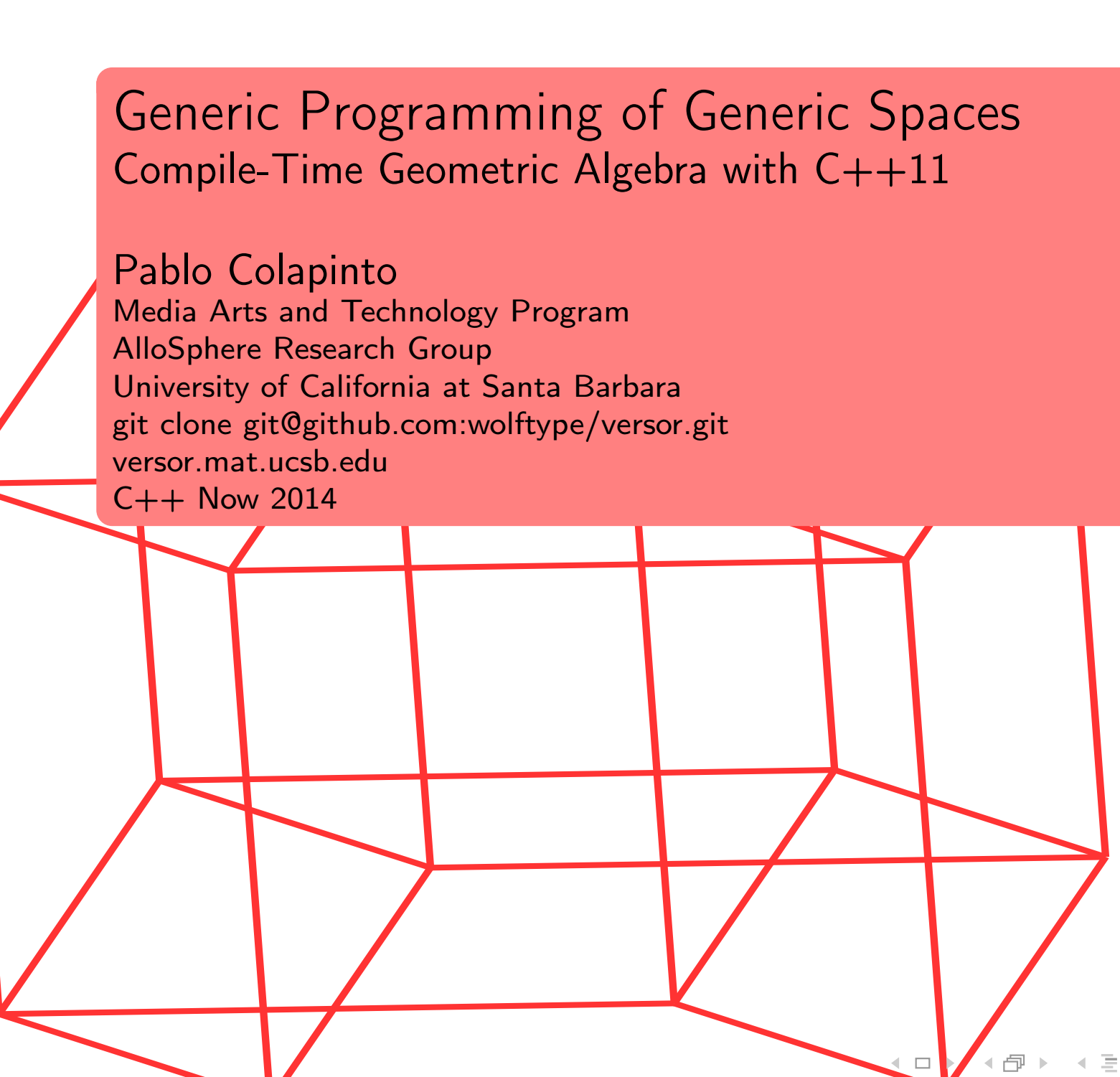
Generic Programming of Generic Spaces

Compile-Time Geometric Algebra with C++11

Pablo Colapinto
Media Arts and Technology Program
AlloSphere Research Group
University of California at Santa Barbara
git clone git@github.com:wolftype/versor.git
versor.mat.ucsb.edu
C++ Now 2014



Goals for the next 90 minutes

- Build Spatial Intuition in **n-Dimensions** with Geometric Algebra
- Explore **Generative Type Creation** with Basic Template Metaprogramming Idioms
- **Challenge** each other

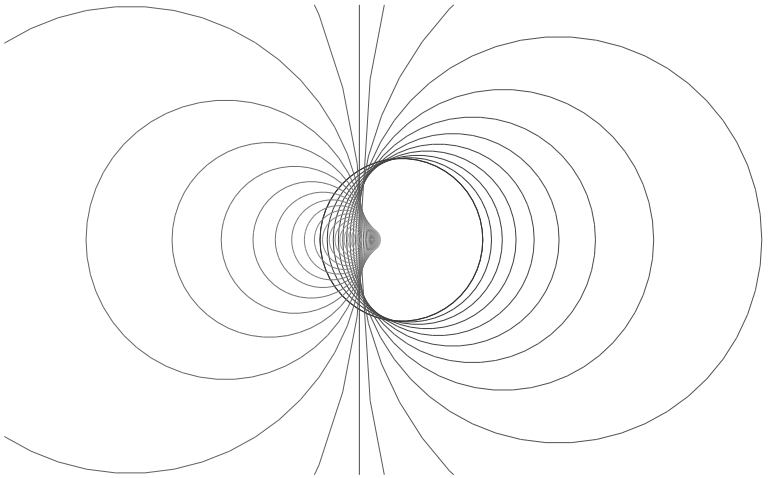
> *We now proceed to do something which must apparently introduce the greatest confusion, but which, on the other hand, increases enormously our powers.*
>
> *-William Clifford*

Geometric Algebra is an ND Spatial Logic Engine

- **Unifies** inductive (algebraic, analytic) and deductive (geometric, synthetic) logic
- **Isomorphisms:** Tensor, vector, matrix, and lie algebras all subsumed in a **compact** way
- **Outermorphism**: problems worked out in lower dimensions can be **extrapolated** to higher dimensions (i.e. curl)
- **Automorphisms**: Concatenation of operations leads to immense **expressivity**
- **Embedded Metrics**: Non-Euclidean and experimental signatures built-in on the ground floor ease the calculus of differential geometry.

Applications of Geometric Algebra Computing

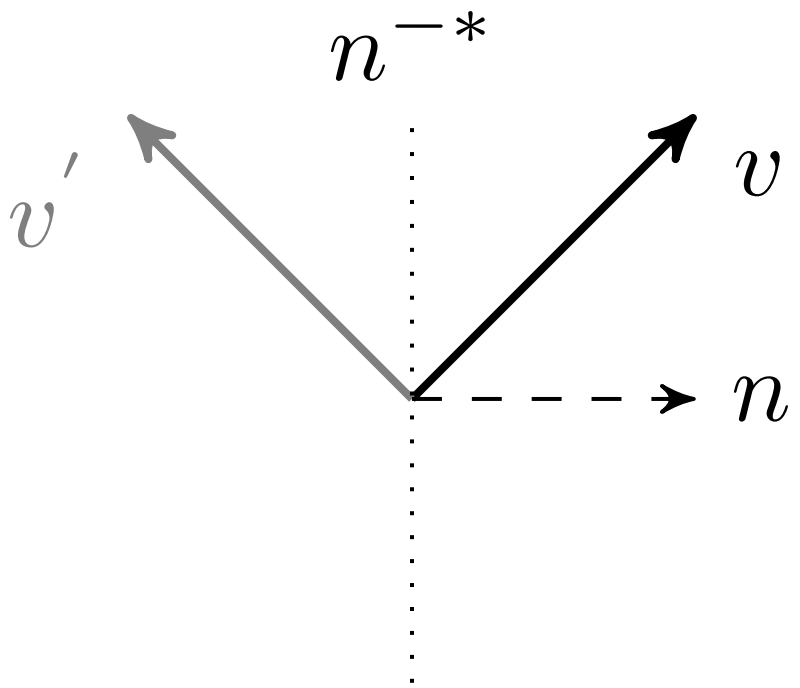
- Classical Mechanics and Particle Physics
- Molecular Modeling and Crystallography
- Electrodynamics and Optics
- Digital Signal Processing and Computer Vision
- Robotics and Kinematics
- Relativistic Physics and Gauge Theory
- 3D Computer Graphics and Experimental Visualization



Conformal Geometric Algebra is a Generalized Homogenous Mathematical System

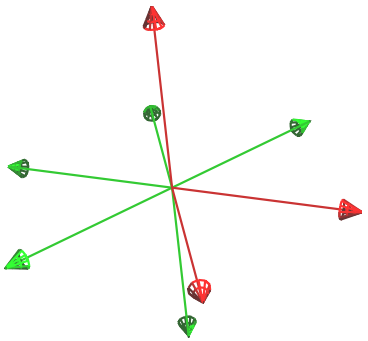
- Points and Vectors are **disambiguated**
- Dot product between points proportional to Euclidean **distance** (squared)
- Algebraic **elements** include Tangent Vectors, Direction Vectors, Circles, Point Pairs, Spheres Lines, Dual LInes
- An additonal array of **operators**: For Translating, Rotating, Twisting, Dilating, Inverting, and Transversing Elements.
- Linearization of all conformal **transformations** simplifies nth-order perturbations.

Reflections in any dimension using geometric products

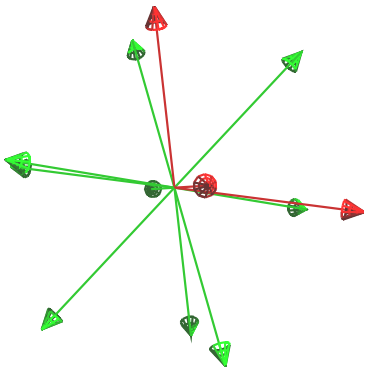


- $v' = -nvn$
- n is a **versor**
- n^{-*} is the **dual** of n

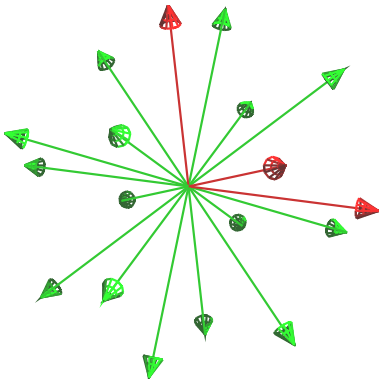
Example: Generating Reflection Groups



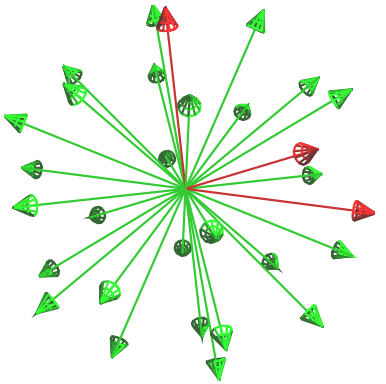
2, 3



3, 3

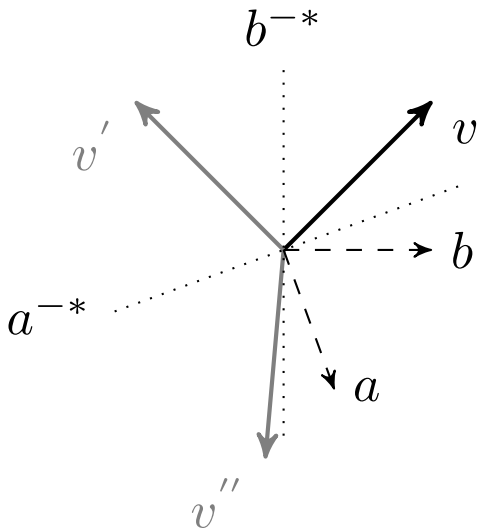


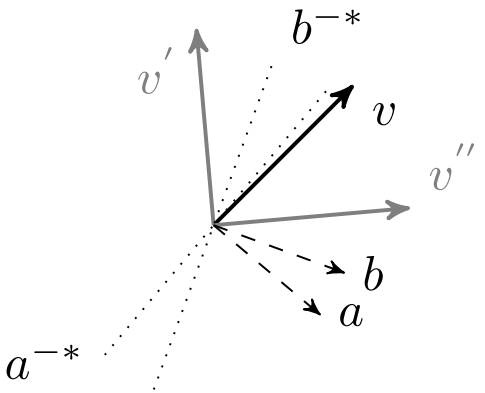
4, 3



5, 3

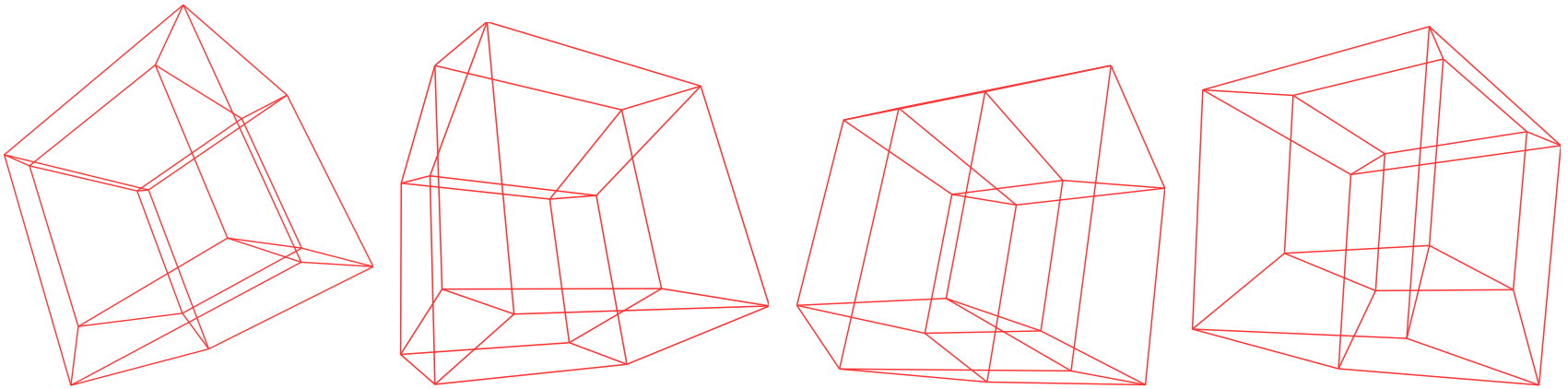
Rotations composed of two reflections $\mathcal{R} = \boldsymbol{ab}$





$v'' = abvba$ defines the reflection of v about two planes with normal versors a and b (a reflection by b followed by a reflection by a). The equation for double reflection is simplifed to $v'' = RvR$ where ab defines the **rotor** R and $\tilde{R}$ defines its reverse ba.

Example: Rotating a Hypercube



Hyperplanes in any dimension using wedge products

We can homogenize a 3D vector a by adding an extra dimension with unit weight.

$$hom : a \mapsto a + e_4$$

Three homogenized vectors are wedged together to define a plane:

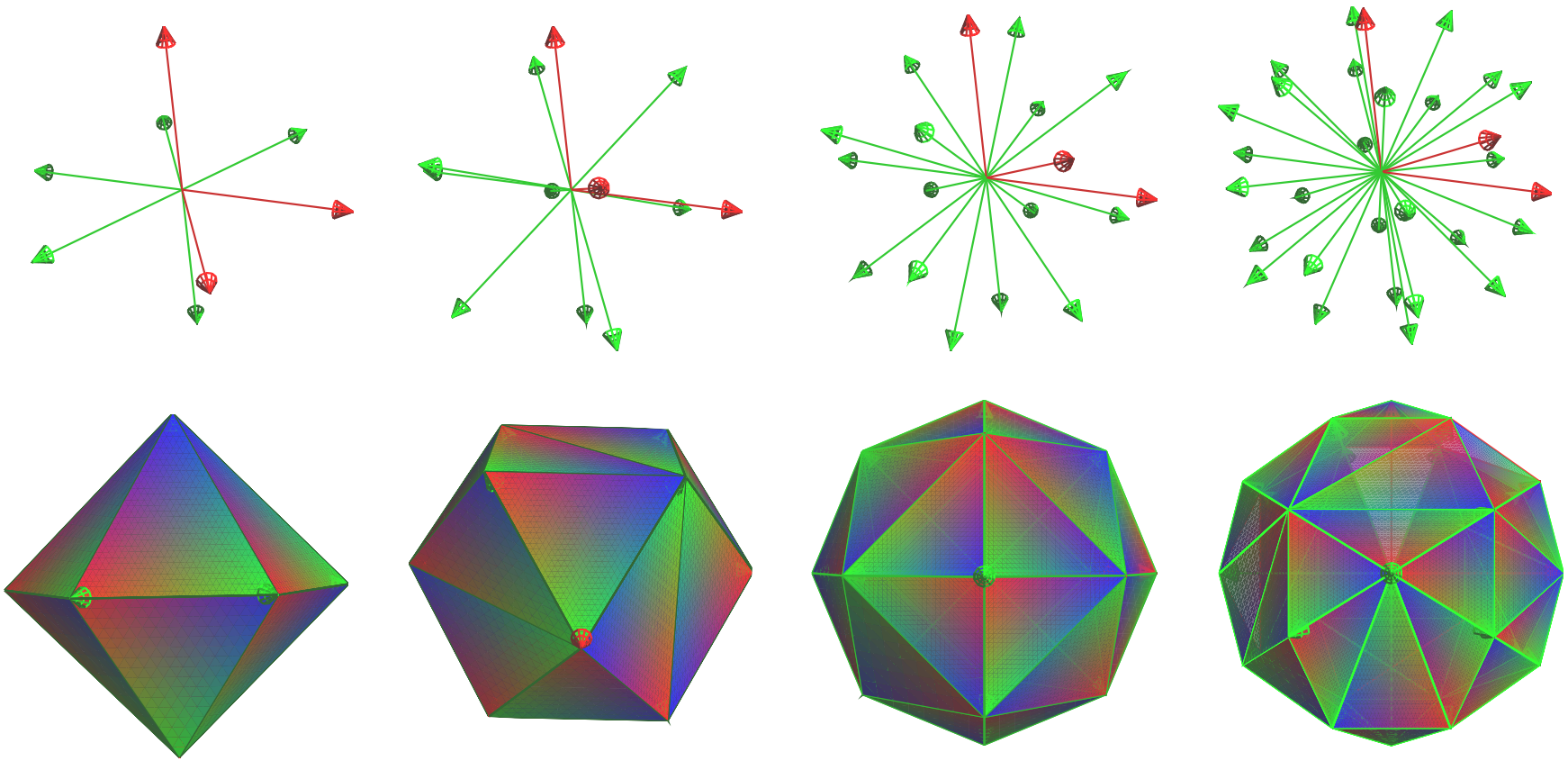
$$\Pi = a \wedge b \wedge c$$

Some d lies on what side of the half-space defined by Π?

$$x = d \cdot \Pi$$

x is the distance of d to Π. It's sign tells us its *orientation* relative to Π.

Example: Convex Hulling



Geometric Algebra Extends the Real Number System

Hypercomplexity

- Many **imaginary** numbers (some for spatial transformations, some for defining duality, some embedded in an indefinite metric)
- More generally, a menagerie of different **types of numbers** of mixed dimensionality. Vectors, complex numbers, quaternions, and other geometric elements and spinors all in one algebraic system.
- In GA this more generic number is called a **multivector**.

Round and Flat Multivectors (and more)

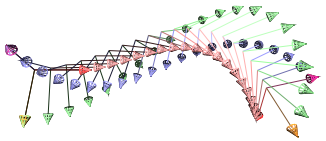
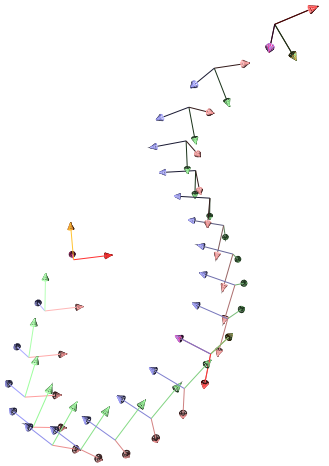
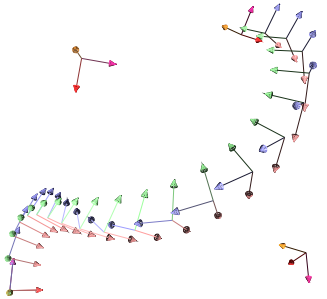
Graphic Symbol	Geometric State	Grade	Algebraic Form	Abbr.
	Point	1	$p = n_o + \boldsymbol{a} + \frac{1}{2}\boldsymbol{a}^2 n_\infty$	`Pnt`
	Point Pair	2	$\tau = p_a \wedge p_b$	`Par`
	Circle	3	$\kappa = p_a \wedge p_b \wedge p_c$	`Cir`
	Sphere	4	$\Sigma = p_a \wedge p_b \wedge p_c \wedge p_d$	`Sph`
	Flat Point	2	$\Phi = p \wedge n_\infty$	`Flp`
	Line	3	$\Lambda = p_a \wedge p_b \wedge n_\infty$	`Lin`
	Dual Line	2	$\lambda = \boldsymbol{B} + \boldsymbol{d} n_\infty$	`Dll`
	Plane	4	$\Pi = p_a \wedge p_b \wedge p_c \wedge n_\infty$	`Pln`
	Dual Plane	1	$\pi = \boldsymbol{v} + \delta n_\infty$	`Dlp`
	Minkowski Plane	2	$E = n_o \wedge n_\infty$	`Mnk`

Basic Rounds and Flats in 5D Conformal Geometric Algebra and their Algebraic Constructions. Bold symbols represent Euclidean elements.

Conformal Versors Completely Represent All Euclidean Transformations

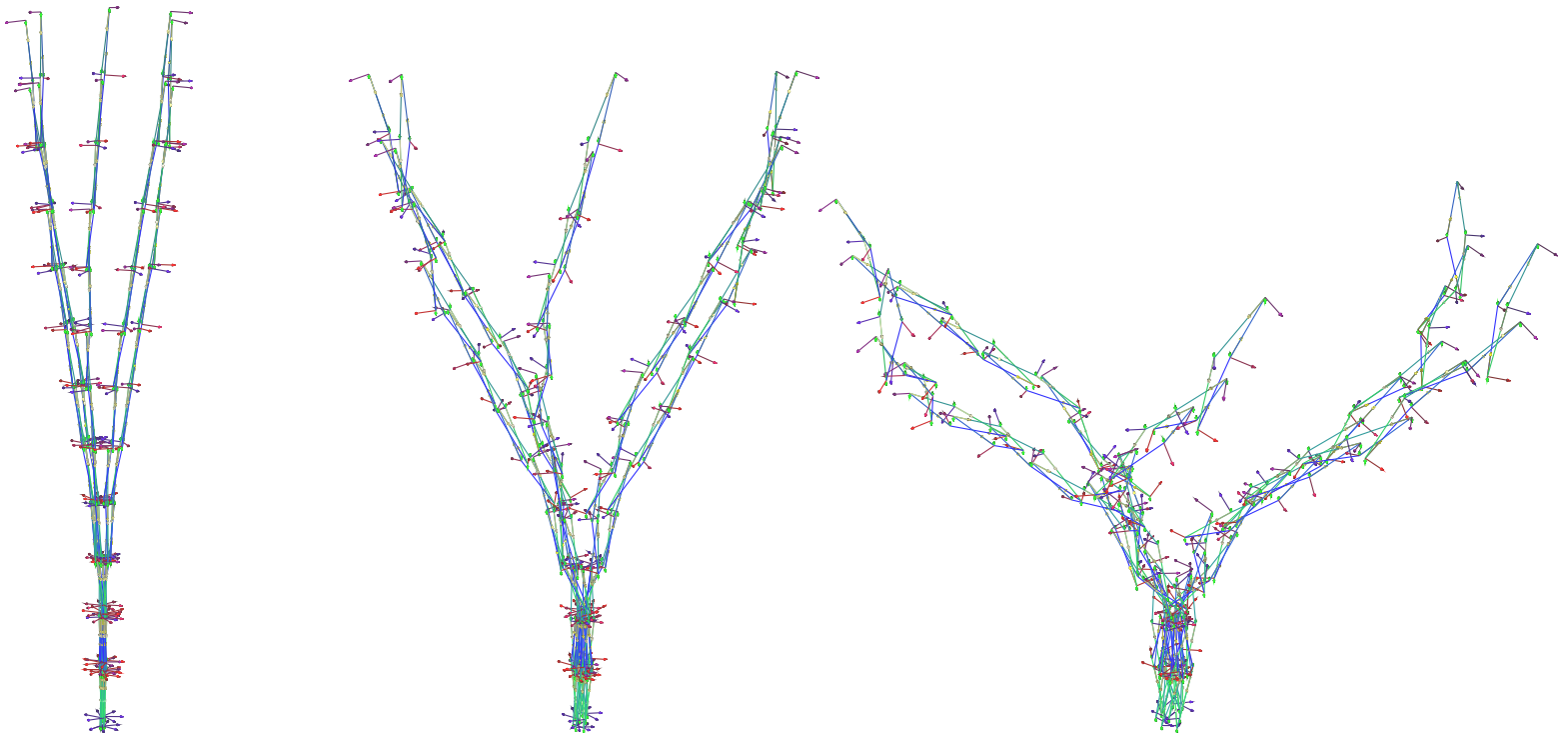
Graphic Symbol	Geometric State	Grade(s)	Algebraic Form	Abbr.
	Rotor	0, 2	$\mathcal{R} = e^{-\frac{\theta}{2}\boldsymbol{I}} = cos\frac{\theta}{2} - sin\frac{\theta}{2}\boldsymbol{I}$	`Rot`
	Translator	0, 2	$\mathcal{T} = e^{\frac{\boldsymbol{d}}{2}\infty} = 1 - \frac{\boldsymbol{d}}{2}\infty$	`Trs`
	Motor	0, 2, 4	$\mathcal{M} = e^{\boldsymbol{B}+\boldsymbol{d}\infty}$	`Mot`
	Dilator	0, 2	$\mathcal{D} = e^{\frac{\lambda}{2}\boldsymbol{E}} = cosh\frac{\lambda}{2} + sinh\frac{\lambda}{2}E$	`Dil`
	Boost	0, 2	$\mathcal{B} = e^{o\boldsymbol{t}} = 1 + o\boldsymbol{t}$	`Trv`

Twists composed of rotations and translations

(a)  (b)  (c) 

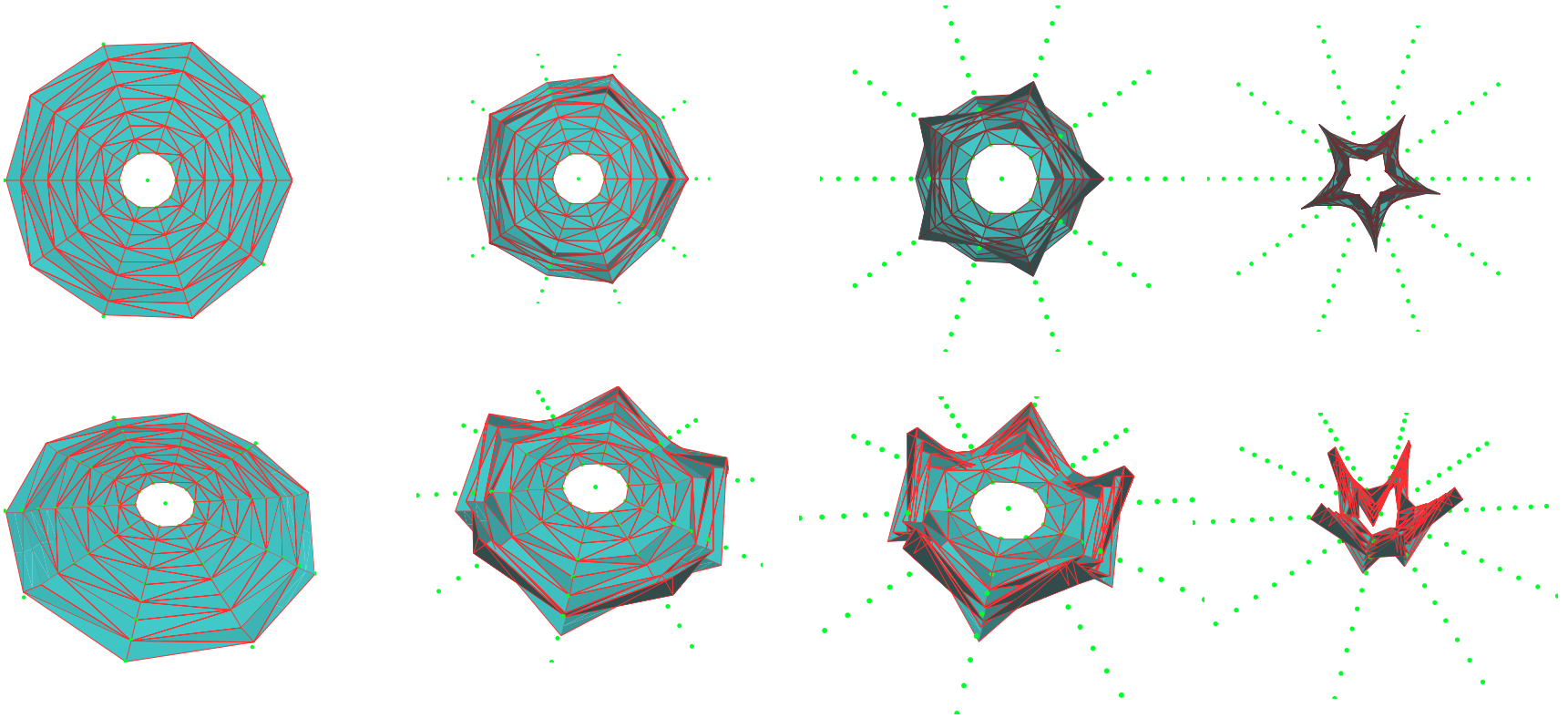
(a) Linear (b) Quadric and (c) Cubic interpolations of motor-generating twists.

Complex Articulations Without Trigonometry



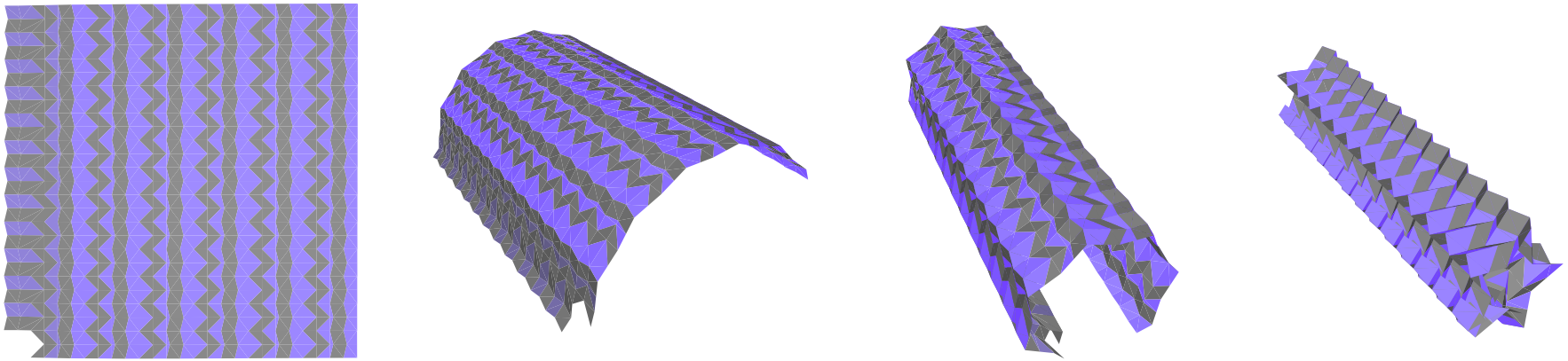
Single degree-of-freedom network of bennet linkages

Complex Articulations Without Trigonometry



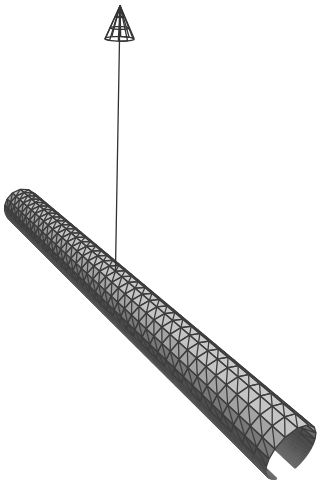
Single degree-of-freedom radial constraint mechanism

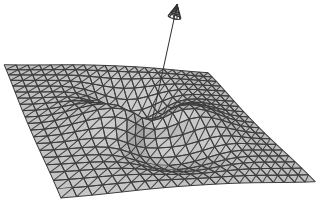
Complex Articulations Without Trigonometry



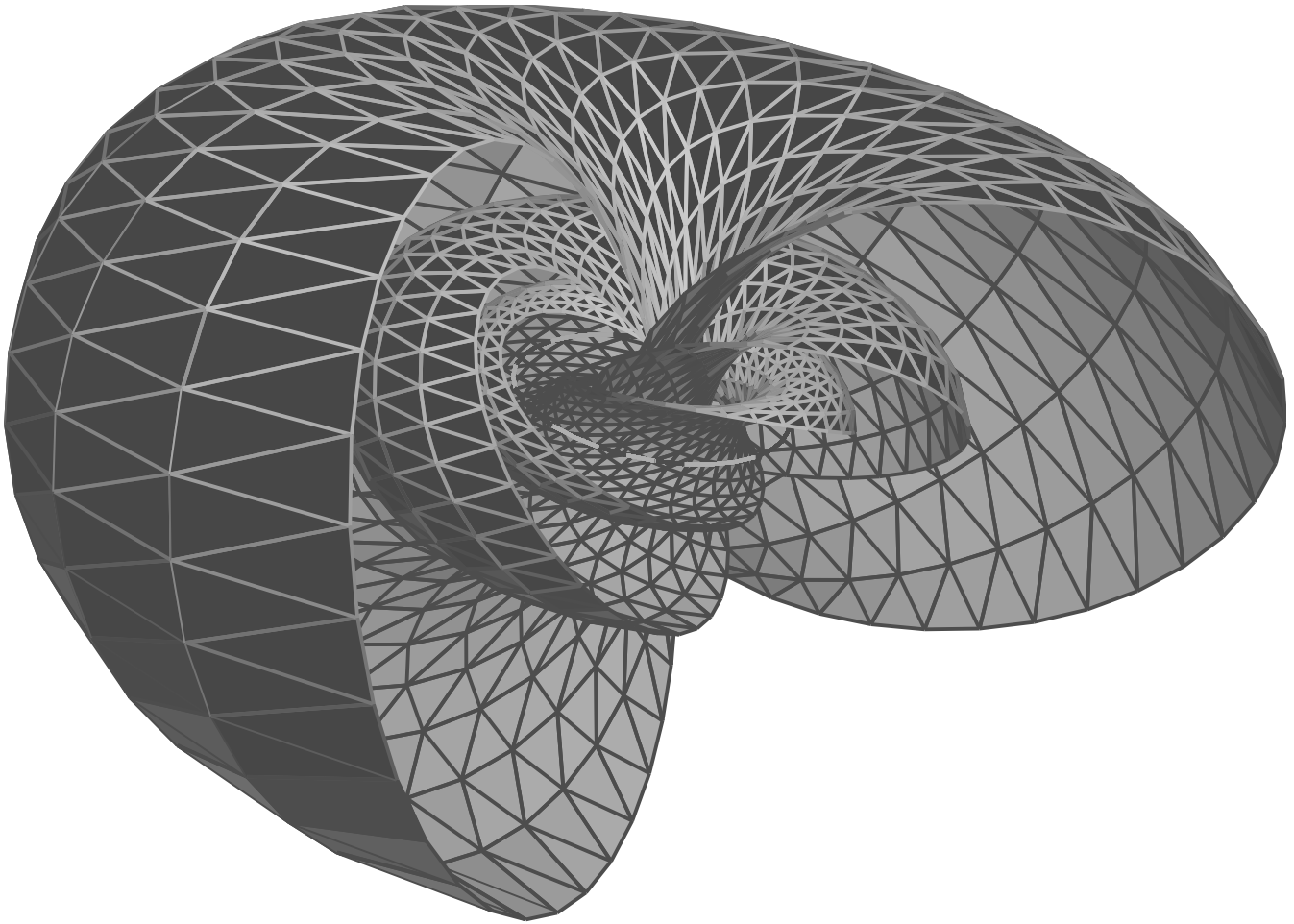
Cylindrical variation of the Miuri-Ori Fold

Bends composed of reflections in spheres



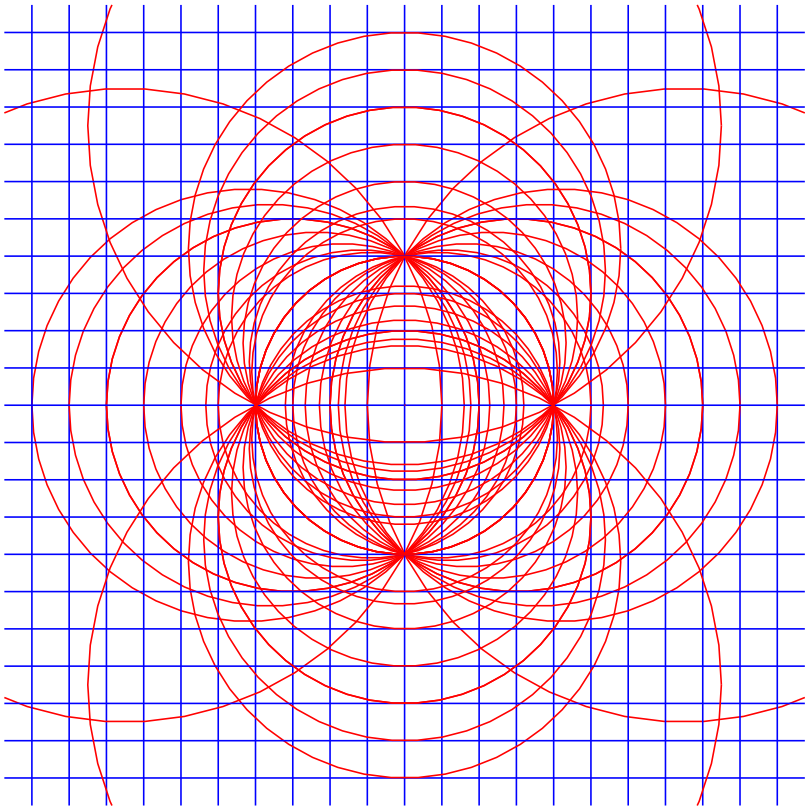


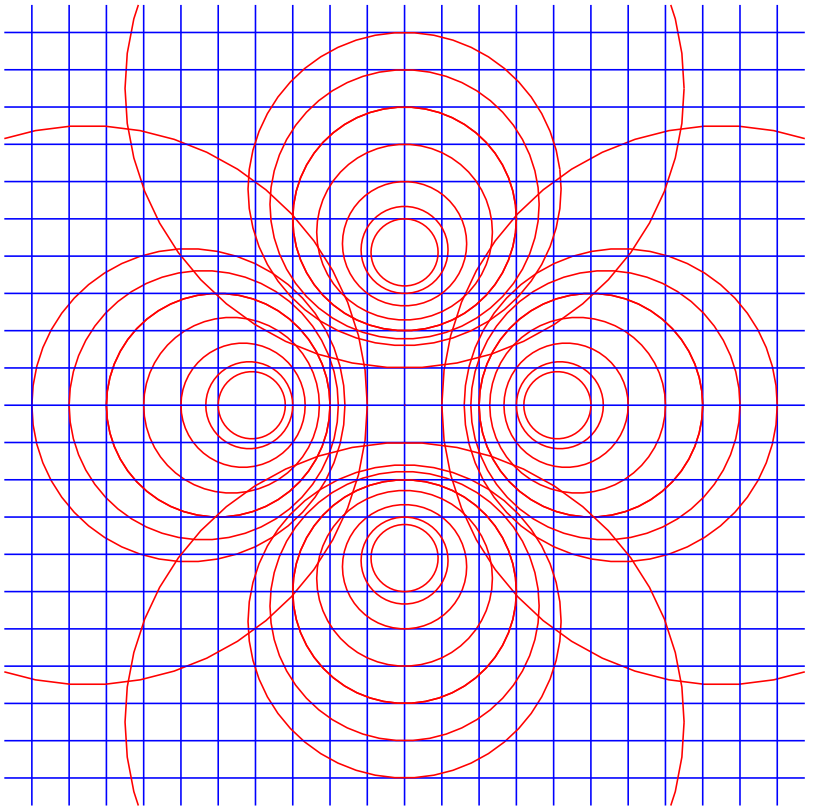
Synthesis of Non-Trivial Topologies



Hopf Fibration

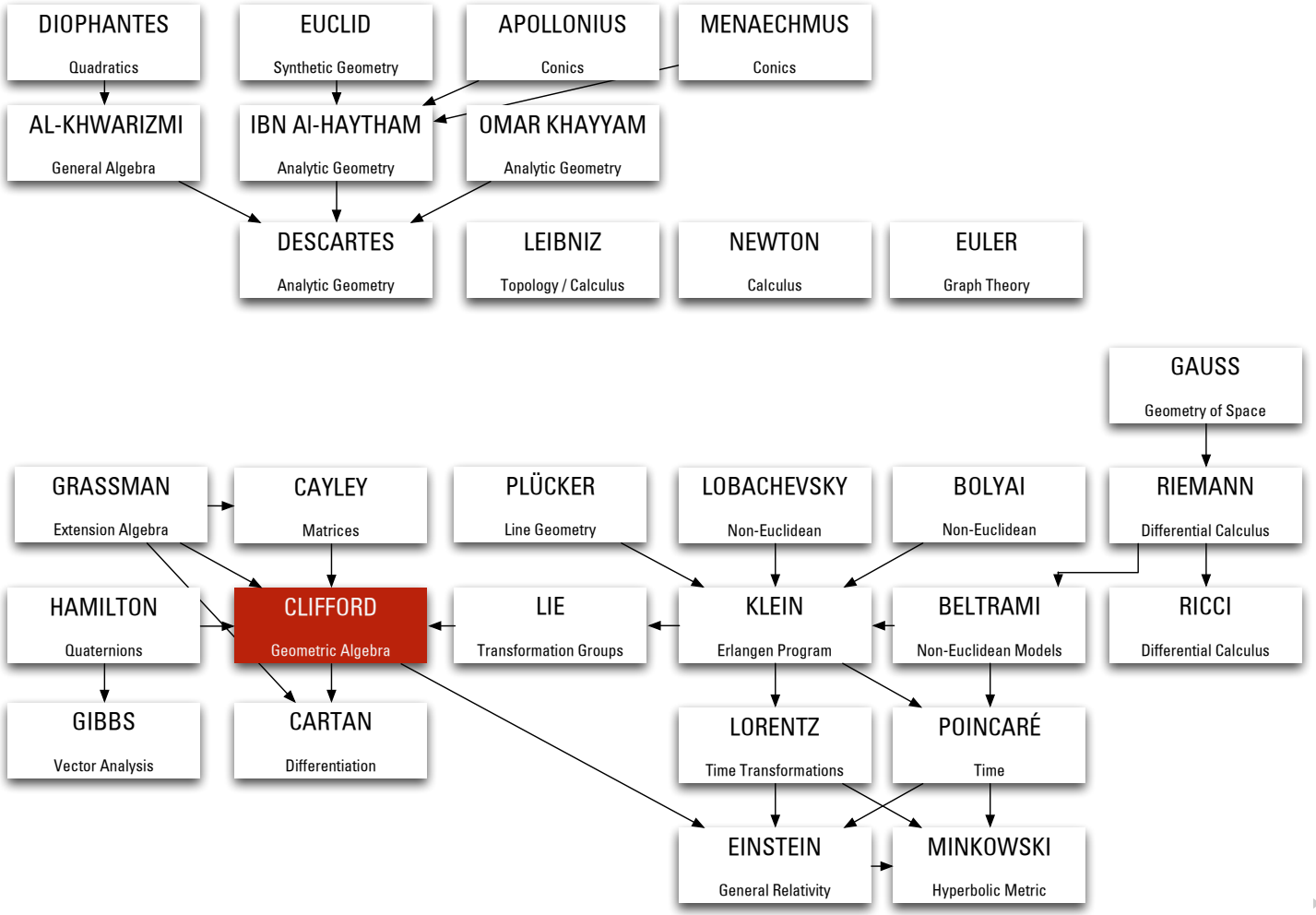
Non-Euclidean Metrics





Spherical and Hyperbolic Spaces

A History of Extension and Abstraction



David Hestenes Resurrects Geometric Algebra in the 60s

- Multivector Calculus
- Spacetime Algebra
- New Foundations for Classical Mechanics
- Geometric Algebra to Geometric Calculus

See Also:

- G. Sobczyk
- A. Lasenby
- E. Bayro-Corrochano
- C. Doran
- C. Perwass
- L. Dorst, S. Mann, D. Fontijne
- J. Lasenby

Primary Contributing Research Centers

- Geometric Algebra Research Group (Cambridge)
- Cognitive Systems (Kiel)
- CINVESTAV (Guadalajara),
- Intelligent Systems Lab (Amsterdam)
- Interactive Graphics Systems Group (Darmstadt)
- Geometric Modeling and Scientific Visualization Research Center (Thuwal)

Implementing GA is Challenging

1. Specialization of Types
2. Combination of Types
3. Arbitrary Metrics and Dimensions
4. $O(n^n)$ Explosive Complexity

D. Hildenbrand, D. Fontijne, C. Perwass and L. Dorst, *Geometric Algebra and its Application to Computer Graphics*, 25th Annual Conference of the European Association for Computer Graphics, EUROGRAPHICS 2004.

Need a system for efficient evaluation of unknown return types in any dimension!

Generative Template Metaprogramming in C++11

- **constexpr** to evaluate basis elements
- **recursion** to process lists of basis elements
- **parameter packs** to template types on basis lists
- **variadic templates** to build types and execution lists
- **partial template specialization** to enable conditional branching
- **auto** return types to alleviate burden of unknown
- **template aliases** to clean up user code

Example: Maybe Types

A Basic Conditional Type Generator

```
1  template<bool, class A, class B>
2  struct Maybe{
3    typedef A Type;
4  };
5
6  template<class A, class B>
7  struct Maybe<false,A,B>{
8    typedef B Type;
9  };
10 /*usage*/
11 typename Maybe<someCompileTimeCheck(),
12   TypeA, TypeB>::Type();
```

Previous Work

- **Gaalet** is a expressions template library for lazy and efficient evaluation of non-degenerate metrics.
- **Clucalc** a C library (evaluated and "graded" at runtime)
- **Gaalop** an optimizer of CluScripts using a precompiler
- **Gaigen** an implementation generator

VERSOR is a Templated Spatial Logic Library

- **Versor** enables nested (tensor) product spaces and a conformal (degenerate) metric, in addition to Euclidean and other diagonalized metrics. (Currently does not use expression templates, but could...)

Versor Usage (Euclidean)

```
1 typedef NEVec<3> Vec;   // 3D Euclidean Vector
2 Vec a(1,.3,.2); Vec b(.3,1,.5);
3 auto sca = a <= b;      // Inner Product
4 auto biv = a ^ b;       // Outer Product
5 auto rot = a * b;       // Geometric Product
```

VERSOR is a Templated Spatial Logic Library

Versor Usage (Conformal)

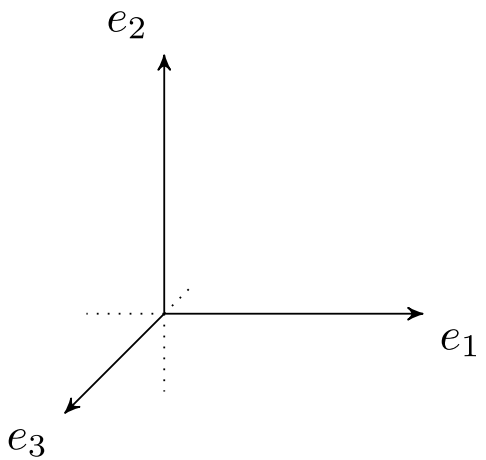
```
1  // Null points (5D conformal)
2  auto a = Vec(1,0,0).null();
3  auto b = Vec(0,1,0).null();
4  auto c = Vec(0,-1,0).null();
5  // Pair of Points
6  auto pair = a ^ b;
7  // Circle through 3 points
8  auto circle = a ^ b ^ c;
9  // Intersection of circle and xz plane
10 auto meet = (circle.dual() ^ Dlp(0,1,0)).dual();
```

Variables in Geometric Algebra Represent Geometric Concepts

Geometric Concept	Algebraic Representation	Dimension
Magnitude	Scalar	0
Direction	Vector	1
Area	Bivector	2
Volume	Trivector	3

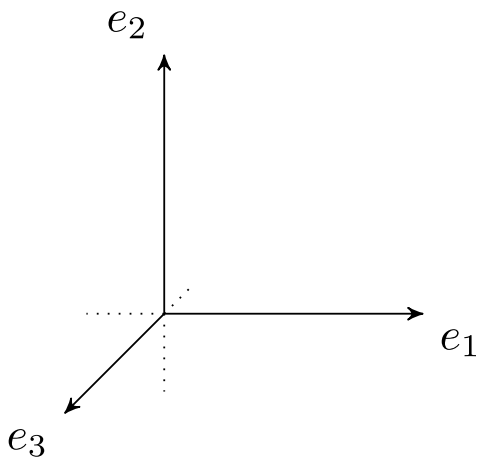
Basic Elements of Geometric Algebra in G^3

The **Basis** frame of G^3 spans the vector space V^3



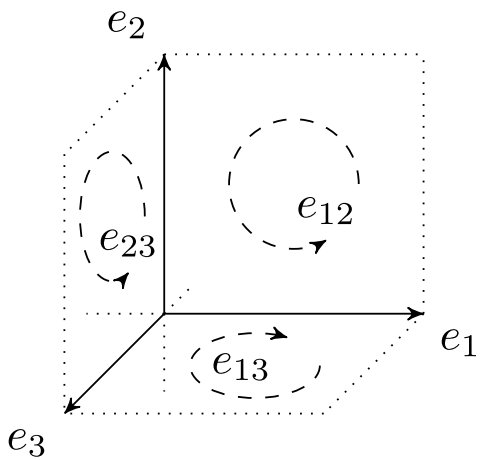
Basis blades e_1,e_2, and e_3 in G^3 represent directed magnitudes (e.g. x, y and z).

Vectors are Directed Magnitudes



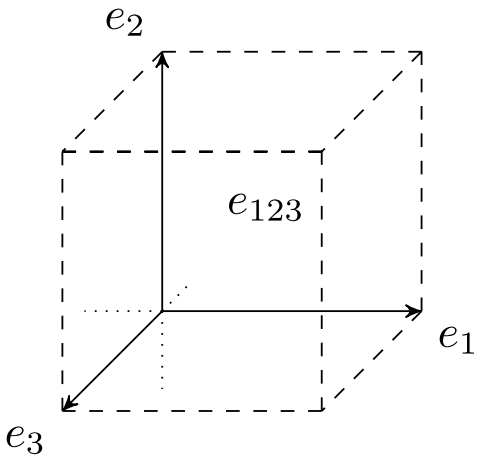
Linear combinations of these basis blades define a vector : $\mathbf{v} = \alpha e_1 + \beta e_2 + \gamma e_3$.

Bivectors are Directed Areas



Basis 2-blades e_{12},e_{13}, and e_{23} in G^3 represent directed unit areas. Linear combinations of these basis blades define a plane or bivector : $\mathbf{B} = \boldsymbol{v}_a \wedge \boldsymbol{v}_b = \alpha e_{12} + \beta e_{13} + \gamma e_{23}$.

The Pseudoscalar is a Directed Volume



The basis trivector e_{123} in G^3 is also known as the pseudoscalar I. As the highest grade blade I is sometimes referred to as the *tangent space*.

$$I = \bigwedge_{i=1}^{n} e_i = e_1 \wedge e_2 \wedge e_3 = e_{123} \tag{1}$$

The metric of the space is embedded axiomatically

The **inner product** of basis blades (in a Euclidean Metric) :

$$e_i \cdot e_j = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

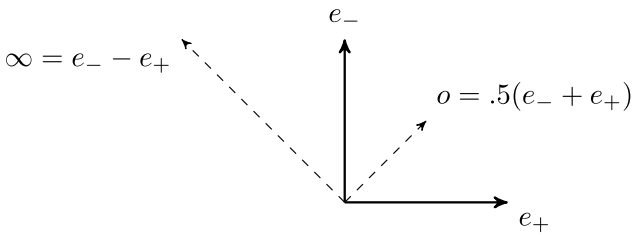
The **outer product** annhilates if the blades are the same :

$$e_i \wedge e_j = \begin{cases} e_{ij} & i \neq j \\ 0 & i = j \end{cases}$$

The **geometric product**, unique to GA, is the sum of these two :

$$e_i e_j = e_i \cdot e_j + e_i \wedge e_j = \begin{cases} e_{ij} & i \neq j \\ 1 & i = j \end{cases}$$

Alternative (non-Euclidean) Metrics are possible



A **null basis** for Minkowski space can be defined through linear combinations of e_- and e_+.

$\mathbb{R}^{1,1}$Metric :

·	e_+	e_-
e_+	1	0
e_-	0	-1

Null Metric :

·	n_∞	n_o
n_∞	0	-1
n_o	-1	0

$$n_\infty = e_- - e_+ \quad n_o = .5(e_- + e_+)$$

$$e_+ = n_o - .5n_\infty \quad e_- = n_o + .5n_\infty$$

The Conformal Model: A 5D Null Metric

$\cdot$	n_o	e_1	e_2	e_3	n_∞
n_o	0	0	0	0	-1
e_1	0	1	0	0	0
e_2	0	0	1	0	0
e_3	0	0	0	1	0
n_∞	-1	0	0	0	0

The Geometric Product is Anti-commutative

Anti-commutation powers the orientability of the algebra (really useful for spatial calculations!)

$$e_{ij} = -e_{ji}$$

so, for example:

$$e_{23}e_2 = e_{232} = -e_{223} = -e_3$$

and:

$$e_{12}^2 = e_{12}e_{12} = e_{1212} = -e_{1122} = -1$$

Geometric Product of Vectors is a Multivector

- Geometric primitives are encoded as linear combinations of basis blades.
- Geometric multiplication is done through distribution of terms in the usual way, and combines both inner and outer products.
- The axioms of the algebra's metric determine the dimensionality of the result.

For vectors $\boldsymbol{a} = \alpha_1 e_1 + \alpha_2 e_2 + \alpha_3 e_3$ and $\boldsymbol{b} = \beta_1 e_1 + \beta_2 e_2 + \beta_3 e_3$:

$$\boldsymbol{ab} = \overbrace{\alpha_1\beta_1 + \alpha_2\beta_2 + \alpha_3\beta_3}^{inner\ \ product} + \underbrace{(\alpha_1\beta_2 - \beta_1\alpha_2)e_{12} + (\alpha_1\beta_3 - \beta_1\alpha_3)e_{13} + (\alpha_2\beta_3 - \beta_2\alpha_3)e_{23}}_{outer\ \ product}$$

Geometric Product of Vectors is a Multivector

$$\begin{aligned} \boldsymbol{ab} &= \alpha_1\beta_1 + \alpha_2\beta_2 + \alpha_3\beta_3 \\ &\quad +(\alpha_1\beta_2 - \beta_1\alpha_2)e_{12} + (\alpha_1\beta_3 - \beta_1\alpha_3)e_{13} + (\alpha_2\beta_3 - \beta_2\alpha_3)e_{23} \end{aligned}$$

Vector Multiplication in G^3

```
1  Rotor gp(const Vec& a, const Vec& b){
2    return Rotor(
3      a[0]*b[0] + a[1]*b[1] + a[2]*b[2], /*s*/
4      a[0]*b[1] - a[1]*b[0],             /*e12*/
5      a[0]*b[2] - a[2]*b[0],             /*e13*/
6      a[1]*b[2] - a[2]*b[1]              /*e23*/
7    )
8  }
```

To efficiently evaluate the product ab . . .

- The very instructions for evaluating the expression must themselves be evaluated at compile time.
- The return type (multivector) created by the expression is to be evaluated at compile time.
- We need to calculate basis blades, combine like terms, then eliminate zero terms, all at compile time.

And of course we must account for different types of a and b (more than just vectors) in arbitrary dimensions.

Basis Blades Bit Representation (Fontijne)

blade	bits	grade
α	000	0
e_1	001	1
e_2	010	1
e_3	100	1
e_{12}	011	2
e_{13}	101	2
e_{23}	110	2
e_{123}	111	3

Binary Representation of Basis Blades in G^3

Grade of blade a is the number of "on" bits

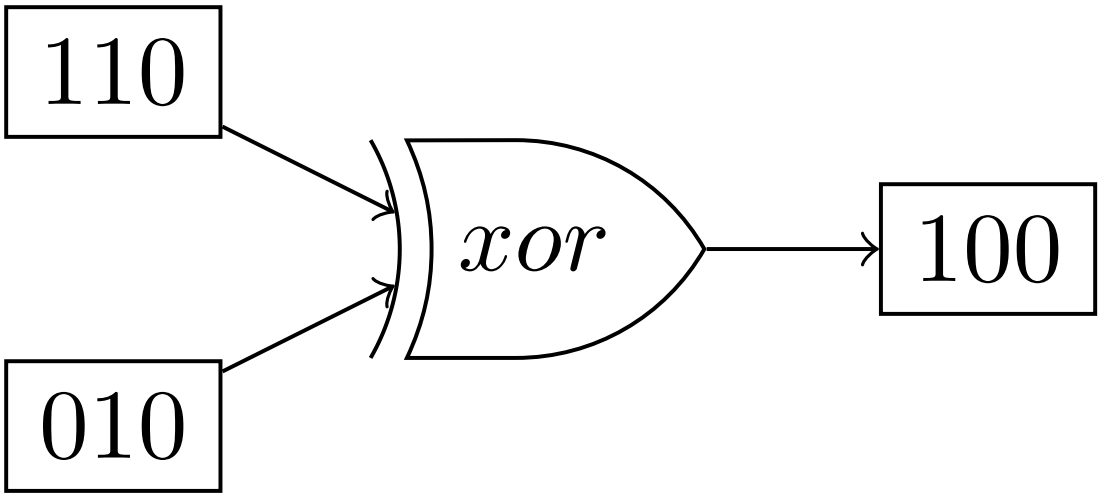
```
function grade(a, c = 0)
    if a > 0 then
        if a&1 then
            return grade(a ≫ 1, c + 1)        ▷ ≫:bitshift right
        else
            return grade(a ≫ 1,c)
    else
        return c
```

Grade of blade *a* is the number of "on" bits

Implementation with constant expressions

```
1 constexpr int grade (int a, int c = 0){
2   return a>0 ?
3     ( a&1 ?
4         grade( a>>1, c+1 )
5       : grade( a>>1, c )
6     )
7     : c;
8 }
```

Geometric Product is an XOR



$$e_{23} * e_2 = -e_3$$

Check for a sign flip when multiplying blades a and b

```
function sign(a,b,c = 0)
    shift ← a ≫ 1
    if shift > 0 then
        return sign(shift, b, c+grade(shift&b))
    else
        if c&1 then
            return true
        else
            return false
```

Check for a sign flip when multiplying blades a and b

Implementation with constant expressions

```
1 constexpr bool sign(int a, int b, int c=0){
2     return (a>>1) > 0 ?
3         (sign(a>>1, b, c + grade((a>>1) & b ))) :
4         ((c & 1) ? true : false);
5 }
```

Check for Outer Product

function outer(a,b)
    return **not** $a\&b$

Implementation with constant expressions

```
1 constexpr bool outer(int a, int b) {
2   return !(a&b);
3 }
```

Check for Inner Product

function inner(a,b)
  $ga \leftarrow \text{grade}(a)$, $gb \leftarrow \text{grade}(b)$, $gab \leftarrow \text{grade}(a \wedge b)$
  if $ga > gb$ **or** $gab \neq gb - ga$ **then**
    return false
  else
    return true

Implementation with constant expressions

```
1 constexpr bool inner(int a, int b) {
2   return !(
3     (grade(a)>grade(b)) ||
4     (grade(a^b) != (grade(b) - grade(a)))
5   );
6 }
```

Comparison for sorting

function compare(a,b)
    $ga \leftarrow$ grade(a)
    $gb \leftarrow$ grade(b)
    if $a == b$ **then**
        return $a < b$
    else
        return $ga < gb$

Multivector Type Generation

A *n*-parameter multivector variable in the algebra is implemented as a C++ type templated on its *n* basis blades. *Multivectors with different basis blades are different types.*

First we define the empty set:

Empty Set Limit Case

```
1 template<int ... XS>
2 struct MV{
3   static const int Num = 0;
4 };
```

Multivector Type Generation

Recursive Template

```
1  template<int X, int ... XS>
2  struct MV<X, XS...>{
3    /*blades*/
4    static const int HEAD = X;
5    typedef MV<XS...> TAIL;
6    /*coordinates*/
7    static const int Num = sizeof...(XS) + 1;
8    float val[Num];
9    /*constructor*/
10   template<typename...Args>
11   constexpr explicit MV(Args...v) :
12     val{ static_cast<float>(v)...} {}
13 };
```

Concatenating Types

Parameter Packs

```
1  template<class A, class B>
2  struct Cat{
3  typedef MV<> Type;
4  };
5
6  template<int ... XS, int ... YS>
7  struct Cat< MV<XS...>, MV<YS...> > {
8  typedef MV<XS..., YS...> Type;
9  };
```

Combinatoric instructions are generated at compile time

When two multivector types are multiplied, a series of instructions are executed.

Vector Multiplication

```
1  Rotor gp(const Vec& a, const Vec& b){
2    return Rotor(
3      a[0]*b[0] + a[1]*b[1] + a[2]*b[2], /*s*/
4      a[0]*b[1] - a[1]*b[0],             /*e12*/
5      a[0]*b[2] - a[2]*b[0],             /*e13*/
6      a[1]*b[2] - a[2]*b[1]              /*e23*/
7    )
8  }
```

This code is not written out manually, but rather generated by the compiler.

Each Execution Instruction is Tokenized

One instruction:

```
a[0]*b[0]
```

Two instructions, summed together:

```
a[0]*b[1] - a[1]*b[0]
```

Three instructions, summed together:

```
a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
```

Execution Instruction Template

```
1  template<bool sign, int A, int B, int IDXA, int IDXB>
2  struct Instruction{
3    /*Result*/
4    static const int Result = A ^ B;
5    /*Inner/Outer Check*/
6    static const bool IP = inner(A,B);
7    static const bool OP = outer(A,B);
8    /*EXECUTION*/
9    template<class TA, class TB>
10   static constexpr typename TA::ValueType
11   Exec( const TA& a, const TB& b){
12     return a[IDXA] * b[IDXB];
13   }
14 };
```

Execution Instruction Template (Partially Specialized)

```
1  template<int A, int B, int IDXA, int IDXB>
2  struct Instruction<true,A,B,IDXA,IDXB>{
3    /*Result*/
4    static const int Result = A ^ B;
5    /*Inner/Outer Check*/
6    static const bool IP = Bits::inner(A,B);
7    static const bool OP = Bits::outer(A,B);
8    /*EXECUTION*/
9    template<class TA, class TB>
10   static constexpr typename TA::ValueType
11   Exec( const TA& a, const TB& b){
12     return -a[IDXA] * b[IDXB]; //<-- Negative Sign
13   }
14 };
```

Execution of Products (overview)

procedure ExecGP(A,B)

instructionList $\leftarrow$ GP(A,B) ▷ Instructions [CT]

$R \leftarrow$ reduce(*instructionList*) ▷ Return Type [CT]

execList $\leftarrow$ index(*instructionList*,R) ▷ Sort Instructions [CT]

return *execList*(A,B) ▷ Execute product [CT or RT]

CT = Compile Time

RT= Run Time

Execution Lists

```
1  template<typename X, typename ... XS >
2  struct XList<X,XS...>{
3   /*Instructions (or Lists of Instructions)*/
4   typedef X HEAD;
5   typedef XList<XS...> TAIL;
6   /*Recursively execute a List of Instructions*/
7   template<class A, class B>
8   static constexpr float Exec(const A& a, const B& b){
9    return X::Exec(a,b) + TAIL::Exec(a,b);
10  }
11  /*Unpack a List of Execution Instructions*/
12  template<class R, class A, class B>
13  static constexpr R Make(const A& a, const B& b){
14   return R(X::Exec(a,b), XS::Exec(a,b)...);
15  }
16 };
```

Lists generated through distributed product

The product return type and the indexing instructions are calculated at compile-time using the algebra's distributive property:
$(a+b+c)(d+e+f) = a(d+e+f)+b(d+e+f)+c(d+e+f)$

procedure GP(A, B, ax=0, bx=0)
  if A=$\emptyset$ **then**
    return $\emptyset$
  else
    first $\leftarrow$ SubGP(A::HEAD, B, ax, bx)
    rest $\leftarrow$ GP(A::TAIL,B, ax+1, bx)
    return cat(*first*, *rest*)

Lists generated through distributed product

The product return type and the indexing instructions are calculated at compile-time using the algebra's distributive property:
$(a+b+c)(d+e+f) = a(d+e+f)+b(d+e+f)+c(d+e+f)$

```
procedure SubGP(a, B, ax, bx)
    if B=∅ then
        return ∅
    else
        sign ← sign(a, B::HEAD)
        first ← instruction(sign, a, B::HEAD, ax, bx)
        rest ← SubGP(a, B::TAIL, ax, bx+1)
        return cat(first, rest)
```

Check for Membership of *a* in M

```
procedure exists(a,M)
    if M = ∅ then return  false
    else
        if a ==M::HEAD then
            return  true
        else
            return exists(a, M::TAIL)
```

Check for Membership of *a* in M

```
1  template<int a, class M>
2  struct Exists{
3    static constexpr bool Call() {
4     return M::HEAD==a ?
5       true
6       : Exists<a, typename M::TAIL>::Call(); }
7  };
8  /*Limit Case of Empty Set*/
9  template<int a> struct Exists<a, MV<>>{
10   static constexpr bool Call() { return false; }
11 };
```

Recursive Compile-Time List Reduction

procedure reduce(I)

$M \leftarrow$ reduce(I::TAIL) ▷ Start at Limit Case

$b \leftarrow$ exists(I::HEAD, M) ▷ Check for membership

$A \leftarrow$ insert(I::HEAD, M) ▷ Add and Sort

$Type \leftarrow$ maybe(b, M, A) ▷ Conditional Branching

return *Type*

Recursive Compile-Time List Reduction

```
1  template<class L> struct Reduce{
2    /*Recursion*/
3    typedef typename
4    Reduce<typename L::TAIL>::Type M;
5    /*Conditional Branching*/
6    typedef typename Maybe<
7      Exists< L::HEAD, M>::Call() ,
8      M,
9      typename Insert< L::HEAD, M >::Type
10   >::Type Type;
11 };
12 /*Limit Case of Empty List*/
13 template<> struct Reduce<List<>>{
14   typedef MV<> Type;
15 };
```

Compile-Time Insert-Sort item *a* into list R

procedure insert(*a*, R, F = MV<>)
  com ← compare(*a*, R::HEAD)
  catF ← cat(cat(*F*,MV<*a*>), *R*)
  catR ← cat(*F*,MV<R::HEAD>)
  return *Type* ← maybe(*com*, *catF*, insert(*a*, R::TAIL, *catR*)

1. *F* is initially empty.
2. If *a* > R::HEAD, head of list R is popped off and added to end of *F*. Recurse with new *R* and *F*.
3. If *a* < R::HEAD, *a* is added to end of *F* and R is added after. Return concatenated type.

Compile-Time Insert Sort

```
1  template<int A, class R, class First=MV<>>
2  struct Insert{
3    typedef typename Maybe<compare<A, R::HEAD>(),
4      typename Cat<
5          typename Cat<First, MV<A>>::Type,
6          R
7      >::Type,
8      typename Insert<
9          A,
10         typename R::TAIL,
11         typename Cat<First, MV<R::HEAD>>::Type
12       >::Type
13     >::Type Type;
14 };
```

Compile-Time Insert Sort

```
1 /*Limit Case of Empty Set*/
2 template <int A, class First>
3 struct Insert<A, MV<>, First>{
4   typedef typename Cat<First, MV<A>>::Type Type;
5 };
```

Gather all instructions that result in N

procedure find(N,A)
  if A=$\emptyset$ **then**
    return $\emptyset$
  else
    $next \leftarrow$ find(N,A::TAIL)
    $cat \leftarrow$ cat(A::HEAD, $next$)
    $list \leftarrow$ maybe(N ==A::HEAD::Result, cat, $next$)
    return $list$

Gather all instructions that result in N

```
1  template<int N, class A>
2  struct FindAll {
3   /*Start at limit case*/
4   typedef typename
5   FindAll<N, typename A::TAIL>::Type Rest;
6   /*Add to list or continue*/
7   typedef typename Maybe<A::HEAD::Result == N,
8    typename XCat<XList<typename A::HEAD>, Rest>::Type,
9    Rest
10  >::Type Type;
11 };
12 /*Limit Case of Empty List*/
13 template<int N> struct FindAll<N, XList<>>{
14  typedef XList<> Type;
15 };
```

Gather a list of execution instruction lists

- A return type has an array of values.
- Each value is calculated using a list of instructions for how to combine the arguments.

procedure index(I,R)
    if R=$\emptyset$ **then**
        return $\emptyset$
    else
        one $\leftarrow$ find(R::HEAD,I)
        list $\leftarrow$ cat(*one*, index(I,R::TAIL))
        return *list*

Gather a list of execution instruction lists

```
1  template< class I, class R >
2  struct Index{
3    /*Pop off head and find all matching instructions*/
4    typedef typename FindAll<R::HEAD, I>::Type One;
5    /*Add to the rest*/
6    typedef typename XCat<
7      XList< One > ,
8      typename Index < I, typename R::TAIL >::Type
9    >::Type Type;
10 };
11 /*Limit Case of Empty List*/
12 template< class I> struct Index< I, MV<> > {
13   typedef XList<> Type;
14 };
```

The Product Templated on its Arguments

```
1  template<class A, class B>
2  struct Product{
3    //Instruction List
4    typedef typename GP<A,B>::Type InstList;
5    //Return Type
6    typedef typename Reduce<InstList>::Type Type;
7    //Sort Instructions using Return Type
8    typedef typename Index< InstList, Type>::Type DO;
9    //Execute function
10   constexpr Type exec(const A& a, const B& b) const{
11     return DO::template Make<Type>(a, b);
12   }
13 };
```

Inner and Outer products of A and B

- Instructions Assembled Similar to Geometric Product
- Additional Check Controls whether Resulting Blade is Added to Return Type

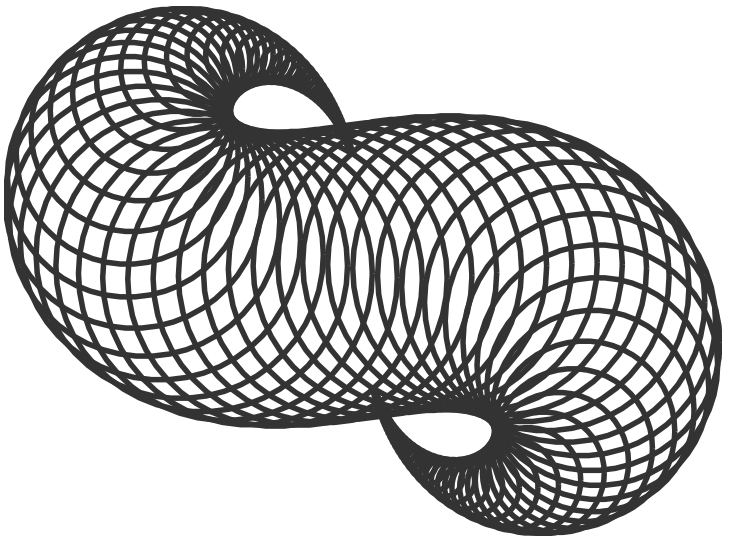
Conformal metric

- Additional Requirements to Split Degenerate Metric into Minkowskian Metric
- Compile time check for presence of n_o or n_∞ blades, convert into appropriate sums of e_- and e_+

Challenges

- Compilation times can take a long time for higher dimensions (> 10), though nesting is possible.
  - e.g. $\mathbb{R}^{16} = \mathbb{R}^8 \otimes \mathbb{R}^2$
- Some generic superclass would be useful for runtime evolutionary algorithms.

Thanks!



With support from the Robert W. Deutsch Foundation