mathematics

MDPI

Article

Developing GA-FuL: A Generic Wide-Purpose Library for Computing with Geometric Algebra

Ahmad Hosny Eid [1,*] and Francisco G. Montoya [2]

[1] Department of Electrical Engineering, Faculty of Engineering, Port Said University, Port Fouad 42523, Egypt
[2] Department of Engineering, University of Almería, 04120 Almeria, Spain; pagilm@ual.es
* Correspondence: ahmad.eid@eng.psu.edu.eg

Abstract: The Geometric Algebra Fulcrum Library (GA-FuL) version 1.0 is introduced in this paper as a comprehensive computational library for geometric algebra (GA) and Clifford algebra (CA), in addition to other classical algebras. As a sophisticated software system, GA-FuL is useful for practical applications requiring numerical or symbolic prototyping, optimized code generation, and geometric visualization. A comprehensive overview of the GA-FuL design is provided, including its core design intentions, data-driven programming characteristics, and extensible layered design. The library is capable of representing and manipulating sparse multivectors of any dimension, scalar kind, or metric signature, including conformal and projective geometric algebras. Several practical and illustrative use cases of the library are provided to highlight its potential for mathematical, scientific, and engineering applications. The metaprogramming code optimization capabilities of GA-FuL are found to be unique among other software systems. This allows for the automated production of highly efficient code, based on powerful geometric modeling formulations provided by geometric algebra.

Keywords: scientificcomputing; geometric algebra; geometric modeling; metaprogramming; software library; power systems; robotics; inverse kinematics

MSC: 15A67; 68W30; 68-04; 65D18; 70B15

Citation: Eid, A.H.; Montoya, F.G. Developing GA-FuL: A Generic Wide-Purpose Library for Computing with Geometric Algebra. *Mathematics* **2024**, *12*, 2272. https://doi.org/10.3390/math12142272

Academic Editor: Michael Voskoglou

Received: 20 June 2024
Revised: 14 July 2024
Accepted: 18 July 2024
Published: 20 July 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Geometric algebra (GA) is a powerful mathematical framework that unifies a variety of mathematical concepts, including complex numbers, quaternions, octonions, vectors, linear subspaces, orthogonal maps, and tensors, into a single uniform algebraic system. GA is capable of algebraically modeling many interesting types of geometry, including Euclidean, elliptic, hyperbolic, and conformal geometries. Research on practical applications of geometric algebra has expanded greatly in recent years. GA has applications in various fields such as computer graphics, robotics, classical mechanics, modern physics, electromagnetics, power systems, geographical information systems, and computer vision [1–5].

As can be expected, there are many software libraries capable of utilizing GA for practical scientific computing and geometric modeling tasks. There are mainly three categories of GA software libraries: numerical and symbolic computational libraries, source code library generators, and optimizing subroutine generators. Computational GA libraries serve the purpose of performing general numeric or symbolic GA operations. GA library generators are designed to perform code generation for producing GA code libraries for practical use based on some specification of a selected GA. Finally, optimizing subroutine generators are capable of producing properly optimized code from GA multivector expressions and algorithms as a single block of subroutine code.

In this paper, the geometric algebra fulcrum library (GA-FuL) is introduced as a generic, wide-purpose software library for computing with geometric algebra. GA-FuL

is capable of performing all three categories of other software libraries within a single, well-composed system. The high-level layered design of GA-FuL is based on a set of core intentions implemented through the principles of data-oriented programming. The ultimate goal of GA-FuL design is to produce a highly readable, maintainable, and extensible code base to properly serve the intended objectives of the library. In addition, the paper contains a brief exposition of some practical applications of GA-FuL, illustrating some of its capabilities.

GA-FuL is implemented in .NET Framework 8 using C# language. Accessing the library capabilities can be done through any .NET language, including, but not limited to, C#, F#, managed C++, and VisualBasic. The code generation capabilities of GA-FuL are generic, meaning any text-based coding language can be targeted using GA-FuL metaprogramming components.

The paper is organized into seven sections. After the introduction, Section 2 contains a brief mathematical overview of geometric algebra. Section 3 provides an exposition of some GA-based software libraries along with their main characteristics. Section 4 details the high-level design of GA-FuL, including core design intentions, data-oriented programming characteristics, and extensible layered design. Section 5 contains several illustrative and practical use cases of the library that highlight its potential for mathematical, scientific, and engineering applications. Section 6 provides some comparisons of GA-FuL with similar GA software libraries. Finally, the conclusions are provided in Section 7.

2. Overview of Geometric Algebra

In this work, real scalars are denoted using regular small letters, matrices are denoted using regular capital letters, vectors are denoted using bold small letters, and multivectors are denoted using bold capital letters.

Assume a vector space $\mathcal{V}$ spanned by an ordered set of n orthonormal basis vectors $\langle \boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n \rangle$, with the property

$$\boldsymbol{e}_i \cdot \boldsymbol{e}_j = \begin{cases} d_i & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases} \qquad d_i \in \{-1, 0, 1\}$$

A geometric algebra $\mathcal{G}$ can be defined by extending the vector space $\mathcal{V}$ using a bilinear product on vectors called the geometric product, defined using the following:

$$\boldsymbol{e}_i \boldsymbol{e}_j = \begin{cases} \boldsymbol{e}_i \cdot \boldsymbol{e}_i & \text{for } i = j \\ -\boldsymbol{e}_j \boldsymbol{e}_i & \text{for } i \neq j \end{cases} \tag{1}$$

The geometric product $\boldsymbol{e}_i \boldsymbol{e}_i = \boldsymbol{e}_i \cdot \boldsymbol{e}_i = d_i$ maps a basis vector $\boldsymbol{e}_i$ to a real scalar $d_i \in \{-1, 0, 1\}$ called the signature of the basis vector while constructing a new irreducible algebraic entity $\boldsymbol{e}_i \boldsymbol{e}_j = -\boldsymbol{e}_j \boldsymbol{e}_i$ from any two different basis vectors. The new entity $\boldsymbol{e}_{i,j} \equiv \boldsymbol{e}_i \boldsymbol{e}_j$ is called a 2-blade basis in the GA literature. Geometrically, a basis vector $\boldsymbol{e}_i$ could be understood to represent an axis-aligned 1-dimensional subspace (i.e., a coordinate axis) in $\mathcal{V}$. As a direct generalization, a 2-blade basis $\boldsymbol{e}_{i,j}$ could be understood to represent an axis-aligned 2-dimensional subspace (i.e., a coordinate plane) spanned by its two basis vectors $\{\boldsymbol{e}_i, \boldsymbol{e}_j\}$ in $\mathcal{V}$. Additionally, the geometric product can operate on any ordered combination of basis vectors $\boldsymbol{E}_r \equiv \boldsymbol{e}_{i_1,i_2,\ldots,i_k} \equiv \boldsymbol{e}_{i_1} \boldsymbol{e}_{i_2} \cdots \boldsymbol{e}_{i_k}$ with $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ and $r = \sum_{j=1}^{k} 2^{i_j - 1}$ to construct an algebraic representation of a k-dimensional subspace spanned by $\{\boldsymbol{e}_{i_1}, \boldsymbol{e}_{i_2}, \ldots, \boldsymbol{e}_{i_k}\}$. This algebraic element is typically called a k-blade basis or a basis blade of grade k. By extension, a basis vector $\boldsymbol{E}_{2^{i-1}} \equiv \boldsymbol{e}_i$ in this system is called a basis 1-blade, and the unit real scalar $\boldsymbol{E}_0 \equiv 1$ is the basis 0-blade. This kind of algebraic formulation is the foundation of one major expressive feature of GA: the representation and computation of subspaces of any dimension. Just as vectors in classical linear algebra represent 1-dimensional subspaces, GA blades represent k-dimensional subspaces. Moreover, common manipulations of subspaces such as projections and reflections are also applicable through blades, replacing matrices in classical linear algebra.

The number of unique k-blades basis in $\mathcal{G}$ is $\binom{n}{k}$. By taking linear combinations of basis blades of the same grade k, a new $\binom{n}{k}$-dimensional vector space Λ_k is obtained which has elements called k-vectors. This definition results in $\Lambda_0 = \mathbb{R}$ and $\Lambda_1 = \mathcal{V}$. Now, the real scalars and classical vectors are just elements in a more general 2^n-dimensional linear space $\Lambda = \Lambda_0 \bigcup \Lambda_1 \bigcup \Lambda_2 \bigcup \cdots \bigcup \Lambda_n$, called the Grassmann space of multivectors. A multivector $A = \sum_{i=0}^{2^n-1} a_i \boldsymbol{E}_i$ is a linear combination of basis blades of all grades. A multivector $\boldsymbol{A}$ can be represented using one of the following equally useful forms:

$$\begin{aligned} \boldsymbol{A} &= \sum_{i=0}^{2^n-1} a_i \boldsymbol{E}_i && (2) \\ \boldsymbol{A} &= \sum_{k=0}^{n} \langle \boldsymbol{A} \rangle_k && (3) \\ \langle \boldsymbol{A} \rangle_k &= \sum_{j=0}^{\binom{n}{k}-1} a_{\Phi(k,j)} \boldsymbol{E}_{\Phi(k,j)} && (4) \end{aligned}$$

The first form (2) is useful for representing multivectors as a linear list of scalars a_i, ordered by the index i of their basis blades $\boldsymbol{E}_i$ in Λ. For example, in a three-dimensional space, the multivector $\boldsymbol{A} = a_0 + a_1\boldsymbol{e}_1 + a_2\boldsymbol{e}_2 + a_3\boldsymbol{e}_{1,2} + a_4\boldsymbol{e}_4 + a_5\boldsymbol{e}_{1,3} + a_6\boldsymbol{e}_{2,3} + a_7\boldsymbol{e}_{1,2,3}$ can be expressed using a flat linear list of scalars: $[a_0, a_1, \ldots, a_7]$. The second form (3) is useful for expressing a multivector $\boldsymbol{A}$ using its k-vector parts $\langle \boldsymbol{A} \rangle_k$. The linear operator $\langle \rangle_k$, called the "grade extraction operator", extracts the terms of $\boldsymbol{A}$ that exclusively belong to the k-vector space Λ_k. The scalars of multivector $\boldsymbol{A} = \sum_{k=0}^{n} \langle \boldsymbol{A} \rangle_k$ can be grouped by the grades of their corresponding basis blades into a graded list $[[a_0], [a_1, a_2, a_4], [a_3, a_5, a_6], [a_7]]$. The index-mapping function $i = \Phi(k, j)$ in (4) maps a graded index (k, j) into the corresponding linear index i, where k is the grade and j is the index of the basis blade in Λ_k, respectively.

The geometric product of two basis blades $\boldsymbol{E}_i\boldsymbol{E}_j$ is always another basis blade $s\boldsymbol{E}_k$ up to a sign $s \in \{-1, 0, 1\}$. A Cayley table of size $2^n \times 2^n$ for the geometric product of basis blades could be constructed using (1). Using the bilinear property of the geometric product for any two multivectors, their geometric product is another multivector in the same GA:

$$\begin{aligned} \boldsymbol{AB} &= \left(\sum_{i=0}^{2^n-1} a_i \boldsymbol{E}_i \right) \left(\sum_{j=0}^{2^n-1} b_j \boldsymbol{E}_j \right) \\ &= \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^n-1} a_i b_j \boldsymbol{E}_i \boldsymbol{E}_j \\ &= \sum_{k=0}^{2^n-1} c_k \boldsymbol{E}_k \end{aligned} \qquad (5)$$

As a concrete example, assume the four-dimensional Euclidean vector space $\mathcal{V}$ with basis vectors $\langle \boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3, \boldsymbol{e}_4 \rangle$ where $\boldsymbol{e}_i \cdot \boldsymbol{e}_i = 1$ and $\boldsymbol{e}_i \cdot \boldsymbol{e}_j = 0$ for $i \neq j$. In this case, the Grassmann space $\Lambda = \Lambda_0 \bigcup \Lambda_1 \bigcup \Lambda_2 \bigcup \Lambda_3 \bigcup \Lambda_4$ has five k-vector spaces with the following basis k-blades for each:

$$\begin{aligned} \mathbb{R} = \Lambda_0 &: \{\boldsymbol{E}_0\} = \{1\} \\ \mathcal{V} = \Lambda_1 &: \{\boldsymbol{E}_1, \boldsymbol{E}_2, \boldsymbol{E}_4, \boldsymbol{E}_8\} = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3, \boldsymbol{e}_4\} \\ \Lambda_2 &: \{\boldsymbol{E}_3, \boldsymbol{E}_5, \boldsymbol{E}_6, \boldsymbol{E}_9, \boldsymbol{E}_{10}, \boldsymbol{E}_{12}\} = \{\boldsymbol{e}_{1,2}, \boldsymbol{e}_{1,3}, \boldsymbol{e}_{1,4}, \boldsymbol{e}_{2,3}, \boldsymbol{e}_{2,4}, \boldsymbol{e}_{3,4}\} \\ \Lambda_3 &: \{\boldsymbol{E}_7, \boldsymbol{E}_{11}, \boldsymbol{E}_{13}, \boldsymbol{E}_{14}\} = \{\boldsymbol{e}_{1,2,3}, \boldsymbol{e}_{1,2,4,}, \boldsymbol{e}_{1,3,4}, \boldsymbol{e}_{2,3,4}\} \\ \Lambda_4 &: \{\boldsymbol{E}_{15}\} = \{\boldsymbol{e}_{1,2,3,4}\} \end{aligned}$$

The full GA has 16 basis blades and any multivector $\boldsymbol{A} = \sum_{i=0}^{15} a_i \boldsymbol{E}_i = \sum_{k=0}^{4} \langle A \rangle_k$ can be expressed in the form (3) using the index-mapping function $\Phi(k, j)$ shown in Table 1.

The geometric product of any two basis blades is simple to compute using (1). For example, $\boldsymbol{E}_7\boldsymbol{E}_9 = \boldsymbol{e}_{1,2,3}\boldsymbol{e}_{2,3} = \boldsymbol{e}_1\boldsymbol{e}_2\boldsymbol{e}_3\boldsymbol{e}_2\boldsymbol{e}_3 = -\boldsymbol{e}_1\boldsymbol{e}_2\boldsymbol{e}_2\boldsymbol{e}_3\boldsymbol{e}_3 = -\boldsymbol{e}_1$. Relation (3) can be used to find the geometric product of any two multivectors.

Table 1. The index-mapping function $\Phi(k, j)$ for a 4-dimensional GA.

Basis Blade	k (Grade)	j (Index in Λ_k)	$i = \Phi(k, j)$ (Index in Λ)
$\boldsymbol{E}_0 = 1 \in \Lambda_0$	0	0	0
$\boldsymbol{E}_1 = \boldsymbol{e}_1 \in \Lambda_1$	1	0	1
$\boldsymbol{E}_2 = \boldsymbol{e}_2 \in \Lambda_1$	1	1	2
$\boldsymbol{E}_3 = \boldsymbol{e}_{1,2} \in \Lambda_2$	2	0	3
$\boldsymbol{E}_4 = \boldsymbol{e}_3 \in \Lambda_1$	1	2	4
$\boldsymbol{E}_5 = \boldsymbol{e}_{1,3} \in \Lambda_2$	2	1	5
$\boldsymbol{E}_6 = \boldsymbol{e}_{1,4} \in \Lambda_2$	2	2	6
$\boldsymbol{E}_7 = \boldsymbol{e}_{1,2,3} \in \Lambda_3$	3	0	7
$\boldsymbol{E}_8 = \boldsymbol{e}_4 \in \Lambda_1$	1	3	8
$\boldsymbol{E}_9 = \boldsymbol{e}_{2,3} \in \Lambda_2$	2	3	9
$\boldsymbol{E}_{10} = \boldsymbol{e}_{2,4} \in \Lambda_2$	2	4	10
$\boldsymbol{E}_{11} = \boldsymbol{e}_{1,2,4} \in \Lambda_3$	3	1	11
$\boldsymbol{E}_{12} = \boldsymbol{e}_{3,4} \in \Lambda_2$	2	5	12
$\boldsymbol{E}_{13} = \boldsymbol{e}_{1,3,4} \in \Lambda_3$	3	2	13
$\boldsymbol{E}_{14} = \boldsymbol{e}_{2,3,4} \in \Lambda_3$	3	3	14
$\boldsymbol{E}_{15} = \boldsymbol{e}_{1,2,3,4} \in \Lambda_4$	4	0	15

Due to limited space, only a very small sample of geometric representations and operations made possible by the algebraic structure of GA is mentioned here. The interested reader can find much valuable information in the GA literature [6–24]. Table 2 shows some basic multivector operations that are typically useful in practice. Please note that the squared norm is a real number that can be positive, zero, or negative. A multivector with a zero squared norm is called a null multivector.

Table 2. Basic operations on multivectors. $\boldsymbol{I} = \boldsymbol{E}_{2^n-1}$ is the space pseudo-scalar.

Operation	Symbol	Equation
Reverse	$\boldsymbol{A}^{\sim}$	$\sum_{k=0}^{n}(-1)^{k(k-1)/2}\langle\boldsymbol{A}\rangle_k$
Grade Involution	$\hat{\boldsymbol{A}}$	$\sum_{k=0}^{n}(-1)^{k}\langle\boldsymbol{A}\rangle_k$
Squared Norm	$\|\boldsymbol{A}\|^2$	$\langle\boldsymbol{A}\boldsymbol{A}^{\sim}\rangle_0$
Inverse	$\boldsymbol{A}^{-1}$	$\frac{1}{\|\boldsymbol{A}\|^2}\boldsymbol{A}^{\sim}$
Dual	$\boldsymbol{A}^{*}$	$\boldsymbol{A}\boldsymbol{I}^{-1}$
Un-dual	$\boldsymbol{A}^{-*}$	$\boldsymbol{A}\boldsymbol{I}$

Some useful bilinear products can be defined using the geometric product, such as the scalar, outer, and left-contraction products, as described in Table 3.

The outer product $\boldsymbol{A} = \boldsymbol{a}_1 \wedge \boldsymbol{a}_2 \wedge \cdots \wedge \boldsymbol{a}_k$ constructs a k-blade $\boldsymbol{A}$ representing a k-dimensional subspace $\mathcal{A} \subseteq \mathcal{V}$, spanned by a set of linearly independent vectors $\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_k\}$. The same blade, up to a scaling factor, could be constructed using any set of linearly independent vectors spanning $\mathcal{A}$. Thus, a blade in GA only encodes the general spatial attitude and dimension of the subspace it represents but not the details of

any specific set of vectors spanning the subspace. The differentiating scalar factor of blades can hold additional information about the subspace, for example, its chirality (orientation), area\volume\hyper-volume, mass, etc.

Table 3. Geometric algebra products.

Operation	Expression	Equation
Outer Product	$\boldsymbol{A} \wedge \boldsymbol{B}$	$\sum_{k=0}^{n} \sum_{l=0}^{n-k} \langle \langle \boldsymbol{A} \rangle_k \langle \boldsymbol{B} \rangle_l \rangle_{k+l}$
Scalar Product	$\boldsymbol{A} * \boldsymbol{B}$	$\sum_{k=0}^{n} \langle \langle \boldsymbol{A} \rangle_k \langle \boldsymbol{B} \rangle_k \rangle_0$
Left-Contraction Product	$\boldsymbol{A} \rfloor \boldsymbol{B}$	$\sum_{k=0}^{n} \sum_{l=k}^{n} \langle \langle \boldsymbol{A} \rangle_k \langle \boldsymbol{B} \rangle_l \rangle_{l-k}$
Right-Contraction Product	$\boldsymbol{A} \lfloor \boldsymbol{B}$	$\sum_{k=0}^{n} \sum_{l=0}^{k} \langle \langle \boldsymbol{A} \rangle_k \langle \boldsymbol{B} \rangle_l \rangle_{k-l}$
Inner Product	$\boldsymbol{A} \cdot \boldsymbol{B}$	$\sum_{k=0}^{n} \sum_{l=0}^{n} \langle \langle \boldsymbol{A} \rangle_k \langle \boldsymbol{B} \rangle_l \rangle_{\lvert l-k \rvert}$

The scalar product of k-blades $\boldsymbol{A} * \boldsymbol{B}$ is one generalization of the inner product of vectors (which are 1-blades in GA). The scalar product is useful in computations that involve the relative orientations of two subspaces $\mathcal{A}, \mathcal{B}$ of the same dimension in the space. For example, in Euclidean 3D space, the angle between two planes, represented by their two blades $\boldsymbol{A}, \boldsymbol{B}$, is easily computed using the scalar product.

The left-contraction product is among the most useful generalizations of the inner product of vectors. The geometric operation represented by the left-contraction product of two blades $\boldsymbol{A} \rfloor \boldsymbol{B}$ is first to project the subspace $\mathcal{A}$ on $\mathcal{B}$; then, the orthogonal complement of the resulting subspace is taken, both in a single unified operation. In contrast to the outer product, which constructs larger subspaces from smaller ones, the left-contraction product takes a kind of difference of two subspaces. Similar interpretations can be associated with the right-contraction and inner products.

In GA, a nonnull blade can be used to represent a reflection operator or a projection operator, in addition to representing a subspace in the base vector space. Given a blade $\boldsymbol{A}$ of grade a, any other subspace blade $\boldsymbol{B}$ of grade b could be reflected or projected on the subspace that $\boldsymbol{A}$ represents using the following two grade-preserving sandwiching product expressions, respectively:

$$\text{Reflection of } \boldsymbol{B} \text{ on } \boldsymbol{A} \quad = \quad (-1)^{b(a+1)} \boldsymbol{A} \boldsymbol{B} \boldsymbol{A}^{-1} \tag{6}$$

$$\text{Projection of } \boldsymbol{B} \text{ on } \boldsymbol{A} \quad = \quad (-1)^{b(a+1)} \boldsymbol{A} \lfloor \boldsymbol{B} \rfloor \boldsymbol{A}^{-1} \tag{7}$$

Using a single nonnull vector $\boldsymbol{v}$, it is possible to perform a reflection of an arbitrary blade $\boldsymbol{A}$ in the $n-1$ dimensional hyperplane orthogonal to $\boldsymbol{v}$ using the grade-preserving sandwiching product expression $\boldsymbol{v} \hat{\boldsymbol{A}} \boldsymbol{v}^{-1}$. By repeating this operation with several nonnull vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k$ and using the Cartan–Dieudonné theorem [25], a simple and powerful algebraic method for representing all orthogonal transformations on subspaces within GA is attained using the versor product: $(-1)^k \boldsymbol{V} \hat{\boldsymbol{A}} \boldsymbol{V}^{-1}$, where $\boldsymbol{V} = \boldsymbol{v}_1 \boldsymbol{v}_2 \cdots \boldsymbol{v}_k$, the geometric product of nonnull vectors, is called a versor.

When a versor $\boldsymbol{R}$ is of unit squared norm $\|\boldsymbol{R}\|^2 = 1 \Leftrightarrow \boldsymbol{R}^{-1} = \boldsymbol{R}^{\sim}$, it is called a rotor, which has many important applications in GA. More specifically, a rotor $\boldsymbol{R}$ can elegantly rotate any subspace $\boldsymbol{A}$ using a simple versor product $\boldsymbol{R} \boldsymbol{A} \boldsymbol{R}^{\sim}$, which is a natural generalization of unit complex numbers and unit quaternions in 2D and 3D, respectively. Additionally, simple rotors can be constructed in many ways, including the exponentiation of a bivector $\boldsymbol{A}$ using the following:

$$\boldsymbol{R} = \exp(\boldsymbol{A}) \quad = \quad \begin{cases} \cos\alpha + (\frac{1}{\alpha} \sin\alpha) \boldsymbol{A} & \|\boldsymbol{A}\|^2 = \alpha^2 > 0 \\ 1 + \boldsymbol{A} & \|\boldsymbol{A}\|^2 = 0 \\ \cosh\alpha + (\frac{1}{\alpha} \sinh\alpha) \boldsymbol{A} & \|\boldsymbol{A}\|^2 = -\alpha^2 < 0 \end{cases} \tag{8}$$

The direction of the bivector defines the plane of rotation, while the norm defines the amount of rotation, which is the usual Euclidean angle when $\|\boldsymbol{A}\|^2 > 0$.

Another important algebraic tool in GA is the outermorphism [12]. An outermorphism $\bar{\mathbf{f}}$ is a direct extension of a linear map $\mathbf{f}$ on vectors that preserves the outer product. For any multivectors $\boldsymbol{X}, \boldsymbol{Y}$ and scalar a, an outermorphism $\bar{\mathbf{f}}$ satisfies the following:

$$\begin{aligned} \bar{\mathbf{f}}[a\boldsymbol{X}] &= a\bar{\mathbf{f}}[\boldsymbol{X}] \\ \bar{\mathbf{f}}[\boldsymbol{X}+\boldsymbol{Y}] &= \bar{\mathbf{f}}[\boldsymbol{X}]+\bar{\mathbf{f}}[\boldsymbol{Y}] \\ \bar{\mathbf{f}}[\boldsymbol{X}\wedge\boldsymbol{Y}] &= \bar{\mathbf{f}}[\boldsymbol{X}]\wedge\bar{\mathbf{f}}[\boldsymbol{Y}] \end{aligned}$$

Similarly, as $\mathbf{f}$ can be expressed using a matrix on the basis spanning $\mathcal{V} = \Lambda_1$, $\bar{\mathbf{f}}$ can be expressed as a set of n matrices, with one matrix per k-vector basis spanning Λ_k [7,26,27]. Many important geometric operations can be represented as outermorphisms, including linear projections, reflections, and versor products, and then invariantly applied to subspaces represented as blades, without the explicit need to express a subspace using any particular set of basis vectors that span the subspace [7,12].

By varying the signatures of GA's basis vectors, it is possible to represent many nonlinear geometric objects using linear subspaces of the GA. For example, the widely used five-dimensional conformal geometric algebra (CGA) [7] has five basis vectors, $\boldsymbol{e}_-, \boldsymbol{e}_+, \boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3$, with signatures $-1, 1, 1, 1, 1$, respectively. CGA is capable of representing Euclidean spheres, circles, point pairs, points, lines, and planes with blades defined using the five basis vectors. These CGA blades can be used as geometric objects and as geometric operators for reflection and projection. Additionally, CGA versors can be used as orthogonal transformation operators to encode conformal maps, including Euclidean transformations and uniform scaling. Projective geometric algebra (PGA) [9,11] is another important GA with many practical applications and highly interesting theoretical implications for geometric modeling. Vector geometric algebra (VGA) and homogeneous geometric algebra (HGA) [7] are two simpler GAs with the same Euclidean basis vector signature $\boldsymbol{e}_i^2 = 1$ but different geometric interpretation for the basis blades. Additionally, a few other GAs exist that can model conic sections and surfaces in two- and three-dimensional Euclidean spaces along with their intersections and transformations [23,28–33].

3. Overview of Existing GA Software

Following the steadily increasing usage of geometric algebra in research and practical applications in recent years, there is a corresponding rise in the number of GA-related software libraries. Most GA-related software serves the direct purpose of performing GA operations on multivectors with numeric or symbolic scalar components. Few software systems, however, are designed to perform code generation for producing GA code libraries for practical usage. Even fewer systems are capable of producing properly optimized code from GA multivector expressions and algorithms. In this overview section, some of the representatively significant GA software systems are discussed briefly. Illustrative comparisons of general capabilities and limitations of several software systems targeting GA can be found in [34,35].

3.1. Computational Libraries

3.1.1. MATLAB Toolboxes

For MATLAB, the Clifford Multivector Toolbox (https://clifford-multivector-toolbox.sourceforge.io accessed on 15 June 2024) [36] has been under development since 2013 and publicly available since 2015. The toolbox enables a MATLAB user to use matrices and arrays of multivectors for practical computation. The toolbox is currently capable of handling GAs of arbitrary signatures with up to 16 dimensions. The Quaternion Toolbox for MATLAB (QTFM) (https://qtfm.sourceforge.io accessed on 15 June 2024) inspired some ideas used in the toolbox. Because it manages numerous algebras instead of just one fixed algebra, the Clifford algebra toolbox is more sophisticated compared to the

quaternion toolbox. Although only one Clifford algebra can be used at a time by the user, it is possible to dynamically switch between algebras while a script or function is running and to maintain the values of any variables that are already in place. This makes it possible to verify that a computation is working in multiple algebras with a single script. As of toolbox version 2, conformal geometric algebras, which are built on an underlying Clifford algebra to enable the numerical calculation, are also available for computation.

More recently, the Symbolic and User-friendly Geometric Algebra Routines (SUGAR) for Computations in MATLAB (https://github.com/distributed-control-systems/SUGAR accessed on 15 June 2024) has been reported [37]. The SUGAR toolbox has been under development since June 2022 and was publicly released in December 2023. SUGAR is intended to make it easier to translate GA ideas into MATLAB and to offer a range of convenient functions designed specifically for GA calculations, including support for symbolic operations. For high-dimensional GAs, it facilitates computations that are both symbolic and numerical. Designed specifically for engineering and applied mathematics, SUGAR is carefully constructed to model geometric components and transformations in two- and three-dimensional projective and conformal geometric algebras, compliant with accepted computational techniques in the literature. Moreover, SUGAR manages multivector functions such as exponential, logarithmic, sinusoidal, and cosine functions with efficiency, which makes it more applicable to a wider range of technical fields such as power electronics, robotics, and control systems. Several examples for using SUGAR are provided in [37], ranging from standard robotics to iconic rigid body dynamics simulations, in addition to new applications such as the use of GA for the modeling and analysis of power electronics.

3.1.2. C++ Template Metaprogramming Libraries

There are several GA-related code libraries targeting C++ for numeric and symbolic computations. In C++, the use of template metaprogramming techniques [38] provides useful compile-time code optimization capabilities. Typically, GA-related template-based C++ implementations are capable of efficiently handling several kinds of scalars including numerical and symbolic. GA-related C++-template-based libraries include GluCat (https://sourceforge.net/projects/glucat/files accessed on 15 June 2024), Versor (https://github.com/wolftype/versor accessed on 15 June 2024) [39], Gaalet (https://sourceforge.net/projects/gaalet accessed on 15 June 2024), GATL (https://github.com/laffernandes/gatl accessed on 15 June 2024) [35], and TbGAL (https://github.com/Prograf-UFF/TbGAL accessed on 15 June 2024) [34]. Additionally, some libraries, such as Gaalet and GATL, apply a form of on-demand computation, called lazy evaluation [40], to increase the time performance efficiency of the code.

One notable GA library is the Tensor-based Geometric Algebra Library (TbGAL) [34], a high-level C++ package supporting GA computations. Even in large-dimensional non-degenerate GA spaces (with $n > 256$), the library achieves great performance by handling blades and versors decomposed as vectors under a tensor structure [41] (Chapter 5). Furthermore, the implementation is ready to be used as a back-end to a Python environment or as a C++ pure library, maintaining the ease of use of the library. This flexibility makes it simple to adjust in accordance with the user's experience without compromising performance. General multivectors cannot be handled by TbGAL. However, in practice, it is not a limitation when one is working with GA, in contrast with Clifford algebra. Visualization capabilities are not provided by the library, but we plan to call an external visualization library, Ganja.js, for this purpose.

3.1.3. Julia and Python Libraries

Leibniz–Grassmann–Clifford–Hestenes geometric algebra, an extended tensor algebra, is used in the Grassmann.jl [42] Julia package to perform computations based on multi-linear algebra, differential geometry, and spin groups. Using staged caching and precompilation, the kernelized operations are constructed using composite sparse tensor

products and Hodge duality, providing high dimensional support for up to 62 indices. Concise yet extremely expandable definitions are made possible by code generation.

Libraries for Python include the numerical GA library Clifford (https://github.com/pygae/clifford accessed on 15 June 2024) and the symbolic GA GAlgebra (https://github.com/pygae/galgebra accessed on 15 June 2024). In addition to providing the common set of GA operations on multivectors, GAlgebra provides many operators for geometric calculus [12,14,21,22] such as geometric derivatives, sub-manifolds, linear transformations, and differential operators.

3.2. Library Generators

This category of GA software systems automatically generates the source code of a complete library based on a selected representation of multivectors, in addition to subroutines to perform basic GA operations on multivectors. The first ever library generator that inspired many of the following systems was the Geometric Algebra Implementation Generator (Gaigen) (https://sourceforge.net/projects/g25 accessed on 15 June 2024) [43]. The last version of Gaigen was capable of generating library code based on GA specifications for C, C++, C# and Java. Gaigen takes textual algebra requirements as input and creates efficient GA implementations. Gaigen also converts functions over such algebras into low-level coordinate-based code from functions expressed in a high-level coordinate-free domain-specific language (DSL) [44]. This code can be released via a customized back-end in any target language. In order to achieve high performance, Gaigen needs to modify the generated code to fit the application that links to it in runtime. A feedback loop based on profiling is used for this purpose. The created code establishes a link with the code generator while it is executing. Information regarding functions that need to be optimized is sent by the resulting code. After registering these data, the code generator returns new type of data. The code is created again using the recorded profile when the program ends.

More recently, a C++ library generator called Garamon (https://github.com/vincentnozick/garamon accessed on 15 June 2024) [45] has been developed to synthesize effective Python and C++ libraries that implement GAs in both low and high dimensions using any given metric. Utilizing the prefix tree formulation, Garamon is able to implement GA operations on high dimensions that were difficult to achieve with previous software implementations. Garamon is intended to generate libraries that are simple to use, straightforward to install, efficient, and numerically stable. Precomputed functions for low dimensions and a smooth transition to the new recursive products for higher dimensions form the foundation of the library's design. In Garamon, multivector basis blade terms are organized internally by grade. A collection of arrays, each devoted to a particular grade, is the selected representation of a multivector. Only arrays that are directly related to grades and expressed by the represented multivector are included in this set; however, an array may still contain some zero values. The majority of real-world GA entities are homogeneous multivectors, which drives this decision. In this circumstance, an array dedicated to a specific grade of an arbitrary object is likely to be full and so far more effective than alternative representations.

One other highly practical GA code generator for JavaScript, C++, C#, Rust, and Python is called Ganja.js (https://github.com/enkimute/ganja.js accessed on 15 June 2024). Ganja.js generates Clifford algebras and sub-algebras of any signature, in addition to implementing algebraic constants and operator overloading. The library internally uses a highly memory-efficient sparse per-grade representation of multivectors. The extensive set of highly informative visual samples presented online (https://enkimute.github.io/ganja.js/examples/coffeeshop.html accessed on 15 June 2024) illustrates GAs with up to 16 dimensions. Visualization capabilities of the library are extensive, including the possibility of creating animated and interactive 2D and 3D graphics, in addition to the powerful capability of visualizing arbitrary blades using ray-marching techniques [46]. Ganja.js can

also be directly used as a computational library for several GAs including 2D\3D VGA, CGA, PGA, and others.

3.3. Optimizing Subroutine Generators

This category of GA software systems can generate optimized source code from a single subroutine described as a series of GA operations on multivectors. Until the development of GA-FuL, only two systems fell into this category: GAALOP and GMac. The **G**eometric Algebra ALgorithms OPtimizer (GAALOP) (https://www.gaalop.de accessed on 15 June 2024) is a software library for compiling and optimizing GA expressions into high-level programming language code. Geometric algebra expressions can be developed using the freely available CLUCalc software [23]. GAALOP optimizes CLUCalc expressions and produces C++ (AMP), OpenCL, CUDA, CLUCalc, or LaTeX output. The optimized code is free of geometric algebra operations and runs very efficiently on various platforms. The GAALOP Precompiler (GPC) is a more development-focused variant of GAALOP that integrates GAALOP directly into CMake-generated C/C++ toolchains. An online interface for GAALOP, called GAALOPWeb, is also available that allows for quickly testing algorithms including a visualization without prior installation.

GMac (https://github.com/ga-explorer/GMac accessed on 15 June 2024) [47], short for Geometric Macro, was initially developed in 2008 based on some of the design ideas in Gaigen, with the intention being to replace the awkward runtime profiling stage that Gaigen requires for efficient code generation. This was accomplished through the use of an optimizing compiler component aided by the extensive symbolic simplification capabilities of Wolfram Mathematica [48]. The user can describe a geometric procedure formulated using GA operations on multivectors expressed textually through a custom domain-specific language called GMacDSL. The user then selects which scalar components of the input and output multivectors are important to the problem. This is similar to how Gaigen works at the initial stage. The GMac optimizing compiler automatically follows scalar component computations of intermediate and output multivectors and performs basic expression simplification and optimizations to produce the final code for a selected programming language. In later developments, GMac was extended to handle GAs of any metric signature [26] and efficient outermorphism computation [27]. GMac was successfully used in a few applications such as in [47,49,50], where the performance of the generated code was significantly better than that of classical manual coding methods. The GMac project was archived in August 2021 in favor of the newly developed Geometric Algebra Fulcrum Library.

4. The Geometric Algebra Fulcrum Library (GA-FuL)

GA-FuL (https://github.com/ga-explorer/GeometricAlgebraFulcrumLib accessed on 15 June 2024) is an extensive .NET framework software library written in the C# programming language [51], capable of performing computational tasks of all three categories of GA software systems. The name *"Geometric Algebra Fulcrum"* signifies two aspects of the system: mathematical and computational. For the mathematical aspect, geometric algebra is a unifying algebraic language that incorporates and generalizes many seemingly diverse algebraic tools under the same framework of mathematical operations [12]. Such algebraic tools include, to name a few, real numbers, complex numbers, quaternions, octonions, and orthogonal matrices. From the experience of many researchers who have worked with GA, its most important feature is, however, its ability to unify the geometric reasoning process among many seemingly diverse fields of application domains [52]. This effectively allows for easier cross-discipline communication and idea exchange on a fundamentally algebraic level. Thus, GA acts as a *fulcrum for geometric reasoning* across diverse scientific and engineering domains.

For the computational aspect, there is a pressing need for software tools and code libraries that act as pivot points, i.e., fulcrums, for developing several classes of computational systems based on multivectors and GA operations. Such computational systems

include numerical prototyping, symbolic mathematical investigations, and optimized code generation, among others. Writing and maintaining a separate code base for each class of systems is impractical. GA-FuL is intended to play the role of a *computational fulcrum* for prototyping, geometric modeling, investigating relevant algorithms, and developing computational software libraries based on the *geometric reasoning fulcrum* of geometric algebra. Additionally, GA-FuL provides a complete application programming interface (API) for handling a wide spectrum of application domains.

4.1. Core Design Intentions

The high-level design of GA-FuL targets a specific set of core design intentions (CDIs). The set of CDIs is a direct result of the experience gained during the development of the predecessor system, GMac. An overview of the CDIs is as follows:

- **CDI-1: Abstracting multivector operations from concrete scalar representations.** In practical computational applications, there are diversely useful representations for the mathematical concept of scalars, as classically introduced in abstract algebra [53]. The most common representation for numerical applications is for real scalars using the IEEE 754 floating-point number format [54]. For modern data-driven and machine learning applications [55], multi-dimensional arrays and tensors are the basic representations for scalar data [56]. Another very useful representation for scalars, used in most Computer Algebra Systems (CAS), is an expression tree which represents a symbolic scalar expression for mathematical manipulation applications [57]. The foundational requirement for GA-FuL design is to provide a unified generic implementation for storing and manipulating multivectors and performing common GA operations on any kind of useful scalar representation.
- **CDI-2: Reducing memory requirements for sparse multivectors.** One major obstacle in the way of using GA for practical computations is the memory storage requirements imposed by the structure of multivectors. Storing a full multivector in a n-dimensional GA requires 2^n scalars, while storing a full k-vector requires $\binom{n}{k}$ scalars. For the simplest of numerical scalar representations, 32-bit floating point numbers, a single multivector in a 30-dimensional GA requires 8 GBytes of memory, while a full 15-vector one requires nearly 1.16 GBytes of memory, assuming the use of memory arrays to store the scalars. For most practical GA applications, however, there is rarely a need for storing full multivectors or full k-vectors, and only sparse multivectors are sufficient. One feature of GA is to enable the creation of linear models of nonlinear geometric objects by embedding the original space into a higher-dimension GA space. This typically results in multivectors representing geometric objects using a significantly reduced number of scalars, not a full-sized multivector or k-vector. For example, a sparse multivector in five-dimensional conformal GA [24], containing only 5 out of 32 scalars, can represent points, planes, and spheres in three-dimensional Euclidean space. As such, another important requirement for the GA-FuL design is to provide a set of generic, memory-efficient data structures for storing the scalars of sparse multivectors in high-dimensional GAs.
- **CDI-3: Providing metaprogramming capabilities.** Generative programming [58,59] in general and metaprogramming [60,61] in particular incorporate the process of creating software systems that treat programs as data, enabling the transformation of existing programs or for the generation of new ones. This was the target of the predecessor system GMac for generating optimized computational numerical code from a series of GA expressions [47,49]. This design goal is carried on to GA-FuL, which would enable code generation targeting high-performance platforms such as CUDA, in addition to classical general purpose programming languages such as C\C++, C#, Java, JavaScript, Python, MATLAB scripts, etc.
- **CDI-4: Introducing a layered system design for a wide spectrum of uses.** The complexity imposed by the previous CDIs must be organized and managed through a layered design of the system. Each layer should specialize in one aspect of the system

such as storage management, processing, algebraic and geometric abstractions, etc. In addition, a system of such capabilities would have a wide range of users and use cases. Typically, a user would use the system to create a numerical\symbolic prototype for some geometric modeling ideas, and then after some experimentation, would use metaprogramming system capabilities to generate optimized code for the final model targeting a specific programming language or environment. The design of GA-FuL attempts to realize this layered approach to allow users of different backgrounds to select the suitable level of coding they can handle. The coding level ranges from the very high level of coordinate-independent prototyping using abstract GA operations up to a fully controlled low-level direct manipulation of scalars and coordinates for high-performance computing purposes.

- **CDI-5: Providing a unified, generic, and extensible API for several classes of applications.** The final design goal of GA-FuL is to expose the system functionality through a good API. The API should have a unified public interface with uniform conventions to aid usability. Additionally, the API should be generic regarding the kinds of scalars and GAs it can handle, reflecting the capabilities of the underlying system. API extensibility is also important for future development of the system to aid in the addition of more features and widening system usage. Finally, the API should support the development of various classes of applications including, but not limited to, numerical prototyping computations, symbolic mathematical manipulations, signal processing, visualization, and metaprogramming.

4.2. Design Overview

During the initial design of GA-FuL, satisfying the set of CDIs using traditional object-oriented programming (OOP) was found to be non-practical. This is mainly due to the tendency of classical OOP practices to increase the code-base complexity of large systems such as GA-FuL. In this context, complexity specifically means the deep coupling of data and behavior code typically imposed by classical OOP principles, especially encapsulation and inheritance. This typically results in complicated relations between system classes and complex inheritance hierarchies, which lead to difficulties in understanding the design of large systems. If not properly mitigated, this can eventually result in reduced code understanding and difficulty in system maintenance and extensibility.

The solution found to be most useful was to use a newly emerging software design paradigm that while being compatible with OOP, also tends to produce a more readable, maintainable, and extensible code-base. The use of data-oriented programming (DOP) principles [62], as the highest-level design paradigm, proved to be highly beneficial to many aspects of GA-FuL system design. The four core principles of DOP as expressed in [62] are follow:

- **DOP-1: Separating behavior code from data.** This is a design tenet that advocates for a distinct division between behavior code and data. Following this DOP principle in OOP entails grouping the behavior code into methods for a static class.
- **DOP-2: Representing data with generic data structures.** DOP is not dogmatic about the programming constructs used to employ and organize the code. Arrays\lists and dictionaries\maps are the two most widely used generic data structures in practice. However, one can also utilize other general data structures, such queues, trees, and sets.
- **DOP-3: Making data immutable.** In DOP, due to isolation of representational data structures from behavior code, data mutation is not permitted. Instead, data modifications are carried out by generating new data structure versions. A variable's reference can be updated to point to a different version of the data, but the actual value of the data must never change.
- **DOP-4: Separating data representation from data schema.** Now that data and code are decoupled and generic immutable data structures are employed to describe it, the challenge is to articulate the shape of the data. The intended shape in DOP is

represented by a data schema that is stored apart from the actual data. DOP-4's primary advantage is that it gives developers the freedom to choose which data elements should have a schema and which ones should not.

In GA-FuL, DOP-1 is implemented using thin wrapper classes around generic data structures holding the actual data. Extension methods [51] in static utility classes operate on the thin-wrapper classes to perform the desired behaviors. As for DOP-2, sparse algebraic objects, such as k-vectors and multivectors, are stored in dictionaries, while dense algebraic objects, such as matrices and multidimensional scalar arrays, are stored in classical array data structures. DOP-3 is accomplished through specialized classes called composers. A composer for a multivector, for example, performs a data transformation\construction transaction that, when completed properly, generates a valid dictionary containing valid data values that a multivector code wrapper class and extension methods can query and manipulate later. Finally, the DOP-4 principle is accomplished through the use of generic interfaces and abstract base classes, where the wrapper classes and extension methods manipulate data with a given generic interface or abstract class regardless of the actual data structure implementing the interface\class at any moment during program execution.

As a specific example of how the DOP principles in GA-FuL are implemented, the interface `IIndexSet` is used as a data schema to represent all kinds of index sets for basis blades (according to DOP-4). For representing a GA basis blade $\boldsymbol{e}_{i_1,i_2,\ldots,i_k}$, concrete class implementations of this interface internally use a sorted set of non-negative integers $\langle i_1, i_2, \ldots, i_k \rangle$, completely independent of any specific GA metric. There are specialized immutable classes implementing the `IIndexSet` interface for the empty index set; a single-element index set, a more efficient index set with largest index less than 64 (internally using a 64-bit unsigned integer); a dense index set of arbitrary size (using an array of integers); and a sparse index set of arbitrary size (internally using a hash-set object for storing the indices) (according to DOP-2,3). The class `XGaBasisBlade` is a thin wrapper around an `IIndexSet` object with member and extension methods for performing basic operations on basis blades such as the geometric and other bilinear products, the reverse operation, etc. (in accordance with DOP-1).

Another example is the generic interface `IReadOnlyDictionary<IIndexSet, T>` that is the main data schema (DOP-4) for storing a sparse list of (basis blade, scalar value) pairs for all kinds of multivectors in GA-FuL. There is a specialized immutable class (DOP-2,3) implementing this interface for zero multivectors, another for storing a single (basis blade, scalar value) pair, and one for an arbitrary sparse list of (basis blade, scalar value) pairs. The internal data of a new multivector can be constructed using the `XGaMultivectorComposer<T>` composer class (DOP-3) acting as a construction transaction management class (DOP-1). The composer class automatically selects the most efficient concrete data structure class implementing the `IReadOnlyDictionary<IIndexSet, T>` interface to be used as internal storage for the constructed multivector.

4.3. GA-FuL Layers

CDI-4 is a core design intention that manages the complexity of GA-FuL components. Figure 1 illustrates the layered design of GA-FuL currently consisting of four layers, each having a set of specific roles and sub-layers.

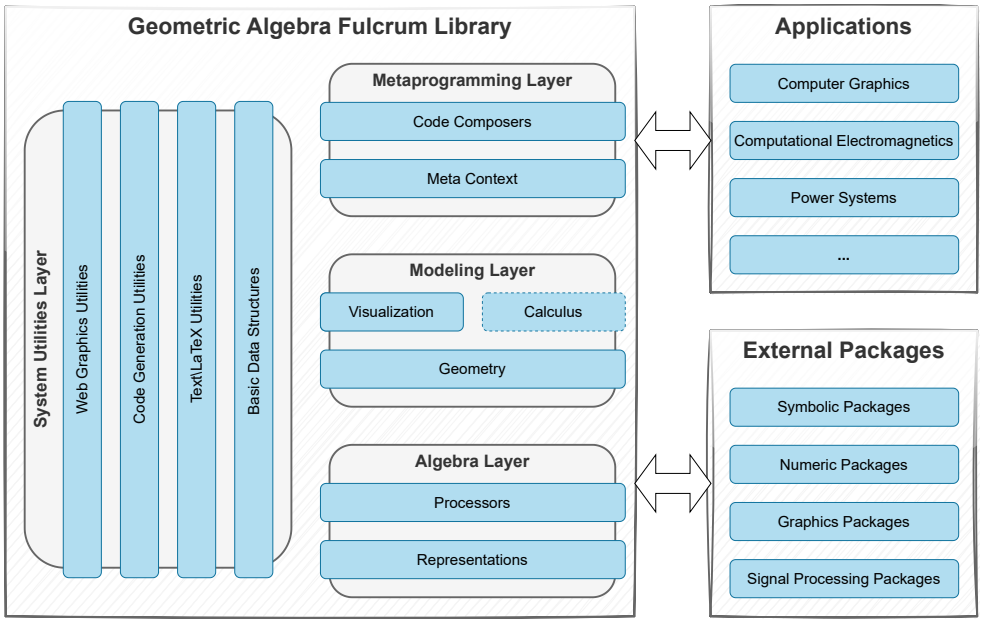

Figure 1. Main layers of the GA-FuL design.

4.3.1. Algebra Layer

At the lowest level, the algebra layer is designed specifically to fulfill CDI-1 and CDI-2, in addition to the four DOP principles. Other layers in GA-FuL eventually utilize the functionalities provided by this layer. Components in the algebra layer mainly perform two functions:

1. DOP-adhering representations for generic scalars, basis blades, multivectors, linear maps, etc.
2. DOP-adhering processing tasks on the representations.

Real scalar representations are considered external to this layer. A scalar can be represented using any desired class or structure, including numeric and symbolic representations provided by external packages. The generic `IScalarProcessor<T>` interface represents a processor to perform basic operations on scalars of arbitrary type `T`. This is one form of DOP-3 adherence in GA-FuL design where a scalar processor transforms scalar data to fulfill desired operations. Such operations include, among others, basic arithmetic (negation, addition, subtraction, multiplication, division, and power), transcendental functions (trigonometric, exponential, logarithms, etc.), and zero equality testing. The derived interface `INumericScalarProcessor<T>` is useful for implementing concrete scalar processors on numerical types. In the current implementation, there are scalar processors for standard single\double precision floating-point real and complex numbers, arbitrary precision decimal\floating-point scalars, and arbitrary-precision rational numbers. In addition, there is a class implementing these operations on NumPy-like multi-dimensional arrays, and another for sampled signals for computational data-driven and signal processing applications. A second derived interface, `ISymbolicScalarProcessor<T>`, is useful for handling symbolic scalars typically used in a CAS. This includes a class capable of processing Wolfram Mathematica symbolic scalars represented by the provided `Expr` class. New implementations can be added at later time to augment GA-FuL with the ability to interact with other symbolic processing systems such as Maple, the MATLAB symbolic toolbox, Python's SymPy package, etc. There is also a generic thin-wrapper class `Scalar<T>` composed over a scalar processor of type `IScalarProcessor<T>` and a scalar value of type `T` . This class is meant to make the GA-FuL API easier to use. Using this class, instead of the complicated scalar processor call `w = scalarProcessor.Add(x, scalarProcessor.Times(y,z))`, the user can simply write `w = x + y * z`. A similar DOP-adhering design is used for storing and manipulating

most mathematical object representations in GA-FuL, including multivectors, the core GA mathematical object.

For representing GA's multivector basis blades $e_{i_1,i_2,\ldots,i_k}$, this layer internally uses a sorted index set $\langle i_1, i_2, \ldots, i_k \rangle$, completely independent of any specific GA metric. As illustrated in the previous section, the interface `IIndexSet` is used to represent such index sets. Basic operations on individual and pairs of basis blades, such as the reverse operation or geometric products, for example, are performed at the lowest level through specialized integer manipulation subroutines. In the current implementation, blades with arbitrary dimensions can be represented using dynamic list-based index sets, while basis blades with dimensions less than 64 can be represented using fixed-length 64-bit integers, where a 1 indicates the presence of a basis vector in the index set of the basis blade, and a 0 indicates its absence. Additionally, GAs with 12 dimensions or less use various lookup tables to accelerate operations on lower-dimensional basis, blades. This structure enables more efficient processing of low-dimensional basis blades while allowing for the handling of arbitrary high-dimensional ones if the application requires.

On the processing side, the class `XGaMetric` is used for basic processing of basis blades with a specified metric signature such as Euclidean, projective, conformal, etc. The signature is specified using two numbers q, r, the number of basis vectors that square to $-1, 0$ respectively. All remaining vectors in a basis blade are assumed to square to 1. In this way, no fixed dimension is predefined for any particular metric computation on basis blades. As in the case of scalars, the thin-wrapper class `XGaSignedBasisBlade` is composed over a `IIndexSet` member, a `XGaMetric` member, and an integer sign member that can only take values $-1, 0, 1$. In this way, operations on basis blades can be easily performed using simple member and static extension methods on the `XGaSignedBasisBlade` class, instead of more complicated calls to methods of an `XGaMetric` object.

The data of a k-vector are stored in an immutable dictionary of (index set, generic scalar) key-value pairs of type `IReadOnlyDictionary<IIndexSet, T>;` with keys of type `IIndexSet` and scalar of generic type `T`. The number of indices per index set for all keys in the dictionary is constant and equal to k, the grade of the k-vector. The data of a multivector are stored in an immutable dictionary containing (grade k, k-vector) key-value pairs of type `IReadOnlyDictionary<int, XGaKVector<T>>;`, where a key holds a unique grade k, and the value is a k-vector part of the multivector. In this way, all linear and bilinear operations on multivectors are reduced to operations on k-vectors, which greatly simplifies the implementation. Additionally, this design enables a highly sparse and flexible representation of multivectors of all kinds in GA-FuL.

The generic `XGaProcessor<T>` class, derived from `XGaMetric`, is the root for all multivector processors in this layer. Most operations on multivectors, some of which are described in Section 2, are implemented using static extension methods taking a `XGaProcessor<T>` object as the main argument. The current version of GA-FuL allows for the representation and manipulation of GA spaces with any number of dimensions. All GA metrics are also possible based on the previous work detailed in [26]. In addition, there are specialized processor classes for Euclidean, conformal [7], and projective [8,9,11] GAs. Additionally, a small hierarchy of thin-wrapper classes, shown in Figure 2, is implemented to simplify the GA-FuL API, as in the case for scalars and basis blades. This scheme allows for the memory-efficient storage of both dense low-dimensional and sparse high-dimensional multivectors.

One downside of this generic scheme is the computational performance for some applications. For this reason, there is a similar class hierarchy, rooted in the `RGaFloat64Multivector` class, optimized specifically for sparse multivectors of standard floating point scalars and GA spaces with fewer than 64 dimensions. For even higher-performance applications, the use of code generation is possible using the metaprogramming layer in GA-FuL described below. This flexible design gives the user a wide set of implementation options for various application domains within a single software framework. Up to the best of the authors' knowledge, no other single GA library provides a similar set of balanced choices simultaneously.

Additional classes for commonly useful GA transformations are also implemented. These include classes for general outermorphisms, general orthogonal operators (using versors in GA), general rotations (using GA rotors), and reflections\projections (using GA subspaces as reflection\projection operators), as illustrated in Figure 3.

In addition to real scalar algebra and geometric algebra, there are other kinds of algebraic representations implemented in the GA-FuL algebra layer. These include algebraic representations for complex numbers, quaternions, polynomials, linear algebra objects (planar angles, classical 2D\3D\4D\nD vectors, matrices, and general linear maps), and sampled signals for signal processing applications.

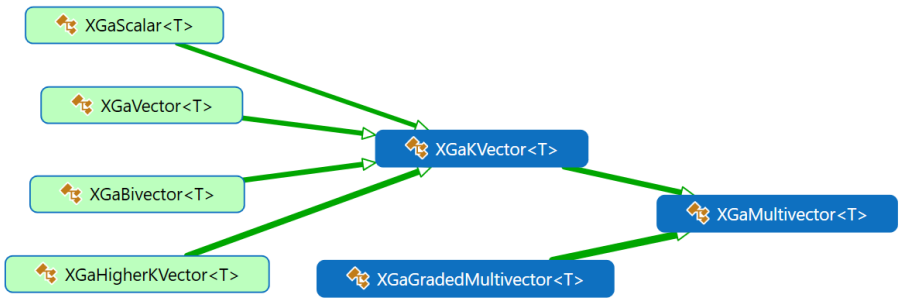

Figure 2. GA-FuL algebra layer generic multivector thin-wrapper classes.

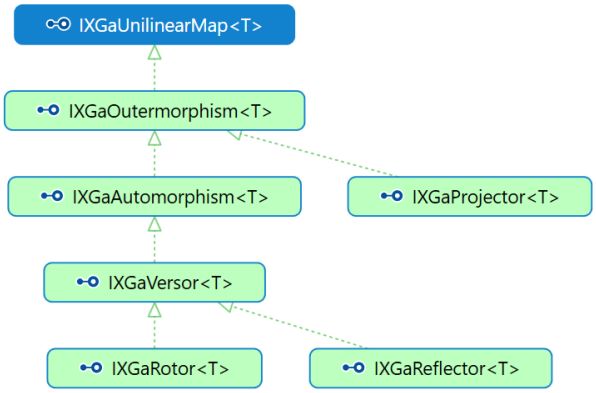

Figure 3. GA-FuL modeling layer interfaces for linear maps on generic multivectors.

4.3.2. Modeling Layer

The modeling layer mainly targets the fulfillment of CDI-5. in this layer, there are mostly thin wrappers around classes from the algebra layer, with specific member and extension methods suitable for the intended functionality of each class. The calculus sub-layer, still in the design stage, is intended to perform geometric calculus operations on multivectors as described in the GA literature [12,14,21,22,52]. The visualization sub-layer is intended to visualize geometric objects using suitable 2D\3D computer graphics methods. Currently, it is possible to generate JavaScript code for the 3D visualization and animation engine Babylon.js (https://www.babylonjs.com accessed on 15 June 2024) based on static and animated geometric objects from the geometry layer. Additionally, more sophisticated videos can be generated by combining individual image frames using the Selenium browser automation project Chrome WebDrive for.NET (https://www.nuget.org/packages/Selenium.WebDriver.ChromeDriver accessed on 15 June 2024). Some illustrative examples are included online under the GA-FuL code repository (https://github.com/ga-explorer/GeometricAlgebraFulcrumLib/tree/main/GeometricAlgebraFulcrumLib.Visualizations accessed on 15 June 2024).

The geometry sub-layer contains the highest level of specialized classes for particular geometries utilizing GA, such as Euclidean, projective, and conformal GAs. For example,

the conformal GA for describing 3D geometric objects can be used, as in the following code example:

```
// The predefined scalar processor for 64-bits
// floating point numbers
var scalarProcessor = ScalarProcessorOfFloat64.Instance;

Create the CGA space object based on the selected
kind of scalars
var cga = XGaConformalSpace5D<double>.Create(scalarProcessor);

// Encode 4 points as CGA null vectors
var a = cga.EncodeIpnsRoundPoint(3.5, 4.3, 2.6);
var b = cga.EncodeIpnsRoundPoint(-2.1, 3.4, 5);
var c = cga.EncodeIpnsRoundPoint(7.4, -1.5, -4.5);
var d = cga.EncodeIpnsRoundPoint(3, -2, 5);

// Use the outer product to define the OPNS blade, encoding a
sphere passing through points a, b, and c
var sphere = a.Op(b).Op(c).Op(d);

// Encode a line passing through a point parallel to
a direction vector
var line = cga.EncodeOpnsFlatLine(
scalarProcessor.Vector3D(3.5, 4.3, 2.6),
scalarProcessor.Vector3D(1, 1, 1)
);

// Project line on sphere to get a circle
var circle = line.ProjectOpnsOn(sphere);

// Decode the circle to separate its individual Euclidean
// geometric components
var circleComponents = circle.DecodeOpnsRound();

// Center, radius, direction bivector, and normal of circle:
var center = circleComponents.CenterToVector3D();
var radius = circleComponents.RealRadius;
var bivector = circleComponents.DirectionToBivector3D();
var normal = circleComponents.NormalDirectionToVector3D();
```

As illustrated in the code, after defining a 5D-CGA space using an instance of the class `XGaConformalSpace5D<T>`, the user can perform the following tasks:

- Encode a geometric object as a CGA blade\multivector. For example, the 5D CGA blades can represent the direction vectors, bivectors, points, point pairs, circles, spheres, lines, and planes of 3D Euclidean space. Additionally, CGA versors can encode all Euclidean and conformal maps such as rotations, translations, inversions, and reflections.
- Perform basic GA multivector algebraic operations on the encoded multivectors in the CGA space.
- Use simple member and extension methods to perform high-level geometric operations on the encoded multivectors. Examples include reflections, intersections, projections, translations, and rotations.
- Decode a CGA blade\multivector into a set of simpler components. For example, a 5D CGA blade representing a circle can be decoded into the circle's center, radius, direction bivector, and normal vector.

In the current implementation of GA-FuL, the base class `XGaConformalSpace<T>` and its two derived classes `XGaConformalSpace4D<T>` , `XGaConformalSpace5D<T>` are capable of

handling not only CGA of any dimension but also PGA through the implementation of the powerful exposition in [63]. The main advantage of handling CGA and PGA within the same algebraic space is the ability to freely mix geometric object representations and their interactions within a single API.

4.3.3. Metaprogramming Layer

The metaprogramming layer mainly targets CDI-3 and is the highest-level layer in GA-FuL. The main purpose of this layer is to generate optimized code in a selected target programming language (TPL), given a sequence of operations on multivectors and other algebraic objects in GA-FuL. In essence, the components of this layer construct an optimizing compiler and code generator that takes an expression tree having scalar parameters and constant numbers as leafs and standard operations on scalars as internal nodes. The expression tree is automatically constructed using GA-FuL algebraic and geometric modeling components and is then optimized and transformed into TPL code through the optimizing compiler and code-generator components.

This layer is useful for software engineers wanting to create specialized code, all or part of which is automatically generated from operations on algebraic objects, especially GA multivectors. The typical sequence for using the components of this layer consists of the following stages:

1. Initialize a metaprogramming context object (MCO) by instancing the `MetaContext` class defined in the meta-context sub-layer.
2. Use the MCO to define algebraic objects acting as input parameters to the computational block.
3. Use algebraic operations, provided by GA-FuL algebra and modeling layers, to describe the intended algebraic steps.
4. Select the expected output variables of the computational block from the algebraic objects computed.
5. Set the TPL names of the input, intermediate, and output scalar variables to be used in the final generated code.
6. Use the MCO to optimize the computational block.
7. Initialize the intended code composer object (CCO) by instancing one of the classes defined in the code composers sub-layer.
8. Generate the final TPL optimized computational code using the CCO.

In the meta-context sub-layer, a special kind of scalar, called a "meta-expression scalar", is used. Essentially, a meta-expression scalar, Figure 4, is an expression tree similar to the ones typically used in computer algebra systems but with additional functionality for metaprogramming tasks. A computational block is constructed step by step by the user's code while being stored and optimized automatically inside the metaprogramming context object. Conceptually, a computational block consists of a sequence of assignment statements to TPL variables defining the required computations at the lowest level of scalar components of the algebraic objects. The conversion from the high-level GA\algebraic operations used in step 3 into low-level scalar operations is performed automatically by the components of the meta context sub-layer and managed by the MCO itself. Some of the variables in the computational block can be assumed by the user as independent, externally defined input parameters, with no attached left-hand-side meta-expressions. Other scalar variables are mostly intermediate ones, except for a few that are selected by the user as output variables of the computational block. The MCO manages the entire computational code construction and optimization process. The MCO contains factory objects to add constant numbers and input parameters to the computational block. The MCO can perform the following optimizations on the computational code before the final code-generation step:

- Propagation of constant values in metaexpressions on the right-hand side.
- Extraction of common subexpressions in right-hand-side metaexpressions into intermediate variables for reuse.

- Optional simplification of metaexpressions on the right hand side using an external computer algebra system\library.
- Pruning of intermediate variables having constant values or repeated right-hand-side metaexpressions or those not being used for computing an output variable.
- Optional reduction of the number of intermediate variables required.
- Optional reduction of computational steps in the code through a genetic programming algorithm [64–66] with a (4 + 1) evolutionary strategy.

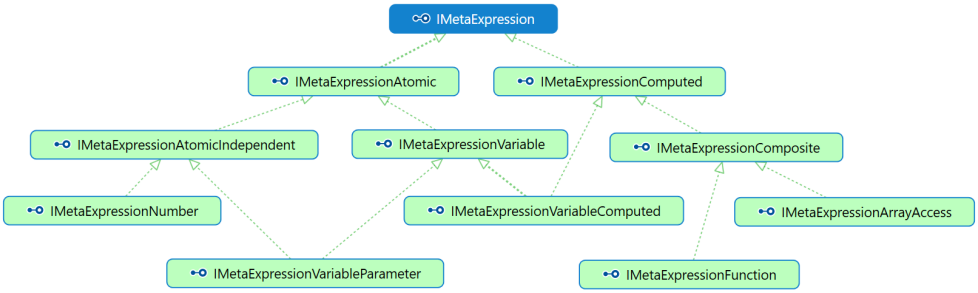

Figure 4. Public interfaces of GA-FuL meta-expressions in the meta-context sub-layer.

The purpose of the CCO is to convert the computational block assignment statements stored in the MCO into TPL code. The low-level metaexpressions of the computational block can only contain standard operations on scalars, such as negation, addition, subtraction, multiplication, division, power, trigonometric functions, exponential, and logarithms. Thus, the code composers sub-layer can be extended to provide code generation capabilities for almost any target programming language that supports such operations. The code composers sub-layer contains abstract classes for additional advanced code-generation tasks. The user can utilize classes of this sub-layer capable of template-based code generation for creating a wide range of general TPL code organizations, with or without using GA computations. These range from small text code with a single code file\module, to a large code library with complicated folder and code file structure.

4.3.4. System Utilities Layer

The utilities layer provides low-level services to the components in the other GA-FuL layers. The basic data structure sub-layer contains a set of data structures to aid in data storage and exchange in the system. For example, several classes that implement the generic `IReadOnlyDictionary<TKey, TValue>` interface are part of this sub-layer. The text\LaTeX utilities sub-layer provides core services for formatted text generation and LaTeX code composition extensively used by the system. The text-generation capabilities of this sub-layer are extensive. There are classes for composing formatted text, and parametric text templates and composers capable of creating full hierarchies of folders containing text files. The code-generation utilities sub-layer performs various low-level code-generation tasks used by the metaprogramming layer and other GA-FuL components. This includes components to represent and construct language-agnostic abstract syntax trees (ASTs) and code generators that can compose code based on the ASTs. Finally, the web graphics utilities sub-layer is used by the visualization sub-layer for generating suitable web-based code for rendering desired graphics from algebraic specifications in GA-FuL.

5. Examples of Applications Using GA-FuL

5.1. Modeling Inverse Kinematics of a 6R Robot

Most GA libraries rely on the direct coding of GA expressions using standard operations on multivectors, as described in Section 2. This approach might be useful for users experienced in GA but less so for general users. In GA-FuL, it is always possible to use the algebra layer for such formulations, as in other libraries. The modeling layer in GA-FuL, on the other hand, provides a much better solution for encoding geometric

operations without much need for explicit coding of expressions on multivectors. Coding the inverse kinematics of the 6R industrial robotic arm described in [37,67] illustrates the use of GA-FuL modeling and metaprogramming layers in producing efficient code for MATLAB. The following listing contains the required GA-FuL code:

```
// Stage 1: Initialize metaprogramming context
1 var context = new MetaContext();
2 context.AttachMathematicaExpressionEvaluator();
3 var cga = context.CreateCGaGeometricSpace5D();
4 var sp = context.ScalarProcessor;

// Stage 2: Define symbolic inputs Staubli robot link lengths
5 var d1 = context["d1"];
6 var a3 = context["a3"];
7 var d4 = context["d4"];

// desired position components of end-effector
8 var pX = context["Px"];
9 var pY = context["Py"];
10 var pZ = context["Pz"];

// Stage 3: Perform algebraic computations
// position of end-effector (a linear algebra vector)
11 var p = sp.Vector3D(pX, pY, pZ);

// origin position (a linear algebra vector)
12 var p0 = sp.ZeroVector3D();

// position of second joint (a linear algebra vector)
13 var p1 = p0 + sp.E3Vector3D(d1);

// plane passing through p0, p1 and p (a CGA IPNS blade)
14 var cgaPlane = cga.EncodeIpnsFlatPlaneFromPoints(p0, p1, p);

// sphere with center at p1 and radius a3 (a CGA IPNS blade)
15 var cgaSphere1 = cga.EncodeIpnsRealRoundSphere(a3, p1);

// sphere with center at p and radius d4 (a CGA IPNS blade)
16 var cgaSphere2 = cga.EncodeIpnsRealRoundSphere(d4, p);

// intersection point-pair of the plane and two spheres (a CGA IPNS blade)
17 var cgaPointPair = cgaSphere2.MeetIpns(cgaSphere1, cgaPlane);

// extract one end-point from point-pair (a linear algebra vector)
18 var p2 = cgaPointPair.DecodeIpnsPointPairVGaPoint1AsVector3D();

// normal vector to plane (a linear algebra vector)
19 var normal = cgaPlane.DecodeIpnsFlatVGaNormalAsVector3D();

// cos of first joint angle theta1 (a scalar)
20 var theta1Cos = normal.GetAngleCosWithE1();

// auxiliary lines l1, l2, and l3 (CGA OPNS blades)
21 var cgaLine1 = cga.EncodeOpnsFlatLineFromPoints(p0, p1);
22 var cgaLine2 = cga.EncodeOpnsFlatLineFromPoints(p1, p2);
23 var cgaLine3 = cga.EncodeOpnsFlatLineFromPoints(p2, p);

// square roots of modules of auxiliary lines l1, l2, and l3 (scalars)
24 var l11 = cgaLine1.SpSquared().Sqrt();
25 var l22 = cgaLine2.SpSquared().Sqrt();
26 var l33 = cgaLine3.SpSquared().Sqrt();

// cos angle between lines l1 and l2 (a scalar)
27 var theta2Cos = cgaLine2.Sp(cgaLine1) / (l11 * l22);

// cos angle between lines l2 and l3 (a scalar)
28 var theta3Cos = cgaLine2.Sp(cgaLine3) / (l22 * l33);
```

```
// Stage 4: Define target language variable names for inputs, outputs,
// and intermediate variables, and optimize internal expression tree
29 d1.SetExternalName("d1");
30 a3.SetExternalName("a3");
31 d4.SetExternalName("d4");

32 pX.SetExternalName("px");
33 pY.SetExternalName("py");
34 pZ.SetExternalName("pz");

35 theta1Cos.SetAsOutput("theta1Cos");
36 theta2Cos.SetAsOutput("theta2Cos");
37 theta3Cos.SetAsOutput("theta3Cos");

38 context = context.OptimizeContext();

39 context.SetComputedExternalNamesByOrder(index => $"temp{index}");

// Stage 5: Define a MATLAB code composer and Generate the final code
40 var codeComposer = context.CreateContextCodeComposer(
GaFuLLanguageServerBase.MatlabFloat64()
);
41 var code = codeComposer.Generate();
42 Console.WriteLine(code);
```

In the first stage of the code (lines 1 to 3), the metaprogramming context and related processors are defined. In stage 2 (lines 5 to 10), the input parameters are defined to be used in the following subroutine computations. These two stages are purely for the initialization overhead of the metaprogramming layer. In stage 3 (lines 11 to 28), actual computations are performed by constructing and manipulating algebraic objects. This can be done explicitly using GA operations on multivectors through the GA-FuL algebra layer or implicitly using GA-FuL modeling layer capabilities. Additionally, during stage 3, an expression tree that relates inputs to outputs is automatically constructed internally. In stage 4 (lines 29 to 39), the modeling layer is again used to define MATLAB variable names for input, output, and intermediate scalar variables in addition to the optimization of the internal expression tree of the metaprogramming context. Finally, stage 5 defines the desired code generator for MATLAB and generates the code based on the optimized expression tree.

The computational code in stage 3 is identical to the cases where a numerical, symbolic, or metaprogramming computation is required for the problem. Additionally, the same GA-FuL code as a whole can be used to generate optimized code for other target languages such as C++, Python, or C# simply by changing the selected target language in line 40. Using the CGA geometric space processor on line 3, geometric objects and operations are described in a higher-level coding style that is easier to manage for most library users. For comparison, the code provided in [37] for the same problem is a direct algebra-to-code translation. Although this gives more control over algebraic expressions, it requires a high degree of familiarity with GA mathematics.

5.2. Library Generation of Arbitrary GAs

Metaprogramming capabilities of GA-FuL can be used not only to generate optimized code for a single algorithm as a TPL subroutine, but it can be scaled up to the level of composing a whole code library for a selected TPL with a sophisticated folder structure and numerous code files. To illustrate this capability, a MATLAB toolbox (https://github.com/ga-explorer/GA-FuL-MATLAB-Toolbox accessed on 15 June 2024) was created purely by code generation using GA-FuL metaprogramming capabilities. The GA-FuL MATLAB toolbox has a separate MATLAB namespace folder for each GA with a given signature. For example, `ga41` is the namespace for 5D CGA, while `ga301` is for 4D PGA. Inside each GA namespace, there is a single class called `Multivector` that internally holds an array of real numbers containing the scalar components of the multivector. For any n-dimensional GA, the `Multivector` class can represent a k-vector of any valid grade $0 \leq k \leq n$ having $\binom{n}{k}$ scalar components or a full multivector having 2^n scalar components. A single scalar component

of a multivector is actually a whole row of 64-bit floating point numbers. This enables the representation of sampled multivector signals and the modeling of sophisticated signal processing tasks using GA operations on multivector signals.

In addition to the GA-FuL MATLAB toolbox, similar code libraries for Python and JavaScript are currently being developed. The following code listing illustrates a simple example for using 3D VGA in the GA-FuL MATLAB toolbox:

```
% Creating two 3D Euclidean GA scalars and two vectors:
a = ga3.EncodeScalar(3.5);
b = ga3.EncodeScalar(-1.25);
v = ga3.EncodeVector([1.2, -2.4, 1.3]);
w = ga3.EncodeVector([-3.1, -1.6, 2.23]);

% Any multivector object has several important properties:
% The Grade of a k-vector, a full multivector returns -1
% in this property
v.Grade

% The number of scalars per multivector, this is dependent
% on the grade and GA space dimensions
v.ScalarCount

% The number of samples in this multivector
v.SampleCount

% The internal data array storing the scalars of this multivector
v.Data

% For any multivector, LaTeX code can be dispalyed as a string:
a.getLaTeX()
b.getLaTeX()
v.getLaTeX()
w.getLaTeX()

% negative, reverse, grade involution, conjugate, dual, undual, inverse
vn = v.negative(); % You can also use vn = -v;
v.reverse().getLaTeX()
v.gradeInvolution().getLaTeX()
v.conjugate().getLaTeX()
v.dual().getLaTeX()
v.unDual().getLaTeX()
v.inverse().getLaTeX()

% Norm, squared norm
v.norm()
v.normSquared()

% Linear combinations:
s = v * 2 - w * 0.5 + a;

% Products: geometric, outer, scalar, left\right contraction, inner
v.gp(w).getLaTeX()
v.op(w).getLaTeX()
v.sp(w)
v.lcp(w).getLaTeX()
v.rcp(w).getLaTeX()
v.fdp(w).getLaTeX()

% Multivector parts
m = a + v + w.dual();
m.getLaTeX()
m.getScalarPart().getLaTeX() % Same as m.getKVectorPart(0).getLaTeX()
m.getVectorPart().getLaTeX() % Same as m.getKVectorPart(1).getLaTeX()
m.getBivectorPart().getLaTeX() % Same as m.getKVectorPart(2).getLaTeX()
m.getTrivectorPart().getLaTeX() % Same as m.getKVectorPart(3).getLaTeX()
m.getDataArray()
```

5.3. Family of Rotations between Two Unit Vectors in 3D

Given two unit vectors $\boldsymbol{u}, \boldsymbol{v}$ in a three-dimensional Euclidean space, there is a parametric family of rotations that map $\boldsymbol{u}$ to $\boldsymbol{v}$ [68]. Symbolic processing capabilities of Wolfram Mathematica, exploited through GA-FuL algebra, and modeling components were used to formulate a simple geometric procedure with closed-form expressions for defining this family of rotations. To the best of the authors' knowledge, the expressions described here are not found elsewhere. A simplified special case, where $\boldsymbol{u} = (0, 0, 1)$ and $\boldsymbol{v} = \frac{1}{\sqrt{3}}(1, 1, 1)$, is detailed in a previous paper by the authors [69]. In addition, GA-FuL visualization components were used to create video animations illustrating the whole procedure (https://github.com/ga-explorer/GeometricAlgebraFulcrumLib/tree/main/GeometricAlgebraFulcrumLib.Visualizations/Rotor%20Family%20Videos accessed on 15 June 2024). Figure 5 illustrates one frame from this video animation.

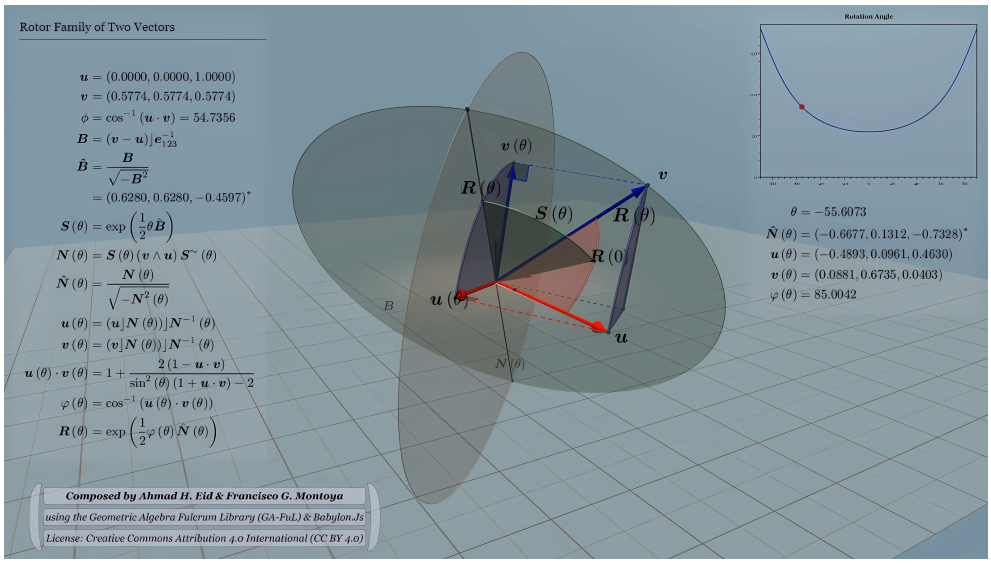

Figure 5. Illustration of the geometric procedure for defining a family of rotations between two unit vectors.

The family of rotations can be defined as a parametric rotor function $\boldsymbol{R}(\theta)$ of a single free parameter $\theta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ such that $\boldsymbol{v} = \boldsymbol{R}(\theta)\boldsymbol{u}\boldsymbol{R}^{\sim}(\theta)$. For each rotor $\boldsymbol{R}(\theta)$ in the family, the angle and unit bivector of rotation are indicated by $\varphi(\theta) \in [\phi, \pi]$ and $\hat{\boldsymbol{N}}(\theta)$, respectively, where $\phi = \cos^{-1}(\boldsymbol{u} \cdot \boldsymbol{v})$, and $\varphi(\theta) = \varphi(-\theta)$ is an even function in the parameter θ satisfying $\varphi\left(-\frac{\pi}{2}\right) = \varphi\left(\frac{\pi}{2}\right) = \pi$ and $\varphi(0) = \phi$.

The geometry of this method requires the definition of a set of pure rotors

$$\boldsymbol{S}(\theta) = \sqrt{\frac{1}{2}(1 + \cos\theta)} + \sqrt{\frac{1}{2}(1 - \cos\theta)}\frac{1}{\sqrt{-\boldsymbol{B}^2}}\boldsymbol{B}$$

where $\boldsymbol{B} = (\boldsymbol{v} - \boldsymbol{u})^*$ is the 2-blade orthogonal to the difference vector $\boldsymbol{v} - \boldsymbol{u}$. Any single rotor $\boldsymbol{S}(\theta)$ performs a rotation in plane $\boldsymbol{B}$ by angle $\theta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$. Next, the constant bivector $\boldsymbol{N} = \boldsymbol{v} \wedge \boldsymbol{u}$ is rotated using $\boldsymbol{S}(\theta)$ to obtain

$$\boldsymbol{N}(\theta) = \boldsymbol{S}(\theta)(\boldsymbol{v} \wedge \boldsymbol{u})\boldsymbol{S}^{\sim}(\theta) = [\boldsymbol{S}(\theta)\boldsymbol{v}\boldsymbol{S}^{\sim}(\theta)] \wedge [\boldsymbol{S}(\theta)\boldsymbol{u}\boldsymbol{S}^{\sim}(\theta)]$$

Then, the vectors $\boldsymbol{u}, \boldsymbol{v}$ are projected onto $\boldsymbol{N}(\theta)$ to get

$$\boldsymbol{u}(\theta) = \boldsymbol{N}(\theta)\lfloor \boldsymbol{u}\rfloor \boldsymbol{N}^{-1}(\theta)$$
$$\boldsymbol{v}(\theta) = \boldsymbol{N}(\theta)\lfloor \boldsymbol{v}\rfloor \boldsymbol{N}^{-1}(\theta)$$

Finally, $\boldsymbol{R}(\theta)$ is fully defined as the rotor mapping unit vector $\boldsymbol{u}(\theta)$ into $v(\theta)$ through the smallest angle between them $\varphi(\theta) = \cos^{-1}(\boldsymbol{u}(\theta) \cdot \boldsymbol{v}(\theta))$. The described procedure can be generalized to higher dimensions $n > 3$ by defining the set of all two blades $\boldsymbol{B}$ that reside in the $(n-1)$-dimensional subspace $\boldsymbol{C} = (\boldsymbol{v} - \boldsymbol{u})^*$ dual to vector $\boldsymbol{v} - \boldsymbol{u}$. A proper closed-form formulation for generalizing this procedure to higher dimensions is currently under investigation.

The GA procedure for defining the family of rotations $\boldsymbol{R}(\theta)$ can be summarized in the following sequence of relations:

$$\begin{aligned}
\cos\varphi(\theta) &= \frac{\cos(2\theta-\phi)+\cos(2\theta+\phi)+2\cos(2\theta)+6\cos(\phi)-2}{\cos(2\theta-\phi)+\cos(2\theta+\phi)+2\cos(2\theta)-2\cos(\phi)+6} \\
&= 1+\frac{2(1-\boldsymbol{u}\cdot\boldsymbol{v})}{\sin^2(\theta)(1+\boldsymbol{u}\cdot\boldsymbol{v})-2} \qquad (9)\\
\text{where} &: -\frac{\pi}{2} \le \theta \le \frac{\pi}{2} \\
&\Updownarrow \\
\theta(\varphi) &= \pm\sin^{-1}\sqrt{2\frac{\boldsymbol{u}\cdot\boldsymbol{v}-\cos\varphi}{(1+\boldsymbol{u}\cdot\boldsymbol{v})(1-\cos\varphi)}} \qquad (10)\\
\text{where} &: \cos^{-1}(\boldsymbol{u}\cdot\boldsymbol{v}) < \varphi < \pi \\
&\Downarrow \\
\boldsymbol{S}(\theta) &= \exp\left(\frac{1}{2}\theta\hat{\boldsymbol{B}}\right) \\
&= \sqrt{\frac{1}{2}(1+\cos\theta)}+\sqrt{\frac{1}{2}(1-\cos\theta)}\frac{1}{\sqrt{-\boldsymbol{B}^2}}\boldsymbol{B} \qquad (11)\\
\text{where} &: \boldsymbol{B} = (\boldsymbol{v}-\boldsymbol{u})^* = (\boldsymbol{u}-\boldsymbol{v})\rfloor\boldsymbol{e}_{123} \\
&\Downarrow \\
\boldsymbol{N}(\theta) &= \boldsymbol{S}(\theta)(\boldsymbol{v}\wedge\boldsymbol{u})\boldsymbol{S}^{\sim}(\theta) \\
&= [\boldsymbol{S}(\theta)\boldsymbol{v}\boldsymbol{S}^{\sim}(\theta)]\wedge[\boldsymbol{S}(\theta)\boldsymbol{u}\boldsymbol{S}^{\sim}(\theta)] \qquad (12)\\
\hat{\boldsymbol{N}}(\theta) &= \frac{1}{\sqrt{-\boldsymbol{N}^2(\theta)}}\boldsymbol{N}(\theta) \qquad (13)\\
&\Downarrow \\
\boldsymbol{R}(\theta) &= \exp\left(\frac{1}{2}\varphi(\theta)\hat{\boldsymbol{N}}(\theta)\right) \\
&= \sqrt{\frac{1}{2}(1+\cos\varphi(\theta))}+\sqrt{\frac{1}{2}(1-\cos\varphi(\theta))}\hat{\boldsymbol{N}}(\theta) \qquad (14)
\end{aligned}$$

An interesting application of this methodology is the simple Kirchhoff rotation (SKR) method [69]. It is implemented via GA-FuL and offers significant computational advantages over traditional methods such as Clarke's transformation while providing a unifying framework for understanding transformations in power systems. In practical applications, the SKR method exhibits linear growth in computational complexity, whereas Clarke's method grows exponentially with the number of dimensions. This is particularly evident in scenarios involving higher-dimensional systems, where SKR outperforms Clarke's method by a substantial margin. For instance, in a 15-phase system, SKR was observed to be over ten times faster than Clarke's method, highlighting its scalability and efficiency for large-scale power system analyses.

Additionally, the SKR method simplifies the mathematical formulation of power system transformations by leveraging the geometric properties of rotors and bivectors. This not only enhances computational performance but also provides clearer geometric insights into the underlying physical processes. The reduced computational burden and

increased intuitive understanding make the SKR method a powerful tool for engineers and researchers working on complex power systems, offering a distinct advantage over traditional matrix-based approaches.

Table 4 illustrates the performance comparison between the Clarke and SKR methods. The results demonstrate that SKR's computational complexity increases linearly, while Clarke's method exhibits exponential growth. This efficiency is particularly beneficial for high-dimensional power systems such as multi-phase electrical machines, making SKR a superior choice for practical applications.

Table 4. Comparison of Computational Efficiency between the Clarke and SKR Methods.

# Dimensions	Clarke Method (ms)	SKR Method (ms)
3	27.86	12.20
4	36.71	13.28
5	48.36	13.62
6	61.47	14.39
7	77.13	15.22
8	94.84	16.14
9	115.63	17.08
10	143.92	17.78
15	294.58	23.88
20	490.91	33.10
25	740.86	43.89
30	1057.39	56.34
35	1399.00	63.22
40	1812.72	71.17
45	2281.55	79.43

5.4. Geometric Models for Power Systems Applications

The GA-FuL library offers specialized computational tools tailored for geometric algebras (GAs), enhancing the practical application of GA in a wide range of engineering fields. Moreover, the GA-FuL library provides bindings for both MATLAB and Mathematica, enabling seamless integration with these popular technical computing platforms. The optimized algorithms and straightforward API make it easy to leverage GA-FuL for efficient GA computations. Recent studies have highlighted its utility, particularly in the fields of power systems and electrical engineering. Using the GA-FuL library, researchers have found a more streamlined approach to implementing GA techniques, facilitating improved analysis of power systems and electrical circuits. This integration suggests that the library could be a valuable asset for those seeking to bridge the gap between theoretical GA constructs and tangible engineering solutions.

In the paper "Geometric Algebra Framework Applied to Symmetrical Balanced Three-Phase Systems for Sinusoidal and Non-Sinusoidal Voltage Supply" [70], the GA-FuL library is leveraged to analyze three-phase circuits under non-sinusoidal supply using GA. The voltage and current are modeled as geometric vectors projected onto different harmonic components. GA-FuL computes the geometric power and decomposes it into active and non-active parts by inverting the voltage vector. This provides new physical insights compared to traditional methods. In this case, the modular GA-FuL design allows for the testing of different problem configurations.

In "Determination of Instantaneous Powers from a Novel Time-Domain Parameter Identification Method of Non-Linear Single-Phase Circuits" [71] and "Combining Real-time Parameter Identification and Robust Control Algorithms for Effective Control of Electrical Machines" [72], GA-FuL is applied to identify parameters of nonlinear loads by fitting their differential equations to measured data. The intrinsic geometric properties of the resulting parametric curves yield the equivalent impedance and admittance. This physics-based modeling approach leverages GA-FuL's support for derivatives, multivector products, and symbolic manipulation.

In a series of two papers "A Geometric Procedure for Computing Differential Characteristics of Multi-phase Electrical Signals using Geometric Algebra" [73] and "Frequency Generalization via Darboux Bivector and Electrical Curves in Multi-Phase Power Systems" [74], GA-FuL is employed for computing geometric frequency and curvatures of multi-phase voltage/current curves. By fitting the discrete signals as continuous parametric curves, their intrinsic differential geometric properties are revealed. GA-FuL is used to compute derivatives, orthogonalization, rotations, and other operations to extract entities such as the Darboux bivector and detect oscillations. This expands the theoretical work on geometric frequency using computational techniques enabled by the efficient GA-FuL implementation. These works significantly expand the geometric frequency concept by generalizing it to multi-phase systems of any complexity using GA-FuL and its ability to handle a high number of dimensions.

In [69], "A Systematic and Comprehensive Geometric Framework for Multiphase Power Systems Analysis and Computing in Time Domain", a new theoretical framework for modeling power systems using GA is presented. The authors build on established GA concepts such as blades and geometric product to formulate electrical properties such as voltage, current, and power in terms of multivectors. GA-FuL is used to symbolically derive expressions for computing power quantities and to numerically validate results on test cases. The efficiency and simplicity of the GA-FuL implementations enable rapid prototyping and testing of the GA power theory formulations.

The paper"Formulating the Geometric Foundation of Clarke, Park and FBD Transformations by Means of Clifford's Geometric Algebra" [50] derives fundamental power system transformations such as Clarke, Park, and FBD using the Euclidean vector GA. Rotations and projections of voltage/current vectors are formulated as GA rotors without matrices or complex numbers. GA-FuL enables the symbolic manipulation of rotors and numerical validation against traditional methods. This provides a geometric grounding for the transformations while showcasing the capabilities of GA-FuL.

The paper "Geometric Algebra Applied to Multiphase Electrical Circuits in Mixed Time-Frequency Domain by Means of Hypercomplex Hilbert Transform" [75] utilizes GA-FuL to implement a hypercomplex Hilbert transform for analyzing nonsinusoidal multi-phase circuits. GA-FuL computes geometric voltage/current vectors, projects them onto different harmonics, and transforms between time and frequency domains. This provides greater insight than do phasor methods. The efficiency of GA-FuL operations facilitated testing of the approach.

5.5. New Efficient Implementation of the Resection Problem (Snellius–Pothenot)

The resection problem, also known as the Snellius–Pothenot problem, involves determining the position of an unknown point P using the positions of three known points A, B, and C, along with relative angular measurements from P. This problem is significant in fields such as geodesy, survey, robotics, and computer vision. Traditionally, solutions to this problem have relied heavily on algebraic methods, which can be complex and challenging to understand.

GA-FuL has played a crucial role in addressing this problem by offering a comprehensive framework that supports various forms of geometric algebra, including VGA, CGA, PGA, and others [76]. It also provides a robust platform for implementing geometric computations by leveraging the strengths of different geometric algebras. For instance, VGA is well-suited for simple 2D and 3D geometric problems, while CGA extends these capabilities to handle conformal transformations, making it ideal for more complex spatial problems. PGA, on the other hand, excels in projective geometry scenarios. This versatility allows GA-FuL to adapt to the specific requirements of the resection problem, optimizing both the computational process and the clarity of the solution.

Moreover, GA-FuL's implementation leverages advanced metaprogramming techniques to generate highly optimized code for geometric algebra operations. This results in significant performance improvements over traditional methods. Table 5 illustrates

the computational efficiency achieved with GA-FuL compared to conventional algebraic approaches. This efficiency is particularly beneficial for applications that require real-time computations, such as robotics and computer vision.

Table 5. Comparison of the Computational Efficiency for the 2D Resection Problem.

Method	Comp. Time (ms)	Remarks
Best algeb. method	21.919	Efficient and less intuitive
CGA method	196.122	More intuitive, less efficient
VGA method	18.997	Most efficient, highly intuitive

6. Comparisons with Similar Software Libraries

6.1. Capability Comparisons

GA-FuL is intended to be an expansive and widely usable software library for GA-based algebraic models, covering the range from initial prototyping to full production code generation. Compared to other GA software libraries, GA-FuL capabilities can be summarized as follows:

- Based on the presentation of Section 3, there are three main classes of GA libraries: numerical\symbolic computational libraries, library generators, and optimizing subroutine generators. Currently, there is no single system capable of performing all three classes of GA-based coding tasksmm except for GA-FuL.
- Very few systems can handle GAs with high dimensions ($n > 20$). Those that do include Gallant, Grassmann.jl, and TbGAL, as reported in [34]. GA-FuL can handle arbitrary GAs with any dimension as long as the number of scalar components in a single sparse multivector is reasonably small.
- GA-FuL is not yet capable of performing specialized matrix-based algorithms on arrays of multivectors (singular-value decomposition, eigendecomposition, etc.), similar to the ones performed by some MATLAB toolboxes. However, the design of GA-FuL's modeling layer can be easily extended to implement such algorithms in the future.
- In contrast with some systems, in GA-FuL, several GAs can be utilized within a single model without any need (or overhead) for switching between algebras.
- In addition to GA-FuL's predecessor system, GMac, only one other active system, GAALOP, can generate individual subroutines containing optimized code for a variety of target languages. GA-FuL is capable of generating similar subroutines with better code-generation capabilities and computational performance.
- Most good GA computational libraries can handle both symbolic and numeric scalar components of multivectors through the use of template metaprogramming methods. The same is present in GA-FuL algebra layer, where any kind of valid numeric or symbolic scalar can be used, including specialized scalars such as arbitrary precision or rational numbers, sampled signals, and symbolic expression trees.
- Some libraries, such as TbGAL, can only handle non-degenerate GAs where no basis vector squares to zero. This limitation is not present in GA-FuL, where any metric signature can be handled seamlessly.
- Very few libraries have intrinsic visualization capabilities, with Ganja.js being one of the best so far. GA-FuL has initially good visualization capabilities, and the addition of interactive animations is currently under development, gradually targeting the level of Ganja.js capabilities.

6.2. Design Comparisons

For any software library, the overall system design highly determines the ability of the library to address its intended objectives and use cases. In addition, system design affects the scalability of the library and its overall capabilities. The main design characteristics of GA-FuL compared to other GA software are:

- Systems such as Garamon have a restrictively inefficient method for representing k-vectors in GAs with high dimensions. The use of dense arrays for representing k-vectors is the reason behind such a problem. Systems like Ganja.js and GA-FuL, avoid this design problem by using sparse data structures for representing k-vectors, which are highly sparse in most practical applications.
- In most GA systems, the most restrictive processing task is the computation of the geometric and derived bilinear products on general multivectors containing m basis blade components. Typical implementations have $O(m^4)$ performance for bilinear products, severely impeding high values for m. In GA-FuL, as well in other systems such as Garamon and Ganja.js, a multivector represents per-grade k-vectors. This enables more efficient $O(m^3)$ implementations for bilinear products.
- GA-FuL's algebra layer is functionally comparable to computational GA libraries based on template metaprogramming methods. The main difference is that GA-FuL's algebra layer is designed for prototyping and efficient memory usage, not for high processing performance. The code-generation facilities of GA-FuL are more than capable of achieving very high processing performance, so this objective is delegated to the metaprogramming layer in GA-FuL.
- Many GA-related programs ignore classical algebraic objects, such as complex numbers, quaternions, classical vectors, and polynomials, in favor of GA's multivectors. In addition, some GA systems, such as TbGAL, focus only on geometric algebra (multivectors are restricted to become blades or versors), not on general Clifford algebras. For practical applications, however, this design choice introduces a significant coding overhead when interfacing with external packages and datasets relying on such classical algebraic objects. GA-FuL is designed to handle, convert, and mix classical algebraic objects as well as GA\CA multivectors within a single modeling task seamlessly.
- Some GA software systems, such as Gaigen, GAALOP, and GMac, use a DSL for expressing GA operations. This approach was found to be very limiting compared to using an API inside the host programming environment. GA-FuL can utilize the full capabilities of its underlying programming environment, the.NET framework, in service of its CDIs. Currently, this is the most common approach for GA software in general.

6.3. Performance Comparisons

Through the GA-FuL metaprogramming layer, the user is able to generate highly efficient code in a selected programming language based on the algebraic and geometric modeling facilities of other GA-FuL layers. Table 6 provides an illustrative comparison with GAALOP as a GA-based system of similar purpose. Five geometric problems were coded using GAALOP and GA-FuL for code generation. As can be seen in the table, the first three problems, which are relatively simple, required very similar execution times. For the final two problems, however, the relative complexity of the problems increased, leading to significant scalability degradation in code generated using GAALOP compared to GA-FuL metaprogramming. For the Snellius–Pothenot problem, it was noticed that the use of symbolic optimization in GAALOP (through the CAS maxima) took a significantly longer time to produce final code. Additionally, the quality of the GAALOP-maxima generated code actually degraded significantly for this problem in terms of complexity of final scalar expressions in contrast to GA-FuL-generated code that scaled gracefully with problem complexity. The main conclusion illustrated by these results is the superiority of GA-FuL compared to GAALOP for more complex problems in terms of scalability, code quality, code-generation time, and execution time. Combining these features with GA-FuL library-generation capabilities leads to a much wider usability of GA-FuL compared to other GA-based systems, including GAALOP. For illustration purposes, and as kindly suggested by the first reviewer, the following listing contains the GAALOPScript code

used for the ReconstructMotor problem. The code is kindly provided by the developers of GAALOPWeb publicly online.

```
ExpApprox = {
1 + _P(1) + _P(1)*_P(1)/2 + _P(1)*_P(1)*_P(1)/6 +_P(1)*_P(1)*_P(1)
*_P(1)/24
}

Motor = {
// computes the motor between two points, lines or planes
// as the sqrt of them
!temp = 1+_P(1)/_P(2);
!abstemp = abs(temp);
temp/abstemp
}

// original points
A1 = createPoint(ax, ay, az);
B1 = createPoint(bx, by, bz);
C1 = createPoint(cx, cy, cz);

// arbitrary transformation
!GT = ExpApprox(0.3*(e1^e3) + 0.2*(e2^e0));
// corresponding target points
?At = GT * A1 * ~GT;
?Bt = GT * B1 * ~GT;
?Ct = GT * C1 * ~GT;

// Transformation from A1 to At
// (translation)
!VA = Motor(At, A1);
!A2 = VA * A1 * ~VA;
!B2 = VA * B1 * ~VA;
!C2 = VA * C1 * ~VA;

// Transformation from B2 to Bt
// based on the rotation from the line L2 to L1
!L1 = *(*At ^ *Bt);
!L2 = *(*At ^ *B2);
!VB = Motor(L1, L2);
!B3 = VB * B2 * ~VB;
!C3 = VB * C2 * ~VB;

// Transformation from C3 to Ct
// based on the rotation of two planes
!P1 = *(*L1 ^*Ct);
!P2 = *(*L1 ^*C3);
!VC = Motor(P1,P2);

// complete transformation
!V = VC * VB * VA;
// interplation motor
lerp = 1 * (1-t) + V * t;
// interpolated points
?AI = lerp * A1 * ~lerp;
?BI = lerp * B1 * ~lerp;
?CI = lerp * C1 * ~lerp;
```

Several GA software libraries attempt to accelerate bilinear products on multivectors using a lookup table per product per GA metric. This approach becomes highly impractical

after about 10 dimensions and prevents the use of several GAs in the same computational task. The use of large lookup tables for accelerating multivector computations, mainly GA bilinear products, is contradictory to GA-FuL CDI-2. As such, only a single lookup table for the geometric product of Euclidean signature GA is created. All bilinear products for all GA metrics are efficiently computed from this single lookup table using the generalized approach in [26]. The measured memory size s in bytes for this table follows the exponential formula $s = 0.258 \times 2^{1.9359n}$, where n is the maximum dimension of the base vector space for the geometric algebra. This size ranges from 2,539,700 bytes for 12 dimensions up to 544,276,344 bytes for 16 dimensions and approximately quadruples for each increase in dimensions by one. For typical code-generations tasks. the memory requirements of GA-FuL are relatively small. For example, the ReconstructMotor example requires fewer than 40 megabytes for full execution, assuming a lookup table for a maximum of 12 dimensions.

Table 6. Average Time Required for 1000 Executions of Generated Code from GAALOP vs. GA-FuL for 5 Problems.

Problem		Execution Time (µs)	
Name	**Description**	**GAALOP**	**GA-FuL**
RotatingSquare	Rotate a fixed square using a parametric angle in 2D [8]	9.582	9.532
SphereCenter	Compute center of sphere given 4 points on the sphere in 3D [7]	20.779	19.013
ThreeSpheresIntersection	Compute the point pair of intersection of 3 spheres in 3D [7]	54.072	55.048
Snellius–Pothenot Problem	Solve resection problem in 2D [76]	600.921	168.923
ReconstructMotor	Reconstructing a motor from exact point correspondences in 3D [8]	1064.652	179.460

7. Conclusions

The Geometric Algebra Fulcrum Library (GA-FuL) is presented in this paper. The library is designed as a wide-purpose computational library for geometric algebra, linear algebra, and other classical algebras. GA-FuL is capable of performing all three categories of tasks typically covered by separate GA libraries. The high-level design of GA-FuL is based on a set of core intentions realized through data-oriented programming principles in a layered software system. The paper provides example use cases for the library and comparisons with similar libraries on levels of capability, design, and run-time performance.

For future developments, the GA-FuL calculus sub-layer is intended for full development. Interactive visualization capabilities are planned for implementation to rival the level of similar systems such as Ganja.js. Several research projects that can greatly benefit from prototyping and code-generation capabilities are currently being planned in various application domains. This includes generating a full numerical computational GA library for Python and other mainstream programming languages.

Author Contributions: Conceptualization, A.H.E. and F.G.M.; methodology, A.H.E. and F.G.M.; software, A.H.E.; validation, A.H.E. and F.G.M.; writing—original draft preparation, A.H.E. and F.G.M.; visualization, A.H.E. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in the study are included in the article and online code repository at https://github.com/ga-explorer/GeometricAlgebraFulcrumLib accessed on 19 June 2024, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

API	application programming interface
AST	abstract syntax tree
CA	Clifford algebra
CAS	computer algebra system
CCO	code composer object
CDI	core design intention
CGA	conformal geometric algebra
DOP	data-oriented programming
DSL	domain-specific language
GA	geometric algebra
HGA	homogeneous geometric algebra
MCO	meta-context object
OOP	object-oriented programming
PGA	projective geometric algebra
SKR	simple Kirchhoff rotation
TPL	target programming language
VGA	vector geometric algebra

References

1. Bayro-Corrochano, E. A Survey on Quaternion Algebra and Geometric Algebra Applications in Engineering and Computer Science 1995–2020. *IEEE Access*. **2021**, *9*, 104326–104355. [CrossRef]
2. Corrochano, E. *Bayro Corrochano: Eduardo Jose*; Springer: Berlin/Heidelberg, Germany, 2019.
3. Bayro-Corrochano, E. *Geometric Algebra Applications Vol. II: Robot Modelling and Control*; Springer International Publishing AG: Cham, Switzerland, 2020.
4. Bhatti, U.; Yu, Z.; Yuan, L.; Zeeshan, Z.; Nawaz, S.; Bhatti, M.; Mehmood, A.; Ain, Q.; Wen, L. Geometric Algebra Applications in Geospatial Artificial Intelligence and Remote Sensing Image Processing. *IEEE Access* **2020**, *8*, 155783–155796. [CrossRef]
5. Breuils, S.; Tachibana, K.; Hitzer, E. New Applications of Clifford's Geometric Algebra. *Adv. Appl. Clifford Algebr.* **2022**, *32*, 17. [CrossRef]
6. Doran, C. *Geometric Algebra for Physicists*; Cambridge University Press: Cambridge, UK, 2013.
7. Dorst, L.; Fontijne, D.; Mann, S. *Geometric Algebra for Computer Science (Revised Edition): An Object-Oriented Approach to Geometry*; Morgan Kaufmann Publishers Inc.: Burlington, MA, USA, 2009.
8. Dorst, L. A Guided Tour to the Plane-Based Geometric Algebra Pga, Version 2.0. 2022. Available online: https://bivector.net/PGA4CS.html (accessed on 15 June 2024).
9. Gunn, C. Geometric Algebras for Euclidean Geometry. *Adv. Appl. Clifford Algebr.* **2016**, *27*, 185–208. [CrossRef]
10. Gunn, C.G. Doing Euclidean Plane Geometry Using Projective Geometric Algebra. *Adv. Appl. Clifford Algebr.* **2016**, *27*, 1203–1232. [CrossRef]
11. Gunn, C.G.; Keninck, S.D. Geometric algebra and computer graphics. In Proceedings of the ACM SIGGRAPH 2019 Courses. ACM, Los Angeles, CA, USA, 28 July 2019. [CrossRef]
12. Hestenes, D.; Sobczyk, G. *Clifford Algebra to Geometric Calculus*; Springer: Dordrecht, The Netherlands, 1984. [CrossRef]
13. Hestenes, D. *New Foundations for Classical Mechanics*, 2nd ed.; Number 99 in Fundamental Theories of Physics; Kluwer Academic Publishers: Dordrecht, The Netherlands, 2003.
14. Hestenes, D. Tutorial on Geometric Calculus. *Adv. Appl. Clifford Algebr.* **2013**, *24*, 257–273. [CrossRef]
15. Hildenbrand, D. *Foundations of Geometric Algebra Computing*; SpringerLink, Springer: Berlin/Heidelberg, Gernamny, 2013.
16. Hildenbrand, D. *Introduction to Geometric Algebra Computing*, 1st ed.; Computer Science, CRC Press, Taylor & Francis Group: Boca Raton, FL, USA; A Chapman & Hall Book: London, UK, 2020.
17. Josipović, M. *Geometric Multiplication of Vectors*; Springer eBooks, Birkhäuser: Cham, Switzerland, 2020.
18. Kanatani, K. *Understanding Geometric Algebra*; Taylor & Francis Group: Milton Park, UK, 2020.
19. Lengyel, E. *Projective Geometric Algebra Illuminated*; Terathon Software LLC: Lincoln, CA, USA, 2024.
20. MacDonald, A. *Linear and Geometric Algebra*, nachdr ed.; Createspace: Scotts Valley, CA, USA, 2012.
21. Macdonald, A. *Vector and Geometric Calculus*; CreateSpace Independent Publishing Platform: Scotts Valley, CA, USA, 2012; Volume 12.
22. Macdonald, A. A Survey of Geometric Algebra and Geometric Calculus. *Adv. Appl. Clifford Algebr.* **2016**, *27*, 853–891. [CrossRef]
23. Perwass, C. *Geometric Algebra with Applications in Engineering*; Number 4 in Geometry and Computing; Springer: Berlin/Heidelberg, Germany, 2009.
24. Wareham, R.; Cameron, J.; Lasenby, J. Applications of Conformal Geometric Algebra in Computer Vision and Graphics. In *Computer Algebra and Geometric Algebra with Applications*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 329–349. [CrossRef]

25. Ivey, T.A. *Cartan for Beginners*, 2nd ed.; Number Volume 175 in Graduate Studies in Mathematics; American Mathematical Society: Providence, RI, USA, 2016.
26. Eid, A.H. An Extended Implementation Framework for Geometric Algebra Operations on Systems of Coordinate Frames of Arbitrary Signature. *Adv. Appl. Clifford Algebr.* **2018**, *28*. [CrossRef]
27. Eid, A.H. A Low-Memory Time-Efficient Implementation of Outermorphisms for Higher-Dimensional Geometric Algebras. *Adv. Appl. Clifford Algebr.* **2020**, *30*, 24. [CrossRef]
28. Breuils, S.; Nozick, V.; Sugimoto, A.; Hitzer, E. Quadric Conformal Geometric Algebra of $\mathbb{R}^{9,6}$. *Adv. Appl. Clifford Algebr.* **2018**, *28*, 35. [CrossRef]
29. Easter, R.B.; Hitzer, E. Double Conformal Geometric Algebra. *Adv. Appl. Clifford Algebr.* **2017**, *27*, 2175–2199. [CrossRef]
30. Easter, R.B.; Hitzer, E. Conic and cyclidic sections in double conformal geometric algebra G8,2 with computing and visualization using Gaalop. *Math. Methods Appl. Sci.* **2019**, *43*, 334–357. [CrossRef]
31. Hitzer, E.; Sangwine, S.J. Foundations of Conic Conformal Geometric Algebra and Compact Versors for Rotation, Translation and Scaling. *Adv. Appl. Clifford Algebr.* **2019**, *29*, 96. [CrossRef]
32. Hrdina, J.; Návrat, A.; Vašík, P. Geometric Algebra for Conics. *Adv. Appl. Clifford Algebr.* **2018**, *28*, 66. [CrossRef]
33. Zamora-Esquivel, J. G 6,3 Geometric Algebra; Description and Implementation. *Adv. Appl. Clifford Algebr.* **2014**, *24*, 493–514. [CrossRef]
34. Sousa, E.V.; Fernandes, L.A.F. TbGAL: A Tensor-Based Library for Geometric Algebra. *Adv. Appl. Clifford Algebr.* **2020**, *30*, 27. [CrossRef]
35. Fernandes, L.A.F. Exploring Lazy Evaluation and Compile-Time Simplifications for Efficient Geometric Algebra Computations. In *Systems, Patterns and Data Engineering with Geometric Calculi*; Springer International Publishing: Berlin/Heidelberg, Germany, 2021; pp. 111–131. [CrossRef]
36. Sangwine, S.J.; Hitzer, E. Clifford Multivector Toolbox (for MATLAB). *Adv. Appl. Clifford Algebr.* **2016**, *27*, 539–558. [CrossRef]
37. Velasco, M.; Zaplana, I.; Dória-Cerezo, A.; Martí, P. Symbolic and User-friendly Geometric Algebra Routines (SUGAR) for Computations in Matlab. *arXiv* **2024**. [CrossRef]
38. Bancila, M. *Template Metaprogramming with C++*; Packt Publishing: Birmingham, UK, 2022.
39. Colapinto, P. Versor: Spatial Computing with Conformal Geometric Algebra. Master's Thesis, University of California, Santa Barbara, CA, USA, 2011. Available online: http://versor.mat.ucsb.edu (accessed on 15 June 2024)
40. Breitner, J. *Lazy Evaluation: From Natural Semantics to a Machine-Checked Compiler Transformation*; KIT Scientific Publishing Karlsruhe Institute of Technology: Karlsruhe, Germany, 2016.
41. Fontijne, D.H.F. *Efficient Implementation of Geometric Algebra*; Universiteit van Amsterdam: Amsterdam, The Nederland, 2007; p. 229.
42. Reed, M. Differential geometric algebra with Leibniz and Grassmann. *Proc. JuliaCon* **2019**, *1*.
43. Fontijne, D. Gaigen 2: A geometric algebra implementation generator. In Proceedings of the 5th International Conference on Generative Programming and Component Engineering. ACM, GPCE06, Portland, OR, USA, 22–26 October 2006. [CrossRef]
44. Wąsowski, A.; Berger, T. *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*; Springer International Publishing: Berlin/Heidelberg, Germany, 2023. [CrossRef]
45. Breuils, S.; Nozick, V.; Fuchs, L. Garamon: A Geometric Algebra Library Generator. *Adv. Appl. Clifford Algebr.* **2019**, *29*, 69. [CrossRef]
46. De Keninck, S. Non-parametric Realtime Rendering of Subspace Objects in Arbitrary Geometric Algebras. In *Lecture Notes in Computer Science*; Springer International Publishing: Berlin/Heidelberg, Germany, 2019; pp. 549–555. [CrossRef]
47. Eid, A.H.A. Optimized Automatic Code Generation for Geometric Algebra Based Algorithms with Ray Tracing Application. Ph.D. Thesis, Faculty of Engineering, Port-Said University, Port Fuad, Egypt, 2010. [CrossRef]
48. Hastings, C. *Hands-on Start to Wolfram Mathematica*, 3rd ed.; Wolfram Media: Champaign, IL, USA, 2020.
49. Eid, A.H.; Soliman, H.Y.; Abuelenin, S.M. Efficient ray-tracing procedure for radio wave propagation modeling using homogeneous geometric algebra. *Electromagnetics* **2020**, *40*, 388–408. [CrossRef]
50. Montoya, F.G.; Eid, A.H. Formulating the geometric foundation of Clarke, Park, and FBD transformations by means of Clifford's geometric algebra. *Math. Methods Appl. Sci.* **2021**, *45*, 4252–4277. [CrossRef]
51. Price, M.J. *C# 12 and.NET 8*, 8th ed.; Packt Publishing: Birmingham, UK, 2023.
52. Hestenes, D. The Genesis of Geometric Algebra: A Personal Retrospective. *Adv. Appl. Clifford Algebr.* **2016**, *27*, 351–379. [CrossRef]
53. Spindler, K. *Abstract Algebra with Applications: Volume 2: Rings and Fields*; Routledge: London, UK, 1993.
54. Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* **1991**, *23*, 5–48. [CrossRef]
55. Steven, L.; Brunton, J.N.K. *Data-Driven Science and Engineering*; Cambridge University Press: Cambridge, UK, 2022.
56. Wei, Y.; Ding, W. *Theory and Computation of Tensors: Multi-Dimensional Arrays*; Academic Press: Cambridge, MA, USA, 2016.
57. Grabmeier, J.; Kaltofen, E.; Weispfenning, V. (Eds.) *Computer Algebra Handbook*; Springer-Verlag GmbH: Berlin, Germany, 2002.
58. Strmečki, D.; Magdalenić, I.; Radosević, D. A Systematic Literature Review on the Application of Ontologies in Automatic Programming. *Int. J. Softw. Eng. Knowl. Eng.* **2018**, *28*, 559–591. [CrossRef]
59. Krysztof Czarnecki, U.E. *Generative Programming: Methods, Tools, and Applications*; Addison-Wesley Professional: Boston, MA, USA, 2000.

60. Lilis, Y.; Savidis, A. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* **2020**, *52*, 1–39. [CrossRef]
61. Štuikys, V.; Damaševičius, R. *Meta-Programming and Model-Driven Meta-Program Development*; Springer: London, UK, 2013. [CrossRef]
62. Sharvit, Y. *Data-Oriented Programming*; Manning Publications Co. LLC.: Shelter Island, NY, USA, 2022; p. 424.
63. Hrdina, J.; Návrat, A.; Vašík, P.; Dorst, L. Projective Geometric Algebra as a Subalgebra of Conformal Geometric algebra. *Adv. Appl. Clifford Algebr.* **2021**, *31*. [CrossRef]
64. Simon, D. *Evolutionary Optimization Algorithms*; Wiley: Hoboken, NJ, USA, 2013.
65. Ashlock, D. (Ed.) *Evolutionary Computation for Modeling and Optimization*; SpringerLink, Springer Science+Business Media, Inc.: New York, NY, USA, 2006.
66. Poli, R.; Langdon, W.B.; McPhee, N.F. *A Field Guide to Genetic Programming*; Lulu Press: Morrisville, NC, USA, 2008; pp. 167–223.
67. Lavor, C. *A Geometric Algebra Invitation to Space-Time Physics, Robotics and Molecular Geometry.*; SpringerBriefs in Mathematics Series; Springer: Cham, Switzerland, 2018.
68. Perwass, C.B.; Farouki, R.T.; Noakes, L. A geometric product formulation for spatial Pythagorean hodograph curves with applications to Hermite interpolation. *Comput. Aided Geom. Des.* **2007**, *24*, 220–237. [CrossRef]
69. Eid, A.H.; Montoya, F.G. A Systematic and Comprehensive Geometric Framework for Multiphase Power Systems Analysis and Computing in Time Domain. *IEEE Access* **2022**, *10*, 132725–132741. [CrossRef]
70. Montoya, F.G.; Baños, R.; Alcayde, A.; Arrabal-Campos, F.M.; Roldán Pérez, J. Geometric algebra framework applied to symmetrical balanced three-phase systems for sinusoidal and non-sinusoidal voltage supply. *Mathematics* **2021**, *9*, 1259. [CrossRef]
71. Montoya, F.G.; De Leon, F.; Arrabal-Campos, F.; Alcayde, A. Determination of instantaneous powers from a novel time-domain parameter identification method of non-linear single-phase circuits. *IEEE Trans. Power Deliv.* **2021**, *37*, 3608–3619. [CrossRef]
72. Uzunović, T.; Montoya, F.G.; Osmanović, A.; Arrabal-Campos, F.M.; Alcayde, A.; Eid, A.H.; Šabanović, A. Combining Real-time Parameter Identification and Robust Control Algorithms for Effective Control of Electrical Machines. In Proceedings of the 2022 International Conference on Electrical Machines (ICEM), Valencia, Spain, 5–8 September 2022; pp. 2391–2396.
73. Eid, A.H.; Montoya, F.G. A geometric procedure for computing differential characteristics of multi-phase electrical signals using geometric algebra. *arXiv* **2022**, arXiv:2208.05917.
74. Montoya, F.G.; Ventura, J.; Arrabal-Campos, F.M.; Alcayde, A.; Eid, A.H. Frequency Generalization via Darboux Bivector and Electrical Curves in Multi-Phase Power Systems. *TechRxiv* **2023**. [CrossRef]
75. Montoya, F.G.; Baños, R.; Alcayde, A.; Arrabal-Campos, F.M.; Roldán-Pérez, J. Geometric algebra applied to multiphase electrical circuits in mixed time–frequency domain by means of hypercomplex hilbert transform. *Mathematics* **2022**, *10*, 1419. [CrossRef]
76. Ventura, J.; Martinez, F.; Manzano-Agugliaro, F.; Návrat, A.; Hrdina, J.; Eid, A.H.; Montoya, F.G. A novel geometric method based on conformal geometric algebra applied to the resection problem in two and three dimensions. *J. Geod.* **2024**, *98*, 47. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.