Interface Specification and Implementation Internals of a Program Module for Geometric Algebra

M. D. Zaharia [a,*], L. Dorst [a]

[a] *Informatics Institute, Faculty of Sciences, University of Amsterdam, Kruislaan 403, 1098 SJ, Amsterdam, The Netherlands*

Abstract

This paper describes a method to implement the main abstract operations characteristic to Clifford algebra. The design goal was to maintain the algorithms simple, their specifications correspond mostly to the basic definitions. The article is addressed to those who want to understand the low-level processing mechanisms underlying the abstract concepts of geometric algebra. Efficiency considerations did not prevail in the present implementation. The package presented is freely available (at source level) and may assist the development of C/C++ applications using the geometric algebra formalism. It could be easily extended in order to fit the specific requirements of a given application. The implementation and correctness proof of a blade factorization algorithm are also presented.

Key words: Clifford algebra, geometric algebra, abstract data type, program module, blade factorization

* Corresponding author. *Tel*: +31 (20)-525.75.17, *Fax*: +31 (20)-525.76.90,
Email address: marius@science.uva.nl (M. D. Zaharia).
URL: http://carol.science.uva.nl/∼marius (M. D. Zaharia).

1 Introduction

Geometric algebra is a formalism capable of expressing diverse geometric ideas. The axioms specific to Clifford algebra and the geometric semantics associated to the algebraic manipulation rules together constitute the foundations of a language suitable to describe problems from various science fields like:

- mathematics (a special mention for the domain of oriented projective geometry [23,22] whose results are applicable in computational geometry problems, linear algebra and determinant theory [3] or geometric function theory that integrates results from real and complex analysis [14]).
- physics (the Pauli and Dirac algebras from quantum mechanics are Clifford algebras suitable to solve specific problems, other diverse applications from crystallography to relativity theory have been presented at geometric algebra conferences [8,7]).
- computer science and engineering (problems in various fields like computer graphics [11,24,25], image processing, computer vision [8,7], were formulated using geometric algebra language, lots of numerical recipes could benefit from the geometric algebra formalism [1]).

This language provides algebraic representations of geometric primitives like points, lines or planes and of relations like incidence or duality [16]. There are even different geometric interpretations that can be associated with all (or only parts) of geometric algebra elements. Sometimes, the research approaches based on geometric algebra bring the elegance and "compactness" specific to synthetic geometry reasoning, in solutions of different problems. The coordinate-free, dimension independent way to describe specific relations that characterize the structure or behavior of different systems constitutes another strength of this formalism.

This paper considers already known the basics of *Clifford algebra*. The interested reader may follow [12,13,15,18]. The main difference between Clifford algebra and geometric algebra is that the latter emphasizes the geometric semantic of algebraic entities (blades, versors, different products etc.) From a formal point of view, the primitives whose implementation is described below are Clifford algebra primitives (they refer the pure algebraic manipulation) but the applications requiring their usage could belong to the geometric algebra field. That is why the reader should not be disturbed by the mixed appearance of both terms in this paper.

The present description of the program module (named GAP [1]) tries to initiate the readers to the detailed manipulation mechanisms specific to geometric

[1] GAP is the acronym of *Geometric Algebra Package*. It is intended to fill the gap between the abstract theory and the application programmer.

algebra and to help them develop own application programs using the abstract operations characteristic to a geometric algebra environment.

This paper should not be considered a user manual. It concerns especially the internals of the implementation and the rational behind the design. More detailed information could be obtained directly from the source program that is freely available (http://carol.wins.uva.nl/∼marius).

A computer algebra system known to the geometric algebra research community is *CLICAL* (developed by P. Lounesto, [20]). It is mainly interactive and is supposed to familiarize the novice user with algebraic concepts such as complex numbers, quaternions or octonions and with their manipulation. Another approach, introducing the geometric algebra concepts is *GABLE* ([21]). It is an interactive, MATLAB based application that allows the user to describe algebraic computations and to visualize their geometric interpretation, limiting itself to geometric algebras defined for 3D Euclidean space.

An interesting approach that constitutes the foundation of an implementation of Clifford algebra primitives (the computational system 'Clifford') is presented in [2]. In contrast with it, the present module treats mainly the case of non-degenerate Clifford algebras whose elements are expressed in an orthogonal basis ($\mathcal{C}\ell_{p,q}$ algebras). The implementation follows an "algorithm driven" approach and this distinguishes it from the work of Ablamowicz ([2]) that could be considered "data driven". However the functionality of our package can be easily extended to support the case of "Clifford algebras whose bilinear forms have antisymmetric parts" and to follow the implementation approach suggested in [2]. Sometimes the work in non-orthogonal bases can lead to more elegant interpretations of the results. That is why this paper refers also to the possibility to extend the functionality of the present module in order to make it sufficiently flexible to satisfy the requirements of different applications. As an example the module GAP was extended to support the null basis of the conformal space. The way this particular extension was incorporated in the package is also presented here. This may constitute a model for analogous developments initiated by the end user.

A brief description of the main notations and concepts used in this paper follows.

If V^n is an n-dimensional *vector space* over the real field R, its associated *geometric algebra* G_n is generated by defining the *geometric product* that, over vector operands, is:

- associative
  $\mathbf{a}(\mathbf{bc}) = (\mathbf{ab})\mathbf{c}$
- distributive
  $\mathbf{a}(\mathbf{b} + \mathbf{c}) = \mathbf{ab} + \mathbf{ac}$

$(\mathbf{b}+\mathbf{c})\mathbf{a} = \mathbf{ba}+\mathbf{ca}$

- and satisfies the conditions:
  $\lambda\mathbf{a} = \mathbf{a}\lambda, \forall\lambda \in R, \mathbf{a} \in V^n$
  $\mathbf{a}^2 \in R$ i.e. $\mathbf{a}^2 = \epsilon_a \mid \mathbf{a} \mid, \forall\mathbf{a} \in V^n$, where $\epsilon_a \in \{+1, 0, -1\}$ is the *signature* of $\mathbf{a}$ and $\mid \mathbf{a} \mid \in R^+$ the modulus of $\mathbf{a}$.

The geometric algebra is said to be *Euclidean* if all its vectors have positive signatures. If vectors with negative signatures exist we will denote by p, q the dimensions of the maximal subspaces of V^n that contain the vectors with positive and negative signatures respectively ($p+q=n$). The corresponding geometric algebra is denoted $G_{p,q}$. If r null basis vectors ($\mathbf{a} \in V^n, \mid \mathbf{a} \mid = 0$) exist the geometric algebra is said to be *degenerate*. It is then denoted $G_{p,q,r}$.

Other specific products (derivable from the geometric product) equally exist: *inner product* (a grade decreasing operator, denoted $\bullet$), *outer product* (a grade increasing, linear operator, anti-commutative when operating on vectors, denoted $\wedge$), *contractions* (denoted $\lrcorner$ or $\llcorner$) as well as other specific operators like *reversion*, *main involution*, *meet* or *join*. The elements of G_n form a 2^n-dimensional algebra, its elements have different *grades* (dimensions), there are for example scalars, vectors, bivectors, trivectors, k-vectors etc. That is why the general elements of the geometric algebra are called *multivectors*; they are sums of homogeneous entities named above k-vectors. All the operators mentioned before are also defined on multivector arguments. A large part of the multivectors from G_n are invertible with respect to the geometric product. An important class of homogeneous multivectors are the *blades*, these characterize (from the geometric algebra point of view) the subspaces of G_n. A blade can be written as an outer-product of linearly independent vectors. In G_n, a blade of maximal grade (i.e. grade n for a non-degenerate algebra) is called *pseudoscalar* of G_n. In fact any blade may be considered the pseudoscalar of a certain subspace of G_n.

A simple geometric product operation produces a lot of information of significant geometric semantics. If $\mathbf{A_r}$ and $\mathbf{B_s}$ are two blades, their geometric product has components of different grades starting from $\mid r-s \mid$ until r+s. Selection of the various grades leads to different products, of which we describe in this paper the inner product, the outer-product and the delta product. Others like the commutator product will even not be mentioned below. It is possible that other "non-standard" products be discovered and demonstrate their utility in different applications. This wealth of geometrically significant information within the geometric product allows the condensed, usually elegant specification of geometric algorithms using the geometric algebra primitives. A better understanding of those semantic aspects could probably lead to new techniques of specification, development or verification of geometric applications. That is why the development of program modules like the one presented in this paper helps not only to a better understanding of the details (low levels) of

the geometric algebra primitives but may also be useful to build applications to improve understanding the higher levels of the formalism.

2 Characteristic objects of geometric algebra

Let G_n be a geometric algebra and V^n its associated metric vector space. The implementation refers (implicitly) to the case of orthogonal basis and allows basis vectors with positive, negative or null signatures. In the description of the present paper, the vectors of the basis will be denoted as $\mathbf{e_0}$, $\mathbf{e_1}$, $\ldots \mathbf{e_{n-1}}$. The metric vector space underlying the geometric algebra is specified by its dimension and the signatures of the basis vectors.

```
typedef struct sp{
 int dim_space; //dimension of the underlying vector space
 int *sign;     //signatures of the basis vectors of
                //corresponding dataspace
} *SPACE;
```

Given a n-dimensional vector space `s` (`s`→`dim_space==n`) and an orthogonal basis of `s`, its vectors satisfy the conditions:

$$\mathbf{e_i} \bullet \mathbf{e_j} = 0,\ \forall\ 0 \leq i,j < n\ ,\ i \neq j$$

$$\mathbf{e_i e_j} = \mathbf{e_i} \wedge \mathbf{e_j},\ \forall\ 0 \leq i,j < n\ ,\ i \neq j$$

$$\mathbf{e_i^2} = (s \rightarrow sign[i]) \mid \mathbf{e_i} \mid^2,\ \forall i,\ 0 \leq i < n$$

The signature of $\mathbf{e_i}$ was specified by a C-like notation. If the basis is orthonormal then the supplementary conditions: $\mid \mathbf{e_i} \mid^2$=1, $\forall 0 \leq i < n$ equally hold.

The main algebraic data objects supported by the present module are: elementary blades, homogeneous multivectors, blades and versors.

The representation of the characteristic objects of geometric algebra could emphasize their additive structure (e.g. a multivector is viewed as a sum of homogeneous components) or the multiplicative structure (e.g. blades are outer-products of vectors). Practically every geometric algebra entity admits an additive representation but only subsets of the multivectors of a geometric algebra have also multiplicative representations. Usually, this last category of representations allows expressing elegant solutions and interpretations of geometric problems.

2.1 Elementary blade

These objects are defined as an outer-product of basis vectors: $\mathbf{e_{i_1}} \wedge \mathbf{e_{i_2}} \wedge \ldots \wedge \mathbf{e_{i_k}}$. The indices of the basis vectors forming the k-tuple characteristic to the elementary blade are always kept in their increasing order. An *elementary blade* is represented by:

```
typedef struct ebld{
  PSPACE vsp;
  int grade; //grade of the elementary blade
             //number of components in array basiscomp
  double coef; //coefficient of the elementary blade
  int *basiscomp;
  //array with the numbers (indices) of basis vectors
  //that compose the elementary blade
} *EBLADE;
```

This storage structure keeps the indices of the basis vectors that together constitute an elementary blade. The product that is implicitly assumed to "glue" them is the outer-product. Different algorithms could however assume different products operating between the basis vectors specified by the EBLADE storage structure. For example, when computing a general geometric product, the same storage structure (EBLADE) is used to express a geometric product of basis vectors. In the case of an orthogonal basis: $\mathbf{e_{i_1}} \wedge \mathbf{e_{i_2}} \wedge \ldots \wedge \mathbf{e_{i_k}} = \mathbf{e_{i_1}} \mathbf{e_{i_2}} \ldots \mathbf{e_{i_k}}$ and, in this case only, the geometric product between basis vectors is anti-commuting. (As already mentioned the outer-product is always anti-commutative, even if its factors are not orthogonal.) This property facilitates considerably the evaluation of a geometric product having as operands geometric products of basis vectors.

2.2 Homogeneous multivector

Homogeneous multivectors are linear combinations of elementary blades with the same grade. The homogeneous multivectors of grade k ($0 \leq k \leq n$) form a $\binom{n}{k}$-dimensional vector subspace of V^n. One particular case of homogeneous multivector is a blade; it is a homogeneous multivector that admits a factorization $\mathbf{B}=\mathbf{v_1} \wedge \mathbf{v_2} \wedge \ldots \wedge \mathbf{v_k}$. The blade represents itself a subspace of V^n ($\{\mathbf{x} \in V^n \mid \mathbf{x} \wedge \mathbf{B} = 0\}$). The data type HMV specifies the storage structure of a homogeneous multivector. The array field "components" contains the coef-

ficients of the elementary blades (on the basis of the $\binom{n}{k}$-dimensional vector subspace). These coefficients are stored in the lexicographic order of the blades they characterize. Let S be the set of elementary blades that constitutes a basis of the space of homogeneous multivectors of grade k. Taking into account the lexicographic ordering convention of the combinations, one may notice that this induces a bijective mapping $S \rightarrow \{0, 1, \ldots, \binom{n}{k}\}$.

The blade and the homogeneous multivector have the same storage structure. In fact, from the Clifford algebra point of view there is no significant difference between a blade and a homogeneous multivector. The differences appear mainly when the related geometric aspects are considered.

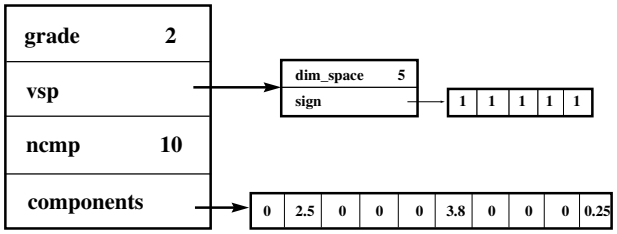

Fig. 1. Storage structure of a homogeneous multivector

For example let us consider in a 5-dimensional space with an associated orthogonal basis $\{\mathbf{e_0}, \mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3}, \mathbf{e_4}\}$ the homogeneous bivector: $\mathbf{B}$=2.5$\mathbf{e_0} \wedge \mathbf{e_2}$+3.8$\mathbf{e_1} \wedge \mathbf{e_3}$ -0.25$\mathbf{e_3} \wedge \mathbf{e_4}$ (in this case n=5, k=2). It may be represented in the "bivector subspace" by the coefficients of the elementary blades that constitute a basis of this subspace. There are $\binom{5}{2}$ =10 such elementary blades, more specifically: $\mathbf{e_0} \wedge \mathbf{e_1}$, $\mathbf{e_0} \wedge \mathbf{e_2}$, $\mathbf{e_0} \wedge \mathbf{e_3}$, $\mathbf{e_0} \wedge \mathbf{e_4}$, $\mathbf{e_1} \wedge \mathbf{e_2}$, $\mathbf{e_1} \wedge \mathbf{e_3}$, $\mathbf{e_1} \wedge \mathbf{e_4}$, $\mathbf{e_2} \wedge \mathbf{e_3}$, $\mathbf{e_2} \wedge \mathbf{e_4}$ and $\mathbf{e_3} \wedge \mathbf{e_4}$. Their coefficients will be stored in the array components exactly in the order specified above i.e. the lexicographical order of the combinations. This ordering assumption will necessitate a set of routines that generates the tuples of the combinations in lexicographic order and establishes the bijective mapping between a tuple (from the $\binom{5}{2}$ possibilities) and an integer index (i$\in$[0, $\binom{5}{2}$]) in the "components" array. Their declarations are:

```
void InitComb(int n, int r, int t[]);
  // generates the first r-tuple combination of n elements
```

```
int NextComb(int n, int r, int t[]);
  // generates next r-tuple combination of n elements.
  // the order is lexicographic
void InitIndexComb1(int n, int r, int t[], int ts[]);
  // generates the index corresponding to a given combination
void CombFromIndex(int index, int n, int r, int t[], int *tsc);
  // generates the combination that corresponds to a given index
```

The storage structure for a homogeneous multivector representation is specified below:

```
typedef struct hcmp {
  int grade;  // grade of the homogeneous multivector
  SPACE vsp;  // characteristics of the underlying vector space
  int ncmp;   // dimension of the blade subspace
  double *components;
  //the coefficients of the blade in the current subspace basis
} *HMULTIVECTOR, *HMV;
```

2.3 Multivector

In Clifford algebra a *multivector* is a sum of homogeneous multivectors. The multivector is represented as a list of (non-zero) homogeneous multivectors that are stored in the increasing order of their grades.

```
typedef struct lcell {
  HMV hcmp;
  //pointer to one homogeneous component of the multivector
  struct lcell *urm;
  // pointer to the next list cell
} *LIST_HCMP;

// Multivector data type
typedef struct mult {
  PSPACE vsp; // characteristics of the underlying data space
  LIST_HCMP hbl;
  // pointer to the homogeneous components list
  // of the multivector
  // the homogeneous components are stored in the increasing
  // order of their grades
} *MULTIVECTOR;
```

If from a computation that produces a homogeneous multivector results a NULL pointer, it will be considered to represent a zero object. Sometimes

the routine performing the computation expects to obtain a result of certain grade, in that case a NULL pointer resulted from the computational process would be interpreted as a zero homogeneous multivector of the grade desired by the calling routine.

2.4 Versor

A *versor* is a geometric algebra entity that can be written as a geometric product of vectors. A blade is a special type of versor, it can be factorized as an outer product of vectors ($\mathbf{B}=\mathbf{v_1} \wedge \mathbf{v_2} \wedge \ldots \wedge \mathbf{v_n}$). That is why the storage structure used to represent a blade is similar to that used for a versor; only the products assumed to glue the vector factors differ. The factors of a blade can always be made orthogonal but they may not coincide with basis vectors. In case of orthogonal factors, the outer-product factorization (characteristic to a blade definition) corresponds to a geometric product factorization (characteristic to a versor definition). The C specification of the corresponding (VERSOR) data type is:

```
typedef struct vers {
  PSPACE vsp; // data space
  int grade; // versor grade
  double **vfact;
  // every row of the array contains the coefficients
  // of a vector factor of the versor
} *VERSOR, *BLADE;
```

This representation emphasizes the multiplicative factors of the definition, but a versor is also multivector and may equally be represented using the MULTIVECTOR data type, which emphasizes the addition between homogeneous components. If we denote by $\mathcal{V}^n$ the versor space over the vector space V^n then obviously $\mathcal{V}^n \subseteq G_n$. The conversion between versors from $\mathcal{V}^n$ and multivectors from G_n is achieved by the functions VTOM() (Versor TO Multivector conversion), BTOM() (Blade TO Multivector conversion) and HMVTOOV() (Homogeneous MultiVector TO Orthogonal Versor factorization). This factorization operation was considered sufficiently powerful and useful in concrete applications (see [25]) in order to deserve a detailed description in a special section (4.8). Further, if we denote by $\mathcal{B}^k$ the space of the blades of grade k, by $\mathcal{M}^k$ the homogeneous multivectors of grade k and by $\mathcal{G}^n$ the multivectors of the geometric algebra G_n then $\mathcal{B}^k \subseteq \mathcal{M}^k \subseteq \mathcal{G}^n$. The module GAP contains the predicate IsBlade() that tests if a given homogeneous multivector is a blade and makes therefore possible the separation of the elements from $\mathcal{M}^k \setminus \mathcal{B}^k$. Its implementation and utility will be detailed later.

3 Classes of operations

The elementary operations used to manipulate the five geometric algebra entities (SPACE, EBLADE, HMV, MULTIVECTOR, BLADE, VERSOR) are grouped in the following classes:

(1) Create/destroy operations (allocate the necessary space of memory for each entity and do the proper initializations)
(2) Input/Output operations
(3) Conversions between different representations
(4) Unary operators (see section 3.2)
(5) Unary predicates
(6) Binary operators (oriented products) (see section 3.4)
(7) Unoriented operators (see section 4.6)
(8) Orthogonal transformations (see section 4.7)
(9) Other operations

Operations from every class are briefly mentioned. Together they form the GAP interface. Emphasis is put on significant examples that show the detailed computations behind the most important abstract operations characteristic to a Clifford algebra. The implementation of the first two classes of operations are not detailed because they were not considered as having a relevant significance from the point of view of understanding geometric algebra concepts. From the operations constituting the third class, only the "Blade to Orthogonal Versor conversion" was detailed in a separate section (4.8). This factorization algorithm is a useful example of application of geometric algebra based techniques to implement higher level algorithms. The next four classes of operations were considered of great relevance for the purpose of the present paper and separate sections were allocated to describe the rationale of their implementation. A separate section (4.9) treats the way this module could be extended to be able to work using non-orthogonal bases representations.

3.1 Conversions between representations of different entities

```
MULTIVECTOR VTOM(VERSOR v);
  // conversion from Versor To Multivector representation
  // VTOM=VersorTOMultivector
MULTIVECTOR VTONNM(VERSOR v);
  // VTONNM=VersorTONonNullMultivector
  // only the non-null vector factors of v are multiplied
  // to give the resulting multivector
VERSOR VersorElimNullFactors(VERSOR v);
```

```
  // returns a new versor formed from the non-null factors of v
HMV ETOHMV(EBLADE e);
  // conversion from an elementary blade to a homogeneous
  // multivector representation
HMV ETOHMV(EBLADE e, double c);
  // conversion from an elementary blade to a homogeneous
  // multivector representation (overloaded)
  // forces a new value of the elementary blade weight
MULTIVECTOR HMVTOM(HMV b);
  // conversion from homogeneous multivector to multivector representation
MULTIVECTOR BTOHMV(HMV h);
  // conversion from blade to homogeneous multivector representation
VERSOR HMVTOOV(HMV v);
  // converts a homogeneous multivector to an orthogonal versor.
  // HMVTOOV=HMVTOOrthogonalVersor
BLADE HMVTOB(HMV b);
  // converts a homogeneous multivector to a blade representation.
  // same as HMVTOOV()
```

3.2 Unary operators

```
double SquaredMagnitude(HMV b);
  // returns the magnitude (squared) of the homogeneous multivector
double SquaredMagnitude(MULTIVECTOR a);
  // returns the magnitude (squared) of a multivector
HMV GradeInvolution(HMV b);
  // the main involution (parity conjugation)
  // of a homogeneous multivector
MULTIVECTOR GradeInvolution(MULTIVECTOR m);
  // the main involution (parity conjugation)
  // of a general multivector
HMV Reverse(HMV b);
  // reverses a homogeneous multivector (Overloaded)
MULTIVECTOR Reverse(MULTIVECTOR a);
  // reverses a general multivector (Overloaded)
VERSOR Inverse(VERSOR v);
  // computes the inverse of the versor v
  // WARNING the versor product is the geometric product
HMV Inverse(HMV b);
  // computes the inverse of a homogeneous multivector
HMV Dual(HMV b);
  // returns the dual of a homogeneous multivector (Overloaded)
MULTIVECTOR Dual(MULTIVECTOR m);
```

```
  // returns the dual of a general multivector (Overloaded)
```

The implementation is based on well known formulas (see [12]) and the algorithms are not computationally expensive. As an example, the *dual* of a blade $\mathbf{B} \in \mathcal{B}^k$ relative to its including space ($\mathcal{B}^n$) is computed with:

$$\mathbf{B}^* = \mathbf{B}\mathbf{I}^{-1} = \mathbf{B} \bullet \mathbf{I}^{-1} \tag{1}$$

where $\mathbf{I}$ denotes the pseudoscalar of this space. The inverse of a unit pseudoscalar coincides with its reverse possibly modulo a sign change that corresponds to a change of orientation. Indeed, remembering our orthonormal basis assumption, the reverse of $\mathbf{I}$ is given by: $reverse(\mathbf{I}) = \mathbf{e_{n-1}e_{n-2}} \dots \mathbf{e_1e_0} = (-1)^{\frac{n(n-1)}{2}}\mathbf{e_0e_1} \dots \mathbf{e_{n-1}}$ and obviously this implies

$$reverse(\mathbf{I})\mathbf{I} = \prod_{i=0}^{n-1} \epsilon_i$$

for a general n-dimensional space. In conformal space, the previous identity gives: $I\tilde{}I = -1$. This implies that: $\mathbf{I}^{-1} = -reverse(\mathbf{I}) = -(-1)^{\frac{n(n-1)}{2}}I$. In the relations above ϵ_i denotes the signature of $\mathbf{e_i}$ (see [24] for other related detailed algebraic computations). The implementation follows:

```
HMV Dual(HMV b)
// Function: computes the dual of a homogeneous multivector
{
  HMV t, pseudscal; int k, p, i;
  if(b==NULL) return NULL;
  pseudscal=CreatePseudoscalar(b->vsp);
  k=(b->vsp->dim_space*(b->vsp->dim_space-1)/2)%2;
  for(i=0, p=1; i<b->vsp->dim_space; i++)
    if(b->vsp->sign[i]<0) p=-p; //computing signatures product
  if(p<0) pseudscal->components[0] = -pseudscal->components[0];
  if(k) pseudscal->components[0] = -pseudscal->components[0];
  t=InnerProduct(b, pseudscal);
  FreeHMV(pseudscal);
  return t;
}
```

3.3 Unary predicates

```
int IsOdd(PMULTIVECTOR m);
  // returns true (1) if the multivector is odd and 0 otherwise
```

```
int IsEven(PMULTIVECTOR m);
  // returns true (1) if the multivector is even and 0 otherwise
int IsBlade(HMV b);
  // returns true (1) if the homogeneous multivector is a blade
int EqualHMV(HMV a, HMV b);
  // returns true (1) if the two homogeneous multivectors are equal
int IsPseudoscalar(HMV b);
  // returns true (1) if the homogeneous multivector b
  // has the same grade as its space dimension
```

A particularly useful predicate IsBlade() tests if a given homogeneous multivector is or is not a blade. Its implementation is detailed in section 4.5. Some algorithms work correctly only when used on blade arguments. The corresponding C routines operate on parameters that instantiate the general HMV data type and consequently could produce wrong results. That is why predicates like IsBlade() can be successfully used in assertions that verify preconditions associated to different algorithms.

3.4 Binary operators and related operations

These operators mainly refer to the characteristic products of the geometric algebra. The products supported by this package are: *geometric product*, *outer product*, the *Hestenes inner product*([12]), *left and right contractions*([19]), *scalar product*, *delta product*([4]). All implementations support both operands of type HMV or both of type MULTIVECTOR.

```
void SetInnerProduct(char c);
  // specifies the currently used  inner product
  // possible values of the argument are:
  // 'H' for Hestenes inner product, c='C' for left contraction
HMV InnerProduct(HMV a, HMV b);
  // computes current inner product of two
  // homogeneous multivectors
MULTIVECTOR InnerProduct(MULTIVECTOR m1, MULTIVECTOR m2);
  // applies the current inner product to two general multivectors
HMV ContractionL(HMV a, HMV b);
  // computes the left contraction between two homogeneous multivectors
MULTIVECTOR ContractionL(MULTIVECTOR m1, MULTIVECTOR m2);
  // left contraction between two arbitrary multivectors
HMV ContractionR(HMV a, HMV b);
  // right contraction between two homogeneous multivectors
MULTIVECTOR ContractionR(MULTIVECTOR a, MULTIVECTOR b);
  // right contraction between two homogeneous multivectors
```

```
HMV InnerProductH(HMV a, HMV b);
  // Hestenes inner product between two homogeneous multivectors
PMULTIVECTOR InnerProductH(PMULTIVECTOR m1, PMULTIVECTOR m2);
  // Hestenes inner product implemented on multivector operands
HMV WedgeProduct(HMV a, HMV b);
  // wedge (outer) product between two homogeneous multivectors
PMULTIVECTOR WedgeProduct(PMULTIVECTOR m1, PMULTIVECTOR m2);
  // outer product between two general multivectors
HMV DeltaProduct(HMV a, HMV b);
  // delta product of two blade operands
  // Precondition: both arguments are blades
EBLADE GeometricProduct(EBLADE a, EBLADE b);
// geometric product between two elementary blades
MULTIVECTOR GeometricProduct(HMV a, HMV b);
  // geometric product between two homogeneous multivectors
  // functions correctly for general Clifford algebras
  // using orthogonal bases
  // and also in the case of conformal null basis
MULTIVECTOR GeometricProduct(MULTIVECTOR a, MULTIVECTOR b);
  // geometric product between two general multivectors
  // functions correctly for orthogonal basis
  // and conformal null basis
double ScalarProduct(HMV a, HMV b);
  // returns the scalar product between two homogeneous multivectors
double ScalarProduct(MULTIVECTOR m1, MULTIVECTOR m2);
  // returns the scalar product between two multivectors
```

The *join* and *meet* form the class of the so-called *unoriented operators*. Two different implementations are supported, the first one is based on [4] and the second uses the classical duality relation between meet and join (see [12]).

```
HMV Join(HMV a, HMV b);
  // Join of two blades using Bouma's algorithm
  // Precondition: both arguments are blades
HMV Meet(HMV a, HMV b);
  // Meet of two blades using Bouma's algorithm
  // Precondition: bothe arguments are blades
HMV MeetandJoin(HMV a, HMV b, HMV *j);
  // returns the Meet as Dual(a).b
  // where the dual is computed with respect to the Join
  // and the Join (*j) is computed with Bouma's algorithm
  // Precondition: arguments a and b are blades
```

Some other useful *auxiliary operations* made available by the package are:

```
HMV GetVersorFactor(int n, PVERSOR v);
  // returns the n-th factor of the versor v.
void RemNullComp(PMULTIVECTOR m);
  // removes all the zero homogeneous components
  // from the multivector m
HMV GetGrade(int g, PMULTIVECTOR m);
  // returns the reference to the homogeneous component
  // of grade g of the multivector m
void AddHMV(HMV b, PMULTIVECTOR m);
  // adds a homogeneous component to a multivector
  // the value referred by m is updated
void AddMultivector(PMULTIVECTOR a, PMULTIVECTOR r);
  // adds the multivector a to r the value of r is updated
```

In geometric algebra, any *orthogonal transformation* acting upon a subspace (denoted $\mathbf{x}$) is expressed in a uniform way using the geometric product. Usually the transformation is described by a versor $\mathbf{B}$. The algebraic specification of the transformed subspace is given by: $\mathbf{BxB}^{-1}$. In the Euclidean space an even versor corresponds to a rotation and an odd versor corresponds to a reflection. The application of an orthogonal transformation to a blade operand is accomplished by the functions:

```
HMV ApplyOrthoTransform(HMV b, HMV x);
  // computes the transformed HMV as: bxb^{-1}
HMV ApplyOrthoTransform(VERSOR v, HMV x);
  // computes the transformed blade as vxv^{-1}
MULTIVECTOR ApplyOrthoTransform(HMV b, MULTIVECTOR x);
  // applies an orthogonal transformation (specified by a HMV)
  // to a multivector argument
MULTIVECTOR ApplyOrthoTransform(VERSOR v, MULTIVECTOR x);
  // applies an orthogonal transformation (specified by a versor)
  // to a multivector argument
```

4 Implementation internals

In realizing the implementation of all these operations, the emphasis was put on the clarity of the algorithms that generally express the semantics of the usual definitions. The purpose of this presentation is mainly to make the reader understand the mechanisms that are behind the main abstract operations of the geometric algebra and are often hidden by the formal notations used in the specialized literature. The efficiency (timing) considerations were

secondary. Other specific implementations (of geometric algebra based computational systems) presented in the specialized literature are those of GABLE and GAIGEN. The first one treats the case of 3-dimensional Clifford algebras of arbitrary signatures. That allows the usage of 8×1 arrays for multivector representation and 8×8 matrices for multivectors when they are first operands of elementary products (see [21]). This kind of representation emphasizes the correspondence between the elementary geometric algebra products and linear transformations. The system GAIGEN uses special bit oriented techniques and tries to eliminate the branching instructions (see [10]). The representation is suitable to be efficiently implemented in parallel environments. This achieves superior efficiency but makes the internal mechanisms more difficult to understand. By contrast, GAP approach emphasizes the combinatorics specific to every abstract operation and tries to use the most intuitive storage representations.

4.1 The outer product implementation

The outer product denoted by the symbol $\wedge$ (wedge) is a linear and antisymmetric product on vectors that is extended to blades by associativity; its geometric semantics is specified in [13]. The properties above allow a straightforward computation. Our implementation supports the following cases:

- both operands are homogeneous multivectors.
- both operands are multivectors

Let $\mathbf{a}=\mathbf{e_{i_1}} \wedge \mathbf{e_{i_2}} \wedge \ldots \wedge \mathbf{e_{i_p}}$ and $\mathbf{b}=\mathbf{e_{j_1}} \wedge \mathbf{e_{j_2}} \wedge \ldots \wedge \mathbf{e_{j_q}}$ be two elementary blades that have to be multiplied (wedged) together; $\mathbf{a} \wedge \mathbf{b}$ is also an elementary blade. If $p + q > n$ (where n is the space dimension) then $\mathbf{a}$ and $\mathbf{b}$ will always have at last one common factor and consequently $\mathbf{a} \wedge \mathbf{b}$ will produce a null result. The statements above are valid also if $\mathbf{a}$ and $\mathbf{b}$ are homogeneous multivectors that satisfy the condition grade($\mathbf{a}$)+grade($\mathbf{b}$)$> n$. If the grade sum of two homogeneous multivectors $\mathbf{a}$ and $\mathbf{b}$ is greater than the space dimension then every pair of elementary blades composing $\mathbf{a}$ and $\mathbf{b}$ will have at last one common factor and that will nullify the global multiplication result.

The outer-multiplication algorithm enumerates (loops /*1*/ and /*2*/ below) all the possible pairs of elementary blades composing $\mathbf{a}$ and respectively $\mathbf{b}$ and accomplishes the multiplication by *merging* their "component" arrays. The function MergeCombinations() implements the outer multiplication of two elementary blades and is based on the well known linear merging algorithm of two increasingly ordered sequences of numbers. In addition, the merging process has to identify if the two elementary blades have at least one common factor; their outer product will then vanish. Otherwise the merging process

keeps track of the number of elementary swaps between the basis vectors $\mathbf{e_{i_k}}$ ($0 \leq k \leq p$) and $\mathbf{e_{j_h}}$ ($0 \leq h \leq q$) in order to obtain the sign of the resulting elementary blade (which has also its factors $\mathbf{e_{i_l}}$ ($0 \leq l \leq p+q$) kept in the increasing order of their indices).

```
HMV WedgeProduct(HMV a, HMV b)
 // Function: The outer-product of two homogeneous multivectors
{HMV r;
 int semn;
 // local static allocations improve considerably the running time
 ...
#ifdef __PROFILE
  WP_COUNT++;
#endif
 if(a==NULL || b==NULL) return NULL;
 if(a->grade+b->grade>a->vsp->dim_space) return CreateHMV(0, a->vsp);
 r=CreateHMV(a->grade+b->grade, a->vsp);
 InitIndexComb1(a->vsp->dim_space, a->grade, ta, tsa);
 InitIndexComb1(b->vsp->dim_space, b->grade, tb, tsb);
 InitIndexComb1(r->vsp->dim_space, r->grade, tr, tsr);
 InitComb(a->vsp->dim_space, a->grade, ta);
 do { /*1*/
  InitComb(b->vsp->dim_space, b->grade, tb);
  do { /*2*/
   //wedging two elementary blades is a special type of merging
   if(MergeCombinations(a->grade, b->grade, ta, tb, tr, &semn)!=-1) {
    r->components[IndexComb(r->vsp->dim_space, r->grade, tr, tsr)] +=
     (semn*a->components[IndexComb(a->vsp->dim_space, a->grade, ta, tsa)]
     *b->components[IndexComb(b->vsp->dim_space, b->grade, tb, tsb)]);
   }
  } while(NextComb(b->vsp->dim_space, b->grade, tb));
 } while(NextComb(a->vsp->dim_space, a->grade, ta));
 return r;
}
```

The worst case analysis gives an asymptotic behavior of O(p+q) for the function MergeCombinations(), where p=a→grade, q=b→grade and the elementary operation is the comparison. If, further, MergeCombinations() is considered an elementary operation in the algorithm specified by the function WedgeProduct() the two loops of this last algorithm will require $\binom{n}{p}+\binom{n}{q}$ elementary operations. In the case of arbitrary multivector operands, the implementation uses the linearity property of the wedge product. Similar implementations can be used for other linear operators (as contraction or geometric

product) when they have multivector arguments.

```
PMULTIVECTOR WedgeProduct(PMULTIVECTOR m1, PMULTIVECTOR m2)
// Function: the outer-product between two multivectors
{ PMULTIVECTOR r;
  LIST_HCMP p, q;
  r=CreateMultivector(m1->vsp);
  for(p=m1->hbl; p!=NULL; p=p->urm)
    for(q=m2->hbl; q!=NULL; q=q->urm) {
      AddHMV(WedgeProduct(p->hcmp, q->hcmp), r);
    }
  RemNullComp(r);
  return r;
}
```

4.2 The geometric product implementation

In the case of elementary blade operands expressed in an orthonormal basis, the geometric product is computed using practically the same algorithm as for the outer-product. Indeed the general relation:

$$\mathbf{ab} = -\mathbf{ba} + 2(\mathbf{a} \bullet \mathbf{b}) \tag{2}$$

valid for any two vectors, becomes in case of two orthogonal vectors $\mathbf{ab}$ = -$\mathbf{ba}$ i.e. has same form as: $\mathbf{a} \wedge \mathbf{b}$ = -$\mathbf{b} \wedge \mathbf{a}$. The difference between the algorithms used to compute the geometric product (in case of arguments that are elementary blades) and the outer product, comes from the fact that $\mathbf{e_i} \wedge \mathbf{e_j}$ = -$\mathbf{e_j} \wedge \mathbf{e_i}$ for any $\mathbf{e_i}$,$\mathbf{e_j}$ basis vectors (so that $\mathbf{e_i} \wedge \mathbf{e_i} = 0$) but $\mathbf{e_i e_j}$ = -$\mathbf{e_j e_i}$ only for i≠j; whereas $\mathbf{e_i e_i}$=$\mathbf{e_i} \bullet \mathbf{e_i}$= $\epsilon_i \mid \mathbf{e_i} \mid^2$, where ϵ_i is the signature of $\mathbf{e_i}$. That is why the merging process of the algorithm that computes the geometric product of two elementary blades treats differently the case when the two current basis vectors have the same index.

```
EBLADE GeometricProduct(EBLADE a, EBLADE b)
// computes the geometric product between
// two elementary blades that can be factorized as
// products of basis vectors from an ORTHONORMAL basis
// in this case, the result is also an elementary blade
{EBLADE r;
 int i, j, k, s;
 static int cmb[NMAXALLOCSTATIC];
 if(a==NULL || b==NULL) return NULL;
 i=0; j=0; k=0; s=1;
```

```
 while(i<a->grade && j<b->grade) //main merging loop
  if(a->basiscomp[i]<b->basiscomp[j]) cmb[k++]=a->basiscomp[i++];
  else if(a->basiscomp[i]>b->basiscomp[j]) {
        cmb[k++]=b->basiscomp[j++];
        if((a->grade-i)%2) s = -s;
        //the sign of the result could be affected if ei>ej
       }
       else {
        //two identical basis vectors were encountered
        s *= a->vsp->sign[a->basiscomp[i]];
        if((a->grade-i-1)%2) s = -s;
        i++; j++;
       }
  while(i<a->grade) cmb[k++]=a->basiscomp[i++];
  while(j<b->grade) cmb[k++]=b->basiscomp[j++];
  r=CreateEBlade(k, a->vsp, s*a->coef*b->coef, cmb);
  return r;
}
```

If the geometric product operands are homogeneous multivectors and they are expressed using an orthonormal vector-basis, the implementation is straightforward and based on the multilinearity of the geometric product ([18]). If an application requires the extension of the package to non-orthogonal bases, the major place where the extension process must operate is the function implementing the geometric product on homogeneous multivector arguments.

The function presented below treats as an extension the case of the *null basis* of the *conformal space*. In this last case, the basis vectors are orthogonal with the notable exception of $\mathbf{e_0}$ and $\mathbf{e_1}$ (denoting $\mathbf{e_0}$ and $\mathbf{e}_\infty$ if we refer to the notation given in [24]); for them holds the equality $\mathbf{e_0e_1}=\mathbf{e_0} \bullet \mathbf{e_1}+\mathbf{e_0} \wedge \mathbf{e_1}= -1+\mathbf{e_0} \wedge \mathbf{e_1}= -1+\mathbf{E}$. That is why, in the merging process, a product like $\mathbf{e_1e_0}$ is written (according to relation (2)) as $\mathbf{e_1e_0} = -\mathbf{e_0e_1} + 2(\mathbf{e_0} \bullet \mathbf{e_1}) = -\mathbf{e_0e_1}-2$. This replacement is implemented by the function conv_conf_GP() that converts an elementary geometric product of basis vectors in (at most) two elementary blades (outer products of basis vectors) by substituting whenever possible $\mathbf{e_0} \wedge \mathbf{e_1}$ with $\mathbf{e_0e_1} - \mathbf{e_0} \bullet \mathbf{e_1} = \mathbf{e_0e_1} + 1$. For example the product $\mathbf{e_0} \wedge \mathbf{e_1} \wedge \mathbf{e_2} \wedge \mathbf{e_4}$ will be converted to the sum of: $\mathbf{e_2} \wedge \mathbf{e_4} = \mathbf{e_2e_4}$ and $\mathbf{e_0e_1e_2e_4}$. The usual multiplication procedure (for orthogonal bases and elementary blade operands) is then applied.

```
 MULTIVECTOR GeometricProduct(HMV a, HMV b)
 // computes the geometric product of two homogeneous multivectors
 // valid for orthogonal basis and for conformal null basis
{MULTIVECTOR r, m=NULL;
 int ka, kb, i, j;
```

```
 EBLADE ea, eb, er, eca[2], ecb[2];
 static int ta[NMAXALLOCSTATIC], tsa[NMAXALLOCSTATIC];
 static int tb[NMAXALLOCSTATIC], tsb[NMAXALLOCSTATIC];
 r=CreateMultivector(a->vsp);
 InitIndexComb1(a->vsp->dim_space, a->grade, ta, tsa);
 InitIndexComb1(b->vsp->dim_space, b->grade, tb, tsb);
 ea=CreateEBlade(a->grade, a->vsp, 0.0, NULL);
 eb=CreateEBlade(b->grade, b->vsp, 0.0, NULL);
 InitComb(a->vsp->dim_space, a->grade, ea->basiscomp);
 do{
  ea->coef=a->components[IndexComb(a->vsp->dim_space, a->grade,
                                   ea->basiscomp, tsa)];
  InitComb(b->vsp->dim_space, b->grade, eb->basiscomp);
  do{
   eb->coef=b->components[IndexComb(b->vsp->dim_space, b->grade,
                                    eb->basiscomp, tsb)];
   if(BasisType=='O') { // orthogonal basis
    er=GeometricProduct(ea, eb);
    AddHMV(ETOHMV(er), r);
    FreeEBlade(er);
   }
   else if(BasisType=='C'){
   // conformal model null basis e0^e1=e0e1-e0.e1
    ka=conv_conf_GP(ea, &eca[0], &eca[1]);
    kb=conv_conf_GP(eb, &ecb[0], &ecb[1]);
    for(i=0; i<ka; i++)
     for(j=0; j<kb; j++) {
      m=GeometricProductConf(eca[i], ecb[j]);
      AddMultivector(m, r);
      FreeMultivector(m);
     }
    if(ka==2) FreeEBlade(eca[1]); if(kb==2) FreeEBlade(ecb[1]);
   }
  }while(NextComb(b->vsp->dim_space, b->grade, eb->basiscomp));
 }while(NextComb(a->vsp->dim_space, a->grade, ea->basiscomp));
 FreeEBlade(ea); FreeEBlade(eb);
 RemNullComp(r);
 FreeMultivector(m);
 return r;
}
```

4.3 The inner product implementation

The *inner product* is related to a metric associated to the vector space that characterizes a certain geometric algebra. In the specialized literature were presented multiple proposals for inner products. In the present package were implemented the *Hestenes inner product* ([12]) and the *contractions* (left and right contractions) that seem to better behave in a computational environment (see [9]). Only some significant implementational details will be presented below.

Let us first take into consideration the left contraction. It is a linear operator defined (when operating on homogeneous multivectors arguments) by:

$$\mathbf{A_r} \rfloor \mathbf{B_s} = \langle \mathbf{A_r B_s} \rangle_{s-r} \tag{3}$$

This corresponds to the following (general) way of implementation:

```
HMV ContractionL(HMV a, HMV b)
// left contraction operating on homogeneous multivectors
{ HMV r;
  PMULTIVECTOR m;
  if(a->grade>b->grade) return CreateHMV(0, a->vsp);
  m=GeometricProduct(a, b);
  r=GetGrade(b->grade-a->grade, m);
  if(r!=NULL) r=CreateHMV(r->grade, a->vsp, r->components);
  FreeMultivectorComplete(m);
  // release the storage allocated to the temporary multivector
  return r;
}
```

The same kind of implementation could be given for the Hestenes inner product that, in case of homogeneous operands, is defined by:

$$\mathbf{A_r} \bullet \mathbf{B_s} = \begin{cases} \langle \mathbf{A_r B_s} \rangle_{|r-s|} & \forall s, r \neq 0 \\ 0 & \text{for r=0 or s=0} \end{cases} \tag{4}$$

This type of implementation holds for the outer product as well. If the geometric product is considered implemented initially, the outer product of two homogeneous multivectors $\mathbf{A_r}$, $\mathbf{B_s}$ is the part of highest possible degree (r+s) of their geometric product $\mathbf{A_r B_s}$.

Another type of implementation of the contraction can be given following the

definition from [19,9]:

- For any scalar $\alpha \in$R and $\mathbf{B}$ an arbitrary grade blade: $\alpha \rfloor \mathbf{B} = \alpha \mathbf{B}$. If $\mathbf{B}$ has no scalar component then $\mathbf{B} \rfloor \alpha = 0$
- $\forall$ $\mathbf{a}$, $\mathbf{b}$ vectors $\Rightarrow$ $\mathbf{a} \rfloor \mathbf{b} = \langle \mathbf{ab} \rangle_0$
- if $\mathbf{a}$ is a vector and $\mathbf{B}$, $\mathbf{C}$ blades, the following expansion formula can be applied (independently of the grade of the second argument):
  $\mathbf{a} \rfloor (\mathbf{B} \wedge \mathbf{C}) = (\mathbf{a} \rfloor \mathbf{B}) \wedge \mathbf{C} + \text{GradeInvolution}(\mathbf{B}) \wedge (\mathbf{a} \rfloor \mathbf{C})$,
- the contraction has a universally valid rewriting rule (which the Hesteness inner-product does not have): $(\mathbf{A} \wedge \mathbf{B}) \rfloor \mathbf{C} = \mathbf{A} \rfloor (\mathbf{B} \rfloor \mathbf{C})$

The rules above lead to the corresponding C recursive implementation (it computes the left contraction of two elementary blades):

```
EBLADE ContractionL(EBLADE a, EBLADE b)
// left contraction with elementary blades operands
// recursive implementation
{int i, j, s, iv;
 int vprim;
 EBLADE r1, r2, r;
 if(a==NULL || b==NULL) return NULL;
 if(a->grade>b->grade) return CreateEBlade(0, a->vsp, 0.0, NULL);
 if(a->grade==0) { //scalar rule
  r=CreateEBlade(b->grade, b->vsp, b->coef, b->basiscomp);
  for(i=0; i<r->grade; i++) r->coef *= a->coef;
  return r;
 }
 if(a->grade==1 && b->grade==1) {  // both operands vectors
  r=CreateEBlade(0, a->vsp, 0, NULL);
  if(a->basiscomp[0]==b->basiscomp[0])
   r->coef=a->coef*b->coef*a->vsp->sign[a->basiscomp[0]];
  return r;
 }
 if(a->grade==1 && b->grade>1) {  //first operand vector
  iv=a->basiscomp[0];
  r=CreateEBlade(b->grade-1, b->vsp, 0.0, b->basiscomp);
  for(i=0; i<b->grade && iv!=b->basiscomp[i]; i++);
  if(i>=b->grade) r->coef=0.0;
  else {
   s=(i%2)?-1:1;
   r->coef= s*a->coef*b->coef*a->vsp->sign[iv];
   for(i=0, j=0; i<b->grade; i++)
    if(iv==b->basiscomp[i]) continue;
    else r->basiscomp[j++]=b->basiscomp[i];
  }
```

```
 return r;
}
//applies the universally valid rewriting rule
vprim=a->basiscomp[0]; // first factor a vector
r1=CreateEBlade(1, a->vsp, a->coef, &vprim);
r2=CreateEBlade(a->grade-1, a->vsp, 1.0, NULL);
for(i=0; i<r2->grade; i++) r2->basiscomp[i]=a->basiscomp[i+1];
r=ContractionL(r2, b);
return ContractionL(r1, r);
}
```

Assuming non-null operands, the running time of the function above satisfies the recurrence:

$$T(r,s) = \begin{cases} \alpha & for\ r > s \\ \beta s & for\ r = 0\ and\ s > 0 \\ \gamma & for\ r = 1\ and\ s = 1 \\ \delta s & for\ r = 1\ and\ s > 1 \\ \lambda + \mu(r-1) + T(r-1,s) + T(1,s-r+1) & otherwise \end{cases}$$

Here, r and s are the grades of the operands and α, β, γ, δ, λ, μ are constants. Expanding this recurrence gives: $T(r,s) \in O(r^2+s^2)$. Consequently the recursive definition of left contraction corresponds to a quadratic algorithm, when applied on elementary blades arguments. As one may immediately notice the implementation (of the same operation) based on (3) is a linear algorithm.

The above=mentioned implementations can be extended to support the case of homogeneous multivector and that of general multivectors operands by using the linearity of the left contraction. For example the implementation of left contraction operating on homogeneous multivectors is obtained from the algorithm presented for the outer-product of multivectors, where the call of WedgeProduct() is substituted by a call of ContractionL() with elementary blade arguments.

4.4 The Delta product

This unconventional product from [4] is used to efficiently compute meet and join (see section 3.5). There is a correspondence between the set theory operations (applied to subspaces of V^n) and geometric algebra operators acting on blade arguments. From this point of view the delta product corresponds to a symmetric difference. The *delta product* is defined only on blade operands

and designates the highest (non-zero) grade part of their geometric product; that is:

```
HMV DeltaProduct(HMV a, HMV b)
// Function returns the delta product of two blades
// (the highest grade blade of their geometric product)
// Precondition: both arguments are blades
{HMV r; PMULTIVECTOR m;
 LIST_HCMP p, pr;
 if(a==NULL || b==NULL) return NULL;
#ifdef __WITH_ASSERT
 assert(IsBlade(a) && IsBlade(b));
#endif
 m=GeometricProduct(a, b);
 if(m==NULL) return NULL;
 if(m->hbl==NULL) {free(m); return NULL;}
 for(p=m->hbl, pr=NULL; p!=NULL; pr=p, p=p->urm);
 r=CreateHMV(pr->hcmp->grade, pr->hcmp->vsp, pr->hcmp->components);
 FreeMultivectorComplete(m);
 return r;
}
```

4.5 Blade characterization

The implementation of predicate IsBlade() can be based on the following proposition (denoted further (PB)) ([6]): "If W is an invertible homogeneous multivector and is not a blade then there exists a basis vector (denoted $\mathbf{e_j}$) such that its projection on W (i.e. $P_W(\mathbf{e_j}) = (\mathbf{e_j} \rfloor \mathbf{W})\mathbf{W}^{-1}$) is not a vector".

If a homogeneous multivector $\mathbf{B}$ does not square to a scalar then it is not a blade. (The proposition "A blade $\mathbf{B}$ squares always to a scalar" follows directly from the blade definition.)

By contrast, if $\mathbf{B}^2$ is a scalar ($\alpha \neq 0$) then $\mathbf{B}$ is not necessarily a blade (a counter-example will follow soon). In this case $\mathbf{B}$ is invertible and $\mathbf{B}^{-1} = \frac{1}{\alpha}\mathbf{B}$. Proposition (PB) is therefore applicable and one may compute the projection of every basis vector $\mathbf{e_i}$ ($0 \leq i < n$) onto $\mathbf{B}$. If every such projection is a vector then $\mathbf{B}$ is a blade. This last fact received an elegant proof in [6]. Implementation follows:

```
int IsBlade(HMV b)
// Function: returns 1 if the homogeneous multivector
// specified by the parameter b is a blade.
{ PMULTIVECTOR m, mp;
```

```
  HMV e, mc;
  LIST_HCMP p; int i, j, k;
  m=GeometricProduct(b, b);
  if(IsNull(b)) {
    if(flag_W) fprintf(stderr, "IsBlade -- null blade1\n");
    return 1;
  }
  if(SquaredMagnitude(m)<TOLERANCE) {
    if(flag_W) fprintf(stderr, "IsBlade -- null blade2\n");
    return 1;
  }
  if(m==NULL) i=0;
  else for(p=m->hbl, i=0; p!=NULL; p=p->urm, i++);
  if((i!=1) || (i==1 && m->hbl->hcmp->grade!=0)) return 0;
  e=CreateHMV(1, b->vsp);
  for(j=0; j<b->vsp->dim_space; j++) {
    for(k=0; k<b->vsp->dim_space; k++)
      e->components[k]=(k==j)?1.0:0.0;
    mp=GeometricProduct(mc=ContractionL(e, b), b);
    if(mp==NULL) i=0;
    else for(p=mp->hbl, i=0; p!=NULL; p=p->urm, i++);
    if(i!=1 || (i==1 && m->hbl->hcmp->grade!=1)) {
        FreeHMV(e); FreeMultivectorComplete(mp); FreeHMV(mc);
        return 0;
    }
  }
  FreeHMV(e); FreeMultivectorComplete(mp); FreeHMV(mc);
  return 1;
}
```

A simpler test, verifying if a homogeneous multivector is a blade could be based on the following

Proposition: In case of geometric algebras over spaces with dimensionality less than 6 a homogeneous multivector $\mathbf{B}$ is blade if and only if $\mathbf{B}^2$ is a non-zero scalar.

Proof: The direct implication holds obviously in any geometric algebra. We proof here the inverse implication. In a geometric algebra over a space with dimension less than 5, the non-zero scalars, vectors and their duals (called respectively pseudo-scalars and pseudo-vectors) are obviously blades. To complete the proof we have to verify that: "In a 5D geometric algebra if $\mathbf{B}$ is a bivector and $\mathbf{B}^2$ is a non-zero scalar then $\mathbf{B}$ is a blade." In conformity with (PB) we have to prove that $(\mathbf{x}\rfloor\mathbf{B})\mathbf{B}^{-1}$ is a vector, for any vector $\mathbf{x}$. Denoting by $\mathbf{k}$ the vector $\mathbf{x}\rfloor\mathbf{B}$ and noticing that $\mathbf{B}^{-1}$ is propor-

tional to $\mathbf{B}$ (when $\mathbf{B}^2$ is a non-zero scalar) then $\mathbf{kB}^{-1}$ is proportional to $\mathbf{kB}{=}\mathbf{k}\rfloor\mathbf{B}+\mathbf{k}\wedge\mathbf{B}$. Consequently our proposition would be valid if $\mathbf{k}\wedge\mathbf{B}=0$. But $\mathbf{k}\wedge\mathbf{B}=\frac{1}{2}(\mathbf{kB}+\mathbf{Bk})=\frac{1}{4}(\mathbf{xB}-\mathbf{Bx})\mathbf{B}+\frac{1}{4}\mathbf{B}(\mathbf{xB}-\mathbf{Bx})=\frac{1}{4}(\mathbf{xB}^2-\mathbf{B}^2\mathbf{x})=0$ q.e.d.

The test from the proposition above was sufficient for the applications developped until now over the GAP module.

Note: In G_6 the trivector $\mathbf{B}=\mathbf{e_0}\wedge\mathbf{e_1}\wedge\mathbf{e_2}+\mathbf{e_3}\wedge\mathbf{e_4}\wedge\mathbf{e_5}$ squares to -2 but is not a blade since its projection on $\mathbf{e_0}$ contains a 5-vector component. This counter-example states clearly the limits of the previous "blade test".

4.6 Unoriented operators

One important category of operations that are suited to be used in real applications are the subspace operations. The most important are meet and join. These are binary operators and have blades as operands. The *meet* of two blades $\mathbf{A}$, $\mathbf{B}$, denoted $\mathbf{A}\cap\mathbf{B}$, specifies the highest grade common subspace of $\mathbf{A}$ and $\mathbf{B}$. It can be compared with the greatest common divisor of two integers or (see the symbol representing it) with the intersection of two sets. The *join* of $\mathbf{A}$ and $\mathbf{B}$, denoted $\mathbf{A}\cup\mathbf{B}$, signifies the smallest grade space that includes both $\mathbf{A}$ and $\mathbf{B}$. It can be compared with the smallest common multiple of two integers or with the reunion of two sets. As was clearly stated by Stolfi (see [23]) if subspaces $\mathbf{A}$, $\mathbf{B}$ are not disjoint, it is impossible to define the orientation of their join (or meet) in a consistent way. In the geometric algebra framework, the blades denote oriented subspaces and their meet (or join) produces an unoriented result; this one is obtained modulo a scalar factor that may be important in certain applications and may be determined from considerations concerning magnitudes of blades (see also [4]).

The implementations of meet and join are based on [4] in which it is proved that the meet of two blades $\mathbf{A}$ and $\mathbf{B}$ corresponds to a subspace characterized by the projection function:

$$\mathbf{P}_{A\cap B}(\mathbf{x})=\frac{\mathbf{P}_B(\mathbf{x})-\mathbf{P}_{A\Delta B}(\mathbf{x})+\mathbf{P}_{(A\Delta B)B^{-1}}(\mathbf{x})}{2} \tag{5}$$

where Δ denotes the delta product. That means the projection of any vector $\mathbf{x}$ on the blade corresponding to $\mathbf{A}\cap\mathbf{B}$ can be computed from (5) where the projection of $\mathbf{x}$ on an arbitrary blade $\mathbf{B}$ is (see [13]) given by: $\mathbf{P}_B(\mathbf{x})=(\mathbf{x}\rfloor\mathbf{B})\mathbf{B}^{-1}$. The routine computing the result of the projection corresponding to $\mathbf{A}\cap\mathbf{B}$ is:

```
HMV ProjMeet(HMV a, HMV b, HMV x)
// implements the projection operator associated to Meet(a, b)
```

```
// x is a 1-grade blade (i.e. vector)
{HMV d; //delta product;
 PMULTIVECTOR m, m1, m2, m3;
 HMV c, c1, c2, p, id, ip, iip, iid, iib, ib;
 double cv[NMAXALLOCSTATIC];
 int i;
 d=DeltaProduct(a, b);
 m=GeometricProduct(iib=ContractionL(x, b), ib=Inverse(b));
 m1=GeometricProduct(iid=ContractionL(x, d), id=Inverse(d));
 c=GetGrade(x->grade, m);
 //applying the NULL BLADE convention
 if(c==NULL) c=CreateHMV(x->grade, x->vsp);
 c1=GetGrade(x->grade, m1);
 if(c1==NULL) c1=CreateHMV(x->grade, x->vsp);
 m2=GeometricProduct(d, Inverse(b)); p=GetGrade(a->grade, m2);
 m3=GeometricProduct(iip=ContractionL(x, p), ip=Inverse(p));
 c2=GetGrade(x->grade, m3);
 if(c2==NULL) c2=CreateHMV(x->grade, x->vsp);
 for(i=0; i<x->ncmp; i++)
  cv[i]=(c->components[i]-c1->components[i]+c2->components[i])/2.0;
//release the temporary storage
 ...
 return CreateHMV(x->grade, a->vsp, cv);
}
```

The reconstruction of the blade from its associated projection operator is accomplished by finding the projection $\mathbf{v_i} = \mathbf{P}_{A\cap B}(\mathbf{e_i})$ of every basis vector and wedging together the maximum possible number of $\mathbf{v_i}$ vectors in order to obtain a non-null result. This result is the blade of maximal degree on which the basis vectors $\mathbf{e_i}$ have same projections as on $\mathbf{A}\cap\mathbf{B}$ (modulo an orientation). The correctness of the algorithm is proven in [4].

```
HMV Meet(HMV a, HMV b)
// Function: implements the Join operation
// Precondition: both arguments are blades
{HMV r, r1, VB, TB; double vunu=1.0;
 int i, j, flag;
#ifdef __WITH_ASSERT
 assert(IsBlade(a) && IsBlade(b));
#endif
 VB=CreateHMV(1, a->vsp);
 r=CreateHMV(0, a->vsp, &vunu);
 flag=0;
 for(i=0; i<a->vsp->dim_space; i++) {
  for(j=0; j<VBouma->vsp->dim_space; j++)
```

```
  //generate the current basis vector
   VB->components[j]=(j==i)?1.0:0.0;
  TB=ProjMeet(a, b, VB);
  r1=WedgeProduct(r, TB);
  if(!IsNull(r1)) {
   flag=1; r=r1;
  }
 }
 return (flag==1)?r:NULL;
}
```

The same technique is used to compute the Join, in this case the corresponding projection operator is:

$$\mathbf{P}_{A\cup B}(\mathbf{x}) = \frac{\mathbf{P}_B(\mathbf{x}) + \mathbf{P}_{A\Delta B}(\mathbf{x}) + \mathbf{P}_{(A\Delta B)B^{-1}}(\mathbf{x})}{2} \tag{6}$$

If the join $\mathbf{J}=\mathbf{A}\cup\mathbf{B}$ is known, the meet $\mathbf{A}\cap\mathbf{B}$ can be computed (see [18]) as: $\mathbf{A}\cap\mathbf{B}=\mathbf{A}^* \bullet \mathbf{B}$, where the dual is considered with respect to the Join i.e. $\mathbf{A}^* = \mathbf{A}\mathbf{J}^{-1}$. This type of reasoning is applied in:

```
HMV MeetandJoin(HMV a, HMV b, HMV *j)
// The function returns the meet of two blades (a and b)
// The Join is returned by *j
// Precondition: arguments a and b are blades
{HMV jrez, r, c, ib;
 PMULTIVECTOR m1;
 jrez=Join(a, b);  *j=jrez;
 if(!IsNull(jrez) || SquaredMagnitude(jrez)!=0.0) {
  m1=GeometricProduct(a, ib=Inverse(jrez));
  c=InnerProduct(m1->hbl->blade, b);
  if(IsBlade(c)) {
   r=CreateHMV(c->grade, c->vsp, c->components);
   FreeMultivectorComplete(m1); FreeHMV(c); FreeHMV(ib);
   return r;
  }
  else {
   fprintf(stderr, "MeetandJoin -- result not a blade\n");
   exit(1);
  }
 }
 return NULL;
}
```

4.7 Orthogonal transformations

This type of transformation is specified briefly and generally in geometric algebra through a construction $\mathbf{BxB}^{-1}$ where B is a versor (see [13]); this has a straight correspondence in the C function:

```
HMV ApplyOrthoTransform(VERSOR v, HMV x)
// applies the orthogonal transformation "sandwich" vxv^{-1}
// the transformation is completely specified by the versor v
{ HMV r, q; MULTIVECTOR m, m1, m2;
#ifdef __WITH_ASSERT
  assert(IsBlade(b));
#endif
  if(x==NULL) return NULL;
  m=GeometricProduct(m1=GeometricProduct(b, x), m2=HMVTOM(Inverse(b)));
  r=GetGrade(x->grade, m);
  q=(r==NULL)?CreateHMV(x->grade, x->vsp):
     CreateHMV(r->grade, r->vsp, r->components);
  FreeMultivectorComplete(m); // release temporary storage
  FreeMultivectorComplete(m1);
  FreeMultivectorComplete(m2);
  return q;
}
```

4.8 Orthogonal Factorization

A very important issue in different applications of geometric algebra is blade factorization. That can be used to apply the *divide and conquer* approach in order to reduce the size of a given problem. The routine described below realizes the factorization of a blade writing it as a product of orthogonal vectors. It is a powerful algorithm since the orthogonality condition (see [5]) satisfied by the resulting factors allows efficient subsequent processing. Implementation details and a correctness proof are given below.

At the beginning, the algorithm finds the elementary blade of maximal coefficient (in modulus) of the given blade $\mathbf{B}$. This kind of selection increases the robustness of the algorithm in the same way the determination of the pivot element does in Gaussian elimination. The previously determined elementary blade will have q=$\mathbf{B}\rightarrow$grade basis vector factors, let them be: $\mathbf{e_{i_1}}$, $\mathbf{e_{i_2}}$,..., $\mathbf{e_{iq}}$. The algorithm eliminates at each stage one vector factor from $\mathbf{B}$. It keeps track of the vectors that were not yet eliminated using a blade $\mathbf{K}$ (initially equal to $\mathbf{B}$) that contains the product of the vectors still to be determined.

The algorithm may be concisely specified in a pseudo-code notation as follows:

(1) find $\mathbf{B}$'s elementary blade component of maximal weight. Let it be:
    $\mathbf{EB}=\mathbf{e_{i_1}} \wedge \mathbf{e_{i_2}} \wedge \ldots \wedge \mathbf{e_{i_q}}$
(2) $\mathbf{K} \leftarrow \mathbf{B}$
(3) $\mathbf{SEB} \leftarrow \{\mathbf{e_{i_1}}, \mathbf{e_{i_2}}, \ldots, \mathbf{e_{i_q}}\}$
(4) for i in { 1, 2, ..., q } do
(5) select $\mathbf{v}$ from $\mathbf{EB}$ so that $\mathbf{v} \bullet \mathbf{K} \neq 0$
(6) $\mathbf{d} \leftarrow (\mathbf{v} \bullet \mathbf{K})\mathbf{K}^{-1}$
(7) add $\mathbf{d}$ to versor $\mathbf{R}$ (as a new factor of $\mathbf{B}$)
(8) $\mathbf{K} \leftarrow \mathbf{v} \bullet \mathbf{K}$
(9) $\mathbf{SEB} \leftarrow \mathbf{SEB} \backslash \{\mathbf{v}\}$
(10) endfor
(11) result $\mathbf{R}$

Proposition: In a non-degenerate geometric algebra, the algorithm for blade orthogonal factorization is correct.

Proof: From now on, it will be assumed that the basis is orthonormal. Let's assume also that, at an arbitrary step of the algorithm (iteration of the for loop) we have to factorize the blade $\mathbf{K}$. Initially $\mathbf{K}$ is equal to $\mathbf{B}$ (assumed of grade greater than 1). The algorithm tries to separate a vector factor $\mathbf{d}$ i.e. write $\mathbf{K}$ as:

$$\mathbf{K} = \mathbf{dA} \tag{7}$$

It looks for $\mathbf{d}$ as the projection onto $\mathbf{K}$ of one of the basis vectors that compose "the most important" elementary blade of $\mathbf{B}$. Let's denote this elementary blade by $\mathbf{EB}$ and the set of its component basis vectors by $\mathbf{SEB}$. If the geometric algebra is non-degenerate, there is at least one $\mathbf{e_{i_k}}$ whose projection on $\mathbf{K}$ is non-zero (otherwise $\mathbf{B}$ will not contain $\mathbf{EB}$ as one of its components). We chose for $\mathbf{d}$ the projection of $\mathbf{e_{i_k}}$ on blade $\mathbf{K}$ i.e.

$$\mathbf{d} = (\mathbf{e_{i_k}} \bullet \mathbf{K})\mathbf{K}^{-1} \tag{8}$$

This implies

$$\mathbf{dK} = \mathbf{e_{i_k}} \bullet \mathbf{K}$$

and substituting the value of $\mathbf{K}$ from (7) results in : $\mathbf{d}^2\mathbf{A} = \mathbf{e_{i_k}} \bullet \mathbf{K}$. Consequently, $\mathbf{A}$ equals (modulo a scalar factor): $\mathbf{e_{i_k}} \bullet \mathbf{K}$. At the next iteration, the algorithm will use as new value for $\mathbf{K}$ the actual value of $\mathbf{A}$. Let's denote this

new value of $\mathbf{K}$ by $\mathbf{K}'$. $\mathbf{K}'$ is a blade orthogonal to $\mathbf{e_{i_k}}$. This last statement follows directly from the basic identity:

$$\mathbf{a} \bullet (\mathbf{b} \bullet \mathbf{K}) = (\mathbf{a} \wedge \mathbf{b}) \bullet \mathbf{K} \tag{9}$$

that is valid if grade($\mathbf{K}$)>1. (In fact, if the inner product used is the contraction, the identity above holds for any grade of $\mathbf{K}$.) Indeed substituting $\mathbf{a}$ and $\mathbf{b}$ with $\mathbf{e_{i_k}}$ formula (9) gives:

$$\mathbf{e_{i_k}} \bullet \mathbf{K}' = \mathbf{e_{i_k}} \bullet (\mathbf{e_{i_k}} \bullet \mathbf{K}) = 0$$

At the next iteration, the algorithm will determine a new vector $\mathbf{d}'$ lying in $\mathbf{K}'$ and consequently orthogonal to $\mathbf{e_{i_k}}$. Let's denote by $\mathbf{e_{s_1}}$, $\mathbf{e_{s_2}}$, ..., $\mathbf{e_{s_q}}$ the sequence of basis vectors composing **EB** in the order they were selected at every iteration of the algorithm. At an arbitrary iteration (let it be denoted by p) the algorithm generates one vector ($\mathbf{R_p}$, the p-th component of the versor $\mathbf{R}$) that is orthogonal to the corresponding elementary blade $\mathbf{K_p}$. The blades $\mathbf{K_p}$ have decreasing grades and form a sequence that satisfies the recurrence relationship: $\mathbf{K_{p+1}} = \mathbf{e_{s_p}} \bullet \mathbf{K_p}$, where $\mathbf{e_{s_p}}$ is the basis vector selected at the iteration p. That is:

- when p=q, $\mathbf{e_{s_q}}$ is the final vector factor (component of versor $\mathbf{R}$)
- when p=q-1 the vector $\mathbf{R}_{q-1}$ must be orthogonal to $\mathbf{e_{s_q}}$
- When p=q-2 the vector $\mathbf{R}_{q-2}$ must be orthogonal to $\mathbf{e_{s_q}} \wedge \mathbf{e_{s_{q-1}}}$

  ...
- when p=q-k the vector $\mathbf{R}_{q-k}$ must be orthogonal to $\mathbf{e_{s_q}} \wedge \mathbf{e_{s_{q-1}}} \wedge \ldots \wedge \mathbf{e_{s_{q-k+1}}}$

  ...
- when p=1 the vector $\mathbf{R}_1$ is orthogonal to $\mathbf{e_{s_q}} \wedge \mathbf{e_{s_{q-1}}} \wedge \ldots \wedge \mathbf{e_{s_2}}$.

The statements above have as consequence that the loop for (label 4.) maintains the invariant:

$$\begin{aligned}&((\bigwedge_{j=1}^{i} \mathbf{R_j}) \wedge \mathbf{K} == \mathbf{B}) \textit{ and } (0 \leq i \leq q) \textit{ and}\\ &(\mathbf{R_j} \textit{ are mutually orthogonal}, \forall 1 \leq j \leq i)\end{aligned} \tag{10}$$

When the loop terminates, the invariant (10) must still be true and consequently:

$$(\mathbf{B} = \bigwedge_{j=1}^{q} \mathbf{R_j}) \textit{ and } (\mathbf{R_j} \textit{ are mutually orthogonal})$$

q.e.d.

The result of the algorithm is in fact a blade. The complete C implementation of factorization algorithm follows.

```
VERSOR HMVTOOV(HMV B)
// Function: "Homogeneous MultiVector To Orthogonal Versor" conversion
// Precondition: the argument is blade
{VERSOR r;
 HMV c, K, ei, d;  PMULTIVECTOR m; int i, j, pmax, p;
 double t, cmax, md;
 static int ta[NMAXALLOCSTATIC], tsa[NMAXALLOCSTATIC],
        flag[NMAXALLOCSTATIC];
 if(B==NULL) return NULL;
#ifdef __WITH_ASSERT
 assert(IsBlade(b));
#endif
 InitIndexComb1(B->vsp->dim_space, B->grade, ta, tsa);
 r=CreateVersor(B->grade, B->vsp);
 // Find the elementary blade of maximal coefficient (in module)
 // from homogeneous multivector B
 for(i=1, pmax=0, cmax=fabs(B->components[0]); i<B->ncmp; i++)
  if((t=fabs(B->components[i]))>cmax) {
   pmax=i;
   cmax=t;
  }
 CombFromIndex(pmax, B->vsp->dim_space, B->grade, ta, tsa);
 //generate the tuple that corresponds to the index
 // of the elementary blade
 ei=CreateHMV(1, B->vsp);
 K=B;
 //flags showing if a basis vector has been previously selected
 for(i=0; i<B->vsp->dim_space; i++) flag[i]=0;
 for(i=0; i<B->grade-1; i++) { /*1*/
 //select the first basis vector of the combination "pmax"
 // that has a non-null projection on blade K and has not
 // been previously selected
  for(p=0; p<B->grade; p++) { /*2*/
   if(flag[ta[p]]==0) {
    for(j=0; j<B->vsp->dim_space; j++)
     ei->components[j]=(j==ta[p])?1.0:0.0;
    c=InnerProduct(ei, K);
    if(SquaredMagnitude(c)>TOLERANCE) { flag[ta[p]]=1; break; }
    FreeHMV(c);
   }//end if
  } // end /*2*/
  if(p>=B->grade) {
```

```
   fprintf(stderr, "BTOOV -- was not able to factorize\n");
   return r;
  }
  //Test that projection (ei.K)K^{-1} is non-null
  m=GeometricProduct(c, Inverse(K));
  // works too m=GeometricProduct(c, K)
  d=GetGrade(1, m);
  if(d==NULL) {
   fprintf(stderr, "BTOOV -- Internal error\n");
   FreeMultivectorComplete(m);
   // in that case the rest of the versor is not modified,
   // it will contain null components
   return r;
  }
  for(j=0, md=0.0; j<d->vsp->dim_space; j++)
   md += d->vsp->sign[j]*d->components[j]*d->components[j];
  if(BasisType=='C') {
   md+=InnerProdTable[0][1]*d->components[0]*d->components[1];
   md+=InnerProdTable[1][0]*d->components[1]*d->components[0];
  }
  // d is a factor of the output versor
  for(j=0; j<B->vsp->dim_space; j++) r->vfact[i][j]=d->components[j];
  K=c;
  if(md!=0.0) for(j=0; j<K->ncmp; j++) K->components[j] /= md;
 }
 for(j=0; j<B->vsp->dim_space; j++)
  r->vfact[B->grade-1][j]=K->components[j];
 return r;
}
```

4.9 Extensions specific to the conformal model

The functionality of the GAP package can be extended by its users, in order to fit the requirements of a specific application. This is a powerful property and confers the necessary flexibility that allows reusing it in a broad range of applications. This section describes an example detailing the way a peculiar extension was implemented. It treats the case of the null basis of the conformal space. The *conformal model* of the Euclidean 3D geometric space is based upon a technique called *projective split* [17,18]). This method consists in establishing a linear mapping between the vectors of a higher dimensionality space V^m and a part of the multivectors of a lower dimensionality geometric algebra G_n ($m > n$). In the concrete case of the so-called conformal split: m=n+2, the original higher dimensional space is a Minkowski space

($V^m = R^{n+1,1} = R^n \oplus R^{1,1}$). An usual case is n=3 then E^3 is the Euclidean space and $R^{4,1}$ the corresponding conformal space. In the orthogonal basis (considered the main basis) of the conformal space three vector components correspond to the Euclidean basis, the other two denoted $\mathbf{e}_+$ and $\mathbf{e}_-$ have the signatures 1 and respectively -1 and constitute the basis of $R^{1,1}$. The negative signature of $\mathbf{e}_-$ implies the existence of null vectors. As a consequence follows the possibility to build from the standard basis a new (null) basis of the conformal space, where $\mathbf{e}_+$ and $\mathbf{e}_-$ are replaced by the reciprocal null vectors: $\mathbf{e_0} = \frac{\mathbf{e}_- - \mathbf{e}_+}{2}$, $\mathbf{e}_\infty = \mathbf{e}_- + \mathbf{e}_+$. In the conformal model a vector from the Euclidean space R^n is regarded as the rejection, relative to the Minkowski plane (with bivector $\mathbf{E}$=$\mathbf{e_0} \wedge \mathbf{e}_\infty$) of the corresponding vector from the Minkowski space $R^{n+1,1}$. Denoting the conformal vectors with bold face lower case letters and the Euclidean vectors with attached arrows, the relations expressing the mapping between the vectors from Euclidean space and those from the conformal space are:

$$\mathbf{x} = \vec{x} + \mathbf{e_0} + \frac{1}{2}\vec{x}^2\mathbf{e}_\infty \tag{11}$$

and conversely:

$$\vec{x} = (\mathbf{x} \wedge \mathbf{E})\mathbf{E} \tag{12}$$

The conformal extension of module GAP contains some routines specific to the conformal model of the Euclidean 3D space. Their declarations follow. They work with 5D space entities expressed using the conformal null basis. These functions are grouped in a separate module whose interface is given in a separate header "conformal.h". As was previously mentioned (see sources from sections 4.2 and 3.2), in order to properly achieve the extension, some functions implementing geometric algebra operators have to be updated.

```
double **CreateConfInnerTable(void);
 // returns a bi-dimensional array containing the conformal
 // inner-multiplication table.
HMV PointEuclideanToConformal(PSPACE s,
      double x, double y, double z);
 // generates a conformal point (null conformal vector)
 // corresponding to the Euclidian point (x, y, z)
HMV VectorEuclidean(PSPACE s, double x, double y, double z);
 // returns the conformal vector (expressed in the null basis)
 // having the Euclidean components (x, y, z)
HMV VectorConformalToEuclidean(HMV x);
 // implements the projection (x^E)E that maps
 // conformal entities onto the Euclidean space
HMV conv_n_nn(HMV b, PSPACE ds);
```

```
 // converts the conformal homogeneous multivector b,
 // from the null basis with signatures (0 0 1 1 1) to the
 // nonnull basis with signatures (1 -1 1 1 1)
PMULTIVECTOR conv_n_nn(PMULTIVECTOR m, PSPACE ds);
 // converts the conformal multivector m,
 // from the null basis with signatures (0 0 1 1 1)
 // to the non null basis with signatures  (1 -1 1 1 1)
HMV conv_nn_n(HMV b, PSPACE ds);
 // converts the homogeneous multivector b in conformal space,
 // from the non null basis with signatures (1 -1 1 1 1)
 // to the null basis with signatures (0 0 1 1 1)
PMULTIVECTOR conv_nn_n(PMULTIVECTOR m, PSPACE ds);
 // converts the conformal multivector m,
 // from the non null basis with signatures (1 -1 1 1 1)
 // to the null basis with signatures (0 0 1 1 1)
int conv_conf_GP(EBLADE a, EBLADE *ra, EBLADE *rb);
int conv_conf_WP(EBLADE a,EBLADE *ra, EBLADE *rb);
PSPACE InitConfSpace(void);
 // returns the data structure specifying a 5D space
 // and the signatures of the null basis vectors
```

4.9.1 Basis conversion

In applications where the vector space is not Euclidean, null vectors (even null basis vectors) can occur in the computational process. This makes difficult processes like the inverse computation, finding projections (see [4]) or blade factorizations ([5]). That is why the temporary change of basis (from a null basis to a non-null basis or vice-versa) during an application could become very useful. For example in an application the main computational flow is accomplished in the non-null conformal basis but the interpretation of results could be facilitated if they are expressed using the null basis. In this basis the simple presence of $\mathbf{e}_\infty$ factor in a conformal elementary blade makes the difference between a plane and a sphere. Parameters characterizing the geometry of different objects can easily be recovered from the null basis representations of conformal entities (see [18]). The null and non-null basis chosen are those specified in the previous section (and in [18]). The null conformal basis has the elements $\mathbf{e_0}$, $\mathbf{e_1}$, $\mathbf{e_2}$, $\mathbf{e_3}$, $\mathbf{e_4}$ where $\mathbf{e_0}$ corresponds to $\mathbf{e_0}$, $\mathbf{e_1}$ to $\mathbf{e}_\infty$ and $\{\mathbf{e_2}$, $\mathbf{e_3}$, $\mathbf{e_4}\}$ to the Euclidean basis. The non-null conformal basis has the elements $\mathbf{e_0}$, $\mathbf{e_1}$, $\mathbf{e_2}$, $\mathbf{e_3}$, $\mathbf{e_4}$ where $\mathbf{e_0}$ and $\mathbf{e_1}$ correspond respectively to $\mathbf{e}_+$ and $\mathbf{e}_-$ and $\mathbf{e_2}$, $\mathbf{e_3}$, $\mathbf{e_4}$ have the same significance as above. The already mentioned order of the basis vectors allows the easy update of the implementation in case of larger space dimensions. The correspondence between $\mathbf{e_0}$, $\mathbf{e}_\infty$ and $\mathbf{e}_+$, $\mathbf{e}_-$ is: $\mathbf{e_0} = \frac{\mathbf{e}_- - \mathbf{e}_+}{2}$, $\mathbf{e}_\infty = \mathbf{e}_- + \mathbf{e}_+$. Inversely $\mathbf{e}_+ = -\mathbf{e_0} + \frac{\mathbf{e}_\infty}{2}$, $\mathbf{e}_- = \mathbf{e_0} + \frac{\mathbf{e}_\infty}{2}$. The chosen normalization condition is $\mathbf{e_0} \bullet \mathbf{e}_\infty$ = -1; it is specified by the definition table

of the inner-product that is built calling the function CreateConfInnerTable(). A simple modification there, can consequently change the normalization condition to the one used in a large part of the geometric algebra literature i.e. $\mathbf{e_0} \bullet \mathbf{e}_\infty = 1$. An implementation example follows, the function converting the representation of a conformal homogeneous multivector expressed in the null basis, in the corresponding representation that uses the non-null basis:

```
HMV conv_n_nn(HMV b, PSPACE ds)
  // converts a homogeneous multivector in conformal space
  // from a null basis (signatures {0, 0, 1, 1, 1})
  // to a non-null Minkowski basis (signatures {1, -1, 1, 1, 1})
{ HMV r;
  EBLADE e1, e2, e;
  int  k1, k, n;
  static int tb[NMAXALLOCSTATIC], tsb[NMAXALLOCSTATIC];
  if(b==NULL) return NULL;
  r=CreateHMV(b->grade, ds);
  InitIndexComb1(b->vsp->dim_space, b->grade, tb, tsb);
  e=CreateEBlade(b->grade, b->vsp, 0.0, NULL);
  InitComb(b->vsp->dim_space, b->grade, e->basiscomp);
  n=0;
  do{
  // the loop processes all the elementary blades
  // of the homogeneous multivector
    e->coef=b->components[n++];
    // test if a null elementary blade was encountered
    if(fabs(e->coef)<TOLERANCE) continue;
    k=conv_n_nn(ds, e, &e1, &e2);//overloaded version for
                                 //elementary blades arguments
    k1=IndexComb(e1->vsp->dim_space, e1->grade, e1->basiscomp, tsb);
    r->components[k1] += e1->coef;
    if(k==2) {
      k1=IndexComb(e2->vsp->dim_space, e2->grade, e2->basiscomp, tsb);
      r->components[k1] += e2->coef;
    }
    FreeEBlade(e1); FreeEBlade(e2);
  } while(NextComb(b->vsp->dim_space, b->grade, e->basiscomp));
  FreeEBlade(e);
  return r;
}
```

4.9.2 How is extension accomplished

In [2] the main operators affected by the existence of an antisymmetric part of the bilinear form are clearly stated, they are reversion and contraction (inner product). In the present example, dealing with the extension of the GAP module to support the null conformal basis, the (symmetric) bilinear form was defined in a matrix form (function CreateConfInnerTable()) and the effect of its skew-diagonal part was treated explicitly in the function computing the geometric product. That is why, when realizing the extension of the module using this technique, the inner product must be implemented selecting the appropriate homogeneous components (conforming to (3) or (4)) from the result given by the geometric product. Other routines necessary for the proper functioning of the extended package in a specific application may be grouped in a separate module.

4.10 Other implementation features

The present implementation contains polynomial (mainly linear but also cubic) algorithms. The frequent usage of dynamic memory allocation procedures increases the heap fragmentation and worsens the response time of intensive applications. Porting the module under a programming environment that supports automatic garbage collection could be useful in the development of applications like those mentioned above. The merit of our implementation approach is that the algorithms are clear, concise and can be easily extended to support different operators, spaces or basis types. This makes it a useful tool for verifying the power of geometric algebra related concepts in solving different real applications.

The module GAP supports different *work modes*. One of them enables the automatic verification of the preconditions. This is particularly useful during the development of an application. Other work modes enable the capturing of profiling information and the automatic garbage collection. The user could generate different versions of this module corresponding to different work modes, by separate compilations. The constants specifying the work mode are defined in a separate header ("globalflags.h").

5 Conclusions

The implementation of GAP was tested in an application for rendering polygonal mesh surfaces that were modeled using a geometric algebra based representation technique and the conformal model of the 3D Euclidean space ([25]).

The package functioned correctly. It clearly emphasizes the types of low-level computational operations that constitute together the specific functionality of the geometric algebra operators. The price paid for clarity and conciseness was a considerably increase of the application running time. That is due to the fact that a lot of computations are done repeatedly. There are more efficient implementations as for example that of GAIGEN ([10]), which is a data-driven implementation. The results of a lot of specific operations are pre-computed and stored for later usage, and that makes the implementation hard to understand by the non-experienced user. The package GAP is intended to be used mainly by those who want to understand the basic, defining mechanisms of Clifford algebra operations. The orthogonal factorization algorithm could constitute an example concerning the use of geometric algebra based techniques for the specification, validity proof and implementation of new applications.

6 Acknowledgements

This paper was developed as part of the research program "Geometric Algebra a New Foundation for Geometric Programming" founded by the Netherlands Organization for Scientific Research (NWO-612.012.006).

The authors gratefully acknowledge Timaeus Bouma for his careful reading of the manuscript as well as his comments (related especially to sections 3.3, 4.6 and 4.8) during the development of this work.

References

[1] R. Ablamowitz, P. Lounesto, J. Parra, *Clifford Algebras with Numeric and Symbolic Computations*, Birkhäuser, Boston, 1996.

[2] R. Ablamowitz, P. Lounesto, On Clifford algebra of a bilinear form with an antisymmetric part, in: *Clifford Algebras with Numeric and Symbolic Computations*, (R. Ablamowitz, P. Lounesto and J. Parra (eds.)), Birkhäuser Boston, pp. 167-187, 1996.

[3] M. Barnabei, A. Brini, G. C. Rota, On the Exterior Calculus of Invariant Theory, *Journal of Algebra*, **96**, pp. 120-160, 1985.

[4] T. Bouma, L. Dorst, H. G. J. Pijls, Geometric Algebra for Subspace Operations, *Acta Applicandae Mathematicae*, **73**, Kluwer Academic Publishers, pp. 285-300, 2002.

[5] T. Bouma, A stable subspace factorization algorithm, *Proceedings of Institute of Mathematics and its Applications Conference "Applications of Geometric Algebra"*, Cambridge, 2002.

[6] T. Bouma, Projection and Factorization in Geometric Algebra, to be published

[7] E. B. Corrochano, G. Sobczyk, *Geometric Algebra with Applications in Science and Engineering*, Birkhäuser, Boston, 2001.

[8] L. Dorst, C. Doran, J. Lasenby, *Applications of Geometric Algebra in Computer Science and Engineering*, Birkhäuser, Boston, 2002.

[9] L. Dorst, The Inner Products of Geometric Algebra, in: *Applications of Geometric Algebra in Computer Science and Engineering*, (L. Dorst, C. Doran and J. Lasenby (eds.)), Birkhäuser, Boston, pp. 35-46, 2002.

[10] D. Fontijne, T. Bouma, L. Dorst, Gaigen: a Geometric Algebra Implementation Generator, *Proceedings of Institute of Mathematics and its Applications Conference "Applications of Geometric Algebra"*, Cambridge, 2002.

[11] Fontijne D, Dorst L. Performance and Elegance of five models of Euclidean Geometry in a ray tracing application, accepted for publication in IEEE Computer Graphics and Applications, expected to be published 2003.

[12] D. Hestenes, G. Sobczyk, *Clifford Algebra to Geometric Calculus*, D. Reidel Dordrecht/Boston, 1984.

[13] D. Hestenes, *New Foundations for Classical Mechanics*, D. Reidel Publishing Co., Dordrecht, 1986.

[14] D. Hestenes, A Unified Language for Mathematics and Physics, in: *Clifford Algebras and their Applications in Mathematical Physics*, (J. S. R. Chisholm and A. K. Commons (eds.)), Reidel, Dordrecht/Boston, pp. 1-23, 1986.

[15] D. Hestenes, The Design of Linear Algebra and Geometry, *Acta Applicandae Mathematicae*, **23**, Kluwer Academic Publishers, pp. 65-93, 1991.

[16] D. Hestenes, R. Ziegler, Projective Geometry with Clifford Algebra, *Acta Applicandae Mathematicae*, **23**, Kluwer Academic Publishers, pp. 25-63, 1991.

[17] D. Hestenes, Old Wine in New Bottles: A New Algebraic Framework for Computational Geometry, in: *Geometric Algebra with Applications in Science and Engineering*, (E. B. Corrochano, G. Sobczyk (eds.)), Birkhäuser, Boston, pp. 3-17, 2001.

[18] H. Li, D. Hestenes, A. Rockwood, Generalized Homogeneous Coordinates for Computational Geometry, in: *Geometric Computing with Clifford Algebra*, (G. Sommer, (ed.)), Springer Verlag, Berlin Heidelberg, 2001.

[19] P. Lounesto, Marcel Riesz's work on Clifford algebras, in: *Clifford Numbers and Spinors*, (E. F. Bolinder and P. Lounesto, (eds.)), Kluwer Academic Publishers, pp. 215-241, 1993.

[20] P. Lounesto, Counterexamples in Clifford algebras with CLICAL, in: *Clifford Algebras with Numeric and Symbolic Computations* (R. Ablamowitz, P. Lounesto and J. Parra (eds.)), Birkhäuser Boston, pp. 3-30, 1996.

[21] S. Mann, L. Dorst, T. Bouma, The making of GABLE: A Geometric Algebra Learning Environment in MATLAB, in: *Geometric Algebra with Applications in Science and Engineering*, (E. B. Corrochano, G. Sobczyk (eds.)), Birkhäuser, Boston, pp. 491-511, 2001.

[22] R. C. Pappas, Oriented Projective Geometry with Clifford Algebra, in: *Clifford Algebras with Numeric and Symbolic Computations* (R. Ablamowicz, P. Lounesto and J. Parra (eds.)), Birkhäuser Boston, pp. 233-250, 1996.

[23] J. Stolfi, *Oriented projective geometry, a framework for geometric computations*, Academic Press, Boston, 1991.

[24] M. D. Zaharia, Computer Graphics from a Geometric Algebra Perspective, *Intelligent Autonomous Systems Technical Report Series*, IAS-UVA-02-05, University of Amsterdam, 2002. (available at: www.science.uva.nl/research/ias)

[25] M. D. Zaharia, L. Dorst, *Modeling and Visualization of 3D polygonal mesh surfaces using geometric algebra*, submitted for publication.

List of Figures