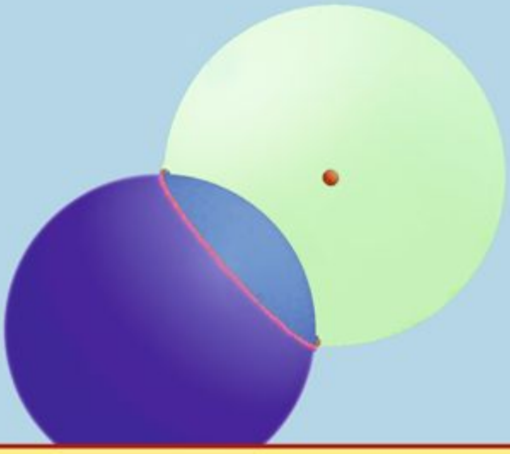
Geometry and Computing 8



Dietmar Hildenbrand

Foundations of Geometric Algebra Computing

Springer

Geometry and Computing

Dietmar Hildenbrand

Foundations of Geometric Algebra Computing

Springer

Dr. Dietmar Hildenbrand
University of Technology Darmstadt
Darmstadt
Germany

ISSN 1866-6795 ISSN 1866-6809 (electronic)
ISBN 978-3-642-31793-4 ISBN 978-3-642-31794-1 (eBook)
DOI 10.1007/978-3-642-31794-1
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012953093

ACM Computing Classification (1998): B.8.2, C.1.4, D.1.3, G.1.2, G.4, I.3.5, F.2.2

Mathematics Subject Classification (2000): 65Dxx, 68U05

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To 150 years of Hermann G. Grassmann's Ausdehnungslehre of 1862

Foreword

If I have seen further it is by standing on the shoulders of giants.
– I. Newton, 15 Feb 1676.

Newton's well-known quote to Hooke finds no better illustration than in the development of what we call Geometric Algebra. The primary set of shoulders in the Geometric Algebra epic belong to Hermann Günther Grassmann, who epitomized the mathematical mind, at least in process, if not in expositional style. He systematically created algebras for geometric concepts by listing essential elements and operations and then reducing them to minimal axiomatic schemes. In his book *Die Lineale Ausdehnungslehre* he laid the foundation for scores of mathematical systems bearing names like projective and exterior geometry and Quaternion, Clifford, Gibbs, Cartan, and Boolean algebra. Unfortunately, his book was densely written with many religious allusions and it shifted without warning between projective and Euclidean spaces.

One of the fundamental ideas of Grassmann was that of different grade (intrinsic dimensions) elements; thus, in addition to scalar and vector elements, with which we are well familiarized today, he would have included bi-vectors, tri-vectors etc., not as combinations of vectors like the cross-product, but as axiomatic elements. His book introduced two fundamental operations between elements, the inner and outer products. One was simply a grade lowering and the other a grade elevating operation between elements. Between vectors, these operations became the familiar dot product (or projection) and the wedge product (or plane through the vectors). The import of these ideas becomes apparent when one considers what else can one say about the relation of two vectors? The grade lowering and grade elevating operators underpin any comprehensive vector algebra.

It was William Kingdon Clifford, who then recognized the sublime value of joining these two operations, inner and outer, into one product via a new algebraic "addition" of the two products. The geometric product of Clifford was cast as the fundamental operation with inner and outer products defined in terms of it. Its subtlety lies in the fact that it is not a binary operation as we are used to thinking of additions; it does not replace two elements with a third "sum" of the same type of element. Instead, it is a prepositional operation that only makes sense in the overall algebraic structure. It is rather like adding real and imaginary parts in complex analysis.

The subsequent evolution of "Clifford" Algebra became a candidate for the mathematical model of nineteenth-century physics, along with the quaternions of Hamilton. The winner of this intellectual contest, however, was eventually another Grassmannian derivative, the vector algebra proposed by Josiah Gibbs, a thermodynamicist. Gibbs' focus was three dimensions, appropriate to his area of research. As such he considered scalars and vectors to be sufficient; the bi-vector could be represented by the cross product, another vector. Hence there was no need for bi-vectors, and certainly not tri-vectors, nor anything higher. Two elements, the scalar and the vector, seem to satisfy Occam's razor. This may well be true, but only for three dimensions. As physics evolved in higher dimensions, the inadequacies of the cross product have become apparent. Pauli and Dirac invented their own versions of spinors, for example, a concept which has since been shown to be fundamental in any comprehensive formulation of an axiomatic physics system.

Our current state of babel has different mathematical systems, which require sophisticated methods of translation between them. They are often so common to us now that we do not see the clumsiness of, say, rewriting complex analysis in terms of 2D vectors, or 3D vectors in terms of quaternions to rotate and rewriting back. Similar situations exist between exterior algebra, Pauli and linear algebras, and the list goes on. A single, comprehensive algebra is both pedagogically and operationally "a consummation devoutly to be wish'd". That is the goal of Geometric Algebra.

In the last 50 years, Clifford Algebra found a strong proponent in physicist David Hestenes, who used it as the basis to describe electron theory and celestial mechanics among other very successful applications. His version of the Geometric Algebra was developed with a strong emphasis on the geometrically intuitive aspects of the algebra.

While Geometric Algebra is generally acknowledged to be a compelling and comprehensive system of mathematics (and it is beginning to find traction in many application areas) one major obstacle exists to its broader adoption, which is the very practical one. How do I compute with it? Without a Maple- or Mathematica-like facility, its usability is vastly limited in today's modern research or engineering environment. The way forward is obvious and a number of researchers have addressed this problem with computer algebraic systems based on Geometric Algebra. Two of the most current and popular are CLUCalc and Gaalop, as described in this book.

Experience with these methods and the innate characteristics of Geometric Algebra now point to the next logical step in the evolution. It is the need to use modern parallel architecture to accelerate Geometric Algebra. We can not only increase the range of realizable applications through speed and efficiency, but it provides unique and valuable insights into the algebra itself. This is the natural evolution and critical path to bringing this richer, more comprehensive system of mathematics, this Geometric Algebra, to the collective scientific consciousness.

This book by Hildenbrand is, in my opinion, the next, necessary and joyful twist in this elegant evolutionary thread stretching back to Grassmann and beyond. He gives a highly readable account of the development of Geometric Algebra. He is able to cook the subject matter like a good meal, and then, pulling all the pieces together, feeds us a very compelling solution for the next steps in creating the most advanced environment for learning, applying and enjoying the beauty of Geometric Algebra.

Bon appetit

Thuwal, Kingdom of Saudi Arabia Prof. Alyn Rockwood
May 2012

Preface

Hermann G. Grassmann's *Ausdehnungslehre* of 1862 laid the foundations for Geometric Algebra as a mathematical language combining geometry and algebra. A hundred and fifty years later, this book is intended to lay the foundations for the widespread use of this mathematical system on various computing platforms.

Seventeen years after Grassmann's first more philosophically written *Ausdehnungslehre* of 1844, he admitted in the preface of his mathematical version of 1862, "I remain completely confident that the labor I have expended on the science presented here and which has demanded a significant part of my life, as well as the most strenuous application of my powers, will not be lost. It is true that I am aware that the form which I have given the science is imperfect and must be imperfect. But I know and feel obliged to state (though I run the risk of seeming arrogant) that even if this work should again remain unused for another seventeen years or even longer, without entering into the actual development of science, still that time will come when it will be brought forth from the dust of oblivion and when ideas now dormant will bring forth fruit." And he went on to say, "there will come a time when these ideas, perhaps in a new form, will arise anew and will enter into a living communication with contemporary developments."

The form that we give to Geometric Algebra in this book has the power to lead easily from the geometric intuition of solving an engineering application to its efficient implementation on current and future computing platforms. We show how easy it is to develop new algorithms in areas such as computer graphics, robotics, computer animation, and computer simulation. Owing to its geometric intuitiveness, compactness, and simplicity, algorithms based on Geometric Algebra can lead to enhanced quality, a reduction in development time and solutions that are more easily understandable and maintainable. Often, a clear structure and greater elegance result in lower runtime performance. However, based on our computing technology, Geometric Algebra implementations can even be faster and more robust than conventional ones.

I really do hope that this book can support the widespread use of Geometric Algebra Computing technology in many engineering fields.

Darmstadt, Germany Dr. Dietmar Hildenbrand
May 2012

Acknowledgments

First of all, I would like to express my thanks to Professor Marc Alexa, who put me in touch with Geometric Algebra in 2002 and, as my advisor, gave me the chance to study for a Ph.D. dealing with the application of Geometric Algebra to computer graphics and robotics. Thanks to Marc, one focus of my dissertation was on the runtime performance of Geometric Algebra algorithms.

I am grateful to Professor Eduardo Bayro-Corrochano for good cooperation in past years, especially concerning the application of Geometric Algebra to robotics. Thanks to Eduardo, I got to know the Cliffordlib Maple library, created by Rafal Ablamowicz and Bertfried Fauser, which we could use advantageously for our first runtime optimization study.

Thanks to Dr. habil. Christian Perwass, I was able to use a very powerful and productive tool for the interactive and visual development and testing of Geometric Algebra algorithms; Christian's tool CLUCalc has accompanied my research into Geometric Algebra from its early beginnings to today. I am very grateful for his good cooperation over the last 10 years.

I would like to thank Professor Andreas Koch and his group for their very good cooperation on compiler technology. Thanks to funding from the DFG (Deutsche Forschungsgemeinschaft), we were able to cooperate on a common research project dealing with a Geometric Algebra compiler for FPGAs.

The molecular dynamics simulation described in this book was a joint work with the high-performance computing center at the University of Stuttgart with special thanks to Florian Seybold.

Many thanks to Professor Vaclav Skala for his GraVisMa initiative.

I would like to thank Professor Kanta Tachibana and Professor Yukio Kaneda for the chance to present my early Geometric Algebra Computing results in some tutorials at the Frontiers of Computational Science Center of Excellence in Nagoya, Japan, as well as for the honor of becoming a member of the advisory committee of this center.

Thanks are due for the enthusiastic work of my former students Holger Griesheimer, Stefan Rockensuess, Jun Zhao, Thomas Kalbe, Haidan Zhang, Yusheng Wang, Sebastian Hartte, Carsten Cibura, Elmar Brendel, Florian Woersdoerfer, Christian Schwinn, Meike Becker, Roman Getto, Crispin Deul, Michael Burger, and Andreas Goerlitz. Thanks to Marco Hutter for proofreading parts of the manuscript. Special thanks to Joe Pitt for the first version of our Gaalop compiler, and to Patrick Charrier and Christian Steinmetz for the current version of Gaalop and the precompilers for C++ and OpenCL, as well as for proofreading of the manuscript.

Thanks to Professor David Hestenes, Professor Hongbo Li, Professor Alyn Rockwood, Prof. Eckhard Hitzer, Professor Rafal Ablamowicz, Professor Gerik Scheuermann, Professor Wolfgang Strasser, Dr. Joan Lasenby, Dr. Leo Dorst, Dr. Daniel Fontijne, Dr. Julio Zamora, Dr. Martin Horn, Professor Paul Drechsel, and Professor Hans-Joachim Petsche for a lot of very fruitful discussions. I would like to thank Prof. Simos and the "*European Society of Computational Methods in Sciences and Engineering*". They awarded me in 2012 with the highest distinction of "Honorary Fellowship" for my work in the field of Applied Mathematics.

I would like to thank Professor Zoubir for his support and I hope that our LOEWE research priority program Cocoon will benefit a lot from this book.

Finally, I would like to thank my wife Carola for her interest shown in my work and her support, understanding and patience during recent months.

Contents

List of Figures

List of Tables

Chapter 2
Mathematical Introduction

Part I of this book provides a theoretical introduction to Geometric Algebra. We focus on 5D Conformal Geometric Algebra because of its intuitive handling of geometric entities, geometric operations, transformations, and motions. We show how a lot of other mathematical systems can be identified within this algebra, and present the fitting of points with the help of spheres and/or planes as an important application.

If you are more interested in a quick interactive, visual overview of Geometric Algebra, you are recommended to start with Part II of this book, especially the tutorial in Chap. 6, and come back to this part whenever you are interested in seeing more details.

This chapter presents mainly the mathematical basics of Geometric Algebra needed in this book. Whenever readers might be interested in a more mathematical and axiomatic approach to Geometric Algebra, we refer to the corresponding sections of [81]. After a short description of the basic algebraic elements and main products of Geometric Algebra we take a look at two specific algebras. With this chapter, we lay the foundations for CGA as described in Chap. 3.

2.1 The Basic Algebraic Elements of Geometric Algebra

While the basis vectors $e_1, e_2, \ldots, e_n$ are the basic algebraic elements of an n-dimensional vector algebra, they are only one part of the algebraic elements of an n-dimensional Geometric Algebra. **Blades** are the basic algebraic elements of Geometric Algebra. An n-dimensional Geometric Algebra consists of blades with **grades** 0, 1, 2, ..., n, where a scalar is a **0-blade** (a blade of grade 0) and the **1-blades** are the basis vectors $e_1, e_2, \ldots, e_n$. The **2-blades** $e_i \wedge e_j$ are blades spanned by two 1-blades, and so on. There exists only one element of the maximum grade n, $I = e_1 \wedge e_2 \ldots \wedge e_n$. It is therefore also called the **pseudoscalar**. A linear

Table 2.1 List of the 8 blades of 3D Euclidean Geometric Algebra

Blade	Grade
1	0
e_1	1
e_2	1
e_3	1
$e_1 \wedge e_2$	2
$e_1 \wedge e_3$	2
$e_2 \wedge e_3$	2
$e_1 \wedge e_2 \wedge e_3$	3

combination of k-blades is called a k-vector (or a vector, bivector, trivector. ...). The sum $e_2 \wedge e_3 + e_1 \wedge e_2$, for instance, is a bivector.

A linear combination of blades with different grades is called a **multivector**. Multivectors are the general elements of a Geometric Algebra.

There are 2^n blades in an n-dimensional Geometric Algebra. Table 2.1, for instance, shows the $8 = 2^3$ blades of 3D Euclidean Geometric Algebra consisting of the scalar, three (basis) vectors, three bivectors, and the pseudoscalar. Additional examples are the 16 blades of 4D Projective Geometric Algebra (Table 2.5) and the 32 blades of the 5D Conformal Geometric Algebra (CGA) (Table 3.1).

2.2 The Products of Geometric Algebra

The main product of Geometric Algebra is called the **geometric product**; many other products can be derived from it.

The three most often used products of Geometric Algebra are the **outer**, the **inner** and the **geometric** product. The notations of these products are listed in Table 2.2. We will use the outer product mainly for the construction and intersection of geometric objects, while the inner product will be used for the computation of angles and distances. The geometric product will be used mainly for the description of transformations.

2.2.1 *The Outer Product*

Geometric Algebra provides an outer product $\wedge$ with the properties listed in Table 2.3.

Property 1 applies only to vectors; the others are generally valid (and so they are also valid for multivectors).

The outer product of two parallel vectors is 0:

$$u \wedge u = -(u \wedge u) = 0. \tag{2.1}$$

Table 2.2 Notations for the geometric algebra products

Notation	Meaning
AB	Geometric product of A and B
$A \wedge B$	Outer product of A and B
$A \cdot B$	Inner product of A and B

Table 2.3 Properties of the outer product $\wedge$

Property	Meaning
Anticommutativity	$u \wedge v = -(v \wedge u)$
Distributivity	$u \wedge (v + w) = u \wedge v + u \wedge w$
Associativity	$u \wedge (v \wedge w) = (u \wedge v) \wedge w$

This is the reason why the outer product can be used as a measure of parallelness. See Chap. 3 of [81] for further details of the products of Geometric Algebra.

In the following, we use two examples from the tutorial in Chap. 6 in order to show how to compute with the outer product.

Computation example 1. We compute the outer product of two vectors according to the tutorial example in Sect. 6.2.1:

$$c = (e_1 + e_2) \wedge (e_1 - e_2)$$

can be transformed based on distributivity to

$$c = (e_1 \wedge e_1) - (e_1 \wedge e_2) + (e_2 \wedge e_1) - (e_2 \wedge e_2);$$

since $u \wedge u = 0$,

$$c = -(e_1 \wedge e_2) + (e_2 \wedge e_1),$$

and because of anticommutativity,

$$c = -(e_1 \wedge e_2) - (e_1 \wedge e_2)$$

or

$$c = -2(e_1 \wedge e_2).$$

In Sect. 6.2.1, we shall see that the geometric meaning of the resulting bivector is a plane element spanned by the two vectors.

Computation example 2. We compute the outer product of three vectors according to the tutorial example in Sect. 6.2.1:

$$d = a \wedge b \wedge c = (e_1 + e_2) \wedge (e_1 - e_2) \wedge e_3.$$

Because of distributivity,

$$d = (\ \underbrace{(e_1 \wedge e_1)}_{0} - (e_1 \wedge e_2) + (e_2 \wedge e_1) - \underbrace{(e_2 \wedge e_2)}_{0}\ ) \wedge e_3$$

$$= (-(e_1 \wedge e_2) + (e_2 \wedge e_1)) \wedge e_3;$$

because of anticommutativity,

$$\begin{aligned} d &= (-(e_1 \wedge e_2) - (e_1 \wedge e_2)) \wedge e_3 \\ &= (-2(e_1 \wedge e_2)) \wedge e_3 \\ &= -2(e_1 \wedge e_2 \wedge e_3) \\ &= -2I. \end{aligned}$$

In Sect. 6.2.1, we shall see that the geometric meaning of the resulting bivector is a volume element spanned by the three vectors a, b, c, which is equal to -2 multiplied by the 3-blade $e_1 \wedge e_2 \wedge e_3$ spanned by the three basis vectors e_1, e_2, e_3, which is equal to the pseudoscalar I.

2.2.2 The Inner Product

For 3D Euclidean space, the inner product of two vectors is the same as the well-known Euclidean scalar product of two vectors. For perpendicular vectors, the inner product is 0; for instance, $e_1 \cdot e_2 = 0$. In Geometric Algebra, however, the inner product is not only defined for vectors. The inner product is grade-decreasing; for example, the result of the inner product of elements with grade 2 and 1 is an element of grade $2-1 = 1$. See Sect. 3.2.7 of [81] for a mathematical treatment of the general inner product.

The inner product of Geometric Algebra contains metric information. Throughout this book, we use Conformal Geometric Algebra (see Chap. 3) and its inner product for the computation of angles and distances. The geometric meaning is treated in Sects. 3.4.1 (inner product and distances) and 3.4.2 (inner product and angles).

2.2.3 The Geometric Product

The geometric product is an amazingly powerful operation., which is used mainly for the handling of transformations. The geometric product of vectors is a combination of the outer product and the inner product. The geometric product of u and v is denoted by uv. For vectors u and v, the geometric product uv can be defined as

$$uv = u \wedge v + u \cdot v. \tag{2.2}$$

We derive the following for the inner and outer products:

$$u \cdot v = \frac{1}{2}(uv + vu), \tag{2.3}$$

$$u \wedge v = \frac{1}{2}(uv - vu). \tag{2.4}$$

but, as noted above, these formulas apply only for vectors, in this form.

See Sect. 3.1 of [81] for an axiomatic approach to the geometric product.

Computation examples. What is the square of a vector?

$$u^2 = uu = \underbrace{u \wedge u}_{0} + u \cdot u = u \cdot u$$

for example

$$e_1^2 = e_1 \cdot e_1 = 1.$$

The geometric product is defined for all kinds of multivectors. Let us calculate, for example, the following geometric product of two bivectors according to Sect. 6.2.3.2:

$$u^2 = (e_1 \wedge e_2)^2$$

Since $e_1e_2 = e_1 \wedge e_2 + \underbrace{e_1 \cdot e_2}_{0} = e_1 \wedge e_2$,

$$u^2 = (e_1 \wedge e_2)^2 = e_1e_2 \underbrace{e_1e_2}_{-e_2e_1} = -e_1 \underbrace{e_2e_2}_{1} e_1 = - \underbrace{e_1e_1}_{1} = -1$$

Because of this property, u can be used such as the imaginary unit i of complex numbers. In the case of $u = e_1 \wedge e_2$ and $v = (e_1 + e_2) \wedge e_3$,

$$\begin{aligned} uv &= (e_1 \wedge e_2)((e_1 + e_2) \wedge e_3), \\ uv &= (e_1e_2)(e_1 \wedge e_3 + e_2 \wedge e_3) \\ &= e_1e_2(e_1e_3 + e_2e_3) \\ &= e_1e_2e_1e_3 + e_1 \underbrace{e_2e_2}_{1} e_3; \end{aligned}$$

since $e_1e_2 = e_1 \wedge e_2 = -e_2 \wedge e_1 = -e_2e_1$,

$$\begin{aligned} uv &= -e_2e_1e_1e_3 + e_1e_3 \\ &= -e_2e_3 + e_1e_3 \\ &= -(e_2 \wedge e_3) + e_1 \wedge e_3. \end{aligned}$$

2.2.3.1 Invertibility

The **inverse** of a blade A is defined by

$$AA^{-1} = 1.$$

The inverse of a vector v, for instance, is

$$v^{-1} = \frac{v}{v \cdot v}.$$

Proof.

$$v\frac{v}{v \cdot v} = \frac{v \cdot v}{v \cdot v} = 1.$$

Example 2.1. The inverse of the vector $v = 2e_1$ results in $0.5e_1$.

Example 2.2. The inverse of the (Euclidean) pseudoscalar $1/I$ is the negative of the pseudoscalar $(-I)$.

Proof.

$$\begin{gathered}
II = (e_1 \wedge e_2 \wedge e_3)(e_1 \wedge e_2 \wedge e_3) = (e_1e_2e_3)(e_1e_2e_3) \\
= \underbrace{e_1e_2}_{-e_2e_1} e_3e_1e_2e_3 = -e_2e_1e_3e_1e_2e_3 = e_2e_3 \underbrace{e_1e_1}_{1} e_2e_3 \\
= e_2e_3e_2e_3 = -e_3e_2e_2e_3 = -e_3e_3 = -1 \\
\rightarrow II = -1 \\
\rightarrow II(I^{-1}) = -I^{-1} \\
\rightarrow I^{-1} = -I.
\end{gathered}$$

2.2.3.2 Duality

Since the geometric product is **invertible**, divisions by algebraic expressions are possible. The **dual** of an algebraic expression is calculated by dividing it by the pseudoscalar I. In the following, the dual of the plane $A = e_2 \wedge (e_1 + e_3)$ is calculated. A superscript * means the dual operator.

$$
\begin{aligned}
(e_2 \wedge (e_1 + e_3))^* &= (e_2 \wedge (e_1 + e_3))(e_1e_2e_3)^{-1} \\
&= (e_2 \wedge (e_1 + e_3))(-e_1e_2e_3) = -(e_2(e_1 + e_3))e_1e_2e_3 \\
&= -e_2 \underbrace{e_1e_1}_{1} e_2e_3 - \underbrace{e_2e_3}_{-e_3e_2} e_1e_2e_3 = -e_2e_2e_3 + e_3e_2e_1e_2e_3 \\
&= -e_3 - e_3e_1e_2e_2e_3 = -e_3 - e_3e_1e_3 \\
&= -e_3 + e_1e_3e_3 = -e_3 + e_1.
\end{aligned}
$$

See [81] for mathematical details.

2.3 Euclidean Geometric Algebra

Euclidean Geometric Algebra includes the well-known vector algebra, and deals with the three Euclidean basis vectors e_1, e_2, e_3. Linear combinations of these basis vectors can be interpreted as 3D vectors or 3D points (a list of all 8 blades of 3D Euclidean Geometric Algebra can be found in Table 2.4). The **scalar product** is identical to the inner product of two vectors. The **cross product** of two Euclidean vectors **u** and **v** can also be written in Geometric Algebra form as

$$\mathbf{u} \times \mathbf{v} = -(\mathbf{u} \wedge \mathbf{v})e_{123}, \tag{2.5}$$

where $e_{123} = e_1 \wedge e_2 \wedge e_3$ is the Euclidean pseudoscalar (a blade of grade 3). In order to prove this equation, we shall calculate first an expression for the outer product of the vectors **u** and **v**:

Table 2.4 The 8 blades of 3D Euclidean Geometric Algebra

Grade	Term	Blades	No.
0	Scalar	1	1
1	Vector	e_1, e_2, e_3	3
2	Bivector	$e_1 \wedge e_2, \quad e_1 \wedge e_3, \quad e_2 \wedge e_3$	3
3	Pseudoscalar	$e_1 \wedge e_2 \wedge e_3$	1

$$
\begin{aligned}
\mathbf{u} \wedge \mathbf{v} &= (u_1e_1 + u_2e_2 + u_3e_3) \wedge (v_1e_1 + v_2e_2 + v_3e_3) \\
&= u_1v_1 \underbrace{(e_1 \wedge e_1)}_{0} + u_1v_2(e_1 \wedge e_2) + u_1v_3(e_1 \wedge e_3) \\
&\quad + u_2v_1(e_2 \wedge e_1) + u_2v_2 \underbrace{(e_2 \wedge e_2)}_{0} + u_2v_3(e_2 \wedge e_3) \\
&\quad + u_3v_1(e_3 \wedge e_1) + u_3v_2(e_3 \wedge e_2) + u_3v_3 \underbrace{(e_3 \wedge e_3)}_{0} \\
&= u_1v_2(e_1 \wedge e_2) + u_1v_3(e_1 \wedge e_3) + u_2v_1(e_2 \wedge e_1) + u_2v_3(e_2 \wedge e_3) \\
&\quad + u_3v_1(e_3 \wedge e_1) + u_3v_2(e_3 \wedge e_2).
\end{aligned}
$$

Using the anticommutativity of the outer product,

$$
\begin{aligned}
\mathbf{u} \wedge \mathbf{v} = u_1v_2(e_1 \wedge e_2) + u_1v_3(e_1 \wedge e_3) - u_2v_1(e_1 \wedge e_2) + u_2v_3(e_2 \wedge e_3) \\
-u_3v_1(e_1 \wedge e_3) - u_3v_2(e_2 \wedge e_3),
\end{aligned}
$$

which leads to the following equation for the outer product of the vectors **u** and **v**:

$$
\mathbf{u} \wedge \mathbf{v} = (u_1v_2 - u_2v_1)(e_1 \wedge e_2) + (u_1v_3 - u_3v_1)(e_1 \wedge e_3) + (u_2v_3 - u_3v_2)(e_2 \wedge e_3). \quad (2.6)
$$

Let us now compute the expression $-(\mathbf{u} \wedge \mathbf{v})e_{123}$:

$$
\begin{aligned}
-(\mathbf{u} \wedge \mathbf{v})e_{123} &= -((u_1v_2 - u_2v_1)e_1e_2 + (u_1v_3 - u_3v_1)e_1e_3 + (u_2v_3 - u_3v_2)e_2e_3)e_{123} \\
&= (u_1v_2 - u_2v_1)e_2e_1e_1e_2e_3 + (u_1v_3 - u_3v_1)e_3e_1e_1e_2e_3 + (u_2v_3 - u_3v_2)e_3e_2e_1e_2e_3 \\
&= (u_1v_2 - u_2v_1)e_2e_2e_3 + (u_1v_3 - u_3v_1)e_3e_2e_3 - (u_2v_3 - u_3v_2)e_3e_1e_2e_2e_3 \\
&= (u_1v_2 - u_2v_1)e_3 - (u_1v_3 - u_3v_1)e_2e_3e_3 - (u_2v_3 - u_3v_2)e_3e_1e_3 \\
&= (u_1v_2 - u_2v_1)e_3 - (u_1v_3 - u_3v_1)e_2 + (u_2v_3 - u_3v_2)e_3e_3e_1 \\
\mathbf{u} \wedge \mathbf{v} &= (u_1v_2 - u_2v_1)e_3 - (u_1v_3 - u_3v_1)e_2 + (u_2v_3 - u_3v_2)e_1
\end{aligned}
$$

leading to the equation

$$
-(\mathbf{u} \wedge \mathbf{v})e_{123} = (u_2v_3 - u_3v_2)e_1 - (u_1v_3 - u_3v_1)e_2 + (u_1v_2 - u_2v_1)e_3, \quad (2.7)
$$

with the righthand side equal to the definition of the cross product

$$
\mathbf{u} \times \mathbf{v} = (u_2v_3 - u_3v_2)e_1 - (u_1v_3 - u_3v_1)e_2 + (u_1v_2 - u_2v_1)e_3. \quad (2.8)
$$

2.4 Projective Geometric Algebra

Projective Geometric Algebra is a 4D Geometric Algebra; its 16 blades are listed in Table 2.5.

An **inhomogeneous** point

$$\mathbf{x} = x_1 e_1 + x_2 e_2 + x_3 e_3 \tag{2.9}$$

is transformed into a **homogeneous** point via

$$X = \mathbf{x} + e_0. \tag{2.10}$$

The origin is mapped to e_0 by this mapping. Vice versa, an arbitrary homogeneous point

$$X = wx_1 e_1 + wx_2 e_2 + wx_3 e_3 + we_0, \quad w \neq 0 \tag{2.11}$$

can first be scaled to the hyperplane

$$X' = x_1 e_1 + x_2 e_2 + x_3 e_3 + e_0 \tag{2.12}$$

and then projected to the inhomogeneous point $\mathbf{x} = x_1 e_1 + x_2 e_2 + x_3 e_3$.

Further details can be found in Sect. 4.2 of [81].

Table 2.5 The 16 blades of 4D projective Geometric Algebra

Grade	Term	Blades	No.
0	Scalar	1	1
1	Vector	e_1, e_2, e_3, e_0	4
2	Bivector	$e_1 \wedge e_2, \quad e_1 \wedge e_3, \quad e_1 \wedge e_0,$ $e_2 \wedge e_3, \quad e_2 \wedge e_0, \quad e_3 \wedge e_0$	6
3	Trivector	$e_1 \wedge e_2 \wedge e_3, \quad e_1 \wedge e_2 \wedge e_0,$ $e_1 \wedge e_3 \wedge e_0, \quad e_2 \wedge e_3 \wedge e_0$	4
4	Pseudoscalar	$e_1 \wedge e_2 \wedge e_3 \wedge e_0$	1

Chapter 3
Conformal Geometric Algebra

In this book, we focus on 5D Conformal Geometric Algebra (CGA). The "conformal" comes from the fact that it handles conformal transformations easily. These transformations leave angles invariant.

You may ask why you should use a 5D Geometric Algebra if your problem is from the 3D real world? One reason is that problems can often be formulated more easily and intuitively in a higher number of dimensions (Fig. 3.1). One advantage of CGA, for instance, is that points, spheres and planes are easily represented as vectors (linear combination of blades of grade 1).

CGA uses the three Euclidean basis vectors e_1, e_2, e_3 and two additional basis vectors e_+, e_- with positive and negative signatures, respectively, which means that they square to $+1$ as usual (e_+) and to -1 (e_-).

$$e_+^2 = 1, \qquad e_-^2 = -1, \qquad e_+ \cdot e_- = 0. \tag{3.1}$$

Another basis e_0, e_∞, with the geometric meaning

- e_0 represents the 3D origin,
- e_∞ represents infinity,

can be defined with the relations

$$e_0 = \frac{1}{2}(e_- - e_+), \qquad e_\infty = e_- + e_+. \tag{3.2}$$

These new basis vectors are null vectors:

$$e_0^2 = e_\infty^2 = 0. \tag{3.3}$$

Taking their inner product results in

$$e_\infty \cdot e_0 = -1, \tag{3.4}$$

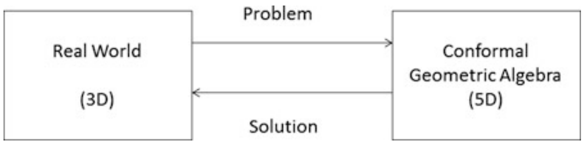


Fig. 3.1 Why 5D Conformal Geometric Algebra for 3D world problems?

Table 3.1 The 32 blades of the 5D Conformal Geometric Algebra (CGA)

Grade	Term	Blades	No.
0	Scalar	1	1
1	Vector	$e_1, e_2, e_3, e_\infty, e_0$	5
2	Bivector	$e_1 \wedge e_2,\ e_1 \wedge e_3,\ e_1 \wedge e_\infty,$ $e_1 \wedge e_0,\ e_2 \wedge e_3,\ e_2 \wedge e_\infty,$ $e_2 \wedge e_0,\ e_3 \wedge e_\infty,\ e_3 \wedge e_0,$ $e_\infty \wedge e_0$	10
3	Trivector	$e_1 \wedge e_2 \wedge e_3,\ e_1 \wedge e_2 \wedge e_\infty,\ e_1 \wedge e_2 \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty,\ e_1 \wedge e_3 \wedge e_0,\ e_1 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty,\ e_2 \wedge e_3 \wedge e_0,\ e_2 \wedge e_\infty \wedge e_0,$ $e_3 \wedge e_\infty \wedge e_0$	10
4	Quadvector	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty,$ $e_1 \wedge e_2 \wedge e_3 \wedge e_0,$ $e_1 \wedge e_2 \wedge e_\infty \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5
5	Pseudoscalar	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	1

since

$$(e_- + e_+) \cdot \frac{1}{2}(e_- - e_+) = \frac{1}{2}(\underbrace{e_- \cdot e_-}_{-1} - \underbrace{e_- \cdot e_+}_{0} + \underbrace{e_+ \cdot e_-}_{0} - \underbrace{e_+ \cdot e_+}_{1}),$$

and their geometric product is

$$e_\infty e_0 = e_\infty \wedge e_0 + e_\infty \cdot e_0 = e_\infty \wedge e_0 - 1 \tag{3.5}$$

or

$$e_0 e_\infty = e_0 \wedge e_\infty + e_\infty \cdot e_0 = -e_\infty \wedge e_0 - 1. \tag{3.6}$$

The outer product $e_\infty \wedge e_0$ is often abbreviated as E.

A list of all 32 blades of 5D CGA can be found in Table 3.1.

3.1 The Basic Geometric Entities

CGA provides a great variety of basic geometric entities to compute with, namely points, spheres, planes, circles, lines, and point pairs, as listed in Table 3.2. These entities have two algebraic representations: the IPNS (inner product null space)

Table 3.2 The two representations (IPNS and OPNS) of conformal geometric entities. The IPNS and OPNS representations are dual to each other, which is indicated by the asterisk symbol

Entity	IPNS representation	OPNS representation
Point	$P = \mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0$	
Sphere	$S = P - \frac{1}{2}r^2 e_\infty$	$S^* = P_1 \wedge P_2 \wedge P_3 \wedge P_4$
Plane	$\pi = \mathbf{n} + d e_\infty$	$\pi^* = P_1 \wedge P_2 \wedge P_3 \wedge e_\infty$
Circle	$Z = S_1 \wedge S_2$	$Z^* = P_1 \wedge P_2 \wedge P_3$
Line	$L = \pi_1 \wedge \pi_2$	$L^* = P_1 \wedge P_2 \wedge e_\infty$
Point pair	$Pp = S_1 \wedge S_2 \wedge S_3$	$Pp^* = P_1 \wedge P_2$

and the OPNS (outer product null space). These representations are duals of each other (a superscript asterisk denotes the dualization operator). See Sect. 3.2 for more details of the IPNS and OPNS.

In Table 3.2, **x** and **n** are in bold type to indicate that they represent 3D entities obtained by linear combinations of the 3D basis vectors e_1, e_2, and e_3:

$$\mathbf{x} = x_1 e_1 + x_2 e_2 + x_3 e_3. \tag{3.7}$$

The $\{S_i\}$ represent different spheres, and the $\{\pi_i\}$ represent different planes. In the OPNS representation, the outer product "$\wedge$" indicates the construction of a geometric object with the help of points $\{P_i\}$ that lie on it. A sphere, for instance, is defined by four points ($P_1 \wedge P_2 \wedge P_3 \wedge P_4$) on this sphere. In the IPNS representation, the meaning of the outer product is an intersection of geometric entities. A circle, for instance, is defined by the intersection of two spheres $S_1 \wedge S_2$ (see Fig. 6.18). In the following, we present the representations of all of the basic geometric entities. For details, see [81], especially Sects. 4.3.4 and 4.3.5.

3.1.1 Points

In order to represent points in 5D conformal space, the original 3D point **x** is projectively extended to a 5D vector by taking linear combinations of the 5D basis vectors e_1, e_2, e_3, e_∞, and e_0 according to the equation

$$P = \mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0, \tag{3.8}$$

where $\mathbf{x}^2$ is the well-known scalar product

$$\mathbf{x}^2 = x_1^2 + x_2^2 + x_3^2. \tag{3.9}$$

For example, for the 3D origin (0,0,0) we get

$$P(0,0,0) = e_0, \tag{3.10}$$

and for the 3D point (0,1,0),

$$P_y = P(0,1,0) = e_2 + \frac{1}{2}e_\infty + e_0. \tag{3.11}$$

3.1.2 Spheres

A sphere can, on the one hand, be represented with the help of its center point P and its radius r as

$$S = P - \frac{1}{2}r^2 e_\infty \tag{3.12}$$

or, using (3.8), as

$$S = \mathbf{x} + \frac{1}{2}(\mathbf{x}^2 - r^2)e_\infty + e_0. \tag{3.13}$$

Note that the representation of a point is simply that of a sphere of radius zero. A sphere can also, on the other hand, be represented with the help of four points that lie on it:

$$S^* = P_1 \wedge P_2 \wedge P_3 \wedge P_4. \tag{3.14}$$

3.1.3 Planes

A plane is defined by

$$\pi = \mathbf{n} + d e_\infty, \tag{3.15}$$

where $\mathbf{n}$ refers to the 3D normal vector of the plane π and d is the distance to the origin. A plane can also be defined with the help of three points that lie on it and the point at infinity:

$$\pi^* = P_1 \wedge P_2 \wedge P_3 \wedge e_\infty. \tag{3.16}$$

Note that a plane is a sphere of infinite radius (details will be given in Sect. 5.1).

3.1.4 Circles

A circle is defined by the intersection of two spheres

$$Z = S_1 \wedge S_2 \tag{3.17}$$

or, with the help of three points that lie on it, by

$$Z^* = P_1 \wedge P_2 \wedge P_3. \tag{3.18}$$

3.1.5 Lines

A line is defined by the intersection of two planes

$$L = \pi_1 \wedge \pi_2 \tag{3.19}$$

or, with the help of two points that lie on it and the point at infinity, by

$$L^* = P_1 \wedge P_2 \wedge e_\infty. \tag{3.20}$$

For example, the y-axis L_y can be described by

$$L_y^* = e_0 \wedge P_y \wedge e_\infty, \tag{3.21}$$

where e_0 represents the origin (see (3.10)) and P_y is the point described by (3.11). Note that a line can be regarded as a circle of infinite radius.

3.1.6 Point Pairs

A point pair is defined by the intersection of three spheres

$$Pp = S_1 \wedge S_2 \wedge S_3 \tag{3.22}$$

or, directly with the help of the two points, by

$$Pp^* = P_1 \wedge P_2. \tag{3.23}$$

We can use the following formula to extract the two points of a point pair Pp [29, 38]:

$$P_\pm = \frac{\pm\sqrt{Pp^* \cdot Pp^*} + Pp^*}{e_\infty \cdot Pp^*}. \tag{3.24}$$

3.2 IPNS and OPNS

The IPNS and OPNS describe the null spaces of algebraic expressions with respect to the inner and outer products, respectively. The IPNS of e_0, for instance, describes all the points P satisfying the equation

$$e_0 \cdot P = 0, \tag{3.25}$$

which gives

$$e_0 \cdot \left(\mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0 \right) = 0. \tag{3.26}$$

With the identity $e_0 \cdot e_\infty = -1$, this is equal to

$$-\frac{1}{2}\mathbf{x}^2 = 0 \tag{3.27}$$

or

$$x_1^2 + x_2^2 + x_3^2 = 0, \tag{3.28}$$

describing exactly the point at the origin.

Let us look at what happens if we subtract some "amount of infinity". What, for instance, does $e_0 - \frac{1}{2}r^2 e_\infty$ mean?

We can compute the IPNS of the expression

$$\left(e_0 - \frac{1}{2}r^2 e_\infty \right) \cdot P = 0, \tag{3.29}$$

which is equal to

$$-\frac{1}{2}\mathbf{x}^2 + \frac{1}{2}r^2 = 0 \tag{3.30}$$

or

$$x_1^2 + x_2^2 + x_3^2 - r^2 = 0, \tag{3.31}$$

describing all points at the same distance r from the origin, namely a sphere at the origin.

For more details, see Chap. 4 of [81] (especially Sect. 4.3.3).

3.3 The Center of a Sphere, Circle, or Point Pair

The center of a sphere can be computed as the following *sandwich* product:

$$P = S e_\infty S. \tag{3.32}$$

This can be proved based on the following steps:

$$P = \left(\mathbf{x} + \frac{1}{2}(\mathbf{x}^2 - r^2) e_\infty + e_0 \right) e_\infty S, \tag{3.33}$$

$$P = \left(\left(\mathbf{x} + \frac{1}{2}(\mathbf{x}^2 - r^2) e_\infty + e_0 \right) e_\infty \right) S, \tag{3.34}$$

$$P = (\mathbf{x} \wedge e_\infty + e_0 \wedge e_\infty - 1)\, S, \tag{3.35}$$

$$P = (\mathbf{x} \wedge e_\infty + e_0 \wedge e_\infty - 1) \left(\mathbf{x} + \frac{1}{2}(\mathbf{x}^2 - r^2)e_\infty + e_0 \right), \tag{3.36}$$

$$\begin{aligned} P &= (\mathbf{x} \wedge e_\infty + e_0 \wedge e_\infty - 1)\,\mathbf{x} \\ &+ (\mathbf{x} \wedge e_\infty + e_0 \wedge e_\infty - 1)\,\frac{1}{2}(\mathbf{x}^2 - r^2)e_\infty \\ &+ (\mathbf{x} \wedge e_\infty + e_0 \wedge e_\infty - 1)\,e_0, \end{aligned} \tag{3.37}$$

$$\begin{aligned} P &= (\mathbf{x} \wedge e_\infty + e_0 \wedge e_\infty - 1)\,\mathbf{x} \\ &+ (\mathbf{x} \wedge e_\infty + e_0 \wedge e_\infty - 1)\,e_0, \end{aligned} \tag{3.38}$$

$$\begin{aligned} P &= -\mathbf{x}^2 e_\infty + \mathbf{x} \wedge e_0 \wedge e_\infty - \mathbf{x} \\ &+ \mathbf{x} \wedge e_\infty \wedge e_0 - \mathbf{x} - 2e_0, \end{aligned} \tag{3.39}$$

leading to the center point of the sphere

$$P = -2\left(\mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0 \right) \tag{3.40}$$

with a homogeneous scaling factor of -2.

The sandwich product of (3.32) can also be used to obtain the centers of circles and point pairs. Note, that circles and point pairs are specific lower-dimensional spheres in CGA. The center of a circle, for instance, can be computed from

$$P = Z e_\infty Z. \tag{3.41}$$

This is used in the robot example of Chap. 8.

3.4 Distances and Angles

In CGA points, planes and spheres are represented as vectors.

Table 3.3 summarizes and details the results of Sect. 3.1. We will now investigate the inner product of conformal vectors and its geometric meaning. Table 3.4 summarizes the geometric meaning of the inner product of two conformal vectors U and V as outlined in Sect. 3.4.1 for distances and in Sect. 3.4.2 for angles.

Table 3.3 Geometric meaning of conformal vectors

Point	$P = \mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0$	$= x_1e_1 + x_2e_2 + x_3e_3 + \frac{1}{2}(x_1^2 + x_2^2 + x_3^2)e_\infty + e_0$
Sphere	$S = P - \frac{1}{2}r^2 e_\infty$	$= s_1e_1 + s_2e_2 + s_3e_3 + \frac{1}{2}(s_1^2 + s_2^2 + s_3^2 - r^2)e_\infty + e_0$
Plane	$\pi = \mathbf{n} + de_\infty$	$= n_1e_1 + n_2e_2 + n_3e_3 + de_\infty$

Table 3.4 Geometric meaning of the inner product of two conformal vectors U and V

$U \cdot V$	Plane	Sphere	Point
Plane	Angle between planes	Euclidean distance from center	Euclidean distance
Sphere	Euclidean distance from center	Distance measure	Distance measure
Point	Euclidean distance	Distance measure	Euclidean distance

3.4.1 Distances

In CGA, points, planes, and spheres are represented as vectors. The inner product of this kind of object is a scalar and can be used as a measure of distances. In the following examples, we will see that the inner product $P \cdot S$ of two vectors P and S can be used for tasks such as

- The Euclidean distance between two points;
- The distance between a point and a plane;
- The decision as to whether a point is inside or outside a sphere.

The inner product of a vector P and a vector S is defined by

$$\begin{aligned} P \cdot S &= (\mathbf{p} + p_4 e_\infty + p_5 e_0) \cdot (\mathbf{s} + s_4 e_\infty + s_5 e_0) \\ &= \mathbf{p} \cdot \mathbf{s} + s_4 \underbrace{\mathbf{p} \cdot e_\infty}_{0} + s_5 \underbrace{\mathbf{p} \cdot e_0}_{0} \\ &\quad + p_4 \underbrace{e_\infty \cdot \mathbf{s}}_{0} + p_4 s_4 \underbrace{e_\infty^2}_{0} + p_4 s_5 \underbrace{e_\infty \cdot e_0}_{-1} \\ &\quad + p_5 \underbrace{e_0 \cdot \mathbf{s}}_{0} + p_5 s_4 \underbrace{e_0 \cdot e_\infty}_{-1} + p_5 s_5 \underbrace{e_0^2}_{0} . \end{aligned}$$

This results in

$$P \cdot S = \mathbf{p} \cdot \mathbf{s} - p_5 s_4 - p_4 s_5 \tag{3.42}$$

or

$$P \cdot S = p_1 s_1 + p_2 s_2 + p_3 s_3 - p_5 s_4 - p_4 s_5 .$$

3.4.1.1 Distance Between Points

In the case of P and S being points we get

$$p_4 = \frac{1}{2}\mathbf{p}^2, p_5 = 1$$

$$s_4 = \frac{1}{2}\mathbf{s}^2, s_5 = 1$$

The inner product of these points is according to Eq. 3.42

$$\begin{aligned}
P \cdot S &= \mathbf{p} \cdot \mathbf{s} - \frac{1}{2}\mathbf{s}^2 - \frac{1}{2}\mathbf{p}^2 \\
&= p_1 s_1 + p_2 s_2 + p_3 s_3 - \frac{1}{2}(s_1^2 + s_2^2 + s_3^2) - \frac{1}{2}(p_1^2 + p_2^2 + p_3^2) \\
&= -\frac{1}{2}(s_1^2 + s_2^2 + s_3^2 + p_1^2 + p_2^2 + p_3^2 - 2p_1 s_1 - 2p_2 s_2 - 2p_3 s_3) \\
&= -\frac{1}{2}((s_1 - p_1)^2 + (s_2 - p_2)^2 + (s_3 - p_3)^2) \\
&= -\frac{1}{2}(\mathbf{s} - \mathbf{p})^2
\end{aligned}$$

We recognize that the square of the Euclidean distance of the inhomogeneous points corresponds to the inner product of the homogeneous points multiplied by -2.

$$(\mathbf{s} - \mathbf{p})^2 = -2(P \cdot S)$$

3.4.1.2 Distance Between a Point and a Plane

For a vector P representing a point, we get

$$p_4 = \frac{1}{2}\mathbf{p}^2, \quad p_5 = 1.$$

For a vector S representing a plane with normal vector $\mathbf{n}$ and distance d, we get

$$\mathbf{s} = \mathbf{n}, \quad s_4 = d, \quad s_5 = 0.$$

The inner product of point and plane is, according to (3.42),

$$P \cdot S = \mathbf{p} \cdot \mathbf{n} - d,$$

which represents the Euclidean distance between the point and the plane, with a sign according to

- $P \cdot S > 0$: $\mathbf{p}$ is in the direction of the normal $\mathbf{n}$;
- $P \cdot S = 0$: $\mathbf{p}$ is on the plane;
- $P \cdot S < 0$: $\mathbf{p}$ is not in the direction of the normal $\mathbf{n}$.

3.4.1.3 Distance Between a Plane and a Sphere

For a vector U representing a plane with normal vector $\mathbf{n}$ and distance d, we get

$$\mathbf{u} = \mathbf{n}, \quad u_4 = d, \quad u_5 = 0.$$

For a vector V representing a sphere, we get

$$v_4 = \frac{1}{2}(\mathbf{s}^2 - r^2), \quad v_5 = 1.$$

The inner product of the plane and the sphere is, according to (3.42),

$$U \cdot V = \pi \cdot S = \mathbf{n} \cdot \mathbf{s} - d, \tag{3.43}$$

which represents the Euclidean distance between the center point of the sphere and the plane (see Sect. 3.4.1.2).

3.4.1.4 Distance Between Two Spheres

We will now compute the inner product of two spheres.

For two vectors S_1 and S_2 representing two spheres, we get

$$u_4 = \frac{1}{2}(\mathbf{s_1}^2 - r_1^2), \quad u_5 = 1,$$

and

$$v_4 = \frac{1}{2}(\mathbf{s_2}^2 - r_2^2), \quad v_5 = 1.$$

The inner product of the two spheres is, according to (3.42),

$$\begin{aligned}
S_1 \cdot S_2 &= \mathbf{s_1} \cdot \mathbf{s_2} - \frac{1}{2}(\mathbf{s_2}^2 - r_2^2) - \frac{1}{2}(\mathbf{s_1}^2 - r_1^2) \\
&= \mathbf{s_1} \cdot \mathbf{s_2} - \frac{1}{2}\mathbf{s_2}^2 + \frac{1}{2}r_2^2 - \frac{1}{2}\mathbf{s_1}^2 + \frac{1}{2}r_1^2 \\
&= \frac{1}{2}r_1^2 + \frac{1}{2}r_2^2 - \frac{1}{2}(\mathbf{s_2}^2 - 2\mathbf{s_1} \cdot \mathbf{s_2} + \mathbf{s_1}^2) \\
&= \frac{1}{2}(r_1^2 + r_2^2) - \frac{1}{2}(\mathbf{s_2} - \mathbf{s_1})^2.
\end{aligned}$$

We get

$$2(S_1 \cdot S_2) = r_1^2 + r_2^2 - (\mathbf{s_2} - \mathbf{s_1})^2. \tag{3.44}$$

This means that twice the inner product of two spheres is equal to the sum of the squares of their radii minus the square of the Euclidean distance between the centers of the spheres.

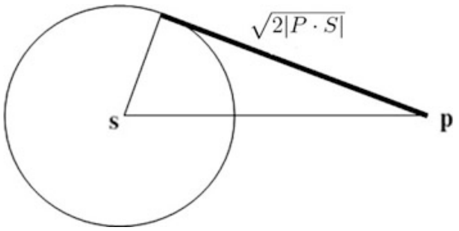


Fig. 3.2 The inner product of a point and a sphere describes the square of the distance between the point and sphere according to (3.46)

3.4.1.5 The Inner Product of a Point and a Sphere

We will now see that the inner product of a point and a sphere can be used to decide whether a point is inside a sphere or not.

For a vector P representing a point, we get

$$p_4 = \frac{1}{2}\mathbf{p}^2, \quad p_5 = 1.$$

For a vector S representing a sphere with radius r, we get

$$s_4 = \frac{1}{2}(s_1^2 + s_2^2 + s_3^2 - r^2), \quad s_5 = 1.$$

The inner product of the point and sphere is, according to 3.42,

$$\begin{aligned} P \cdot S &= \mathbf{p} \cdot \mathbf{s} - \frac{1}{2}(\mathbf{s}^2 - r^2) - \frac{1}{2}\mathbf{p}^2 \\ &= \mathbf{p} \cdot \mathbf{s} - \frac{1}{2}\mathbf{s}^2 + \frac{1}{2}r^2 - \frac{1}{2}\mathbf{p}^2 \\ &= \frac{1}{2}r^2 - \frac{1}{2}(\mathbf{s}^2 - 2\mathbf{p} \cdot \mathbf{s} - \mathbf{p}^2) \\ &= \frac{1}{2}r^2 - \frac{1}{2}(\mathbf{s} - \mathbf{p})^2. \end{aligned}$$

We get

$$2(P \cdot S) = r^2 - (\mathbf{s} - \mathbf{p})^2 \tag{3.45}$$

or

$$-2(P \cdot S) = (\mathbf{s} - \mathbf{p})^2 - r^2. \tag{3.46}$$

Figure 3.2 illustrates this formula. The triangle shown is right-angled. According to Pythagoras' theorem, $\sqrt{2|P \cdot S|}$ is equal to the distance between **p** and the tangent point to the sphere.

3.4.1.6 Is a Point Inside or Outside a Sphere?

In terms of the Euclidean distance d, where

$$(d + r)^2 = (\mathbf{s} - \mathbf{p})^2 = d^2 + 2dr + r^2,$$

we get

$$2(P \cdot S) = r^2 - (d^2 + 2dr + r^2),$$
$$2(P \cdot S) = -d^2 - 2dr,$$

or

$$P \cdot S = D(d) = -\frac{d}{2}(d + 2r).$$

With the help of some curve sketching, we can see that this is a parabola with

$$D(0) = 0, \quad D(-2r) = 0,$$

and a maximum at

$$D(-r) = \frac{1}{2}r^2.$$

We can now see that

- If $P \cdot S > 0$, $\mathbf{p}$ is inside the sphere;
- If $P \cdot S = 0$, $\mathbf{p}$ is on the sphere;
- If $P \cdot S < 0$, $\mathbf{p}$ is outside the sphere.

3.4.2 Angles

This section presents some relations between the inner products of geometric objects and their angles.

Angles between two objects o_1, o_2 such as two lines or two planes can be computed using the inner product of the normalized OPNS representation of the objects:

$$\cos(\theta) = \frac{o_1^* \cdot o_2^*}{|o_1^*|\,|o_2^*|} \tag{3.47}$$

or

$$\theta = \angle(o_1, o_2) = \arccos \frac{o_1^* \cdot o_2^*}{|o_1^*|\,|o_2^*|}. \tag{3.48}$$

See [63] for more details.

Let us derive, as one example, an expression for the angle between two planes based on the observation expressed in (3.42). For a vector π_1 representing a plane with normal vector $\mathbf{n_1}$ and distance d_1, we get

$$\mathbf{u} = \mathbf{n_1}, \quad u_4 = d_1, \quad u_5 = 0.$$

For a vector π_2 representing another plane, we get

$$\mathbf{v} = \mathbf{n_2}, \quad v_4 = d_2, \quad v_5 = 0.$$

The inner product of the two planes is

$$\pi_1 \cdot \pi_2 = \mathbf{n_1} \cdot \mathbf{n_2}, \tag{3.49}$$

representing the scalar product of the two normals of the planes. Based on this observation, the angle θ between the two planes can be computed as follows:

$$\cos(\theta) = \pi_1 \cdot \pi_2. \tag{3.50}$$

This corresponds to (3.48) taking into account the fact that the planes are normalized and that the dualization operation switches between the two possible angles between planes.

3.5 Transformations

Transformations can easily be described in CGA. All kinds of transformations of an object o can be done with the help of the following geometric product:

$$o_{transformed} = V o \tilde{V}, \tag{3.51}$$

where V is a **versor** and $\tilde{V}$ is its reverse. (In the reverse of a multivector the blades are with reversed order of their outer product components, for instance the reverse of $1 + e_1 \wedge e_2$ is equal to $1 + e_2 \wedge e_1$ or $1 - e_1 \wedge e_2$).

We focus here on rotations and translations; see Sect. 6.4 of [7] for a treatment of other conformal transformations such as reflections, dilations, and involutions. Readers interested in the mathematical details of how transformations such as rotations, and translations are represented by combinations of reflections are referred to Sects. 1.5.4 and 4.3 of [81]. Section 3.3 of that book describes versors in some detail.

3.5.1 Rotation

The operator

$$R = e^{-(\frac{\phi}{2})L} \tag{3.52}$$

describes a **rotor**. L is the rotation axis, represented by a normalized bivector, and ϕ is the rotation angle around this axis. R can also be written as

$$R = \cos\left(\frac{\phi}{2}\right) - L\sin\left(\frac{\phi}{2}\right). \tag{3.53}$$

The rotation of a geometric object o is performed with the help of the operation

$$o_{rotated} = Ro\tilde{R}.$$

There are strong relations between rotations in CGA and quaternions and dual quaternions. In Chap. 4 you will see that quaternions are in principle rotors with a rotation axis through the origin, and dual quaternions turn out to be versors describing rotations with an arbitrary rotation axis, not passing through the origin.

3.5.2 *Translation*

In CGA, a translation can be expressed in a multiplicative way with the help of a **translator** T defined by

$$T = e^{-\frac{1}{2}\mathbf{t}e_\infty}, \tag{3.54}$$

where $\mathbf{t}$ is a vector

$$\mathbf{t} = t_1e_1 + t_2e_2 + t_3e_3.$$

Application of the Taylor series

$$T = e^{-\frac{1}{2}\mathbf{t}e_\infty} = 1 + \frac{-\frac{1}{2}\mathbf{t}e_\infty}{1!} + \frac{(-\frac{1}{2}\mathbf{t}e_\infty)^2}{2!} + \frac{(-\frac{1}{2}\mathbf{t}e_\infty)^3}{3!} + \dots$$

and the property $(e_\infty)^2 = 0$ results in the translator

$$T = 1 - \frac{1}{2}\mathbf{t}e_\infty. \tag{3.55}$$

Example. Versors do not only transform points but are also able to transform complete geometric objects. Let us, for instance, translate the sphere

$$S = -e_\infty + e_0 \tag{3.56}$$

(see Fig. 3.3) in the x-direction by the translation vector

$$\mathbf{t} = 4e_1. \tag{3.57}$$

Note that, according to (3.13), this is a sphere with its center at the origin and with $r^2 = 2$.

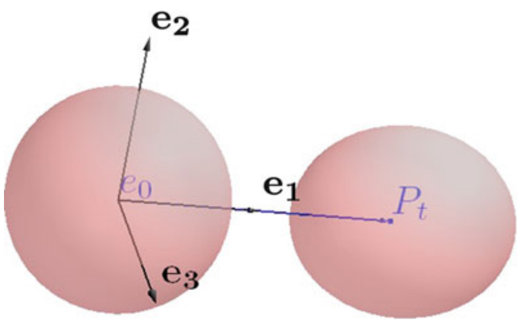


Fig. 3.3 Translation of a sphere from the origin to the point P_t

The translator in this example has the form

$$T = 1 - 2e_1 e_\infty, \tag{3.58}$$

and its reverse is

$$\tilde{T} = 1 + 2e_1 e_\infty. \tag{3.59}$$

The translated sphere can now be computed as the versor product

$$S_{translated} = TS\tilde{T} \tag{3.60}$$

$$= (1 - 2e_1 e_\infty)(-e_\infty + e_0)(1 + 2e_1 e_\infty)$$

$$= (1 - 2e_1 e_\infty)(-e_\infty - 2\underbrace{e_\infty e_1 e_\infty}_{0} + e_0 + 2e_0 e_1 e_\infty)$$

$$= (1 - 2e_1 e_\infty)(-e_\infty + e_0 - 2e_1 e_0 e_\infty)$$

$$= -e_\infty + e_0 - 2e_1 e_0 e_\infty + 2e_1 \underbrace{e_\infty e_\infty}_{0} - 2e_1 e_\infty e_0 + 4e_1 e_\infty e_1 e_0 e_\infty$$

$$= -e_\infty + e_0 - 2e_1 \underbrace{(e_0 e_\infty + e_\infty e_0)}_{-2} + 4e_1 e_\infty e_1 e_0 e_\infty$$

$$= 4e_1 - e_\infty + e_0 + 4\underbrace{e_1 e_\infty e_1}_{-e_\infty} e_0 e_\infty$$

$$= 4e_1 - e_\infty + e_0 - 4e_\infty \underbrace{e_0 e_\infty}_{-e_\infty \wedge e_0 - 1}$$

$$= 4e_1 - e_\infty + e_0 - 4\underbrace{e_\infty(-e_\infty \wedge e_0 - 1)}_{-2e_\infty},$$

resulting in

$$S_{translated} = 4e_1 + 7e_\infty + e_0. \tag{3.61}$$

According to (3.13), this is a sphere with the same radius $r^2 = 2$, but with a translated center point

$$P_t = \mathbf{t} + \frac{1}{2}\mathbf{t}^2 e_\infty + e_0 = 4e_1 + 8e_\infty + e_0. \tag{3.62}$$

3.6 Rigid-Body Motion

In CGA, a rigid-body motion, including both a rotation and a translation, is described by a **displacement versor** D, sometimes also called a **motor**,

$$D = RT, \tag{3.63}$$

where R is a rotor and T is a translator (see Sect. 3.5). A rigid-body motion of an object o is described by

$$o_{rigid\_body\_motion} = Do\tilde{D}.$$

We see, that both rotation and translation can be expressed in one algebraic expression. In CGA, this is also possible for rotational and translational kinematics and dynamics.

If we consider a time-dependent displacement versor $D(t)$, its differentiation leads to

$$\dot{D} = \frac{1}{2}VD. \tag{3.64}$$

The velocity screw V has the form

$$V = e_\infty \mathbf{v} - e_{123}\boldsymbol{!}, \tag{3.65}$$

where $\mathbf{v}$ is the linear velocity vector, e_{123} is the Euclidean pseudoscalar $e_1 \wedge e_2 \wedge e_3$, and $\boldsymbol{!}$ is the angular (or rotational) velocity. Its differentiation leads to a combined acceleration expression consisting of both a force and a torque. See [50] for details of the treatment of kinematics and dynamics with CGA.

All of these equations are used in our molecular dynamics application in Chap. 13.

3.7 The Horizon Example

This section presents an example that will be used throughout this book for the explanation of various topics. Given a sphere S describing the Earth and a viewpoint P of an observer on a beach (see Fig. 3.4), we wish to find an algebraic expression for the horizon as seen by the observer, provided there is no occlusion of any sort other than the Earth itself in the scene.

Fig. 3.4 Horizon of an observer on a beach

First of all, we compute the distance from the viewpoint to the horizon. According to (3.46) and Fig. 3.2, the square of this distance d can easily be computed based on the inner product of the Earth and the viewpoint:

$$d^2 = -2(S \cdot P). \tag{3.66}$$

Given this squared distance, we may construct another sphere K around P with that distance as its radius ($r^2 = d^2$):

$$K = P - \frac{1}{2}d^2 e_\infty \tag{3.67}$$

or

$$K = P + (S \cdot P)e_\infty. \tag{3.68}$$

The circle representing the horizon may then be calculated by calculating the intersection (via the outer product) of the two spheres. Figure 3.5 illustrates the calculation:

$$C = S \wedge K \tag{3.69}$$

or

$$C = S \wedge (P + (S \cdot P)e_\infty). \tag{3.70}$$

This is a very compact description of the horizon circle C seen from the viewpoint P on the Earth S. Note that we did not need any coordinates for its formulation.

Only if we have to make concrete computations do we have to use coordinates. Let m_x, m_y, m_z be the 3D coordinates of the Earth's center; then M has the following representation in 5D conformal space:

$$M = m_x e_1 + m_y e_2 + m_z e_3 + \frac{1}{2}\mathbf{m}^2 e_\infty + e_0, \tag{3.71}$$

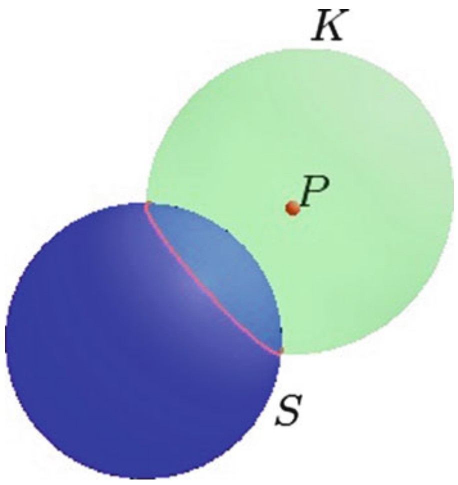


Fig. 3.5 Calculation of the intersection circle (horizon)

where

$$\mathbf{m}^2 = m_x^2 + m_y^2 + m_z^2. \tag{3.72}$$

The viewpoint P of an observer with coordinates p_x, p_y, p_z can be represented accordingly as

$$P = p_x e_1 + p_y e_2 + p_z e_3 + \frac{1}{2}\mathbf{p}^2 e_\infty + e_0, \tag{3.73}$$

where

$$\mathbf{p}^2 = p_x^2 + p_y^2 + p_z^2, \tag{3.74}$$

and the sphere S describing the Earth, with center point M and radius r, can be represented as

$$S = M - \frac{1}{2}r^2 e_\infty. \tag{3.75}$$

Assuming the center of the sphere to be at the origin of the coordinate system, S can also be defined as

$$S = e_0 - \frac{1}{2}r^2 e_\infty. \tag{3.76}$$

The horizon example will be used throughout this book in order to explain several different topics. We use it especially to explain the software tools **CLUCalc** and **Gaalop** in the following sections:

- Section 6.5: CLUCalc, a tool for the interactive and visual development of Geometric Algebra algorithms
- Section 10.1: the Gaalop compiler for high-performance implementations;
- Section 11.2: Gaalop GPC for C++ programs;
- Section 12.2.1: Gaalop GPC for OpenCL programs;
- Section 12.2.2: Gaalop GPC for CUDA programs.

Chapter 4
Maple and the Identification of Quaternions and Other Algebras

The goal of this chapter is to identify some mathematical systems in CGA and to investigate what their geometric meaning is (Fig. 4.1).

Chapter 3 showed how geometric objects can be represented as algebraic expressions.

Figure 1.2 indicates that spheres and planes can be represented as vectors, and circles and lines as bivectors in CGA. We will see in this chapter, that, for instance, quaternions can be identified based on their imaginary units i, j, k. This is the reason why transformations such as rotations can be handled within the algebra, although they are traditionally handled based on separate rotation matrices.

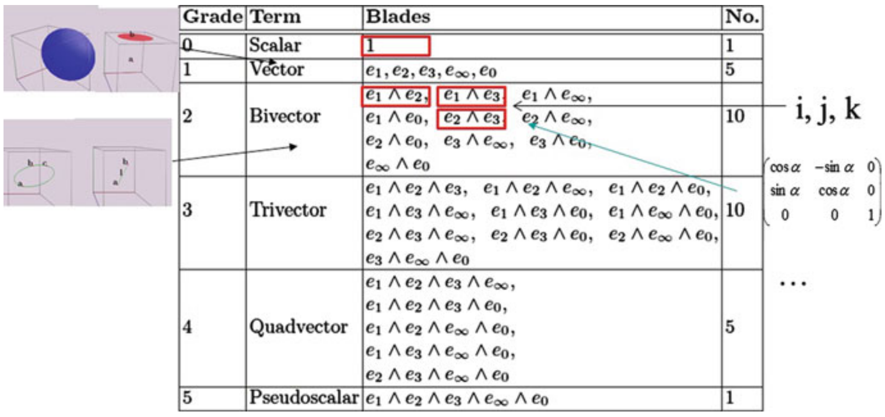
Besides quaternions, we will see the identification of other mathematical systems in CGA such as complex numbers (see Fig. 4.2), Plücker coordinates (see Fig. 4.7) and dual quaternions (see Fig. 4.11). Many of the calculations presented in this chapter are supported by Maple.

4.1 Using Maple for Symbolic Geometric Algebra Computing

In order to deal with symbolic Geometric Algebra computations based on Maple, we use a library called Cliffordlib [3], developed by Rafal Ablamowicz and Bertfried Fauser; see [4] for download and installation hints. The most important operations of the Clifford package are presented in Table 4.1. We use the left contraction (LC) operation (see [29]) for the inner product.

Besides these main operations, we also need some methods such as `scalarpart()` and `vectorpart()` for extracting the scalar or the vector part of a multivector.

In order to perform CGA computations, we have to load the Clifford package, set the corresponding metric (positive signature for the basis vectors $e_1, \dots, e_4$ and negative signature for e_5), set aliases for basic blades (optional), and define e_0 and e_∞ as shown in the following Maple listing:



Grade	Term	Blades	No.
0	Scalar	$\boxed{1}$	1
1	Vector	$e_1, e_2, e_3, e_\infty, e_0$	5
2	Bivector	$\boxed{e_1 \wedge e_2}$, $\boxed{e_1 \wedge e_3}$, $e_1 \wedge e_\infty$, $e_1 \wedge e_0$, $\boxed{e_2 \wedge e_3}$, $e_2 \wedge e_\infty$, $e_2 \wedge e_0$, $e_3 \wedge e_\infty$, $e_3 \wedge e_0$, $e_\infty \wedge e_0$	10
3	Trivector	$e_1 \wedge e_2 \wedge e_3$, $e_1 \wedge e_2 \wedge e_\infty$, $e_1 \wedge e_2 \wedge e_0$, $e_1 \wedge e_3 \wedge e_\infty$, $e_1 \wedge e_3 \wedge e_0$, $e_1 \wedge e_\infty \wedge e_0$, $e_2 \wedge e_3 \wedge e_\infty$, $e_2 \wedge e_3 \wedge e_0$, $e_2 \wedge e_\infty \wedge e_0$, $e_3 \wedge e_\infty \wedge e_0$	10
4	Quadvector	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$, $e_1 \wedge e_2 \wedge e_3 \wedge e_0$, $e_1 \wedge e_2 \wedge e_\infty \wedge e_0$, $e_1 \wedge e_3 \wedge e_\infty \wedge e_0$, $e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5
5	Pseudoscalar	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	1

i, j, k
$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$
...

Fig. 4.1 The blades of CGA and the geometric meaning of some of them

Table 4.1 Notation for Geometric Algebra operations in Maple

Notation	Meaning
`a &c b`	Geometric product
`a &w b`	Outer product
`LC(a,b)`	Inner product
`-(a) &c e12345`	Dualization
`reversion()`	Reversion

```
> with(Clifford);
> B:=linalg[diag](1, 1, 1, 1, -1);
> eval(makealiases(5, "ordered"));
> e0:=-0.5*e4+0.5*e5;
> einf:=e4+e5;
```

There is the possibility to write user-specific functions such as the following often-needed functions for the computation of the conformal representation of a 3D point,

```
> conformal := proc(x)
>   local conf;
>   global einf, e0;
>   conf := x + 1/2 * x &c x &c einf + e0;
> RETURN(conf);
> end:
```

and for the computation of the dual,

```
> dual := proc(x)
>   local dual;
>   global e12345;
>   dual := - x &c e12345;
> RETURN(dual);
> end:
```

in accordance with Sect. 2.2.3.1.

In Chap. 9, you will see how advantageously Maple can be used for the optimization of a computer animation application.

In the following sections, we will use Maple to investigate how different algebras can be identified within CGA.

4.2 Complex Numbers

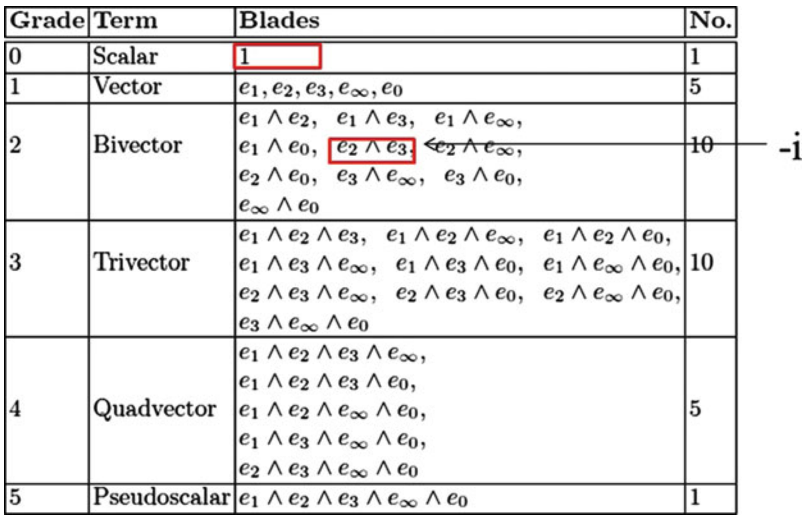
Complex numbers are linear combinations of scalars and the imaginary unit i, which squares to -1. This imaginary unit can be identified in CGA as one of the 2-blades $e_1 \wedge e_2,\ e_1 \wedge e_3$, and $e_2 \wedge e_3$, spanned by the three Euclidean basis vectors e_1, e_2, e_3. We select

$$i = e_3 \wedge e_2 = -e_2 \wedge e_3 \tag{4.1}$$

in order to be in line with the imaginary units of the quaternions as described in Sect. 4.3. We have

$$i^2 = (e_3 \wedge e_2)^2 = e_3 e_2 \underbrace{e_3 e_2}_{-e_2 e_3} = -e_3 \underbrace{e_2 e_2}_{1} e_3 = -\underbrace{e_3 e_3}_{1} = -1 \tag{4.2}$$

Of the 32 blades of CGA, only two blades are needed for complex numbers, namely the scalar and one 2-blade, as indicated in Fig. 4.2. From a geometric point of view, the imaginary unit i according to (4.1) represents a line along the x-axis: Planes are represented according to Eq. (3.15) by their 3D normal vector and their distance



Grade	Term	Blades	No.
0	Scalar	$\boxed{1}$	1
1	Vector	$e_1, e_2, e_3, e_\infty, e_0$	5
2	Bivector	$e_1 \wedge e_2,\ e_1 \wedge e_3,\ e_1 \wedge e_\infty,$ $e_1 \wedge e_0,\ \boxed{e_2 \wedge e_3},\ e_2 \wedge e_\infty,$ $e_2 \wedge e_0,\ e_3 \wedge e_\infty,\ e_3 \wedge e_0,$ $e_\infty \wedge e_0$	10
3	Trivector	$e_1 \wedge e_2 \wedge e_3,\ e_1 \wedge e_2 \wedge e_\infty,\ e_1 \wedge e_2 \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty,\ e_1 \wedge e_3 \wedge e_0,\ e_1 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty,\ e_2 \wedge e_3 \wedge e_0,\ e_2 \wedge e_\infty \wedge e_0,$ $e_3 \wedge e_\infty \wedge e_0$	10
4	Quadvector	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty,$ $e_1 \wedge e_2 \wedge e_3 \wedge e_0,$ $e_1 \wedge e_2 \wedge e_\infty \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5
5	Pseudoscalar	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	1

Fig. 4.2 Complex numbers in CGA. The imaginary unit can be identified in CGA as one of the 2-blades $e_1 \wedge e_2$, $e_1 \wedge e_3$, and $e_2 \wedge e_3$, each with a different geometric meaning

to the origin. e_3 represents a plane through the origin in the direction of the x-axis and the y-axis. A line is represented as the intersection of two planes (see (3.19)). Therefore $i = e_3 \wedge e_2$ represents the intersection of the two planes represented by its normal vectors e_3 and e_2. This results in a line along the x-axis.

Let us now investigate a rotation by an angle ϕ around the line represented by the imaginary unit $i = e_3 \wedge e_2$. According to (3.52) this rotation can be described by a rotor R, where

$$R = e^{-\frac{\phi}{2} e_3 \wedge e_2}.$$

With the help of a Taylor series, we can write

$$R = 1 + \frac{-e_3 \wedge e_2 \frac{\phi}{2}}{1!} + \frac{(-e_3 \wedge e_2 \frac{\phi}{2})^2}{2!} + \frac{(-e_3 \wedge e_2 \frac{\phi}{2})^3}{3!} + \frac{(-e_3 \wedge e_2 \frac{\phi}{2})^4}{4!} + \frac{(-e_3 \wedge e_2 \frac{\phi}{2})^5}{5!} + \ldots$$

or

$$R = 1 - \frac{e_3 \wedge e_2 \frac{\phi}{2}}{1!} + \frac{(e_3 \wedge e_2 \frac{\phi}{2})^2}{2!} - \frac{(e_3 \wedge e_2 \frac{\phi}{2})^3}{3!} + \frac{(e_3 \wedge e_2 \frac{\phi}{2})^4}{4!} - \frac{(e_3 \wedge e_2 \frac{\phi}{2})^5}{5!} + \ldots$$

or, according to (4.2),

$$R = 1 - \frac{(\frac{\phi}{2})^2}{2!} + \frac{(\frac{\phi}{2})^4}{4!} - \frac{(\frac{\phi}{2})^6}{6!} + \ldots - e_3 \wedge e_2 \frac{\frac{\phi}{2}}{1!} + e_3 \wedge e_2 \frac{(\frac{\phi}{2})^3}{3!} - e_3 \wedge e_2 \frac{(\frac{\phi}{2})^5}{5!} + \ldots,$$

and therefore

$$R = \cos\left(\frac{\phi}{2}\right) - e_3 \wedge e_2 \sin\left(\frac{\phi}{2}\right) \tag{4.3}$$

or

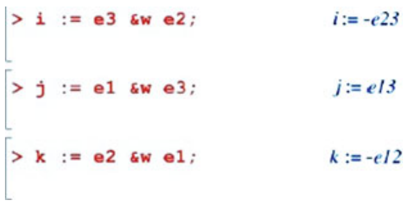
$$R = \cos\left(-\frac{\phi}{2}\right) + e_3 \wedge e_2 \sin\left(-\frac{\phi}{2}\right). \tag{4.4}$$

This means that a complex number identified by the imaginary unit $i = e_3 \wedge e_2$ represents a rotation around the x-axis. Identifying the imaginary unit with one of the other bivectors, that square to -1 would mean a rotation around the y-axis or the z-axis.

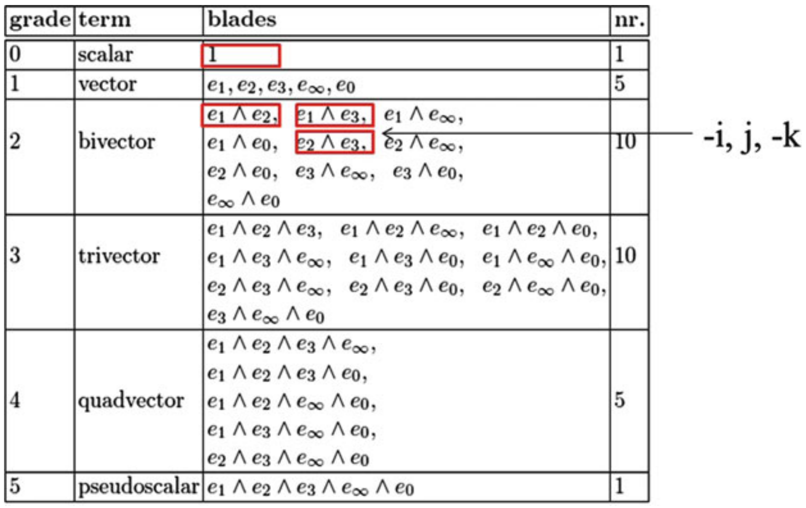
4.3 Quaternions

There is a lot of literature about quaternions represented in Euclidean Geometric Algebra (e.g., [47, 65, 70, 76, 86]). In this section, we will see how they can be embedded in CGA in a very intuitive way. The main observation is that an arbitrary line through the origin represents the rotation axis for a quaternion if we use the following definitions for the imaginary units (defined in Maple according to Fig. 4.3):

Fig. 4.3 Definition of quaternions in Maple



```
> i := e3 &w e2;                    i:= -e23
> j := e1 &w e3;                    j:= e13
> k := e2 &w e1;                    k:= -e12
```



grade	term	blades	nr.
0	scalar	$\boxed{1}$	1
1	vector	$e_1, e_2, e_3, e_\infty, e_0$	5
2	bivector	$\boxed{e_1 \wedge e_2,}$ $\boxed{e_1 \wedge e_3,}$ $e_1 \wedge e_\infty,$ $e_1 \wedge e_0,$ $\boxed{e_2 \wedge e_3,}$ $e_2 \wedge e_\infty,$ $e_2 \wedge e_0,$ $e_3 \wedge e_\infty,$ $e_3 \wedge e_0,$ $e_\infty \wedge e_0$	10
3	trivector	$e_1 \wedge e_2 \wedge e_3,$ $e_1 \wedge e_2 \wedge e_\infty,$ $e_1 \wedge e_2 \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty,$ $e_1 \wedge e_3 \wedge e_0,$ $e_1 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty,$ $e_2 \wedge e_3 \wedge e_0,$ $e_2 \wedge e_\infty \wedge e_0,$ $e_3 \wedge e_\infty \wedge e_0$	10
4	quadvector	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty,$ $e_1 \wedge e_2 \wedge e_3 \wedge e_0,$ $e_1 \wedge e_2 \wedge e_\infty \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5
5	pseudoscalar	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	1

Fig. 4.4 Quaternions in CGA

$$i = e_3 \wedge e_2, \tag{4.5}$$

$$j = e_1 \wedge e_3, \tag{4.6}$$

$$k = e_2 \wedge e_1. \tag{4.7}$$

Figure 4.4 shows the four blades for quaternions, representing the scalar part and the three imaginary parts.

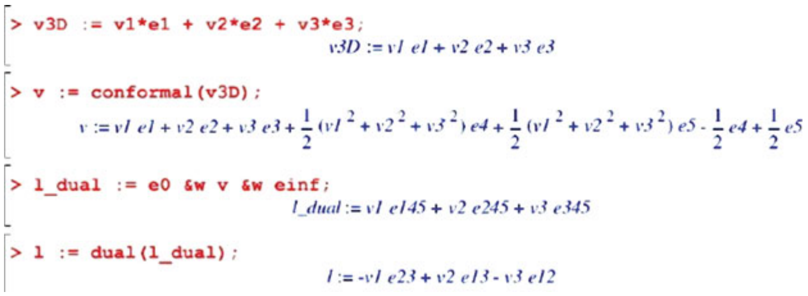
Let $\mathbf{v}$ be an arbitrary *normalized Euclidean 3D vector*

$$\mathbf{v} = v_1 e_1 + v_2 e_2 + v_3 e_3. \tag{4.8}$$

The conformal representation of the Euclidean point (v_1, v_2, v_3) is

$$P = \mathbf{v} + \frac{1}{2}\mathbf{v}^2 e_\infty + e_0, \tag{4.9}$$

According to Table 3.2, the line through the origin e_0 and the point P is described by their outer product with the point at infinity e_∞,



```
> v3D := v1*e1 + v2*e2 + v3*e3;
                          v3D := v1 e1 + v2 e2 + v3 e3
> v := conformal(v3D);
    v := v1 e1 + v2 e2 + v3 e3 + 1/2 (v1^2 + v2^2 + v3^2) e4 + 1/2 (v1^2 + v2^2 + v3^2) e5 - 1/2 e4 + 1/2 e5
> l_dual := e0 &w v &w einf;
                       l_dual := v1 e145 + v2 e245 + v3 e345
> l := dual(l_dual);
                          l := -v1 e23 + v2 e13 - v3 e12
```

Fig. 4.5 Computation of the representation of a line in Maple

$$L^* = e_0 \wedge P \wedge e_\infty. \tag{4.10}$$

A dualization calculation leads to the standard representation of the line (see Fig. 4.5 for a screenshot of the Maple computations)

$$L = v_1(e_3 \wedge e_2) + v_2(e_1 \wedge e_3) + v_3(e_2 \wedge e_1) \tag{4.11}$$

or

$$L = v_1 i + v_2 j + v_3 k. \tag{4.12}$$

We will see in the following sections that a rotation around this axis L by an angle of ϕ can be computed using the following quaternion:

$$Q = \cos\left(\frac{\phi}{2}\right) + L \sin\left(\frac{\phi}{2}\right). \tag{4.13}$$

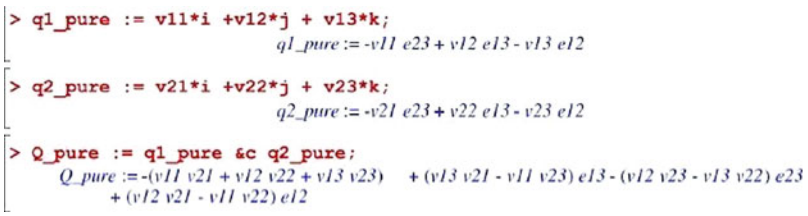
4.3.1 The Imaginary Units

For the imaginary units defined in (4.5), (4.6), and (4.7), we can derive the following properties:

$$i^2 = (e_3 \wedge e_2)^2 = e_3 e_2 \underbrace{e_3 e_2}_{-e_2 e_3} = -e_3 \underbrace{e_2 e_2}_{1} e_3 = -\underbrace{e_3 e_3}_{1} = -1,$$

$$j^2 = (e_1 \wedge e_3)^2 = e_1 e_3 \underbrace{e_1 e_3}_{-e_3 e_1} = -e_1 \underbrace{e_3 e_3}_{1} e_1 = -\underbrace{e_1 e_1}_{1} = -1,$$

$$k^2 = (-e_1 \wedge e_2)^2 = e_1 e_2 \underbrace{e_1 e_2}_{-e_2 e_1} = -e_1 \underbrace{e_2 e_2}_{1} e_1 = -\underbrace{e_1 e_1}_{1} = -1.$$



```
> q1_pure := v11*i +v12*j + v13*k;
                         q1_pure := -v11 e23 + v12 e13 - v13 e12
> q2_pure := v21*i +v22*j + v23*k;
                         q2_pure := -v21 e23 + v22 e13 - v23 e12
> Q_pure := q1_pure &c q2_pure;
   Q_pure := -(v11 v21 + v12 v22 + v13 v23)  + (v13 v21 - v11 v23) e13 - (v12 v23 - v13 v22) e23
         + (v12 v21 - v11 v22) e12
```

Fig. 4.6 The product of pure quaternions in Maple

For the multiplication of i and j, we get

$$ij = (e_3 \wedge e_2)(e_1 \wedge e_3) = e_3e_2e_1e_3 = e_2e_3e_3e_1 = e_2 \wedge e_1 = k.$$

Accordingly,

$$jk = i,$$

$$ki = j,$$

and

$$ijk = ii = -1.$$

We recognize that the three imaginary units i, j, k are represented as the three axes in CGA. As with the imaginary unit of Sect. 4.2, i represents the x-axis and, in addition, j and k represent the y-axis and z-axis, respectively.

4.3.2 Pure Quaternions and Their Geometric Product

A pure quaternion Q_1 has no scalar part. Its CGA form is

$$Q_1 = v_{11}(e_3 \wedge e_2) + v_{12}(e_1 \wedge e_3) + v_{13}(e_2 \wedge e_1) \tag{4.14}$$

or, using the definitions (4.5), (4.6), and (4.7),

$$Q_1 = v_{11}i + v_{12}j + v_{13}k, \tag{4.15}$$

where (v_{11}, v_{12}, v_{13}) is a normalized 3D vector.

The geometric product of a pure quaternion Q_1 and a pure quaternion Q_2, where

$$Q_2 = v_{21}(e_3 \wedge e_2) + v_{22}(e_1 \wedge e_3) + v_{23}(e_2 \wedge e_1),$$

is, according to the Maple evaluation in Fig. 4.6,

$$Q_1Q_2 = -(v_{11}v_{21} + v_{12}v_{22} + v_{13}v_{23}) \tag{4.16}$$

$$+(v_{12}v_{23}-v_{13}v_{22})(e_3\wedge e_2)+(v_{13}v_{21}-v_{11}v_{23})(e_1\wedge e_3)+(v_{11}v_{22}-v_{12}v_{21})(e_2\wedge e_1),$$

which is equivalent to the product of quaternions

$$Q_1Q_2 = -(v_{11}v_{21}+v_{12}v_{22}+v_{13}v_{23})+(v_{12}v_{23}-v_{13}v_{22})i+(v_{13}v_{21}-v_{11}v_{23})j \\ +(v_{11}v_{22}-v_{12}v_{21})k. \tag{4.17}$$

Note that the square of a pure quaternion therefore is

$$Q_1^2 = -(v_{11}v_{11}+v_{12}v_{12}+v_{13}v_{13}) = -1. \tag{4.18}$$

4.3.3 Rotations Based on Unit Quaternions

Rotations based on quaternions are restricted to rotations with a rotation axis passing through the origin. They can be defined by

$$Q = e^{\frac{\phi}{2}L}, \tag{4.19}$$

where

$$L = v_1 i + v_2 j + v_3 k$$

represents a normalized line through the origin according to the Euclidean direction vector of (4.8). This leads to the well-known definition of general quaternions

$$Q = \cos\left(\frac{\phi}{2}\right) + L\sin\left(\frac{\phi}{2}\right). \tag{4.20}$$

Note. With the help of the Taylor series and the property $L^2 = -1$ (see (4.18)), we have

$$\begin{aligned} Q &= e^{\frac{\phi}{2}L} \\ &= 1+\frac{L\frac{\phi}{2}}{1!}+\frac{(L\frac{\phi}{2})^2}{2!}+\frac{(L\frac{\phi}{2})^3}{3!}+\frac{(L\frac{\phi}{2})^4}{4!}+\frac{(L\frac{\phi}{2})^5}{5!}+\frac{(L\frac{\phi}{2})^6}{6!}+\dots \\ &= 1-\frac{(\frac{\phi}{2})^2}{2!}+\frac{(\frac{\phi}{2})^4}{4!}-\frac{(\frac{\phi}{2})^6}{6!}+\dots \\ &\quad +L\frac{\frac{\phi}{2}}{1!}-L\frac{(\frac{\phi}{2})^3}{3!}+L\frac{(\frac{\phi}{2})^5}{5!}+\dots \\ &= \cos(\frac{\phi}{2})+L\sin(\frac{\phi}{2}). \end{aligned}$$

There is only a slight difference between the quaternion of (4.20) and the rotor of Sect. 3.5. The sign difference indicates that the rotations are in different directions.

In CGA, we rotate an object o with the help of the operation

$$o_{rotated} = Qo\tilde{Q}, \tag{4.21}$$

where $\tilde{Q}$ is the reverse of Q,

$$\tilde{Q} = \cos\left(\frac{\phi}{2}\right) - L\sin\left(\frac{\phi}{2}\right), \tag{4.22}$$

which is also referred to as the **conjugate** of a quaternion.

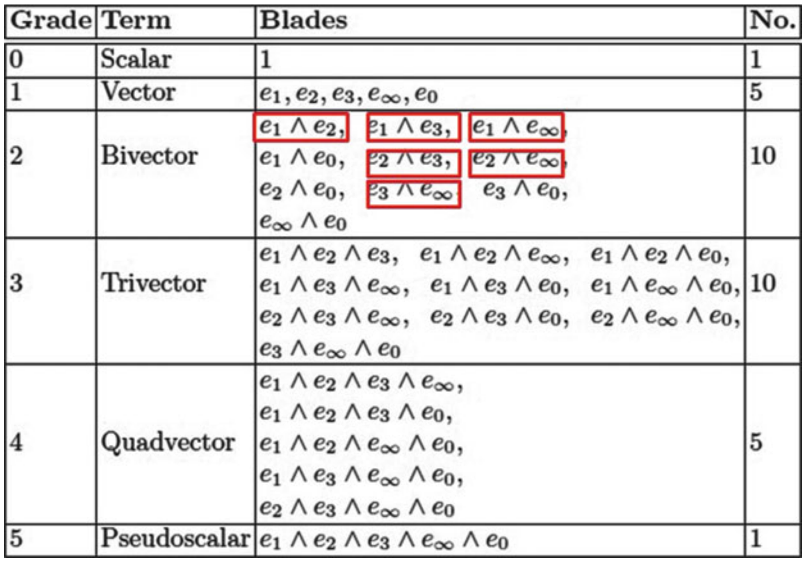
Note that for $\phi = \pi$, the quaternion

$$Q = v_1 i + v_2 j + v_3 k \tag{4.23}$$

represents a line through the origin and the 3D point represented by the normalized 3D vector (v_1, v_2, v_3), and also a rotation by an angle $\phi = \pi$ about this line. We will use this property advantageously in the inverse kinematics algorithm described in Chap. 9.

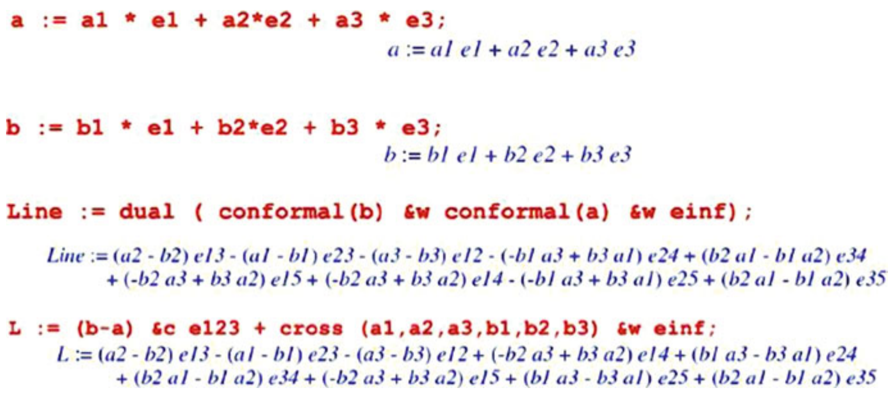
4.4 Plücker Coordinates

We will see in this section that Plücker coordinates can be identified in CGA based on the six 2-blades indicated in Fig. 4.7. Since Plücker coordinates describe an arbitrary line, we first of all compute the representation of a line. According to the



Grade	Term	Blades	No.
0	Scalar	1	1
1	Vector	$e_1, e_2, e_3, e_\infty, e_0$	5
2	Bivector	$\boxed{e_1 \wedge e_2,}$ $\boxed{e_1 \wedge e_3,}$ $\boxed{e_1 \wedge e_\infty}$, $e_1 \wedge e_0$, $\boxed{e_2 \wedge e_3,}$ $\boxed{e_2 \wedge e_\infty}$, $e_2 \wedge e_0$, $\boxed{e_3 \wedge e_\infty}$ $e_3 \wedge e_0$, $e_\infty \wedge e_0$	10
3	Trivector	$e_1 \wedge e_2 \wedge e_3$, $e_1 \wedge e_2 \wedge e_\infty$, $e_1 \wedge e_2 \wedge e_0$, $e_1 \wedge e_3 \wedge e_\infty$, $e_1 \wedge e_3 \wedge e_0$, $e_1 \wedge e_\infty \wedge e_0$, $e_2 \wedge e_3 \wedge e_\infty$, $e_2 \wedge e_3 \wedge e_0$, $e_2 \wedge e_\infty \wedge e_0$, $e_3 \wedge e_\infty \wedge e_0$	10
4	Quadvector	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$, $e_1 \wedge e_2 \wedge e_3 \wedge e_0$, $e_1 \wedge e_2 \wedge e_\infty \wedge e_0$, $e_1 \wedge e_3 \wedge e_\infty \wedge e_0$, $e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5
5	Pseudoscalar	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	1

Fig. 4.7 Plücker coordinates in CGA



```
a := a1 * e1 + a2*e2 + a3 * e3;
                                   a := a1 e1 + a2 e2 + a3 e3


b := b1 * e1 + b2*e2 + b3 * e3;
                                   b := b1 e1 + b2 e2 + b3 e3

Line := dual ( conformal(b) &w conformal(a) &w einf);

   Line := (a2 - b2) e13 - (a1 - b1) e23 - (a3 - b3) e12 - (-b1 a3 + b3 a1) e24 + (b2 a1 - b1 a2) e34
          + (-b2 a3 + b3 a2) e15 + (-b2 a3 + b3 a2) e14 - (-b1 a3 + b3 a1) e25 + (b2 a1 - b1 a2) e35

L := (b-a) &c e123 + cross (a1,a2,a3,b1,b2,b3) &w einf;
   L := (a2 - b2) e13 - (a1 - b1) e23 - (a3 - b3) e12 + (-b2 a3 + b3 a2) e14 + (b1 a3 - b3 a1) e24
        + (b2 a1 - b1 a2) e34 + (-b2 a3 + b3 a2) e15 + (b1 a3 - b3 a1) e25 + (b2 a1 - b1 a2) e35
```

Fig. 4.8 Computations of Plücker coordinates in Maple

Maple program shown in Fig. 4.8, this results in

$$L = -(a_3 - b_3)e_1 \wedge e_2 + (a_2 - b_2)e_1 \wedge e_3 - (a_1 - b_1)e_2 \wedge e_3 \tag{4.24}$$

$$+(a_2b_3 - a_3b_2)e_1 \wedge e_\infty - (a_1b_3 - a_3b_1)e_2 \wedge e_\infty + (a_1b_2 - a_2b_1)e_3 \wedge e_\infty.$$

Another form is

$$L = (a_1 - b_1)i + (a_2 - b_2)j + (a_3 - b_3)k \tag{4.25}$$

$$+(a_2b_3 - a_3b_2)e_1 \wedge e_\infty - (a_1b_3 - a_3b_1)e_2 \wedge e_\infty + (a_1b_2 - a_2b_1)e_3 \wedge e_\infty$$

or

$$L = (\mathbf{b} - \mathbf{a})e_{123} + [\mathbf{a} \times \mathbf{b}] \wedge e_\infty, \tag{4.26}$$

resulting in

$$L = \mathbf{u}e_{123} + \mathbf{m} \wedge e_\infty, \tag{4.27}$$

where $\mathbf{u} = \mathbf{b} - \mathbf{a}$ is a Euclidean direction vector and $\mathbf{m} = \mathbf{a} \times \mathbf{b}$ is a moment vector, (Fig. 4.9). The corresponding six Plücker coordinates are

$$(\mathbf{u} : \mathbf{m}) = (u_1 : u_2 : u_3 : m_1 : m_2 : m_3). \tag{4.28}$$

Note that, based on the above equations, the pair $(\mathbf{u} : \mathbf{m})$ uniquely determines the line L up to a common (non zero) factor which depends on the distance between **a** and **b**. That is, the coordinates may be considered as homogeneous coordinates for L, in the sense that all pairs $(\lambda\mathbf{u} : \lambda\mathbf{m})$, for $\lambda \neq 0$, can be produced by points on L and only L, and any such pair determines a unique line as long as **u** is not zero and $\mathbf{u} \cdot \mathbf{m} = 0$ (see [2]).

The cross product $\mathbf{m} = \mathbf{a} \times \mathbf{b}$ can also be written as $\mathbf{m} = -(\mathbf{a} \wedge \mathbf{b})e_{123}$ according to (2.5); $\mathbf{u}e_{123}$ is equal to $u_1e_{23} - u_2e_{13} + u_3e_{12}$, since

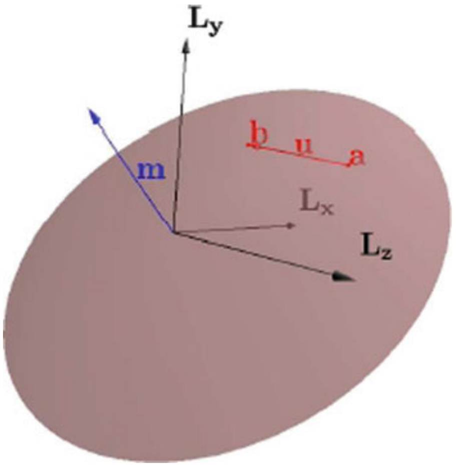


Fig. 4.9 The six Plücker coordinates of the line through a and b consist of the coordinates of the direction vector $\mathbf{u}$ and the moment vector $\mathbf{m}$

$$
\begin{aligned}
&(u_1e_1 + u_2e_2 + u_3e_3)e_1e_2e_3 \\
&= u_1 \underbrace{e_1e_1}_{1} e_2e_3 + u_2 \underbrace{e_2e_1}_{-e_1e_2} e_2e_3 + u_3 \underbrace{e_3e_1}_{-e_1e_3} \underbrace{e_2e_3}_{-e_3e_2} \\
&= u_1e_2e_3 - u_2e_1e_2e_2e_3 + u_3e_1e_3e_3e_2 \\
&= u_1e_2e_3 - u_2e_1e_3 + u_3e_1e_2.
\end{aligned}
$$

Example. For the two Euclidean points $\mathbf{a} = (1, 0, 1)$ and $\mathbf{b} = (1, 0, 0)$, we get

$$\mathbf{u} = -e_3$$

and

$$\mathbf{m} = e_2,$$

leading to the line

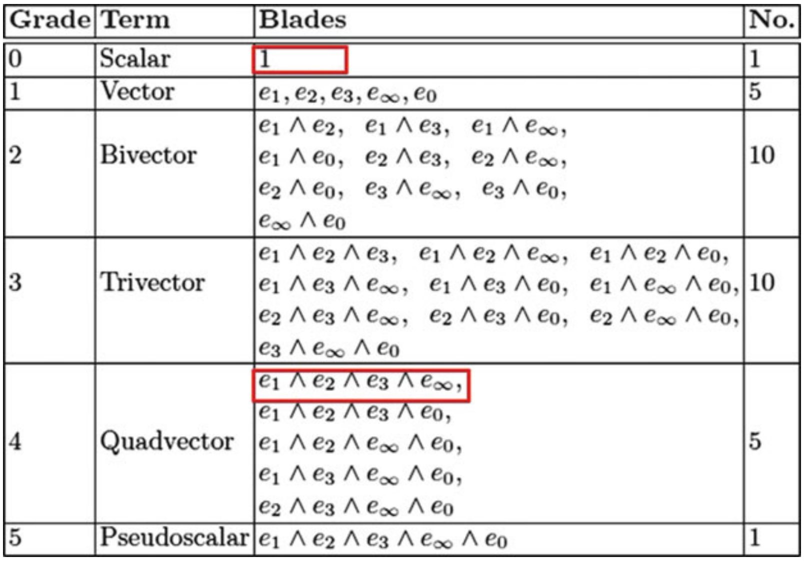
$$L = -e_3e_{123} + e_2 \wedge e_\infty = -e_{12} + e_2 \wedge e_\infty \tag{4.29}$$

with the six Plücker coordinates $(0 : 0 : -1 : 0 : 1 : 0)$.

4.5 Dual Numbers

A dual number is a number $x + \epsilon y$, where x, y are scalars and ϵ is a unit with the property that $\epsilon^2 = 0$. In CGA, there are many blades with this property of squaring to zero, since $e_0^2 = e_\infty^2 = 0$. We use

$$\epsilon = e_1 \wedge e_2 \wedge e_3 \wedge e_\infty, \tag{4.30}$$



Grade	Term	Blades	No.
0	Scalar	$\boxed{1}$	1
1	Vector	$e_1, e_2, e_3, e_\infty, e_0$	5
2	Bivector	$e_1 \wedge e_2,\ e_1 \wedge e_3,\ e_1 \wedge e_\infty,$ $e_1 \wedge e_0,\ e_2 \wedge e_3,\ e_2 \wedge e_\infty,$ $e_2 \wedge e_0,\ e_3 \wedge e_\infty,\ e_3 \wedge e_0,$ $e_\infty \wedge e_0$	10
3	Trivector	$e_1 \wedge e_2 \wedge e_3,\ e_1 \wedge e_2 \wedge e_\infty,\ e_1 \wedge e_2 \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty,\ e_1 \wedge e_3 \wedge e_0,\ e_1 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty,\ e_2 \wedge e_3 \wedge e_0,\ e_2 \wedge e_\infty \wedge e_0,$ $e_3 \wedge e_\infty \wedge e_0$	10
4	Quadvector	$\boxed{e_1 \wedge e_2 \wedge e_3 \wedge e_\infty,}$ $e_1 \wedge e_2 \wedge e_3 \wedge e_0,$ $e_1 \wedge e_2 \wedge e_\infty \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5
5	Pseudoscalar	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	1

Fig. 4.10 Dual numbers in CGA

as used in [67], which has this property, to obtain dual numbers and also dual quaternions (see Sect. 4.6). We can write a dual number in CGA as follows:

$$x + \epsilon y = x + e_1 \wedge e_2 \wedge e_3 \wedge e_\infty y. \tag{4.31}$$

Figure 4.10 shows the subset of the 32 blades of CGA needed for dual numbers, namely the scalar and ϵ.

4.6 Dual Quaternions

A dual quaternion is defined by

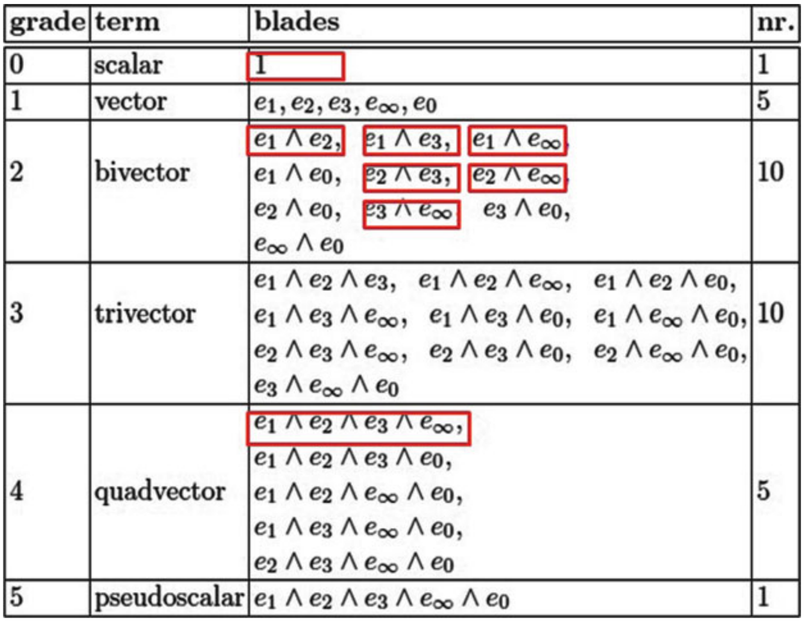
$$Q = Q_1 + \epsilon Q_2, \tag{4.32}$$

with the quaternions

$$Q_i = s_i + v_{i1} i + v_{i2} j + v_{i3} k, \tag{4.33}$$

and where ϵ is a unit with the property that $\epsilon^2 = 0$.

We will see in this section that dual quaternions – as often used in robotics – can be identified in CGA based on the six 2-blades used for Plücker coordinates, together with the scalar and one 4-blade ($\epsilon = e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$), as indicated in Fig. 4.11. We will also see in the following that there is a geometrically intuitive



grade	term	blades	nr.
0	scalar	$\boxed{1}$	1
1	vector	$e_1, e_2, e_3, e_\infty, e_0$	5
2	bivector	$\boxed{e_1 \wedge e_2,}$ $\boxed{e_1 \wedge e_3,}$ $\boxed{e_1 \wedge e_\infty}$ $e_1 \wedge e_0,$ $\boxed{e_2 \wedge e_3,}$ $\boxed{e_2 \wedge e_\infty}$ $e_2 \wedge e_0,$ $\boxed{e_3 \wedge e_\infty}$ $e_3 \wedge e_0,$ $e_\infty \wedge e_0$	10
3	trivector	$e_1 \wedge e_2 \wedge e_3,$ $e_1 \wedge e_2 \wedge e_\infty,$ $e_1 \wedge e_2 \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty,$ $e_1 \wedge e_3 \wedge e_0,$ $e_1 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty,$ $e_2 \wedge e_3 \wedge e_0,$ $e_2 \wedge e_\infty \wedge e_0,$ $e_3 \wedge e_\infty \wedge e_0$	10
4	quadvector	$\boxed{e_1 \wedge e_2 \wedge e_3 \wedge e_\infty,}$ $e_1 \wedge e_2 \wedge e_3 \wedge e_0,$ $e_1 \wedge e_2 \wedge e_\infty \wedge e_0,$ $e_1 \wedge e_3 \wedge e_\infty \wedge e_0,$ $e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5
5	pseudoscalar	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	1

Fig. 4.11 Dual quaternions are represented in CGA based on eight blades: the scalar, six 2-blades and one 4-blade

relation between quaternions and a versor describing, for instance, a rotation around an arbitrary rotation axis.

Quaternions can be written in CGA form as

$$Q_i = s_i + v_{i1}(e_3 \wedge e_2) + v_{i2}(e_1 \wedge e_3) + v_{i3}(e_2 \wedge e_1). \tag{4.34}$$

Using (4.30), we can write a dual quaternion in CGA as follows:

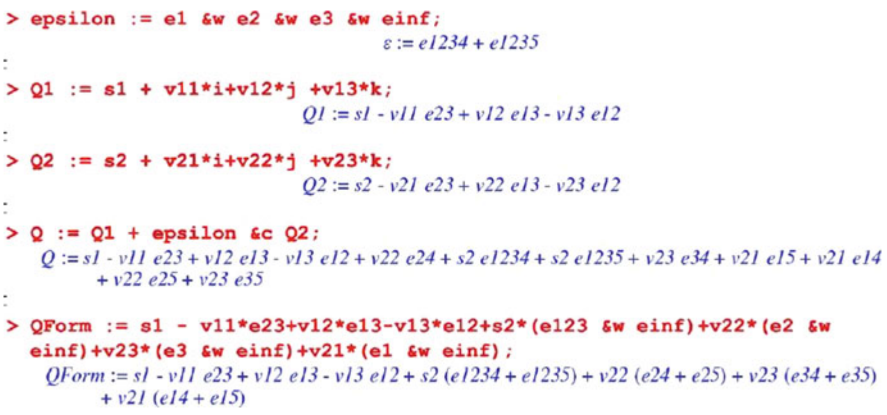
$$Q = Q_1 + \epsilon Q_2 = Q_1 + e_1 \wedge e_2 \wedge e_3 \wedge e_\infty Q_2. \tag{4.35}$$

This can be written in the form of a linear combination of the above-mentioned eight blades (Fig. 4.12),

$$Q = s_1 - v_{11}(e_2 \wedge e_3) + v_{12}(e_1 \wedge e_3) - v_{13}(e_1 \wedge e_2) \tag{4.36}$$

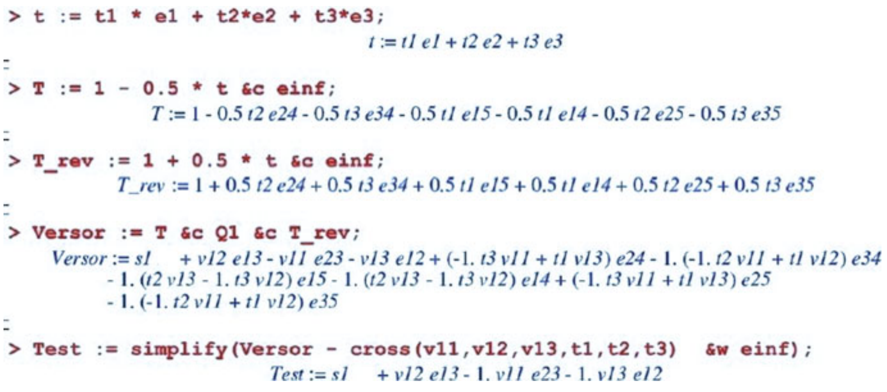
$$+s_2(e_1 \wedge e_2 \wedge e_3 \wedge e_\infty) + v_{21}(e_1 \wedge e_\infty) + v_{22}(e_2 \wedge e_\infty) + v_{23}(e_3 \wedge e_\infty).$$

We will see now that there is a geometric relation between dual quaternions and a versor describing a rotation with an arbitrary rotation axis. This operation is especially helpful for describing the movement of a robot at a revolute joint.



```
> epsilon := e1 &w e2 &w e3 &w einf;
                                    ε := e1234 + e1235
:
> Q1 := s1 + v11*i+v12*j +v13*k;
                              Q1 := s1 - v11 e23 + v12 e13 - v13 e12
:
> Q2 := s2 + v21*i+v22*j +v23*k;
                              Q2 := s2 - v21 e23 + v22 e13 - v23 e12
:
> Q := Q1 + epsilon &c Q2;
   Q := s1 - v11 e23 + v12 e13 - v13 e12 + v22 e24 + s2 e1234 + s2 e1235 + v23 e34 + v21 e15 + v21 e14
        + v22 e25 + v23 e35
:
> QForm := s1 - v11*e23+v12*e13-v13*e12+s2*(e123 &w einf)+v22*(e2 &w
  einf)+v23*(e3 &w einf)+v21*(e1 &w einf);
   QForm := s1 - v11 e23 + v12 e13 - v13 e12 + s2 (e1234 + e1235) + v22 (e24 + e25) + v23 (e34 + e35)
        + v21 (e14 + e15)
```

Fig. 4.12 Dual-quaternion computations in Maple



```
> t := t1 * e1 + t2*e2 + t3*e3;
                                    t := t1 e1 + t2 e2 + t3 e3
:
> T := 1 - 0.5 * t &c einf;
             T := 1 - 0.5 t2 e24 - 0.5 t3 e34 - 0.5 t1 e15 - 0.5 t1 e14 - 0.5 t2 e25 - 0.5 t3 e35
:
> T_rev := 1 + 0.5 * t &c einf;
           T_rev := 1 + 0.5 t2 e24 + 0.5 t3 e34 + 0.5 t1 e15 + 0.5 t1 e14 + 0.5 t2 e25 + 0.5 t3 e35
:
> Versor := T &c Q1 &c T_rev;
   Versor := s1    + v12 e13 - v11 e23 - v13 e12 + (-1. t3 v11 + t1 v13) e24 - 1. (-1. t2 v11 + t1 v12) e34
        - 1. (t2 v13 - 1. t3 v12) e15 - 1. (t2 v13 - 1. t3 v12) e14 + (-1. t3 v11 + t1 v13) e25
        - 1. (-1. t2 v11 + t1 v12) e35
:
> Test := simplify(Versor - cross(v11,v12,v13,t1,t2,t3)  &w einf);
                              Test := s1    + v12 e13 - 1. v11 e23 - 1. v13 e12
```

Fig. 4.13 Dual quaternion computations in Maple

An arbitrary rotation according to a quaternion Q_1 can be described based on a translation to the origin, a rotation Q_1 about the origin, and a translation back, as expressed in the versor product

$$TQ_1\tilde{T}. \tag{4.37}$$

This is equal to (Fig. 4.13)

$$V = s_1 - v_{11}(e_2 \wedge e_3) + v_{12}(e_1 \wedge e_3) - v_{13}(e_1 \wedge e_2) \tag{4.38}$$

$$+(-t_2v_{13} + t_3v_{12})(e_1 \wedge e_\infty) + (t_1v_{13} - t_3v_{11})(e_2 \wedge e_\infty) + (t_2v_{11} - t_1v_{12})(e_3 \wedge e_\infty).$$

This means that there is a relation between versors and dual quaternions as follows:

$$s_2 = 0, \tag{4.39}$$

$$v_{21} = -t_2 v_{13} + t_3 v_{12}, \tag{4.40}$$

$$v_{22} = t_1 v_{13} - t_3 v_{11}, \tag{4.41}$$

$$v_{23} = t_2 v_{11} - t_1 v_{12}. \tag{4.42}$$

In a nutshell, there is the following relation between dual quaternions and the versor $TQ_1\tilde{T}$:

$$TQ_1\tilde{T} = Q_1 + \epsilon Q_2, \tag{4.43}$$

together with the cross product

$$\mathbf{v_2} = \mathbf{v_1} \times \mathbf{t}.$$

Note that a normalization according to $\mathbf{v_2}$ is needed in order to handle quaternions correctly.

Chapter 5
Fitting of Planes or Spheres to Sets of Points

One big advantage of CGA is its easy handling of objects such as spheres and planes. Many problems in computer graphics are related to these kinds of objects.

In this chapter, a set of points $\mathbf{p}_i \in \mathbb{R}^3,\ i \in \{1, \ldots, n\}$, will be approximated with the help of the best-fitting plane or sphere [95]. Planes and spheres in conformal space are both vectors of the form

$$S = s_1 e_1 + s_2 e_2 + s_3 e_3 + s_4 e_\infty + s_5 e_0, \tag{5.1}$$

and points are specific vectors of the form

$$P_i = \mathbf{p}_i + \frac{1}{2}\mathbf{p}_i^2 e_\infty + e_0. \tag{5.2}$$

In order to solve the fitting problem, we do the following:

- Use the distance measure between a point and a sphere or plane with the help of the inner product.
- Use a least-squares approach to minimize the squares of the distances between the points and the sphere or plane.
- Solve the resulting eigenvalue problem.

The main benefit of this approach is that it fits either a plane or a sphere, depending on which one fits better. Infinity plays a key role if we wish to obtain a deeper understanding of the transition between these two geometric objects. The next section investigates how infinity is represented in CGA and how a plane can be created by increasing the radius of a sphere, infinitely.

5.1 The Role of Infinity

In Sect. 3.8, we saw that the conformal basis vector e_0 represents the origin of Euclidean space (see (3.10)). But, what about e_∞? We will see in this section that

e_∞ can be interpreted either as a sphere of infinite radius or as a point or a plane at infinity.

5.1.1 Sphere of Infinite Radius

Let us look first at a sphere with its center point at the origin. According to (3.12), this "origin sphere" ($P = e_0$) is represented as

$$S = -\frac{1}{2}r^2 e_\infty + e_0. \tag{5.3}$$

Another homogeneous representation of this origin sphere is the product of the above representation with the scalar $-\frac{2}{r^2}$,

$$S' = -\frac{2}{r^2}S = e_\infty - \frac{2}{r^2}e_0. \tag{5.4}$$

Based on this formula and the fact that S and S' represent the same sphere, we can easily see that an origin sphere of infinite radius is represented by e_∞:

$$\lim_{r\to\infty} S' = e_\infty.$$

It can be shown that this is true not only for an origin sphere but also for a sphere with an arbitrary center.

5.1.2 Point at Infinity

Let us now assume an arbitrary Euclidean point $\mathbf{x}$ (not equal to the origin) represented by a conformal vector P, where

$$P = \mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0,$$

with a Euclidean normal vector $\mathbf{n}$ in the direction of $\mathbf{x}$ (see Fig. 5.1),

$$\mathbf{x} = t\mathbf{n},\ t > 0,\ \mathbf{n}^2 = 1.$$

Another homogeneous representation of this point is its product with the scalar $\frac{2}{\mathbf{x}^2}$,

$$P' = \frac{2}{\mathbf{x}^2}(\mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0),$$

$$P' = \frac{2}{\mathbf{x}^2}\mathbf{x} + e_\infty + \frac{2}{\mathbf{x}^2}e_0.$$

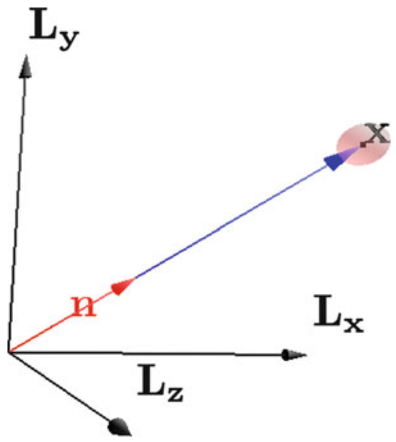


Fig. 5.1 The point at infinity

We use this form to compute the limit $\lim_{t\to\infty} P'$ for increasing $\mathbf{x}$. Since $\mathbf{x} = t\mathbf{n}$, we get

$$P' = \frac{2}{t^2\mathbf{n}^2} t\mathbf{n} + e_\infty + \frac{2}{t^2\mathbf{n}^2} e_0$$

and, since $\mathbf{n}^2 = 1$,

$$P' = \frac{2}{t}\mathbf{n} + e_\infty + \frac{2}{t^2} e_0.$$

Based on this formula and the fact that P and P' represent the same Euclidean point, we can easily see that the point at infinity for any direction vector $\mathbf{n}$ is represented by e_∞:

$$\lim_{t\to\infty} P' = e_\infty.$$

5.1.3 *Plane at Infinite Distance from the Origin*

Let us consider a plane at an arbitrary distance $d \neq 0$ from the origin. According to (3.15), this plane is represented as

$$\pi = \mathbf{n} + d e_\infty, \tag{5.5}$$

with a 3D normal vector $\mathbf{n}$. Another homogeneous representation of this plane is its product with the scalar $\frac{1}{d}$,

$$\pi' = \frac{1}{d}\mathbf{n} + e_\infty. \tag{5.6}$$

Based on this formula and the fact that π and π' represent the same plane, we can easily see that a plane at an infinite distance from the origin is represented by e_∞:

$$\lim_{d\to\infty} \pi' = e_\infty.$$

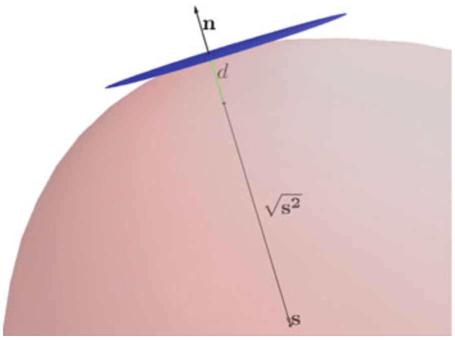


Fig. 5.2 A sphere with a *center* **s** (in the direction opposite to a normal vector **n**) that goes to infinity (while the radius of the sphere changes accordingly), results finally in a plane with a normal vector **n** and a distance d from the origin

5.1.4 Planes as a Limit of a Sphere

Spheres and planes, are both vectors in CGA. In this section, we will see how a sphere

$$S = \mathbf{s} + \frac{1}{2}(\mathbf{s}^2 - r^2)e_\infty + e_0, \tag{5.7}$$

with a Euclidean center point **s** and radius r, degenerates to a plane as the result of a limiting process.

According to the construction shown in Fig. 5.2, the minimum distance from the origin to a sphere with its center in the direction opposite to a normal vector **n** is

$$d = r - \sqrt{\mathbf{s}^2}, \tag{5.8}$$

and the radius is the sum of the length of the 3D vector **s** and d, i.e.,

$$r = \sqrt{\mathbf{s}^2} + d, \tag{5.9}$$

or

$$r^2 = \mathbf{s}^2 + 2d\sqrt{\mathbf{s}^2} + d^2. \tag{5.10}$$

The sphere can be written as

$$S = \mathbf{s} + \frac{1}{2}(\mathbf{s}^2 - \mathbf{s}^2 - 2d\sqrt{\mathbf{s}^2} - d^2)e_\infty + e_0 \tag{5.11}$$

or, equivalently,

$$S = \mathbf{s} + \frac{1}{2}(-2d\sqrt{\mathbf{s}^2} - d^2)e_\infty + e_0. \tag{5.12}$$

Now we introduce S', a scaled version of the algebraic expression for the sphere S representing geometrically the same sphere, as follows:

$$S' = -\frac{S}{\sqrt{\mathbf{s}^2}} = -\frac{\mathbf{s}}{\sqrt{\mathbf{s}^2}} + \frac{1}{2}\left(2d + \frac{d^2}{\sqrt{\mathbf{s}^2}}\right)e_\infty - \frac{e_0}{\sqrt{\mathbf{s}^2}}. \tag{5.13}$$

Since the ratio of the 3D vector **s** to its length $\sqrt{\mathbf{s}^2}$ corresponds to the negative normal vector **n** (see the construction in Fig. 5.2),

$$\lim_{\mathbf{s}^2\to\infty}\left(-\frac{S}{\sqrt{\mathbf{s}^2}}\right) = \mathbf{n} + \lim_{\mathbf{s}^2\to\infty}\frac{1}{2}\left((2d+\frac{d^2}{\sqrt{\mathbf{s}^2}}\right)e_\infty - \lim_{\mathbf{s}^2\to\infty}\frac{e_0}{\sqrt{\mathbf{s}^2}}. \tag{5.14}$$

This is equivalent to

$$\lim_{\mathbf{s}^2\to\infty}\left(-\frac{S}{\sqrt{\mathbf{s}^2}}\right) = \mathbf{n} + de_\infty, \tag{5.15}$$

which is a representation of a plane with a normal vector **n** and a distance d from the origin.

5.2 Distance Measure

From Sect. 3.4.1.5, we already know that a distance measure between a point P_i and a sphere/plane S can be defined with the help of their inner product

$$P_i \cdot S = \left(\mathbf{p}_i + \frac{1}{2}\mathbf{p}_i^2 e_\infty + e_0\right) \cdot (\mathbf{s} + s_4 e_\infty + s_5 e_0). \tag{5.16}$$

According to (3.42), this results in

$$P_i \cdot S = \mathbf{p_i} \cdot \mathbf{s} - s_4 - \frac{1}{2}s_5\mathbf{p}_i^2$$

or, equivalently,

$$P_i \cdot S = \sum_{j=1}^{5} w_{i,j}s_j, \tag{5.17}$$

where

$$w_{i,k} = \begin{cases} p_{i,k}, & k \in \{1,2,3\} \\ -1, & k = 4 \\ -\frac{1}{2}\mathbf{p}_i^2, & k = 5. \end{cases}$$

5.3 Least-Squares Approach

In the least-squares approach, we consider the minimum of the sum of the squares of the distances (expressed in terms of the inner product) between all of the points considered and the plane/sphere,

$$\min \sum_{i=1}^{n} (P_i \cdot S)^2. \tag{5.18}$$

In order to obtain this minimum, it can be rewritten in bilinear form as

$$\min(s^T Bs), \tag{5.19}$$

where

$$s^T = (s_1, s_2, s_3, s_4, s_5),$$

and the 5×5 matrix

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} & b_{4,5} \\ b_{5,1} & b_{5,2} & b_{5,3} & b_{5,4} & b_{5,5} \end{pmatrix}$$

has entries

$$b_{j,k} = \sum_{i=1}^{n} w_{i,j} w_{i,k}.$$

The matrix B is symmetric, since $b_{j,k} = b_{k,j}$. We consider only normalized results such that $s^T s = 1$. A conventional approach to such a constrained optimization problem is to introduce

$$L = s^T Bs - 0 = s^T Bs - \lambda(s^T s - 1),$$

$$s^T s = 1,$$

$$B^T = B.$$

The necessary conditions for a minimum are

$$0 = \nabla L = 2 \cdot (Bs - \lambda s) = 0$$

$$\rightarrow Bs = \lambda s.$$

The solution of the minimization problem is given by the eigenvector of B that corresponds to the smallest eigenvalue.

Figures 5.3 and 5.4 illustrate two properties of the distance measure in this approach, dealing with the double squaring of the distance and the limiting process for the distance in the case of a plane considered as a sphere of infinite radius.

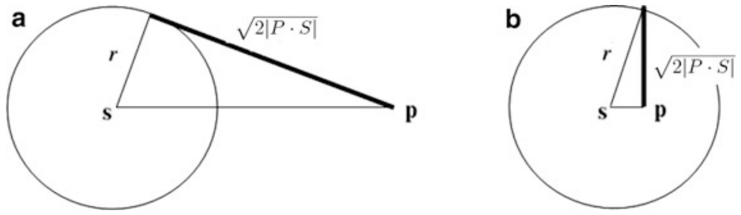


Fig. 5.3 The inner product $P \cdot S$ of a point and a sphere on the one hand already describes the square of a distance, but on the other hand has to be squared again in the least-squares method, since the inner product can be positive or negative depending on whether (**a**) the point **p** lies outside the sphere or (**b**) the point **p** lies inside the sphere

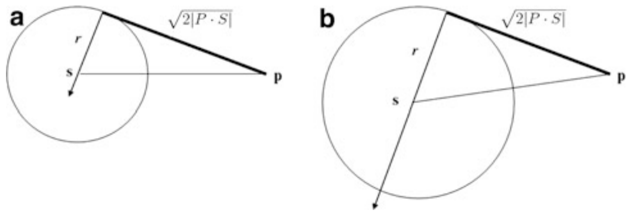


Fig. 5.4 The constraint $s^T s = 1$ leads implicitly to a scaling of the distance measure such that it gets smaller with increasing radius; if the radius increases from the one in (**a**) via the radius in (**b**) and further to an infinite radius, the distance measure gets zero for a plane considered as a sphere of infinite radius

5.4 Example

Three distinct (not collinear) points are needed to describe a plane, whereas four distinct (and not coplanar) points exactly describe a sphere. In this example, we use five points in order to demonstrate that our approach really is able to fit the best-fitting object, whether it is a sphere or a plane. First, let us look at an example where we have the following five points, four of them being coplanar:

Point	x	y	z
$\mathbf{p}_1$	1	0	0
$\mathbf{p}_2$	1	1	0
$\mathbf{p}_3$	0	0	1
$\mathbf{p}_4$	0	1	1
$\mathbf{p}_5$	−1	0	1

A least-squares calculation results in

$$S = -0.301511e_1 + 0.301511e_2 - 0.301511e_3$$
$$-0.603023e_\infty + 0.603023e_0.$$

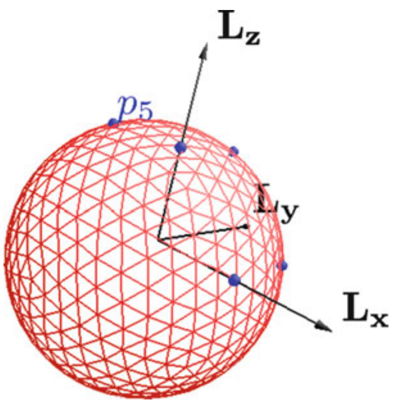


Fig. 5.5 Fitting a sphere to a set of five points

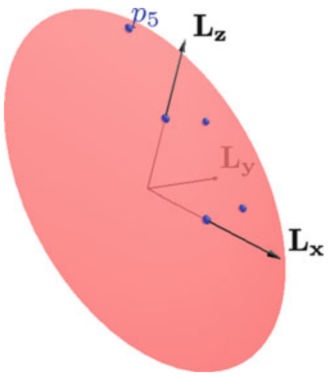


Fig. 5.6 Fitting a plane to a set of five points

Another scaled representation describing the same object is

$$S = -\frac{1}{2}e_1 + \frac{1}{2}e_2 - \frac{1}{2}e_3 - e_\infty + e_0.$$

This corresponds to a sphere with center $\mathbf{s} = (0.5, 0.5, -0.5)$ and with the square of its radius $r^2 = 2.75$ (see Fig. 5.5).

Let us now change the fifth point in order that all of the points lie in one plane:

Point	x	y	z
$\mathbf{p}_1$	1	0	0
$\mathbf{p}_2$	1	1	0
$\mathbf{p}_3$	0	0	1
$\mathbf{p}_4$	0	1	1
$\mathbf{p}_5$	−1	0	2

Now, the result is

$$S = 0.57735e_1 + 0.57735e_3 + 0.57735e_\infty,$$

representing a plane according to Fig. 5.6.

Chapter 6
A Tutorial on Geometric Algebra Using CLUCalc

Part II of this book is written in a tutorial-like style in order to encourage the reader to gain his/her own experience in developing Geometric Algebra algorithms. The relevant tool, CLUCalc, written by Christian Perwass, the tutorial examples, and the robotics applications in this part can be downloaded free of charge. While the focus of Part I was more on the mathematical foundations, the focus of Part II is more on the interactive handling of Geometric Algebra. These two parts are almost independent of each other. Readers are encouraged to start with whichever introduction helps them best to understand the basics of Geometric Algebra.

We use the CLUCalc software package to compute interactively with Geometric Algebra and to visualize the results of these computations. CLUCalc is freely available for download at [82] (please download version 4.3 in order to run the examples in this book). With the help of CLUCalc, you will be able to edit and run scripts called **CLUScripts**. The screenshot in Fig. 6.1 shows how the example in Sect. 1.1 of the intersection of two spheres can be calculated in an interactive, visual way.

CLUCalc provides the following three windows:

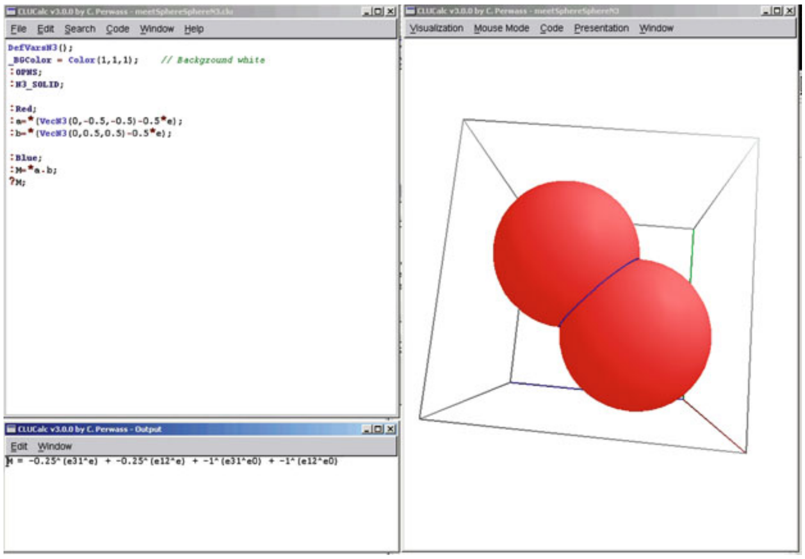
- An editor window;
- A visualization window;
- An output window.

With the help of the editor window, you can easily edit your formulas, and in the visualization window you will be able to see the spheres and the circle as directly visualized results.

There is an almost one-to-one correspondence between formulas and code. The formulas

$$S_1 = P_1 - \frac{1}{2}r_1^2 e_\infty,$$

$$S_2 = P_2 - \frac{1}{2}r_2^2 e_\infty,$$


CLUCalc v3.0.0 by C. Perwass - meetSphereSphereN3.clu
File Edit Search Code Window Help
CLUCalc v3.0.0 by C. Perwass - Output
Edit Window
CLUCalc v3.0.0 by C. Perwass - meetSphereSphereN3
Visualization Mouse Mode Code Presentation Window

```
DefVarsN3();
_BGColor = Color(1,1,1);    // Background white
:OPNS;
:N3_SOLID;

:Red;
:a=*(VecN3(0,-0.5,-0.5)-0.5*e);
:b=*(VecN3(0,0.5,0.5)-0.5*e);

:Blue;
:M=*a.b;
?M;
```

```
M = -0.25^(e31^e) + -0.25^(e12^e) + -1^(e31^e0) + -1^(e12^e0)
```

Fig. 6.1 Interactive and visual development of algorithms using CLUCalc. In the editor window, the intersection of two spheres is defined, which is immediately visualized in the visualization window

and

$$z = S_1 \wedge S_2$$

are coded in CLUCalc as follows:

```
s1 = p1 - 0.5*r1*r1*einf;
s2 = p2 - 0.5*r2*r2*einf;
z = s1^s2;
```

In this chapter, we present a tutorial-like introduction to Geometric Algebra. It can be used by readers interested in a quick, interactive, and visual overview of Geometric Algebra. In order to be usable without preconditions, this tutorial is written more or less independently of Part I of the book.

The tutorial is based on some simple examples highlighting aspects of Geometric Algebra. These examples are meant as starting points for your own experiments. We hope that they will inspire you to make changes interactively and gain your own experience with Geometric Algebra. Each of the figures in this tutorial is related to one of these examples. They were all generated by CLUCalc scripts, indicated in the caption of the figure. All of these short scripts can be easily typed in by yourself or downloaded from

http://www.gaalop.de.

For specific details of CLUCalc, see [81] or the CLUCalc online help in [82].

6.1 Blades and Vectors

The basis vectors $e_1, e_2, \ldots, e_n$ are the basic algebraic elements of an n-dimensional vector algebra. In Geometric Algebra, there are a variety of basic elements called **blades**, including these basis vectors and all combinations of these basis vectors with dimensions 0 to n (this dimension is usually called the **grade** of the blade).

The Geometric Algebra of Euclidean 3D space consists of blades of grade 0, 1, 2, and 3. A scalar is a **0-blade** (a blade of grade 0). The **1-blades** are the three basis vectors e_1, e_2, e_3. The **2-blades** are plane elements spanned by two basis vectors.

The basis vectors (or 1-blades) and the 2-blade $e_1 \wedge e_2$ are visualized in the following CLUScript:

```
DefVarsE3();                 // 3D Euclidean space

:Blue;
:a=e1;
:b=e2;
:c=e3;

:Red;
:PE = e1^e2;
```

Here, **:DefVarsE3();** indicates calculations in 3D Euclidean space. **:Blue;** and **:Red;** mean that the following objects will be drawn in the corresponding color. The leading colon means that the object is not only computed but also visualized.

Figure 6.2 shows the visualization of this script. The basis vectors e_1, e_2, and e_3 are drawn in blue and the 2-blade $e_1 \wedge e_2$, considered as a plane element spanned by the two basis vectors e_1 and e_2, is drawn in red. The additional annotations were produced with specific LaTeX functions according to the CLUScript BasisElementsE3.clu.

Note that for these functions to work, you need to have LaTeX and the software packages dvips and Ghostscript installed. Under a Linux standard distribution, all

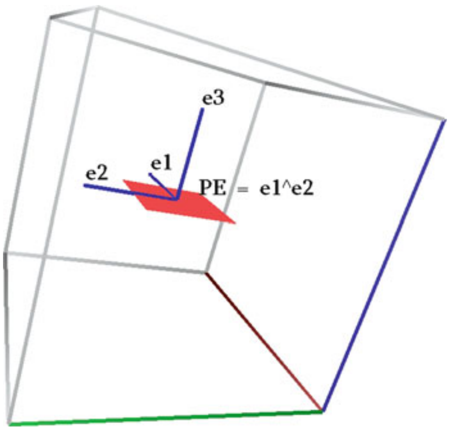


Fig. 6.2
BasisElementsE3.clu

Table 6.1 List of the eight blades of 3D Euclidean Geometric Algebra

Blade	Grade	Abbreviation
1	0	1
e_1	1	e1
e_2	1	e2
e_3	1	e3
$e_1 \wedge e_2$	2	e12
$e_1 \wedge e_3$	2	e13
$e_2 \wedge e_3$	2	e23
$e_1 \wedge e_2 \wedge e_3$	3	I

this software should already be installed. Under Windows, you will need to install MiKTeX, from www.miktex.org (or something comparable), and AFPL Ghostscript v8.13 or later, from www.ghostscript.com. They both come with an installer for Windows and are easy to install.

The Geometric Algebra of Euclidean 3D space also contains a **3-blade** $e_1 \wedge e_2 \wedge e_3$ spanned by all three basis vectors. A linear combination of k-blades is called a k-**vector** (also called vectors, bivectors, trivectors, ...). Table 6.1 lists the eight blades of Geometric Algebra of the Euclidean 3D space, including the scalar, three vectors, three bivectors, and one trivector.

6.2 The Products of Geometric Algebra

Geometric Algebra offers three main products:

- The outer product;
- The inner product;
- The geometric product.

6.2.1 The Outer Product and Parallelness

Geometric Algebra provides an outer product $\wedge$ with the properties listed in Table 6.2.

The outer product is needed in order to generate high-dimensional blades and vectors. Since the outer product of two parallel vectors is 0, it can be used as a measure of parallelness. The outer product of two vectors is a **bivector**. It can be visualized as a plane element spanned by these two vectors. The following CLUScript, bivectorE3.clu (Fig. 6.3), computes and visualizes the bivector c based on the two vectors a and b:

```
DefVarsE3();
:Blue;
:a = e1 + e2;
```

Table 6.2 Properties of the outer product

	Property	Meaning
1.	Anticommutativity	$u \wedge v = -(v \wedge u)$
2.	Distributivity	$u \wedge (v + w) = u \wedge v + u \wedge w$
3.	Associativity	$u \wedge (v \wedge w) = (u \wedge v) \wedge w$

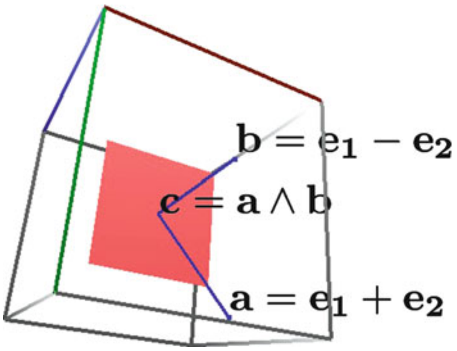

$\mathbf{b} = \mathbf{e_1} - \mathbf{e_2}$
$\mathbf{c} = \mathbf{a} \wedge \mathbf{b}$
$\mathbf{a} = \mathbf{e_1} + \mathbf{e_2}$

Fig. 6.3 bivectorE3.clu

```
:b = e1 - e2;
:Red;
:c = a ^ b;
```

The two vectors $a = e_1 + e_2$ and $b = e_1 - e_2$ are drawn in blue. The result c of taking their outer product (c) is a bivector. It is visualized as a plane element in red.

A question mark in front of a variable indicates that its algebraic representation should be shown in a separate window. So, for instance,

```
?c;  // output in separate window
```

leads to the following result for the bivector c:

```
c  = -2 e12
```

According to Table 6.1, this is the same as $-2(e_1 \wedge e_2)$. The following outer-product computation,

```
?d = a ^ a;
```

leads to

```
d  = 0
```

indicating that the two vectors of the outer product are parallel.

In the following line (of the CLUScript bivectorE3.clu),

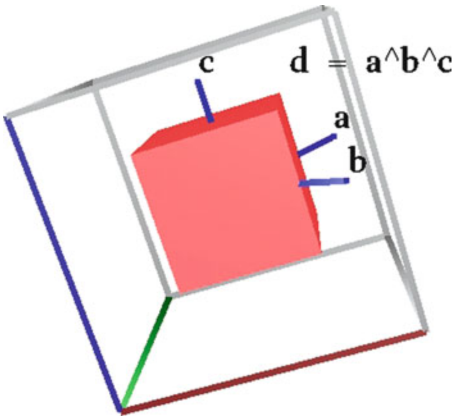
```
?reverse = ~c;
```

the **reverse** of c is computed. This results in

```
reverse  = 2 e12
```

since taking the reverse of a blade simply reverses the order of its basis vectors.

Fig. 6.4 trivectorE3.clu



We can see that the resulting plane element is

- Twice the plane element spanned by the basis vectors e_2 and e_1, or
- Twice the plane element spanned by the basis vectors e_1 and e_2 with an inverted **orientation**.

Note that $\tilde{c}$ (the reverse of c) is equal to

$$\tilde{c} = 2(e_1 \wedge e_2).$$

A **trivector** is a volume element resulting from the outer product of three vectors. The following CLUScript computes and draws a simple trivector in E3:

```
DefVarsE3();

:Blue;
:a = e1 + e2;
:b = e1 - e2;
:c = e3;

:Red;
:d = a ^ b ^ c;
?d;
```

The three vectors a, b, c are drawn in blue and their outer product d in red (Fig. 6.4).

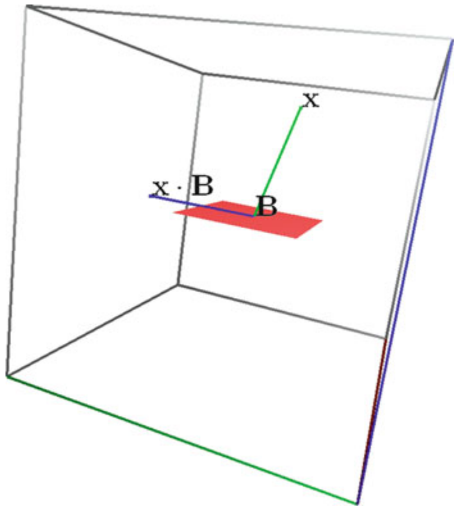
If you are interested in the mathematical computation corresponding to this example, see Sect. 2.2.1, which shows the following result:

$$d = a \wedge b \wedge c = (e_1 + e_2) \wedge (e_1 - e_2) \wedge e_3$$

$$= -2(e_1 \wedge e_2 \wedge e_3) = -2I.$$

This means that the resulting geometric object $a \wedge b \wedge c$ is equal to -2 multiplied by the volume element spanned by the three basis vectors e_1, e_2, e_3. This is often denoted by I, called the **pseudoscalar**.

Fig. 6.5 innerProductE3.clu



6.2.2 The Inner Product and Perpendicularity

Geometric Algebra offers an inner product, denoted by $A \cdot B$ (A.B in CLUScript). For Euclidean spaces, the inner product of two vectors is the same as the well-known Euclidean **scalar product** of two vectors.

The result of the following CLUScript,

```
DefVarsE3();
B = e1+e2;
?length = sqrt(B.B);
```

is

```
length = 1.41421,
```

the length of the vector $e_1 + e_2$.

As in vector algebra, the result of taking the inner product of two basis vectors,

```
DefVarsE3();
?InnerProduct = e1.e2;
```

is

```
InnerProduct = 0,
```

since the two vectors are **perpendicular** to each other.

In Geometric Algebra, the inner product is not only defined for vectors; see Sect. 3.2.7 of [81] for a mathematical treatment. The following CLUScript, innerProductE3.clu (Fig. 6.5), computes and draws the inner product of a vector and a bivector:

```
DefVarsE3();

:Red;
:B = e1 ^ e2;

:Green;
:x = e1+e3;

:Blue;
// xiB is a vector in the B-plane
// perpendicular to x
:xiB = x.B;
```

The result of taking the inner product of the vector $x = e_1 + e_3$ and the bivector B is a vector in the plane (represented by the bivector B) which is **perpendicular** to x.

Remark. the inner product is grade-decreasing; for example, in the example above, the result of taking the inner product of an element of grade 2 and an element of grade 1 is an element of grade $2 - 1 = 1$.

6.2.3 The Geometric Product and Invertibility

The geometric product is a combination of the outer product and the inner product. The geometric product of *u* and *v* is denoted by *uv* (u*v in CLUScript). As we will see, it is an amazingly powerful operation.

6.2.3.1 The Geometric Product of Vectors

For vectors *u* and *v*, the geometric product *uv* is defined as

$$uv = u \wedge v + u \cdot v. \tag{6.1}$$

We can derive the following for the inner and the outer product:

$$u \cdot v = \frac{1}{2}(uv + vu), \tag{6.2}$$

$$u \wedge v = \frac{1}{2}(uv - vu). \tag{6.3}$$

Example 1. What is the **square of a vector**? We have

$$a^2 = aa = a \wedge a + a \cdot a = a \cdot a;$$

for example,

$$e_1 e_1 = e_1 \cdot e_1 = 1.$$

Example 2. What is $(e_1 + e_2)(e_1 + e_2)$?

```
DefVarsE3();
?(e1+e2)*(e1+e2);
```

results in

```
Constant = 2
```

This can be expressed mathematically as

$$(e_1 + e_2)(e_1 + e_2) = (e_1 + e_2) \cdot (e_1 + e_2)$$

$$= e_1 \cdot e_1 + e_1 \cdot e_2 + e_2 \cdot e_1 + e_2 \cdot e_2$$
$$= e_1 \cdot e_1 + e_2 \cdot e_2 = 2.$$

Example 3. What is $e_1 e_2$?

```
DefVarsE3();
?e1*e2;
```

results in

```
Constant = e12
```

This can be expressed mathematically as

$$e_1 e_2 = e_1 \wedge e_2 + e_1 \cdot e_2 = e_1 \wedge e_2.$$

Example 4. What is $e_1(e_1 + e_2)$?

```
DefVarsE3();
?e1*(e1+e2);
```

results in

```
Constant = e12 +1
```

This can be expressed mathematically as

$$e_1(e_1 + e_2) = e_1 e_1 + e_1 e_2 = 1 + e_1 \wedge e_2.$$

Note that the result of this calculation is a linear combination of different types of blades (in this example, a scalar and a bivector). This kind of expression is called a **multivector**.

6.2.3.2 Extension of the Geometric Product to General Multivectors

The geometric product is defined not only for vectors but also for all kinds of multivectors. Let us calculate, for example, the geometric product of two bivectors:

```
DefVarsE3();
:Red;
:B = e1 ^ e2;
?i_square = B*B;
```

The surprising result for the square product B^2 of the bivector $B = e_1 \wedge e_2$ is -1. This is why this bivector can be used like the imaginary unit i of complex numbers.

The geometric product of the following two (unequal) bivectors

```
DefVarsE3();
?(e1^e2)*((e1+e2)^e3);
```

results in the bivector

```
Constant = - e23 - e31
```

See Sect. 2.2 for a proof of this example.

6.2.3.3 Invertibility

The **inverse** of a blade A is defined by

$$AA^{-1} = 1.$$

The inverse of a vector v is

$$v^{-1} = \frac{v}{v \cdot v}.$$

Proof.

$$v\frac{v}{v \cdot v} = \frac{v \cdot v}{v \cdot v} = 1.$$

Example 1. What is the inverse of the vector $v = 2e_1$?

```
DefVarsE3();
:v=2*e1;
? 1/v;
```

results in the CLUCalc output 0.5e1.

Example 2. What is the **inverse of the pseudoscalar**?

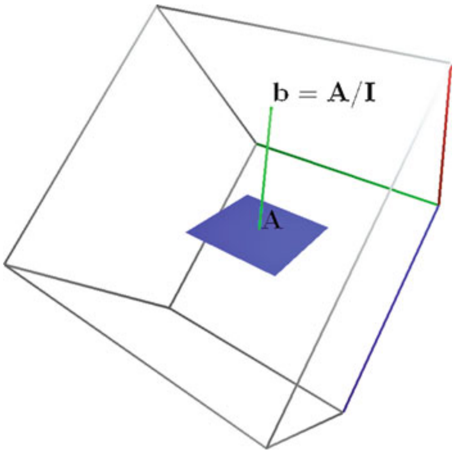
```
DefVarsE3();
? 1/I;
```

results in the negative of the pseudoscalar $(-I)$,

```
Constant = - I
```

See Sect. 2.2 for a proof of this example.

Fig. 6.6 DualE3.clu



6.2.3.4 Duality

Since the geometric product is **invertible**, divisions by geometric objects are possible.

The **dual** of a geometric object is calculated by dividing it by the pseudoscalar I. A superscript is used to denote the dual operator. In CLUScript, this is denoted by a leading asterisk.

In the following CLUScript DualE3.clu (Fig. 6.6) the dual of the plane A is calculated:

```
DefVarsE3();

:Blue;
:A= e2 ^ (e1+e3);

:Green;
:b= A/I;
?b;
```

The resulting vector b,

```
b = e1 - e3
```

corresponds to the normal vector of the plane. See Sect. 2.2 for a proof of this example.

6.3 Geometric Operations

Geometric operations can be expressed easily in Geometric Algebra.

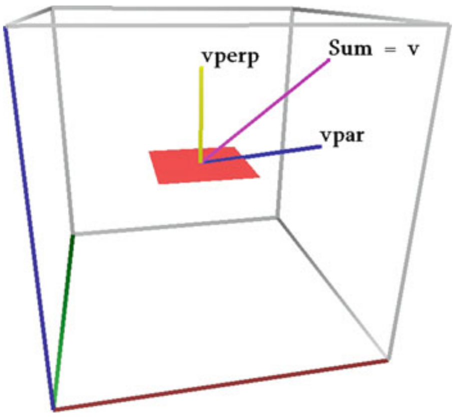


Fig. 6.7 ProjectE3.clu

6.3.1 *Projection and Rejection*

In the following example ProjectE3.clu (Fig. 6.7) we compute and draw the projection and rejection of a vector v onto a plane B.

The projection is calculated with the help of

$$v_{par} = (v \cdot B)/B,$$

and the rejection with the help of

$$v_{perp} = (v \wedge B)/B :$$

```
DefVarsE3();
:Red;
:B = e1^(e1+e2);
v = 1.5*e1 + e2/3 +e3;
```

The plane B and the vector v are computed. The plane B is drawn in red.

Remark. The vector v is computed but not drawn, because of the missing colon.

In the following code

```
:Blue;
:vpar = (v.B)/B;
?vpar;
```

v_{par} is computed as $v_{par} = (v \cdot B)/B$ and drawn in blue. It is the part of v parallel to B. In the following,

```
:Yellow;
:vperp= (v ^ B)/B;
?vperp;
```

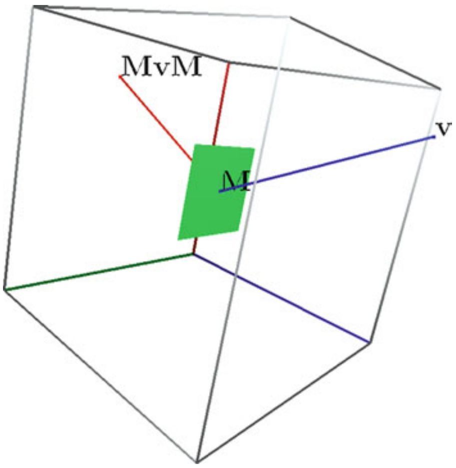


Fig. 6.8 ReflectE3.clu

v_{perp} is computed as $v_{perp} = (v \wedge B)/B$ and drawn in yellow. It is the part of v perpendicular to B. In the following,

```
:Magenta;
:Sum = vpar + vperp;
?Sum;
```

the calculation of Sum (the sum of the two vectors v_{par} and v_{perp}) results in the original vector v.

6.3.2 *Reflection*

The reflection of a vector v from a plane M is defined by

$$v_{refl} = MvM.$$

In the following example ReflectE3.clu (Fig. 6.8) we reflect a vector from a plane.

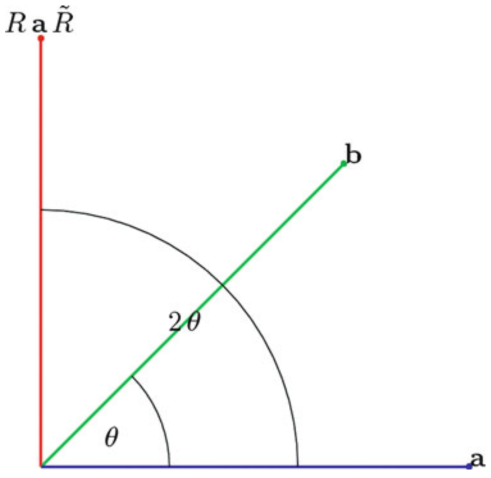
```
DefVarsE3();
:Blue;
:v=e1+2*e3;
:Green;
:M = e1 ^ e2;
```

The vector v is drawn in blue and the plane M in green.

```
:Red;
:vrefl = M*v*M;
?vrefl;
```

With the help of the geometric product MvM, the reflected vector v_{refl} is calculated, drawn, and printed.

Fig. 6.9 Rotor2d.clu



6.3.3 Rotation in 2D

In Geometric Algebra, the geometric product $R := \mathbf{ba}$ of two normalized vectors describes the rotation between these two vectors (by twice the angle between a and b). In the following example Rotor2d.clu (Fig. 6.9) we rotate the vector $\mathbf{a}$ with the help of

$$\mathbf{c} = R\mathbf{a}\tilde{R}.$$

R is called a rotor and $\tilde{R}$ is the reverse of R.

```
DefVarsE3();

:Blue;
:a = e1;

:Green;
:b = 1/sqrt(2)*(e1+e2);
```

The vector a is drawn in blue, and the vector b in green.

```
?R = b*a;
```

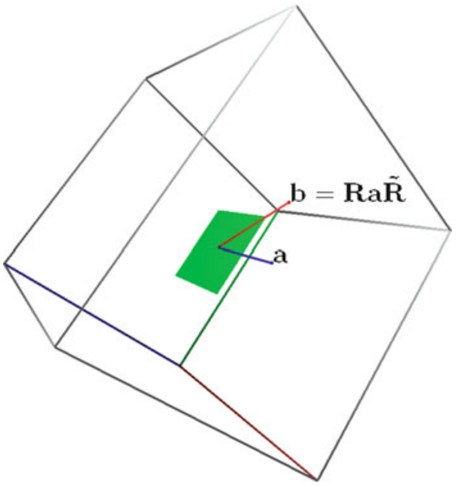
The rotation operator R is calculated as the product of the vector b and the vector a.

```
:Red;
:c = R*a*~R;
```

The rotated vector c is calculated and drawn in red. We see that R rotates a by twice the angle between a and b.

The rotation operator can also be calculated with the help of an exponential function:

Fig. 6.10
Rotate_EXP_E3.clu



```
DefVarsE3();
:Green;
:i = e1 ^ e2;
```

The plane i is drawn in green.

```
R=exp(-i * (Pi/4)/2);
```

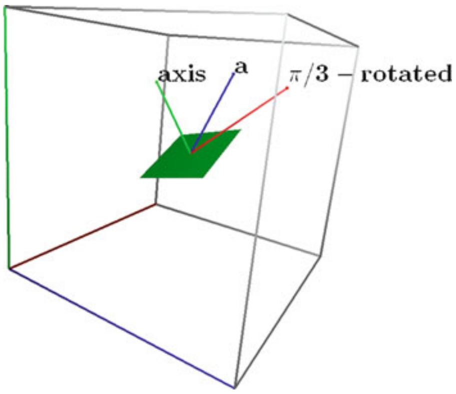
The rotation operator is calculated with the help of i and the specific angle `Pi/4` ($\pi/4$).

```
:Blue;
:a = e1;
:Red;
:b = R*a*~R;
```

The operator $R = e^{-i\frac{\phi}{2}}$, with $i = e_1 \wedge e_2$, can be decomposed as follows (Fig. 6.10). With the help of the Taylor series and the fact that $i^2 = -1$ (see Sect. 6.2.2),

$$
\begin{aligned}
R &= e^{-i\frac{\phi}{2}} \\
&= 1 + \frac{-i\frac{\phi}{2}}{1!} + \frac{(-i\frac{\phi}{2})^2}{2!} + \frac{(-i\frac{\phi}{2})^3}{3!} \\
&+ \frac{(-i\frac{\phi}{2})^4}{4!} + \frac{(-i\frac{\phi}{2})^5}{5!} + \frac{(-i\frac{\phi}{2})^6}{6!} \dots \\
&= 1 - \frac{(\frac{\phi}{2})^2}{2!} + \frac{(\frac{\phi}{2})^4}{4!} - \frac{(\frac{\phi}{2})^6}{6!} \dots \\
&+ -i\frac{\frac{\phi}{2}}{1!} + -i\frac{(\frac{\phi}{2})^3}{3!} - -i\frac{(\frac{\phi}{2})^5}{5!} \dots \\
&= \cos(\frac{\phi}{2}) - i\sin(\frac{\phi}{2})
\end{aligned}
$$

Fig. 6.11 Rotor3d.clu



6.3.4 *Rotation in 3D*

The operator $R = e^{-\frac{\phi}{2}p}$ describes a rotor in 3D with p being a normalized plane. The normal vector (or the dual) of this plane is used as the rotation axis.

In the following example Rotor3d.clu (Fig. 6.11) we rotate the vector **a** with the help of $R\mathbf{a}\tilde{R}$:

```
DefVarsE3();
:Blue;
:a=e1+e2;

:Green;
axis = -3*e1 + 6*e2 - 2*e3;

:axis =axis/sqrt(axis.axis);
:p=*axis;

angle = Pi/3;
?R=exp(-0.5*angle*p);
:Red;
:rot=R*a*~R;
```

The vector a is drawn in blue. The rotated vector c is calculated and drawn in red.

6.4 Conformal Geometric Algebra

Up to now, we have dealt with the well-known Euclidean space. In this section, we will extend our investigations to one specific non-Euclidean space, called conformal space. Conformal Geometric Algebra (CGA) is a five-dimensional Geometric Algebra (see Chap. 3 for details). In this algebra, points, spheres and planes are easily represented as vectors (grade-1 blades).

Table 6.3 Meanings of the coefficients of the two additional coordinates in CGA

	$s_5 = 0$	$s_5 \neq 0$
$s_4 = 0$	Origin plane	Origin sphere/point
$s_4 \neq 0$	Plane	Sphere/point

6.4.1 Vectors in CGA

A vector can be written as

$$S = s_1e_1 + s_2e_2 + s_3e_3 + s_4e_\infty + s_5e_0, \tag{6.4}$$

where the two additional basis vectors e_∞ and e_0 have the following geometric meanings:

- e_0 represents the 3D origin;
- e_∞ represents infinity.

The point $\mathbf{s} = s_1e_1 + s_2e_2 + s_3e_3$ denotes an inhomogeneous point in the Euclidean space. (Note that points $\mathbf{s}$ in bold type in this book mean $\mathbf{s} \in \mathbb{R}^3$.)

The meanings of the coefficients of the two additional coordinates of CGA are listed in Table 6.3.

6.4.1.1 Spheres

A sphere S with an inhomogeneous center point $\mathbf{s}$ and radius r is represented as

$$S = \mathbf{s} + s_4e_\infty + e_0, \tag{6.5}$$

where

$$s_4 = \frac{1}{2}(s_1^2 + s_2^2 + s_3^2 - r^2) = \frac{1}{2}(\mathbf{s}^2 - r^2).$$

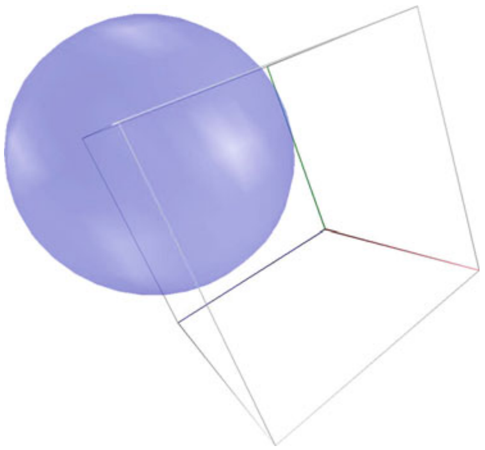
The radius of the sphere is obtained from

$$r^2 = \mathbf{s}^2 - 2s_4 = s_1^2 + s_2^2 + s_3^2 - 2s_4.$$

In the example OneSphereN3.clu (Fig. 6.12),

```
DefVarsN3();
:IPNS;
:N3_SOLID;
:S = e2 +e3 - einf +e0;
```

Fig. 6.12 OneSphereN3.clu



calculation of the radius of the sphere $S = e_2 + e_3 - e_\infty + e_0$ results in

$$r^2 = 1 + 1 - 2 * (-1) = 4.$$

Here, **:DefVarsN3();** indicates conformal-space calculations. **:IPNS;** means that we describe the sphere with the help of the inner product null space (IPNS). The outer product null space (OPNS) would be used if we wished to describe the sphere with the help of its dual representation (quadvector instead of vector). See Sect. 3.2 for more details about the IPNS and OPNS. **:N3_SOLID;** is needed in order to visualize the sphere as a solid sphere instead of a wired one (N3_WIRED).

6.4.1.2 Points

Points are degenerate spheres with radius $r = 0$. An inhomogeneous point $\mathbf{p}$ is represented as

$$X = \mathbf{p} + \frac{1}{2}\mathbf{p}^2 e_\infty + e_0. \tag{6.6}$$

6.4.1.3 Planes

Planes are degenerate spheres with infinite radius. They are represented as a vector with $s_5 = 0$,

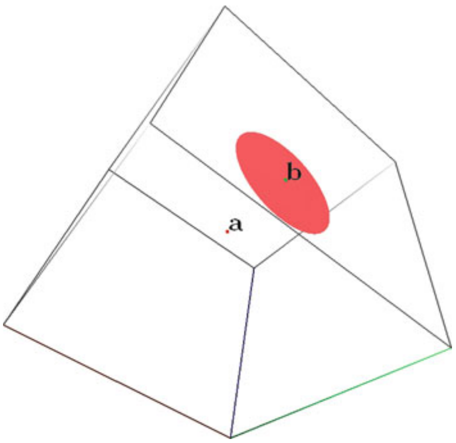
$$Plane = n_1 e_1 + n_2 e_2 + n_3 e_3 + d e_\infty, \tag{6.7}$$

with a normal vector (n_1, n_2, n_3), such that

$$n_1^2 + n_2^2 + n_3^2 = 1,$$

and where d is the distance of the plane from the origin.

Fig. 6.13 PlaneN3.clu



In the following CLUScript PlaneN3 (Fig. 6.13).clu the plane $e_2 + e_\infty$ is drawn in red. The point at infinity e_∞ is indicated by einf.

```
DefVarsN3();
:N3_IPNS;

:Red;
:a=VecN3(0,0,0);
:Plane=e2+einf;
:Green;
:b=VecN3(0,1,0);
```

The normal vector of the plane is $(n_1, n_2, n_3) = (0, 1, 0)$ and the distance is 1 (indicated by the red point a at the origin and the green point b). These points in conformal space are generated by the function **VecN3()**.

6.4.2 Bivectors in CGA

The representation of bivectors of CGA are circles and lines. Lines are circles with infinite radius.

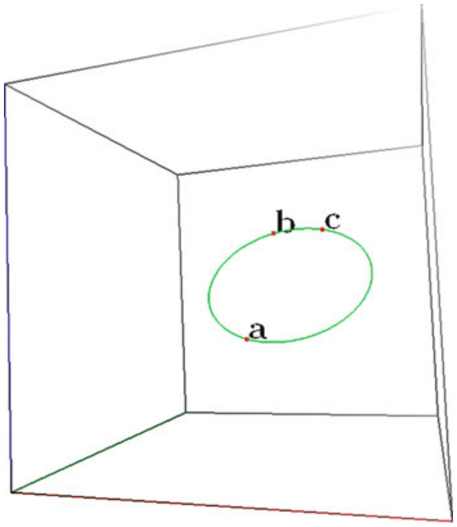
6.4.2.1 Circles

A circle can be defined by three points. Its algebraic description in CGA is the dual of the outer product of these three points.

In the following CLUScript CircleN3.clu (Fig. 6.14) a circle is shown in green, based on the red points a, b, c:

```
DefVarsN3();
:IPNS;
```

Fig. 6.14 CircleN3.clu



```
:Red;
:a=VecN3(0,-0.5,-0.5);
:b=VecN3(0,0.5,0.5);
:c=VecN3(0.5,0.5,0.5);

:Green;
:Circle=*(a^b^c);

?Circle;
```

The resulting bivector is calculated, drawn, and printed.

6.4.2.2 Lines

A line, represented as a degenerate circle with infinite radius, can be defined by two points and the point at infinity. Its algebraic description in CGA is the dual of the outer product of these three points.

In the following CLUScript LineN3.clu (Fig. 6.15) a line is shown in green, based on the red points a, b:

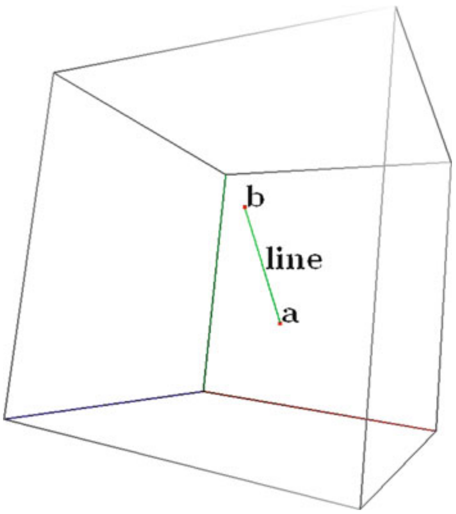
```
DefVarsN3();
:IPNS;

:Red;
:a=VecN3(0,-0.5,-0.5);
:b=VecN3(0,0.5,0.5);

:Green;
:line=* (a ^ b ^ einf );

?line;
```

Fig. 6.15 LineN3.clu



The point at infinity e_∞ is indicated by einf. The resulting bivector is calculated, drawn, and printed.

6.4.3 Dual Vectors in CGA

In Sect. 6.4.2, we saw circles and lines represented as the duals of trivectors based on the outer product of three points. In a similar way we are able to define spheres and planes as the dual of the outer product of four points (in the IPNS representation) or as the outer product of four points (in the OPNS representation). The dual of a vector in CGA is a four-vector (or quadvector).

In the following CLUScript DualSphereN3.clu (Fig. 6.16) a sphere generated by four points is visualized. Please notice that in this script, exceptionally, :OPNS is used instead of :IPNS :

```
DefVarsN3();
:OPNS;
:N3_SOLID;

:Red;
:A=VecN3(-0.5,0,1);

:Blue;
:B=VecN3(1,-0.5,2);

:Green;
:C=VecN3(0,1.5,3);

:Black;
:D=VecN3(0,2,2);
```

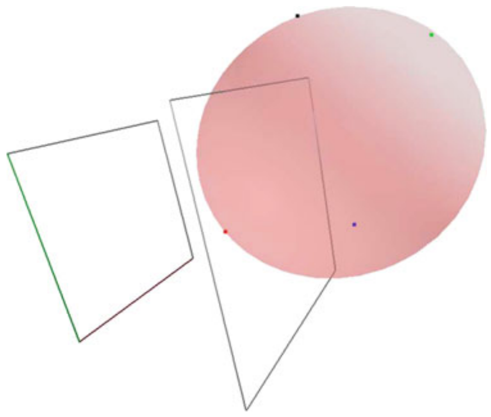


Fig. 6.16 DualSphereN3.clu

```
:Yellow;
:Sphere=A^B^C^D;

?Sphere;
```

The sphere is generated by the outer product of the four points A, B, C, D. These points are indicated by different colors. The resulting quadvector is shown in the output window.

6.4.4 Is a Point Inside or Outside the Circumcircle of a Triangle?

The reader is encouraged to verify that the following CLUScript PointInsideCircleN3.clu (Fig. 6.17) is able to decide whether a point is inside or outside the circumcircle of a triangle:

```
DefVarsN3();

:IPNS;
:N3_SOLID;

:Red;
:A=VecN3(-0.5,0,1);
:Blue;
:B=VecN3(1,-0.5,2);

:Green;
:C=VecN3(0,1.5,3);
:Black;
:X=VecN3(0,4,4);
```

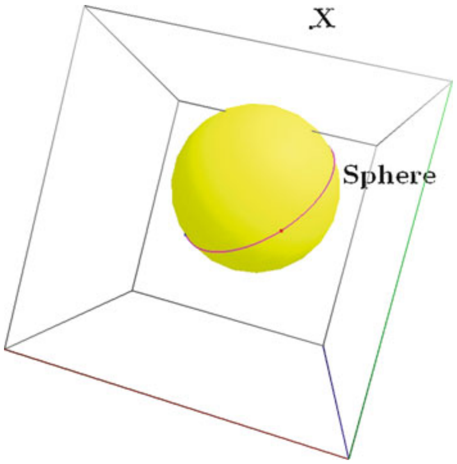


Fig. 6.17
PointInsideCircleN3.clu

```
:Magenta;
:Circle=*(A^B^C);
Plane=*(A^B^C^einf);

:Yellow;
:Sphere=Circle*Plane;
?Distance=Sphere.X;
```

6.4.5 Intersections

If we use the IPNS representation, we are able to use the outer product of various objects such as spheres, lines, and planes to compute their intersections.

6.4.5.1 Intersection of Two Spheres

In the following CLUScript intersectSphereSphereN3.clu (Fig. 6.18) the intersection of two spheres is calculated:

```
DefVarsN3();

:IPNS;
:N3_SOLID;

:Red;
:s1=VecN3(0,-0.5,-0.5)-0.5*einf;
:s2=VecN3(0,0.5,0.5)-0.5*einf;
?C=s1^s2;

:Blue;
:C;
```

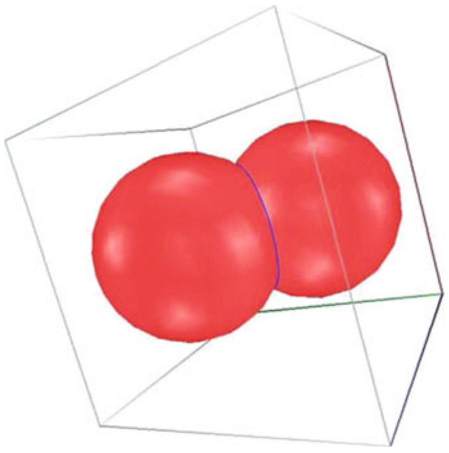


Fig. 6.18
intersectSphereSphereN3.clu

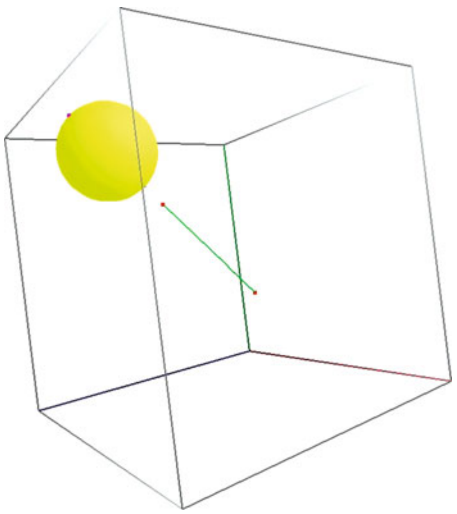


Fig. 6.19
intersectSphereLineN3.clu

Two spheres, defined as vectors, are drawn in red. The intersection of these spheres is calculated with the help of the outer product, and the resulting circle is drawn in blue.

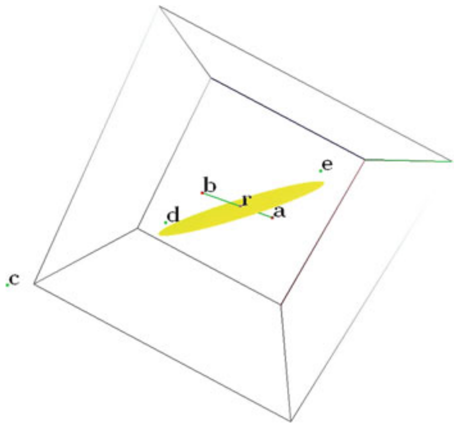
6.4.5.2 Intersection of a Line and a Sphere

In the following CLUScript intersectSphereLineN3.clu (Fig. 6.19) the intersection of a sphere s and a line l is calculated:

```
DefVarsN3();

:IPNS;
:N3_SOLID;
```

Fig. 6.20
intersectPlaneLineN3.clu



```
:Red;
:a=VecN3(0,-0.5,-0.5);
:b=VecN3(0,0.5,0.5);

:Green;
:l=*(a^b^einf);
?l;

:Yellow;
:s=VecN3(0,1,1)-0.1*einf;

:Magenta;
:r=s^l;
```

The intersection of the line l (defined by the points a and b) and the sphere s is a **point pair**. This geometric object is visualized in magenta. A point pair is a trivector in CGA.

6.4.5.3 Intersection of a Line and a Plane

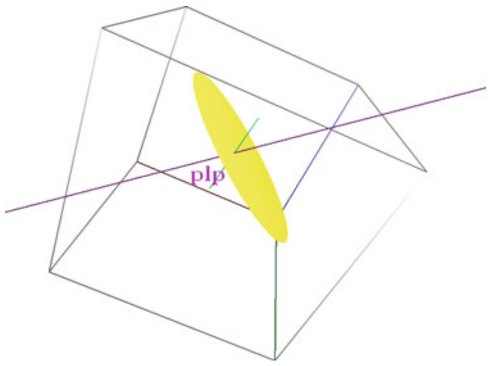
In the following CLUScript intersectPlaneLineN3.clu (Fig. 6.20) the intersection of a plane p and a line l is calculated with the help of the outer product:

```
DefVarsN3();
:IPNS;

:Red;
:a=VecN3(0,-0.5,-0.5);
:b=VecN3(0,0.5,0.5);

:Green;
:l=*(a^b^einf);
?l;
```

Fig. 6.21 ReflectN3.clu



```
:c=VecN3(2,1,2);
:d=VecN3(1,-1,1);
:ev=VecN3(-1,-2,-1);

:Yellow;
:p=*(c^d^ev^einf);

:Magenta;
:r=p^l;
?r;
```

The plane p is defined with the help of the three points c, d, ev and the point at infinity e_∞. The intersection point r with the line l (defined with the help of a, b, e_∞) is visualized in magenta.

6.4.6 Reflection

In the following CLUScript ReflectN3.clu (Fig. 6.21) we visualize the reflection of a line **l** in a plane **p** with the help of the operation **plp**:

```
DefVarsN3();
:IPNS;

a=VecN3(0,-0.5,-0.5);
b=VecN3(0,2,2);

:Green;
:l=*(a^b^einf);
?l;

c=VecN3(2,1,2);
d=VecN3(1,-1,1);
e_=VecN3(-1.5,-2,-1);
```

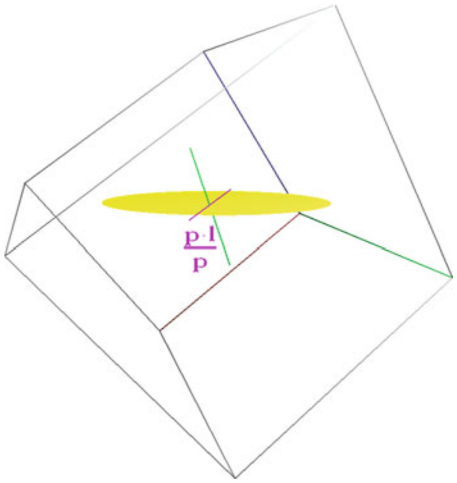


Fig. 6.22 ProjectN3.clu

```
:Yellow;
:p=*(c^d^e_^einf);

:Magenta;
:r=p*l*p;
?r;
```

The result is one reflected line, drawn in magenta.

6.4.7 Projection

In the following CLUScript ProjectN3.clu (Fig. 6.22) we visualize the projection of a line onto a plane with the help of the operation $\frac{\mathbf{p} \cdot \mathbf{l}}{\mathbf{p}}$:

```
DefVarsN3();
:IPNS;

a=VecN3(0,-0.5,-0.5);
b=VecN3(0,2,2);

:Green;
:l=*(a^b^einf);
?l;

c=VecN3(2,1,2);
d=VecN3(1,-1,1);
e_=VecN3(-1.5,-2,-1);

:Yellow;
:p=*(c^d^e_^einf);
```

```
:Magenta;
:r=(p.l)/p;
?r;
```

The result is the projected line, drawn in magenta.

6.5 CLUCalc Implementation of the Horizon Example

The following CLUScript visualizes the horizon example of Sect. 3.7. It computes the horizon circle on the Earth S as seen from a viewpoint P:

```
:Black;
:P = VecN3(px,py,pz);     // viewpoint
 M = VecN3(mx,my,mz);     // center point of earth
:Blue;
:S = M-0.5*r*r*einf;      // sphere representing earth
:Color(0,1,0,0.2);
:K = P+(P.S)*einf;        // sphere around P
:Red;
:C=S^K;                   // intersection circle
```

The variables **px**, **py**, **pz** and **mx**, **my**, **mz** and the radius **r** are free variables that have to be assigned before the script is run. The function **VecN3** computes the conformal points from the 3D points for the viewpoint and the center of the Earth. The sphere K is computed according to (3.68) and the horizon circle is computed as the intersection of S and K.

The leading colons cause the visualization of the geometric objects according to the colors that have been defined (see Fig. 6.23). Note that the **Color(R,G,B,A)** function defines the RGB (red, green, blue) values and one translucency color component A, all ranging between 0 and 1.

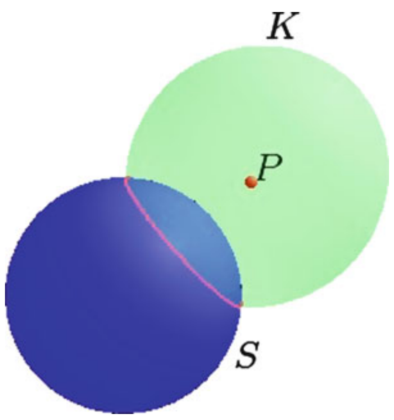


Fig. 6.23 Visualization of the horizon example

6.6 CLUCalc Implementation of Motions

CLUScripts can also be animated. The following script describes motion according to Sect. 3.5. In this example we rotate the blue sphere *Earth* around the yellow sphere *Sun* located at the origin (see Fig. 6.24).

The script begins as follows:

```
_DoAnimate = 1;
```

This script is animated (see the online help of CLUCalc for details). The sphere *Earth* is rotated continuously according to a continuously changing angle. This angle is computed depending on the elapsed time:

```
DefVarsN3();
angle = ((Time * 45) % 360) * RadPerDeg;

:IPNS;
:N3_SOLID;
:Red;
:a = VecN3(0,-2,0);
:b=VecN3(0,2,0);

:Green;
axis = *(a^b^einf);
:axis;
axis=axis/abs(axis);
?axis;
```

:a = VecN3(0,–2,0); assigns the five-dimensional representation of a three-dimensional point to the variable *a* according to Table 3.2. With the help of **axis = *(a** $\wedge$ **b** $\wedge$ **einf);** a bivector representing a line *axis* is computed. According to Table 3.2 the dual representation of a line is the outer product of two points and e_∞, the point at infinity. The resulting bivector, after dualization, is normalized with the help of the *abs* function, and visualized and printed.

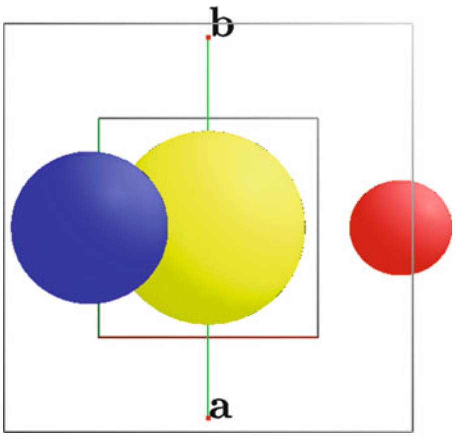


Fig. 6.24 Visualization of a CLUScript describing motion

```
:Yellow;
:Sun = e0 -0.5*einf;

:Red;
:Earth =VecN3(2,0,0)-0.125*einf;

?R = exp(-angle/2*axis);

:Blue;
:R * Earth * ~R;
```

Sun is centered at the origin e_0, with radius $r = 1$ (see Table 3.2). It is drawn as a yellow sphere. The red sphere is used as the basis sphere for the rotation of $Earth$. It is located away from the origin and has half the radius of Sun. The blue sphere representing the Earth is rotated with the help of the product R Earth $\tilde{R}$ (see Sect. 3.5). The rotation operator depends on the fixed $axis$ and the continuously changing $angle$. See CLUCalc online help [82] for further details of CLUScript.

The inverse kinematics algorithm described in Chap. 7, and the grasping algorithm in Chap. 8 were implemented using CLUCalc. The corresponding CLUScripts can be downloaded from the home page

```
http://www.gaalop.de
```

These example scripts show how easy it is to develop algorithms based on CGA.

Chapter 7
Inverse Kinematics of a Simple Robot

This chapter presents the inverse kinematics application of a simple robot. All the figures in this chapter are screenshots of a CLUCalc application that can be downloaded from

`http://www.gaalop.de.`

We present the Geometric Algebra algorithm for this application step by step. The geometrically intuitive operations of CGA make it easy to compute the joint angles of this robot that have to be set in order for the robot to reach its new position.

Objects such as robots and virtual humans (see Chap. 9) can be modeled as a set of rigid links connected together at various joints. These objects are described as kinematic chains.

The robot shown in Fig. 7.1 consists of three links and one gripper:

- The three joint points are denoted by P_0, P_1, and P_2;
- The three link distances are denoted by d_1, d_2, and d_3;
- The distance from the last joint P_2 to the "T" intersection of the gripper is denoted by d_4.

The robot has five degrees of freedom obtained by means of the following five joint angles $\theta_1, \ldots, \theta_5$:

- θ_1, rotation of the robot (around L_y);
- $\theta_2, \theta_3, \theta_4$, acting in the plane π_1 (the blue plane in Figs. 7.3–7.6);
- θ_5, rotation of the gripper.

The plane π_1 is defined by the origin e_0, the point P_y (see (3.11)) on the y-axis, and the target point P_t (see Fig. 7.2).

According to (3.16) we get

$$\pi_1^* = e_0 \wedge P_y \wedge P_t \wedge e_\infty. \tag{7.1}$$

Our goal is to find the joint angles in terms of the target position P_t and the orientation of the gripper plane π_t (the red plane in Fig. 7.2). In CGA, this inverse

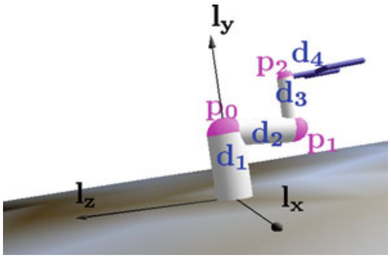


Fig. 7.1 Kinematic chain of the example robot

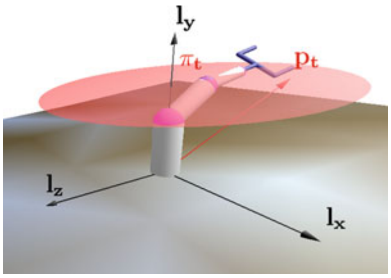


Fig. 7.2 Target point and gripper plane

kinematics problem can be solved in a geometrically very intuitive way owing to the easy handling of intersections of spheres, circles, and planes, etc. Our approach is based on the papers [13, 53].

For convenience, we define the gripper plane π_t as being parallel to the ground plane. Since a plane can be described using (3.15), we get

$$\pi_t = e_2 + P_{t,y}\; e_\infty, \tag{7.2}$$

where $P_{t,y}$ is the y-coordinate of the target point P_t.

In the following steps, we first calculate the three locations P_0, P_1, P_2. Based on these points, we will be able to calculate the five joint angles $\theta_1, \ldots, \theta_5$.

7.1 Computation of P_0

In the first step, the point P_0 is calculated (Fig. 7.3). Its 3D representation is $(0, d_1, 0)$. Using (3.8), we get

$$P_0 = d_1 e_2 + \frac{1}{2} d_1^2 e_\infty + e_0. \tag{7.3}$$

The CLUCalc code of this step is as follows:

```
p0 = VecN3(0,d1,0);
PIt = e2 + pty*einf;
PI1_DUAL = e0 ^ pt ^ p0 ^ einf;
PI1 = *PI1_DUAL;
```

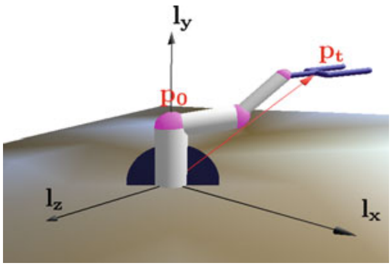


Fig. 7.3 Computation of P_0

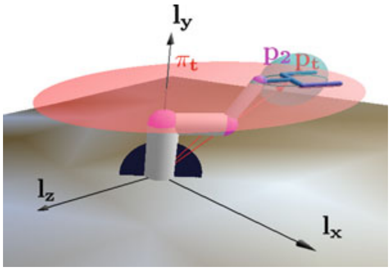


Fig. 7.4 Computation of P_2

7.2 Computation of P_2

In the second step, the point P_2 is calculated (Fig. 7.4). This is the location of the joint with the last link of the robot. This means that it has to lie on the sphere S_t with center point P_t and with the length d_4 of the displacement between P_t and P_2 as the radius. Using (3.12), we get

$$S_t = P_t - \frac{1}{2} d_4^2 e_\infty. \tag{7.4}$$

Since the gripper also has to lie in the orientation plane π_t, we have to intersect this plane with S_t. The result is the circle Z_t (see (3.17)),

$$Z_t = S_t \wedge \pi_t. \tag{7.5}$$

Remember that a plane is simply a sphere of infinite radius.

Since P_2 also has to lie in the plane π_1, the intersection of π_1 with the circle Z_t results in a point pair (see (3.22))

$$Pp_2 = Z_t \wedge \pi_1. \tag{7.6}$$

From the point of view of mechanics, only one of these two points is applicable, which we choose as our point P_2 (we use (3.24) to extract the two points of the point pair).

The CLUCalc code of this step is as follows:

```
St = pt - 0.5*d4*d4*einf;
zt = St^PIt;

Pp2= *(PI1 ^ zt);
// choose one of the two points
p2 = DissectFirst(Pp2);
```

7.3 Computation of P_1

In the third step, the point P_1 is calculated (Fig. 7.5).

Computing this point is usually a difficult task, because it is the intersection of two circles. However, using CGA we can determine it by intersecting the spheres S_1 and S_2 with the plane π_1:

$$S_1 = P_0 - \frac{1}{2}d_2^2 e_\infty, \tag{7.7}$$

$$S_2 = P_2 - \frac{1}{2}d_3^2 e_\infty, \tag{7.8}$$

and

$$Pp_1 = S_1 \wedge S_2 \wedge \pi_1. \tag{7.9}$$

Again, we have to choose one point from the resulting point pair.

The CLUCalc code of this step is as follows:

```
s1 = p0 - 0.5*d2*d2*einf;
s2 = p2 - 0.5*d3*d3*einf;
Pp1 = s1^s2^PI1;
// choose one of the two points
p1 = DissectSecond(*Pp1);
```

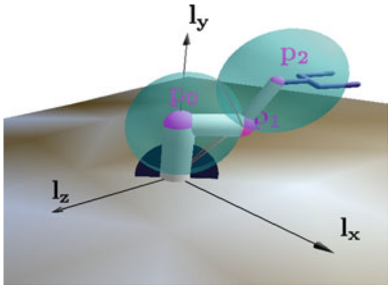


Fig. 7.5 Computation of P_1

7.4 Computation of the Joint Angles

First, all of the auxiliary planes and lines that are needed for the computation of the angles of the joints are calculated. We need the following:

- The plane π_2 (orange in Fig. 7.6) spanned by the x-axis and the y-axis. Since the z-axis is perpendicular to this plane, we get

$$\pi_2 = e_3. \tag{7.10}$$

- The (blue) line L_1 through P_0 and P_1,

$$L_1^* = P_0 \wedge P_1 \wedge e_\infty. \tag{7.11}$$

- The (green) line L_2 through P_1 and P_2,

$$L_2^* = P_1 \wedge P_2 \wedge e_\infty. \tag{7.12}$$

- The (magenta) line L_3 through P_2 and P_t,

$$L_3^* = P_2 \wedge P_t \wedge e_\infty. \tag{7.13}$$

The CLUCalc code of this step is as follows:

```
PI2_DUAL = e3 * I;
l1_dual = p0^p1^einf;
l2_dual = p1^p2^einf;
l3_dual = p2 ^ pt ^ einf;
```

Now, we are able to compute all the joint angles

$$\theta_1 = \angle(\pi_1, \pi_2), \tag{7.14}$$

$$\theta_2 = \angle(L_1, L_y), \tag{7.15}$$

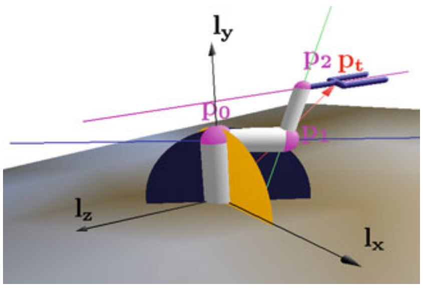


Fig. 7.6 Visualization of step 4

$$\theta_3 = \angle(L_1, L_2), \tag{7.16}$$

$$\theta_4 = \angle(L_2, L_3), \tag{7.17}$$

using (3.48), where o_1, o_2 are either two lines or two planes. In our simplified example,

$$\theta_5 = 0 \tag{7.18}$$

since the gripper should be parallel to the ground plane.

See [61] for an application to the inverse kinematics of a humanoid robot. For more details concerning inverse kinematics algorithms based on CGA, see [7].

Chapter 8
Robot Grasping an Object

In this chapter, we present a Geometric Algebra algorithm for the grasping process of the robot Geometer (Fig. 8.1) constructed at Cinvestav, Guadalajara [7, 62]. We present both the Geometric Algebra algorithm and the algorithm in standard mathematics in order to highlight the difference in the symbolic descriptions. Most of the figures are screenshots of a CLUCalc application that can be downloaded from

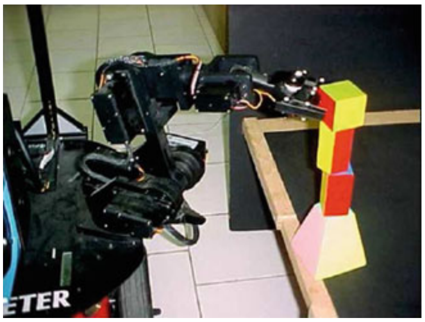
```
http://www.gaalop.de.
```

8.1 The Geometric Algebra Algorithm

The goal of our grasping algorithm is to move a gripper to an object, where the gripper is modeled by a circle and the object is modeled by a cylinder. First we compute the grasping circle Z_t of the object, second, the gripper circle Z_h is estimated, and third, the translation and rotation required for the movement are computed.

8.1.1 Computation of the Bounding Volume of the Object

First of all, we compute a cylinder that is the bounding volume of the object to be grasped. We need four points identifying this object. In a real application, they would be taken from a calibrated stereo pair of images of the object. In order to assign these points, we compute the distance between each of the four points and the plane spanned by the other three points. The point with the greatest distance d_a is called the apex point x_a (see the visualization in Fig. 8.2). The other three points are called the base points $x_{b_1}, x_{b_2}, x_{b_3}$.



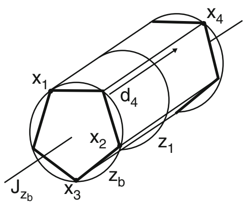
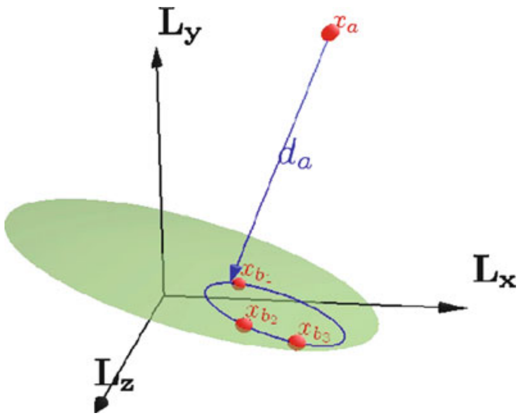


Fig. 8.1 The robot Geometer grasping an object

Fig. 8.2 Assigning points for the bounding cylinder of the object to be grasped



8.1.2 Computation of the Grasping Circle Z_t

To compute the grasping plane π_t, we compute the base circle

$$Z_b^* = x_{b_1} \wedge x_{b_2} \wedge x_{b_3}, \tag{8.1}$$

based on the outer product of the three base points (see Table 3.2). We expect to grasp the object near its center, and therefore we translate the circle Z_b in the direction of the 3D vector $\frac{d_a}{2}$ by the magnitude of that vector. Then, with the help of the translator

$$T = 1 + \frac{1}{4} d_a e_\infty, \tag{8.2}$$

we compute the grasping circle Z_t (see the visualization in Fig. 8.3),

$$Z_t = T Z_b \tilde{T}. \tag{8.3}$$

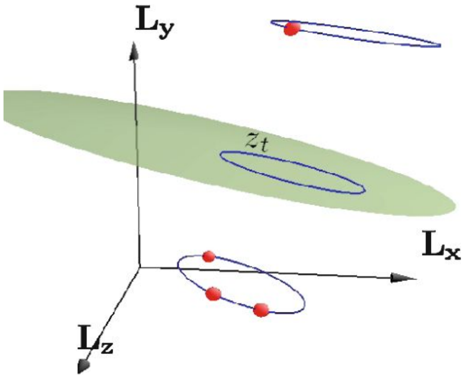


Fig. 8.3 Grasping circle Z_t

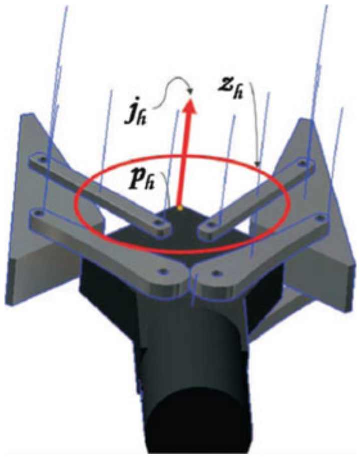


Fig. 8.4 Gripper

8.1.3 Gripper Circle

While the gripper circle of our simulation is computed by default (see the visualization in Fig. 8.5), the gripper circle of the real robot is estimated by tracking its metallic screws. Figure 8.4 shows the position of the point P_h. We create a sphere with center point P_h and radius r (equal to the width of the center of the aperture of the gripper) according to Table 3.2

$$S_h = P_h - \frac{1}{2}r^2 e_\infty. \tag{8.4}$$

Now, by tracking two additional points a and b on the gripper, we create the plane π_h:

$$\pi_h^* = P_h \wedge a \wedge b \wedge e_\infty \tag{8.5}$$

(note that a plane is created here with the help of the outer product of three points and the point at infinity).

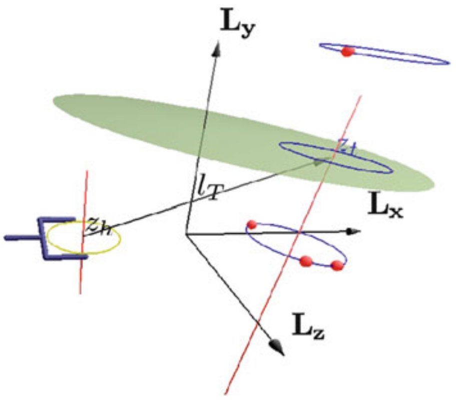


Fig. 8.5 Gripper circle Z_h, grasping circle Z_t and their axes L_h and L_t

Finally, we calculate the gripper circle as the intersection of the sphere and the plane:

$$Z_h = S_h \wedge \pi_h. \tag{8.6}$$

8.1.4 Estimation of Translation and Rotation

Now, we compute the transformation needed to move the gripper circle to the grasping circle. The translation axis can be computed easily from the centers of the circles. The center point P of a circle Z can be computed easily with the help of a sandwich product:

$$P = Ze_\infty Z \tag{8.7}$$

according to Sect. 3.3. The distance d between the circles can be computed with the help of

$$l_T^* = P_h \wedge P_t \wedge e_\infty. \tag{8.8}$$

It is given by $d = |l_T^*|$. The rotation axis is computed using the axes of the circles L_h^* and L_t^* (see the red lines in Fig. 8.5):

$$L_h^* = Z_h \wedge e_\infty, \tag{8.9}$$

$$L_t^* = Z_t \wedge e_\infty. \tag{8.10}$$

These axes L_h and L_t lead to the plane π_{th}^* given by

$$\pi_{th}^* = L_t^* \wedge (L_h^* E), \qquad E = e_0 \wedge e_\infty, \tag{8.11}$$

and therefore the rotation axis is

$$L_r^* = P_h \wedge \pi_{th} \wedge e_\infty. \tag{8.12}$$

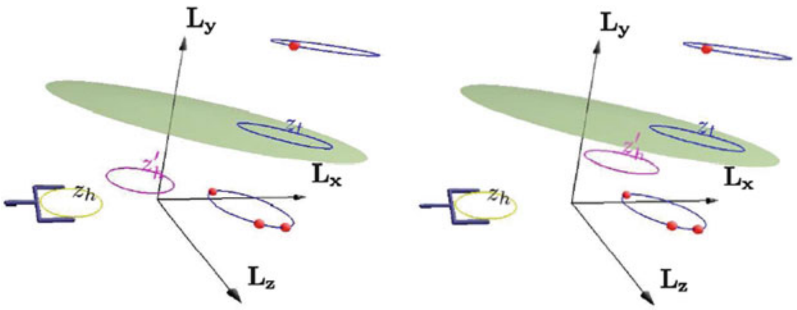


Fig. 8.6 Moving the gripper circle Z_h towards the grasping circle Z_t

The angle θ between the two circles can be computed based on the inner product of the two lines L_h^* and L_t^* (see the visualization in Fig. 8.5):

$$\cos(\theta) = \frac{L_t^* \cdot L_h^*}{|L_t^*||L_h^*|}. \tag{8.13}$$

Once we have estimated the rotation and translation axes (see Sect. 3.5),

$$R = e^{-\frac{1}{2}\Delta\theta L_r}, \tag{8.14}$$

$$T = 1 + \frac{1}{2}\Delta d L_t e_\infty, \tag{8.15}$$

we are able to move the gripper circle Z_h step by step (z_h') towards the grasping circle Z_t (see the visualization in Fig. 8.6):

$$z_h' = TRZ_h\widetilde{R}\widetilde{T}. \tag{8.16}$$

For more details of algorithms for grasping, kinematics, and dynamics see [7].

8.2 The Algorithm Using CLUCalc

A very basic version of the CLUCalc algorithm is given in Fig. 8.7, with the input parameters as described in Fig. 8.8. These parameters include the points x_{b_1}, x_{b_2}, x_{b_3} and x_a (see Fig. 8.10) and the center point of the gripper sphere ($g1, g2, g3$) and its radius $g4$.

As described in Sect. 8.1 we start with the construction of the base circle that represents the bottom of the object. One way to compute a circle in Geometric Algebra is to construct the outer product of three points that lie on that circle. So, we can use three base points to construct the base circle according to (8.1).

```
z_b_d = xb1 ^ xb2 ^ xb3;
?z_b = *z_b_d;
?Pi_b = *(z_b_d ^ einf);
NVector = (Pi_b * einf).e0;
NLength = abs(NVector);
NVector = NVector/NLength;
Plane = Pi_b/NLength;
d_a=(xa.Plane)*NVector;
?T = 1 + 0.25 * d_a * einf;
?z_t = T * z_b * ~T;
S_h = VecN3(g1,g2,g3)
      - 0.5*(g4*g4)*einf;
Pi_h = -e2;
?z_h = S_h ^ Pi_h;
s_h = -0.5*S_h*einf*S_h;
S_t = *z_t / ((*z_t) ^ einf);
s_t = -0.5*S_t*einf*S_t;
?l_T = s_h ^ s_t ^ einf;
?d = abs(l_T);
?l_T = l_T / d;
l_h = z_h ^ einf;
l_t = z_t ^ einf;
?Pi_th = l_t ^ (l_h*(einf^e0));
l_R_d = s_h ^ (*Pi_th) ^ einf;
?l_R = *l_R_d / abs(l_R_d);
?phi = acos((l_t.l_h)
       / (abs(l_t)*abs(l_h)));
?R = cos(0.5*phi) - l_R*sin(0.5*phi);
?T = 1 + 0.5*d*l_T*einf;
?z_h_new = T*R*z_h*~R*~T;
```

Fig. 8.7 CLUCalc code of the grasping algorithm

```
g1 = -1.15;
g2 = 0;
g3 = 0;
g4 = 0.6;
:xb1 = VecN3(0.6, 0.2, 0.2);
:xb2 = VecN3(0.7, 0.2, 1.0);
:xb3 = VecN3(1.0, 0.15, 1.0);
:xa  = VecN3(1.1, 1.9, 0.2);
```

Fig. 8.8 Input parameters of the CLUCalc algorithm

```
z_b_d = xb1 ^ xb2 ^ xb3;
:z_b = *z_b_d;
```

Fig. 8.9 Construction and visualization of the base circle z_b

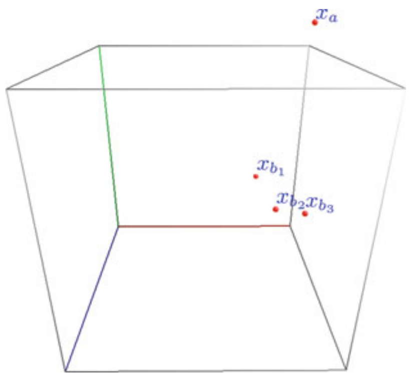


Fig. 8.10 Base points

Because we would like to visualize the circle and use the IPNS in our CLUScripts by default, we have to dualize the circle z_b_d. Later on in the algorithm, we have to use z_b_d for some computations, so we construct both representations here.

Fig. 8.11 Base circle

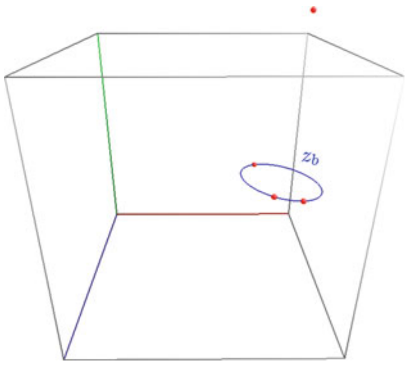


Fig. 8.12 Translation vector

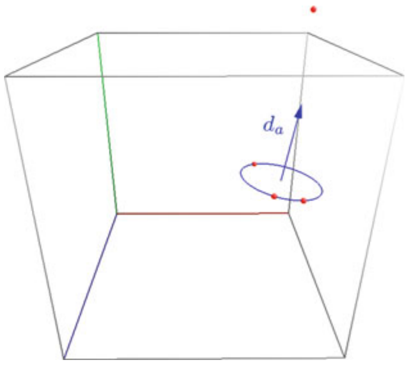
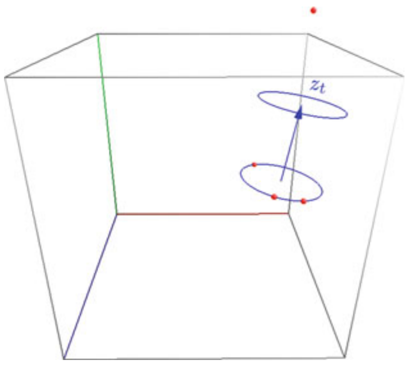


Fig. 8.13 Target circle



Finally, we put a colon in front of the line and get the code listed in Fig. 8.9. This leads to the situation shown in Fig. 8.11.

To construct the target circle `z_t`, we have to translate `z_b` along the vector shown in Fig. 8.12. The necessary translator is computed in the lines 3–9 in Fig. 8.7. Finally, the translation is done in line 10. Putting a colon in place of the question mark leads to Fig. 8.13.

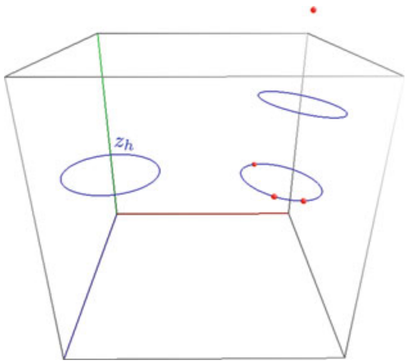


Fig. 8.14 Gripper circle

Now that we know the final position of the gripper, we have to compute the necessary translation and rotation. For this, we have to estimate the current position of the gripper. This is done by extracting the center of the gripper from the stereo image and approximating it by constructing the circumscribed sphere `S_h`. The gripper sphere is then intersected with the plane `Pi_h` that the gripper lies in to construct the gripper circle `z_h`. In the code used here, we assume the gripper to be parallel to the floor. In a real application, the plane has to be constructed using two additional points on the gripper. Now we have the situation depicted in Fig. 8.14 and are ready to proceed to the computation of the final translation and rotation.

To compute the final translation, we have to compute the translation axis `l_T`, which passes through the center points of the gripper circle and the target circle. This is done in lines 14–17, where first the center points `s_h` and `s_t` are computed and then they are used to construct the axis. For us to be able to construct the wanted translator, the axis has to be normalized, which is done in the following two lines.

The next thing that we have to compute is the rotation axis. This is a little more complicated because the rotation axis is perpendicular to a plane that is itself perpendicular to both the gripper circle and the target circle.

For this, we compute the two axes of the circles in lines 20 and 21 and construct a plane from them in line 22. With this plane, we are able to compute the rotation axis perpendicular to it through the center of the gripper circle, and normalize it. Finally, we compute the rotation angle between the two axes of the circles and construct the rotor.

Now all that is left to be done is the computation of the new position of the gripper circle. This is done in line 28, where the translator and rotor are used to move the gripper to its new position. This leads finally to Fig. 8.15. To animate the script and show an actual movement of the gripper circle along a linear path, one could use only fractions of `d` and `phi` in the construction of `T` and `R`, respectively.

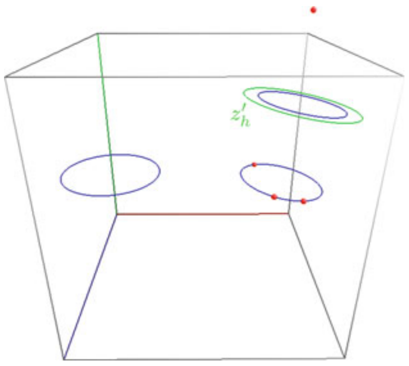


Fig. 8.15 Final position

8.3 Geometric Algebra Versus Conventional Mathematics

In [112], a comparison was made between the performance of the grasping algorithm of this chapter using on the one hand Geometric Algebra and on the other hand standard vector algebra with matrix calculations. Using Gaalop (see Chap. 10), both the CPU implementation and the CUDA implementation were faster than an implementation based on conventional linear algebra.

The high potential of Geometric Algebra for very efficient implementations in terms of runtime performance will be discussed in detail in Part III of this book. Here, we will discuss mainly the differences between the two mathematical descriptions.

8.3.1 *The Base Circle and Its Center*

Whereas in CGA the base circle can easily be computed based on three points of it using

$$z_b^* = x_{b_1} \wedge x_{b_2} \wedge x_{b_3}$$

and can therefore be described using only one algebraic expression, there is no explicit description of a circle in vector algebra. And, whereas in CGA the center of the circle can easily be computed using the sandwich product

$$z_b e_\infty z_b,$$

in vector algebra the computation has to be performed in the following steps.

The circle z_b is the circumscribed circle of the triangle Δ_b which is formed by the three base points. To compute its center, we have to construct two perpendicular bisectors; the intersection of them is the center p_b of z_b. First we compute the center points m_{12} and m_{13} of two sides of Δ_b:

$$l_{12} = x_{b2} - x_{b1}, \tag{8.17}$$

$$l_{13} = x_{b3} - x_{b1}, \tag{8.18}$$

$$m_{12} = \frac{1}{2}(x_{b1} + x_{b2}), \tag{8.19}$$

$$m_{13} = \frac{1}{2}(x_{b1} + x_{b3}). \tag{8.20}$$

Next we have to compute the direction vectors d_{12} and d_{13} that are needed to construct the perpendicular bisectors. For this, we need the normal vector n_b of the plane formed by the base points:

$$n_b = l_{12} \times l_{13}, \tag{8.21}$$

$$d_{12} = l_{12} \times n_b, \tag{8.22}$$

$$d_{13} = l_{13} \times n_b. \tag{8.23}$$

From this, we are able to compute the perpendicular bisectors and their intersection p_b:

$$pb_{12} = m_{12} + \lambda_{12} \cdot d_{12},\ \lambda_{12} \in \mathbb{R}, \tag{8.24}$$

$$pb_{13} = m_{13} + \lambda_{13} \cdot d_{13},\ \lambda_{13} \in \mathbb{R}, \tag{8.25}$$

$$p_b = m_{12} + \lambda_{12_S} \cdot d_{12} = m_{13} + \lambda_{13_S} \cdot d_{13}. \tag{8.26}$$

With the help of (8.26), we can compute the parameters λ_{12_S}, λ_{13_S}, and finally p_b.

Remark. Using vector algebra, we have to take care of three independent parts to describe a circle, namely its center p_b, its normal vector n_b and its radius. Also, we have to solve the equation system in the last line to get p_b.

8.3.2 The Transformation of the Gripper

Whereas in Geometric Algebra transformations are described based on elements of the algebra, namely versors such as

$$T = 1 + \frac{1}{2} d\, l_T\, e_\infty,$$

$$R = \cos\left(\frac{1}{2}\phi\right) - l_R \sin\left(\frac{1}{2}\phi\right),$$

in vector algebra matrices are needed to describe transformations.

To perform the final rotation in vector algebra, we have to normalize the rotation vector $n_{th} = (n_1, n_2, n_3)$, translate the rotation axes to the origin using RT_{orig}, compute the rotation R, and finally translate the axes back using RT_{back}:

$$RT_{orig} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_{h_1} & -p_{h_2} & -p_{h_3} & 1 \end{bmatrix} \tag{8.27}$$

$$RT_{back} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p_{h_1} & p_{h_2} & p_{h_3} & 1 \end{bmatrix} \tag{8.28}$$

$$c = \cos(\phi), \tag{8.29}$$

$$s = \sin(\phi), \tag{8.30}$$

$$m = 1 - \cos(\phi), \tag{8.31}$$

$$R = \begin{bmatrix} n_1^2 m + c & n_1 n_2 m + n_3 s & n_1 n_3 m - n_2 s & 0 \\ n_1 n_2 m - n_3 s & n_2^2 m + c & n_2 n_3 m + n_1 s & 0 \\ n_1 n_3 m + n_2 s & n_2 n_3 m - n_1 s & n_3^2 m + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.32}$$

In our grasping application, we have to transform circles. Whereas in Geometric Algebra the transformation of complete circles is explicitly possible, in vector algebra we are able only to transform points.

Chapter 9
Efficient Computer Animation Application in CGA

For a long time, Geometric Algebra was known primarily as an elegant mathematical language. It could indeed be used in order to develop new algorithms, but to implement them efficiently, standard linear algebra was required. This was due to the low runtime performance of naively implemented Geometric Algebra algorithms. In 2006 [59], we presented for the first time an implementation of a computer animation algorithm that was faster than the standard implementation. This chapter presents this algorithm and some results of implementation based on Gaigen 2 (see Sect. 9.3.1) and on our Maple-based optimization approach (see Sect. 9.3.2). The remaining chapters of Part III present our Geometric Algebra Computing technology for the easy integration of Geometric Algebra into standard programming languages with the goal of achieving fast and robust implementations.

Here, we present the efficient implementation of the inverse kinematics of a human-arm-like model. In the animation of humanoid models, inverse kinematics solutions are important as a basic building block for path planning. The standard model for arms (and also legs) is a kinematic chain with seven degrees of freedom (DOF), with three DOF ($\theta_1, \theta_2, \theta_3$) at the shoulder, one DOF at the elbow (θ_4), and three DOF at the wrist ($\theta_5, \theta_6, \theta_7$). A standard tool for solving the inverse problem of mapping from a given end effector state to the configuration space $\{\theta_i\}$ was given by Tolani, Goswami, and Badler [109]. Our analytic solution to the inverse problem in CGA is an improvement of that in [56], and its derivation is considerably simpler than that in affine or projective geometry. Perhaps more importantly for the prospective user, our approach also turns out to be faster when implemented using Gaigen 2 or optimized based on our Maple-based approach.

9.1 Optimizations Based on Quaternions

Normally, the goal of an inverse kinematics algorithm is the computation of the joint angles (see Chap. 7). But in the application described here, the goal is to compute

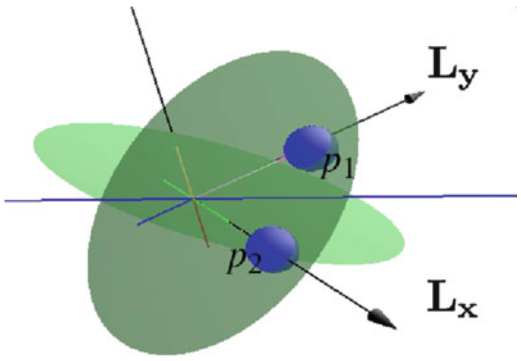


Fig. 9.1 Rotation based on the midline between two points through the origin

quaternions in order to perform SLERP operations related to the motion of the arm (see [111] for details of this motion interpolation procedure).

In Chap. 4, we found that quaternions can be handled directly in a CGA algorithm. For this reason, we improved our algorithm in order to compute the needed quaternions, directly and efficiently.

9.1.1 Direct Computation of Quaternions

For efficiency reasons, our inverse kinematics approach uses quaternions directly in the algorithm. In this way, we avoid the effort of translating between different mathematical systems such as transformation matrices and quaternions. In this section, we directly compute a quaternion that rotates an object from one point P_1 to another point P_2 as shown in Fig. 9.1. The two points are at the same distance from the origin and the rotation is about the blue midline L_m between the two points and passing through the origin. First, we calculate L_m. In CGA, a midplane of two points is described by their difference (see page 155 of [81]),

$$\pi_m = P_1 - P_2. \tag{9.1}$$

We calculate the midline with the help of the intersection of this plane and the plane through the origin and the points P_1 and P_2,

$$\pi_e^* = e_0 \wedge P_1 \wedge P_2 \wedge e_\infty, \tag{9.2}$$

and get

$$L_m = \pi_e \wedge \pi_m. \tag{9.3}$$

Second, in order to rotate from P_1 to P_2, we have to rotate around the midline with radius π. This results in a quaternion identical to the normalized line (see Sect. 4.3.3)

$$Q = \frac{L_m}{|L_m|}. \tag{9.4}$$

9.1.2 Efficient Computation of Quaternions

For efficiency reasons, we use an approach to calculating quaternions that does not use trigonometric functions. According to (4.13), a quaternion describing a rotation can be computed with the help of half of an angle and a normalized rotation axis. For example, if $L = i = e_3 \wedge e_2$, the resulting quaternion

$$\begin{aligned} Q &= \cos(\frac{\phi}{2}) + i\sin(\frac{\phi}{2}) \\ &= \cos(\frac{\phi}{2}) + (e_3 \wedge e_2)\sin(\frac{\phi}{2}) \end{aligned}$$

represents a rotation around the x-axis. The angle between two lines or two planes is defined according to Sect. 3.4.2 as follows:

$$\cos(\theta) = \frac{o_1^* \cdot o_2^*}{|o_1^*|\,|o_2^*|}. \tag{9.5}$$

We already know the cosine of the angle. For this reason, we are able to compute the quaternion in a more direct way using the following two properties of trigonometric functions:

$$\cos(\frac{\phi}{2}) = \pm\sqrt{\frac{1+\cos(\phi)}{2}} \tag{9.6}$$

and

$$\sin(\frac{\phi}{2}) = \pm\sqrt{\frac{1-\cos(\phi)}{2}}. \tag{9.7}$$

This leads to the formulas

$$\cos(\frac{\phi}{2}) = \pm\sqrt{\frac{1+\frac{o_1^*\cdot o_2^*}{|o_1^*||o_2^*|}}{2}} \tag{9.8}$$

and

$$\sin(\frac{\phi}{2}) = \pm\sqrt{\frac{1-\frac{o_1^*\cdot o_2^*}{|o_1^*||o_2^*|}}{2}}. \tag{9.9}$$

The signs of these formulas depend on the application.

9.2 The Inverse Kinematics Algorithm

Here, we describe the inverse kinematics of the human-arm-like model step by step. Our goal is for the wrist to reach the chosen point P_w. An arbitrary orientation of the

Table 9.1 Input/output parameters of the inverse kinematics algorithm

Parameter	Meaning
P_w	Target point of wrist
ϕ	Swivel angle
d_1, d_2	Lengths of the forearm and upper arm
Q_s	Shoulder quaternion
Q_e	Elbow quaternion

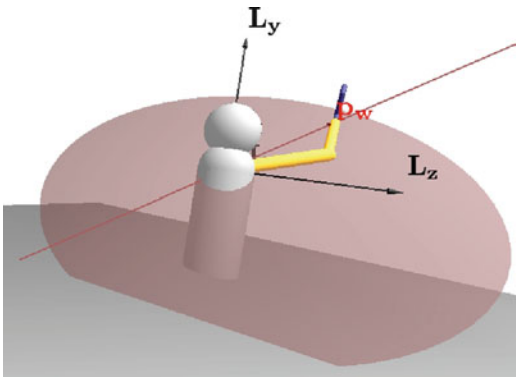


Fig. 9.2 Swivel plane

gripper is not investigated in this book. The length of the arm and the swivel angle are additional input parameters to the algorithm. The results of the algorithm are the quaternions for the shoulder and the elbow. A summary of the input and output parameters of the algorithm is given in Table 9.1.

9.2.1 *Computation of the Swivel Plane*

In accordance with [109], we use the swivel angle ϕ as one free degree of redundancy. The swivel plane is a plane rotated by ϕ around the line L_{sw} through the shoulder (at the origin) and the target point P_w of the wrist (see Fig. 9.2):

$$L_{sw} = (e_0 \wedge P_w \wedge e_\infty)^*. \tag{9.10}$$

Note that the OPNS representation of a line is defined with the help of two points and the point at infinity. The quaternion Q_{swivel} is defined according to (4.13):

$$Q_{swivel} = \cos(\frac{\phi}{2}) + \frac{L_{sw}}{|L_{sw}|}\sin(\frac{\phi}{2}). \tag{9.11}$$

Initially, the swivel plane is defined with the help of the origin, the point P_w, the point P_z ($\mathbf{x} = e_3$), given by

$$P_z = e_3 + \frac{1}{2}e_\infty + e_0, \tag{9.12}$$

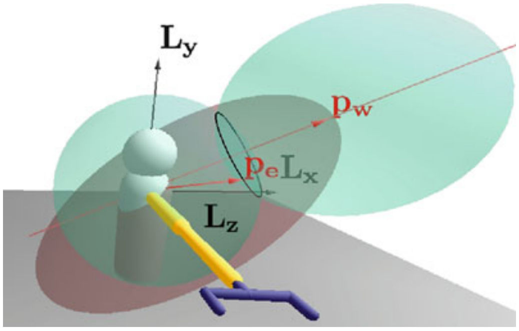


Fig. 9.3 Computation of the elbow point

and the point at infinity (see Table 3.2):

$$\pi_{swivel} = (e_0 \wedge P_z \wedge P_w \wedge e_\infty)^*. \tag{9.13}$$

Its final rotated location is

$$\pi_{swivel} = Q_{swivel}\ \pi_{swivel}\ \tilde{Q}_{swivel}. \tag{9.14}$$

For details of computing rotations in CGA, see Sect. 3.5.

9.2.2 The Elbow Point P_e

With the help of the two spheres

$$\begin{aligned} S_1 &= P_w - \frac{1}{2}d_2^2 e_\infty, \\ S_2 &= e_0 - \frac{1}{2}d_1^2 e_\infty, \end{aligned} \tag{9.15}$$

with center points P_w and e_0 and radii d_2 and d_1 we are able to compute the circle determining all possible locations of the elbow as the intersection of these spheres (see Table 3.2).

$$Z_e = S_1 \wedge S_2. \tag{9.16}$$

The intersection with the swivel plane delivers the point pair

$$Pp = Z_e \wedge \pi_{swivel}, \tag{9.17}$$

and we decide on one of the two possible elbow points and call it P_e. See 3.24 for extracting the points of a point pair. Figure 9.3 shows the elbow point P_e as the intersection of the swivel plane with the two spheres centered at the shoulder and at the target point P_w.

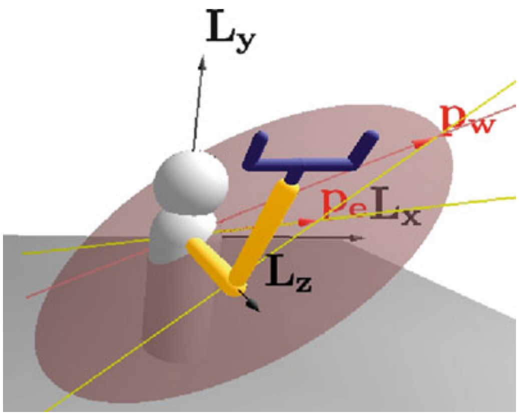


Fig. 9.4 Using the elbow quaternion

9.2.3 Calculation of the Elbow Quaternion Q_e

The elbow angle θ_4 is computed with the help of the line L_{se} through the shoulder and the elbow,

$$L_{se} = (e_0 \wedge P_e \wedge e_\infty)^*, \tag{9.18}$$

and the line L_{ew} through the shoulder and the wrist,

$$L_{ew} = (P_e \wedge P_w \wedge e_\infty)^*. \tag{9.19}$$

Based on these two lines, we are able to compute the angle between them (using the notation $c_i = \cos(\theta_i)$), from

$$c_4 = \cos(\theta_4) = \frac{L^*_{se} \cdot L^*_{ew}}{|L^*_{se}|\,|L^*_{ew}|}, \tag{9.20}$$

according to (9.5).

Now we are able to compute the quaternion Q_e according to (4.13):

$$Q_e = \cos(\theta_4/2) + \sin(\theta_4/2)i. \tag{9.21}$$

This represents a rotation around the local x-axis by an angle θ_4. The optimized version of this quaternion is

$$Q_e = \sqrt{\frac{1+c_4}{2}} - \sqrt{\frac{1-c_4}{2}}i, \tag{9.22}$$

according to (9.8) and (9.9). This quaternion rotates the upper arm by an angle corresponding to the angle between the two yellow lines in Fig. 9.4.

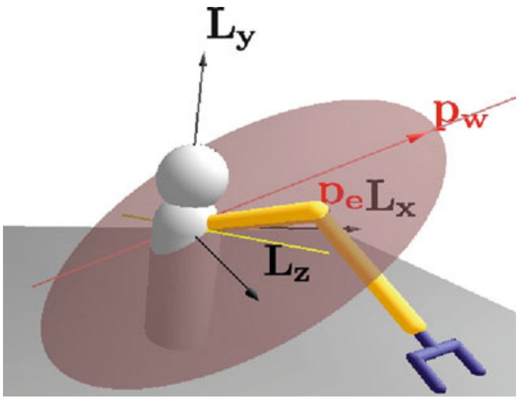


Fig. 9.5 Rotation to the elbow position

9.2.4 Rotation to the Elbow Position

First we calculate the midline L_m through the origin at the same distance from the points P_e and P_{ze} (see Sect. 9.1.1), where

$$P_{ze} = d_1 e_3 + \frac{1}{2} d_1^2 e_\infty + e_0. \tag{9.23}$$

We will need the line L_m in the next step in order to rotate around this line. To compute L_m, we use the midplane (given by the difference between the two points P_e and P_{ze})

$$\pi_m = P_{ze} - P_e \tag{9.24}$$

and the plane through the origin and the points P_e and P_{ze}

$$\pi_e^* = e_0 \wedge P_{ze} \wedge P_e \wedge e_\infty, \tag{9.25}$$

and intersect them:

$$L_m = \pi_e \wedge \pi_m. \tag{9.26}$$

In order to rotate the elbow towards our already computed point P_e we have to rotate around the midline determined in the previous step by an angle π. This results in a quaternion identical to the normalized midline (see Sect. 9.1.1),

$$Q_{12} = \frac{L_m}{|L_m|}. \tag{9.27}$$

Figure 9.5 shows this rotation from the z-axis L_z to the elbow point with the help of the yellow midline.

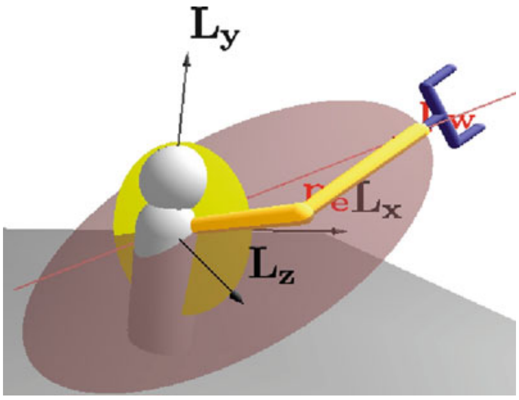


Fig. 9.6 Rotation to the wrist location

9.2.5 Rotation to the Wrist Location

The angle θ_3 (and the resulting quaternion Q_3) is computed with the help of the y–z plane rotated by the quaternion Q_{12}, and the swivel plane. The y–z plane (with normal vector e_1 and zero distance from the origin) is computed using

$$\pi_{yz} = e_1. \tag{9.28}$$

The rotated plane π_{yz2} is obtained from

$$\pi_{yz2} = Q_{12}\,\pi_{yz}\,\tilde{Q}_{12}. \tag{9.29}$$

Based on these two planes, we are able to compute the angle between them, from

$$c_3 = \cos(\theta_3) = \frac{\pi^*_{yz2} \cdot \pi^*_{swivel}}{\left|\pi^*_{yz2}\right| \left|\pi^*_{swivel}\right|} \tag{9.30}$$

according to (9.5), and we get the quaternion

$$Q_3 = \cos(\theta_3/2) + \sin(\theta_3/2)k. \tag{9.31}$$

This represents a rotation around the local z-axis by an angle θ_3. The optimized version of this quaternion is

$$Q_3 = \pm\sqrt{\frac{1+c_3}{2}} + \sqrt{\frac{1-c_3}{2}}k, \tag{9.32}$$

according to (9.8) and (9.9). The sign of this quaternion depends on which side of the plane π_{yz2} the point P_w lies on. This can be computed easily with the help of the inner product

$$\pi_{yz2} \cdot P_w.$$

This quaternion rotates the arm to the wrist location as shown in Fig. 9.6.

The resulting quaternion for the shoulder rotation can now be computed as the product of Q_{12} and Q_3,

$$Q_s = Q_{12}Q_3. \tag{9.33}$$

Taking this together with the elbow quaternion Q_e, we have all the information needed for an interpolation based on SLERP in order to reach the target.

9.3 Approaches to Runtime Optimization

We developed and simulated our algorithm visually, based on CLUCalc, as also used in our tutorial in Chap. 6. Then we had to implement it on the target platform, a virtual reality system written in C++. Originally, the inverse kinematics was implemented using IKAN [109], a widely used C++ library.

When we first implemented our algorithm with Gaigen, the runtime benchmarks were worse than for the IKAN implementation. But, when we used our optimization techniques, our approaches clearly outperformed the IKAN implementation. In this section, we present two different optimization approaches, one is based on Maple, and the other on the code generator Gaigen 2.

9.3.1 Optimization with Gaigen 2

With the help of the Gaigen 2 toolkit from the University of Amsterdam [29, 35] we were able to implement algorithms and integrate them into our target platform. Gaigen 2 is a C++ code generator for geometric algebras; see [36] for information and to download the package.

If runtime efficiency is a major issue, one can use some additional techniques from the Gaigen 2 toolkit to optimize the resulting C++ code. The philosophy behind Gaigen 2 is based on two ideas: generative programming and specializing to the structure of Geometric Algebra. Gaigen 2 takes a succinct specification of a geometric algebra and transforms it into an implementation. The resulting implementation is very similar to what someone would program by hand and can be linked directly to an application.

In many types of programs, not every variable needs a linear combination of all of the 32 blades of CGA (see Table 3.1); instead, it has a fixed "specialized" multivector type. The inverse kinematics algorithms of this chapter, for instance, use variables with multivector types such as *line* and *sphere*. If the Geometric Algebra implementation can work directly with these leaner specialized multivector types, performance will be greatly increased. As an implementation of this insight, Gaigen 2 allows the user to define specialized types along with the algebra specification, and generates classes for each of them. These specialized multivector classes require much less storage than the generic multivector but, as a result, they are of course

Table 9.2 Input/output parameters of the inverse kinematics algorithm using Gaigen 2

Parameter	Gaigen	Meaning
P_w	`pw`	Target point of wrist
ϕ	`Sangle`	Swivel angle
d_1, d_2	`d1, d2`	Lengths of the forearm and upper arm
Q_s	q_s	Shoulder quaternion
Q_e	q_e	Elbow quaternion

unable to store an arbitrary multivector type. For example, a *line* variable cannot be stored in a *sphere* variable.

The big advantage of solutions developed with Gaigen 2 or with Gaalop GPC (see Chap. 11) is that one can think in terms of geometry, and program directly in geometric elements.

The following detailed description of the inverse kinematics algorithm is offered as an explicit example of how one can think in terms of geometry, and program directly in geometric elements. We present the Gaigen 2 code for the above inverse kinematics algorithm. The goal of our inverse kinematics algorithm is to compute the output parameters Q_s and Q_e based on the input parameters (see Table 9.2).

9.3.1.1 Computation of the Elbow Point P_e

The Gaigen 2 implementation of (9.15)–(9.17) is as follows:

```
Sphere s1 = _Sphere(pw - 0.5f*d2*d2*einf); // einf means e∞

originSphere s2 = _originSphere(e0 - 0.5f*d1*d1*einf);

Circle Z_e = _Circle(s1^s2);
```

With the help of the two spheres s1 and s2 with center points P_w (the target point of the wrist) and e_0 (the shoulder located at the origin) and radii d_2 and d_1, we are able to compute the circle determining all possible locations of the elbow as the intersection of these spheres.

We use the following specializations for the geometric objects that we need. Both the target point P_w and the sphere S_1 are assigned to a multivector type called `Sphere` (remember that a point in CGA is simply a sphere of zero radius). The type `Sphere` is defined as follows:

```
specialization: blade Sphere(e0=1, e1, e2, e3, einf);
```

The type `Sphere` means a linear combination of basis blades where the coefficient of e_0 is 1. For the second sphere S_2, with its center at the origin e_0, we use the type `originSphere`:

```
specialization: blade originSphere(e0, einf);
```

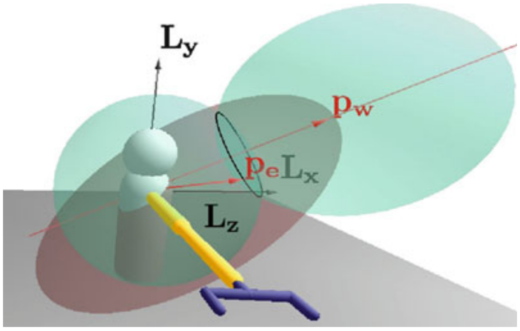


Fig. 9.7 Computation of the elbow point

since `e0-0.5f* d1* d1*einf` needs only the blades `e0` and `einf`. The result of the intersection of the spheres $Z_e = S_1 \wedge S_2$ is of type `Circle`, defined as follows:

```
specialization: blade Circle(e0^e1, e0^e2, e0^e3, e1^einf,
e2^einf, e3^einf, e0^einf);
```

The computation is illustrated in Fig. 9.7.

9.3.1.2 Computation of the Swivel Plane

The corresponding Gaigen 2 implementation for the computation of the swivel plane is as follows:

```
originLine l_sw = _originLine(dual(e0^pw^einf));

originPlane SwivelPlane = _originPlane(dual(pz^pw^e0^einf));
```

Note that `pz` is defined as a constant equal to e_3.

```
quaternion SwivelRot = _quaternion(cos( SAngle/2)
+sin( SAngle/2) * (l_sw*(1.0f/_float(GAnorm(l_sw)))));

SwivelPlane = _originPlane(SwivelRot*SwivelPlane*reverse
(SwivelRot));

pointPair pp2 = _pointPair(dual(Z_e^SwivelPlane));

Sphere p_e = _Sphere(
-(-sqrt(_float(GAnorm(lcont(pp2,pp2))))+pp2)*
(inverse(lcont(einf,pp2))));
```

The line L_{sw} passes through the origin; its algebraic representation is a Euclidean bivector. We define `originLine` as follows:

```
specialization: blade originLine(e1^e2, e2^e3, e3^e1);
```

The plane π_{swivel} is a linear combination of the blades defined by the type `originPlane`:

```
specialization: blade originPlane(e1, e2, e3);
```

We do not define it as the type `Plane`, because the distance between π_{swivel} and origin is 0, which means that the e_∞ part can be omitted. We can contrast this with the definition of an arbitrary `Plane`:

```
specialization: blade Plane(e1, e2, e3, einf);
```

The quaternion Q_{swivel} matches the definition of `quaternion`:

```
specialization: versor quaternion(1.0, e1^e2, e2^e3, e3^e1);
```

The point pair pp_2 matches the definition of `pointPair`, and the point P_e is assigned naturally to `Sphere`.

9.3.1.3 Computation of the Elbow Quaternion Q_e

This step is based on Sect. 9.2.3. The Gaigen 2 implementation is as follows:

```
originLine l_se = _originLine(dual(e0^pe^einf));
Line l_ew = _Line(dual((p_e^pw^einf)));
```

The line L_{se} passes through the origin, so it is of type `originLine`:

```
specialization: blade originLine(e1^e2, e2^e3, e3^e1);
```

The line L_{ew} does not go through the origin. We define the type `Line` as follows:

```
specialization: blade Line(e1^e2, e1^e3, e2^e3,
e1^einf,e2^einf,e3^einf);
```

```
mv::Float cosi = (-_Float(scp(l_se,l_ew)))/( d1* d2);
mv::Float cos_2 = sqrt((1+cosi)/2.0);
mv::Float sin_2 = sqrt((1-cosi)/2.0);
quaternion_i q_e = _quaternion_i(cos_2+sin_2*quati);
```

Note the difference between L_{se} and L_{ew}. `quati` is defined as a constant for the quaternion i. The quaternion Q_e describes a rotation around the x-axis only using $e_3 \wedge e_2$ as a blade. This matches the definition of `quaternion_i`:

```
specialization: versor quaternion_i(1.0, e3^e2);
```

Table 9.3 Computation of the shoulder quaternion

$$\pi_{yz2} = Q_{12} * e_1 * \tilde{Q}_{12}$$
$$c_3 = \frac{\pi_{yz2} \cdot \pi_{swivel}}{|\pi_{yz2}||\pi_{swivel}|}$$
$$sign = \pi_{yz2} \cdot P_w$$
$$Q_3 = \sqrt{\frac{1+c_3}{2}} + \sqrt{\frac{1-c_3}{2}}k$$
$$Q_s = Q_{12} * Q_3$$

9.3.1.4 Rotation to the Elbow Position

The Gaigen 2 implementation of the computation of Q_{12} is as follows:

```
Sphere p_ze = _Sphere( d1*e3);
originPlane pi_m = _originPlane(p_ze-p_e);
originPlane pi_e = _originPlane(dual(p_ze^p_e^e0^einf));
originLine l_m = _originLine(pi_e^pi_m);
pureQuaternion q_12 =_pureQuaternion
    (l_m*(1.0f/_float(GAnorm(l_m))));
```

The point P_{ze} is assigned to the type `Sphere`. The planes π_m and π_e are not assigned to the type `Plane`, because they pass through the origin (the coefficient of e_∞ is 0). This matches the definition of the multivector type `originPlane`. The intersection line L_m passes through the origin, so it is assigned to the type `originLine`. The quaternion Q_{12} has no scalar part, so it is assigned to the type `pureQuaternion`:

```
specialization: blade pureQuaternion (e3^e2, e1^e3, e2^e1);
```

9.3.1.5 Rotation to the Wrist Location

The Gaigen 2 implementation is as follows, in accordance with Table 9.3 (in the first formula, we use e_1 directly instead of π_{yz}):

```
originPlane plane_yz2 = _originPlane(q_12*e1*reverse(q_12));
cosi=computeCos(plane_yz2,SwivelPlane);
mv:: Float sign= _float(scp(plane_yz2,pw));
sign = sign/abs(sign);
cos_2 = sqrt((1+cosi)/2.0)*sign;
sin_2 = sqrt((1-cosi)/2.0);
quaternion_k q_3 = _quaternion_k(cos_2+sin_2*quatk);
q_s = q_12 * q_3;
```

The plane π_{yz2} is not assigned to the type `Plane`, because it has distance of 0 from the origin. So, the coefficient of e_∞ is 0. This matches the definition of the

Table 9.4 Input/output parameters of the inverse kinematics algorithm using Maple

Parameter	Maple	Meaning
P_w	pw(pwx, pwy, pwz)	Target point of wrist
ϕ	sangle	Swivel angle
d_1, d_2	d1, d2	Lengths of the forearm and upper arm
Q_s	qs	Shoulder quaternion
Q_e	qe	Elbow quaternion

multivector type `originPlane`. The quaternion Q_3 describes a rotation around the z-axis, so it is assigned to the type `quaternion_k`:

```
    specialization: versor quaternion_k(1.0, e2^e1);
```

The quaternion Q_s is assigned to the quaternion type.

The above-mentioned versor multiplication that rotates the plane π_{yz} with the help of a quaternion Q_{12} can be optimized as follows. The C++ code for this equation reads

```
Plane plane_yz2 = applyVersor(q_12, e1);
```

where **q_12** is a quaternion and **e1** is a constant. The final result is output as the following optimized C++ function:

```
inline Plane applyVersor(
    const quaternion& V, const __e1_ct__& X)
{
  return Plane(
    V.c[0]*V.c[0] - V.c[1]*V.c[1] +
    V.c[2]*V.c[2] - V.c[3]*V.c[3] ,
    -2*V.c[0]*V.c[1] + 2*V.c[2]*V.c[3],
     2*V.c[2]*V.c[1] + 2*V.c[0]*V.c[3]);
}
```

9.3.2 *Optimization with Maple*

We used Maple in order to obtain the most elementary relationship between the input and output parameters of our inverse kinematics algorithm (see Table 9.4). The specific technique is described in Sect. 4.1.

9.3.2.1 Inverse Kinematics Algorithm in Maple

The goal of our inverse kinematics algorithm is to compute the output parameters Q_s and Q_e based on the input parameters (see Table 9.4). In this section, we present the Maple code of the inverse kinematics algorithm:

- **Computation of the elbow point** P_e:

```
> pw:=pwx*e1+pwy*e2+pwz*e3+0.5*
        (pwx^2+pwy^2+pwz^2)*einf+e0;
> S1:=pw-0.5*d2*d2*einf;
> S2:=e0-0.5*d1*d1*einf;
> Z_e:=S1 &w S2;
> // now compute the swivel plane ...
> l_sw:=-(e0 &w pw &w einf)&c e12345; // dualization operation
> pi_swivel:=-(pz &w pw &w e0 &w einf)
              &c e12345;
> norm_l_sw:=sqrt(l_sw &c reversion(l_sw));
> q_swivel:=cos(sangle/2)+sin(sangle/2)
             *(l_sw / norm_l_sw);
> pi_swivel:=q_swivel &c pi_swivel
              &c reversion(q_swivel);
> PP:=-(Z_e &w pi_swivel) &c e12345;
> PP:=vectorpart(PP,2);
> einf_PP:=LC(einf,PP);
> norm_einf_PP:=einf_PP &c
                  reversion(einf_PP);
> inv_einf_PP:=einf_PP/norm_einf_PP;
> p_e:=-(-sqrt(scalarpart(LC(PP,PP)))
       +PP) &c inv_einf_PP;
> p_e:=vectorpart(p_e,1);
```

- **Computation of the quaternion** Q_e **for the elbow joint:**

```
> l_se:=-(e0 &w p_e &w einf)&c e12345;
> l_ew:=-(p_e &w pw &w einf)&c e12345;
> c4:=-LC(l_se,l_ew)/(d1*d2)/Id;
> qe:=sqrt((1+c4)/2)+sqrt((1-c4)/2)*(-qi);
```

- **Rotation to the elbow position:**

```
> p_ze:=d1*e3+0.5*d1^2*einf+e0;
> pi_m:= p_ze-p_e;
> pi_e:=-(p_ze &w p_e &w e0 &w einf)&c e12345;
> l_m:=pi_e &w pi_m;
> q12:=l_m/(sqrt(l_m &c reversion(l_m)));
```

- **Rotation to the wrist location.** We compute the quaternion Q_s for the shoulder joint. This will let the robot wrist reach the given target P_w:

```
> pi_yz2:=q_12 &c e1 &c reversion(q_12);
> _sign:=scalarpart(LC(pw,pi_yz2));
> _sign:=_sign/abs(_sign);
> norm_pi_swivel:=sqrt(pi_swivel &c
                   reversion(pi_swivel));
> c3:=scalarpart(-LC(pi_yz2,pi_swivel))
      /(norm_pi_swivel);
> q3:=sqrt((1+c3)/2)+sqrt((1-c3)/2)
      *_sign*qk;
> qs:=q12 &c q3;
```

The quaternions Q_s and Q_e are the required results of our algorithm.

9.3.2.2 Optimized Inverse Kinematics Algorithm

With the help of Maple our Geometric Algebra formulas could be simplified and combined into very efficient expressions because of the symbolic-computation features of Maple. For instance, we obtained the following result for the first few lines of the algorithm in Sect. 9.3.2.1:

```
Z_e = 0.5*(1-d1^2)*(pwx*e15+pwy*e25+pwz*e35)-
      0.5*(1+d1^2)*(pwx*e14+pwy*e24+pwz*e34)+
      0.5*e45*(pwx^2+pwy^2+pwz^2+d1^2-d2^2)
```

which uses only some simple multiplications and additions. We then obtain

$$|L_{sw}| = \sqrt{p_{w_x}^2 + p_{w_y}^2 + p_{w_z}^2}. \tag{9.34}$$

The coefficients of the swivel plane are

$$\pi_{swivel\,x} = \frac{(2\cos\frac{\phi}{2}\sin\frac{\phi}{2}p_{w_z}p_{w_x} - p_{w_y}|L_{sw}| + 2p_{w_y}|L_{sw}|\cos\frac{\phi}{2}^2)}{|L_{sw}|} \tag{9.35}$$

$$\pi_{swivel\,y} = \frac{(2\cos\frac{\phi}{2}\sin\frac{\phi}{2}p_{w_z}p_{w_y} + p_{w_x}|L_{sw}| - 2p_{w_x}|L_{sw}|\cos\frac{\phi}{2}^2)}{|L_{sw}|} \tag{9.36}$$

$$\pi_{swivel\,z} = \frac{-2\sin\frac{\phi}{2}\cos\frac{\phi}{2}(p_{w_x}^2 + p_{w_y}^2)}{|L_{sw}|} \tag{9.37}$$

The coefficients of the point pair PP are

$$PP_i = \frac{1}{2}\pi_{swivel\,x}(p_{w_x}^2 + p_{w_z}^2 + p_{w_y}^2 + d_1^2 - d_2^2), \tag{9.38}$$

$$PP_j = \frac{1}{2}\pi_{swivel\,y}(p_{w_x}^2 + p_{w_z}^2 + p_{w_y}^2 + d_1^2 - d_2^2), \tag{9.39}$$

$$PP_k = \frac{1}{2}\pi_{swivel\,z}(p_{w_x}^2 + p_{w_z}^2 + p_{w_y}^2 + d_1^2 - d_2^2), \tag{9.40}$$

$$PP_{14} = \frac{1}{2}(1 - d_1^2)(p_{w_y}\pi_{swivel\,z} - p_{w_z}\pi_{swivel\,y}), \tag{9.41}$$

$$PP_{15} = \frac{1}{2}(1 + d_1^2)(p_{w_z}\pi_{swivel\,y} - p_{w_y}\pi_{swivel\,z}), \tag{9.42}$$

$$PP_{24} = \frac{1}{2}(1 - d_1^2)(p_{w_z}\pi_{swivel\,x} - p_{w_x}\pi_{swivel\,z}), \tag{9.43}$$

$$PP_{25} = \frac{1}{2}(1 + d_1^2)(p_{w_x}\pi_{swivel\,z} - p_{w_z}\pi_{swivel\,x}), \tag{9.44}$$

$$PP_{34} = \frac{1}{2}(d_1^2 - 1)(p_{w_y}\pi_{swivel_x} - p_{w_x}\pi_{swivel_y}), \tag{9.45}$$

$$PP_{35} = \frac{1}{2}(1 + d_1^2)(p_{w_y}\pi_{swivel_x} - p_{w_x}\pi_{swivel_y}). \tag{9.46}$$

The elbow point P_e is extracted from the point pair PP as follows:

$$einf\_PP = (PP_{35} - PP_{34})^2 + (PP_{14} - PP_{15})^2 + (PP_{25} - PP_{24})^2 \tag{9.47}$$

$$\begin{aligned} tmp_1 =& - PP_i^2 - PP_j^2 - PP_k^2 - PP_{14}^2 + PP_{15}^2 - PP_{24}^2 \\ &+ PP_{25}^2 - PP_{34}^2 + PP_{35}^2 \end{aligned} \tag{9.48}$$

$$tmp_{sqrt} = \sqrt{tmp_1} \tag{9.49}$$

$$\begin{aligned} p_{e_x} =& (PP_j(PP_{34} - PP_{35}) + PP_k(PP_{25} - PP_{24}) \\ &+ tmp_{sqrt}(PP_{15} - PP_{14}))/einf\_PP \end{aligned} \tag{9.50}$$

$$\begin{aligned} p_{e_y} =& (PP_i(PP_{35} - PP_{34}) + PP_k(PP_{14} - PP_{15}) \\ &+ tmp_{sqrt}(PP_{25} - PP_{24}))/einf\_PP \end{aligned} \tag{9.51}$$

$$\begin{aligned} p_{e_z} =& (PP_j(PP_{15} - PP_{14}) + PP_i(PP_{24} - PP_{25}) \\ &+ tmp_{sqrt}(PP_{35} - PP_{34}))/einf\_PP \end{aligned} \tag{9.52}$$

The quaternion Q_e for the rotation at the elbow joint is given by

$$Q_e = \sqrt{\frac{1 + \frac{p_{e_x}^2 - p_{e_x}p_{w_x} + p_{e_y}^2 - p_{e_y}p_{w_y} - p_{e_z}p_{w_z} + p_{e_z}^2}{d_1 d_2}}{2}} + \tag{9.53}$$

$$\sqrt{\frac{1 - \frac{p_{e_x}^2 - p_{e_x}p_{w_x} + p_{e_y}^2 - p_{e_y}p_{w_y} - p_{e_z}p_{w_z} + p_{e_z}^2}{d_1 d_2}}{2}}\, e_{23} \tag{9.54}$$

The result for the quaternion Q_{12} is

$$\begin{aligned} tmp_2 =& d_1^4 p_{e_y}^2 - 2d_1^3 p_{e_y}^2 p_{e_z} + d_1^2 p_{e_y}^2 p_{e_z}^2 + \\ & d_1^4 p_{e_x}^2 - 2d_1^3 p_{e_x}^2 p_{e_z} + d_1^2 p_{e_x}^2 p_{e_z}^2 + \\ & d_1^2 p_{e_x}^4 + 2d_1^2 p_{e_x}^2 p_{e_y}^2 + d_1^2 p_{e_y}^4 \end{aligned} \tag{9.55}$$

$$|L_m| = \sqrt{tmp_2} \tag{9.56}$$

$$q_{12_i} = \frac{d_1 p_{e_x}(d_1 - p_{e_z})}{|L_m|} \tag{9.57}$$

$$q_{12_j} = \frac{d_1 p_{e_y}(d_1 - p_{e_z})}{|L_m|} \tag{9.58}$$

$$q_{12_k} = \frac{d_1(p_{e_x}^2 + p_{e_y}^2)}{|L_m|} \tag{9.59}$$

The last rotation at the shoulder joint is given by c_3:

$$c_3 = (\pi_{swivel_x}(q_{12_k}^2 + q_{12_j}^2 - q_{12_i}^2) - \tag{9.60}$$

$$2q_{12_i}(q_{12_j}\pi_{swivel_y} + q_{12_k}\pi_{swivel_z}))/ \tag{9.61}$$

$$\sqrt{\pi_{swivel_x}^2 + \pi_{swivel_y}^2 + \pi_{swivel_z}^2} \tag{9.62}$$

$$sign = p_{w_x}(q_{12_i}^2 - q_{12_k}^2 - q_{12_j}^2) + \tag{9.63}$$

$$2q_{12_i}(q_{12_k} p_{w_z} + q_{12_j} p_{w_y}) \tag{9.64}$$

$$sign = \frac{sign}{|sign|} \tag{9.65}$$

$$q_{3_{scalar}} = \sqrt{\frac{1 + c_3}{2}} \tag{9.66}$$

$$q_{3_k} = \sqrt{\frac{1 - c_3}{2}}\, sign \tag{9.67}$$

The final result for Q_s is

$$\begin{aligned} Q_s = &- q_{12_k} \cdot q_{3_k} - (q_{12_i} \cdot q_{3_{scalar}} + q_{12_j} \cdot q_{3_k})e_{23} \\ &- (q_{12_i} \cdot q_{3_k} - q_{12_j} \cdot q_{3_{scalar}})e_{13} - (q_{12_k} \cdot q_{3_{scalar}})e_{12} \end{aligned} \tag{9.68}$$

We can code these results of the Maple optimization process into C/C++ easily. For better computation efficiency, some frequently used given variables could be defined as constant and some repeatedly computed expressions could be assigned to auxiliary variables.

9.4 Results

First we developed and simulated our algorithm at a high level based on CLUCalc. Then we implemented it on the target platform Avalon, a virtual reality system written in C++ using Visual Studio.NET 2003. The inverse kinematics was implemented first using IKAN [109] in the conventional way.

Our Maple-based approach clearly outperformed the IKAN implementation. The Gaigen 2 approach also outperformed IKAN in a similar way. In a nutshell, the CGA-based algorithm was 43 % faster than IKAN, and 240 % faster when the conversion from matrices to quaternions was taken into account.

See Sect. 14.1 for a description of a hardware implementation of our algorithm leading to an additional speedup by a factor of about 100.

Chapter 10
Using Gaalop for High-Performance Geometric Algebra Computing

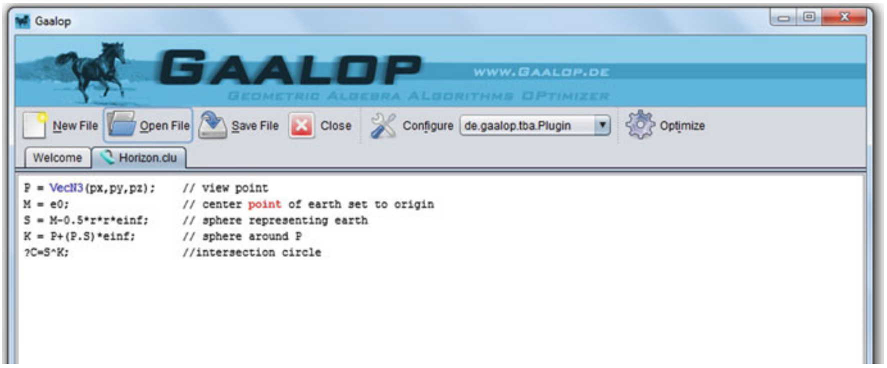
The Maple-based approach of Chap. 9 was the basis for the development of our Gaalop (Geometric algebra algorithms optimizer) compiler, using the CLUCalc language of Part II as the input language (see [61]). In this chapter, we introduce Gaalop based on the horizon example and present our new compilation approaches to going from Geometric Algebra algorithms to optimized code. Gaalop is our main tool for the efficient implementation of Geometric Algebra algorithms; most of Part III is dedicated to it. Gaalop is also the base for the integration of Geometric Algebra algorithms into programming languages, as introduced in Chap. 11. You are able to download Gaalop from www.gaalop.de.

10.1 The Horizon Example with Gaalop

```
P = VecN3(px,py,pz);        // view point
M = e0;                     // center point of earth set to origin
S = M-0.5*r*r*einf;         // sphere representing earth
K = P+(P.S)*einf;           // sphere around P
?C=S^K;                     // intersection circle
```

Listing 10.1 Gaalop input for the horizon example

The CLUCalc code for the computation of the horizon example of Sect. 3.7 is shown in Listing 10.1. This computes the horizon circle C seen from the viewpoint P on the Earth S. The variables **px**, **py**, **pz** and the radius **r** are free variables. This script can be used as an input to Gaalop. The free variables are handled in Gaalop as symbolic variables. Figure 10.1 shows a screenshot of the Gaalop input for the horizon example. First of all, we can see that the CLUCalc code in Listing 10.1 is exactly what Gaalop expects as input. A question mark at the beginning of a line indicates a multivector variable that has to be explicitly computed by Gaalop. This means that Gaalop is able to optimize not only single statements, but a



```
P = VecN3(px,py,pz);     // view point
M = e0;                  // center point of earth set to origin
S = M-0.5*r*r*einf;      // sphere representing earth
K = P+(P.S)*einf;        // sphere around P
?C=S^K;                  //intersection circle
```

Fig. 10.1 The CLUCalc input code of the *horizon* example as requested by *Gaalop*

number of Geometric Algebra statements written in CLUScript. In Listing 10.1, the expressions for P, M, S, and K are used only by Gaalop, in order to compute an optimized result for the horizon circle C (see the question mark in the last line of the listing).

Gaalop writes the content of this multivector in C syntax in an output file. More exactly, it computes the coefficients of the multivector and assigns them to an array representing this multivector. Listing 10.2 shows the result for the optimized circle C as an array with entries 8, 9, 11, 12, 13, 14, and 15 (all the other entries are zero) according to Table 10.1.

```
C[8] = 0.5f * px * r * r;   // e1^einf
C[9] = - px;                // e1^e0
C[11] = 0.5f * py * r * r;  // e2^einf
C[12] = - py;               // e2^e0
C[13] = 0.5f * pz * r * r;  // e3^einf
C[14] = - pz;               // e3^e0
C[15] = - r * r;            // einf^e0
```

Listing 10.2 Gaalop output for the horizon example

Figure 10.2 shows two alternative ways to perform the calculation of the horizon, namely two alternative calculations of the sphere K. Although one might think that one or other of the two alternatives would result in better optimized code, the interesting thing is that the two calculations lead to exactly the same result. This shows that what Gaalop is doing is optimal in some sense.

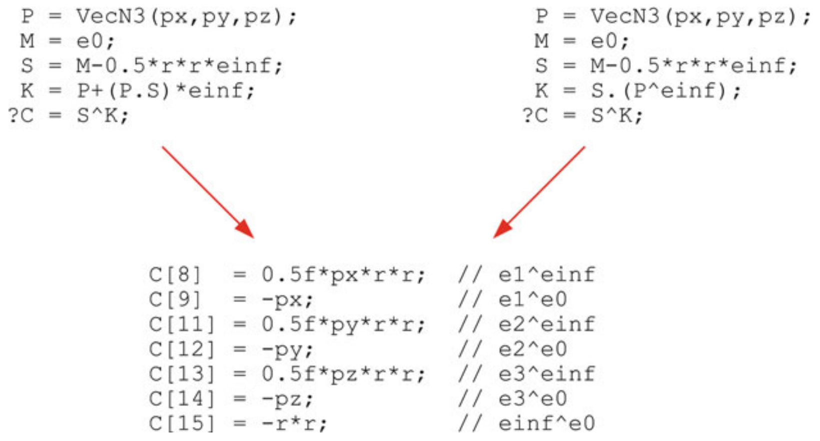
10.2 The Geometric Algebra Computing Approach

How can we combine the properties of Geometric Algebra (see Sect. 1.1) with high-performance implementations? Multivectors of n-dimensional Geometric Algebra are 2^n-dimensional. At first glance, this would seem to be computationally very

Table 10.1 The 32 blades of 5D CGA that compose a multivector. The entry in the first column is the index of the corresponding blade. The negated entries are needed for the selectors of the Geometric Algebra Parallelism Programs (GAPP) described in Chap. 14

Index	Negative index	Blade	Grade
0	-0	1	0
1	-1	e_1	1
2	-2	e_2	1
3	-3	e_3	1
4	-4	e_∞	1
5	-5	e_0	1
6	-6	$e_1 \wedge e_2$	2
7	-7	$e_1 \wedge e_3$	2
8	-8	$e_1 \wedge e_\infty$	2
9	-9	$e_1 \wedge e_0$	2
10	-10	$e_2 \wedge e_3$	2
11	-11	$e_2 \wedge e_\infty$	2
12	-12	$e_2 \wedge e_0$	2
13	-13	$e_3 \wedge e_\infty$	2
14	-14	$e_3 \wedge e_0$	2
15	-15	$e_\infty \wedge e_0$	2
16	-16	$e_1 \wedge e_2 \wedge e_3$	3
17	-17	$e_1 \wedge e_2 \wedge e_\infty$	3
18	-18	$e_1 \wedge e_2 \wedge e_0$	3
19	-19	$e_1 \wedge e_3 \wedge e_\infty$	3
20	-20	$e_1 \wedge e_3 \wedge e_0$	3
21	-21	$e_1 \wedge e_\infty \wedge e_0$	3
22	-22	$e_2 \wedge e_3 \wedge e_\infty$	3
23	-23	$e_2 \wedge e_3 \wedge e_0$	3
24	-24	$e_2 \wedge e_\infty \wedge e_0$	3
25	-25	$e_3 \wedge e_\infty \wedge e_0$	3
26	-26	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$	4
27	-27	$e_1 \wedge e_2 \wedge e_3 \wedge e_0$	4
28	-28	$e_1 \wedge e_2 \wedge e_\infty \wedge e_0$	4
29	-29	$e_1 \wedge e_3 \wedge e_\infty \wedge e_0$	4
30	-30	$e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	4
31	-31	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5

expensive. For example, in CGA their size is limited to 32 blades (see Table 10.1). The multivector storage for CGA therefore has to save a maximum of 32 blade coefficients. A naive approach might therefore be to simply save the maximum number of coefficients in an array. The problem with this approach is that the number of blades grows exponentially with the dimensionality. The 9D algebra of Julio Zamora [115], for example, has 512 blades and 512 blade coefficients, which are too many to save efficiently. Since we want to support even higher numbers of dimensions, this is not an option. Fortunately, the simple observation that the majority of the blade coefficients of a multivector are equal to zero helps us to overcome this problem. The obvious solution is to save only nonzero blade coefficients.



```
 P = VecN3(px,py,pz);
 M = e0;
 S = M-0.5*r*r*einf;
 K = P+(P.S)*einf;
?C = S^K;
```

```
 P = VecN3(px,py,pz);
 M = e0;
 S = M-0.5*r*r*einf;
 K = S.(P^einf);
?C = S^K;
```

```
C[8]  = 0.5f*px*r*r;  // e1^einf
C[9]  = -px;          // e1^e0
C[11] = 0.5f*py*r*r;  // e2^einf
C[12] = -py;          // e2^e0
C[13] = 0.5f*pz*r*r;  // e3^einf
C[14] = -pz;          // e3^e0
C[15] = -r*r;         // einf^e0
```

Fig. 10.2 Two alternative calculations of the horizon application according to Sect. 3.7, with the same optimized result

In a nutshell, there is a lot of potential for optimization and parallelization of the handling of multivectors:

- The possibility of precomputing Geometric Algebra expressions:
  - Determination of which of the coefficients are needed for the resulting multivector;
  - Symbolic simplification of the computations of the nonzero coefficients;
- The possibility of parallelization of the remaining coefficient computations:
  - All coefficients of one multivector can be computed in parallel;
  - Each coefficient computation can benefit from instruction-level parallelism (see Chap. 14).

Our Geometric Algebra Computing approach uses two specific layers:

- A Geometric Algebra compilation layer;
- A platform layer.

In the compilation layer, Geometric Algebra operations on multivectors such as calculating the geometric product, outer product, inner product, dual, and reverse are handled. This is compiled in a second step to the platform layer. In this layer, only basic arithmetic operations on multivectors with a high potential for efficient computation on sequential and parallel platforms are available.

Our Geometric Algebra architecture is presented in Fig. 10.3. Algorithms (described by the Geometric Algebra programming language CLUCalc [82]) are compiled to an intermediate representation using either a Maple-based or a table-based approach (see Sect. 10.3). Based on this representation, implementations

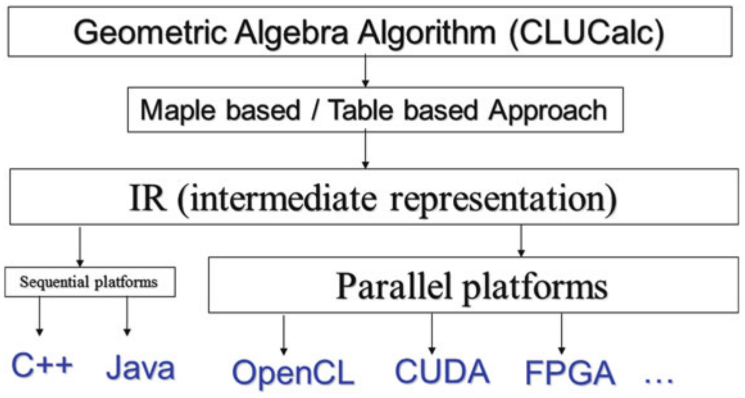

Geometric Algebra Algorithm (CLUCalc)
Maple based / Table based Approach
IR (intermediate representation)
Sequential platforms
Parallel platforms
C++
Java
OpenCL
CUDA
FPGA
...

Fig. 10.3 The Geometric Algebra Computing architecture. Algorithms are compiled to an *intermediate representation* for compilation to different computing platforms

for different sequential and parallel platforms can be derived. Some examples of Geometric Algebra Computing platforms are presented in this book:

- Based on C++, in Chap. 11;
- Based on OpenCL/CUDA, in Chap. 13;
- Based on an FPGA platform, in Chap. 14.

In order to achieve highly efficient implementations, Geometric Algebra algorithms have first to be optimized. We use two different compilation approaches, a table-based approach, described in Sect. 10.3, and a Maple-based approach. The Maple-based compilation needs the commercial Maple package and is restricted to Geometric Algebras with dimension less than or equal to nine. The Maple-based compilation uses the powerful symbolic-computation feature of Maple (see Sect. 4.1). Since all of the results of Geometric Algebra operations on multivectors are again multivectors, we symbolically compute and simplify the resulting multivectors in order to determine which of the coefficients are actually needed and what is the simplest expression for each coefficient (in the Maple sense).

The table-based compilation approach does not need a commercial product such as Maple and is, in principle, open for arbitrary Geometric Algebras.

Gaalop can be downloaded from [58] free of charge.

10.3 Table-Based Compilation Approach

The table-based compilation approach of Gaalop, as introduced in [54], uses precomputed multiplication tables (see Sect. 10.3.1) in order to transform Geometric Algebra algorithms into a representation that does not contain any Geometric Algebra. In a second optimization step (see Sect. 10.3.7), Gaalop is able to perform

Table 10.2 Multiplication table for the geometric product of 2D Geometric Algebra. This algebra consists of the following basic algebraic objects: an object of grade (dimension) 0, the scalar; objects of grade 1, the two basis vectors e_1 and e_2; and an object of grade 2, the bivector $e_1 \wedge e_2$

	1	e_1	e_2	$e_1 \wedge e_2$
1	1	e_1	e_2	$e_1 \wedge e_2$
e_1	e_1	1	$e_1 \wedge e_2$	e_2
e_2	e_2	$-e_1 \wedge e_2$	1	$-e_1$
$e_1 \wedge e_2$	$e_1 \wedge e_2$	$-e_2$	e_1	-1

Table 10.3 Multiplication table of 2D Geometric Algebra in terms of its basis blades E_1, E_2, E_3 and E_4

		b	b_0	b_1	b_2	b_3
			E_0	E_1	E_2	E_3
a			1	e_1	e_2	$e_1 \wedge e_2$
a_0	E_0	1	E_0	E_1	E_2	E_3
a_1	E_1	e_1	E_1	E_0	E_3	E_2
a_2	E_2	e_2	E_2	$-E_3$	E_0	$-E_1$
a_3	E_3	$e_1 \wedge e_2$	E_3	$-E_2$	E_1	$-E_0$

further optimizations based on the symbolic-computation tool Maxima [77]. The benefits of this approach are:

- Computations with arbitrary algebras are possible;
- No commercial tool is needed (as in the Maple-based approach), since Maxima is free of charge;
- No external tool is needed, since the optimization of Maxima is optional.

10.3.1 Multiplication Tables

The multiplication tables needed for the table-based compilation approach are discussed in detail in this section.

In order to compute Geometric Algebra algorithms, the rules for the computation of the products of multivectors have to be known. These products can be summarized (and precomputed) in multiplication tables that describe the products of the various blades of the algebra. Table 10.2, for instance, presents the multiplication table for the geometric product of 2D Geometric Algebra. This algebra consists of the following basic algebraic objects: an object of grade (dimension) 0, the scalar; objects of grade 1, the two basis vectors e_1 and e_2; and an object of grade 2, the bivector $e_1 \wedge e_2$. We can recognize that the product of two blades is again a blade, sometimes with a negative sign. This is visualized in Table 10.3, which shows the product of two blades E_i and E_j in terms of another blade E_k with positive or negative sign.

The geometric product, the outer product, and the inner product are linear products. They are distributive over addition (see [29] or Chap. 3 of [81] for details).

Let us now compute the geometric product

$$x = ab = \left(\sum_i a_i E_i\right)\left(\sum_j b_j E_j\right) \tag{10.1}$$

of two arbitrary multivectors $a = \sum\limits_i a_i E_i$ and $b = \sum\limits_j b_j E_j$.

This can be written as

$$x = \sum_i \sum_j a_i b_j (E_i E_j) \tag{10.2}$$

or as a linear combination of blades $E_{i,j}$

$$x = \sum_i \sum_j a_i b_j (m_{i,j} E_{i,j}), \tag{10.3}$$

where $m_{i,j}$ is 0, 1 or -1, or as

$$x = \sum_i \sum_j c_{i,j} E_{i,j}, \tag{10.4}$$

with coefficients $c_{i,j} = m_{i,j} a_i b_j$. This can be rearranged to

$$x = \sum_k c_k E_k, \tag{10.5}$$

with

$$c_k = \sum_{i,j:E_{i,j}=E_k} m_{i,j} a_i b_j. \tag{10.6}$$

For details of the steps of the computation of products with multiplication tables, see Sect. 10.3.2.

This is described in Table 10.3 for 2D Euclidean Geometric Algebra . Each entry $m_{i,j} E_{i,j}$ describes the geometric product of two basis blades E_i and E_j, expressed in terms of the basis blades E_k with positive or negative sign. Each coefficient c_k of the product $x = ab$ can be computed by summing the products $\pm a_i * b_j$ based on the table entries for E_k, for instance, $c_0 = a_0 * b_0 + a_1 * b_1 + a_2 * b_2 - a_3 * b_3$ for the table entries for E_0.

Some examples of multiplication tables for 3D Euclidean Geometric Algebra can be found in the Tables 10.4–10.6. Table 10.4, for instance, describes the geometric products of the $8 = 2^3$ blades. Based on this information, the geometric product of two multivectors, each defined as a linear combination of all of the blades $v = \sum_{i=0}^{7} v_i E_i$, can easily be derived, as described in the caption of Table 10.4.

The same procedure can be used for other products. Table 10.6, for instance, describes the outer product of 3D Euclidean Geometric Algebra. Note that many of the entries are zero, corresponding to the outer product of two identical basis vectors $e_i \wedge e_i = 0$.

10.3.2 Table-Based Multiplication Algorithm

To clarify the concept, we provide below some simple algorithmic steps in pseudocode for the computation of the product of two general multivectors $a = \sum_i a_i E_i$ and $b = \sum_j b_j E_j$:

```
Set all blades of result multivector $c$ to zero;

for each blade coeff $a_i$ of multivector a
    for each blade coeff $b_j$ of multivector b
        Look up target blade $E_{i,j}$ and
        sign $m_{i,j}$ from multiplication table;
        ($i$ is the row index and
         $j$ is the column index.)

        Add simple arithmetic product $a_i * b_j$
        with sign $m_{i,j}$ to target blade $E_{i,j}$
        of result multivector $c$:
        $c[E_{i,j}] = c[E_{i,j}] + m_{i,j} * (a_i * b_j)$;
    end;
end;
```

10.3.3 Example

As an example, we will compute the expression

$$f = a \wedge (a + ab), \tag{10.7}$$

which contains two 3D vectors a and b. This can be expressed in terms of a CLUCalc script as follows:

```
  a=a1*e1+a2*e2+a3*e3;
  b=b1*e1+b2*e2+b3*e3;
  f=a^(a+a*b);
```

Table 10.4 Multiplication table describing the geometric product of two multivectors $a = \sum a_i E_i$ and $b = \sum b_i E_i$ for 3D Euclidean Geometric Algebra. Each coefficient c_k of the product $c = ab$ can be computed by summing the products $\pm a_i * b_j$ based on the table entries for E_k; for instance, $c_0 = a_0 * b_0 + a_1 * b_1 + a_2 * b_2 + a_3 * b_3 - a_4 * b_4 - a_5 * b_5 - a_6 * b_6 - a_7 * b_7$ for the table entries for E_0. In other words, a particular blade E_k of the result multivector c is computed by summing the products $\pm a_i * b_j$ of the table entries marked by E_k. See Sect. 10.3.2 for details of the algorithmic steps for computation with multiplication tables

	b	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7
a		1	e_1	e_2	e_3	e_{12}	e_{13}	e_{23}	e_{123}
E_0	1	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7
E_1	e_1	E_1	E_0	E_4	E_5	E_2	E_3	E_7	E_6
E_2	e_2	E_2	$-E_4$	E_0	E_6	$-E_1$	$-E_7$	E_3	$-E_5$
E_3	e_3	E_3	$-E_5$	$-E_6$	E_0	E_7	$-E_1$	$-E_2$	E_4
E_4	e_{12}	E_4	$-E_2$	E_1	E_7	$-E_0$	$-E_6$	E_5	$-E_3$
E_5	e_{13}	E_5	$-E_3$	$-E_7$	E_1	E_6	$-E_0$	$-E_4$	E_2
E_6	e_{23}	E_6	E_7	$-E_3$	E_2	$-E_5$	E_4	$-E_0$	$-E_1$
E_7	e_{123}	E_7	E_6	$-E_5$	E_4	$-E_3$	E_2	$-E_1$	$-E_0$

First, the compiler has to transform complex expressions to expressions that can easily be handled by multiplication tables. A corresponding CLUCalc script with only simple expressions could look as follows:

```
a=a1*e1+a2*e2+a3*e3;
b=b1*e1+b2*e2+b3*e3;
c=a*b;
d=a+c;
f=a^d;
```

Let us now compile this script step by step.

The first two lines are used for the definition of the two multivectors $a = a_1 E_1 + a_2 E_2 + a_3 E_3$ and $b = b_1 E_1 + b_2 E_2 + b_3 E_3$ (a_1, a_2, a_3, b_1, b_2, and b_3 are regular scalar variables). For both of these lines, only the entries 1, 2, and 3 are needed, since these entries correspond to the three basis vectors e_1, e_2, e_3. Table 10.5 shows the corresponding multiplication table for this product. This is derived from Table 10.4 with empty rows and columns for the multivector entries not needed for a and b. The resulting multivector c needs only the coefficients for the blades E_0, E_4, E_5, E_6 (see Table 10.5). Each coefficient $c[k]$ can be computed by summing the products $\pm a_i b_j$ based on the table entries for E_k , for instance $c_0 = a_1 b_1 + a_2 b_2 + a_3 b_3$.

The corresponding C++ code is as follows:

```
c[0]=a[1]*b[1]+a[2]*b[2]+a[3]*b[3];
c[4]=a[1]*b[2]-a[2]*b[1];
c[5]=a[1]*b[3]-a[3]*b[1];
c[6]=a[2]*b[3]-a[3]*b[2];
```

Table 10.5 Multiplication table describing the geometric product of two vectors $a = a_1e_1 + a_2e_2 + a_3e_3$ and $b = b_1e_1 + b_2e_2 + b_3e_3$ for 3D Euclidean Geometric Algebra. Note that all rows and columns for basis blades not needed for the vectors are set to zero

		b		$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{b_3}$				
			E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7
a			1	e_1	e_2	e_3	e_{12}	e_{13}	e_{23}	e_{123}
	E_0	1	0	0	0	0	0	0	0	0
$\mathbf{a_1}$	E_1	e_1	0	E_0	E_4	E_5	0	0	0	0
$\mathbf{a_2}$	E_2	e_2	0	$-E_4$	E_0	E_6	0	0	0	0
$\mathbf{a_3}$	E_3	e_3	0	$-E_5$	$-E_6$	E_0	0	0	0	0
	E_4	e_{12}	0	0	0	0	0	0	0	0
	E_5	e_{13}	0	0	0	0	0	0	0	0
	E_6	e_{23}	0	0	0	0	0	0	0	0
	E_7	e_{123}	0	0	0	0	0	0	0	0

In the fourth line of the CLUCalc script, the two multivectors a and c are added, resulting in the multivector d:

```
d[0]=c[0];
d[1]=a[1];
d[2]=a[2];
d[3]=a[3];
d[4]=c[4];
d[5]=c[5];
d[6]=c[6];
```

This sets the coefficients of the blades [0, 4, 5, 6] of the multivector c as coefficients of the blades [0, 4, 5, 6] of the multivector d. The coefficients [1, 2, 3] of a are set as coefficients [1, 2, 3] of d.

The evaluation of the outer product $a \wedge d$ of a with this multivector d leads to

```
f[1]=a[1]*d[0];
f[2]=a[2]*d[0];
f[3]=a[3]*d[0];
f[4]=a[1]*d[2]-a[2]*d[1];
f[5]=a[1]*d[3]-a[3]*d[1];
f[6]=a[2]*d[3]-a[3]*d[2];
f[7]=a[1]*d[6]-a[2]*d[5]+a[3]*d[4];
```

The multiplication table in Table 10.6 can be used for this computation. Associating the rows with the multivector a and the columns with d, we can set rows 0, 4, 5, 6, and 7 and column 7 to 0. We recognize that the remaining entries are for the coefficients 1, 2, 3, 4, 5, 6, and 7, for instance E_1 in the second row, and the first column is associated with the product $a_1 * d[0]$. For clarity, the needed table entries are shown in Table 10.7.

Table 10.6 Multiplication table describing the outer product of two general multivectors $a = \sum a_i E_i$ and $b = \sum b_i E_i$ for 3D Euclidean Geometric Algebra

	b	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7
a		1	e_1	e_2	e_3	e_{12}	e_{13}	e_{23}	e_{123}
E_0	1	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7
E_1	e_1	E_1	0	E_4	E_5	0	0	E_7	0
E_2	e_2	E_2	$-E_4$	0	E_6	0	$-E_7$	0	0
E_3	e_3	E_3	$-E_5$	$-E_6$	0	E_7	0	0	0
E_4	e_{12}	E_4	0	0	E_7	0	0	0	0
E_5	e_{13}	E_5	0	$-E_7$	0	0	0	0	0
E_6	e_{23}	E_6	E_7	0	0	0	0	0	0
E_7	e_{123}	E_7	0	0	0	0	0	0	0

Table 10.7 Part of the multiplication table in Table 10.6 describing the outer product of two specific multivectors $a = \sum_{i=1}^{3} a_i E_i$ and $d = \sum_{i=0}^{6} d_i E_i$ for 3D Euclidean Geometric Algebra

	d	E_0	E_1	E_2	E_3	E_4	E_5	E_6
a		1	e_1	e_2	e_3	e_{12}	e_{13}	e_{23}
E_1	e_1	E_1	0	E_4	E_5	0	0	E_7
E_2	e_2	E_2	$-E_4$	0	E_6	0	$-E_7$	0
E_3	e_3	E_3	$-E_5$	$-E_6$	0	E_7	0	0

10.3.4 Cascading Multiplications

The evaluation of

$$b = e_2 \wedge e_3 \wedge e_1 = e_2 \wedge (e_3 \wedge e_1), \tag{10.8}$$

for instance, can be done by using Table 10.6 twice. Calculating $e_3 \wedge e_1$ results in $-E_5$, and

$$b = e_2 \wedge (-E_5) \tag{10.9}$$

results in

$$b = -(-E_7) = E_7 = e_{123}. \tag{10.10}$$

In general, products with n multivector operands lead to computations of sums of products with n factors.

According to Sect. 10.3.1, each coefficient of the result of a multiplication can be computed as a sum of products

$$x_p = \sum_{i,j:E_{i,j}=E_p} (m_{i,j} a_i) b_j, \tag{10.11}$$

with two factors $m_{i,j}a_i$ and b_j. For instance, the geometric product

$$y = abc = \left(\sum_i a_i E_i\right)\left(\sum_j b_j E_j\right)\left(\sum_k c_k E_k\right) \tag{10.12}$$

of three arbitrary multivectors $a = \sum\limits_i a_i E_i$, $b = \sum\limits_j b_j E_j$ and $\sum\limits_k c_k E_k$ can be written as

$$y = abc = \left(\sum_p x_p E_p\right)\left(\sum_k c_k E_k\right) \tag{10.13}$$

or

$$y = \sum_p \sum_k x_p c_k (E_p E_k), \tag{10.14}$$

or as a linear combination of blades $E_{p,k}$,

$$y = \sum_p \sum_k x_p c_k (m_{p,k} E_{p,k}), \tag{10.15}$$

where $m_{p,k}$ is 0, 1 or -1, or as

$$y = \sum_p \sum_k y_{p,k} E_{p,k}, \tag{10.16}$$

with coefficients $y_{p,k} = m_{p,k} x_p c_k$. This can be rearranged to

$$y = \sum_q y_q E_q, \tag{10.17}$$

where

$$y_q = \sum_{p,k:E_{p,k}=E_q} m_{p,k} x_p c_k, \tag{10.18}$$

$$y_q = \sum_{p,k:E_{p,k}=E_q} m_{p,k} \left(\sum_{i,j:E_{i,j}=E_p} m_{i,j} a_i b_j\right) c_k, \tag{10.19}$$

or

$$y_q = \sum_{p,k:E_{p,k}=E_q} \sum_{i,j:E_{i,j}=E_p} m_{p,k} m_{i,j} a_i b_j c_k, \tag{10.20}$$

which is equal to a sum of products with three factors $a_i b_j c_k$ with signs defined by $m_{p,k} m_{i,j}$.

Table 10.8 A subset of the 5D geometric-product multiplication table of Geometric Algebra. This lists only 5 out of the 32 possible blades for each multivector. The expression E denotes the outer product $E = \mathbf{e}_\infty \wedge \mathbf{e}_0$

	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_3$	$\mathbf{e}_\infty$	$\mathbf{e}_0$
$\mathbf{e}_1$	1	$\mathbf{e}_{12}$	$-\mathbf{e}_{31}$	$\mathbf{e}_1 \wedge \mathbf{e}_\infty$	$\mathbf{e}_1 \wedge \mathbf{e}_0$
$\mathbf{e}_2$	$-\mathbf{e}_{12}$	1	$\mathbf{e}_{23}$	$\mathbf{e}_2 \wedge \mathbf{e}_\infty$	$\mathbf{e}_2 \wedge \mathbf{e}_0$
$\mathbf{e}_3$	$\mathbf{e}_{31}$	$-\mathbf{e}_{23}$	1	$\mathbf{e}_3 \wedge \mathbf{e}_\infty$	$\mathbf{e}_3 \wedge \mathbf{e}_0$
$\mathbf{e}_\infty$	$-\mathbf{e}_1 \wedge \mathbf{e}_\infty$	$-\mathbf{e}_2 \wedge \mathbf{e}_\infty$	$-\mathbf{e}_3 \wedge \mathbf{e}_\infty$	0	$-1 + E$
$\mathbf{e}_0$	$-\mathbf{e}_1 \wedge \mathbf{e}_0$	$-\mathbf{e}_2 \wedge \mathbf{e}_0$	$-\mathbf{e}_3 \wedge \mathbf{e}_0$	$-1 - E$	0

10.3.5 Linear Operation Tables

Some operations, such as taking the reverse, are linear operations, meaning that they are distributive over addition (see [29] for details). For instance, the reverse of a multivector a,

$$\tilde{a} = \widetilde{\left(\sum_i a_i E_i\right)}, \tag{10.21}$$

is equal to

$$\tilde{a} = \sum_i a_i \tilde{E}_i. \tag{10.22}$$

Every coefficient a_i is assigned to its corresponding blade E_i. The reverse operation assigns a_i to its mutually exclusive reverse blade $\tilde{E}_i$.

10.3.6 Multiplication Tables with a Non-Euclidean Metric

Geometric Algebra Computing with non-Euclidean metrics can be a little more complicated than with standard Euclidean metrics. In particular, this leads to more complex multiplication tables for CGA, where sums of blades are valid entries. The geometric product of $\mathbf{e}_\infty$ and $\mathbf{e}_0$, for instance, can be written as the following difference of the two blades $\mathbf{e}_\infty \wedge \mathbf{e}_0$ and 1, as in Table 10.8.

$$\mathbf{e}_\infty \mathbf{e}_0 = \mathbf{e}_\infty \wedge \mathbf{e}_0 - 1 = E - 1. \tag{10.23}$$

In terms of the corresponding multiplication table entry $E_{i,j}$, this means that the product $a_i * b_j$ has to be considered not only for one blade but also for two blades.

10.3.7 Additional Symbolic Optimizations Using Maxima

The table-based approach on its own already produces quality code. Nevertheless, there is still some potential for additional optimizations, such as

- Support for multiplications with more than two operands (if this is not done by cascading multiplications as described in Sect. 10.3.4);
- Merging of several algorithmic steps into one optimized step;
- Symbolically finding additional zero entries in multivectors.

In order to use this potential for optimization, CLUScripts have to be extended by question marks "?" at the beginning of statements, in order to indicate which multivectors have to be computed explicitly. All other multivectors can be treated as intermediate results to improve the runtime performance (see the example below).

The result for the example considered in Sect. 10.3.3 can be improved further. In the following CLUScript, for instance, the same multivectors as before have to be computed, except for the multivector d, which can be treated as an intermediate result:

```
?a=a[1]*e1+a[2]*e2+a[3]*e3;
?b=b[1]*e1+b[2]*e2+b[3]*e3;
?c=a*b;
d=a+c;
?f=a^d;
```

Inserting the (intermediate) result for d (see Sect. 10.3.3) into f leads to the following (not yet optimized) result for f:

```
f[1]=a[1]*c[0];
f[2]=a[2]*c[0];
f[3]=a[3]*c[0];
f[4]=a[1]*a[2]-a[2]*a[1];
f[5]=a[1]*a[3]-a[3]*a[1];
f[6]=a[2]*a[3]-a[3]*a[2];
f[7]=a[1]*c[6]-a[2]*c[5]+a[3]*c[4];
```

This can be optimized further by the symbolic-computing engine, since the multivector entries 4, 5, and 6 lead to zero entries. For instance, $a[1] * a[2]$ is equal to $a[2] * a[1]$. Their difference is therefore zero, being the right-hand side of $f[4]$. The same rule applies to the right-hand sides of $f[5]$ and $f[6]$:

```
f[1]=a[1]*c[0];
f[2]=a[2]*c[0];
f[3]=a[3]*c[0];
f[7]=a[1]*c[6]-a[2]*c[5]+a[3]*c[4];
```

The computation of the multivector entries f[4], f[5], and f[6] and of the entire multivector d is now no longer needed.

Chapter 11
Collision Detection Using the Gaalop Precompiler

In order to simplify the use of the Geometric Algebra Computing technology, we have developed Gaalop GPC [20], a precompiler, which integrates Gaalop into standard programming languages such as C++, OpenCL, and CUDA. Figure 11.1 outlines the concept for the C++ programming language. With Gaalop GPC, we are able to enhance ordinary C++ code with Geometric Algebra code and automatically generate optimized C++ code.

This chapter provides information on the basic concept of Gaalop GPC and describes its application to the horizon example and a collision detection application.

11.1 Basic Concept of Gaalop GPC

A precompiler is an elegant way of extending the features of a programming language. For Geometric Algebra Computing, it is of high interest to use both the power of high-performance languages such as C++/OpenCL/CUDA and the elegance of expression of a domain-specific language such as CLUCalc. We have therefore embedded CLUCalc code into C++/OpenCL and CUDA code, and compile it by utilizing the precompiler concept and the fast optimizations and code generation features of Gaalop.

Gaalop GPC enhances standard programs with:

- Embedding of Geometric Algebra code using multivectors;
- Functionality to interact with multivectors.

It transforms them to optimized standard programs without any explicit Geometric Algebra functionality.

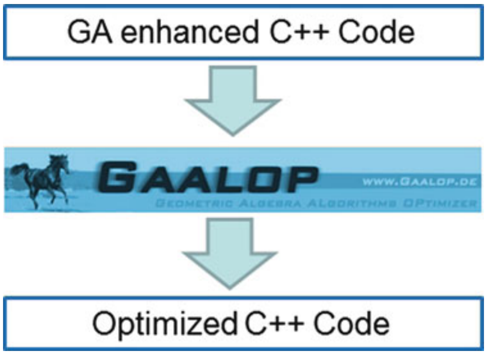


Fig. 11.1 Gaalop GPC for C++

The embedding of Geometric Algebra code is done based on pragmas with the following structure:

```
#pragma gpc begin
  ...
  Import of multivectors (if needed)
  ...
  #pragma clucalc begin
    ...
    Geometric Algebra code based on CLUCalc
    ...
  #pragma clucalc end
  ...
  Export of multivectors (if needed)
  ...
#pragma gpc end
```

The embedding of Geometric Algebra code in standard programs needs functionality to interact with multivectors. Several multivector functions are defined in Table 12.1. The purpose of these functions is to provide transformations between multivectors and C++/OpenCL/CUDA language concepts. In particular, we use the function **mv_get_bladecoeff()** for access to multivectors. This is responsible for extracting the coefficient of a blade from a multivector.

11.2 The Horizon Example Revisited in Gaalop GPC for C++

The horizon example described in Sect. 3.7 is implemented in C++ using Gaalop GPC as follows. Listing 11.1 computes and visualizes the observer point, the spheres S and K, and the horizon circle (Fig. 11.2). The multivectors for the variables P, S, K, C are computed explicitly (see the colons in the listing), while the radius r is only used implicitly as a constant.

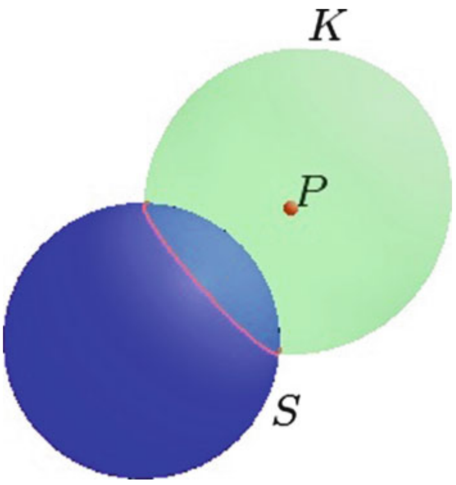


Fig. 11.2 Visualization of the horizon example

```
void horizon() {
  #pragma clucalc begin
    :Black;
    :P = VecN3(1,1,0);
     r = 1;
    :Blue;
    :S = e0-0.5*r*r*einf;
    :Color(0,1,0,0.2);
    :K = P+(P.S)*einf;
    :Red;
    :C = S^K;
  #pragma clucalc end
}
```

Listing 11.1 Code for horizon example in Gaalop GPC for C++ with visualization

The above example does not have any interaction with the surrounding C++ program. In contrast, the following example (Listing 11.2) is intended to illustrate the multivector access functionality. Note that "gp.h" has to be included in order to be able to use the Geometric Algebra functionality. Here, we are interested not only in the horizon circle but also in its center *homogeneousCenter* and the corresponding Euclidean 3D point *EuclideanCenter*.

```
#include <iostream>
#include <gp.h>

int main() {
  #pragma gpc begin
    #pragma clucalc begin
      P = VecN3(1,1,0);
      r = 1;
      S = e0-0.5*r*r*einf;
      C = S^(P+(P.S)*einf);
      ?homogeneousCenter = C*einf*C;
      ?scale = -homogeneousCenter.einf;
      ?EuclideanCenter = homogeneousCenter / scale;
    #pragma clucalc end
    std::cout << mv_get_bladecoeff(EuclideanCenter,e1)
    << "," << mv_get_bladecoeff(EuclideanCenter,e2)
    << "," << mv_get_bladecoeff(EuclideanCenter,e3);
  #pragma gpc end
  return 0;
}
```

Listing 11.2 Code for the example *horizon.cpg* in Gaalop GPC

We use the values for the point P, the radius r, the sphere S, and the horizon circle C as intermediate results only. Since they are computed only implicitly, this leads to a higher runtime performance. We explicitly compute the center of the horizon circle based on the multivector variable $homogeneousCenter$ (see Sect. 3.3), compute its scale factor $scale$ and the Euclidean coordinates of the center $EuclideanCenter$, and print the values of the center out based on the multivector access function **mv_get_bladecoeff**() (see Table 12.1).

11.3 Collision Detection

Collision detection is needed, for instance, in computational simulation of cloth. A piece of cloth in computer graphics might be constructed from a large number of triangles, making a connection between many points in three-dimensional space. All of these triangles may collide with other triangles on the same piece of cloth, which is called self-collision, with other pieces of cloth, or even with rigid bodies. Theoretically, to check for collisions between the triangles, we must assume that they are all potential colliders, and we must therefore check for collisions between every triangle and every other triangle and with every other object. Without any further information, this test would not be computationally manageable for larger scenes. The common methods used to solve this problem are primarily hierarchical methods to break down the number of tests. Such a method might, for example, be to use a hierarchy of spheres (see, for instance, the bounding-sphere algorithm using CGA in Chap. 22 of [7]) that can be traversed by the following rules:

1. Perform a sphere–sphere test, starting with the root level.
2. If we have no collision, then stop the test. If we do have a collision, then recursively traverse the underlying spheres (broad-phase testing).
3. If we hit the leaves of the hierarchy, perform tests on the underlying geometry, namely the triangles. Those tests come down to two individual cases (narrow-phase testing):
   - A point-versus-triangle test,
   - An edge-versus-edge test.

As an example, we present the **point–triangle test** in the following. The triangle has a thickness h and is a prism, mathematically. Figure 11.3 illustrates the triangle,

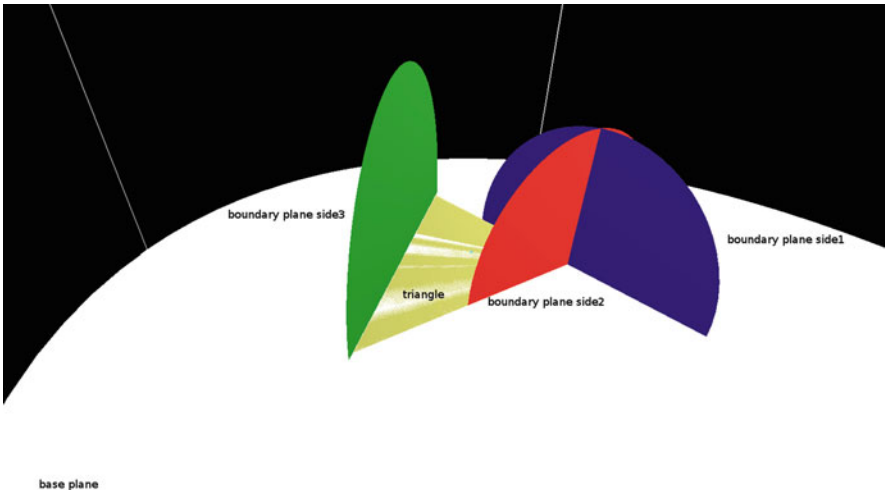


Fig. 11.3 Point–triangle intersection in CLUCalc. This picture shows the triangle, the plane it is embedded in, and its three boundary planes

together with the planes of the boundary faces of the triangular prism. These boundary planes include one edge of the triangle and are perpendicular to the base plane of the triangle. For convenience, we shall still call the triangular prism a "triangle".

Listing 11.3 is based on Gaalop GPC for C++ and performs a test for a collision between a triangle t and a point p

```
bool pointTriangleTest(const float t1x, const float t1y,
const float t1z, const float t2x, const float t2y, const float t2z,
const float t3x, const float t3y, const float t3z, const float px,
const float py, const float pz, const float h) {

  #pragma gpc begin
    #pragma clucalc begin
      TrianglePoint1 = VecN3(t1x, t1y, t1z);
      TrianglePoint2 = VecN3(t2x, t2y, t2z);
      TrianglePoint3 = VecN3(t3x, t3y, t3z);
      TestPoint = VecN3(px, py, pz);

      // construct base plane
      plane=*(TrianglePoint1 ^ TrianglePoint2 ^ TrianglePoint3 ^ einf);
      // compute signed distance of TestPoint to base plane
      ?d = plane . TestPoint;
      // extract triangle normal
      ?normal_ = plane - (plane . e0) ^ einf;

      // construct boundary planes
      side1 = *(TrianglePoint1 ^ TrianglePoint2 ^ normal_ ^ einf);
      side2 = *(TrianglePoint2 ^ TrianglePoint3 ^ normal_ ^ einf);
      side3 = *(TrianglePoint3 ^ TrianglePoint1 ^ normal_ ^ einf);
      // compute distances
      ?d1 = side1 . TestPoint;
      ?d2 = side2 . TestPoint;
      ?d3 = side3 . TestPoint;
    #pragma clucalc end
    const float d_SCALAR = mv_get_bladecoeff(d,1);
    const float d1_SCALAR = mv_get_bladecoeff(d1,1);
    const float d2_SCALAR = mv_get_bladecoeff(d2,1);
    const float d3_SCALAR = mv_get_bladecoeff(d3,1);
  #pragma gpc end

  if (d_SCALAR * d_SCALAR > h * h)
        return false;
  if (d1_SCALAR <= 0.0f && d2_SCALAR <= 0.0f
              && d3_SCALAR <= 0.0f || d1_SCALAR >= 0.0f
              && d2_SCALAR >= 0.0f && d3_SCALAR >= 0.0f)
        return true;
  return false;
}
```

Listing 11.3 A collision detection test written with Gaalop GPC for C++, checking for the collision of a “triangle” t of thickness h with a point p

The code executes the following steps.

1. Define the three “triangle points” (the vertices of the triangle) and the test point based on the corresponding C++ variables (t1x, t1y, ...).
2. Construct the base plane based on the triangle points (see Sect. 3.1.3).
3. Compute the signed distance d between the base plane and the test point based on the inner product (see Sect. 3.4.1).
4. Compute the normal vector to the plane.
5. Using this normal vector and the triangle points, compute the boundary planes, for example, the planes that are perpendicular to the base plane and pass through every combination of the three points.
6. Compute the signed distances d_1, d_2, and d_3 between the test point and the three boundary planes and assign them to C++ variables.
7. The condition for a collision is satisfied if d^2 is less than or equal to the square of the thickness h of the triangle, and all signed distances d_1, d_2, and d_3 have the same sign.

Note that algorithms similar to this one occur in the field of ray tracers. A recent work based on CGA is [18]; it shows very promising results.

Chapter 12
The Gaalop Precompiler for GPUs

Thanks to powerful GPGPU techniques (see for instance [45]), one can expect impressive results using the powerful language of Geometric Algebra. In this chapter, we present the basics of Gaalop GPC for GPUs, a precompiler for parallel programming of heterogeneous systems using OpenCL and CUDA. While CUDA is vendor-specific, OpenCL is an open industry standard, maintained by the Khronos Group [68].

The purpose of Gaalop GPC for OpenCL is to enhance ordinary OpenCL code with Geometric Algebra code and generate OpenCL code, optimized by Gaalop, as outlined in Fig. 12.1. Gaalop GPC may be adapted to new developments in High Performance Computing on GPUs like OpenACC or OpenHMPP as those increase in popularity.

GPUs are able to execute thousands of so-called kernels in parallel. These kernels execute the same code with different data. The kernels are managed by one host. Currently, the most efficient way to provide communication between the host and the kernels is to use strided arrays, as introduced in the next section.

12.1 Strided Arrays

In the following examples, it is very important to understand the concept of a strided array. We will therefore briefly explain it.

Nonstrided arrays are simply a concatenation of instances in memory. For example, having a structure

```
struct Vec {
        float x;
        float y;
        float z;
};
```

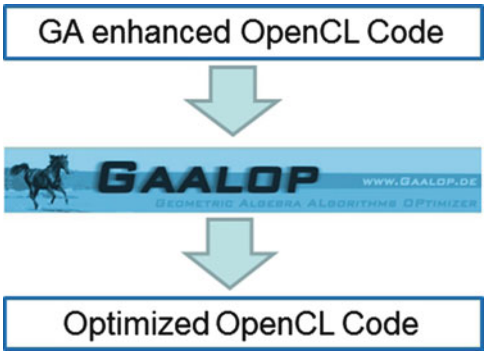


Fig. 12.1 Gaalop GPC for OpenCL

and creating an array containing N instances of this structure will yield the following image in memory:

x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	$\dots$	x_{N-1}	y_{N-1}	z_{N-1}

Strided arrays are simply a concatenation of individual structure elements in memory, thereby breaking with the traditional memory layout: As can be seen, we

x_0	x_1	x_2	$\dots$	x_{N-1}	y_0	y_1	y_2	$\dots$	y_{N-1}	z_0	z_1	z_2	$\dots$	z_{N-1}

now put into the array all x-elements of all structures, followed by all y-elements of all structures, followed by all z-elements. This may seem unusual at first, but yields coalesced memory access when the GPU threads read their data into the register space. Coalesced reads are by far the fastest reads possible from GPU global memory. Similar advantages arise from coalesced writes, which also occur in the following examples.

12.2 The Horizon Example on a GPU

We now describe how the horizon example, as presented in Sect. 3.7, is implemented in Gaalop GPC for OpenCL and CUDA.

12.2.1 *OpenCL Implementation*

The main difference from the implementation in Sect. 11.2 is that the horizon calculation is done in a kernel and not in a main program.

The CLUCalc code of the horizon example may be embedded into an OpenCL kernel, resulting in the code in Listing 12.1.

```
__kernel void horizonKernel(__global
float* circleCenters, __global const float* points,
const unsigned int num_points)
{
  const int id = get_global_id(0);
  #pragma gpc begin
    P = VecN3(points[id],
              points[id+num_points],
              points[id+2*num_points]);
    #pragma clucalc begin
      r = 1;
      S = e0-0.5*r*r*einf;
      C = S^(P+(P.S)*einf);

      ?homogeneousCenter = C*einf*C;
      ?scale = -homogeneousCenter.einf;
      ?EuclideanCenter = homogeneousCenter / scale;
    #pragma clucalc end
    circleCenters = mv_to_stridedarray(EuclideanCenter,
                                        id, num_points, e1,e2,e3);
  #pragma gpc end

}
```

Listing 12.1 Horizon example in OpenCL

First, each kernel has to request its identifier **id**, which is used to obtain its Euclidean observer point P from the strided array **points**. The actual computation is equivalent to the sequential example in Sect. 11.2. Finally, the e_1, e_2, e_3-components of the multivector **EuclideanCenter** are written to the strided array **circleCenters** (see Table 12.1 for the functions for constructing and accessing multivectors).

12.2.2 CUDA Implementation

The CLUCalc code for the horizon example may be embedded into a CUDA kernel, resulting in the code in Listing 12.2.

```
__kernel void horizonKernel(__global float* circleCenters,
__global const float* points)
{
  const int id = get_global_id(0);

  #pragma gpc begin
    P = VecN3(points[id],
              points[id+num_points],
              points[id+2*num_points]);
    #pragma clucalc begin
      r = 1;
      S = e0-0.5*r*r*einf;
      C = S^(P+(P.S)*einf);

      ?homogeneousCenter = C*einf*C;
      ?scale = -homogeneousCenter.einf;
      ?EuclideanCenter = homogeneousCenter / scale;
    #pragma clucalc end
    circleCenters = mv_to_stridedarray(EuclideanCenter,
                                         id, num_points, e1,e2,e3);
  #pragma gpc end
}
```
Listing 12.2 Horizon example in CUDA

12.3 List of Multivector Functions

Several multivector functions are defined in Table 12.1. The purpose of these functions is the transformation between multivectors and C++/OpenCL/CUDA language concepts such as **float** variables, arrays, and vectors. For example, **mv_get_bladecoeff**() is responsible for extracting a blade coefficient from a multivector, whereas **mv_from_array**() constructs a multivector from a C-like array.

Table 12.1 Gaalop GPC functions for constructing and accessing multivectors

coeff = mv_getbladecoeff(mv,blade);	Get the coefficient of blade **blade** of multivector **mv**
array = mv_to_array(mv, blades ,...);	Write the blades **blades ,...** of multivector **mv** to array **array**. Example **array = mv_to_array (mv,e1,e2,e3,e0, einf);**
array = mv_to_stridedarray(mv,index,stride ,blades ,...);	Write the blades **blades ,...** of multivector **mv** to array **array** with stride **stride**. Example **array = mv_to_array (mv,nummvs, e1,e2,e3,e0, einf);**
vector = mv_to_vector(mv, blades ,...);	Write the multivector **mv** to vector **vector**
mv = mv_from_vector(vector,blades ,..);	Construct multivector **mv** from vector **vector**
mv = mv_from_array(array,blades,..);	Construct multivector **mv** from array **array**
mv = mv_from_stridedarray(array,index,stride,blades ,...);	Construct multivector **mv** from array **array** with stride **stride**
mv_draw(mv,color);	Visualizes the geometric object represented by the multivector **mv**. The optional **color** parameter describes the R, G, B, A values of the color of the object

Chapter 13 Molecular Dynamics Using Gaalop GPC for OpenCL

A molecular dynamics simulation of the kind described here [101] models the point-pair interactions of a system of molecules, each molecule consisting of several atoms, and numerically solves Newton's and Euler's equations of motion for each molecule. This can be expressed in the mathematical language of Conformal Geometric Algebra . This chapter presents a molecular dynamics simulation based on Gaalop GPC for OpenCL (Fig. 13.1).

The nice result is that the Gaalop GPC implementation is faster than the conventional implementation, which is not self-evident for CGA-based implementations of such complexity. Further tests have shown that Gaalop GPC also yields a higher numerical stability in terms of energy conservation. This might be due to the fact that the advanced symbolic simplification performed by Gaalop GPC minimizes the number of operations, which otherwise would have been potential sources for numerical errors.

13.1 Molecular Dynamics in a Nutshell

In the following, we describe very briefly how molecular dynamics is modeled in our simulation (see Fig. 13.2):

- A molecule is a compound object consisting of several atoms, which are assumed to be static inside the molecule.
- Every atom sends out attraction or repulsion forces to every other atom.
- These forces then result in movement of the molecules according to Newton's and Euler's laws. This is simulated for thousands of molecules in parallel.

A potential function describes the dependance of the energy between two atoms and the distance between them. A popular approximation to the potential between real physical atoms is the Lennard-Jones potential (Fig. 13.3), which we make strong use of in this application.

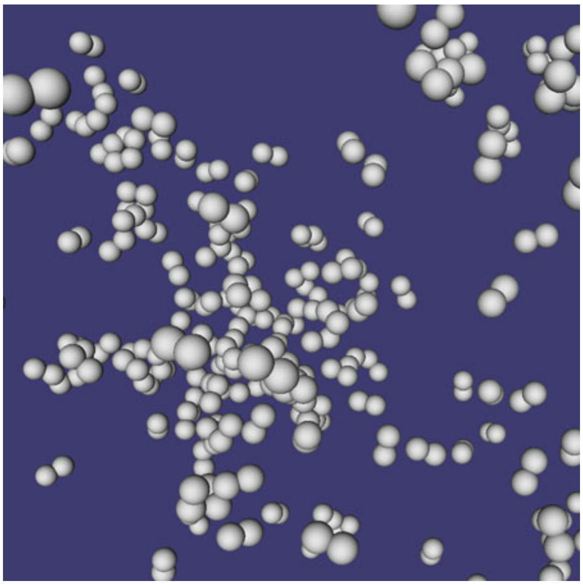


Fig. 13.1 Screenshot of a molecular dynamics simulation using CGA

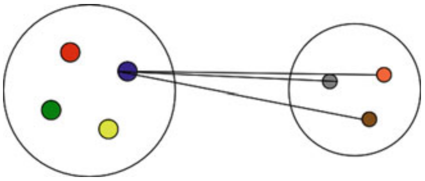


Fig. 13.2 The forces between all of the atoms in the molecules result in movement of the molecules

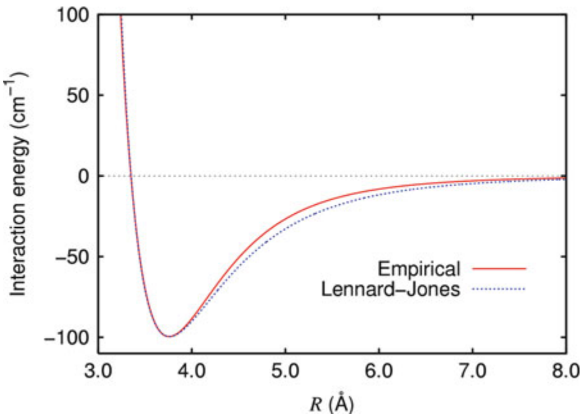


Fig. 13.3 The Lennard-Jones potential describes the dependence of the energy between two atoms and the distance between them (Image source: www.wikipedia.org)

The usual method here is to derive the forces on each atom from the potentials. Mathematically, and in the context of molecular dynamics, the force pointing in the direction of the lowest local energy is defined as the negative gradient $-\nabla\Phi(\mathbf{d})$, where $\mathbf{d} = \mathbf{p}_{ji}$ is the distance between the two atom positions and Φ is the Lennard-Jones potential function

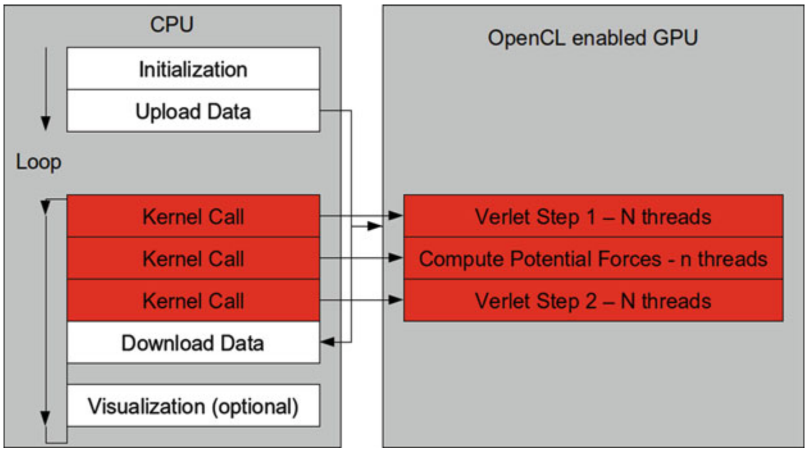


Fig. 13.4 Code architecture of the molecular dynamics application

$$\Phi(r) = 4\epsilon \left\{ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right\}, \tag{13.1}$$

where ϵ is a scale factor and σ is the distance at which the repulsive part outweighs the attractive part of the potential.

13.2 Software Architecture

An initialization on the host (CPU) is needed prior to running the simulation itself (see Sect. 13.3). This consists of loading a so-called MOLD file, which is basically a snapshot of the real physical state of the molecules, including their definition.

The actual simulation is done by the OpenCL solver for the molecular dynamics computations, initiated by some kernel calls. The OpenCL solver is separated into the following three parts (see Fig. 13.4):

1. **Molecule Verlet time integration step 1.** The kernel described in Sect. 13.4 updates a molecule's position and orientation; N computations are required for N molecules.
2. **Computation of potential forces.** This updates each molecule's force and torque; $n(n-1)$ computations are required for n atoms. The kernel described in Sect. 13.5 is responsible for this step.
3. **Molecule Verlet time integration step 2.** The kernel described in Sect. 13.6 updates the molecule's linear and angular velocities. N computations are required for N molecules.

The downloaded data can be used for the visualization of the motion of the molecules.

The following listings show code extracted from the OpenCL versions of the solver, implemented in Gaalop GPC for OpenCL.

13.3 Initialization

Listing 13.1 shows the host/CPU code for the initialization of the molecular dynamics simulation.

```
void convertStandardModelToSolverModel(const BaseModel& model) {
  const MoleculeVector& molecules = model.molecules;
  const AtomVector& atoms = model.atoms;
  const int numMolecules = molecules.size();
  const size_t numAtomPositions = atoms.size();
  for (int index = 0; index < numMolecules; ++index) {
    // get molecule
    const Molecule& molecule = molecules[index];

    #pragma gpc begin
      //map to multivectors
      lp = mv_from_array(molecule.lpos,e1,e2,e3);
      rotor = mv_from_array(molecule.arot,1,
                                e2^e3,e3^e1,e1^e2);
      lv = mv_from_array(molecule.lvel,e1,e2,e3);
      av = mv_from_array(molecule.avel,e1,e2,e3);

      #pragma clucalc begin
        // compute start values
        translator = 1 - 0.5 * lp^einf;
        ?D_in = translator*rotor;
        ?V_in = einf*lv - e1^e2^e3*av;
      #pragma clucalc end

      //map from multivectors
      host_mol_D0 = mv_to_stridedarray(D_in,index,numMolecules,
        1,e1^e2,e1^e3,
        e1^einf,e2^e3,
        e2^einf,e3^einf,
        e1^e2^e3^einf);
      host_mol_V0 = mv_to_stridedarray(V_in,index,numMolecules,
        e1^e2,e1^e3,
        e1^einf,e2^e3,
        e2^einf,e3^einf);
    #pragma gpc end

  }

  // fill device buffer, copy host buffer to device buffers
  commandQueue.enqueueWriteBuffer(dev_mol_D0.getBuffer(),
    CL_TRUE, 0,
        host_mol_D0.size() * sizeof(float), &host_mol_D0.front() );
  commandQueue.enqueueWriteBuffer(dev_mol_V0.getBuffer(),
     CL_TRUE, 0,
        host_mol_V0.size() * sizeof(float), &host_mol_V0.front() );
  std::fill(host_mol_V1.begin(), host_mol_V1.end(), 0.0f);
  commandQueue.enqueueWriteBuffer(dev_mol_V1.getBuffer(),
     CL_TRUE, 0,
        host_mol_V1.size() * sizeof(float), &host_mol_V1.front() );
  std::fill(host_mol_lmom.begin(), host_mol_lmom.end(), 0.0f);
  commandQueue.enqueueWriteBuffer(dev_mol_lmom.getBuffer(),
     CL_TRUE, 0,
        host_mol_lmom.size() * sizeof(float),
          &host_mol_lmom.front() );
  std::fill(host_mol_amom.begin(), host_mol_amom.end(), 0.0f);
  commandQueue.enqueueWriteBuffer(dev_mol_amom.getBuffer(),
     CL_TRUE, 0,
        host_mol_amom.size() * sizeof(float),
          &host_mol_amom.front() );

}
```

Listing 13.1 Gaalop GPC code for the conversion of the properties (position, orientation, and linear and angular velocity) of the molecules into the CGA representation as a versor **Din** and a velocity screw **Vin**

The goal is to convert the data of all the molecules into the Geometric Algebra repesentation and to prepare this data for the GPU. The location and orientation of each molecule is transformed to a displacement versor **Din** (see Sect. 3.5). The linear and angular velocity are defined through the molecule's velocity screw **Vin**, an expression for the combined linear and angular velocity (see Sect. 3.6 and [48, 50]).

The sequence of steps is as follows:

1. Take the position **lp** from the molecule position array **lpos** and the orientation **rotor** from the quaternion **arot**.
2. Take the linear velocity **lv** from the molecule velocity array **lvel** and the angular velocity **av** from the molecule array **avel**.
3. Define a **translator** from the Euclidean translation vector **lp**.
4. The displacement versor **Din** is simply the geometric product of **translator** and **rotor**.
5. The velocity screw **Vin** is defined according to (3.65), namely as the difference between the geometric product of e_∞ with the Euclidean linear-velocity vector **lv**, and the geometric product of $e_1 \wedge e_2 \wedge e_3$ with the Euclidean angular-velocity vector **av**.
6. Move the **Din** and **Vin** data for each molecule to the strided arrays **host_mol_D0** and **host_mol_V0**. These are versor-type multivectors with entries as indicated in Fig. 4.11, consisting of the scalar, six 2-blades, and one 4-blade. Note that versors are even multivectors with blades of even grade.
7. Copy all the host buffers to the corresponding device buffers.

13.4 Velocity Verlet Integration Step 1

Wisely chosen OpenCL kernels, such as the one in Listing 13.2, are called one or many times per frame. This kernel computes the first half of an implicit *velocity Verlet integration* for a molecule. A velocity Verlet numerical integrator propagates the position and velocity of a mass point for a given time step.

```
__kernel void verletStep1(__global float* array_D0,
                          __global float* array_V0,
                          __global const float* array_V1,
                          const float dt,
                          const unsigned int numMolecules) {
    // compute index
    const unsigned int index = get_global_id(0);

    // clamp
    if(index >= numMolecules)
        return;
    #pragma gpc begin
      //map to multivectors
      D0_t = mv_from_stridedarray(array_D0,index,numMolecules,
                                  1,e1^e2,e1^e3,e1^einf,
                                  e2^e3,e2^einf,e3^einf,
                                  e1^e2^e3^einf);
      V0_t = mv_from_stridedarray(array_V0,index,numMolecules,
                                  e1^e2,e1^e3,e1^einf,
                                  e2^e3,e2^einf,e3^einf);
      V1_t = mv_from_stridedarray(array_V1,index,numMolecules,
                                  e1^e2,e1^e3,e1^einf,
                                  e2^e3,e2^einf,e3^einf);
      #pragma clucalc begin
        ?D1_t = 0.5 * D0_t * V0_t;
        ?D2_t = 0.5 * D1_t * V0_t + 0.5 * D0_t * V1_t;

        ?D0_t_dt = D0_t + D1_t * dt + 0.5 * D2_t * dt * dt;
        ?V0_t_05dt = V0_t + 0.5 * V1_t * dt;
      #pragma clucalc end
        //map from multivectors
        array_D0 = mv_to_stridedarray(D0_t_dt,index,numMolecules,
                                      1,e1^e2,e1^e3,e1^einf,
                                      e2^e3,e2^einf,e3^einf,
                                      e1^e2^e3^einf);
        array_V0 = mv_to_stridedarray(V0_t_dt,index,numMolecules,
                                      e1^e2,e1^e3,e1^einf,
                                      e2^e3,e2^einf,e3^einf);
    #pragma gpc end
}
```

Listing 13.2 Compute-intensive Gaalop GPC for OpenCL code for the first step of the velocity Verlet numerical integration of the displacement versor and velocity screw for a molecule

The code performs the displacement propagation and computes the midpoint velocity, as described in the following equations:

Displacement propagation:	$D(t+\Delta t)$	$= D(t) + \dot{D}(t)\Delta t + \frac{1}{2}\ddot{D}(t)(\Delta t)^2.$
Midpoint velocity:	$V_b\left(t+\frac{\Delta t}{2}\right)$	$= V_b(t) + \dot{V}_b(t)\frac{\Delta t}{2}.$
Acceleration:	$\dot{V}_b(t+\Delta t)$	$= e_\infty \dot{v}_b(t+\Delta t) - e_{123}\dot{\omega}_b(t+\Delta t).$
Velocity propagation:	$V_b(t+\Delta t)$	$= V_b\left(t+\frac{\Delta t}{2}\right) + \frac{1}{2}\dot{V}_b(t+\Delta t)\Delta t.$

Here,

$$\dot{D} = \frac{1}{2}DV_b\left(= \frac{1}{2}V_b D = \frac{1}{2}DV_bD^{-1}D\right),$$

$$\ddot{D} = \frac{1}{2}\dot{D}V_b + \frac{1}{2}D\dot{V}_b = \frac{1}{4}DV_b^2 + \frac{1}{2}D\dot{V}_b,$$

and the Euclidean pseudoscalar is

$$e_{123} = e_1 \wedge e_2 \wedge e_3.$$

The state of a molecule is described by $D(t)$, its displacement versor in the inertial frame; $V_b(t)$, its velocity screw in the body frame ($V = DV_b(\tilde{D})$ in an inertial frame); $\dot{v}_b(t)$, its translational acceleration; and $\dot{\omega}_b(t)$, its rotational acceleration.

The convention for the variable names in the code is as follows: displacement versors and velocity screws are indicated by the letters D and V, followed by the number of differentiations (D1, for instance, means the first derivative of the displacement versor).

A more detailed description of this approach can be found in [101].

13.5 Accumulation of Forces Per Atom

After execution of the first part of the velocity Verlet algorithm, it is now required to update the forces and torques acting on all molecules.

The net force acting on a molecule, $\mathbf{f}_m$, is equal to the sum of the forces $\sum_i \mathbf{f}_i$ acting on its atoms. The net torque $\mathbf{t}_i$ acting upon a molecule is equal to the sum of the cross products $\sum_i \mathbf{f}_i \times \mathbf{r}_i = \sum_i -(\mathbf{f}_i \wedge \mathbf{r}_i)e_{123}$ of the molecule's atoms, where $\mathbf{r}_i$ is the position of atom i. In Sect. 2.3, we proved that the product $-(u \wedge v)e_{123}$ is equal to the cross product $u \times v$.

The kernel **__kernel void accumulateForcesPerAtom**() computes $\mathbf{f}_i$ and $\mathbf{f}_i \wedge \mathbf{r}_i$ on a per-atom basis. The summation is then performed consecutively in **void computeMoleculeForceAndTorque**(), as explained below.

```
__kernel void accumulateForcesPerAtom(
__global float* array_mol_lmom_temp,
__global float* array_mol_amom_temp,
__global const float* array_mol_D0,
__global const float* array_atom_pos,
__global const unsigned int* array_atom_pos_ind,
__global const unsigned int* array_atom_mol_ind,
const float epsilon,const float sigma,
const unsigned int numMolecules,
const unsigned int numAtoms,
const unsigned int numAtomPositions) {
    // compute index
    const unsigned int atom_index1 = get_global_id(0);

    // clamp
    if(atom_index1 >= numAtoms)
        return;

    // get atom data
    const unsigned int atom_mol_ind1
        = array_atom_mol_ind[atom_index1];

    /*
     * Precache a number of BLOCK_SIZE versors into fast local
            memory.
     * BLOCK_SIZE is a preprocessor definition supplied by the
     * host at compilation time.
     * Let every thread load from the strided global memory array
     * into non-strided local memory array in parallel.
     */
    __local float* versor1
        = &array_versor_cache_block[8 * get_local_id(0)];
    __local float array_versor_cache_block[8 * BLOCK_SIZE];
    {
        unsigned int shiftedIndex;
        versor1[0] = array_mol_D0[shiftedIndex
                                  = atom_mol_ind1];
        versor1[1] = array_mol_D0[shiftedIndex
                                  += numMolecules];
        versor1[2] = array_mol_D0[shiftedIndex
                                  += numMolecules];
        versor1[3] = array_mol_D0[shiftedIndex
                                  += numMolecules];
        versor1[4] = array_mol_D0[shiftedIndex
                                  += numMolecules];
        versor1[5] = array_mol_D0[shiftedIndex
                                  += numMolecules];
        versor1[6] = array_mol_D0[shiftedIndex
                                  += numMolecules];
        versor1[7] = array_mol_D0[shiftedIndex
                                  += numMolecules];
    }
    __local unsigned int array_atom_mol_ind2_block[BLOCK_SIZE];
    __local float4 array_gpos2_block[BLOCK_SIZE];

    float4 pos1,gpos1;
    {
        const unsigned int atom_pos_ind1
```

```
            = array_atom_pos_ind[atom_index1];
        pos1 = (float4)(array_atom_pos[atom_pos_ind1],
                        array_atom_pos[numAtomPositions
                        + atom_pos_ind1],
                        array_atom_pos[(numAtomPositions << 1)
                        + atom_pos_ind1],0.0f);

        //map to multivectors
        #pragma gpc begin
          p1 = VecN3(pos1);
          D1 = mv_from_array(versor1,
                             1,e1^e2,e1^e3,e1^einf,
                             e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
          #pragma clucalc begin
            // calculate
            ?gp1 = D1*p1*(~D1);
          #pragma clucalc end
          //map from multivectors
          gpos1 = mv_to_vector(gp1,e1,e2,e3);
        #pragma gpc end
    }

    // accumulate forces
    float4 accumulated_forces = (float4)(0.0f);
    {
        for(unsigned int atom_index2_block = 0;
            atom_index2_block < numAtoms;
            atom_index2_block += BLOCK_SIZE)
        computeLennardJonesForce(&accumulated_forces,gpos1,
                array_mol_D0,array_atom_pos,
                array_atom_mol_ind,array_atom_pos_ind,
                array_atom_mol_ind2_block,array_gpos2_block,
                epsilon,sigma,
                atom_mol_ind1,atom_index2_block,
                numMolecules,numAtoms,numAtomPositions);
    }

    // transform and save molecule's force and torque
    computeMoleculeForceAndTorque(array_mol_lmom_temp,
                                  array_mol_amom_temp,
                                  versor1,pos1,
                                  accumulated_forces,
                                  atom_index1,numAtoms);
}
```

The following two utility device functions,

- **void computeLennardJonesForce()**
- **void computeMoleculeForceAndTorque()**

are invoked by **__kernel void accumulateForcesPerAtom()**. Their sole purpose is the computation of the forces acting between two atoms in different molecules.

The catch here is that the atom positions are expressed in their molecule's frame. A transformation to inertial-frame coordinates is therefore required prior to computing the forces on them. The inertial-frame transformation $D_j \mathbf{p}_{ji} \tilde{D}_j$ with molecule versor D_j and atom position $\mathbf{p}_{ji}$ performs this job for atom 2, whereas the position of atom 1 has already been pretransformed by **__kernel void accumulateForcesPerAtom()** prior to calling the device function **void computeLennardJonesForce()**.

Once both atoms have been transformed, the computation of the Lennard-Jones potential forces is performed easily by **void computeLennardJonesForceSimple()**. Mathematically, the force pointing in the direction of the lowest local energy is defined as the negative gradient $-\nabla\Phi(\mathbf{d})$, where $\mathbf{d} = \mathbf{p}_{ji}$ is the distance between the two atom positions.

```
void computeLennardJonesForce(
float4* accumulated_forces,
const float4 gpos1,
__global const float* array_mol_D0,
__global const float* array_atom_pos,
__global const unsigned int* array_atom_mol_ind,
__global const unsigned int* array_atom_pos_ind,
__local unsigned int* array_atom_mol_ind2_block,
__local float4* array_gpos2_block,
const float epsilon, const float sigma,
const unsigned int atom_mol_ind1,
const unsigned int atom_index2_block,
const unsigned int numMolecules,
const unsigned int numAtoms,
const unsigned int numAtomPositions) {

    if(atom_index2_block + get_local_id(0) < numAtoms) {
        const unsigned int atom_mol_ind2
            = array_atom_mol_ind[atom_index2_block
            + get_local_id(0)];
        array_atom_mol_ind2_block[get_local_id(0)]
            = atom_mol_ind2;
        const unsigned int atom_pos_ind2
            = array_atom_pos_ind[atom_index2_block
            + get_local_id(0)];

      #pragma gpc begin
        D2 = mv_from_array(array_mol_D0, 1,e1^e2,e1^e3,e1^einf,
                             e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
        p2 = mv_from_stridedarray(array_atom_pos,
                 atom_pos_ind2,numAtomPositions,e1,e2,e3);
        #pragma clucalc begin
          // calculate
          ?gp2 = D2*p2*(~D2);
        #pragma clucalc end
        array_gpos2_block[get_local_id(0)]
            =mv_to_vector(gp2,e1,e2,e3);
      #pragma gpc end
    }
    // sync for shared memory consistency
    barrier(CLK_LOCAL_MEM_FENCE);

    // compute lennard jones force using transformed positions
    for(unsigned int index_block = 0;
        index_block < BLOCK_SIZE;
        ++index_block)
        if(atom_index2_block + index_block < numAtoms
           && atom_mol_ind1
           != array_atom_mol_ind2_block[index_block])
        computeLennardJonesForceSimple(accumulated_forces,
        gpos1, array_gpos2_block[index_block], epsilon, sigma);
}

void computeLennardJonesForceSimple(float4* accumulated_forces,
                                    const float4 pos1,
                                    const float4 pos2,
                                    const float epsilon,
                                    const float sigma) {
    // compute lennard jones potential force
    const float4 distVec = pos1 - pos2;
    const float distPow2 = distVec.x * distVec.x
                         + distVec.y * distVec.y
                         + distVec.z * distVec.z;
    const float distPow6 = distPow2 * distPow2 * distPow2;
    const float distPow8 = distPow6 * distPow2;
    const float distPow14 = distPow8 * distPow6;
    const float sigmaPow6 = sigma * sigma * sigma
                          * sigma * sigma * sigma;
    const float sigmaPow12 = sigmaPow6 * sigmaPow6;
    const float factor = (24.0f * epsilon)
                       * (sigmaPow12 / distPow14
                       - sigmaPow6 / distPow8);

    accumulated_forces[0].x += distVec.x * factor;
    accumulated_forces[0].y += distVec.y * factor;
    accumulated_forces[0].z += distVec.z * factor;
}
```

The code in Listing 13.3 is responsible for transforming a force into a molecule's local coordinate system, computing the torque by multiplying the transformed force by the position it acts upon, and saving both of these quantities in memory. The explicit steps are as follows:

1. Transform the force into the molecule body frame using the operation $\tilde{D} f D$, where f is the force and D is the molecule's versor.
2. The torque is computed by taking a simple outer product $\wedge$ of the application point of the force and the force itself. The operands and the result are expressed in terms of the body frame.

```
void computeMoleculeForceAndTorque(__global float* atom_lmom_temp,
                                   __global float* atom_amom_temp,
                                   __local const float* versor1,
                                   const float4 localPos,
                                   const float4 globalForce,
                                   const unsigned int atom_index,
                                   const unsigned int numAtoms)
{
    #pragma gpc begin
    //map to multivectors
    moleculeVersor = mv_from_array(versor1,
               1,e1^e2,e1^e3,e1^einf,
               e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
    #pragma clucalc begin
        posLocal = VecN3(localPos);
        forceGlobal = VecN3(globalForce);

      // calculate
      ?local_force = ~moleculeVersor
                   * forceGlobal
                   * moleculeVersor;
      ?local_torque = posLocal ^ local_force;
    #pragma clucalc end
      //map from multivectors
      atom_lmom_temp = mv_to_stridedarray(local_force,
          atom_index, numAtoms,e1,e2,e3);
      atom_amom_temp = mv_to_stridedarray(local_torque,
          atom_index, numAtoms,e1,e2,e3);
  #pragma gpc end
}
```

Listing 13.3 Gaalop GPC for OpenCL code for the computation of force and torque

13.6 Velocity Verlet Integration Step 2

This kernel performs the second step of the velocity Verlet implicit integration. Mathematically, it integrates acceleration and velocity propagation as defined below:

Acceleration:	$\dot{V}_b(t+\Delta t)$	$= e_\infty \dot{v}_b(t+\Delta t) - e_{123}\dot{\omega}_b(t+\Delta t)$
Velocity propagation:	$V_b(t+\Delta t)$	$= V_b\left(t+\frac{\Delta t}{2}\right) + \frac{1}{2}\dot{V}_b(t+\Delta t)\Delta t$

```
__kernel void verletStep2(__global float* array_V0,
                          __global float* array_V1,
                          __global float* array_lmom,
                          __global float* array_amom,
                          __global const float* array_masses,
                          __global const float* array_inertia,
                                    const float dt,
                                    const unsigned int numMolecules) {
    // compute index
    const unsigned int index = get_global_id(0);

    // clamp
    if(index >= numMolecules)
        return;

    #pragma gpc begin
      //map to multivectors
      lmom = mv_from_stridedarray(array_lmom,index,
           numMolecules,e1,e2,e3);
      amom = mv_from_stridedvec(array_amom,index,
           numMolecules,e1,e2,e3);
      V0 = mv_from_stridedarray(array_V0,index,numMolecules,
                            e1^e2,e1^e3,e1^einf,
                            e2^e3,e2^einf,e3^einf);
      const float mass = array_masses[index];
      const float I_1 = array_inertia[shiftedIndex
                                       = index];
      const float I_2 = array_inertia[shiftedIndex
                                      += numMolecules];
      const float I_3 = array_inertia[shiftedIndex
                                      += numMolecules];
      #pragma clucalc begin
        avel1_t = V023;
        avel2_t = V013;
        avel3_t = V012;
        v1_t_dt = lmom / mass;

        // temporary values
        w1_t_dt_1 = (am23 - (I_3 - I_2) * avel2_t * avel3_t) / I_1;
        w1_t_dt_2 = (am13 - (I_1 - I_3) * avel3_t * avel1_t) / I_2;
        w1_t_dt_3 = (am12 - (I_2 - I_1) * avel1_t * avel2_t) / I_3;
        w1_t_dt = e1 * w1_t_dt_1 + e2 * w1_t_dt_2 + e3 * w1_t_dt_3;

        // calculate verlet step 2
        ?V1_t_dt = einf * v1_t_dt - e1^e2^e3 * w1_t_dt;
        ?V0_t_dt = V0_t_05dt + 0.5 * V1_t_dt * dt;
      #pragma clucalc end
      //map from multivectors
      array_V0 = mv_to_stridedarray(V0_t_dt,index,numMolecules,
                                  e1^e2,e1^e3,e1^einf,
                                  e2^e3,e2^einf,e3^einf);
      array_V1 = mv_to_stridedarray(V1_t_dt,index,numMolecules,
                                  e1^e2,e1^e3,e1^einf,
                                  e2^e3,e2^einf,e3^einf);
    #pragma gpc end
}
```

Chapter 14
Geometric Algebra Computers

How should a computer for Geometric Algebra be designed? This chapter investigates different computing architectures with the goal of implementing Geometric Algebra algorithms with as high a performance as possible.

There are already pure hardware solutions available (see Sect. 1.3). We have realized a combined software and hardware solution based on Gaalop with reconfigurable FPGAs (see Sect. 14.1). Another goal of Gaalop is to adapt Geometric Algebra as much as possible to modern parallel computing platforms (see Sect. 14.2). In the hardware industry, the need for suitable parallel computing architectures is quite evident. After Geometric Algebra has been adapted to current architectures, future architectures might even be influenced and driven by the Geometric Algebra Computing technology. Therefore, we present an approach to a Geometric Algebra machine instruction set in Sect. 14.3. This has already been used as an internal representation in Gaalop, but will be used in future Geometric Algebra computers as instruction set (see Sect. 14.4).

14.1 FPGA Implementation of Geometric Algebra Algorithms

Here, we describe our approach to automatically generating FPGA implementations of Geometric Algebra algorithms based on Gaalop. This is joint work with the Embedded Applications Group of Professor Andreas Koch, financed by the Deutsche Forschungsgemeinschaft (DFG). Details can be found in the paper [66].

There are general FPGA (field programmable gate arrays) implementations for geometric products available, as indicated in Sect. 1.3. Because of the complexity of Geometric Algebra computations, however, they have some restrictions, for instance concerning the dimension of the algebra. Our approach using Gaalop (see Chap. 10) differs from these general solutions, as we compile Geometric Algebra algorithms first into simplified algorithms that can be handled more easily by

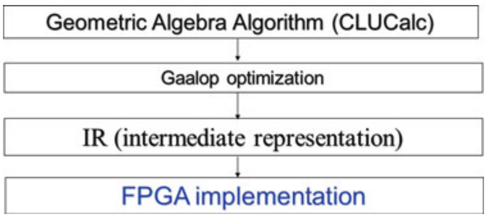

Geometric Algebra Algorithm (CLUCalc)
Gaalop optimization
IR (intermediate representation)
FPGA implementation

Fig. 14.1 Generation of optimized FPGA implementations from Geometric Algebra algorithms

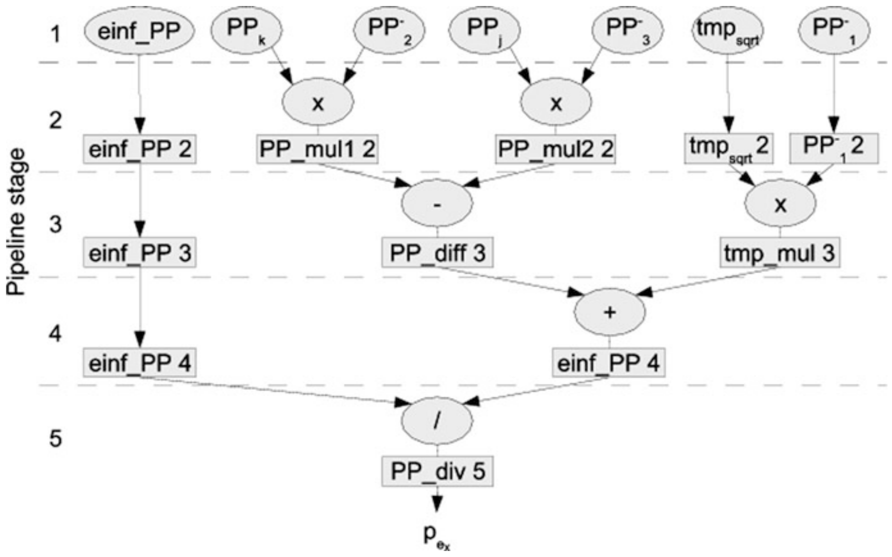


Fig. 14.2 Pipeline schedule for the coefficient p_{ex} of a multivector. All of the computations specified by (14.1) for all of the pipeline stages can be done in parallel

FPGAs; see Fig. 14.1 for our compilation process. These FPGA implementations are always application-specific. As one proof of concept of our approach, we implemented the inverse kinematics algorithm of Chap. 9. First, we used our Maple-based compilation approach, and the software implementation of the optimized algorithm became three times faster than the conventional solution [59].

The FPGA implementation of the optimized algorithm used the Verilog programming language. See Fig. 14.2 for the data flow and the pipeline schedule of the computation of the following part of the algorithm (one coefficient of one multivector according to Sect. 9.3.2.2):

$$p_{ex} = (PP_j(PP_{34} - PP_{35}) + PP_k(PP_{25} - PP_{24}) \qquad (14.1)$$

$$+tmp_{sqrt}(PP_{15} - PP_{14}))/einf\_PP.$$

The main advantage of this kind of implementation on reconfigurable hardware is that we are able to realize parallelism in two dimensions as follows:

- Computation of all coefficients of one or more multivectors in parallel;
- Use of the pipeline structure (computations are possible in all pipeline stages at the same time).

As a result, the implementation became about 300 times faster [60] (three times faster by software optimization and 100 times by additional hardware optimization).

14.2 Adaptation of Geometric Algebra to Current Computer Architectures

One goal of Geometric Algebra Computing is to combine the two recent trends illustrated in Fig. 1.1 in order to adapt Geometric Algebra algorithms as much as possible to current computer architectures [55]:

- The development of mathematics from the exterior algebra of Hermann Grassmann via Clifford algebra to the Geometric Algebra of David Hestenes and, especially, the 5D conformal model, and further development to higher-dimensional algebras, leading to an increasing number of engineering applications. Examples are the 6/8D Geometric Algebra of Christian Perwass, the 8D Geometric Algebra of Selig and Bayro-Corrochano [99] and the 9D Geometric Algebra of Julio Zamora [115]. Algebras for conics were presented in [81]. Selig and Bayro-Corrochano [99] showed how velocities, momenta, and inertia can be represented by elements of an 8D Geometric Algebra. Zamora [115] showed how quadrics can be represented in a 9D Geometric Algebra.
- During the past few decades, especially from 1986 to 2002, processor performance doubled every 18 months, based on higher clock frequencies. Owing to physical limitations, the primary method of gaining processor performance now is through parallelism, and we can recognize a shift to parallel systems; most likely, these systems will dominate in the future. Thanks to multicore architectures and powerful GPGPUs (see for instance [45]), especially based on the new OpenCL technology, one can expect impressive results using the powerful language of Geometric Algebra.

Most multivector calculations derived from the table-based compilation approach presented in Sect. 10.3 break down to scalar multiplications of equal-sized n-dimensional vectors. The instruction sets of modern microcomputers contain operations that are performed on vectors in parallel. A vector in this context is simply a concatenation of scalar values. Multiplying two vectors means multiplying all elements of vector a by the corresponding elements of vector b in parallel. Streaming SIMD (single instruction multiple data) extensions (SSEs) are an addition to the standard x86-platform instruction set. They provide a variety of instructions for computing the elements of vectors of four floating-point values concurrently and also provide special cache instructions to optimize the runtime performance. We can utilize SSE instructions to evaluate the dot product in a parallel way (see

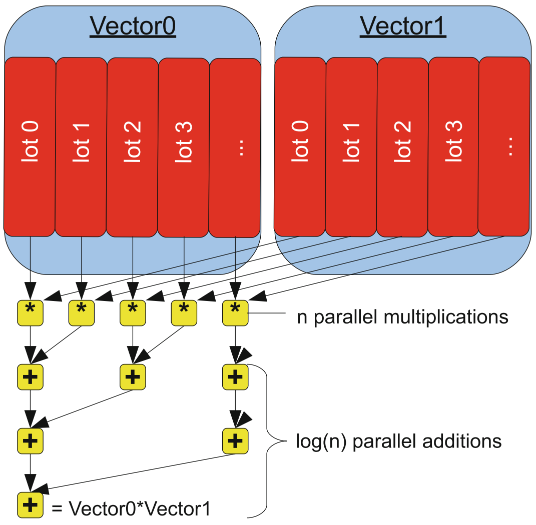


Fig. 14.3 Parallel dot product of two n-dimensional vectors Vector0 and Vector1 (n parallel products followed by $\log(n)$ parallel addition steps)

Fig. 14.3). The above statements are true in particular for modern GPUs. OpenCL provides vector operations that can be performed in parallel. The best performance of OpenCL kernels is obtained if they can execute the same instruction stream without too much branching. They benefit from the presence of fewer conditional statements in Geometric Algebra algorithms owing to its higher generality.

Finding such fine-grained parallelism is usually a hard task, and is a very common problem in compiler construction. Most interestingly, Geometric Algebra and the table-based approach intrinsically expose instruction-level parallelism. The Gaalop precompiler adapts Geometric Algebra as much as possible to modern parallel computing platforms (see Chap. 12).

14.3 Geometric Algebra Parallelism Programs (GAPP)

The scalar multiplication (or dot product) mentioned above, which is a recurring pattern in Geometric Algebra Computing, intrinsically contains a high level of parallelism. This motivates the abstract language defined in this section [57].

One goal of this new instruction set is to make it possible to develop future computing architectures in such a way that they are suited as much as possible to Geometric Algebra. The GAPP language is defined in Table 14.1. In the following sections, we will use code examples based on this language to explain its basics.

The dot product in our abstract GAPP language is defined by the *dotVectors* command. The language is designed to perform as many operations in parallel as possible. The dot product of two vectors is decomposed into one step consisting of parallel multiplications of each of their components and $log_2(n)$ parallel addition steps, where n is the dimensionality of the vectors; see the example in Fig. 14.3.

Table 14.1 The main commands of the Geometric Algebra Parallelism Programs (GAPP) language; a more detailed list can be found in [105]

Command:	**assignInputsVector**
Syntax:	assignInputsVector inputsVector = [var_1,var_2,...,var_n];
Description:	Assigns scalar inputs variables var_1, var_2, ..., var_n to the vector of the inputs.
Arguments:	var_i: the i-th scalar input variable.
Example:	assignInputsVector inputsVector = [x1,x2,x3,y1,y2,y3];
Command:	**resetMv**
Syntax:	resetMv *multivector*;
Description:	Creates a multivector and initializes it with zeros.
Arguments:	*multivector*: the name of the multivector which should be created.
Example:	resetMv v1;
Command:	**setMv**
Syntax:	setMv *dest*[sel_{dest_1},sel_{dest_2},...,sel_{dest_n}] = *src*[sel_{src_1},sel_{src_2},...,sel_{src_n}];
Description:	Copies certain blades from a source to a destination multivector. The order of the selector lists is important.
Arguments:	*dest*: the destination multivector. sel_{dest_i}: the i-th component of the destination multivector. *src*: the source multivector/vector. sel_{src_i}: the i-th component of the source multivector.
Example:	setMv v1[1,2] = inputsVector[0,3];
Command:	**setVector**
Syntax:	setVector *dest* = {arg_1,arg_2,...,arg_n};
Description:	Creates a vector from a list of arguments
Arguments:	*dest*: the destination vector. arg_i: the i-th argument to be assigned in the destination vector, which can be a constant (e.g. 2.0) or an extract from a multivector/vector (e.g. $mv[1,2]$).
Example:	setVector ve0 = {0.5,v1[1,2],v3[1],v1[4,5]};
Command:	**dotVectors**
Syntax:	dotVectors *dest*[*sel*] = ⟨$vector_1$,$vector_2$,...,$vector_n$⟩;
Description:	Calculates the dot product of the given vectors and stores it in a component of a multivector.
Arguments:	*dest*: the destination multivector. *sel*: the component index of the destination multivector.
Example:	dotVectors p1[4] = ⟨ve0,ve1,ve2⟩;
Command:	**assignMv**
Syntax:	assignMv *dest*[sel_{dest_1},sel_{dest_2},...,sel_{dest_n}] = [val_1,val_2,...,val_n];
Description:	Assigns values to components of a multivector.
Arguments:	*dest*: the destination multivector. sel_{dest_i}: The i-th component of the destination multivector. val_i: the i-th constant value.
Example:	assignMv p1[5,6] = [1.0,3.0];

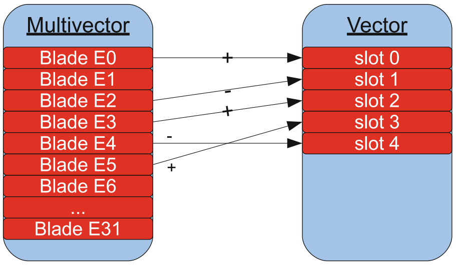


Fig. 14.4 Selected blades of a **multivector** are stored in a five-dimensional **vector** in the form E_0,$-E_2$, E_3, E_5, and $-E_4$

A vector in this sense is simply a concatenation of selected blades, as in the command *setVector*. It does not contain any information about the blades itself, which distinguishes it from a multivector. All of this information is removed by *setVector*. A vector is nothing more than the storage of signed real numbers, and is a preparation of the data for the actual computation performed by *dotVectors*. For example, if we select blades E_0,$-E_2$, E_3, E_5, and $-E_4$, these blades are stored in a five-dimensional vector in the same order, with the specified signs. Figure 14.4 visualizes this mapping of parts of multivectors to vector components.

Depending on the implementation, vectors may also be higher-dimensional, with all remaining elements set to zero, because zero elements do not have any impact on the result of calculating the dot product.

A blade selector consists of the index of a multivector entry and its sign. For CGA, the multivector entry index is equal to one of the indices in Table 10.1. For example, selecting the index 10 returns the coefficient of the blade $e_2 \wedge e_3$. Selecting the index -10 returns the negated coefficient of the same blade.

14.3.1 Example

As an example of the application of GAPP, we will use the example presented in Sect. 10.3.3 in the form

```
?a=a1*e1+a2*e2+a3*e3;
?b=b1*e1+b2*e2+b3*e3;
?c=a*b;
?d=a+c;
?f=a^d;
```

Note that this example is based on 3D Euclidean Geometric Algebra, whereas we have used mostly 5D CGA in this book.

Let us now compile this script step by step into a GAPP representation according to Table 14.1. We will also show the corresponding representation in C++ code for better understanding.

First, all of the input variables are stored in one specific vector:

```
assignInputsVector inputsVector = [a1,a2,a3,b1,b2,b3];
```

The next two lines are used for the definition of the multivector $a = a_1 E_1 + a_2 E_2 + a_3 E_3$ ($a1, a2, a3$ are regular scalar variables):

```
resetMv a;
setMv a[1,2,3] = inputsVector[0,1,2];
//a[1] = inputsVector[0]
//a[2] = inputsVector[1]
//a[3] = inputsVector[2]
```

The multivector $b = b_1 E_1 + b_2 E_2 + b_3 E_3$ is defined by

```
resetMv b;
setMv b[1,2,3] = inputsVector[3,4,5];
```

As stated earlier in Sect. 10.3.3, the optimized computation for the variable c is

```
c[0]=a[1]*b[1]+a[2]*b[2]+a[3]*b[3];
c[4]=a[1]*b[2]-a[2]*b[1];
c[5]=a[1]*b[3]-a[3]*b[1];
c[6]=a[2]*b[3]-a[3]*b[2];
```

Expressed in the GAPP language according to Table 14.1, the code looks as follows:

```
//c[0] = ((a[3] * b[3]) + (a[2] * b[2])) + (a[1] * b[1])
resetMv c;
setVector ve0 = {a[3,2,1]};
setVector ve1 = {b[3,2,1]};
dotVectors c[0] = <ve0,ve1>;

//c[4] = (a[1] * b[2]) - (a[2] * b[1])
setVector ve2 = {a[1,-2]};
setVector ve3 = {b[2,1]};
dotVectors c[4] = <ve2,ve3>;

//c[5] = (a[1] * b[3]) - (a[3] * b[1])
setVector ve4 = {a[1,-3]};
setVector ve5 = {b[3,1]};
dotVectors c[5] = <ve4,ve5>;

//c[6] = (a[2] * b[3]) - (a[3] * b[2])
setVector ve6 = {a[2,-3]};
setVector ve7 = {b[3,2]};
dotVectors c[6] = <ve6,ve7>;
```

In the fourth line of the CLUCalc script, the two multivectors a and c are added, resulting in the multivector d:

```
d[0]=c[0];
d[1]=a[1];
d[2]=a[2];
d[3]=a[3];
d[4]=c[4];
d[5]=c[5];
d[6]=c[6];
```

In the GAPP language,

```
resetMv d;
setMv d[0,4,5,6] = c[0,4,5,6];
setMv d[1,2,3] = a[1,2,3];
```

This sets the coefficients of blades [0,4,5,6] of the multivector c as the coefficients of blades [0,4,5,6] of the multivector d. The coefficients [1,2,3] of a are set as the coefficients [1,2,3] of d.

The evaluation of the outer product of a with this multivector d just computed leads to the following C++ code:

```
f[1]=a[1]*d[0];
f[2]=a[2]*d[0];
f[3]=a[3]*d[0];
f[4]=a[1]*d[2]-a[2]*d[1];
f[5]=a[1]*d[3]-a[3]*d[1];
f[6]=a[2]*d[3]-a[3]*d[2];
f[7]=a[1]*d[6]-a[2]*d[5]+a[3]*d[4];
```

This corresponds to the following GAPP code:

```
resetMv f;
//f[1] = a[1] * d[0]
setVector ve8 = {a[1]};
setVector ve9 = {d[0]};
dotVectors f[1] = <ve8,ve9>;

//f[2] = a[2] * d[0]
setVector ve10 = {a[2]};
setVector ve11 = {d[0]};
dotVectors f[2] = <ve10,ve11>;

//f[3] = a[3] * d[0]
setVector ve12 = {a[3]};
setVector ve13 = {d[0]};
dotVectors f[3] = <ve12,ve13>;

//f[4] = (a[1] * d[2]) - (a[2] * d[1])
setVector ve14 = {a[1,-2]};
setVector ve15 = {d[2,1]};
dotVectors f[4] = <ve14,ve15>;

//f[5] = (a[1] * d[3]) - (a[3] * d[1])
setVector ve16 = {a[1,-3]};
setVector ve17 = {d[3,1]};
dotVectors f[5] = <ve16,ve17>;

//f[6] = (a[2] * d[3]) - (a[3] * d[2])
setVector ve18 = {a[2,-3]};
setVector ve19 = {d[3,2]};
dotVectors f[6] = <ve18,ve19>;
```

```
//f[7] = ((a[1] * d[6]) - (a[2] * d[5])) + (a[3] * d[4])
setVector ve20 = {a[1,-2,3]};
setVector ve21 = {d[6,5,4]};
dotVectors f[7] = <ve20,ve21>;
```

The result of this example can be improved further according to Sect. 10.3.7. The Geometric Algebra code without the explicit computation of the variable d,

```
?a=a1*e1+a2*e2+a3*e3;
?b=b1*e1+b2*e2+b3*e3;
?c=a*b;
d=a+c;
?f=a^d;
```

leads to the following optimized algorithm for the multivector f:

```
f[1]=a[1]*c[0];
f[2]=a[2]*c[0];
f[3]=a[3]*c[0];
f[7]=a[1]*c[6]-a[2]*c[5]+a[3]*c[4];
```

The corresponding simplified GAPP code looks as follows:

```
resetMv f;
//f[1] = a[1] * c[0]
setVector ve8 = {a[1]};
setVector ve9 = {c[0]};
dotVectors f[1] = <ve8,ve9>;

//f[2] = a[2] * c[0]
setVector ve10 = {a[2]};
setVector ve11 = {c[0]};
dotVectors f[2] = <ve10,ve11>;

//f[3] = a[3] * c[0]
setVector ve12 = {a[3]};
setVector ve13 = {c[0]};
dotVectors f[3] = <ve12,ve13>;

//f[7] = ((a[1] * c[6]) - (a[2] * c[5])) + (a[3] * c[4])
setVector ve14 = {a[1,-2,3]};
setVector ve15 = {c[6,5,4]};
dotVectors f[7] = <ve14,ve15>;
```

See [105] for details of the implementation of GAPP in Gaalop.

14.3.2 Parallelization Concepts Supported by GAPP

The dotVectors command is the central command of the GAPP language. It supports SIMD operations as explained in Sect. 14.2. Additionally, since the computations of the coeffcients of a multivector are independent of each other, they can be computed

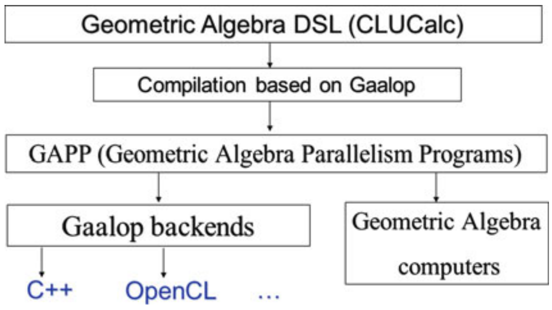

Geometric Algebra DSL (CLUCalc)
Compilation based on Gaalop
GAPP (Geometric Algebra Parallelism Programs)
Gaalop backends
Geometric Algebra computers
C++
OpenCL
...

Fig. 14.5 Geometric Algebra computers based on Gaalop are able to use GAPP as instruction set for Geometric Algebra computations

on different processor cores (see [54]). This has to be done only for the non zero coeffcients.

14.4 Geometric Algebra Computers Based on Gaalop

GAPP has been already used in Gaalop as internal representation for C++ and OpenCL programs. Figure 14.5 shows the process from CLUCalc as a domain specific language (DSL) for Geometric Algebra to the corresponding backends for C++ and OpenCL.

Our vision is that

- Computers in the future will use GAPP directly as instruction set for Geometric Algebra computations;
- Geometric Algebra will become part of standard programming languages such as OpenCL.

Please find always the most up-to-date information and software on http://www.gaalop.de.

References

1. Geomerics ltd. home page available at http://www.geomerics.com.
2. Plücker coordinates. Available at http://en.wikipedia.org.
3. Rafal Ablamowicz. Clifford algebra computations with Maple. In W. E. Baylis, editor, *Clifford (Geometric) Algebras*, pages 463–501. Birkhäuser, 1996.
4. Rafal Abłamowicz and Bertfried Fauser. Clifford/bigebra, a Maple package for Clifford (co)algebra computations. Available at http://www.math.tntech.edu/rafal/, 2011.
5. Eduardo Bayro-Corrochano. Geometric neural computing. *IEEE Transactions on Neural Networks*, 12(5):968–986, 2001.
6. Eduardo Bayro-Corrochano. Robot perception and action using conformal geometry. In Eduardo Bayro-Corrochano, editor, *Handbook of Geometric Computing: Applications in Pattern Recognition, Computer Vision, Neurocomputing and Robotics*, chapter 13, pages 405–458. Springer, 2005.
7. Eduardo Bayro-Corrochano. *Geometric Computing for Wavelet Transforms, Robot Vision, Learning, Control and Action*. Springer, 2010.
8. Eduardo Bayro-Corrochano and Vladimir Banarer. A geometric approach for the theory and applications of 3D projective invariants. *Journal of Mathematical Imaging and Vision*, 16:131–154, 2001.
9. Eduardo Bayro-Corrochano, Kostas Daniilidis, and Gerald Sommer. Motor algebra for 3d kinematics: The case of the hand–eye calibration. *Journal of Mathematical Imaging and Vision*, 13:79–99, 2000.
10. Eduardo Bayro-Corrochano and Gerik Scheuermann, editors. *Geometric Algebra Computing in Engineering and Computer Science*. Springer, 2010.
11. Eduardo Bayro-Corrochano and Garret Sobczyk, editors. *Geometric Algebra with Applications in Science and Engineering*. Birkhäuser, 2001.
12. Eduardo Bayro-Corrochano, Refugio Vallejo, and Nancy Arana-Daniel. Geometric preprocessing, geometric feedforward neural networks and Clifford support vector machines for visual learning. *Neurocomputing*, 67:54–105, 2005.
13. Eduardo Bayro-Corrochano and Julio Zamora-Esquivel. Inverse kinematics, fixation and grasping using conformal geometric algebra. In *IROS 2004, Sendai, Japan*, 2004.
14. Eduardo Bayro-Corrochano and Julio Zamora-Esquivel. Kinematics and differential kinematics of binocular robot heads. In *proceedings of ICRA conference, Orlando, USA*, 2006.
15. John Browne. The Grassmann algebra book home page. Available at http://sites.google.com/site/grassmannalgebra/, 2009.
16. Sven Buchholz, Eckhard M. S. Hitzer, and Kanta Tachibana. Optimal learning rates for Clifford neurons. In *International Conference on Artificial Neural Networks*, volume 1, pages 864–873, Porto, Portugal, 2007. 9–13.

17. Sven Buchholz, Eckhard M. S. Hitzer, and Kanta Tachibana. Coordinate independent update formulas for versor Clifford neurons. In *Proceeding Joint 4th International Conference on Soft Computing and Intelligent Systems (SCIS) and 9th International Symposium on advanced Intelligent Systems (ISIS 2008)*, Nagoya, Japan, 2008.
18. Michael Burger. Das effiziente Raytracen von Dreiecksnetzen auf Mehrkernprozessoren, GPUs und FPGAs mittels geometrischer Algebra. Master's thesis, TU Darmstadt, 2011.
19. Jonathan Cameron and Joan Lasenby. Oriented conformal geometric algebra. *Proceedings of ICCA7*, 2005.
20. Patrick Charrier and Dietmar Hildenbrand. Geometric algebra enhanced precompiler for C++ and OpenCL. In *AGACSE conference La Rochelle*, 2012.
21. William Kingdon Clifford. *Applications of Grassmann's Extensive Algebra*, volume 1 of *American Journal of Mathematics*, pages 350–358. The Johns Hopkins University Press, 1878.
22. William Kingdon Clifford. On the classification of geometric algebras. In Robert Tucker, editor, *Mathematical Papers*, pages 397–401. Macmillan, London, 1882.
23. Alfred Differ. Clados home page. Available at http://sourceforge.net/projects/clados/, 2002.
24. Chris Doran. *Geometric Algebra and its Application to Mathematical Physics*. PhD thesis, Cambridge University, 1994.
25. Chris Doran and Anthony Lasenby. *Geometric Algebra for Physicists*. Cambridge University Press, 2003.
26. Leo Dorst. Honing geometric algebra for its use in the computer sciences. In Gerald Sommer, editor, *Geometric Computing with Clifford Algebra*. Springer, 2001.
27. Leo Dorst, Chris Doran, and Joan Lasenby, editors. *Applications of Geometric Algebra in Computer Science and Engineering*. Birkhäuser, 2002.
28. Leo Dorst and Daniel Fontijne. 3D euclidean geometry through conformal geometric algebra (a GAViewer tutorial). *available at http://www.science.uva.nl/ga*, 2003.
29. Leo Dorst, Daniel Fontijne, and Stephen Mann. *Geometric Algebra for Computer Science, An Object-Oriented Approach to Geometry*. Morgan Kaufmann, 2007.
30. Leo Dorst and Joan Lasenby, editors. *Guide to Geometric Algebra in Practice*. Springer, 2011.
31. Leo Dorst and Stephen Mann. Geometric algebra: A computational framework for geometrical applications (part i: Algebra). *Computer Graphics and Application*, 22(3):24–31, 2002.
32. Julia Ebling. Clifford Fourier transform on vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):469–479, 2005.
33. Ahmad Hosney Awad Eid. *Optimized Automatic Code Generation for Geometric Algebra Based Algorithms with Ray Tracing Application*. PhD thesis, Suez Canal University, Port Said, 2010.
34. Patrick Fleckenstein. C++ template classes for geometric algebras. *Available at http://www.nklein.com/products/geoma*.
35. Daniel Fontijne. *Efficient Implementation of Geometric Algebra*. PhD thesis, University of Amsterdam, 2007.
36. Daniel Fontijne, Tim Bouma, and Leo Dorst. Gaigen: A geometric algebra implementation generator. *Available at http://www.science.uva.nl/ga/gaigen*, 2005.
37. Daniel Fontijne, Tim Bouma, and Leo Dorst. Gaigen 2: A geometric algebra implementation generator. Available at http://staff.science.uva.nl/~fontijne/gaigen2.html, 2007.
38. Daniel Fontijne and Leo Dorst. Performance and elegance of 5 models of geometry in a ray tracing application. *Software and other downloads available at http://www.science.uva.nl/~fontijne/raytracer*, 2002.
39. Daniel Fontijne and Leo Dorst. Modeling 3D euclidean geometry. *IEEE Computer Graphics and Applications*, 23(2):68–78, 2003.
40. S. Franchini, A. Gentile, M. Grimaudo, C.A. Hung, S. Impastato, F. Sorbello, G. Vassallo, and S. Vitabile. A sliced coprocessor for native Clifford algebra operations. In *Euromico Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, 2007.

41. Antonio Gentile, Salvatore Segreto, Filippo Sorbello, Giorgio Vassallo, Salvatore Vitabile, and Vincenzo Vullo. Cliffosor, an innovative FPGA-based architecture for geometric algebra. In *ERSA 2005*, pages 211–217, 2005.
42. Hermann Grassmann. *Die Ausdehnungslehre. Vollstaendig und in strenger Form begruendet*. Verlag von Th. Chr. Fr. Enslin, Berlin, 1862.
43. Hermann Grassmann. Ueber den Ort der Hamilton'schen Quaternionen in der Ausdehnungslehre. In *Mathematische Annalen*, 1877.
44. Klaus Gürlebeck and Wolfgang Sprössig. *Quaternionic and Clifford Calculus for Physicists and Engineers*. Wiley, 1998.
45. Mark Harris. GPGPU home page. Available at http://www.gpgpu.org, 2011.
46. David Hestenes. *Space-Time Algebra (Documents on Modern Physics)*. Gordon and Breach, 1966.
47. David Hestenes. *New Foundations for Classical Mechanics*. Springer, 1999.
48. David Hestenes. Old wine in new bottles: A new algebraic framework for computational geometry. In Eduardo Bayro-Corrochano and Garret Sobczyk, editors, *Geometric Algebra with Applications in Science and Engineering*. Birkhäuser, 2001.
49. David Hestenes. Grassmann's legacy. In H-J. Petsche, A. Lewis, J. Liesen, and S. Russ, editors, *From Past to Future: Grassmann's Work in Context*. Birkhäuser, 2011.
50. David Hestenes. New tools for computational geometry and rejuvenation of screw theory. In Eduardo Bayro-Corrochano and Gerik Scheuermann, editors, *Geometric Algebra Computing in Engineering and Computer Science*, volume 1, pages 3–33. Springer, May 2010.
51. David Hestenes and Ernest D. Fasse. Homogeneous rigid body mechanics with elastic coupling. In Leo Dorst, Chris Doran, and Joan Lasenby, editors, *Applications of Geometric Algebra in Computer Science and Engineering*. Birkhäuser, 2002.
52. David Hestenes and Garret Sobczyk. *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics*. Springer, 1987.
53. Dietmar Hildenbrand. Geometric computing in computer graphics using conformal geometric algebra. *Computers & Graphics*, 29(5):802–810, 2005.
54. Dietmar Hildenbrand. Geometric algebra computers. In *Proceedings of the GraVisMa workshop, Plzen*, 2009.
55. Dietmar Hildenbrand. From Grassmann's vision to geometric algebra computing. In H. J. Petsche, A. Lewis, J. Liesen, and S. Russ, editors, *From Past to Future: Grassmann's Work in Context*. Birkhäuser, 2011.
56. Dietmar Hildenbrand, Eduardo Bayro-Corrochano, and Julio Zamora-Esquivel. Advanced geometric approach for graphics and visual guided robot object manipulation. In *Proceedings of ICRA Conference, Barcelona*, 2005.
57. Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz, and Andreas Koch. Specialized machine instruction set for geometric algebra. In *AGACSE conference La Rochelle*, 2012.
58. Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz, and Joachim Pitt. Gaalop home page. Available at http://www.gaalop.de, 2012.
59. Dietmar Hildenbrand, Daniel Fontijne, Yusheng Wang, Marc Alexa, and Leo Dorst. Competitive runtime performance for inverse kinematics algorithms using conformal geometric algebra. In *Eurographics Conference Vienna*, 2006.
60. Dietmar Hildenbrand, Holger Lange, Florian Stock, and Andreas Koch. Efficient inverse kinematics algorithm based on conformal geometric algebra using reconfigurable hardware. In *GRAPP Conference Madeira*, 2008.
61. Dietmar Hildenbrand, Joachim Pitt, and Andreas Koch. Gaalop – high performance parallel computing based on conformal geometric algebra. In Eduardo Bayro-Corrochano and Gerik Scheuermann, editors, *Geometric Algebra Computing in Engineering and Computer Science*. Springer, May 2010.
62. Dietmar Hildenbrand, Julio Zamora-Esquivel, and Eduardo Bayro-Corrochano. Inverse kinematics computation in computer graphics and robotics using conformal geometric algebra. In *ICCA7, 7th International Conference on Clifford Algebras and Their Applications*, 2005.

63. Eckhard Hitzer. Angles between subspaces. In *proceedings of the GraVisMa workshop, Brno*, 2010.
64. Eckhard Hitzer. New views of crystal symmetry guided by profound admiration of the extraordinary works of Grassmann and Clifford. In H. J. Petsche, A. Lewis, J. Liesen, and S. Russ, editors, *From Past to Future: Grassmann's Work in Context*. Birkhäuser, 2011.
65. Martin Erik Horn. Quaternionen und geometrische Algebra. In *Didaktik der Physik der DPG, Beitraege zur Fruehjahrstagung Kassel 2006*, 2006.
66. Jens Huthmann, Peter Mueller, Florian Stock, Dietmar Hildenbrand, and Andreas Koch. Accelerating high-level engineering computations by automatic compilation of geometric algebra to hardware accelerators. In *proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2010.
67. Thomas Kalbe. Beschreibung der Dynamik elastisch gekoppelter Koerper in konformaler geometrischer Algebra. Master's thesis, TU Darmstadt, 2006.
68. Khronos-Group. OpenCL home page. Available at http://www.khronos.org/opencl/, 2009.
69. Joan Lasenby, Eduardo Bayro-Corrochano, Anthony Lasenby, and Gerald Sommer. A new methodology for computing invariants in computer vision. In *Proceedings of ICPR 96*, 1996.
70. Joan Lasenby, William J. Fitzgerald, Anthony Lasenby, and Chris Doran. New geometric methods for computer vision: An application to structure and motion estimation. *International Journal of Computer Vision*, 3(26):191–213, 1998.
71. Paul C. Leopardi. GluCat home page. Available at http://glucat.sourceforge.net/, 2001.
72. Hongbo Li. *Invariant Algebras and Geometric Reasoning*. World Scientific, 2008.
73. Hongbo Li, David Hestenes, and Alyn Rockwood. Generalized homogeneous coordinates for computational geometry. In G. Sommer, editor, *Geometric Computing with Clifford Algebra*, pages 27–59. Springer, 2001.
74. Pertti Lounesto. The CLICAL home page. Available at http://www.helsinki.fi/~lounesto/CLICAL.htm, 1987. Last visited 15 Sep. 2003.
75. Stephen Mann and Leo Dorst. Geometric algebra: a computational framework for geometrical applications (part ii: Applications). *Computer Graphics and Application*, 22(4):58–67, 2002.
76. Stephen Mann, Leo Dorst, and Tim Bouma. The making of GABLE, a geometric algebra learning environment in Matlab. pages 491–511, 2001.
77. Maxima Development Team. Maxima, a computer algebra system. version 5.18.1. Available at http://maxima.sourceforge.net/, 2009.
78. Biswajit Mishra and Peter R. Wilson. Color edge detection hardware based on geometric algebra. In *European Conference on Visual Media Production (CVMP)*, 2006.
79. Ambjorn Naeve and Alyn Rockwood. Course 53 geometric algebra. In *SIGGRAPH Conference, Los Angeles*, 2001.
80. Christian Perwass. *Applications of Geometric Algebra in Computer Vision*. PhD thesis, Cambridge University, 2000.
81. Christian Perwass. *Geometric Algebra with Applications in Engineering*. Springer, 2009.
82. Christian Perwass. The CLU home page. Available at http://www.clucalc.info,2010.
83. Christian Perwass and Wolfgang Förstner. Uncertain geometry with circles, spheres and conics. In Reinhard Klette, R. Kozera, L. Noakes, and J. Weickert, editors, *Geometric Properties from Incomplete Data*, volume 31 of *Computational Imaging and Vision*, pages 23–41. Springer, 2006.
84. Christian Perwass, Christian Gebken, and Gerald Sommer. Implementation of a Clifford algebra co-processor design on a field programmable gate array. In Rafal Ablamowicz, editor, *Clifford Algebras: Application to Mathematics, Physics, and Engineering*, Progress in Mathematical Physics, pages 561–575. 6th International Conference on Clifford Algebras and Applications, Cookeville, TN., Birkhäuser, 2003.
85. Christian Perwass, Christian Gebken, and Gerald Sommer. Geometry and kinematics with uncertain data. In A. Leonardis, H. Bischof, and A. Pinz, editors, *9th European Conference on Computer Vision, ECCV 2006, May 2006, Graz, Austria*, number 3951 in LNCS, pages 225–237. Springer, Berlin Heidelberg, 2006.

86. Christian Perwass and Dietmar Hildenbrand. Aspects of geometric algebra in euclidean, projective and conformal space. Technical report, University of Kiel, 2004.
87. Christian Perwass and Joan Lasenby. A Geometric Analysis of the Trifocal Tensor. In G. Gimel'farb Klette, Reinhard and R. Kakarala, editors, *Image and Vision Computing New Zealand, IVCNZ'98, Proceedings*, pages 157–162. The University of Auckland, 1998.
88. Christian Perwass and Joan Lasenby. A Unified Description of Multiple View Geometry. In Gerald Sommer, editor, *Geometric Computing with Clifford Algebra*. Springer, 2001.
89. Christian Perwass and Gerald Sommer. The inversion camera model. In *28. Symposium für Mustererkennung, DAGM 2006, Berlin, 12.-14.09.2006*. Springer, Berlin, Heidelberg, 2006.
90. H-J. Petsche, A. Lewis, J. Liesen, and S. Russ, editors. *From Past to Future: Grassmann's Work in Context*. Birkhäuser, 2011.
91. Hans-Joachim Petsche. The Grassmann Bicentennial Conference home page. Available at http://www.uni-potsdam.de/u/philosophie/grassmann/Papers.htm,2009.
92. Minh Tuan Pham, Kanta Tachibana, Eckhard M. S. Hitzer, Tomohiro Yoshikawa, and Takeshi Furuhashi. Classification and clustering of spatial patterns with geometric algebra. In *AGACSE conference Leipzig*, 2008.
93. Wieland Reich and Gerik Scheuermann. *Analyzing Real Vector Fields with Clifford Convolution and Clifford Fourier Transform*, volume 1. Springer, 2010.
94. Leo Reyes-Lozano, Gerard Medioni, and Eduardo Bayro-Corrochano. Registration of 3D points using geometric algebra and tensor voting. *Journal of Computer Vision*, 75(3):351–369, 2007.
95. Alyn Rockwood and Dietmar Hildenbrand. Engineering graphics in geometric algebra. In Eduardo Bayro-Corrochano and Gerik Scheuermann, editors, *Geometric Algebra Computing in Engineering and Computer Science*. Springer, May 2010.
96. Bodo Rosenhahn. *Pose Estimation Revisited*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2003.
97. Bodo Rosenhahn and Gerald Sommer. Pose estimation in conformal geometric algebra. *Journal of Mathematical Imaging and Vision*, 22:27–70, 2005.
98. C.J. Scriba and P. Schreiber. *5000 Jahre Geometrie*. Springer, 2009.
99. Jon Selig and Eduardo Bayro-Corrochano. Rigid body dynamics using Clifford algebra. In *Advances in Applied Clifford Algebras*. Springer, 2008.
100. Florian Seybold. Gaalet – a C++ expression template library for implementing geometric algebra, 2010.
101. Florian Seybold, Patrick Charrier, Dietmar Hildenbrand, M. Bernreuther, and D. Jenz. Runtime performance of a molecular dynamics model using conformal geometric algebra. Slides available at http://www.science.uva.nl/~leo/agacse2010/talks_world/Seybold.pdf, 2010.
102. Gerald Sommer, editor. *Geometric Computing with Clifford Algebra*. Springer, 2001.
103. Gerald Sommer. Applications of geometric algebra in robot vision. In Hongbo Li, Peter J. Olver, and Gerald Sommer, editors, *Computer Algebra and Geometric Algebra with Applications*, volume 3519 of *LNCS*, pages 258–277. 6th International Workshop IWMM 2004, Shanghai, China and International Workshop GIAE 2004, Xian, China, Springer, Berlin Heidelberg, 2005.
104. Gerald Sommer, Bodo Rosenhahn, and Christian Perwass. The twist representation of free-form objects. In Reinhard Klette, R. Kozera, L. Noakes, and J. Weickert, editors, *Geometric Properties from Incomplete Data*, volume 31 of *Computational Imaging and Vision*, pages 3–22. Springer, 2006.
105. Christian Steinmetz. Optimizing a geometric algebra compiler for parallel architectures using a table-based approach. In *Bachelor thesis TU Darmstadt*, 2011.
106. E. Study, G. Scheffers, and F. Engel. *Hermann Grassmann's gesammelte mathematische und physikalische Werke, Die Abhandlungen zur Geometrie und Analysis*. Teubner, 1904.
107. Jaap Suter. Clifford. *Formerly available at http://www.jaapsuter.com*, 2003.
108. SymPy Development Team. Sympy: Python library for symbolic mathematics. Available at http://www.sympy.org, 2010.

109. Deepak Tolani, Ambarish Goswami, and Norman I. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical Models*, 62(5):353–388, 2000.
110. Rich Wareham, Jonathan Cameron, and Joan Lasenby. Applications of conformal geometric algebra in computer vision and graphics. *Lecture Notes in Computer Science*, 3519:329–349, 2005.
111. Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques*. Addison-Wesley, 1992.
112. Florian Woersdoerfer, Florian Stock, Eduardo Bayro-Corrochano, and Dietmar Hildenbrand. Optimization and performance of a robotics grasping algorithm described in geometric algebra. In *Iberoamerican Congress on Pattern Recognition 2009, Guadalajara, Mexico*, 2009.
113. Marius Dorian Zaharia and Leo Dorst. Interface specification and implementation internals of a program module for geometric algebra. *Journal of Logic and Algebraic Programming*, 2003.
114. Marius Dorian Zaharia and Leo Dorst. Modeling and visualization of 3D polygonal mesh surfaces using geometric algebra. *Computers & Graphics*, 29(5):802–810, 2003.
115. Julio Zamora-Esquivel. G6,3 geometric algebra. In *ICCA9, 7th International Conference on Clifford Algebras and their Applications*, 2011.

Index