Exploring Lazy Evaluation and Compile-Time Simplifications for Efficient Geometric Algebra Computations

Leandro A. F. Fernandes

Abstract Mathematical function libraries for scientific computation play an essential role in scientific development. These libraries allow researchers to focus their efforts on solving higher-level problems while the implementations provided by the libraries make good use of available computer resources. The Geometric Algebra Template Library (`GATL`) is a C++ library of data structures and mathematical functions for arbitrary Geometric Algebras (GAs). `GATL` uses template metaprogramming to implement a lazy evaluation strategy at compile-time. This way, `GATL` is capable of performing optimizations on the programs during the compilation of executable files, reducing the computational cost that programs will have at runtime. More specifically, we have designed `GATL` to automatically execute low-level algebraic manipulation in the procedures described by the programmer using GA operations. The aim of `GATL` at compile-time is to simplify each described procedure by performing symbolic optimizations on expressions, leading to more efficient programs.

1 Introduction

It is well-known that Geometric Algebra (GA) is a powerful mathematical system encompassing concepts like Complex Numbers, Quaternion Algebra, Grassmann-Cayley Algebra, and Plücker Coordinates under the same framework [9, 16, 18, 21]. GA is mainly based on Clifford Algebra, but with a strong emphasis on geometric interpretation. As such, it is an appropriate mathematical tool for modeling and solving geometric problems in physics, chemistry, engineering, and computer science.

L. A. F. Fernandes (✉)
Instituto de Computação, Universidade Federal Fluminense (UFF),
Av. Gal. Milton Tavares de Souza, Niterói, Rio de Janeiro 24210-346, Brazil
e-mail: laffernandes@ic.uff.br

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

S. Xambó-Descamps (ed.), *Systems, Patterns and Data Engineering with Geometric Calculi*, SEMA SIMAI Springer Series 13, https://doi.org/10.1007/978-3-030-74486-1_6

This contribution discusses the application of the lazy evaluation strategy at compile time as a promising approach for the implementation of efficient programs based on GA. Lazy evaluation defers the evaluation of expressions until other computations need the expressions' results. We present the Geometric Algebra Template Library (`GATL`) as proof of concept of how to explore lazy evaluation at compile time to reduce both computational cost and memory footprint of programs. `GATL` is a high-level C++ library that includes a data structure that represents GA expressions that operate multivectors in arbitrary metric spaces and dozens of operations of this algebra. In contrast to other lazy solutions such as `Gaalet` [25], `GATL` conducts symbolic optimizations on expressions during compilation. Solutions like `Gaalop` [5, 17] also execute algebraic manipulations at compile time to simplify expressions. In this case, the solution uses an external tool to perform symbolic optimizations and replaces the original source code by the optimized counterpart version of it. `GATL`, on the other hand, performs symbolic optimizations by the ingenious use of the metaprogramming template capabilities of C++.

In Sect. 2, we present an overview of the implementation strategies that have been employed in defining code optimizers, libraries, and library generators for GA. Section 3 describes the internal structure of `GATL` and how it implements lazy evaluation and compile-time simplification. The performance of other solutions for GA is compared to `GATL` in Sect. 4. Finally, we draw our conclusions in Sect. 5.

! Attention

It is assumed that the reader is familiar with GA. Please refer to Dorst *et al.* [9], Kanatani [18], and Perwass [21] for a complete reference on the subject. Here you find the notational conventions used throughout this contribution:

$\mathcal{C}\ell_{r,p,q}$	Clifford Algebra with signature (p, q, r), where $n = p + q + r$.
e_i	i-th Unit basis vector that square to $+1$ in an orthonormal basis.
e_+, e_-	Extra unit basis vectors that square to $+1$ and -1, respectively.
n_o, n_∞	Null basis vectors interpreted, respectively, as origin point and point at infinity in conformal model.
ϕ	Angle in radians.
x, y, z	Scalar value, typically coordinates in 3-dimensional spaces.
a, b	General vector (1-blade).
$\mathcal{R}$	General rotor.
A, B	General multivector.
AB	Geometric product of A and B.
$A \wedge B$	Outer product of A and B.
$\widetilde{A}$	Reverse of A.

2 Overview of Implementation Strategies for Geometric Algebra

One can classify the solutions that provide data structures and mathematical functions to work with GA in terms of their type as *code optimizers*, *libraries*, or *library generators*. The solutions can employ two different strategies for evaluating the implemented operations. Namely, *lazy evaluation* and *eager evaluation*. We call *implementation strategy* the combination of a type of solution with an evaluation strategy. Table 1 relates some existing solutions with their classification regarding their type, evaluation strategy, supported data types at runtime, and programming languages.

Section 1 introduces the lazy evaluation as the strategy that defers the evaluation of expressions until other computations need the expressions' results. In eager evaluation, the arguments to a function are always evaluated completely before the function is applied. It is important to emphasize that the actual evaluation strategy is intrinsic to the programming language. All programming languages mentioned in this section employ the eager evaluation strategy. But in all of them, it is possible to implement data structures that simulate laziness. Therefore, the solutions indicated as "lazy" make use of implementation tricks to simulate laziness in eager languages.

Table 1 Classification of solutions for GA regarding their type, evaluation strategy, supported data types at runtime, and the programming languages

Solution	Type	Evaluation Strategy	Runtime Data types	Programming languages
`clifford` [1]	Library	Eager	Numeric	Python 3
`Gaalet` [25]	Library	Lazy	Numeric, Symbolic†	C++
`Gaalop` [5, 17]	Code Optimizer	Eager	Numeric	C, C++, CUDA, OpenCL, MATLAB
`Gaigen` [13]	Library Generator	Eager	Numeric	C, C++, C#, Java
`galgebra` [3, 23]	Library	Eager	Symbolic	Python 2, Python 3
`Gallant` [8, 12]	Library	Eager	Numeric	Java
`ganja.js` [7]	Library Generator	Eager	Numeric	C++, C#, Javascript, Python 3, Rust
`Garamon` [2]	Library Generator	Eager	Numeric	C++
`GATL`	Library	Lazy	Numeric, Symbolic	C++
`GluCat` [19]	Library	Eager	Numeric	C++, Python 2
`GMac` [10]	Code Optimizer	Eager	Numeric	C++, C#, VB.NET, F#, IronPython
`Grassmann.jl` [24]	Library	Eager	Numeric, Symbolic	Julia
`Klein` [20]	Library	Eager	Numeric	C++
`Liga` [4]	Library	Eager	Numeric	Julia
`TbGAL` [26]	Library	Eager	Numeric, Symbolic†	C++, Python 2, Python 3
`Versor` [6]	Library	Eager	Numeric, Symbolic†	C++

†As a template-based C++ solution, it is likely to support symbolic data types for multivector coefficients at runtime. However, this capability has not been asserted by the authors

Code Optimizers. When a programmer uses a code optimizer, he/she writes snippets of the source code of his/her program using the representation language provided by the optimizer. The representation language is not necessarily the same used to write the rest of the program. Before compiling the whole program, the optimizer analyzes the snippets of specialized code, maps the input variables to symbols, and converts the calls to operations of the algebra into mathematical expressions. Such expressions, in turn, are manipulated algebraically in order to simplify them. The result of the manipulation is then translated into source code in the programming language chosen by the programmer, and compiled with the rest of the program.

`Gaalop` [17] is an example of code optimizer for GA. It is a software that expects procedures implemented using `CLUScript` [22] as input and converts them into simpler C, C++, CUDA, OpenCL, or MATLAB code. `Gaalop` uses the `Maxima` system for the manipulation of symbolic expressions. Charrier *et al.* [5] developed a `Gaalop` Pre-Compiler for C++, CUDA, and OpenCL that takes `CLUScript` snippets declared in `pragma` directives [27] and optimize them producing inline code for a given source file. The source code produced by `Gaalop` is evaluated eagerly. Nevertheless, in theory, it does not show any performance issues at runtime since the simplification process runs before the compiling process is triggered.

`GMac` [10] is another code optimizer that produces code fragments from a description of an algorithm in a domain-specific language. It can be configured to generate textual code files using an API that is accessible through any .NET language, including C++, C#, VB.NET, F#, and IronPython. Currently, `GMac` depends on `Wolfram Mathematica` for symbolic processing. The generated code is evaluated eagerly.

Libraries. When using libraries, the programmer declares variables whose types are data structures provided by the solution and calls subroutines that represent GA operations implemented by the library. `clifford` [1], `galgebra` [3, 23], `Gallant` [8, 12], `GluCat` [19], `Grassmann.jl` [24], `Klein` [20], `Liga` [4], `TbGAL` [26], and `Versor` [6] are examples of GA libraries that employ the eager evaluation strategy. For eager libraries, the only possible optimizations are those that affect individual calls of subroutines, because eager evaluation solves each subroutine call before passing its result as an argument for the next call.

`Versor` uses the metaprogramming capabilities of C++ templates to perform compile-time specialization of subroutines based on the arguments passed to them, producing results that compute and store only the multivector components whose coefficients may be non-zero at runtime. To perform such simplification at compile-time, `Versor` assumes that the coefficients stored by the input multivectors are non-zero values since the actual values will only be known at runtime. In contrast, the basis blade associated with each coefficient is known at compile time. Through the algebraic manipulation of the basis blades of the input multivectors, the library can predict which components of the resulting multivector will be equal to zero and which will be different from zero, eliminating the need for calculating the former and maintaining the storage and computation of the latter. However, as illustrated in the following code,

eager evaluation prevents the library from suppressing intermediate results that would be unnecessary when considering a sequence of operations.

Program Code

Example of source code using the `Versor` library. The comments on the right present the state of each variable at runtime:

```
1  #include <vsr/space/vsr_ega3D_types.h>
2  #include <vsr/detail/vsr_generic_op.h>
3
4  using namespace vsr::ega;
5  using namespace vsr::nga;
6
7  int main() {
8    double x, y, z, phi_deg;
9    std::cin >> x >> y >> z; // x: 10, y: 20, z: 30
10   std::cin >> phi_deg;      // phi_deg: 90
11
12   double phi = phi_deg * (M_PI/180.0);    // phi: 1.5708
13
14   auto a = Vec(x, y, z);                  // a: 10*e1 + 20*e2 + 30*e3
15   auto R = Gen::rot(Biv::xy * (-phi/2)); // R: 0.7071 - 0.7071*e12
16   auto b = R * a * ~R;                    // b: -20*e1 + 10*e2 + 30*e3 + 0*e123
17
18   return EXIT_SUCCESS;
19 }
```

The * and ~ operators implement the geometric product and the reverse, respectively.

Discussion The rotation of a vector $a = x_1 e_1 + y_1 e_2 + z_1 e_3$ made by a sandwiching product with a rotor $\mathcal{R} = \cos(\phi/2) - \sin(\phi/2) e_1 \wedge e_2$ under Euclidean metric, where ϕ and $e_1 \wedge e_2$ are, respectively, the rotation angle and the unit rotation plane, produces a vector $b = \mathcal{R} a \widetilde{\mathcal{R}} = x_2 e_1 + y_2 e_2 + z_2 e_3$. By the grade preservation property of outermorphisms, the multivector components with grade different than 1 will always be zero in b. But the example shows that `Versor` does not suppress the computation of the component of grade 3 in `b` (line 16). In eager solutions, suppression is only possible by implementing specialized routines, such as `lhs.spin(rhs)`, which returns the multivector resulting from applying the rotor `rhs` to `lhs`.

Other eager libraries presented in Table 1 do not simplify subroutine according to their arguments at compile time. The only exception is `clifford` [1], which uses the just-in-time (JIT) compilation feature of Python to improve the performance of subroutine calls. The JIT functionality was extended for $\mathcal{C}\ell_{4,1}$ by `Gajit` [15].

Libraries for GA that employ the lazy evaluation strategy provide two types of data structures to represent multivectors. The more straightforward kind of data structure represents *concrete multivectors*, *i.e.,* multivectors that store their components in memory at runtime. The other kind of data structure encodes (lazy) *multivector expressions*, *i.e.,* expressions obtained by operating concrete multivectors and other expressions. Typically, a multivector expression `arg` can be evaluated implicitly by assigning it to an existing concrete multivector variable or explicitly by calling functions like `eval(arg)` and `arg.eval()` that return a concrete multivector.

To the best of our knowledge, Gaalet [25] and GATL are the only libraries for GA that employ the lazy evaluation strategy. By combining C++ templates metaprogramming and lazy evaluation, both solutions extend the compile-time specialization capability of Versor to include the suppression of unnecessary computations over sequences of operations, inline function calls, and avoid storage of temporary values. The main difference between Gaalet and GATL is that GATL's lazy evaluation system of template expressions implements an algebraic manipulator that conducts symbolic manipulations equivalent to those performed by Gaalop, but without the need for an external tool. Also, unlike Gaalet and Versor, GATL allows multivector coefficients to assume known values at compile-time, thus reducing storage cost at runtime. This latter feature is especially useful in representing points with unit homogeneous coordinate and constant multivector.

The code examples that follow illustrate, respectively, the use of Gaalet and GATL in solving the same rotation case presented for Versor.

Program Code

Example of source code using the Gaalet library. The comments on the right present the state of each concrete variable at runtime, or the expression encoded by the non-concrete multivector variable b_ as nested operations:

```
1  #include <gaalet.h>
2
3  using ga3e = gaalet::algebra<gaalet::signature<3, 0>, double>;
4
5  int main() {
6    double x, y, z, phi_deg;
7    std::cin >> x >> y >> z; // x: 10, y: 20, z: 30
8    std::cin >> phi_deg;     // phi_deg: 90
9
10   double phi = phi_deg * (M_PI/180.0); // phi: 1.5708
11
12   ga3e::mv<1, 2, 4>::type a{x, y, z};              // a: 10*e1 + 20*e2 + 30*e3
13   ga3e::mv<0, 3>::type R{cos(phi/2), -sin(phi/2)}; // R: 0.7071 - 0.7071*e12
14
15   auto b_ = grade<1>(R * a * ~R); // b_: grade<1, gp<gp<mv<0, 3>, mv<1, 2, 4> >,
16                                   //        reverse<mv<0, 3> > > >
17
18   auto b = eval(b_);              // b: -20*e1 + 10*e2 + 30*e3
19
20   return EXIT_SUCCESS;
21 }
```

The * and ~ operators implement the geometric product and the reverse, respectively.

Discussion This example is equivalent to the one presented for Versor. However, notice in line 15 that the variable b_ represents a multivector expression instead of a concrete multivector. The last step in this expression is the extraction of the grade 1 components of the multivector resulting from the application of the rotor $\mathcal{R}$ to vector a. The nested structure of the expression in this variable is defined at compilation time. At runtime, variable b (line 18) receives a concrete multivector that does not include the calculation of the suppressed grade 3 component.

Program Code

Example of source code using the GATL library. The comments on the right present the state of each concrete variable at runtime, or the expression encoded by the non-concrete multivector variables a_, phi_, R_, and b_:

```
#include <gatl/ga3e.hpp>

using namespace ga3e;

int main() {
  double x, y, z, phi_deg;
  std::cin >> x >> y >> z; // x: 10, y: 20, z: 30
  std::cin >> phi_deg;     // phi_deg: 90

  double phi = phi_deg * (M_PI/180.0); // phi: 1.5708

  auto a = vector(x, y, z);            // a: 10*e1 + 20*e2 + 30*e3

  auto lazy = make_lazy_context(a, scalar(phi));
  auto [a_, phi_] = lazy.arguments();  // a_: x_*e1 + y_*e2 + z_*e3, phi_: phi_

  auto R_ = cos(phi_/c<2>) - sin(phi_/c<2>)*e1^e2; // R_: cos(phi_/2)
                                                   //     - sin(phi_/2)*e12
  auto b_ = R_ * a_ * ~R_; // b_: (cos(phi_)*x_ - sin(phi_)*y_)*e1
                           //     + (sin(phi_)*x_ + cos(phi_)*y_)*e2 + z_*e3

  auto b = lazy.eval(b_);  // b: -20*e1 + 10*e2 + 30*e3

  return EXIT_SUCCESS;
}
```

The *, ^, and ~ operators implement the geometric product, outer products, and reverse, respectively. The helper functions vector(arg1, ...) and scalar(arg) in lines 12 and 14 create multivector structures from arithmetic types or scalar multivectors. The make_lazy_context(arg1, ...) function in line 14 takes a set of multivectors as input arguments and creates a data structure that relates each concrete coefficient and basis blade in the components of the input to symbols composing the multivector expressions returned by the lazy.arguments() function in line 15. Details of its operation will be presented in Sect. 3. The helper template c<arg> in line 17 initializes a scalar multivector that wraps a static constant value.

Discussion This example is equivalent to those presented for Versor and Gaalet. By comparing the expressions presented here for b_ with the expression presented for the variable b_ in the Gaalet's example, we observe that, unlike Gaalet, GATL does not store expressions by nesting operations. The existence of the lazy context (line 14) allows algebraic manipulations to be performed at compile time while defining each new expression, leading to simpler computations (line 19) when calling the lazy.eval(arg) function at runtime (line 22).

Library Generators. For this kind of solution, the programmer first defines the parameters of his/her programming language and GA of interest in the generator software. The generator then produces implementations of data structures and GA operations from scratch. This set of implementations, in turn, can be used by the programmer like conventional libraries in writing his/her programs. Gaigen [13], ganja.js [7], and Garamon [2] are examples of library generators for GA (Table 1). The three solutions generate eager libraries but only Gaigen implements an optimization mechanism

for individual subroutines based on use cases. When using Gaigen, the programmer must generate the initial library without optimizations and use it in his/her program with the profiling functionality enabled. Profiling data is then interpreted by Gaigen, which produces an optimized version of the GA library by pruning unused multivector components. The final compilation of the program must be done considering the optimized version of the generated library.

3 The Geometric Algebra Template Library (GATL)

GATL is a C++17 template library defined in its headers files. There is no binary library to link to, no configured header file, or dependencies to external libraries. Therefore, if you want to use GATL, you can use the header files right away. Section 3.1 presents an overview of GATL's front-end, *i.e.,* the set of data structure and subroutines available to the programmer while using the library. The internal organization and implementation of the library correspond to the back-end presented in Sect. 3.2.

3.1 GATL's Front-End

According to the GATL's conventions, the root directory for the header files that the programmer will include in his/her source files is the gatl folder. The header file that encloses all GATL implementations is gatl/ga.hpp. From this header, the programmer has access to the ga namespace, which is the main namespace of the GATL library. In C++, namespaces are declarative regions that provide scope to the names of the types, subroutines, and constants inside it. The following classes correspond to the most important data structures in GATL's front-end:

`clifford_expression<CoefficientType, Expression>`
: A Clifford expression. The template parameter CoefficientType can be either a native arithmetic type (*e.g.,* double, float, int) or third-party classes implementing arbitrary-precision arithmetic or symbolic computation. It specifies the data type of the multivector's coefficients. The Expression parameter is a type describing the internal structure of the Clifford expression. Depending on the definition of this parameter, the Clifford expression will be classified as *concrete multivector* or (lazy) *multivector expression* (see Sect. 2).

`lazy_context<CliffordExpression₁, CliffordExpression₂, ...>`
: A class to define lazy arguments for lazy evaluation of Clifford expressions. It keeps references to the set of instances of clifford_expression<...> informed as input argument and produces multivector expressions having the concrete coefficients and basis blades in the input set replaced by symbols.

metric_space<MetricSpaceType>
The base metric space class from which all specialized metric space classes derive. The parameter MetricSpaceType must be one of those specialized spaces.

That's it. Only three classes! And most of the time, the programmer does not have to worry about parameterizing these classes since GATL provides helper functions and auxiliary headers for pre-defined GAs. Also, it is strongly recommended to use the auto placeholder type specifier [27] whenever possible.

The last program code presented in Sect. 2 illustrates the use of GATL. In this example, gatl/ga3e.hpp (line 1) is an auxiliary header defining the namespace ga3e (line 3) for a GA for 3-dimensional Euclidean geometry ($\mathcal{C}\ell_{3,0}$) with basis vectors $\{e_1, e_2, e_3\}$. The specialized metric space class in this example is euclidean_metric_space<3>. It is used inside the header to declare the static constant object space. This object is implicitly passed as argument to all metric and non-metric products called in this example, such as the geometric and outer products (lines 17 and 19). e1 and e2 (line 17) are also static constant objects defined in the namespace ga3e. The constants e1 and e2 and the variables a, a_, phi_, R_, b_, and b are instances of the clifford_expression<...> class assuming different types in the Expression parameter. Notice the use of the auto placeholder to deduce the type of a variable from the initializer. In line 14, the lazy variable is of type lazy_context<...>. Typically, we use the helper function make_lazy_context(arg1, ...) to initialize it. We also use the helper functions vector(arg1, ...) and scalar(arg) to initialize Clifford expressions in lines 12 and 14. In line 17, the helper template c<arg> initializes a clifford_expression<...> whose Expression parameter wraps the constant scalar value 2. The difference between a wrapped constant value and a regular constant value is that the former can be handled at compile time by the symbolic simplification mechanism implemented by GATL's back-end (Sect. 3.2).

In its current version, GATL includes the following set of namespaces for specific GAs. These namespaces already use the ga namespace. Also, they overload all metric operations in ga by setting their respective metric:

ga1e, ga2e, ga3e, ga4e, ga5e
GAs for Euclidean geometry ($\mathcal{C}\ell_{n,0}$), with basis vectors $\{e_1, e_2, \cdots, e_n\}$.

ga1h, ga2h, ga3h, ga4h
GAs for homogeneous geometry ($\mathcal{C}\ell_{d+1,0}$), with basis vectors $\{e_1, e_2, \cdots, e_d, e_+\}$.

ga1m, ga2m, ga3m
GAs for Minkowski spaces ($\mathcal{C}\ell_{d+1,1}$), with basis vectors $\{e_1, e_2, \cdots, e_d, e_+, e_-\}$.

ga1c, ga2c, ga3c
GAs for conformal geometry ($\mathcal{C}\ell_{d+1,1}$), with basis vectors $\{e_1, e_2, \cdots, e_d, n_o, n_\infty\}$.

The header file for each namespace is its name followed by the .hpp extension, *e.g.*, gatl/ga3e.hpp, gatl/ga3h.hpp, gatl/ga3c.hpp, and so on. Please, refer to examples in the GATL repository [11] to see how to declare Clifford Algebras $\mathcal{C}\ell_{r,p,q}$ with arbitrary (p, q, r) signatures and assuming arbitrary metric matrices.

The documentation in [11] also includes the complete reference to helper functions and GA operations implemented by `GATL`. Among them, we highlight:

- `+rhs` — Unary plus.
- `-rhs` — Unary minus.
- `lhs + rhs` — Addition.
- `lhs - rhs` — Subtraction.
- `gp(lhs, rhs [, mtr])` — Geometric product (same as `lhs * rhs`).
- `op(lhs, rhs [, mtr])` — Outer product (same as `lhs ^ rhs`).
- `rp(lhs, rhs [, mtr])` — Regressive product.
- `lcont(lhs, rhs [, mtr])` — Left contraction (same as `lhs < rhs`).
- `rcont(lhs, rhs [, mtr])` — Right contraction (same as `lhs > rhs`).
- `dot(lhs, rhs [, mtr])` — Dot product (same as `lhs | rhs`).
- `hip(lhs, rhs [, mtr])` — Hestenes' inner product.
- `sp(lhs, rhs [, mtr])` — Scalar product.
- `cp(lhs, rhs [, mtr])` — Commutator product.
- `dp(lhs, rhs [, tol] [, mtr])` — Delta product.
- `conjugate(arg)` — Clifford conjugation.
- `involute(arg)` — Grade involution.
- `reverse(arg)` — Reversion (same as `~arg`).
- `rnorm_sqr(arg [, mtr])` — Squared reverse norm.
- `rnorm(arg [, mtr])` — Reverse norm.
- `inv(arg [, mtr])` — Inverse of the given versor.
- `dual(arg [, pseudoscalar [, mtr]])` — Dualization operation.
- `undual(arg [, pseudoscalar [, mtr]])` — Undualization operation.

According to `GATL` conventions, `lhs` and `rhs` are informal shorthand for, respectively, the left-hand side and the right-hand side arguments of binary operations. The `mtr` argument must be an instance of the `metric_space<...>` class, while all other arguments can be either instances of the `clifford_expression<...>` class, native arithmetic types, or third-party classes implementing arbitrary-precision arithmetic or symbolic computation.

3.2 *GATL's Back-End*

All namespaces mentioned in Sect. 3.1 declare a nested `detail` namespace. This is the namespace where the magic happens, *i.e.,* the namespace of `GATL`'s back-end. All data types described in this section are defined in the `detail` namespace.

3.2.1 Expression Structure

The behavior of GATL at compile time and runtime is related to the definition of the Expression parameter of the instances of clifford_expression<...> involved in each operation. Recall that as a template parameter, Expression is a type, not an instance. It represents the description of the structure of the respective instance of clifford_expression<...>. The possible types for Expression are:

component<Coefficient, BasisBlade>
: A single multivector component whose coefficient is described by the template parameter Coefficient as a real-valued expression, and a basis blade described by the template parameter BasisBlade.

add<Component$_1$, Component$_2$, ...>
: The addition of two or more components of type component<Coefficient, BasisBlade> having different basis blades.

When the BasisBlade parameter is constant_basis_blade<BasisVectors>, it means that the basis blade of the component is known at compile time. Here, BasisVectors is an unsigned integer value whose bitset represents an unit basis blade. For instance, in Euclidean geometry, $1 = 0001_b$ stands for e_1, $2 = 0010_b$ stands for e_2, $3 = 0011_b$ stands for $e_1 \wedge e_2$, and so on.

Basis blades defined at runtime are represented by setting BasisBlade to dynamic_basis_blade<PossibleGrades, Bitset>, where PossibleGrades is an unsigned integer value known at compile time, indicating the grades that the runtime-defined component may assume (*e.g.,* $1 = 0001_b$ stands for grade 0, $2 = 0010_b$ stands for grade 1, $3 = 0011_b$ stands for grades 0 and 1, and so on). It is important to notice that one may predict the possible grades of the outcome of a GA operation if you know the grades of the arguments, even when the actual basis blades are unknown. For instance, the grade of the outer product of a 2-blade and a 3-blade will be 5 unless $n < 5$. In this case, the resulting grade may be set to any value, and the resulting coefficient will be 0 for sure. GATL explores this observation to perform simplifications at compile-time, even on expressions with runtime-defined multivector. In components with dynamic basis blades, the Bitset parameter is a type describing a bitwise expression with at least one bitset defined at runtime.

So far, only two templates parameters where not described in detail: Coefficient and Bitset. In GATL, the atomic types for Coefficient expressions are:

constant_value<Value>
: This type wraps a compile-time defined integer value.

stored_value
: This type indicates that the value of the coefficient is stored by the current instance of clifford_expression<...>.

get_value<Tag, Index> and get_map_values<Tag, Index>
: These types can be interpreted as variables within an algebraic expression. The pair of compile-time defined integer values Tag and Index makes it possible for a lazy_context<...> to unequivocally identify the value stored by one of the instances of clifford_expression<...> informed to it as initialization argument. Tag indexes the argument passed to the lazy context and Index indexes the value or values stored by this argument. To different lazy contexts do not confuse their arguments by defining conflicting Tag values, each lazy_context<...> instance deduces the minimum value it can assign to Tag by checking which values have already been used in its arguments.

function<Name, Argument$_1$, Argument$_2$, ...>
: The function represented by this type is defined by the Name parameter at compile time. The values that Name can assume includes, but is not limited to, add, mul, power, sine, cosine, and if_else. The expected number of arguments depends on the function's name, and they can encode real-valued expressions or logical expressions defined on the same set of atomic types.

The atomic types for Bitset are similar to those defined for Coefficient: constant_value<Value>, stored_bitset, get_bitset<Tag, Index>, get_map_bitsets<Tag, Index>, and function<Name, Argument$_1$, ...>. The difference is that atomic types for Bitset expressions are related to basis blades instead to the values of coefficients. Thus, among the possible values for the Name parameter, we highlight bitwise_and, bitwise_or, bitwise_xor, and if_else.

When the Expression parameter of a clifford_expression<...> only includes components with constant_basis_blade<...>, constant_value<...>, and function<...> types, we say that we have a *concrete multivector completely defined at compile time*. These multivectors do not occupy space in the compiled program because they do not store anything. Also, they do not depend on other multivectors. The e1 object in the sample code is an example of such multivector type:

```
clifford_expression<          // e1
  long,                       // CoefficientType
  component<                  // Component
    constant_value<1>,        //   Coefficient (same as 1)
    constant_basis_blade<1>   //   BasisBlade (1 = 0001_b, or e_1)
  >
>
```

The objects e2 and c<2> are also concrete multivectors defined at compile time.

The object b is an example of *concrete multivector defined at runtime*. As can be seen in its type definition, b stores three double-precision floating-point values:

```
clifford_expression<          // b
  double,                     // CoefficientType
  add<
    component<                // First Component
      stored_value,           //   Coefficient (runtime defined)
      constant_basis_blade<1> //   BasisBlade (1 = 0001_b, or e_1)
    >,
    component<                // Second Component
      stored_value,           //   Coefficient (runtime defined)
      constant_basis_blade<2> //   BasisBlade (2 = 0010_b, or e_2)
    >,
    component<                // Third Component
      stored_value,           //   Coefficient (runtime defined)
      constant_basis_blade<4> //   BasisBlade (4 = 0100_b, or e_3)
    >
  >
>
```

Thus, the size of b at runtime is 3×8 bytes $= 24$ bytes. In GATL, concrete multivectors store their runtime-defined coefficients and bitsets using sequence containers of type std::array or associative containers of type std::map. The associative container is used when more than one component of the multivector resulting from an operation have overlapping sets of possible grades, and runtime-defined bitsets. In this case, the description of the Expression parameter is simplified in the final clifford_expression<...> type and all components having overlapping PossibleGrades sets are put in the same std::map. Otherwise, the sequential container is chosen to store the data. Notice that it is possible for a clifford_expression<...> type to use std::array and std::map simultaneously to store componets with different configurations.

Lazy multivector expressions depend on concrete multivectors to have the values of their coefficients or the bitsets of their components computed at runtime. In GATL, the Expression parameter of all lazy clifford_expression<...> objects includes at least one getter type. For instance, the type of the R_ object in the sample code includes get_value<2, 0>. Thus, the object R_ depends on the 1st coefficient (std::array is zero-base indexed [27]) of the 2nd argument of the lazy context:

```
clifford_expression<          // R_
  long,                       // CoefficientType
  add<
    component<                // First Component
      function<               //   Coefficient (same as cos(φ/2))
        cosine,
        function<
          mul,
          function<power, constant_value<2>, constant_value<-1> >,
          get_value<2, 0>
        >
      >,
      constant_basis_blade<0> //   BasisBlade (0 = 0000_b, or 1)
    >,
    component<                // Second Component
      function<               //   Coefficient (same as -sin(φ/2))
        mul,
        constant_value<-1>,
        function<
          sine,
          function<
            mul,
            function<power, constant_value<2>, constant_value<-1> >,
            get_value<2, 0>
          >
        >
      >,
      constant_basis_blade<3> //   BasisBlade (3 = 0011_b, or e1 ∧ e2)
    >
  >
>
```

Lazy multivector expressions do not store anything. Thus, variables of this kind do not take up memory space at runtime. Of course, if you compile your program in debug mode, then these variables will have 1 byte each to be addressed by the debugger. However, in release mode, they are usually optimized by the compiler.

3.2.2 Expression Simplification and Evaluation

When we state that GATL employs the lazy evaluation strategy, we are referring to the possibility for the programmer to start a lazy context, operate the lazy context arguments by calling GA operations, and evaluate such expressions later. However, strictly speaking, the explicit use of a lazy context by the programmer is optional. This means that the sample program code presented for GATL could have been written without using the lazy context. However, in this case, the evaluation of the sequence of operations would be in an eager fashion, and the lazy multivectors would be concrete multivectors with coefficients defined at runtime. In addition, variable b would have the extra component of grade 3 observed in the sample code of the Versor library, since the suppression of this component would not be foreseen by GATL. The programmer

would choose to use a lazy context if he/she thinks that there are pertinent simplifications to be made in a certain part of his/her program.

Nevertheless, all subroutines implemented by GATL initializes a lazy context to benefit from any simplifications that may arise from the algebraic manipulation of the components of its arguments. It means that the implementation of all GATL subroutines looks like this:

```
template<class CT1, class E1, class CT2, class E2, class MST>
decltype(auto) foo(
  clifford_expression<CT1, E1> const &arg1,
  clifford_expression<CT2, E2> const &arg2,
  metric_space<MST> const &mtr
) {
  auto [lazy, arg1_, arg2_] = make_lazy_context_tuple(arg1, arg2);
  // Compute 'result_' using 'arg1_', 'arg2_', and 'mtr'.
  return lazy.eval(result_);
}
```

Here, CT*i* and E*i* encode the types assumed by, respectively, the template parameters CoefficientType and Expression of the *i*-th argument of the function foo(…), MST is the MetricSpaceType parameter, and make_lazy_context_tuple(…) is a helper shorthand function for calling lazy = make_lazy_context(…) and […] = lazy.arguments(). C++14 introduced decltype(auto) to delay the return type deduction after the dust of template instantiation has settled [27].

In high-level operations like dualization, inversion, and versor application, the implementation placed between the creation of the lazy context and the evaluation of the result is similar to what is expected from the user of the library. That is, by calling products and grade-dependent sign-change operations implemented by GATL as subroutines. Core operations, on the other hand, are implemented in a special way. They apply generative template metaprogramming to process the Expression parameters of the input arguments and produce the type set to the Expression parameter of the non-concrete multivector result_. It is at this moment that GATL applies algebraic simplification. For example, geometric product, outer product, regressive product, and inner products implement a set of recursive templates to apply distributivity over addition, and partial template specialization to enable conditional branching to suppress arithmetic operations that equal compile-time defined constant values. More specifically, when the compiler instantiates a template that implements a core operation, GATL's lexicographic expression ordering convention tries to put the subexpressions defined on the same get_value<…> (or get_bitset<…>) types close to each other. Whenever predefined patterns are identified, they are replaced by simpler equivalent expressions.

Calling the lazy.eval(result_) function causes the instantiation of a new clifford_expression<…> object. The Expression parameter of the new object will be derived from the Expression parameter describing result_. The coefficients and bitsets stored by the new object (if any) are computed by the lazy context by traversing the result_'s Expression parameter and solving

subexpressions defined on the occurrences of get_value<...>, get_map_values<...>, get_bitset<...>, and get_map_bitsets<...> related to instances of the clifford_expression<...> objects given as input. The current lazy context instance does not evaluate subexpressions defined on coefficients and bitsets with Tag values coming from other lazy contexts, nor subexpressions completely defined on constant values. It is because it is clear that the evaluation of these subexpressions must be deferred to another part of the program.

4 Experimental Results

We have used the GA Benchmark Project [14] to assess the execution time of GATL in comparison to other C++ solutions for GA. The GA Benchmark Project started from informal conversations among developers who attended to AGACSE 2018, in Campinas, Brazil. The idea was to build a suitable environment to compare GA solutions. It is an effort to define standards and methodologies for benchmarking GA code optimizers, libraries, and library generators. The goal of the project is to help physicists, chemists, engineers, and computer scientists to choose the GA solution that best suits their practical needs, as well as to push further the improvement of the compared solutions and to motivate the development of new tools. GA Benchmark is built on the Google Benchmark, a open source library to benchmark code snippets.

The GA Benchmark version 2.0.3 includes the evaluation of twelve binary operations (commutator product, geometric product, inverse geometric product, dot product, Hestenes' inner product, left contraction, right contraction, scalar product, outer product, regressive product, addition, and subtraction) and ten unary operations (dualization, undualization, versor inversion, normalization under reverse norm, squared reverse norm, unary minus, unary plus, Clifford conjugation, grade involution, and reversion) applied to k-blades having grades ranging from $k = 0$ to $k = n$ on eleven models of geometry (Euclidean models with basis vectors $\{e_1, \cdots, e_n\}$ and $n \in \{2, 3, 4, 5\}$, homogeneous models with basis vectors $\{e_1, \cdots, e_{n-1}, e_+\}$ and $n \in \{3, 4, 5\}$, Minkowski spaces with basis vectors $\{e_1, \cdots, e_{n-2}, e_+, e_-\}$ and $n \in \{4, 5\}$, and conformal models assuming $\{e_1, \cdots, e_{n-2}, n_o, n_\infty\}$ and $n \in \{4, 5\}$). GA Benchmark also includes the evaluation of an inverse kinematics algorithm assuming the conformal model of 3-dimensional Euclidean space, where $n = 5$ and the set of basis vectors depends on the solution.

For each possible configuration of input grades in unary and binary operation, GA Benchmark generates random blades (or pairs of random blades) and measures the mean execution times of 30 evaluations of the operation. For the inverse kinematics algorithm, the inputs are random sets having five angular values each. The time required to build input data is not considered when calculating the execution times.

The benchmark is ready to compare seven C++ solutions for GA. Namely, Gaalet [25], Gaalop [5], Garamon [2], GATL, GluCat [19], TbGAL [26], and Versor [6]. For GluCat, the benchmark includes comparison considering framed-based and matrix-based multivectors. We performed the comparison in a notebook running

Table 2 Ranking of performance of the compared solutions on binary and unary operations according to the gold first method

Solutions	Medals								Compilation errors
	1	2	3	4	5	6	7	8	
`GATL`	247	75	1	0	0	0	0	0	0
`Versor`	246	76	1	0	0	0	0	0	0
`Gaalop`	246	18	0	0	0	0	0	0	59
`Gaalet`	238	26	0	0	0	0	0	0	59
`TbGAL`	34	56	97	107	26	3	0	0	0
`Garamon`	26	255	42	0	0	0	0	0	0
`GluCat` (framed)	6	60	151	72	22	12	0	0	0
`GluCat` (matrix)	0	3	45	108	151	16	0	0	0

Ubuntu 18.04 operating system (Linux kernel version 4.4.0) on bare metal. The computer was equipped with 16 Gb of RAM and one Intel Core i7-8550U processor with 1.99 GHz and 8 cores. The C++ source codes were compiled using GCC 7.4.0 with `O3` optimization in release mode and single thread. The tables and charts that follow only show results considering conformal, Euclidean, and Minkowski geometries, since homogeneous and Euclidean models are equivalent. The complete set of log files produced for the experiments, as well as detailed charts for all operations and models of geometry, are available at the GitHub repository of the GA Benchmark project as the results reported on February 5th, 2020.

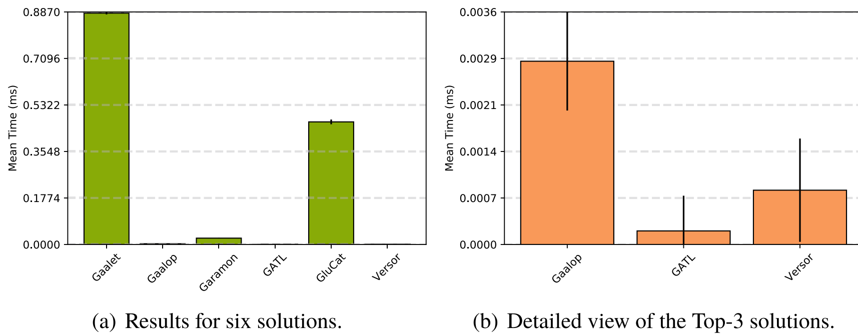
Table 2 classifies the compared solutions using the gold first method, *i.e.,* based first on the number of gold medals, then silver, and so on. A solution receives a gold medal (medal #1) whenever its performance is better than that of other solutions in a particular case of binary or unary operation with input arguments having specific grades. The second best-placed solution receives a silver medal (medal #2), and so on. The medals are distributed among the solutions for testing cases implemented as native subroutines and models of geometry by all of them. Therefore, only Euclidean and Minkowski models were considered here, because `Gaalet` and `GluCat` implement conformal geometry assuming $\{e_1, e_2, \cdots, e_+, e_-\}$ as basis vectors (like the Minkowski model) instead of $\{e_1, e_2, \cdots, n_o, n_\infty\}$. Regarding the set of available operations, unfortunately, the front-end of most of the solutions is incomplete. As can be seen in Table 3, the only operations implemented by all solutions are geometric product, outer product, and reversion. From the results in Table 2, it is possible to conclude that the code optimizer (*i.e.,* `Gaalop`) and libraries that explore template metaprogramming to perform compile-time specialization of subroutines (*i.e.,* `Gaalet`, `GATL`, and `Versor`) present equivalent performance when the mean processing time of common operations are considered. The last column of Table 2 shows the number of cases where the benchmark program could not be compiled due to errors raised by the solution.

Table 3 Binary and unary operations implemented as native subroutines by the compared solutions

	Binary Operations												**Unary Operations**										
	Main Products			Inner Products					Non-Metric Products		Sum Operations		Dualization Operations		Norm-Based Operations			Plus/Minus Operations		Sign-Change Operations		
Solutions	Commutator Product	Geometric Product	Inverse Geometric Product	Dot Product	Hestenes' Inner Product	Left Contraction	Right Contraction	Scalar Product	Outer Product	Regressive Product	Addition	Subtraction	Dualization	Undualization	Versor Inversion	Normalization	Squared Reverse Norm	Unary Minus	Unary Plus	Clifford Conjugation	Grade Involution	Reversion
`Gaalet`	-	✓	-	-	✓	-	-	✓	✓	-	✓	✓	✓	-	✓	-	-	-	-	-	-	✓
`Gaalop`	-	✓	✓	-	✓	-	-	-	✓	-	✓	✓	✓	-	-	-	-	✓	-	-	-	✓
`Garamon`	-	✓	✓	-	✓	✓	✓	✓	✓	-	✓	✓	✓	-	✓	-	✓	✓	-	-	-	✓
`GATL`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`GluCat`	-	✓	✓	-	✓	✓	-	-	✓	-	✓	✓	-	-	✓	-	✓	✓	-	✓	✓	✓
`TbGAL`	-	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`Versor`	✓	✓	✓	-	-	✓	-	-	✓	-	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓

The comparison of the mean execution times of complete algorithms aims to verify the ability of each solution to induce the optimization of sequences of operations and not only of individual subroutines. Figure 1 shows the mean execution times of the inverse kinematics algorithm. The GA Benchmark does not included `TbGAL` in this comparison because the original algorithm performs the addition of general multivectors and `TbGAL` only implements the addition and subtraction operations of scalar values, vectors, pseudovectors, and pseudoscalars. It is because `TbGAL` stores blades and versors as collections of scalar and vector factors instead of as the weighted sums of basis blades. In Fig. 1(a), we only include the result obtained for the frame-based version of `GluCat`, because the mean execution time of the matrix-based version of the library is four times higher for this algorithm. Since both frame and matrix-based versions belong to the same library, we used the one with better performance. In Fig. 1(b) we highlight the Top-3 solutions. The vertical lines on each of the bars on the graph indicate one standard deviation confidence interval.

Surprisingly, according to Fig. 1(a), `Gaalet` proved to be the least efficient solution. We believe that better results can be achieved if the user carefully inspects the outcome of each sequence of operations. In this way, he/she will be able to instruct



(a) Results for six solutions. (b) Detailed view of the Top-3 solutions.

Fig. 1 Mean execution times of the inverse kinematics algorithm implemented using six C++ solutions for GA (a), and the detailed view of the results of the Top-3 solutions (b)

the library to suppress unnecessary components in the intermediate results. But such a task can be difficult and laborious.

When comparing `Garamon` and `GluCat`, we conclude that `Garamon` achieved better results because it stores the multivector components as per grade arrays, while `GluCat` uses dictionaries to store individual components. The high cost of both solutions is related to the time needed to manage the dynamic memory used by the algorithm's intermediate variables.

The detailed view of the results obtained for `Gaalop`, `GATL`, and `Versor` is presented in Fig. 1(b). The proposed library achieved better results because the lazy evaluation and compile-time simplification mechanisms were able to eliminate unnecessary multivector components and arithmetic operations. The same degree of optimization is not achieved by the eager evaluation strategy employed by `Versor`. The optimized codes generated by `Gaalop` show that the lack of performance of the solution in this algorithm is related to the substitution of expressions such as x^2, x^3, etc., by calls to the `std::pow(base, exponent)` function. `GATL`, on the other hand, uses one or two multiplications to raise x to the powers 2, 3, and 4. The `std::pow(...)` function requires many more processing cycles than multiplication.

5 Concluding Remarks

This contribution presents `GATL`, a C++ library that applies the lazy evaluation strategy and template metaprogramming to conduct symbolic optimizations on expressions at compile-time to improve the runtime performance of GA-based programs.

In addition to runtime performance, `GATL` is concerned with being user friendly and intuitive. In other words, we expect the implementation of equations written with GA to be as straightforward as possible without compromising the proper use of available computing resources. To this end, `GATL` offers dozens of GA operations ready for use and completely integrated with the lazy evaluation concept.

Currently, GATL does not include modules for data visualization. However, in our experience, the results produced with GATL can be easily integrated with other visualization solutions, such as ganja.js [7].

GATL supports Clifford Algebras assuming metric matrices with arbitrary (p, q, r) signatures and arbitrary sets of basis vectors. The practical use of GATL's current version is limited to spaces with up to $n = p + q + r = 7$ dimensions, except in special situations where the operated multivectors are made up of a small amount of components. This limitation is also observed in other templates-based GA libraries such as Versor and Gaalet. It is related to the ability of the compiler to analyze and instantiate the templates. Fortunately, each new version of the C ++ language includes features that allows us to replace complex templates by simpler counterparts. Consequently, the expectation is that in the future it will be possible to work with algebras in higher dimensions and produce programs that compile in less time.

The development of computational tools for GA is a living ecosystem. We hope that the ideas presented in this contribution will help developers to improve existing solutions and serve as inspiration for the development of innovative tools.

Acknowledgements This work was sponsored by CNPq-Brazil (grant 311.037/2017-8) and FAPERJ (grant E-26/202.718/2018).

References

1. Arsenovic, A., Hadfield, H., Antonello, J., Kern, R., Boyle, M.: Numerical geometric algebra module for Python (2018). https://github.com/pygae/clifford
2. Breuils, S., Nozick, V., Fuchs, L.: Garamon: a geometric algebra library generator. Adv. Appl. Clifford Algebras **29**(4), 69 (2019)
3. Bromborsky, A.: Symbolic geometric algebra/calculus package for SymPy (2015). https://github.com/brombo/galgebra
4. Castelani, E.V.: Library for geometric algebra (2017). https://github.com/evcastelani/Liga.jl
5. Charrier, P., Klimek, M., Steinmetz, C., Hildenbrand, D.: Geometric algebra enhanced precompiler for C++, OpenCL and Mathematica's OpenCLLink. Adv. Appl. Clifford Algebras **24**(2), 613–630 (2014)
6. Colapinto, P.: Versor: spatial computing with conformal geometric algebra. Master's thesis, University of California at Santa Barbara (2011)
7. De Keninck, S.: Javascript geometric algebra generator for Javascript, C++, C#, Rust, Python
8. Dijkman, D.H.F.: Efficient implementation of geometric algebra. Ph.D. thesis, Universiteit van Amsterdam (2007)
9. Dorst, L., Fontijine, D., Mann, S.: Geometric Algebra for Computer Science: An Object Oriented Approach to Geometry. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann Publishers, Amsterdam (2007)
10. Eid, A.H.: An extended implementation framework for geometric algebra operations on systems of coordinate frames of arbitrary signature. Adv. Appl. Clifford Algebras **28**(1), 16 (2018)
11. Fernandes, L.A.F.: GATL: geometric algebra template library (2020). https://github.com/laffernandes/gatl
12. Fontijne, D.: Implementation of Clifford algebra for blades and versors in $O(n^3)$ time. In: Talk at International Conference on Clifford Algebra (2005)

13. Fontijne, D.: Gaigen 2: a geometric algebra implementation generator. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, pp. 141–150 (2006)
14. GA Developers: GA-Benchmark: a benchmark for geometric algebra libraries, library generators, and code optimizers (2020). https://github.com/ga-developers/ga-benchmark
15. Hadfield, H., D., H., A., A.: Gajit: symbolic optimisation and JIT compilation of geometric algebra in Python with GAALOP and Numba. In: Gavrilova, M., Chang, J., Thalmann, N., Hitzer, E., Ishikawa, H. (eds.) Advances in Computer Graphics – Computer Graphics International Conference (CGI). Springer (2019)
16. Hestenes, D.: New Foundations for Classical Mechanics. Reidel Publishing Company (1987)
17. Hildenbrand, D., Pitt, J., Koch, A.: Gaalop - high performance parallel computing based on conformal geometric algebra. In: Bayro-Corrochano, E., Scheuermann, G. (eds.) Geometric Algebra Computing, pp. 477–494. Springer, London (2010)
18. Kanatani, K.: Understanding Geometric Algebra: Hamilton, Grassmann, and Clifford for Computer Vision and Graphics. CRC Press (2015)
19. Leopardi, P.C.: GluCat: generic library of universal Clifford algebra templates (2007). http://glucat.sourceforge.net/
20. Ong, J.: $p(r*_{3,0,1})$ specialized SIMD geometric algebra library (2020). https://github.com/jeremyong/klein
21. Perwass, C.: Geometric Algebra with Applications in Engineering. Springer Publishing Company (2009)
22. Perwass, C., Gebken, C., Grest, D.: CluViz: interactive visualization (2004). http://cluviz.de
23. Pythonic Geometric Algebra Enthusiasts: Symbolic geometric algebra/calculus package for SymPy (2017). https://github.com/pygae/galgebra
24. Reed, M.: <Leibniz-Grassmann-Clifford-Hestenes> differential geometric algebra multivector simplicial-complex (2017). https://github.com/chakravala/Grassmann.jl
25. Seybold, F.: Gaalet: geometric algebra algorithms expression templates (2010). https://sourceforge.net/projects/gaalet/
26. Sousa, E.V., Fernandes, L.A.F.: TbGAL: a tensor-based library for geometric algebra. Adv. Appl. Clifford Algebras **30**(2), 27 (2020)
27. Standard C++ Foundation: ISO International Standard ISO/IEC 14882:2017(E) – Programming Language C++. IOS (2017). https://isocpp.org/std/the-standard