

C++ Template Class for Geometric Algebras

nklein software (www.nklein.com)

v1.0.2000.05.17

Contents

1	Version	2
2	License	3
3	Introduction	3
4	The GeometricAlgebra Template Class	4
4.1	Data Members	4
4.2	Methods	5
4.2.1	Constructor	5
4.2.2	Copy Constructor	6
4.2.3	Assignment Operator	6
4.2.4	Coefficient Accessors	6
4.2.5	Addition	7
4.2.6	Subtraction	8
4.2.7	Negation	9
4.2.8	Grade Involution	10
4.2.9	Reversion	11
4.2.10	Conjugation	12
4.2.11	Multiplication	13
4.3	Class	15
5	Test Code	16
6	The GeomMultTable Template Class	17
6.1	Data	18
6.2	Methods	18
6.2.1	Constructor	18
6.2.2	Query Method	20
6.3	Class	20

7	The <code>GeomGradeTable</code> Template Class	20
7.1	Data	21
7.2	Methods	21
	7.2.1 Constructor	21
	7.2.2 Query Method	22
7.3	Class	22
8	Source Files	23
8.1	<code>geoma.h</code>	23
8.2	<code>geomaData.h</code>	23
8.3	<code>geoma.cc</code>	24

1 Version

2 $\langle version\ 2 \rangle \equiv$ (23b)
 "v1.0.2000.05.17"

2 License

Adapted from <http://www.nklein.com/etc/copyright.html>

We at nklein software made all text, software, and other stuff in this package. We authorize you to do anything you like with these so long as you do not restrict the rights of others to do what they like with them. We're not saying you have to give away your products. We're just saying that all of the items in this package have a Universal, Non-Exclusive License.

For example, if you wanted to take some of this text or some of this software and plaster your name on them and sell them, fine. But, you cannot keep Sally Q. Public from taking those same items and plastering her name on them and selling them. You just can't. It's all as hers as it is yours.

All of that said, it'd please us plenty if you slung appreciation, accolades, credit, and/or cash our way as you see fit.

3 Introduction

The geometric algebras or Clifford algebras are very useful in a variety of areas. There are some packages out there to deal with them in Maple and Java and such. But, to our knowledge, this is the only publically available C++ template class to implement them.

The Clifford algebra $\mathcal{C}\ell_{p,q}$ is an algebra generated by vectors from a quadratic space. The first p unit vectors contribute positively to the norm and the other q unit vectors contribute negatively to the norm. For unit vectors \mathbf{e}_i and \mathbf{e}_j ,

$$\mathbf{e}_i \mathbf{e}_j = \mathbf{e}_{ij} = \begin{cases} 1 & 1 \leq i = j \leq p \\ -1 & p < i = j \leq p + q \\ -\mathbf{e}_j \mathbf{e}_i = -\mathbf{e}_{ji} & i \neq j \end{cases} \quad (1)$$

In other words, a vector $\mathbf{r} = \sum_{i=1}^{p+q} a_i \mathbf{e}_i$ obeys

$$\mathbf{r} \mathbf{r} = \mathbf{r}^2 = \langle \mathbf{r}, \mathbf{r} \rangle = \sum_{i=1}^p a_i^2 - \sum_{i=p+1}^{p+q} a_i^2 \quad (2)$$

All of the cross terms here cancel out because $\mathbf{e}_{ij} = -\mathbf{e}_{ji}$ when $i \neq j$.

In a general multiplication of two vectors \mathbf{a} and \mathbf{b} , these terms do not cancel out. But, because of the anticommutativity of the cross-terms, we can always sort the order of the subscripts and only affect the sign of the coefficient. For example:

$$\begin{aligned} \mathbf{e}_{3142} &= -\mathbf{e}_{3124} \\ &= \mathbf{e}_{1324} \\ &= -\mathbf{e}_{1234} \end{aligned}$$

And, identical subscripts annihilate each other when adjacent. For example, in $\mathcal{C}^{\ell_{1,3}}$

$$\begin{aligned} \mathbf{e}_{2142} &= -\mathbf{e}_{2124} \\ &= \mathbf{e}_{1224} \\ &= -\mathbf{e}_{14} \end{aligned}$$

and

$$\begin{aligned} \mathbf{e}_{2141} &= -\mathbf{e}_{2114} \\ &= -\mathbf{e}_{24} \end{aligned}$$

There are some excellent introductions to Clifford algebras available on the web. Some of these are:

- <http://www.mrao.cam.ac.uk/~clifford>—The Geometric Algebra Group at Cambridge
- <http://www.hit.fi/~lounesto>—Pertti Lounesto whose excellent book *Clifford Algebras and Spinors* got me started with Clifford algebras.
- <http://www.clifford.org>—The International Clifford Algebra Society (though this page is fairly out of date).

The class implemented in this document is a template class that requires three template parameters: the data type for scalars, the value of p , and the value of q . The data type for scalars must support addition, subtraction, multiplication, assignment from another member of the same type, and assignment from the integer 0. The multiplication need not be commutative.

4 The GeometricAlgebra Template Class

This is the main template class generated in this document. It implements addition of a multivector and scalar, subtraction of a scalar from a multivector, and (left or right) multiplication of a multivector by a scalar. It implements the negation, addition, subtraction, multiplication, coefficient access, grade-involution, reversion, and conjugation of arbitrary elements of $\mathcal{C}^{\ell_{p,q}}$.

4.1 Data Members

The coefficients of the various k -forms are stored in an array. The array has to contain 2^{p+q} coefficients. These are the $\binom{p+q}{k}$ k -forms for all $0 \leq k \leq p+q$.

The binary digits of the index specify which unit vectors make up this k -form. A k -form will have k -bits set in the index. Because every k -form can be reordered with transpositions (with a possible change of sign), we only need to track which unit vectors compose a given k -form. We do not need to track

the order in which they appear. The n -th bit of the index will specify the unit vector \mathbf{e}_{n+1} .

Here are some examples of k -forms from $\mathcal{C}\ell_{1,3}$ and their indices in base 2 and base 10.

$$\begin{aligned}
 \mathbf{1} &= 0000_2 = 0_{10} \\
 \mathbf{e}_2 &= 0010_2 = 2_{10} \\
 \mathbf{e}_{23} &= 0110_2 = 6_{10} \\
 \mathbf{e}_{14} &= 1001_2 = 9_{10} \\
 \mathbf{e}_{134} &= 1101_2 = 13_{10} \\
 \mathbf{e}_{1234} &= 1111_2 = 15_{10}
 \end{aligned} \tag{3}$$

So, the array of coefficients must hold 2^{p+q} elements of the data type `Type` given to the template.

```
5a  <data members 5a>≡ (15)
      Type coef[ 1U << (P+Q) ];
```

4.2 Methods

Herein lie the implementations of the constructor, the coefficient accessor, the addition, subtraction, multiplication, and various involutions.

4.2.1 Constructor

In the constructor for the `GeometricAlgebra` template class, we simply initialize all of the coefficients to zero if the `init` parameter is `true`. The `init` parameter defaults to `true`. But, we allow one to skip initialization in order to allow one to optimize out the initialization of things that will just be assigned over immediately. This is especially useful if assigning the integer 0 to an element of type `Type` is expensive.

```
5b  <public methods 5b>≡ (15) 6a▷
      GeometricAlgebra( bool init = true )
      {
          if ( init ) {
              for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
                  coef[ii] = 0;
              }
          }
      }
```

Uses `GeometricAlgebra 15`.

4.2.2 Copy Constructor

In the copy constructor for the `GeometricAlgebra` template class, we employ the use of the assignment operator.

```
6a  <public methods 5b>+≡ (15) <5b 6b>
      inline
      GeometricAlgebra( const GeometricAlgebra<Type,P,Q>& b )
      {
          *this = b;
      }
```

Uses `GeometricAlgebra` 15.

4.2.3 Assignment Operator

In the assignment operator for the `GeometricAlgebra` template class, we employ the use of the assignment operator for `Type` and simply copy the coefficients.

```
6b  <public methods 5b>+≡ (15) <6a 6c>
      inline GeometricAlgebra<Type,P,Q>&
      operator =( const GeometricAlgebra<Type,P,Q>& b )
      {
          for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
              coef[ii] = b.coef[ii];
          }
          return *this;
      }
```

Uses `GeometricAlgebra` 15.

4.2.4 Coefficient Accessors

In the coefficient accessor, we make sure that the index is within the valid range. If it is too big, we huck an exception. Otherwise, we return the requested coefficient.

```
6c  <public methods 5b>+≡ (15) <6b 7a>
      inline Type&
      operator [] ( const unsigned int index )
      {
          if ( index >= ( 1U << (P+Q) ) ) {
              throw out_of_range( "index" );
          }
          return coef[ index ];
      }
```

And, we made a const version of the same thing.

```
7a  <public methods 5b>+≡ (15) <6c 10>
      inline const Type&
      operator [] ( const unsigned int index ) const
      {
          if ( index >= ( 1U << (P+Q) ) ) {
              throw out_of_range( "index" );
          }
          return coef[ index ];
      }
```

4.2.5 Addition

The addition of a multivector and a scalar is quite simple. We simply add the scalar to the zero-th coefficient of the multivector.

```
7b  <friend methods 7b>≡ (23a) 7c>
      template <class Type, const unsigned int P, const unsigned int Q=0>
      inline
      GeometricAlgebra<Type,P,Q>
      operator + ( const GeometricAlgebra<Type,P,Q>& a,
                  const Type& b )
      {
          GeometricAlgebra<Type,P,Q> c = a;

          c[0] = a[0] + b;

          return c;
      }
```

Uses GeometricAlgebra 15.

And, left-addition of a scalar is basically the same thing.

```
7c  <friend methods 7b>+≡ (23a) <7b 8a>
      template <class Type, const unsigned int P, const unsigned int Q=0>
      inline
      GeometricAlgebra<Type,P,Q>
      operator + ( const Type& a,
                  const GeometricAlgebra<Type,P,Q>& b )
      {
          GeometricAlgebra<Type,P,Q> c = b;

          c[0] = a + b[0];

          return c;
      }
```

Uses GeometricAlgebra 15.

The addition of two multivectors is fairly straightforward. We simply add the corresponding components.

```
8a  <friend methods 7b>+≡ (23a) <7c 8b>
    template <class Type, const unsigned int P, const unsigned int Q=0>
      inline
      GeometricAlgebra<Type,P,Q>
      operator + ( const GeometricAlgebra<Type,P,Q>& a,
                  const GeometricAlgebra<Type,P,Q>& b )
      {
        GeometricAlgebra<Type,P,Q> c(false);

        for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
          c[ii] = a[ii] + b[ii];
        }

        return c;
      }
```

Uses GeometricAlgebra 15.

4.2.6 Subtraction

The subtraction of a scalar from a multivector is quite simple. We simply subtract the scalar from the zero-th coefficient of the multivector.

```
8b  <friend methods 7b>+≡ (23a) <8a 9a>
    template <class Type, const unsigned int P, const unsigned int Q=0>
      inline
      GeometricAlgebra<Type,P,Q>
      operator - ( const GeometricAlgebra<Type,P,Q>& a,
                  const Type& b )
      {
        GeometricAlgebra<Type,P,Q> c = a;

        c[0] = a[0] - b;

        return c;
      }
```

Uses GeometricAlgebra 15.

At this point, we have opted not to subtract multivectors from scalars. If you want this functionality, you will just have to employ left-scalar addition and multivector negation.

The subtraction of two multivectors is fairly straightforward. We simply subtract the corresponding components.

```
9a  <friend methods 7b>+≡ (23a) <8b 9b>
    template <class Type, const unsigned int P, const unsigned int Q>
      inline
      GeometricAlgebra<Type,P,Q>
      operator - ( const GeometricAlgebra<Type,P,Q>& a,
                  const GeometricAlgebra<Type,P,Q>& b )
      {
        GeometricAlgebra<Type,P,Q> c(false);

        for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
          c[ii] = a[ii] - b[ii];
        }

        return c;
      }
}
```

Uses GeometricAlgebra 15.

4.2.7 Negation

The negation of a multivector is quite straightforward. We simply negate each coefficient of the multivector. Rather than require the Type of the coefficients to support unary negation, we will just subtract from a zero scalar.

```
9b  <friend methods 7b>+≡ (23a) <9a 13a>
    template <class Type, const unsigned int P, const unsigned int Q=0>
      inline
      GeometricAlgebra<Type,P,Q>
      operator - ( const GeometricAlgebra<Type,P,Q>& a )
      {
        GeometricAlgebra<Type,P,Q> b(false);
        Type zero = 0;

        for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
          b[ii] = zero - a[ii];
        }

        return b;
      }
}
```

Uses GeometricAlgebra 15.

4.2.8 Grade Involution

The grade involution of a multivector is a bit funky. It toggles the sign of every coefficient of an odd-graded element. In code, this amounts to whether the bottom bit of the grade is set. Rather than require the Type of the coefficients to support unary negation, we will just subtract it from a zero scalar.

```
10  <public methods 5b>+≡ (15) <7a 11>
    inline
    GeometricAlgebra<Type,P,Q>
    GradeInvolution( void ) const
    {
        GeometricAlgebra<Type,P,Q> a(false);
        Type zero = 0;

        for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
            if ( ( GetGrade( ii ) & 1 ) != 0 ) {
                a[ii] = zero - coef[ii];
            } else {
                a[ii] = coef[ii];
            }
        }

        return a;
    }
}
```

Uses GeometricAlgebra 15.

4.2.9 Reversion

The reversion of a multivector is fairly hairy. It reverses the order of the subscripts for each k -form. For example, the 3-form \mathbf{e}_{123} becomes $\mathbf{e}_{321} = -\mathbf{e}_{123}$ while the 4-form \mathbf{e}_{1234} does not change signs.

Now, the sign only depends upon the odd-ness or even-ness of the number of transpositions required to get things back in order. This obeys a simple recurrence relationship. Let $T(n)$ be the number of transpositions required to revert an n -form. Then, we can see that $T(n + 1) = T(n) + n - 1$ because it will require $T(n)$ transpositions to reorder the first n subscripts and $n - 1$ transpositions to get the $n + 1$ -th subscript from one end of the list to the other.

With this recurrence relationship, we can see that the odd-ness or even-ness of $T(n + 4)$ is the same as that of $T(n)$, because

$$\begin{aligned} T(n + 4) &= T(n + 3) + n + 2 \\ &= T(n + 2) + 2n + 3 \\ &= T(n + 1) + 3n + 3 \\ &= T(n) + 4n + 2 \end{aligned}$$

And, because $T(0)$ and $T(1)$ are even while $T(2)$ and $T(3)$ are odd, we have that an n -form requires an odd number of transpositions to revert iff $n \equiv 2$ or $n \equiv 3$ modulo 4. In code, this translates to whether the second bit of the grade is set.

```

11  <public methods 5b>+≡ (15) <10 12>
      inline
      GeometricAlgebra<Type,P,Q>
      Reversion( void ) const
      {
          GeometricAlgebra<Type,P,Q> a(false);
          Type zero = 0;

          for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
              if ( ( GetGrade( ii ) & 2 ) != 0 ) {
                  a[ii] = zero - coef[ii];
              } else {
                  a[ii] = coef[ii];
              }
          }

          return a;
      }

```

Uses GeometricAlgebra 15.

4.2.10 Conjugation

The conjugation of a multivector is a grade involution and a reversion (in either order). Thus, this code is similar to the code in the previous two sections. The sign of the coefficient changes iff either but not both of the grade involution and reversion would change it.

```
12  <public methods 5b>+≡ (15) <11
      inline
      GeometricAlgebra<Type,P,Q>
      Conjugation( void ) const
      {
          GeometricAlgebra<Type,P,Q> a(false);
          Type zero = 0;

          for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
              switch ( GetGrade( ii ) & 3 ) {
                  case 0:
                  case 3:
                      a[ii] = coef[ii];
                      break;
                  case 1:
                  case 2:
                      a[ii] = zero - coef[ii];
                      break;
              }
          }

          return a;
      }

```

Uses GeometricAlgebra 15.

4.2.11 Multiplication

The multiplication of a multivector by a scalar is rather straightforward. We simply multiply each coefficient in the multivector by the scalar.

```
13a  <friend methods 7b>+≡ (23a) <9b 13b>
      template <class Type, const unsigned int P, const unsigned int Q=0>
      inline
      GeometricAlgebra<Type,P,Q>
      operator * ( const GeometricAlgebra<Type,P,Q>& a,
                  const Type& b )
      {
          GeometricAlgebra<Type,P,Q> c;

          for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
              c[ ii ] = a[ ii ] * b;
          }

          return c;
      }
```

Uses GeometricAlgebra 15.

Here, since multiplication need not be commutative in Type, we must take to preserve this.

```
13b  <friend methods 7b>+≡ (23a) <13a 14>
      template <class Type, const unsigned int P, const unsigned int Q=0>
      inline
      GeometricAlgebra<Type,P,Q>
      operator * ( const Type& a,
                  const GeometricAlgebra<Type,P,Q>& b )
      {
          GeometricAlgebra<Type,P,Q> c;

          for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
              c[ ii ] = a * b[ ii ];
          }

          return c;
      }
```

Uses GeometricAlgebra 15.

The multiplication of two multivectors is a bit more complicated. Here, we have to sum up all of the terms that contribute to each coefficient in the product. Fortunately, with the index-scheme that we defined in 4.1 on page 4, the product of the i -th term of the first vector and the j -th term of the second vector contributes to the $i \otimes j$ coefficient where \otimes is a bitwise XOR. The sign of the product is stored in the `GeomMultTable`.

```

14  <friend methods 7b>+≡ (23a) <13b
    template <class Type, const unsigned int P, const unsigned int Q=0>
      inline
      GeometricAlgebra<Type,P,Q>
      operator * ( const GeometricAlgebra<Type,P,Q>& a,
                  const GeometricAlgebra<Type,P,Q>& b )
      {
        GeometricAlgebra<Type,P,Q> c;

        for ( unsigned int ii=0; ii < ( 1U << (P+Q) ); ++ii ) {
          for ( unsigned int jj=0; jj < ( 1U << (P+Q) ); ++jj ) {
            unsigned int index = ( ii ^ jj );
            if ( c.IsPositive( ii, jj ) ) {
              c[ index ] = c[ index ] + ( a[ ii ] * b[ jj ] );
            } else {
              c[ index ] = c[ index ] - ( a[ ii ] * b[ jj ] );
            }
          }
        }

        return c;
      }

```

Uses `GeometricAlgebra` 15.

4.3 Class

The actual `GeometricAlgebra` template class itself inherits from the `GeomMultTable` template class which is described in section 6 on page 17. The `GeometricAlgebra` template class uses the `GeomMultTable` template class to maintain the lookup tables that are used in the multiplication, grade involution, reversion, and conjugation. Beyond that, it contains some data members of its own and the methods that were implemented above.

```
15  <template class 15>≡ (23a)
    template <class Type, const unsigned int P, const unsigned int Q=0>
      class GeometricAlgebra
        : public nklein_priv::GeomMultTable<P,Q>
        {
        protected:
          <data members 5a>
        public:
          <public methods 5b>
        };
```

Defines:

`GeometricAlgebra`, used in chunks 5–14, 16, and 23b.

Uses `GeomMultTable` 20b.

5 Test Code

The test code creates a couple of $C\ell_{2,1}$ multivectors and does a variety of things to them. At each step, the multivector is printed so that the reader can verify that things function as claimed. This code also creates several other types of multivectors. The point of this is to ensure that multiple types of multivectors can be created without the templates losing their heads. Additionally, one can check with a tool such as `nm(1)` to ensure that this code only creates one `gradeTable` (for $n = 3$) and two `multTables` (one for $p = 2, q = 1$ and one for $p = 3, q = 0$).

```
16  <test code 16>≡ (24)
    int
    main( void )
    {
        nklein::GeometricAlgebra< int, 2, 1 > a;
        nklein::GeometricAlgebra< int, 2, 1 > b;
        nklein::GeometricAlgebra< int, 2, 1 > c(false);
        nklein::GeometricAlgebra< int, 3 > d;
        nklein::GeometricAlgebra< double, 3 > e;
        nklein::GeometricAlgebra< complex< double >, 3 > f;

        a[0] = 1;
        a[1] = 1;
        a[3] = 1;
        a[7] = 1;

        b[1] = 1;
        b[2] = 1;
        b[4] = 1;

        c = a;
        cout << "a = " <<
            <test code print c 17>;

        c = b;
        cout << "b = " <<
            <test code print c 17>;

        c = 2 * a - b;
        cout << "2*a - b = " <<
            <test code print c 17>;

        c = a.GradeInvolution();
        cout << "\\hat{a} = " <<
            <test code print c 17>;
```



```

c = a.Reversion();
cout << "\\tilde{a} = " <<
    <test code print c 17>;

c = a.Conjugation();
cout << "\\bar{a} = " <<
    <test code print c 17>;

c = b*b;
cout << "b*b = " <<
    <test code print c 17>;

c = a*b;
cout << "a*b = " <<
    <test code print c 17>;

c = b*a;
cout << "b*a = " <<
    <test code print c 17>;

return 0;
}

```

Uses GeometricAlgebra 15.

To print out a $\mathcal{C}\ell_{2,1}$ multivector, we simply emit each coefficient with the appropriate k -form.

```

17 <test code print c 17>≡ (16)
                                c[0]
    << " + " << c[1] << "e_1"
    << " + " << c[2] << "e_2"
    << " + " << c[4] << "e_3"
    << " + " << c[3] << "e_{12}"
    << " + " << c[5] << "e_{13}"
    << " + " << c[6] << "e_{23}"
    << " + " << c[7] << "e_{123}"
    << endl

```

6 The GeomMultTable Template Class

The general GeometricAlgebra template class uses the GeomMultTable template class. This template class is not in the nklein namespace alongside the GeometricAlgebra template class. It is not intended for general use. But, it helps conserve memory.

6.1 Data

The thrust of this table is that it holds the sign of the multiplication of a given j -form by a given k -form. Which indices correspond to which k -forms is describe in section 4.1 on page 4.

18a \langle template multiplication table private data 18a $\rangle \equiv$ (20b)
static int multTable[1U << (P+Q)][1U << (P+Q)];

18b \langle template multiplication table data 18b $\rangle \equiv$ (23b)
template<const unsigned int P, const unsigned int Q=0>
int GeomMultTable<P,Q>::multTable[1U << (P+Q)][1U << (P+Q)];
Uses GeomMultTable 20b.

6.2 Methods

Herein lie the implementations of the constructor and the query methods for this template class.

6.2.1 Constructor

To fill the table, we run through every combination of coefficient ii by coefficient jj and we track how many times we have to “move” bits of jj past bits of ii . The $iiTopBits$ variable keeps track of how many bits we may still have to slide past in ii .

18c \langle template multiplication methods 18c $\rangle \equiv$ (20b)
GeomMultTable()
{
for (unsigned ii=0; ii < (1U << (P+Q)); ++ii) {
unsigned int iiInitialTopBits = GetGrade(ii);

for (unsigned jj=0; jj < (1U << (P+Q)); ++jj) {
unsigned int iiTopBits = iiInitialTopBits;

 \langle template multiplication table calculate sign 19a \rangle

multTable[ii][jj] = sign;
}
}
}

Uses GeomMultTable 20b.

To calculate the sign of the multiplication of ii and jj , we go through each of the bits which are set in jj . We know that the sign is a function of the parity of the number of transpositions that this bit must incur to navigate into the proper spot in the result. The number of transpositions is tracked in `iiTopBits` which tells how many bits in ii are above the kk -th bit. And, if the bit is also set in ii , then we have translated the bit next to an adjacent one. If that is the case, then we must annihilate the like subscripts.

19a `<template multiplication table calculate sign 19a>`≡ (18c)
`int sign = 1;`

```
for ( unsigned int kk=0; kk < (P+Q); ++kk ) {
    unsigned int bit = ( 1U << kk );
```

`<template multiplication table update iiTopBits 19b>`

```
if ( ( jj & bit ) != 0 ) {
    sign *= ( iiTopBits & 1 ) ? -1 : 1;
```

```
    if ( ( ii & bit ) != 0 ) {
        <template multiplication table annihilate like indices 19c>
    }
}
```

```
}
```

If the current bit is set in ii , then we must decrement the number of bits that the bit in jj will have to pass on its way into position.

19b `<template multiplication table update iiTopBits 19b>`≡ (19a)
`if ((ii & bit) != 0) {`
 `--iiTopBits;`
`}`

If the unit vector that we are sliding is one of the first p , then the sign is fine the way it is. But, if it is one of the other q , then we have to toggle the sign to annihilate the indices.

19c `<template multiplication table annihilate like indices 19c>`≡ (19a)
`if (kk >= P) {`
 `sign *= -1;`
`}`

6.2.2 Query Method

One can check the sign adjustment of the multiplication of the `ii` coefficient by the `jj` coefficient by calling this method. It simply range checks the indices and then returns whether the `multTable` entry for those indices is positive.

```
20a  <template multiplication public methods 20a>≡ (20b)
      static inline bool IsPositive( unsigned int ii, unsigned int jj )
      {
          if ( ii >= ( 1U << (P+Q) ) ) {
              throw out_of_range( "first index" );
          }
          if ( jj >= ( 1U << (P+Q) ) ) {
              throw out_of_range( "second index" );
          }
          return ( multTable[ii][jj] >= 0 );
      }
```

6.3 Class

The multiplication table inherits from the `GeomGradeTable` which is described in section 7 on page 20. The class itself contains some private data, a protected constructor, and the public query method.

```
20b  <template multiplication table 20b>≡ (23a)
      template<const unsigned int P, const unsigned int Q=0>
      class GeomMultTable : public GeomGradeTable<P+Q> {
      private:
          <template multiplication table private data 18a>
      protected:
          <template multiplication methods 18c>
      public:
          <template multiplication public methods 20a>
      };
```

Defines:

`GeomMultTable`, used in chunks 15 and 18.

Uses `GeomGradeTable` 22b.

7 The GeomGradeTable Template Class

The general `GeometricAlgebra` template class and the `GeomMultTable` template class use the `GeomGradeTable` template class. This template class is not in the `nklein` namespace alongside the `GeometricAlgebra` template class. It is not intended for general use. But, it helps conserve memory.

7.1 Data

The thrust of this table is that it holds the grade of each coefficient.

```
21a <template grade table private data 21a>≡ (22b)
    static int gradeTable[ 1U << (N) ];
```

```
21b <template grade table data 21b>≡ (23b)
    template<const unsigned int N>
        int GeomGradeTable<N>::gradeTable[ 1U << (N) ];
```

Uses GeomGradeTable 22b.

7.2 Methods

Herein lie the implementations of the constructor and the query methods for this template class.

7.2.1 Constructor

We simply count the number of bits that are set in *ii*. This is the grade of the coefficient with index *ii*.

```
21c <template grade methods 21c>≡ (22b)
    GeomGradeTable()
    {
        for ( unsigned ii=0; ii < ( 1U << (N) ); ++ii ) {
            unsigned int iiBits = 0;
            <template grade table count bits in ii 21d>
            gradeTable[ii] = iiBits;
        }
    }
```

Uses GeomGradeTable 22b.

To count the bits in *ii*, we loop through each byte of *ii* with the help of the `char*` called *ptr*. At each byte, we add in the number of bits which are set in the low nibble and the high nibble with the help of the lookup table *lut*.

```
21d <template grade table count bits in ii 21d>≡ (21c)
    char* ptr = (char*)&ii;

    for ( unsigned int kk=0; kk < sizeof(unsigned int); ++kk ) {
        static const unsigned int lut[] = {
            0, 1, 1, 2, 1, 2, 2, 3,
            1, 2, 2, 3, 2, 3, 3, 4
        };
        iiBits += lut[ ( ptr[kk] >> 0 ) & 0x0F ];
        iiBits += lut[ ( ptr[kk] >> 4 ) & 0x0F ];
    }
```

7.2.2 Query Method

The `GeomGradeTable` class has this accessor method to retrieve the grade of a given index. It simply range checks the index and then returns the table entry for the given index.

```
22a  <template grade public methods 22a>≡ (22b)
      static inline unsigned int GetGrade( unsigned int index )
      {
          if ( index >= ( 1U << (N) ) ) {
              throw out_of_range( "index" );
          }
          return gradeTable[index];
      }
```

7.3 Class

The grade table template class contains some private data, a protected constructor, and its public query method.

```
22b  <template grade table 22b>≡ (23a)
      template<const unsigned int N>
      class GeomGradeTable {
      private:
          <template grade table private data 21a>
      protected:
          <template grade methods 21c>
      public:
          <template grade public methods 22a>
      };
```

Defines:

`GeomGradeTable`, used in chunks 20 and 21.

8 Source Files

In order to use all of this stuff, we will have to break it out into source files. We have broken it up into three source files which will hopefully ensure the best use of memory.

8.1 geoma.h

The `geoma.h` file contains the declarations and implementations of each of the classes described above. These are all wrapped in the `nklein` namespace to hopefully avoid collisions with anything else in the free world.

```
23a <geoma.h 23a>≡
    namespace nklein {
        namespace nklein_priv {
            <template grade table 22b>
            <template multiplication table 20b>
        };

        <template class 15>
        <friend methods 7b>
    };
```

8.2 geomaData.h

The `geomaData.h` file contains all of the static variable declarations needed by the template classes in `geoma.h`. We separated this out from the rest of `geoma.h` so that if one is using the same vector type across multiple source files, one would only need to have the `geomaData.h` included in one of them. This avoids having the table declared in multiple places.

```
23b <geomaData.h 23b>≡
    namespace nklein {
        namespace nklein_priv {
            <template multiplication table data 18b>
            <template grade table data 21b>
        };

        static const char* GeometricAlgebraVersion
            = "nklein::GeometricAlgebra::version: " <version 2>;
    };
```

Uses `GeometricAlgebra 15`.

8.3 geoma.cc

And, the `geoma.cc` simply includes the test code above. In practice, you will not need this file at all. It simply demonstrates how you would go about employing this template class.

```
24  <geoma.cc 24>≡
      #include <iostream>
      #include <stdexcept>
      #include <complex>
      #include "geoma.h"
      #include "geomaData.h"

      <test code 16>
```

Document Information

This document was created using `vi`, `noweb`, and `LATEX`.

Noweb Index

GeometricAlgebra: 5b, 6a, 6b, 7b, 7c, 8a, 8b, 9a, 9b, 10, 11, 12, 13a, 13b, 14, [15](#),
16, 23b

GeomGradeTable: 20b, 21b, 21c, [22b](#)

GeomMultTable: 15, 18b, 18c, [20b](#)