Gaigen 2.5 User Manual

Daniel Fontijne
University of Amsterdam

August 23, 2010

1 Introduction

Gaigen 2.5 is a code generator for geometric algebra. It compiles an XML specification of an algebra into an implementation. Supported output languages are C and C++, C# and Java. C# and Java support is functional but incomplete: some shortcuts and operator overloading should still be added (this should be finished in August 2010). Matlab support is under development. The tool itself is written in C#.

2 History, Background, Licensing

Gaigen stands for *Geometric Algebra Implementation GENerator*. The first version of Gaigen was written in 2001. It supported only C++ as output language and the performance of the generated code was two to five times slower than the equivalent use of linear algebra. At that time it was the fastest general purpose GA implementation available. Gaigen 1 supported only the general multivector type and used coordinate compression and profiling to increase performance. The tool itself was written in C++.

The second version of Gaigen was released in 2006. It added support for specialized multivector types and supported C++ and Java. The tool itself was written in Java. The generated code was competitive with linear algebra (faster for some problems, slower for others). Again Gaigen 2 was the fastest general purpose GA implementation available, although a C++ template library named Clifford was equally fast (this library is no longer available, it seems).

This version (2.5) is a re-implementation of Gaigen 2. It removes a lot of 'dead weight' and primarily aims at making Gaigen suitable for a production environment, with an emphasis on testing, programming language support, scalability and extendibility. Some functions may be slightly faster in Gaigen 2.5 than Gaigen 2 due to minor optimizations.

The Gaigen 2.5 tool (`g25`) is covered by a GPL license. The generated (compiled) code is your own and can be covered by any license.

See [1] and [2] for background information on (implementation of) geometric algebra.

3 Installing

An installer is provided on SourceForge for each platform. On Linux and OSX, Mono is required because Gaigen 2.5 is written in C#. Mono is a free, open source implementation of the Microsoft .Net platform.

- **Windows:** Run `InstallG25.msi`. By default, the files are installed in `c:/program files/g25`. The system path is updated to include this directory.
- **Linux:** Make sure Mono and ANTLR are installed, then install the RPM using `rpm -i g25-2.5.X.rpm`. Use `rpm -U g25-2.5.X.rpm` to update. By default four small scripts are installed in `/usr/bin`, the actual program is installed in `/usr/share/g25`.
- **Mac OS X:** Install Mono, then run `g25.pkg`. Four small scripts are installed in `/usr/bin`, the actual program is installed in `/usr/share/g25`.

ANTLR is required to compile ANTLR grammars for parsing multivector strings. To avoid its use, simply ask for the built-in parser (instead of ANTLR) in the algebra specification.

4 Sample Algebra Specificatopms

The `g25_test_generator` can be used to generate a few sample algebra specifications that can be used as a starting point for your own specs. Use the following command to generate them: `g25_test_generator -sa .`

A directory `TestG25` will be created. In that directory will be a number algebra specifications. Build-scripts, test-scripts and Makefiles are also provided (these scripts are platform dependent).

The sample algebras are:

- **e2ga**. 2-D Euclidean geometric algebra.
- **e3ga**. 3-D Euclidean geometric algebra.
- **p3ga**. Geometric algebra for the homogeneous model of 3-D space (so it is a 4-D algebra). Multivector types for points, planes, lines and so on are defined.
- **c3ga** Geometric algebra for the conformal model of 3-D space (so it is a 5-D algebra). Multivector types for points, spheres, circles, lines and so on are defined.

5 Running

To compile an algebra specification, run `g25 specfile.xml` from the command line.

The first time you run `g25` for a particular language it may be a bit slow because it is compiling a lot of code templates on the fly. Compiled templates are stored in a temporary directory and recycled in future runs. On Linux and

OSX, the first run can take a even more time because Mono is a bit slower than Microsofts CLR implementation. On OSX this problem is worse, because OSX seems to delete the file in the temp directory on reboot (to do: check if this is true, is there a solution?).

The following command line options are available:

`-h -help -?`: display help.
Example: `g25 -h`

`-v, -version`: display version information.
Example: `g25 -v`

`-s, -save`: read specification file, then save it.
Example: `g25 -s spec_out.xml spec_in.xml`

`-f, -filelist`: save a list of generated files to a file.
Example: `g25 -f filelist.txt spec.xml`

`-d, -deterministic`: whether code generation should be deterministic (`true` or `false`). Names of test functions and dependencies have a number suffix to avoid name clashes. These numbers are not assigned in a deterministic manner when multiple threads are running concurrently. The `xml_test` script uses the `-d true` option to make sure that the names of functions are consistent between runs. This option is also useful for debugging.
Examples:

```
g25 -d true spec.xml
g25 -d false spec.xml
```

6 Generated Files

When Gaigen 2.5 compiles an algebra specification. a number of files are generated. In this paragraph, the `namespace` of the algebra is `A` and the name of the general multivector type is `MV`.

6.1 Generated Files for the C Programming Language

By default, the following files and generated:

- `A.h` : main header of algebra implementation.
- `A.c` : main definitions of algebra functions.

If testing is enabled:

- `A_test_main.c` : test suite (`main()` function included).

If the Mersenne Twister random generator is enabled:

- `A_mt.c` : Mersenne Twister random generator source.
- `A_mt.h` : Mersenne Twister random generator header.

If a parser is enabled:

- `A_parse_MV.c` : main header of algebra implementation.

If the ANTLR parser is enabled:

- `A.g` : ANTLR grammar.

6.2 Generated Files for the C++ Programming Language

By default, the following files and generated:

- `A.h` : main header, inlined functions.
- `A.cpp` : non-inlined definitions of algebra functions.

If testing is enabled:

- `A_test_main.cpp` : test suite (`main()` function included).

If the Mersenne Twister random generator is enabled:

- `A_mt.cpp` : Mersenne Twister random generator source.
- `A_mt.h` : Mersenne Twister random generator header.

If a parser is enabled:

- `A_parse_MV.cpp` : main header of algebra implementation.

If the ANTLR parser is enabled:

- `A.g` : ANTLR grammar. Note: after compilation using ANTLR, rename `A_parser.c` to `A_parser.cpp`, and `A_lexer.c` to `A_lexer.cpp`.

6.3 Generated Files for the C# Programming Language

To do.

6.4 Generated Files for the Java Programming Language

To do.

7 Documentation of Generated Code

The generated code is self-documenting, although this feature is not fully finished yet (the overview page is not implemented yet and not all functions are documented). Run `doxygen` to extract the documentation from the code.

The generated test code is also a good way to get an example of use of each function.

8 Testing the Generated Code

Gaigen 2.5 can generate a test suite with each algebra. This is controlled via option `testSuite` in the specification file.

The sample algebras (Section 4) generate the test code by default. Building them will result in executables named `test` which test whether the generated code is working correctly.

8.1 Testing of Gaigen 2.5

A special tool called `g25_test_generator` is included with Gaigen 2.5. Its purpose is to generate variations on algebra specifications for thorough testing of the code generator. It is used for regression testing during development.

If you are interested in running it, run `g25_test_generator -r 5000 -s` on the command line. This should generate a directory named `TestG25`. Inside that directory, run the `build` script to see if all the generated algebras actually build. Then run the `test` script to see if all algebras work correctly. Run the `xml_test` script to see if loading and saving specification XML files works correctly. Run the `clean` script to get rid of all intermediate build and test files.

The `-r` option reduces the number of algebras that are generated. Without it, many thousands of algebras would be generated, which would take a long time to build. Building takes a long time anyway since the test algebras include an unrealistic large number of functions.

The command line options to `g25_test_generator` are:

`-r -reduce` : reduce the number of test algebras by approximately that factor. Example: `g25_test_generator -r 5000`

`-s -shuffle`: shuffle the order in which the test algebras are built, tested. Example: `g25_test_generator -s`

`-sa -sample_algebras`: generate the sample algebras instead of the randomly selected test algebras. Example: `g25_test_generator -sa`

9 Building from source

To build Gaigen 2.5 from source, first download the source code as a tarball `g25-2.5.X.tar.gz` or from SourceForge SVN.

Using Visual Studio or MonoDevelop, open the main project `g25.sln` in `g25/vs2008`. (When using MonoDevelop you will get a warning that the Windows installer project cannot be loaded; please ignore). Do `Build->Build Solution` to build all.

If you want to build from the command line on Windows, open a Visual Studio Command Prompt and go to the directory `g25/vs2008`. Do `msbuild g25.sln /p:Configuration=Release`.

If you want to build from the command line on Linux or OSX, go to the directory `g25/vs2008` and do

```
export MONO_IOMAP=all  # makes mono tools case/slash insensitive
cd g25/vs2008
xbuild g25.sln /p:Configuration=Release
cd ../../g25_diff/vs2008
xbuild g25_diff.csproj /p:Configuration=Release
cd ../../g25_test_generator/vs2008
xbuild g25_test_generator.csproj /p:Configuration=Release
cd ../../g25_copy_resource/vs2008
xbuild g25_copy_resource.csproj /p:Configuration=Release
```

Installers / packagers for each platform are in `g25/setup_win`, `g25/setup_osx` and `g25/setup_linux`.

10 Writing Algebra Specifications

The format of the specification file is described in the next section. A good starting point for writing you own specifications are the sample algebras, see Section 4 for how to obtain them.

If you'd like to write a totally new specification, typically you'd take the top part of one of the sample algebras (up to where the specialized multivectors (`smv`) are defined). Then you'd change the dimension of the space to what you need, edit the names of the basis vectors, and set the metric. Then adjust all the other settings to your needs.

If you need them, add your own specialized multivectors (defining types like points, vectors, bivectors).

Finally you need to ask for the functions you want to have implemented using `function` entries. The easiest way is to ask for these functions to be implemented for general multivectors. Specialized multivectors automatically convert to general multivectors when passed as arguments to such functions, so if you do not care about speed they are all you need.

If you want to optimize your code, add specialized functions as needed. You can also use the `reportUsage` feature to get a report on what functions could be optimized.

11 Algebra Specification File XML Format

An XML specification file starts with the opening element `g25spec`.

This element can have the following attributes:

- **license**. The license of the generated code. The value can be `custom`, `gpl` or `bsd`. The license is case insensitive. If the license is custom, a `customLicense` element is expected later on in the specification.
- **language**. The value can be `c`, `cpp`, `java`, `csharp`, `python`, `matlab` currently. (the fact that a value is valid does not means that it is actually implemented . . .). The language names are case insensitive.
- **namespace**. The name and the namespace/package of the generated code (always required, because it is also used as a prefix/part of generated filenames).
- **coordStorage**. The value can be `array` (coordinates are stored in arrays) or `variables` (one variable for each coordinate). Determines whether coordinates are stored in arrays or in single variables. This only applies to specialized multivectors.
- **defaultOperatorBindings**. The value can be `true` or `false`. If `true`, the default operator bindings for the output language are used (for example, the + symbol is bound to the `add` function).

- **dimension**. The dimension of the space of the algebra. Must be $>= 1$. For values above 7, consider using `gmvCode="runtime"`, see below. Values above 12 probably lead to code that cannot be compiled because it is too large.

- **testSuite**. Whether to generate extra code to test the generated code. Can be `true` or `false`.

- **reportUsage**. The value can be `true` or `false`. When `true`, print statements are added to the code to report usage of non-optimized functions (i.e., functions involving specialized multivectors which were implicitly converted to general multivectors). Also, a member variable is added to the general multivector type which keeps track of the original specialized type of the multivector. This option has no effect in the `C` language because it does not support implicit conversion.

- **gmvCode**. Possible values are `expand` and `runtime`. The code for general multivectors can be very large. For example a geometric product of two GMVs in 10-D takes in the order of 1024*1024 multiplications and additions. If the code for this product is written explicitly into the code (the default option), the code size would also be in the order of megabytes.

  Because of this, Gaigen 1 and Gaigen 2 could only generate algebras up to about 7-D. To overcome this limitation, Gaigen 2.5 supports 'run-time' computation of geometric products and all other functions without explicitly generating code for every single multiply/add. The default option `expand` writes out all code, is fast, but cannot realistically be used above 7-D.

  The option `runtime` performs the computations at run-time, using (among others) tables which must be initialized at startup. If the option `runtime` is used, the metric must be diagonal. To compute the tables of a non-diagonal metric, symmetric eigenvalue computation is required, and it would be a burden to require eigenvalue code for every output language. Note that the run-time code is approximately two times slower than the expanded code.

- **parser**. What type of multivector string parser to generate. The default is `none`. Other options are `builtin` (for a parser hand-written for Gaigen 2.5) and `antlr` for an ANTLR based parser. Both these parsers have the same functionality and interface, but their internal implementation is different. For the ANTLR parser, you need to invoke `java org.antlr.Tool` on the generated `.g` grammar and link with the ANTLR run-time.

- **copyright**. The copyright notice of the generated code.

Inside the main `g25spec` element, the following elements can be present:

- **customLicense**. The custom license text. This element must be present when `license="custom"`. The text is copied verbatim to the top of each the generated file.

- **outputDirectory**. Where the generated files should go. The `path` attribute sets the directory where the output should go. By default, the output goes to the current working directory.

- **outputFilename**. Allows the name of individual generated files to be modified. For example, if the code generator would generate a file named `foo.cpp`, but the user wants this file to be named `bar.cpp`, then setting attributes `defaultName="foo.cpp"` and `customName="bar.cpp"` allows the filename to be overridden. Attributes:

  – **defaultName** (required). Default name of the file; do not include the full path.

  – **customName** (required). Custom name for file; do not include the full path.

- **inline**. What types of functions to inline. Possible attributes are `constructors`, `set`, `assign`, `operators` and `functions`. The value of the attributes can be `true` or `false`.

- **floatType**. Specifies the float type used for storing coordinates. Multiple float types can be defined in the same algebra. This element can have the following attributes:

  – **type**. (required). The value should be a floating point type (e.g. `float` or `double`).

  – **suffix**. (optional). The suffix applied to multivector/outermorphism classes when instantiated with this floating point type. For example if there is a specialized multivector type called `vectorE3GA` and the suffix for the float type `float` is `_f` then `vectorE3GA` instantiated with `float` will be called `vectorE3GA_f`.

  – **prefix**. (optional). The prefix applied to multivector/outermorphism classes when instantiated with this floating point type.

- **basisVectorNames**. This element lists the names of basis vectors of the algebra. The number of basis vectors must match the dimension `N` of the space. The attributes of the element are `name1`, `name2`, ..., `nameN`. Each attribute is assigned the name of its respective basis vector, for example `name1="e1" name2="e2"`.

- **metric**. A `metric` element specifies the inner product between one or more pairs of basis vectors. By default, all inner product between basis vectors are assumed to be 0. By using `metric` elements, one can set the inner product to different values. Inside a single algebra, different metrics can be used, e.g. a conformal one and a Euclidean one. Having a Euclidean metric is useful, e.g., for blade factorization algorithms.

  An example of a metric element is

  ```
  <metric name="default">no.ni=-1</metric>
  ```

  This line says that the inner product between basis vectors `no` and `ni` is `-1`. The attribute `name="default"` says that this line belongs to the `default` metric and may be left out (because the default value for this attribute *is* `"default"`).

One may also specify multiple metrics at once, as in

```
<metric>e1.e1=e2.e2=e3.e3=1</metric>
```

Inside `function` elements, a non-default metric name may be specified by using the `metric="name"` attribute, e.g., `metric="conformal"`.

Due to floating point round-off errors in eigenvalue computation, values or coordinates that should be (e.g.)1.0 may become (e.g., $1 + 1e^{-16}$). This makes the generated code less efficient, is annoying to read and propagates the round-off errors.

For that reason, there is the option to round coordinates after a metric product. The default is to round, but when the final metric is diagonal, it is forced to no rounding because there is not need to use it in that case. The user can explicitly force the rounding using the `round="false"` or `round="true"` attribute, but when the metric is diagonal, it will still be forced to no rounding. When rounding is enabled, coordinates which are very close to an integer value are rounded to that value. The threshold for being 'very close' is $1e^{-14}$.

- **unaryOperator**. This element allows you to bind a unary operator symbol to a one-argument function (in languages which support this feature). The attributes of this element are:
  - `symbol`. The operator symbol, for example `++`.
  - `prefix`. Only for operators `++` and `--`: Whether this operator is prefix (e.g. `++a`) or postifx (`a++`). Use `true` for prefix, and `false` for postfix.
  - `function`. The name of the function to bind to, for example `increment`.
- **binaryOperator**. This element allows you to bind an binary operator symbol to a two-argument function (in languages which support this feature). The attributes of this element are:
  - `symbol` The operator symbol, for example $\wedge$.
  - `function` The name of the function to bind to, for example `op`.
- **mv**. This element specifies the properties of the general multivector. It is one of the most involved elements. Some examples are given below in Section 11.1. Its attributes are:
  - `name`. The name of the general multivector type, for example `mv`.
  - `compress`. How to compress the multivector coordinates: `byGrade` or `byGroup`.
  - `coordinateOrder`. The order of coordinates: `default` or `custom`.
  - `memAlloc`. How to allocate memory for coordinates: `full`,`parityPure` or `dynamic`.

- **smv**. An smv specifies a specialized multivector type. The smv element should contain the basis blades of the type. These may have constant assignments, and if the type is constant const="true", then all basis blades must have a constant assignment. An example of a specialized multivector definition is

```
<smv name="normalizedPoint" type="blade">no=1 e1 e2 e3</smv>.
```

  The attributes of a smv element are:

  – name. The name of the specialized multivector type, for example vector.

  – const (optional). Can either be true or false. When true, the type is a constant type with no variable coordinates. In that case, all basis blades must have a constant value assigned to it. If the const attribute is not specified it is assumed to be false. A constant with the name will be generated and the actual name of the type will have an _t suffix.

  – type. The type attribute specified whether instances of the specifialized multivector class will contain only blades (type="blade"), rotors (type="rotor"), versors (type="versor") or any type of multivector (type="multivector") value. This may be used for optimizations and for sanity checks by the code generator.

- **constant**. This element is used to generate a constant value in the output. This is useful is you want a constant value of non-constant type. The constant has a name, a type, and a value. Some examples of a constant are:

```
<constant name="vectorE1" type="vectorE3GA">e1=1</constant>
<constant name="pointAtOrigin" type="normalizedPoint"></constant>
```

  Coordinates which are zero do not need to be specified. The attributes of a constant element are:

  – name. The name of the constant.

  – type. The type of the constant. Currently only specialized multivector constants are supported (smv).

  The constant element contains the values of the coordinates of the constant, and optionally a comment element.

- **om**. Specifies the general outermorphism matrix representation type. This allows for efficient application of linear transformations using the applyOM function.

  The outermorphism has a domain and a range, both of which may be specified, but they can also be left to the defaults. An example of an outermorphism with default coordinate order is:

```
<om name="om" coordinateOrder="default" />
```

A 3-D example of an outermorphism with a custom domain and range is:

```
<om name="om" coordinateOrder="custom">
<domain>scalar e1 e2 e3 e1^e2 e2^e3 e3^e1 e1^e2^e3</domain>
<range>scalar e1 e2 e3 e1^e2 e2^e3 e3^e1 e1^e2^e3</range>
```

In this example, it was redundant to specify the range since it is identical to the domain. Leaving the `range` element out would have the same effect. Note that all basis blades must be present in an general outermorphism's range and domain.

The attributes of a `om` element are:

  - `name`. The name of the outermorphism type, for example `om`.
  - `coordinateOrder`. This can be `default` or `custom`. If `custom` is used, the `domain` and possibly the `range` should be specified. If the `range` is left out, it is assumed to be identical to the `domain`.

- **som**. A `som` element specifies a specialized outermorphism. It is pretty much that same as a general outermorphism except it does not need to have all basis blades in its domain and range. An example of a `som` element is:

```
<som name="flatPointOM">
<domain>e1^ni e2^ni e3^ni no^ni</domain>
<range>e1^ni e2^ni e3^ni no^ni</range>
</som>
```

  The `som` element has only one attribute, since the `coordinateOrder` is always custom:

  - `name`. The name of the outermorphism type, for example `om`.

- **function**. This element specifies a request to the code generator back-end to implement a specific function for specific arguments. See Section 12 for the supported GA functions and Section 13 for information on converters.

  The attributes are:

  - `name`. The name of the function, as it is known to the code generator (see Section 12 for a list of function names). This name is also the name of the generated function unless an `outputName` attribute is specified.

    To generate a converter ('underscore constructor'), the name of the function should be an underscore plus the name of the destination type, e.g., `_vectorE3GA`. This first (and only) argument should be the source type.
  - `outputName`. Optional. Changes the name of the generated function to the value of the attribute. For example, allows you to rename a function `gp` to `geometricProduct`. Sometimes this attribute is required to avoid name-clashes, for example if you want the define the same function for two different metrics.

- `returnType`. Optional. By default, the code generator will determine the return type of the functions it generates, but it is possible to override this default by setting it explicitly.

  The return type should be the name of a specialized multivector. However, the return type may also be `scalar` or any of the floating point typenames used in the algebra. If the return type is `scalar`, then a float will be returned, automatically adapted to the floating point type of the function.

- `argN`. Specifies the type of argument `N`. If no `argN` attribute is given, the code generator will fill in the default (general) types automatically. Otherwise, the correct number of `argN` attributes should be specified for the function (running from 1 up to the number of arguments of the function).

  Not all combinations of argument types are possible. For example, currently it is not possible to mix general and specialized multivectors. It *is* possible to mix floats and general multivectors though.

- `argNameN`. Specifies the name of argument `N`. This only affects the name of the argument inside the generated function. Specifying this name may be superfluous, but it can improve readability, especially for code completion.

- `optionX`. Specifies an option `X`. For example, the `exp` functions can generate more efficient code when it knows what the sign of the square of the argument is. In that case, one may use for example `optionSquare="1.0"`.

- `floatType`. Multiple `floatType` attributes may be present in a single `function` element. By default, the code generator will generate code for all floating point types of the specification, but using the `floatType` attribute(s) this may be limited to only the set of listed floating point types.

- `metric`. The optional `metric` attribute specifies the usage of a non-default metric (case insensitive). By default, the metric `"default"` is used. By using this attribute a different metric may be used for the function, e.g., `metric="euclidean"`.

- `comment`. Use the this optional attribute to add any extra user comments to the function documentation. For example, one could use the comment to explain what a certain function is used for. These comments will appear in the documentation generated by Doxygen.

• **verbatim**. This element is used to add verbatim code to the output files. It can be useful,for example to include some headers or packages, or to add some custom functions or documentation. The `verbatim` element can contain the code as text, or can point to a file using the `codeFilename` attribute. The attributes are:

  - `filenameX`. The filename(s) of the files to modify. Multiple files can be modifed with one `verbatim` element. The `X` in the attribute can be any string (including empty). If multiple `filenameX` attributes are specified, multiple files are modified.

- `position`. Where to place the verbatim code. The values can be `top` (at the top of the file), `bottom` (at the bottom of the file), `before` (before some marker string) or `after` (before some marker string).
- `marker`. If `position` is `before` or `after`, then this attribute specifies the string before or after which the verbatim code should be inserted.
- `codeFilename`. The verbatim code can be directly inside the `verbatim` element but for long code it may be easier to put the code in a separate file. The name of this file is specified using this attribute.

11.1 Multivector Compression, Coordinate Order and Memory Allocation

Memory of general multivector variables can be allocated in different ways, each with its own advantages and disadvantages.

Memory can be allocated for all possible coordinates (`memAlloc="full"`). For example, this would allocate 32 coordinates for a 5-D algebra.

Memory can also be allocated for halve the coordinates (`memAlloc="parityPure"`), it it is known that multivector values will always be parity-pure (only even-grade, or only odd-grade).

Another option is dynamically allocate just the memory that is required (`memAlloc="dynamic"`).

Compression of multivector coordinates can be done per grade part (`compress="byGrade"`) or per user-defined group (`compress="byGroup"`).

If compression is done by grade, then the attribute value `coordinateOrder="default"` can be used. In that case the coordinate or does not have to be specified.

But it is also allowed to have a custom coordinate order. In that case the `mv` element must contain a list of basis blades all, i.e., the order of coordinates. The basis blades should not be in `group` elements, just listed in the order you want them to be. The basis blades should be listed in ascending grade order. For example:

```
<mv compress="byGrade" coordinateOrder="custom" memAlloc="parityPure">
   scalar
   no e1 e2 e3 ni
   no^e1 no^e2 no^e3 e1^e2 e2^e3 e3^e1 e1^ni e2^ni e3^ni no^ni
   e2^e3^ni e3^e1^ni e1^e2^ni no^e3^ni no^e1^ni no^e2^ni no^e2^e3 no^e1^e3 no^e
   e1^e2^e3^ni no^e2^e3^ni no^e1^e3^ni no^e1^e2^ni no^e1^e2^e3
   no^e1^e2^e3^ni
</mv>
```

If compression is done by group, each group of basis blades must be specified inside the `mv` element inside a `group` element. A group cannot contain basis blades of different grades. This example splits the coordinates of the 5-D conformal algebra into three basic groups (`no`, `ni`, and `e1, e2, e3`) for all grades. For example:

```
<mv compress="byGroup" coordinateOrder="custom" memAlloc="parityPure">
    <group>scalar</group>
    <group>no</group>
```

```
    <group>e1 e2 e3</group>
    <group>ni</group>
    <group>no^e1 no^e2 no^e3</group>
    <group>e1^e2 e2^e3 e3^e1</group>
    <group>e1^ni e2^ni e3^ni</group>
    <group>no^ni</group>
    <group>e2^e3^ni e3^e1^ni e1^e2^ni</group>
    <group>no^e3^ni no^e1^ni no^e2^ni</group>
    <group>no^e2^e3 no^e1^e3 no^e1^e2</group>
    <group>e1^e2^e3</group>
    <group>e1^e2^e3^ni</group>
    <group>no^e2^e3^ni no^e1^e3^ni no^e1^e2^ni</group>
    <group>no^e1^e2^e3</group>
    <group>no^e1^e2^e3^ni</group>
</mv>
```

12 Supported Functions

The section lists the functions that Gaigen 2.5 can generate out of the box (plugins may add more functions). Each entry lists the name of the function, gives a short description and some examples.

add

Adds two multivectors.

```
<function name="add" arg1="mv" arg2="mv" />
<function name="add" arg1="vectorE3GA" arg2="vectorE3GA" />
<function name="add" arg1="scalar" arg2="bivector" returnType="rotor" />
```

subtract

Subtracts two multivectors.

```
<function name="subtract" arg1="mv" arg2="mv" />
<function name="subtract" arg1="vectorE3GA" arg2="vectorE3GA" />
```

applyOM

Applies an outermorphism to a multivector.

```
<function name="applyOM" arg1="om" arg2="mv"/>
<function name="applyOM" arg1="om" arg2="normalizedPoint"/>
<function name="applyOM" arg1="grade1OM" arg2="vectorE3GA"/>
```

applyVersor

Applies a versor $\mathbf{V}$ to a multivector $\mathbf{X}$.
For even versors, returns $\mathbf{V}\,\mathbf{X}\,\widetilde{\mathbf{V}}/(\mathbf{V}\,\widetilde{\mathbf{V}})$.
For odd versors, returns $\mathbf{V}\,\hat{\mathbf{X}}\,\widetilde{\mathbf{V}}/(\mathbf{V}\,\widetilde{\mathbf{V}})$.

A custom metric can be used via the `metric="name"` attribute.

```
<function name="applyVersor" arg1="mv" arg2="mv"/>
<function name="applyUnitVersor" arg1="rotorE3GA" arg2="vectorE3GA"/>
<function name="applyVersor" arg1="rotorE3GA" arg2="normalizedPoint"
          metric="euclidean"/>
```

applyUnitVersor

Applies a unit versor $\mathbf{V}$ to a multivector $\mathbf{X}$ under the assumption that $\widetilde{\mathbf{V}} = \mathbf{V}^{-1}$. This identity does not hold generally in non-Euclidean metrics, so be careful.
For even versors, returns $\mathbf{V}\,\mathbf{X}\,\widetilde{\mathbf{V}}$.
For odd versors, returns $\mathbf{V}\,\hat{\mathbf{X}}\,\widetilde{\mathbf{V}}$.

A custom metric can be used via the `metric="name"` attribute.

```
<function name="applyUnitVersor" arg1="mv" arg2="mv"/>
<function name="applyUnitVersor" arg1="evenVersor" arg2="line"/>
```

applyVersorWI

Applies a versor $\mathbf{V}$ to a multivector $\mathbf{X}$ given the explicit $\mathbf{VI} = \mathbf{V}^{-1}$.
Note that the function have three arguments ($\mathbf{V}$, $\mathbf{X}$ and $\mathbf{VI}$).
For even versors, returns $\mathbf{V}\,\mathbf{X}\,\mathbf{VI}$.
For odd versors, returns $\mathbf{V}\,\hat{\mathbf{X}}\,\mathbf{VI}$.

A custom metric can be used via the `metric="name"` attribute.

```
<function name="applyVersorWI" arg1="mv" arg2="mv" arg3="mv"/>
<function name="applyVersorWI" arg1="rotorE3GA" arg2="bivectorE3GA"
          arg3="rotorE3GA"/>
```

cgaPoint

Returns a (normalized) conformal point. The position of the point can be specified as

- coordinates,
- avector,
- a 'flat point' (the outer product of a conformal point and infinity).

```
<function name="cgaPoint" arg1="vectorE3GA"/>
<function name="cgaPoint" arg1="double" arg2="double" arg3="double"
          floatType="double"/>
<function name="cgaPoint" arg1="float" arg2="float" arg3="float"
          optionOrigin="no"  optionInfinity="ni" floatType="float"/>
<function name="cgaPoint" arg1="flatPoint"/>
<function name="cgaPoint" arg1="normalizedFlatPoint"/>
```

randomCgaPoint

Returns a conformal point at a random position. The point lies inside a cube centered around the origin. The first and only argument to the function is 'radius' of the cube.

```
<function name="randomCgaPoint"/>
```

div

Divides a multivector by a scalar.

```
<function name="div" arg1="mv" arg2="double"
          floatType="double"/>
<function name="div" arg1="vector" arg2="double"
          floatType="double"/>
```

dual

Computes the dual ($\mathbf{D} = \mathbf{X}\,\mathbf{I}^{-1}$) of a multivector. A custom metric can be specified.

```
<function name="dual" arg1="mv" />
<function name="dual" arg1="double" floatType="double" />
<function name="dual" arg1="pointPair" />
```

undual

Computes the undual ($\mathbf{U} = \mathbf{X}\,\mathbf{I}$) of a multivector. A custom metric can be specified.

```
<function name="undual" arg1="mv" />
<function name="undual" outputName="undual_em" arg1="sphere"
          metric="euclidean"/>
```

equals

Check for equality of two multivectors, up to a scalar epsilon (difference threshold) value.

```
<function name="equals" arg1="mv" arg2="mv" arg3="double"
          floatType="double"/>
<function name="equals" arg1="rotor" arg2="bivector" arg3="double"
          floatType="double"/>
```

extractGrade

Extracts one or more grade parts from a multivector.

```
<function name="extractGrade" arg1="mv"/>
<function name="extractGrade" arg1="evenVersor"/>
```

extractGradeX

Extracts grade X from a multivector.

```
<function name="extractGrade1" arg1="mv"/>
<function name="extractGrade4" arg1="evenVersor"/>
```

gp

Computes the geometric product of two multivectors. A custom metric can be specified.

```
<function name="gp" arg1="mv" arg2="mv"/>
<function name="gp" arg1="vectorE3GA" arg2="vectorE3GA"
          returnType="evenVersor"/>
<function name="gp" outputName="gp_em" arg1="mv" arg2="mv"
          metric="euclidean"/>
```

gradeBitmap

Computes a bitmap which specifies which grade parts of a multivector are non-zero, up to a scalar epsilon (difference threshold) value.

```
<function name="gradeBitmap" arg1="mv" arg2="double"
          floatType="double"/>
<function name="gradeBitmap" arg1="rotor" arg2="double"
          floatType="double"/>
```

hp

Computes the hadamard product (coordinate-wise multiplication) of two multivectors. This is not a true geometric algebra operations, and the hadamard product is not well-defined when the orientation of basis-blades does not match between the two multivectors. It is useful nonetheless, for example when modulating color vectors in computer graphics.

```
<function name="hp" arg1="mv" arg2="mv"/>
<function name="hp" arg1="vector" arg2="vector"/>
```

ihp

Computes the inverse hadamard product (coordinate-wise division) of two multivectors. See the limitations for `hp` above.

```
<function name="ihp" arg1="mv" arg2="mv"/>
<function name="ihp" arg1="plane" arg2="sphere"/>
```

igp

Computes the inverse geometric product of two multivectors ($\mathbf{A}\,\widetilde{\mathbf{B}}/(\mathbf{B}\,\widetilde{\mathbf{B}})$). A custom metric can be specified.

```
<function name="igp" arg1="mv" arg2="mv"/>
<function name="igp" arg1="line" arg2="circle"
          metric="euclidean">
```

increment

Increments a multivector value by one.

```
<function name="increment" arg1="mv"/>
<function name="increment" arg1="bivector" returnType="rotor"/>
<function name="increment" arg1="rotor"/>
```

decrement

Decrements a multivector value by one.

```
<function name="increment" arg1="mv"/>
<function name="increment" arg1="bivector" returnType="rotor"/>
<function name="increment" arg1="rotor"/>
```

hip

Computes the Hestenes inner product of two multivectors (which is zero for scalars). A custom metric can be specified.

```
<function name="hip" outputName="ip" arg1="mv" arg2="mv"/>
<function name="hip" outputName="ip" arg1="oddVersor"
          arg2="rotorE3GA"/>
```

mhip

Computes the modified Hestenes inner product of two multivectors. A custom metric can be specified.

```
<function name="mhip" arg1="mv" arg2="mv"/>
<function name="mhip" arg1="bivector" arg2="scalar"/>
```

lc

Computes the left contraction inner product of two multivectors. The left contraction is zero when the grade of the left argument is higher than the grade of the right argument. A custom metric can be specified.

```
<function name="lc" arg1="mv" arg2="mv"/>
<function name="lc" arg1="vector" arg2="bivector"/>
```

rc

Computes the right contraction inner product of two multivectors. The right contraction is zero when the grade of the right argument is higher than the grade of the left argument. A custom metric can be specified.

```
<function name="rc" arg1="mv" arg2="mv"/>
<function name="rc" arg1="bivector" arg2="vector"/>
```

sp

Computes the scalar product (scalar part of the geometric product) of two multivectors. A custom metric can be specified.

```
<function name="sp" arg1="mv" arg2="mv"/>
<function name="sp" outputName="dotProduct"
          arg1="vector" arg2="vector"
          metric="euclidean"/>
```

log

Computes the logarithm of a rotor. The type of rotor must be specified. Currently, only 3D Euclidean rotors are supported (`optionType="euclidean"`).

```
<function name="log" arg1="mv" optionType="euclidean"/>
<function name="log" arg1="rotor" optionType="euclidean" floatType="double"/>
```

norm

Computes the norm of a multivector. The absolute norm squared is used, i.e., the value returned is $\sqrt{|< \mathbf{X}\,\widetilde{\mathbf{X}} >_0|}$. A custom metric can be specified.

```
<function name="norm" arg1="mv"/>
<function name="norm" arg1="rotor"/>
```

signedNorm

Not implemented yet.

norm2

Computes the squared norm of a multivector. The value returned is $< \mathbf{X}\,\widetilde{\mathbf{X}} >_0$. A custom metric can be specified.

```
<function name="norm2" arg1="mv"/>
<function name="norm2" outputname="norm2_em"
          arg1="evenVersor" metric="euclidean"/>
```

op

Computes the outer product of two multivectors.

```
<function name="op" arg1="mv" arg2="mv"/>
<function name="op" arg1="vector" arg2="vector" />
```

random_blade

Generates a random blade (stored in the general multivector type).

```
<function name="random_blade"/>
<function name="random_blade" floatType="float"/>
```

random_versor

Generates a random versor (stored in the general multivector type). A custom metric can be specified.

```
<function name="random_versor"/>
<function name="random_versor" outputName="random_versor_eucl" metric="euclidea
```

random_scalar

Generates a random floating point value. The `scalar` part of the function name should be replaced with a floating point type, as in the examples below. Use the option `optionGen` to specify the random generator. Currently supported are Mersenne Twister `mt` and the C standard library random generator `libc`.

```
<function name="random_double" outputName="genrand" optionGen="libc"/>
<function name="random_float" optionGen="mt"/>
```

random_smv

Generates a random specialized multivector. The `smv` part of the function name should be replaced with the name of the specialized multivector type.

For scalar, pseudoscalar, vector, dual vector and types defined with `type="multivector"`, random coordinates are generated.

For all other types, random vectors are generated and multiplied using the geometric product. A custom metric can be specified.

```
<function name="random_dualSphere" floatType="double"/>
<function name="random_normalizedPoint" floatType="float"/>
<function name="random_sphere"/>
```

sas

`sas` stands for *scale, add scalar*. It is a function that is used internally by the `exp`, `cos` and `sin` functions. It scales a multivector by a certain factor, then adds a scalar.

```
<function name="sas" arg1="mv" arg2="double" arg3="double" floatType="double"/>
<function name="sas" arg1="bivectorE3GA" arg2="double" arg3="double" floatType=
```

exp

Computes the exponential of a multivector. A custom metric can be specified.

An option `optionSquare="value"` can be used to specify the sign of the square. `value` can be `-1`, `0`, `1`. If the sign of the square is known, much more effective code can be genererated, avoiding a (slow and inprecise) series evaluation. Most of the time though the option is not needed since Gaigen can figure out the sign of the square on its own using symbolic GA.

```
<function name="exp" arg1="mv" />
<function name="exp" outputName="exp_em" arg1="mv" metric="euclidean" />
<function name="exp" arg1="pointPair" />
<function name="exp" arg1="bivectorE3GA" optionSquare="1"/>
```

sin

Computes the sine of a multivector. A custom metric can be specified.

```
<function name="sin" arg1="mv" />
<function name="sin" arg1="flatPoint" />
<function name="sin" arg1="bivectorE3GA" optionSquare="1"/>
```

cos

Computes the cosine of a multivector. A custom metric can be specified.

```
<function name="cos" arg1="mv" />
<function name="cos" outputName="cos_em" arg1="bivector" metric="euclidean"/>
```

sinh

Computes the hyperbolic sine of a multivector. A custom metric can be specified.

```
<function name="sinh" arg1="mv" />
<function name="sinh"arg1="bivector"/>
```

cosh

Computes the hyperbolic cosine of a multivector. A custom metric can be specified.

```
<function name="cosh" arg1="mv" />
```

negate

Negate a multivector.

```
<function name="negate" arg1="mv"/>
<function name="negate" arg1="vector"/>
```

reverse

Reverses a multivector.

```
<function name="reverse" arg1="mv"/>
<function name="reverse" arg1="rotor"/>
```

cliffordConjugate

Computes the Clifford conjugate of a multivector.

```
<function name="cliffordConjugate" arg1="mv"/>
<function name="cliffordConjugate" arg1="evenVersor"/>
```

gradeInvolution

Computes the grade involution of a multivector.

```
<function name="gradeInvolution" arg1="mv"/>
<function name="gradeInvolution" arg1="vector"/>
```

unit

Computes the unit of a multivector. The norm is evaluated using the `norm` function. A custom metric can be specified.

```
<function name="unit" arg1="mv"/>
<function name="unit" outputName="unit_em" arg1="rotor" metric="euclidean"/>
```

versorInverse

Computes the versor inverse of a multivector ($\mathbf{X}/(\mathbf{X}\,\widetilde{\mathbf{X}})$). The norm is evaluated using the `norm` function. A custom metric can be specified.

```
<function name="versorInverse" arg1="mv"/>
<function name="versorInverse" arg1="rotor"/>
```

zero

Checks whether a multivector is zero, up to a scalar epsilon (difference threshold) value.

```
<function name="zero" arg1="mv" arg2="double"
          floatType="double"/>
<function name="zero" arg1="rotor" arg2="float"
          floatType="float"/>
```

13 Converters

By default, converters are generated to convert specialized multivectors to general multivectors and vice versa.

To convert one type of specialized multivector into another, the user should explicitly ask for it to be generated. For example, to request a converter from the specialized multivector type `normalizedPoint` to the specialized multivector type `vectorE3GA`, use:

```
<function name="_vectorE3GA" arg1="normalizedPoint"/>
```

The underscore prefix in the `name` attribute indicates that the function is a converter. This comes from Gaigen 2 where these functions were called *underscore constructors*.

Converters silently drop coordinates that cannot be represented by the destination type. In the above example, the conversion from point to 3-D Euclidean vector loses the `no` (origin) and `ni` (infinity) coordinates.

References

[1] L. Dorst and D. Fontijne and S. Mann. *Geometric Algebra for Computer Science: An Object Oriented Approach to Geometry.* Morgan Kaufmann, revised edition 2009.

[2] D. Fontijne. *Efficient Implementation of Geometric Algebra.* PhD. Thesis, University of Amsterdam, 2007.