Clifford Multivector Toolbox (for MATLAB)

Stephen J. Sangwine and Eckhard Hitzer

Abstract. MATLAB® is a numerical computing environment oriented towards manipulation of matrices and vectors (in the linear algebra sense, that is arrays of numbers). Until now, there was no comprehensive toolbox (software library) for MATLAB to compute with Clifford algebras and matrices of multivectors. We present in the paper an account of such a toolbox, which has been developed since 2013, and released publically for the first time in 2015. The paper describes the major design decisions made in implementing the toolbox, gives implementation details, and demonstrates some of its capabilities, up to and including the LU decomposition of a matrix of Clifford multivectors.

Mathematics Subject Classification (2010). Primary 68-04; Secondary 11E88, 15A66.

Keywords. MATLAB, software library, toolbox, Clifford algebra.

1. Introduction

Modern applications of Clifford algebras[1] exist in many fields as diverse as robotics (*e.g.*, collision avoidance, pose estimation), computer vision (motion capture, orientation and tracking) and (colour) image processing (ray tracing, image segmentation), materials science (optical tomography, space group symmetry visualization), applied physics (vector fields, scattering and diffraction of electromagnetic waves), and many others. For an overview of these and further applications please see [1, 2].

In many applications, computation is important, particularly for modelling and simulation, but numerical computation can also be used to study

[1]In the literature further common names of these algebras are *geometric algebras*, and *Clifford('s) geometric algebras*.

algorithms and verify algebraic results used in algorithms. Therefore computational tools tailored to Clifford algebras are of wide applicability and indeed there are in existence several computational libraries and tools for computing with Clifford algebras. However, until now there has not been a *comprehensive* software library for computing with Clifford algebras in MATLAB [3]. In this paper we present details of a new library (known in MATLAB as a *toolbox*) designed so as to *extend* MATLAB in a natural MATLAB-like manner to handle *arrays* (including, but not limited to, vectors[2] and matrices) with elements which are Clifford multivectors in an arbitrarily chosen Clifford algebra. The toolbox presented here is a numerical toolbox at the present time, but since MATLAB includes a symbolic toolbox, there remains open the possibility of adding symbolic manipulation at a later date.

A paper such as this, covering mathematical software, of necessity uses terminology from both mathematics and computing. We have tried to make clear what we mean by technical terms using footnotes or parenthetical explanations.

2. Mathematical preliminaries

Definition 1 (Clifford's geometric algebra [4, 5, 6]). *Let* $\{\mathbf{e}_1, \ldots, \mathbf{e}_p, \mathbf{e}_{p+1}, \ldots, \mathbf{e}_{p+q}, \mathbf{e}_{p+q+1}, \ldots, \mathbf{e}_n\}$*, with* $n = p + q + r$, $\mathbf{e}_k^2 = Q(\mathbf{e}_k)1 = \varepsilon_k$,

$$\varepsilon_k = \begin{cases} +1 & : \quad k = 1, \ldots, p \\ -1 & : \quad k = p+1, \ldots, p+q \\ \ \ 0 & : \quad k = p+q+1, \ldots, n, \end{cases} \tag{1}$$

be an orthonormal base *of the inner product vector space* $(\mathbb{R}^{p,q,r}, Q)$*,* Q *the quadratic form, with a geometric product according to the multiplication rules*

$$\mathbf{e}_k\mathbf{e}_l + \mathbf{e}_l\mathbf{e}_k = 2\varepsilon_k\delta_{k,l}, \qquad k, l = 1, \ldots n, \tag{2}$$

where $\delta_{k,l}$ *is the Kronecker symbol with:*

$$\delta_{k,l} = \begin{cases} 1 & : \quad k = l \\ 0 & : \quad k \neq l. \end{cases}$$

This non-commutative product and the additional axiom of associativity *generate the* m*-dimensional Clifford geometric algebra denoted* $C\ell_{p,q,r}$ *over*[3] $\mathbb{R}$*, with* $m = 2^n$*. The set* $\{\mathbf{e}_A : A \subseteq \{1, \ldots, n\}\}$ *with* $\mathbf{e}_A = \mathbf{e}_{h_1}\mathbf{e}_{h_2}\ldots\mathbf{e}_{h_k}$*,* $1 \leq h_1 < \ldots < h_k \leq n$*,* $\mathbf{e}_\emptyset = 1$ *the unity in the Clifford algebra, forms a graded (blade) basis of* $C\ell_{p,q,r}$[4]*. The grades* k *range from* 0 *for scalars,* 1 *for vectors,* 2 *for bivectors,* s *for* s*-vectors, up to* n *for pseudoscalars. The quadratic space* $(\mathbb{R}^{p,q,r}, Q)$ *is embedded into* $C\ell_{p,q,r}$ *as a subspace, which is*

[2]The word vector here means a degenerate matrix with one row or column, not a vector in the sense used in Clifford algebras.

[3] For quadratic non-degenerate Clifford algebras (see [5, § 14.3]), it would equally be possible to replace the field of real numbers $\mathbb{R}$ in this definition by the field of complex numbers $\mathbb{C}$. This would also be no problem for computations with MATLAB.

[4]In the rest of the paper we denote the unity element as $\mathbf{e}_0$.

identified with the subspace of 1*-vectors. All linear combinations of basis elements of grade* k*,* $0 \leq k \leq n$*, form the subspace* $C\ell^k_{p,q,r} \subset C\ell_{p,q,r}$ *of* k*-vectors. The general elements of* $C\ell_{p,q,r}$ *are real linear combinations of basis blades* $\mathbf{e}_A$*, called Clifford numbers, multivectors or hypercomplex numbers.*

In general $\langle \mathbf{A} \rangle_k$ denotes the grade k part of $\mathbf{A} \in C\ell_{p,q,r}$. Following [6], the parts of grade 0 and $k+s$, respectively, of the geometric product of a k-vector $\mathbf{A}_k \in C\ell_{p,q,r}$ with an s-vector $\mathbf{B}_s \in C\ell_{p,q,r}$

$$\mathbf{A}_k * \mathbf{B}_s := \langle \mathbf{A}_k \mathbf{B}_s \rangle_0 , \qquad \mathbf{A}_k \wedge \mathbf{B}_s := \langle \mathbf{A}_k \mathbf{B}_s \rangle_{k+s} , \tag{3}$$

are called *scalar product* and *outer product*, respectively. They are bilinear products mapping a pair of multivectors to a resulting product multivector in the same algebra. The outer product is also associative, the scalar product is not. Compare [5, 7] for further common derived products, *e.g.*, left and right contractions are algebra derivations, which generalize the inner product of vectors to k-vectors. All these products extend by linearity to products of general multivectors.

For non-Euclidean vector spaces with non-degenerate quadratic form ($r = 0$) it is conventional to write $C\ell_{p,q}$ rather than $C\ell_{p,q,0}$. Every k-vector $\mathbf{B}$ that can be written as the outer product $\mathbf{B} = \mathbf{b}_1 \wedge \mathbf{b}_2 \wedge \ldots \wedge \mathbf{b}_k$ of k vectors $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k \in \mathbb{R}^{p,q,r}$ is called a *simple* k-vector or *blade*.

Multivectors $\mathbf{M} \in C\ell_{p,q,r}$ have k-vector parts ($0 \leq k \leq n$): scalar part $\langle \mathbf{M} \rangle_0 \in \mathbb{R}$, vector part $\langle \mathbf{M} \rangle_1 \in \mathbb{R}^{p,q,r}$, bivector part $\langle \mathbf{M} \rangle_2 \in \bigwedge^2 \mathbb{R}^{p,q,r}$, $\ldots$, and pseudoscalar part $\langle \mathbf{M} \rangle_n \in \bigwedge^n \mathbb{R}^{p,q,r}$

$$\mathbf{M} = \sum_A \mathbf{M}_A \mathbf{e}_A = \langle \mathbf{M} \rangle_0 + \langle \mathbf{M} \rangle_1 + \langle \mathbf{M} \rangle_2 + \cdots + \langle \mathbf{M} \rangle_n . \tag{4}$$

A multivector matrix or Clifford matrix is an $a \times b$ matrix A of multivector elements, *i.e.*, an array of a rows and b columns with each array entry A_{ij}, $1 \leq i \leq a$, $1 \leq j \leq b$, itself being a full multivector $\mathsf{A}_{ij} \in C\ell_{p,q,r}$.

3. Introduction to the toolbox

In the rest of this paper we present an account of a new open-source[5] toolbox (software library) which enables MATLAB [3] to compute with *arrays* (including matrices) of Clifford multivectors, *i.e.*, elements of $C\ell_{p,q,r}$. This toolbox is obviously not the first software library for computing with Clifford algebras and it is unlikely to be the last. In the 1980s, Lounesto and others created a stand-alone calculator-like program called CLICAL [8], running under MS-DOS, which was probably the first software for calculating with Clifford multivectors. In the 1990s, GABLE [9, 10] was created and is of interest here because it is a MATLAB package. GABLE implements only Clifford algebras applicable to 3-dimensional space (*i.e.*, with three basis vectors), and it can represent only single multivectors (not arrays or matrices of multivectors). Thus, although it is a MATLAB package, it does not *extend* MATLAB to handle

[5]GNU General Public License v3

arrays or matrices of multivectors, rather the authors used the MATLAB platform to support a Clifford package. The primary aim of GABLE appears to have been education, and it does permit visualization of vectors, bivectors, geometric and inner products *etc.* Development halted around 2001, and although MATLAB has moved on through many versions since then, the package still works on current versions of MATLAB.

Other libraries and packages exist for computing with Clifford algebras. They include libraries, template libraries, and code generators for C++, Java and other languages designed to be used as components in other software. Examples are: GAIGEN [11], GluCat [12], Gaalet [13], and GAALOP [14]. There are also packages which run in an environment provided by a software tool (examples of such tools include MATLAB, MAPLE, Mathematica, Maxima). Examples (in addition to GABLE, mentioned above) are the CLIFFORD package for Maple [15] which provides very sophisticated symbolic manipulation capabilities, and a Clifford package for Mathematica [16] (however it is not clear where to obtain this package).

The new toolbox has been developed by the authors since 2013 and was first released publicly (version 0.5) in March 2015 at Sourceforge [17]. The current state of implementation, as of version 0.9, is:

- the underlying structure of the toolbox is complete (initialization of algebras, internal representation of multivector arrays);
- the user can compose and decompose multivector arrays from numeric data (for example to construct multivector arrays from data read in from an external file);
- the MATLAB colon notation for sophisticated indexing of arrays, and concatenation of arrays using square brackets are implemented for both reading and assignment (we have overloaded[6] the MATLAB `subsref`, `subsindex` and `subsasgn` functions to make this possible);
- arithmetic functions are implemented, including the full geometric product of two multivectors (discussed in § 4) (see Table 1); complete with test code for a range of algebra specific multiplication tables;
- grade extraction, involutions (conjugate, reverse *etc.*) are implemented; (see Table 1)
- LU decomposition and matrix inverse work, but not yet for larger algebras $C\ell_{p,q}$ with more than 6 grades, *i.e.*, $p + q > 6$ (these functions are not public yet and will be the subject of a future paper).

Some ideas used in the toolbox are derived from the Quaternion Toolbox for MATLAB (QTFM) [18] which has been developed since 2005 and has been

[6] *Overloading* in computer languages means that multiple functions with the same name are provided, each operating on a different type of data, and the appropriate function is called automatically according to the data type of the parameters provided at run-time. The QFTM toolbox, for example, provides two overloadings of the `abs` function, which in MATLAB computes the absolute value of its argument (real or complex). The quaternion toolbox provides quaternion and octonion versions of this function each of which computes the absolute value (magnitude) of its quaternion or octonion argument (and being vectorised, computes this for each element of an array argument).

TABLE 1. Summary of key functions provided by the toolbox.

Function name	Computes
abs($\mathbf{M}$)	Modulus $\lvert\mathbf{M}\rvert = \sqrt{\lvert\lvert\mathbf{M}\rvert\rvert}$
bivector($\mathbf{M}$)	Bivector $\langle\mathbf{M}\rangle_2$
cast($\mathbf{M}$, T)	Casts/converts coefficients of $\mathbf{M}$ to data type T
conj($\mathbf{M}$)	Clifford conjugate of a multivector
dual($\mathbf{M}$)	Dual of a multivector[a] $\mathbf{M}^* = \mathbf{M}I^{-1}$
even($\mathbf{M}$)	Even part $\sum_{k,\text{even}} \langle\mathbf{M}\rangle_k$
grade($\mathbf{M}$, k)	Grade extraction $\langle\mathbf{M}\rangle_k$
left_contraction($\mathbf{M}$, $\mathbf{N}$)	Left contraction $\mathbf{M}\rfloor\mathbf{N}$
-/minus($\mathbf{M}$, $\mathbf{N}$)	Subtraction $\mathbf{M} - \mathbf{N}$
*/**mtimes**($\mathbf{M}$, $\mathbf{N}$)	Matrix product $\mathbf{MN}$
normm($\mathbf{M}$)	Norm $\lvert\lvert M\rvert\rvert = \sum_A M_A^2$, see (4)
odd($\mathbf{M}$)	Odd part $\sum_{k,\text{odd}} \langle\mathbf{M}\rangle_k$
part($\mathbf{M}$, k)	Coefficient[b] $\mathbf{M}_k \in \mathbb{R}$, $1 \le k \le m$
+/**plus**($\mathbf{M}$, $\mathbf{N}$)	Addition $\mathbf{M} + \mathbf{N}$
pseudoscalar($\mathbf{M}$)	Pseudoscalar $\langle\mathbf{M}\rangle_n$
reshape	Reshapes an array
reverse($\mathbf{M}$)	Computes the reverse of a multivector (an involution)
right_contraction($\mathbf{M}$, $\mathbf{N}$)	Right contraction $\mathbf{M}\lfloor\mathbf{N}$
scalar($\mathbf{M}$)	Scalar $\langle\mathbf{M}\rangle_0$
scalar_product($\mathbf{M}$, $\mathbf{N}$)	Scalar product of two multivectors
size	Dimensions of a multivector array
squeeze	Removes singleton dimensions from an array
sum	Sums an array of multivectors
.*/**times**	Elementwise (geometric) product
transpose	Transpose of a matrix
trivector($\mathbf{M}$)	Trivector $\langle\mathbf{M}\rangle_3$
unit($\mathbf{M}$)	Normalise to unit modulus[c]: $\mathbf{M}/\lvert\mathbf{M}\rvert$
vector($\mathbf{M}$)	Vector $\langle\mathbf{M}\rangle_1$
wedge($\mathbf{M}$, $\mathbf{N}$, ...)	Wedge product[d] $\mathbf{M} \wedge \mathbf{N} \wedge \dots$

Bold function names are overloadings of standard MATLAB functions. All except **mtimes**, **reshap**e, **size**, **squeeze**, **sum** and **transpose** operate elementwise on their arguments (that is the operation is applied to each element of an array of multivectors).

[a] For non-degenerate quadratic form Clifford algebras. I denotes the pseudoscalar.

[b] In fact, the toolbox permits coefficients to be complex, as discussed in § 7.3.

[c] For multivectors with non-zero modulus.

[d] Alternative common name: outer product.

downloaded over 13,000 times. The quaternion toolbox includes many high-level algorithms including quaternion Fourier transforms, the LU, QR, SVD and EVD matrix decompositions, some of which have been the subject of journal papers [19, 20, 21]. Since release 2 in 2013, QTFM also includes basic infrastructure and some functions for computing with octonions. The quaternion toolbox is highly vectorized[7] and many functions are overloadings of MATLAB functions and hence are easy to learn, since they operate largely in the same way as the original MATLAB functions.

The new toolbox is more complex than the quaternion toolbox because it handles multiple algebras, rather than one fixed algebra. The user can only compute with one algebra at a time but it is possible to switch dynamically from one Clifford algebra to another in the middle of a running script or function, and to preserve the values of existing variables, making it feasible to check that a computation works in many algebras with one script.

The authors plan to implement some functions in higher-dimensional algebras by recursion to lower-dimensional algebras (that is, the toolbox itself will internally switch to a lower-dimensional algebra matrix representation of variables from a higher-dimensional algebra, and then return to the original algebra before return of results). This depends on successfully implementing isomorphisms as discussed by Lounesto [5, Chapter 16], which requires some detailed but not fundamentally difficult programming to rearrange multivector data into a different format and handle the switch between two algebras (most likely using a stack to preserve details of the previous algebra ready for the return later).

We chose to create a new toolbox *for* MATLAB (and not for some other language or maths tool) for several reasons including: experience with the quaternion toolbox QTFM; the fact that MATLAB is a (mainly) numerical tool widely used in many fields across academia and industry; the ability of MATLAB to compute fast using processor-level vectorization; MATLAB's notation for vector and matrix operations using the colon notation to select slices or segments of an array for read and write operations; and the existence of a large library of functions which can be used in conjunction with a user-defined toolbox (*e.g.*, functions for plotting/image viewing, file I/O, statistics).

4. Initializing an algebra

Since the toolbox can handle any Clifford algebra, the user must initialize an algebra before computing with it. This is done by specifying the number of basis vectors that square to $+1$, -1, and 0 (the 'signature', (p, q, r) respectively) as defined in Definition 1.

[7]Vectorization in numerical computing refers to the efficient computation of operations on arrays of data in memory without individual access to the array elements by the user's code. It also usually means that the operations are implemented by single-instruction multiple-data machine instructions rather than a sequence of machine instructions or repeated machine instructions operating on each datum one-by-one.

The initialization process computes and stores a multiplication table of size $m \times m$ for the specified algebra (compare Definition 1), tabulating the result of multiplication of each of the m basis elements[8] $\mathbf{e}_j, j = 0, \ldots, m-1$ with any other. The entries in this table contain two items of information about the product $\mathbf{e}_a \mathbf{e}_b = \epsilon_{ab} \mathbf{e}_c$ $(a, b, c \in \{0, 1, 2, 3, \ldots, m-1\})$:

1. the sign[9]: $\epsilon_{ab} \in \{-1, 0, +1\}$;
2. the index c of the result (this depends only on $n = p + q + r$ and not on the individual values of p, q and r in the signature).

In describing how the toolbox works, and in thinking about it ourselves, we have to map between multiple systems of indexing. Internally, the toolbox uses a simple index from 1 to m to access multivector elements (MATLAB's indices start at 1). This system is used in computing products using the table described above, but it is not what the user sees. The first element of a multivector (the scalar part) is usually denoted by index zero, and therefore the unit scalar is denoted by `e0` when the user types it. Similarly, we provide a means for the user to enter basis elements such as $\mathbf{e}_{23}$ by typing `e23`. We call the sequence of index values '23' (with strictly ascending order) a lexical index. The toolbox handles the conversion between lexical and numeric indices (used for indexing into the multivector internally) in a way that should seem natural to the user (mostly by hiding the details).

The multiplication table is used to compute the full multivector (geometric) product using table lookup. A concise mathematical definition of the product of two multivectors is given by Perwass *et al.* [22, § 2, p. 461] using the Einstein summation convention and assuming a tensor (3-dimensional array) which expresses the multiplication table between basis elements. In practice, because the tensor is sparse (mostly zero), the arrangement described above with separate sign and index tables is much more compact. An equivalent is stated by Schulz *et al.* in [23, Equation 1, § II.B] for the case of hypercomplex algebras in general. The matrices T described in [23] are planes within the tensor of [22]. The practical implementation is almost trivial: a multivector variable[10] is stored as an array of m numeric arrays (in fact in a MATLAB cell array of length m, which we explain in more detail in § 7.1) and the m values of one multivector are each multiplied in turn with the m values of the other, the appropriate sign is applied to the result from the table[11], and the product is stored into or added to the appropriate element of the result using the index from the table.

[8] *Basis element* here means the set containing the scalar $\mathbf{e}_0$, the n basis vectors $\mathbf{e}_i$, $i = 1, \ldots, n$, and the lexically ordered canonic combinations of the basis vectors $\mathbf{e}_{12}$ up to the pseudoscalar (with index $m-1$).

[9] Note that ϵ here is not the same ε as given in Definition 1 although it is derived from the rule given in (2). The 'sign' may be zero because some algebras (with $r > 0$) have basis elements that square to zero.

[10] *Multivector variable* means a MATLAB variable containing an array of multivectors (of which a single multivector is just a special case).

[11] The full truth is this: if the sign is zero, clearly it is a waste of time to compute the product, so in fact this step is omitted by checking the sign first.

Entries in the sign table are computed from the rules given in Definition 1, which are sufficient to define directly all the products of basis elements up to $\mathbf{e}_n$. The rules for basis elements $\mathbf{e}_{n+1} \ldots \mathbf{e}_m$ are computed by counting the number of swaps needed to put the subscripts of a product into canonical order. For example, the product of $\mathbf{e}_{24}$ with $\mathbf{e}_{134}$ yields a notional result $\mathbf{e}_{24134}$ which must be 'reduced' to a canonic basis element with appropriate sign. In this example, moving either of the '4's to make them adjacent requires two swaps and therefore two changes of sign, which cancel out, yielding $\mathbf{e}_{21344}$. The duplicated index '4' is removed and the sign of $\mathbf{e}_4^2$ is applied to the result. Finally, the remaining unique indices must be placed in lexical order, involving a further swap, which requires a change of sign, giving $\pm\mathbf{e}_{123}$, the sign depending on the algebra. In practice, much of this is implemented using bitwise logical operations, and the reader is referred to the code for further details [17, function `clifford_sign_table`].

Entries in the index table are computed straightforwardly by mapping lexical indices into numeric indices using a sort.

Since the computation of the multiplication table is intricate, we have taken steps to make verification possible by comparison with other Clifford algebra software. To do this we have implemented an export function that can export the multiplication table to a comma-separated values file, with the hope that other authors will do the same. Comparison between the exported tables is straightforward. However, our internal test code also verifies the multiplication table, by a somewhat different computation to that used to construct it.

There are in fact two functions which compute the full multivector product, in order to implement the two standard MATLAB multiplications: one (`times` or `.*`) computes the elementwise product of two arrays (the Hadamard product), the other (`mtimes` or `*`) computes the matrix product (of conformable matrices or vectors). In both cases, the individual multiplications of multivectors within the matrices are full geometric products. The only difference in the coding of the two functions is which of the two standard MATLAB products is used to compute the individual numerical products of multivector coefficients. (Matrix multiplication of matrices of Clifford multivectors is based on the underlying MATLAB matrix product, thus making use of highly optimized and fast code developed and enhanced over many years by the Mathworks, and of course, hardware level parallelism which MATLAB employs.)

Once an algebra has been initialized, all multivector variables created thereafter are implicitly elements of the initialized algebra. Any multivector variables already in existence are not destroyed but they cannot be used until the 'signature' is restored to the value that was in use when they were created (every multivector variable contains a (hidden) copy of the signature to enable this check). The function `clifford_signature` with parameters initializes an algebra as shown in Figure 1. Without parameters it displays a summary of the currently initialized algebra.

```
>> clifford_signature(0,3)
>> clifford_signature
Algebra Cl(0,3)
Dimensionality:   8
Number of grades: 4
Multiplication table:
      | e0    e1    e2    e3    e12   e13   e23   e123
------+-----------------------------------------------
e0    | e0    e1    e2    e3    e12   e13   e23   e123
e1    | e1   -e0    e12   e13  -e2   -e3    e123 -e23
e2    | e2   -e12  -e0    e23   e1   -e123 -e3    e13
e3    | e3   -e13  -e23  -e0    e123  e1    e2   -e12
e12   | e12   e2   -e1    e123 -e0    e23  -e13  -e3
e13   | e13   e3   -e123 -e1   -e23  -e0    e12   e2
e23   | e23   e123  e3   -e2    e13  -e12  -e0   -e1
e123  | e123 -e23   e13  -e12  -e3    e2   -e1    e0
```

FIGURE 1. Initializing an algebra

5. Constructing multivector arrays

All variables in MATLAB are arrays (even π is represented as a 1×1 matrix containing the numeric value $3.14159\ldots$), therefore *by design* all multivector variables are also arrays, since they are composed of standard MATLAB numeric arrays (each multivector variable contains a cell array of m standard MATLAB numerical arrays). The reason for this design choice is to make it possible to utilise as much as possible of the existing MATLAB infrastructure for arithmetic, array indexing, display of numerical values in the command window *etc.* Our toolbox is designed to extend MATLAB, not just use it as a platform for Clifford algebra computations. In § 6 below we discuss an example of this extension concept: the LU decomposition of a matrix of Clifford multivectors. We provide an overloading of the MATLAB `lu` function to operate on matrices of multivectors, using as far as possible the same parameter profile as the existing MATLAB function. This concept is not limited to named functions: it also applies to the use of indexing using parentheses and the colon notation; to the use of square brackets to perform concatenation of arrays; and to the use of the dotted operator notation for elementwise operations such as `.*` (elementwise product) and `.^` (elementwise exponentiation).

There are several ways to build a multivector variable from given data, such as data read in from a file:

1. Use a constructor function and supply each numerical component of the multivector as a parameter. The constructor function is called `clifford`.
2. Multiply numeric quantities with basis elements and add them. The basis elements ($\mathbf{e}_1$, $\mathbf{e}_{123}$ *etc.*) are provided as parameterless functions called `e1`, `e123` and so on. (These are created by the initialization process.)

```
>> p = clifford(1,2,3,4,5,6,7,8)

p = 1.0000 e0
  + 2.0000 e1  + 3.0000 e2  + 4.0000 e3
  + 5.0000 e12 + 6.0000 e13 + 7.0000 e23
  + 8.0000 e123

>> q = 1 * e0  + 2 * e1  + 3 * e2 + 4 * e3 + 5 * e12 ...
     + 6 * e13 + 7 * e23 + 8 * e123

q = 1.0000 e0
  + 2.0000 e1  + 3.0000 e2  + 4.0000 e3
  + 5.0000 e12 + 6.0000 e13 + 7.0000 e23
  + 8.0000 e123

>> p == q

ans = 1
```

FIGURE 2. Constructing multivectors using the constructor function (top), then the parameterless basis functions (below). (The output has been slightly edited to remove white space and blank lines. Both methods will also work with arrays.)

3. For testing, a random multivector generation function is provided, like MATLAB's `rand` function. It returns an array of unit modulus multivectors.

The first two of these methods are illustrated in Figure 2. At the end of the transcript, the equality operator (implemented for multivectors by the toolbox) compares the variables p and q. The result is 1 (a MATLAB logical (or Boolean) value), indicating `true`.

There are also ways to extract the geometric and numerical components of a multivector variable. Figure 3 shows grade extraction and coefficient extraction. The `bivector` function extracts the bivector part of its argument. Named functions are provided to extract the scalar, vector, bivector, trivector and pseudoscalar grades. The more general `grade` function must be used for higher grades, or where it is necessary to index through the grades. The grade extraction functions return a multivector variable (even in the case of the scalar or pseudoscalar grades), whereas the coefficient extraction function `part` returns a numeric variable. Grades are indexed from zero (scalar) following mathematical convention. Coefficients are indexed from one (MATLAB indexing convention). Thus in the example with $n = 3$, $m = 2^n = 8$, the pseudoscalar part is grade 3 (there are four grades in this algebra with indices $0, 1, 2, 3$), but it is accessed as a coefficient at index 8 (the basis coefficients are indexed from 1 to 8).

```
>> bivector(p)

ans = 5.0000 e12 + 6.0000 e13 + 7.0000 e23

>> grade(p, 2)

ans = 5.0000 e12 + 6.0000 e13 + 7.0000 e23

>> grade(p, 3)

ans = 8.0000 e123

>> part(p, 8)

ans =  8
```

FIGURE 3. Accessing parts of a multivector (the functions will also work on multivector arrays).

6. A non-trivial test case: the LU decomposition

Now we present results from a non-trivial matrix function that has been successfully implemented in the toolbox. The LU decomposition is a factorization of a matrix into a lower (L) and an upper (U) triangular matrix. MATLAB implements this in the form `[L, U, P] = lu(A)` where P is an (optional) permutation matrix such that $\mathsf{LU} = \mathsf{PA}$. The Clifford toolbox currently implements this decomposition for algebras where $r = 0$ (it won't work when some of the basis elements square to zero). Of course, there can be a problem with computing any decomposition if the matrix to be decomposed has elements which are divisors of zero (algebraically non-invertible multivectors), and we will address this in a forthcoming paper. The Clifford `lu` function takes parameter profiles identical to the MATLAB function or raises an error if a parameter profile is not supported, as you would expect. The LU decomposition was implemented first because it needs no complicated computations apart from inverses and row/column products and standard arithmetic. It does however, require indexed access for both read and write, so this function provides a good test case for verifying much of the functionality of the toolbox. The decomposition requires many fundamental multivector operations: multivector inverses, full multivector (geometric) products, multivector additions, and outer (matrix products) of rows and columns of multivectors. *The LU function is not yet available in the public release, but will be the subject of a forthcoming paper.*

Figure 4 shows results from the LU decomposition of a small (3×3) array in algebra $C\ell_{1,1}$. The 3×3 matrix A is created by the `randm` function which returns an array of unit modulus random multivectors. Thus A is a 3×3 matrix with (random, unit modulus) multivectors as elements. The `lu` function computes the decomposition of A into two matrices L and U, each of

```
>> clifford_signature(1,1)
>> A = randm(3);
>> [L, U, P] = lu(A)

L = 3x3 Cl(1,1) multivector array

U = 3x3 Cl(1,1) multivector array

P =
     1     0     0
     0     0     1
     0     1     0

>> show([L, U])

e0 *

    1.0000         0         0    0.6759   -0.3119    0.2112
   -0.6785    1.0000         0         0   -0.8486   -0.5983
    0.7731    0.3155    1.0000         0         0   -0.5460

+ e1 *

         0         0         0   -0.2391   -0.8004    0.5038
   -0.3510         0         0         0   -0.4131    0.3472
   -0.1410   -0.3898         0         0         0    0.2240

+ e2 *

         0         0         0   -0.6844   -0.2096    0.7260
   -0.5940         0         0         0   -0.9474    1.4190
   -0.7386   -0.1936         0         0         0   -0.7052

+ e12 *

         0         0         0   -0.1326    0.4670   -0.4177
    0.3806         0         0         0    0.0146   -0.3652
    0.0991    0.9669         0         0         0    0.2932

>> max(max(abs(L * U - P * A)))

ans = 3.0022e-16
```

Figure 4. LU decomposition demonstration

which is a lower (respectively) upper triangular matrix, again with elements which are multivectors. P, as in the MATLAB LU decomposition, is a real permutation matrix such that $\mathsf{LU} = \mathsf{PA}$. The `show` command displays the numerical data within a multivector array (whereas by default only size and

```
>> clifford_signature(4,1)
>> A = randm(50)

A = 50x50 Cl(4,1) multivector array

>> tic, [L, U, P] = lu(A); toc
Elapsed time is 22.673430 seconds.
>> max(max(abs(L * U - P * A)))

ans = 7.9795e-09
```

Figure 5. LU decomposition in a larger algebra with a larger matrix

data type information is output). The format of the output from `show` is based on MATLAB's representation of numeric arrays: each non-zero component of the multivector array formed by concatenating L and U is displayed, prefixed by its basis element. The choice here is based on two factors: compactness, and simplicity (of use and of implementation). If we tried to show the matrix of multivectors by displaying every element of the multivector variable in a tabular layout representing the matrix, there would be insufficient space in the command window to do so, without wrapping lines and making a very confusing display. Notice the use of square brackets inside the `show` command to concatenate L and U horizontally, and thus display them side-by-side. The numeric format is controlled by MATLAB, so altering the format setting will alter the format here (*e.g.*, the number of decimal places displayed). The L and U arrays are not output by default – but some information about them is shown. P is a numeric matrix, so MATLAB displays it numerically. To verify the result, the final command computes the difference between LU and PA. Since the result is a multivector matrix (of rounding errors), we use `abs` (the modulus function) to compute the moduli of the rounding errors. The MATLAB `max` functions applied to the result find the largest element of the modulus array, which confirms that $\mathsf{LU} = \mathsf{PA}$ to within rounding error.

To demonstrate that this process works for larger algebras and larger arrays, Figure 5 shows a similar computation in the conformal geometric algebra $C\ell_{3+1,1}$, with 32 coefficients in each multivector, on a matrix with 50 rows and columns (there are 80,000 numeric values in this matrix). The rounding error is larger, as would be expected, but still small compared to the magnitude of the multivector values. The elapsed time to compute this example is shown by the `tic` and `toc` function pair to be around 20 seconds.

7. Key design and implementation issues

Key design issues that we faced were:

- Whether to permit use of one algebra at a time, or multiple algebras:

  - we chose one at a time because it does not make sense to combine multivectors from different algebras;
  - this is how CLICAL operates [8].

  Working with one algebra at a time means that the algebra is always implicit when we create or compute with variables, whereas with multiple algebras the algebra would have to be specified for each operation, or inferred from the algebra of each multivector variable (in an arithmetic expression, for example).
- How to represent multivector variables internally for efficiency of computation and simplicity of coding.
- How to represent each algebra internally.

These decisions affect all later work, whereas many other design choices can be altered fairly easily (*e.g.*, the display format for the numeric values of multivectors, the names of functions *etc.*).

7.1. How multivectors are represented

Recall (from § 5) that in MATLAB, all variables are arrays. A multivector such as:

$$\mathbf{B} = a\,\mathbf{e}_0 + b\,\mathbf{e}_1 + c\,\mathbf{e}_2 + d\,\mathbf{e}_{12}$$

must have arrays (of the same size and type) for the coefficients a, b, c, d. This applies whether $\mathbf{B}$ is a single multivector (a special case, referred to in MATLAB as a *scalar*[12]) or an array of multivectors. For example, if $\mathbf{B}$ is a 2×2 matrix of multivectors, each coefficient a, b, c, d will be stored internally by the toolbox as a standard MATLAB 2×2 matrix, each in a different cell of a 'cell' array[13] indexed from 1 to m, where $m = 2^n$ is the number of coefficients in a multivector. The indexing capability is the reason for using cell arrays, as well as the capability to store a conventional matrix or array in each cell. The choice of cell arrays to store the m numeric coefficients of a multivector variable largely avoids the need to construct our own indexing infrastructure for arrays of multivectors (we still have to provide certain overloadings of MATLAB functions, but nothing like the complexity that would be needed to re-implement the entirety of MATLAB's capabilities). The same idea is used in QTFM [18], for the same reason: simplicity of coding. The details of the cell array implementation are transparent to the user, of course. The data within the cells must be of the same type (*e.g.*, double precision floats, 32-bit integers).

[12]Of course the term *scalar* from linear (or matrix) algebra is not the same as the geometric algebra concept of a scalar ($a\mathbf{e}_0$ in the case of $\mathbf{B}$). It is perfectly possible to have a matrix of scalars in the geometric algebra sense, just as it is possible to have a matrix of pseudoscalars or bivectors *etc.* All of these cases are just special cases of matrices of multivectors.

[13]MATLAB cell arrays are a special type of array, usually used to store disparate types of data in each element. This is contrary to the conventional computer science concept of an array, which requires the ability to index through the array and apply the same operation to every element, but it provides a very useful addition to the normal array concept.

An important issue, which was considered in the original design of the quaternion toolbox [18], but which is more important in Clifford algebras of large dimension, is the storage of zero coefficients. In the quaternion toolbox, a so-called *pure* quaternion with zero scalar part is stored with an empty array for the scalar part. (An empty array in MATLAB has dimension 0×0 or $0 \times k$, and is easily tested for using the built-in function `isempty`.) This means that the multiplication of two pure quaternions takes only 12 floating-point multiplications rather than the 16 that are needed for a full quaternion product. Additionally, the code in the toolbox is sometimes specific to the pure quaternion case, which is easily detected by the empty scalar part.

In a Clifford algebra, computation with vectors, bivectors *etc.* (that is, sparse multivectors with most grades null) is highly likely and were the zero parts of each multivector to be stored explicitly, a great deal of wasted computation would be performed. Therefore, we took the decision that coefficients of a multivector variable which are *exactly* zero are not stored explicitly. Instead a zero coefficient is stored as an 'empty' array. This choice also reduces the amount of memory needed to store an array with zero coefficients. The empty arrays are not visible to the user under normal circumstances: if the user asks for the numeric value of an empty coefficient, the toolbox supplies an appropriately sized array of zeros (matching the size of the non-zero coefficients). When the value of a single multivector is displayed, empty coefficients are suppressed, not displayed as zeros (as shown in Figure 3). When an array of multivectors is displayed, coefficients which are empty arrays are suppressed, but a coefficient which contains some zeros and some non-zero values is displayed in standard MATLAB fashion, as shown in Figure 4. A vector or bivector will be displayed on one line, and the empty grades will be suppressed from the displayed value. Figure 6 shows the use of empty cells within a multivector. We have implemented a simple dump function to permit inspection of the content of a multivector variable, for debugging purposes, and the figure shows the output of this function for the case of a multivector variable with only two non-zero coefficients.

Every multivector variable contains a copy of the three-element array $[p, q, r]$ (the signature in force when the multivector was created). This is shown in Figure 6 in the first part of the output from the dump function. This is done so that key functions such as multiplication can check that the current algebra signature matches the signature inside the multivector: if it does not, an error is raised and the computation stops.

Although it is possible to compute with only one algebra at a time, variables can exist in memory (the MATLAB workspace) simultaneously from more than one algebra. This is an essential feature to permit recursion through algebras, as we discussed in § 3. Now it is possible to explain this in slightly more detail: we will provide a function (possibly internal to the toolbox) that will be able to construct an isomorphic copy of an existing multivector variable in an algebra different to the current algebra using the matrix isomorphisms in Lounesto's book [5, Chapter 16]. Explicitly, each element of the

```
>> M = eye(2) .* (e1 + e3)

M = 2x2 clifford multivector array

>> dump(M)
Signature:
     1     2     0

Multivector:
    [] [2x2 double] [] [2x2 double] [] [] [] []
```

Figure 6. Example of empty cells within a multivector variable: there are two 2×2 identity matrices stored in the positions corresponding to $\mathbf{e}_1$ and $\mathbf{e}_3$. MATLAB indicates an empty matrix by the notation `[]`.

original multivector array will be represented in the new multivector array by a (block) array of multivectors in the new algebra. This new multivector variable can clearly be constructed with a signature different to the current algebra. Then the function will call the `clifford_signature` function to initialise the new algebra (or use a cached copy of the descriptor for the new algebra, as described in the next section). The process of constructing the new multivector is merely a re-arrangement of the numeric values into a cell array of different size. The isomorphisms themselves will likely be coded as tables in the code, not computed by the toolbox.

7.2. How is each algebra represented?

When the user initializes an algebra, the function `clifford_signature` computes some internal information and stores it in a global variable called the *descriptor* which is normally hidden from the user (it can be made visible by a `global` statement issued in the command window, if desired). The descriptor is a data structure with the following fields:

- the signature – the values of p, q, r;
- the number of basis vector elements ($n = p + q + r$);
- the number of basis blade elements in each multivector ($m = 2^n$)
- the multiplication table;
- a table mapping coefficients onto grades;
- an indexing table mapping from lexical indices to numeric indices into the cell array (1 to m).

This information is reasonably compact, and easily saved and restored if we switch from one algebra to another. (The multiplication table requires 3 bytes of memory per entry, and m^2 entries, which is 3 MB of data for an algebra with $n = 10$, $m = 1024$.)

7.3. Data types

The numeric arrays stored inside a multivector can be of any data type supported by MATLAB. The default data type is the MATLAB `double` floating-point type, but integer types are possible. We have implemented a `cast` function for converting between data types, overloading the MATLAB function of the same name. Since the data within a multivector coefficient is a standard MATLAB numeric array, multivectors can have complex coefficients without any special coding in the toolbox. However, as the toolbox develops, some special coding will be necessary to ensure that meaningful results are obtained with complex multivectors, just as in the quaternion toolbox. As in the quaternion toolbox, we will implement checking that all the components of a multivector have the same type, but this is not yet fully done[14], and again, as in QTFM we will provide convenient functions for creating multivector variables with complex coefficients, and extracting the real and imaginary parts.

7.4. The basis elements $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_{12}$ etc.

MATLAB does not provide a way to implement constants, so we cannot define these important multivectors as constants. It is very important to provide these values in a form that appears natural to the user, so that it is possible to type something like: `2 * e2 + 3 * e13` and obtain the expected result $2\mathbf{e}_2+3\mathbf{e}_{13}$. We have therefore implemented the basis elements as *parameterless functions*:

- For any given algebra there are m parameterless functions from `e0` up to `e123...n` which return multivectors with the values $\mathbf{e}_0$ up to $\mathbf{e}_{123\ldots n}$.
- These functions are created and written to disk during initialization of an algebra. If they already exist on disk, *and have the correct content*, they are not modified.
- The user must have write access to the disk space where the toolbox is installed, in order to initialize the files for the first time.

The parameterless function files are not distributed with the toolbox — a large number of files would be needed for algebras with larger values of n. The system we have implemented means that a user who never initialises a large algebra never creates the parameterless functions with long index values (*e.g.*, the file which implements $\mathbf{e}_{123456789}$).

We have limited the parameterless function files to the canonic (lexical) ordering of the basis vector indices, because to do otherwise would result in a huge number of function files being created[15]. This means of course that the user cannot type `e21` without error, but it is of course possible to type `e2 * e1` and obtain the correct result. We could implement an alternative means of creating basis elements with non-canonic ordering. For example, a function

[14]The omission of this checking does not impact on the usability of the toolbox if the user uses only the default `double` data type.

[15]The number of permutations of '123456789' is $9! = 362880$ and this would be the number of files we would need to provide in place of just *one* parameterless function file.

named `e` with a string parameter would allow `e('21')` (the same approach is described for Mathematica in [16]). We have code that does almost this in our test code, and it could be extended with suitable error checking (*e.g.*, to guard against `e('22')`.

There is clearly a problem with the parameterless functions if the user creates a variable with the same name as one of the parameterless functions, but a similar problem also occurs in MATLAB itself, for example, if the user creates a variable with the name `pi`, this will hide the existing MATLAB built-in value of the same name, which will then cause erroneous results if the user's variable does not have the value π.

8. Conclusion

The current implementation of the Clifford Multivector Toolbox provides a basis for future work to develop its capabilities up to and beyond those of the quaternion toolbox. Future work includes the following.

- Many non-trivial functions require implementation and in some cases, research to find how they can be optimally implemented.
- Clifford Fourier transforms [24], or at least a framework for the user to implement them.
- Exponential function, trigonometric functions *etc.*
- Matrix isomorphisms to express multivectors in higher-dimensional algebras using matrices of multivectors in lower-dimensional algebras.
- Better or more appropriate random multivector functions.
- More test code.
- HTML documentation in a similar manner to that in the quaternion toolbox.

Further, we recognise that some users will want to use the toolbox with Octave [25]. The most recent version of this software (Octave 4) has only recently been released, and it is this version that we expect to adapt the toolbox to work with.

E. H. would like to urge readers to apply the knowledge communicated in this paper in accordance with the Creative Peace License [26].

Acknowledgements

Development of this toolbox was partially funded by a Scheme 4 (Research in Pairs) grant number 41408 from the London Mathematical Society. This grant funded a two-week visit by Stephen Sangwine to the International Christian University in Tokyo to work with Eckhard Hitzer in March 2015. We express our thanks to the Programme Committee of the London Mathematical Society for this grant which enabled us to develop the toolbox to an initial public release (version 0.5) from which we have been able to develop a rapid series of incremental releases.

This paper is based on an invited presentation by Stephen Sangwine given on 29 July 2015 at the Applied Geometric Algebra in Computer Science and Engineering Conference held in Barcelona, Spain.

References

[1] Eckhard Hitzer, Tohru Nitta, and Yasuaki Kuroe. Applications of Clifford's geometric algebra. *Advances in Applied Clifford Algebras*, 23 (2):377–404, June 2013. doi:10.1007/s00006-013-0378-4. Available in preprint: `http://arxiv.org/abs/1305.5663`.

[2] Dietmar Hildenbrand. *Foundations of Geometric Algebra Computing*, volume 8 of *Geometry and Computing*. Springer, Berlin, 2013. ISBN 978-3-642-31793-4.

[3] The MathWorks Inc. MATLAB, 1984–2015. `http://www.mathworks.com/products/matlab/`.

[4] M. I. Falcao and H. R. Malonek. Generalized exponentials through Appell sets in $\mathbb{R}^{n+1}$ and Bessel functions. In *AIP Conference Proceedings*, volume 936, pages 738–741, 2007.

[5] Pertti Lounesto. *Clifford Algebras and Spinors*. Number 286 in London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge, second edition, 2001. ISBN 978-0-521-00551-7.

[6] D. Hestenes and G. Sobczyk. *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics*. Springer, Heidelberg, 1984. ISBN 978-9027725615.

[7] E. Hitzer. Introduction to Clifford's geometric algebra. *SICE Journal of Control, Measurement, and System Integration*, 51(4):338–350, April 2012. Available in preprint: `http://arxiv.org/abs/1306.1660`.

[8] P. Lounesto, R. Mikkola, and V. Vierros. CLICAL user manual: Complex number, vector space and Clifford algebra calculator for MS-DOS personal computers. Technical report, Institute of Mathematics, Helsinki University of Technology, 1987. Compiled MS-DOS software application, available from `http://users.aalto.fi/~ppuska/mirror/Lounesto/CLICAL.htm`.

[9] S. Mann, L. Dorst, and T. Bouma. The making of a geometric algebra package in Matlab. Research Report CS-99-27, Computer Science Department, University of Waterloo, Canada, 1999. Available at `https://cs.uwaterloo.ca/research/tr/1999/27/CS-99-27.pdf`.

[10] S. Mann, L. Dorst, and T. Bouma. The making of GABLE: a geometric algebra package in Matlab. In E. Bayro Corrochano and G. Sobczyk, editors, *Geometric Algebra with Applications in Science and Engineering*, chapter 24, pages 491–511. Birkhäuser, Boston, 2001.

[11] Daniel Fontijne. Gaigen 2.5. [Online], 2010. Software library available at: `http://g25.sourceforge.net/`.

[12] Paul C. Leopardi. GluCat: Generic library of universal Clifford algebra templates. [Online], 2007. Software library available at: `http://glucat.sourceforge.net/`.

[13] Florian Seybold. Gaalet - Geometric Algebra ALgorithms Expression Templates. [Online], 2010. Software library available at: `http://gaalet.sourceforge.net/`.

[14] Joachim Pitt, Dietmar Hildenbrand, Christian Schwinn, Patrick Charrier, and Christian Steinmetz. GAALOP - Geometric Algebra ALgorithms OPtimizer. [Online], 2008–2016. Software library available at: `http://www.gaalop.de/`.

[15] R. Abłamowicz and B. Fauser. Clifford/bigebra, a Maple package for Clifford (co)algebra computations. Available at `http://www.math.tntech.edu/rafal/`, 2011. ©1996-2011, RA&BF.

[16] G. Aragon-Camarasa, G. Aragon-Gonzalez, J. L. Aragon, and M. A. Rodriguez-Andrade. Clifford algebra with Mathematica. Preprint `http://arxiv.org/abs/0810.2412`, October 2008.

[17] Stephen J. Sangwine and Eckhard Hitzer. Clifford Multivector Toolbox. [Online], 2015. Software library available at: `http://clifford-multivector-toolbox.sourceforge.net/`.

[18] Stephen J. Sangwine and Nicolas Le Bihan. Quaternion Toolbox for Matlab®, version 2 with support for octonions. [Online], 2013. Software library available at: `http://qtfm.sourceforge.net/`.

[19] Stephen J. Sangwine and Nicolas Le Bihan. Quaternion singular value decomposition based on bidiagonalization to a real or complex matrix using quaternion householder transformations. *Applied Mathematics and Computation*, 182(1):727–738, 1 November 2006. doi:10.1016/j.amc.2006.04.032.

[20] Nicolas Le Bihan and Stephen J. Sangwine. Jacobi method for quaternion matrix singular value decomposition. *Applied Mathematics and Computation*, 187(2):1265–1271, 15 April 2007. doi:10.1016/j.amc.2006.09.055.

[21] Salem Said, Nicolas Le Bihan, and Stephen J. Sangwine. Fast complexified quaternion Fourier transform. *IEEE Trans. Signal Process.*, 56(4): 1522–1531, April 2008. ISSN 1053-587X. doi:10.1109/TSP.2007.910477.

[22] C. Perwass, C. Gebken, and G. Sommer. Estimation of geometric entities and operators from uncertain data. In W. G. Kropatsch, R. Sablatnig, and A. Hanbury, editors, *PATTERN RECOGNITION, PROCEEDINGS, 27th Annual Meeting of the German Association for Pattern Recognition, Vienna University of Technology, Vienna, Austria, 31 August – 2 September*, volume 3663 of *Lecture Notes in Computer Science*, pages 459–467, Berlin, 2005. Springer-Verlag.

[23] Dominik Schulz, Jochen Seitz, and Joao Paulo C. Lustosa da Costa. Widely linear SIMO filtering for hypercomplex numbers. In *IEEE Information Theory Workshop (ITW 2011), 16-20 October*, Paraty, Brazil, 2011. IEEE.

[24] Fred Brackx, Eckhard Hitzer, and Stephen J. Sangwine. History of quaternion and Clifford Fourier transforms and wavelets. In Eckhard Hitzer and Stephen J. Sangwine, editors, *Quaternion and Clifford Fourier Transforms and Wavelets*, pages xi–xxvii. Birkhäuser/Springer, Basel, Switzerland, 2013. ISBN 978-3-0348-0602-2. doi:10.1007/978-3-0348-0603-9.

[25] John W. Eaton et al. GNU Octave, 1994–2015. Open source software application available at: `http://www.gnu.org/software/octave/index.html`.

[26] Eckhard Hitzer. The Creative Peace License, 15 July 2015. Available at: `https://gaupdate.wordpress.com/2011/12/14/the-creative-peace-license-14-dec-2011/`.

Stephen J. Sangwine
School of Computer Science and Electronic Engineering, University of Essex,
Wivenhoe Park, Colchester, CO4 3SQ, United Kingdom.
e-mail: `sjs@essex.ac.uk`

Eckhard Hitzer
Osawa 3-10-4, House M472, Mitaka-shi 181-0015, Tokyo, Japan
e-mail: `hitzer@icu.ac.jp`