Garamon: a Geometric Algebra Library Generator

Stéphane Breuils, Vincent Nozick and Laurent Fuchs

Abstract. This paper presents both a recursive scheme to perform Geometric Algebra operations over a prefix tree, and Garamon, a C++ library generator implementing these recursive operations. While for low dimension vector spaces, precomputing all the Geometric Algebra products is an efficient strategy, it fails for higher dimensions where the operation should be computed at run time. This paper describes how a prefix tree can be a support for a recursive formulation of Geometric Algebra operations. This recursive approach presents a much better complexity than the usual run time methods. This paper also details how a prefix tree can represent efficiently the dual of a multivector. These results constitute the foundations for Garamon, a C++ library generator synthesizing efficient C++ / Python libraries implementing Geometric Algebra in both low and higher dimensions, with any arbitrary metric. Garamon takes advantage of the prefix tree formulation to implement Geometric Algebra operations on high dimensions hardly accessible with state-of-the-art software implementations. Garamon is designed to produce easy to install, easy to use, effective and numerically stable libraries. The design of the libraries is based on a data structure using precomputed functions for low dimensions and a smooth transition to the new recursive products for higher dimensions.

Mathematics Subject Classification (2010). Primary 99Z99; Secondary 00A00.

Keywords. Geometric Algebra, Clifford Algebra, Prefix tree.

1. Introduction

Geometric Algebra is a powerful tool to create and manipulate geometric objects. Its theory is more and more investigated in various research fields like physics, mathematics or computer sciences. Since Geometric Algebra presents a strong potential for applied sciences, a significant effort on how it

can be implemented on a computer has also been conducted. These implementations can take the form of libraries, library generators, code generators, packages included into larger systems or specialized programs, each of them dedicated for a specific use. Information about those different systems can be found in some books [9, 22, 18] or in papers dealing with Geometric Algebra implementations [12, 1, 4, 16].

This paper presents Garamon (Geometric Algebra Recursive and Adaptive Monster), a library generator written in C++ programming language and producing specialized Geometric Algebra (GA) libraries also in C++, with a binding in Python. It can be compared to generators producing C++ programming code like Gaigen [14], Gaalop [6], GMac [13, 12] and libraries written in C++ programming language like Gaalet [23], Versor [7] and GluCat [19].

These software implementations differ mainly in the way they represent multivectors and in their optimizing level of the algebra operations. Compared to the linear algebra framework, the fundamental entities of Geometric Algebra, the multivectors, are of a higher dimension and thus require larger data structures. Actually, even if multivectors could be very large (2^d coordinates if the base vector space of the algebra is of dimension d) they are in practice often very sparse. So, to be efficient, a Geometric Algebra implementation may aim at both representing a geometric object with as little information as possible and designing the algebra operators from algorithms that use efficiently this information. In that perspective, different strategies have been conducted. Gaigen [14] generates optimized libraries defined from an algebra specification. Gaalop [6] and Gmac [13, 12] produce optimized code fragments from a description of an algorithm in a domain specific language. Then, the generated code can be integrated into the target program. Gaalet [23] and Versor [7] are using C++ metaprogramming techniques like expression templates to define types representing expressions to be computed at compile time. Thus, expressions are computed only when needed to produce an efficient code. GluCat is following another way using real matrix representation for Clifford algebra [20] and a dedicated version of fast Fourier transform to improve Clifford product. GluCat was benchmarked and found its performance to be similar to CLU [15].

All these approaches present some very interesting properties, however some improvements can be achieved to make them easier to use, to have better memory requirements or to range over wider dimension spaces. These points are the initial motivation to create *Garamon*.

2. Garamon overview and motivations

We propose a new library called *Garamon*. It is a C++ template library generator dedicated to Geometric Algebra. The generated Geometric Algebra libraries are designed to be user friendly and efficient both in term of computation speed as well as memory consumption. The full project is available online*.

The generated libraries are built from a short configuration file describing the targeted algebra. This configuration file specifies the algebra signature, the name of the basis vectors and some optimization options. This file is restricted to the minimum information such it can be filled very easily.

2.1. Efficient

The generated GA libraries handles both "low dimensional" (base vector space of dimension roughly up to 10) and "high dimensional", with a hard-coded limit to dimension 31. The "low dimensional" operations are pre-computed, whereas the "high dimensional" computations run on a new recursive scheme based on a prefix tree multivector representation. This prefix tree representation presents some very efficient runtime optimizations in term of time complexity, as well as the property to encode easily the dual of the considered multivector. The transition from "low dimensional" to "high dimensional" is smooth, such that "high dimensional" GA libraries may still include some pre-computed instructions for some products. Note there exist some applications for such high dimensional spaces. First, [5] is an example of such high dimensional framework. One can also imagine an extension of [5] to deal with cubic surfaces and quartic surfaces.

2.2. User friendly

The generated libraries are dedicated to being very easy to install and to use. They are multi-platform, run and compile with only one dependency, i.e. the header only library `eigen` [17] for hidden vector and matrix manipulations. Any generated library contains its own dedicated installation file (cmake), as well as a dedicated sample code. The generated libraries handle any arbitrary Geometric Algebra signature, such that the user do not have to care about basis change. The embedded basis change takes a special care about numerical stability. Moreover, since all the generated libraries are identified by a namespace, multiple Geometric Algebra libraries can be used together.

3. Notations

Following the state-of-the-art usages of [9] and [22], upper-case bold letters denote blades (blade $\mathbf{A}$) whose grade is higher than 1. Multivectors and k-vectors are denoted with upper-case non-bold letters (multivector A).

*git clone https://git.renater.fr/anonscm/git/garamon/garamon.git

Lower-case bold letters refer to vectors and lower-case non-bold to multivector coordinates. Lower-case and Frakture letters denote multivector expressed over a tree structure. For example, $\mathfrak{a}$ represents a multivector over a tree structure, this notion is presented in the Section 8.1. The k-grade part of a multivector A is denoted by $\langle A \rangle_k$. Finally, the vector space dimension is denoted by 2^d, where d is the number of basis blades $\mathbf{e}_i$ of grade 1.

4. Structure of the paper

This paper is organized as follows. Section 5 introduces the data structure used in Garamon to store the multivector components. Section 6 details how GA operations can be pre-computed in advance for low dimension vector spaces and why this approach does not hold for higher dimensions. In these high dimensions, GA products should be computed at run time, and using trees as a support of GA computation leads to better performance. Section 7 introduces how a prefix tree can be designed to represent GA multivectors and their dual. Section 8 introduces tree traversal methods to access efficiently to the different components of a multivector. Then, Section 9 defines a mapping between the prefix tree approach and the binary tree method defined in [3]. This mapping is used in Section 10 to prove the correctness of recursive formulas of basic vector space operations over a prefix tree, and in Section 11 to define and prove the correctness of recursive formulation of GA products over prefix trees. Section 12 extends these recursive products to dual operations. Section 13 presents a new numerical scheme to enhance the numerical stability of metric dependent products with non-orthogonal metrics. Section 14 presents the technical characteristics of Garamon and Section 15 details our experimental results.

5. Multivector Data Structure

5.1. Multivectors and arrays

For a d dimensional vector space, the potential amount of information that could be stored to represent fundamental elements of linear algebra (vectors and matrices) drastically differs from the information represented in GA (multivectors). For linear algebra, it is of order $\mathcal{O}(d^2)$ whereas it is of order $\mathcal{O}(2^d)$ for GA, often with very sparse data. This difference usually influences their respective implementation. Hence, linear algebra implementations are frequently expressing and storing all the data composing a vector or a matrix (except if they are known to be sparse) whereas GA implementations are mostly trying to only store non-zero elements. One way to implement this constraint is to use a linked list of non-zero elements as in many Geometric Algebra implementations [14, 6, 23, 7].

In this paper, we follow a different approach by storing multivector elements by grade. This concept is motivated by the observation that for many

GA, geometric objects are blades, i.e. multivectors of a single grade. As an example, Figure 1 shows that for Conformal Geometric Algebra (CGA), all geometric objects are blades and only versors may be non-blades multivectors.

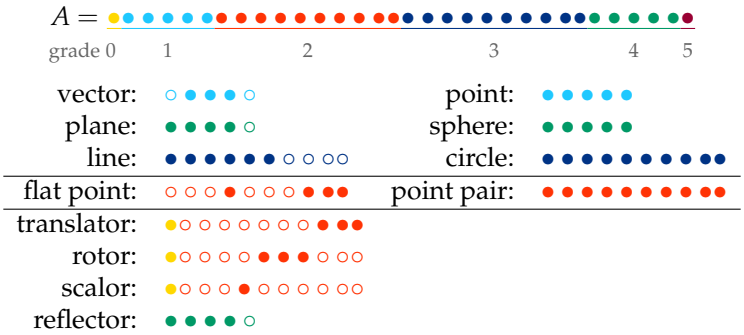


FIGURE 1. This represents how objects of Conformal Geometric Algebra of $\mathbb{R}^3$ tend to be restricted to a single grade.

More specifically, we consider a multivector as a set of arrays, all dedicated to a specific grade. This set contains only arrays related to grades explicitly expressed by the represented multivector, but still, an array may contain some zero values, as depicted in Figure 2. This choice is motivated by the

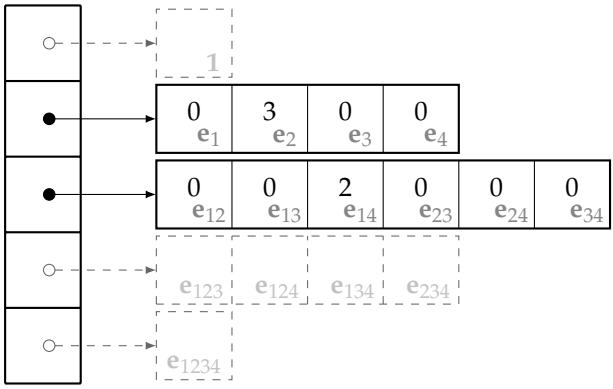


FIGURE 2. Data structure, example with $\mathbf{x} = 3\mathbf{e}_2 + 2\mathbf{e}_{14}$.

fact that most Geometric Algebra entities consist in homogeneous multivectors, i.e. multivectors with elements all having the same grade. In this

situation, the array dedicated to the specific grade of an arbitrary object is likely to be full and thus much more effective than a linked list.

In practice, storing this Geometric Algebra elements as per grade arrays is also motivated by some code optimization using SIMD registers to perform up to eight operations simultaneously on the data. These optimizations are straightforward for vector addition, multiplication with a constant, etc. In that case, even storing some zero often does not affect the computation speed.

Within this framework, a full multivector of dimension 24 (with 2^{24} elements) with coefficients stored with `float32` weights ∼100 MB, and dimension 32 leads to a ∼3 GB full multivector.

Thus, dimension 24 looks like an upper bound for this framework in term of memory usage. Therefore, a `uint32` binary representation of each basis vector defining a basis blade is still acceptable and leads to a hardcoded upper bound of dimension 31. In this binary representation , a 1-bit at a position i indicates the presence of the basis vector $\mathbf{e}_i$ in the basis blade. For example, in a 4-dimensional vector space, the basis blade $\mathbf{B} = \mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_4$ will be represented by the binary number 1101. Note that this way of indexing basis blades is widely used in GA community, see [22, 15, 12].

5.2. More about zeros

As mentioned in section 5.1, GA implementations aim to avoid to store zero data. In practice, GA implementations of products also tend to avoid manipulation of zeros. To be more precise and clearly define the kind of optimisation targeted in this paper, we define four types of zeros that can be encountered in GA operations. Let us consider the following example:

$$C = A \wedge B$$

with

$$A = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 3 & 2 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$
$$\begin{array}{cccccccc} 1 & \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \mathbf{e}_{12} & \mathbf{e}_{13} & \mathbf{e}_{23} & \mathbf{e}_{123} \end{array}$$

$$B = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 4 & 2 & 1 & 0 \\ \hline \end{array}$$
$$\begin{array}{cccccccc} 1 & \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \mathbf{e}_{12} & \mathbf{e}_{13} & \mathbf{e}_{23} & \mathbf{e}_{123} \end{array}$$

$$C = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ \hline \end{array}$$
$$\begin{array}{cccccccc} 1 & \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \mathbf{e}_{12} & \mathbf{e}_{13} & \mathbf{e}_{23} & \mathbf{e}_{123} \end{array}$$

The naive double loop over the elements of A and B used to compute the outer product would encounter different types of zeros. Considering that GA objects usually have limited different grade elements, these zeroes can be listed as:

1. **structural zero**: when an operation leads to zero due to the nature of the product (i.e. $\underbrace{3\mathbf{e}_2}_{A} \wedge \underbrace{4\mathbf{e}_{12}}_{B}$).

2. **object zero**: when A and B are homogeneous GA objects, computing products with elements of grade not related to the object is useless (i.e. $\underbrace{3\mathbf{e}_2}_{A} \wedge \underbrace{0\mathbf{e}_1}_{B}$).
3. **data zero**: when not all the components of grade k are used to express a GA object of grade k (i.e. $\underbrace{0\mathbf{e}_1}_{A} \wedge \underbrace{1\mathbf{e}_{23}}_{B}$).
4. **computational zero**: when an element should theoretically be zero but is numerically non-zero due to numerical errors.

A data storage based on a list will naturally avoid "object zeros" and "data zeros" when a per grade array storage may compute useless "data zeros". In any cases, "computational zeros" are very difficult to avoid. Indeed, a program will hardly identify if a value $x = 10^{-9}$ is an expectable value in the considered problem or a numerical error (like in $\mathtt{x} = \mathtt{0.1f} - \mathtt{0.1L} \simeq 10^{-9}$ in C/C++ language).

In the list above, the most interesting useless product is the "structural zeros" that seems unavoidable. This point is discussed latter on Section 8 and shows how some recursive expressions of the outer products over a prefix tree can intrinsically avoid these useless operations.

6. Products in low dimensional vector spaces

6.1. Per grade products

Considering the per grade data structure defined in section 5.1, a very efficient way to process any product is to pre-compute it. Since the outer, inner and geometric products are distributive over the addition, each "per grade product" can be extracted and computed independently. Let $\langle X \rangle_k$ be the part of the multivector X of grade k, and $D_X = \{\langle X \rangle_i \neq 0\}_{i \in [0,d]}$ be the set of all k-vector $\langle X \rangle_k$ of any grade present in X, where d is the dimension of the vector space. Then, most of the products $\odot$ between the multivectors A and B can be computed by the double loop algorithm as presented in Algorithm 1 (geometric product is a special case). In practice, these two loops are likely to contain only one call, in the case where A and B are homogeneous multivectors.

In low dimensional spaces, each product `product_k`$_a$`_k`$_b$, i.e. a function implementing the considered product between a k_a-vector and a k_b-vector, can be pre-computed, according to the specified GA signature. In this situation, the production of a GA library mostly consists in automatically generating such pre-computed functions. The main specificity of Garamon here is the memory management and some operations, like multivector addition, or multiplication by a scalar, that are performed using parallel computing with SIMD instructions.

Algorithm 1: Per grade loop

```
input : multivectors A and and B,
        a product ⊙ distributive over the addition
w output: multivector: C = A ⊙ B
foreach k-vector ⟨A⟩_{k_a} ∈ D_A do
    foreach k-vector ⟨B⟩_{k_b} ∈ D_B do
        k_c = find_grade(⊙, k_a, k_b)
        ⟨C⟩_{k_c} = product_k_a_k_b(⊙, A, B)
return C
```

6.2. Dual computation

For any full-rank GA signature, the dual multivector computation can be optimized in advance. By definition, the dual of a multivector is given by:

$$A^* = A \cdot \mathbf{I}^{-1} = \frac{A \cdot \widetilde{\mathbf{I}}}{\mathbf{I} \cdot \widetilde{\mathbf{I}}} \tag{6.1}$$

This expression requires the computation of two inner products, a reverse and a scalar division, that can be pre-computed. Due to the symmetry property of the Pascal's triangle, a k-vector A and its dual A^* both have the same number of elements. In the array based data structure of section 5.1, computing the dual of a k-vector thus just consists in changing the "grade label" of the corresponding array from k to $d - k$ (for a vector space of dimension d), permuting some array elements and eventually multiplying them by some constant according to the metric of the algebra. Extending this method to a multivector means to apply it to all non-null blades of the multivector. Concerning the implementation, the dual is merely computed by pre-computing both an array that stores the required permutation for each array and a vector that stores the coefficients to apply to each resulting array. Some of these operations are well suited to SIMD optimization.

6.3. Precomputation, the limits

Every library implementing GA operators and data faces the exponential growth of the subspace dimensions of the graded algebra structure. In practice, this constraint means that the number of pre-computed operations and the number of instructions they contain will grow exponentially with respect to the underlying vector space dimension.

For this reason, the upper bound limit for Gaigen [14] is set to dimension 12. Higher dimensions would produce a source code with unreasonable size. For Versor [7], this limit varies according to the required operations and may range from dimension 6 using massive computations to dimension 10 for very simple operations. The limitation here is related to the required RAM memory during compilation and compilation time. Finally, the upper bound dimension for the table based plugin of Gaalop [6] is dimension 10.

This limitation is due to the use of tables to compute GA products. Figure 1 clearly highlights the exponential behaviour of the memory consumption. The required memory in dimension 10 is around 710 MB, higher dimension would result in a program that exceeds the 1GB limit of the JVM (Java Virtual Machine) required to execute the program.

TABLE 1. Memory consumption of Gaalop, including tables.

Dimension	5	6	7	8	9	10	11
Memory occupation (MB)	1.1	3.7	13	40	142	710	-

Table 2 summarises the vector space upper bound limit for some well-known libraries. It should be noticed that using a GA library with dimension near to the limit can be troublesome, and the "comfortable use" limit is slightly below.

TABLE 2. Usual dimension upper limit.

	max dimension	reason
Gaigen [14]	~ 12	code size
Versor [7]	~ 7	compilation memory and time

7. Geometric Algebra and prefix tree

As presented in section 6.3 as well as in [4], GA products pre-computation is a good strategy for low dimensional spaces, however this approach fails for higher dimensional spaces, due to memory overload or to complexity issue. Indeed, in such situation, GA products should be computed at run time and may suffer from non-optimized algorithmic structure. Breuils et al. [3] detailed how a binary tree can represent efficiently multivector components and leads to an effective recursive formulation of the products used in GA for high dimensions. In the following sections, we introduce a variation of this formulation, using a prefix tree that presents some interesting properties leading to very efficient optimization in recursive GA products. Moreover, this prefix tree formulation also includes a natural dual multivector representation well suited to an efficient dual computation algorithm, particularly useful for high dimensions.

7.1. Multivectors and prefix trees

This section presents the prefix tree structure of the basis blades of a Geometric Algebra. Each basis blade is associated to a node of a prefix tree and the nodes of depth k in the prefix tree correspond to the basis blades of grade k. Thus, the scalar basis blade, denoted by $\mathbf{1}$, is associated to the root node, the vector basis blades are associated to the children of the root node, the

bivector basis blades are associated to the children of those nodes, and so on, as illustrated on Figure 3. By construction of the prefix tree, the index of a basis blade associated to a node is prefixed by the indexes of the basis blades associated with the parent nodes.

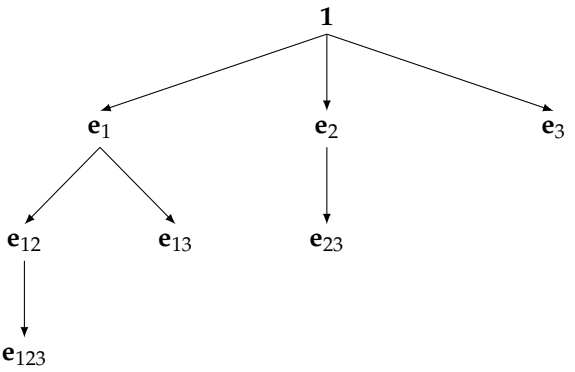


FIGURE 3. Prefix tree structure of the basis blades for a Geometric Algebra whose underlying vector space is of dimension 3.

As in several GA implementations [13, 6, 14], the indexes of the basis blades are represented with a binary label, as illustrated in Figure 4. This binary label is very useful to optimize paths in the prefix tree. More details about these optimizations are given in section 8.3. The binary label of a node is recursively computed using the binary label of its parent node. A node with binary label `u` has its first child binary label computed by:

$$\text{child\_label}(\texttt{u},\texttt{msb}) = \texttt{u} + \texttt{msb} \tag{7.1}$$

where $+$ is the binary addition and `msb` is the binary label of the basis vector "added" to the basis blade by the outer product. So, `msb` contains only a single bit set to 1. Note that this bit set to 1 in `msb` cannot be a bit already set to 1 in `u`, otherwise the parent node and its child would have the same grade.

The contribution of `msb` is the most significant bit of child_label(`label`,`msb`), i.e. the first bit to 1 encountered while reading the binary label from the left, which corresponds to the position of the 1-bit of `msb`.

Moreover, the labels of the siblings of a child (with a direct common parent) can be easily computed by means of left-shifting (i.e. multiplying by two) of `msb`. This is illustrated in Figure 5.

It can be noted that this tree representation is not well suited for an efficient

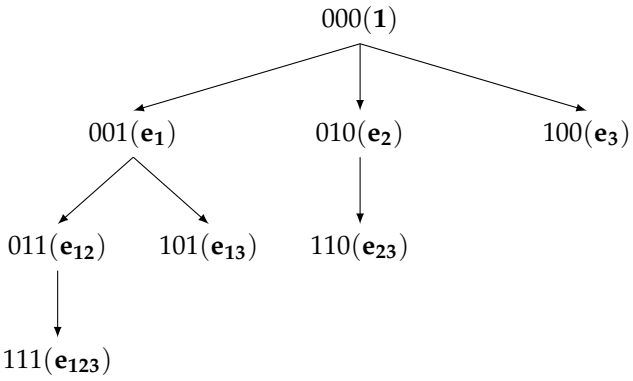


FIGURE 4. Binary labelling of the prefix tree nodes in the case of Geometric Algebra whose underlying vector space is of dimension 3.

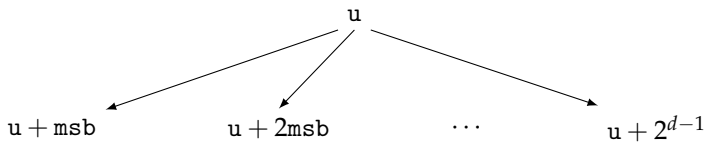


FIGURE 5. Binary labelling of the siblings of a child node.

data storage due to the difficulty to cut useless parts of the tree. Therefore, Garamon uses only the data structure defined in section 5.1 and includes a mapping from the tree representation to the data structure of section 5.1. This mapping consists in two pre-computed look-up tables. The first one extracts the grade of a node according to its binary label and the second defines its position, again according to this binary label.

Finally, it is noteworthy that, due to their structure and the way they are constructed, the prefix trees presented in this section are also binomial trees [25]. The use of "prefix tree" denomination is used here to stress the link between nodes labelling and nodes grade.

7.2. Dual and prefix tree

In the prefix tree structure of the basis blades, the basis blades of grade k are associated to the nodes of depth k. So, the pseudo-scalar is associated to the deepest node of the prefix tree. Looking at the tree "upside-down", the node corresponding to the pseudo-scalar corresponds to the root of the tree and can be associated to the dual of the pseudo-scalar. In the same way, the nodes just over the pseudo-scalar correspond to the dual basis blades of the vector basis blades if their sign is correctly set. We can go further and

reach the scalar basis blade which is now associated to the deepest node of the "upside-down" tree and hence corresponds to the dual of the pseudo-scalar (eventually up to a sign change). In short, obtaining the dual basis blades just consists in reading "upside-down" the prefix tree structure of the basis blades, eventually with some sign changes and some metric coefficient update, as shown in Figure 6. With this model, the dual basis blades also have a prefix tree structure. Moreover, the binary labels of the dual prefix tree can be computed by subtracting the binary label `msb` of the dual child to the binary label of the dual parent node. Hence we have a dual version of Equation (7.1):

$$\text{dual\_child\_label}(\mathtt{u},\mathtt{msb}) = \mathtt{u} - \mathtt{msb} \tag{7.2}$$

where `u` is the binary label of the dual parent node. Note that the binary label of the dual prefix tree root is now the binary label $(\mathtt{1} << \mathtt{d}) - \mathtt{1}$ where $<<$ is the left shift operator shifting on the left the digits of a label. To take into account sign and coefficient changes, both the sign and the metric coefficients can be stored in a single array of size 2^d where d is the dimension of the underlying vector space of the algebra.

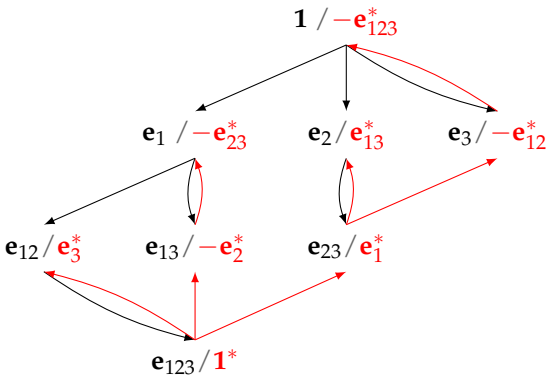


FIGURE 6. Primal form of a tree data structure of an Euclidean 3 dimensional vector space, and its dual counterpart in red

The dual and primal prefix tree representations are the support of an efficient recursive expression of GA products, coupled with the per grade data structure of section 5.1. As for the primal prefix tree, the dual prefix tree is just a support for the recursive products, the data are staying stored into the "per grade" data structure of section 5.1. The main goal of this dual prefix tree is to compute some product between dual multivectors without computing the costly multivector dualization.

8. Prefix tree traversal algorithms

Given a geometric algebra, the prefix tree structure of its basis blades defined in Section 7 can be used to represent its multivectors and to efficiently compute some products. The idea is to associate the multivector coefficients to the nodes of the prefix tree. Then, the different products of the algebra are defined recursively over the prefix tree representation. The basis blades of the geometric algebra also have a binary tree structure and some recursive products over binary trees, defined by Fuchs and Théry [16] and developed by Breuils et al. [3]. These products (outer product, inner products, contractions, ...) are computed in $\mathcal{O}(3^d)$ time complexity, where d is the dimension of the underlying vector space of the algebra. More precisely, in the worst case, the number of elementary products performed by the recursive product is:

$$\sum_{i=1}^{d} 3^i = \frac{3}{2}\left(3^d - 1\right) \tag{8.1}$$

This result does not hold for the geometric product computed with a $\mathcal{O}(4^d)$ complexity.

In [3], Breuils et al. already showed that the recursive approach achieves better time complexity than the state-of-the-art methods which are in $\mathcal{O}(d \times 4^d)$. This section both defines how multivectors are represented by prefix trees and how traversal of a prefix tree can be optimized with respect to the grade of the considered multivector.

8.1. Multivector basis vectors and prefix tree

ism. Namely, we want to express the operations of the algebra using this prefix tree representation. This requires expressing two multivectors with the prefix tree representation. Just like [4] expressed a multivector as a binary tree with recursive formalism, we seek for an expression of the Geometric Algebra with prefix tree. We consider a multivector A in $\mathbb{G}_{p,q}$ where $p+q=d$, we note a component of A at index $\mathtt{u}$ in the prefix tree formulation as $\mathfrak{a}_\mathtt{u}$. The children of $\mathfrak{a}_\mathtt{u}$ in the prefix tree formulation are shown in Figure 7. Let us first introduce how to express a multivector over a prefix tree. Given a multivector of a specific geometric algebra whose underlying vector space is of dimension d, we note $\mathfrak{a}_\mathtt{u}$ the node labelled by $\mathtt{u}$ in a prefix tree. Like in Figure 5, the children of $\mathfrak{a}_\mathtt{u}$ in the prefix tree can be found as depicted in Figure 7.

For the following parts, we define a notation to manipulate prefix trees. A prefix tree with root $\mathfrak{a}_\mathtt{u}$ and children $\mathfrak{a}_\mathtt{u}$, $\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}$, ..., $\mathfrak{a}_{\mathtt{u}+2^{d-1}}$ is noted by:

$$\left(\mathfrak{a}_\mathtt{u}, \left(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}}\right)\right) \tag{8.2}$$

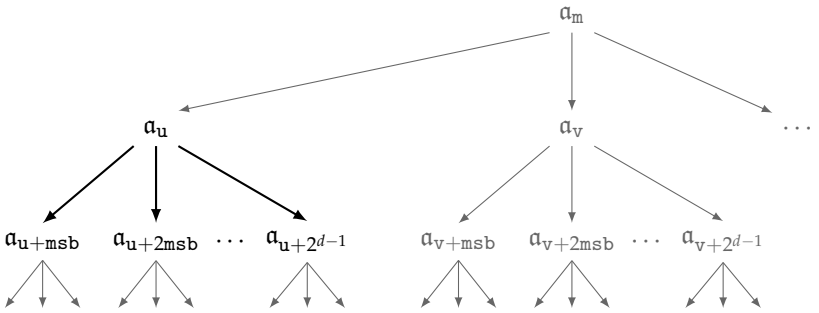


FIGURE 7. Multivector prefix tree representation and link from a node to its children.

A prefix tree defining a multivector a_u must be interpreted as an algebra element. This is achieved by the following interpretation.

Definition 8.1. Interpretation
Let a_u be a node of the prefix tree and $\mathbf{e}_i$, $i \in 1, \cdots, d$, be the basis vector associated with msb. The link between a node a_u and its direct children can be interpreted as Geometric Algebra operations, as follows:

$$a_u + \mathbf{e}_i \wedge a_{u+msb} + \mathbf{e}_{i+1} \wedge a_{u+2msb} + \cdots + \mathbf{e}_d \wedge a_{u+2^{d-1}} \tag{8.3}$$

Repeating recursively this interpretation for all nodes of the prefix tree fully describes the corresponding multivector and clearly associates a basis blade to each node. As an example, a multivector of an algebra $\mathbb{G}_{p,q}$ with $d = p + q = 3$, at first recursion depth, is noted by:

$$\Big(a_{000}, (a_{001}, a_{010}, a_{100})\Big) \tag{8.4}$$

The development after $d = 3$ recursive steps yields:

$$\Big(a_{000}, \Big(a_{001}, \big(a_{011}, (a_{111})\big), a_{101}\Big), \Big(a_{010}, \big(a_{110}\big)\Big), a_{100}\Big)\Big) \tag{8.5}$$

Using the interpretation of the tree given in Equation (8.3) results in:

$$\begin{aligned} & a_{000} + \mathbf{e}_1 \wedge \Big(a_{001} + \mathbf{e}_2 \wedge \big(a_{011} + \mathbf{e}_3 \wedge (a_{111})\big) + \mathbf{e}_3 \wedge a_{101}\Big) \\ & + \mathbf{e}_2 \wedge \Big(a_{010} + \mathbf{e}_3 \wedge a_{110}\Big) + \mathbf{e}_3 \wedge a_{100} \\ = \; & a_{000} + a_{001}\mathbf{e}_1 + a_{011}\mathbf{e}_1 \wedge \mathbf{e}_2 + a_{111}\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 + a_{101}\mathbf{e}_1 \wedge \mathbf{e}_3 + a_{010}\mathbf{e}_2 \\ & + a_{110}\mathbf{e}_2 \wedge \mathbf{e}_3 + a_{100}\mathbf{e}_3 \end{aligned} \tag{8.6}$$

This corresponds to a general multivector in a 3-dimensional vector space.

8.2. Prefix tree traversal algorithm

This section explains the depth-first traversal of a prefix tree knowing the maximum grade of the multivector it represents. Let us start with a non-optimized approach presented in Algorithm 2.

Algorithm 2: Recursive deep-first traversal of a multivector A whose maximal grade is k_a

```
1 Function traverse
      Input: A: the multivector to be traversed,
             k_a: the grade of the multivector.
             label_a: the recursive position on each tree.
2     if grade(label_a) == k_a then                    // end of recursion
3         return
4     else                                             // recursive calls
5         msb_a = labelToMsb(label_a)
6         foreach msb < 2^d do
7             label = label_a + msb
8             traverse(A, k_a, label_a + msb)
```

In this algorithm, `labelToMsb(label)` computes `msb`, the most significant bit from the considered label, i.e. the first `1` encountered in the binary word `label` when reading from left to right.

All nodes corresponding to a basis blade whose grade is smaller or equal than the grade of the multivector are traversed, when not all those basis blades are actually used in the multivector. This traversal leads to a lot of useless recursive calls. To avoid them, we propose an improvement of this algorithm consisting in ignoring each branch that never reaches the grade of the considered multivector. This method uses `msb`.

8.3. Bounds for node labels

The most significant bit, `msb`, is a valuable information leading to an effective optimization in the tree traversal. The key idea of this optimization is based on the following proposition.

Proposition 8.2. *Given a node of the prefix tree with label* `label`*, there exists at least one path from the considered node to a node whose grade is k if the following condition, computed in constant time, is satisfied:*

$$\texttt{label} + \texttt{msb}(2^{k-\texttt{grade(label)}} - 1) < 2^d \tag{8.7}$$

where $\texttt{msb}(2^{k-\texttt{grade(label)}} - 1)$ *is the most significant bit of* $2^{k-\texttt{grade(label)}} - 1$*,* $\texttt{grade}(\texttt{label})$ *is the number of* 1*-bit in* `label` *(i.e. Hamming weight) and d is the dimension of the underlying vector space of the algebra.*

Proof. Here, the key point is to prove that the number of recursive calls from a given node to a node of grade k can be lower bounded and that the computation of this lower bound can be performed in constant time.

By definition, at each recursive depth, the grade is incremented. Furthermore, if $\texttt{label} + \texttt{msb}$ corresponds to the first child of the current node. After two recursive calls, the label of the new node is lower bounded by (left-most child):

$$\texttt{label} + \texttt{msb} + 2\texttt{msb}, \tag{8.8}$$

the grade is increased and is now $\texttt{grade}(\texttt{label}) + 2$. The index of the last traversed vector is simply $2\texttt{msb}$. After three recursive calls, this label is lower bounded by

$$\texttt{label} + \texttt{msb} + 2\texttt{msb} + 4\texttt{msb} \tag{8.9}$$

and the grade is $\texttt{grade}(\texttt{label}) + 3$. The index of the last traversed vector is $4\texttt{msb}$. The general law that we establish for the lower bound of the label of the children after n recursive calls is:

$$\texttt{label} + \sum_{i=0}^{n-1} 2^i \texttt{msb} \tag{8.10}$$

This can be also rewritten as:

$$\texttt{label} + \texttt{msb} \sum_{i=0}^{n-1} 2^i, \text{ with } n \in [1, d]. \tag{8.11}$$

And the general law for the grade is $\texttt{grade}(\texttt{label}) + n$. For the index of the last traversed vector the general law is $2^{n-1}\texttt{msb}$. Let us prove it by induction. The base case holds since after one recursive call the child of the node has label:

$$\texttt{label} + \texttt{msb} = \texttt{label} + \texttt{msb} \sum_{i=0}^{0} 2^i \tag{8.12}$$

This corresponds to the first child thus to a lower bound. The grade of the label is $\texttt{grade}(\texttt{label}) + 1$ and the index of the last traversed vector is $\texttt{msb}$, as seen in the definition.

Let us assume that the Formula (8.11) holds for a number of recursive calls noted m. Thus, the label of the children of the current node is lower bounded by:

$$\texttt{label} + \texttt{msb} \sum_{i=0}^{m-1} 2^i \tag{8.13}$$

By definition of the child of a node, the last traversed vector is $2^m\texttt{msb}$. Thus, the left-most child of the node after m recursive traversals is defined as:

$$\texttt{label} + \texttt{msb} \sum_{i=0}^{m-1} 2^i + 2^m\texttt{msb} \tag{8.14}$$

This can be rewritten as:

$$\texttt{label} + \texttt{msb} \sum_{i=0}^{(m+1)-1} 2^i \tag{8.15}$$

As the grade after m recursive calls is $\texttt{grade}(\texttt{label}) + m$ and we computed the child of this label thus the grade was incremented, and the grade of the leading node is $\texttt{grade}(\texttt{label}) + m + 1$.

Thus, the Formula 8.11 holds for $m + 1$ recursive calls. Finally, by induction, this formula holds for any number of recursive calls.

Furthermore, the term $\sum_{i=0}^{(m+1)-1} 2^i$ is known to be a geometric series whose first term is 1 and its common ratio is 2. The general formula of such geometric series is given as:

$$\sum_{i=0}^{(m+1)-1} 2^i = \frac{1 - 2^m}{1 - 2} = 2^m - 1 \tag{8.16}$$

Finally, the Formula 8.11 can be rewritten as:

$$\texttt{label} + \texttt{msb}(2^m - 1) \tag{8.17}$$

Note that this expression can be computed in a constant time $\mathcal{O}(1)$. Furthermore, we know the upper bound of the labels when the underlying vector space of the algebra is of dimension d. This is the label $11\cdots1$ associated to the pseudo-scalar. Thus, when the underlying vector space of the algebra is of dimension d, all labels are upper bounded by the label `label_max`:

$$\texttt{label\_max} = 2^d \tag{8.18}$$

This means that if the lower bound $\texttt{label} + \texttt{msb}(2^n - 1)$ exceeds `label_max`, then this is not a label for the basis blades of the algebra, meaning that we have a constant time function to know whether after n recursive calls, there exists a child of the current node in the prefix tree representing a multivector. Moreover, it requires exactly $k - \texttt{grade}(\texttt{label})$ recursive calls to reach a node corresponding to a multivector of grade k. This latter result along with the upper bound of Formula (8.18) and the lower bound Formula (8.17) yields to the targeted following inequality:

$$\texttt{label} + \texttt{msb}(2^{k - grade(\texttt{label})} - 1) < 2^d \tag{8.19}$$

□

Formula (8.19) is a way to know whether there exists a reachable child of a node `label` with grade k.

This leads to its associated function $\texttt{gradeKReachable}(k, \texttt{label}, \texttt{msb})$, defined in Algorithm 3.

With Algorithm 3, some branches of the trees can be left unvisited, according

Algorithm 3: check whether it exists one child of the node `label` whose grade is k.

```
1 Function gradeKReachable
    Input: label: the recursive position
           msb: a label of the last traversed vector
           k: the considered grade.
2   labelChildK ← label + msb(2^{k−grade(label)} − 1)
3   return labelChildK < 2^d
```

to a formula that can be computed in constant time. Indeed, the computation of $2^{k-grade(\texttt{label})} - 1$ only requires bit shifting (constant time), one integer multiplication and three additions. And when the dimension increases, this number of operations remains the same thus the algorithm is constant time. Finally, this decision only depends on grades and label, and does not require any memory allocation, thus is also constant in terms of memory complexity.

8.4. Optimized prefix tree traversal algorithm

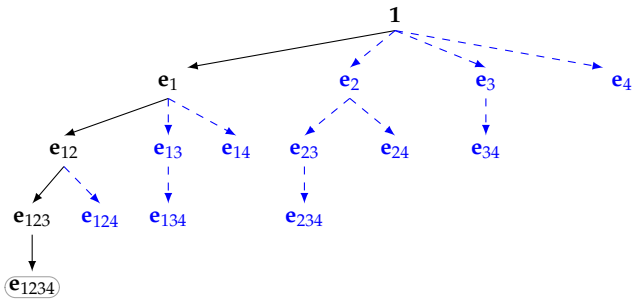
The previous algorithm is the base to define an improved version of the recursive traversal defined in Algorithm 2. The resulting pseudo-code is shown in Algorithm 4. The useless visits are avoided with the grade test

Algorithm 4: Recursive traversing of a multivector A whose maximal grade is k_a

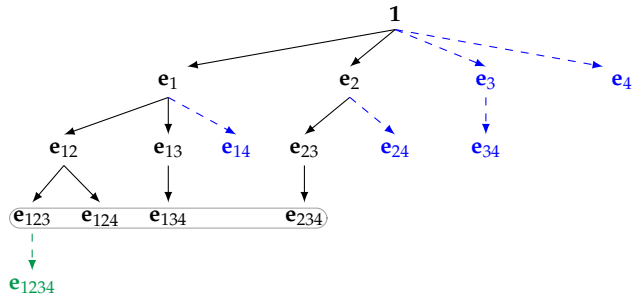
```
1 Function traverse
    Input: A: the multivector to be traversed,
           k_a: the grade of the multivector.
           label_a: the recursive position on each tree.
2   if grade(label_a) == k_a then                    // end of recursion
3     return
4   else                                             // recursive calls
5     msb_a = labelToMsb(label_a)
6     foreach msb such that gradeKReachable(k_a,msb)== true
        do
7       label = label_a + msb
8       traverse(A,k_a,label_a + msb)
```

in Algorithm 4, line 6. Equipped with this new algorithm, it is now possible to traverse the trees as shown in black on Figure 8. This algorithm can result in high improvement of runtime performance. Indeed, the number of useless recursive calls grows exponentially as the dimension grows. Figure 8

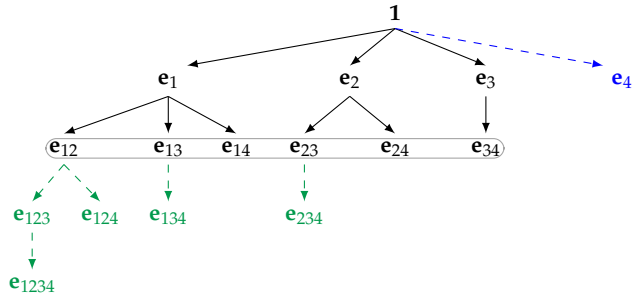
depicts these kind of situations where some nodes (in green) are not visited since their corresponding grade is higher than the targeted grade. Some others are (in blue) are avoided with the test loop in line 6.



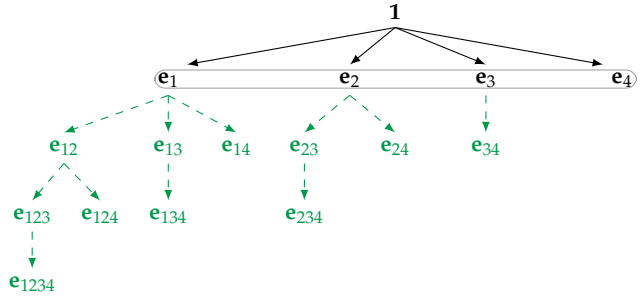
(A)



(B)



(C)



(D)

FIGURE 8. Tree structure for some resulting multivectors of grade 4 (A), grade 3 (B), grade 2 (C), grade 1 (D) in a 4-dimensional vector space. Useless branches are depicted in green dashed arrows above the targeted multivector and in blue below. The targeted nodes are surrounded by a black rectangle.

This section defined the prefix tree both in terms of structure and traversal algorithms. The next sections define the link between the prefix tree and the binary tree, and the recursive products with their corresponding pseudo-code.

9. Mapping between prefix tree and binary tree

The definition of each recursive product requires a proof of correctness. The proof can be given using Definition 8.1 for some recursive definitions. However, the proof using this definition becomes too cumbersome and error-prone for some recursive products. An easier alternative is to define a mapping between the prefix tree and the binary tree to establish an equivalence between them. The proof of the correctness of the recursive algorithms over the binary tree given in [3] will imply the correctness of the corresponding algorithms over the prefix tree. This section defines this mapping.

Definition 8.1 and formula 8.2 are the support to define a mapping between the prefix tree and the binary tree frameworks defined bellow.

Definition 9.1. Mapping prefix tree and binary tree
Let ψ be the mapping associating a prefix tree multivector

$$\left(\mathfrak{a}_{\mathtt{u}}, \left(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}}\right)\right) \tag{9.1}$$

to its counterpart in the binary tree framework [3]:

$$\left(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \left(\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \left(\cdots \left(\mathfrak{a}_{\mathtt{u}+2^{d-1}}, \mathfrak{a}_{\mathtt{u}}\right)\right)\right)\right)_b \tag{9.2}$$

Proposition 9.2.
The interpretation of the binary tree given in Equation 9.2 *results in interpretation of Equation* 8.3.

Proof. Let us consider the pair:

$$\left(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}\right)_b \tag{9.3}$$

where

$$\mathfrak{a} = \left(\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \left(\cdots \left(\mathfrak{a}_{\mathtt{u}+2^{d-1}}, \mathfrak{a}_{\mathtt{u}}\right) \cdots\right)\right)_b \tag{9.4}$$

Assumption of Definition 8.1 denotes that the basis vector associated with $\mathtt{msb}$ is $\mathbf{e}_i$, with $i \in 1, \cdots, d$. The interpretation of Section 3 of [16] means that the pair composed of $\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}$ as a left sub-tree can be written as:

$$\mathbf{e}_i \wedge \mathfrak{a}_{\mathtt{u}+\mathtt{msb}} + \mathfrak{a} \tag{9.5}$$

By reiterating the same computation for the nested pairs in $\mathfrak{a}$ yields:

$$\mathbf{e}_i \wedge \mathfrak{a}_{\mathtt{u}+\mathtt{msb}} + \mathbf{e}_{i+1} \wedge \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}} + \cdots + \mathbf{e}_d \wedge \mathfrak{a}_{\mathtt{u}+2^{d-1}} + \mathfrak{a}_{\mathtt{u}} \tag{9.6}$$

This corresponds to Equation 8.3. □

10. Vector space operations

10.1. Addition and scalar multiplication

In order to define the recursive Geometric Algebra products over the prefix tree structure, we must define and prove the correctness of the vector space operations, namely addition and scalar multiplication. We start with the definition of the recursive formula of the addition between two prefix tree multivector.

Proposition 10.1. *Let us consider two multivectors A and B whose recursive construction are respectively:*

$$\Big(\mathfrak{a}_{\mathtt{u}}, \big(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}}\big)\Big), \tag{10.1}$$

and

$$\Big(\mathfrak{b}_{\mathtt{u}}, \big(\mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{b}_{\mathtt{u}+2^{d-1}}\big)\Big). \tag{10.2}$$

The recursive construction of the addition $C = A + B$ can be computed as:

$$\begin{aligned} & \Big(\mathfrak{c}_{\mathtt{u}}, \big(\mathfrak{c}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{c}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{c}_{\mathtt{u}+2^{d-1}}\big)\Big) \\ = \; & \Big(\mathfrak{a}_{\mathtt{u}}, \big(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}}\big)\Big) + \Big(\mathfrak{b}_{\mathtt{u}}, \big(\mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{b}_{\mathtt{u}+2^{d-1}}\big)\Big) \\ = \; & \Big(\mathfrak{a}_{\mathtt{u}} + \mathfrak{b}_{\mathtt{u}}, \big(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}} + \mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}} + \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}} + \mathfrak{b}_{\mathtt{u}+2^{d-1}}\big)\Big) \end{aligned} \tag{10.3}$$

Proof. We use the interpretation of the prefix tree defined in Equation 8.3. The operation to be performed is:

$$\begin{aligned} & \mathfrak{a}_{\mathtt{u}} + \mathbf{e}_i \wedge \mathfrak{a}_{\mathtt{u}+\mathtt{msb}} + \mathbf{e}_{i+1} \wedge \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}} + \cdots + \mathbf{e}_d \wedge \mathfrak{a}_{\mathtt{u}+2^{d-1}} + \mathfrak{b}_{\mathtt{u}} + \mathbf{e}_i \wedge \mathfrak{b}_{\mathtt{u}+\mathtt{msb}} \\ + \; & \mathbf{e}_{i+1} \wedge \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}} + \cdots + \mathbf{e}_d \wedge \mathfrak{b}_{\mathtt{u}+2^{d-1}} \end{aligned} \tag{10.4}$$

The distributive property of the outer product yields:

$$\begin{aligned} & \mathfrak{a}_{\mathtt{u}} + \mathfrak{b}_{\mathtt{u}} + \mathbf{e}_i \wedge (\mathfrak{a}_{\mathtt{u}+\mathtt{msb}} + \mathfrak{b}_{\mathtt{u}+\mathtt{msb}}) + \mathbf{e}_{i+1} \wedge (\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}} + \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}) + \cdots \\ + \; & \mathbf{e}_d \wedge (\mathfrak{a}_{\mathtt{u}+2^{d-1}} + \mathbf{e}_i \wedge \mathfrak{b}_{\mathtt{u}+\mathtt{msb}} + \mathbf{e}_{i+1} \wedge \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}} + \cdots + \mathbf{e}_d \wedge \mathfrak{b}_{\mathtt{u}+2^{d-1}} \end{aligned} \tag{10.5}$$

Finally, by identification, the above formula is the interpretation of the prefix tree multivector:

$$\Big(\mathfrak{a}_{\mathtt{u}} + \mathfrak{b}_{\mathtt{u}}, \big(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}} + \mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}} + \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}} + \mathfrak{b}_{\mathtt{u}+2^{d-1}}\big)\Big) \tag{10.6}$$

□

Proposition 10.2. *The recursive construction of the multiplication of one multivector A by a scalar $\lambda \in \mathbb{R}$ can be computed as $C = \lambda A$:*

$$\begin{aligned} & \Big(\mathfrak{c}_{\mathtt{u}}, \big(\mathfrak{c}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{c}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{c}_{\mathtt{u}+2^{d-1}}\big)\Big) \\ = \; & \lambda\Big(\mathfrak{a}_{\mathtt{u}}, \big(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}}\big)\Big) \\ = \; & \Big(\lambda\mathfrak{a}_{\mathtt{u}}, \big(\lambda\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \lambda\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \lambda\mathfrak{a}_{\mathtt{u}+2^{d-1}}\big)\Big) \end{aligned} \tag{10.7}$$

Proof. Once more, we use the interpretation of the prefix tree defined in Equation 8.3. The operation to be performed is:

$$\lambda\Big(\mathfrak{a}_{\mathtt{u}} + \mathbf{e}_i \wedge \mathfrak{a}_{\mathtt{u+msb}} + \mathbf{e}_{i+1} \wedge \mathfrak{a}_{\mathtt{u+2msb}} + \cdots + \mathbf{e}_d \wedge \mathfrak{a}_{\mathtt{u}+2^{d-1}}\Big) \tag{10.8}$$

The distributive property of both the scalar multiplication and the outer product yields:

$$\lambda\mathfrak{a}_{\mathtt{u}} + \mathbf{e}_i \wedge \lambda\mathfrak{a}_{\mathtt{u+msb}} + \mathbf{e}_{i+1} \wedge \lambda\mathfrak{a}_{\mathtt{u+2msb}} + \cdots + \mathbf{e}_d \wedge \lambda\mathfrak{a}_{\mathtt{u}+2^{d-1}} \tag{10.9}$$

Thus, by identification, this results in:

$$\Big(\lambda\mathfrak{a}_{\mathtt{u}}, (\lambda\mathfrak{a}_{\mathtt{u+msb}}, \lambda\mathfrak{a}_{\mathtt{u+2msb}}, \cdots, \lambda\mathfrak{a}_{\mathtt{u}+2^{d-1}})\Big) \tag{10.10}$$

□

10.2. Anti-commutativity recursive operator

In order to efficiently compute permutation required for some Geometric Algebra operators, we define the anti-commutativity operator denoted as an overline (e.g. multivector $\overline{A}$). This operator is recursively defined over the prefix tree. It is complicated to prove the correctness of the proposed recursive algorithms using the interpretation of the prefix tree. An easier alternative is to use the mapping between the prefix tree and the binary tree to establish an equivalence between them. The proof of the correctness of the recursive algorithms over the binary tree given in [3] will imply the correctness of the corresponding algorithms over the prefix tree.

Proposition 10.3. *Given a multivector A whose recursive construction is:*

$$\Big(\mathfrak{a}_{\mathtt{u}}, (\mathfrak{a}_{\mathtt{u+msb}}, \mathfrak{a}_{\mathtt{u+2msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}})\Big), \tag{10.11}$$

the recursive construction of the anticommutativity of this multivector $C = \overline{A}$ can be computed as:

$$\begin{aligned} & \Big(\mathfrak{c}_{\mathtt{u}}, (\mathfrak{c}_{\mathtt{u+msb}}, \mathfrak{c}_{\mathtt{u+2msb}}, \cdots, \mathfrak{c}_{\mathtt{u}+2^{d-1}})\Big) \\ = & \Big(\overline{\mathfrak{a}_{\mathtt{u}}}, (\overline{\mathfrak{a}_{\mathtt{u+msb}}}, \overline{\mathfrak{a}_{\mathtt{u+2msb}}}, \cdots, \overline{\mathfrak{a}_{\mathtt{u}+2^{d-1}}})\Big) \\ = & \Big(\overline{\mathfrak{a}_{\mathtt{u}}}, (-\overline{\mathfrak{a}_{\mathtt{u+msb}}}, -\overline{\mathfrak{a}_{\mathtt{u+2msb}}}, \cdots, -\overline{\mathfrak{a}_{\mathtt{u}+2^{d-1}}})\Big) \end{aligned} \tag{10.12}$$

Proof. The method followed here consists in proving that the commutative diagram of the anti-commutative operator shown below holds.

$$\begin{array}{ccc} \Big(\mathfrak{c}_{\mathtt{u}}, (\mathfrak{c}_{\mathtt{u+msb}}, \cdots, \mathfrak{c}_{\mathtt{u}+2^{d-1}})\Big) & \xrightarrow{\ \psi\ } & \Big(\mathfrak{c}_{\mathtt{u+msb}}, \Big(\mathfrak{c}_{\mathtt{u+2msb}}, \big(\cdots(\mathfrak{c}_{\mathtt{u}+2^{d-1}}, \mathfrak{c}_{\mathtt{u}})\big)\Big)\Big)_b \\ \Big\downarrow \overline{\cdot} & & \Big\downarrow \overline{\cdot} \\ \Big(\overline{\mathfrak{c}_{\mathtt{u}}}, (\overline{\mathfrak{c}_{\mathtt{u+msb}}}, \cdots, \overline{\mathfrak{c}_{\mathtt{u}+2^{d-1}}})\Big) & \xrightarrow{\ \psi\ } & \Big(\overline{\mathfrak{c}_{\mathtt{u+msb}}}, \Big(\overline{\mathfrak{c}_{\mathtt{u+2msb}}}, \big(\cdots(\overline{\mathfrak{c}_{\mathtt{u}+2^{d-1}}}, \overline{\mathfrak{c}_{\mathtt{u}}})\big)\Big)\Big)_b \end{array} \tag{10.13}$$

On the one hand, mapping the prefix tree $\mathfrak{c}$ of Equation (10.12) in the binary tree using ψ of Equation (9.2) results in:

$$\begin{aligned} & \left(\mathfrak{c}_{\mathtt{u}+\mathtt{msb}}, \left(\mathfrak{c}_{\mathtt{u}+2\mathtt{msb}}, \left(\cdots\left(\mathfrak{c}_{\mathtt{u}+2^{d-1}}, \mathfrak{c}_{\mathtt{u}}\right)\right)\right)\right)_b \\ = & \left(-\overline{\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}}, \left(-\overline{\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}}, \left(\cdots\left(-\overline{\mathfrak{a}_{\mathtt{u}+2^{d-1}}}, \overline{\mathfrak{a}_{\mathtt{u}}}\right)\right)\right)\right)_b \end{aligned} \tag{10.14}$$

On the other hand, the recursive formula of the anti-commutativity defined in [16] as:

$$\left(\mathfrak{c}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{c}_{\mathtt{u}}\right) = \left(-\overline{\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}}, \overline{\mathfrak{a}_{\mathtt{u}}}\right)_b \tag{10.15}$$

After developing this formula at one recursion depth, this formula becomes:

$$\left(\mathfrak{c}_{\mathtt{u}+\mathtt{msb}}, \left(\mathfrak{c}_{\mathtt{u}+2\mathtt{msb}}, \mathfrak{c}_{\mathtt{u}}\right)\right)_b = \left(-\overline{\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}}, -\overline{\left(\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \overline{\mathfrak{a}_{\mathtt{u}}}\right)}\right)_b \tag{10.16}$$

After further developing this formula, we find:

$$\begin{aligned} & \left(\mathfrak{c}_{\mathtt{u}+\mathtt{msb}}, \left(\mathfrak{c}_{\mathtt{u}+2\mathtt{msb}}, \left(\cdots\left(\mathfrak{c}_{\mathtt{u}+2^{d-1}}, \mathfrak{c}_{\mathtt{u}}\right)\right)\right)\right)_b \\ = & \left(-\overline{\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}}, \left(-\overline{\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}}, \left(\cdots - \overline{\mathfrak{a}_{\mathtt{u}+2^{d-1}}, \overline{\mathfrak{a}_{\mathtt{u}}}}\right)\right)\right)_b \end{aligned} \tag{10.17}$$

Equations (10.17) and (10.14) are equivalent, thus the commutative diagram holds. □

11. Products in high dimensional vector space

11.1. Recursive Outer product

This section presents the recursive formulation of the outer product over a prefix tree. Let us consider the product $C = A \wedge B$, where the maximum grades of A, B, C are respectively k_a, k_b, k_c.

Proposition 11.1. *Given two multivectors A and B whose recursive constructions are respectively:*

$$\left(\mathfrak{a}_{\mathtt{u}}, \left(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}}\right)\right), \tag{11.1}$$

and

$$\left(\mathfrak{b}_{\mathtt{u}}, \left(\mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{b}_{\mathtt{u}+2^{d-1}}\right)\right). \tag{11.2}$$

the recursive construction of the outer product $C = A \wedge B$ can be computed as:

$$\begin{aligned}
& \Big(\mathfrak{c}_{\mathfrak{u}}, \big(\mathfrak{c}_{\mathfrak{u}+\mathtt{msb}}, \mathfrak{c}_{\mathfrak{u}+2\mathtt{msb}}, \cdots, \mathfrak{c}_{\mathfrak{u}+2^{d-1}}\big)\Big) \\
= \quad & \Big(\mathfrak{a}_{\mathfrak{u}}, \big(\mathfrak{a}_{\mathfrak{u}+\mathtt{msb}}, \mathfrak{a}_{\mathfrak{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathfrak{u}+2^{d-1}}\big)\Big) \wedge \Big(\mathfrak{b}_{\mathfrak{u}}, \big(\mathfrak{b}_{\mathfrak{u}+\mathtt{msb}}, \mathfrak{b}_{\mathfrak{u}+2\mathtt{msb}}, \cdots, \mathfrak{b}_{\mathfrak{u}+2^{d-1}}\big)\Big) \\
= \quad & \Big(\mathfrak{a}_{\mathfrak{u}} \wedge \mathfrak{b}_{\mathfrak{u}}, \big(\quad \mathfrak{a}_{\mathfrak{u}+\mathtt{msb}} \wedge \mathfrak{b}_{\mathfrak{u}} + \overline{\mathfrak{a}_{\mathfrak{u}}} \wedge \mathfrak{b}_{\mathfrak{u}+\mathtt{msb}}, \\
& \qquad\qquad \mathfrak{a}_{\mathfrak{u}+2\mathtt{msb}} \wedge \mathfrak{b}_{\mathfrak{u}} + \overline{\mathfrak{a}_{\mathfrak{u}}} \wedge \mathfrak{b}_{\mathfrak{u}+2\mathtt{msb}}, \\
& \qquad\qquad\qquad\qquad \vdots \\
& \qquad\qquad \mathfrak{a}_{\mathfrak{u}+2^{d-1}} \wedge \mathfrak{b}_{\mathfrak{u}} + \overline{\mathfrak{a}_{\mathfrak{u}}} \wedge \mathfrak{b}_{\mathfrak{u}+2^{d-1}}\big)\Big)
\end{aligned} \tag{11.3}$$

Proof. Again, we aim at proving that commutative diagram of the outer product holds. On the one hand, mapping the prefix tree $\mathfrak{c}$ of Equation (11.3) in the binary tree using ψ of Equation (9.2) results in:

$$\begin{aligned}
& \Big(\mathfrak{c}_{\mathfrak{u}+\mathtt{msb}}, \big(\mathfrak{c}_{\mathfrak{u}+2\mathtt{msb}}, \big(\cdots(\mathfrak{c}_{\mathfrak{uu}+2^{d-1}}, \mathfrak{c}_{\mathfrak{u}})\big)\big)\Big)_b \\
= \quad & \Big(\mathfrak{a}_{\mathfrak{u}+\mathtt{msb}} \wedge \mathfrak{b}_{\mathfrak{u}} + \overline{\mathfrak{a}_{\mathfrak{u}}} \wedge \mathfrak{b}_{\mathfrak{u}+\mathtt{msb}}, \big(\mathfrak{a}_{\mathfrak{u}+2\mathtt{msb}} \wedge \mathfrak{b}_{\mathfrak{u}} + \overline{\mathfrak{a}_{\mathfrak{u}}} \wedge \mathfrak{b}_{\mathfrak{u}+2\mathtt{msb}}, \big(\cdots \\
& \big(\mathfrak{a}_{\mathfrak{u}+2^{d-1}} \wedge \mathfrak{b}_{\mathfrak{u}} + \overline{\mathfrak{a}_{\mathfrak{u}}} \wedge \mathfrak{b}_{\mathfrak{u}+2^{d-1}}, \mathfrak{a}_{\mathfrak{u}} \wedge \mathfrak{b}_{\mathfrak{u}}\big)\big)\big)\Big)_b
\end{aligned} \tag{11.4}$$

On the other hand, the recursive formula of the outer product between two multivectors in the binary tree framework is the pair:

$$\Big(\mathfrak{c}_{\mathfrak{u}+\mathtt{msb}}, \mathfrak{c}_{\mathfrak{u}}\Big)_b = \Big(\mathfrak{a}_{\mathfrak{u}+\mathtt{msb}} \wedge \mathfrak{b}_{\mathfrak{u}} + \overline{\mathfrak{a}_{\mathfrak{u}}} \wedge \mathfrak{b}_{\mathfrak{u}+\mathtt{msb}}, \mathfrak{a}_{\mathfrak{u}} \wedge \mathfrak{b}_{\mathfrak{u}}\Big)_b \tag{11.5}$$

After developing this formula at one recursion depth, this formula becomes:

$$\begin{aligned}
& \Big(\mathfrak{c}_{\mathfrak{u}+\mathtt{msb}}, \Big(\mathfrak{c}_{\mathfrak{u}+2\mathtt{msb}}, \mathfrak{c}_{\mathfrak{u}}\Big)\Big)_b \\
= \quad & \Big(\mathfrak{a}_{\mathfrak{u}+\mathtt{msb}} \wedge \mathfrak{b}_{\mathfrak{u}} + \overline{\mathfrak{a}_{\mathfrak{u}}} \wedge \mathfrak{b}_{\mathfrak{u}+\mathtt{msb}}, \Big(\mathfrak{a}_{\mathfrak{u}+2\mathtt{msb}} \wedge \mathfrak{b}_{\mathfrak{u}} + \overline{\mathfrak{a}_{\mathfrak{u}}} \wedge \mathfrak{b}_{\mathfrak{u}+2\mathtt{msb}}, \mathfrak{a}_{\mathfrak{u}} \wedge \mathfrak{b}_{\mathfrak{u}}\Big)\Big)_b
\end{aligned} \tag{11.6}$$

Then the formula takes the following form:

$$\begin{aligned} & \left(\mathtt{c}_{\mathtt{u}+\mathtt{msb}}, \left(\mathtt{c}_{\mathtt{u}+2\mathtt{msb}}, \left(\cdots \left(\mathtt{c}_{\mathtt{uu}+2^{d-1}}, \mathtt{c}_{\mathtt{u}}\right)\right)\right)\right)_b \\ = & \left(\mathtt{a}_{\mathtt{u}+\mathtt{msb}} \wedge \mathtt{b}_{\mathtt{u}} + \overline{\mathtt{a}_{\mathtt{u}}} \wedge \mathtt{b}_{\mathtt{u}+\mathtt{msb}}, \left(\mathtt{a}_{\mathtt{u}+2\mathtt{msb}} \wedge \mathtt{b}_{\mathtt{u}} + \overline{\mathtt{a}_{\mathtt{u}}} \wedge \mathtt{b}_{\mathtt{u}+2\mathtt{msb}}, \left(\cdots \right.\right.\right. \\ & \left.\left.\left.\left(\mathtt{a}_{\mathtt{u}+2^{d-1}} \wedge \mathtt{b}_{\mathtt{u}} + \overline{\mathtt{a}_{\mathtt{u}}} \wedge \mathtt{b}_{\mathtt{u}+2^{d-1}}, \mathtt{a}_{\mathtt{u}} \wedge \mathtt{b}_{\mathtt{u}}\right)\right)\right)\right)_b \end{aligned} \tag{11.7}$$

Equations (11.7) and (11.4) are equivalent, thus the commutative diagram holds. □

This recursive formula is the base to develop the pseudo-code of the outer product. As a reminder, the recursive formula is defined as:

$$\begin{aligned} & \left(\mathtt{c}_{\mathtt{u}}, \left(\mathtt{c}_{\mathtt{u}+\mathtt{msb}}, \mathtt{c}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathtt{c}_{\mathtt{u}+2^{d-1}}\right)\right) \\ = & \left(\mathtt{a}_{\mathtt{u}} \wedge \mathtt{b}_{\mathtt{u}}, \left(\begin{array}{r} \mathtt{a}_{\mathtt{u}+\mathtt{msb}} \wedge \mathtt{b}_{\mathtt{u}} + \overline{\mathtt{a}_{\mathtt{u}}} \wedge \mathtt{b}_{\mathtt{u}+\mathtt{msb}}, \\ \mathtt{a}_{\mathtt{u}+2\mathtt{msb}} \wedge \mathtt{b}_{\mathtt{u}} + \overline{\mathtt{a}_{\mathtt{u}}} \wedge \mathtt{b}_{\mathtt{u}+2\mathtt{msb}}, \\ \vdots \\ \mathtt{a}_{\mathtt{u}+2^{d-1}} \wedge \mathtt{b}_{\mathtt{u}} + \overline{\mathtt{a}_{\mathtt{u}}} \wedge \mathtt{b}_{\mathtt{u}+2^{d-1}} \end{array} \right)\right) \end{aligned} \tag{11.8}$$

We highlight the main parts of the recursive formula and their equivalent in the pseudo-code shown in Algorithm 5.

11.2. Complexity

This paragraph investigates the complexity of the outer product. At each depth of the tree, the number of recursive calls is multiplied by 2. Furthermore, to a considered depth corresponds the same grade and thus the same number of recursive calls. Finally, the number of nodes of the same grade k is given by $\binom{d}{k}$ and in the worst case the depth may vary from 0 to d. Thus, the number of recursive calls is upper bounded by:

$$\sum_{i=0}^{d} \binom{d}{k} 2^i \tag{11.9}$$

From the binomial theorem this formula can be rewritten as:

$$\sum_{i=0}^{d} \binom{d}{k} 2^i = 3^d \tag{11.10}$$

Thus, the number of recursive calls is upper bounded by 3^d which is lower than the number of recursive calls obtained with the two previous methods

Algorithm 5: Recursive outer product $C = A \wedge B$

```
1  Function outer
       Input: A, B: two multivectors,
              C: resulting multivector,
              k_a, k_b and k_c: the respective grade of each multivector.
              label_a, label_b, label_c: recursive position on each tree.
              sign: recursive sign index.
              complement: recursive value ( ±1).
2      if grade(label_c) == k_c then                          // end of recursion
3          C[label_c] += sign × A[label_a] × B[label_b]
4      else                                                   // recursive calls
5          msb_a = labelToMsb(label_a)
6          msb_b = labelToMsb(label_b)
7          msb_c = labelToMsb(label_c)
8          foreach msb such that
               gradeKReachable(k_c, msb, label_c) == true do
9              label = label_c + msb
10             if gradeKReachable(k_a, msb, label_a) then
11                 outer(A, B, C, k_a, k_b, k_c, label_a +
                   msb, label_b, label, sign × complement, complement)
12             if gradeKReachable(k_b, msb, label_b) then
13                 outer(A, B, C, k_a, k_b, k_c, label_a, label_b +
                   msb, label, sign, −complement)
```

presented up to now in:

$$\sum_{i=1}^{d} 3^i = \frac{3}{2}(3^d - 1) \tag{11.11}$$

11.3. GA products using metric

11.3.1. Left, right contractions and inner product. Section 11.1 defines the recursive outer product over a prefix tree. This section details various products and algorithms that depend on the metric. In this context, we assume that a diagonal metric *diagMetric* is stored as a vector whose size is the dimension d. This vector is such that:

$$\begin{aligned} diagMetric(0) &= \mathbf{e}_1 \cdot \mathbf{e}_1 \\ diagMetric(1) &= \mathbf{e}_2 \cdot \mathbf{e}_2 \\ &\vdots \\ diagMetric(d-1) &= \mathbf{e}_d \cdot \mathbf{e}_d \end{aligned} \tag{11.12}$$

Using a metric in the recursive functions results in an additional parameter, namely `metric`, corresponding to the elements of *diagMetric*. The definition of the left contraction is shown in Algorithm 6.

Algorithm 6: Recursive left contraction product $C = A \rfloor B$

Function `leftcont`
  Input: A, B: two multivectors.
  C: resulting multivector.
  k_a, k_b and k_c: respective grade of each multivector.
  $\texttt{label}_a$, $\texttt{label}_b$, $\texttt{label}_c$: recursive position on each tree.
  `sign`: a recursive sign index.
  `complement`: recursive value (± 1).
  `metric`: coefficients related to the metric.
  if $\texttt{grade}(\texttt{label}_b) == k_b$ **then** // end of recursion
    $C[\texttt{label}_c] += \texttt{metric} \times \texttt{sign} \times A[\texttt{label}_a] \times B[\texttt{label}_b]$
  else // recursive calls
    $\texttt{msb}_a = \texttt{labelToMsb}(\texttt{label}_a)$
    $\texttt{msb}_b = \texttt{labelToMsb}(\texttt{label}_b)$
    $\texttt{msb}_c = \texttt{labelToMsb}(\texttt{label}_c)$
    foreach `msb` **such that** $\texttt{gradeKReachable}(k_b, \texttt{msb}, \texttt{label}_b) == \texttt{true}$ **do**
      $\texttt{label} = \texttt{label}_b + \texttt{msb}$
      if $\texttt{gradeKReachable}(k_a, \texttt{msb}, \texttt{label}_a)$ **then**
        $\texttt{leftcont}(A, B, C, k_a, k_b, k_c, \texttt{label}_a + \texttt{msb}, \texttt{label}, \texttt{label}_c, \texttt{sign} \times \texttt{complement}, -\texttt{complement}, \texttt{metric} \times \mathit{diagMetric}(\mathit{grade}(\texttt{label}_b)))$
      if $\texttt{gradeKReachable}(k_c, \texttt{msb}, \texttt{label}_c)$ **then**
        $\texttt{leftcont}(A, B, C, k_a, k_b, k_c, \texttt{label}_a, \texttt{label}, \texttt{label}_c + \mathit{msb}, \texttt{sign}, -\texttt{complement}, \texttt{metric})$

The right contraction is simply a variation of the left contraction. The resulting pseudo-code is detailed in Algorithm 7.

11.3.2. Recursive geometric product. The last product to define is a major one, namely the geometric product.

Proposition 11.2. *Given two multivectors A and B whose recursive constructions are respectively:*

$$\left(\mathfrak{a}_{\mathtt{u}}, \left(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}}\right)\right), \tag{11.13}$$

and

$$\left(\mathfrak{b}_{\mathtt{u}}, \left(\mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{b}_{\mathtt{u}+2^{d-1}}\right)\right), \tag{11.14}$$

Algorithm 7: Recursive right contraction product $C = A \lfloor B$

```
1  Function rightcont
       Input: A,B: two multivectors.
              C: resulting multivector.
              k_a, k_b and k_c: respective grade of each multivector.
              label_a,label_b,label_c: recursive position on each tree.
              sign: recursive sign index.
2             complement: recursive value (±1).
3             metric: coefficients related to the metric.
4      if grade(label_b) == k_b then                          // end of recursion
5          C[label_c] += metric × sign × A[label_a] × B[label_b]
6      else                                                   // recursive calls
7          msb_a = labelToMsb(label_a)
8          msb_b = labelToMsb(label_b)
9          msb_c = labelToMsb(label_c)
10         foreach msb such that
             gradeKReachable(k_a,msb,label_a)== true do
11             label = label_a + msb
12             if gradeKReachable(k_b,msb,label_b) then
13                 rightcont(A, B, C, k_a, k_b,k_c, label, label_b + msb,
                     label_c, sign × complement, −complement,
                     metric × diagMetric(grade(label_b)))
14             if gradeKReachable(k_c,msb,label_c) then
15                 rightcont(A,B,C,k_a,k_b,k_c,label,label_b,label_c +
                     msb,sign,−complement,metric))
```

the recursive construction of the geometric product $C = A * B$ can be computed as:

$$
\begin{aligned}
& \left(\mathfrak{c}_{\mathtt{u}}, \left(\mathfrak{c}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{c}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{c}_{\mathtt{u}+2^{d-1}}\right)\right) \\
= \; & \left(\mathfrak{a}_{\mathtt{u}}, \left(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}}\right)\right) * \left(\mathfrak{b}_{\mathtt{u}}, \left(\mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{b}_{\mathtt{u}+2^{d-1}}\right)\right) \\
= \; & \Big(\mathfrak{a}_{\mathtt{u}} * \mathfrak{b}_{\mathtt{u}} + \mathit{diagMetric}(\mathtt{msb})\overline{\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}} * \mathfrak{b}_{\mathtt{u}+\mathtt{msb}} \\
& + \mathit{diagMetric}(2\mathtt{msb})\overline{\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}} * \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}} \\
& \qquad\qquad \vdots \\
& + \mathit{diagMetric}(2^{d-1})\overline{\mathfrak{a}_{\mathtt{u}+2^{d-1}}} * \mathfrak{b}_{\mathtt{u}+2^{d-1}}, \\
& \left(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{a}_{\mathtt{u}+2\mathtt{msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \right. \\
& \left. \mathfrak{a}_{\mathtt{u}+2^{d-1}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+2^{d-1}}\right)\Big)
\end{aligned}
\tag{11.15}
$$

Proof. This proof follows a similar outline as with the recursive outer product. First, the mapping the prefix tree $\mathfrak{c}$ of Equation (11.15) in the binary tree using ψ of Equation (9.2) results in:

$$
\begin{aligned}
&\left(\mathfrak{c}_{\mathtt{u+msb}}, \left(\mathfrak{c}_{\mathtt{u+2msb}}, \left(\cdots (\mathfrak{c}_{\mathtt{uu}+2^{d-1}}, \mathfrak{c}_{\mathtt{u}})\right)\right)\right)_b \\
= &\left(\mathfrak{a}_{\mathtt{u+msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u+msb}}, \left(\mathfrak{a}_{\mathtt{u+2msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u+2msb}}, \left(\cdots\right.\right.\right. \\
&(\mathfrak{a}_{\mathtt{u}+2^{d-1}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+2^{d-1}}, \mathfrak{a}_{\mathtt{u}} * \mathfrak{b}_{\mathtt{u}} + \mathit{diagMetric}(\mathtt{msb})\overline{\mathfrak{a}_{\mathtt{u+msb}}} * \mathfrak{b}_{\mathtt{u+msb}} \\
&+\mathit{diagMetric}(\mathtt{2msb})\overline{\mathfrak{a}_{\mathtt{u+2msb}}} * \mathfrak{b}_{\mathtt{u+2msb}} \\
&\qquad\qquad\vdots \\
&\left.\left.\left.+\mathit{diagMetric}(2^{d-1})\overline{\mathfrak{a}_{\mathtt{u}+2^{d-1}}} * \mathfrak{b}_{\mathtt{u}+2^{d-1}})\right)\right)\right)_b
\end{aligned}
\tag{11.16}
$$

On the other hand, the recursive formula of the geometric product between two multivectors in the binary tree framework can be the pair:

$$
\begin{aligned}
&\left(\mathfrak{c}_{\mathtt{u+msb}}, \mathfrak{c}_{\mathtt{u}}\right)_b \\
= &\left(\mathfrak{a}_{\mathtt{u+msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u+msb}}, \mathfrak{a}_{\mathtt{u}} * \mathfrak{b}_{\mathtt{u}} + \mathit{diagMetric}(\mathtt{msb})\overline{\mathfrak{a}_{\mathtt{u}} + \mathtt{msb}} * \mathfrak{b}_{\mathtt{u+msb}}\right)_b
\end{aligned}
\tag{11.17}
$$

After developing this formula at one recursion depth and having in mind the recursive structure of the binary tree, the above formula becomes:

$$
\begin{aligned}
&\left(\mathfrak{c}_{\mathtt{u+msb}}, \left(\mathfrak{c}_{\mathtt{u+2msb}}, \mathfrak{c}_{\mathtt{u}}\right)\right)_b \\
= &\left(\mathfrak{a}_{\mathtt{u+msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u+msb}}, \left(\mathfrak{a}_{\mathtt{u+2msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u+2msb}}, \mathfrak{a}_{\mathtt{u}} * \mathfrak{b}_{\mathtt{u}}\right.\right. \\
&+\mathit{diagMetric}(\mathtt{msb})\overline{\mathfrak{a}_{\mathtt{u}} + \mathtt{msb}} * \mathfrak{b}_{\mathtt{u+msb}} \\
&\left.\left.+\mathit{diagMetric}(\mathtt{2msb})\overline{\mathfrak{a}_{\mathtt{u}} + \mathtt{2msb}} * \mathfrak{b}_{\mathtt{u+2msb}}\right)\right)_b
\end{aligned}
\tag{11.18}
$$

Thus,

$$
\begin{aligned}
&\left(\mathfrak{c}_{\mathtt{u}+\mathtt{msb}}, \left(\mathfrak{c}_{\mathtt{u}+2\mathtt{msb}}, \left(\cdots(\mathfrak{c}_{\mathtt{uu}+2^{d-1}}, \mathfrak{c}_{\mathtt{u}})\right)\right)\right)_b \\
= &\left(\mathfrak{a}_{\mathtt{u}+\mathtt{msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \left(\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, (\cdots\right.\right. \\
&(\mathfrak{a}_{\mathtt{u}+2^{d-1}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+2^{d-1}}, \mathfrak{a}_{\mathtt{u}} * \mathfrak{b}_{\mathtt{u}} + \mathit{diagMetric}(\mathtt{msb})\overline{\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}} * \mathfrak{b}_{\mathtt{u}+\mathtt{msb}} \\
&+\mathit{diagMetric}(2\mathtt{msb})\overline{\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}} * \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}} \\
&\qquad\qquad\vdots \\
&\left.\left.+\mathit{diagMetric}(2^{d-1})\overline{\mathfrak{a}_{\mathtt{u}+2^{d-1}}} * \mathfrak{b}_{\mathtt{u}+2^{d}})\right)\right)\Big)_b
\end{aligned}
\tag{11.19}
$$

Equations (11.19) and (11.16) are equivalent, thus the commutative diagram holds for the geometric product. □

Algorithm: This recursive formula is the base to develop the pseudo-code of the geometric product. As a reminder, the recursive formula is defined as:

$$
\begin{aligned}
&\left(\mathfrak{c}_{\mathtt{u}}, (\mathfrak{c}_{\mathtt{u}+\mathtt{msb}}, \mathfrak{c}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{c}_{\mathtt{u}+2^{d-1}})\right) \\
= &\left(\mathfrak{a}_{\mathtt{u}} * \mathfrak{b}_{\mathtt{u}} + \mathit{diagMetric}(\mathtt{msb})\overline{\mathfrak{a}_{\mathtt{u}+\mathtt{msb}}} * \mathfrak{b}_{\mathtt{u}+\mathtt{msb}}\right. \\
&+\mathit{diagMetric}(2\mathtt{msb})\overline{\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}}} * \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}} \\
&+\cdots \\
&+\mathit{diagMetric}(2^{d-1})\overline{\mathfrak{a}_{\mathtt{u}+2^{d-1}}} * \mathfrak{b}_{\mathtt{u}+2^{d-1}}, (\mathfrak{a}_{\mathtt{u}+\mathtt{msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+\mathtt{msb}}, \\
&\left.\mathfrak{a}_{\mathtt{u}+2\mathtt{msb}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+2\mathtt{msb}}, \cdots, \mathfrak{a}_{\mathtt{u}+2^{d-1}} * \mathfrak{b}_{\mathtt{u}} + \overline{\mathfrak{a}_{\mathtt{u}}} * \mathfrak{b}_{\mathtt{u}+2^{d-1}})\right)\Big)
\end{aligned}
\tag{11.20}
$$

We highlight the main parts of the recursive formula and their equivalent in the pseudo-code presented in Algorithm 8.

11.4. Complexity

Concerning the left and right contractions, the number of recursive calls is the same as the recursive outer product. Hence, these two recursive products require 3^d recursive calls in the worst case. Again, this is a lower complexity than state of the art methods. As for the geometric product, we note that the computation of the sign is performed by a constant time operation. By an operation similar to the computation of the complexity of the outer product, we prove that the number of recursive calls of the geometric product is 4^d. We can note that this approach performs better than the binary tree approach whose number of recursive calls was $\frac{4}{3}(4^d - 1)$.

Algorithm 8: Recursive geometric product $C = AB$

```
1  Function geometric
       Input: A, B: two multivectors.
              C: resulting multivector.
              k_a, k_b and k_c: respective grade of each multivector.
              label_a, label_b, label_c: recursive position on each tree.
              sign: a recursive sign index.
              complement: recursive value (±1).
              metric: coefficients related to the metric.
              depth: current depth in the prefix tree.
2      if grade(label_b) == k_b and grade(label_a) == k_a then
3          C[label_c] += metric × sign × A[label_a] × B[label_b]
                                                  // end of recursion
4      else
5          msb_a = labelToMsb(label_a)
6          msb_b = labelToMsb(label_b)
7          msb_c = labelToMsb(label_c)
8          for i in 2^depth, 2^(depth+1), ··· , 2^(d-1) do
9              if gradeKReachable(k_b, i, label_b) then
10                 if gradeKReachable(k_a, i, label_a) then
11                     geometric(A, B, C, k_a, k_b, k_c, label_a + i, label_b +
                         i, label_c, sign ×
                         complement, −complement, metric ×
                         diagMetric(i), depth + 1))
12             if gradeKReachable(k_a, i, label_a) then
13                 geometric(A, B, C, k_a, k_b, k_c, label, label_b, label_c +
                     msb, sign ×
                     complement, complement, metric, depth + 1))
14             if gradeKReachable(k_b, i, label_b) then
15                 geometric(A, B, C, k_a, k_b, k_c, label_a, label_b +
                     i, label_c + i, sign, −complement, metric), depth +
                     1)
```

12. Products with dual multivectors

A recursive product where one or both of the operands are dual multivectors can be optimized by an extension of Algorithm 5 adapted to the dual tree defined in Section 7.2. In a certain sense, it is like if the recursive product algorithm is dualized instead of the multivectors. In this situation, potential costly dualizations can be avoided.

In practice, we already have a recursive method to compute the inner product as well as a method to traverse the dual prefix tree of the resulting

multivector C using the structure *dualCoefficients*. The resulting pseudo-code is shown in Algorithm 9 when the grade of A is lower than the grade of B. When the grade of B is higher than the grade of A then the algorithm is very similar (extracted from the right contraction algorithm).

Algorithm 9: Recursive outer product between a primal multivector and the dual of another multivector: $C = A \wedge B^*$

```
1  Function outerPrimalDual
       Input: A,B: two multivectors.
              C: resulting multivector.
              k_a, k_b and k_c: respective grade of each multivector.
              label_a,label_b,label_c: recursive position on each tree.
              sign: a recursive sign index.
2             complement: recursive value (±1).
3             metric: coefficients related to the metric.
4      if grade(label_b) == k_b then                          // end of recursion
5          dualCoefficients(label_c) × C[label_c] += dualCoefficients × sign × A[label_a] × B[label_b]
6      else                                                   // recursive calls
7          msb_a = labelToMsb(label_a)
8          msb_b = labelToMsb(label_b)
9          msb_c = labelToMsb(label_c)
10         foreach msb such that gradeKReachable(k_b,msb,label_b)== true do
11             label = label_b + msb
12             if gradeKReachable(k_a,msb,label_a) then
13                 outerPrimalDual(A,B, C, k_a, k_b, k_c, label_a + msb, label, label_c, sign × complement, −complement, metric × diagMetric(grade(label_b)))
14             if gradeKReachable(k_c,msb,label_c) then
15                 outerPrimalDual(A,B,C,k_a,k_b,k_c,label_a,label,label_c − msb,sign,−complement,metric)
```

13. Non orthogonal metric

13.1. Automatic basis change and numerical clean up

For ergonomic purposes, any optional basis changes required by an arbitrary metric are automatically handled by the generated library. This basis change is included in the precomputed functions during the precomputation process and is explicitly computed for the recursive products before

and after the recursive calls. The library generator first checks if the metric is a valid symmetric matrix. If the matrix is identity, all the generated products are left unchanged. If the matrix is a diagonal matrix (but not identity), the metric coefficients are inserted in the products. In any other cases, we follow [12] and proceed to a basis change, however we also add some numerical robustness pre-processing. As an example, let us consider the Conformal Geometric Algebra of $\mathbb{R}^2$ with metric M and its eigen decomposition $M = PDP^{-1}$:

$$\begin{aligned} M &= \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0.707 & 0.707 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ -0.707 & 0.707 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.707 & 0 & 0 & -0.707 \\ 0.707 & 0 & 0 & 0.707 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{aligned}$$

For such very common metrics, an eigen decomposition leads to square roots in the eigen vector components. For a better numerical robustness, we automatically upscale the matrix P such that it is composed of integers and downscale accordingly its inverse P^{-1}:

$$\begin{aligned} M &= \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 & -0.5 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{aligned}$$

Then, all the components of the resulting matrices are subject to a numerical clean up, by adjusting each value to the nearest integer, inverse power of two or decimal. Thus, this clean up removes the numerical errors generated by the eigen decomposition and is validated if the resulting decomposition still results in the original metric. In all the GA we encountered, this process removes all the numerical approximations.

13.2. Computing transformation matrices

The final stage consists in generating both transformation and inverse transformation matrices for any grade of the algebra. In practice, these transformation matrices are very sparse and are stored in the efficient `eigen` sparse matrices [17]. The algorithm followed to achieve this is explained in the following section.

13.2.1. Algorithm. We explain the computation of the transformation matrix P_k that maps any k-basis vector in the non-orthogonal basis to the k-basis vector in the orthogonal basis. First, let us consider the orthogonal basis as $(\mathbf{e}_1, \cdots, \mathbf{e}_d)^\top$ and the non-orthogonal basis as $(\mathbf{n}_1, \cdots, \mathbf{n}_d)^\top$. We assume that $P(i)$ denotes the i^th line of P and p_{ij} represents the element of the i^th line

and j^th column of P. The transformation matrix P maps these basis vectors as follows:

$$\begin{pmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_d \end{pmatrix} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1d} \\ p_{21} & p_{22} & \cdots & p_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ p_{d1} & p_{d2} & \cdots & p_{dd} \end{bmatrix} \begin{pmatrix} \mathbf{n}_1 \\ \mathbf{n}_2 \\ \vdots \\ \mathbf{n}_d \end{pmatrix} = \begin{bmatrix} P(1) \\ P(2) \\ \vdots \\ P(d) \end{bmatrix} \begin{pmatrix} \mathbf{n}_1 \\ \mathbf{n}_2 \\ \vdots \\ \mathbf{n}_d \end{pmatrix} \tag{13.1}$$

Thus, by definition:

$$\mathbf{e}_i = \mathrm{P}(i) \begin{pmatrix} \mathbf{n}_1 \\ \mathbf{n}_2 \\ \vdots \\ \mathbf{n}_d \end{pmatrix} \forall i \in [1,d], \tag{13.2}$$

For a k-basis blade defined as:

$$\mathbf{e}_{pqr\cdots u} = \mathbf{e}_p \wedge \mathbf{e}_q \wedge \mathbf{e}_r \wedge \cdots \wedge \mathbf{e}_u \tag{13.3}$$

Using formula 13.2, this yields:

$$\mathbf{e}_{pqr\cdots u} = P(p) \begin{pmatrix} \mathbf{n}_1 \\ \mathbf{n}_2 \\ \vdots \\ \mathbf{n}_d \end{pmatrix} \wedge \mathrm{P}(q) \begin{pmatrix} \mathbf{n}_1 \\ \mathbf{n}_2 \\ \vdots \\ \mathbf{n}_d \end{pmatrix} \wedge \mathrm{P}(r) \begin{pmatrix} \mathbf{n}_1 \\ \mathbf{n}_2 \\ \vdots \\ \mathbf{n}_d \end{pmatrix} \wedge \cdots \wedge \mathrm{P}(u) \begin{pmatrix} \mathbf{n}_1 \\ \mathbf{n}_2 \\ \vdots \\ \mathbf{n}_d \end{pmatrix} \tag{13.4}$$

Hence, the main point of determining the transformation matrices lies in computing the outer product between lines of the vector transformation matrix. The resulting pseudo-code is shown in Algorithm 10.

Algorithm 10: Compute the k-vector transformation matrix from the vector transformation matrix P

```
1 Function computeKvectorTransformationMatrix
    Input: P: the vector transformation matrix,
           d: vector space dimension,
           k: the grade of the transformation matrix to be computed.
2   foreach e_{pqr...u}, idx in k-basis blades do // idx:index of the blade
      in the k basis blades
3     ⟨P⟩_k(idx) ← vector whose dimension is (d k)
4     mv ← P(p)(n_1, n_2, ..., n_d)^⊤
5     foreach e_v in e_{qr...u} do
6       mv ← mv ∧ (P(v)(n_1, n_2, ..., n_d)^⊤)
7     ⟨P⟩_k(idx) ← mv
8   return ⟨P⟩_k
```

13.2.2. Data structure to use. As stated in the last section, the obtained transformation matrices are sparse. An example of the transformation matrix $\langle \mathrm{P} \rangle_3$ of 3-basis blades is shown below:

$$\langle \mathrm{P} \rangle_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{13.5}$$

We remark that for over a total of $10 \times 10 = 100$ elements, only 16 elements are non-null, giving to this matrix a sparsity score of $\frac{84}{100} = 0.84$. This score tends to be stable for extension of conformal Geometric Algebra in higher dimension. That is why, we use sparse matrix data structure of *Eigen*.

Note also that the inverse transformation matrices can be computed using the same process as with the transformation matrices. The only difference will be in the vector transformation matrix used as input. In this case, we will use the inverse of the vector transformation matrix. All the remaining algorithm remains the same.

Finally, in the case of non-full-rank metrics, the process remains unchanged, however the dual functions are not generated.

14. Resulting library generator: Garamon

The resulting implementation is a C++ template library generator dedicated to Geometric Algebra. The generator itself runs in C++ and generate optimized C++ code. These generated GA libraries are dedicated to being user-friendly and efficient both in term of computation speed and memory consumption.

14.1. Define an algebra

Each C++ library is generated for a specific GA, according to a dedicated configuration file. The required information concerns mainly the name of the algebra, its dimension, the name of the vectors and the metric (symmetric matrix). All other parameters can be set to their default values. An example is depicted in Figure 9.

14.2. Hybridization

For high dimension GA, the generated libraries include a soft transition between precomputed products and recursive products. The criteria for a

```
# Modeling 3D Geometry in the Clifford Algebra R(4,4),
# Juan Du, Ron Goldman and Stephen Mann, 2017

<namespace>
p3ga2
</namespace>

<metric>
 0   0   0   0   0.5 0   0   0
 0   0   0   0   0   0.5 0   0
 0   0   0   0   0   0   0.5 0
 0   0   0   0   0   0   0   0.5
 0.5 0   0   0   0   0   0   0
 0   0.5 0   0   0   0   0   0
 0   0   0.5 0   0   0   0   0
 0   0   0   0.5 0   0   0   0
</metric>

<basis vector name>
0 1 2 3 d0 d1 d2 d3
</basis vector name>

...
```

FIGURE 9. Example of a configuration file (its main part). The considered algebra is $\mathbb{R}^{4,4}$ used in [10]

product to be implemented either with precomputed functions or recursively is defined by a user defined threshold on the size of the two k-vectors involved in the product. With this approach, a GA library over a 10 dimension vector space can entirely be implemented in precomputed functions and a GA library over a 15 dimension vector space will have at least the products of vectors implemented with precomputed functions. Within this framework, the hybridization is completely transparent to the user. Figure 10 shows the resulting source code memory consumption with and without the hybridization. Figure 11 depicts the binary memory consumption with the hybridization. Up to dimension 10, the growth is exponential. Then, for dimension 11 and 12, some products are computed at run time and not precomputed anymore. Since the removed products are the most memory consuming, this hybridization has a big impact on the overall memory consumption. The same phenomenon appears from dimension 13. For higher dimension, at least the product between vectors will remain precomputed, leading to a linear growth of the memory.

14.3. Generated code

A generated library consists in few source code files, its own dedicated installation file (cmake), as well as a dedicated sample code to help the user to start using the library. It also includes a dedicated cheetsheat listing all

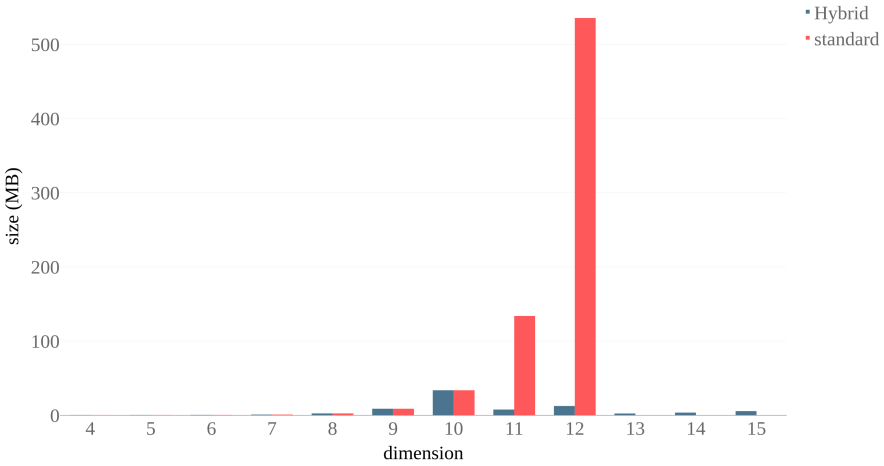


FIGURE 10. Memory requirement for the generated source code with and without hybridization.

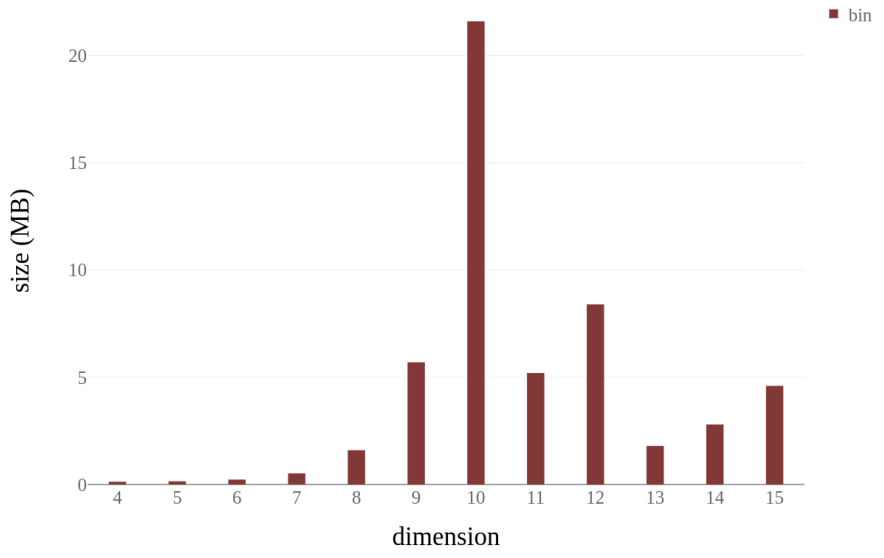


FIGURE 11. Memory requirement for the generated binary with hybridization.

the available operations. All the source files are well commented and documented with `Doxygen` [24].

The Listing 1 shows an example of some operations in CGA using Garamon.

LISTING 1. Example of code with Conformal Geometric Algebra of $\mathbb{R}^3$.

```
#include <iostream>
#include <c3ga/Mvec.hpp>

void plop(){

   c3ga::Mvec<double> mv1;
   mv1[c3ga::scalar] = 1.0;
   mv1[c3ga::E0] = 42.0;
   std::cout << "mv1 : " << mv1 << std::endl;

   c3ga::Mvec<double> mv2;
   mv2[c3ga::E0] = 1.0;
   mv2[c3ga::E1] = 2.0;
   mv2 += c3ga::I<double>() + 2*c3ga::e01<double>();

   // some products
   std::cout << "outer: " <<(mv1 ^ mv2)<< std::endl;
   std::cout << "inner: " <<(mv1 | mv2)<< std::endl;
   std::cout << "geo : " <<(mv1 * mv2)<< std::endl;
   std::cout << "left cont : "<<(mv1 < mv2)<< std::endl;
   std::cout << "right cont : "<<(mv1 > mv2)<< std::endl;

   // some tools
   std::cout << "grade : " << mv1.grade() << std::endl;
   std::cout << "norm : " << mv1.norm() << std::endl;
   mv1.clear();
}
```

Since all the generated libraries are identified by a namespace, multiple GA libraries can be used together as shown in Listing 2.

LISTING 2. Example of code using simultaneously the space time algebra of $\mathbb{R}^3$ and an Eclidean GA of $\mathbb{R}^3$.

```
#include <st3ga/Mvec.hpp>
#include <e3ga/Mvec.hpp>

void plop(){
   st3ga::Mvec<double> mv1 = ...;
   e3ga::Mvec<double> mv2 = ...;
}
```

15. Experimental results

We conducted some tests on high quality consumer grade hardware over several platforms (Ubuntu-16.04, MacOs-10.12 and Windows-10), with gcc-5.4, clang-9.0 and MinGW-7.2 and Visual Studio compilers. The compilers just need to be compatible with C++14. These tests mainly concern the speed

of the products, the size of the binary file, the size of the stored data and the dimension range. To get a better understanding of the results, we compared Garamon with some of the most efficient existing GA libraries in C++, namely Gaalop [6], Gaigen [14] and Versor [7].

15.1. High dimensions

In this section, the term dimension d refers to the dimension of the vector space used to build a GA composed of 2^d elements. As stated in [14], the maximum dimension supported by Gaigen is dimension 12. The tests we conducted on Versor showed that a single vector product could run in an Euclidean GA at most in dimension 10, due to compilation memory overloads. This maximum dimension falls to dimension 7 when the program tested involves various grades of k-vectors and various associated products.

Garamon is designed to be compatible with high dimension algebras. Due to some technical choices, Garamon has a hardcoded limit of dimension 31. However, in practice, while generating a library based on an Euclidean algebra of dimension 20 takes few seconds, generating a library based on a conformal vector space (including basis changes) of the same dimension 20 may requires hours. Then, the compilation may also be long, but should be done only once since our compilation process includes a full precompiled version for `float` and `double`.

For practical applications, we conducted some tests on both Double Projective Geometric Algebra of $\mathbb{R}^{4,4}$ [10] and Triple Conformal Geometric Algebra of $\mathbb{R}^{9,3}$ [11]. For higher dimensional algebra, we tested Garamon on the Quadric Conformal Geometric Algebra [5] built over a 15-dimensional vector space for real-time applications. There would be some interests to also conduct these tests on high dimension Euclidean GA dedicated to GIS sytems [26].

15.2. Speed computation

The speed computation tests were conducted on basic operations like outer products $C = A \wedge B$, inner products $C = A|B$, or some combinations $D = (A \wedge B)|C$. For more complex operations, we expect Gaalop [6] to provide some efficient code reduction such it becomes the best solution every time.

For Gaalop, we followed its standard usage and generated a set of functions with general signature like "`void myProduct(A,B,C)`", that are clearly efficient since no memory allocation nor memory copy are required. However, these functions are far from easy to use when combining several products. For the other tested libraries, we used the already defined functions, such as $\mathtt{c} = \mathtt{a} \wedge \mathtt{b}$. In most of the implementations, these operations require a memory allocation to locally store the result, and a copy to the final variable.

For each tested library, the speed performance can vary according to the platform, the compiler and the algebra dimension. However, the trend of these benchmarks tends to show that Gaalop and Versor are almost every time the fastest. Garamon presents the same performances as Gaigen, and

surprisingly performs sometimes better than Gaalop on products such as $D = (A \wedge B) \cdot C$.

The code profiling shows that a large part of the product in Garamon is actually used for the memory allocation. This situation is especially true when the result of a product has several grades, like is some geometric products where the memory allocation is performed for all independent grades and not once, like in Gaigen. The memory handling of Garamon, however, presents some good property when manipulating a large amount of data, as described in the section 15.3.

15.3. Memory consumption

The data memory consumption tests were conducted by generating both many random vectors and bivectors. Let d be the dimension of the vector space supporting the algebra, Table 3 and 4 show that the per-grade arrays has a memory storage roughly linear in d when the full multivector has a memory complexity of $\mathcal{O}(2^d)$.

TABLE 3. Memory requirement (in MB) to store 50000 random vectors.

dimension	5	6	7	8	10	15
Gaigen [14]	12.8	25.6	51.2	102.4	409.6	–
Versor [7]	4.6	5.0	5.5	6.3	–	–
Garamon	2.1	2.5	2.9	3.4	4.3	6.4

TABLE 4. Memory requirement (in MB) to store 50000 random bivectors.

dimension	5	6	7	8	10	15
Gaigen [14]	12.8	25.6	51.20	102.4	409.6	–
Versor [7]	7.9	11.8	16.6	22.1	–	–
Garamon	5.3	7.9	11.2	15.3	24.7	57.6

16. Conclusion

This paper presents Garamon, a Geometric Algebra library generator synthesizing C++ libraries implementing Geometric Algebra of low and high dimensions for any arbitrary metrics. The objective of Garamon is to be as user friendly as possible, without too much computation speed repercussions, and to have a good behaviour in term of memory consumption. The libraries are generated from a simple specification file. The "per grade" data structure used in Garamon is an efficient compromise between data storage,

computation efficiency and user friendliness. According to the base vector space dimension, the generated specialized libraries are implemented either with full precomputed operations or also based on a new recursive scheme following a prefix tree multivector representation for higher dimensions. An "upside down" reading of the prefix tree leads to recursive products of the dual multivector without any explicit dualization. Finally, Garamon can handle any arbitrary algebra signatures with a numerically robust basis change implementation. We consider Garamon as an efficient tool to easily test and investigate GA algorithms. Then, the final version of the method can be optimized only once with Gaalop.

Further work is in process to deal with even higher dimension, namely for vector space dimensions of up to 30 for handling cubic and quartic surfaces, for example. This would require a framework whose vector space dimension is higher than 20. To achieve this, we would develop new algorithms to compute only required products at runtime. To achieve this, cache-oblivious [8] of Geometric Algebra operators will be developed. These algorithms will be based on the prefix tree approach. In order to compute as low products as possible we would also base our approach on stochastic acceptance [21] of the products of Geometric Algebra with respects to the products computed by the user. Another interesting ongoing work consists in making *Garamon* capable of handling non-constant metrics. Applications of such works are wide and could consist in finding a way to compute the electric and magnetic part of the Riemann tensor as outlined in [2].

References

[1] Benger, W., and Dobler, W. Massive Geometric Algebra: Visions for C++ implementations of geometric algebra to scale into the big data era. *Advances in Applied Clifford Algebras 27* (2017), 2153–2174.

[2] Benger, W., Hamilton, A., Folk, M., Koziol, Q., Su, S., Schnetter, E., Ritter, M., and Ritter, G. Using geometric algebra for navigation in riemannian and hard disc space. *Proceedings of Computer Graphics, Computer Vision and Mathematics, Plzen, Czech Republic* (2008), 80–92.

[3] Breuils, S., Nozick, V., and Fuchs, L. A geometric algebra implementation using binary tree. *Advances in Applied Clifford Algebras 27*, 3 (Sep 2017), 2133–2151.

[4] Breuils, S., Nozick, V., Fuchs, L., Hildenbrand, D., Benger, W., and Steinmetz, C. A hybrid approach for computing products of high-dimensional geometric algebras. In *Proceedings of the Computer Graphics International Conference, ENGAGE* (Hiyoshi, Japan, 2017), CGI '17, ACM, pp. 43:1–43:6.

[5] Breuils, S., Nozick, V., Sugimoto, A., and Hitzer, E. Quadric conformal geometric algebra of $\mathbb{R}^{9,6}$. *Advances in Applied Clifford Algebras 28*, 2 (Mar 2018), 35.

[6] Charrier, P., Klimek, M., Steinmetz, C., and Hildenbrand, D. Geometric algebra enhanced precompiler for C++, OpenCL and Mathematica's OpenCLLink. *Advances in Applied Clifford Algebras 24*, 2 (2014), 613–630.

[7] COLAPINTO, P. *Spatial computing with conformal geometric algebra*. PhD thesis, University of California Santa Barbara, 2011.

[8] DEMAINE, E. D. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets 8*, 4 (2002), 1–249.

[9] DORST, L., FONTIJNE, D., AND MANN, S. *Geometric Algebra for Computer Science, An Object-Oriented Approach to Geometry*. Morgan Kaufmann, 2007.

[10] DU, J., GOLDMAN, R., AND MANN, S. Modeling 3D Geometry in the Clifford Algebra $\mathbb{R}^{4,4}$. *Advances in Applied Clifford Algebras 27*, 4 (Dec 2017), 3039–3062.

[11] EASTER, ROBERT BENJAMIN AND HITZER, ECKHARD. Triple conformal geometric algebra for cubic plane curves. *Mathematical Methods in the Applied Sciences* (2017). mma.4597.

[12] EID, A. H. An extended implementation framework for geometric algebra operations on systems of coordinate frames of arbitrary signature. *Advances in Applied Clifford Algebras 28*, 1 (Feb 2018), 16.

[13] EID, A. H. A. Optimized automatic code generation for geometric algebra based algorithms with ray tracing application. *arXiv preprint arXiv:1607.04767* (2016).

[14] FONTIJNE, D. Gaigen 2.5 user manual. `https://sourceforge.net/projects/g25/`.

[15] FONTIJNE, D. *Efficient Implementation of Geometric Algebra*. PhD thesis, University of Amsterdam, 2007.

[16] FUCHS, L., AND THÉRY, L. Implementing geometric algebra products with binary trees. *Advances in Applied Clifford Algebras 24*, 2 (2014), 589–611.

[17] GUENNEBAUD, G., JACOB, B., ET AL. Eigen v3. http://eigen.tuxfamily.org, 2010.

[18] HILDENBRAND, D. *Foundations of Geometric Algebra Computing*. Springer, 2013.

[19] LEOPARDI, P. GluCat: Generic library of universal Clifford algebra templates. `http://glucat.sourceforge.net/`.

[20] LEOPARDI, P. A generalized FFT for Clifford algebras. *Bulletin of Belgian Mathematical Society 11* (2004), 663–688.

[21] LIPOWSKI, A., AND LIPOWSKA, D. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications 391*, 6 (2012), 2193–2196.

[22] PERWASS, C. *Geometric algebra with applications in engineering*, vol. 4 of *Geometry and Computing*. Springer, 2009.

[23] SEYBOLD, F., AND WÖSSNER, U. Gaalet-a C++ expression template library for implementing geometric algebra. In *6th High-End Visualization Workshop* (2010).

[24] VAN HEESCH, D. Doxygen, 2004.

[25] VUILLEMIN, J. A data structure for manipulating priority queues. *Communications of the ACM 21*, 4 (April 1978), 309–314.

[26] ZHU, S., YUAN, S., LI, D., LUO, W., YUAN, L., AND YU, Z. Mvtree for hierarchical network representation based on geometric algebra subspace. *Advances in Applied Clifford Algebras 28*, 2 (Apr 2018), 39.

Stéphane Breuils
CNRS JFLI, UMI 3527,
National Institute of Informatics,
Tokyo 101-8430, Japan
Laboratoire d'Informatique Gaspard-Monge, Equipe A3SI,
UMR 8049, Université Paris-Est Marne-la-Vallée, France
e-mail: breuils@nii.ac.jp

Vincent Nozick
Laboratoire d'Informatique Gaspard-Monge, Equipe A3SI,
UMR 8049, Université Paris-Est Marne-la-Vallée, France
e-mail: vincent.nozick@u-pem.fr

Laurent Fuchs
XLIM-ASALI, UMR 7252,
Université de Poitiers, Poitiers, France
e-mail: Laurent.Fuchs@univ-poitiers.fr