hence

$$\mathbf{a} \wedge \mathbf{b} = \sin\phi\, \mathbf{I}_2.$$

This shows that the size of a 2-blade is equal to the surface area of the parallelogram spanned by its two factors.

Factorizing Blades under the Geometric Product

Every k-blade can be written as the geometric product of k vectors. I.e., for every blade $\mathbf{v}_1 \wedge \mathbf{v}_2 \wedge \ldots \wedge \mathbf{v}_k$ we can find vectors $\mathbf{w}_i$ such that

$$\mathbf{v}_1 \wedge \mathbf{v}_2 \wedge \ldots \wedge \mathbf{v}_k = \mathbf{w}_1\, \mathbf{w}_2\, \ldots\, \mathbf{w}_k.$$

Note that this does not imply that a k-blade is always a k-versor: some of the $\mathbf{w}_i$ may be non-invertible.

To prove this, note that the blade spans its own vector space, with its own metric. For every metric, an orthogonal basis can be found (see Section 2.5.2 and Section 5.3.2). The grade k geometric product of the k vectors of this basis is equal up to scale to the k blade, by definition of the outer product. The other grade parts of the geometric product must be zero, since all factors are orthogonal. This also implies that invertible blades are versors, as depicted earlier in Figure 2.3.

When can a Left Contraction be Replaced by a Geometric Product?

The left contraction $\mathbf{A}_k \rfloor \mathbf{B}_l$ is equal to the geometric product $\mathbf{A}_k\, \mathbf{B}_l$ when blade $\mathbf{A}_k$ is contained in blade $\mathbf{B}_l$,

$$\mathbf{A}_k \rfloor \mathbf{B}_l = \mathbf{A}_k\, \mathbf{B}_l \quad \text{if} \quad \mathbf{A}_k \subseteq \mathbf{B}_l. \tag{2.23}$$

Proof. We prove this recursively. One at a time, we split off a factor $\mathbf{a}_k$ and rewrite it to form a geometric product with $\mathbf{B}_l$. Because of containment, the outer product $\mathbf{a}_k \wedge \mathbf{B}_l$ is always 0.

$$\begin{aligned}
\mathbf{A}_k \rfloor \mathbf{B}_l &= (\mathbf{A}_{k-1} \wedge \mathbf{a}_k) \rfloor \mathbf{B}_l \\
&= \mathbf{A}_{k-1} \rfloor (\mathbf{a}_k \rfloor \mathbf{B}_l) \\
&= \mathbf{A}_{k-1} \rfloor (\mathbf{a}_k\, \mathbf{B}_l - \mathbf{a}_k \wedge \mathbf{B}_l) \\
&= \mathbf{A}_{k-1} \rfloor (\mathbf{a}_k\, \mathbf{B}_l) \\
&= \ldots \\
&= \mathbf{A}_k\, \mathbf{B}_l
\end{aligned}$$

□

2.8 Basic Constructions in Geometric Algebra

Now that we have defined the basic products and operations of geometric algebra, more interesting geometric constructions can be built upon that foundation. In this section, we treat constructions like projection and orthogonalization, still

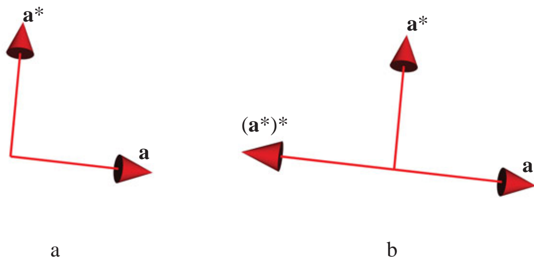


Figure 2.6: *The dual in 2-D space. The left figure shows a vector* $\mathbf{a}$ *and its dual* $\mathbf{a}^\star$*. The right figure also shows the dual applied twice, which is not the same as the original* $\mathbf{a}$ *in all spaces.*

using the blades-represent-subspaces interpretation. More interesting interpretations of blades and versors will follow shortly in the sections on the homogeneous and conformal models.

Many operations are defined only for blades because of geometric significance, but through linearity and distributivity can in principle work for general multivectors. In most examples and illustrations we assume a Euclidean metric.

2.8.1 Dualization

Dualization of blades is defined as

$$\text{dualization:} \quad \mathbf{A}^\star = \mathbf{A} \rfloor \mathbf{I}_n^{-1}, \tag{2.24}$$

where $\mathbf{I}_n$ is the blade with respect to which one would like to dualize. Often, $\mathbf{I}_n$ is the pseudoscalar of the space.

The subspace interpretation of dualization is taking the orthogonal complement. This is illustrated for vectors in 2-D in Figure 2.6a. The dual of $\mathbf{A}$ is the part of the space $\mathbf{I}_n$ that is not contained in $\mathbf{A}$. Note that $\mathbf{I}_n$ does not have to be the pseudoscalar of the full space, it can be an any blade that fully contains $\mathbf{A}$.

One would expect a double dualization to result in the original blade, as in

$$(\mathbf{A}^\star)^\star = \mathbf{A} \qquad \text{(not true in general!)}$$

However, this is not true in general. The problem is illustrated for the 2-D case in Figure 2.6b. In general, we find that

$$(\mathbf{A}^\star)^\star = (\mathbf{A} \rfloor \mathbf{I}_n^{-1}) \rfloor \mathbf{I}_n^{-1} = \mathbf{A}\, \mathbf{I}_n^{-1}\, \mathbf{I}_n^{-1}$$

and to fix this, *undualization* is defined as

$$\text{undualization:} \quad \mathbf{A}^{-\star} = \mathbf{A} \rfloor \mathbf{I}_n, \tag{2.25}$$

such that

$$(\mathbf{A}^\star)^{-\star} = (\mathbf{A} \rfloor \mathbf{I}_n^{-1}) \rfloor \mathbf{I}_n = \mathbf{A}\, \mathbf{I}_n^{-1}\, \mathbf{I}_n = \mathbf{A}.$$

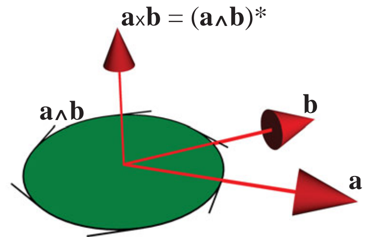


Figure 2.7: *The 3-D-only cross product.*

Duality Relationships

Two important duality relationships that we will use are

$$\mathbf{A}\rfloor(\mathbf{B}^\star) = \mathbf{A}\rfloor(\mathbf{B}\rfloor\mathbf{I}) = (\mathbf{A}\wedge\mathbf{B})\rfloor\mathbf{I} = (\mathbf{A}\wedge\mathbf{B})^\star, \tag{2.26}$$

by Equation 2.6. And

$$(\mathbf{A}\rfloor\mathbf{B})^\star = (\mathbf{A}\rfloor\mathbf{B})\rfloor\mathbf{I} = \mathbf{A}\wedge(\mathbf{B}\rfloor\mathbf{I}) = \mathbf{A}\wedge(\mathbf{B}^\star), \tag{2.27}$$

which we do not prove here (see Appendix B of [12]).

2.8.2 The 3-D Cross Product

Using dualization we can retrieve the cross product of vectors from linear algebra. The cross product $\mathbf{a}\times\mathbf{b}$ is a product for vectors that works only in 3-D. It computes a vector at a right angle to both $\mathbf{a}$ and $\mathbf{b}$, and is the dual of the outer product:

$$\text{cross product:} \quad \mathbf{a}\times\mathbf{b} = (\mathbf{a}\wedge\mathbf{b})^\star = (\mathbf{a}\wedge\mathbf{b})\rfloor\mathbf{I}_3^{-1}. \tag{2.28}$$

Using the cross product in geometric algebra is entirely superfluous, as we can use the 2-blade $\mathbf{a}\wedge\mathbf{b}$ directly, instead of its dual representation.

2.8.3 Orthogonal Projection

In Section 2.5.5, we stated that a left contraction $\mathbf{A}\rfloor\mathbf{B}$ is the combination of orthogonal projection of $\mathbf{A}$ onto $\mathbf{B}$, followed by taking the orthogonal complement of the projection with respect to $\mathbf{B}$. If we want to obtain just projection, we need to undo the orthogonal complement, as in:

$$\text{orthogonal projection of } \mathbf{A} \text{ onto } \mathbf{B}: \quad (\mathbf{A}\rfloor\mathbf{B}^{-1})\rfloor\mathbf{B} = (\mathbf{A}\rfloor\mathbf{B}^{-1})\,\mathbf{B}. \tag{2.29}$$

See Figure 2.8 for an illustration. Note that we are allowed to rewrite the second contraction to a geometric product because of containment of $(\mathbf{A}\rfloor\mathbf{B})$ in $\mathbf{B}$.

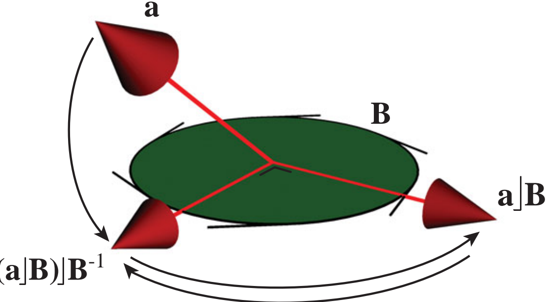


Figure 2.8: *Orthogonal projection, displayed as the un-doing of the orthogonal-complement-step of the left contraction (see also Figure 2.4).*

It is simple to prove that this actually is a projection. The condition for this is that it is idempotent, i.e., applying the projector twice should give the same result:

$$\begin{aligned}
(((\mathbf{A}\rfloor\mathbf{B}^{-1})\,\mathbf{B})\rfloor\mathbf{B}^{-1})\,\mathbf{B} &= (((\mathbf{A}\rfloor\mathbf{B}^{-1})\,\mathbf{B})\,\mathbf{B}^{-1})\,\mathbf{B} \\
&= ((\mathbf{A}\rfloor\mathbf{B}^{-1})\,\mathbf{B})\,\mathbf{B}^{-1}\,\mathbf{B} \\
&= (\mathbf{A}\rfloor\mathbf{B}^{-1})\,\mathbf{B}.
\end{aligned}$$

2.8.4 Intersection of Blades

The intersection of two blades is their common factor. It can be computed as:

$$\text{Intersection of blades } \mathbf{A} \text{ and } \mathbf{B}: \quad \mathbf{A}^{\star}\rfloor\mathbf{B}. \tag{2.30}$$

The intuitive interpretation of this equation is "remove from **B** the part that is not like **A**". Note that the dual must be computed with respect to the lowest-grade blade that fully contains both **A** and **B**. For now we will assume that **A** and **B** span the whole space, so we get

$$\mathbf{A}^{\star}\rfloor\mathbf{B} = (\mathbf{A}\rfloor\mathbf{I}^{-1})\rfloor\mathbf{B},$$

but in cases where this assumption does not hold, the `join` should be used to compute the appropriate **I** (see Section 2.13).

Some favor the following — more symmetric — equivalent of Equation 2.30

$$(\mathbf{A}^{\star}\wedge\mathbf{B}^{\star})^{-\star}.$$

This equation can be derived from Equation 2.30 using the duality relationship Equation 2.26. The intuitive interpretation is "combining everything that is unlike **A** and unlike **B**, and take the dual of that".

Intersection of blades is an operation which is only concerned with blades and their factorization. So it is an area where we can (and sometimes *should*) perform computations using a Euclidean metric, as was discussed in Section 2.5.6.

2.8.5 Reciprocal Frames

Even though geometric algebra is coordinate-free, computer implementations internally use coordinates to describe multivectors. It may be required to compute the coordinates of a vector on a non-orthogonal basis. For this purpose, the definition of reciprocal frames is often used in geometric algebra.

Suppose we have a frame defined by k basis vectors $\mathbf{b}_i$, which are not necessarily orthogonal. We want to compute the coordinates x_i of some vector $\mathbf{x}$ with respect to the frame such that

$$\mathbf{x} = x_1\,\mathbf{b}_1 + x_2\,\mathbf{b}_2 + \ldots + x_k\,\mathbf{b}_k$$

This is easily accomplished by computing the *reciprocal vectors*

$$\mathbf{b}^i = (-1)^{i-1}(\mathbf{b}_1 \wedge \mathbf{b}_2 \wedge \cdots \wedge \breve{\mathbf{b}}_i \wedge \cdots \wedge \mathbf{b}_k) \rfloor \mathbf{I}_k^{-1}. \tag{2.31}$$

The notation $\breve{\mathbf{b}}_i$ means to leave $\mathbf{b}_i$ out of the product. So $\mathbf{e}_1 \wedge \breve{\mathbf{e}}_2 \wedge \mathbf{e}_3 = \mathbf{e}_1 \wedge \mathbf{e}_3$. Note that the reciprocal vectors use superscript indices, whereas the original frame used subscript indices. Using the reciprocal vectors we compute coordinates as

$$x^i = \mathbf{x} \cdot \mathbf{b}^i. \tag{2.32}$$

Each $\mathbf{b}^i$ is orthogonal to all $\mathbf{b}_j$, except for $i = j$, in which case $\mathbf{b}^i \cdot \mathbf{b}_i = 1$:

$$\begin{aligned}
\mathbf{b}_i \cdot \mathbf{b}^j &= (-1)^{j-1}\mathbf{b}_i \rfloor \Big((\mathbf{b}_1 \wedge \cdots \wedge \breve{\mathbf{b}}_j \cdots \wedge \mathbf{b}_k) \rfloor \mathbf{I}_k^{-1}\Big) \\
&= (-1)^{j-1}(\mathbf{b}_i \wedge \mathbf{b}_1 \wedge \cdots \wedge \breve{\mathbf{b}}_j \wedge \cdots \wedge \mathbf{b}_k) \rfloor \mathbf{I}_k^{-1} \\
&= \delta_i^j\,(\mathbf{b}_1 \wedge \cdots \wedge \mathbf{b}_i \cdots \wedge \mathbf{b}_k) \rfloor \mathbf{I}_k^{-1} \\
&= \delta_i^j\,\mathbf{I}_k \rfloor \mathbf{I}_k^{-1} \\
&= \delta_i^j,
\end{aligned} \tag{2.33}$$

where δ_i^j is the Kronecker delta function, defined to be 1 when $i = j$ and 0 otherwise. It is now straightforward to verify that $x^i = \mathbf{x} \cdot \mathbf{b}^i$:

$$\mathbf{x} \cdot \mathbf{b}^i = (\sum x^j \mathbf{b}_j) \cdot \mathbf{b}^i = \sum x^j (\mathbf{b}_j \cdot \mathbf{b}^i) = \sum x^j \delta_j^i = x^i.$$

Coordinates are independent of metric, so working with reciprocal frames in order to obtain coordinates may be done in a Euclidean metric, as long as the

same metric is used for computing the frame (Equation 2.31) as for using it (Equation 2.32).

We note that computing a reciprocal frame is related to computing an inverse of a matrix. If the columns of the $n \times k$ matrix B are the vectors of the basis $\mathbf{b}_i$, the columns of the $n \times k$ matrix B_r are the reciprocal vectors, and M is the $n \times n$ metric matrix, then

$$\begin{aligned} B_r^T M\, B &= I, \\ B_r^T &= B^{-1} M^{-1}, \\ B_r &= M^{-1} B^{-T}. \end{aligned}$$

Numerically speaking, explicitly inverting a matrix is not a smart idea, and instead of using reciprocal frames, numerical analists would recommend to solve the system

$$B^T M\, y = x.$$

in order to find coordinates of x with respect to the B-basis. This can be done for instance using QR factorization and back-substitution [23]. This is both cheaper and more stable than explicitly using the reciprocal frame method described above.

2.8.6 Gram-Schmidt Orthogonalization

Gram-Schmidt orthogonalization is a well-known algorithm from linear algebra. It computes an orthogonal basis given a set of non-orthogonal vectors. It is easily rephrased in geometric algebra.

Suppose we have a set of k vectors $\mathbf{v}_i$, and we want to form an orthonormal basis $\mathbf{b}_i$ for the subspace they span. We assume here that the vectors $\mathbf{v}_i$ are independent. We start by setting:

$$\mathbf{b}_1 = \mathbf{v}_1.$$

The next orthogonal vector is computed as:

$$\mathbf{b}_2 = (\mathbf{v}_2 \wedge \mathbf{b}_1)\, \mathbf{b}_1^{-1}.$$

So we compute $\mathbf{b}_2$ by spanning the subspace containing the first basis vector $\mathbf{b}_1$ and the second input vector $\mathbf{v}_2$, and then computing the orthogonal complement of the space already spanned by basis vectors (i.e., by dividing by $\mathbf{b}_1^{-1}$). We can continue this scheme to compute the rest of the vectors:

$$\mathbf{b}_i = (\mathbf{v}_i \wedge \mathbf{b}_{i-1} \wedge \cdots \wedge \mathbf{b}_1)\, (\mathbf{b}_1 \wedge \cdots \wedge \mathbf{b}_{i-1})^{-1}.$$

Note that as such, the produced vectors $\mathbf{b}_i$ are not unit length. This requires normalization of each vector.

As with reciprocal frames, practical implementation of orthogonalization had better be done using existing linear algebra algorithms. E.g., using QR factorization when the metric is orthogonal, and using the eigenvalue decomposition when it the metric is non-orthogonal (see Section 5.3.2).

2.9 Orthogonal Transformations

The ability to handle orthogonal transformations using the geometric product is one of the greatest feats of geometric algebra. In this section we first discuss reflection, the basic building block of orthogonal transformations. This is followed by rotations. When we discuss the conformal model in Section 2.11 we will show how other types of transformations can be orthogonalized.

2.9.1 Reflection

In Equation 2.16 we already showed that

$$\mathbf{a}\,\mathbf{x}\,\mathbf{a} = 2\,(\mathbf{a}\cdot\mathbf{x})\,\mathbf{a} - (\mathbf{a}\cdot\mathbf{a})\,\mathbf{x},$$

and noted that this is the classical expression for reflection. It reflects the vector $\mathbf{x}$ in the 'line' $\mathbf{a}$. In the above equation $\mathbf{a}$ must be unit length. We can drop this unit requirement by using an inverse geometric product, as in:

$$\text{Reflection of } \mathbf{x} \text{ in line } \mathbf{a}: \quad \mathbf{a}\,\mathbf{x}\,\mathbf{a}^{-1} = 2\,(\mathbf{a}\cdot\mathbf{x})\,\mathbf{a}^{-1} - \mathbf{x}. \tag{2.34}$$

It is then easy to show that such a reflection is an orthogonal (metric preserving) transformation:

$$\begin{aligned}(\mathbf{a}\,\mathbf{x}\,\mathbf{a}^{-1})\cdot(\mathbf{a}\,\mathbf{y}\,\mathbf{a}^{-1}) &= (2\,(\mathbf{a}\cdot\mathbf{x})\,\mathbf{a}^{-1} - \mathbf{x})\cdot(2\,(\mathbf{a}\cdot\mathbf{y})\,\mathbf{a}^{-1} - \mathbf{y}) \\ &= -2\,(\mathbf{a}\cdot\mathbf{x})\,(\mathbf{a}^{-1}\cdot\mathbf{y}) - 2\,(\mathbf{a}\cdot\mathbf{y})\,(\mathbf{a}^{-1}\cdot\mathbf{x}) + \\ &\quad\; 4\,(\mathbf{a}\cdot\mathbf{x})\,(\mathbf{a}\cdot\mathbf{y})\,(\mathbf{a}^{-1}\cdot\mathbf{a}^{-1}) + \mathbf{x}\cdot\mathbf{y} \\ &= \mathbf{x}\cdot\mathbf{y}.\end{aligned} \tag{2.35}$$

Figure 2.9 shows that there are actually two types of reflections that can be implemented using variants of the formula $\mathbf{a}\,\mathbf{x}\,\mathbf{a}^{-1}$. The first is reflection in the line spanned by vector $\mathbf{a}$. The other is reflection in the (hyper-)plane orthogonal to $\mathbf{a}$, and uses an extra negation: $-\mathbf{a}\,\mathbf{x}\,\mathbf{a}^{-1}$. This variant is actually the preferred one because it has a determinant of -1 in spaces of any dimension, see Chapter 7.1 of [12].

We can apply plane-reflection to an arbitrary blade $\mathbf{X}$:

$$\text{Reflection of blade } \mathbf{X} \text{ in plane orthogonal to } \mathbf{a}: \quad \mathbf{a}\,\widehat{\mathbf{X}}\,\mathbf{a}^{-1},$$

and as a result the whole blade gets reflected properly. We use the grade involution of $\mathbf{X}$ to handle negation regardless of the grade of $\mathbf{X}$:

$$\mathbf{a}\,\widehat{\mathbf{X}}\,\mathbf{a}^{-1} = (-\mathbf{a}\,\mathbf{x}_1\,\mathbf{a}^{-1}) \wedge (-\mathbf{a}\,\mathbf{x}_2\,\mathbf{a}^{-1}) \wedge \cdots \wedge (-\mathbf{a}\,\mathbf{x}_n\,\mathbf{a}^{-1}). \tag{2.36}$$

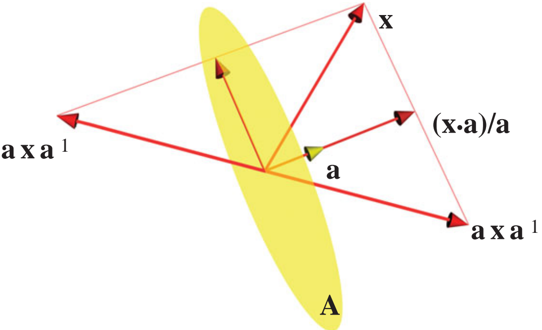


Figure 2.9: *Reflection of a vector* $\mathbf{x}$ *in a vector* $\mathbf{a}$*. Depending on the sign, we get either a line reflection (*$\mathbf{a}\,\mathbf{x}\,\mathbf{a}^{-1}$*) or a plane reflection (*$-\mathbf{a}\,\mathbf{x}\,\mathbf{a}^{-1}$*).*

2.9.2 Rotations

The expression for reflection from the previous section may look like an algebraic curiosity at first, but it is actually one of the main reasons why the geometric product is so useful. Using reflections, all orthogonal transformations can be constructed. For example, two consecutive reflections form a rotation, even in arbitrary metrics. Hence we can perform also rotations using the geometric product. If we reflect a vector $\mathbf{x}$, first in $\mathbf{a}$, then in $\mathbf{b}$, we get

$$\mathbf{b}\,(\mathbf{a}\,\mathbf{x}\,\mathbf{a}^{-1})\,\mathbf{b}^{-1} = (\mathbf{b}\,\mathbf{a})\,\mathbf{x}\,(\mathbf{a}^{-1}\,\mathbf{b}^{-1}) = \mathbf{R}\,\mathbf{x}\,\mathbf{R}^{-1}.$$

We can identify

$$\mathbf{R} = \mathbf{b}\,\mathbf{a}$$

as the *rotation versor*. We call the 'sandwich product' $\mathbf{R}\,\mathbf{x}\,\mathbf{R}^{-1}$ the *application* of $\mathbf{R}$ to $\mathbf{x}$. The geometry is illustrated in Figure 2.10.

To reveal the rotation properties of $\mathbf{R}$ (the plane and the angle of rotation), assume for the moment that $\mathbf{R}$ has a unit reverse norm, such that $\mathbf{R}\,\mathbf{x}\,\mathbf{R}^{-1} = \mathbf{R}\,\mathbf{x}\,\widetilde{\mathbf{R}}$. We split $\mathbf{x}$ into a part orthogonal to $\mathbf{b} \wedge \mathbf{a}$ and a part parallel (contained in) $\mathbf{b} \wedge \mathbf{a}$ such that

$$\mathbf{x} = \mathbf{x}^{\perp} + \mathbf{x}^{\parallel}.$$

It is easy to show that $\mathbf{x}^{\perp}$ is not affected by the application of $\mathbf{R}$. $\mathbf{x}^{\perp}$ commutes

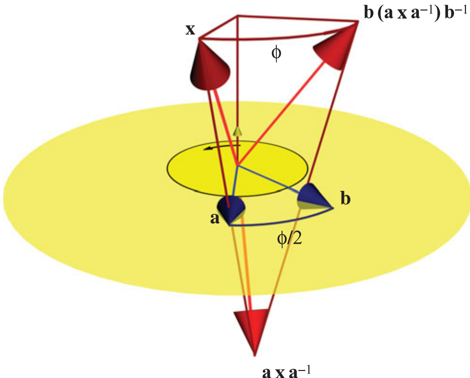


Figure 2.10: *Rotation of vector* $\mathbf{x}$*, executed as two* line-*reflections: first in* $\mathbf{a}$ *and then in* $\mathbf{b}$*.*

with $\mathbf{b} \wedge \mathbf{a}$.

$$\begin{aligned}
\mathbf{R}\,\mathbf{x}^{\perp}\,\mathbf{R}^{-1} &= (\mathbf{b}\cdot\mathbf{a} + \mathbf{b}\wedge\mathbf{a})\,\mathbf{x}^{\perp}\,(\mathbf{b}\cdot\mathbf{a} + \mathbf{a}\wedge\mathbf{b}) \\
&= ((\mathbf{b}\cdot\mathbf{a})\,\mathbf{x}^{\perp} + \mathbf{b}\wedge\mathbf{a}\wedge\mathbf{x}^{\perp})\,(\mathbf{b}\cdot\mathbf{a} + \mathbf{a}\wedge\mathbf{b}) \\
&= (\mathbf{x}^{\perp}\,(\mathbf{b}\cdot\mathbf{a}) + \mathbf{x}^{\perp}\wedge\mathbf{b}\wedge\mathbf{a})\,(\mathbf{b}\cdot\mathbf{a} + \mathbf{a}\wedge\mathbf{b}) \\
&= \mathbf{x}^{\perp}\,(\mathbf{b}\cdot\mathbf{a} + \mathbf{b}\wedge\mathbf{a})\,(\mathbf{b}\cdot\mathbf{a} + \mathbf{a}\wedge\mathbf{b}) \\
&= \mathbf{x}^{\perp}\,\mathbf{R}\,\mathbf{R}^{-1} \\
&= \mathbf{x}^{\perp}.
\end{aligned}$$

Since $\mathbf{x}^{\perp}$ is not affected, we have determined that the rotation must be in the $\mathbf{b} \wedge \mathbf{a}$-plane.

It remains to determine the angle of rotation in that plane. The rotation is composed of two reflections, which are orthogonal (angle preserving) transformations. So we can pick any vector in the $\mathbf{b} \wedge \mathbf{a}$-plane to determine that angle. We pick $\mathbf{a}$, such that we get:

$$\begin{aligned}
\mathbf{R}\,\mathbf{a}\,\mathbf{R}^{-1} &= \mathbf{b}\,\mathbf{a}\,\mathbf{a}\,\mathbf{a}^{-1}\,\mathbf{b}^{-1} \\
&= \mathbf{b}\,\mathbf{a}\,\mathbf{b}^{-1},
\end{aligned}$$

where $\mathbf{b}\,\mathbf{a}\,\mathbf{b}^{-1}$ is the reflection of $\mathbf{a}$ in $\mathbf{b}$. From this it is clear that the rotation must be over twice the angle between $\mathbf{b}$ and $\mathbf{a}$, since the angle between $\mathbf{a}$ and $\mathbf{b}\,\mathbf{a}\,\mathbf{b}^{-1}$ is twice the angle between $\mathbf{a}$ and $\mathbf{b}$.

2.9.3 Rotors as Exponentials of Bivectors

A special class of versors can be written as the exponentials of bivectors. E.g., our rotation versors have the exponential form

$$\mathbf{R} = \cos(\tfrac{1}{2}\phi) - \sin(\tfrac{1}{2}\phi)\,\mathbf{I}_2 = e^{-\frac{1}{2}\phi\,\mathbf{I}_2}. \tag{2.37}$$

Being able to write transformations as exponentials allows for easy interpolation and filtering of transformations because the *log-space* of rotors is linear, since it is the space of bivectors. We will not discuss interpolation further here; for details see Chapter 7 of [12].

The exponential is defined by the usual power series

$$e^{\mathbf{A}} = \sum_{0}^{\infty} \frac{\mathbf{A}^n}{n!} = 1 + \mathbf{A} + \frac{\mathbf{A}^2}{2!} + \frac{\mathbf{A}^3}{3!} + \frac{\mathbf{A}^4}{4!} + \cdots$$

Sine and cosine are defined by similar series:

$$\begin{aligned} \sin\mathbf{A} &= \mathbf{A} - \frac{\mathbf{A}^3}{3!} + \frac{\mathbf{A}^5}{5!} - \cdots, \\ \cos\mathbf{A} &= 1 - \frac{\mathbf{A}^2}{2!} + \frac{\mathbf{A}^4}{4!} - \cdots. \end{aligned}$$

Using these series, we can easily verify Equation 2.37 (using Euclidean metric where $\mathbf{I}_2^2 = -1$):

$$\begin{aligned} e^{\psi\,\mathbf{I}_2} &= 1 + \psi\mathbf{I}_2 + \frac{(\psi\mathbf{I}_2)^2}{2!} + \frac{(\psi\mathbf{I}_2)^3}{3!} + \cdots, \\ &= (1 - \frac{\psi^2}{2!} + \frac{\psi^4}{4!} - \cdots) + (\psi - \frac{\psi^3}{3!} + \frac{\psi^5}{5!} - \cdots)\,\mathbf{I}_2 \\ &= \cos\psi + \mathbf{I}_2\,\sin\psi, \end{aligned}$$

where we have used $\psi = -\frac{1}{2}\phi$ to simplify notation. In general we find for any bivector $\mathbf{A} \in \bigwedge\mathbb{R}^n$ with a scalar square:

$$\exp\mathbf{A} = \begin{cases} \cos\alpha + \mathbf{A}\,\frac{\sin\alpha}{\alpha} & \text{if} \quad \mathbf{A}^2 = -\alpha^2 \\ 1 + \mathbf{A} & \text{if} \quad \mathbf{A}^2 = 0 \\ \cosh\alpha + \mathbf{A}\,\frac{\sinh\alpha}{\alpha} & \text{if} \quad \mathbf{A}^2 = \alpha^2 \end{cases}$$

This equation is useful in the conformal model because in that model many transformations are defined as exponentials of bivectors.

Rotors

Rotors are even versors whose reverse is equal to their inverse. All exponentials of bivectors are rotors, and most even unit versors are rotors, but the precise correspondences are rather subtle, see Chapter 7.4 of [12].

As the name suggests, rotors can only rotate. But this is at the algebraic level. Our interpretation of a specific algebra may cause a rotor to represent other types of motion than just rotation. For example in the conformal model, we will find rotation-rotors, scaling-rotors, translation-rotors, and so on.

Quaternions

Quaternions are just 3-D Euclidean rotors taken out of their proper geometric algebra context. A quaternion q is usually treated as a unit 4-vector with scalar coordinates q_s, q_i, q_j, q_k such that

$$q = q_s + q_i\, i + q_j\, j + q_k\, k,$$

where i, j and k are 'complex vectors' that square to -1 and anti-commute. To convert this back to geometric algebra, we identify these complex vectors with the real basis of 2-blades in 3-D space:

$$i = \mathbf{e}_3\,\mathbf{e}_2 \qquad j = \mathbf{e}_1\,\mathbf{e}_3 \qquad k = \mathbf{e}_2\,\mathbf{e}_1$$

But be aware that other definitions of i, j, and k are in use, differing only in signs.

Logarithms

When you write something in the form of an exponential it makes sense to provide the logarithm too. Unfortunately, a generic closed-form formula for computing the logarithm of an arbitrary rotor is not known. One has to find a specific formula for each case. Here we find the logarithm of a rotor $\mathbf{R}$ in a Euclidean metric.

When $\mathbf{R} = 1$, the $\log(\mathbf{R})$ must be 0, since $e^0 = 1$. When $\mathbf{R} = -1$, the logarithm is not unique since there is no well determined plane of rotation: any unit 2-blade $\mathbf{B}$ gives $e^{\pi\,\mathbf{B}} = -1$. Otherwise, the rotor is of the form

$$\mathbf{R} = \cos(\psi) + \sin(\psi)\,\mathbf{I}_2 = e^{\psi\,\mathbf{I}_2}.$$

We use as plane of rotation

$$\mathbf{I}_2 = \frac{\langle\mathbf{R}\rangle_2}{\|\langle\mathbf{R}\rangle_2\|}.$$

The angle must then be

$$\psi = \operatorname{atan2}(\|\langle\mathbf{R}\rangle_2\|, \langle\mathbf{R}\rangle_0),$$

resulting in the logarithm

$$\mathbf{R}\text{ is not scalar:}\qquad \log(\mathbf{R}) = \operatorname{atan2}(\|\langle\mathbf{R}\rangle_2\|, \langle\mathbf{R}\rangle_0)\,\frac{\langle\mathbf{R}\rangle_2}{\|\langle\mathbf{R}\rangle_2\|}. \tag{2.38}$$

2.10 The Homogeneous Model

The basic use of geometric algebra as discussed so far gets us an algebra of oriented subspaces through the origin. It is possible to reflect and rotate these subspaces, intersect them, project them, measure their properties, and so on, all using elegant and mostly generic equations.

However, this is not enough to implement the type of geometry that is used in computer science. So far, our subspaces represent only directions of various dimensionality, while at the very least we want be able to represent arbitrary points, lines and planes. We should also be able to apply transformation to these objects.

A thoughtless implementation of these demands usually leads to an ad hoc approach, such as the one described for linear algebra in Chapter 1. We can do quite a bit better by using a more structured approach called the *homogeneous model* – and a lot better using the *conformal model* of the next section.

By separating the origin of the algebra from the origin of the represented space, the homogeneous model can represent points, lines and planes as blades. This idea has been long used in traditional geometry implementations in the form of *homogeneous coordinates*, for instance by the computer graphics library `OpenGL` [39]. By merging homogeneous coordinates and geometric algebra, we obtain the homogeneous model, which extends and structures the existing approach.

We intentionally limit the discussion of the homogeneous model, because it serves only as a stepping stone towards the conformal model. Overall, we allow ourselves to be less rigorous in these sections than in the preceding ones, both to limit the size of this chapter and because — within this thesis — these so-called *models of geometry* serve mainly as source of examples in later chapters. Full details on both models may be found in [12].

2.10.1 Blades in the Homogeneous Model

In the homogeneous model there is a much clearer distinction between the algebraic blade and its geometric interpretation. Up to this point, we have semi-implicitly interpreted blades as subspaces. In this section, we interpret blades by their intersection with a particular plane offset from the origin, which results in 'flat objects' such as points and lines.

The Basis and Points

Figure 2.11a illustrates the fundamental idea of the homogeneous model and homogeneous coordinates: the *Euclidean plane* (where we want to do geometry) is embedded in a space with one more dimension. So, assuming we want to do geometry in the 2-D plane (the *base space*), we embed it in a 3-D space, which we call the *embedding space*. The embedding space is put at a unit distance from the algebraic origin O. The basis vector for the extra dimension is usually called e_0.

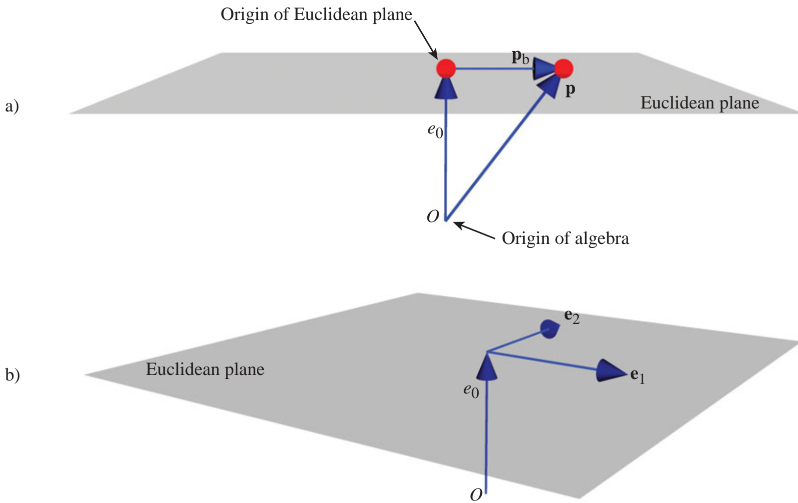


Figure 2.11: *Points and the basis in the homogeneous model.* a) *The difference between the algebraic origin and the origin of the represented space. Points* $\mathbf{p}$ *are constructed by adding a vector* $\mathbf{p}_b$ *to the origin* e_0 *of the represented space.* b) *A basis for the space. The basis vectors* $\mathbf{e}_1$ *and* $\mathbf{e}_2$ *are shown as lying the base space, but this is just for illustration.*

Vectors are interpreted by their intersection with the Euclidean plane, and thus e_0 is interpreted as the *Origin of the Euclidean plane.*

Figure 2.11a also shows a second vector, interpreted as the point $\mathbf{p}$. A point can be split into a part in the base space ($\mathbf{p}_b$) and a part orthogonal to that (e_0). We will use the subscript $_b$ to indicate that a blade that is parallel to the base space.

Figure 2.11b shows a typical 3-D basis for the homogeneous model of 2-D space. $\mathbf{e}_1$ and $\mathbf{e}_2$ are the basis vectors that span the base space, e_0 points to the origin.

Let $\mathbf{p}_b$ be a vector indicating a point in the base space relative to the origin, then it is clear that the homogeneous representation of a point at this location is

$$\mathbf{p} = \mathbf{p}_b + e_0. \tag{2.39}$$

The tip of this vector lies precisely in the plane of the base space. From here on we will often refer to such vectors as points, making no distinction between blades and the geometric objects they represent.

Because vectors are interpreted as their intersection with a plane, the magnitude of the vector is in principle irrelevant. That is, $\mathbf{p}$ and $\alpha\,\mathbf{p}$ represent the same point, as long as $\alpha \neq 0$. Nevertheless, the magnitude can be assigned a useful meaning:

- One may interpret the magnitude as something physically significant like weight, or use it as a counter.

- When points are added, the magnitude of the points determines the relative contribution of each points to the result. This allows for weighted summing of points, which is useful in interpolation.

- In intersection / incidence equations, the magnitude gives information about the (sine of the) intersection angle.

The magnitude may also be negative. This gives points an orientation. When taken into account properly, this orientation is carried on through all computations, which may also be useful in some applications.

Lines

By computing the outer product of several points, we obtain *flats*, i.e., n-dimensional flat subspaces in general position. In a 3-D base space, this means lines and planes, but the principles generalize to any dimension.

For example, to represent a line, we need to construct a 2-blade that intersects the base space in the right way. This is straightforward given two points $\mathbf{p}$ and $\mathbf{q}$ on the line:

$$\mathbf{L} = \mathbf{p} \wedge \mathbf{q}.$$

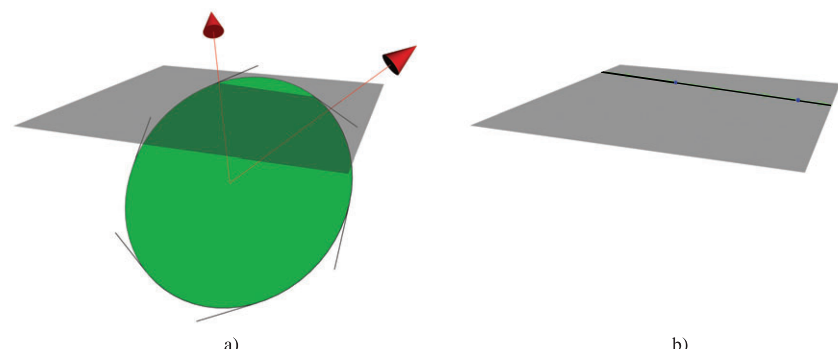


Figure 2.12: *A 2-blade spanned by two vectors is interpreted as the line through the points.* a) *The vectors and blades as subspaces.* b) *The geometric interpretation.*

The line is simply the outer product of the two points. This works because this 2-blade intersects the plane in a line through both points, as illustrated for 2-D in Figure 2.12.

By writing $\mathbf{q} = \mathbf{p} + (\mathbf{q}_b - \mathbf{p}_b)$, we can find an alternative expression based on position and direction:

$$\begin{aligned}\mathbf{L} &= \mathbf{p} \wedge \mathbf{q} \\ &= \mathbf{p} \wedge (\mathbf{p} + (\mathbf{q}_b - \mathbf{p}_b)) \\ &= \mathbf{p} \wedge (\mathbf{q}_b - \mathbf{p}_b).\end{aligned}$$

One might say that $\mathbf{q}_b - \mathbf{p}_b$ is the direction of the line. Hence, the outer product of a point and a 'direction vector' results in a line through the point in the direction of the vector.

Like points, a line has an orientation and a magnitude. Hence, $\mathbf{p} \wedge \mathbf{d}_b$ and $\mathbf{d}_b \wedge \mathbf{p}$ are two lines with the same attitude but different orientations. $\mathbf{p} \wedge 2\,\mathbf{d}_b$ is a line with twice the magnitude as $\mathbf{p} \wedge \mathbf{d}_b$. In Figure 2.12 the orientation of the line is shown using little hooks along the edges. One possible interpretation of magnitude is the 'speed' of the line.

It is easy to show that any interpolated point $\alpha\,\mathbf{p} + \beta\,\mathbf{q}$ lies on the line, by showing containment in the blade $\mathbf{p} \wedge \mathbf{q}$:

$$(\alpha\,\mathbf{p} + \beta\,\mathbf{q}) \wedge \mathbf{p} \wedge \mathbf{q} = \alpha\,\mathbf{p} \wedge \mathbf{p} \wedge \mathbf{q} + \beta\,\mathbf{q} \wedge \mathbf{p} \wedge \mathbf{q} = 0.$$

Planes

We have been limiting ourselves to a 2-D base space for the sake of illustration. If we use a 3-D base space, we can also represent planes. This entirely analogous

to representing lines. Given three points $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$, the plane through each of them is

$$\mathbf{P} = \mathbf{p} \wedge \mathbf{q} \wedge \mathbf{r}.$$

As with lines, this can be rewritten to separate the position from the direction:

$$\begin{aligned}
\mathbf{q} &= \mathbf{p} + (\mathbf{q}_b - \mathbf{p}_b), \\
\mathbf{r} &= \mathbf{p} + (\mathbf{r}_b - \mathbf{p}_b), \\
\mathbf{P} &= \mathbf{p} \wedge \mathbf{q} \wedge \mathbf{r} \\
&= \mathbf{p} \wedge (\mathbf{p} + (\mathbf{q}_b - \mathbf{p}_b)) \wedge (\mathbf{p} + (\mathbf{r}_b - \mathbf{p}_b)) \\
&= \mathbf{p} \wedge (\mathbf{q}_b - \mathbf{p}_b) \wedge (\mathbf{r}_b - \mathbf{p}_b) \\
&= \mathbf{p} \wedge \mathbf{A}_b
\end{aligned}$$

where $\mathbf{A}_b = (\mathbf{q}_b - \mathbf{p}_b) \wedge (\mathbf{r}_b - \mathbf{p}_b)$, a 2-blade in the base space.

k-dimensional Flats

The principle of representing flat subspaces works in any dimension: through the custom interpretation of vectors, we have turned the outer product into a generic constructor for k-dimensional flats.

2.10.2 Infinite Points and Attitudes

A vector $\mathbf{p}_b$ (with no e_0 component) cannot represent a point, since it does not intersect the base space. There are two possible interpretations for such vectors:

- As a point at infinity. One may justifying this by taking the point definition $\mathbf{p}_b + e_0$ and making $\mathbf{p}_b$ infinitely large relative to e_0. In the limit, e_0 becomes so small relative to $\mathbf{p}_b$ that we might as well leave e_0 out. The vector $\mathbf{p}$ then points to the direction where the (infinitely far away) point is located. This turns out very well: for example, using this interpretation, the intersection of two lines in the 2-D plane always results in a point, be it local (for intersecting lines) or at infinity (for parallel lines). We may compute the outer product of two points at infinity, resulting in a line at infinity, and so on.

- As a direction vector. This allows us to work with an algebra of oriented attitudes, along with an algebra of k-flats.

As the need arises, one may switch between both interpretations, although this is the start of another ad hoc approach, that of extending the homogeneous model.

2.10.3 Limitations with the Homogeneous Model

There are several problems in the homogeneous model. One is that fundamental transformations such as translation, reflection and scaling cannot be represented as a versors. Rotation can be represented as a versor, but without translation versors, rotations can only be around the origin. Another problem is that most metric operations (such as orthogonal projection) do not work properly – at least when one uses the standard geometric algebra equations.

The fundamental reason why metric operations involving e_0 can never work properly in the homogeneous model is that metric products involving e_0 are not translation invariant. For example, the inner product of the origin with itself is not equal when a translated version of the origin is used:

$$e_0 \cdot e_0 \neq (e_0 + \mathbf{t}_b) \cdot (e_0 + \mathbf{t}_b) = e_0 \cdot e_0 + \mathbf{t}_b \cdot \mathbf{t}_b.$$

We will now discuss the conformal model, which fixes this problem and significantly extends the homogeneous model.

2.11 The Conformal Model

The conformal model can be considered an 'upgrade' of the homogeneous model. It fixes many of the problems related to the metric and significantly extends the number of objects that can be directly represented. It can be used to model various conformal (angle preserving) geometries. Euclidean geometry is the most interesting one for computer science applications, and we will limit ourselves to that in this section.

One can define a geometry by the properties that transformations should keep constant. "Conformal" means angle-preserving, and all angle-preserving transformations can be represented by versors in the conformal model. A proper subset of these transformations are the Euclidean transformations, which keep all distances between points fixed.

Being able to represent all transformations of a geometry as versors is significant because it unifies their treatment and allows for universal application of transformations to objects and to other transformations. Another advantage is that many of these transformations have logarithms, which simplifies interpolation.

Besides representing the transformations as versors, the conformal model can represent flat objects, rounds objects and several other useful types of objects such as "a location with a direction" as blades. This increases its power over the homogeneous model, which can represent only flat objects and has limited support for directions.

One could say that two techniques underlie the power of the conformal model: the interpretation of blades and the special metric.

Interpretation of Blades in the Conformal Model

One can think of the blade-interpretation-process in the conformal model as a two-step process, in which we move from blades in the $n+2$-dimensional representational space to objects in the n-dimensional base space:

- First the blades in the representational space are intersected with a (hyper-) plane, just as in the homogeneous model. This allows arbitrary flats in an $(n+1)$-dimensional space to be represented, as in the homogeneous model.
- These flats in $(n+1)$-D space are then intersected with a n-dimensional paraboloid, the *horosphere* (see Figure 2.13). This intersection is projected onto an n-dimensional (hyper-) plane orthogonal to the axis of symmetry of paraboloid. This results in (oriented) circles, lines, planes, spheres, and so on, in the n-dimensional base space.

This is a simplistic description, and we will return to the subject shortly in Section 2.11.3. In any case, two dimensions are 'lost' in the interpretation process, so to model an n-dimensional geometry using the conformal model, we need a geometric algebra over an $(n+2)$-dimensional vector space.

2.11.1 The Basis and Metric

The basis vectors for the base space are usually called just $\mathbf{e}_1$, $\mathbf{e}_2$ and so on. We prefer to call the two basis vectors for the extra dimensions o and ∞ and pronounce them as 'no' and 'ni', for *Null vector representing the Origin*, and *Null vector representing Infinity*, respectively. In this context, the symbol ∞ should not be confused with the scalar value infinity for which it is usually employed.

The metric multiplication table of the conformal model is as follows:

	o	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_3$	∞
o	0	0	0	0	-1
$\mathbf{e}_1$	0	1	0	0	0
$\mathbf{e}_2$	0	0	1	0	0
$\mathbf{e}_3$	0	0	0	1	0
∞	-1	0	0	0	0

(2.40)

This defines a so-called *Minkowski metric* $\mathbb{R}^{n+1,1}$ which is also used in physics for space-time.

Note that the basis we use is not orthogonal. An alternative basis for the conformal model uses two vectors named e and $\bar{e}$, with $e \cdot e = 1$ and $\bar{e} \cdot \bar{e} = -1$. This leads to an orthogonal metric matrix. The relation between these vectors and o and ∞ is

$$
\begin{aligned}
o &= \frac{1}{\sqrt{2}}(e + \bar{e}), \\
\infty &= \frac{1}{\sqrt{2}}(\bar{e} - e).
\end{aligned}
$$

Since we are mainly concerned with Euclidean geometry, using the $\{e, \bar{e}\}$-basis complicates matters because they have less geometrical significance in Euclidean geometry: e and $\bar{e}$ represent spheres, while o and ∞ represent the origin and infinity, respectively[3]. Additionally, the $\{o, \infty\}$-basis may have a performance advantage, as shown in Section 7.3.2. The orthogonal e, $\bar{e}$-basis does have its use for computing products of basis blades, and we will encounter it again in Section 3.1.6.

2.11.2 Representation of Points

One of the reasons for the strange metric of the conformal model becomes apparent when we define points and study their inner product. Conformal points $\mathbf{p}$ are represented by vectors:

$$\mathbf{p} = \mathbf{p}_b + o + \tfrac{1}{2}\mathbf{p}_b^2\infty. \tag{2.41}$$

$\mathbf{p}_b$ is a vector in the base space. o serves the same role as e_0 in the homogeneous model. The $\frac{1}{2}\mathbf{p}_b^2\infty$-part of the definition specifies that all points lie on a parabola in the ∞ direction. The inner product between two points is then proportional to their distance squared:

$$\begin{aligned}
\mathbf{p}\cdot\mathbf{q} &= (\mathbf{p}_b + o + \tfrac{1}{2}\mathbf{p}_b^2\infty)\cdot(\mathbf{q_b} + o + \tfrac{1}{2}\mathbf{q_b}^2\infty) \\
&= \mathbf{p}_b\cdot\mathbf{q_b} + \tfrac{1}{2}\mathbf{q_b}^2 o\cdot\infty + \tfrac{1}{2}\mathbf{p}_b^2\infty\cdot o \\
&= \mathbf{p}_b\cdot\mathbf{q_b} - \tfrac{1}{2}\mathbf{q_b}^2 - \tfrac{1}{2}\mathbf{p}_b^2 \\
&= -\tfrac{1}{2}(\mathbf{p}_b - \mathbf{q_b})^2.
\end{aligned} \tag{2.42}$$

Note that the inner product of points is independent of the origin: o does not appear on the last line of Equation 2.42. Because distance between a point and itself is 0, points must be represented by null vectors:

$$\mathbf{p}\cdot\mathbf{p} = \tfrac{1}{2}(\mathbf{p}_b - \mathbf{p_b})^2 = 0.$$

We can use Equation 2.42 to derive a function to compute the Euclidean distance between two (normalized) points:

$$\mathrm{dist}(\mathbf{p}, \mathbf{q}) = \sqrt{-2\,\mathbf{p}\cdot\mathbf{q}}.$$

Normalization of Points

We consider a point $\mathbf{p}$ to be *normalized* when $-\infty\cdot\mathbf{p} = 1$. This just means that the o-component of the vector has a size 1. So a point is 'normalized' using

$$\mathbf{p}_n = \frac{\mathbf{p}}{-\infty\cdot\mathbf{p}}.$$

[3]Note that the e and $\bar{e}$ which are used in [12] have a different scale. The e and $\bar{e}$ which are used here are the result of our (automatically generated) implementation of Chapter 4 and are normalized in the Euclidean sense (through the use of the eigenvalue decomposition).

We can easily adjust dist() to take non-normalized points into account:

$$\text{dist_nn}(\mathbf{p}, \mathbf{q}) = \sqrt{\frac{-2\,\mathbf{p} \cdot \mathbf{q}}{(\infty \cdot \mathbf{p})\,(\infty \cdot \mathbf{q})}}.$$

2.11.3 Conformal Blades and their Interpretation

In this section we discuss how blades are interpreted in the conformal model, and what types of geometric objects this leads to. This ultimately results in the classification tree of blades and versors shown in Figure 2.15.

∞ is the Point at Infinity

In the section on transformations, we will see that ∞ is not affected by rotation and translation. This suggests that ∞ represents the point at infinity. The inner product of a finite point $\mathbf{p}$ with infinity is

$$\mathbf{p} \cdot \infty = (\mathbf{p}_b + o + \tfrac{1}{2}\mathbf{p}_b^2 \infty) \cdot \infty = -1,$$

which seems a bit strange, since one would expect the distance to be infinite. However, if we naively plug ∞ into the distance function, we get (in the limit)

$$\text{dist_nn}(\mathbf{p}, \infty) = \sqrt{\frac{-2\,\mathbf{p} \cdot \infty}{(\infty \cdot \mathbf{p})\,(\infty \cdot \infty)}} = \sqrt{\frac{2}{0}} = \bowtie,$$

which fixes our intuition (we use $\bowtie$ as the symbol to denote the regular scalar value infinity).

The Horosphere

Equation 2.41 specified how the conformal model uses $(n+2)$-D vectors to represent n-D points. It is repeated here for convenience:

$$\mathbf{p} = \mathbf{p}_b + o + \tfrac{1}{2}\mathbf{p}_b^2 \infty.$$

The role of the o-component is the same as the role of e_0 in the homogeneous model. It allows us to position arbitrary flat subspaces anywhere in the $(n+1)$-D space spanned by the base space plus ∞.

We can draw those flats in $(n+1)$-D space for $n = 2$, which helps us to understand the blade interpretation process of the conformal model. See for example Figure 2.13a. It shows the 2-D Euclidean plane (spanned by $\mathbf{e}_1$ and $\mathbf{e}_2$) with the horosphere above it. The axis of symmetry of the horosphere is in the ∞-direction.

Conformal blades are interpreted by intersecting them with this horosphere, and projecting the intersection down onto the Euclidean base plane. In Figure 2.13a this results in a circle.

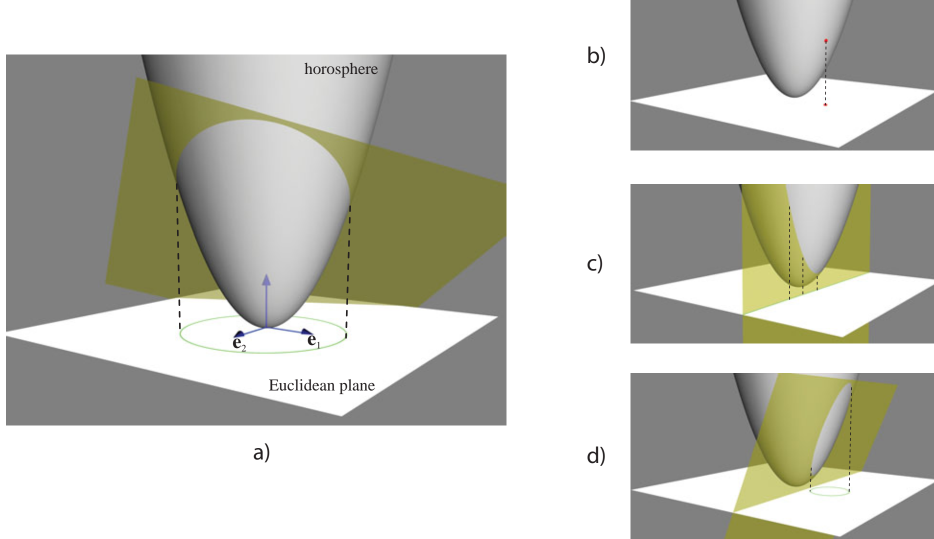


Figure 2.13: a) *The central idea for interpreting blades in the conformal model: flats in the* $(n+1)$*-D space are intersected with the* horosphere*, and the resulting conic is projected onto the* Euclidean plane*. In this figure, a circle results. The projections are indicated (approximately) by the dashed lines. The basis for conformal space (except o) is also shown.* b) *Points are represented by null vectors on the horosphere.* c) *Lines are represented by planes parallel to the axis of symmetry of the horosphere.* d) *Circles are represented by planes which are not parallel to the axis of symmetry of the horosphere.*

The horosphere is made of all null vectors which represent points. Hence, the basic principle for interpreting conformal blades is finding the null vectors (points) they contain.

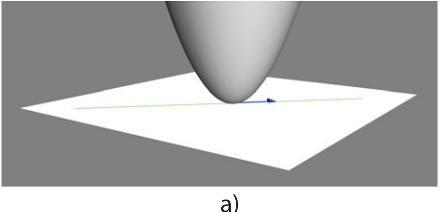
Points

An n-D point is just a $(n+1)$-D point that lies precisely on the parabola. See Figure 2.13b. In later chapters, we will refer to these points as *conformal points.*

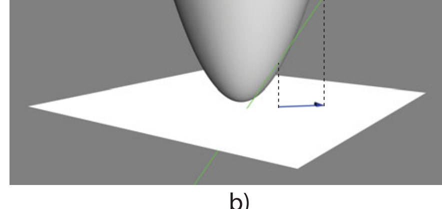
Euclidean Flats

A flat that runs parallel to the axis of symmetry of the horosphere is interpreted as a flat in the Euclidean plane. Figure 2.13c shows illustrates this case for a line. Such a line is created as the outer product of two points on the line and infinity:

$$\mathbf{L} = \mathbf{p} \wedge \mathbf{q} \wedge \infty.$$



a)



b)

Figure 2.14: *Tangent blades.* a) *The green line is a flat in* $(n+1)$*-D space. It touches the horosphere in the origin. It is interpreted as the blue* tangent vector. b) *When a tangent blade is translated, it also rotates in* $(n+1)$*-D space in such a way that it remains tangent to the horosphere.*

Another way to view this situation is to determine the null vectors that are in range of the subspace spanned by $\mathbf{p}$, $\mathbf{q}$ and ∞. These are all points on the line through $\mathbf{p}$ and $\mathbf{q}$:

$$\mathbf{x} = \alpha\mathbf{p} + \beta\mathbf{q} + \gamma\infty.$$

γ is then used to add exactly the right amount of ∞ to make $\mathbf{x}$ a null vector.

Rounds

If we create a flat that does not run parallel to the axis of the horosphere, the intersection is an ellipsoid. Figure 2.13d illustrates this situation for a circle. A circle is therefore constructed as the outer product of three points (which do not lie on a line):

$$\mathbf{C} = \mathbf{p} \wedge \mathbf{q} \wedge \mathbf{r}.$$

Note that flats that do not intersect the paraboloid can be interpreted as *imaginary rounds*, which we do not discuss here.

Tangent blades

Figure 2.14 shows two flats that barely touch the horosphere. They obviously should have a point-like interpretation since they touch the horosphere at one specific point. But they also have another component that may be interpreted as a direction. The simplest way to construct such a flat is at the origin (Figure 2.14a)

$$\mathbf{T} = o \wedge \mathbf{A}_b,$$

where $\mathbf{A}_b$ is some blade from the base space (i.e., a vector in Figure 2.14a). No other null vectors than o itself are in range of this blade. We may employ $\mathbf{A}_b$ to indicate a direction specific to the point o. That is why we have called them *tangent blades*.

One may worry that if a tangent vector at the origin is translated to some other position, unintended null vectors may come into range of the blade. But as seen in Figure 2.14b, it does not work out this way. When a tangent blade is translated through a translation versor, it is also automatically rotated in $(n+1)$-D space just so that it remains tangent to the horosphere. See Figure 2.14b. One may derive that the general form of a tangent vector at location $\mathbf{p}$ is:

$$\mathbf{p} \wedge (\mathbf{p} \rfloor (\mathbf{A}_b \wedge \infty)).$$

Free blades

The final type of conformal blade is of the form

$$\mathbf{F} = \mathbf{A}_b \wedge \infty.$$

Since this blade has no o component, it is interpreted as a flat at infinity in $(n+1)$-D space, see Section 2.10.2. Because of that, we cannot draw an image of it as we did for the other three types of conformal blades (flat, round, tangent). It also has no points in its range. For this reason we call such blades *free blades*: inside conformal space, they are everywhere and they are nowhere. These free blades behave like directional elements with no location: translating them has no effect since the ∞ factor kills any attempt to displace $\mathbf{A}_b$. Rotation just rotates the $\mathbf{A}_b$ part of a free blade, as it should.

Classification Complete

In all, there are four main groups of blades (flats, rounds, tangents, frees) in the conformal model. This is an exhaustive list [13], and it is illustrated in Figure 2.15. The figure also shows the versors (with transversion not discussed here). Note that many blades can be used as transformations since invertible blades are versors.

Chapter 14.2 of [12] gives more detail on the classification of conformal blades, and discusses how to extract their parameters (such as location, radius, direction and so on).

2.11.4 Transformations

This section describes the versor representation of Euclidean transformations. Some transformations (those that can be performed using infinitely many infinitesimal steps) are not just versors but rotors, so they also have an exponential form. We demonstrate the correctness of the action of the transformations on points; their action on arbitrary blades follows through the outermorphism property.

There are more transformations in the conformal model than are discussed here. Examples are transversion, negative scaling, and reflection in a sphere. Full details can be found in Chapter 16 of [12].

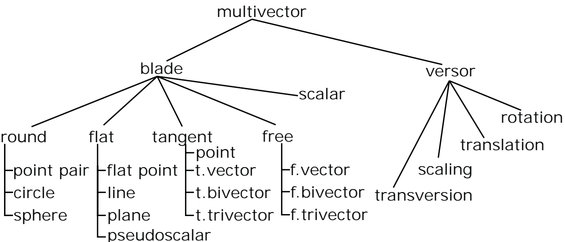


Figure 2.15: *Classification of the blades and versors of the conformal model of 3-D geometry. The strange location of the* point *in the tree can be justified by considering it to be a 'tangent scalar', i.e., a scalar with a position. Note that the classification is incomplete and not fully precise. For instance, some blades are versors too (e.g., planes), and versors can be combined (e.g., rotation-translation versors).*

Reflection in a Plane

We start with reflection in a plane because it is the fundamental building block of the Euclidean transformations. As we will see later on, a dual plane through the origin has the form

$$\mathbf{P}^\star = \mathbf{n}_b,$$

where $\mathbf{n}_b$ is a (normal) vector in the base space. When we apply $\mathbf{P}^\star$ to a point using 'versor-sandwiching', we find

$$\begin{aligned} -\,\mathbf{n}_b\,(\mathbf{p}_b + o + \tfrac{1}{2}\mathbf{p}_b^2\infty)\,\mathbf{n}_b^{-1} &= -\mathbf{n}_b\,\mathbf{p}_b\,\mathbf{n}_b^{-1} - \mathbf{n}_b\,o\,\mathbf{n}_b^{-1} - \tfrac{1}{2}\mathbf{n}_b\,\mathbf{p}_b^2\infty\,\mathbf{n}_b^{-1} \\ &= -\mathbf{n}_b\,\mathbf{p}_b\,\mathbf{n}_b^{-1} + o + \tfrac{1}{2}(-\mathbf{n}_b\,\mathbf{p}_b\,\mathbf{n}_b^{-1})^2\infty \\ &= -\mathbf{n}_b\,\mathbf{p}_b\,\mathbf{n}_b^{-1} + o + \tfrac{1}{2}\mathbf{p}_b^2\infty. \end{aligned}$$

In other words, the base-space-part of the points gets reflected in the plane while o and ∞ are unaffected: the correct result. We do not have to prove that this reflection-construction works for arbitrary planes (not through the origin), since we are free to choose the origin (this result is presented right below).

Translation

A translation can be performed by applying two consecutive reflections in parallel planes. A translation over vector $\mathbf{t}_b$ is represented by the rotor

$$\mathbf{T} = 1 - \tfrac{1}{2}\mathbf{t}_b\,\infty = e^{-\frac{1}{2}\mathbf{t}_b\,\infty}. \quad (2.43)$$

Showing that this rotor works correctly on an arbitrary point $\mathbf{p}$ is rather involved:

$$\begin{aligned}
\mathbf{T}\,\mathbf{p}\mathbf{T}^{-1} &= (1-\tfrac{1}{2}\mathbf{t}_b\,\infty)\,(\mathbf{p}_b+o+\tfrac{1}{2}\mathbf{p}_b^2\infty)\,(1+\tfrac{1}{2}\mathbf{t}_b\,\infty)\\
&= (1-\tfrac{1}{2}\mathbf{t}_b\,\infty)\,\mathbf{p}_b\,(1+\tfrac{1}{2}\mathbf{t}_b\,\infty)\,+\\
&\quad\ (1-\tfrac{1}{2}\mathbf{t}_b\,\infty)\,o\,(1+\tfrac{1}{2}\mathbf{t}_b\,\infty)\,+\\
&\quad\ (1-\tfrac{1}{2}\mathbf{t}_b\,\infty)\,\tfrac{1}{2}\mathbf{p}_b^2\infty\,(1+\tfrac{1}{2}\mathbf{t}_b\,\infty)\\
&= (\mathbf{p}_b+\tfrac{1}{2}\mathbf{p}_b\,\mathbf{t}_b\,\infty-\tfrac{1}{2}\mathbf{t}_b\,\infty\,\mathbf{p}_b)+(o+\mathbf{t}_b-\tfrac{1}{2}\mathbf{t}_b^2\,\infty)+(\tfrac{1}{2}\mathbf{p}_b^2\infty)\\
&= (\mathbf{p}_b+\tfrac{1}{2}\mathbf{p}_b\,\mathbf{t}_b\,\infty+\tfrac{1}{2}\mathbf{t}_b\,\mathbf{p}_b\,\infty)+(o+\mathbf{t}_b-\tfrac{1}{2}\mathbf{t}_b^2\,\infty)+(\tfrac{1}{2}\mathbf{p}_b^2\infty)\\
&= \mathbf{p}_b+\mathbf{t}_b+o+\tfrac{1}{2}\mathbf{p}_b^2\infty+\mathbf{p}_b\cdot\mathbf{t}_b\infty+\tfrac{1}{2}\mathbf{t}_b^2\infty\\
&= (\mathbf{p}_b+\mathbf{t}_b)+o+\tfrac{1}{2}(\mathbf{p}_b+\mathbf{t}_b)^2\,\infty.
\end{aligned}$$

Once we have translation as a rotor, we are free to prove translation invariant geometric theorems at any finite point in space. We can simply translate the whole space as we see fit. This means that there is no reason to pick any a particular point as the origin, which simplifies many proofs because we can prove them relative to our origin of choice. To illustrate this, let us prove that a translation rotor works correctly on any point, by showing it works correctly on the origin

$$\begin{aligned}
\mathbf{T}\,o\,\mathbf{T}^{-1} &= (1-\tfrac{1}{2}\mathbf{t}_b\,\infty)\,o\,(1+\tfrac{1}{2}\mathbf{t}_b\,\infty)\\
&= (o-\tfrac{1}{2}\mathbf{t}_b\,\infty\,o)\,(1+\tfrac{1}{2}\mathbf{t}_b\,\infty)\\
&= \mathbf{t}_b+o+\tfrac{1}{2}\mathbf{t}_b^2\,\infty.
\end{aligned}$$

This is a lot shorter than the version above, which we had to prove first, before allowing ourselves to apply this pick-your-own-origin technique.

Rotation

Rotation about an axis through the origin can be performed by applying two consecutive reflections in planes through the origin. The rotation rotor we already used in Section 2.9.2 can be directly used in the conformal model:

$$\mathbf{R}=\cos(\tfrac{1}{2}\phi)-\,\sin(\tfrac{1}{2}\phi)\,\mathbf{I}_b=e^{-\frac{1}{2}\phi\,\mathbf{I}_b}.$$

Being composed of two plane-reflections through the origin, it rotates the base-space part of points, and leaves o and ∞ unaffected:

$$\begin{aligned}
\mathbf{R}\,(\mathbf{p}_b+o+\tfrac{1}{2}\mathbf{p}_b^2\infty)\,\mathbf{R}^{-1} &= \mathbf{R}\mathbf{p}_b\mathbf{R}^{-1}+o+\tfrac{1}{2}(\mathbf{R}\,\mathbf{p}_b\,\mathbf{R}^{-1})^2\,\infty)\\
&= \mathbf{R}\mathbf{p}_b\mathbf{R}^{-1}+o+\tfrac{1}{2}\mathbf{p}_b^2\infty.
\end{aligned}$$

By combining rotation and translation rotors, we can obtain general Euclidean motions in rotor form.

Scaling

Uniform scaling can be accomplished by two consecutive reflections in spheres with the same center point. We do not discuss sphere-reflections here. The rotor

$$\mathbf{S} = \cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma) o \wedge \infty = e^{\frac{1}{2}\gamma o \wedge \infty} \tag{2.44}$$

scales (relative to the origin) by e^{γ}. To prove this, we first note that:

$$\begin{aligned}
(o \wedge \infty)^2 &= 1 \\
(o \wedge \infty)\, o &= -o = -o\,(o \wedge \infty) \\
(o \wedge \infty)\, \infty &= \infty = -\infty\,(o \wedge \infty),
\end{aligned}$$

and then derive:

$$\begin{aligned}
\mathbf{S}\,\mathbf{p}\,\mathbf{S}^{-1} &= (\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)\,(\mathbf{p}_b + o + \tfrac{1}{2}\mathbf{p}_b^2\infty)\,(\cosh(\tfrac{1}{2}\gamma) - \sinh(\tfrac{1}{2}\gamma)) \\
&= (\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)\,\mathbf{p}_b\,(\cosh(\tfrac{1}{2}\gamma) - \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty) + \\
&\quad (\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)\, o\,(\cosh(\tfrac{1}{2}\gamma) - \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty) + \\
&\quad (\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)\, \tfrac{1}{2}\mathbf{p}_b^2\infty\,(\cosh(\tfrac{1}{2}\gamma) - \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty) \\
&= \mathbf{p}_b + \\
&\quad (\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)\,(\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)\, o + \\
&\quad (\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)\,(\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)\, \tfrac{1}{2}\mathbf{p}_b^2\infty \\
&= \mathbf{p}_b + \\
&\quad (\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)^2\, o + \\
&\quad (\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)^2\, \tfrac{1}{2}\mathbf{p}_b^2\infty \\
&= \mathbf{p}_b + (\cosh(\tfrac{1}{2}\gamma) + \sinh(\tfrac{1}{2}\gamma)\, o \wedge \infty)^2\,(o + \tfrac{1}{2}\mathbf{p}_b^2\infty) \\
&= \mathbf{p}_b + e^{\gamma o \wedge \infty}\,(o + \tfrac{1}{2}\mathbf{p}_b^2\infty) \\
&= \mathbf{p}_b + (\cosh\gamma + \sinh\gamma o \wedge \infty)\,(o + \tfrac{1}{2}\mathbf{p}_b^2\infty) \\
&= \mathbf{p}_b + (\cosh\gamma - \sinh\gamma) o + (\cosh\gamma + \sinh\gamma)\tfrac{1}{2}\mathbf{p}_b^2\infty \\
&= \mathbf{p}_b + e^{-\gamma} o + e^{\gamma}\tfrac{1}{2}\mathbf{p}_b^2\infty.
\end{aligned}$$

Through normalization we get:

$$\frac{\mathbf{S}\,\mathbf{p}\,\mathbf{S}^{-1}}{e^{-\gamma}} = e^{\gamma}\mathbf{p}_b + o + \tfrac{1}{2}(e^{\gamma}\mathbf{p}_b)^2\infty.$$

Hence the rotor scales by e^{γ}.

2.12 Outermorphisms

So far we have not treated outermorphisms in detail because we did not want to disrupt the flow towards the conformal model. They are however an important part of our additive implementation discussed in Chapter 4.

Any grade preserving linear transformation of vectors ($\mathsf{f} : \mathbb{R}^n \to \mathbb{R}^n$) has the properties

$$\begin{cases} \mathsf{f}(\alpha \mathbf{x}) & = & \alpha \, \mathsf{f}(\mathbf{x}) \\ \mathsf{f}(\mathbf{x} + \mathbf{y}) & = & \mathsf{f}(\mathbf{x}) + \mathsf{f}(\mathbf{y}) \end{cases}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors. Because of linearity, it is straightforward to extend f such that it also works on blades in a way that preserves the structure of the outer product

$$\mathsf{f}(\mathbf{x} \wedge \mathbf{y}) = \mathsf{f}(\mathbf{x}) \wedge \mathsf{f}(\mathbf{y}).$$

Because of this, these transformations are called *outermorphisms*. They preserve the properties of the outer product. Examples of outermorphisms are projection and rotation.

A nice property of linear transformations is that they can be represented by matrices. These matrices can then be used to apply transformations to vectors as is regularly done in linear algebra. In general, manipulating the mapping is done most efficiently using pure geometric algebra, while applying them is in general more efficient using matrices.

This idea can also be extended to work on blades. Applying transformations to blades using matrices can be more efficient than applying them using the original geometric algebra expression that generated the mapping.

We first show an example (in 3-D) of how this works out for vectors, and then generalize to arbitrary blades. Let L^1 is an $1 \times n$ matrix containing a basis for the grade 1. For example,

$$L^1 = [\mathbf{e}_1 \quad \mathbf{e}_2 \quad \mathbf{e}_3] \, .$$

The basis does not have to be orthogonal. Using L^1, we can represent a vector $\mathbf{a}$ as an $n \times 1$ matrix $[\mathbf{a}]$, as in

$$\mathbf{a} = L^1 \, [\mathbf{a}] = [\mathbf{a}]_1 \, \mathbf{e}_1 + [\mathbf{a}]_2 \, \mathbf{e}_2 + \ldots + [\mathbf{a}]_3 \, \mathbf{e}_3,$$

where L^1 is an $1 \times n$ matrix containing the grade 1 basis vectors.

Let matrix $[\mathsf{f}]^1$ represent the linear transformation f (we use the superscript 1 to denote that this matrix is for grade 1 blades). The columns of $[\mathsf{f}]^1$ must be the images of the basis vectors under the transformation, because the outcome of a matrix multiplication (e.g., $[\mathsf{f}]^1[\mathbf{a}]$) is a weighted sum of the columns of $[\mathsf{f}]^1$, where the weights are specified by the respective rows of $[\mathbf{a}]$:

$$\begin{aligned} [\mathsf{f}(\mathbf{a})] &= [\mathsf{f}([\mathbf{a}]_{[1]} \, \mathbf{e}_1) + \mathsf{f}([\mathbf{a}]_{[2]} \, \mathbf{e}_2) + \ldots + \mathsf{f}([\mathbf{a}]_{[n]} \, \mathbf{e}_n)] \\ &= [[\mathbf{a}]_{[1]} \, \mathsf{f}(\mathbf{e}_1) + [\mathbf{a}]_{[2]} \, \mathsf{f}(\mathbf{e}_2) + \ldots + [\mathbf{a}]_{[n]} \, \mathsf{f}(\mathbf{e}_n)] \\ &= [\mathbf{a}]_1 \, [\mathsf{f}]^1_{[:,1]} + [\mathbf{a}]_2 \, [\mathsf{f}]^1_{[:,2]} + \ldots + [\mathbf{a}]_n \, [\mathsf{f}]^1_{[:,n]}, \end{aligned}$$

hence

$$[\mathsf{f}]^1 = [\ [\mathsf{f}(\mathbf{e}_1)] \quad [\mathsf{f}(\mathbf{e}_2)] \quad [\mathsf{f}(\mathbf{e}_3)]\ ] \, .$$

We can extend this idea to other grades for the case of the additive representation. For clarity of discussion, we use a 2-blade in this example, but the results can

straightforwardly be extended to all grades and spaces. We can again define a basis for grade 2 in 3-D space, and store it in a matrix L^2:

$$L^2 = [\ \mathbf{e}_1 \wedge \mathbf{e}_2 \quad \mathbf{e}_2 \wedge \mathbf{e}_3 \quad \mathbf{e}_3 \wedge \mathbf{e}_1\ ].$$

Let $\mathbf{B}$ be some 2-blade. Using L^2, we can represent $\mathbf{B}$ using an $n \times 1$ matrix $[\mathbf{B}]$ such that

$$\mathbf{B} = L^2[\mathbf{B}] = [\mathbf{B}]_{[1]}\, \mathbf{e}_1 \wedge \mathbf{e}_2 + [\mathbf{B}]_{[2]}\, \mathbf{e}_2 \wedge \mathbf{e}_3 + [\mathbf{B}]_{[3]}\, \mathbf{e}_3 \wedge \mathbf{e}_1.$$

We now want to find a matrix $[\mathsf{f}]^2$ that can be used to compute $[\mathsf{f}(\mathbf{B})]$.

$$\begin{aligned} [\mathsf{f}(\mathbf{B})] &= [\mathsf{f}([\mathbf{B}]_{[1]}\, \mathbf{e}_1 \wedge \mathbf{e}_2) + \mathsf{f}([\mathbf{B}]_{[2]}\, \mathbf{e}_2 \wedge \mathbf{e}_3) + \mathsf{f}([\mathbf{B}]_{[3]}\, \mathbf{e}_3 \wedge \mathbf{e}_1)] \\ &= [[\mathbf{B}]_{[1]}\, \mathsf{f}(\mathbf{e}_1 \wedge \mathbf{e}_2) + [\mathbf{B}]_{[2]}\, \mathsf{f}(\mathbf{e}_2 \wedge \mathbf{e}_3) + [\mathbf{a}]_{[3]}\, \mathsf{f}(\mathbf{e}_3 \wedge \mathbf{e}_1)] \\ &= [\mathbf{B}]_1\, [\mathsf{f}]^2_{[:,1]} + [\mathbf{B}]_2\, [\mathsf{f}]^2_{[:,2]} + [\mathbf{B}]_3\, [\mathsf{f}]^2_{[:,3]}. \end{aligned}$$

Hence, this matrix $[\mathsf{f}]^2$ should be:

$$[\mathsf{f}]^2 = [\ [\mathsf{f}(\mathbf{e}_1 \wedge \mathbf{e}_2)] \quad [\mathsf{f}(\mathbf{e}_2 \wedge \mathbf{e}_3)] \quad [\mathsf{f}(\mathbf{e}_3 \wedge \mathbf{e}_1)]\ ].$$

In other words, if we create a matrix $[\mathsf{f}]^2$ whose columns are the images of the grade 2 basis blades under the transformation, then we can use this matrix to transform arbitrary grade 2 blades that are also represented on this basis.

Such a matrix $[\mathsf{f}]^k$ can be constructed for any grade $k \geq 1$. Note that these matrices are highly dependent on the basis vectors, the orientation of the basis blades, and order of basis blades. We will return on the practical issues of using these techniques in Section 4.6.7 (additive implementation) and Section 5.2.14 (multiplicative implementation).

2.13 `meet` and `join`

The `meet` and `join` are two non-linear products which we have mostly avoided so far. They compute blade intersection and union, respectively. Due to non-linearity the `meet` and `join` cannot be extended to versors or general multivectors. The `join` $\mathbf{A} \cup \mathbf{B}$ is the lowest-grade blade that contains both operands $\mathbf{A}$ and $\mathbf{B}$. The `meet` $\mathbf{A} \cap \mathbf{B}$ is the highest-grade blade that is contained in both operands. For the definition of containment, see Section 2.4.5.

The definitions of `meet` and `join` say nothing about the scale of these blades; only the attitude matters, not the orientation or scale. So in many equations below we will use $\sim$ instead of $=$ ("is proportional to" instead of "is equal to").

We can use a Euclidean metric to work with the `meet` and `join` because they do not involve metric (see Section 2.5.6). Hence we assume a Euclidean metric for the rest of this section.

We already discussed intersection, for example in Section 2.8.4, so one may wonder why there is a need for two special products to compute blade intersection

and union. However, to compute an intersection using Equation 2.30 always requires a pseudoscalar with respect to which the dual is computed. In low-dimensional spaces (relevant for many computer science applications), one can often assume a certain value for this pseudoscalar, which speeds up computations. But in general, the `join` must be used to compute it. In this light, Equation 2.30 tells you only how to compute the `meet` given the `join`, which is not very useful in general.

Another way to understand why we need special products to compute blade intersection and union is linearity. Intersection is not a linear operation in general. For example

$$\mathbf{e}_1 \cap (\mathbf{e}_1 + \mathbf{e}_2) \quad \not\sim \quad \mathbf{e}_1 \cap \mathbf{e}_1 + \mathbf{e}_1 \cap \mathbf{e}_2,$$

because

$$\begin{aligned} \mathbf{e}_1 \cap (\mathbf{e}_1 + \mathbf{e}_2) \quad &\sim \quad 1 \\ \mathbf{e}_1 \cap \mathbf{e}_1 + \mathbf{e}_1 \cap \mathbf{e}_2 \quad &\sim \quad \mathbf{e}_1 + \alpha, \end{aligned}$$

with scalar $\alpha \neq 0$.

2.13.1 Basic Properties of the `meet` and `join`

Suppose we have two blades $\mathbf{A}$ and $\mathbf{B}$ and factor them orthogonally as follows:

$$\begin{aligned} \mathbf{A} &= \mathbf{A}^r\,\mathbf{C} \\ \mathbf{B} &= \mathbf{C}\,\mathbf{B}^r. \end{aligned} \tag{2.45}$$

Here $\mathbf{C}$ is the common factor of $\mathbf{A}$ and $\mathbf{B}$, and $\mathbf{A}^r$ and $\mathbf{B}^r$ are 'the remainders' of $\mathbf{A}$ and $\mathbf{B}$, respectively (we use superscript indices to avoid confusion with grade part selection). Given this factorization, it is easy to see that

$$\begin{aligned} \mathbf{A} \cap \mathbf{B} \quad &\sim \quad \mathbf{C}, \\ \mathbf{A} \cup \mathbf{B} \quad &\sim \quad (\mathbf{A}^r\,\mathbf{C}) \wedge \mathbf{B}^r. \end{aligned}$$

However, this insight does not help with computing the `meet` and `join`, since the common factor $\mathbf{C}$ is just the `meet`. Computing the `meet` and `join` is hinted at in the section below, and treated thoroughly in Section 3.4.7 and 5.2.10.

Even though `meet` and `join` do not have not particular scale, we can enforce a relationship between them such that the `meet` and `join` of the same pair of blades is consistent with the factorization 2.45, as in:

$$\mathbf{J} = \mathbf{A} \cup \mathbf{B} \quad = \quad \mathbf{A} \wedge (\mathbf{M}^{-1} \rfloor \mathbf{B}), \tag{2.46}$$

$$\mathbf{M} = \mathbf{A} \cap \mathbf{B} \quad = \quad (\mathbf{B} \rfloor \mathbf{J}^{-1}) \rfloor \mathbf{A}. \tag{2.47}$$

2.13.2 The Delta Product

Computation of the `meet` and `join` is non-trivial in the additive representation. We use a custom algorithm as described in Section 3.4.7. This algorithm uses a special (non-linear) product called the *delta product* [6]. In set-theoretic terminology, it computes the symmetric difference of two blades.

The delta product is defined as the highest non-zero grade part *max* of a geometric product:

$$\mathbf{A} \triangle \mathbf{B} = \langle \mathbf{A}\,\mathbf{B} \rangle_{\mathrm{max}}. \tag{2.48}$$

Thus — with blades $\mathbf{A}$ and $\mathbf{B}$ factorized as above — the delta product behaves as follows

$$\mathbf{A} \triangle \mathbf{B} = \langle \mathbf{A}^r\,\mathbf{C}\,\mathbf{C}\,\mathbf{B}^r \rangle_{\mathrm{max}} = \langle (\mathbf{C}\cdot\mathbf{C})\mathbf{A}^r\,\mathbf{B}^r \rangle_{\mathrm{max}} = (\mathbf{C}\cdot\mathbf{C})\mathbf{A}^r \wedge \mathbf{B}^r, \tag{2.49}$$

where $\mathrm{max} = \mathrm{grade}(\mathbf{A}^r) + \mathrm{grade}(\mathbf{B}^r)$.

2.14 Executive Summary

2.14.1 Structure

Geometric algebra brings structure to geometry, for mathematicians and programmers alike. It does so by allowing many more geometric objects and transformations to have a *direct representation* in the algebra than classic approaches do, and by allowing these to interact in universal ways. By 'direct representation' we mean that a single geometric concept (e.g., a line) can be represented by a single element in the algebra, as opposed to classic approaches, which often use assemblies of multiple elements to represent these concepts (e.g., a line is the intersection of two planes). By 'universal' we mean that equations often work on many different types of objects or transformations.

2.14.2 How Geometric Algebra Does This

Geometric algebra achieves this by making subspaces and (orthogonal) transformations into elements of computation. Orthogonal transformations are represented by *versors* which are created using the linear, invertible *geometric product*. The geometric product is the fundamental product of geometric algebra, since other useful products can be derived from it. One of them is the *outer product*, which is used to create *blades*. Blades in turn represent the subspaces or objects. Both blades and versors are *multivectors*, which is what general elements of the algebra are called.

2.14.3 Blades and the Outer Product

If $\mathbf{a}$ and $\mathbf{b}$ are vectors , then

$$\mathbf{S} = \mathbf{a} \wedge \mathbf{b}$$

is the *2-blade* which represents the oriented subspace which contains $\mathbf{a}$ and $\mathbf{b}$. We say that $\mathbf{S}$ is a blade of *grade* 2 because it spans a two-dimensional subspace. We define that vectors are 1-blades, and scalars are 0-blades. The outer product of three vectors results in a blade of grade 3, and so on.

The outer product of dependent vectors is always 0, hence in an n-dimensional space, there are no grade $n+1$ blades. The (unit) top-grade blade of an algebra is called the *pseudoscalar* and is often denoted as $\mathbf{I}$. The definition of the outer product is given in Section 2.4.1.

Grade Part Selection

To indicate that a blade $\mathbf{A}$ is of grade k, we use the notation $\mathbf{A}_k$. One may add blades of different grades, so some elements of the algebra may consist of multiple grade parts. *Grade part selection* may be used to extract a particular grade part. The notation to extract grade k from a multivector $\mathbf{A}$ is $\langle \mathbf{A} \rangle_k$.

2.14.4 Versors and the Geometric Product

The geometric product is denoted by a half space. It has a surprisingly simple definition, which is given in Section 2.6.2. The outcome of a geometric product of vectors is the sum of a scalar and a 2-blade:

$$\mathbf{a}\,\mathbf{b} = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \wedge \mathbf{b}.$$

The geometric product may be used to reflect vectors in vectors. For example,

$$\mathbf{a}\,\mathbf{x}\,\mathbf{a}^{-1}$$

reflects vector $\mathbf{x}$ in $\mathbf{a}$. Once reflections can be handled like this, any orthogonal transformation can be. For example, the following double reflection (first in $\mathbf{a}$, then in $\mathbf{b}$)

$$\mathbf{b}\,\mathbf{a}\,\mathbf{x}\,\mathbf{a}^{-1}\mathbf{b}^{-1}$$

rotates $\mathbf{x}$ over twice the angle between $\mathbf{a}$ and $\mathbf{b}$ (see Section 2.9.2). If we define $\mathbf{V} = \mathbf{b}\,\mathbf{a}$, then $\mathbf{V}$ is called a *versor* (this particular rotation versor is traditionally known as a quaternion). In general, a versor is the geometric product of any number of invertible vectors. When we write $\mathbf{V}\mathbf{X}\mathbf{V}^{-1}$, we say that we *apply* versor $\mathbf{V}$ to blade $\mathbf{X}$. We note that invertible blades are versors, too, so they can be applied to other blades in the same manner.

Versors can be applied to blades of any grade, for example to a 3-blade $\mathbf{x}\wedge\mathbf{y}\wedge\mathbf{z}$:

$$\mathbf{V}\,(\mathbf{x} \wedge \mathbf{y} \wedge \mathbf{z})\,\mathbf{V}^{-1} = (\mathbf{V}\,\mathbf{x}\,\mathbf{V}^{-1}) \wedge (\mathbf{V}\,\mathbf{y}\,\mathbf{V}^{-1}) \wedge (\mathbf{V}\,\mathbf{z}\,\mathbf{V}^{-1}).$$

This equation shows that applying a versor to a blade is the same as applying the versor to each of the factors of the blade. This type of equation is one of the key powers of geometric algebra, as it allows for easy manipulation with transformations.

2.14.5 Interpretation of the Algebra and the Conformal Model

To do anything useful with geometric algebra, blades and versors must be interpreted in such a way that they actually represent geometric concepts. The best example of this is the conformal model, which is introduced in Section 2.11.

The conformal model embeds n-dimensional space in an $n+2$-dimensional geometric algebra with a Minkowski metric. In this setup, null vectors are points $\mathbf{p}$, and the outer product of points represents objects like lines ($\mathbf{p}_1 \wedge \mathbf{p}_2 \wedge \infty$), circles ($\mathbf{p}_1 \wedge \mathbf{p}_2 \wedge \mathbf{p}_3$), planes ($\mathbf{p}_1 \wedge \mathbf{p}_2 \wedge \mathbf{p}_3 \wedge \infty$) and spheres ($\mathbf{p}_1 \wedge \mathbf{p}_2 \wedge \mathbf{p}_3 \wedge \mathbf{p}_4$). The ∞ symbol is a special null vector. It is interpreted as the point at infinity.

Because of the metric of the conformal model, it is possible to reflect points (and hence all other objects) in planes and spheres using the 'versor-application-equation' $\mathbf{V}\,\mathbf{X}\,\mathbf{V}^{-1}$. Reflections in planes and spheres are the building blocks of all conformal (angle preserving) transformations, hence the conformal model can represent transformations like translation, rotation, reflection and dilation (scaling) as versors. Those transformations that can be done with infinitesimal steps have logarithms, which is useful for interpolation.

2.14.6 Products Derived from the Geometric Product

The outer product and several *metric products* can be derived from the geometric product, using grade part selection:

$$
\begin{array}{rrcl}
\text{Outer Product} & \mathbf{A}_k \wedge \mathbf{B}_l & = & \langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{l+k} \\
\text{Left Contraction} & \mathbf{A}_k \rfloor \mathbf{B}_l & = & \langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{l-k} \\
\text{Right Contraction} & \mathbf{A}_k \lfloor \mathbf{B}_l & = & \langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{k-l} \\
\text{Scalar Product} & \mathbf{A}_k * \mathbf{B}_l & = & \langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{0}
\end{array}
$$

This is described in Section 2.7. The metric products are used to measure relations of blades. Examples are the norm of a blade, the angle between two blades, or their intersection. The geometric algebra community has not yet settled on what specific metric products to use, and they are all rather complementary. In this thesis, we mainly use the left contraction ($\rfloor$), though we sometimes write this as a dot product when only vectors are involved.

2.14.7 Dualization

The *dual* of a blade is its orthogonal complement. It is always computed with respect to some other blade, usually the pseudoscalar $\mathbf{I}$:

$$\mathbf{A}^{\star} = \mathbf{A}\,\mathbf{I}^{-1}.$$

2.14.8 Reversion, Inversion

Reversing the order of factors of a blade (or versor) is called reversion:

$$(\mathbf{x} \wedge \mathbf{y} \wedge \mathbf{z})^{\sim} = \mathbf{z} \wedge \mathbf{y} \wedge \mathbf{x}$$

For blades, this amounts to a mere change of sign , i.e.,

$$(\mathbf{x} \wedge \mathbf{y} \wedge \mathbf{z})^{\sim} = -\mathbf{x} \wedge \mathbf{y} \wedge \mathbf{z}$$

or, in other words, a change of orientation. Since the geometric product of a versor with its reverse is a scalar

$$\widetilde{\mathbf{V}}\,\mathbf{V} = \mathbf{V}\,\widetilde{\mathbf{V}} = (\mathbf{v}_1\,\mathbf{v}_2\,\ldots\,\mathbf{v}_k)\,(\mathbf{v}_k\,\ldots\,\mathbf{v}_2\,\mathbf{v}_1) = \mathbf{v}_k \cdot \mathbf{v}_k\,\ldots\,\mathbf{v}_2 \cdot \mathbf{v}_2\,\mathbf{v}_1 \cdot \mathbf{v}_1,$$

reversion may be used to compute the inverse of those:

$$\mathbf{V}^{-1} = \frac{\widetilde{\mathbf{V}}}{\mathbf{V}\,\widetilde{\mathbf{V}}}.$$

2.14.9 Multivectors

The general elements of the algebra, obtained as the sum of arbitrary blades, are called *multivectors*. Many multivectors do not have a geometric interpretation and as such are not very useful for doing geometry. In general, only the blades and versors are useful for geometry. Figure 2.3 shows a Venn diagram of special types of multivectors we identify in this thesis.

2.14.10 Basis

Any multivector can be written as the sum of (basis) blades. We can use this to span a basis for all multivectors in a particular algebra. For example, for $n = 3$ we could use the basis

$$\Big\{ \underbrace{1}_{\textit{scalars}}, \underbrace{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3}_{\textit{vector space}}, \underbrace{\mathbf{e}_1 \wedge \mathbf{e}_2,\ \mathbf{e}_2 \wedge \mathbf{e}_3,\ \mathbf{e}_3 \wedge \mathbf{e}_1}_{\textit{2-blades}}, \underbrace{\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3}_{\textit{3-blades}} \Big\}.$$

The basis vectors are called $\mathbf{e}_i$ by convention, but in some models different symbols are used for special basis vectors. Examples are o and ∞ in the conformal model, which stand for the origin and infinity, respectively.

In the above example, there is 1 scalar, 3 basis vectors, 3 basis 2-blades and 1 basis 3-blade. In general, 2^n coordinates are required to represent an arbitrary multivector in n-D space.

2.14.11 Outermorphisms

Grade preserving linear transformations are *outermorphisms*. Examples of outermorphisms are rotation, intersection and projection. As with versors, this means that applying the outermorphism O to some blade $\mathbf{x} \wedge \mathbf{y} \wedge \mathbf{z}$ gives the same result as applying it to each of the factors, for example:

$$\mathsf{O}(\mathbf{x} \wedge \mathbf{y} \wedge \mathbf{z}) = \mathsf{O}(\mathbf{x}) \wedge \mathsf{O}(\mathbf{y}) \wedge \mathsf{O}(\mathbf{z}).$$

This is another example of the universal nature of equations in geometric algebra.

When using an additive basis, it is (computationally) useful to 'summarize' an outermorphism into a matrix representation. One matrix is computed for each grade. This matrix specifies the image of each of the basis blades of that grade. The matrix representation can be used to apply the outermorphism to blades using regular matrix multiplication, and this is usually more efficient than applying the outermorphism using regular geometric algebra expressions. See section 2.12.

2.14.12 Non-Linear Products

Two important non-linear products are the `meet` and the `join` (Section 2.13). They compute the subspace (blade) intersection and union, respectively, and hence we use the cap $\cap$ and cup $\cup$ symbols to denote these products.

Because they are non-linear, they must be computed using specialized algorithms (Section 3.4.7).

Chapter 3

Implementation of Geometric Algebra on an Additive Basis

In this chapter we describe the *additive approach* to implementing geometric algebra. We call it the additive approach because multivectors are represented as a sum of basis blades.

In this representation, any multivector $\mathbf{A}$ from an n-dimensional geometric algebra is represented as a column vector $[\mathbf{A}]$ with 2^n elements, containing the coefficients of $\mathbf{A}$ on the basis of blades. $\mathbf{A}$ can be retrieved from the column vector $[\mathbf{A}]$ by multiplying it with a symbolic row vector containing the basis elements. In a typical 3-D example, this would give:

$$\mathbf{A} = [1, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_1 \wedge \mathbf{e}_2, \mathbf{e}_2 \wedge \mathbf{e}_3, \mathbf{e}_1 \wedge \mathbf{e}_3, \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3]\,[\mathbf{A}].$$

From this starting point, the rest of the implementation follows.

Most products and operations are fairly simple to implement for basis blades, and once we have done so, we can employ distributivity and linearity to implement those operations for arbitrary multivectors.

However, some operations are non-linear, and we will have to design special algorithms to implement them. Examples are computing the `meet` and `join`, and inversion. This cuts our implementation neatly into three levels, and this chapter is structured accordingly:

1. Selecting the basis and basis blades, and implementing the basic operations on them.
2. Implementing linear operations for multivectors.
3. Implementing non-linear operations on multivectors.

We describe each of these levels in this chapter. The actual use of geometric algebra in an application could be considered the fourth level, and we present a case study in Chapter 6.

We have implemented the algorithms of this chapter in a 'reference implementation', the source code of which may be inspected at (and downloaded from) our website `http://www.geometricalgebra.net`, [11]. The code fragments which are shown in this chapter are excerpts from this implementation. Due to its simplicity, the reference implementation is extremely inefficient. It should be seen as purely educational. Chapter 4 describes our efficient `Gaigen 2` implementation, which is intended for use in actual applications.

3.0.1 A Note on Time Complexity

For the algorithms in Section 3.3 and 3.4 we give the order of the time complexity. However, realistically bounding the time complexity is hard because it depends on what type of element is being processed. For example, the geometric product of two versors may involve many more operations than the geometric product of blades. This difference is caused by the number of coordinates that are zero. To simplify the estimation we make the following assumptions:

- When the operands of an operation are known to be multivectors, we assume they have 2^n non-zero coordinates.
- When operands are versors, we assume they have 2^{n-1} non-zero coordinates.
- When operands are known to be blades, we assume we know their respective grades k and that they have $\frac{n!}{k!(n-k)!}$ non-zero coordinates.

These assumptions will give worst-case time complexities.

However, we believe the time complexities are actually not that valuable, as the number of coordinates is in the order of $O(2^n)$ anyway. This limits the usefulness of the additive implementation to low dimensional spaces, since the multiplicative implementation of Chapter 5 requires only $O(n^2)$ coordinates and is more efficient in high-dimensional spaces. In low dimensional spaces, the constant factor of the time complexity plays an important role, and we will see in the Chapter 4 that optimizing the additive implementation method is more about fine tuning the architecture of the implementation than about theoretical time complexities: bad design choices can easily make an implementation 100 times slower than 'optimal', regardless of the theoretical time complexity of the algorithms used.

Another consideration is the cost of processor instructions, and what to optimize for. Traditionally scalar multiplication and scalar division took many clock cycles to compute and a lot of effort was put into reducing the *multiplicative* complexity of algorithms. On present-day processors multiplication is as cheap as addition. Instead one should avoid (unpredictable) memory access and (unpredictable) conditional jumps and allow for parallel (SIMD) computation of addition and multiplication. This is due to the relatively low speed of memory compared to the processor, and due to the (deep) pipelining of the processor. As a result, classic research on multiplicative complexity of bilinear forms [9] is of less use to optimizing low-dimensional geometric algebra implementations, because it focuses on reducing the number of multiplications (often at the cost of many more additions, which are now just as expensive as multiplications). This is visible in our benchmarks, too: avoiding unpredictable memory access and jumps can save you a factor of 50 (see for example Sections 7.4.2 and 7.4.4), while avoiding multiplications and additions can save you a factor of three (this number is for a contrived example, see for Section 7.3.2). That is not to say that applying theory developed for reducing multiplicative complexity of bilinear forms should not be considered. We include it as a suggestion for future research at the end of Chapter 4.

3.1 The Basis and Operations on Basis Blades

In this section we show how to represent basis blades, and how to implement the basic geometric algebra operations for them. We illustrate this with an occasional

Java source code listing, which we found to be the simplest way to precisely describe the algorithms.

The functions described in this section — such as those in Figure 3.1 and 3.2 — are are typically not used at run-time. Most implementations will instead pre-compute the outcome of these functions for all (combinations of) basis blades, and store the results in a table.

3.1.1 A Basis of Blades

We showed in Section 2.4.8 that from every set of n basis vectors $\mathbf{e}_i$ automatically follows a basis of blades that span the 2^n dimensional linear multivector space. To obtain these basis blades, we simply take every combination of basis vectors (including none).

We are free to pick the orientation of basis vectors in each basis blade — in most spaces there is little rationale for picking one order over the other. For example in 3-D, we can pick the grade 2 such that they are cyclic and dual with respect to the basis vectors, as in

$$\Big\{ \underbrace{1}_{\textit{scalars},} \quad \underbrace{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3,}_{\textit{vector space}} \ \underbrace{\mathbf{e}_2 \wedge \mathbf{e}_3,\ \mathbf{e}_3 \wedge \mathbf{e}_1,\ \mathbf{e}_1 \wedge \mathbf{e}_2,}_{\textit{bivector space}} \ \underbrace{\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3}_{\textit{trivector space}} \Big\}.$$

but this type of ordering does not generalize to higher dimensions.

Below we use a *canonical order* for the basis vectors in a basis blades. We assume the basis vectors are ordered from low index to high index, so, for example, $\mathbf{e}_1 \wedge \mathbf{e}_3$ and not $\mathbf{e}_3 \wedge \mathbf{e}_1$. This greatly simplifies the description of algorithms.

3.1.2 The Bitmap Representation of Basis Blades

To compute with basis blades, we need some way of representing them. We use the compact *bitmap representation* that also works out well for algorithms that compute the linear operations of geometric algebra. This method — suggested by Bouma [5] — was developed for `Gaigen 1` and extended for `Gaigen 2`. The bitmap representation itself is probably not new although we could not find any references to it earlier than `Gaigen 1`. What *is* new is automated handling of non-orthogonal non-Euclidean metrics.

A basis blade is a non-zero outer product of a number of basis vectors, or a unit scalar. Hence a basis blade either contains a specific basis vector or not. Each basis blade in an algebra can be represented by a list of boolean values, where each boolean indicates the presence or absence of the respective basis vector. The list of booleans is naturally implemented using a bitmap, which is why we call this the bitmap representation of basis blades.

The bits in the bitmap are assigned in the logical order: bit 0 stands for $\mathbf{e}_1$, bit 1 stands for $\mathbf{e}_2$, bit 2 stands for $\mathbf{e}_3$, and so on. Figure 3.1.2 shows how this works out. The unit scalar does not contain any basis vectors, so it is represented by `0`.

basis blade	bitmap representation
1	0
$\mathbf{e}_1$	1
$\mathbf{e}_2$	10
$\mathbf{e}_1 \wedge \mathbf{e}_2$	11
$\mathbf{e}_3$	100
$\mathbf{e}_1 \wedge \mathbf{e}_3$	101
$\mathbf{e}_2 \wedge \mathbf{e}_3$	110
$\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$	111
$\mathbf{e}_4$	1000
...	...

Table 3.1: *The bitmap representation of basis blades. Note how the representation scales up with the dimension of the algebra.*

A blade like $\mathbf{e}_1 \wedge \mathbf{e}_3$ contains $\mathbf{e}_1$ and $\mathbf{e}_3$, so it is represented by $\mathsf{001} + \mathsf{100} = \mathsf{101}$. (In this chapter, we use a sans serif font for binary numbers).

Visually, the bits are stored in reversed order relative to how we would write them as basis blades. In a basis blade, $\mathbf{e}_1$ is the left-most basis vector, while in bitmap representation it is the right-most bit. While this is slightly inconvenient for reasoning about them, it is more consistent for implementation: if an extra dimension is required, the next most significant bit in the bitmap is used and previous results are automatically absorbed.

To represent weighted basis blades, one would accompany the bitmap with a floating point number. This combination of the bitmap and the floating point weight is implemented in the `BasisBlade` class of the reference implementation.

3.1.3 Notation of Bitwise Boolean Operations

In the following, we describe the essential algorithms for basis blades that act on the bitmap representation, using bitwise boolean operations. The algorithms are described easily in `Java` source code. The `Java` operators we use are:

symbol	operation
&	bitwise boolean 'and'
^	bitwise boolean 'exclusive or'
>>>	unsigned bitwise 'shift right'

3.1.4 The Outer Product of Basis Blades

To compute the outer product of two basis blades in the bitmap representation, we first check whether they are dependent. That is, we check if the two basis blades have a common factor. If they do, their bitmaps have at least one bit

in common, so dependence of basis blades is checked simply with a binary *and*. When the basis blades are dependent, the outcome of the outer product is 0 and the algorithm described below does not have to be executed.

On the other hand, if the basis blades are independent, their outer product is computed (up to a scalar factor) as the bitwise *exclusive or* of the bitmaps. For example,

$$(\mathbf{e}_2 \wedge \mathbf{e}_3) \wedge \mathbf{e}_1 = \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$$

is computed as

$$\mathsf{110} \wedge \mathsf{001} = \mathsf{111}.$$

The hard part in implementing the outer product is computing the correct sign for the result. The bitmap representation assumes that the basis vectors are in canonical order, so we should compute a scalar multiplier to get the correct orientation. E.g., we represent $\mathbf{e}_3 \wedge \mathbf{e}_1$ as $-1.0 * (\mathbf{e}_1 \wedge \mathbf{e}_3)$.

When one computes the sign of the outer product of basis blades by hand, it is standard procedure to count how many basis vectors have to swap positions to get the result into the required order. Due to anti-commutativity, each swap of position causes a flip of sign of the result. In our example, all that is required is that $\mathbf{e}_3$ and $\mathbf{e}_1$ swap positions, so we get a negative sign.

To compute this sign algorithmically, we have to compute, for each 1-bit in the first operand, the number of less significant 1-bits in the second operand. This is implemented somewhat like a convolution, in that the bitmaps are slid over each other bit by bit. For each position, the number of matching 1-bits counted. Figure 3.1 shows the implementation of this idea.

The implementation of the outer product itself is in Figure 3.2. This function also implements the geometric product, depending on the value of the boolean `outer` argument. Computing the geometric product is described in the next section.

3.1.5 The Geometric Product of Basis Blades in an Orthogonal Metric

The implementation of the geometric product is similar to that of the outer product as long as we stay in an orthogonal metric with respect to the orthonormal basis $\{\mathbf{e}_i\}_{i=1}^n$. Such a metric has $\mathbf{e}_i \cdot \mathbf{e}_j = 0$ for $i \neq j$, and $\mathbf{e}_i \cdot \mathbf{e}_i = m_i$, that is, its metric matrix is diagonal:

$$\mathbf{e}_i \cdot \mathbf{e}_j = m_i \, \delta^i_j, \tag{3.1}$$

where δ^i_j is the *Kronecker delta function*, returning 1 when i equals j, and zero otherwise. A particular example is the Euclidean metric with $m_i = 1$, for all i, so that the metric matrix is diagonal. In general, the m_i can have any real value, but -1, 0 and 1 will be the most prevalent in many applications.

There are now two cases:

```
// Computes 'reordering sign' to get into canonical order.
// Arguments 'a' and 'b' are both bitmaps representing basis blades.
double canonicalReorderingSign(int a, int b)
{
  // Count the number of basis vector swaps required to
  // get 'a' and 'b' into canonical order.
  a = a >>> 1;
  int sum = 0;
  while (a != 0)
  {
    // the function bitCount() counts the number of
    // 1-bits in the argument
    sum = sum + subspace.util.Bits.bitCount(a & b);
    a = a >>> 1;
  }

  // even number of swaps -> return 1
  // odd number of swaps -> return -1
  return ((sum & 1) == 0) ? 1.0 : -1.0;
}
```

Figure 3.1: *This function computes the sign change due to the reordering of two basis blades into canonical order.*

- For basis blades consisting of *different* orthogonal factors, we have the usual equivalence of the outer product and the geometric product:

$$\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \cdots \wedge \mathbf{e}_k = \mathbf{e}_1\,\mathbf{e}_2\,\cdots\,\mathbf{e}_k.$$

- However, when two factors are in common between the multiplicands, the outer product leads to a zero result, whereas the geometric product does not. Instead, it annihilates the dependent basis vectors, effectively replacing them by metric factors. For example,

$$(\mathbf{e}_1 \wedge \mathbf{e}_2)\,(\mathbf{e}_2 \wedge \mathbf{e}_3) = m_2\,\mathbf{e}_1 \wedge \mathbf{e}_3.$$

In the bitmap representation, these two cases are easily merged. Both results may be computed as the bitwise *exclusive or* operation (i.e., $\mathtt{011} \wedge \mathtt{110} = \mathtt{101}$). In this sense, the geometric product acts as a 'spatial exclusive or' on basis blades.

As for the outer product, the result of the geometric product 'exclusive or' should be given the correct sign, to distinguish between the outcomes $\mathbf{e}_1 \wedge \mathbf{e}_3$ and $\mathbf{e}_3 \wedge \mathbf{e}_1$. We therefore also have to perform reordering techniques to establish the sign of the result on the standard basis. Those sign changes are identical to those for the outer product: we count the number of basis vector swaps required to get the result into canonical order and use this to determine the sign.

In a Euclidean metric, all diagonal metric factors are 1, so this bit pattern and sign computation is all there is to the geometric product. The function that

```
// computes the geometric and outer product of basis blades
BasisBlade gp_op(BasisBlade a, BasisBlade b, boolean outer)
{
  // if outer product: check for independence:
  if (outer && ((a.bitmap & b.bitmap) != 0))
    return new BasisBlade(0.0);

  // compute the bitmap:
  int bitmap = a.bitmap ^ b.bitmap;

  // compute the sign change due to reordering:
  double sign = canonicalReorderingSign(a.bitmap, b.bitmap);

  // return result:
  return new BasisBlade(bitmap, sign * a.scale * b.scale);
}
```

Figure 3.2: *This function will compute either the geometric product (in Euclidean metric only) or the outer product, depending on the value of argument* `outer`*.*

computes the geometric product in this simple Euclidean case is therefore virtually the same as the function for computing the outer product (see Figure 3.2), the only difference being the dependence check required for the outer product.

We can easily extend the function in Figure 3.2 to non-Euclidean metrics, as long as the metric matrix is diagonal. We simply need to incorporate the metric coefficients of the annihilated basis vectors into the scale of the resulting blade. Which basis vectors are annihilated is determined with a bitwise *and*. The function which implements this is shown in Figure 3.3.

3.1.6 The Geometric Product of Basis Blades in non- Orthogonal Metrics

In practical geometric algebra we naturally encounter non-orthonormal bases. For example, in the conformal model we may want to represent o and ∞ directly as basis vectors. This leads to a non-diagonal multiplication table and therefore non-diagonal metric matrix:

	o	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_3$	∞
o	0	0	0	0	−1
$\mathbf{e}_1$	0	1	0	0	0
$\mathbf{e}_2$	0	0	1	0	0
$\mathbf{e}_3$	0	0	0	1	0
∞	−1	0	0	0	0

The orthogonal-metric-method of the previous section does not seem to apply to this case. Yet the methods employed for the orthogonal metrics are more

```
// Computes the geometric product of two basis blades in limited non-Euclidean metric.
// 'm' is an array of doubles giving the metric for each basis vector.
public static BasisBlade gp(BasisBlade a, BasisBlade b, double[] m)
{
  // compute the geometric product in Euclidean metric:
  BasisBlade result = gp_op(a, b, false);

  // compute the meet (bitmap of annihilated vectors):
  int mt = a.bitmap & b.bitmap;

  // change the scale according to the metric:
  int i = 0;
  while (mt != 0) {
    if ((mt & 1) != 0) result.scale *= m[i];
    i++;
    mt = mt >>> 1;
  }

  return result;
}
```

Figure 3.3: *This function computes the geometric product, taking into account the metric as specified by* `m`*, which is the diagonal of the metric matrix.*

general than they seem. We already discussed in Section 2.5.2 how a matrix can always be brought into diagonal form by an appropriate orthogonal coordinate transformation.

Therefore we can compute geometric products in the conformal model by temporarily switching to a new basis that is orthonormal. Such a basis is computed by finding the eigenvalue decomposition of the metric matrix. For our conformal model example, such a basis would be:

$$\begin{aligned}
\mathbf{f}_1 &= \mathbf{e}_1 \\
\mathbf{f}_2 &= \mathbf{e}_2 \\
\mathbf{f}_3 &= \mathbf{e}_3 \\
\mathbf{f}_4 &= \tfrac{1}{2}\sqrt{2}(o - \infty) \\
\mathbf{f}_5 &= \tfrac{1}{2}\sqrt{2}(o + \infty).
\end{aligned}$$

The $\mathbf{f}_i$ are all basis vectors with $\mathbf{f}_i \cdot \mathbf{f}_i = 1$, except for $\mathbf{f}_5 \cdot \mathbf{f}_5 = -1$. The new basis is therefore one of the orthogonal metrics of the previous section, and we can revert to the previous methods. So, to compute the geometric product in arbitrary metric:

1. Compute the eigenvectors and eigenvalues of the metric matrix. This has to be done only once, when the object which represents the metric is initialized.

2. Apply a change-of-basis to the input such that it is represented with respect to the eigenbasis.
3. Compute the geometric product on this new basis (the eigenvalues specify the metric).
4. Apply another change of basis to the result, to get back to the original basis.

The code for this algorithm is rather involved, so we do not show the implementation here. Interested readers may find it it in the `subspace.basis` package [11].

One detail to note is that, in general, evaluating the geometric product of two basis blades does not result in a single basis blade. When switching back and forth between one basis and another, a single basis blade can convert into a sum of multiple basis blades. This is in agreement with what one would expect. An example is easily given: in the conformal model, $o\,\infty = -1 + o \wedge \infty$.

3.1.7 The Metric Products of Basis Blades

Several metric products are in use in geometric algebra. In this thesis we use mainly the left contraction, but the other products are similar, from an implementational viewpoint. They can all be derived from the geometric product by extracting the appropriate grade parts of the result (the same could be be done for the outer product, if you wish). Here we give the precise rules to do so.

Let $\mathbf{A}$ and $\mathbf{B}$ be two basis blades of grade a and b, respectively. Then the rules for deriving a particular metric product from the geometric product of two basis blades are:

- Left contraction: If $a \leq b$, $\mathbf{A} \rfloor \mathbf{B} = \langle \mathbf{A}\,\mathbf{B} \rangle_{b-a}$, otherwise 0.
- Right contraction: If $a \geq b$, $\mathbf{A} \lfloor \mathbf{B} = \langle \mathbf{A}\,\mathbf{B} \rangle_{a-b}$, otherwise 0.
- Dot product: If $a \leq b$, then the dot product is equal to the left contraction. Otherwise it is equal to the right contraction. (In our reference implementation, we call this product the Modified Hestenes inner product)
- Hestenes inner product: like the dot product, but 0 when either $\mathbf{A}$ or $\mathbf{B}$ is a scalar.
- scalar product: $\mathbf{A} * \mathbf{B} = \langle \mathbf{A}\,\mathbf{B} \rangle_0$.

The grade of a basis blade is determined simply by counting the number of 1-bits in the bitmap. We can therefore implement the inner products exactly as defined. For example, to compute the left contraction of a grade-1 blade and a grade-3 blade, it extracts the grade $3 - 1 = 2$ part from their geometric product.

If one were to write a dedicated version of the inner product implementation, an optimization would be to check whether one lower-grade basis blade is fully

contained in the higher-grade basis blade, before performing the actual product. If this condition is not satisfied, the inner product (of whatever flavor) will always be 0. This containment is tested for by checking whether all set bits in the lower-grade blade are also set in the higher-grade blade. For non-orthogonal metrics this check should be done after the change to the eigenbasis.

3.1.8 Grade Dependent Sign Operators on Basis Blades

Implementing the reversion and grade involution (see Section 2.4.6) is straightforward for basis blades. These 'grade dependent sign operators' toggle the sign of basis blades according to a specific multiplier based on the grade k of the blade:

operation	multiplier for grade k	pattern
reversion:	$(-1)^{k(k-1)/2}$	$++--++--$
grade involution:	$(-1)^k$	$+-+-+-+-$

As an example, the reverse of $\mathbf{e}_1 \wedge \mathbf{e}_2$ is computed by first determining the grade (which is 2) and then applying the correct multiplier (in this case $(-1)^{2(2+1)/2} = -1$) to the `scale` of the blade.

The last column of the table shows the repetitive 'pattern' over the varying grades that describes the behavior of the each operation: '$-$' stands for multiplication by -1 while '$+$' stands for multiplication with $+1$. It shows for instance that the grade involution toggles the sign of all odd grade parts while it leaves the even grade parts unchanged.

3.2 Representing Multivectors and Coordinate Compression

Given the basis, representing multivectors is straightforward in principle. For example, a multivector can be represented by a vector of 2^n coordinates, or as a list of weighted basis blades.

An important point to take into consideration when deciding how to represent multivectors, is that they are inherently 'sparse': versors are either even or odd and a blade has only one non-zero grade part, by definition. So for all geometrically useful multivectors, at least half the coordinates are zero.

Naively handling all of these 'zero coordinates' wastes memory and processing time. Coordinate compression should be used to avoid this. Coordinate compression reduces the number of zero coordinates that are stored and processed.

Several types of compressions are possible, and which type of compression is used in an implementation is one of the most important factors that determines performance of an implementation. Some alternatives are:

- *No Compression.* Simply store all coordinates of a multivector, even though many of those coordinates are 0. This is very inefficient, it can easily be 10 times slower than an optimal implementation.

- *Per-Coordinate Compression.* The most straightforward compression method is per-coordinate compression. Only the non-zero coordinates are stored, and each coordinate is tagged with the basis blade it refers to. Our reference implementation uses this type of compression.

- *Per-Grade Compression.* As we stated above, multivectors are typically sparse in a per-grade fashion. This suggests grouping coordinates that belong to the same grade part: when a grade part is not used by a multivector, the entire group of coordinates for that grade is not stored. Instead of signaling the presence of each individual coordinate, we signal the presence of the entire group.

  For low-dimensional spaces, per-grade compression is a good balance between per-coordinate compression and no compression at all. Per-coordinate compression is slow because each coordinate has to be processed individually, leading to excessive conditional branching, which slows down modern processors. Using no compression at all is slow because lots of zero coordinates are processed needlessly. Per-grade compression reduces the number of zero coordinates, while still allowing sizable groups of coordinates to be processed without conditional branching.

- *Per-Group Compression.* Per-grade compression misses out on some significant compression opportunities. For example, a flat point in the conformal model (a 2-blade) requires only four coordinates, while the grade 2 part of a general 5-D multivectors requires ten coordinates. Thus per-grade compression stores at least six zero coordinates for each flat point. As the dimension and structure of the algebra increases, this storage of zero coordinates becomes more of an issue. A more refined grouping method may alleviate this problem, but what the 'groups' are depends heavily on the use and interpretation of the algebra at hand. E.g., in the conformal model for Euclidean geometry, one could group coordinates based on the presence of a ∞ factor in the basis blades, in addition to grouping them based on grade; but if the conformal model would be used for hyperbolic geometry, a grouping based on the presence of an e factor might be better.

- *Type-based compression*

  A new compression method which we describe in detail in the next chapter, is *type-based.* For each multivector type (e.g., vector, line, translation versor), a distinct implementation type (class) may be created. This class allocates storage only for the coordinates which can be non-zero for its multivector type (e.g., only three coordinates for a 3-D vector). The other coordinates are assumed to be zero.

Different compressions methods may also influence what is the simplest way to implement the linear operations on multivectors. We describe two alternatives next.

3.3 Linear Operations on Multivectors

Once we have implemented the linear operations for basis blades and have selected a suitable multivector representation, it is rather trivial to extend the linear operations to multivectors through distributivity. In this section we discuss how to do so for the geometric product, the outer product and the metric products. We also discuss some examples of unary linear operations such as addition, reversion, and grade extraction

We present two ways to extend the operations from basis blades to arbitrary multivectors. The first approach uses linear algebra to encode the multiplying element as a square matrix acting on the multiplied element, which is encoded as a column matrix. We present this approach since the matrix ideas would be the standard mathematical way to implement linear products. It is used by the Matlab package GABLE [32]. However, in practice this method is not used much because it does not exploit the sparseness of most elements in geometric algebra.

The second approach literally distributes the work of computing the products and operations to the level of basis blades. This automatically employs the sparseness of geometric algebra and provides a more natural path towards the optimizations of Chapter 4. This is the approach used by our reference implementation.

3.3.1 The Linear Algebra Approach

A multivector from an n-dimensional geometric algebra can be stored as a $2^n \times 1$ column matrix . Each element in the matrix is a coordinate that refers to a specific basis blade. To formalize this, let us define a row matrix L whose elements are the basis blades of the basis, listed in order. For example, in 3-D:

$$L = [1,\ \mathbf{e}_1,\ \mathbf{e}_2,\ \mathbf{e}_3,\ \mathbf{e}_1 \wedge \mathbf{e}_2,\ \mathbf{e}_2 \wedge \mathbf{e}_3,\ \mathbf{e}_1 \wedge \mathbf{e}_3,\ \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3]$$

Using L we can represent a multivector $\mathbf{A}$ by a $2^n \times 1$ matrix $[\mathbf{A}]$ with elements:

$$\mathbf{A} = L\,[\mathbf{A}].$$

We use the following notation to indicate equivalence between the actual geometric algebra operations and their implementation in linear algebra:

$$\mathbf{A} \rightleftharpoons [\mathbf{A}].$$

Implementing the Linear Operations

The elements of geometric algebra form a linear space, and these linear operations are implemented trivially in the matrix approach:

- Addition of elements is performed by adding the matrices:

$$\mathbf{A} + \mathbf{B} \rightleftharpoons [\mathbf{A}] + [\mathbf{B}].$$

Scalar multiplication is implemented as multiplication of the matrix $[A]$ by the scalar:

$$\alpha\,\mathbf{A} \rightleftharpoons \alpha\,[\mathbf{A}].$$

- The unary linear operations of reversion and grade involution can also be implemented as matrices. The entries of these matrices need to be set according to the corresponding operations on the basis blades in L.

  For instance, for the reversion we define the (constant) diagonal matrix $[\mathsf{R}]$ as

  $$[\mathsf{R}]_{[i,i]} = (-1)^{\text{grade}(L_{[i]})(\text{grade}(L_{[i]})-1)/2}.$$

  Reversion is then implemented as

  $$\widetilde{\mathbf{A}} \rightleftharpoons [\mathsf{R}]\,[\mathbf{A}].$$

- Extracting the k-th grade part of a multivector is also a unary linear operator. For each grade we need to construct a diagonal selection matrix $[\mathsf{S}^k]$, so that the operation can be performed as the matrix operator:

  $$\langle\mathbf{A}\rangle_k \rightleftharpoons [\mathsf{S}^k]\,[\mathbf{A}].$$

  The entries of the matrix $[\mathsf{S}^k]$ must be defined as

  $$[\mathsf{S}^k]_{[i,j]} = \begin{cases} 1 & \text{if grade}(L_{[i]}) = k \text{ and } i = j \\ 0 & \text{otherwise.} \end{cases}$$

Implementing the Linear Products

The linear products can all be implemented using matrix multiplication. For if we consider the geometric product $\mathbf{A}\,\mathbf{B}$, the result is linear in $\mathbf{B}$, so $\mathbf{A}$ is like a linear operator acting on $\mathbf{B}$. In the L-based representation, that $\mathbf{A}$-operator can be represented by matrix $[\mathbf{A}^G]$ acting on column matrix $[\mathbf{B}]$ (the superscript G denotes that $\mathbf{A}$ acts on $\mathbf{B}$ by the geometric product). That gives

$$\mathbf{A}\,\mathbf{B} \rightleftharpoons [\mathbf{A}^G][\mathbf{B}].$$

Here $[\mathbf{A}^G]$ is a $2^n \times 2^n$ matrix; we need to construct it so that it corresponds to the geometric product. As we do so, we find that its entries are certain linear combinations of the coefficients of $\mathbf{A}$ on the L-basis. Hence a single matrix representation of the geometric product does not exist: each element acts through its own matrix. By the same reasoning, each element $\mathbf{A}$ also has an associated outer product matrix $[\mathbf{A}^O]$, which describes the action of the operation '$\mathbf{A}\wedge$', and a left contraction matrix $[\mathbf{A}^L]$ for the operation '$\mathbf{A}\rfloor$', and so on.

Whenever we want to compute a product, we therefore need to construct the corresponding matrix. Let us describe how that is done for the geometric product matrix $[\mathbf{A}^G]$.

$$
[\mathbf{A}^G] = \begin{bmatrix}
+\mathbf{A}_0 & +\mathbf{A}_1 & +\mathbf{A}_2 & +\mathbf{A}_3 & -\mathbf{A}_{12} & -\mathbf{A}_{23} & -\mathbf{A}_{13} & -\mathbf{A}_{123} \\
+\mathbf{A}_1 & +\mathbf{A}_0 & +\mathbf{A}_{12} & +\mathbf{A}_{13} & -\mathbf{A}_2 & -\mathbf{A}_{123} & -\mathbf{A}_3 & -\mathbf{A}_{23} \\
+\mathbf{A}_2 & -\mathbf{A}_{12} & +\mathbf{A}_0 & +\mathbf{A}_{23} & +\mathbf{A}_1 & -\mathbf{A}_3 & +\mathbf{A}_{123} & +\mathbf{A}_{13} \\
+\mathbf{A}_3 & -\mathbf{A}_{13} & -\mathbf{A}_{23} & +\mathbf{A}_0 & -\mathbf{A}_{123} & +\mathbf{A}_2 & +\mathbf{A}_1 & -\mathbf{A}_{12} \\
+\mathbf{A}_{12} & -\mathbf{A}_2 & +\mathbf{A}_1 & +\mathbf{A}_{123} & +\mathbf{A}_0 & -\mathbf{A}_{13} & +\mathbf{A}_{23} & +\mathbf{A}_3 \\
+\mathbf{A}_{23} & +\mathbf{A}_{123} & -\mathbf{A}_3 & +\mathbf{A}_2 & +\mathbf{A}_{13} & +\mathbf{A}_0 & -\mathbf{A}_{12} & +\mathbf{A}_1 \\
+\mathbf{A}_{13} & -\mathbf{A}_3 & -\mathbf{A}_{123} & +\mathbf{A}_1 & -\mathbf{A}_{23} & +\mathbf{A}_{12} & +\mathbf{A}_0 & -\mathbf{A}_2 \\
+\mathbf{A}_{123} & +\mathbf{A}_{23} & -\mathbf{A}_{13} & +\mathbf{A}_{12} & +\mathbf{A}_3 & +\mathbf{A}_1 & -\mathbf{A}_2 & +\mathbf{A}_0
\end{bmatrix}
$$

$$
[\mathbf{A}^O] = \begin{bmatrix}
+\mathbf{A}_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
+\mathbf{A}_1 & +\mathbf{A}_0 & 0 & 0 & 0 & 0 & 0 & 0 \\
+\mathbf{A}_2 & 0 & +\mathbf{A}_0 & 0 & 0 & 0 & 0 & 0 \\
+\mathbf{A}_3 & 0 & 0 & +\mathbf{A}_0 & 0 & 0 & 0 & 0 \\
+\mathbf{A}_{12} & -\mathbf{A}_2 & +\mathbf{A}_1 & 0 & +\mathbf{A}_0 & 0 & 0 & 0 \\
+\mathbf{A}_{23} & 0 & -\mathbf{A}_3 & +\mathbf{A}_2 & 0 & +\mathbf{A}_0 & 0 & 0 \\
+\mathbf{A}_{13} & -\mathbf{A}_3 & 0 & +\mathbf{A}_1 & 0 & 0 & +\mathbf{A}_0 & 0 \\
+\mathbf{A}_{123} & +\mathbf{A}_{23} & -\mathbf{A}_{13} & +\mathbf{A}_{12} & +\mathbf{A}_3 & +\mathbf{A}_1 & -\mathbf{A}_2 & +\mathbf{A}_0
\end{bmatrix}
$$

$$
[\mathbf{A}^L] = \begin{bmatrix}
+\mathbf{A}_0 & +\mathbf{A}_1 & +\mathbf{A}_2 & +\mathbf{A}_3 & -\mathbf{A}_{12} & -\mathbf{A}_{23} & -\mathbf{A}_{13} & -\mathbf{A}_{123} \\
0 & +\mathbf{A}_0 & 0 & 0 & -\mathbf{A}_2 & 0 & -\mathbf{A}_3 & -\mathbf{A}_{23} \\
0 & 0 & +\mathbf{A}_0 & 0 & +\mathbf{A}_1 & -\mathbf{A}_3 & 0 & +\mathbf{A}_{13} \\
0 & 0 & 0 & +\mathbf{A}_0 & 0 & +\mathbf{A}_2 & +\mathbf{A}_1 & -\mathbf{A}_{12} \\
0 & 0 & 0 & 0 & +\mathbf{A}_0 & 0 & 0 & +\mathbf{A}_3 \\
0 & 0 & 0 & 0 & 0 & +\mathbf{A}_0 & 0 & +\mathbf{A}_1 \\
0 & 0 & 0 & 0 & 0 & 0 & +\mathbf{A}_0 & -\mathbf{A}_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & +\mathbf{A}_0
\end{bmatrix}
$$

Figure 3.4: *Symbolic matrices for computing the geometric product ($[\mathbf{A}^G]$), outer product ($[\mathbf{A}^O]$) and left contraction ($[\mathbf{A}^L]$).* $\mathbf{A}$ *notational shorthand is used for readability. I.e.,* $\mathbf{A}_0$ *is the scalar coordinate of* $[\mathbf{A}]$, $\mathbf{A}_1$ *is the coordinate from* $[\mathbf{A}]$ *that refers to* $\mathbf{e}_1$; $\mathbf{A}_{123}$ *is the* $\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$*-coordinate, etc.*

For simplicity, we temporarily assume that the algebra has a diagonal metric matrix (e.g., a Euclidean metric). To devise a rule on how to fill in the entries of $[\mathbf{A}^G]$, we consider the geometric product of two basis blades weighted by their respective coordinates from $[\mathbf{A}]$ and $[\mathbf{B}]$. These are scalars, and they get multiplied by the geometric algebra elements from L — that is where the structure of geometric algebra enters the multiplication. The product of two such elements $L_{[k]}$ and $L_{[j]}$ is a third element $L_{[i]}$, with a scalar $s_i^{k,j}$ determined by the metric (in a Euclidean metric, $s_i^{k,j} = \pm 1$, with the minus sign occurring for instance for basis 2-blades, so these are not simply the m_j of the metric matrix in Equation 3.1).

$$L_{[k]}\, L_{[j]} = s_i^{k,j}\, L_{[i]}. \tag{3.2}$$

Think of the scalars $s_i^{k,j}$ as the 'structure coefficients' of the algebra. As their definition shows, they involve the products of basis blades (the components of L), so they can be determined efficiently by the methods of Section 3.1.

We can now specify what the product of the matrices $[\mathbf{A}]$ and $[\mathbf{B}]$ should satisfy, coefficient-by-coefficient:

$$([\mathbf{A}]_{[k]} L_{[k]})\,([\mathbf{B}]_{[j]} L_{[j]}) = s_i^{k,j} [\mathbf{A}]_{[k]} [\mathbf{B}]_{[j]}\, L_{[i]}. \tag{3.3}$$

To achieve the equivalent of this equation through matrix multiplication (that is $[\mathbf{A}^G]\,[\mathbf{B}] = [\mathbf{C}]$), it is clear that $s_i^{k,j}[\mathbf{A}]_{[k]}[\mathbf{B}]_{[j]}$ should end up in row i of $[\mathbf{C}]$, so $s_i^{k,j}[\mathbf{A}]_{[k]}$ should be on row i of $[\mathbf{A}^G]$. The fact that $s_i^{k,j}[\mathbf{A}]_{[k]}$ should combine with $[\mathbf{B}]_{[j]}$ means that $s_i^{k,j}[\mathbf{A}]_{[k]}$ should be in column j. In summary,

$$([\mathbf{A}]_{[k]} L_{[k]})\,([\mathbf{B}]_{[j]} L_{[j]}) = [\mathbf{A}]_{[k]} [\mathbf{B}]_{[j]}\, s_i^{k,j} L_{[i]} \quad \rightarrow \quad [\mathbf{A}^G]_{[i,j]} = s_i^{k,j} [\mathbf{A}]_{[k]}. \tag{3.4}$$

This final rule does not involve the symbolic basis list L anymore, yet is based fully on the geometric algebra structure it contributes. By executing this rule for all indices k, j, we obtain, by linearity, a full geometric product matrix $[\mathbf{A}^G]$. An example of the geometric product matrix $[\mathbf{A}^G]$ is shown in Figure 3.4 for a 3-D Euclidean metric.

When the metric matrix is non-diagonal, things get slightly more complicated, because the geometric product of two basis blades can result in a sum of basis blades:

$$L_{[k]}\, L_{[j]} = \sum_i s_i^{k,j}\, L_{[i]}.$$

It is clear that we should propagate this change to the computation of the matrix $[\mathbf{A}^G]$:

$$([\mathbf{A}]_{[k]} L_{[k]})\,([\mathbf{B}]_{[j]} L_{[j]}) = [\mathbf{A}]_{[k]} [\mathbf{B}]_{[j]} \sum_i s_i^{k,j} L_{[i]}. \tag{3.5}$$

Thus we have to execute the rule in Equation 3.4 for each $s_i^{k,j}[\mathbf{A}]_{[k]}$. Summarizing, the following algorithm computes the geometric product matrix for a multivector $\mathbf{A}$:

1. Initialize the matrix $[\mathbf{A}^G]$ to a $2^n \times 2^n$ null matrix.

2. Loop over all indices k, j to compute the geometric product of all basis blades as in Equation 3.2. How to compute the products of basis blades was described in Section 3.1.

3. Add each $s_i^{k,j}[\mathbf{A}]_{[k]}$ to the correct element of $[\mathbf{A}^G]$ according to the rule on the right hand side of Equation 3.4.

In principle, these product matrices need to be computed only once, to bootstrap the implementation. The results can be stored symbolically, leading to matrices such as the one shown in Figure 3.4.

After the symbolic matrices have been initialized, computing an actual product $\mathbf{C} = \mathbf{A}\,\mathbf{B}$ is reduced to creating an actual matrix $[\mathbf{A}^G]$ according to the appropriate symbolic matrix and the coordinates of $[\mathbf{A}]$, and computing $[\mathbf{C}] = [\mathbf{A}^G][\mathbf{B}]$.

The same algorithm can compute the matrix for any linear product derived from the geometric product. Figure 3.4 also shows matrices for the outer product ($[\mathbf{A}^O]$) and the left contraction ($[\mathbf{A}^L]$). Note how these matrices are mostly identical to the geometric product matrix $[\mathbf{A}^G]$, but with zeros at specific entries. The reason is of course that these products are merely selections of certain grade parts of the more encompassing geometric product.

3.3.2 The 'List of Basis Blades' Approach

Instead of representing multivectors as 2^n-vectors and using matrices to implement the products, we can alternatively represent a multivector by a list (sum) of basis blades. That is how our reference implementation works. The linear products and operations are distributive over addition, e.g., for the left contraction and the reverse we have

$$\begin{aligned}
(\mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_n) \rfloor (\mathbf{b}_1 + \mathbf{b}_2 + \cdots + \mathbf{b}_m) &= \sum_{i=1}^{n}\sum_{j=1}^{m} \mathbf{a}_i \rfloor \mathbf{b}_j, \\
(\mathbf{b}_1 + \mathbf{b}_2 + \cdots + \mathbf{b}_n)^{\sim} &= \sum_{i=1}^{n} \widetilde{\mathbf{b}}_i.
\end{aligned}$$

Therefore we can straightforwardly implement any linear product or operation for multivectors, using their implementations of basis blades of the previous section. As an example, Figure 3.5 gives `Java` code from the reference implementation that computes the outer product.

The explicit loops over basis blades and the actual basis blade product evaluations are quite expensive computationally, and implementations based on this principle are about one or two orders of magnitude slower than implementations that expand the loops and optimize them. We will get back to this in Chapter 4.

```
// Returns outer product of 'this' and 'x'
public Multivector op(Multivector x) {
   ArrayList result = new ArrayList(blades.size() * x.blades.size());

   // loop over basis blade of 'this'
   for (int i = 0; i < blades.size(); i++) {
      BasisBlade B1 = (BasisBlade)blades.get(i);

      // loop over basis blade of 'x'
      for (int j = 0; j < x.blades.size(); j++) {
         BasisBlade B2 = (BasisBlade)x.blades.get(j);
         // compute actual outer product of the basis blades ...
         // ... and add to result:
         result.add(BasisBlade.op(B1, B2));
      }
   }
   return new Multivector(simplify(result));
}
```

Figure 3.5: *Implementation of the outer product of multivectors based in the list-of-blades approach. The* `Multivector` *class has a member variable* `blades`*, which is an* `ArrayList` *of* `BasisBlades`*. The function* `simplify()` *simplifies a list of* `BasisBlades` *by adding those blades that are equal up to scale.*

3.3.3 Time Complexity of Linear Operations

Below, we determine the time complexity of the 'linear algebra' approach and the 'list of basis blades' approach, which are quite similar. In fact, when both implementations are implemented with care, they should have identical performance. The differences come from our assumptions that:

- the linear algebra approach does not make use of the fact that some of the matrices are sparse.
- the linear algebra approach does not use coordinate compression.
- the operands for the list of blades approach are versors.

Time Complexity of the Linear Algebra Approach

If we naively implement the unary operations reversion and grade involution, they require one matrix-vector multiply each. The matrix has size $2^n \times 2^n$, so these operations have $O(2^{2n})$ time complexity.

Addition and subtraction requires only $O(2^n)$ time as these operations map directly to standard vector addition and subtraction.

The geometric product requires initialization of a $2^n \times 2^n$ and one matrix-vector multiply. Initializing the matrix has $O(2^{2n})$ time complexity. The matrix-vector multiplication has $O(2^{2n})$ time complexity, too.

operations	linear algebra approach	list of blades approach
reversion, grade involution	$O(2^{2n})$	$O(2^{n-1})$
addition, subtraction	$O(2^{n})$	$O(2^{n-1})$
geometric product,	$O(2^{2n})$	$O(2^{2n})$
metric products, outer product	$O(2^{2n})$	$O(2^{2n})$

Table 3.2: Time complexities of the linear operations.

The metric products and the outer product are a bit cheaper to compute, if we make use of the fact that these matrices fill only one half of the matrix (see Figure 3.4). But half of $O(2^{2n})$ is still $O(2^{2n})$, so for the time complexity this does not matter.

Time Complexity of the List of Blades Approach

In the list of blades approach, reversion and grade involution take only $O(2^n)$ time, as these operations simply toggle the sign of selected coordinates, based on they grade of the basis blade they refer to. Likewise, addition and subtraction have $O(2^n)$ time complexity.

To compute the geometric product of two versors, each combination of coordinates from both operands must be processed (see the double loop in Figure 3.5). So the geometric product has $O(2^{n+n}) = O(2^{2n})$ time complexity.

The metric products and the outer product also require each combination of coordinates to be processed in some way, so they too have $O(2^{2n})$ time complexity.

Summary

Figure 3.2 summarizes the time complexities of both approaches. For the remainder of the thesis, we will use the time complexities of the list of blades approach as given. We note that when carefully implemented, the scalar product (a metric product) has only $O(2^{n-1})$ time complexity, as it is similar to the regular dot product from linear algebra.

3.4 Non-Linear Operations on Multivectors

To get a full geometric algebra implementation, we need more than just the linear operations on multivectors. Some essential operations are non-linear and for these we need to devise special algorithms. This section describes known algorithms

from literature (citations are given for the non-trivial algorithms). The non-linearity results in more complex algorithms, still reasonably efficient but typically an order of magnitude more time-consuming than the linear operations.

We describe algorithms for the inverse, for exponentiation, for testing whether a multivector is a blade or a versor, for blade factorization, and for the efficient computation of `meet` and `join`. These are the algorithms we had to implement for our day-to-day use in actual applications.

3.4.1 Inverse of Versors (and Blades)

In Section 2.6.6 we presented an efficient way to invert versors. We call the it the *versor inverse* method because, in general, it works only on versors. To compute the inverse of a versor, use:

$$\mathbf{V}^{-1} = \widetilde{\mathbf{V}}/(\mathbf{V}\,\widetilde{\mathbf{V}}). \tag{3.6}$$

We remind you that this equation also works for invertible blades, because they are versors, too.

If the operand is a versor, computing the scalar product $\mathbf{V}\,\widetilde{\mathbf{V}}$ has $O(2^{n-1})$ time complexity. Computing the scalar division also has $O(2^{n-1})$ time complexity, so inverting a versor has $O(2^{n-1})$ time complexity. Likewise, inverting a blade has $O(\frac{n!}{k!(n-k)!})$ time complexity. This means that inversion of blades and versors has the same time complexity as reversion, which is relatively low.

3.4.2 Inverse of Multivectors

Within the bounds of geometric algebra, the versor inverse method should cover all inversion needs, since all multivectors with a geometric interpretation are blades or versors. The logarithms of rotors — which are not blades in general — form the exception, but we do not need to invert those. However, for completeness we present a technique to invert *any* invertible multivector. It is based on the linear algebra implementation approach described in Section 3.3.1. It is based on the observation that

$$\mathbf{A}\,\mathbf{B} \rightleftharpoons [\mathbf{A}^G][\mathbf{B}],$$

from which it follows that

$$\mathbf{A}^{-1}\,\mathbf{B} \rightleftharpoons [\mathbf{A}^G]^{-1}[\mathbf{B}].$$

To see why, just left-multiply both sides by $\mathbf{A}$ and $[\mathbf{A}^G]$. So the matrix of the inverse of $\mathbf{A}$ is the inverse of the matrix of $\mathbf{A}$. Thus we can invert any invertible multivector using the following steps:

- Compute the geometric product matrix $[A^G]$.
- Invert this matrix.

- Extract $[\mathbf{A}^{-1}]$ as the first column of $[\mathbf{A}^G]^{-1}$.

To understand why the last step is correct, inspect the algorithm for computing the geometric product matrix. With the scalar 1 as the first element of L, the first column of the matrix $[A^G]$ is constructed from the geometric product of A with the first element of L, which is 1. Therefore the first column of a geometric product matrix $[\mathbf{A}^G]$ is $[\mathbf{A}]$ itself. This may also be observed in Figure 3.4. Hence we find the coefficients of $\mathbf{A}^{-1}$ as the first column of $[(\mathbf{A}^{-1})^G] = [\mathbf{A}^G]^{-1}$.

This inversion method has two disadvantages: it is slow and it is numerically imprecise due to floating point round-off errors caused by the matrix inversion. The versor inverse method of Section 3.4.1 is numerically more stable and also more efficient, so in practice we always use that method instead.

The time complexity of this inversion algorithm is dominated by the matrix inversion, which has an $O(2^{3n})$ time complexity.

3.4.3 Exponential, Sine and Cosine of Multivectors

The exponential and trigonometric functions were introduced for blades in Section 2.9.3. For reference, we repeat the equation for the polynomial expansion of the exponential:

$$\exp(\mathbf{A}) = 1 + \frac{\mathbf{A}}{1!} + \frac{\mathbf{A}^2}{2!} + \cdots \tag{3.7}$$

If the square of $\mathbf{A}$ is a scalar, this series can be easily computed using standard trigonometric or hyperbolic functions. We repeat the equation from Section 2.9.3:

$$\exp(\mathbf{A}) = \begin{cases} \cos\alpha + \mathbf{A}\,\frac{\sin\alpha}{\alpha} & \text{if} \quad \mathbf{A}^2 = -\alpha^2 \\ 1 + \mathbf{A} & \text{if} \quad \mathbf{A}^2 = 0 \\ \cosh\alpha + \mathbf{A}\,\frac{\sinh\alpha}{\alpha} & \text{if} \quad \mathbf{A}^2 = \alpha^2 \end{cases}, \tag{3.8}$$

where α is a real scalar.

In practice, one needs to compute exponentials of bivectors only. If the above equation applies (i.e., the bivector has a scalar square), then evaluating it requires one scalar product and one scalar multiply. This means that the equation has $O(\frac{n!}{2!(n-2)!})$ time complexity. This is about the same order as reversion of blade.

Unfortunately, we will need to take exponentials of general bivectors, which may not have a scalar square. The exponentials of general bivectors generate the continuous motions in a geometry, and only in fewer than 4 dimensions are bivector always 2-blades.

When the special cases do not apply, the series can be evaluated explicitly up to a certain order. This tends to be slower and less precise. Experience shows that evaluating the polynomial series up to order 10 to 12 gives the best results for 64 bit floating point values. A rescaling technique should be used that will increase accuracy, as follows.

Suppose you want to compute $\exp(\mathbf{A})$. If $\mathbf{A}$ is a lot larger than unity, $\mathbf{A}^k$ can be so large that the series overflows the accuracy of the floating point representation before it converges. To prevent this problem, we can scale $\mathbf{A}$ to near unity before evaluating the series, because of the identity

$$\exp(\mathbf{A}) = \left(\exp\left(\frac{\mathbf{A}}{s}\right)\right)^s.$$

For our purposes, any $s \approx \|\mathbf{A}\|$ will do. In a practical implementation, we choose s to be a power of two, so that we can efficiently compute $\left(\exp\left(\frac{\mathbf{A}}{s}\right)\right)^s$ by repeatedly squaring $\exp\left(\frac{\mathbf{A}}{s}\right)$.

The time complexity of Equation 3.7 is $O(p2^{2n})$, where p is the order up to which the series is evaluated, so $10 \leq p \leq 12$ in practice. (We list p in the time complexity even though it is a constant: in practice, the additive implementation method is not usable in spaces where the dimensionality n is much larger than 10, so in most cases, p is has the same order of magnitude as n).

A point of future research might be to find closed-form exponentiation equations for specific applications. For example, one might imagine that it is possible to find a closed-form equations for the exponent of a conformal translation-rotation-scaling bivector. This would avoid the costly explicit series evaluation, and likely increase precision.

3.4.4 Logarithm of Rotors

As stated before, a general algorithm for computing the logarithm of an arbitrary rotor is not known. However, for many useful special cases a closed form solution can be found. See for example Section 2.9.3 (rotation) and Section 16.3.4 (positively scaled rigid body motion) of [12].

3.4.5 Multivector Classification

At times, it is useful to have an algorithm that classifies a multivector as either a blade, a versor, or a non-versor. For instance, classification is useful as a sanity check of interactive input, or to verify whether a result could be displayed geometrically.

Yet testing whether a multivector is a versor or a blade is non-trivial in our additive representation. A blade test that simply requires that the multivector is of a single grade is insufficient (for example, $\mathbf{e}_1 \wedge \mathbf{e}_2 + \mathbf{e}_3 \wedge \mathbf{e}_4$ is of uniform grade 2, but not a blade). Adding the rule that the square of the multivector must be a scalar does not help (since $\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 + \mathbf{e}_4 \wedge \mathbf{e}_5 \wedge \mathbf{e}_6$ squares to -2, but it is not a blade).

Figure 3.6 shows the pseudo-code for the a classification algorithm that performs this test. It is based on [7], and is described in more detail below.

The algorithm can be used both for the versor test and for the blade test. It is natural that these tests are structurally similar, for all invertible blades are

```
// Pseudo-code for the multivector classification algorithm
MVType multivectorType(Multivector A, Metric M) {
  // step 1: compute versor inverse, check that it is the actual inverse:
  Multivector Avi = versorInverse(A, M);
  Multivector Agi = gradeInvolution(A);
  if (grade(gp(Agi, Avi, M)) != 0) return MULTIVECTOR;
  if ((gp(Agi, Avi, M) - gp(Avi, Agi, M)) != 0.0) return MULTIVECTOR;

  // Step 2: check that the images of basis vectors are vectors:
  for (int i = 0; i < A.spaceDim(); i++) {
     Multivector e_i = new Multivector(new BasisBlade(1 << i));
     if (grade(Avi.gp(ei, M).gp(Agi, M)) != 1)
       return MULTIVECTOR;
  }

  // Step 3: if A is homogeneous, it must be a blade, otherwise it is a versor:
  if (homogeneous(A)) return BLADE;
  else return VERSOR;
}
```

Figure 3.6: *This function returns the multivector type of a multivector* `A`*. The type can be* `MULTIVECTOR`*,* `VERSOR` *or* `BLADE`*. This function was rewritten to pseudo-code for presentation. The original code can be found at* [11]*.*

homogeneous versors. As you glance through it, you notice that the algorithm uses inverses. This is fine for a versor test, since versors need to be invertible by definition. However, a blade need not be invertible, and null blades are still blades.

In fact, "being a blade" is defined as "factorizable by the outer product", and that does not depend on a metric at all, see Section 2.5.6. If you are testing for whether $\mathbf{V}$ is a blade, you must perform the geometric products in the algorithm using a Euclidean metric. This choice of convenience eliminates all null blades that would have had to be taken into account without actually affecting factorizability, and that simplifies the algorithm. As in example, the multivector $\mathbf{e}_1 \wedge \infty$ is a blade, but in the conformal metric contains a null factor ∞. This makes the blade non-invertible, but it is of course still a blade. By using a Euclidean metric, ∞ is treated as a regular vector, thus the blade becomes invertible and the test can run as it would for $\mathbf{e}_1 \wedge \mathbf{e}_2$.

On the other hand, determining whether a multivector V is a versor (i.e., a geometric product of invertible vectors) clearly depends on the metric. So for a versor test, you have to run this algorithm using the actual metric of the algebra of the versor. For this purpose, the algorithm in Figure 3.6 allows the user to specify the metric to be used during the test.

The classification process for $\mathbf{V}$ consists of three parts:

1. Test if the versor inverse $\widetilde{\mathbf{V}}/(\mathbf{V}\,\widetilde{\mathbf{V}})$ is truly the inverse of the multivector V.

This involves the following tests:

$$\begin{aligned} \text{grade}(\widehat{\mathbf{V}}\,\mathbf{V}^{-1}) &\stackrel{?}{=} 0, \\ \widehat{\mathbf{V}}\,\mathbf{V}^{-1} &\stackrel{?}{=} \mathbf{V}^{-1}\,\widehat{\mathbf{V}}. \end{aligned}$$

If either of these test fails, we can report that the multivector is a non-versor, and hence a non-blade.

The use of grade involution in the two equations above is an essential technique from [7]. It prevents multivectors that have both odd and even grade parts from sneaking through the test: If $\mathbf{V}$ is an even versor then $\widehat{\mathbf{V}} = \mathbf{V}$. If $\mathbf{V}$ is an odd versor, $\widehat{\mathbf{V}} = -\mathbf{V}$. But if $\mathbf{V}$ has both odd and even grade parts, the grade involution prevents odd and even parts from recombining in a way that they cancel each other out.

2. The second test is on the grade preservation properties of the versor: applying a versor (and hence an invertible blade) to a vector should not change the grade. So for each basis vector $\mathbf{e}_i$ of the vector space $\mathbb{R}^n$, the following test should hold if $\mathbf{V}$ is a versor:

$$\text{grade}(\widehat{\mathbf{V}}\,\mathbf{e}_i\,\widetilde{\mathbf{V}}) \stackrel{?}{=} 1.$$

When the multivector does not pass this test, we report a non-versor (and hence a non-blade), otherwise we know that it is either a versor or a blade.

3. The final part makes the distinction between blades and versors by simply checking whether the multivector is homogeneous; if so, it is a blade, otherwise it is a versor.

One problem related to determining whether a multivector is a blade (or a versor) is that of finding the blade (or versor) closest to some non-blade (or non-versor). We currently have no general solution to this problem. A practical application of such an algorithm might be to correct numerical 'drift' due to accumulating floating point round-off errors. Unfortunately, a general algorithm for 'projecting' near-blades and near-versor onto the blade- or versor-manifold respectively is not known.

The time complexity of the algorithm is dominated by the main loop (step 2 in Figure 3.6). The versor inverse and the grade involution in step 1 of the algorithm each have a time complexity of $O(2^n)$ (we assume the algorithm can be handed an arbitrary multivector). The geometric products in step 1 of the algorithm have $O(2^{2n})$ time complexity. The first geometric product in the main loop (`Avi.gp(ei, M)`) has only $O(2^n)$ time complexity as it computes the geometric product of a single basis vector with an arbitrary multivector. The second product on the other hand has a time complexity of $O(2^{2n})$. And since the loop runs n times (worst case), the total time complexity of the loop is $O(n2^{2n})$, which is also the time complexity of the full algorithm.

3.4.6 Blade Factorization

This section deals with generic factorization of blades. That is the problem to find, for a given blade $\mathbf{B}_k$, a set of k vectors $\mathbf{b}_i$ such that

$$\mathbf{B}_k = \mathbf{b}_1 \wedge \mathbf{b}_2 \wedge \cdots \mathbf{b}_k.$$

One reason to factorize blades is to pass them (in factorized form) to libraries that cannot handle blades directly, or to implement another low level algorithm such as the computation of the `meet` and `join`, as discussed below.

We are only concerned with the outer product and consequently (as in the previous section) are allowed to choose any convenient metric. To avoid any problems with null vectors, we use a Euclidean metric for all metric products in the algorithm that follows.

A first step towards a useful factorization algorithm is finding potential factors of a blade $\mathbf{B}$. This can be done using projection. Take any candidate vector $\mathbf{c}$ (for example, a basis vector), and project it onto $\mathbf{B}_k$:

$$\mathbf{f} = (\mathbf{c} \rfloor \mathbf{B}_k)\, \mathbf{B}_k^{-1}.$$

If $\mathbf{f}$ is 0, then another candidate vector should be tried. For better numerical stability, we should also try another candidate when $\mathbf{f}$ is close to 0. Or better yet, try many different candidate vectors and use the one that results in the largest $\mathbf{f}$. In any case, when $\mathbf{f}$ is a usable factor we can remove it from $\mathbf{B}_k$ simply by dividing it out:

$$\mathbf{B}_{k-1} = \mathbf{f}^{-1} \rfloor \mathbf{B}_k. \tag{3.9}$$

Then we can write

$$\mathbf{B}_k = \mathbf{f} \wedge \mathbf{B}_{k-1},$$

so that we have found our first factor of $\mathbf{B}_k$. We now repeat this process on the blade $\mathbf{B}_{k-1}$, iteratively, until we are left with a final vector factor $\mathbf{B}_1$. [1]

This basic idea works, but the procedure may be inefficient in high-dimensional spaces. Many candidate vectors may result in a zero value for $\mathbf{f}$, which is a waste of effort. It is better to use the structure of $\mathbf{B}$ itself to limit the search for its factors in an efficient, stable technique, as follows. The blade $\mathbf{B}$, as given, is represented as coordinates relative to a basis of blades. We take the basis blade with the absolute largest coordinate on this basis. Let that be $\mathbf{E} = \mathbf{e}_{i_1} \wedge \mathbf{e}_{i_2} \wedge \cdots \wedge \mathbf{e}_{i_k}$. We then use the basis vectors that make up $\mathbf{E}$ as candidate vectors for projection. This selection procedure guarantees that the projection of each of the candidate vectors is non-zero (we show this at the end of this section).

One final issue is the scaling of each of the factors. Due to the projection, the factors do not have a predictable scale, which is a numerically awkward property.

[1] We remove the factor $\mathbf{b}_f$ from $\mathbf{B}_k$ because we want the factors to be orthogonal. If one simply needs factors that make up $\mathbf{B}_k$, and orthogonality or scale is not an issue, one may adjust the algorithm to use the original $\mathbf{B}_k$ in each loop of the final algorithm below (inear independence of the factors is ensured by picking the largest basis blade $\mathbf{E}$).

Our implementation of the factorization algorithm normalizes the factors and returns the scale of the blade as a separate scalar.[2]

The final algorithm (based on [6]) for factorizing a blade becomes:

1. Input: a non-zero blade $\mathbf{B}$ of grade k.
2. Determine the norm of $\mathbf{B}$: $s = \|\mathbf{B}\|_E$.
3. Find the basis blade $\mathbf{E}$ in the representation of $\mathbf{B}$ with the largest coordinate. Determine the k basis vectors $\mathbf{e}_i$ that span $\mathbf{E}$.
4. Let the current input blade be $\mathbf{B}_c \leftarrow \mathbf{B}/s$.
5. For all but one of the basis vectors $\mathbf{e}_i$ of $\mathbf{E}$:
    (a) Project $\mathbf{e}_i$ onto $\mathbf{B}_c$: $\mathbf{f}_i = (\mathbf{e}_i \rfloor \mathbf{B}_c)\,\mathbf{B}_c$.
    (b) Normalize $\mathbf{f}_i$. Add it to the list of factors.
    (c) Update $\mathbf{B}_c$: $\mathbf{B}_c \leftarrow \mathbf{f}_i^{-1} \rfloor \mathbf{B}_c$.
6. Obtain the last factor: $\mathbf{f}_k = \mathbf{B}_c$. Normalize it.
7. Output: the factors $\mathbf{f}_i$ and the scale s.

Some notes on metric are in order. First, the algorithm also works for null blades, since no blade is actually null in the Euclidean metric that is used during the factorization algorithm itself. Second, the output of the algorithm is a set of orthonormal factors in the Euclidean metric that was used within the algorithm. That may not be the metric of the space of interest. If orthonormality is required in the some other metric, the metric matrix of the factors may be constructed, decomposed using the eigenvalue decomposition, and used to construct an orthonormal set of factors (see also Sections 2.5.2 and 2.7.1).

To see why the projection of $\mathbf{e}_i$ on $\mathbf{B}_c$ is never zero (step 5a), note that there always exists a rotor $\mathbf{R}$ that turns $\mathbf{E}$ to the original $\mathbf{B}$. This rotor will never be over 90 degrees (for that would imply that $\mathbf{E} \rfloor \mathbf{B} = 0$, yet we know that $\mathbf{E} \rfloor \mathbf{B}$ must be non-zero to be the basis blade with the largest coordinate in $\mathbf{B}$). We may not be able to compute $\mathbf{R}$ easily, but we can find $\mathbf{R}^2 = \mathbf{R}\,\mathbf{R} = \mathbf{B}\mathbf{E}^{-1}$. A rotation over twice the angle between two vectors is constructed as simply the geometric product of the vectors, see Section 2.9.2.

Since $\mathbf{R}$ is never over 90 degrees, $\mathbf{R}^2$ will never be over 180 degrees. Because of this, the quantity $\frac{1}{2}(\mathbf{e}_i + \mathbf{R}^2\,\mathbf{e}_i/\mathbf{R}^2)$ must be non-zero: no $\mathbf{e}_i$ is rotated far enough

[2]Other solutions are to premultiply the first factor with the scale, or to apportion the scale evenly over each factor. Which method is most convenient may partly depends on the subsequent use of the factorization. In any case, the unit factors with a separate scale can be transformed into any of the other representations easily.

by $\mathbf{R}^2$ to become its own opposite. We rewrite this and find:

$$\begin{aligned}
0 &\neq \tfrac{1}{2}(\mathbf{e}_i + \mathbf{R}^2\,\mathbf{e}_i/\mathbf{R}^2) \\
&= \tfrac{1}{2}(\mathbf{e}_i + \mathbf{B}\,\mathbf{E}^{-1}\,\mathbf{e}_i\mathbf{E}\mathbf{B}^{-1}) \\
&= \tfrac{1}{2}(\mathbf{e}_i - \mathbf{B}\,\mathbf{E}^{-1}\widehat{\mathbf{E}}\,\mathbf{e}_i\mathbf{B}^{-1}) \\
&= \tfrac{1}{2}(\mathbf{e}_i - \widehat{\mathbf{B}}\,\mathbf{e}_i\,\mathbf{B}^{-1}) \\
&= (\mathbf{e}_i \rfloor \mathbf{B})\,\mathbf{B}^{-1}.
\end{aligned}$$

Therefore none of the $\mathbf{e}_i$ from $\mathbf{E}$ projects to zero on $\mathbf{B}$, so $\mathbf{f}_1$ is non-zero in the first pass of the algorithm,. After removal of $\mathbf{f}_1$, the same argument can be applied to the next $\mathbf{e}_i$ on $\mathbf{B}_c$, et cetera. Hence none of the $\mathbf{f}_i$ are zero.

Tightly bounding the time complexity of the blade factorization algorithm is rather hard due to step 5 of the algorithm. Steps 2, 3 and 4 all have $O(\frac{n!}{k!(n-k)!})$ time complexity. Step 5a projects a basis vector onto the blade. The first part of the projection $(\mathbf{e}_i \rfloor \mathbf{B}_c)$ has a time complexity of $O(\frac{n!}{(k-i+1)!(n-k+i+1)!})$, where i is the iteration count of the algorithm. The second part of the projection $((\mathbf{e}_i \rfloor \mathbf{B}_c)\,\mathbf{B}_c)$ is a geometric product of two blades of grade $k-i$ and $k-i+1$, respectively. This is clearly the dominant operation of the loop in terms of time complexity. However, tightly bounding the time complexity of the full loop (which runs k times) would lead to a complex expression with little informative value. On the other hand, if we state that the time complexity of the second geometric product is of order $O(2^{2n})$ (an overstatement), then we can bound the time complexity of the full algorithm to order $O(k2^{2n})$.

3.4.7 `meet` and `join` of Blades

The `meet` and `join` products (introduced in Section 2.13) are the geometrical versions of set intersection and union, respectively, applied to blades rather than sets. They really work on the basis spanning the blades, and are in that sense discrete.

Figure 3.7 shows an example. Suppose we have to compute the `meet` and `join` of two blades $\mathbf{A}$ and $\mathbf{B}$ which can be factored as

$$\begin{aligned}
\mathbf{A} &= \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \mathbf{c}, \\
\mathbf{B} &= \mathbf{c} \wedge \mathbf{b}_1,
\end{aligned}$$

where $\mathbf{c}$ is the largest common factor — a vector in this example. Figure 3.7a illustrates this; each block in the figure is a vector, and grey lines surround $\mathbf{A}$ and $\mathbf{B}$. The `join` (Figure 3.7b) is then

$$\mathbf{A} \cup \mathbf{B} \sim \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \mathbf{c} \wedge \mathbf{b}_1.$$

We use the symbol $\sim$ for “is proportional to”. The `meet` (Figure 3.7c) is

$$\mathbf{A} \cap \mathbf{B} \sim \mathbf{c}.$$

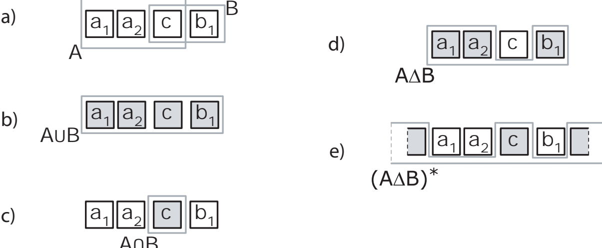


Figure 3.7: *'Venn diagrams' illustrating* `meet`*,* `join`*, and the delta product of two blades.*

The symmetric difference (delta product, Figure 3.7d) is

$$\mathbf{A} \bigtriangleup \mathbf{B} \sim \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \mathbf{b}_1.$$

We need the delta product to compute the grade of the `meet` or `join` based on **A** and **B**. The delta product was defined in Section 2.13.2 as the highest non-zero grade part *max* of a geometric product:

$$\mathbf{A} \bigtriangleup \mathbf{B} = \langle \mathbf{A}\,\mathbf{B} \rangle_{\text{max}}.$$

We also need the dual (with respect to the full space) of the delta product, which is shown in Figure 3.7e. This dual includes **c**, and the rest of the space that is not in either **A** or **B**.

The algorithm (based on [6] with improvements [3]) we present in this section first computes the required grade of the `meet` and `join`, using the delta product. Then it works towards constructing either the `meet` or the `join`. As soon as it finds either, it computes the other using Equation 2.47 and return the result.

Computing the Grade of the `meet` and `join`

To compute the grade of the `meet` and `join` based on the operands, the delta product can be used. The delta product $\mathbf{A} \bigtriangleup \mathbf{B}$ computes a blade that contains those factors of **A** and **B** that are not common to both of them. The grade of this blade helps us to compute `meet` and `join`.

To explain how, we assume the `meet` $\mathbf{C} = \mathbf{A} \cap \mathbf{B}$ is already known, so we can write

$$\begin{aligned} \mathbf{A} &= \mathbf{A}^r\,\mathbf{C}, \\ \mathbf{B} &= \mathbf{C}\,\mathbf{B}^r, \end{aligned}$$

where $\mathbf{A}^r$ and $\mathbf{B}^r$ are the 'remainders' of **A** and **B** under the factorization. Using the above factorization **A** and **B**, it is easy to see that the sum of the grade of **A**

and $\mathbf{B}$ is:

$$\begin{aligned}\text{grade}(\mathbf{A}) + \text{grade}(\mathbf{B}) &= \text{grade}(\mathbf{A}^r\,\mathbf{C}) + \text{grade}(\mathbf{C}\,\mathbf{B}^r) \\ &= \text{grade}(\mathbf{A}^r) + \text{grade}(\mathbf{B}^r) + 2\,\text{grade}(\mathbf{C}).\end{aligned}$$

If we now note that

$$\mathbf{A} \bigtriangleup \mathbf{B} = \langle(\mathbf{A}^r\,\mathbf{C})\,(\mathbf{C}\,\mathbf{B}^r)\rangle_{\text{max}} \sim (\mathbf{C}\cdot\mathbf{C})\mathbf{A}^r \wedge \mathbf{B}^r \sim \mathbf{A}^r \wedge \mathbf{B}^r,$$

and thus

$$\text{grade}(\mathbf{A} \bigtriangleup \mathbf{B}) = \text{grade}(\mathbf{A}^r) + \text{grade}(\mathbf{B}^r),$$

we may compute the grade of the `join` as

$$\text{grade}(\mathbf{A} \cup \mathbf{B}) = \frac{\text{grade}(\mathbf{A}) + \text{grade}(\mathbf{B}) + \text{grade}(\mathbf{A} \bigtriangleup \mathbf{B})}{2}. \tag{3.10}$$

The reasoning for computing the grade of the `meet` is analogous and leads to

$$\text{grade}(\mathbf{A} \cap \mathbf{B}) = \frac{\text{grade}(\mathbf{A}) + \text{grade}(\mathbf{B}) - \text{grade}(\mathbf{A} \bigtriangleup \mathbf{B})}{2}. \tag{3.11}$$

Computing the `meet` and `join`

The algorithm for actually computing the `meet` and `join` initially assumes that the `meet` is a scalar, and tries to expand it through the outer product with new vectors until it arrives at the true `meet`. Potential factors of the `meet` satisfy the following conditions:

- They are not factors of the delta product.
- They are factors of $\mathbf{A}$ and $\mathbf{B}$.

You can verify this using Figure 3.7. Likewise, the algorithm initially assumes that the `join` is the pseudoscalar of the full space, and removes factors from it, until the true `join` is obtained. Factors that should not be in the `join` satisfy the following conditions:

- They are factors of the dual of the delta product.
- They are not factors of $\mathbf{A}$ and $\mathbf{B}$.

We now have enough building blocks to construct an algorithm to compute the `meet` and `join`:

1. Input: two blades $\mathbf{A}$, $\mathbf{B}$, and possibly a threshold ϵ for the delta product.
2. If $\text{grade}(\mathbf{A}) > \text{grade}(\mathbf{B})$, swap $\mathbf{A}$ and $\mathbf{B}$.

3. Compute the dual of the delta product: $\mathbf{S} = (\mathbf{A} \triangle \mathbf{B})^\star$. A threshold ϵ may be used to suppress floating point noise when computing the delta product (specifically, when determining what is the top non-zero grade part of $\mathbf{A}\,\mathbf{B}$).

4. Factorize $\mathbf{S}$ into factors $\mathbf{s}_i$.

5. Compute the required grade of the `meet` and `join` (Equation 3.10 and Equation 3.11).

6. Set $\mathbf{M} \leftarrow 1$, $\mathbf{J} \leftarrow \mathbf{I}_n$ ($\mathbf{I}_n$, the pseudoscalar of the total space).

7. For each of the factors $\mathbf{s}_i$:

   (a) Compute the projection $\mathbf{p}_i$ and rejection $\mathbf{r}_i$ of $\mathbf{s}_i$ and $\mathbf{A}$:

$$\begin{aligned} \mathbf{p}_i &= (\mathbf{s}_i \rfloor \mathbf{A}) \rfloor \mathbf{A}^{-1} \\ \mathbf{r}_i &= \mathbf{s}_i - \mathbf{p}_i \end{aligned}$$

   (b) If the projection is not zero, then wedge it to the `meet`: $\mathbf{M} \leftarrow \mathbf{M} \wedge \mathbf{p}_i$. If the new grade of $\mathbf{M}$ is the required grade of the `meet`, then compute the `join` using Equation 2.46, and break the loop. Otherwise continue with $\mathbf{s}_{i+1}$.

   (c) If the rejection is not zero, then remove it from the `join`: $\mathbf{J} \leftarrow \mathbf{r}_i \rfloor \mathbf{J}$. If the new grade of $\mathbf{J}$ is the required grade of the `join`, then compute the `meet` using Equation 2.47, and break the loop. Otherwise continue with $\mathbf{s}_{i+1}$.

8. Output: $\mathbf{A} \cap \mathbf{B}$ and $\mathbf{A} \cup \mathbf{B}$.

For added efficiency, step 4 could be integrated into the main loop of the algorithm (i.e., only find factors of $\mathbf{S}$ as required).

Note that the algorithm always returns both $\mathbf{A} \cap \mathbf{B}$ and $\mathbf{A} \cup \mathbf{B}$, while only one of them may be required. This is because the algorithm searches for both, but terminates as soon as it finds either one of them. Benchmarks have shown that this is more efficient than an algorithm that searches specifically for either the `meet` or the `join`, as it will terminate as soon as it finds either of them. The returned `meet` and `join` are based on the same factorization, so that we can use relationships of Equation 2.46 and Equation 2.47 to compute one given the other.

In the worst case scenario, the algorithm loops $n-1$ times, where n is the dimension of the vector space. This can be understood by analyzing the following example:

$$\mathbf{A} = \mathbf{e}_n, \qquad \mathbf{B} = \mathbf{e}_n,$$

in which case

$$\mathbf{A} \triangle \mathbf{B} = 1, \qquad \mathbf{S} = (\mathbf{A} \triangle \mathbf{B})^\star = \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \cdots \wedge \mathbf{e}_n.$$

The algorithm will start by projecting and rejecting $\mathbf{e}_1$. The projection will be zero, so the $\mathbf{M}$ will not 'grow' towards the actual `meet` (likewise for $\mathbf{e}_2 \cdots \mathbf{e}_{n-1}$). When the projection is zero, the rejection is obviously non-zero, so each of the rejections is removed from the $\mathbf{J}$. While this brings $\mathbf{J}$ closer and closer to the actual `join`, $\mathbf{J}$ has to 'shrink' all the way until it is of grade 1, which will not happen until all $\mathbf{e}_1 \cdots \mathbf{e}_{n-1}$ have been processed, leading to $O(n)$ cycles of the loop.

We should note that when the input blades $\mathbf{A}$ and $\mathbf{B}$ are in general position, the projection and rejection are likely to be non-zero in each cycle of the loop. In that case the required number of cycles is $\min(\text{grade}(\texttt{meet}), \text{grade}(\texttt{join})) < n/2$. For numerical stability, we should require that the projections and rejections have some minimum weight, which will increase the number of cycles (because some projections and rejections will not be used for computation).

In terms of time complexity, the `meet` and `join` algorithm is similar to the blade factorization algorithm: the time complexity is dominated by the products in the main loop (the left contractions of step 7a, to be precise). As with the blade factorization algorithm, tightly bounding the time complexity of these products would lead to a complex non-informative expression, so we take the easy route and assume that the second left contraction of step 7a has a time complexity of $O(2^{2n})$. If we set

$$k = \max(\text{grade}(\mathbf{A}), \text{grade}(\mathbf{B})),$$

then the loop of step 7 makes $O(k)$ loops, and the time complexity of the full algorithm is $O(k2^{2n})$.

3.5 Discussion and Summary

This chapter described the building blocks of an additive geometric algebra implementation. The next chapter will describe our approach to optimize such an implementation. It is natural to divide such an implementation into three layers, and each layer has its own considerations for efficiency. The first layer (geometric algebra on basis blades) is precomputed in practice, so an efficient implementation is not critical. For the efficiency of the other second layer (linear operations), two important choices must be made:

- How to compress the coordinates of multivectors
- How to implement the linear operations, given this compression method.

We describe our choice for `Gaigen 2` in next chapter. We will not focus on optimization of the third layer. It can be built upon the second layer and would thus 'inherit' its (in-)efficiency, or one may devise custom implementations for each algorithm. Experiments [3] have shown us that the latter option may be fruitful, but this is future work.

[3] We wrote a custom implementations of blade factorization and pf the `meet` and `join`. These implementations were only about 10 times as expensive as a bilinear product, on average. This

In terms of time complexity. there are roughly three groups of operations / algorithms:

- Simple operations (reversion, addition, inversion): $O(2^n)$ time complexity.
- Bilinear operations (products): $O(2^{2n})$ time complexity.
- Non-linear operations (classification, exponentiation, factorization, meet and join): $O(x2^{2n})$ time complexity, where x is either the grade of the operands k, the dimensionality of the space n or some constant value of the same order of magnitude as n.

The $O(2^n)$ or higher time complexity limits the use of the additive implementation method to low dimensional spaces; we will see in Chapter 5 that the multiplicative implementation method is more efficient when $n \geq 10$, approximately. In low dimensional spaces, the constant factor of the time complexity has a large influence on the actual performance of the implementation, relative to variables such as n and k. This means that making sure the constant factor is as low as possible is important, especially to compete with traditional geometry implementations that are usually also very optimized. As the benchmarks in Chapter 7 will show, at the time we started our work, no existing geometric algebra implementation came close to traditional geometry implementations, in terms of performance. This prompted us to try to achieve this goal, by the methods described in the next chapter.

should be compared to an implementation on top of `Gaigen 2` (Chapter 4), which turned out about 100 tunes as expensive as a bilinear product.

Chapter 4

Optimizing the Additive Implementation

The previous chapter described the most common method used to implement geometric algebra, the additive implementation. This chapter describes our approach to optimizing that implementation method, in such a way that geometric-algebra-based geometry can perform on a par with classical (linear-algebra-based) geometry implementations.

Our goal is not implementations of one-off equations, small algebras or optimized special-purpose code: these can be programmed by hand, although this is tedious work. The type of implementations we aim at in this chapter are general purpose, low dimensional (up to about 10-D) geometric algebra implementations. An important example is the implementation of the 5-D algebra required for the conformal model of 3-D space. We use this algebra as a source of examples throughout this chapter.

Our method is to automatically generate such implementations from a specification of the algebra. This specification is written by the user of `Gaigen 2`. It describes the details of the algebra (such as dimension and metric), functions over the algebra, and profiling information. The functions are written in a small domain specific programming language. All this information is transformed automatically by `Gaigen 2` into an optimized implementation written in a mainstream language such as `C++` or `Java`. The generated source code is compiled by a regular compiler and linked to the rest of the *user application*. The user application can be any program that uses geometry, for example a ray tracer or an optical motion capture system (these are our two main applications so far). So in this context the *user* is a programmer who uses our generated implementation to implement his or her geometry.

The first part of this chapter discusses existing implementations, the bottlenecks that make these implementations slow, and our solutions to these problems. Because our implementation is automatically generated, we also briefly discuss various generative programming approaches to motivate the particular approach we used. The second part of the chapter is a detailed description of the resulting code generator, `Gaigen 2`, which stands for Geometric Algebra Implementation Generator 2.

4.1 Existing Additive Implementations

There are many geometric algebra implementations based on the additive implementation principle. Most of them are written in `C++`.

`GABLE` [32] is a geometric algebra implementation intended for learning geometric algebra. It runs in Matlab. Multivectors are stored as non-compressed $2^n \times 1$ matrices. $2^n \times 2^n$ matrices are used to compute the linear products, using the method of Section 3.3.1. As a result, `GABLE` is very slow. One or our benchmarks showed it to be around 15000 times slower than traditional (i.e., linear algebra based) geometry implementations, see Section 7.4.3.

`Gaigen 1` was our first Geometric Algebra Implementation Generator. It

started out as an optimized version of `GABLE`. Coordinates are stored using per-grade compression. Profiling is used to optimized the generated code for a particular application. We showed that `Gaigen 1` is about about 2 to 5 times slower in a ray tracer application than traditional geometry implementations. See Chapter 7 and [19].

`CLU` [36] is a geometric algebra implementation that is part of a larger package which also includes a 'visual calculator'. `CLU` stores multivectors in non-compressed arrays of 2^n coordinates. A minor optimization in the implementation is that it skips zero coordinates during product evaluation. However, this is not enough to obtain an efficient implementation, and `CLU` is about around 100 times slower than traditional geometry implementations. We show extensive benchmarks in Chapter 7.

`MV` is a proof-of-concept implementation by Ian Bell [3]. It uses a clever per-coordinate compression scheme where a single bitmap is used to indicate which coordinates are present. When that bitmap is too small for the dimensionality of the algebra, another bitmap is used for the next 'level'. It is the only additive implementation know to us that can function in spaces of $n = 15$ and up.

`Clifford` [41] is a proof-of-concept geometric algebra implementation by Jaap Suter. It uses `C++` meta-programming to achieve an efficient implementation, approaching the efficiency of a traditional geometry implementations. It uses type-based compression (new types can be created using template instantiation).

We do not discuss each implementations known to us here. Others are for example `Geoma` [15] and `Glucat` [31]. We have casually benchmarked these implementations and found their performance to be similar to `CLU`.

4.2 Issues in Efficient Implementation

There are a number of issues inherent to geometric algebra that make it hard to create an efficient additive implementation:

- Multivectors are the general elements of computation in geometric algebra, but they are 'big' compared to the dimensionality of the vector space: a multivector has 2^n coordinates in an n-D space. A multivector can represent any multivector type (vector, rotor, circle, and so on) in a geometric algebra. This abstraction is great from a mathematical point of view, but it makes multivectors sparse in practice: only a limited number of the 2^n coordinates are non-zero for each variable. This enforces overhead on the representation: either memory is wasted by storing zero coordinates, or processing time is wasted on compression of those coordinates. We already discussed various approaches to coordinate compression in Section 3.2.

- The number of basic geometric algebra operations on multivectors is quite large. (By 'operation' we mean all products, and other functions we discussed in Chapter 2, like addition, reversion, dualization, projection). Prefer-

ably we would like every operation to be implemented for every (combination of) multivector type(s), but this leads to a combinatorial explosion when encoded individually for maximum efficiency.

- The metric is a basic feature of a geometric algebra, affecting the most basic products. Many different metrics are useful and need to be allowed. Looking up the metric at run-time (e.g., from a table) is too costly. We should therefore not just implement a general geometric algebra and use it in a particular situation; for efficiency, we need a special implementation for every metrically different geometric algebra.

There are also some issues that affect all algebras with relatively 'small' elements:

- The execution of each individual product or operation requires only a few processor instructions. Therefore any overhead imposed by the implementation (such as a conditional branch due to looping) results in a significant performance degradation.
- Expressions consisting of multiple products and operations can often be executed much more efficiently by folding them into one calculation rather than executing them one by one through a series of function calls.

4.3 Resolving the Issues

We argue that if the following (relatively generic) optimization goals are achieved, these issues are resolved. I.e., the resulting geometric algebra implementation would then have performance comparable to traditional geometry implementations.

1. Waste as little memory as possible. I.e., store each variable as compactly as possible.
2. Implement functions over the algebra efficiently. To do this we need several things:
   (a) Process as few 'zero' coordinates as possible (which coincides with Goal 1),
   (b) Minimize (unpredictable) memory access. This coincides with Goal 1 and Goal 2a, but also demands avoiding lookup from tables, etc.
   (c) Avoid conditional branches. When a modern processor mispredicts a conditional branch, a large number of processor cycles is lost.
   (d) Unroll loops (e.g., over the dimension of the algebra, or the grade of a blade), whenever possible. This avoid branches (coinciding with Goal 2c) and also allows us to apply further optimizations.

(e) Optimize non-trivial expressions. Avoid implementing them as a series of function calls.

Except for Clifford, the implementations described in Section 4.1 typically satisfy none of these goals, or only the first goal (waste as little memory as possible), and more than ten times slower than standard linear algebra based geometry.

4.3.1 Overview of Our Implementation Approach

Our approach is built on top of the ideas of Chapter 3, and fulfills the optimization goals from the previous section. It can be seen as an optimization of the 'list of basis blades' approach from Section 3.3.2. To write out expressions on a basis, we use the bitmap techniques of Section 3.1.

One could think of `Gaigen 2` as a 'geometry compiler'. That is, `Gaigen 2` is a code generator that converts a high-level algebra specification to low-level source code. Code generation is used because it is too much work to write our type of implementations by hand. For example, the conformal geometric algebra implementation used by the ray tracer of Chapter 6 counts around 25.000 lines of code.

To represent multivectors, two types of classes are generated: general and specialized. The general multivector class can hold any multivector value, but it is relatively slow and bulky. The specialized multivector classes use type-based compression: they can hold only specific multivector types, and hence are lean and mean. Specialized multivector classes store only the non-zero coordinates for that specific type. For example, a 3-D vector class only stores the $\mathbf{e}_1$-, $\mathbf{e}_2$- and $\mathbf{e}_3$-coordinates.

Specialized multivector classes that represent useful constants (such as the basis vectors or the pseudoscalar) can also be generated. These classes serve only as symbolic tokens, as they are ultimately optimized away when they are used in equations.

The user (the programmer who uses `Gaigen 2`) writes functions over multivectors in a Domain Specific Language (DSL). `Gaigen 2` can instantiate these DSL functions with general or specialized multivectors, and emits optimized source code for each instantiation. The metric of the algebra is coded directly into these optimized functions. The optimized functions are then compiled by a regular (`C++` or `Java`) compiler, and the user can call the generated functions from a regular (`C++` or `Java`) program.

Profiling is used to automatically determine what DSL functions to instantiate with what (specialized) multivector types to arrive at the optimal implementation.

To speed up the application of outermorphisms, `Gaigen 2` can also generate outermorphism classes, again general and specialized. These classes contain matrix representations of the outermorphisms they represent.

Figure 4.1: *Basic tool chain from source code to running application with three points where code generation can take place (see text).*

4.4 Generative Programming

Generative programming [8] is the craft of developing programs that synthesize other programs. One of its uses is to connect software components (such as geometric algebra implementations) without loss of performance. Ideally, each component adapts dynamically to its context. In our case, we would like to use generative programming to transform the specification of a geometric algebra into an optimized implementation that is tailored to the needs of the rest of the program.

There are several points in the tool chain from source code to fully linked and running program where this transformation can take place. This is illustrated in Figure 4.1. We list three obvious examples, but other approaches (or hybrids thereof) are also possible:

1. The most explicit approach is generating an implementation of the algebra before the actual compilation of the program takes place. This is the way classical code generators like `lex` and `yacc` work. Advantages are that generated code is directly available to the user as a library, and that this method does not interfere with the rest of the chain. The main disadvantage is that it does not integrate well since a separate program is required to generate the code. Another disadvantage is that some form of feedback from the final, running program (i.e., profiling) is required to allow the implementation to adapt itself to the context. This feedback loop may require an extra code generation pass.

2. The transformation can also take place at compile-time, if the programming language permits this. This is called meta-programming, where the implementation is set up such that the compiler 'generates' parts of the implementation at compile time. For example, in `C++` this is possible though the use of the *templates* feature, which was originally added to the language for generic programming. The definite advantage of this method is the good integration with the language. A disadvantage is the limited number of mainstream programming languages that support meta-programming. Disadvantages specific to the `C++` programming language are the complicated template syntax, hard to decipher compiler error messages (both for user of the library and its developer) and the long compile times. The `boost::math::clifford` library [41] uses this approach.

3. A third option is to delay code generation until the program is up and running. The program would generate the code as soon as it is required (e.g., the first time a function of the algebra is called with specific arguments). This is called JIT (just-in-time) compilation or compile-on demand. The advantages are that the algebra implementation can adapt itself to the actual input and the actual hardware that it runs on (e.g., available instruction set extensions, the speed of registers compared to cache compared to main memory, etc. [2]). Disadvantages are the slower startup time of the final program (due to the run-time code generation) and the fact that the method is rather unconventional and hard to implement. At the time of writing we are not aware of such a geometric algebra implementation.

Our implementation `Gaigen 2` uses the first method. This method was chosen because it is language independent (`Gaigen 2` in fact provides multiple language back ends), and offers a large degree of freedom in what is generated. It is also easy to inspect the generated code.

4.5 `Gaigen 2` Implementation

The rest of the chapter describes `Gaigen 2` in full detail. `Gaigen 2` is a `Java` program which takes as input a `.gs2` file and one or more `.gp2` files. The `.gs2` file contains the algebra specification and DSL functions definitions, while the `.gp2` files contain the optional profiling information. `Gaigen 2` converts this information into source files (currently `C++` or `Java`) and `ANTLR` [35] grammars. (The `ANTLR` grammars are only for parsing human-readable multivector strings, so they do not form an core part of the implementation).

Figure 4.2 shows an overview of the `Gaigen 2` code generation process. Our discussion follows this figure from top-left to bottom-right, and ends by closing the loop with profiling feedback (top-right).

In the examples below, we use `C++` as the target language because this is the currently the most often used target. Throughout this section we use the conformal geometric algebra used for the ray tracer (Chapter 6) as the main source of examples.

4.6 Implementation: Specification of the Algebra

The content of the specification of the algebra is:

- Low-level code generation details.
- The basis and the metric of the algebra.
- The definition of the general multivector type.
- The definition of the specialized multivector types.

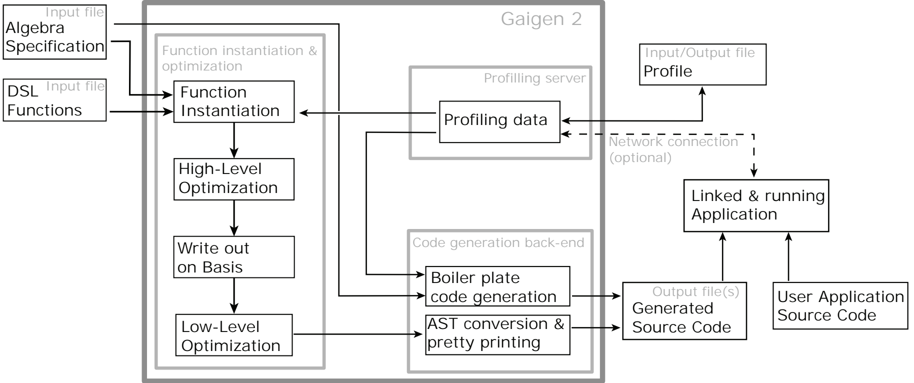


Figure 4.2: *The full* `Gaigen 2` *code generation process.*

- The definition of constants.
- The definition of the general outermorphism type.
- The definition of the specialized outermorphism types.

We discuss each of these items below.

4.6.1 Low-level Code Generation Details

A `.gs2` file starts with some practical details required for code generation, such as the license and namespace of the generated code; an example is shown in Figure 4.3. Many entries are target language dependent, such as the operator bindings for `C++`. The example lists (in order):

- The **license** under which the generated code is placed (can be proprietary license or open source).
- The **language** in which the generated code is emitted (currently, this can be `C++` or `Java`).
- How to **pass** variables to functions (by reference or value).
- The **namespace** or package of the generated code.
- What functions to **inline** (e.g., constructors, assignment operators, regular functions, and so on).
- How to handle **exceptional** conditions (do nothing, perform a callback, throw an exception).

- Whether to convert the value of each variable to a string for debugging purposes: **debug_coordinate_string**.

- Where to output *visual debugging* information, if any: **debug_output**.

- Enable or disable **profiling**.

- How to bind (C++) operators to functions: **cpp_operator_default_bindings** and **bind**.

- Whether to allocate full memory or parity pure for multivectors (the option **multivector_mem_alloc**) and what type of floating point variables to use for coordinates **storage**.

4.6.2 Basis and Metric

An example of the definition of the basis and the metric is:

```
// excerpt from c3ga.gs2 specification file
 basis: no, e1, e2, e3, ni;
 metric: e1.e1 = e2.e2 = e3.e3 = 1;
 metric: no.ni = -1;
```

This example just specifies the metric of the conformal model, as shown in Equation 2.40. The `basis` line of this example specifies that there are five basis vectors in the algebra: `no`, `e1`, `e2`, `e3` and `ni`. The next two `metric` lines specify the metric by listing the non-null inner products between basis vectors. Unspecified inner products are assumed to be zero.

4.6.3 General Multivector Definition

Every algebra implementation contains a *general multivector class* that can hold arbitrary multivector values. The coordinates of the general multivector class are stored in an array using per-grade compression, as was discussed in Section 3.2. The user can specify the name of the multivector class and the order in which the coordinates are stored. Specifying the coordinate-order can be useful when coordinates are passed directly from the array to a library that expects the coordinates to be in a certain order. Figure 4.4 shows an example from our conformal algebra.

4.6.4 Specialized Multivector Types

As stated above, `Gaigen 2` allows the user to statically type multivector variables. This is done by generating classes for each of the multivector types the user requires. Some examples of multivector types are `point`, `sphere` or `EXForm` (short for Euclidean transform), which would be defined as follows in `Gaigen 2`:

```
// excerpt from c3ga.gs2 specification file
  license: GPL;
  language: cpp;
  pass: reference;
  namespace: c3ga;
  inline: constructors set assign operators functions;
  exception: throw;

  debug_coordinate_string: disable;
  debug_output: disabled;
  profile: ;

  cpp_operator_default_bindings;
  bind binary operator & to extract_coord;
  bind binary assign operator & to extract_coord;

  multivector_mem_alloc: parity_pure;
  storage: double '%s';
```

Figure 4.3: *Implementation details as listed in a Gaigen 2 specification* `.gs2` *file.*

```
// excerpt from c3ga.gs2 specification file
  multivector: mv(
    // grade 0:
    1.0,
    // grade 1:
    no, e1, e2, e3, ni,
    // grade 2:
    no^e1, no^e2, no^e3,
    e1^e2, e2^e3, e3^e1,
    e1^ni, e2^ni, e3^ni,
    no^ni,
    // grade 3:
    e2^e3^ni, e3^e1^ni, e1^e2^ni,
    no^e3^ni, no^e1^ni, no^e2^ni,
    no^e2^e3, no^e1^e3, no^e1^e2, e1^e2^e3,
    // grade 4:
    e1^e2^e3^ni,
    no^e2^e3^ni, no^e1^e3^ni, no^e1^e2^ni,
    no^e1^e2^e3,
    // grade 5:
    no ^ e1 ^ e2 ^ e3 ^ ni
  );
```

Figure 4.4: *General multivector definition in* `.gs2` *file. The name of the multivector class is* `mv`*, as specified on the* `multivector` *line. The other lines specify the order of the basis-blade-coordinates.*

```
// excerpt from c3ga.gs2 specification file
  // point:
  specialization: blade point(no, e1, e2, e3, ni);

  // sphere:
  specialization: blade sphere(e1^e2^e3^ni, e1^e2^no^ni,
    e1^e3^no^ni, e2^e3^no^ni, e1^e2^e3^no);

  // EXForm:
  specialization: versor EXForm(1.0,                                    // grade 0
    e1^e2, e1^e3, e2^e3, e1^ni, e2^ni, e3^ni, no^ni,                   // grade 2
    e1^e2^no^ni, e1^e3^no^ni, e2^e3^no^ni, e1^e2^e3^ni);              // grade 4
```

Each specialization defines the name of the type, and lists the required basis blades. These are the basis blades for which the coordinates are non-zero in general. Optionally the user can specify that variables of the defined type should always be `blade`s or `versor`s. The intention is that run-time safety checks on the value of the variables are performed, using the algorithm from Section 3.4.5, but this feature is not implemented yet in `Gaigen 2`.

Basis-blade-coordinates that are not listed in the `specialization` definition are assumed to have a constant value of 0. `Gaigen 2` also supports a generalization of this idea: it allows specified coordinates to have constant values. For example, a normalized point in the conformal model has an o coordinate which is 1 by definition. In `Gaigen 2` a specialized multivector type that takes advantage of this would be defined as:

```
// excerpt from c3ga.gs2 specification file
  specialization: blade normalizedPoint(no = 1, e1, e2, e3, ni);
```

The `no = 1` part of the definition specifies that the o coordinate is always 1. The class generated for this specialization would allocate storage for just the *four* non-constant coordinates. The constant `no` coordinate is not stored. However, whenever a `normalizedPoint` is used in an expression, the correct value for `no` is automatically used during static evaluation, as will be discussed in Section 4.8.5.

4.6.5 Generated Specialized Multivector Types

`Gaigen 2` also generates definitions of specialized multivector types on its own, for the following reasons:

- When the return type of DSL functions is a general multivector, it is automatically changed to the 'tightest' specialized multivector type whenever possible. If this type has not been defined by the user yet, then `Gaigen 2` generates a definition. This will be discussed in Section 4.8.7.

- To initialize the columns of outermorphism matrix representations, each column is given its own specialized type. This allows the columns of these matrices to be used directly as blade variables. This will be discussed in Section 4.7.4.

4.6.6 Constants

Constant multivector values that are used often inside the user application may be encoded as true constants. For example, the unit pseudoscalar in our conformal model example can be defined as:

```
// excerpt from c3ga.gs2 specification file
  constant: I5 = "no ^ e1 ^ e2 ^ e3 ^ ni";
```

This definition causes `Gaigen 2` to generate a special class named `__I5_ct__` for this constant multivector value, and one global variable of this type, named `I5`. Whenever such a constant is used in actual code, the goal of the optimization process is to optimize it away, as described in Section 4.8.6. Hence they serve mostly as symbolic tokens.

4.6.7 Outermorphism Matrix Representations

`Gaigen 2` allows outermorphisms to be represented by matrices. The theory behind this was described in Section 2.12. As with multivectors, two types of *outermorphism classes* can be defined: general and specialized.

The general outermorphism class can be applied to any type of blade. Square transformation matrices are allocated for each grade part, except the scalar part. (The scalar part is not affected by an outermorphism). For a 5-D conformal algebra, this means two 5×5 matrices (grades 1 and 4), two 10×10 matrices (grades 2 and 3) and one 1×1 matrix (grade 5). On initialization of a general outermorphism variable, each of these matrices is initialized such that it can be used to transform a blade of their respective grade. This will be discussed in Section 4.7.4. As with general multivector classes, the definition of the general outermorphism class consists of the name and the order of coordinates. For example:

```
// excerpt from c3ga.gs2 specification file
outermorphism: om(
  // list of basis blades ... (omitted)
);
```

The name of the class is `om`. The list of basis blades was omitted from the listing since it is the same as the list of the general multivector definition.

The general outermorphism class can transform any blade. Specialized outermorphism classes on the other hand can only transform blades from a certain domain to a certain range. For example, a specialized outermorphism class `omFlatPoint` — which can be used to transform conformal flat points — is defined using:

```
// excerpt from c3ga.gs2 specification file
  om_specialization: transpose omFlatPoint(e1^ni, e2^ni, e3^ni, no^ni);
```

The `transpose` keyword is used to specify that the matrix should be stored in transposed form. This can be useful when the matrix is passed to other software.

This class `omFlatPoint` would work particularly well with the graphics library `OpenGL`, as described in Section 11.13 of [12].

The domain and range of `omFlatPoint` (as shown above) are the same. In principle, the domain and range of a specialized outermorphism class can be different. For example, to define an outermorphism class that can only transform flat points that lie on the $o \wedge \mathbf{e}_1 \wedge \infty$-line to flat points which lie on the $o \wedge \mathbf{e}_2 \wedge \infty$-line, one would use:

```
// artificial example from .gs2 specification file
// (e1^ni, no^ni) is the domain, (e2^ni, no^ni) is the range
  om_specialization: strangeOmFlatPoint(e1^ni, no^ni)(e2^ni, no^ni);
```

However, we have not found a practical use for this feature yet.

4.7 Implementation: Boiler Plate Code Generation

Directly from the algebra specification, a significant amount of relatively straightforward 'boiler plate' source code is generated by `Gaigen 2`. This includes:

- Class definitions for the general and specialized multivector classes.
- Class definitions for the general and specialized outermorphism classes.
- Functions for pretty printing multivector values to human readable form.
- `ANTLR` [35] grammars for parsing the human readable input.
- Code for implementing constants.

The actual code generation is performed by a custom back end, designed for a specific target language. Currently, we have one back end for `C++` and one back end for `Java`.

In this section we first discuss the general principles behind the code generation. This is followed by a short discussion of each of the boiler-plate code generation items listed above.

4.7.1 Code Generation Details

Although most of the boiler plate code is pretty straightforward, generating it does require a fair amount of control logic (e.g., to loop over all coordinates of a specialized multivector) and string substitutions (e.g., to output the name of a class). Initially we implemented this control logic in hand-written `Java`: the generated code was simply emitted using `print()` statements. In practice this became unreadable because of the way control logic and emitted code are mixed.

```
// Excerpt from template file

// set to 'coordinates specified'
<%MODE.inlineString()%>void <%SMV.name()%>::set(
  __<%SMV.name()%>_coordinates__
  <%for (int i = 0; i < SMV.nbCoordinates(); i++) {%>
    , Float c_<%SMV.getBE(i).symbolicName(SPEC)%><%}%>) {
    // set coordinates
    <%for (int i = 0; i < SMV.nbCoordinates(); i++) {%>
      m_c[<%i%>] = c_<%SMV.getBE(i).symbolicName(SPEC)%>;
    <%}%>
    <%if (SPEC.isEnabled(DEBUG_COORDINATE_STRING)) {%>ucs();<%}%>
<%if (SPEC.isEnabled(VIZ)) { /* visual debug instrumentation */%>
    const bool assign = true;
    doVizCallbackThis(assign);
<%}%>
}
```

Figure 4.5: *Example of an active template.* `Java` *code which will be executed at code-generation-time is enclosed in* <%...%>*-delimiters. The text which will be emitted verbatim is grey.*

We solved this by switching to *active templates*. These are templates that consist of both plain text (to be emitted verbatim), mixed with `Java` code that is executed at code generation time. The `Java` code handles the control logic and the string substitutions. The plain text can be any type of text. In our case, the plain text consists of bits and pieces of `C++`, `Java` and `ANTLR` code.

Active templates improved the readability considerably, although they are still somewhat cryptic. An example is shown in Figure 4.5. In these templates, the <%... %> delimiters contain the active parts of the templates (written in `Java`); the rest is the plain text. Two types of active parts are used:

- String substitutions. For example `<%SMV.name()%>` just prints the name of the specialized multivector type represented by `SMV`. I.e. `name()` is just a method of the variable `SMV`.
- Control logic. For example
  `<%for (int i = 0; i < SMV.nbCoordinates(); i++)%>`
  loops over all coordinates of the specialized multivector represented by `SMV`.

The particular template shown in Figure 4.5 generates a function that initializes a specialized multivector from its coordinates. An example of the output of the template is shown in Figure 4.7. Of course, the precise output of the template depends on the value of the variables used by the template (`MODE`, `SMV`, `SPEC`). These variables are passed as arguments to the template.

To execute the templates, they must first be compiled using a simple 'template compiler'. The template compiler is just another `Java` program that parses the

```
// Generated Java source code (which generates source code).
public static void emit(subspace.cog.IndentationEngine __E__,
            subspace.cog.CoG COG,
            subspace.g2.cpp.CppCodeGenerationMode MODE,
            subspace.g2.Specification SPEC,
            subspace.g2.SpecializedMVType SMV)
{
  __E__.indent("// set to 'coordinates specified' \n");
  __E__.indent("\n");
  __E__.indent("" + (MODE.inlineString()));
  __E__.indent("void ");
  __E__.indent("" + (SMV.name()));
  __E__.indent("::set(__");
  __E__.indent("" + (SMV.name()));
  __E__.indent("_coordinates__");

  for (int i = 0; i < SMV.nbCoordinates(); i++) {
    __E__.indent(", Float c_");
    __E__.indent("" + (SMV.getBE(i).symbolicName(SPEC)));
  }
  __E__.indent(") {\n");
  __E__.indent("\n");
  __E__.indent("\t");
  __E__.indent("// set coordinates\n");
  __E__.indent("\n");
  __E__.indent("\t");

  for (int i = 0; i < SMV.nbCoordinates(); i++) {
     __E__.indent("\t");
     __E__.indent("\t");
     __E__.indent("m_c[");
     __E__.indent("" + (i));
     __E__.indent("] = c_");
     __E__.indent("" + (SMV.getBE(i).symbolicName(SPEC)));
     __E__.indent(";\n");
     __E__.indent("\n");
     __E__.indent("\t");
  }
  __E__.indent("\t");

  if (SPEC.isEnabled(DEBUG_COORDINATE_STRING)) {
     __E__.indent("ucs();");
  }
  __E__.indent(" \n");
  __E__.indent("\n");

   if (SPEC.isEnabled(VIZ)) { /* visual debug instrumentation */
     __E__.indent("\t");
     __E__.indent("const bool assign = true;\n");
     __E__.indent("\n");
     __E__.indent("\t");
     __E__.indent("doVizCallbackThis(assign);\n");
     __E__.indent("\n");
  }
  __E__.indent("}\n");
}
```

Figure 4.6: *The result of compiling Figure 4.5.*

```
// Generated C++ source code

// set to 'coordinates specified'
inline void vectorE3GA::set(__vectorE3GA_coordinates__,
   Float c_e1, Float c_e2, Float c_e3)
{
  // set coordinates
  m_c[0] = c_e1;
  m_c[1] = c_e2;
  m_c[2] = c_e3;
}
```

Figure 4.7: *One example of output of the active template in Figure 4.5.*

templates (finding all the <%... %> parts), and converts them into regular `Java` source code. This parser can see the difference between method calls that return strings and control logic. The template in Figure 4.5 is compiled into the code shown in Figure 4.6.

When it is time for `Gaigen 2` to generate the final code, the compiled templates are run with the required arguments, resulting in the type of code shown in Figure 4.7. All code discussed in the following sections (up to Section 4.8) is generated using this template method.

4.7.2 Implementation of the General Multivector Class

The goal of the general multivector class is twofold:

- As an alternative to the specialized multivector classes: Users who cannot — or do not want to — statically type their multivector variables, can use the general multivector class. Statically typing multivector variables is simply not possible for some algorithms where the multivector types of variables change during execution (such as `meet` and `join`, Section 3.4.7). Also, some users may deliberately choose to not statically type their variables, because they simply do not care about performance.

- As a fallback option during profiling: For optimal performance, algebra implementations should be adapted to the rest of the application using profiling, which will be discussed in Section 4.9. As long as this adaptation is not complete, some functions over the algebra may be defined only for general multivectors. When such functions are invoked with specialized multivectors as actual parameters, these specialized multivectors are automatically converted to general multivectors before being passed to the functions. In essence, the general multivector implementation is used as the (slow) fallback option for functions that have not been instantiated for the right specialized multivector types yet.

We should therefore provide a general multivector class that is at least reasonably efficient. What type of coordinate compression (Section 3.2) is used and how that compression is handled during computations is the most important factor in this. For `Gaigen 2` we use per-grade compression. We had previously shown that per-grade compression is reasonably efficient (see e.g. [19]), and large parts of the code generation process can be shared with the specialized multivector classes.

Class definition, floating point type and coordinate storage

The multivector class is basically just a container for the (compressed) coordinates, plus constructors and some other functions. In the conformal algebra used for the ray tracer, the general multivector class looks like:

```
// generated C++ code
class mv {
public:

typedef double Float;

   // constructors (omitted)
   // set function (omitted)
   // assignment operators (omitted)
   // compression and expansion functions (omitted)
   // largest coordinate functions (omitted)
   // coordinate extraction functions (omitted)
   // pretty printing functions (omitted)

   Float m_c[16];            // coordinate storage
   unsigned int m_gu;        // grade usage
};
```

We have omitted the functions, as these are discussed below. At the start of the class definition, the `Float` type of the multivector is specified. In this example, we use `double` precision (64-bit) floating point values, as was specified in the `.gs2` specification in Figure 4.3.

The last two lines of the class define the member variables used for coordinate storage. The coordinates are stored in the statically allocated `m_c[]` array. Only 16 (as opposed to 32) coordinates are allocated in the example since `parity_pure` storage was specified. The integer `m_gu` is used as the *grade usage bitmap*. For each grade, a bit of the integer is used to indicate the presence or absence of that grade part. If bit 0 of `m_gu` is 1, then the grade 0 (scalar) coordinate is stored in the first entry of `m_c[]`. If bit 1 is 1, then the grade 1 coordinates are stored in the next five free entries, and so on.

Note that if the multivector is not parity pure, then more than 16 coordinates must be stored in `m_gu` and an error occurs because only 16 entries have been allocated. Run-time checks can be made to avoid catastrophic consequences. However, with decent use of geometric algebra (e.g., not summing even and odd

blades or versors, which is a geometrically meaningless operation), this situation should never occur.

In `Gaigen 1` we offered the option to dynamically allocate the array `m_c[]`. `Gaigen 1` would then allocate memory as needed to reduce memory usage (i.e., most conformal multivectors require less than 16 coordinates). In our experience this option was not used much because it was slower than static allocation. Moreover, the general multivector implementation in `Gaigen 2` is less important than in `Gaigen 1`. This is because in `Gaigen 2` specialized multivector types should be used for computationally intense work, while the original `Gaigen 1` had *only* the general multivector implementation. Hence we decided not to implement dynamic memory allocation in `Gaigen 2`.

Compression and expansion

Functions are provided to compress and expand the coordinates of the general multivector class.

The coordinate *compression* function is used when a user supplies coordinates that are not compressed to initialize a general multivector, and when the result of a computation must be stored in a general multivector variable. (An example of the latter will be shown at the bottom of Figure 4.16, at the end of the function.) The compression function works by simply checking for each grade parts whether it is all zero or not. Grade parts that are zero are not stored and the grade usage bitmap is set accordingly.

The coordinate *expansion* function expands the general multivector to an array of $n + 1$ pointers to `Float` arrays. It is used by DSL functions when they handle general multivectors; an example of this is shown at the top of Figure 4.16

Constructors, assignment operators and `set()` functions

Constructors are functions that create instances of a particular class. They should initialize the instance to some valid state value according to the provided arguments. The general multivector class has constructors for:

- Initializing to 0.
- Initializing to the value of some other general multivector (copy constructor).
- Initializing to specified non-compressed coordinates. The coordinates are passed in an array. The compressions function are used to compress the coordinates.
- Initializing to specified compressed coordinates. The user supplies the grade usage bitmap and the compressed coordinates.

- Initializing from a specialized multivector. One constructor is supplied for each type of specialized multivector. These *converting constructors* are important because they allow functions over general multivectors to be invoked with specialized multivector values.

- Initializing from a constant multivector. These converting constructors are again required for invoking functions over general multivectors with constant values as arguments.

The assignment operators make sure that you can write `a = b` for multivector variables `a` and `b`. With them you can assign any multivector value (general or specialized) to a general multivector variable.

The `set()` functions allow one to set the value of an existing general multivector variable to a particular value, as in

```
a.set(b); // sets a to value of b
```

The `set()` functions are available in the same variants as the constructors. In fact, most constructors and all assignment operators simply invoke the respective `set()` function to do the real work, so this functionality is provided only in one place.

Figure 4.8 shows an interesting example of a `set()` function. It converts the specialized multivector type `normalizedTranslator` to a general multivector. A normalized translator is of the form

$$\mathbf{T} = 1 + c_0\,\mathbf{e}_1 \wedge \infty + c_1\,\mathbf{e}_2 \wedge \infty + c_2\,\mathbf{e}_3 \wedge \infty.$$

So it has a grade 0 (scalar) and a grade 2 part. The scalar part has the constant value '1' due to normalization. All this is clearly visible in the function: The grade part usage is set to $2^0 + 2^2 = 1 + 4 = 5$ by the line `gu(5);`. The scalar coordinate of the general multivector is set to '1' using the line `m_c[0] = (Float)1.0;`. The three non-constant coordinates are copied near the end of the function. Note that most of the grade 2 coordinates are set to 0 because the grade 2 part of a 5-D multivector has ten coordinates.

One may verify that the coordinates are copied to the correct entries using Figure 4.4 which specifies the coordinate order for `mv` and the definition of the `normalizedTranslator`, which reads:

```
// excerpt from c3ga.gs2 specification file
  specialization: versor normalizedTranslator(1.0 = 1, e1^ni, e2^ni, e3^ni);
```

Largest coordinate function

For a number of algorithms, it is useful to efficiently extract the absolute largest coordinate of a multivector. Examples are blade factorization (Section 3.4.6), and the normalization technique used for exponentiation (Section 3.4.3). Hence

```
// Generated C++ code
// set to normalizedTranslator
inline void mv::set(const normalizedTranslator & arg1) {
  // set grade usage
  gu(5);

  // set coordinates
  m_c[0] = (Float)1.0;
  m_c[1] = (Float)0;
  m_c[2] = (Float)0;
  m_c[3] = (Float)0;
  m_c[4] = (Float)0;
  m_c[5] = (Float)0;
  m_c[6] = (Float)0;
  m_c[7] = arg1.m_c[0];
  m_c[8] = arg1.m_c[1];
  m_c[9] = arg1.m_c[2];
  m_c[10] = (Float)0;
}
```

Figure 4.8: *Example of a* `set()` *function. This function sets a general multivector variable (type* `mv`*) to the value of specialized multivector variable (type* `normalizedTranslator`*).*

`Gaigen 2` generates a function which returns this coordinate, plus a variant that also returns the basis blade bitmap corresponding to this coordinate (required for blade factorization). These functions are also supplied for all specialized multivectors.

4.7.3 Implementation of the Specialized Multivector Classes

Specialized multivector classes are very simple containers for coordinates. They provide a subset of the functionality of the general multivector class. An example is of a specialized multivector class is:

```
// generated C++ code
class normalizedTranslator {
public:
  typedef double Float;
  // constructors and functions (omitted)

  // coordinate storage
  Float m_c[3];
};
```

`m_c[3]` holds the three non-constant coordinates of the `normalizedTranslator`. The `set()` functions allow a specialized multivector to be set 0, a copy of another specialized multivector, to specified coordinates, or to a general multivector value. Because specialized multivector classes are basically containers for coordinates,

```
// Generated C++ code
// Set normalized point to mv
inline void normalizedPoint::set(const mv & arg1) {
  int gidx = 0;

  if (arg1.gu() & 1) gidx += 1;

  if (arg1.gu() & 2) {
    m_c[0] = arg1.m_c[gidx + 1];
    m_c[1] = arg1.m_c[gidx + 2];
    m_c[2] = arg1.m_c[gidx + 3];
    m_c[3] = arg1.m_c[gidx + 4];
  }
  else {
    m_c[0] = (Float)0.0;
    m_c[1] = (Float)0.0;
    m_c[2] = (Float)0.0;
    m_c[3] = (Float)0.0;
  }
}
```

Figure 4.9: *Converting a general multivector to a `normalizedPoint`.*

there is not much to say about them at this point. We only discuss conversion from general multivectors to specialized multivectors.

Converting from Specialized to General Multivector Variables

Figure 4.9 shows an example of how a general multivector is converted to a specialized multivector. Notice how the function skips the scalar coordinate when it is present, and copies the `e1`-, `e2`-, `e3`- and `ni`-coordinates when grade 1 is present. The `no`-coordinate is ignored, so the function assumes the `no` coordinate is '1'; safety checks could in principle be implemented to verify this at run-time. The coordinates for grade 2 and higher are also ignored. Again run-time safety checks could be implemented to verify that all these coordinates are indeede 0.

Underscore Constructors

In `C++`, there is a problem with automatic conversion from general multivectors to specialized multivectors. We provide a converting constructor from specialized to general. If we also provided the opposite (general to specialized), the compiler would complain about ambiguity in many cases. So we have opted to not provide these converting constructors. Instead, we provide what we have called *underscore constructors*. These are just regular functions, but their name is an underscore plus the name of the specialized multivector class. These must be invoked explicitly by the user (the programmer using `Gaigen 2`), thus avoiding the ambiguity. An example of an understore constructor is:

```
// Generated C++ code
inline normalizedPoint _normalizedPoint(const mv &arg1) {
  return normalizedPoint(arg1, 0);
}
```

All this function does is forward the call to a real constructor. This real constructor has an extra integer argument (again to avoid ambiguities).

Although the underscore constructors are just a necessary hack to prevent compilation problems, they are useful from time to time to extract certain parts of multivector variables. For example, to extract the base space part of a point, one may use

```
vectorE3GA v = _vectorE3GA(somePoint);
```

The underscore constructor `_vectorE3GA` drops the o and ∞ coordinates. The formally sound alternative to this hack would be using a projection like

```
vectorE3GA v = somePoint << I3 * I3i;
```

where `I3` $= \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$ and `I3i = inverse(I3)`. In practice, we find the former solution clearer even though it not proper use of geometric algebra.

4.7.4 Implementation of the General Outermorphism Class

The general outermorphism class is a container for transformation matrices – one for each grade, as discussed in Section 4.6.7. The entries of these matrices are all stored in one big array. The implementation of the general outermorphism class of our conformal algebra looks like:

```
// generated C++ code
class om {
public:
   typedef double Float;
   // constructors, functions, etc (omitted)

   Float m_c[251]; // matrix entry storage
};
```

Note that in this example 251 entries are allocated for the `m_c` array in order to store the matrices for each grade, except the scalar grade because scalars are not affected by linear transformations of the vector space. So this gives:

$$5 \times 5 + 10 \times 10 + 10 \times 10 + 5 \times 5 + 1 = 251.$$

Constructors and `set()` functions are provided to initialize / set the class to identity, to images of basis vectors and to specialized outermorphism types. We now discuss initialization from images of basis vectors.

Initialization from images of basis vectors

The most straightforward way for a user to specify a linear mapping is through the images of the basis vectors. `Gaigen 2` generates functions to initialize outermorphism classes using these images. The user simply constructs the n images of the basis vectors by applying the transformation to each of the basis vectors, and then passes them to the outermorphism class, which uses them to initialize its matrices.

As explained in Section 2.12, each column of each matrix of the general outermorphism class holds the image of a specific basis blade. `Gaigen 2` generates efficient code for implementing this. The idea is to initialize the grade 1 matrix first, and work up to grade 2, 3 and so on, forming the required basis-blade-images along the way. Each time an existing basis-blade-image is required as an intermediate result, it is re-used.

The actual code for our conformal algebra counts far to many lines to print here, so instead we show a pseudo-code example (initialization code for for a specialized outermorphism is much shorter, and we do show an actual example of that in the next section):

```
// pseudo code example, written by hand

// input: images of basis vectors (ino, ie1, ie2, ie3, ini)
// output: columns of outermorphism matrices (m1_c1, m1_c2, and so on)

// initialize grade 1 matrix columns:
   m1_c1 = ino;    // image of no
   m1_c2 = ie1;    // image of e1
   m1_c3 = ie2;    // image of e2
   m1_c4 = ie3;    // image of e3
   m1_c5 = ini;    // image of ni

// initialize grade 2 matrix columns:
   m2_c1 = m1_c1 ^ m1_c2;   // image of (no ^ e1)
   // . . .
   m2_c5 = m1_c3 ^ m1_c4;   // image of (e2 ^ e3)
   // . . .

// initialize grade 3 matrix columns:
   m3_c1 = m2_c5 ^ m1_c5;   // image of ((e2^e3) ^ ni)
   // . . .

// initialize grade 4,5 matrix columns:
   // ...

// Done: return results.
```

The pseudo-code example starts by copying the supplied images (`ino`, `ie1`, and so on) to the respective grade-1-matrix columns (`m1_c1`, `m1_c2`, and so on). After that, the grade 2 matrix is initialized (only 2 our of 10 lines are shown). The grade 2 images are computed as the outer product of the columns of the grade 1

matrix. Then the grade 3 matrix is initialized, re-using existing grade 1 and grade 2 images (only 1 out of 10 lines is shown). I.e., the first column of the grade 3 matrix is the outer product of the fifth column of the grade 2 and the fifth column of the grade 1 matrix.

4.7.5 Implementation of Specialized Outermorphism Classes

While the general outermorphism class is efficient at applying transformations to blades, the class allocates a lot of memory because matrices for all grade parts are stored. It also takes time to initialize the general outermorphism variables because all these matrices must be computed (the (relative) cost of initialization depends on how many times a particular outermorphism will be applied). A user may want to apply an outermorphism just to one specific type of blade (e.g., only to flat points), in which case the general outermorphism class is a bit of an overkill.

For that reason, specialized outermorphism classes were introduced. These store only a transformation matrix from a specified domain to a specified range, and thus require less memory – and less time — to be initialized. On top of that, the entries of the matrix are packed closer in memory, which is also beneficial for (cache-) performance. Another advantage is that these matrices can easily be exchanged with other application or libraries.

A nice example of the latter is the `omFlatPoint` example we used earlier. This specialized outermorphism class transforms conformal flat points of the form $\mathbf{p} \wedge \infty$, and stores this transform in a 4×4 matrix. The graphics library `OpenGL` uses the same type of matrices to represent transformations, so they can be passed back and forth between `OpenGL` and `omFlatPoint` with ease. (The flat points of the conformal model are essentially the regular points of the homogeneous model, which `OpenGL` uses).

The implementation of such a specialized outermorphism class looks much like a general outermorphism class, except that it allocates much less matrix entries. I.e., the `m_c` array of the `omFlatPoint` requires just 16 entries to hold the 4×4 matrix:

```
// Generated C++ code
class omFlatPoint {
public:
typedef double Float;
  // constructors, functions, etc (omitted)

  Float m_c[16]; // matrix entry storage
};
```

Initialization from images of basis vectors

Outermorphism classes are typically initialized from images of basis vectors or basis blades. Figure 4.10 shows a function that initializes an instance of `omFlatPoint`

```
// generated C++ code (edited for readability)
inline void set(omFlatPoint& OM,
  const point& ie1, const point& ini,
  const point& ie2, const point& ie3, const point& ino)
  {
  OM.m_c[0] = ((ini.m_c[4] * ie1.m_c[1]) + (-1.0 * ini.m_c[1] * ie1.m_c[4]));
  OM.m_c[1] = ((ini.m_c[4] * ie1.m_c[2]) + (-1.0 * ini.m_c[2] * ie1.m_c[4]));
  OM.m_c[2] = ((-1.0 * ini.m_c[3] * ie1.m_c[4]) + (ini.m_c[4] * ie1.m_c[3]));
  OM.m_c[3] = ((ini.m_c[4] * ie1.m_c[0]) + (-1.0 * ini.m_c[0] * ie1.m_c[4]));
  OM.m_c[4] = ((ie2.m_c[1] * ini.m_c[4]) + (-1.0 * ie2.m_c[4] * ini.m_c[1]));
  OM.m_c[5] = ((ie2.m_c[2] * ini.m_c[4]) + (-1.0 * ie2.m_c[4] * ini.m_c[2]));
  OM.m_c[6] = ((ie2.m_c[3] * ini.m_c[4]) + (-1.0 * ie2.m_c[4] * ini.m_c[3]));
  OM.m_c[7] = ((ie2.m_c[0] * ini.m_c[4]) + (-1.0 * ie2.m_c[4] * ini.m_c[0]));
  OM.m_c[8] = ((ie3.m_c[1] * ini.m_c[4]) + (-1.0 * ie3.m_c[4] * ini.m_c[1]));
  OM.m_c[9] = ((-1.0 * ie3.m_c[4] * ini.m_c[2]) + (ie3.m_c[2] * ini.m_c[4]));
  OM.m_c[10] = ((-1.0 * ie3.m_c[4] * ini.m_c[3]) + (ie3.m_c[3] * ini.m_c[4]));
  OM.m_c[11] = ((-1.0 * ie3.m_c[4] * ini.m_c[0]) + (ie3.m_c[0] * ini.m_c[4]));
  OM.m_c[12] = ((ino.m_c[1] * ini.m_c[4]) + (-1.0 * ino.m_c[4] * ini.m_c[1]));
  OM.m_c[13] = ((-1.0 * ino.m_c[4] * ini.m_c[2]) + (ino.m_c[2] * ini.m_c[4]));
  OM.m_c[14] = ((-1.0 * ino.m_c[4] * ini.m_c[3]) + (ino.m_c[3] * ini.m_c[4]));
  OM.m_c[15] = ((ino.m_c[0] * ini.m_c[4]) + (-1.0 * ino.m_c[4] * ini.m_c[0]));
}
```

Figure 4.10: *Specialized outermorphism class initialization. The result is returned in* `OM`. `ie1`, `ini`, `ie2`, `ie1`, `ie3`, `ino` *are the images of basis vectors. (Variable names were edited for readability).*

from the images of the five basis vectors. The function in this figure may be the first convincing example in this chapter that one would prefer not to write this type of code by hand.

`Gaigen 2` implements these functions by generating the appropriate DSL function that describes the initialization in the high-level coordinate-free DSL language. In later stages of the code generation process, `Gaigen 2` converts this DSL function to regular code.

4.7.6 Parsing and Pretty Printing of Multivectors

Parsing and pretty printing are each other's opposites. A parser converts human readable text to a representation that is efficient for a computer. A pretty printer converts the computer representation to human readable form. We provide both parsing and pretty printing for multivector variables.

In the case of `Gaigen 2`, pretty printing may be used to present multivector values to humans, or to store them in text-based formats such as XML. Parsing may be used to parse human input, or to read pretty printed text back into a program (e.g., from an XML file).

Pretty Printing

Pretty printing multivector values is straightforward. Only the non-zero coordinates are printed, together with their respective basis blades. For example, a 3-D rotor may look like:

```
0.41 - 0.68*e2^e3 - 0.17*e3^e1 + 0.47*e1^e2
```

The precision of floating point numbers can be specified using the regular format specifiers (we used `"%1.2f"` in the above example). When a coordinate would show up as `0` with respect to that precision, it is not printed.

The main pretty printing function only prints out general multivector variables. Specialized multivector variables are pretty printed by converting them to general multivectors first (performance is not much of an issue here since pretty printing is mostly I/O-bound).

Parsing

As usual, parsing is more complicated than pretty printing. The multivector parser can parse the output of the pretty printer, and also somewhat more general input.

To facilitate parsing of multivector strings, `Gaigen 2` generates `ANTLR` grammars (lexers and parsers). The lexer splits the input into tokens: numbers, basis vectors, white space and some operators ($+, -, *, \wedge$). The parser combines these, first into scaled basis blades, and then into multivector values. The resulting parser implementations always return a general multivector, but these can easily be converted to specialized multivectors using an underscore constructor.

The `ANTLR` grammars do not differ much between algebras, so generating them is a bit superfluous. Only superficial differences between algebras influence the grammars, such as namespaces, the class name of the general multivector type, the number and names of basis vectors, the floating point type, and so on. We already had the code generation framework setup, so generating the grammars was the simplest way to implement these parsers.

The generated grammars must be compiled by `ANTLR`, which transforms them into regular `C++` or `Java` code.

4.8 Implementation: DSL Functions

Up to this point, we have discussed how an algebra is defined in `Gaigen 2`, and how the generated implementation handles such basic tasks as storage of multivectors, initializing matrix representations of outermorphisms, pretty printing and parsing.

In this section, we discuss the more interesting part of `Gaigen 2`: the definition and implementation of functions over the algebra. These functions are defined in

a high-level coordinate-free language, and Gaigen 2 converts them to low-level coordinate-based C++ or Java.

The sole purpose of the Gaigen 2 Domain Specific Language (DSL) is to define functions over the algebra which should be converted to functions in the output language. An example of a function written in the DSL is:

```
// DSL function, written by hand
bivectorE3GA op(vectorE3GA x, vectorE3GA y) {
  return ::g2bi::op(x, y);
}
```

All this function does is define a function op() that takes two vectorE3GA arguments and returns the outer product of these two arguments. The outer product is 'computed' by calling the function ::g2bi::op(), which is a built-in function of the DSL.

The idea is that Gaigen 2 converts this DSL function to C++ or Java code. It does so by expanding the function to the level of coordinates, performing static analysis, and converting the result to the output language. The generated C++ code for the example would be:

```
// Generated C++ code
inline bivectorE3GA op(const vectorE3GA& x, const vectorE3GA& y) {
  return bivectorE3GA(bivectorE3GA_e1e2_e2e3_e3e1,
    ((x.m_c[0] * y.m_c[1]) + (-1.0 * x.m_c[1] * y.m_c[0])),
    ((x.m_c[1] * y.m_c[2]) + (-1.0 * x.m_c[2] * y.m_c[1])),
    ((-1.0 * x.m_c[0] * y.m_c[2]) + (x.m_c[2] * y.m_c[0])));
}
```

In this example we have used specialized multivectors (vectorE3GA) as arguments types. In practice, one would use general multivectors as argument types, and use profiling to instantiate the general function with specialized arguments. Profiling will be discussed in Section 4.9.

Inside Gaigen 2, the processing of DSL functions works like a pipeline (with a loop at the end if profiling is used), and we will discuss the inner workings in the order of the pipeline:

1. First we discuss the grammar of the DSL and how it is parsed.

2. Then we briefly discuss high-level optimization of DSL function (or the lack thereof).

3. This is followed by writing out the functions on the basis and low-level optimization.

4. We conclude with code emission in the target language.

```
<input> ::= (<function> | <namespace>)* ;

<namespace> ::= "namespace" <identifier> "{" (<function> | <namespace>)* "}" ;

<function> ::= <type> <identifier>
        "(" (<formal argument> (",", <formal argument>)* )? ")"
        "{" <statement list> "}" ;

<statement list> ::= <statement>* ;

<statement> ::=
        <if-else stmt> |
        <for stmt> |
        <while stmt> |
        <do stmt> |
        <return stmt> |
        <variable declaration> |
        ( <expression> ";" ) ;


<variable declaration> ::= <type> <variable declaration list>
        ( "," <variable declaration list> )* ";" ;

<variable declaration list> ::= <identifier> ( "=" <expression> )? ;

<expression> ::= (
        identifier | literal | operator |
        ( "(" <expression> ")" ) |
        ( "[" <expression> "]" )
        )* ;
```

Figure 4.11: *Partial grammar of the* `Gaigen 2` *DSL language. See text for details.*

4.8.1 Language Description

The `Gaigen 2` DSL is a simple language with a C-like syntax. The most important part of its grammar is shown in Figure 4.11. The grammar shows that the language allows only functions and namespaces at the top-level input. Functions are made up of statement lists, where individual statements can be the familiar C control statements (such as `for` and `if-else`), and expressions.

Expressions are parsed as lists of literals, identifiers operators. This is done because of the special way we parse expressions, which is explained in more detail in Section 4.8.3.

Literals are floating point numbers, boolean values, and string literals. Operators are the same symbols which are operators in C. Types are the regular types in C (or `C++`), such as `int` and `bool`, plus `string`s and all the multivector types defined in the algebra specification.

Built-in functions

The DSL language has a large number of built-in functions for doing geometric algebra. Some examples are addition, all linear products, grade extraction, reversion, negation, grade involution and dualization (with respect to the full space). The regular scalar operations (such as addition, multiplication, division, square root, cosine, and so on) are also available. Many of these built-in functions are mapped to operators. For example the $\wedge$ operator maps to the outer product when applied to multivectors.

Some Examples of DSL functions

Figure 4.12 shows some more functions that can be defined in the DSL. Although these functions are usually hand-written by the user of `Gaigen 2`, `Gaigen 2` itself also generates some of them. For example the DSL functions that initialize the columns of an outermorphism matrix are generated (see Section 4.7.5).

Unfortunately we have not gotten to implementing larger DSL functions than the ones which are shown here. Initially we set out to also implement larger algorithms such as blade factorization and `meet` and `join` in the DSL language.

4.8.2 DSL Functions Parsing

`Gaigen 2` parses the DSL language in a two-step process. First the overall structure of the language is parsed using an `ANTLR` [1] parser, after which the result is refined to form the final abstract syntax tree (AST).

This two-step parsing process was used such that we could re-use the first parsing step for another program. This program uses an interpreted language

[1]This parser is of course distinct from the `ANTLR` parser that `Gaigen 2` generates to parse human readable multivector strings.

```
// Hand-written Gaigen 2 DSL functions

// Returns outer product of 'x' and 'y'
mv op(mv x, mv y) {
  return x ^ y;
}

// Returns unit of 'x' under reverse norm
mv unit_r(mv x) {
  scalar r2 = ::g2bi::scp(x, ::g2bi::reverse(x));
  scalar ir = ::g2bi::inverse(::g2bi::scalar_sqrt(::g2bi::scalar_abs(r2)));
  return x ir;
}

// Extracts the 'y'-coordinate of 'x'
mv extract_coord(mv x, mv y) {
  return ::g2bi::scp_em(x, ::g2bi::reverse(y));
}

// Applies a versor to some multivector 'x', given the versor 'v' and its inverse 'vI'
mv apply_versor(mv v, mv x, mv vI) {
  if (::g2bi::packed(x)) return v x vI;
  else return ::g2bi::select_grade_by_bitmap(v x vI, ::g2bi::grade_usage_bitmap(x));
}
```

Figure 4.12: *Some examples of DSL functions.*

```
<input> ::= (<statement list> | <namespace>)* ;

<namespace> ::= "namespace" <identifier> "{" (<statement list> | <namespace>)* "}" ;

<statement list> ::= <statement>* ;

<statement> ::=
        <if-else stmt> |
        <for stmt> |
        <while stmt> |
        <do stmt> |
        <return stmt> |
        <try-catch stmt> |
        <statement list> |
        <expression> ";" ;

<expression> ::=
        (
        <operator> |
        <literal> |
        <identifier> |
        <function argument> |
        <array index> )* ;

<function argument> ::= "(" <expression> ")" ;
<statement list>    ::= "{" <statement>* "}" ;
<array index>       ::= "[" <expression> "]" ;


<for stmt> ::= "for" "(" <expression> ";" <expression> ";" <expression> ")"
        (<statement list> | ( <expression> ";" ) ) ;
```

Figure 4.13: *Partial grammar of the Curly Bracket Language Parser (CBLP).*

whose syntax is similar to the `Gaigen 2` DSL, but dissimilar enough that they cannot share the whole grammar.

The first parsing step is implemented as the *Curly Bracket Language Parser* (CBLP). This parser accepts any C-like language that uses curly brackets to denote statement lists. Figure 4.13 shows a simplified grammar of the CBLP; Only the `for stmt` is shown as an example of the other statements (such as cdwhile, `if-else` and so on). Using this grammar, a function like

```
int func(int f) {
  return f * f;
}
```

is analyzed as an expression `int func` followed by a function argument `(int f)`, followed by a statement list containing one statement, a `return` statement.

The CBLP recognizes only keywords required for statements (`for`, `if-else`, and so on) and literals (like `true` and `false`). Type keywords such as `int`, `float` are not recognized, because they depend on the actual language. So, to the CBLP, `int func` looks like an expression consisting of two identifiers. The output of the

CLBP is stored in an intermediate AST.

The second parsing step is actually more of an analyzer than a parser. It traverses the AST returned by CBLP, and analyzes it according to the grammar of the `Gaigen 2` DSL. In the process it transforms the AST into another AST. This second parser is a regular `Java` program, written by hand — not an `ANTLR`-based parser.

4.8.3 DSL Expression Parsing

Another reason for using a two-step parsing process is that the grammar for expressions is not context free. We wanted operator symbols to be intuitive for geometric algebra, but also keep backwards compatibility with C-like programming languages. Hence the meaning and precedence of operators depends on their context.

For example, in C-like programming languages the wedge symbol $\wedge$ means *exclusive or* and has a low precedence. In geometric algebra the same symbol means *outer product* has the highest precedence. With respect to parsing this means that in a C context, we parse an expression like $1 + a \wedge b$ as

$$\mathsf{1 + a{\wedge}b = (1{+}a) \wedge b,}$$

whereas in a geometric algebra context one would want this to be parsed as

$$\mathsf{1 + a{\wedge}b = 1 + (a{\wedge}b).}$$

To implement this, we created a custom parser for expressions called the *Free Expression Parser*.

Besides regular operator overloading (different meaning of symbols in a different context), this parser also allows for multiple precedences for the same symbol. The precedence depends on the surrounding operands. For example when a $\wedge$-symbol is surrounded by scalar operands, it means *exclusive or* and has a low precedence. But when one of the operands is a multivector variable, the precedence is high and the meaning becomes outer product.

We implemented this as follows. The DSL parser accepts lists of operators, literals identifiers and expressions in (round or square) brackets as 'expressions'. In order to refine these rough expressions, first of all function applications (like `f()`), array addressing (like `a[]`) and namespace resolution (`::x::y`) sub-expressions are isolated. Then the type of each literal, identifier and sub-expression is determined. I.e., the raw input expressions are transformed to lists of types and operators. These lists are sent to the Free Expression Parser.

An instance of the Free Expression Parser is created by specifying the operators it has to handle. Operators can be unary, binary or ternary. An example of the latter is the `?:` operator in C. Each operator specification contains a rule that specifies to which types it can bind, what the precedence level is, to what function the operator resolves. When all operators have been specified, the Free Expression

Parser checks if the operator specification is consistent to avoid ambiguity at parse-time. For example, one cannot have the same unary operator bind both to the left and to the right at the same precedence level. All operators specifications are stored in a table.

To parse the lists of operators and types, the Free Expression Parser builds a priority queue. The entries of the queue are operators from the list that can be bound to neighboring types (the operands) The higher the operator precedence, the higher up in the queue it is placed. Only those operators that can be bound are put in the queue. For example, the center `+` operator in the following expression

```
a++ + ++b
```

cannot be bound because it is surrounded by other operators. However, at some point the example expression will have been parsed to

```
(a++) + (++b)
```

and then the `+` operator can bind (if types and rules permit).

The parser starts processing the operators on the priority queue, binding operators to types. Each time an operator is bound to it operand(s), the priority queue is updated according to the changes: neighboring operators may become valid, and operators already on the queue may become invalid.

The parser also allows for white space to be an operator. When it finds two operands in the list without any operators between them, it assumes the space operators is to be applied, and adds it to the priority queue according to the operator table.

The end result is a AST of the expression. For the example expression above we would get:

```
a++ + ++b = ((a++) + (++b)) =
  ::g2bi::add(::g2bi::post_increment(a), ::g2bi::pre_increment(b))
```

Of course, the parsing process can halt if an expression is invalid given the operator specifications. For example, if `a∧∧b` has no meaning according to the operator table, an error is reported, as with regular parsers.

The Free Expression Parser can be extended at run-time. That is, new operators can be added or existing ones removed. This feature is not used by `Gaigen 2`, but is used by the successor to GAViewer, which can (un-)load new classes at run-time. These classes may define their own new operators at the time when they are loaded.

4.8.4 High-Level Optimization of DSL Functions

The original intent was for `Gaigen 2` to apply custom high-level optimization passes to the DSL functions, once they were fully parsed and stored in an AST. One example of such an optimization could be to try to discover parts of a geometric computation that are performed in a subspace of the full space, and use

this knowledge to optimize the computation. However, due to time constraints we were not able to implement any high-level optimizations.

4.8.5 Expanding Multivector Variables on the Basis

The Gaigen 2 implementation works with an the additive multivector basis, so at some point the coordinate-free expressions of the DSL must be expanded to the level of coordinates. The first step in achieving this goal is replacing all multivector variables (general, specialized, constant) by their equivalent value on the basis.

For example, take the following function, which computes the outer product of two normalized points:

```
pointPair op(normalizedPoint a, normalizedPoint b) {
  return a ^ b;
}
```

After parsing, this function becomes:

```
pointPair op(normalizedPoint a, normalizedPoint b) {
  return ::g2bi::op(a, b);
}
```

Replacing the variables a and b by their sum of basis blades results in:

```
pointPair op(normalizedPoint a, normalizedPoint b) {
  return ::g2bi::op(1*no + a.c0*e1 + a.c1*e2 + a.c2*e3 + a.c3*ni,
                    1*no + b.c0*e1 + b.c1*e2 + b.c2*e3 + b.c3*ni);
}
```

Note that the no basis blade has a constant scalar coordinate (i.e., 1*no), while the other coordinates refer to the respective coordinate entry of their variable (e.g., a.c0*e1). This is also the place where constant multivector values 'disappear': they are converted to a constant sum of basis blades, and no reference to their coordinates is made. For example, I5 would be converted to 1*no∧e1∧e2∧e3∧ni and any reference to the original object I5 would disappear.

Outermorphism Matrix Representations

Since outermorphism matrix representations are inherently coordinate-based, they can not be expanded on a basis as multivectors can. That is, an outermorphism cannot be expanded in isolation, only in the context of *application* to some multivector variable. This complicates the expansion process somewhat, since expanding the outermorphism variable must be delayed until application.

Figure 4.14 shows the expansion process of a function which exposes the apply-outermorphism-functionality to the outside world. The built-in function to apply an outermorphism matrix is called ::g2bi::apply_om() and it is bound to the * operator. The figure shows the five steps in the process of converting the input function to the final C++ code.

```
// 1. Input: User-written DSL function
flatPoint applyOM(omFlatPoint M, flatPoint p) {
  return M * p;
}

// 2. After parsing:
flatPoint applyOM(omFlatPoint M, flatPoint p) {
  return ::g2bi::apply_om(M, p);
}

// 3. Multivector variables written out:
flatPoint applyOM(omFlatPoint M, flatPoint p) {
  return ::g2bi::apply_om(M, p.c0 * e1ni + p.c0 * e2ni + p.c0 * e3ni + p.c0 * noni);
}

// 4. Outermorphism application written out:
flatPoint applyOM(omFlatPoint M, flatPoint p) {
  return
    p.c0 * ::g2bi::extractColumn(M, 0) +
    p.c1 * ::g2bi::extractColumn(M, 1) +
    p.c2 * ::g2bi::extractColumn(M, 2) +
    p.c3 * ::g2bi::extractColumn(M, 3);
}

// 5. Built-in function ::g2bi::extractColumn() statically executed:
flatPoint applyOM(omFlatPoint M, flatPoint p) {
  return
    p.c0 * (M[0*4+0] * e1ni+M[0*4+1] * e2ni+M[0*4+2] * e3ni+M[0*4+3] * noni)+
    p.c1 * (M[1*4+0] * e1ni+M[1*4+1] * e2ni+M[1*4+2] * e3ni+M[1*4+3] * noni)+
    p.c2 * (M[2*4+0] * e1ni+M[2*4+1] * e2ni+M[2*4+2] * e3ni+M[2*4+3] * noni)+
    p.c2 * (M[3*4+0] * e1ni+M[0*4+1] * e2ni+M[3*4+2] * e3ni+M[3*4+3] * noni);
}

// 6. Output: generated C++ code:
inline flatPoint applyOM(const omFlatPoint &M, const flatPoint &p) {
  return flatPoint(flatPoint_e1ni_e2ni_e3ni_noni,
    p.m_c[0]*M.m_c[0*4+0]+p.m_c[1]*M.m_c[1*4+0]+
      p.m_c[2]*M.m_c[2*4+0]+p.m_c[3]*M.m_c[3*4+0],
    p.m_c[0]*M.m_c[0*4+1]+p.m_c[1]*M.m_c[1*4+1]+
      p.m_c[2]*M.m_c[2*4+1]+p.m_c[3]*M.m_c[3*4+1],
    p.m_c[0]*M.m_c[0*4+2]+p.m_c[1]*M.m_c[1*4+2]+
      p.m_c[2]*M.m_c[2*4+2]+p.m_c[3]*M.m_c[3*4+2],
    p.m_c[0]*M.m_c[0*4+3]+p.m_c[1]*M.m_c[1*4+3]+
      p.m_c[2]*M.m_c[2*4+3]+p.m_c[3]*M.m_c[3*4+3]);
}
```

Figure 4.14: *Example of expanding an outermorphism application on the basis. Note that the intermediate functions (2, 3, 4, 5) never exist as the human-readable text shown here, but only as ASTs.*

4.8.6 Static Evaluation of DSL Functions

After all multivector and outermorphism variables have been expanded on the basis, a term rewrite system [1] is used to process the resulting expressions. The term rewrite system converts all geometric algebra operations to regular arithmetic operations by statically evaluating them as far as possible. So for example an expression like;

```
::g2bi::reverse(e1^e2^e3)
```

is converted to:

```
-e1^e2^e3
```

In essence, the term rewrite system contains a symbolic implementation of Sections 3.1 and 3.3, plus rewrite rules for simplifying regular arithmetic.

The rules of the term rewrite system are a mix of rewrite rules (for matching and substitution) and `Java` code (for custom logic). Figure 4.15 shows an example of a few rules. Every rule consists of a matching-clause, an arrow, and a substitution-clause. `Java` code is contained in {} parentheses.

For example, the first rule in Figure 4.15 matches `add()` terms. When a term is matched, it sorts the operands of the add node based on being `scalar` or not. The `scalar` nodes are stored in a list `S` and if this list is not empty, it transforms the `add()` node into another node with `scalar`s summed.

These rewrite rules are compiled by a small custom compiler. This compiler converts the matching-rules and substitution-rules to regular `Java`, which can then be linked to the rest of `Gaigen 2`.

4.8.7 Tight Return Type

Once all geometric algebra operations have been evaluated, `Gaigen 2` optionally determines the 'tight' return type of DSL functions. This is done only for DSL functions that return general multivector values. To determine the return type, `Gaigen 2` finds all return-statements of the DSL function and determines the set of basis blades that can be returned by the statement. The union of all these sets of basis blades is used to find or generate the tight return type.

`Gaigen 2` then searches for a specialized multivector type that precisely matches this set of basis blades. If a suitable type is not found, it is generated from scratch. This also means that the boiler plate code for these new types should be generated. So the process of writing out must actually be performed *before* generating the boiler plate code.

When a suitable return type has been found or created, `Gaigen 2` simply replaces the return type of the function with the tight return type.

Two notes are in order:

```
// Some examples of rewrite rules from the term rewrite system:

// 1) add scalars and place scalars up front
add([R, S]:scalar) {S.size() > 0}
  ->
  add({woob().addScalars(S)}, R);

// 2) turn Euclidean metric into algebra metric if possible
ip<metric, ipType>(X) {(metric == Woob.EUCLIDEAN) && (woob().ipEuclidean())}
  ->
  ip<{Woob.ALGEBRA}, ipType>(X);

// 3) compute outer product of basis elements
op(X, be1:be, be2:be, Y)
  ->
  op(X, {Be.op(be1, be2)}, Y);

// 4) Convert reversion to sum of grade parts multiplied by +1 or -1.
reverse(x)
  ->
  {
    ArrayList L = new ArrayList();
    for (int i = 0; i <= woob().dim(); i++) {
      L.add(woob().mul(
        woob().scalar( (((i>>1) & 1) == 0) ? SF.ONE : SF.NEG_ONE),
        woob().gs(x, woob().scalar(SF.scalar(i)))));
    }
   }
   add({L});
```

Figure 4.15: *Some example of rules from the term rewrite system.* 1) `add()` *nodes are found, scalars isolated and summed, a new* `add()` *node is substituted.* 2) *The inner products which use a Euclidean metric are converted to the regular algebra metric if the metric of the algebra is Euclidean.* 3) *The outer product of basis blades is evaluated.* 4) *The reverse is implemented by selecting each grade part (using* `gs()`*) and multiplying it with scalar* `+1` *or* `-1`*.*

- When a basis blade has a constant coordinate, this is taken into account when searching for the return type. I.e., the return type must have matching constant coordinates. This way, constant coordinates propagate through the type system.
- In certain situations `Gaigen 2` does not substitute the return type. Currently this occurs when basis blades of both even and odd grades are present in the set of result blades. This usually happens in functions over general multivectors.

4.8.8 DSL Function Code Generation

When all DSL functions have been transformed into low-level ASTs, they are converted to the final source code in the target language by the language dependent back end. This is the same back end that does the boiler plate code generation.

Our current back ends transform the low-level AST one more time using another term rewriting system. This term rewriting system implements trivial language-dependent modifications, such as making sure that a 32 bit floating point number will print out with a trailing `f`.

The back end then traverses the AST to pretty print it. Traversal may occur multiple times to emit the right pieces of source code to the right files. For example, in `C++` the user may select which functions to inline. Inlined functions should go into the `header` file, while all other functions go into the general `source` file.

DSL Function Implementation involving General Multivectors

As an example of the type of code that is generated for DSL functions, we first present a partial listing of the generated code for the geometric product of two general multivectors. We show this function to make clear how compression works out in practice, why general multivectors are slow, and also why we should have picked different method to implement these functions (see the discussion, Section 4.11).

The DSL function that computes the geometric product is:

```
// user-written DSL function
mv gp(mv x, mv y) {
  return x y;
}
```

All this function does is expose the built-in geometric product of the DSL language to the outside world in the form of a functions named `gp()`. The generated function is rather huge (489 lines). It is shown (partially) in Figure 4.16. The functions first arranges the coordinates of the two arguments into two easily accessible arrays of pointers to arrays of doubles. Then it iterates over all combinations of grade parts, applying the geometric product to those grades that are present.

Note that only the scalar part of `x` is processed in that part of the function that is actually shown; the rest of the function has been omitted. After all combinations of grade parts have been processed, the function compresses the result and returns it.

This sort of code is representative for all operations involving general multivectors: it is large and contains many conditional statements. The conditional statements that check for the presence of grade parts are generated by the back end: the write-out-phase assumes that all grade parts are present and generates an AST accordingly. The back end processes the AST to find and collect the coordinates into their respective grade-dependent groups.

DSL Function Implementation involving Specialized Multivectors

To demonstrate the much sleeker result created from a function involving just specialized multivectors, we show the `C++` function generated from the following DSL function:

```
// user-written DSL function
line op(normalizedFlatPoint x, dualPlane y) {
  return x ^ y;
}
```

This function computes the outer product of a flat point and a dual plane, which is the line through the point, intersecting the plane dually. The resulting `C++` code is:

```
// Generated C++ code
inline line op(const normalizedFlatPoint& x, const dualPlane& y) {
  return line(line_e1e2ni_e1e3ni_e2e3ni_neg_noe1ni_neg_noe2ni_neg_noe3ni,
    ((x.m_c[1] * y.m_c[0]) + (-1.0 * x.m_c[0] * y.m_c[1])),
    ((x.m_c[2] * y.m_c[0]) + (-1.0 * x.m_c[0] * y.m_c[2])),
    ((-1.0 * x.m_c[1] * y.m_c[2]) + (x.m_c[2] * y.m_c[1])),
    y.m_c[0],
    y.m_c[1],
    y.m_c[2]);
}
```

4.9 Implementation: Profiling

So far, we have been using many examples of DSL functions where the arguments and return types are specialized multivectors. For example, above we used the following function as an example:

```
// user-written DSL function
line op(normalizedFlatPoint x, dualPlane y) {
  return x ^ y;
}
```

```
// Generated C++ code
mv gp(const mv& x, const mv& y) {
  mv __temp_var_1__;
  double __tmp_coord_array_2__[32] ;
  mv_zero(__tmp_coord_array_2__, 32);
  const double* __x_xpd__[6] ;
  x.expand(__x_xpd__, true);
  const double* __y_xpd__[6] ;
  y.expand(__y_xpd__, true);
  if (((x.m_gu & 1) != 0)) {
    if (((y.m_gu & 1) != 0)) {
      __tmp_coord_array_2__[0] += (__x_xpd__[0][0] * __y_xpd__[0][0]);

    }
    if (((y.m_gu & 2) != 0)) {
      __tmp_coord_array_2__[1] += (__x_xpd__[0][0] * __y_xpd__[1][0]);
      __tmp_coord_array_2__[2] += (__x_xpd__[0][0] * __y_xpd__[1][1]);
      __tmp_coord_array_2__[3] += (__x_xpd__[0][0] * __y_xpd__[1][2]);
      __tmp_coord_array_2__[4] += (__x_xpd__[0][0] * __y_xpd__[1][3]);
      __tmp_coord_array_2__[5] += (__x_xpd__[0][0] * __y_xpd__[1][4]);

    }
    if (((y.m_gu & 4) != 0)) {
      __tmp_coord_array_2__[6] += (__x_xpd__[0][0] * __y_xpd__[2][0]);
      __tmp_coord_array_2__[7] += (__x_xpd__[0][0] * __y_xpd__[2][1]);
      __tmp_coord_array_2__[8] += (__x_xpd__[0][0] * __y_xpd__[2][2]);
      __tmp_coord_array_2__[9] += (__x_xpd__[0][0] * __y_xpd__[2][3]);
      __tmp_coord_array_2__[10] += (__x_xpd__[0][0] * __y_xpd__[2][4]);
      __tmp_coord_array_2__[11] += (__x_xpd__[0][0] * __y_xpd__[2][5]);
      __tmp_coord_array_2__[12] += (__x_xpd__[0][0] * __y_xpd__[2][6]);
      __tmp_coord_array_2__[13] += (__x_xpd__[0][0] * __y_xpd__[2][7]);
      __tmp_coord_array_2__[14] += (__x_xpd__[0][0] * __y_xpd__[2][8]);
      __tmp_coord_array_2__[15] += (__x_xpd__[0][0] * __y_xpd__[2][9]);

    }
    if (((y.m_gu & 8) != 0)) {
      __tmp_coord_array_2__[16] += (__x_xpd__[0][0] * __y_xpd__[3][0]);
      __tmp_coord_array_2__[17] += (__x_xpd__[0][0] * __y_xpd__[3][1]);
      __tmp_coord_array_2__[18] += (__x_xpd__[0][0] * __y_xpd__[3][2]);
      __tmp_coord_array_2__[19] += (__x_xpd__[0][0] * __y_xpd__[3][3]);
      __tmp_coord_array_2__[20] += (__x_xpd__[0][0] * __y_xpd__[3][4]);
      __tmp_coord_array_2__[21] += (__x_xpd__[0][0] * __y_xpd__[3][5]);
      __tmp_coord_array_2__[22] += (__x_xpd__[0][0] * __y_xpd__[3][6]);
      __tmp_coord_array_2__[23] += (__x_xpd__[0][0] * __y_xpd__[3][7]);
      __tmp_coord_array_2__[24] += (__x_xpd__[0][0] * __y_xpd__[3][8]);
      __tmp_coord_array_2__[25] += (__x_xpd__[0][0] * __y_xpd__[3][9]);

    }
    if (((y.m_gu & 16) != 0)) {
      __tmp_coord_array_2__[26] += (__x_xpd__[0][0] * __y_xpd__[4][0]);
      __tmp_coord_array_2__[27] += (__x_xpd__[0][0] * __y_xpd__[4][1]);
      __tmp_coord_array_2__[28] += (__x_xpd__[0][0] * __y_xpd__[4][2]);
      __tmp_coord_array_2__[29] += (__x_xpd__[0][0] * __y_xpd__[4][3]);
      __tmp_coord_array_2__[30] += (__x_xpd__[0][0] * __y_xpd__[4][4]);

    }
    if (((y.m_gu & 32) != 0)) {
      __tmp_coord_array_2__[31] += (__x_xpd__[0][0] * __y_xpd__[5][0]);

    }

  }
  // Function continues for 425 more lines! (omitted)
  // ...
  // ...

  // The end of the function:
  __temp_var_1__ = mv_compress(__tmp_coord_array_2__);
  return __temp_var_1__;
}
```

Figure 4.16: *Partial listing of generated `C++` source code for the geometric product (generated from `c3ga.gs2`).*

Writing this type of function directly is not the intended use of Gaigen 2. The idea is to write functions over general multivectors, like

```
// user-written DSL function
mv op(mv x, mv y) {
  return x ^ y;
}
```

and to have Gaigen 2 instantiate these functions with specialized arguments.

What functions to instantiate with what specialized arguments should be determined by *profiling* the user application. Gaigen 2 cannot perform a static analysis of the user application source code (before compilation). Instead, it must extract the required information by running the application with profiling instrumentation added to it. Profiling an application proceeds as follows:

1. The generated source code is re-generated with the profiling instrumentation.

2. The user application is recompiled and run through a *representative* input. The instrumentation records the usage profile of each DSL functions. This usage profile consists of the specialized multivector types used as the arguments, and the number of invocations with those arguments. It is important that the user application 'visits' all parts of its geometry code, otherwise the profile may miss information.

3. The profile is stored in a file.

4. Gaigen 2 re-generates the source code, this time without profiling instrumentation, but with the extra function instantiations, as specified by the profile. I.e., Gaigen 2 uses the profile to instantiate the DSL functions with the specialized arguments.

5. The user application is recompiled.

Although this may seem a straightforward solution, there is a problem that prevents it from working well. If the application uses nested calls to DSL functions, the profiling information cannot be determined in a single run through the above list, for reasons explained below.

When DSL functions are instantiated and optimized, the return type changes from general multivectors to some tight specialized multivector type. Gaigen 2 determines this specialized type by statically analyzing all possible return values of the function (see Section 4.8.7). When calls are nested, the return types of the nested calls change after a profiling run. Thus on the next profiling run, more information will be extracted, and so on, until the type information has 'bubbled' through all nested function calls. An example is given in Figure 4.17. It shows the abstract syntax tree for the function calls in the following piece of code.

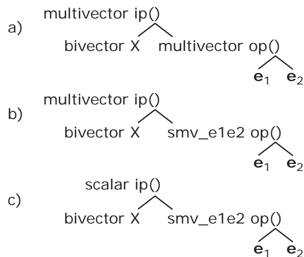


Figure 4.17: *C++ AST of example code during various stages of optimization.*

```
// C++ code, written by user
bivector X;
ip(X, op(e1, e2));
```

Assume that both `ip()` and `op()` are DSL functions (inner product, outer product). When the generated code is not optimized at all (i.e., no profile information), the C++ AST looks like Figure 4.17a. After the first profiling run, an optimized function for the call `op(e1, e2)` has been created. It returns a newly generated type `smv_e1e2`. `Gaigen 2` makes up symbolic names for the types it has to create on the fly. In this example, it determines that the outer product of constant `e1` and constant `e2` is another constant `e1` $\wedge$ `e2`. The code is regenerated, the application recompiled and linked again, and in the second profiling run, the use of function `ip(bivector, smv_e1e2)` is recorded (Figure 4.17b). This function is instantiated (return type `scalar`) and after another round of code generation and compilation the final result is Figure 4.17c.

Note that the generated code cannot easily determine the return types at runtime, because this is a non-trivial computation. I.e., the generated code would have to contain a duplicate expression rewriting system used by `Gaigen 2`.

In our own user applications, we have found that nested function calls can easily be five levels deep, thus requiring five iterations of the profiling process before the optimal code is generated. We found this very tiresome, and to avoid it we have adjusted the profiling process as follows.

When the user application starts, the profiling instrumentation connects to `Gaigen 2`, which is running in the background. On each DSL function invocation, the profiling instrumentation sends the ID of the functions and the type-IDs of the arguments to the code generator (each DSL function and each specialized multivector type gets it own unique ID). The code generator instantiates the function with the arguments, and determines the specialized multivector return type. The ID of the return type is sent back to the profiling instrumentation, which uses it to set the type variable of the general multivector which is returned by the function. This way, general multivectors will always have a specialized multivector type (unless of course the user program explicitly creates a general

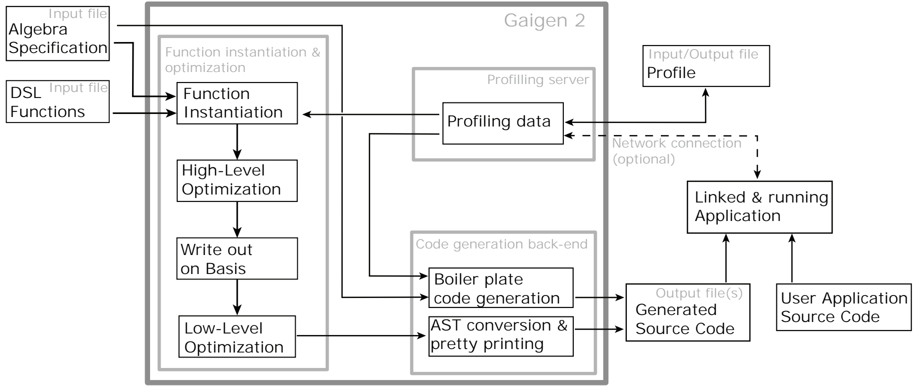


Figure 4.18: *The full* `Gaigen 2` *code generation process. (Repeated for reference; this figure is identical to Figure 4.2).*

multivector without any type).

When the application terminates, the profiling connection to `Gaigen 2` is closed. In response, `Gaigen 2` stores the profile in a file, and optionally regenerates the algebra implementation immediately. Thus all required information is extracted in a single run though the profiling process.

4.10 Implementation: Full System Overview

The full `Gaigen 2` code generation process is illustrated in Figure 4.18. The top left corner shows the algebra specification and the DSL function definitions. These are processed by the various stages of `Gaigen 2` (the big block in the middle). The resulting source files (on the right) are linked to the rest of the application. If required, the application is profiled: the profiling information and type IDs are sent back and forth over the network connection between the application and `Gaigen 2`. The generated code is regenerated with the correct optimizations, after which the application is compiled and linked again.

4.11 Discussion

We have built a system that generates general-purpose geometric algebra implementations, for low-dimensional cases. The main goal was that these implementations would come close to traditional implementations in terms of performance. Our results (see Chapter 7) were better than we initially expected. Our prediction [16] — before actually building the system — was that the conformal ray tracing

application would be about 2 times slower, but we were able to bring this down to about 1.25 times.

Throughout the system we have used generative programming [8]. On the geometric algebra side, this was possible because geometric algebra is such a well-structured framework. We also used code generation for intermediate stages, such as compilation of templates and term rewriting systems.

For the term rewriting system, which is a central part in `Gaigen 2`, we tried multiple term-rewriting and transformation systems. Initially we wanted to use `ASF+SDF` [44], but we found it too bulky, and hard to integrate with other programming languages like `C++` or `Java`. Secondly, we tried a leaner language called `Q` [24]. Using that language we implemented the first version of the rewrite system. But, as with `ASF+SDF`, it too did not integrate with well with other programming languages[2]. In the end we settled for a relatively simple home-grown term rewriting system. As an added benefit, this system was about twice as fast as the `Q`-based implementation (this was measured using our term rewrite system which expands multivectors and equations on the basis).

Overall, we find `Gaigen 2` over-designed. Initially, we had the goal of performing high-level optimizations of large DSL functions. Due to time constraints, we never got to that point. This means the DSL language and its implementation could have been simpler. `Gaigen 2` also should have been more modular, and its construction should have followed a more step-by-step process:

- Start with creating a system that can write out simple GA expressions. Make this a stand-alone tool/library.
- Allow these expressions to be combined to form simple DSL functions (without general multivectors and so one). Write a tool which can convert these functions to `C++`, `Java`, etc.
- Construct something akin to what `Gaigen 2` is now (general multivectors, instantiation, profiling).
- Possibly add high-level optimizations on top of this.

This design would have probably allowed us to produce and release results faster.

Specialized multivectors, general and specialized outermorphism matrices all worked out well in `Gaigen 2`. However, the general multivector should not have been per-grade compressed, and functions working on general multivectors should not have been implemented as shown in the example of Figure 4.16. This approach leads to a huge amount of code. This is the main reason why `Gaigen 2` code is limited to about 6-D currently [3]. We should have allowed the user to choose between per-grade compression and per-coordinate compression, and to choose

[2] Support for embedding `Q` in C and `C++` programs has since been added to `Q`, possibly in response to our feedback.

[3] Note that this does not negatively impact the performance of `Gaigen 2`. The code may be large, but if it is never (or seldomly) invoked, then all it does is waste some memory.

between fully written out products and on-the-fly computation of products, such as used in `MV` [3].

Overall, we find `Gaigen 2` a useful tool and we use it regularly for our own applications. But there should be a '`Gaigen 2.5`' that addresses the shortcomings listed here.

Further low-level optimizations in the code generated by `Gaigen 2` may be possible by considering the approaches used for reducing the multiplicative complexity of bilinear forms [9]. The goal of this field of research is to find optimimal (in terms of number of number of multiplications) algorithms for implementing bilinear forms such as matrix multiplication, or the geometric product. Nowadays, minimizing the number of multiplications may not be useful, but the general principles may still be valid. In fact, we have unknownly built one such optimization into `Gaigen 2` in an ad hoc way: by allowing for a non-orthogonal basis, we can reduce the complexity of operations on certain (conformal) objects (such as lines). Our synthetic benchmark (Section 7.3.2) shows that (in selected cases) this may lead to a speed-up of around three, so it is worth considering if this ad hoc approach could be generalized and automated.

Chapter 5

Implementation of Geometric Algebra on a Multiplicative Basis

5.1 Introduction

Most numerical geometric algebra implementations use the additive representation method of Chapter 3. This works well for low-dimensional spaces (up to around 6-D to 10-D, depending on the implementation), but becomes inefficient for high-dimensional spaces, since the method requires $O(2^n)$ coordinates to represent a multivector in an n-dimensional space. Various compression methods are used to make this approach more efficient, as described in Section 3.2. While these compression methods can offer some relief, ultimately — as n gets large enough — the $O(2^n)$ storage complexity will defeat them.

It is clear that for high-dimensional spaces, a more efficient representation of blades and versors is required. In this chapter, we describe our solution, which is based on a multiplicative, factorized representation. This representation has an $O(n^2)$ storage complexity. Using this representation, operations can be implemented such that they have a $O(n^3)$ (or less) time complexity. In high-dimensional spaces, this is clearly an advantage over the $O(2^n)$ storage complexity and the $O(2^{2n})$ time complexity of the bilinear products in the additive representation.

Note that the implementation method described in this chapter should — up to floating point noise – yield the same numerical results as the additive implementation of Chapter 3. Even though we use factorizations like the singular value decomposition, only the `join` algorithm uses this factorization to discard factors that fall below a certain epsilon value (the additive implementation works on comparable principles through the delta product). Note that the outer product algorithm also uses singular values to check if the outcome of the product is 0 or not, but one may choose to omit this check, essentially continuing computations with a blade that is very small instead of exactly 0.

Existing Work

Our factored representation is an extension of the "simplex representation" of Stolfi [40] in the context of projective geometry. Stolfi describes:

- How to represent homogeneous flats using matrices containing the factorization under the outer product.
- How to compute the `meet` and `join` of these flats.
- How to dualize them.

In essence, Stolfi implements a Grassmann-Cayley algebra. We generalize and extend this to blades, include versors and describe how to implement a full geometric algebra.

The variant of the QR factorization algorithm that uses Householder reflections [23] also represents (and applies) orthogonal transformations in factored form. Again, our method to handle versors is an extension and generalization of this idea.

A Note on Matrix Usage

In this chapter, we will represent blades and versors using matrices; i.e., each column of a matrix is a factor of the blade or versor it represents. This conveniently merges an idea (factorized representation) with an implementation (through linear algebra).

For blades the matrix representation is entirely natural, and it coincides closely with the common practice in linear algebra that the column space of a matrix defines a subspace. This has been extended (e.g., in [40]) to oriented subspaces. We can then make use of standard techniques such as QR decomposition, and computing determinants to implement geometric algebra operations on blades.

For versors, the matrix representation is new but convenient. Its use permits use to describe the versor simplification algorithm in Section 5.4 in a straightforward manner.

5.2 Blades

In this section we discuss the representation of blades, and the implementation of all relevant geometric algebra operations. The next section will do the same for the versors.

5.2.1 Blade Representation

A blade is a multivector that can be factored as the outer product of k vectors:

$$\mathbf{B} = s\,\mathbf{f}_1 \wedge \mathbf{f}_2 \wedge \ldots \wedge \mathbf{f}_k, \tag{5.1}$$

where s is a scalar, and where k can be 0. In our representation, we split the blade into its magnitude and its direction (the oriented subspace it spans). The magnitude is represented by a single scalar. The direction is represented by a matrix. In the following, if a blade is called $\mathbf{B}$, the matrix representing its oriented subspace will be called $[\mathbf{B}]$. $\mathbf{B}^s$ is the scale of the blade $\mathbf{B}$.

Each column of such a matrix represents a unit vector $\mathbf{f}_j$ $(1 \leq j \leq k)$ in one possible factorization of the blade. I.e., the outer product of the columns of the matrix is equal to the blade, up to scale. The columns are chosen to be orthonormal in a Euclidean metric. Orthonormality is not a requirement of Equation 5.1, but a requirement of our representational method.

Most blades allow for infinitely many factorizations, so a particular blade representation is often not unique (Stolfi [40] uses an upper triangular representation that *is* unique). But, as shown in the following sections, this does not cause any problems. Note that we are not just interested in the range of the matrix $[\mathbf{B}]$ — which determines the subspace as kernel($[\mathbf{B}]$) — but also in the specific ordering of the columns of the matrix — which determines the orientation of the subspace.

The individual vectors $\mathbf{f}_i$ are represented as a list of coordinates with respect to an arbitrary basis of vectors that span the n-dimensional space. We will refer

$$3\sqrt{2}\begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2}\sqrt{2} \\ 0 & -\frac{1}{2}\sqrt{2} \end{bmatrix}$$

Figure 5.1: *The blade* $3\,\mathbf{e}_1 \wedge (\mathbf{e}_2 - \mathbf{e}_3)$ *stored in factored representation. The oriented subspace spanned by the blade is represented by the two orthonormal columns of the matrix. The magnitude of the blade is* $3\sqrt{2}$.

to these basis vectors as $\mathbf{e}_i$ $(1 \leq i \leq n)$. It can be more efficient to choose these basis vectors as orthonormal as possible for the required metric, but this is not an absolute requirement.

Figure 5.1 shows an example. Suppose our basis is $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$, then the blade $3\mathbf{e}_1 \wedge (\mathbf{e}_2 - \mathbf{e}_3)$ could be represented as shown in the matrix. The magnitude of the blade $(3\sqrt{2})$ is stored separately from the matrix. The two columns of the matrix span the oriented subspace of the blade. Flipping the order of the columns flips the orientation of the blade.

Note that for simplicity we often ignore the scale $(\mathbf{B}^s)$ of blades when its effects on an algorithm are trivial.

5.2.2 Reverse and Grade Involution

As discussed in Section 2.4.6, the reverse of a grade k blade is computed as:

$$\widetilde{\mathbf{A}}_k = (-1)^{\frac{1}{2}k(k-1)}\mathbf{A}_k.$$

This is trivially implemented by multiplying the scalar factor of the blade representation by $(-1)^{\frac{1}{2}k(k-1)}$. Likewise, the grade involution of a blade is implemented by multiplying the sign of the magnitude by $(-1)^k$.

5.2.3 Inner Product of Two Blades of the Same Grade, using a Euclidean Metric

The generic implementation of the metric products is discussed later on, but up front we require an implementation of the inner product for two blades of the same grade, in a Euclidean metric. (To indicate an inner product between vectors in a Euclidean metric, we use the symbol $\odot$; to indicate an inner product between vectors in the algebra metric, we use the regular $\cdot$ symbol).

Let $\mathbf{A}_k$ and $\mathbf{B}_k$ be two blades of the same grade k. This means their inner product $\mathbf{C} = \mathbf{A}_k \odot \mathbf{B}_k$ is a scalar. We can compute:

$$\gamma = \mathbf{A}_k \odot \mathbf{B}_k$$

as

$$\begin{aligned} [\mathbf{C}] &= 1, \\ \mathbf{C}^s &= (-1)^{\frac{1}{2}k(k-1)} \det([\mathbf{A}_k]^T [\mathbf{B}_k]). \end{aligned} \tag{5.2}$$

Proof:
Since the inner product is linear, let us assume unit blades $\mathbf{A}_k$ and $\mathbf{B}_k$. $\mathbf{A}_k$ can be split up into a part parallel ($\mathbf{A}_{\|}$) and a part orthogonal ($\mathbf{A}_{\perp}$) to $\mathbf{B}$. By definition, the orthogonal part does not contribute to the inner product:

$$\begin{aligned} \mathbf{A}_k \odot \mathbf{B}_k &= (\mathbf{A}_{\|} + \mathbf{A}_{\perp}) \odot \mathbf{B}_k \\ &= \mathbf{A}_{\|} \odot \mathbf{B}_k \\ &= \alpha \mathbf{B}_k \odot \mathbf{B}_k \\ &= (-1)^{\frac{1}{2}k(k-1)} \alpha \widetilde{\mathbf{B}}_k \odot \mathbf{B}_k \\ &= (-1)^{\frac{1}{2}k(k-1)} \alpha, \end{aligned}$$

where α is the scale of $\mathbf{A}_{\|}$ relative to $\mathbf{B}_k$. This is just the oriented volume of the hyperparallelepiped spanned by the columns of $\mathbf{A}_k$ and we can show that this is the determinant of the matrix $[\mathbf{A}_k]^T[\mathbf{B}_k]$.

To see why, note that we can apply an orthogonal transformation R to $[\mathbf{B}_k]$ such that the result is a $k \times k$ identity matrix stacked on top of a $(n-k) \times k$ null matrix:

$$R\,[\mathbf{B}_k] = \begin{bmatrix} I_{k \times k} \\ 0_{(n-k)\times(k)} \end{bmatrix}.$$

If we apply the same transform to $[\mathbf{A}_k]$ and set the lower $n-k$ rows of the result to 0, it should be clear that this is a (possibly non-orthogonal) representation of $\mathbf{A}_{\|}$. So to retrieve the scale of $\mathbf{A}_{\|}$ relative to $\mathbf{B}_k$, we can use

$$\begin{aligned} \alpha &= \det([\mathbf{A}_{\|}]^T\,[\mathbf{B}_k]) = \det((R\,[\mathbf{A}_k])^T\,R\,[\mathbf{B}_k]) = \det([\mathbf{A}_k]^T\,R^T\,R\,[\mathbf{B}_k]) \\ &= \det([\mathbf{A}_k]^T\,[\mathbf{B}_k]). \end{aligned}$$

□
Note that the proof is still correct when the columns of $[\mathbf{A}]$ are not orthonormal, because we only require that the columns of $[\mathbf{B}]$ are orthonormal.

5.2.4 Outer Product

Let $\mathbf{A}_k$ and $\mathbf{B}_l$ be two unit blades. We want to compute the outer product

$$\mathbf{C}_{k+l} = \mathbf{A}_k \wedge \mathbf{B}_l.$$

We first detect trivial cases such as when one of the blades is a scalar or when $(k+l) > n$. These are handled separately.

Otherwise we construct a new matrix O by appending the columns of $[\mathbf{B}_l]$ to $[\mathbf{A}_k]$, i.e.,

$$O = [\ [\mathbf{A}_k] \quad [\mathbf{B}_l]\ ]\,.$$

In principle, the columns of O form a valid factorization for $\mathbf{C}_{k+l}$, except that they are not necessarily orthonormal. Also, if the columns are linearly dependent, that should be detected and the result of the product set to zero. For this purpose we use the singular value decomposition (SVD).

The SVD [23] computes matrices U, S, V such that

$$O = U\, S\, V^T. \tag{5.3}$$

Both U and V are orthogonal. The diagonal of S contains the *singular values* of O. The convention is to order them from large to small. Let ϵ be an appropriate threshold value, then if $S_{k+l,k+l} \leq \epsilon$, we have detected that $\mathbf{A}_k \wedge \mathbf{B}_l = 0$. Otherwise, we set $[\mathbf{C}]$ to the first k columns of U, as these columns span the same subspace as O.

It remains to compute the scalar factor $\mathbf{C}^s$. Although the outer product is non-metric, we can compute the scaling factor conveniently using a Euclidean metric on the (arbitrary) basis we use to represent the subspace. The relative scale of two blades that span the same subspace is computed as $s = \mathbf{X} \odot \mathbf{Y}^{-1}$. When $\mathbf{Y}$ is unit, this is equivalent to

$$s = \mathbf{X} \odot \widetilde{\mathbf{Y}} = (-1)^{\frac{1}{2}k(k-1)} \mathbf{X} \odot \mathbf{Y}.$$

This is implemented as:

$$\mathbf{C}^s = (-1)^{\frac{1}{2}k(k-1)} \left((-1)^{\frac{1}{2}k(k-1)} \det(O^T\, [\mathbf{C}]) \right) = \det(O^T\, [\mathbf{C}]). \tag{5.4}$$

5.2.5 Creating a Blade from a Matrix

Let A be an $n \times k$ matrix that represents a subspace through the span of its columns. The columns are assumed to be independent, so $\mathrm{rank}(A) = k$. Such a matrix may come from some algorithm or other software that does not use geometric algebra, and we may want to continue computations with geometric algebra. This means that we should convert it into a blade. We may choose to treat this subspace as scaled or oriented. In any case, we want to compute a blade $\mathbf{A}$ which represents the same subspace as A. The QR decomposition [23] can be used to compute a valid blade representation of the subspace spanned by A, i.e.:

$$\begin{aligned} A &= Q\, R, \\ [\mathbf{A}] &= Q. \end{aligned}$$

If required, we can retain scale and orientation by setting

$$\mathbf{A}^s = \det(A^T [\mathbf{A}]),$$

otherwise, $\mathbf{A}^s = 1$.

5.2.6 Dual using a Euclidean Metric

In this section we show how to compute the dual of a blade with respect to the full space, in a Euclidean metric. The dual using an arbitrary metric is discussed later on. Dualization of a blade $\mathbf{A}$ is defined in Section 2.8.1 as

$$\mathbf{D} = \mathbf{A}^\star = \mathbf{A} \rfloor \mathbf{I}^{-1} = \mathbf{A}\,\mathbf{I}^{-1}.$$

where $\mathbf{I}$ is the unit pseudoscalar of the full space. If $\mathbf{A}$ is unit, we may derive:

$$\mathbf{A}^{-1}\,\mathbf{D} = \mathbf{A}^{-1} \wedge \mathbf{D} = \widetilde{\mathbf{A}} \wedge \mathbf{D} = \mathbf{I}^{-1} = \widetilde{\mathbf{I}}.$$

This makes it clear that $\mathbf{A}^{-1}$ and $\mathbf{D}$ must span the entire space, and must have a particular orientation relative to $\mathbf{I}$. We factor out the scalar multiplier due to reversion of $\mathbf{A}$, and multiply by $\mathbf{I}$:

$$(-1)^{\frac{1}{2}k(k-1)}(\mathbf{A} \wedge \mathbf{D})\,\mathbf{I} = 1, \tag{5.5}$$

with $k = \text{grade}(\mathbf{A})$.

To compute $\mathbf{D}$ in our matrix representation, we set $[\mathbf{D}]$ to any orthogonal complement of $[\mathbf{A}]$. To compute this orthogonal complement, we can use for example the full QR decomposition and discard the first k the columns of Q that span the space of $[\mathbf{A}]$. We then compute a new matrix O by appending the columns of $[\mathbf{D}]$ to $[\mathbf{A}]$:

$$O = \left[\;[\mathbf{D}] \quad [\mathbf{A}]\;\right]. \tag{5.6}$$

This is a full-rank $n \times n$ matrix representing (up to scale) the outer product of $\mathbf{A}$ and its dual $\mathbf{D}$. We compute $\mathbf{D}^s$ such that Equation 5.5 holds:

$$\mathbf{D}^s = (-1)^{\frac{1}{2}(k(k-1)+n(n-1))}\det(O\,I) = (-1)^{\frac{1}{2}(k(k-1)+n(n-1))}\det(O) \tag{5.7}$$

Note that the columns of $[\mathbf{A}]$ form an orthonormal set, as do the columns of $[\mathbf{D}]$. Since $[\mathbf{A}]$ and $[\mathbf{D}]$ are each others orthogonal complement, $\det(O) = \pm 1$.

To summarize the algorithm for computing the dual using a Euclidean metric:

- Compute the orthogonal complement of $[\mathbf{A}]$, e.g., using full QR decomposition. This is (up to scale) the matrix representation of $[\mathbf{D}]$.
- Construct the matrix O as in Equation 5.6.
- Use Equation 5.7 to compute $\mathbf{D}^s$.

5.2.7 Representation of Metric

The *metric matrix* representation of metric, as discussed in Section 2.5.2, can be directly used to implement metric products in the multiplicative representation.

Each entry in such a metric matrix is the inner product of the two respective basis vectors. For example, the metric matrix for the conformal model of 3-D space would be (see also Section 2.11.1):

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

We usually use M to denote the metric matrix.

5.2.8 Dual using an Arbitrary Metric

We can generalize the dual using a Euclidean metric to the dual using any metric. To do this, we should evaluate the left contraction on the RHS of the following equation using the metric specified by some metric matrix M:

$$\mathbf{D} = \mathbf{A}^\star = \mathbf{A} \rfloor \mathbf{I}^{-1}.$$

The left contraction can be evaluated using equations 2.5 and 2.6, which we repeat for reference:

$$\begin{aligned} \mathbf{A} \rfloor (\mathbf{B} \wedge \mathbf{C}) &= (\mathbf{A} \rfloor \mathbf{B}) \wedge \mathbf{C} + (-1)^{\mathrm{grade}(\mathbf{B})} \mathbf{B} \wedge (\mathbf{A} \rfloor \mathbf{C}), \\ (\mathbf{A} \wedge \mathbf{B}) \rfloor \mathbf{C} &= \mathbf{A} \rfloor (\mathbf{B} \rfloor \mathbf{C}). \end{aligned}$$

These two equations give a primitive algorithm for evaluating a left contraction, and they show that:

- The metric in a left contraction ultimately originates from simple dot products between vectors.
- The result of a left contraction is a weighted sum of factors of the second operand.

This gives us a way to adjust the dual-using-Euclidean-metric algorithm from Section 5.2.6 such that it works for non-Euclidean metric: we correctly evaluate left contractions between vectors, while not changing the factors of the second operand. This can be accomplished by replacing $[\mathbf{A}]$ with $M\,[\mathbf{A}]$. The matrix product $M\,[\mathbf{A}]$ replaces each column of $[\mathbf{A}]$ (that is, each factor $\mathbf{a}_i$ of $\mathbf{A}$) by $M\,[\mathbf{a}_i]$, such that left contractions of vectors are executed properly as

$$[\mathbf{a}_i]^T\, M\, [\mathbf{x}],$$

where vector $\mathbf{x}$ is a factor of the second operand. Remember that M is symmetric so it does not have to be transposed. See also Section 2.5.1.

The algorithm for computing $\mathbf{D} = \mathbf{A}^\star$ using a metric specified by metric matrix M then becomes:

- Compute the matrix representation of the dual, up to scale, so $[\mathbf{D}]$ is the orthogonal complement of $M\,[\mathbf{A}]$.
- Construct the matrix $O = \left[\ [\mathbf{D}] \quad (M\,[\mathbf{A}])\ \right]$.
- Use O to compute $\mathbf{D}^s$ through Equation 5.7.

5.2.9 Metric Products

This section discusses the generic implementation of the various metric products. We implement the left contraction and derive the other inner products from it.

First, if (grade($\mathbf{A}$) > grade($\mathbf{B}$)) then $\mathbf{A}\rfloor\mathbf{B} = 0$. If (grade($\mathbf{A}$) = grade($\mathbf{B}$)), we use (from Section 5.2.3)

$$\begin{aligned} \mathbf{C} &= \mathbf{A}\rfloor\mathbf{B} \rightarrow \\ [\mathbf{C}] &= 1, \\ \mathbf{C}^s &= (-1)^{\frac{1}{2}k(k-1)}\det([\mathbf{A}]^T M[\mathbf{B}]), \end{aligned}$$

where M is the metric matrix.

Otherwise (grade($\mathbf{A}$) < grade($\mathbf{B}$)), and we implement the left contraction through its adjoint relationship with the outer product, i.e., as a dual of an outer product with a dual blade (see Section 2.8.1):

$$\mathbf{A}\rfloor\mathbf{B} = \left(\mathbf{A} \wedge (\mathbf{B}\,\mathbf{I}^{-1})\right)\,\mathbf{I}. \tag{5.8}$$

In this equation, duality should be evaluated using the desired metric.

The right contraction can be specified in terms of the left contraction using dualization:

$$\mathbf{A}\lfloor\mathbf{B} = \widetilde{\widetilde{\mathbf{B}}\rfloor\widetilde{\mathbf{A}}}.$$

The other metric products can be derived using the rules from Section 3.1.7.

Alternative Implementation

Evaluating a metric product using Equation 5.8 seems rather indirect and computationally inefficient. Possibly one may also implement the metric products using a projection followed by an orthogonal complement, as suggested by Figure 2.4. Both projection and orthogonal complement are elementary operations in linear algebra. However, this is future work and we have not worked out the details.

5.2.10 `join`

The join $\mathbf{J} = \mathbf{A} \cup \mathbf{B}$ is the union of blades. Computing it through SVD is straightforward. The `join` is like an outer product, but without the requirements

that $\mathbf{A}$ and $\mathbf{B}$ are independent. We simply append the columns of $[\mathbf{B}]$ to $[\mathbf{A}]$, and compute the SVD:

$$\begin{bmatrix} [\mathbf{A}] & [\mathbf{B}] \end{bmatrix} = U\,S\,V^T.$$

We then find the first entry on the diagonal of S that has $S_{i,i} \leq \epsilon$ and set $[J]$ to the first $i-1$ columns of U.

Note that it makes no sense to use this convenient way of computing the `join` for the additive representation (Section 3.4.7): for that to work, we would have to factor (Section 3.4.6) the input blades first, which is approximately as expensive as directly using the additive `meet` and `join` algorithm itself.

We fix the scale of $\mathbf{A} \cup \mathbf{B}$ to 1 since the `join` has no absolute magnitude (see Section 2.13).

5.2.11 `meet`

We compute the `meet` using the `meet-join` relationship Equation 2.47, repeated here for convenience:

$$\begin{aligned} \mathbf{J} = \mathbf{A} \cup \mathbf{B} &= \mathbf{A} \wedge (\mathbf{M}^{-1} \rfloor \mathbf{B}), \\ \mathbf{M} = \mathbf{A} \cap \mathbf{B} &= (\mathbf{B} \rfloor \mathbf{J}^{-1}) \rfloor \mathbf{A}. \end{aligned}$$

Since the `meet` is non-metric, we use the more efficient Euclidean metric to implement it.

5.2.12 Addition

As shown in Section 2.4.7, the sum of non-zero two blades will be another blade if and only if:

- They have the same grade k.
- The grade of their `meet` is k or $k-1$.

To compute $\mathbf{C} = \mathbf{A} + \mathbf{B}$, we first check for some some trivial cases (scalar, vectors) and handle those separately. For the non-trivial cases, we check if the grades of $\mathbf{A}$ and $\mathbf{B}$ are equal. Then we compute $\mathbf{A} \cap \mathbf{B}$ and check that it satisfies the second condition. When these conditions are not met, the sum of the blades is a multivector and not a blade, see below.

If the grade of $\mathbf{M} = \mathbf{A} \cap \mathbf{B}$ is equal to k, then they span the same subspace. So we set $[\mathbf{C}] = [\mathbf{A}]$. The orientation of $\mathbf{B}$ relative to $\mathbf{A}$ is $\mathbf{A} \odot \widetilde{\mathbf{B}}$, so we compute:

$$\mathbf{C}^s = \mathbf{A}^s + \det([\mathbf{A}][\mathbf{B}]^T)\mathbf{B}^s. \tag{5.9}$$

Note that $\det([\mathbf{A}][\mathbf{B}]^T) = \pm 1$. Otherwise, the grade of $\mathbf{M}$ is equal to $k-1$, and we compute:

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = \mathbf{M} \wedge (\mathbf{M}^{-1} \odot \mathbf{A} + \mathbf{M}^{-1} \odot \mathbf{B}). \tag{5.10}$$

All operations used in this equation have already been discussed above.

Summing Blades to Form Multivectors

Of course, even if the addition algorithm for blades fails, we can 'add' two blades by forming a multivector, which in the multiplicative representation is basically a list of blades. However, the sum of such blades does not have geometric significance in general. We see this as a canonical difference between Clifford algebra and geometric algebra: for the latter only 'blade-preserving' addition should be allowed.

5.2.13 Inversion

Computing the inverse of a blade is easy, if it exists:

$$\mathbf{A}^{-1} = \frac{\widetilde{\mathbf{A}}}{\widetilde{\mathbf{A}}\,\mathbf{A}}. \tag{5.11}$$

This equation is described in detail in Section 2.6.6 and can be directly implemented using the techniques discussed so far. Of course not all blades have inverses, since they may contain null factors when factored under the geometric product, which results in a zero denominator in Equation 5.11.

5.2.14 Linear Transformations

Let L be an $n \times n$ matrix representing a linear transformation. Such a linear transformation can easily be applied to blades in factored representation. This is possible because linear transformations are outermorphisms, see Section 2.12. We can apply the linear transformation represented by L to a blade $\mathbf{A}$ as $L\,[\mathbf{A}]$. However, if L is not an orthonormal transformation (in a Euclidean metric), re-orthonormalization of the resulting matrix may be required. The technique described in section 5.2.5 can be used to obtain a valid blade representation again. If L is not full rank, the resulting blade may have 0 scale, which is detected when the scale of the blade representation is computed during re-orthonormalization.

5.3 Versors

In this section, we discuss the representation of versors, and the implementation of the relevant geometric algebra operations. The algorithm for simplification of versors is presented its own section, because we needed to develop a novel low-level algorithm for that purpose.

5.3.1 Versor Representation

A versor is the geometric product of k invertible vectors (Section 2.6.5):

$$\mathbf{V} = s\,\mathbf{f}_1\,\mathbf{f}_2\,\ldots\,\mathbf{f}_k \tag{5.12}$$

where s is a scalar, and where k can be 0.

In our multiplicative representation, we store the k factors of a versor in the columns of a $k \times n$ matrix. All columns are unit (Euclidean metric). The scale of the versor is kept in a separate scalar. The columns need not be orthonormal (as for the blades) since the inner product between the factors is an essential property of the versor. During some versor-processing algorithms the columns may temporarily be non-unit, but we re-normalize them after running the respective algorithm.

5.3.2 Conversion from Blade Representation to Versor Representation

Invertible blades are versors and are used as such in geometric algebra. We would like to convert from the blade representation to the versor representation. This requires some work because the columns of the matrix that represents the blade are orthonormal in a Euclidean metric, whereas the equivalent versor representation must have orthogonal columns in the metric of the algebra. That is, the outer product of the blade factors is not equal to their geometric product in general:

$$\mathbf{f}_1 \wedge \mathbf{f}_2 \wedge \ldots \wedge \mathbf{f}_k \neq \mathbf{f}_1\, \mathbf{f}_2\, \ldots\, \mathbf{f}_k. \tag{5.13}$$

To convert from the blade representation to the versor representation we have to find an orthogonal factorization in the metric of the algebra. We already discussed a related problem for the additive representation in Section 3.1.6.

Let $\mathbf{A}$ be a blade in our representation. We want to compute an eigenbasis for the blade in some metric M. We first compute the inner product of every factor of the blade with itself:

$$P = [\mathbf{A}]^T M [\mathbf{A}]. \tag{5.14}$$

P is a real symmetric $k \times k$ matrix and each entry $P_{[i,j]} = P_{[j,i]}$ contains the inner product of factors i and j in metric M. We compute the eigenvalue decomposition

$$P = V \Lambda V^{-1}$$

such that the columns of V are the eigenvectors of P.

To compute the eigenvectors of the blade in the original space, we transform it according to V

$$[\mathbf{W}] = [\mathbf{A}]V,$$

such that

$$\Lambda = V^{-1}[\mathbf{A}]^T M [\mathbf{A}] V = [\mathbf{W}]^T M [\mathbf{W}],$$

where $[\mathbf{W}]$ is the matrix representation of the versor $\mathbf{W}$ we were searching for: the columns of $[\mathbf{W}]$ are orthogonal in the metric specified by M. As the last step, we compute the scale of the new versor using the following familiar equation:

$$\mathbf{W}^s = \det([\mathbf{W}]^T [\mathbf{A}])\, \mathbf{A}^s.$$

5.3.3 Typical Usage of Versors in Geometric Algebra

Unlike blades, we do not have to implement a lot of products for versors, because versors are mostly used to represent and compose transformations. This requires implementation of a few operations, which are:

- Computing exponentials of bivectors.
- Composition or two versors (the geometric product): $\mathbf{V} = \mathbf{W}\,\mathbf{U}$.
- Application of a versor $\mathbf{V}$ to a blade or another versor $\mathbf{X}$: $\mathbf{Y} = \mathbf{V}\mathbf{X}\mathbf{V}^{-1}$. This in turn requires:
- Inversion and reversion.

We discuss each operations below, in reverse order.

5.3.4 Versor Reverse

To reverse a versor, it suffices to simply reverse the order of the columns of its matrix representation.

5.3.5 Versor Inverse

Versor inversion was discussed in Section 2.6.6, and can be implemented using Equation 2.18, which is repeated here for convenience:

$$\mathbf{V}^{-1} = \frac{\widetilde{\mathbf{V}}}{\widetilde{\mathbf{V}}\,\mathbf{V}}.$$

Computing the geometric product $\widetilde{\mathbf{V}}\,\mathbf{V}$ in Equation 3.6 is straightforward, since we have direct access to the factorization under the geometric product $\mathbf{V} = \mathbf{v}_1\,\mathbf{v}_2\,\ldots\,\mathbf{v}_k$ such that

$$\begin{aligned}
\widetilde{\mathbf{V}}\,\mathbf{V} &= (\mathbf{v}_k\,\ldots\,\mathbf{v}_2\,\mathbf{v}_1)\,(\mathbf{v}_1\,\mathbf{v}_2\,\ldots\,\mathbf{v}_k) \\
&= \mathbf{v}_k\,\ldots\,\mathbf{v}_2\,\mathbf{v}_1\,\mathbf{v}_1\,\mathbf{v}_2\,\ldots\,\mathbf{v}_k \\
&= \prod_{i=1}^{k} \mathbf{v}_i^2.
\end{aligned}$$

Formulated in matrix multiplications $\widetilde{\mathbf{V}}\,\mathbf{V}$ is the product of the diagonal entries of $[\mathbf{V}]^T M[\mathbf{V}]$.

5.3.6 Versor Application

Versors are applied to other versors and blades using expressions like $\mathbf{V}\,\mathbf{X}\,\widetilde{\mathbf{V}}$, $\widetilde{\mathbf{V}}\,\mathbf{X}\,\mathbf{V}$, $\mathbf{V}\,\mathbf{X}\,\mathbf{V}^{-1}$ and $\mathbf{V}^{-1}\,\mathbf{X}\,\mathbf{V}$. All these expressions can be implemented using the same approach. The example we use here is for $\mathbf{V}\,\mathbf{X}\,\mathbf{V}^{-1}$. First we show that applying a versor can be done factor by factor:

- if $\mathbf{X}$ is a blade, the following is true:

$$\begin{aligned} \mathbf{V}\,\mathbf{X}\,\mathbf{V}^{-1} &= \mathbf{V}\,(\mathbf{x}_1 \wedge \mathbf{x}_2 \wedge \ldots \wedge \mathbf{x}_k)\,\mathbf{V}^{-1} \\ &= (\mathbf{V}\,\mathbf{x}_1\,\mathbf{V}^{-1}) \wedge (\mathbf{V}\,\mathbf{x}_2\,\mathbf{V}^{-1}) \wedge \ldots \wedge (\mathbf{V}\,\mathbf{x}_k\,\mathbf{V}^{-1}). \end{aligned} \tag{5.15}$$

- if $\mathbf{X}$ is a versor, the following is true:

$$\begin{aligned} \mathbf{V}\,\mathbf{X}\,\mathbf{V}^{-1} &= \mathbf{V}\,(\mathbf{x}_1\,\mathbf{x}_2\,\ldots\,\mathbf{x}_k)\,\mathbf{V}^{-1} \\ &= (\mathbf{V}\,\mathbf{x}_1\,\mathbf{V}^{-1})\,(\mathbf{V}\,\mathbf{x}_2\,\mathbf{V}^{-1})\,\ldots\,(\mathbf{V}\,\mathbf{x}_k\,\mathbf{V}^{-1}). \end{aligned} \tag{5.16}$$

So it suffices to show that we can compute $\mathbf{V}\,\mathbf{a}\,\mathbf{V}^{-1}$ for some vector $\mathbf{a}$.

In the blade-case, we must to re-orthonormalize the factors after applying the versor: even though versor-application is an orthogonal transformation, this is orthogonal in the metric of the algebra, and not the Euclidean metric we use for the matrix representation of the blade.

To find out how to apply the versor to the vector, we rewrite the versor in factored form, assuming unit factors:

$$\mathbf{V}\,\mathbf{a}\,\mathbf{V}^{-1} = s\,s_i\,\mathbf{f}_1\,\mathbf{f}_2\,\ldots\,(\mathbf{f}_k\,\mathbf{a}\,\mathbf{f}_k)\,\ldots\,\mathbf{f}_2\,\mathbf{f}_1, \tag{5.17}$$

where s and s_i are the scalar magnitude of $\mathbf{V}$ and $\mathbf{V}^{-1}$ respectively. A geometric product of three vectors in a 2-D subspace is a vector. In particular, the middle part of Equation 5.17

$$\mathbf{f}_k\,\mathbf{a}\,\mathbf{f}_k \tag{5.18}$$

is just the reflection of $\mathbf{a}$ in $\mathbf{f}_k$, at least up to scale (see Section 2.9.1). Repeated application of Equation 5.18 implements Equation 5.17, which can in turn be used to implement equations 5.16 and 5.15.

5.3.7 Exponentials of Bivectors

Computing the exponential of bivectors is an important component of any geometric algebra implementation, see Section 2.9.3. Currently we can only handle exponentials of 2-blades that have a scalar square. For those blades, we use Equation 3.8 to get a scalar-plus-2-blade-representation, which can be factored into a valid versor representation.

However, in practice one also wants to compute exponentials of bivectors and blades that do not have a scalar square. We cannot handle this as of yet, but we

hope to apply the techniques described in [37] to 'unravel' such exponentials of bivectors into a geometric product of simpler exponentials, as in

$$e^{\mathbf{X}} = e^{\mathbf{B}_1}\, e^{\mathbf{B}_2}\, \dots\, e^{\mathbf{B}_n},$$

where $\mathbf{X}$ is a bivector with a non-scalar square and each of the $\mathbf{B}_i$ is a 2-blade with a scalar square. Each of the $e^{\mathbf{B}_i}$ can be evaluated using Equation 3.8, and the their geometric product can be simplified using the techniques from the next section.

5.4 Simplifying Versor Representations

On our list of typical versor usage one problem remains: composition, i.e., the geometric product of versors. In principle, 'computing' the geometric product of two versors is as simple as concatenating their factors. Of course, the representation of versors would then grow unboundedly large under repeated composition. Thus we should try to simplify the representation of the outcome of the geometric product as much as possible. This is the most complicated part of the multiplicative implementation, and we have not fully solved this problem for general geometric algebras. We present a solution that works for many practical geometric algebras and metrics, namely for $\mathbb{R}^{n,0}, \mathbb{R}^{0,n}, \mathbb{R}^{n-1,1}$ and $\mathbb{R}^{1,n-1}$ (this includes the conformal model). We explain below why our method fails for other metrics.

The problem is, given two versors $\mathbf{V}$ and $\mathbf{W}$, to compute the factorized representation of their geometric product $\mathbf{V}\,\mathbf{W}$. The factorized representation should have the smallest number of factors possible for that versor. And we would like this geometric product simplification algorithm to have $O(n^3)$ time complexity.

Intuition suggests that in an n-dimensional (sub-)space, versors have at most n factors: the outer product (which is the top grade part of a geometric product) of more than n would be zero due to linear dependency. This would appear to lower the grade of the versor by two. For example, the highest non-zero grade part of a geometric product of $n+1$ vectors would be of grade $n-1$, in general. Hence, one may conclude that $n-1$ vectors suffice to factorize such a versor.

However this is not true in all metrics. I.e., one can construct versors in an n-dimensional (sub-)space which have a minimal factorization of more than n factors. We give a simple example of a seemingly 2-versor $\mathbf{V}$ from a 3-D subspace, which is actually a 4-versor:

$$\mathbf{V} = (\mathbf{e}_1 + \mathbf{n}_1)\,(\mathbf{e}_1 + \mathbf{n}_2)\,\mathbf{e}_1\,(\mathbf{e}_1 + \mathbf{n}_1 - \mathbf{n}_2) = 1 + \mathbf{n}_1 \wedge \mathbf{n}_2, \tag{5.19}$$

where $\mathbf{e}_1^2 = 1$ and the subspace spanned by $\mathbf{n}_1$ and $\mathbf{n}_2$ has a signature $\mathbb{R}^{0,0,2}$. This means that $\mathbf{n}_1 \cdot \mathbf{n}_1 = \mathbf{n}_2 \cdot \mathbf{n}_2 = \mathbf{n}_1 \cdot \mathbf{n}_2 = 0$.

The versor from Equation 5.19 can never be factored as a geometric product of two vectors, since those vectors would have to be contained in the $\mathbf{n}_1 \wedge \mathbf{n}_2$ subspace (to form $\mathbf{n}_1 \wedge \mathbf{n}_2$), and thus are null vectors. The geometric product of

vectors from this subspace always has a zero scalar part, and thus factorization using only two factors is impossible.

To create the situation of Equation 5.19, a space with a signature $\mathbb{R}^{m,k}$ or $\mathbb{R}^{k,m}$ with $(m > k \geq 2)$ is required. This is why our method is restricted to the aforementioned signatures where $k < 2$. Interestingly, these are the same spaces in which '*every isometry connected to the identity is an exponential of an anti-symmetric transformation*' [37], although we have not been able to relate those two properties.

We will now describe our algorithm, which is able to simplify versor factorizations in spaces with the aforementioned signatures. The foundation for this algorithm is a technique that can simplify the geometric product of an existing k-versor with a single vector. This technique has a $O(n^2)$ time complexity. Because the algorithm applies this technique in the order of n times, we achieve our desired $O(n^3)$ time complexity.

Below, we first discuss our algorithm for the special case of Euclidean metrics (which also works for anti-Euclidean metrics). Then we extend it to include the other metrics.

5.4.1 Versor Representation: Extra Administration

To achieve our desired time complexity, we require a significant amount of extra administration to be present in the versor representation. The reasons for the extra administration will become clear during the discussion of the algorithm. Here we just list what is required. We also give each matrix a label to simplify the discussion. The required matrices are:

- A matrix F containing the factors of the k-versor. The factors are stored in the columns and are unit length, in a Euclidean metric. Note that we want to simplify the geometric product of versors $\mathbf{V}\,\mathbf{W}$, so F is just a shorthand for $[\mathbf{V}]$.
- A matrix O that is the blade representation of the top-grade part of F. I.e., the columns of O are orthonormal (in a Euclidean metric) and span the same subspace as the columns of F.
- A matrix W that contains the transformation from O to F, such that $O = F\,W$, and conversely $F = O\,W^{-1}$.
- A matrix M_F that contains the Euclidean-metric inner product of each column of F with each other column: $M_F = F^T\,F$.

The extra administration can be permanently maintained in each versor representation. It can also be constructed in $O(n^3)$ time, when it is required: first compute M_F, then compute the eigenvalue decomposition $M_F = W \Lambda W^{-1}$, and set $O = F\,W$. However, it is essential that the refactoring algorithm described below properly updates the administration itself in order for the entire versor simplification algorithm to have $O(n^3)$ time complexity.

5.4.2 Detecting Linear Dependency of the New Factor

Our goal is to add an extra factor $\mathbf{a}$ to an existing k-versor $\mathbf{V}$ and simplify the outcome, with $O(n^2)$ time complexity. The first step of our algorithm is to determine if $\mathbf{a}$ is linearly dependent on the factors of $\mathbf{V}$. If so, we determine the coordinate vector c such that $F\,c = a$, where a is the column vector $[\mathbf{a}]$. We derive:

$$\begin{aligned} F\,c &= O\,W^{-1}\,c = a, \\ c &= W\,O^T\,a. \end{aligned}$$

When $\mathbf{a}$ is not linearly dependent on the columns of F then the 'residual' $r = a - F\,c \neq 0$. In that case, we do not have to simplify the representation and the algorithm returns F^{k+2}, O^{k+2}, W^{k+2} and M_F^{k+2} as follows:

$$\begin{aligned} F^{k+2} &= \begin{bmatrix} F & a \end{bmatrix}, \\ r &= a - F\,c, \\ l &= \sqrt{r \odot r}, \\ O^{k+2} &= \begin{bmatrix} O & \frac{r}{l} \end{bmatrix}, \\ W^{k+2} &= \begin{bmatrix} W & -\frac{c}{l} \\ \mathbf{0} & \frac{1}{l} \end{bmatrix}, \\ M_F^{k+2} &= \begin{bmatrix} M_F & F^T\,a \\ a^T\,F & 1 \end{bmatrix}, \end{aligned}$$

where we assume that a has unit length. (The superscript indices $k+2$ are used to keep the labels of the output variables consistent with those used below).

5.4.3 Factoring out the Dependent Factor

When $\mathbf{a}$ is dependent on space the columns of F, we want to refactor the versor $\mathbf{f}_1\,\mathbf{f}_2\,\ldots\,\mathbf{f}_k$ such that $\mathbf{a}$ appears on the right side of the factorization:

$$\mathbf{f}_1\,\mathbf{f}_2\,\ldots\,\mathbf{f}_k\,\mathbf{a} = \mathbf{g}_1\,\mathbf{g}_2\,\ldots\,\mathbf{g}_{k-1}\,\mathbf{a}\,\mathbf{a},$$

so that we can eliminate the scalar $\mathbf{a}^2$ to reduce the number of factors of the versor. We do this by factoring $\mathbf{a}$ out of the versor, from left to right. The idea is to repeatedly refactor neighboring pairs of factors $\mathbf{f}_i$ and $\mathbf{f}_{i+1}$ in such a way that $\mathbf{a}$ becomes the right-most factor.

Example

As a concrete example consider simplifying the geometric product of a 2-versor $\mathbf{V} = \mathbf{f}_1\,\mathbf{f}_2$ and the vector $\mathbf{a}$, where

$$\begin{aligned}
\mathbf{f}_1 &= \mathbf{e}_1, \\
\mathbf{f}_2 &= \mathbf{e}_2, \\
\mathbf{a} &= \mathbf{e}_1 + \mathbf{e}_2.
\end{aligned}$$

We want to find a factorization of $\mathbf{V}$ such that

$$\mathbf{V}\,\mathbf{a} = (\mathbf{f}_1\,\mathbf{f}_2)\,\mathbf{a} = (\mathbf{g}_1\,\mathbf{a})\,\mathbf{a}.$$

Computing $\mathbf{g}_1$ is straightforward

$$\mathbf{g}_1 = \frac{\mathbf{f}_1\,\mathbf{f}_2}{\mathbf{a}}.$$

By substituting the values of the example we find:

$$\begin{aligned}
\mathbf{g}_1 &= \frac{\mathbf{e}_1\,\mathbf{e}_2}{\mathbf{e}_1 + \mathbf{e}_2} = \tfrac{1}{2}(\mathbf{e}_1 - \mathbf{e}_2), \\
\mathbf{V}\,\mathbf{a} &= \mathbf{e}_1\,\mathbf{e}_2\,(\mathbf{e}_1 + \mathbf{e}_2) \\
&= \tfrac{1}{2}(\mathbf{e}_1 - \mathbf{e}_2)\,(\mathbf{e}_1 + \mathbf{e}_2)\,(\mathbf{e}_1 + \mathbf{e}_2) \\
&= \tfrac{1}{2}(\mathbf{e}_1 - \mathbf{e}_2)\,(\mathbf{e}_1 + \mathbf{e}_2)^2 \\
&= \mathbf{e}_1 - \mathbf{e}_2.
\end{aligned}$$

For k-versors with $k > 2$ we apply this step for each pair of factors, starting at the left-most pair.

Refactoring Implementation

The vector

$$c = W\,O^T\,a$$

contains the coordinates of $\mathbf{a}$ relative to the columns of F. It specifies how much of each factor $\mathbf{f}_i$ should be shuffled out to the right to arrive at the desired factorization.

The algorithm for factoring out the dependent factor step is described next. The main steps are:

- Computing the transformation matrix for each pair of factors.
- Updating the versor representation and the respective extra administration matrices.

In the following, the superscript j on the variables indicates the iteration count, and ranges from 1 to $k+2$, with 1 being the input, and $k+2$ being the final result. So to bootstrap the algorithm, we set $F^1 = F$, $O^1 = O$, $W^1 = W$, $M_F^1 = M_F$ and $c^1 = c$. When equations are valid for every iteration, we sometimes omit the superscript indices to simplify notation. Subscript indices in brackets indicate matrix entries. The regular subscripts indicate the index of the factor.

If $c^j_{[j]} = 0$ we do not have to refactor for step j. In that case, we set the $j+1$ value of each variable to its j value ($F^{j+1} = F^j$, and so on). Otherwise we set

$$\mathbf{f}^{j+1}_{j+1} = c^j_{[j]}\mathbf{f}^j_j + c^j_{[j+1]}\mathbf{f}^j_{j+1}.$$

such that we shuffle the required component of $\mathbf{a}$ (as specified by coordinates $c^j_{[j]}$ and $c^j_{[j+1]}$) to the right. To keep the value of the versor constant we set $\mathbf{f}^{j+1}_j$ to the vector

$$\mathbf{f}^{j+1}_j = \frac{\mathbf{f}^j_j\,\mathbf{f}^j_{j+1}}{\mathbf{f}^{j+1}_{j+1}}.$$

To this end, two coefficients α, β can be computed such that

$$\mathbf{f}^{j+1}_j = \alpha\mathbf{f}^j_j + \beta\mathbf{f}^j_{j+1}.$$

Using these values, we can now initialize a 2×2 matrix R^j that is useful to perform the full update

$$R^j = \begin{bmatrix} \alpha & c^j_{[j]} \\ \beta & c^j_{[j+1]} \end{bmatrix}. \tag{5.20}$$

We embed this matrix in a $k \times k$ block diagonal matrix as follows

$$Q^j = \begin{bmatrix} I_{(j-1)} & 0 & 0 \\ 0 & R^j & 0 \\ 0 & 0 & I_{(k-j-1)} \end{bmatrix}$$

where each $I_{(x)}$ is an $x \times x$ identity matrix. We can now compute F^{j+1}

$$F^{j+1} = F^j\,Q^j.$$

It remains to update the other matrices. O^{j+1} is set to O^j since the subspace spanned by the columns of F^j is not altered by multiplying F with a full rank matrix Q^j. From this fact we derive how to update W^j:

$$O^j = F^j\,W^j = O^{j+1} = F^{j+1}\,W^{j+1} = F^j\,Q^j\,W^{j+1},$$

from which it follows that

$$W^j = Q^j W^{j+1},$$

and thus

$$W^{j+1} = (Q^j)^{-1} W^j.$$

Computing the inverse of Q^j requires only the inversion of the 2×2 non-singular matrix R^j. To update M_F^j, recall that $M_F^j = {F^j}^T F^j$ and derive

$$M_F^{j+1} = {F^{j+1}}^T F^{j+1} = {Q^j}^T {F^j}^T F^j Q^j = {Q^j}^T M_F^j Q^j.$$

Finally we update c by setting

$$c^{j+1} = c^j,$$

with the exception that $c^{j+1}_{[j]} = 0$ and $c^{j+1}_{[j+1]} = 1$. We note that — when implemented with care — each of these steps have $O(n)$ time complexity, since only two columns or rows of each affected matrix are modified.

5.4.4 Removing the Dependent Factor

The algorithm from the previous section factors out the dependent factor $\mathbf{a}$, such that it appears twice on the right of the factorization. The geometric product $\mathbf{a}\,\mathbf{a}$ then results in a scalar, and we have simplified our factorization. However, we have to update all matrices to account for the removal of the last column of F^{k+1}:

- F^{k+2} is set to F^{k+1}, with the last column removed.
- M_F^{k+2} is set to M_F^{k+1} with the last row and column removed.
- O^{k+2} should be computed such that its rows form an orthonormal set of vectors that span the rows of F^{k+2}. To do this, we should remove the reciprocal (with respect to the other columns of F^{k+1}) of $\mathbf{a}$ from the subspace spanned by O^{k+1}. (Simply removing $\mathbf{a}$ would not work in most cases, since $\mathbf{a}$ is not orthogonal to the other columns in F^{k+1}, in general). Note that the reciprocal is computed using the same Euclidean metric as is used to make the O orthonormal.

  The reciprocal of $\mathbf{a}$ is obtained as the last row of $W\,O^T$. To see why, recall from Section 2.8.5 that a reciprocal frame stored in matrix form has the property

  $$F^{-1}\,N\,F = I,$$

  where N is the metric matrix. In updating O, we use a Euclidean metric, so $N = I$ and we get

  $$F^{-1}\,F = I.$$

  From this we derive

  $$F^{-1}\,F = F^{-1}\,(O\,W^{-1}) = I,$$

  from which it follows that

  $$F^{-1} = (O\,W^{-1})^{-1} = W\,O^T.$$

Hence the reciprocal of **a** is the last row of $F^{-1} = W\,O^T$. Now that the reciprocal of **a** is known, a single reflection can be applied to reflect the last row of O^{k+1} to become scalar multiple of this reciprocal. The last row of this reflected O^{k+1} is then discarded to obtain O^{k+2}.

- Lastly, we have to compute a W^{k+2} that matches O^{k+2} (recall that $O = F\,W$). In principle, one could use $W^{k+2} = F^{-1}\,O$, since F^{k+2} and O^{k+2} are both known. But this operations is too expensive ($O(n^3)$). Instead, we update W much like we update O:

  The last column of W^{k+1} contains the coordinates of the last column of O^{k+1} with respect to the F^{k+1}-frame. We want to reflect it towards the vector which contains the coordinates of the reciprocal of **a** in the F^{k+1}-frame. We derive the following in order to compute that vector:

$$
\begin{aligned}
F^{-1} &= W\,O^T = W\,W^T F^T, \\
F^{-T} &= F\,W\,W^T.
\end{aligned}
$$

  This last equation shows that the last column of $W\,W^T$ contains the coordinates of the reciprocal of the last column in F. So to compute W^{k+2}, we perform a reflection that reflects the last row of W^{k+1} to the last column of $W^{k+1}{W^{k+1}}^T$, and discard the last row of the outcome. Note that this reflection must performed using the metric specified by the metric matrix M_F^{k+1}. The entire reason for keeping track of M_F during the algorithm was to make sure that the update of W^{k+2} has $O(n^2)$ time complexity!

5.4.5 Modifications for Signatures $\mathbb{R}^{n-1,1}$ and $\mathbb{R}^{1,n-1}$

Two modifications to the versor simplification algorithm are required to support signatures $\mathbb{R}^{n-1,1}$ and $\mathbb{R}^{1,n-1}$. In spaces with these signatures, null vectors exist, and this complicates the refactoring. I.e., during the refactoring of a versor, there could be a point where it is required to refactor a pair of vectors such that the second vector becomes a null vector, for example

$$e\,\bar{e} = \mathbf{x}\,(e + \bar{e})$$

(with $e \cdot e = 1, \bar{e} \cdot \bar{e} = -1$). This equation cannot be solved.

Our solution is to process *three* factors at a time (as opposed to two) during the refactoring algorithm. This allows us to safely embed any null vectors in the last two factors of the triplet. We call this the 3-versor refactoring problem, and discuss it in detail below.

The second modification is more straightforward. It only affects is the final step in the simplification algorithm: After the refactoring is complete, the last two factors of the versor and the new factor **a** collapse into a single vector **b**:

$$\mathbf{f}_{k-1}^{k+2}\mathbf{f}_{k}^{k+2}\mathbf{a} = \mathbf{b}$$

The vector $\mathbf{v}$ that is no longer spanned by the new factorization then is

$$\mathbf{v} = \mathbf{b} \odot (\mathbf{f}_{k-1}^{k+2} \wedge \mathbf{f}_{k}^{k+2}).$$

The reciprocal of $\mathbf{v}$ should be removed from the span of O and all other matrices should be updated accordingly. This can be done using the same methods as used in Section 5.4.4.

The 3-Versor Refactoring Problem

We define the 3-versor refactoring problem as follows. Given a 3-versor

$$\mathbf{V} = \mathbf{f}_1\,\mathbf{f}_2\,\mathbf{f}_3,$$

refactor $\mathbf{V}$ such that it is the product of three vectors

$$\mathbf{V} = \mathbf{g}_1\,\mathbf{g}_2\,\mathbf{g}_3$$

such that some arbitrary 'target vector' $\mathbf{t}$ that is specified as a weighted sum of $\mathbf{f}_1$, $\mathbf{f}_2$ and $\mathbf{f}_3$

$$\mathbf{t} = \alpha_1\,\mathbf{f}_1 + \alpha_2\,\mathbf{f}_2 + \alpha_3\,\mathbf{f}_3$$

can be constructed as the weighted sum of just $\mathbf{g}_2$ and $\mathbf{g}_3$:

$$\mathbf{t} = \beta_2\,\mathbf{g}_2 + \beta_3\,\mathbf{g}_3.$$

I.e., the essence is that we want to refactor $\mathbf{f}_1\,\mathbf{f}_2\,\mathbf{f}_3$ such that the target vector $\mathbf{t}$ can be formed as a weighted sum of the second and third factor. This 3-versor refactoring technique can then be applied repeatedly to shuffle $\mathbf{t}$ to the right of the entire versor, as in Section 5.4.3. We now show the solution to the 3-versor refactoring problem in spaces with signature $\mathbb{R}^{n,0}$, $\mathbb{R}^{0,n}$ $\mathbb{R}^{n-1,1}$ and $\mathbb{R}^{n-1,1}$.

If we assume for the moment that $\mathbf{t}$ could never be null, we could simply compute

$$\mathbf{g}_1\,\mathbf{g}_2\,\mathbf{g}_3 = (\mathbf{g}_1\,\mathbf{g}_2)\,\mathbf{t} = (\mathbf{f}_1\,\mathbf{f}_2\,\mathbf{f}_3\,\mathbf{t}^{-1})\mathbf{t},$$

that is, we divide $(\mathbf{f}_1\,\mathbf{f}_2\,\mathbf{f}_3)$ by $\mathbf{t}$, refactor the resulting 2-versor into two factors $(\mathbf{g}_1\,\mathbf{g}_2)$ and we are done.

However, the very reason for solving the 3-versor refactoring is that $\mathbf{t}$ *can* be null. So instead we proceed as follows. First we find a vector $\mathbf{g}_1$ (the first factor of the new factorization) that is orthogonal to $\mathbf{t}$ (in a Euclidean metric). That way we know that $\mathbf{t}$ will be contained in the second and third factors, because it definitely is not contained in the first factor. To obtain this vector $\mathbf{g}_1$, we compute the dual of $\mathbf{t}$ with respect to the subspace spanned by $\mathbf{f}_1, \mathbf{f}_2$ and $\mathbf{f}_3$:

$$\mathbf{B} = \mathbf{t} \odot (\mathbf{f}_1 \wedge \mathbf{f}_2 \wedge \mathbf{f}_3)^{-1}.$$

This $\mathbf{B}$ is a 2-blade. We factor $\mathbf{B}$ under the outer product to obtain two orthogonal (in a Euclidean metric) vectors $\mathbf{u}$ and $\mathbf{v}$:

$$\mathbf{B} = \mathbf{u} \wedge \mathbf{v}.$$

We could use either $\mathbf{u}$ or $\mathbf{v}$ as $\mathbf{g}_1$, as long as they are not null. If both $\mathbf{u}$ and $\mathbf{v}$ are null (in the algebra metric), we use $\mathbf{g}_1 = \mathbf{u} + \mathbf{v}$. If we now compute

$$\mathbf{g}_1\,(\mathbf{g}_1^{-1}\,\mathbf{f}_1\,\mathbf{f}_2\,\mathbf{f}_3) = \mathbf{g}_1\,\mathbf{g}_2\,\mathbf{g}_3,$$

we have solved the 3-versor refactoring problem. Since $\mathbf{g}_1$ and $\mathbf{t}$ are orthogonal in a Euclidean metric, $\mathbf{t}$ must be a weighted sum of $\mathbf{g}_2$ and $\mathbf{g}_3$: in that sense, we have shuffled $\mathbf{t}$ (or its components) one step to the right.

Modifications to the Refactoring Algorithm

Because the refactoring now affects three factors at a time, the matrix $\mathbf{R}$ of Equation 5.20 becomes a 3×3 matrix. This matrix must be initialized such that it transforms the factors (columns) $\mathbf{f}_1$, $\mathbf{f}_2$ and $\mathbf{f}_3$ to $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$, respectively. The fact that the matrix is now 3×3 does not fundamentally change the time complexity of the refactoring algorithm. Otherwise, no changes are required.

5.5 Complexity Analysis

In this section we describe the time and storage complexity of our multiplicative implementation method. We omit the constant factors on the complexities.

5.5.1 Storage Complexity

Our method stores k-blades in $n \times k$ matrices ($k \leq n$), so blades have $O(n^2)$ storage complexity; k-versors are stored in $n \times k$ matrices, and thus have $O(n^2)$ storage complexity. If we store the extra administration required for versor simplification along with each versor, three extra matrices are required for each versor. Matrix O is $n \times k$, matrix W is $k \times k$ and matrix M_F is $k \times k$. This is only a constant factor increase in the storage complexity.

5.5.2 Time Complexity

Table 5.1 enumerates the linear algebra operations [23] that are used in our implementation. Table 5.2 lists the time complexities of the blade algorithms we implement. The linear algebra operations used for each algorithm are also listed (see the shorthands in Table 5.1). Table 5.3 lists time complexity of the versor algorithms. In the versor table, we give two time complexities for some operations because they depend on whether the metric matrix is diagonal or not (the lower complexity being the diagonal matrix case).

The geometric product (or versor simplification) algorithm is a special case because it uses a custom algorithm. It applies the following steps $O(n)$ times (that is, once for each factor of the second operand):

algorithm	shorthand	time complexity
matrix composition	MC	$O(n^2)$
matrix-matrix multiply	MMM	$O(n^3)$
matrix-vector multiply	MVM	$O(n^2)$
diagonal matrix-matrix multiply	DMMM	$O(n^2)$
diagonal matrix-vector multiply	DMVM	$O(n)$
vector dot product	DP	$O(n)$
determinant	DET	$O(n^3)$
reduced QR decomposition	RQR	$O(n^3)$
full QR decomposition	FQR	$O(n^3)$
singular value decomposition	SVD	$O(n^3)$
eigenvalue decomposition of symmetric matrix	EIGS	$O(n^3)$

Table 5.1: *Linear algebra operations used by multiplicative implementation.*

operation	linear algebra usage	time complexity
reverse, grade inv.	-	$O(1)$
Euclidean dot product	MMM, DET	$O(n^3)$
outer product	MC, SVD, DET,	$O(n^3)$
matrix $\rightarrow$ blade	RQR, MMM, DET	$O(n^3)$
Euclidean dual	FQR, MC, DET	$O(n^3)$
general dual	MMM, FQR, MC, DET	$O(n^3)$
metric products	MMM, SVD, FQR, MC, DET	$O(n^3)$
`join`	MC, SVD	$O(n^3)$
`meet`	SVD, FQR, MC, DET	$O(n^3)$
addition	SVD, FQR, MC, DET	$O(n^3)$
inversion	MMM, FQR, MC, DET	$O(n^3)$
linear trans. of blades	MMM, RQR	$O(n^3)$

Table 5.2: *Time complexity of blade algorithms.*

operation	linear algebra usage	time complexity
blade $\rightarrow$ versor conversion	MMM, EIGS, DET	$O(n^3)$
reverse	MC	$O(n^2)$
inverse	$O(n)\times$ DMVM/MVM, DP	$O(n^3)$ or $O(n^2)$
application to blade / versor	$O(n^2)\times$ DVMV/MVM, DP	$O(n^4)$ or $O(n^3)$
2-blade exponential	MMM, RQR, FQR, MC, DET, EIGS	$O(n^2)$
geometric product	*custom algorithm*	$O(n^4)$ or $O(n^3)$

Table 5.3: *Time complexity of versor algorithms.*

- *Linear dependency check.* This uses a matrix-vector multiplication, so this step has a $O(n^2)$ time complexity.
- *Factoring out the dependent factor.* This step must be repeated $O(n)$ times. Updating the administration matrices is done in $O(n)$ time. So total time complexity for this step is $O(n^2)$.
- *Removing the dependent column.* This step uses matrix-vector multiplications and Householder transformations of matrices. The total time complexity is $O(n^3)$ for a general metric, or $O(n^2)$ when the metric is diagonal.

Hence the time complexity of the whole versor simplification algorithm is thus $O(n^4)$ (general metric) or $O(n^3)$ (diagonal metric).

In principle, an implementation could use a diagonal metric matrix internally, while making it appear to the user that any metric can be used. I.e., input / output should then also be converted appropriately, which takes $O(n^3)$ time per conversion. Hence we can conclude that each of the geometric algebra operations have $O(n^3)$ time complexity.

5.6 Implementation

We implemented our multiplicative method in a `Java` package called `Gallant` [17]. For linear algebra, we used the `Colt` library, which contains a `Java` port of `LAPACK`.

There are not many implementation details worth mentioning. Everything is implemented as described in the preceding sections. For some unary operations (such as computing the orthogonal complement), we cache the result, in case the result is required again later on. Otherwise we have not put much effort into low-level optimizations. Our main interest was in achieving $O(n^3)$ time complexity, regardless of the constant factor.

5.7 Benchmarks

We did not perform many benchmarks of the `Gallant` implementation. Our main goal was to achieve the theoretical time complexity of $O(n^3)$, and we did not care much about the constant factor. Figure 5.7 shows one benchmark where we measured the time required to perform 1,000 geometric products of blades. We used `MV` (see Section 4.1 and Section 7.4.4) as the baseline additive implementation; `MV` was the only additive implementation that we found to be able to handle high-dimensional spaces.

We have no explanation for the bump in the graph for `MV` around 12-D. It may be caused by some low-level detail of the CPU architecture, such as memory/cache access pattern or branch prediction.

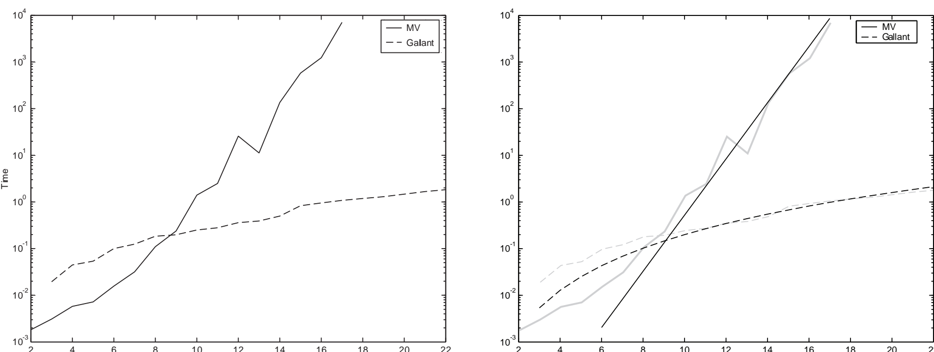


Figure 5.2: *Benchmarks of the multiplicative implementation (*`Gallant`*) and the additive implementation (*`MV`*).* On the left: *Benchmark of the time required to compute 1,000 geometric products of blades. Note that the y-axis is in log-scale.* On the right: *Theoretical time complexity (a rough fit by hand). To draw the graph we used* $t = 2^{2n}/2,000,000$ *for the additive implementation, and* $t = n^3/5,000$ *for the multiplicative implementation. Note that the benchmarks from the left figure are visible as light grey graphs. Also note that the start of the theoretical additive graph is missing because the values do not agree with the actual benchmarks: the overhead (constant factor) is much higher in for small values of* n.

5.8 Comparison to the Additive Implementation

The theoretical time and storage complexity makes clear that the multiplicative implementation should be superior to the additive implementation in terms of efficiency. In practice, the multiplicative implementation is slower for low dimensional spaces due to a relatively high constant factor. Benchmarks show that the tipping point lies around 10-D. Below that, applying linear algebra algorithms (which are often optimized for large matrices) to small matrices is more expensive than simply multiplying and adding some coordinates.

It is interesting to look at which operations are hard in each implementation. In the additive implementation computing the `meet` and `join` is hard, as is blade factorization. Both algorithms require a factorization of the blades, which is not easy to acquire in the additive representation.

On the other hand, addition of blades is hard in the multiplicative implementation. Moreover, we have not implemented an algorithm for addition of versors, nor do we expect that there is an efficient algorithm for doing so. This is one reason why we cannot compute exponentials of arbitrary bivectors yet (no addition of versors). Implementation of the geometric product is hard because it has no linear algebra equivalent (or we were not aware of it).

5.9 Discussion

Our multiplicative implementation method shows that there is an upper bound to the time- and storage-complexity of geometric algebra, which is $O(n^3)$ and $O(n^2)$, respectively. It also shows an important connection between geometric algebra and linear algebra, which may be useful both in practice and from an educational point of view.

In a way, our implementation shows that linear algebra already had all the pieces (fundamental algorithms) required to create a structured geometry framework. What geometric algebra does is show how to fit these pieces together. I.e., linear algebra is the 'assembly language' of geometry, where the instructions are operations such as matrix multiplication and QR factorizations. Geometric algebra can be seen as a high-level language on top of that. And as with any high level language, one may write an 'optimizing compiler' that maps geometric algebra operations to linear algebra operations in smarter ways than we have done in this chapter.

There are still some gaps in our multiplicative implementation; we cannot compute exponentials of arbitrary bivectors, and cannot simplify all versors in metrics like $\mathbb{R}^{4,2}$. The problem with the exponentials may be solved by converting the bivector into a sum of 2-blades-with-scalar-square. Versor simplification in metrics like $\mathbb{R}^{4,2}$ may be the harder problem. In those metrics there exist versors such as the one in Equation 5.19 that have more factors than their top-grade-part suggests. This affects the versor simplification algorithms profoundly, as it

is built on the assumption that the factors of a versor (in minimal factorization) are linearly independent.

Other future work includes:

- Replacing the use of the SVD in the outer product and the `join` with a more efficient algorithm. This algorithm would likely be based on a slightly modified version of QR factorization.
- Optimizing the representation of blades and versors. Blades may be stored as triangular as possible. Blades may be stored in dual form when they have more than $n/2$ factors [40]. This saves memory and processing time.
- Finding a more direct way to compute metric products. This may be possible by doing a projection followed by orthogonal complement.
- Finding a more direct way to compute `meet`.
- Developing a `C++` implementation with attention to low-level optimizations.
- An error analysis of the geometric product simplification algorithm.

Chapter 6

Case study: Using the Algebra to Implement a Ray Tracer

In this chapter we put the additive implementation to work by using it to implement a ray tracer in the `C++` programming language. A ray tracer is a program that takes as input the description of a scene (including light sources, models, and camera) and produces an image of that scene as seen from the camera. In this thesis, the ray tracer serves as a practical example of the use of geometric algebra and of our implementation `Gaigen 2` in particular. We will show how the conformal model of Euclidean geometry is useful for the specification and computation of the basic operations. Spelling out such an application to the source code level will bring up interesting choices between the representational possibilities the algebra offers. It allows us to show how the choice of (conformal) primitives affects performance, and vice versa.

Elegance vs Performance

Before we discuss the conformal ray tracer in detail, we first discuss some of the context in which this ray tracer was developed. The reasons for developing the ray tracer(s) were two-fold:

- To show — in a relatively objective way — how the 'elegance' of essential ray tracing equations depends on the choice of the model of geometry.
- To measure the performance of each model of geometry (and its implementation).

The idea was to implement the same basic ray tracing algorithm multiple times. Each time, we used a different model of geometry and/or a different implementation of that model. In total, about 20 versions of the ray tracer were implemented, over two generations. The following models of geometry were used:

- 3-D linear algebra. This is the most basic model, which uses vectors to represent anything from points to planes to directions. 3×3 matrices are used to represent rotations.
- 3-D geometric algebra. Much like 3-D linear algebra approach. 2-blades are used instead of normal vectors, and rotors are used to represent rotations. We also implemented a variant where outermorphism matrix representations are used to apply rotations.
- 4-D linear algebra (homogeneous coordinates), with Plücker coordinates used to represent lines and planes. 4×4 matrices are used to implement rotations, scaling and translations.
- 4-D geometric algebra (homogeneous model). Blades represent objects. Matrix representations of outermorphisms are used to implement rotations, scaling and translations.

- 5-D geometric algebra: the conformal model. This is the model we use in this chapter. Each geometric primitive has its own type of blade. All transformations are implemented using versors.

Once we had implemented all variants of the ray tracer, we collected those equations and primitives that are central to ray tracing for each variant. Appendix B shows these equations in tables, one table for representation of ray tracing primitives, and one table for ray tracing equations. From these tables, it is easy to see that the general trend is that the more advanced models use simpler, more generic representations and equations, with the conformal model version being the most elegant. We do not elaborate further on the relative elegance of the models; see [19] for full details.

As far as performance is concerned, the general trend is that the more advanced models have somewhat lower performance than the simple ones. These results are discussed in detail in Section 7.1.

6.1 Ray Tracing Basics

When you view a scene in the real world, rays originating from light sources interact with objects, 'picking up' their colors before they reach your eye to form an image. In standard ray tracing [20], this process is reversed: starting from a pixel of the image, a ray is cast through the optical center of the camera, and traced into the world. It encounters objects, reflects, refracts, and ultimately may reach one or more light sources. The product of all these interactions with the material and spectral properties of the objects and light sources determines the color, shading and intensity that needs to be rendered in the original pixel. This is called the 'shading computation'. To do this physically exact is impossible or slow, so many convenient shortcuts and simplifications have been introduced. Often, performing the shading computation requires new rays to be cast and traced. Among those are the 'shadow rays', which are used to check whether a direct path exists between some intersection point and a light source. New rays must also be cast if the material at the intersection point is reflective or if it is transparent (which requires a new ray to be created after refraction according to Snell's law).

From the algebraic point of view, the representation of rays and their intersections with objects are central to a ray tracer. The possibilities the conformal model offers for these elements will be carefully considered in their advantages and disadvantages, both algebraically and implementationally. Most of the time, algebraic elegance and computational efficiency go hand-in-hand, but sometimes we will be forced to choose and then we will opt for efficiency. Yet we will always remain within the framework of the conformal model and benefit fully from its capability of specifying all operations in terms of the geometrical elements rather than their coordinates.

A point to take into account when selecting conformal blades is the optimizations that `Gaigen 2` is able to make. Some seemingly expensive algebraic operations can be surprisingly cheap. For instance, to extract the Euclidean direction $\mathbf{u}$ from a line $\mathbf{L}$, we can write $\mathbf{u} = (\infty \wedge o) \rfloor \mathbf{L}$. This computation requires little effort since `Gaigen 2` simplifies it to collecting some of the coordinates of $\mathbf{L}$ into a new vector $\mathbf{u}$. Dualization with respect to basis elements, in particular the pseudoscalar, is also implemented by coordinate manipulation and does not require any real computation. Such implementational considerations will affect our representational choices in the ray tracer considerably.

The ray tracer is complete but limited. It includes a simple modeler with `OpenGL` visualization, in which one can interactively create the scene to be rendered. It can render only still frames (no animations). It can render only polygonal meshes (no curved surfaces). No kinematic chains are possible. It has support for texturing and bump mapping. Shading is adopted from the standard `OpenGL` shading model (ambient, diffuse, specular, see [39], Chapter 5). It supports reflection and refraction. We apply no optimizations to improve ray-model intersection test efficiency except simple bounding spheres and BSP trees.

The equations in this chapter are directly shown as `C++` code, so to follow the text one should read the code fragments. Since the code is usually close to the mathematical notation, it should be easily readable even for non-`C++` literate. The only catches are:

- The '`.`' denotes not the inner product, but access to a member variable.
- Since the '`.`' is already taken, we use the `<<` symbol to denote the left contraction $\rfloor$.
- We write `ni` for ∞ and `no` for o.

The code examples we show are sometimes polished or slightly changed for presentation. Most changes just involve renaming of variables. The full, unedited source code can be inspected at our website [11].

6.2 The Ray Tracing Algorithm

We present the basic outline of the ray tracing algorithm to define our terms and geometrical subproblems.

To render an image, the ray tracer casts rays through the optical center of the camera position and each pixel of the image. For anti-aliasing, multiple rays per pixel can be cast, each with a small offset.

For each cast ray, we want to find the closest intersection with the surface of a model. A quick intersection test is performed by first checking the intersection with a bounding sphere that encloses the entire model. If that test turns out positive, a more complex intersection test is done. The polygons of the models are stored in binary space partition (BSP) trees. A BSP tree recursively splits

space into two halves. This allows one to find the polygons of a model faces efficiently through a '3-D' binary search. We descend down the BSP tree until we discover what polygons the ray intersects. We pick the intersection that was the closest to the start of the ray and return it as the result of the intersection test.

Once the closest intersection point is known, we query the appropriate polygon to supply information about the material at the intersection point. Material properties include color, reflectivity, transparency, refractive index and surface direction.

Using the material properties and light source information, local shading computations are made. The shading equations that we use are the same as those employed for the fixed function pipeline of `OpenGL`, except that we do shadow checks: for a light source to illuminate the intersection point, there must be a clear path between them. So a *shadow ray* is cast from the intersection point towards each of the light sources to check the path. If the shadow ray cannot reach a light source unobstructed, that light source does not contribute to diffuse and specular lighting for the intersection point. Note that transparent objects are also treated as obstructions, because refraction causes our shadow ray to bend away from the light source, and we cannot easily compensate for that (see e.g., [29] for a solution).

The outcome of the shading computations contributes to the final color of the pixel that the ray originated from. If the material is reflective or refractive, new rays are cast appropriately. The colors returned by these rays also contribute to the final color of the pixel that the ray originated from.

In the next three sections we discuss the low level geometric details of each part of the ray tracer. We start with the representation, followed by the tracing of the rays, and end with shading computations. We do not discuss interactive modeling of the scene, which involves construction of various types of versors. This is discussed in [12].

6.3 Representing Meshes

To represent the shape of objects in the scene we need to model them. We will use a triangular mesh, consisting of a set of vertices and triangular faces, bounded by edges. Figure 6.1 shows an example of such a mesh in both solid and wire frame rendering.

Meshes are read from `.rtm` files, which describe the faces and vertices of the mesh. The description of a face is simply a list of indices of the three vertices. The vertex description is more complicated, specifying position, local surface direction and texture coordinates. Finally, the mesh file contains material properties, like reflectivity, transparency, and optional texture information.

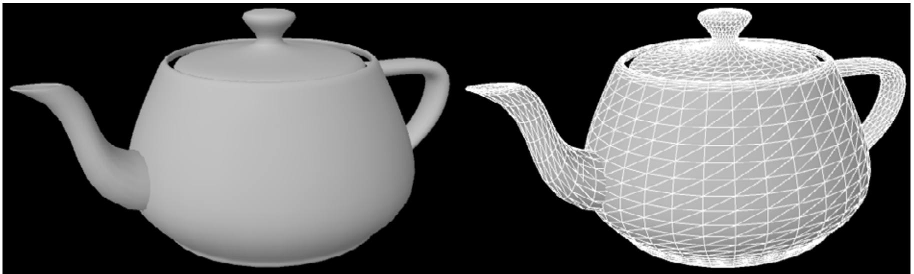


Figure 6.1: A polygonal mesh rendered in solid and with its wire frame model superimposed.

Vertices

Vertices are stored in the `vertex` class. Vertices naturally require a 3-D position $\mathbf{x}$. We choose to use a normalized conformal point (of the form $o+\mathbf{x}+\frac{1}{2}\mathbf{x}^2\,\infty$) to represent the position of a vertex. Such a normalized conformal point requires four coordinates for storage (the constant o does not have to be stored). Alternatively we could have used a flat point (of the form $(o+\mathbf{x})\wedge\infty$) which would require one less coordinate, but this would be annoying in the algebra of the spanning operation (in which the plane determined by three points is the outer product of infinity and three regular points, not flat points).

To initialize the points, we read the x, y, and z coordinates from the `.rtm` and construct a point from them:

```
normalizedPoint pt = cgaPoint(x * e1 + y * e2 + z * e3);
```

`cgaPoint()` is a function that constructs a conformal point. It implements Equation 2.41:

```
// Returns a point at the location specified by 3-D Euclidean vector p
normalizedPoint cgaPoint(const vectorE3GA &p)
{
   return p + no + 0.5 * norm_e2(p) * ni;
}
```

The `norm_e2()` function computes the squared Euclidean norm, i.e., the sum of the squares of all coordinates.

It is common practice to approximate the look of a smooth surface by interpolating the local surface direction at the vertices across each faces. This affects only the shading computations, and does not change the actual shape of the model (hence the contours of a model show that it is actually a polygonal mesh). The mesh files specify the *normals* for each vertex. We construct a free vector (see Section 2.11.3) from these coordinates. This free vector is then dualized into a free 2-blade that is used to represent the surface direction:

```
// nx, ny and nz are floats.
// They represent the classical surface normal as a direction.
freeBivector att = dual((nx * e1 + ny * e2 + nz * e3) ^ ni);
```

Each vertex also has an associated 2-D point that is used for *texture mapping*, which is a common way of coloring the surface of a 3-D model by 'wrapping' a 2-D image around it. In computer graphics, texture mapping is also used to apply 'bumps' to the surface of a model (bump mapping, or displacement mapping), to specify the surface transparency, and so on. Because we will not be doing a lot of geometry on the texture coordinates, we use *normalized flat 2-D points* to represent them (this requires only 2 coordinates). Note that these points live in a different space than the rest of our geometry, namely in the 2-dimensional carrier space of the texture image. So, putting it all together, the storage part of the `vertex` class looks like

```
class vertex {
  normalizedPoint pt;               // position of the vertex
  freeBivector att;                 // local surface direction
  normalizedFlatPoint2d texPoint;   // position in 2-D texture image(s)
};
```

Representing Vertices with Tangent 2-Blades

An alternative way to represent the position and the local surface direction of a vertex is to use a tangent 2-blade (see Section 2.11.3). We can combine point $\mathbf{p}$ and attitude $\mathbf{A}$ into one 'vertex primitive' with their combined properties and the clearer semantics of a tangent 2-blade: $\mathbf{V} = \mathbf{p} \rfloor (\mathbf{p} \wedge \mathbf{A} \wedge \infty)$.

However, there are some efficiency concerns with this tangent 2-blade representation, for we cannot extract its position 'for free', though its direction can be. The direction of a tangent 2-blade $\mathbf{V}$ is the free 2-blade algebraically given by $-(\infty \rfloor \mathbf{V}) \wedge \infty$, and this can be extracted by taking the $o \wedge \mathbf{e}_i \wedge \mathbf{e}_j$ coordinates of $\mathbf{V}$ and making them into $\mathbf{e}_i \wedge \mathbf{e}_j \wedge \infty$-coordinates of an attitude element $\mathbf{A} \wedge \infty$. (`Gaigen 2` performs such optimizations automatically). But the position of a tangent 2-blade cannot be extracted in this manner from $\mathbf{V}$. A method to get the flat point at the location of $\mathbf{V}$ is $(\infty \rfloor \mathbf{V}) \rfloor (\mathbf{V} \wedge \infty)$, which you may recognize as the intersection of the line perpendicularly through $\mathbf{V}$ with the plane that contains $\mathbf{V}$. On the coordinate level, implementing this equation involves some multiplications and additions, rather than merely coordinate transfer. Since the position of the vertices is required often during rendering to do interpolation, the cost of its extraction made us decide against using the tangent 2-blade representation of vertices for the ray tracer. This is an example where an algebraically more elegant possibility is overruled by efficiency considerations.

Faces

The `face` class describes the faces of the polygonal model. The main data in this class are the indices of the vertices that make up the face. These indices are read

from the `.rtm` file and stored in an integer array `vtxIdx` and accessed through a function `vtx()`, which returns the vertex.

The ray tracer then precomputes some blades to speed up later computations using standard conformal operations:

- The plane that contains the face:

```
// wedge together the position of vertex 0, 1, 2 and infinity
pl = vtx(0).pt ^ vtx(1).pt ^ vtx(2).pt ^ ni;
```

- The three conformal lines along the three edges of the polygon:

```
// each edge line is its (start vertex)^(end vertex)^infinity
edgeLine[0] = vtx(0).pt ^ vtx(1).pt ^ ni;
edgeLine[1] = vtx(1).pt ^ vtx(2).pt ^ ni;
edgeLine[2] = vtx(2).pt ^ vtx(0).pt ^ ni;
```

The edge lines are used to compute whether a specific line intersects the face and they come in handy to compute the direction of edges. For example,

```
((ni ^ no) << edgeLine[0]) ^ ni
```

is the free vector along the direction of edge 0. With these details, the storage part of the `face` class looks like:

```
class face {
  // indices of the three vertices
  int vtxIdx[3];

  // lines along the edges of the face
  line edgeLine[3];

  // the plane of the face
  plane pl;
};
```

Computing the Bounding Sphere

For each mesh, a bounding sphere is computed that contains all the vertices of the mesh. This sphere is stored as a regular conformal sphere. As we mentioned before, this is a common trick to speed up intersection computations: when we need to test for the intersection of some primitive with a model, we do not do a detailed intersection test straight away but instead first check if it intersects the bounding sphere. If so, then we proceed with the detailed intersection test, otherwise we can report that no intersection was found. The bounding sphere does not need to be tight (although the tighter the better). For simplicity we compute a sphere of which the center is at half the extent of the data set in the directions of the coordinate axes. The code is shown in Figure 6.2.

```
/*
Compute the center of the bounding sphere:
-create three orthogonal planes through the origin:
-compute minimal and maximal signed distance of each
vertex to these planes
*/
dualPlane pld[3] = {
   dual(no ^ e2 ^ e3 ^ ni),
   dual(no ^ e3 ^ e1 ^ ni),
   dual(no ^ e1 ^ e2 ^ ni)
};
double minDist[3] = {1e10, 1e10, 1e10};
double maxDist[3] = {-1e10, -1e10, -1e10};
for (int i = 0; i < vertices.size(); i++) {
   for (int j = 0; j < 3; j++) {
      minDist[j] = min(Float(pld[j] << vertices[i].pt)), minDist[j]);
      maxDist[j] = max(Float(pld[j] << vertices[i].pt)), maxDist[j]);
   }
}

// Compute the center 'c' of the bounding sphere.
normalizedPoint c(cgaPoint(
   0.5 * (minDist[0] + maxDist[0]) * pld[0] +
   0.5 * (minDist[1] + maxDist[1]) * pld[1] +
   0.5 * (minDist[2] + maxDist[2]) * pld[2]));


// Compute the required radius 'r' (actually, -radius^2/2) of the bounding sphere:
double r = 0.0;
for (int i = 0; i < vertices.size(); i++)
   r = min(vertices[i].pt << center, r);

// Construct the bounding dual sphere:
boundingSphere = c + r * ni;
```

Figure 6.2: *Computing the bounding sphere of a mesh given the* `vertices`*.*

Constructing the BSP Tree

To do a detailed intersection test of a model with, say, a line, we could simply check the intersection of the line with every face of the model. For complex models with many faces, this can become computationally expensive.

To speed up the intersection test, we apply a standard technique similar to bounding spheres, but employed recursively. First we search a plane that has about half of the vertices of the model at its back side, and the other half of the vertices in front. We sort the vertices according to what side of the plane they lie, and then recurse: for each of the halves of the vertex sets, we search for another plane that splits these halves in half. And so on, until we have partitioned the space into chunks that each contain a reasonable number of vertices. This is called a *binary space partition tree* (BSP tree). The intersection test of a line with such a BSP tree is discussed 6.5.3.

During the construction of the BSP tree, the main geometric computation is the selection of the plane to partition the space. We use the following approach: for the subdivision planes, we cycle, modulo 3, through translated versions of the three orthogonal planes $o \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \infty$, $o \wedge \mathbf{e}_3 \wedge \mathbf{e}_1 \wedge \infty$ and $o \wedge \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \infty$. The remaining difficulty is to compute the translation that halves the vertex set in the chosen direction. This is done through applying the Quicksort algorithm. All of these computations are just standard techniques cast into a conformal notation. The code is shown in Figure 6.3.

6.4 Modeling the Scene

We need a way to place our light sources, cameras and polygon models. This requires methods to represent and apply Euclidean transformations, and methods to interactively manipulate these transformations.

In the conformal model, the natural choice to represent Euclidean transformations is using rotors. In Section 2.11 we showed how conformal rotors can be used to represent and apply rotations, translations, reflections and uniform scaling. These are the most basic transformations used in computer graphics, and they would classically be represented with 4×4 matrices. We use rotors throughout the ray tracer to handle transformations.

To apply a conformal rotor to a blade, code like the following is used:

```
// xf is the rotor (xf is short for 'transform').
// ptLocal is a point that we want to take from a
// local frame to the global frame.
pt = xf * ptLocal * reverse(xf);
```

Our ray tracer also contains an interactive modeler that allows the user to modify these transformations interactively, but we do not discuss it here. There interested reader may refer to [12].

```
/*
Get the 'base plane' we are going to use for partitioning the space:
The integer 'basePlaneIdx' tells us what plane to use:
*/
dualPlane dualBasePlane;
if (basePlaneIdx == 0) dualBasePlane = dual(no ^ e2 ^ e3 ^ ni);
else if (basePlaneIdx == 1) dualBasePlane = dual(no ^ e3 ^ e1 ^ ni);
else dualBasePlane = dual(no ^ e1 ^ e2 ^ ni);

/*
Compute the signed distance of each vertex to the plane.
This distance is stored inside the vertices,
so we can use it to sort them:
*/
for (int i = 0; i < nbVertices; i++)
  vtx(i).setSignedDistance(
    dualBasePlane << getVertex(vertex[i].idx).pt());

/*
Use quicksort to sort the vertices.
The function 'constructBSPTreeSort()' compares the
signed_distance fields of the vertices.
*/
qsort(vertex, nbVertices, sizeof(bspSortData), constructBSPTreeSort);

/*
The required translation of the base plane is now simply the
average of the signed distance of the two vertices that came out
in the middle after sorting:
*/
splitIdx = nbVertices / 2;
double d = 0.5 * (vertex[splitIdx-1].signedDistance + vertex[splitIdx].signedDistance);

/*
We now translate the dual base plane, (un)dualize it and store
it in the pl of the BSP node:
*/
pl = dual(dualBasePlane - d * ni);
```

Figure 6.3: *Constructing a bisection plane for a BSP tree.*

6.5 Tracing the Rays

With our scenes (interactively) modeled and represented, we arrive at the core of the ray tracer. In this section we discuss how rays are represented, how they are cast, and how the ray tracer computes the intersection of rays with the models.

6.5.1 The Representation of Rays

Since rays interact with almost every part of our ray tracer, we should choose their representation carefully. We consider several alternatives and pick the one that we believe works best in the context of the operations that need to be performed by the ray tracer.

Classical Representation

A ray is a directed half-line, with a position (from where it was cast) and a direction. Classically we would represent this as the point (requiring three or four coordinates) and a direction vector (requiring three coordinates).

Point-Line Representation

A sensible generalization of this classical representation in CGA would be to use a conformal point and a conformal line through it. This is of course somewhat redundant, since the line contains partial position information too. But it would seem convenient to have the ray in line representation so it can be used directly in intersection testing.

However, this representation requires a lot of coordinates for storage (four for the point, six for the line). A regular conformal point might seem a good way to represent position, but it is awkward in a ray tracer. When we intersect the ray (line) with a plane (the most common intersection operation in the ray tracer), we get a flat point as a result. To compute the representation of a cast ray at that location, that flat point would have to be converted to a regular point, which is extra work not warranted by the geometry of the situation. Also, the point/line combination is expensive to transform to and from coordinate frames. Finally, even though it is algebraically possible to reflect and refract the lines by substituting them in the classical equations for reflection and refraction of directions, this is computationally rather expensive.

Tangent Vector Representation

A third idea is to use a tangent vector to represent rays. It seems perfect: a tangent vector has position and direction, all in one. This would represent a ray as a single blade, conforming to the geometric algebra philosophy of representing one object by one blade, whenever possible. Besides, a tangent vector can be turned into its carrier line at any time (just 'add' infinity using the outer product).

Yet this representation suffers from the same problems as the point-line representation. A tangent vector requires ten coordinates for storage, and this makes it expensive in operations, even more so because the locational part of a tangent vector cannot be extracted 'for free' by coordinate transfer (this problem was already described for tangent bivectors in Section 6.3).

Rotor Representation

Another possible ray representation is as a rotor, since a translation/rotation rotor naturally represents a position and a direction (the axis of a general rotation). It requires just seven coordinates for storage. But the position and direction information cannot be extracted 'for free' from a rotor by coordinate extraction. Instead, we have to apply the rotor to a blade to determine such properties: $\mathbf{V}\, o\, \mathbf{V}^{-1}$ is the position of the ray (as a conformal point), and $\mathbf{V}\,(\infty \wedge \mathbf{e}_3)\,\mathbf{V}^{-1}$ is the direction of the ray. Both these equations are relatively expensive to evaluate.

Flat Point-Free Vector Representation

For our most efficient implementation we decided to use the less elegant flat point-free vector representation. The flat point gives the position, the free vector gives the direction. The choice for a flat point may seem unnatural. You cannot directly create a line from a flat point $\mathbf{p} \wedge \infty$ and a free vector $\mathbf{u} \wedge \infty$, since they both contain an infinity factor. This is solved by removing the infinity factor from the free vector, before wedging them together:

```
// fp is a flat point, fv is a free vector
line l = fp ^ (-no << fv)
```

The number of coordinates of a 'flat point, free vector' pair is as low as the classical representation. But, unlike the classical representation, the flat point and free vector automatically have clear semantics in operations. The point responds to both translation and rotation, the free vector only to rotation (it is translationally invariant).

Still, the disadvantage of this mixed CGA representation is that we miss out on automatic treatment semantics for the ray as a whole; the programmer has to remember that the flat point and the free vector belong together and what their combination 'means'. Had we used a tangent vector, there would be no possibility to get its ray-like properties wrong in transformations. Now the programmer must specify semantics that are not 'intrinsic' to the algebra by putting the blades together in one class and adding comments that give detailed information on what each blade represents. For instance, our `ray` class becomes:

```
class ray {
public:
   flatPoint pos;          // position of the ray
   freeVector direction;   // direction of the ray
};
```

Another downside of the 'flat point with free vector' representation is that distance computations become less elegant. In ray tracing, when we search along the ray for the first intersection we need to compute the distance of candidate intersections with the start of the ray. If we had the ray position as a regular conformal point, we could do that with a simple inner product. With the ray position as a flat point, we have to use the classical approach of subtracting the two points and computing the Euclidean norm (squared).

6.5.2 Casting Rays

There are four occasions where we have to cast new rays:

- The initial casting of a ray through camera center and image plane.
- For shadow rays, to test whether a light source is visible from a specific point.
- When reflecting a ray.
- When refracting a ray.

The geometry involved in doing reflection and refractions is treated in separate sections. Here we just show how camera rays and the shadow rays are cast.

Camera Rays

To render an image, we trace rays through each pixel in the image. If anti-aliasing is required, we cast multiple rays through each pixel, each ray offset by a small random vector. The code for casting the initial rays is shown in Figure 6.4:

Note that one could say that we make direct use of (image) coordinates to specify the directions of the rays ($x\,\mathbf{e}_1,\ y\,\mathbf{e}_2$), which is a somewhat against our philosophy, but we believe this is the most sensible way to start the rays. Trying to avoid coordinate usage here would make things more awkward instead of simpler.

Shadow Rays

Shadow rays are cast to check whether there is a clear path between a surface point and a light source. The surface point is naturally represented as a flat point, since it comes from an intersection computation.

The light source position is encoded in a rotor. We can extract a point from that rotor by using it to transform the flat point at the origin:

```
flatPoint lightPoint =
   light->getXF() *
   (no ^ ni) *
   reverse(light->getXF());
```

```
double pixelWidth = cameraFOVwidth / imageWidth;
double pixelHeight = cameraFOVheight / imageHeight;

// for each pixel in the image:
for (int y = 0; y < imageHeight; y++ {
   for (int x = 0; x < imageWidth; x++ {
   /*
   Compute the direction of the ray if it has to
   got to go through sensor pixel [x, y]:
   */
   freeVector rayDirection =
      ((x * pixelWidth - 0.5 * cameraFOVwidth) * e1 +
      (y * pixelHeight - 0.5 * cameraFOVheight) * e2 +
      e3) ^ ni;


   // sample multiple times, accumulate the result in pixelColor:
   color pixelColor;
   for (int s = 0; s < multisample; s++) {
      /*
      Add a small perturbation within the square.
      To generate random numbers, we call mt_rand(),
      the Mersenne twister random number generator.
      */
      freeVector perturbedRayDirection =
         rayDirection +
         (((mt_rand() - 0.5) * e1 * pixelWidth
         (mt_rand() - 0.5) * e2 * pixelHeight) ^ ni);

      // make the direction unit:
      freeVector unitRayDirection = unit_e(perturbedRayDirection);

      // trace the ray from camera position towards 'unitRayDirection':
      ray R((C.getPosition(), unitRayDirection);
      pixelColor += trace(R, maxRecursion);
   }
}
```

Figure 6.4: *Casting rays for each pixel of the camera sensor.*

```
// compute the line representation of the ray:
line rayLine = ray.pos ^ (-no << ray.direction);

// intersect the line and the sphere
pointPair intersection = dual(rayLine) << boundingSphere;

/*
Check if intersection is a real circle by computing
the radius squared:
*/
if ((intersection << intersection) > 0) {
   /*
   Intersection with bounding sphere detected:
   . . .
   */
}
```

Figure 6.5: *Computing whether a ray intersects a bounding sphere.*

This computation can be done before we start rendering the scene, since the position of the light does not change during the rendering of a single frame.

The ray-model intersection test returns the `surfacePoint` where the ray intersects the model, so we now have two flat points that we can simply subtract to get a free vector that points from the model surface to the light:

```
/*
'surfacePoint' is a flat point at the surface of some model.
'lightPoint' is a flat point at the position of the light for
which we have to check visibility:

We can subtract the points because they are both normalized.
*/
freeVector shadowRayDirection =
   unit_e(lightPoint - surfacePoint);
```

This gets us the representation of the shadow ray as `surfacePoint` and `shadowRayDirection`.

6.5.3 Ray-Model Intersection

A typical ray tracer spends most of its time finding if and where rays intersect models. Our ray tracer does this in two steps: it first checks if the ray intersects the bounding sphere. If so, a descent down the BSP tree follows, until it eventually finds actual intersections with faces of the model.

Bounding Sphere Test

The bounding sphere computation is quite simple. It is shown in Figure 6.5.

A Trip Down the BSP Tree

The descent down the BSP tree is more complicated. Remember that the BSP tree recursively splits space into halves. The leaves of the tree contain the model faces that lie in that particular partition of space. During an intersection test, the goal of the BSP tree is to quickly arrive in those faces that the ray actually intersects. For instance, the first partition plane of the BSP tree divides space into two halves. Unless the ray happens to be parallel to this plane, it always intersects both halves, so we split the line in two, and each line segment goes down to the appropriate half of the BSP tree to be tested for intersections. Of course, we can take advantage of the fact that no faces lie outside of the bounding sphere: before we start our descent, we clip the ray against the bounding sphere, resulting in the initial line segment.

On the geometric algebra side of things, what this all boils down to is that we require a representation for line segments. This specific representation must be picked so that it works well while descending down a BSP tree.

We decided to use a two flat points to represent such line segments, one point for the start of the segment and one point for the end of the segment. We now show what the BSP descent algorithm looks with this choice of representation. Afterwards, we briefly discuss the alternative representation of CGA point pairs to represent line segments.

To bootstrap the BSP descent algorithm, we need to have a line segment that represents the ray as clipped against the bounding sphere. As input for the recursive descent, we take the point pair named 'intersection' from the previous code fragment, and dissect that into two flat points using the following point-pair dissection code:

```
// Dissects a point pair 'pp' into two flat points 'fpt1' and 'fpt2'
void dissectPointPair(const pointPair &pp,
    normalizedFlatPoint &fpt1, normalizedFlatPoint &fpt2) {
  double n = sqrt(pp << pp);
  dualPlane v1 = ni << pp;
  dualPlane v2 = n * v1;
  dualSphere v3 = v1 << pp;

  double scale = 1.0 / (v3 << ni);
  fpt1 = (scale * ni) ^ (v3 + v2);
  fpt2 = (scale * ni) ^ (v3 - v2);
}
```

We sort the two points according to distance from the start of the ray. This is useful because it allows us to prune part of the BSP tree descent: we want to find the closest intersection of the ray with the model. So we first check the parts of the BSP tree closest to the start of the ray to see if we find any intersection there. If find an intersection there, then we don't have to test to the other halve.

We determine what part of the tree to descend in first by checking on what side of the plane the start of the ray lies:

```
// partitionPlane is the plane that splits the space in two halves.
if ((ray.pos << dual(partitionPlane)) > 0.0) {
    // descend front side first . . .
}
else if ((ray.pos << dual(partitionPlane)) < 0.0) {
    // descend back side first . . .
}
else {
    // descend the side that's in the ray direction
}
```

For the actual descent of the tree, we first check on what side(s) of the partition plane our line segment lies (this is not necessarily the same side as the start of the ray):

```
/*
partitionPlane is the plane that splits the space in two halves.
We compute the signed distance of the two flat points to the
plane:
*/
double sd1 = no << (dual(partitionPlane) << fp1);
double sd2 = no << (dual(partitionPlane) << fp2);
```

If the line segment lies on both sides of the partition plane, we have to compute the intersection point of the line segment and the plane. Otherwise, we can simply descend down the appropriate node in the BSP tree.

The following code checks if we have to compute the intersection point, and computes it:

```
// If one signed distance is negative, and the other is positive,
// then the line segment must lie on both sides of the plane:
if (sd1 * sd2 < 0.0) {
    // Compute the intersection point:
    // First compute the line representation of the ray:
    line rayLine = ray.pos ^ (-no << ray.direction);
    // Then intersect rayLine line and the partitionPlane:
    flatPoint fpI = unit_r(dual(partitionPlane) << line);

    // descend further down both sides of the BSP tree
    // . . .
}
```

A degenerate case occurs when the line segment lies exactly in the partition plane. We then simply descend to both children of the current BSP node.

Representing Line Segments with Point Pairs

Instead of pairs of flat points, we could use point pairs (i.e., 0-spheres) to represent line segments. But at some point we will have to dissect the point pair into two points to form two new point pairs (each on a different side of the partition plane), and this would then require the rather expensive point-pair dissection code shown above. It is computationally more efficient to have the ends of the segments readily available, and use the dissection code only once at the start of the descent.

6.5.4 Reflection

The following function implements reflection of a ray in a model surface. It does so by forming a plane parallel to the local surface attitude, and reflecting the ray direction (a free vector) in it. See Section 2.9.1 for discussion of the reflection equation (using the reverse instead of the inverse is allowed because `spt.getAtt()` is a unit blade).

```
color scene::reflect(const ray &R, int maxRecursion, const surfacePoint &spt) const {
   /*
   Reflect the ray direction in the surface attitude,
   and spawn a new ray with that direction at the surface point.
   */
   ray reflectedRay(
      spt.getPt(),                       // starting position of new ray
      -((spt.getAtt() ^ no) *
      R.get_direction() *
      reverse(spt.getAtt() ^ no))      // direction of new ray
   );

   return trace(reflectedRay, maxRecursion-1);
}
```

6.5.5 Refraction

Like reflections, refractions are also computed as part of the shading computation. Originally, we formulated a pure geometric algebra implementation of the refraction function that operated directly on conformal lines (which we used at that time to represent rays), and which constructed several spheres in the process. But since this version turned out significantly slower than the classic solution, we opted to use the latter instead. The equation for that solution reads

$$\mathbf{u}' = \left(\mathrm{sign}\left(\mathbf{n}\cdot\mathbf{u}\right) \sqrt{1-\eta^2+\left(\mathbf{n}\cdot\mathbf{u}\right)^2\eta^2} - \left(\mathbf{n}\cdot\mathbf{u}\right)\eta \right)\mathbf{n} + \eta\mathbf{u}, \qquad (6.1)$$

where $\mathbf{n}$ is the surface normal, $\mathbf{u}$ is the ray direction and η is the refractive constant. Conversion of this equation to `C++` is rather straightforward (but rather long), so we refrain from showing the full code here. The interested reader can read it at `http://www.geometricalgebra.net`.

6.6 Shading

The ray tracer performs shading computations for every cast ray that intersects a model (unless it is a shadow ray). The shading computations should approximate how much light the material reflects along the direction of the ray (and thus towards the camera). We do not discuss the shading computations in detail because they do not differ much from the classical computations. Even in the

conformal model, shading computations are most conveniently performed using 3-D Euclidean vectors. For completeness, we briefly list the steps involved in shading:

- Before the ray tracer can perform any shading computations, it must interpolate the surface properties defined at the vertices for the intersection point.
- If the model has a bump map applied to it, we have to modulate the surface attitude according to that bump map. This again involves interpolation.
- Finally we have to perform the actual shading computations. Our ray tracer mimics the simple fixed function pipeline shading model of `OpenGL` [39], although we perform shading per intersection point, instead of per vertex.

6.7 Evaluation

We have shown most of the geometry involved in implementing a classical ray tracer using the conformal model. We have emphasized that there are usually several ways to represent computer graphics primitives in the conformal model of Euclidean geometry.

To built the most efficient conformal model-based ray tracer, we picked the most efficient representations we could find. This required us to 'add some water to the wine'. I.e., to achieve high performance we used some unelegant tricks such as representing a ray as the combination of a flat point and a free vector, instead of a single tangent vector. In other, less computing-intensive applications, one pick the more elegant representations, since we do think it is ultimately beneficial represent each geometric concept with a single type of blade. (In fact, we also implemented a *nice* version of the conformal ray tracer. In that version, we used sematically best representations regardless of their performance. The effect of these choices is discussed in the next chapter.)

Still, the direct use of coordinates in the equations and the source code has been eliminated, except for places where they were actually useful (e.g., systematically generating the positions of all pixels in a 2-D image). This would not be possible with classical homogeneous coordinates. One may pick up any computer graphics text on the same topic and find many pages filled with coordinate-based equations.

The equations used in the conformal ray tracer are also more generic and elegant than what would be used for different models of geometry, as may be seen in the tables of Appendix B.

In the next chapter we will compare the performance of our ray tracer to that of more traditional implementations (that is, using linear algebra to implement geometry). It turns out that the use of the conformal model makes the ray tracer about 25% slower. The *nice* version of the conformal ray tracer is 59% slower.

Chapter 7

Benchmarks

This chapter presents benchmarks of our additive geometric algebra implementation `Gaigen 2`. The main benchmark is based on the ray tracer of Chapter 6. Through the various implementations of the ray tracer, we can compare the performance of geometric-algebra-based geometry to that of traditional alternatives. We also present several smaller benchmarks.

We did not extensively benchmark the multiplicative implementation of Chapter 5. The advantage of this implementation approach should be clear from its theoretical time and storage complexity. The benchmark in Figure 5.7 shows that this also works out in practice.

7.1 Ray Tracer Benchmarks

As explained in the introduction of Chapter 6, one of the reasons to develop the ray tracer was to compare the performance of geometric algebra to that of the traditional approaches in a semi-realistic application. The idea was to implement the same basic ray tracing algorithm multiple times, each time using a different model of geometry. The five models we used are listed at the start of Chapter 6.

The ray tracer went through two generations. The first generation [19] used `Gaigen 1` to implement the geometric algebra. It also compared the performance of `Gaigen 1` to `CLU`, which is an alternative GA implementation (see Section 4.1). The second generation [12, 18] was a clean-up of the code and equations of the first generation, used `Gaigen 2` to implement geometric algebra, and no longer made the comparison to `CLU`.

7.1.1 First Generation Ray Tracer

The benchmarks of the first generation are shown in Table 7.1. The benchmarks show that `Gaigen 1` is about 2.5× (*3-D GA*) to 6× (*5-D GA*) slower than the *3-D LA* model. They also show that a naive GA implementation like `CLU` is two orders of magnitude slower than the hand-coded optimized *3-D LA* implementation.

Some notes on the benchmarks are in order:

- The *model* column states the geometry approach (from the list above). The model named *3-D GA OM* is 3-D geometric algebra with outermorphism matrix representations used to apply rotations.
- The *implementation* column specifies how the models were implemented. The linear algebra implementations were written and optimized by hand.
- We benchmarked the rendering time in two ways. The *full rendering time* column is how long a regular rendering run took. However, a significant portion of that time is spent on finding ray-object intersections. This is a relatively repetitive use of geometry, i.e., only ray-plane intersections and ray-sphere intersections are performed. Therefore such a benchmark measures mostly how fast each implementation can perform these. By also

model	implementation	full render time	render time w.o. BSP	exec. size	run time mem. usage
3-D LA	hand-written	1.00×	1.00×	52KB	6.2MB
3-D GA	`Gaigen 1`	3.94×	2.39×	64KB	6.7MB
3-D GA OM	`Gaigen 1`	2.56×	1.86×	64KB	6.7MB
4-D LA	hand-written	1.05×	1.22×	56KB	6.4MB
4-D GA	`Gaigen 1`	2.97×	2.62×	72KB	7.7MB
5-D GA	`Gaigen 1`	5.71×	4.58×	100KB	9.9MB
3-D GA	`CLU`	129×	72.0×	164KB	12.6MB
4-D GA	`CLU`	164×	97.1×	176KB	14.7MB
5-D GA	`CLU`	482×	178×	188KB	19.0MB

Table 7.1: First generation ray tracer performance benchmarks. (*impl* is short for implementation; *exec. size* is short for the size of the executable; *mem. usage* is short for run-time memory usage).

performing the benchmark with those intersections precomputed, we got a second set of benchmarks 'for free'. Those results are shown in the *rendering time w.o. BSP* column.

- We also measured memory usage and the size of the executable.
- The benchmarks were run on a Pentium III 700 MHz notebook, with 256 MB memory, running Windows 2000[1]. Programs were compiled using Visual C++ 6.0. All support libraries, such as fltk, libpng and libz were linked dynamically to get the executable size as small as possible. Run time memory usage was measured using the task manager.

7.1.2 Second Generation Ray Tracer

To benchmark the performance of `Gaigen 2`, we cleaned up the code of the first generation ray tracers, and ported them to `Gaigen 2`. We dropped support for `CLU` because we already knew its (lack of) performance, and we did not expect this performance to change much in the second generation ray tracer.

The resulting performance benchmarks are listed in Table 7.2. The benchmarks are for full rendering only. The benchmarks were run on a 3GHz Pentium 4 notebook with 1GB memory, running Windows XP. Programs were compiled using the Intel C++ Compiler 9.0 with SSE SIMD optimizations turned on. At the bottom of the list is the `Gaigen 1` benchmark, which may be used to compare the two generations. The (worse) figure for `Gaigen 1` (7.22×) is likely caused by multiple factors, including the scene, the compiler, and better optimizations done

[1]These benchmarks were performed in 2002.

model	implementation	rendering time relative to 3-D LA
3-D LA	hand-written	1.00×
4-D LA	hand-written	1.22×
3-D GA OM	`Gaigen 2`	0.98×
3-D GA	`Gaigen 2`	1.05×
4-D GA	`Gaigen 2`	1.2×
5-D GA	`Gaigen 2`	1.26×
5-D GA nice	`Gaigen 2`	1.59×
5-D GA	`Gaigen 1`	7.22×

Table 7.2: Second generation ray tracer performance benchmarks.

for the baseline (*3-D LA*) code. This goes to show that we cannot give absolute figures of the performance of geometric algebra, only a likely 'range'.

The *5-D GA nice* version

Table 7.2 has an entry for the *5-D GA nice* version of the ray tracer. This is a variant of the regular conformal (*5-D GA*) version in which we used all the semantically nice representations, instead of focusing on efficiency. The most significant differences with the regular *5-D GA* version are:

- Rays are represented by tangent vectors, as opposed to a flat-point-free-vector combination.
- Points (and their local surface attitudes) are represented by tangent bivectors, as opposed to a point-free-vector combination.

So the *5-D GA nice* benchmark shows the impact of using what is most elegant without taking efficiency into consideration.

Compilation Time Benchmarks

Because the code size of the (conformal) geometric algebra implementations generated by `Gaigen 2` can grow quite large, we were curious about the compilation times of the various implementations. Long compilation times reduce the productivity of programmers because they cannot get feedback on changes swiftly. Table 7.3 shows the results. The figures shows the compilation times for the full ray tracer, as measured on the same 3GHz machine that was used for the run-time benchmarks above.

As is evident from the data, the size of the implementation does have some influence, but this influence is limited. The simplest model *3-D LA* compiles in 37 seconds. The model with the largest code-size *5-D GA nice* compiles in 50

model	impl.	comp.	header size	source size	total size
3-D LA	hand-written	37s	20KB	4KB	24KB
4-D LA	hand-written	38s	18KB	2KB	20KB
3-D GA (OM)	`Gaigen 2`	40s	134KB	63KB	197KB
3-D GA	`Gaigen 2`	40s	135KB	60KB	195KB
4-D GA	`Gaigen 2`	45s	235KB	126KB	361KB
5-D GA	`Gaigen 2`	101s	460KB	358KB	818KB
5-D GA (nice)	`Gaigen 2`	50s	624KB	385KB	1009KB
5-D GA	`Gaigen 1`	75s	39KB	154KB	193KB

Table 7.3: Source code size and compilation times for second generation ray tracer. The source code size is the size of the geometry implementation `.h` and `.cpp` files, respectively. (*impl.* is short for implementation; *comp.* is short for compilation time).

seconds. The two outliers are the *5-D GA* models implemented through `Gaigen 2` and `Gaigen 1`. They take much longer to compile because of the interactive user interface that these two versions of the ray tracer have built-in, which the others do not. However, we predict that these *5-D GA* would compile in 45 to 50 seconds if they did not have the UI.

7.2 Inverse Kinematics

Using `Gaigen 2`, we optimized the implementation of an inverse kinematics algorithm described in [28]. The algorithm uses conformal geometric algebra to compute joint rotations of a human-like arm such that the hand ends up a required position and orientation relative to the shoulder. The algorithm was designed using `CLUCalc` [36] and then implemented using `Gaigen 1` by the computer graphics group of the TU Darmstadt. Because performance was unsatisfactory at that point, we were asked to port it to `Gaigen 2`.

After the port to `Gaigen 2`, the algorithm was about 40% *faster* than its open source linear-algebra-based counterpart [43]. One of the reason that our implementation performed better is that the the conformal model excels at constructing transformations.

[28] reports that when the entire context of the algorithm is considered, our implementation is even about 3.4 times faster: the inverse kinematics algorithm is part of a larger application which uses rotors (or quaternions) to represent rotation. Hence for the linear-algebra-based algorithm, rotor-matrix-rotor conversions are required, and this lowers performance. However, we find the '3.4 times faster' claim somewhat exaggerated, since we believe that these conversions were not implemented very efficiently.

7.3 Minor Benchmarks

7.3.1 Outermorphism Matrix Representation

For our book [12] we performed a simple benchmark to find out the efficiency increase due to using the matrix representation for an outermorphism. In this benchmark we performed 10,000,000 projections using either the matrix representation or regular geometric algebra. Using the matrix representation, this took about 0.13 seconds, using regular geometric algebra this took about 0.26 seconds (on the 3GHz notebook detailed above). In general we expect that applying transformations using the outermorphism matrix representation is more than ≥ 1.5 times faster than the regular geometric algebra implementation, but this of course depends on the problem at hand.

Again, using the outermorphism matrix representation only makes sense when the same transformation has to be applied many times. Otherwise the cost for initializing the matrices exceeds the performance gain of applying them.

7.3.2 Choice of Basis

To show the importance of picking the appropriate basis vectors we present one more (synthetic) benchmark.

In the conformal model, the vectors o and ∞ are very important, as they represent the origin and infinity. However, o and ∞ are not orthogonal (i.e., $o \cdot \infty = -1$). Hence, using o and ∞ as basis vectors complicates the computation of products of basis blades (Section 3.1.6).

Thus, most GA implementations use a different basis: two orthogonal vectors e and $\bar{e}$ are used, as described in Section 2.11.1. In this metric, o and ∞ are then constructed as a weighted sum of e and $\bar{e}$.

The problem with using e and $\bar{e}$ instead of o and ∞ is that the structure of the conformal model makes it more efficient to use the $\{o, \infty\}$-basis, at least when doing Euclidean geometry. For example, the flat objects (lines and planes) contain ∞ as a factor. This means that they have fewer coordinates on the $\{o, \infty\}$-basis than on the $\{e, \bar{e}\}$-basis.

We developed a small synthetic benchmark [18] that demonstrates the effect of the choice of basis. Our hypothesis was that translating lines would be affected by the choice of basis vectors because both the lines and the 'translators' have ∞ as a factor. Rotating circles should not be affected, as neither rotors nor circles have ∞ as a factor. The example was designed specifically for this purpose, so the effect is exaggerated:

	$\{o, \infty\}$-basis	$\{e, \bar{e}\}$-basis
translating lines	0.26s	0.81s
rotating circles	0.45s	0.46s

The times are for repeating the operation 3,000,000 times on the 3 GHz notebook. The figures show that translating lines is more than 3× more efficient on the $\{o, \infty\}$-basis than on the $\{e, \bar{e}\}$-basis. As predicted, the choice of basis vectors does not affect the performance of rotating circles.

7.3.3 Constant Coordinates

We also performed a small synthetic benchmark to prove the point of using constant coordinates in `Gaigen 2`. In many applications, the coordinates of certain 'normalized' blades and versors are constant. For example, the o coordinate of a normalized conformal point is always 1, because that is the definition of normalization in this case. The same goes for the scalar coordinate of a normalized conformal translation versor.

The benchmark we used consists of performing 100,000,000 translations of conformal points using a translation versors. In one setup we used the regular point type, which has a non-constant o coordinate. We also use a regular conformal translation versor. In the other setup we use normalized points and translation versors, with a constant o-coordinate and scalar coordinate, respectively.

The results were that the latter version was 1.4 times faster than the former (constant coordinates version: 3.34 seconds, regular version: 4.79 seconds).

7.4 Comparison of Various Implementations

In this section we compare the benchmarks of various additive implementations that we have collected. See Section 4.1 for some more information on these implementations.

7.4.1 `Gaigen 1` Compared to `Gaigen 2`

On average, we found that `Gaigen 2` is about 3 times faster than `Gaigen 1`, as measured in a full application The speedup is more pronounced for the conformal model than for simple 3-D algebras. `Gaigen 2` is faster than `Gaigen 1` mainly because it uses type-based compression (`Gaigen 1` uses per-grade compression), and because it optimizes functions for specialized multivectors.

Note that in synthetic benchmarks the performance difference between `Gaigen 1` and `Gaigen 2` is usually larger. The reason for this is that a full application does more than just geometry.

7.4.2 `Gaigen 1` and `Gaigen 2` Compared to `CLU`

The first generation ray tracer benchmark shows that `Gaigen 1` is about 30 to 80 times faster than `CLU`. Because `Gaigen 2` is on average more than twice as fast as `Gaigen 1`, these figures more than double for the second generation ray tracer.

dimension	`Gaigen 1` (seconds)	`MV` (seconds)
4-D	0.32	2.7
5-D	0.48	6.4
6-D	0.76	19.4
7-D	1.27	52.7
8-D	2.70	174

Table 7.4: Benchmarks of `Gaigen 1` and `MV`. The figures are executions times for performing 1,000,000 linear products on random blades, in 4-D to 8-D space. Benchmarks performed on a 1.83 GHz Core2 Duo notebook.

7.4.3 `Gaigen 1` and `Gaigen 2` Compared to `Matlab GABLE`

We were requested to implemented an algorithm to detect singularities in vector fields, as described in [34]. The authors of that paper had used Matlab GA-Package `GABLE` [14] for a prototype implementation, which was basically unusable because of its low speed. (The inefficiency of `GABLE` is due to its total lack of coordinate compression, and the heavy use of *Matlab objects* which are known to be slow).

Our `Gaigen 1` version was $6,000$ times faster than the `GABLE` version. Later on, we ported our `Gaigen 1` version to `Gaigen 2`, for use as an example in our book [12]. This resulted in a performance increase of about $2.5\times$. This performance difference between `Gaigen 1` and `Gaigen 2` is consistent with the ray tracer benchmarks. Also, by multiplying these figures ($2.5 \times 6,000$), we can extrapolate that `GABLE` is about $15,000\times$ slower than `Gaigen 2`.

7.4.4 `Gaigen 1` Compared to `MV`

We performed a synthetic benchmark to compare `Gaigen 1` to `MV` [3]. The primary reason for performing this benchmark was to obtain the graph of Figure 5.7; the comparison between `Gaigen 1` and `MV` was secondary.

The benchmark consisted of computing various linear products on random blades. The results are in Table 7.4. We found that for low dimensional algebras ($N \leq 5$), the performance of `Gaigen 1` is about 10 times that of `MV`. As the dimension increases, `Gaigen 1` becomes more and more efficient relative to `MV`, up to 60 times faster for $N = 8$. (`Gaigen 1` can not go beyond 8-D, while `MV` scales up to 64-D).

We note that `MV`'s per-coordinate compression scheme is at a disadvantage in this benchmark. We used random blades as input. These random blades are generated as the outer product of random vectors. In general, random blades do not have any zero coordinates for their respective grade. This makes `MV`'s per-coordinate compression less usefull. The benchmark in the next section is fairer with respect `MV`'s coordinate compression method.

7.4.5 `Gaigen 2` Compared to `MV`

We also performed a synthetic benchmark that compared `Gaigen 2` to `MV` in a somewhat more realistic setting than the 'random blades' approach of the previous section. The benchmark consisted of computing linear products of combinations of objects from the conformal model of 3-D Euclidean geometry (so a 5-D algebra was used). The objects which were used are: scalars, flat points, lines, planes, points, circles, spheres, tangent vectors, tangent bivectors, tangent trivectors, free vectors, free bivectors and free trivectors (see Section 2.11.3 for a classification of conformal model blades). The objects were constructed by randomly translating and rotating canonical versions of the same objects. For example, a random line was constructed by randomly translating and rotating the line through the origin in the direction of $\mathbf{e}_1$, i.e., $o \wedge \mathbf{e}_1 \wedge \infty$.

Using these conformal model objects (as opposed to fully random blades) allows `MV` to show off its per-coordinate compression, as many of these objects contain basis vectors as factors. Indeed, the performance of `MV` increases from 6.4 seconds to 4.7 seconds, for computing 1,000,000 linear products. However, this is still 10 times slower than `Gaigen 1` (see the 5-D entry of Table 7.4).

The same benchmark implemented for `Gaigen 2` required 0.08 seconds to compute 1,000,000 operations. This makes `Gaigen 2` around 60 times faster than `MV`.

7.4.6 `Gaigen 2` Compared to `Clifford`

`Clifford` is similar to `Gaigen 2` in that it supports specialized multivectors, so we expected similar performance for basic geometric algebra usage. `Clifford` uses a different generative programming method than `Gaigen 2` to support specialization (`C++` meta-programming, see Section 4.4). `Clifford` is not publically available, so we requested the author of `Clifford` to perform a benchmark for us [42]. The outcome of this benchmark was that `Gaigen 2` and `Clifford` perform equally, as measured in the *3-D GA* model.

However, `Clifford` does not support some of the features that `Gaigen 2` has, such as the use of non-orthogonal bases and outermorphism matrix representations. Hence we expect that when used for the conformal model of Euclidean geometry, `Clifford` will be somewhat slower than `Gaigen 2` (see Section 7.3.2 for the impact on performance).

7.5 Discussion and Conclusion

The ray tracer benchmarks show that geometric algebra can be similar in efficiency to linear algebra. However, it is important to use the right implementation technique, otherwise performance can easily be two order of magnitude worse (i.e., `Gaigen 2` compared to `CLU`).

We have also shown that the compile time of programs using our `Gaigen 2` is not prohibitively longer than programs using classical geometry implementations.

The most complicated conformal model version of the ray tracer took 35% longer to compile than the basic *3-D LA* version.

The inverse kinematics application offers the prospect of applications that can actually run significantly faster through the use of (conformal) geometric algebra. The core inverse kinematics algorithm itself is faster, and the added benefit is that fewer rotor-to-matrix conversions are required to fit the solution into its context.

The difference in performance between these two applications (ray tracer, inverse kinematics) perhaps exposes fundamental strengths and weaknesses of (conformal) geometric algebra: the general performance trend may be that geometric algebra excels at constructing and processing transformations (but not at applying them), and lags somewhat (or at most plays even) in other areas. Of course, more detailed benchmarks and comparisons are required to confirm this trend. One may conclude that the ray tracer was a bad example to show the performance of geometric algebra, but we selected this application it on purpose because it contains a good mix of geometric operations.

Through a series of synthetic benchmarks we have shown the benefit of several distinctive features of `Gaigen 2` in isolation. These features are outermorphism matrix representations, the use of a non-orthogonal basis, and constant coordinates. In real-world applications each of these features may cause a speed-up of up to 10%. Having them is essential to close the gap between geometric algebra and hand-optimized linear algebra.

Another series of benchmarks shows the performance of `Gaigen 2` relative to other geometric algebra implementations. The only real competition to `Gaigen 2` comes from `Clifford`. `Clifford` is based on the same principles as `Gaigen 2`, but implemented through a different generative programming technique (`C++` meta programming). However, `Clifford` lacks the aforementioned distinctive features of `Gaigen 2` (it cannot fully take advantage of the structure of the conformal model), which would make it somewhat slower in most applications. We also expect compile times for `Clifford` to be longer (because the compiler has to handle all the template-rewriting), but we have not verified this.

Chapter 8

Discussion and Conclusion

This chapter concludes the thesis. It summarizes findings, but also discusses topics not covered by the main text, such as open problems in the conformal model.

8.1 Suitability of Geometric Algebra for Implementing Geometry on a Computer

The main point of this thesis is to show that geometric algebra is a suitable alternative for implementing geometry on a computer. The theoretical advantages should be obvious: equations become more generic, more primitives and transformations can be represented directly, and so on. This is visible in the tables of Appendix B. What we have shown is that — when properly implemented — the performance of geometric algebra is comparable to that of classical methods.

This is true even for the conformal model, despite its high dimensionality (e.g., 32-D for 3-D geometry). In two different applications we got an approximate speeds of 0.7 times and 1.25 times that of the classical implementation. In general we expect that the relative performance of a well-implemented conformal model application to be within one half and twice the speed of its classical equivalent.

8.2 `Gaigen 2`

Through `Gaigen 2` we have shown that geometric algebra can perform on a par with classical geometry implementations. In our main benchmarking platform (the ray tracer), performance of geometric algebra in the 3-D and 4-D models is approximately equal to that of the linear algebra equivalent. In these models, geometric algebra even has a slight edge due to minor optimizations that are easy to implement using `Gaigen 2`. The 5-D conformal model version of the ray tracer is somewhat slower, but not by much.

In another conformal model application (inverse kinematics [28]) we actually measured a performance *increase* of 40% compared to the classical implementation. This application deals mostly with constructing transformations, which is an area where the conformal model excels.

We have also shown that geometric algebra implementations can be automatically generated from their specifications. This is made relatively easy by the structured nature of geometric algebra, which allows high-level equations to be 'expanded' to the level of coordinates automatically. Using generative programming not only saves the programmer from writing low-level coordinate-code by hand, but also allows for optimizations that would be tedious to do manually.

We believe that the generative programming ideas behind `Gaigen 2` can also be applied to other algebras. We are especially interested in automatic generation of implementations of linear algebra algorithms for *low*-dimensional spaces. LAPACK implementations are typically designed for high-dimensional spaces, but

lack performance for small problems. We have found that hand-optimized implementations for solving a 3×3 system with partial pivoting and computing the SVD are ten times faster than the `LAPACK`[1] equivalent. We expect similar speedups for more complex algorithms. Examples of the algorithms we would like to have are singular value decomposition and eigenvalue decomposition, for 3×3 to 5×5 matrices. Techniques similar to those we used for `Gaigen 2` could be applied to automate the code generation of such algorithms.

Besides its overall performance, there are several nice properties of `Gaigen 2` that we repeat here:

- Explicit typing of multivector variables (Section 4.6.4). To get optimal performance out of `Gaigen 2`, specialized multivector variables should be used. Explicit typing may seem a drawback compared to the one-size-fits-all general multivector type, but in practice it makes code more readable. I.e., the multivector type of each variable can be easily read from the code. The universal applicability of equations is retained due to the way `Gaigen 2` is set up.

- Constant-coordinate optimizations (Sections 4.6.4 and 4.6.6). These are easy to add to an existing application. One only has to add the required specialized multivector type and regenerate the algebra. Constant-coordinates increase performance and save memory by up to 40%.

- Matrix representation of outermorphisms (Section 4.8.5). These are essential for high performance implementation of transformations as performance can easily double. A downside is that in most cases one has to keep track of both the matrix representation and the geometric algebra elements that generate the transform. The former is used to apply the transform, while the latter is used to manipulate it.

- Choice of basis (Section 4.6.2). The choice of basis matters in the conformal model, and `Gaigen 2` allows one to pick any basis, not just an orthogonal one. This makes conformal model implementations more efficient.

`Gaigen 2` also has some problems:

- The design of the `Gaigen 2` code generator is not modular enough. This makes it harder than it should be to add new features or make changes.

- The generated code for general multivectors in high-dimensional spaces ($n > 6$) can grow large (more than 100.000 lines). This makes `Gaigen 2` practically unusable for such spaces because many compilers fail on such large files. It would be better to have a code generation mode that emits code that processes such multivectors directly using the bitmap representation instead of explicitly expanding all the code to the level of coordinate manipulation.

[1]We used the optimized `LAPACK` implementation from Intels Math Kernel Library

- Using only per-grade compression for the general multivector implementation is too limiting. Per-grade compression can be inefficient in high-dimensional spaces, so per-coordinate or per-group compression should also be available as an option.

- The amount of 'glue code' in the generated implementations is rather large. By glue code we means code that is only required to get compilation working and does not actually compute anything. For example, the code of the conformal model implementation for the ray tracer is 50% actual code and 50% glue code. (While the glue code is optimized away by the compiler, it slows down the compilation process somewhat).

- Profiling the final application is an awkward activity. In practice we tend to postpone profiling as long as possible.

The first three problems could be fixed in a next version. The last two problems are fundamental to the generative programming method we have used.

8.3 Multiplicative Representation

We have shown that implementing geometric algebra using a multiplicative representation of blades and versors is feasible. The expected time complexity of $O(n^3)$ instead of $O(2^n)$ was matched by our benchmarks. This is a great improvement over the $O(2^n)$ or worse time complexity of the additive representation.

The main problem with the multiplicative representation is the versor simplification problem in metrics $\mathbb{R}^{m-k,k}$ or $\mathbb{R}^{k,m-k}$ with $(m-2) \geq k \geq 2$. The versor simplification algorithm cannot work on some versors in these metrics. We do not know how severe this is. It may be that such versors are not used in practice, but since we have no applications that use such metrics, we currently cannot tell.

Another open problem is computing the exponential of arbitrary bivectors. We hope to solve this by decomposing such bivectors into smaller parts, each of which individually is a simple 2-blade with a scalar square.

8.4 Conformal model

We find the conformal model of great value for implementing geometry. One of the reasons we like it so much is the ease with which one can perform 'visual debugging' of geometry, by going back and forth between the application and an interactive visualization program like `GAViewer`. We owe this to the inherent property of the geometric algebra that each of the elements of computation has a geometrical interpretation. The large number of primitives and transformations that can be directly represented is also a great asset. However, we have also some criticisms. Some of these points may be directions for future research.

- To get the highest possible performance out of the conformal ray tracer, we had to do 'optimizations' by picking the best (in terms of performance) representation for each geometric concept. These representations were not always the best semantically. For example, a tangent vector is the best representation for ray, semantically speaking. But splitting a tangent vector into a flat point and a free vector leads to a more efficient implementation.

  To measure the impact of using the semantically best representations anyway, we implemented the *5-D GA nice* version of the ray tracer (see Section 7.1.2). This version turned out 1.59× slower than the *3-D LA* version, or 1.26× slower than the *5-D GA* conformal version. So we may conclude that even using the semantically best choices can still result in acceptable performance.

  One would of course prefer to work directly with the semantically best representations, and have a geometry compiler perform the optimizations automatically. However, to perform such optimizations the compiler would have to do a full-program-analysis. I.e., the compiler should have insight in how each variable is used, and understand how it can apply optimizations. We do not believe that this is attainable presently.

- The numerical stability of conformal geometric algebra should be studied rigorously. The null vectors that represent points lie on the horosphere (a n-D parabola). This means that one of the coordinates of such vectors is proportional to *square* of the respective distance of the point to the origin. This is true regardless of the implementation method (additive or multiplicative). Hence this coordinate is large when the point is far away from the origin. Due to limited precision of floating point numbers and the resulting rounding errors, such points will not lie exactly on the horosphere. This causes them to be interpreted as small spheres. This problem propagates to other operations. For example, we have had problems when repeatedly applying scaling versors to points even though they were not far from the origin; the errors accumulated. In our work so far, such situations have occurred several times. We usually ignored such problems, or fixed them through quick hacks. However, a thorough study is required to determine what the numerical limitations of the conformal model are.

- A US patent [26] on the conformal model has been granted. We believe this may hinder adoption of conformal geometric algebra. The alternative — in the form of homogeneous coordinates or the homogeneous model — is far better known, more widely available, used in a lot of software, and has hardware optimized for it. The conformal model is already fighting an up-hill battle, and the prospect of having to pay license fees for commercial applications is not going to make this fight any easier.

Appendix A: Subspace Products from the Geometric Product

The non-interested reader may skip the rest of this section, and just assume that the grade part selection rules are correct. No techniques which are essential for the rest of the thesis are discussed here. The proofs and most of the text in this section come directly from Appendix C of [12].

A.1 Outer Product from Geometric Product (for Vectors)

To prove that the grade selection rules are correct in general, we first need a weaker result, namely that they are correct when one of the arguments is a vector.

For vectors $\mathbf{a}$, we should prove that:

$$\mathbf{a} \wedge \mathbf{B} = \tfrac{1}{2}(\mathbf{a}\,\mathbf{B} + \widehat{\mathbf{B}}\,\mathbf{a}), \tag{A.1}$$

$$\mathbf{B} \wedge \mathbf{a} = \tfrac{1}{2}(\mathbf{B}\,\mathbf{a} + \mathbf{a}\,\widehat{\mathbf{B}}), \tag{A.2}$$

We check the earlier defining properties of Section 2.4.1.

- If $\mathbf{B}$ is a scalar β, we get $\mathbf{a} \wedge \beta = \beta\,\mathbf{a} = \beta \wedge \mathbf{a}$.
- If $\mathbf{B}$ is a vector, we obtain anti-symmetry, as we saw above.
- To demonstrate associativity, let us develop $(\mathbf{a} \wedge \mathbf{B}) \wedge \mathbf{c}$ and $\mathbf{a} \wedge (\mathbf{B} \wedge \mathbf{c})$.

  $$(\mathbf{a} \wedge \mathbf{B}) \wedge \mathbf{c} = \tfrac{1}{4}\,(\mathbf{a}\,\mathbf{B}\,\mathbf{c} + \mathbf{a}\,\mathbf{c}\,\widehat{\mathbf{B}} + \widehat{\mathbf{B}}\,\widehat{\mathbf{c}}\,\mathbf{a} + \widehat{\mathbf{c}}\,\mathbf{B}\,\mathbf{a})$$

  $$\mathbf{a} \wedge (\mathbf{B} \wedge \mathbf{c}) = \tfrac{1}{4}\,(\mathbf{a}\,\mathbf{B}\,\mathbf{c} + \widehat{\mathbf{B}}\,\mathbf{a}\,\mathbf{c} + \mathbf{c}\,\widehat{\mathbf{a}}\,\widehat{\mathbf{B}} + \mathbf{c}\,\mathbf{B}\,\widehat{\mathbf{a}})$$

  The two are equivalent if $\mathbf{a}\,\mathbf{c}\,\widehat{\mathbf{B}} - \mathbf{c}\,\widehat{\mathbf{a}}\,\widehat{\mathbf{B}} = \widehat{\mathbf{B}}\,\mathbf{a}\,\mathbf{c} - \widehat{\mathbf{B}}\,\widehat{\mathbf{c}}\,\mathbf{a}$. But this can be simplified using Equation 2.15 to $(\mathbf{a}\cdot\mathbf{c})\,\widehat{\mathbf{B}} = \widehat{\mathbf{B}}\,(\mathbf{a}\cdot\mathbf{c})$. That holds by the commutativity of scalars under the geometric product, so the two expressions are indeed equivalent. In particular, this demonstrates the associativity law for vectors

  $$(\mathbf{a} \wedge \mathbf{b}) \wedge \mathbf{c} = \mathbf{a} \wedge (\mathbf{b} \wedge \mathbf{c}).$$

  By applying it recursively we get general associativity for blades.

- Linearity and distributivity over addition trivially hold by the corresponding properties of the geometric product. They permit extending the results to general multivectors B.

This demonstrates that the two equations above indeed identify the same outer product structure that we had before, at least when one of the factors is a vector. Since we have associativity, this can be extended to general blades, and by linearity to general multivectors. The case of two scalars is formally not yet included, but this will be a trivial addition in the full proof below which uses grade part selection.

A.2 Left Contraction from Geometric Product (for Vectors)

We also need the equivalent of the previous section for the left contraction:

$$\mathbf{a} \rfloor \mathbf{B} = \tfrac{1}{2}(\mathbf{a}\,\mathbf{B} - \widehat{\mathbf{B}}\,\mathbf{a}), \tag{A.3}$$

$$\mathbf{B} \lfloor \mathbf{a} = \tfrac{1}{2}(\mathbf{B}\,\mathbf{a} - \mathbf{a}\,\widehat{\mathbf{B}}). \tag{A.4}$$

We focus on the formula for the left contraction $\mathbf{a} \rfloor \mathbf{B}$, and show consistency with the earlier definition of Equation 2.2 through Equation 2.6. The right contraction $\mathbf{B} \lfloor \mathbf{a}$ is completely analogous.

- If $\mathbf{B}$ is a scalar β, we get $\mathbf{a} \rfloor \beta = 0$.
- If $\mathbf{B}$ is a vector $\mathbf{b}$, we obtain symmetry and equivalence to the classical inner product $\mathbf{a} \cdot \mathbf{b}$ of Equation 2.15.
- To demonstrate the distribution of the left contraction over an outer product, we first show

  $$\begin{aligned} \mathbf{a} \rfloor (\mathbf{b}\,\mathbf{C}) &= \tfrac{1}{2}(\mathbf{a}\,\mathbf{b}\,\mathbf{C} + \mathbf{b}\,\widehat{\mathbf{C}}\,\mathbf{a}) = \tfrac{1}{2}(\mathbf{a}\,\mathbf{b}\,\mathbf{C} + \mathbf{b}\,\mathbf{a}\,\mathbf{C} - \mathbf{b}\,\mathbf{a}\,\mathbf{C} + \mathbf{b}\,\widehat{\mathbf{C}}\,\mathbf{a}) \\ &= (\mathbf{a} \cdot \mathbf{b})\,\mathbf{C} - \mathbf{b}\,(\mathbf{a} \rfloor \mathbf{C}). \end{aligned} \tag{A.5}$$

  Similarly, you may derive that

  $$\mathbf{a} \rfloor (\widehat{\mathbf{C}}\,\mathbf{b}) = (\mathbf{a} \rfloor \widehat{\mathbf{C}})\,\mathbf{b} + \mathbf{C}\,(\mathbf{a} \cdot \mathbf{b}). \tag{A.6}$$

  These equations are important by themselves, for they specify the distributivity of the contraction over the geometric product. Adding them produces $\mathbf{a} \rfloor (\mathbf{b} \wedge \mathbf{C}) = (\mathbf{a} \cdot \mathbf{b}) \wedge \mathbf{C} - \mathbf{b} \wedge (\mathbf{a} \rfloor \mathbf{C})$. Applied repeatedly, this equation gives us the desired result Equation 2.5 for blades.
- If we follow the same procedure we used to demonstrate associativity of the outer product, we get a surprise. That derivation does not show the associativity of the contraction (fortunately!), but the associativity of a 'sandwich' combination of the two contractions:

  $$(\mathbf{a} \rfloor \mathbf{B}) \rfloor \mathbf{c} = \mathbf{a} \rfloor (\mathbf{B} \rfloor \mathbf{c}).$$

  For vectors this becomes the rather trivial $(\mathbf{a} \rfloor \mathbf{b}) \rfloor \mathbf{c} = \mathbf{a} \rfloor (\mathbf{b} \rfloor \mathbf{c})$ which holds because both sides are zero.
- That leaves us with no tools to prove the remaining defining equation Equation 2.6 for the earlier contraction: $\mathbf{A} \rfloor (\mathbf{B} \rfloor \mathbf{C}) = (\mathbf{A} \wedge \mathbf{B}) \rfloor \mathbf{C}$. In fact, we cannot even compute both sides simultaneously, since we always need a vector as an argument to the equation $\mathbf{a} \cdot \mathbf{B} = \frac{1}{2}(\mathbf{a}\,\mathbf{B} - \widehat{\mathbf{B}}\,\mathbf{a})$,

As with the outer product, the proposed equations do not define the contraction, and do therefore not prove it to be identical with the earlier contraction from Section 2.5.4; but we have at least shown that they are consistent with it.

A.3 The Outer Product from the Geometric Product (General)

Our new definition of the outer product of two blades $\mathbf{A}_k$ and $\mathbf{B}_l$ is:

$$\mathbf{A}_k \wedge \mathbf{B}_l = \langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{l+k}$$

We show the correctness of the equation for the outer product using Equation A.1. Write $\mathbf{A}_k$ as $\mathbf{A}_k = \mathbf{a}_k \wedge \mathbf{A}_{k-1}$. Then

$$\begin{aligned}
\langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{l+k} &= \langle (\mathbf{a}_k \wedge \mathbf{A}_{k-1})\, \mathbf{B}_l \rangle_{l+k} \\
&= \tfrac{1}{2} \langle \mathbf{a}_k \mathbf{A}_{k-1} \mathbf{B}_l + \widehat{\mathbf{A}}_{k-1} \mathbf{a}_k \mathbf{B}_l \rangle_{l+k} \\
&= \tfrac{1}{2} \langle \mathbf{a}_k \mathbf{A}_{k-1} \mathbf{B}_l + \widehat{\mathbf{A}}_{k-1} \mathbf{a}_k \mathbf{B}_l + \widehat{\mathbf{A}}_{k-1} \widehat{\mathbf{B}}_l \mathbf{a}_k - \widehat{\mathbf{A}}_{k-1} \widehat{\mathbf{B}}_l \mathbf{a}_k \rangle_{l+k} \\
&= \tfrac{1}{2} \langle (\mathbf{a}_k \mathbf{A}_{k-1} \mathbf{B}_l + \widehat{\mathbf{A}}_{k-1} \widehat{\mathbf{B}}_l \mathbf{a}_k) + (\widehat{\mathbf{A}}_{k-1} \mathbf{a}_k \mathbf{B}_l - \widehat{\mathbf{A}}_{k-1} \widehat{\mathbf{B}}_l \mathbf{a}_k) \rangle_{l+k} \\
&= \tfrac{1}{2} \langle (\mathbf{a}_k (\mathbf{A}_{k-1} \mathbf{B}_l) + (\widehat{\mathbf{A}}_{k-1} \widehat{\mathbf{B}}_l) \mathbf{a}_k) + (\widehat{\mathbf{A}}_{k-1} (\mathbf{a}_k \mathbf{B}_l) - \widehat{\mathbf{A}}_{k-1} (\widehat{\mathbf{B}}_l \mathbf{a}_k)) \rangle_{l+k} \\
&= \langle \mathbf{a}_k \wedge (\mathbf{A}_{k-1} \mathbf{B}_l) + \widehat{\mathbf{A}}_{k-1} (\mathbf{a}_k \rfloor \mathbf{B}_l) \rangle_{l+k} \\
&= \langle \mathbf{a}_k \wedge (\mathbf{A}_{k-1} \mathbf{B}_l) \rangle_{l+k} \quad \text{[other term at most of grade } l+k-2\text{]} \\
&= \mathbf{a}_k \wedge \langle \mathbf{A}_{k-1} \mathbf{B}_l \rangle_{l+k-1} \\
&\;\;\vdots \\
&= \mathbf{A}_k \wedge \langle \mathbf{B}_l \rangle_l \\
&= \mathbf{A}_k \wedge \mathbf{B}_l
\end{aligned}$$

This proof does not work when $\mathbf{A}_k$ is a scalar α of grade 0. But then $\langle \alpha \mathbf{B}_l \rangle_l = \alpha \langle \mathbf{B} \rangle_l = \alpha \wedge \mathbf{B}_l$, so the result still holds.

A.4 The Metric Products from the Geometric Product (General)

Our new definitions of the metric products are:

$$\begin{aligned}
\mathbf{A}_k \rfloor \mathbf{B}_l &= \langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{l-k} \\
\mathbf{A}_k \lfloor \mathbf{B}_l &= \langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{k-l} \\
\mathbf{A}_k * \mathbf{B}_l &= \langle \mathbf{A}_k\, \mathbf{B}_l \rangle_0
\end{aligned}$$

We now prove that these definitions are consistent with earlier definitions of these products:

- The scalar properties Equation 2.2 and 2.3 are straightforward:

$$\mathbf{A}_k \rfloor \alpha = \langle \mathbf{A}_k \alpha \rangle_{-k} = 0 \quad \text{if} \quad k \neq 0$$

$$\alpha \rfloor \mathbf{A}_k = \langle \alpha \mathbf{A}_k \rangle_k = \alpha \mathbf{A}_k$$

- The inner product correspondence property Equation 2.4 is rather obvious: $\langle \mathbf{a}\,\mathbf{b}\rangle_0 = \mathbf{a} * \mathbf{b} = \mathbf{a} \cdot \mathbf{b}$.
- The property Equation 2.5, which reads: $\mathbf{a} \rfloor (\mathbf{b} \wedge \mathbf{B}_l) = (\mathbf{a} \rfloor \mathbf{b})\mathbf{B}_l - \mathbf{b} \wedge (\mathbf{a} \rfloor \mathbf{B}_l)$ for a blade $\mathbf{B}_l$ of grade l takes some more work:

$$\begin{aligned}
&(\mathbf{a} \rfloor \mathbf{b})\mathbf{B}_l - \mathbf{b} \wedge (\mathbf{a} \rfloor \mathbf{B}_l) = \\
&\quad = \quad \langle (\mathbf{a} \cdot \mathbf{b})\mathbf{B}_l - \mathbf{b}(\mathbf{a} \rfloor \mathbf{B}_l) \rangle_l \\
&\quad = \quad \langle (\mathbf{a} \cdot \mathbf{b})\mathbf{B}_l - \mathbf{b} \wedge (\mathbf{a} \rfloor \mathbf{B}_l) - \mathbf{b} \rfloor (\mathbf{a} \rfloor \mathbf{B}_l) \rangle_l \\
&\quad = \quad \langle (\mathbf{a} \cdot \mathbf{b})\mathbf{B}_l - \mathbf{b} \wedge (\mathbf{a} \rfloor \mathbf{B}_l) \rangle_l \quad \text{[last term of too low grade]} \\
&\quad = \quad \langle \mathbf{a} \rfloor (\mathbf{b} \wedge \mathbf{B}_l) \rangle_l \\
&\quad = \quad \langle \mathbf{a}(\mathbf{b} \wedge \mathbf{B}_l) - \mathbf{a} \wedge (\mathbf{b} \wedge \mathbf{B}_l) \rangle_l \\
&\quad = \quad \langle \mathbf{a}(\mathbf{b} \wedge \mathbf{B}_l) \rangle_l \quad \text{[last term of too high grade]} \\
&\quad = \quad \mathbf{a} \rfloor (\mathbf{b} \wedge \mathbf{B}_l)
\end{aligned}$$

- We can now prove the final contraction property Equation 2.6.

$$\begin{aligned}
&(\mathbf{A}_k \wedge \mathbf{a}) \rfloor \mathbf{B}_l = \\
&\quad = \quad \langle (\mathbf{a} \wedge \widehat{\mathbf{A}}_k)\mathbf{B}_l \rangle_{l-k-1} \\
&\quad = \quad \tfrac{1}{2} \langle \mathbf{a}\widehat{\mathbf{A}}_k\mathbf{B}_l + \mathbf{A}_k\mathbf{a}\mathbf{B}_l \rangle_{l-k-1} \\
&\quad = \quad \tfrac{1}{2} \langle \mathbf{A}_k\mathbf{a}\mathbf{B}_l - \mathbf{A}_k\widehat{\mathbf{B}}_l\mathbf{a} + \mathbf{A}_k\widehat{\mathbf{B}}_l\mathbf{a} + \mathbf{a}\widehat{\mathbf{A}}_k\mathbf{B}_l \rangle_{l-k-1} \\
&\quad = \quad \langle \mathbf{A}_k(\mathbf{a} \rfloor \mathbf{B}_l) + \mathbf{a} \wedge (\widehat{\mathbf{A}}_k\mathbf{B}_l) \rangle_{l-k-1} \\
&\quad = \quad \langle \mathbf{A}_k(\mathbf{a} \rfloor \mathbf{B}_l) \rangle_{l-k-1} \quad \text{(lowest grade last term } |l-k|+1\text{, too high)} \\
&\quad = \quad \mathbf{A}_k \rfloor (\mathbf{a} \rfloor \mathbf{B}_l),
\end{aligned}$$

  and recursion over the factors of $\mathbf{A}_k$ then proves the general result $(\mathbf{A} \wedge \mathbf{A}') \rfloor \mathbf{B} = \mathbf{A} \rfloor (\mathbf{A}' \rfloor \mathbf{B})$.

So we see that the symbol $\langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{l-k}$ which we proposed as the grade-based alternative definition of the left contraction inner product has all the desired properties. Therefore it is algebraically identical to the contraction of Section 2.5.4.

For the right contraction we could repeat this proof, or just show that its correspondence equation $\mathbf{B} \lfloor \mathbf{A} = (\widetilde{\mathbf{A}} \cdot \widetilde{\mathbf{B}})^{\sim} = (-1)^{a(b+1)}\mathbf{A} \cdot \mathbf{B}$ (with $a = \text{grade}(\mathbf{A})$ and $b = \text{grade}(\mathbf{B})$) with the left contraction holds:

$$\mathbf{A}_k \lfloor \mathbf{B}_l = \langle \mathbf{A}_k\, \mathbf{B}_l \rangle_{k-l} = \langle (\widetilde{\mathbf{B}}_l\, \widetilde{\mathbf{A}}_k)^{\sim} \rangle_{k-l} = \langle \widetilde{\mathbf{B}}_l\, \widetilde{\mathbf{A}}_k \rangle_{k-l}^{\ \sim} = (\mathbf{B}_l \lfloor \mathbf{A}_k)^{\sim}.$$

Since that is universal for all blades (and can be extended for multivectors), the right contraction defined by Equation 2.21 must be identical to our earlier right contraction.

The scalar product of blades of equal grade should be consistent with how it can be defined as a special case of the contraction: $\mathbf{A}_k * \mathbf{B}_k = \mathbf{A}_k \rfloor \mathbf{B}_k = \langle \mathbf{A}_k\, \mathbf{B}_k \rangle_0$.

For non-equal grades, the scalar part of $\mathbf{A}_k \, \mathbf{B}_l$ is zero (as it should be), since the lowest grade present always exceeds zero when $k \neq l$. Therefore Equation 2.22 for the scalar product is correct.

Appendix B: Ray Tracer Tables

This appendix contains two tables. The first table lists five essential ray tracing primitives and how they are represented in each of the five models we listed in at the start of Chapter 6. The second table lists five essential ray tracing operations and how they are implemented in each of the five models. The conformal model tables correspond to the description in [19] (1st generation of our ray tracer) and thus do not fully correspond to Chapter 6, which describes the 2nd generation of our ray tracer.

Representation of 5 important ray tracing primitives in 5 models of 3D Euclidean Geometry

	3D LA	3D GA	4D LA	4D GA	5D GA
point	$\mathbf{q}$: vector from the origin to the point	$\mathbf{q}$: vector from the origin to the point	$\vec{q}$: vector from origin to point $\mathbf{q}$ $\mathbf{q} = (\vec{q} : 1)$	$\vec{q}$: vector from origin $\mathbf{e}_0$ to point $\mathbf{q}$ $\mathbf{q} = \vec{q} + \mathbf{e}_0$	$\vec{q}$: vector from origin $\mathbf{e}_0$ to point $\mathbf{q}$ $\mathbf{q} = \vec{q} + \mathbf{e}_0 - \frac{1}{2}(\vec{q} \cdot \vec{q})\,\mathbf{e}_\infty$
line	$\mathbf{q}$: vector from origin to a point on the line $\mathbf{u}$: direction of the line	$\mathbf{q}$: vector from origin to a point on the line $\mathbf{u}$: direction of the line	$\mathbf{q}_1 = (\vec{q}_1 : 1)$, $\mathbf{q}_2 = (\vec{q}_2 : 1)$: two points $\mathbf{l} = (\vec{q}_1 - \vec{q}_2 : \vec{q}_1 \times \vec{q}_2)$	$\mathbf{q}_1, \mathbf{q}_2$: two points $\mathbf{l} = \mathbf{q}_1 \wedge \mathbf{q}_2$	$\mathbf{q}_1, \mathbf{q}_2$: two points $\mathbf{l} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{e}_\infty$
plane	$\mathbf{n}$: normal vector of the plane δ: distance of plane to the origin	$\mathbf{p}$: bivector of the plane δ: distance of plane to the origin	$\vec{n}$: normal vector of the plane δ: distance of plane to the origin $\mathbf{p} =$ [illegible]	$\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$: three points $\mathbf{p} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3$	$\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$: three points $\mathbf{p} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3 \wedge \mathbf{e}_\infty$
sphere	$\mathbf{q}$: vector from the origin to the center of the sphere ρ: radius of the sphere	$\mathbf{q}$: vector from the origin to the center of the sphere ρ: radius of the sphere	$\mathbf{q}$: sphere center point ρ: sphere radius	$\mathbf{q}$: sphere center point ρ: sphere radius	$\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3, \mathbf{q}_4$: four points $\mathbf{s} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3 \wedge \mathbf{q}_4$ or: $\mathbf{q}$: sphere center point ρ: sphere radius $\mathbf{s} = \left(\mathbf{q} + \frac{1}{2}\rho^2\mathbf{e}_\infty\right)^*$
rotation / translation	$\mathbf{R}$: rotation matrix $\mathbf{t}$: translation vector $\mathbf{R}_1, \mathbf{t}_1$: transformation 1 $\mathbf{R}_2, \mathbf{t}_2$: transformation 2 $\mathbf{R}_c = \mathbf{R}_2\mathbf{R}_1$ $\mathbf{t}_c = \mathbf{t}_2 + \mathbf{R}_2\mathbf{t}_1$	$\mathbf{b}$: rotation plane ϕ: rotation angle $\mathbf{R} = e^{-\frac{\phi}{2}\mathbf{b}}$ $\mathbf{t}$: translation vector $\mathbf{R}_1, \mathbf{t}_1$: transformation 1 $\mathbf{R}_2, \mathbf{t}_2$: transformation 2 $\mathbf{R}_c = \mathbf{R}_2\mathbf{R}_1$ $\mathbf{t}_c = \mathbf{t}_2 + \mathbf{R}_2\mathbf{t}_1\mathbf{R}_2^{-1}$	$\mathbf{R}$: 3×3 rotation matrix $\vec{t}$: translation vector $\mathbf{M} = \begin{bmatrix} \mathbf{R} & \vec{t} \\ \vec{0} & 1 \end{bmatrix}$ $\mathbf{M}_1, \mathbf{M}_2$: two transformations $\mathbf{M}_c = \mathbf{M}_2\mathbf{M}_1$	f: transformation function satisfying $f(\mathbf{a} \wedge \mathbf{b}) = f(\mathbf{a}) \wedge f(\mathbf{b})$ $\mathbf{M}_f$: outermorphism operator constructed from f $\mathbf{M}_{f1}, \mathbf{M}_{f2}$: two outermorphism operators $\mathbf{M}_c = \mathbf{M}_{f2}\mathbf{M}_{f1}$	$\mathbf{b}$: rotation plane ϕ: rotation angle $\mathbf{R} = e^{-\frac{\phi}{2}\mathbf{b}}$ $\vec{t}$: translation vector $\mathbf{T} = \mathbf{1} + \frac{1}{2}\vec{t} \wedge \mathbf{e}_\infty = e^{\frac{1}{2}\mathbf{t} \wedge \mathbf{e}_\infty}$ $\mathbf{V}_1, \mathbf{V}_2$: two transformations $\mathbf{V}_c = \mathbf{V}_2\mathbf{V}_1$

Table B.1: Ray tracer primitives.

Implementation of 5 important ray tracing operations in 5 models of 3D Euclidean Geometry

	3D LA	3D GA	4D LA	4D GA	5D GA
rotation / translation	$\mathbf{R}, \mathbf{t}$: rotation matrix, translation vector point: $\mathbf{q}$: point $\mathbf{q}' = \mathbf{R}\mathbf{q} + \mathbf{t}$ line: $\mathbf{q}_l$: point on line $\mathbf{u}$: line direction $\mathbf{q}_l' = \mathbf{R}\mathbf{q}_l - \mathbf{t}$ $\mathbf{u}' = \mathbf{R}\mathbf{u}$ plane: $\mathbf{n}$: normal vector of plane δ: distance of plane to origin $\mathbf{n}' = \mathbf{R}\mathbf{n}$ $\delta' = \delta + \mathbf{t} \cdot \mathbf{n}'$	$\mathbf{R}, \mathbf{t}$: rotor, translation vector point: $\mathbf{q}$: point $\mathbf{q}' = \mathbf{R}\mathbf{q}\mathbf{R}^{-1} + \mathbf{t}$ line: $\mathbf{q}_l$: point on line $\mathbf{u}$: line direction $\mathbf{q}_l' = \mathbf{R}\mathbf{q}_l\mathbf{R}^{-1} - \mathbf{t}$ $\mathbf{u}' = \mathbf{R}\mathbf{u}\mathbf{R}^{-1}$ plane: $\mathbf{p}$: bivector of the plane δ: distance of plane to origin $\mathbf{p}' = \mathbf{R}\mathbf{p}\mathbf{R}^{-1}$ $\delta' = \delta + (\mathbf{t} \wedge \mathbf{p}')^*$	$\mathbf{M}$: 4×4 transformation matrix $\mathbf{q}$: point $\mathbf{p}$: plane $\mathbf{q}' = \mathbf{M}\mathbf{q}$ $\mathbf{p}' = \mathbf{M}^{-T}\mathbf{p}$ $\mathbf{l} = (\vec{u} : \vec{v})$: line $\mathbf{q}' = (\vec{q}' : 1) = \mathbf{M}(\vec{v} \times \vec{u} : 1)$ $\mathbf{l}' = (\mathbf{M}\vec{u} : \mathbf{M}\vec{u} \times \vec{q}')$	$\mathbf{M}$: outermorphism operator $\mathbf{x}$: any primitive $\mathbf{x}' = \mathbf{M}\mathbf{x}$	$\mathbf{x}$: any primitive $\mathbf{V}$: versor representing rotation, translation $\mathbf{x}' = \mathbf{V}\mathbf{x}\mathbf{V}^{-1}$
line-plane intersection	$\mathbf{q}_l$: point on line $\mathbf{u}$: direction of line $\mathbf{n}$: normal vector of plane δ: distance of plane to origin $\mathbf{q}_i = \mathbf{q}_l - \frac{((\mathbf{q}_l \cdot \mathbf{n}) - \delta)\,\mathbf{u}}{\mathbf{u} \cdot \mathbf{n}}$	$\mathbf{q}_l$: point on line $\mathbf{u}$: line direction $\mathbf{p}$: bivector of the plane δ: distance of plane to origin $\mathbf{q}_i = \mathbf{q}_l - \frac{((\mathbf{q}_l \wedge \mathbf{p})^* - \delta)\,\mathbf{u}}{(\mathbf{u} \wedge \mathbf{p})^*}$	$\mathbf{l} = (\vec{u} : \vec{v})$: line $\mathbf{p} = [\vec{n} : \delta]$: plane $\mathbf{q} =$ [illegible]	$\mathbf{l}, \mathbf{p}$: line, plane $\mathbf{q} = \mathbf{p}^* \cdot \mathbf{l}$	$\mathbf{l}, \mathbf{p}$: line, plane $\mathbf{q} \wedge \mathbf{e}_\infty = \mathbf{p}^* \cdot \mathbf{l}$ plus one flat point decomposition, if required
line-sphere intersection	$\mathbf{q}_l$: point on line $\mathbf{u}$: direction of line $\mathbf{q}_s$: center point of sphere ρ: radius of the sphere $\mathbf{q}_c = \mathbf{q}_l + ((\mathbf{q}_s - \mathbf{q}_l) \cdot \mathbf{u})\,\mathbf{u}$ $\delta_n^2 = \frac{(\mathbf{q}_c - \mathbf{q}_s)\cdot(\mathbf{q}_c - \mathbf{q}_s)}{\rho^2}$ $\mathbf{q}_\pm = \mathbf{q}_c \pm \rho\sqrt{1 - \delta_n^2}\,\mathbf{u}$	$\mathbf{q}_l$: point on line $\mathbf{u}$: direction of line $\mathbf{q}_s$: center point of sphere ρ: radius of the sphere $\mathbf{q}_c = \mathbf{q}_l + ((\mathbf{q}_s - \mathbf{q}_l) \cdot \mathbf{u})\,\mathbf{u}$ $\delta_n^2 = \frac{(\mathbf{q}_c - \mathbf{q}_s)^2}{\rho^2}$ $\mathbf{q}_\pm = \mathbf{q}_c \pm \rho\sqrt{1 - \delta_n^2}\,\mathbf{u}$	$\mathbf{l} = (\vec{u} : \vec{v})$: line $\mathbf{q}_s = (\vec{s} : 1)$: center point of sphere ρ: radius of the sphere $\mathbf{l}_t = (\vec{u} : \vec{v} - \vec{u} \times \vec{t}) = (\vec{u}_t : \vec{v}_t)$ $\mathbf{q}_c = \left(\frac{\vec{v}_t \times \vec{u}_t}{\vec{u}_t} : 1\right)$ $\delta_n^2 =$ [illegible] $\mathbf{q}_\pm = \mathbf{q}_c \pm \rho\sqrt{1 - \delta_n^2}\,\vec{u}_t$	$\mathbf{l}$: line $\mathbf{q}_s$: center point of sphere ρ: radius of sphere $\mathbf{t} = -\mathbf{q}_s + \mathbf{e}_0$ $\mathbf{l}_t = \mathbf{l} + \mathbf{t} \wedge (\mathbf{e}_0 \cdot \mathbf{l})$ $\mathbf{q}_c = (\mathbf{e}_0 \cdot \mathbf{l}_t)\mathbf{l}_t^{-1} - \mathbf{t}$ $\delta_n^2 =$ [illegible] $\mathbf{q}_\pm = \mathbf{q}_c \pm \rho\sqrt{1 - \delta_n^2}\,(\mathbf{e}_0 \cdot \mathbf{l})$	$\mathbf{l}, \mathbf{s}$: line, sphere $\mathbf{r} = \mathbf{s}^* \cdot \mathbf{l}$ $\mathbf{q}_\pm = \frac{\pm\sqrt{\mathbf{r} \cdot \mathbf{r}} + \mathbf{r}}{\mathbf{e}_\infty \cdot \mathbf{r}}$
re ection	$\mathbf{q}_l$: point on line $\mathbf{u}$: direction of line $\mathbf{n}$: normal vector of plane δ: distance of plane to origin $\mathbf{u}' = -2(\mathbf{n} \cdot \mathbf{u})\mathbf{n} + \mathbf{u}$ plus one line-plane intersection	$\mathbf{q}_l$: point on line $\mathbf{u}$: direction of line $\mathbf{p}$: bivector of the plane δ: distance of plane to the origin $\mathbf{u}' = \mathbf{p}\mathbf{u}\mathbf{p}$ plus one line-plane intersection	$\mathbf{l} = (\vec{u} : \vec{v})$: line $\mathbf{p} = [\vec{n} : \delta]$: plane $\vec{u}' = -2(\vec{n} \cdot \vec{u})\vec{n} + \vec{u}$ plus one line-plane intersection	$\mathbf{l}, \mathbf{p}$: line, plane $\mathbf{q} = \mathbf{p}^* \cdot \mathbf{l}$, $\mathbf{t} =$ [illegible] $\mathbf{l}_t = \mathbf{l} - \mathbf{t} \wedge (\mathbf{e}_0 \cdot \mathbf{l})$ $\mathbf{p}_t = \mathbf{p} - \mathbf{t} \wedge (\mathbf{e}_0 \cdot \mathbf{p})$ $\mathbf{l}_t' = \mathbf{p}_t\mathbf{l}_t\mathbf{p}_t$ $\mathbf{l}' = \mathbf{l}_t' + \mathbf{t} \wedge (\mathbf{e}_0 \cdot \mathbf{l}_t')$	$\mathbf{l}, \mathbf{p}$: line, plane $\mathbf{l}' = \mathbf{p}\mathbf{l}\mathbf{p}^{-1}$
refraction	$\mathbf{q}_l$: point on line $\mathbf{u}$: direction of line $\mathbf{n}$: normal vector of plane δ: distance of plane to origin $\eta = \frac{\eta_2}{\eta_1}, \gamma = \mathbf{n} \cdot \mathbf{u}$: constants $\kappa = \mathrm{sign}(\gamma)\sqrt{1 - \eta^2 + \gamma^2\eta^2} - \gamma\eta$ $\mathbf{u}' = \kappa\mathbf{n} + \eta\mathbf{u}$ plus one line-plane intersection	$\mathbf{q}_l$: point on line $\mathbf{u}$: direction of line $\mathbf{n} = \mathbf{p}^*$: normal vector of plane δ: distance of plane to origin $\eta = \frac{\eta_2}{\eta_1}, \gamma = \mathbf{n} \cdot \mathbf{u}$: constants $\kappa = \mathrm{sign}(\gamma)\sqrt{1 - \eta^2 + \gamma^2\eta^2} - \gamma\eta$ $\mathbf{u}' = \kappa\mathbf{n} + \eta\mathbf{u}$ plus one line-plane intersection	$\mathbf{l} = (\vec{u} : \vec{v})$: line $\mathbf{p} = [\vec{n} : \delta]$: plane $\eta = \frac{\eta_2}{\eta_1}, \gamma = \vec{n} \cdot \vec{u}$: constants $\kappa = \mathrm{sign}(\gamma)\sqrt{1 - \eta^2 + \gamma^2\eta^2} - \gamma\eta$ $\vec{u}' = \kappa\vec{n} + \eta\vec{u}$ plus one line-plane intersection plus one line construction	$\mathbf{l}, \mathbf{p}$: line, plane $\mathbf{u} = \mathbf{e}_0 \cdot \mathbf{l}$: direction of line $\mathbf{n} = \mathbf{p}^* - (\mathbf{e}_0 \cdot \mathbf{p}^*)\,\mathbf{e}_0$: normal of plane $\eta = \frac{\eta_2}{\eta_1}, \gamma = \mathbf{n} \cdot \mathbf{u}$: constants $\kappa = \mathrm{sign}(\gamma)\sqrt{1 - \eta^2 + \gamma^2\eta^2} - \gamma\eta$ $\mathbf{u}' = \kappa\mathbf{n} + \eta\mathbf{u}$, $\mathbf{l}' = (\mathbf{p}^* \cdot \mathbf{l}) \wedge \mathbf{u}'$	$\mathbf{l}, \mathbf{p}$: line, plane $\eta = \frac{\eta_2}{\eta_1}, \gamma = \mathbf{n} \cdot \mathbf{u}$: constants [illegible]

Table B.2: Ray tracer operations.

Appendix C: Samenvatting
(Summary in Dutch)

Dit proefschrift behandelt de efficiënte implementatie van geometrische algebra. Dit is een algebra die, zoals de naam suggereert, bijzonder geschikt is om geometrische (ruimtelijke) berekeningen mee uit te voeren. Geometrie wordt overal in de informatica gebruikt. Voorbeelden van toepassings- gebieden zijn:

- CAD/CAM: ontwerp en produktie van producten met behulp van computers.
- 2-D weergave van 3-D scenes: voor speciale effecten in films, voor computerspellen, voor medische apparatuur.
- Aansturen van (industriële) robots.

De geometrie in deze toepassingen wordt traditioneel geformuleerd met gebruik van lineaire algebra. Lineaire algebra formaliseert vectorruimten, vectoren en lineaire transformaties. Het heeft echter een beperkt 'vocabulaire': alles moet in termen van vectoren (of combinaties daarvan) worden uitgedrukt. Een lijn kan bijvoorbeeld worden voorgesteld als de combinatie van een vector die de plaats van een punt op de lijn aangeeft, plus een tweede vector die de richting van de lijn aangeeft. Het beperkte vocabulaire leidt al snel tot het gebruik van truckjes en uitbreidingen zoals quaternionen en Plücker coördinaten. Door de ongestrureerde manier waarop dit gebeurt, moet iedereen moet het wiel opnieuw uitvinden om deze uitbreidingen tot één geheel te smeden. Dit ad hoc knutselwerk leidt weer makkelijk tot kostbare fouten in computer programma's.

Geometrische algebra biedt een krachtig alternatief voor lineaire algebra, omdat het een veel completere 'taal' te bieden heeft. Naast vectoren zitten er in een geometrische algebra veel meer elementen die gebruikt kunnen worden om geometrische concepten te representeren. De zogenaamde *blades* worden gebruikt om objecten voor te stellen, de *versors* stellen orthogonale transformaties voor. Ook zijn er producten om met deze elementen te werken. Het *uitproduct* (*outer product*) spant blades op, het *inproduct* (*inner product*) wordt gebruikt om metrische relaties tussen blades te meten, en het *geometrische product* (*geometric product*) maakt versors en past deze toe op blades en andere versors.

In hun meest eenvoudige interpretatie stellen de blades deelruimtes door de oorsprong voor. De versoren stellen rotaties en reflecties voor. Deze interpretatie biedt echter te weinig mogelijkheden om er serieus geometrie mee te implementeren, tenzij we vervallen in hetzelfde soort truckjes als in lineaire algebra wordt gebruikt. Maar door een meer geavanceerde interpretatie te gebruiken, kunnen met blades en versors veel meer concepten worden gerepresenteerd. Het beste voorbeeld hiervan is het zogenaamde *conforme model*. Dit model bedt een n-dimensionale ruimte in een $n+2$-dimensionale ruimte (dus bijvoorbeeld een 5-D algebra voor een 3-D ruimte). De twee extra dimensies stellen het punt in de oorsprong en het punt op het oneindige voor. Blades kunnen hierdoor onder andere punten, circles, lijnen, vlakken, bollen en raaklijnen voorstellen. Door ook nog een slimme metriek te gebruiken kunnen versoren alle conforme (hoekbehoudende) transformaties voorstellen. De Euclidische transformaties rotatie,

translatie en reflectie vallen hieronder, en dit zijn precies de transformaties die voor veel informatica-toepassingen nodig zijn.

Dit alles levert een veel meer gestructureerde taal op dan lineaire algeba. Maar helaas moet hiervoor wel een prijs worden betaald. Een geometrische algebra over n-dimensionale vectorruimte is eigenlijk een 2^n-dimensionale lineaire ruimte. Dus om met het conforme model over 3-D te kunnen rekenen is een 32-dimensionale lineaire ruimte nodig ($2^5 = 32$). Door deze hoog-dimensionaliteit is een naïeve implementatie van geometrische algebra al snel twee ordes van grootte langzamer dan zijn klassieke lineaire algebra tegenhanger. In dit proefschift laten we op twee manieren zien hoe deze inefficiëntie kan worden overkomen.

De eerste manier buit de structuur van de algebra uit om overbodige berekeningen te voorkomen. Het is een optimalisatie van bestaande implementatie-ideeën, gebruik makend van automatische code generatie. In deze method worden de elementen van de algebra voorgesteld als een gewogen som van *basis blades*. Er zijn 2^n basis blades nodig om de ruimte van een n-dimensionale geometrische algebra op te spannen, en om een element uit de algebra voor te stellen heb je dus ook 2^n coördinaten nodig. Gelukkig worden in de praktijk slechts een deel van de 2^n coördinaten benut voor elk element. Zo zijn er slechts zes getallen nodig om een lijn in 3-D voor te stellen, en niet 32. Onze implementatie maakt hiervan gebruik door specifieke typen (klassen) te genereren voor specifieke elementen van de algebra. Door deze en nog vele andere optimalisaties toe te passen, is het mogelijk om geometrische algebra (bijna) net zo efficiënt te implementeren als lineaire algebra. In sommige gevallen is geometrische algebra zelfs sneller. We laten dit zien aan de hand van een groot aantal benchmarks.

Ondanks onze optimalisaties voor de additive representatie komt er een moment dat deze onbruikbaar wordt, doordat 2^n natuurlijk zeer groot wordt als n groot wordt. Bijvoorbeeld wanneer $n = 10$, dan $2^n = 1.024$; als $n = 20$, dan $2^n = 1.048.576$. Dit zou betekenen dat de implementatie van een geometrische algebra over bijvoorbeeld een 20-dimensionale ruimte onbruikbaar langzaam zou zijn. Echter, de meeste nuttige elementen in een geometrische algebra hebben een vector-factorisatie: blades zijn het uitproduct van vectoren, versors zijn het geometrisch product van vectoren. Door blades en versors op te slaan als producten van vectoren, is veel minder opslag nodig. Er zijn namelijk nog slechts maximaal n^2 coordinaten nodig, dus bijvoorbeeld voor 20-D nog maar maximaal 400 coordinaten. Voor deze *multiplicatieve representatie* hebben wij uitgezocht hoe alle operaties van geometrische algebra kunnen worden implementeerd. Het blijkt dat dit mogelijk is, en we laten zien dat vanaf ongeveer 10-D de multiplicatieve representatie efficienter is dan de additieve.

Tot slot laten we nog een praktisch voorbeeld van het gebruik van geometrische algebra zien. We beschrijven de implementatie van een *ray tracer* met behulp van het conforme model. Een ray tracer is een type programma wat wordt gebruikt om beelden van bijvoorbeeld 3-D animatiefilms uit te rekenen. We laten zien hoe de mogelijkheden van geometrische algebra de implementatie keuzes beinvloeden. Ook gebruiken we de ray tracer als meetplatform om aan te tonen dat

geometrische algebra niet fundamenteel langzamer is dan de klassieke aanpak, mits goed geïmplementeerd.

Appendix D: Abstract

Most of the geometry which is used in fields like computer vision and computer graphics is classically formalized using linear algebra. The limitation of linear algebra is that all geometric concepts have to be represented by (combinations of) vectors and matrices. This limited 'vocabulary' leads to all kind of tricks, hacks and extensions, which in turn leads to programming errors.

Geometric algebra offers a powerful, more elegant alternative. It makes oriented subspaces and orthogonal transformations regular elements of the algebra. A rich set of products and operations is available to work with these elements. Because of the unified treatment of subspaces and transformations, many functions over the algebra are universally applicable to all types of elements. By interpreting the elements of the algebra in special ways, they can represent geometric concepts such as points, lines, circles, spheres, tangents, rotations, reflections and so on.

However, this does not come for free. The elements of a geometric algebra over n-D space live in a 2^n dimensional linear space. In essence, this is the way one pays for the greater level of abstraction. For example, the conformal model of n-D space (a particularly useful application of geometric algebra), requires an $n+2$-dimensional algebra, which means that to do 3-D geometry, we need to work in a 32-dimensional space ($2^{3+2} = 32$). This suggests that geometric algebra is hopelessly inefficient in practice.

Early, naive implementations seemed to confirm this. However, in this thesis we show that geometric algebra is not inherently inefficient. By properly exploiting the structure of geometric algebra we can get its run-time performance close to (or in certain cases, beyond) that of classical linear-algebra-based geometry implementations.

We discuss two approaches. The first approach is a significant optimization of an existing implementation method. We show that it is possible to automatically generate practically usable geometric algebra implementations from specification, and that these implementations have a run-time performance comparable to that of traditional geometry implementations.

The second approach is a more fundamental innovation. It establishes an upper-bound to the storage complexity and time complexity of geometric algebra. These bounds are $O(n^2)$ for the memory required to store an element, and $O(n^3)$ for the time required to evaluate a product. This is much lower than the $O(2^n)$ mentioned above, and makes it possible to use geometric algebra in high-dimensional spaces. The core idea is to represent the elements in factored form. Linear algebra techniques are used to implement the products and operations on top of this.

Bibliography

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

[2] O. Beckmann, A. Houghton, P. Kelly, and M. Mellor. Run-time code generation in c++ as a foundation for domain-specific optimisation. *Lecture Notes in Computer Science*, 3016:291–306, 2004.

[3] I. Bell. Ian Bell's pages on geometric algebra. `http://www.iancgbell.clara.net/maths/geoalg1.htm`, 2007.

[4] J. Blinn. *Jim Blinn's Corner Notation, Notation, Notation.* Morgan Kaufmann, 2002.

[5] T.A. Bouma. Personal communication, 2001.

[6] T.A. Bouma, L. Dorst, and H.G.J. Pijls. Geometric algebra for subspace operations. *Acta Mathematicae Applicandae*, pages 285–300, 2002.

[7] T.A. Bouma and G. Memowich. Invertible homogeneous versors are blades. Available at `http://www.science.uva.nl/ga/publications`, 2001.

[8] K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications.* Addison-Wesley Pub Co, 2000.

[9] H. F. de Groote. *Lectures on the complexity of bilinear problems (Lecture notes in computer science 245).* Springer, 1987.

[10] C. Doran and A. Lasenby. *Geometric Algebra for Physicists.* Cambridge University Press, 2003.

[11] L. Dorst, D.Fontijne, and S. Mann. Companion website to geometric algebra for computer science: An object oriented approach to geometry. Available at `http://www.geometricalgebra.net`, 2007.

[12] L. Dorst, D.Fontijne, and S. Mann. *Geometric Algebra for Computer Science: An Object Oriented Approach to Geometry.* Morgan Kaufmann, 2007.

[13] L. Dorst and D. Fontijne. An algebraic foundation for object-oriented Euclidean geometry. In *ITM 2003 proceedings*, 2003.

[14] L. Dorst, S. Mann, and T. Bouma. GABLE: A matlab tutorial for geometric algebra. Available at `http://www.wins.uva.nl/~leo/clifford/gable.html`, 1999.

[15] P. Fleckenstein. C++ template classes for geometric algebras. Available at `http://www.nklein.com/products/geoma`.

[16] D. Fontijne. Performance and elegance of five models of 3-D Euclidean geometry in a ray tracing application. Talk at IMA Conference on Applications of Geometric Algebra, 5 - 6 September 2002, Trinity College, Cambridge, 2002.

[17] D. Fontijne. Implementation of Clifford algebra for blades and versors in $o(n^3)$ time. Talk at International Conference on Clifford Algebra 7, May 19 - May 29, 2005, Toulouse, France, 2005.

[18] D. Fontijne. Gaigen 2: a geometric algebra implementation generator. In *Proceedings of GPCE'06.*, 2006.

[19] D. Fontijne and L. Dorst. Modeling 3-D Euclidean geometry. *IEEE Computer Graphics and Applications*, 23(2):68–78, March-April 2003.

[20] A. S. Glassner. *An Introduction To Ray Tracing.* Academic Press, 1989.

[21] R. Goldman. Illicit expressions in vector algebra. *ACM Transactions on Graphics*, 4(3), July 1985.

[22] R. Goldman. The ambient spaces of computer graphics and geometric modeling. *IEEE Computer Graphics and Applications*, 20:76–84, 2000.

[23] G. Golub and C. Van Loan. *Matrix Computations, 3rd edition.* Johns Hopkins University Press, 1997.

[24] A. Gräf. Q - equational programming language. Available at `http://q-lang.sourceforge.net`, 2007.

[25] D. Hestenes. *New Foundations for Classical Mechanics.* Reidel, 2nd edition, 2000.

[26] D. Hestenes, A. Rockwood, and H. Li. System for encoding and manipulating models of objects. U.S. Patent 6,853,964, granted February 8, 2005.

[27] D. Hestenes and G. Sobczyk. *Clifford Algebra to Geometric Calculus.* Reidel, 1984.

[28] D. Hildenbrand, D. Fontijne, Y. Wang, M. Alexa, and L. Dorst. Competitive runtime performance for inverse kinematics algorithms using conformal geometric algebra. In *Proceedings of Eurographics 2006, Vienna*, 2006.

[29] H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, Ltd., 2001.

[30] J.B. Kuipers. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace and Virtual Reality*. Princeton University Press, 2002.

[31] P. Leopardi. Glucat. Available from `http://glucat.sourceforge.net`.

[32] S. Mann, L. Dorst, and T. Bouma. The making of GABLE, a geometric algebra learning environment in Matlab. In E. Bayro-Corrochano and G. Sobczyk, editors, *Geometric Algebra with Applications in Science and Engineering*, chapter 24, pages 491–511. Birkhäuser, 2001.

[33] S. Mann, N. Litke, and T. DeRose. A coordinate free geometry adt. Technical Report CS-97-15, University of Waterloo, 1997.

[34] S. Mann and A. Rockwool. Using geometric algebra to compute singularities in 3-D vector fields. In *IEEE Visualization 2002 Proceedings*, pages 283–289, 2002.

[35] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1995.

[36] C. Perwass. Clifford algebra library and utilities. Available from `http://www.clucalc.info`.

[37] M. Riesz. *Clifford Numbers and Spinors*. Kluwer Academic, 1993.

[38] K. Shoemake. Animating rotation with quaternion curves. In *Proceedings of ACM SIGGRAPH*, volume 19:3, pages 245–254, 1985.

[39] D. Shreiner, M. Woo, J. Neider, T. Davis, and OpenGL Architecture Review Board. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4, Fourth Edition*. Addison-Wesley Pub Co, 2003.

[40] J. Stolfi. *Oriented Projective Geometry*. Academic Press, 1991.

[41] J. Suter. Clifford (software), 2003. Used to be available at `http://www.jaapsuter.com`.

[42] Jaap Suter. Personal communication, 2004.

[43] D. Tolani, A. Goswami, and N.I. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical Models*, 62(5):353–388, September 2000.

[44] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, , and J. Visser. The asf+sdf meta-environment: a component-based language development environment. In *Proceedings of Compiler Construction (CC'01), LNCS 2027*, 2001.

Acknowledgements

Many people have helped me making it to the completion of my PhD thesis. First of all, I would like to thank Leo Dorst for introducing me to geometric algebra, helping me to learn it, and for the friendly and open work environment. “One day you are a member of someone’s a M.Sc. committee, the next moment you are cooperating for ten years”.

Thanks to Dick Grune for serving as my sparring partner for discussions on actual implementation and programming languages, and for the help writing [18].

Thanks to Jack Hanlon for his friendly e-mail conversations and for his avid usage of GAViewer. Thanks to Robert Valkenburg who was one the most faithful `Gaigen 1` users (and testers), detecting bugs and suggesting improvements.

Credits go to Tim Bouma for suggesting to store versors as ordered list of vectors for $O(N^2)$ storage, which formed the starting point for Chapter 5. Hongbo Li pointed out that some of our assumptions for the geometric product originally used in this chapter were false.

I would also like to thank Christian Perwass, Dietmar Hildenbrand, Jaap Suter, Stephen Mann, and Ian Bell for their geometric algebra implementations and applications which I used for the benchmarks in Chapter 7. Thanks to Ian Bell for the discussion on the `meet` and `join` algorithm, which led to improvements of the algorithm in Section 3.4.7.

Besides writing a thesis, I also co-authored the GA4CS book [12] for which I would like to thank Frans Groen for enduring the book writing process; no doubt it delayed my thesis work. Also, I would like to thank Stephen Mann and Leo Dorst for giving me the opportunity to contribute to the book, and Morgan Kaufmann for allowing me to re-use parts of that book for this thesis.

Finally I would like to thank Yvonne, my parents and my parents-in-law for the many other developments in my life: building a house, working on motion capture, animation and performances, having children. Thanks to you my PhD study was a lively time!