GARAMON: GEOMETRIC ALGEBRA LIBRARY GENERATOR

Stéphane Breuils and Vincent Nozick and Laurent Fuchs

LIGM (UMR 8049),
F-77454, Marne-la-Vallée, France
stephane.breuils@u-pem.fr

JFLI (UMI 3527), CNRS, NII,
Tokyo 101-8430, Japan
vincent.nozick@u-pem.fr [presenter, corresponding]

XLIM-ASALI (UMR 7252), CNRS,
Université de Poitiers, Poitiers, France
Laurent.Fuchs@univ-poitiers.fr

ABSTRACT. This paper presents Garamon, a C++ library generator synthesizing efficient C++ libraries implementing Geometric Algebras in both low and higher dimensions, with any arbitrary constant metric. The generator is designed to produce easy to install, easy to use, effective and numerically stable libraries. For low dimension vector spaces, the Geometric Algebra products are precomputed, and smoothly switch to recursive product computations for higher dimensions. The recursive scheme is based on an optimized prefix tree structure that naturally encodes dual multivectors.

1. INTRODUCTION

Many implementations of geometric algebra exist taken 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 [7, 19, 16] or in papers dealing with implementations [10, 1, 3, 14].

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++. It could be compared to generators producing C++ programming code like Gaigen [13], Gaalop [5], GMac [11, 10] and libraries written in C++ programming language like Gaalet [20], Versor [6] and GluCat [17].

Those softwares differ mainly by the way they are representing the multivectors and optimizing 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, very sparse. So, to be efficient, a GA implementation may tend to represent as few information as possible, and to design operator algorithms that use efficiently this information. To do that, different strategies have been conducted. Gaigen [13] generates optimized libraries defined from an algebra specification. Gaalop [5] and Gmac [11, 10] are producing optimized code fragments from the description of an algorithm in a domain specific language. The drawback is the necessity to manually integrate the generated code into the target program, that can become annoying when investigating some algebra equations. Gaalet [20] and Versor [6] 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 efficient code. GluCat is following another way using real matrix representation for Clifford algebra [18] and a dedicated
version of fast Fourier transform to improve Clifford product. GluCat was benchmarked and found its performance to be similar to CLU [12].

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

2. LIBRARY GENERATOR

Garamon 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 be user friendly and efficient both in term of computation speed as well as memory consumption. The full project is available online\(^1\).

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 hardcoded limit to dimension 31. The “low dimensional” operations are precomputed, whereas the “high dimension” computations run on a new recursive scheme based on a prefix tree multivector representation. This prefix tree representation presents some very effective optimization 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 still include some precomputed instructions for some products.

2.2. User friendly. The generated libraries are dedicated to be 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 [15]. 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 GA libraries can be used together.

3. MULTIVECTOR DATA STRUCTURE

3.1. 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) strongly differs from the information represented in GA (multivectors). For linear algebra, it is of order \(O(d^2)\) whereas for it is of order \(O(2^d)\) for GA. 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 [13, 5, 20, 6]. In this paper, a different approach is followed by storing multivector elements by grade.

More specifically, a multivector is considered 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 zero values, as depicted in figure 1. This choice is motivated by the fact that most GA entities consist in homogeneous multivectors, i.e. multivectors with

\(^1\)git clone https://git.renater.fr/garamon.git
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 GA elements as per grade arrays is also motivated by some code optimization using SIMD registers. In that case, even storing some zero often does not affect the computation speed since the SIMD registers process several multivector product operations simultaneously.

4. Products in Low Dimensional Space

4.1. Per grade products. Considering the per grade data structure defined in section 3.1, the most efficient way to process any product is to precompute it in advance. Since the outer, inner and geometric products are distributive over the addition, each “per grade product” can be extracted and computed independently. Let \( x_{(k)} \) be the part of the multivector \( x \) of grade \( k \), and \( D_x = \{ x_{(k)} \neq 0 \} \) be the set of all \( k \)-vector \( x_{(k)} \) of any grade present in \( x \). 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).

**Algorithm 1: Per grade loop**

**Input**: multivectors \( a \) and \( b \),

a product \( \odot \) distributive over the addition

**Output**: multivector: \( c = a \odot b \)

1. **foreach** \( k \)-vector \( a_{(k_a)} \in D_a \) do
2.     **foreach** \( k \)-vector \( b_{(k_b)} \in D_b \) do
3.         \( k_c = \text{find\_grade}(\odot, k_a, k_b) \)
4.         \( c_{(k_c)} = \text{product\_k}_{a\_k_b}(\odot, a, b) \)
5. **return** \( c \)

In practice, these two loops are likely to contain only one call, in the case where \( a \) and \( b \) are homogeneous multivectors. Moreover, each product \( \text{product\_k}_{a\_k_b} \) can be precomputed in advance, according to the specified GA signature. More details about this embedded functionality is given in section 7.
4.2. **SIMD instructions.** As presented in Algorithm 1, a multivector product is divided in sub-products of homogeneous grade. Each sub-product \( c_{(k_i)} = a_{(k_i)} \odot b_{(k_i)} \) consists in a list of atomic basic contributions of the form \( c_l = wa_nb_n \) where \( c_l, w, a_m \) and \( b_m \) are elements of the base field used to build the algebra. These instructions are easily converted to C++ code but the conversion to SIMD is not straightforward since some low level memory constraints should be considered. First, “writing” a new value in a variable is more costly than just reading it. Second, dealing with consecutive array elements should be highly preferred. Moreover, a specific care must be given to data cache transfer minimization.

For a given product, all the atomic instructions \( c_l = wa_nb_n \) are sorted according to first the resulting k-vector element index \( l \), then if necessary according to the left and right operand index \( n \) and \( m \).

In the current version of Garamon, the SIMD instructions uses only `mavx` intrinsics and avoid `mavx2` functions such that it can run on almost any computer. In case of incompatibility, the SIMD instructions can just be disabled such the program automatically use instead the default C++ functions.

In practice, the SIMD implementations of GA products are actually not as impressive as expected and leads to roughly the same performances as the regular C++ version compiled with \(-O2\) option. However, further investigations may lead to a more consequent speed up.

4.3. **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:

\[
a^* = a \cdot I^{-1} = \frac{a \cdot \tilde{I}}{1 \cdot \tilde{I}}
\]

This expression requires the computation of two inner products, a reverse and a scalar division, that can be precomputed in advance. 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 3.1, 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 precomputing both an array that stores the required permutation for each array and a vector that stores the coefficients to apply to each resulting array. These operations are well suited to SIMD optimization.

5. **HIGH DIMENSION VECTOR SPACE**

As presented in [3], the common optimization of GA products used in most of the GA libraries fail in high dimension space, due to memory overload or to complexity issue. Breuils et al. [2] detailed how a binary tree can represent efficiently multivector components and lead to a 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.

5.1. **Multivectors and prefix trees.** As detailed in [3], a multivector can be represented by a binary tree. This approach can be modified to represent a multivector with a prefix tree, where each node is associated to a basis vector. More precisely, the nodes of depth \( k \) of the tree represent all the basis vector of grade \( k \). Thus, the root node, denoted by \( 1 \), represents the scalar
part of a multivector, the children of the root node are denoting the vector part, the children of those nodes are denoting the bivector part, and so on. Moreover, by nature of the prefix tree structure, each label of basis vector of the tree is prefixed by the labels its parents, as illustrated in figure 2, where \( e_{ijk} \) stands for \( e_i \wedge e_j \wedge e_k \).

![Figure 2. Prefix tree data structure.](image)

Each node of the tree is labeled by a binary word encoding the basis vector representing the node. This binary labeling, where each bit represents a basis vector, is used in numerous GA implementations [11, 5, 13]. Traversing a prefix tree representing a multivector can be achieved using an index called \( \text{msb} \) (most significant blade). This index represents the binary label of the last basis vector (of grade 1) encountered from the previous node to reach the considered node. Thus, this index actually contains only a single bit to 1. Consequently, the label of a child of a node with binary identifier \( \text{label} \) and index \( \text{msb} \) is computed by:

\[
(5.1) \quad \text{child\_label}(\text{label}, \text{msb}) = \text{label} + \text{msb}
\]

where + is the binary addition. In this formulation, the contribution of \( \text{msb} \) corresponds to the most significant bit of \( \text{child\_label}(\text{label}, \text{msb}) \).

We can note that this tree representation is not well suited for an efficient data storage due to the difficulty to cut useless parts of the tree. Therefore, Garamon includes a mapping from the tree representation to the array structure defined in section 3.1. This mapping consists in two precomputed lookup tables that extract both the grade and position on the array of a given label.

5.2. Dual and prefix tree. Given a multivector \( a \), then the nodes at depth \( k \) of the prefix tree represent the components of \( a \) of grade \( k \). Hence, the root of the tree is always the scalar component, and the deepest leaf corresponds to the pseudoscalar component of \( a \). It is noteworthy to observe that this formulation also implicitly describes the dual \( a^* \) of \( a \) by reading the tree from the pseudoscalar leaf to the scalar root, as shown in figure 3. This dual “upside down” representation of the prefix tree involves some basis sign changes adjustments. Moreover, some nodes of the dual prefix tree are affected by the metric of the specified algebra. To include these 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 vector space supporting the algebra.

In practice, the dual tree traversal requires a function to indicate the children of a given node. This function corresponds to a dual version of Eq. (5.1) and is derived from the binary labeling of the nodes, by:

\[
(5.2) \quad \text{dual\_child\_label}(\text{label}, \text{msb}) = \text{label} - \text{msb}
\]

Note that the label of the root is now the binary label \((1 << d) - 1\) where \( << \) is the left shift operator that shifts on the left the digits of a label.
FIGURE 3. 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 3.1. 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 3.1. The main goal of this dual prefix tree is to compute some product between dual multivectors without computing the costly multivector dualization.

6. PRODUCTS IN HIGH DIMENSIONAL VECTOR SPACE

6.1. Recursive products. The recursive products over binary trees, defined by Fuchs and Théry [14] and developed by Breuils et al. [2], can be adapted to the prefix tree with the same efficient time complexity, i.e. $O(3^d)$ for the outer product instead of the usual $O(4^d)$. However, this prefix tree structure is more compact since all the nodes represent a multivector component, where only the leaves of the binary tree store these data. This compact structure presents some advantages concerning the ranges of possible optimization. Indeed, according to the product to be processed, some branches of the tree can be left unvisited. This decision is computed with a very simple rule related to a boundary on the $\text{msb}$ index. Moreover, since the grade corresponding to a node depends on its depth, this depth indicates when to stop the recursive call along the tree, i.e. when the grade of the result multivector is reached.

In the following, we consider the outer product. The inner and geometric products are computed in a very similar way. The outer product $c_{<k>} = a_{<k>} \wedge b_{<k>}$ can be computed recursively in efficiently traversing the prefix tree of $a$, $b$ and $c$, as shown in Algorithm 2. In this pseudocode, $\text{labelToMsb(label)}$ computes the most significant bit from the considered label. The function $\text{gradeKReachable(grade, msb)}$ indicates whether at least a child of a node reached by reading the basis vector ‘$\text{msb}$’ can reach the grade ‘$\text{grade}$’. This function is used to avoid several recursive calls, as shown on Figure 4. Furthermore, as the dimension grows, this number of avoided useless recursive calls grows exponentially.

6.2. 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 2 adapted to the dual tree defined in Section 5.2. 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.

6.3. Hybridization. For high dimension GA, the generated libraries include a soft transition between precomputed products and recursive products. The criteria for a 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
Algorithm 2: Recursive outer product $c_{<k_c>} = a_{<k_a>} \wedge b_{<k_b>}$

Function outer

Input: $a, b$: two multivectors,
$c$: resulting multivector,
$k_a, k_b$ and $k_c$: the respective grade of each multivector.
$\text{label}_a, \text{label}_b, \text{label}_c$: the recursive position on each tree.
sign: a recursive sign index.

if grade($\text{label}_c$) == $k_c$ then // end of recursion
    $c[\text{label}_c] += \text{sign} \times a[\text{label}_a] \times b[\text{label}_b]$
else // recursive calls
    $\text{msb}_a = \text{labelToMsb}(\text{label}_a)$
    $\text{msb}_b = \text{labelToMsb}(\text{label}_b)$
    $\text{msb}_c = \text{labelToMsb}(\text{label}_c)$
    foreach $\text{msb}$ such that gradeKReachable($k_c, \text{msb}$) == true do
        $\text{label} = \text{label}_c + \text{msb}$
        if gradeKReachable($k_a, \text{msb}$) then
            outer($a, b, c, k_a, k_b, k_c, \text{label}_a + \text{msb}, \text{label}_b, \text{label}_c, -\text{sign}$)
        if gradeKReachable($k_b, \text{msb}$) then
            outer($a, b, c, k_a, k_b, k_c, \text{label}_a, \text{label}_b + \text{msb}, \text{label}_c, \text{sign}$)

library over a 15 dimension vector space will have at least the products of vectors implemented with precomputed functions.

Figure 4. Tree structure for a resulting multivector of grade 3 in a 4-dimensional vector space. Useless branches are depicted with gray dashed arrows. The targeted nodes (of grade 3) are surrounded by a black rectangle. We can remark that 6 useless traversals are ignored over 15 theoretic traversals.

7. Non orthogonal metric

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 [10] and proceed to a basis change, however we also
add some numerical robustness preprocessing. As an example, let us consider the Conformal Geometric Algebra of \( \mathbb{R}^2 \) with metric \( M \) and its eigen decomposition \( M = PDP^{-1} \):

\[
M = \begin{bmatrix}
0 & 0 & 0 & -1 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
-1 & 0 & 0 & 0
\end{bmatrix}
= \begin{bmatrix}
0.707 & 0.707 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
-0.707 & 0.707 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
0.707 & 0 & 0 & -0.707 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
\]

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 \( P^{-1} \):

\[
M = \begin{bmatrix}
0 & 0 & 0 & -1 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
-1 & 0 & 0 & 0
\end{bmatrix}
= \begin{bmatrix}
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
-1 & 1 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
0.5 & 0 & 0 & -0.5 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
\]

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 remove all the numerical approximations. The final stage consists in the generation of both transformations and inverse transformation matrix for any grade of the algebra. In practice, these transformation matrices are very sparse and are stored in the efficient eigen sparse matrices [15]. In the case of non-full-rank metrics, the process remains the same, however the dual functions are not generated.

8. 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 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 [5], Gaigen [13] and Versor [6].

8.1. 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. According to [3], the highest dimension usable with Gaalop is around 10, due to memory overload. As stated in [13], 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 \( \mathbb{R}^{4,4} \) [8] and Triple Conformal Geometric Algebra of \( \mathbb{R}^{9,3} \) [9]. For higher dimensional algebra, we tested Garamon on the Quadric Conformal Geometric Algebra [4] 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 [21].
8.2. **Speed computation.** The speed computation tests were conducted on basic operations like outer products $c = a \wedge b$, inner products $c = a \mid b$, or some combinations $d = (a \wedge b) \mid c$. For more complex operations, we expect Gaalop [5] 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 $c = a \wedge b$. 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 libraries, 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 that Gaalop on products such as $d = (a \wedge b) \mid c$.

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 8.3.

8.3. **Memory consumption.** The data memory consumption tests were conducted by generating both a large number of random vectors and bivectors. Let $d$ be the dimension of the vector space supporting the algebra. Table 1 and 2 show that the per-grade arrays has a memory storage roughly linear in $d$ when the full multivector has a memory complexity of $O(2^d)$.

<table>
<thead>
<tr>
<th>dimension</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaalop / Gaigen</td>
<td>12.8</td>
<td>25.6</td>
<td>51.2</td>
<td>102.4</td>
<td>409.6</td>
<td>–</td>
</tr>
<tr>
<td>Versor</td>
<td>4.6</td>
<td>5.0</td>
<td>5.5</td>
<td>6.3</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Garamon</td>
<td>2.1</td>
<td>2.5</td>
<td>2.9</td>
<td>3.4</td>
<td>4.3</td>
<td>6.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dimension</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaalop / Gaigen</td>
<td>12.8</td>
<td>25.6</td>
<td>51.2</td>
<td>102.4</td>
<td>409.6</td>
<td>–</td>
</tr>
<tr>
<td>Versor</td>
<td>7.9</td>
<td>11.8</td>
<td>16.6</td>
<td>22.1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Garamon</td>
<td>5.3</td>
<td>7.9</td>
<td>11.2</td>
<td>15.3</td>
<td>24.7</td>
<td>57.6</td>
</tr>
</tbody>
</table>

In term of binary file size and as an indication, Garamon weights 2.6 MB in dimension 5 and 7.8MB in dimension 18.

9. **Conclusion**

This paper presents Garamon, a geometric algebra library generator synthesizing C++ libraries implementing geometric algebras of low and high dimensions for any arbitrary metrics. A first objective of Garamon is to be as user friendly as possible, with an easy installation procedure and intuitive commands. Moreover, the user do not have to handle any basis changes relative
to Geometric Algebra products. A second objective of Garamon is to have good computational performances in term of speed and memory consumption. 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 target application can be optimized only once with Gaalop.

REFERENCES