HAL
open science

A Geometric Algebra Implementation using Binary Tree

Stéphane Breuils, Vincent Nozick, Laurent Fuchs

To cite this version:

Stéphane Breuils, Vincent Nozick, Laurent Fuchs. A Geometric Algebra Implementation using Binary Tree. Advances in Applied Clifford Algebras, 2017, 1, pp.1-19. 10.1007/s00006-017-0770-6 . hal-01510078v2

HAL Id: hal-01510078

https://hal.science/hal-01510078v2

Submitted on 3 May 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Geometric Algebra Implementation using Binary Tree

Stéphane Breuils, Vincent Nozick and Laurent Fuchs

Abstract. This paper presents an efficient implementation of geometric algebra, based on a recursive representation of the algebra elements using binary trees. The proposed approach consists in restructuring a state of the art recursive algorithm to handle parallel optimizations. The resulting algorithm is described for the outer product and the geometric product. The proposed implementation is usable for any dimensions, including high dimension (e.g. algebra of dimension 15). The method is compared with the main state of the art geometric algebra implementations, with a time complexity study as well as a practical benchmark. The tests show that our implementation is at least as fast as the main geometric algebra implementations.

Keywords. geometric algebra, implementation, binary trees.

1. Introduction

In this paper, we present a new method for computing geometric algebra products. This method is based on a recursive way of presenting the algebra. Currently, different implementations of geometric algebra exist. Gaigen [2], developed by Fontijne and Dorst, is considered as the first efficient implementation of geometric algebra. This library is available in C++. At compile time, a routine is generated from a specified geometric algebra. Products are optimized with respect to the type of multivector, leading to a specialized implementation. The optimized implementation includes specialized multivector classes and methods for computing geometric algebra products. Specialized multivectors do not store unspecified data. Moreover, products between these multivectors are hardcoded. This leads to a minimization of memory access and storage. All these optimizations hold when the type of each multivector is known before the compilation, see Fontijne [1]. Otherwise, a general multivector class is used. This general multivector stores only specified data and the grade of the multivector but the products of geometric algebra are not

hardcoded. These products will consist in computing the geometric product between each pair of non-zero elements from the two multivectors.

Gaalop [5], developed by Hildenbrand, is the most recent efficient implementation of the geometric algebra. This implementation uses a precompiler to generate a specialized code of an operation at compile time. This implementation converts a Clucalc code [3] into either serial C++ code or parallel Cuda/OpenCL code that already includes a developed form of the computations. That is to say, a significant part of the computations are performed during this "compilation" step. In practice, a multivector is represented by a list that stores each of its non-zero coefficients. At compile time, this implementation produces some 2D-tables indicating the resulting blade and sign of the product of two input k-vectors. Geometric algebra products between a pair of basis blades become constant time algorithm.

Geometric algebra can be used in a wide range of applications. An increasing number of these applications use high dimensional space. For example, Easter and Hitzer [10] introduced a new geometric algebra space, enabling the modelling of quadric surfaces in a 2^{10} dimensional space. In such situation, the storage of tables may be memory expensive. Moreover, for non-specific grade operation, Gaigen computes the products of two arbitrary blades with the general class representing multivectors. In such a case, this implementation may compute the product of a pair of multivectors composed by 2^d non-zero coefficients, leading to $2^{2d} = 4^d$ products. When the product requires less than 4^d products, useless products will be computed. Furthermore, the cost of computation of the sign requires d operations per product and is thus time consuming.

The method we propose is specifically well suited for high dimensional spaces, leading to a complementary approach to Gaigen and Gaalop. In order to compute efficiently each product, our approach is based on [4] developed by Fuchs and Théry. In this approach, each product is explicitly defined using recursive definitions. This approach leads to efficient computation of products. However this method does not handle parallel algorithm implementations.

The method we propose fixes the issue just raised above and consists in restructuring the recursive functions of [4] such that the products can be specialized according to the grade or a particular coefficient of the resulting multivector. The proposed approach yields to parallel algorithms. Indeed, we construct the products of geometric algebra which is somewhat related to the approach explained by Clifford in [11]. From this approach a specialized multivector class is extracted. The generated products are vectorized using SIMD instructions.

2. Definitions and Notations

This section presents the notations used for the sequel. Lower-case bold letters will denote basis blades and multivectors (multivector $\mathbf{a}$). Lower-case letters refers to multivector coordinates. For example, a_i is the ith coordinate of the multivector $\mathbf{a}$. The vector space dimension is denoted by 2^d, where d is the number of basis blades $\mathbf{e}_i$ of grade 1.

2.1. Binary Trees and Multivectors

A multivector $\mathbf{a}$ is represented as a sum of basis blades weighted by coefficients a_i, i.e.:

$$\mathbf{a} = \sum_{i=0}^{2^d-1} a_i \mathbf{E}_i \tag{2.1}$$

where $\mathbf{E}_i$ denotes a basis blade whose grade ranges from 0 to d (i.e. $\mathbf{E}_7 = \mathbf{e}_{123}$ with $d = 3$).

Fuchs and Théry [4] proposed a recursive representation of multivectors over binary trees. This binary tree is recursively defined as follows:

$$\mathfrak{a}^n = (\mathfrak{a}_1^{n+1}, \mathfrak{a}_0^{n+1})^n \tag{2.2}$$

where n is the depth of recursion. This depth may vary from 0 (root level) to d (leaf level). In this representation, each node of the binary tree $\mathfrak{a}$ is considered as multivector which contributes to the construction of the multivector $\mathbf{a}$. Each depth of the binary tree corresponds a basis vector (of grade 1). This basis vector contributes to the construction of the multivector of upper grades. A binary tree $(\mathfrak{a}_1^{n+1}, \mathfrak{a}_0^{n+1})^n$ is thus interpreted as $(\mathbf{e}_n \wedge \mathfrak{a}_1) + \mathfrak{a}_0$ at each depth n, that is to say $\mathfrak{a}^n$ is divided into two subtrees, namely the tree $\mathfrak{a}_1$ containing $\mathbf{e}_n$ and $\mathfrak{a}_0$ which does not, as shown on Figure 1. Considering that a leaf represents a basis blade, a multivector is defined by the mapping between these leaves and the multivector coefficients (see coefficients a_i on Figure 1). For the following, we define a node $\mathfrak{a}^n$ as a multivector composed of basis blades whose maximum grade is n.

3. Outer product

In this section, we consider the computation of the outer product in any d-dimensional space. Let $\mathbf{c} = \mathbf{a} \wedge \mathbf{b}$ be the outer product between two multivectors $\mathbf{a}$ and $\mathbf{b}$, then we have:

$$\mathbf{c} = \sum_{k=0}^{2^d-1} c_k \mathbf{E}_k = \sum_{i=0}^{2^d-1} a_i \mathbf{E}_i \wedge \sum_{j=0}^{2^d-1} b_j \mathbf{E}_j \tag{3.1}$$

The distributivity of the outer product leads to:

$$\mathbf{c} = \sum_{i=0}^{2^d-1} \sum_{j=0}^{2^d-1} a_i b_j \ \mathbf{E}_i \wedge \mathbf{E}_j \tag{3.2}$$

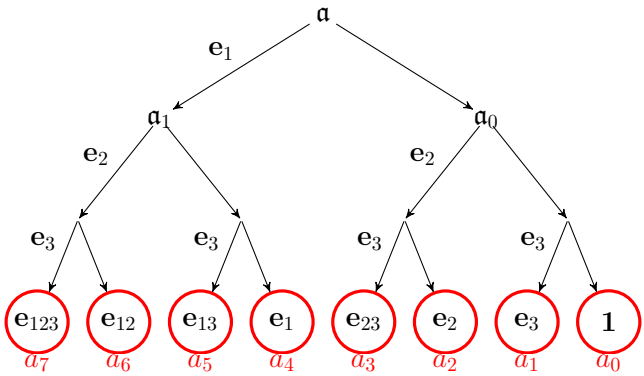

FIGURE 1. Binary tree of $\mathfrak{a}$, representing a multivector in a 2^3-dimensional space. There is a mapping between the coefficients a_i of the multivector and the leaves of the tree.

Finally a coefficient c_k of $\mathbf{c}$ can be expressed as:

$$c_k = \sum_{\substack{i,j \text{ such that} \\ \mathbf{E}_i \wedge \mathbf{E}_j = s_{ij}\mathbf{E}_k}} s_{ij}\ a_i \wedge b_j = \sum_{\substack{i,j \text{ such that} \\ \mathbf{E}_i \wedge \mathbf{E}_j = s_{ij}\mathbf{E}_k}} s_{ij}\ a_i b_j \tag{3.3}$$

where s_{ij} is -1 or 1 according to parity of the number of permutations required to change $\mathbf{E}_i \wedge \mathbf{E}_j$ in $\mathbf{E}_k$. In order to extract each coefficient c_k, one may think of computing each product $\mathbf{E}_i \wedge \mathbf{E}_j$, as performed by the non-specialized form of Gaigen. In this case, some products will lead to useless operations where $\mathbf{E}_i \wedge \mathbf{E}_j = 0$, for example where $i = j$. General multivectors $\mathbf{a}$ and $\mathbf{b}$ may have up to 2^d non-zero coefficients, leading to $2^d \times 2^d = 4^d$ products. This number of operations can be reduced by avoiding useless products.

In [4], each product is explicitly defined using a recursive definition over binary trees. The outer product between $\mathfrak{a}$ and $\mathfrak{b}$ is defined as follows:

$$\begin{array}{l} \mathfrak{a}^n \wedge \mathfrak{b}^n = (\mathfrak{a}_1^{n+1}, \mathfrak{a}_0^{n+1})^n \wedge (\mathfrak{b}_1^{n+1}, \mathfrak{b}_0^{n+1})^n \\ \text{if } n < d\ ,\ \mathfrak{a}^n \wedge \mathfrak{b}^n = \\ \quad \left(\mathfrak{a}_1^{n+1} \wedge \mathfrak{b}_0^{n+1} + \overline{\mathfrak{a}_0}^{n+1} \wedge \mathfrak{b}_1^{n+1}\ , \mathfrak{a}_0^{n+1} \wedge \mathfrak{b}_0^{n+1}\right)^n \\ \text{if } n = d\ ,\ \mathfrak{a}^n \wedge \mathfrak{b}^n = \mathfrak{a}^d \wedge \mathfrak{b}^d \end{array} \tag{3.4}$$

where $\overline{\mathfrak{a}}$ expresses the anticommutativity of the outer product. The outer product of two multivectors consists in developing the hereabove formula from the root to the leaves. From this formula, we may derive that for each depth n, the number of recursive calls is multiplied by 3. Thus the total number of recursive calls for a d-dimensional space is:

$$\sum_{i=1}^{d} 3^i = \frac{3}{2}\left(3^d - 1\right) \tag{3.5}$$

Therefore, the number of recursive calls is thus in $O(3^d)$ which is less than $O(4^d)$ corresponding to the complexity of the naive method.

3.1. Proposed method

To reduce the number of products of Equation (3.2) and to use parallel optimizations, we present a rewriting of the recursive functions. The contributions of our method are:

- to extract all products $a_i b_j$ for a considered coefficient $\mathfrak{c}_k$, and the sign associated with each product,
- to determine the products involved in $\mathfrak{c}$ knowing the grade of $\mathfrak{a}$ and $\mathfrak{b}$.

3.1.1. Binary trees labelling.

We first define a label for each node of the binary tree, derived from Huffman labelling, as illustrated in Figure 2. The label of a leaf provides the path from

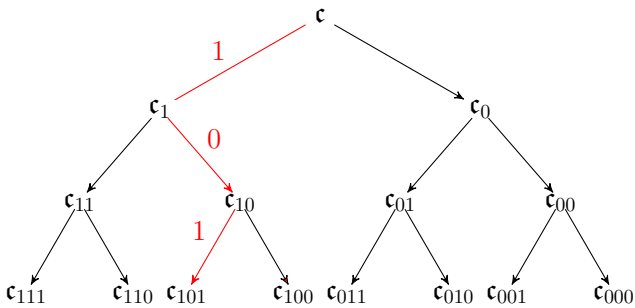

FIGURE 2. Labelling of a binary tree $\mathfrak{c}$ in a 3-dimensional space.

this leaf to the root. For example, the path that connects the leaf labelled 101 to the root is (left, right, left), as shown in red Figure 2.

3.1.2. From trees to lists. We know, from Equation 3.3, that each leaf c_k is expressed as a sum of products $a_i b_j$. In order to compute these products, we want to identify the lists a_i and b_j involved in the computation of each coefficient c_k. For example, the computation of c_5 will be based on the extraction of the list (a_5, a_4, a_1, a_0) and (b_0, b_1, b_4, b_5), see Table 1.

In order to identify these products for any dimension and for any coefficient c_k, we transform the recursive functions. More precisely, instead of reducing a set of products to a base case (recursive function), we start with a base case (root node), then we build some sequences forward from the base case. To achieve this, we use the labelling on Equation 3.4 to identify each node by a word u. A binary tree $\mathfrak{a}^n$ can be expressed as $\mathfrak{a}^n_u$, with its left child rewritten as $\mathfrak{a}^{n+1}_{u1}$ and its right child as $\mathfrak{a}^{n+1}_{u0}$.

$\mathbf{e}_{123}$	$\mathbf{e}_{12}$	$\mathbf{e}_{13}$	$\mathbf{e}_1$	$\mathbf{e}_{23}$	$\mathbf{e}_2$	$\mathbf{e}_3$	$\mathbf{1}$
c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
a_7b_0	a_6b_0	a_5b_0	a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
a_6b_1	a_4b_2	a_4b_1	a_0b_4	a_2b_1	a_0b_2	a_0b_1	
$-a_5b_2$	$-a_2b_4$	$-a_1b_4$		$-a_1b_2$			
a_4b_3	a_0b_6	a_0b_5		a_0b_3			
a_3b_4							
$-a_2b_5$							
a_1b_6							
a_0b_7							

TABLE 1. Outer product table in a 3-dimensional space.

Using this labelling, we can also rearrange the recursive definition of the outer product (Equation 3.4) as follows:

$$\begin{aligned}
&\mathfrak{a}_u^n \wedge \mathfrak{b}_v^n = (\mathfrak{a}_{u1}^{n+1}, \mathfrak{a}_{u0}^{n+1})^n \wedge (\mathfrak{b}_{v1}^{n+1}, \mathfrak{b}_{v0}^{n+1})^n \\
&\text{if } n < d \ , \ \mathfrak{a}_u^n \wedge \mathfrak{b}_v^n = \\
&\ \left(\mathfrak{a}_{u1}^{n+1} \wedge \mathfrak{b}_{v0}^{n+1} + \overline{\mathfrak{a}_{u0}}^{n+1} \wedge \mathfrak{b}_{v1}^{n+1} \ , \mathfrak{a}_{u0}^{n+1} \wedge \mathfrak{b}_{v0}^{n+1}\right)^n \\
&\text{if } n = d \ , \ \mathfrak{a}_u^n \wedge \mathfrak{b}_v^n = \ \mathfrak{a}_u^d \wedge \mathfrak{b}_v^d
\end{aligned} \tag{3.6}$$

In order to identify the components a_i and b_j involved in the computation of each coefficient c_k, we extract a construction of labels of $\mathfrak{a}$ and $\mathfrak{b}$ from Equation (3.6). Let *Alist* and *Blist* be these recursive constructions for labels of $\mathfrak{a}$ and $\mathfrak{b}$ respectively. In the following, we will only consider the construction of labels of $\mathfrak{a}$. We can prove by induction that labels of $\mathfrak{b}$ are the labels of $\mathfrak{a}$ in reverse order. The recursive definition of this *Alist* is the following:

$$\begin{aligned}
&\text{if } n < d \ , \ \{Alist^n\} = \\
&\ \left(\{Alist(1)^{n+1}, Alist(0)^{n+1}\}, \{Alist(0)^{n+1}\}\right) \\
&\text{if } n = d \ , \ \{Alist^n\} = \ \{Alist\}
\end{aligned} \tag{3.7}$$

where $\{. \ , \ .\}$ denotes the merge of two lists of labels, and $Alist(r)$ indicates that the elements of the list are suffixed by the letter r. The resulting tree of dimension 2^3 is depicted on Figure 3.

3.1.3. Construction of *Alist*. The recursive construction of the *Alist* provides the number p of products associated to a label u of $\mathfrak{c}$:

$$p = 2^{h(u)} \tag{3.8}$$

where $h(u)$ denotes the Hamming weights of u (i.e. the number of ones in a binary word u).

We now introduce an approach to determine the evolution of the labels of $\mathfrak{a}$ involved in the computation of each leaf $\mathfrak{c}$. The recursive function (3.7) shows that each left subtree is both suffixed by 0 and 1, meaning a duplication of the number of elements of the list, whereas the right subtree is only suffixed by 0.

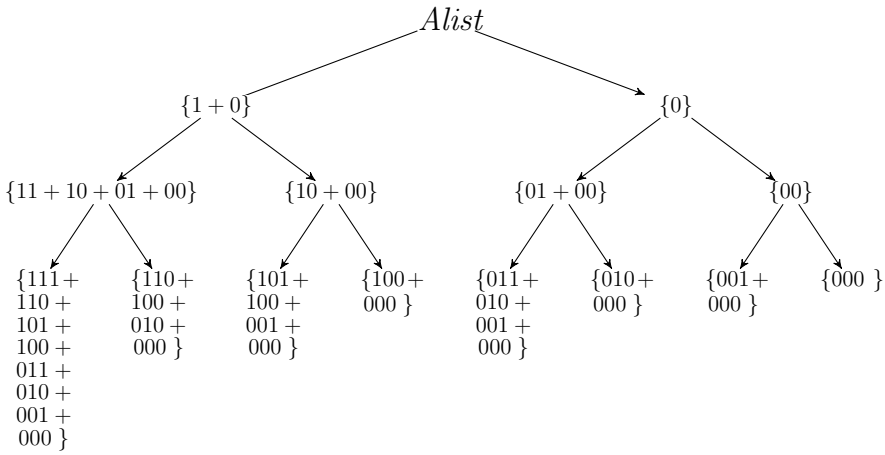

FIGURE 3. Recursive definition of *Alist* pushed down to the leaves.

From this result, an algorithm to construct *Alist* at only one level is extracted. In this algorithm, *Alist* is represented by a list of binary sequences. A binary sequence is an integer representing a label. Algorithm 1 gives pseudo-code for the method.

As an example, let us consider the *Alist* for the node $\mathfrak{c}_{101}$. This construction is equivalent to computing each product $a_i b_j$ for the leaf c_5 (for example $5_{(10)} = 101_{(2)}$). The computation of the *Alist* for the coefficient $\mathfrak{c}_{101}$ is described Table 2. In this table, the binary words $(000, 001, 100, 101)$ correspond to the coefficients (a_0, a_1, a_4, a_5) in table 1.

Algorithm 2 shows the pseudo-code of our method for a considered coefficient of $\mathfrak{c}$. The function `reverse`(*binaryWord*, *dimension*) computes the operation $(2^d - 1 - binaryWord)$, in order to compute the *Blist* coefficients from a *Alist*. Thus Algorithm 2 enables us to compute any coefficients of $\mathfrak{c}$ independently. Therefore the algorithm can be used to compute different coefficients of $\mathfrak{c}$ in parallel.

The latter algorithm can be further improved when the grade of the two multivectors are known. Let M be the grade of $\mathfrak{a}$, N the grade of $\mathfrak{b}$ and L the grade of $\mathfrak{a} \wedge \mathfrak{b}$. Then the grade of $\mathfrak{a} \wedge \mathfrak{b}$ is $L = M + N$. Thus, the computation of the outer product is equivalent to identifying and computing each coefficient $\mathfrak{c}_k$ whose grade is L. The labelling of the binary tree enables to efficiently extract the leaves of $\mathfrak{c}$ whose grade is L. An algorithm is produced and consists in traversing the binary tree of $\mathfrak{c}$. At each depth of this tree, if the grade of the label of $\mathfrak{c}$ is L then the products at this label can be computed and the children of this node don't have to be traversed. This enables

Algorithm 1: Outer product at one level

```
Function oneLevelOuterProduct
    Input: list: list of binary sequences
           b: bit of a label
    tmpList ← { }
    if b == 1 then
        tmpList.push(addBitList(list, 1, 0))
    else
        tmpList.push(addBitList(list, 0))
    return tmpList

Function addBitList(list, bit)
    listRes ← { }
    foreach label u of list do
        listRes.push(concat(u, bit))
    return listRes

// Overload the function above
Function addBitList(list, bitA, bitB)
    listRes ← { }
    foreach label u of list do
        listRes.push(concat(u, bitA))
        listRes.push(concat(u, bitB))
    return listRes
```

bit	subtree	*Alist*
Beginning	root	ϵ
1	left subtree	$(0, 1)$
	$\downarrow$	
0	right subtree	$(00, 10)$
	$\downarrow$	
1	left subtree	$(000, 001, 100, 101)$

TABLE 2. *Alist* computed at the node $\mathfrak{c}_{101}$.

us to efficiently compute the products. The algorithm used is shown in Algorithm 3. This algorithm is used to implement a per-grade specialization of our implementation.

Finally, note that the considered label is enough to compute the *Alist* elements. Firstly the binary 0 is in this list. Then the other elements can be defined by the list of binary words whose length is d such that the logical AND operator between this binary word and the label is non-zero. This method is

Algorithm 2: Outer product corresponding to a coefficient $\mathfrak{c}_k$

Data:
- $\mathfrak{a}, \mathfrak{b}$: multivectors
- *Alist*: sequence of binary words
- k: label of a coefficient of the multivector $\mathfrak{c}$
- d: dimension

Result: $\mathfrak{c}_k$: k^{th} coefficient of the multivector $\mathfrak{c} = \mathfrak{a} \wedge \mathfrak{b}$

```
Alist ← { }
Sign ← { }
SignOuter(Sign, 1, 1, d)                    // refer to Algorithm 4
foreach bit k_i of k do
    Alist ← oneLevelOuterProduct(Alist, k_i)
c_k ← 0
i ← 0
foreach label u of Alist do
    c_k ← c_k + Sign[i] · a_u · b_reverse(u,d)
    i ← i + 1
```

derived from the Hamming expression of Equation (3.8).

3.1.4. Complexity of this method.

The performance of our method is estimated from Algorithm 2. Firstly, the cost of the function `oneLevelOuterProduct`() in Algorithm 2 is linear to the size of the list and more precisely proportional to the hamming weight of the coefficient of the node. Thus, this operation is repeated d times. The latter computation is repeated 2^d times in the worst case (2^d non-zero coefficients) which leads to a complexity proportional to 3^d. The performance of this approach is thus asymptotically equivalent to the cost of the previous approach [4].

3.1.5. Sign computation.

Finally, the sign of each product $a_u b_v$ is computed. From our experimental results described in section 5, this computation might be the most time-consuming part of the outer product. The first method that we explored consisted in a "convolution" between the considered label in *Alist* and in *Blist* in a similar way to [1]. The convolution consists of right-shifting each bit of the label in *Alist* until the label is zero. At each iteration, we count the number of ones in common between the shifted label and the other label. The sign is obtained by raising -1 to the power of the number of ones.

The performance of this approach is estimated by computing the total number of right-shifting in $\mathfrak{a} \wedge \mathfrak{b}$. For a label a in *Alist*, the maximum number of right-shifting is $\lfloor log_2(a) \rfloor$. As mentioned in the previous subsection, the occurrence of a label a in the computation of the outer product is given by $2^{d-h(a)}$, with h

Algorithm 3: Outer product of two multivectors, with a known resulting grade

```
Data:
Sign: Sequence of signs
𝔞, 𝔟: multivectors
Function perGradeOuterP
    Input:
        Alist: sequence of binary words
        𝔠_k: node of 𝔠 = 𝔞 ∧ 𝔟
        L: grade of 𝔠
        d: dimension
        lPathNum: number of ones in the label of the node of 𝔠
        depth: depth of the current node
    Result: 𝔠 = 𝔞 ∧ 𝔟
    // lPathNum represents the grade of the node
    if lPathNum == L then
        i ← 0
        foreach label u of Alist do
            v ← 2^(d−depth) · u                    // from node to leaf
            𝔠_k ← 𝔠_k + Sign[i] · 𝔞_v · 𝔟_reverse(v,d)
            i ← i + 1
    else
        list ← { }
        depth ← depth + 1
        if 𝔠_k is not a leaf then
            list ← oneLevelOuterProduct(Alist,1)
            perGradeOuterP(list, 𝔠_k1, L, d, lPathNum + 1,depth)
            list ← oneLevelOuterProduct(Alist,0)
            perGradeOuterP(list, 𝔠_k0, L, d, lPathNum, depth)
```

the hamming weight of a. Instead of computing the leading formula, we give an upper bound to the number of total right-shifting. From the Equation 2.1, an upper bound to the coefficient is 2^d. Therefore, for any label $a \geq 1$, the upper bound of the maximum number of right-shifting is $\lfloor log_2(2^d) \rfloor = d$. Thus, the number of right-shifting for the total number of occurrences of the label a is bounded as follows:

$$2^{d-h(a)} \cdot \lfloor log_2(a) \rfloor \leq d \cdot 2^{d-h(a)} \tag{3.9}$$

Finally, the total number of right-shifting is obtained by summing over the number of coefficients in a multivector:

$$\sum_{a=1}^{2^d-1} 2^{d-h(a)} \cdot \lfloor log_2(a) \rfloor \leq \sum_{a=1}^{2^d-1} d \cdot 2^{d-h(a)} \tag{3.10}$$

Due to the linearity of the summation, we can rearrange the upper bound by extracting the dimension the following way:

$$\sum_{a=1}^{2^d-1} 2^{d-h(a)} \cdot \lfloor log_2(a) \rfloor \leq d \cdot \sum_{a=1}^{2^d-1} 2^{d-h(a)} \tag{3.11}$$

Here, the Hamming weight $h(a)$ is ranging from 1 to d. The number of coefficients whose hamming weight is k, is $\binom{n}{k}$. Hence the upper bound can be rewritten as $\sum_{k=1}^{d} \binom{n}{k} 2^k$. From the binomial theorem, the upper bound is thus proportional to $d \cdot 3^d$.

We now introduce our method to reduce this number of arithmetic operations. The key point of the method lies in the fact that the sequence of signs over the binary tree remains the same for any coefficient $\mathfrak{c}$. This is explained by the structure of the recursive definition of the sign explained in [4]. The recursive definition of the sign is as follows:

$$\begin{array}{l} \text{if } n < d \ , \ \overline{\mathfrak{a}_u}^n = \ \left(- \overline{\mathfrak{a}_{u1}}^{n+1}, \overline{\mathfrak{a}_{u0}}^{n+1} \right) \\ \text{if } n = d \ , \ \overline{\mathfrak{a}}^n \ = \ \mathfrak{a} \end{array} \tag{3.12}$$

This formula combined with the recursive definition of the outer product *Alist* is the following:

$$\begin{array}{l} \text{if } n < d \ , \ \left\{ Alist^n \right\} = \\ \left(\left\{ Alist(1)^{n+1}, \overline{Alist(0)}^{n+1} \ \right\}, \left\{ Alist(0)^{n+1} \right\} \right) \\ \text{if } n = d \ , \ \left\{ Alist^n \right\} = \ \left\{ Alist \right\} \end{array} \tag{3.13}$$

From Equation (3.13), the sequence of signs is left unchanged for each right subtree. Therefore, the sequence of signs is completely determined by the sequence of signs of the far left leaf of $\mathfrak{c}$.

Thus, we only have to compute the sequence of signs for the far left leaf and store the sequence. Then for each leaf, the outer product algorithm goes through the elements of this sequence. Algorithm 4 gives pseudo-code for our method.

In this algorithm, the variable *comp* enables the flip of sign. From this algorithm, the number of operations to be performed is 2^d for a d-dimensional space. Therefore, using this algorithm in the computation of the outer product takes 3^d proportional time.

In order to confirm these results, the two algorithms described in this section will be compared through some tests in Section 5.

Algorithm 4: Computation of the sequence of signs

```
Function SignOuter
    Input:
        Sign: list to store the resulting signs
        currentSign: sign
        comp: complementary operator
        depth: depth of the current node
        d: dimension
    if depth == d then
        Sign.push(currentSign)
    else
        SignOuter(comp × currentSign, comp, depth + 1, d)
        SignOuter(currentSign,−comp, depth + 1, d)
```

4. Geometric product

4.1. Euclidean space

We now consider the computation of the geometric product of two multivectors $\mathfrak{a}$ and $\mathfrak{b}$ in an Euclidean space. We first describe the geometric product in an Euclidean space and then in an non-Euclidean space. We denote the geometric product between $\mathfrak{a}$ and $\mathfrak{b}$ by $\mathfrak{c} = \mathfrak{a} * \mathfrak{b}$. The overall method is equivalent to the method described in Section 3.1.2. The labelled recursive equation of the geometric product gives:

$$\begin{aligned}
&\mathfrak{a}_u^n * \mathfrak{b}_v^n = (\mathfrak{a}_{u1}^{n+1}, \mathfrak{a}_{u0}^{n+1})^n * (\mathfrak{b}_{v1}^{n+1}, \mathfrak{b}_{v0}^{n+1})^n \\
&\text{if } n < d \ , \ \mathfrak{a}_u^n * \mathfrak{b}_v^n = \\
&\ (\mathfrak{a}_{u1}^{n+1} * \mathfrak{b}_{v0}^{n+1} + \overline{\mathfrak{a}}_{u0}^{n+1} * \mathfrak{b}_{v1}^{n+1} \ , \overline{\mathfrak{a}}_{u1}^{n+1} * \mathfrak{b}_{v1}^{n+1} + \mathfrak{a}_{u0}^{n+1} * \mathfrak{b}_{v0}^{n+1})^n \\
&\text{if } n = d \ , \ \mathfrak{a}_u^n * \mathfrak{b}_v^n = \ \mathfrak{a}_u^d * \mathfrak{b}_v^d
\end{aligned} \tag{4.1}$$

The development of this recursive formula in a 3-dimensional space is presented in Table 3, where the sign of each product is computed with a equivalent method used for the outer product, see Algorithm 5.
As for the outer product, we extract a recursive construction of the set of labels of $\mathfrak{a}$ and $\mathfrak{b}$. *Alist* and *Blist* are again these recursive constructions of labels. The recursive definitions of *Alist* and *Alist* are the following:

$$\begin{aligned}
&\text{If } n < d \ , \ \{Alist^n\} = \\
&\ \big(\{Alist(1)^{n+1}, Alist(0)^{n+1}\}, \{Alist(0)^{n+1}, Alist(1)^{n+1}\}\big) \\
&\text{If } n = d \ , \ \{Alist^n\} = \ \{Alist\}
\end{aligned} \tag{4.2}$$

$$\begin{aligned}
&\text{If } n < d \ , \ \{Blist^n\} = \\
&\ \big(\{Blist(0)^{n+1}, Blist(1)^{n+1}\}, \{Blist(0)^{n+1}, Blist(1)^{n+1}\}\big) \\
&\text{If } n = d \ , \ \{Blist^n\} = \ \{Blist\}
\end{aligned} \tag{4.3}$$

$\mathbf{e}_{123}$	$\mathbf{e}_{12}$	$\mathbf{e}_{13}$	$\mathbf{e}_{1}$	$\mathbf{e}_{23}$	$\mathbf{e}_{2}$	$\mathbf{e}_{3}$	$\mathbf{1}$
c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
a_7b_0	a_7b_1	$-a_7b_2$	$-a_7b_3$	a_7b_4	a_7b_5	$-a_7b_6$	$-a_7b_7$
a_6b_1	a_6b_0	a_6b_3	a_6b_2	$-a_6b_5$	$-a_6b_4$	$-a_6b_7$	$-a_6b_6$
$-a_5b_2$	$-a_5b_3$	a_5b_0	a_5b_1	a_5b_6	a_5b_7	$-a_5b_4$	$-a_5b_5$
a_4b_3	a_4b_2	a_4b_1	a_4b_0	a_4b_7	a_4b_6	a_4b_5	a_4b_4
a_3b_4	a_3b_5	$-a_3b_6$	$-a_3b_7$	a_3b_0	a_3b_1	$-a_3b_2$	$-a_3b_3$
$-a_2b_5$	$-a_2b_4$	$-a_2b_7$	$-a_2b_6$	a_2b_1	a_2b_0	a_2b_3	a_2b_2
a_1b_6	a_1b_7	$-a_1b_4$	$-a_1b_5$	$-a_1b_2$	$-a_1b_3$	a_1b_0	a_1b_1
a_0b_7	a_0b_6	a_0b_5	a_0b_4	a_0b_3	a_0b_2	a_0b_1	a_0b_0

TABLE 3. Geometric product table in a 3-dimensional space.

4.1.1. Iterative construction of *Alist* and *Blist*.

In order to extract a construction, an analysis of these recursive definitions is performed. Figures 4 and 5 shows the development of these recursive formula in a 3-dimensional space.

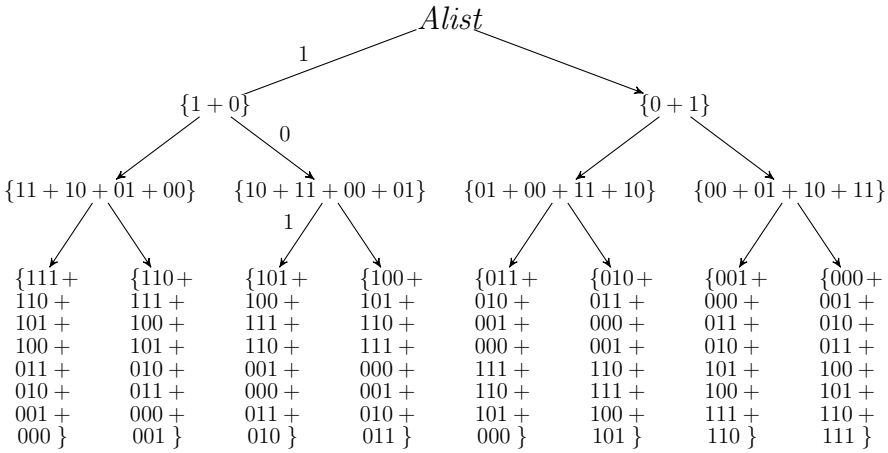

FIGURE 4. *Alist* development in a 3-dimensional space.

Let us now analyse these formula. Firstly, for any recursion level on *Alist* and *Blist*, the number of labels is multiplied by 2. Hence, the number p of products for a d-dimensional space is the following:

$$p = 2^d \tag{4.4}$$

Moreover we observe that each subtree is suffixed by the same binary word. We also observe and we can show that the obtained labels are composed of the binary words whose length n is ranging from $(\underbrace{00\cdots00}_{n})$ to $(\underbrace{11\cdots11}_{n})$.

Finally the construction of labels of *Alist* is extracted from Equation (4.2). We first define a list of labels *Rlist*, composed of consecutive binary words whose

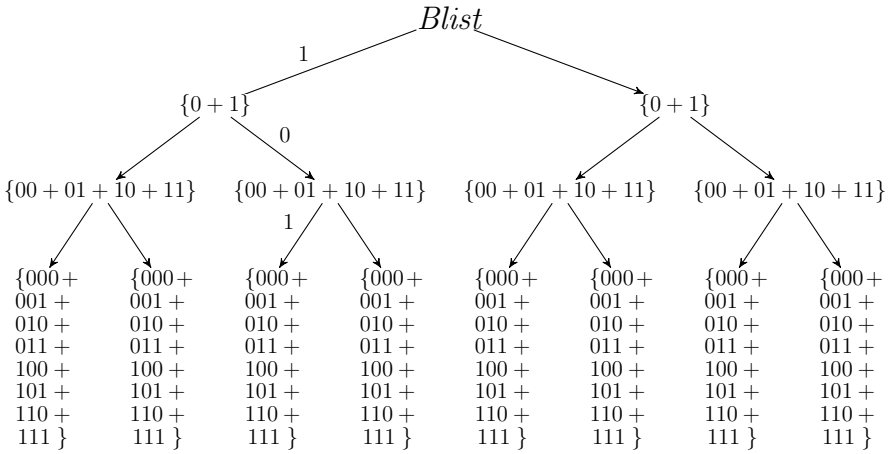

FIGURE 5. *Blist* development in a 3-dimensional space.

length is n and ranging from $(\underbrace{00\cdots00}_{n})$ to $(\underbrace{11\cdots11}_{n})$. Therefore, the labels of *Alist* are determined by the bitwise XOR logical operation ($\oplus$ operator) between elements of *Rlist* and the binary word of the considered node. For example, the construction of *Alist* for the node 101 is presented in Figure 6.

Rlist	k	*Alist*
000	101	$101 \oplus 000 = 101$
001	101	$101 \oplus 001 = 100$
010	101	$101 \oplus 010 = 111$
011	101	$101 \oplus 011 = 110$
100	101	$101 \oplus 100 = 001$
101	101	$101 \oplus 101 = 000$
110	101	$101 \oplus 110 = 011$
111	101	$101 \oplus 111 = 010$

FIGURE 6. Construction of *Rlist* and *Alist* for the node 101.

The method to construct the geometric product leads to Algorithm 5. The computation of the sign is adapted from Algorithm 4 and shown in Algorithm 5.

4.2. Non-Euclidean space

In this section, we show the construction of the geometric product for a non-Euclidean space. We assume in this paper that the basis used is orthogonal. If non-orthogonal basis is needed, then a change of basis can be performed, similarly to what is explained in [1]. In order to construct the geometric product, the quadratic form ϕ is required. The representation of the quadratic

Algorithm 5: Geometric product corresponding to a coefficient $\mathfrak{c}_k$

```
Data: 𝔞, 𝔟, 𝔠: multivectors,
Alist, Blist: sequences of binary words,
k: label of a coefficient of the multivector 𝔠,
d: dimension
Result: 𝔠_k: k^th coefficient of the multivector 𝔠 = 𝔞 * 𝔟
Alist ← { }
Blist ← { }
for binaryWords from 0 to 2^d − 1 do
    Alist.push(binaryWords ⊕ k)
    Blist.push(binaryWords)
Sign ← SignGeo(1, 0, k, d, 1) 𝔠_k ← 0
i ← 0
foreach label u, v of Alist and Blist do
    𝔠_k ← 𝔠_k + Sign[i] · 𝔞_u · 𝔟_v
    i ← i + 1

Function SignGeo
    Input:
        CurrentSign: sign
        level:Integer
        k:coefficient
        d: dimension
        comp: Integer
    Result: Sign: sequence of signs
    if level==d then
        Sign = Sign.push(CurrentSign)
    else
        b ← level^th bit of k
        if b==1 then
            SignGeo(comp * currentSign, level + 1, k, d , comp)
            SignGeo(currentSign, level + 1, k, d , −comp)
        else
            SignGeo(comp * currentSign, level + 1, k, d , −comp)
            SignGeo(currentSign, level + 1, k, d , comp)
```

used in the sequel is picked up from [4]. The values of this quadratic form are represented with d-tuple. We aim now at determining the construction of the geometric product with this quadratic form. In [4], Fuchs and Théry

define the geometric product with the quadratic form as follows:

$$\begin{aligned}
&\mathfrak{a}_u^n * \mathfrak{b}_v^n = (\mathfrak{a}_{u1}^{n+1}, \mathfrak{a}_{u0}^{n+1})^n * (\mathfrak{b}_{v1}^{n+1}, \mathfrak{b}_{v0}^{n+1})^n \\
&\text{if } n < d \ , \ \mathfrak{a}_u^n * \mathfrak{b}_v^n = \\
&\ (\mathfrak{a}_{u1}^{n+1} * \mathfrak{b}_{v0}^{n+1} + \overline{\mathfrak{a}}_{u0}^{n+1} * \mathfrak{b}_{v1}^{n+1} \ , \mathfrak{a}_{u0}^{n+1} * \mathfrak{b}_{v0}^{n+1} + \phi_{n+1}\overline{\mathfrak{a}}_{u1}^{n+1} * \mathfrak{b}_{v1}^{n+1})^n \\
&\text{if } n = d \ , \ \mathfrak{a}_u^n * \mathfrak{b}_v^n = \ \mathfrak{a}_u^d * \mathfrak{b}_v^d
\end{aligned} \tag{4.5}$$

where $\phi_{n+1} = \phi(\mathbf{e}_{n+1}, \mathbf{e}_{n+1})$ denotes the $(n+1)^{th}$ element of this quadratic form, and represents the squared of $\mathbf{e}_{n+1}$.

Firstly, the sequence of labels of $\mathfrak{a}$ and $\mathfrak{b}$ remain the same. Therefore we determine the sequence of ϕ in the binary tree. Let $\phi list$ denote this construction. This construction is computed with a list of elements of ϕ in an equivalent way as the sequence of labels. It is extracted from Equation (4.5) and defined as follows:

$$\begin{aligned}
&\text{If } n < d \ , \ \{\phi list^n\} = \\
&\ (\{\phi list^{n+1}, \phi list^{n+1} \ \}, \{\phi list^{n+1}, \phi^{n+1}.\phi list(1)^{n+1} \ \}) \\
&\text{If } n = d \ , \ \{\phi list^n\} = \ \{\phi list\}
\end{aligned} \tag{4.6}$$

Algorithm 6 shows the pseudo-code for the construction of this sequence.

Algorithm 6: Geometric product corresponding to a coefficient $\mathfrak{c}_k$

```
Function quadraticForm
    Input:
        currentCoef: coefficient
        metric: list of coefficients
        level: Integer
        k: coefficient
        d: dimension
    Result: seq: sequence of metric elements for a leaf c_k
    if level==d then
        seq.push(CurrentCoef)
    else
        φ ← level^th element of metric
        b ← level^th bit of k
        if b==1 then
            quadraticForm(currentCoef, metric, level + 1, k , d)
            quadraticForm(currentCoef, metric, level + 1, k , d)
        else
            quadraticForm(φ * currentCoef, metric, level + 1, k , d)
            quadraticForm(currentCoef, metric, level + 1, k , d)
```

This construction is inserted in the implementation of the geometric product.

4.3. Complexity of geometric product

For each coefficient of $\mathfrak{c}$, the number of sequence construction is proportional to 2^d and the number of products is 2^d, leading to a total number of products of 4^d. This number of products is equivalent to the performance of the geometric product algorithm described in [1] and [4]. However our approach enables us to compute independently each leaf of $\mathfrak{c}$.

5. Experimental results

5.1. Implementations

In order to test our method, we first realized a generalized multivector implementation in C++. This implementation stores the 2^d coefficients in an array and the products are performed using the method proposed in this article. The computation of the products is performed at compile time using metaprogramming described in [9]. We implement also specialized multivector classes to represent some decided geometric objects. In this implementation, multivectors are represented with optimized memory storage, according to their grade. Geometric algebra products are also specialized according to the grade of each operand. This leads to an equivalent of Gaigen. Additionally, the products are vectorized using SIMD instructions described in [8], enabling us to operate up to four leaves of the tree at the same time by using SIMD registers. The SIMD code is inserted on line 6 of Algorithm 3. For each group of 4 operations, a SIMD code is produced. This code consists in shuffling the indices of the two multivectors. Using SIMD code, the addition and multiplication are then performed. Our approach is well suited for this kind of optimizations because our products are explicitly defined for each leaf of $\mathfrak{c}$ (result of the product between $\mathfrak{a}$ and $\mathfrak{b}$). More precisely, from a coefficient of $\mathfrak{c}$, we can directly extract each product of a and b which contributes to the construction of the leaf of $\mathfrak{c}$ which have a known grade. Finally, each coefficient of $\mathfrak{c}$ can be computed independently. We thus benefit from the high degree of parallelizability of our approach.

5.2. Tests

5.2.1. Practical benchmark. We first compared the two approaches to compute the sign described in subsection 3.1.5. Actually we compare three methods. The first computes the outer product without any sign computation. The two last approaches differ in that they do not compute the sign the same way. If we get rid of the sign, they all have the same runtime performance (proportional to 3^d). For these methods, the outer product was computed with general multivectors whose dimension may vary between 5 (conformal geometric algebra space, see [1]) and 10 (double conformal geometric algebra space, see [10]). The leading runtime performance are recorded. We observe

that the ratio between the runtime of the sign computed by convolution and the runtime without sign computation is linear to the dimension. This confirms the result in Section 3.1.5. Another interesting thing is the runtime performance for the sign computed using sequences and the performance of the algorithm whose performance is proportional to 3^d. These two algorithms have the same asymptotic behaviour. This also confirms that the runtime performance is proportional to 3^d.

The resulting implementations are compared, through different tests, to the Gaigen implementation of the general multivector and an another implementation derived from the table based approach explained in [5].

5.2.2. First tests:
This serie of tests consist in computing the geometric product and the outer product between a multivector whose grade is 1 and another whose grade is 2 for different dimensions ranging from 4 to 10. This computation is repeated 10^5 times. Firstly we compare our generalized method to the Gaigen implementation of the general multivector and an another implementation derived from the table based approach. The runtime performance shows an averaged 20% relative gain over the other generalized multivector implementations.

5.2.3. Second tests:
These tests consist in computing the intersection between two hyperspheres whose radii are fixed. The dimensions of the multivectors are ranging from 4 to 10. Here, the 5-dimensional space corresponds to the conformal geometric algebra space (e_0,e_1,e_2,e_3,e_∞), and the 10-dimensional space which has the same number of basis vectors than the double conformal space explained in [10]. From the radius and the centre, we construct these hyperspheres. With this test, the intersection is simply computed with the outer product between the vectors of the two hyperspheres. The computation of each intersection between two hyperspheres is repeated 10^5 times. At each iteration, the position of the second hypersphere is set. The performance is determined by comparing the runtime for the three approaches previously described. The runtime performance for dimensions from 4 to 10 show relative gain ranging from 15% in the 4 dimensional space to 30% in the 9 dimensional space.

Our implementation can now be compared to Gaigen [2]. These specialized implementations are compared with the same protocol used for the generalized version. The specialized version are also tested on the computation of the intersection of two spheres, as described in [5]. The results show here an overall 25% relative gain over Gaigen mainly due to the SIMD implementation.

5.2.4. Third tests:
These tests consists in generating our library in a 15-dimensional spaces. The number of generated functions is quadratic to the dimension. Thus, the size of the leading library is roughly 200 MB and the size of the binary file is 1 GB. That would be the approximate size of Gaigen library if Gaigen generated it. If one used only multiplication tables, the size of the tables would be a problem. If we consider the storage of unsigned integers (32 bits) at each element of this table, then, the memory cost would be approximately 12 GB (32 bits/element$\times 4^{15}$ elements/table $\times 3$ tables). For these high dimensions, none of these methods are suitable for practical use due to the huge memory requirement. However, we can directly use online computations without any precomputed functions. In that case, our recursive method has $O(3^d)$ complexity where the same approach with Gaigen has $O(4^d)$ complexity.

6. Conclusion

In this paper, we present a new method to compute geometric algebra products, defined in any dimension. We show that the proposed method is at least as fast as the main state of the art specialized geometric algebra implementations. As future work, we want to extend our method to handle high dimension applications, namely higher than 15. For these kind of spaces, the number of specialized functions to generate might be too numerous.

References

[1] Dorst, Leo and Fontijne, Daniel and Mann, Stephen, *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry* (2007), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] Fontijne, Daniel *Gaigen 2:: a geometric algebra implementation generator*, Proceedings of the 5th international conference on Generative programming and component engineering,pages 141–150, (2006), ACM.

[3] Perwass, Christian *CLUCal/CLUViz Interactive Visualization [online]*, http://www.clucalc.info/, (2010).

[4] Fuchs, Laurent and Théry, Laurent *Implementing Geometric Algebra Products with Binary Trees*, Advances in Applied Clifford Algebras, pages 22, (2014).

[5] Hildenbrand, Dietmar *Foundations of Geometric Algebra Computing*, Springerr-Verlag Berlin Heidelberg, (2013).

[6] Fontijne, Daniel *GAViewer Documentation Version 0.84*, http://www.science.uva.nl/research/ias/ga/viewer, University of Amsterdam.

[7] Kanatani, Kenichi *Understanding Geometric Algebra: Hamilton, Grassmann, and Clifford for Computer Vision and Graphics*,CRC Press, (2015).

[8] Fog, Agner *Optimizing software in C++ An optimization guide for Windows, Linux and Mac platforms*, (2014).

[9] Abrahams, David and Gurtovoy, Aleksey *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley Professional (2004).

[10] Easter, Robert Benjamin and Hitzer, Eckhard *Double Conformal Geometric Algebra for Quadrics and Darboux Cyclides.*, CGI2016.

[11] Clifford, William Kingdon*Applications of Grassmann's Extensive Algebra.*, American Journal of Mathematics Pure and Applied 1, 350-358 (1878).

Stéphane Breuils
Laboratoire d'Informatique Gaspard-Monge, Equipe A3SI,
UMR 8049, Université Paris-Est Marne-la-Vallée, France
e-mail: `stephane.breuils@u-pem.fr`

Vincent Nozick
Laboratoire d'Informatique Gaspard-Monge, Equipe A3SI,
UMR 8049, Université Paris-Est Marne-la-Vallée, France
JFLI UMI 3527,
CNRS, NII, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
e-mail: `vincent.nozick@u-pem.fr`

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