Geometric Algebra

Overview

This Geometric Algebra (GA) package performs GA operations in n-dimensions for any n. The package can be configured for either Grassmann or Clifford algebras. The latter can be configured to use either the +--- or -+++ convention, and can be configured for space or space-time (with additional time dimensions possible for those who wish to explore that.) This package uses everyday standard (i.e., subscript) notation and uses the symbols e_1, e_2, ... for its orthonormal basis.

The package has a palette to simplify entering of multivectors so that entering subscripts is made easy. The palette also shows all the available functions, discusses them in tooltips (i.e., hover the mouse over a palette entry), and displays examples. All commands can be used with symbols as well as numerical values.

Quick Start describes how to install the palette. To use the commands in this package, put the file GeomAlg2017June.m in one of the directories listed in your $Path. The preferred location is $UserBaseDirectory. After that, to use the package simple enter Needs["GeomAlg2017June`"] at the top of a notebook and then start using any of these commands along with usual Mathematica commands

1 Begin Package

Set Up

In[6514]:=
```
(* Mathematica Package
   :Title:Geometric Alagebra in n-Space
   :Context:GeomAlg2017June`
   :Author:Dr. Bud Simrin
   :Date:2019-08-11

   :Package Version:0.3
   :Mathematica Version:11.0.1
   :Copyright:(c) 2019 Dr. Bud Simrin
   :Keywords: geometric algeba, Clifford algebra, wedge product, dot product, exterior
    product, interior product, Hodge dual, clif, multivector, bivector, rotor, rotation,
    spacetime, n-dimensional, quaternion

   :Discussion:

USAGE:

    SetDirectory[NotebookDirectory[]];
 *)

ClearAll["GeomAlg2017June`*"];

BeginPackage["GeomAlg2017June`"];
SetOptions[$FrontEnd,InputAliases→{"slc"→"⌋","src"→"⌊"}];
```

Debug Flags (True = On, False = Off, Default = False)

In case of trouble, one or more of these flags can be enabled to trace the flow of events. The output can be prodigious so try to only enable the ones you might need. This capability is coded. It is not the built-in Mathematica debug structure.

In[6517]:=
```
debug1;  (* Initialization, both section 3 and section 5 *)
debug2;  (* MaxDimG *)
debug3;  (* GeomPrdtG *)
debug4;  (* GradePpieceG *)
debug5;  (* ClifFormatG *)
debug6;  (* WedgePrdtG, DotPrdtG, LeftContractionG, RightContractionG *)
debug7;  (* ReverseG *)
debug8;  (* ScalarPrdtG *)
debug9;  (* FreeTermG, ConstantG, ClifToListG *)
```

Initialization Variables

algebraType

1 Clifford algebra

2 Grassmann algebra i.e., $e_k{}^2 = 0$ for all k

The following are only relevant for Clifford algebra:

signatureType

-1 Physicists + - - - i.e., $e_k{}^2 = +1$ if $k \leq$ numTimelike; else $e_k{}^2 = -1$

+1 Mathematicians - + + + i.e., $e_k{}^2 = -1$ if $k \leq$ numTimelike; else $e_k{}^2 = +1$

numTimelike

0 Space

1 Space-time

2+ More than 1 time dimension, allowed for those who wish to explore this

These are global symbols (i.e., context Global`) so they are not listed in the Usage section (e.g., context GeomAlg2017June`)

If user forgets to initialize these variables, a warning will be issued and defaults will be implemented:

Defaults are: Clifford, Mathematician ++++, Space

Caution: If user has set Notebook to use a unique context, then only execute initialization of these variables after package has

been invoked with Needs statement so that their context is already assigned to Global`

Usage and one Warning Statement

In[6526]:=
```
e; (* MUST pass e as a global variable. This causes Context[e] = GeomAlg2017June` in any u
        that invokes this package, and thus operations like eu_→1 work correctly *)

BladeG::usage="BladeG[p,q,...,r] generates the blade epeq...er";
BiVectorG::usage="BiVectorG[a,n] generates an n-dimensional bivector";
ClifFormatG;
ClifToListG::usage="ClifToListG[x] converts multivector x into a list of its components";
CollectG::usage="CollectG[x] groups the terms of multivector x by blades eiej...ek. It also
ComplexG::usage="ComplexG[a,b] generates a complex number a + b i, where i is the dimensio
ConstantG::usage="ConstantG[x] picks out the constant term(s) of multivector x";
ConstantToListZeroG;
ConstantToZeroG;
DotPrdtG::usage="DotPrdtG[x,y,...,z] computes dot product of several multivectors";
EijTermG::usage="EijTermG[x,e1e3e4] returns the components of x, if any, having e1e2e4 as a
eSubscriptListG::usage="eSubscriptListG[x] computes a list of terms ei,j,... corresponding 
EvenClifG::usage="EvenClifG[b,n] generates an n-dimensional multivector having only even-g
ExpandG::usage="ExpandG[x] expands multivector x, including reduction of possible lingerin
FreeTermG::usage="FreeTermG[x,c] computes clif, x, minus constant term, c";
GeomPrdtBladeG;
GeomPrdtG::usage="GeomPrdtG[x,y,...,z] computes the geometric product of several multivect
GormG::usage="GormG[x] computes the gorm, xReverse · x, of multivector x";
GradeListG::usage="GradeListG[x] generates a list of grades matching the components of mu
GradePTermG::usage="GradePTerm[x,p] finds the components of multivector x that are of grad
HodgeDualG::usage="HodgeDualG[x,n] generates the Hodge Dual, x ∘ i, of multivector x, wher
HodgeDual2G::usage="HodgeDual2G[x,n] generates an alternative Hodge Dual, x ∘ i^-1, of mult
InitializeG::usage="InitializeG[x] reduces any ei^2 terms in multivector x according to use
InitializeG::warning="Warning: Implementing initialization defaults. Use Palette if wish t
InverseG::usage="InverseG[x] computes the inverse, if it exists, of multivector x";
LeftContractionG::usage="LeftContractionG[x,y,...,z] computes the left contraction of seve
ListToClifG::usage="ListToclifG[xList] converts a list into its corresponding multivector"
MaxDimG::usage="MaxDimG[x] finds the maximum dimension among blades in the multivector";
nClifG::usage="nClifG[a,n] generates a general symbolic multivector of dimension n with co
NormG::usage="Norm[x] generates the norm of a multivector x when it exists";
nVectorG::usage="nVectorG[a,n] generates a general (1-dimensional) vector of dimension n w
pBladeG::usage="pBladeG[a,p,n] outputs a general blade of grade p in n-space having coeff
PrdtG;
PseudoScalarG::usage="PseudoScalarG[n] generates the dimension n positive unit pseudoscala
QuaternionG::usage="QuaternionG[a,b,c,d]";
ReverseG::usage="ReverseG[x] generates the reverse of multivector x. That is, it changes s
RightContractionG::usage="RightContractionG[x,y,...,z] computes the right contraction of s
RotorG::usage="RotorG[m,n,θ} generates a rotor that spins the m n-axis counter-clockwise b
ScalarPrdtG::usage="ScalarPrdtG[x,y,...,z] computes the scalar product of several multivec
SignatureG::usage="SignatureG[list] computes the signature of a list; i.e., ±1 depending u
SubscriptListG::usage="SubscriptListG[x] generates a list of subscripts of the terms in mu
WedgePrdtG::usage="WedgeProductG[x,y,...,z] computes the wedge product of multivectors x,
```

2 Begin Private (i.e., define functions)

In[6569]:=
```
Begin["`Private`"];
```

3 Operator Symbols

In[6570]:=
```
SmallCircle:=GeomPrdtG  (* Enter clifA Esc sc Esc clifB: clifA ∘ clifB *)
Wedge:=WedgePrdtG  (* Enter clifA Esc ^ Esc clifB: clifA ∧ clifB *)
CenterDot:=DotPrdtG  (* Enter Esc . Esc: clifA · clifB  *)
(* In Mathematica, can't use Right Floor, ⌋ as a binary operator (i.e., infix). So use :
CircleDot:=ScalarPrdtG  (* Enter Esc c. Esc: clifA ⊙ clifB  *)
SquareSuperset:=LeftContractionG  (* Enter Esc slc Esc  *)
SquareSubset:=RightContractionG  (* Enter Esc src Esc  *)

(* Example: wedge product of 3 vectors: vector1 ^ vector2 ^ vector3  *)
(* Future: Add unary operators (i.e., postfix)
        Consider SuperStar (unfortunately considered a power I believe) or
        OverHat for Hodge Dual and OverTilde for Reverse  *)
```

4 Typing Aids: Multivector Generators

Vectors, Blades, Multivectors, Rotors, Complex Numbers, Quaternions

In[6576]:=
```
ModifyContextPathG:=Module[{len,ContextPath},len=Length[$ContextPath];
ContextPath=Permute[$ContextPath,Cycles[{{1,len}}]]] (* Move pkg context to end of path *)


(* Together the BladeG definitions below enable expressions like Blade[u,v,w,x] = eueveWex
BladeG[eu_]:=eu
BladeG[u_]:=eu
BladeG[eu_,v_]:=euBladeG[ev]
    (* These blades are simple products of basis vectors, ei.
    In general, blades can be from any independent vectors, but basis elements are conven
BladeG[u_,v__]:=BladeG[u]BladeG[v] (* Ex: BladeG[1,2,3,4] = e1e2e3e4  *)

    (*End of BladeG Module*)

pBladeG[c_,0,n_]:=c0
pBladeG[c_,p_,n_]:=
```

```
    (* Note: this is a generic homogeneous clif of grade p in n-Space.
        Ex: pBladeG[a,2,4] = e_1e_2a_{1,2} + e_1e_3 + e_1e_4a_{1,4} + e_2e_3a_{2,3} + e_2e_4a_{2,4} + e_3e_4a_{3,4} *)
    Module[{pBlade,tupleList,cList,eList},
        Cases[SubValues[Subscript],dv_/;FreeQ[dv,c]];DownValues[Subscript];
        (* Clear all c-subscripted variables *)

        pBlade=0;
        If[p≤n,
            tupleList=Subsets[Range[n],{p}];
                (* {{1,2},{1,3},{2,3}}  *)
            cList=Subscript[c,##]&@@@Subsets[Range[n],{p}];
                (* {c_{1,2},c_{1,3},c_{2,3}}  *)
            eList=Product[Subscript[e,i],{i,{##}}]&@@@Subsets[Range[n],{p}];
                (* {e_1e_2, e_1e_3, e_2e_3}   *)
            pBlade=cList.eList
                (* e_1e_2c_{1,2} + e_1e_3c_{1,3} + e_2e_3c_{2,3}  *)
        ,
            Print["Error: pBladeG requires p ≤ n."]
        ];
        pBlade
    ]       (* End of pBladeG Module *)


EvenClifG[b_,0]:=c_0
EvenClifG[b_,1]:=c_0

EvenClifG[b_,n_]:= (* Create an n-dimensional Clif using user-provided base "c"
INPUTS:
    b - Coefficient letter to use in Clif
    n - Number of desired dimension (i.e., the maximum grade) of the clif

PROCESS:
    Cases command clears out any possible values or definition for the base b
    eArray generates a list of all possible e_ie_j... products of grade ≤ n
        Example: {1, e_1, e_2, e_1e_2}
        The subset formula fails for grades 0 and 1 so those are handled by brute force
    bList generates a matching of b_{i,j,...}coefficients. Example: {b_0, b_1, b_2, b_{1,2}}
    Since the constant b_0 does not change sign, we modify it to be b_{0,0} in bLis1
        Example: {b_{0,0}, b_1, b_2, b_{1,2}}
    Then we make a list of the lengths of terms b_{i,j,...} in bList 1. Example: {3, 2, 2, 3}
    We replace the odd lengths (like b_{1,2}) by 1 and the even lengths (like b_3) by 0
        We name this list evenList. Example: {1, 0, 0, 1}
    The desired clif is the dot product of evenList with the result of the product of
        bList and eArray.
        Example:
            {1, 0, 0, 1} . [ {b_0, b_1, b_2, b_{1,2}} {1, e_1, e_2, e_1e_2} ]
```

```
            = {1, 0, 0, 1} . {b₀, b₁e₁, b₂e₂, e₁e₂b₁,₂}
            = b₀ + e₁ e₂ b₁,₂

OUTPUT:
    An array of even-grade members, like
        b₀ + e₁e₂b₁,₂ + e₁e₃b₁,₃ + e₁e₄b₁,₄ + e₂e₃b₂,₃  + e₂e₄b₂,₄ + e₃e₄b₃,₄ +  e₁e₂e₃e₄b₁,₂,₃,₄  *)

    Module[{bList,eArray,bList1,evenList,evenClif},

        Cases[SubValues[Subscript],dv_/;FreeQ[dv,b]];
        DownValues[Subscript];
        (* Clear all b-subscripted variables *)

        eArray=Subsets[∏ᵢⁿ eᵢ, n];  (* List of all eᵢeⱼ... products *)
        bList=Flatten[Append[{b₀},Subscript[b,##]&@@@Rest[Subsets[Range[n]]]]];
        If[bList[[1]]==b₀,bList1=ReplacePart[bList,1→b₀,₀],bList1=bList];
        evenList=Length/@bList1/.{u_/;OddQ[u]→1,u_/;EvenQ[u]→0};
        evenClif=evenList.(bList eArray);
        evenClif
    ]   (* End of EvenClifG Module  *)



nClifG[c_,0]:=c₀
nClifG[c_,1]:=c₀+c₁e₁

nClifG[c_,n_]:= (* Create an n-dimensional Clif using user-provided base "c"
INPUTS:
    c - Coefficient letter to use in Clif, often a for 1st array and b for a second one
    n - Number of desired dimension (i.e., the maximum grade) of the clif

PROCESS:
    Cases command clears out any possible values or definition for the base c
    eArray generates a list of all possible eᵢeⱼ... products of grade ≤ n
        The subset formula fails for grades 0 and 1 so those are handled by brute force
    cList generates a matching of cᵢ,ⱼ,...coefficients
    The desired clif is the dot product of the 2 lists

OUTPUT:
    An array like b₀ + b₁e₁ + b₂e₂ + b₃e₃ + e₁e₂b₁,₂ + e₁e₃b₁,₃ + e₂e₃b₂,₃ + e₁e₂e₃b₁,₂,₃   *)

    Module[{cList,eArray},

        Cases[SubValues[Subscript],dv_/;FreeQ[dv,c]];
            DownValues[Subscript];
```

```
        (* Clear all c-subscripted variables *)

        eArray=Subsets[∏_i^n e_i, n];  (* List of all e_i e_j... products *)

        cList=Flatten[Append[{c_0},Subscript[c,##]&@@@Rest[Subsets[Range[n]]]]];

        cList.eArray
    ]   (* End of nClifG Module  *)

ClifFormatG[n_]:=  (*
INPUT:
    n = largest subscript on any e_i term in clif(s) under consideration for simplifying pr

PROCESS:
    First form the product e_1e_2...e_n. For example, for n = 3 we form e_1e_2e_3
    Then make a list of all subsets of the product. For example, for n = 3, the list is
        {e_1, e_2, e_3, e_1e_2, e_1e_3, e_2e_3, e_1e_2e_3} = Basis for GA[3]
    Finally, Mathematica works better with this list in reverse order
    This function is used by function CollectG and can also be used directly by the user

OUTPUT:
    collectTerms - the list above, reversed   *)

    Module[{collectTerms},
        If[n>1,
            collectTerms=Subsets[∏_i^n e_i,{1,n}]; (* Terms we wish to collect by *)
            collectTerms=Reverse[collectTerms]
                (* Reverse order to force Mathematica to collect terms correctly *)
        ,
            collectTerms={e_1}
        ];
        If[debug5,Print["Arrange List = ",collectTerms]];
        collectTerms
    ]       (*End of ClifFormatG Module*)

ComplexG[a_,b_]:=
    Module[{complexNum},
        i=e_1e_2;complexNum=a+b i
    ]

PseudoScalarG[n_]:=Product[e_i,{i,n}]  (* e_1e_2...e_n *)

QuaternionG[a_,b_,c_,d_]:=
    Module[{quaternionNum},
        i=e_2e_3;j=-e_1e_3;k=e_1e_2;quaternionNum=a+b i+c j+d k;
        quaternionNum
```

```
    ]

RotorG[i_,j_,θ_]:=Cos[θ/2]+Sin[θ/2]eᵢeⱼ
(* Create rotor from rotation angle plane of rotation (denoted by the pair of axis numbers

BiVectorG[c_,n_]:=
    (* Note: bivector is visualized as a 2D region spanned by 2 vectors in a space of
        dimension 2 or higher   *)

    Module[{pairList,cList,eList,bivectorC},
        Cases[SubValues[Subscript],dv_/;FreeQ[dv,c]];DownValues[Subscript];
        (* Clear all c-subscripted variables *)

        pairList=Subsets[Table[i,{i,n}],{2}];
        cList=pairList/.{u_,v_}→c_{u,v};
        eList=pairList/.{u_,v_}→e_u e_v;

        If[n>1,
            bivectorC=cList.eList,
            Print["Error: 2nd entry of BiVectorG must be greater than 1."]
        ];

        bivectorC      (*End of BivectorG Module*)
    ]

nVectorG[c_,n_]:=   (*  Note: n-vector is a vector of dimension 1 in n-dimensional space
    Module[{vectorC},
        Cases[SubValues[Subscript],dv_/;FreeQ[dv,c]];DownValues[Subscript];
        (* Clear all c-subscripted variables *)

        vectorC=Sum[c_i e_i,{i,n}]
    ]    (*End of nVectorG Module*)
```

5 Multivector Support Functions

In[6596]:=

```
CollectG[clifInput_,n_:-1]:=  (*

INPUTS:
    clifInput is the clif to be simplified
    n - Optional input. Max value of any e_i subscript for purposes of collecting terms in p

PROCESS:
    Call function MaxDimG to find the size of the largest e-subscript in the clif
    Modify multivector clifInput according to user's initialization rules
    Use Mathematica's Collect to collect the terms by e1, ..., en, e1e2,
```

```
        e1e3, ..., e1e2...en
    This function is used by other functions and can also be called directly by the unser

OUTPUT:
    clifOutput = clif with all multiplications carried out and then arranged by product o
    This is same as Mathematica's Collect function except for implementation of user's in

    Module[{m,clifOutput},
        If[n==-1,(* if user doesn't supply an input *)
            m=MaxDimG[clifInput]
        ,
            m=n
        ];

        clifOutput=Collect[ExpandG[clifInput],ClifFormatG[m]];
        clifOutput
]       (* End of CollectG Module    *)

ConstantG[clif_]:= (*
    INPUT: A multivector
    PROCESS: Should be the same as 1st 2 lines of ClifToListG. See ClifToList Process
    OUTPUT: The constant term(s)of clif *)

    Module[{clif1,const},
        clif1=Expand[clif];
        const=clif1/.eu_→0; (* Constant, may be zero or consist of one or more non-zero te
(*
        Print["ConstantG: clif = ",clif,", Context of e: ",Context[e]];
*)
        const
    ]
    (*  End of ConstantG  *)


EijTermG[clif_,eij_]:= (*
INPUTS:
    clif = a multivector.  Example: e1e2 + 2e1e2e3 + 3e1e2e3e4
    eij = a product like e2 or e3e4 or e1e3e4e5 or else the constant 1.  Example: e1e2e3
PROCESS:
    Convert clif to a list.  Example: {e1e2, 2e1e2e3, 3e1e2e3e4}
    Use Cases to find the sublist of members containing the product eij.
        Example: {2e1e2e3, 3e1e2e3e4}
        Note: eij doesn't match the pattern ___ eij so have to get that case separately
    Change list back to an expression.  Example: 2e1e2e3 + 3e1e2e3e4
    If, in the epxression, the eij term has one or more ei factors, set those terms to 0
        Example: 3e1e2e3e4 → 0  leading to 2e1e2e3
    Find the eij coefficient by dividing out eij.  Example: 2
```

```
OUTPUT:
    eijPart = eij coefficients, if any, in clif
    In the case eij = 1, eijPart is just the constant term of clif  *)

    Module[{list,eijPart1,eijPart},
        list=ClifToListG[clif];
        eijPart1=CollectG[Join[Cases[list,eij],Cases[list,___ eij]]/.List→Plus];
        eijPart=(eijPart1/.eu_eij→0)/eij;
        eijPart
    ]  (* End of EijTermG  *)


ExpandG[clifInput_]:=Expand[InitializeG[clifInput]]  (*
Same as Mathematica's Expand function except:
    Multivector is modified first according to user's initialization rules shown in Palett


FreeTermG[0]:=0

FreeTermG[clif_]:= (*
    INPUT: A clif
    PROCESS:
        The obvious definition is a simple subtraction:  clif - constant term(s)
        But... this causes unwieldy Mathematica problems in certain situations
        Namely, if user limits precision of clif (say, 4 significant digits for outputtin
            Then Mathematica throws in a roundoff term (a constant like 10^-5) to the subtr
            Thus the "free" term retains a constant part
            This can be elimininated using algebraic operations, but that causes yet wors
            As Mathematica expands and collects terms, it changes the form original expre
            After multiplication, Mathematica cannot find the necessary simplifications t
        It is safer just to find the constant and free terms by manipulating lists withou
        Since this requires making the clif list, we call ClifToList and grab the free te
        See ClifTlList for a description of the process
    OUTPUT: clifFree = clif - constant term
*)
Module[{conTerm,clifFree,freeList},
    ClifToListG[clif,conTerm,clifFree,freeList];

    clifFree
]   (* End of FreeTermG  *)

GradePTermG[0,p_]:=0

GradePTermG[clif_,p_]:=  (* User's Clif is c0 + c1 e1 + c1,2 e1e2 + ...
INPUTS:
    clif = a Clif
```

```
    p = grade of terms we wish to select

PROCESS:
    This function finds terms in clif of grade p
    Call other functions to generate4 2 arrays:
        cliflist = list of terms in clif, with all constants grouped into term 1
        gradesClif = list of grades of terms in cliflist
    From gradesClif, create an array with 1's for all grade p terms and 0's otherwise
    Then simiply dot multiply this array with the list of terms from clif

    Note 1: p = 0 requires special treatment since we are setting non-grade p terms to 0

    Note 1: The constant term have been rounded up into term 1, and due to Note 1
        the grade has been changed from 0 to -1

OUTPUT:
    gradePClif = clif composed of just the grade-p terms, if any
*)
    Module[{clifList,gradesClif,gradeParray,gradePclif},

    If[debug4,Print[];
        Print["Clif being examined = ",Expand[clif],"   (GradePTermG)"];
        Print["Grade being investigated = ",p]
    ];

    gradesClif=GradeListG[clif]/.x_/;x==0→-1;
        (*  Get list of grades of terms
            A grade of zero corresponds to a non-zero constant term as first term in list
            If 1st term has grade 0, change it to -1 so as not to interfere with
                two statements below after "Else"  *)

    If[p>0,
        gradeParray=gradesClif/.x_/;x≠p:>0;  (* For p>0, set non-grade p terms to 0 *)
        gradeParray=gradeParray/p , (* and divide the grade p term by p to get 1 *)
    (* Else *)
        gradeParray=gradesClif/.x_/;x≠-1:>0;  (* For p=0, set non-constants to 0 *)
        gradeParray=-gradeParray  (* and change the constant term from -1 to +1  *)
    ];

    clifList=ClifToListG[clif];

    gradePclif=gradeParray.clifList;

    If[debug4,
        Print["List of grades of ",clif,
                " with constant (if any) set to -1: ",gradesClif];
```

```
        Print["Array, with 1's at grade ",p," position(s) (if any) = ",gradeParray];
        Print["List of any grade ",p," terms: ",gradePclif,".  If grade > 0, 1st term",
        " is 0. List is product of array with 1's and 0's with list of the clif terms."];
        Print["Clif term(s) of grade ",p," = ",gradePclif]
    ];

    gradePclif
    ]    (*  End of GradePTermG Module  *)


InitializeG[clifInput_]:=
 (*
INPUT:
    clifInput = A multivector

PROCESS:
    Internal function called only by ExpandG, which in turn is called by CollectG
    Modifies multivector clifInput according to user's initialization rules for terms eᵢ²,
        set up either in Palette or manually
    algebraType: 1 = Clifford algebra,  2 = Grassmann algebra
    signatureType: 1 = Mathematicians - + + + ,  -1 = Physicists + - - -
    numTimelike = Number of timelike dimensions.
        0 = Pure Space   1 = Spacetime   2 or more time dimensions are allowed, time alwa
        the lowest ubscripts

OUTPUT:  clifOutput = Multivector with any square eᵢ terms reduced:
                    = 0 if Grassmann algebra;
               else = signatureType if u ≤ numTimelike;
               else = signatureType

CONTEXT SHADOWING:
    The context of this package is GeomAlgPkg` or whatever is in BeginPackage, above
    If user calls this package from his environment having a diffenet context, say Contex
        then e = GeomAlgPkg`Private`e but we need e = ContextUser`e in order for /.e_u_²→1
    The user only enters clif1 (and possibly clif2), leading here to clifInput. He does n
        either e or his context.
    Here is how we find both:
        Suppose clifInput = 1 + 2e₁ + e₁²e₂
        leaves = Level[clifInput,{-1}] = {1,2,e,1,e,1,2,e,2}, the leaves (or atomic eleme
            Note 1: The leaves list memebers have context ContextUser`, what we are after
            Note 2: The symbol "leaves" does not. It has context GeomAlgPkg`Private`.
        symbols1 = Cases[leaves,_Symbol] = {e,e,e}, lists the symbols, if any, among the
        symbols2 = Append[symbols1,e]     = {e,e,e,e} appends an e to the end
            The appended e has context GeomAlgPkg`Private`, which is NOT what we want
            However, the appended e will only be chosen if symbols1 is an empty list; tha
            has no eᵢterms
                If ithere are no eᵢ terms, then eᵢ²→1 doesn't do anything, so no harm occu
```

```
              also no warning message occurs due to operating on an empty symbols1 list
        symbolA = First[symbol2] = e
            Note: As in Notes 1 and 2, above, e has the context we want; symbolA does not
                  But... the rhs e is not available for processing; we only have access t
        e = Evaluate[symbolA]
            e is rhs e (of symbolA) and has the user's context, the one we want
        contextUser = Context[symbolUser] is the user's context in list form
        Since contextUser is a list, eList = contextUser<>"e" is a list containing user's
        ToExpression[eList] = e in user's context

    The 3 init symbols (algebraType, signatureType, and numTimelike) similarly will have 
        context the first time InitializeG is called within a session. In that case the f
        if-test will be true and the ccontext shadow is set not only for e but for the in
    The user may or may not have previously used the palette to set the init symbols. If 
        the 2nd "Head" if-test will be true and default values are set.
    In subsequent calls of InitializeG these if-tests will be false and so redundant shad
        correcting will not occur.
    Finally, we need e and contextUser to be global within context GeomPkg`Private`so the
        repeatedly duricng a session
    We need the 3 init symbols to be global and available to the user's context because t
        settings are dynamic and will change or be changed in sync with the init symbols
*)

Module[{algebraType,signatureType,numTimelike,clifOutput},

    If[Head[signatureType]==Symbol,  (* If true, this signatureType symbol has GeomAlgPkg 
        algebraType=Global`algebraType;
        signatureType=Global`signatureType;  (* Change context to Global` for init symbol
        numTimelike=Global`numTimelike
    ];
    If[Head[signatureType]==Symbol,  (* If true, then Global` signatureType has not yet be
        Message[InitializeG::warning];
        Global`algebraType=1;
            (* "Evaluate" causes Global`algebraType = 2; else would set GeomAlgPkg`algebr
        Global`signatureType=1;  (* Set values for init variables *)
        Global`numTimelike=0
    ];

    clifOutput=clifInput/.
        {e_u_^2/;algebraType==2:>0,e_u_^2/;u≤numTimelike:>-signatureType,e_u_^2:>signatureType};

    If[debug1,
        Print[];
        Print["InitializeG: algebraType = ",algebraType,", signatureType = ",
            signatureType,", and numTimelike = ",numTimelike];
        Print["Context of e: ",Context[e]];
```

```
        Print["Clif Input = ",clifInput,", and Clif Ouput = ",clifOutput]
    ];
    clifOutput
]
(* End of InitializeG function   *)

(* ShadowSymbol[symbol_]:=ToExpression[contextUser<>ToString[symbol]] *)

MaxDimG[clif_]:=    (*
    INPUT: clif - A users clif. Coefficients can be either values or symbols
    PROCESS: Flatten the subscript list and find the maximum
    OUTPUT: maxDim = largest subscript value on any ei. Used when collecting terms in clif

    Module[{flatSubscriptList,maxDim},
        flatSubscriptList=Flatten[SubscriptListG[clif]];
            (* Want maxDim ≥ 1. Thus,c use clif rather than clif - const  *)
        maxDim=Max[1,Max[flatSubscriptList]]; (* Extra max takes care of clif = 0 *)

        If[debug2,
            Print[];
            Print["MaxDim: Flattened list of subscripts = flatSubscriptList = ",
                flatSubscriptList];
            Print["Max dim = largest subscript = ",maxDim]
        ];
    maxDim
    ]     (* End of MaxDimG Module    *)
```

6 List Operations and Support

In[6606]:=

```
ClifToListG[clif_]:=Module[{conTerm,freeTerm,clifFreeList,clifList},
    clifList=ClifToListG[clif,conTerm,freeTerm,clifFreeList]]

ClifToListG[clif_,conTerm_]:=Module[{freeTerm,clifList,clifFreeList},
    clifList=ClifToListG[clif,conTerm,freeTerm,clifFreeList];clifList]

ClifToListG[clif_,conTerm_,freeTerm_,clifFreeList_]:=  (* Optional output: conTerm, freeTe
(*
    INPUT: A clif (i.e, a multivector)
    PROCESS: Expand the list to simplify later multiplications.
                Want coefficients of blades to be single terms (not sums), to greatly red
                number of ways a product could be simplified, thus simplifying the code g
             Find the constant term(s) by setting all other terms to zero
             Find the list of constant terms by generating the clif's list and setting al
                and then deleting zero terms from list
```

```
                  Caution 1. Must delete both infinite and non-infinite precision 0's (i.e.
                  Caution 2. It might seem easier to just make a list from the already-comp
                      But, if some terms have precision set but not others, it is possible
                      Mathematica to combine terms in the constant but not in the list.
                      We need the constant list to exactly match the constants in the whole
                      the constants in the constant term
               Make a list from the constant term(s), separating the terms by commas
               To make a list from the clif where all constant terms are combined to be the
                  Make a list from the clif
                  Drop the constant terms = the first n terms where n = length of list of c
                  Prepend the constant term(s) as a single element
               If constant ≠ 0, the free term = the clif list without the first term (i.e.,
                  Unless... the clif itself is just a constant.
                  Then the free term = 0 and the free term list = {0}
               Note: If user has set any precision for any symbol, then 0 receives Machine
                  and IF TEST for 0 must also test for 0. (machine precision version of zer

    OUTPUT:  clifList, a list of the separate elements of the clif (directly returned)
             conTerm = Constant terms (if any), collected into 1st term of list (returned
             freeTerm = Non-constant terms, if any (returned via arg list)
             clifFreeList = a list of the non-constant terms of the clif
             conTerm, freeTerm, and clifFreeList are optional outputs.
             Use 2nd definition if only conTerm is needed.
             Use first definition if neither are needed.
             Note: Only ReverseG needs more than conTerm, and it needs all 3 optional out
                ReverseG thus uses the 3 definition
                Three definitions are used in order to avoid redundant recomputing of the
               *)

Module[{clif1,conList,len,clifList0,clifList},
    clif1=Expand[clif];
    conTerm=clif1/.e_u_→0;

    If[Head[clif1]===Plus,clifList0=List@@clif1,clifList0=List[clif1]];
    conList=DeleteCases[DeleteCases[clifList0/.e_u_→0,0],0.];
    len=Length[conList];

    If[conTerm===0||conTerm===0.,
          clifList = clifList0;
          freeTerm = clif1;
          clifFreeList = clifList,

          clifList = Prepend[Drop[clifList0, len], conTerm];
          clifFreeList = Drop[clifList, 1];
          freeTerm = clifFreeList /. List -> Plus;
          If[freeTerm===0||freeTerm===0.,clifFreeList={0}]
```

```
      ];

    If[debug9==True,
        Print[];Print["ClifToListG: Expanded clif = ",clif1];
        Print["Constant term = ",conTerm,",  Constant-term  list = ",  conList,",  Length
        Print["Precision of constant term = ",Precision[conTerm]];
        Print["Clif list with separated constant terms (if any) = ",clifList0];
        Print["Clif list with combined constant terms (if any) = ", clifList];
        Print["Free term = ", freeTerm];
        Print["Free list = ", clifFreeList]
      ];

    clifList
]  (* End of ClifToListG  *)

ConstantToZeroG[clif_,list_,const_]:=  (*
    INPUT: Clif, a list of clif subscripts or e-subscripts, and the constant term of the
    OUTPUT: list1 = Clif list with 1st term replaced by 0 if constant is non-zero  *)

    Module[{list1},
    If[const===0||const===0.,list1=list,list1=ReplacePart[list,1→0]];
    list1]  (* Sends constant term of list, if any, to 0 *)

eSubscriptListG[0]:={0}

eSubscriptListG[clif_]:= (*
    INPUT: clif = A multivector
    PROCESS:
    OUTPUT: eList = list of e_{i,j,...}corresponding to e_i e_j... terms in clif
            1st term is replaced by 0 if constant term of clif is non-zero
            eList = { 0 } if clif = 0
            Example: 5 + 3 e_1 e_2 → eList = {0, e_{1,2}}  *)

    Module[{clifList,eList,const},
        clifList=ClifToListG[clif,const]/.e_{u_}^2→e_{u,u};
        (* Reduce square terms, if any. Example: e_3^2 e_4^2 changes to e_{3,3} e_{4,4}
            Note: The only time there may be square terms is during computation of geomet
        eList=ConstantToZeroG[clif,clifList,const]
            //.{e_{u__} e_{v__}->e_{u,v},w_ e_{u__}→e_u,e_{u__}:→Subscript[e,Sequence@@Sort[List[u]]]};
        (* The repeated replace changes clif list into a e-subscript list with sorted sub
            ConStantToZeroG changees the 1st element in the list to 0 if constant term ≠
        eList
    ] (* End of eSubscriptListG *)

GradePTermG[0]:={0}
```

```
GradeListG[clif_]:=  (* User's Clif is c0 + c1 e1 + c1,2 e1e2 + ...  *)

(* INPUTS:
    clifC = a Clif

    PROCESS:
    This is an internal function called by GradePTermG and ReverseG
    It creates a list of the grades of the terms
    The process starts by grabbing the subscript list using SubscriptListG
    The length of ei,j,... = 1 + # of subscripts since base e is part of the length
    The constant term(s) could be of various lengths, so its grade is simply set to zero

    OUTPUT:
    gradeList = A list, matching the ordering of ClifToListG[clif], of the grades of
                each term
*)

    Module[{clif1,gradeList,const},
    clif1=Expand[clif];
    const=ConstantG[clif1];
    gradeList=ConstantToZeroG[clif1,Length/@eSubscriptListG[clif1]-1,const];
        (*  List of grades of clif terms (= number of subscripts on e terms)
            Constant 1st term (if any) gets assigned value of 0
            For other terms subtract 1 because Length counts base e along with
                subscripts *)
    If[debug4,Print["GradeListG: List of grades of terms of ",clif1," = ",gradeList]];

    gradeList
    ]     (* End of GradeListG Module  *)

ListToClifG[clifList_]:=clifList/.List→Plus

SignatureG[list_]:=
(*
    INPUT: A list of integers, letters or word, or other items that Mathematica
            can sort by structure

    PROCESS:  Find the list of permutations cycles that convert this list to an ordered or
              Replace each cycle by its length
              Replace even lengths with -1 and odd lengths with +1
              The signature is the product of the ±1's

    OUTPUT: The signature of the list:
                Let n = minimum # of pairwise permutations required to put the list
                    in order
                Note: If an integer repeats, this means not to permute adjacent equal ite
                Signature = +1 if n is even
```

```
                Signature = -1 if n is odd *)

    Module[{cycles,listOfCycles,permLength,evenOddLength,evenOddLength2,signature},
        cycles=FindPermutation[list];
        (*  We desire the cycles length to make this list ordered.
            This is the list of permutation cycles that convert this list to an ordered o
            Note: Technically, this converts an ordered list into this one, but the
                result is the same
         *)
        listOfCycles=List@@cycles /. {{{{u__}}}→{{u}},{{{u__},v__}}→{{u},v}};
            (*  Replace head (i.e., "Cycles") with "List"
                Then reduce one level of braces *)
        permLength=Length/@listOfCycles/. u_/;u==0→1;
            (*  Find the lengths of the cycles in the list
                Note: null cycles map to zero so we change them to 1. See example   *)
        evenOddLength=2Mod[permLength,2]-1;
            (* Map even length cycles to -1 and odd length cycles to +1 *)
        signature=evenOddLength/. List→Times; (* Remove one level of brackets

        Ex .   list={1, 1, 2, 4 }   This list shows that null cycles have to be handled
                cycles = Cycles[{}]
                listOfCycles = { { } }
                permLength = {1}
                evenOddLength = {1}
                signature = 1

        Ex      list={2, 3, 4, 1, 2 }
                    This list shows that a simple product (e2e3e4∧e1e2)can have more than
                cycles = Cycles[ {1, 4, 2}, { 3, 5} ]
                listOfCycles = { { 1, 4, 2}, { 3, 5} }
                permLength = {3,2}
                evenOddLength = {1,-1}
                signature = -1
*)

        If[debug3,
            Print["SignatureG: List of Product Blade = ",list,",  Cycles of list: ",cycle
            Print["Cycles sans head (Cycles): ",listOfCycles,", Length of cycles: ",permL
            Print["Even cycle -> -1, odd -> +1: ",evenOddLength,", Signature = Product of
                signature]
        ];
        signature
]  (* End of SignatureG Module *)


SubscriptListG[clif_]:=  (*
    INPUT: Clif = A multivector
```

```
    PROCESS:
        Use eSubscriptList to obtain a list of e-subscripts of terms of clif
            Example: 2 + 3 e2 e4 → { 0, e2,4 }
        Use /. replace all to change e-subscripts like e2e4 to a list of subscripts like {
        If constant term ≠ 0, change 1st term of list from 0 to {0}
        If entire clif = 0, make subscript list = { {0} }
    OUTPUT: subList1 = List of the subscripts of the e-terms
            1st term = {0} if either constant is non-zero or entire clif = 0
            Example: 5 + 3 e1e2 → { {0 }, {1,2} )
            Example: 0 → { {0} }
            Example: 3 e1e2 → { {1,2} }   *)

    Module[{subList0,subList1},
        subList0=eSubscriptListG[clif]/.eu__→{u};
        If[ConstantG[clif]===0||ConstantG[clif]===0.,subList1=subList0,subList1=ReplacePa
        If[clif===0||clif===0.,subList1={{0}}];
        subList1
    ]
```

7 Secondary Geometric Algebra Operations:

Hodge Dual, Norm, Gorm, Reverse, Inverse

In[6617]:=

```
GormG[clif_]:=Simplify[ScalarPrdtG[ReverseG[clif],clif]]

HodgeDualG[clif_,n_]:=clif∘PseudoScalarG[n]  (*

INPUTS: A clif and space dimension n

METHOD: The Hodge Dual (or Hodge Star) of a clif, clif2, of grade p ≤n is the unique
    clif, *clif2, that satisfies
        clif1 ∧ *clif2 =( clif1 · clif2) ∘ i  for every clif1 of same grade as clif2,
    where i is the PseudoScalar e1e2...en

    This definition extends linearly to a general clif of dimension n which
    is the sum of clifs of grades 0 - n.

    This package can be used to check that the following definition satisfies
    the condition:
        *clif2 = clif2 ∘ i

    Note: i ∘ clif2, i^-1 ∘ clif2, and clif2 ∘ i^-1 all fail to satisfy this condition
        The 3rd product is the alternate definition, below

OUTPUT: The Hodge Dual of clif  *)
```

```
    (* End of HodgeDualG  *)

HodgeDual2G[clif_,n_]:=clif∘InverseG[PseudoScalarG[n]]
    (* Alternate version *)


InverseG[clif_]:=
    Module[{gormClif,clifInv},
        gormClif=GormG[clif];
        If[(gormClif===0||gormClif===0.)&&NumberQ[gormClif],
            clifInv=1;Print["Caution: Inverse of ",clif," may not exist"],
            clifInv=ReverseG[clif]/gormClif
        ];
        clifInv
    ]


NormG[clif_]:=Sqrt[Abs[GormG[clif]]]   (*/;NumberQ[GormG[clif]]==True*)


ReverseG[0]:=0

ReverseG[clif_]:=
(*
    INPUT: A clif  Example: 3 + e2 + e2e 3 + e1e2e3e4 + e1e3e5
    PROCESS:
        Use ClifToListG to get a list of clif terms, like {3,e2,e2 e3,e1 e2 e3 e4,e1 e3 e5}
        Use SubscriptListG to get list of subscripts, like {{0}, {2}, {2, 3}, {1, 2, 3, 4
        Use Reverse to get reversed list of subscripts, like {{0},{2},{3,2},{4,3,2,1},{5,
        Use Signature to determine whether reversed members are even (+1) or odd (-1) per
        Multiply the clif list by the signature list and convert result to a multivector 
    OUTPUT: The reverse clif
*)
    Module[{clifList,subscriptList,reverseList,signatureList,reverseClif},
        clifList=ClifToListG[clif];
        subscriptList=SubscriptListG[clif];
        reverseList=Reverse/@subscriptList;
        signatureList=Signature/@reverseList;
        reverseClif=signatureList.clifList;

        If[debug7,
            Print["ReverseG: List of ",clif," = ",clifList];
            Print["Subscript list = ",subscriptList];
            Print["List of reversed subscripts = ",reverseList];
            Print["Signature List = ",signatureList," is ±1, measuring # of pairwise tran
```

```
                "to restore reverse list to original"];
            Print["Reverse of clif = ",reverseClif]
        ];
        reverseClif
    ](*  End of ReverseG Module  *)
```

8 Geometric Algebra Operations

Geometric Product, Dot Product, Wedge Product, Left Contraction

In[6624]:=

```
PrdtG[0,x_,p_]:=0
PrdtG[x_,0,p_]:=0

PrdtG[clif1_,clif2_,prdtType_]:=
    (*INPUTS:
        clif1, clif2: Two clifs whose dot/wedge/left or right contraction product is to b
        prdtType: The type of product: wedge, dot, right or left contraction

      PROCESS:
        Example: Wedge Product of clif1 = e1 + e2 e3 and clif2 = 4 - e3 e4
        Make lists of the terms of the 2 clifs: clifList1 ={ e1, e2 e3 } and clifList2 { 4
        Use Outer to take the Geometric Product of every term from clif1 with every term
           prdtBladeList = {4 e1, -e1 e3 e4, 4 e2 e3, -e2 e4}
        Notice: every term is a simple blade a e1e2...en
        Find the eSubscripts and then the grade of each blade:
           eSubscripts: {{e1},{e2,3}} and {{0},{e3,4}}
           grades: { 1, 2 } and { 0, 2 }
           Note: grade of e2,4 = 2 = Length [ e2,4 ] - 1
           Note: length of a constant, like 5, is 0, so subtracting 1 gives a grade of -1
                 The correct grade is 0, so Length - 1 is clipped to be ≥ 0
        Use Outer to make a list of the target grades of every term from clif1 with every
            Target grade for Wedge Product of 2 blades is sum of the two grades
            Thus, targetGradeList = {1 + 0, 1 + 2, 2 + 0, 2 + 2} = { 1, 3, 2, 4 }
            Target grade for Dot Product is |grade1 - grade2|
            Target grade for Right Contraction is grade1 - grade2
            Caution: For left contraction, desire (grade2 - grade1).
                But, to mimic the outer order of prdtBlade, we must enter blade 1 first
                That is, we cannot enter Outer [ Subtract, gradeList2, gradeList1 ]
                Rather, we enter this as - Outer [ Subtract, gradeList1, gradeList2 ]
                The latter gives, of course, the opposite sign from grade 1 - grade 2 for
        Use Inner to make a list of pairs, where:
            First element of pair is the geometric product of 2 blades
```

```
            Second element of pair is the target grade for that geometric product
            For Wedge Product, bladeGradePairList = {{4 e1, 1},{-e1 e3 e4, 3},{4 e2 e3, 2}
        Use GradePTerm to extract the part of each geometric product, if any, having the
            Note: We need a list of GradePTerm[blade, tgt grade] so we APPLY GradePTerm a
            For Wedge Product, { 4e1, -e1 e3 e4, 4 e2 e3, 0 }
        Lastly, we change the list of target pieces into its series expression
            Wedge Product = targetPrdt = 4 e1  +4 e2 e3 - e1 e3 e4

      OUTPUT:
        The dot/wedge/contraction product of the two clifs, defined as the sum of the pro
        every term from clif1 with every term from clif2  *)

    Module[{dummy1,dummy2,dummy3,dummy4,clifList1,clifList2,prdtBladeList,eSubscriptList1
            gradeList2,targetGradeList,bladeGradePairList,targetPrdt},

        clifList1=ClifToListG[clif1];clifList2=ClifToListG[clif2];
        prdtBladeList=Flatten[Outer[GeomPrdtG,clifList1,clifList2]];
        eSubscriptList1=eSubscriptListG/@clifList1;
        eSubscriptList2=eSubscriptListG/@clifList2;
        gradeList1=Clip[Length@@@eSubscriptList1-1,{0,∞}];
        gradeList2=Clip[Length@@@eSubscriptList2-1,{0,∞}];
        If[prdtType == "Wedge",targetGradeList=Outer[Plus,gradeList1,gradeList2]];
        If[prdtType == "Dot",targetGradeList=Abs[Outer[Subtract,gradeList1,gradeList2]]];
        If[prdtType == "LC",targetGradeList=-Outer[Subtract,gradeList1,gradeList2]];
        If[prdtType == "RC",targetGradeList=Outer[Subtract,gradeList1,gradeList2]];
        targetGradeList=Flatten[targetGradeList];
        bladeGradePairList=Inner[List,prdtBladeList,targetGradeList,List];
        targetPrdt=CollectG[Apply[GradePTermG,bladeGradePairList,{1}]/.List→Plus];

        If[debug6==True,
            Print[];Print["clif lists: ",clifList1,",  ",clifList2];
            Print["e-Subscript lists: ",eSubscriptList1,",  ",eSubscriptList2];
            Print["grade lists: ",gradeList1,",  ",gradeList2];
            Print["Blade geom prdt list: ",prdtBladeList];
            Print["Target grade list: ",targetGradeList];
            Print["List of {blade geom prdt, tgt grade}: ",bladeGradePairList];
            Print[prdtType," product = ",targetPrdt]
        ];
        targetPrdt
    ]

DotPrdtG[clif1_,clif2_]:=
    Module[{dotPrdt},
        dotPrdt=PrdtG[clif1,clif2,"Dot"];
        dotPrdt]
```

```
DotPrdtG[u_,v_,w__]:=DotPrdtG[DotPrdtG[u,v],w]
DotPrdtG[u_,v_]:=0/;u==0||v==0
    (*  End of DotPrdtG Module  *)


GeomPrdtBladeG[blade1_,blade2_]:= (*
    INPUTS: 2 simple blades, blade1 & bladd2  (e.g. blade1 = e1e2e4)
    PROCESS:
        Combine the 2 subscript lists, preserving blade1 subscripts before blade2
        Use SignatureG to get the signature of the combined list
    OUTPUT:
        bladePrdt = algebraic product of signature, blade1, and blade 2
        Note: ei^2 terms are NOT reduced until after this function is called by GeomPrdtG
*)
    Module[{maxDim,subscriptList1,subscriptList2,subscriptList12,signature12,bladePrdt},

            If[debug3,
                Print[];Print["GeomPrdtBLADEG: Blade1 = ",blade1,", Blade2 = ",blade2]
            ];

                subscriptList1=Flatten[SubscriptListG[blade1]];
            (* Example: 5 + 3 e1e2 → { {0}, {1,2} ) → {0, 1, 2}  *)
        maxDim=Max[subscriptList1]; (*  Example: maxDim = 3  *)
        subscriptList2=Flatten[SubscriptListG[blade2]/.u_/;u==0→maxDim];
            (* Example:  4 → {0} → {3}   *)

        subscriptList12=Join[subscriptList1,subscriptList2];  (* Example: {1,3,4}  *)
        If[debug3,Print[];
            Print["GeomPrdtBLADEG: Flattened list of Blade1 subscripts = ", subscriptList
                ", Flattened list of Blade2 subscripts = ",subscriptList2];
            Print["Joined flatten subscript lists = ",subscriptList12]
        ];
        signature12=SignatureG[subscriptList12];  (* Example: signature = +1  *)

        bladePrdt=signature12 blade1 blade2;

        If[debug3,
            Print["GeomPrdtBLADEG: Signature = ",signature12,", product blade = ",bladePr
        ];

        bladePrdt
    ]

GeomPrdtG[0,y_]:=0
GeomPrdtG[x_,0]:=0
```

```
GeomPrdtG[clif1_,clif2_]:=   (*

    INPUTS:
        clif1, clif2 - Users input clifs, with either numeric or symbolic coefficients
    PROCESS:
        1. Put the 2 clifs into lists
        2. Take outer product of function GeopmPrdtBladeG applied to the lists to create
            list of geometric products between all terms (blades) of clif1 with all terms
        3. Convert the list back to an expression
        4. Use CollectG to evaluate all terms ei^2 and to collect terms by blades
    OUPUT:
     clif12: The geometric product clif *)

    Module[{dummy1,dummy2,dummy3,dummy4,clif1a,clif2a,clifList1,clifList2,clifList12,clif

        clif1a=Expand[clif1];clif2a=Expand[clif2];
        If[debug3,Print[];Print["GeomPrdtG: clif1 = ",clif1a,", Clif2 = ",clif2a]];

        clifList1=ClifToListG[clif1a];
        clifList2=ClifToListG[clif2a];

        If[debug3,Print[];Print["GeomPrdtG: Clif1 List = ",clifList1,",  Clif2 List = ",c

        clifList12=Outer[GeomPrdtBladeG,clifList1,clifList2];
        clif12 = CollectG[clifList12/.List→Plus];

        If[debug3,Print[];
            Print["GeomPrdtG: Product clif list = ",clifList12];
            Print["Product clif = ",clif12]
        ];
        clif12
    ]

GeomPrdtG[u_,v_,w__]:=GeomPrdtG[GeomPrdtG[u,v],w]
    (* Enables expressions like GeomPrdtG[x,y,z,u,v]  *)

GeomPrdtG[u_,v_]:=0/;u==0||v==0

     (* End of GeomPrdtG Module  *)

LeftContractionG[clif1_,clif2_]:=
    Module[{leftContraction},
        leftContraction=PrdtG[clif1,clif2,"LC"];
        leftContraction]
LeftContractionG[u_,v_,w__]:=LeftContractionG[LeftContractionG[u,v],w]
LeftContractionG[u_,v_]:=0/;u==0||v==0
```

```
    (* End of Function LeftContractionG  *)

RightContractionG[clif1_,clif2_]:=
    Module[{rightContraction},
        rightContraction=PrdtG[clif1,clif2,"RC"];
        rightContraction]
RightContractionG[u_,v_,w__]:=RightContractionG[RightContractionG[u,v],w]
RightContractionG[u_,v_]:=0/;u==0||v==0
    (* End of Function RightContractionG  *)

ScalarPrdtG[clif1_,clif2_]:= (*

INPUT: multivectors
PROCESS:
    As with function PrdtG, prdtBladeList is a list of blades in the geometric product
    Similar to function PrdtG, eSubscriptList is a list of the e-subscripted terms of the
    However, when eSubscriptListG is MAPPED (/@) to prdtBladeList, we can get duplicate e
        If clifList = { ((1 + √2) e1 }, then eSubscriptList computes to { {e1, e1} }
    Map DeleteDuplicates to get eList = { { e1 }
    Apply [ Length, eSubscriptList, {1} ] to get the list of lengths of the e-subscripted
    Subtract one to send grade 0 terms to -1 and all other terms to [ 0, ∞ ]
    Clip the non-grade 0 terms to make them all 0, and multiply by -1 to make the grade z
    The scalar product is the grade zero terms in the geometric product:
        prdtBladeList . zeroOnegradeList
OUTPUT: The scalar product *)

    Module[{clifList1,clifList2,prdtBladeList,eSubscriptList,eList,zeroOnegradeList,scala
        clifList1=ClifToListG[clif1];clifList2=ClifToListG[clif2];
        prdtBladeList=Flatten[Outer[GeomPrdtG,clifList1,clifList2]];
        eSubscriptList=eSubscriptListG/@prdtBladeList;
        eList=DeleteDuplicates/@eSubscriptList;
        zeroOnegradeList = -Clip[Length@@@eList-1, {-1, 0}];
        scalarPrdt = prdtBladeList.zeroOnegradeList;

        If[debug8 == True,
              Print[]; Print["clif lists: ", clifList1, ",  ", clifList2];
              Print["Blade geom prdt list: ", prdtBladeList];
              Print["e-Subscript list: ", eSubscriptList];
              Print["Reduced e-Subscript list: ",eList];
              Print["grade list: ", zeroOnegradeList];
              Print["Scalar product = ", scalarPrdt]];
        scalarPrdt]

ScalarPrdtG[u_,v_,w__]:=ScalarPrdtG[ScalarPrdtG[u,v],w]
ScalarPrdtG[u_,v_]:=0/;u==0||v==0
```

```
WedgePrdtG[clif1_,clif2_]:=
    (*INPUTS:
        The two clifs whose Wedge product is to be taken
      PROCESS:
        See Function PrdtG
      OUTPUT:
        The Wedge product of the two clifs, defined as the sum of the Wedge products of
        every term from clif1 with every term from clif2  *)
    Module[{wedgePrdt},
        wedgePrdt=PrdtG[clif1,clif2,"Wedge"];
        wedgePrdt]
WedgePrdtG[u_,v_,w__]:=WedgePrdtG[WedgePrdtG[u,v],w]
WedgePrdtG[u_,v_]:=0/;u==0||v==0
    (*  End of WedgePrdtG Module  *)

End[]
EndPackage[]
```

Out[6648]= GeomAlg2017June`Private`

In[6650]:= "GeomAlg2017June`Private`"

Out[6650]= GeomAlg2017June`Private`

In[6651]:= "GeomAlg2017`Private`"

Out[6651]= GeomAlg2017`Private`

Version Changes

2019 August

Improved warning message. Fixed a few typos. Improved documentation of PrdtG function.

version 52d

Simplified expressions for both Hodge duals (no change to calculations; functions are just more readable)

version 52cc

Replaced multiple instances of list1 list2 /. List -> Plus with list1 . list2

Renamed GradePpieceG to GradePTermG

Renamed EijPieceG to EijTermG

Corrected Hodge2G calculation

version52bc

Added EvenClifG and EijPieceG functions

version 52

Switched algebraType reference to match change in palette: Now, 1 = Clifford Algebra, 2 = Grassmann Algebra

Added function EiEjPieceG

Corrected definition of Gorm to use scalar product rather than dot product

Added “Simplify” to the Gorm result since it usually leads to clearer answers

Fixed ReverseG which would make some incorrect calculation for grades 4 and higher

This fixed GormG and NormG, both of which use ReverseG

version 51g

Deleted all reference to SortG and ExpandSortG as they did not help in sorting the way I hoped

version 51f

Corrected ClifToList for cases where some parts of clif have different precision than other parts

Change manner of computing list of constant terms

Version 51e

Replaced Apply [Length [list, {1}] by Length @@@ list in many places; (just a readability change)

Added internal SortG function to sort terms if multivector has 2 or more terms

Improve sorting of output multivectors by applying Mathematica’s Sort function to multivectors with 2 or more terms

Correct freeTermList calculation when the multivector is a constant. freeTermList should be { 0 }, not { }

Version 51c

Completely rewrote logic for a core function, ClifToListG

Now handles multivector inputs that have had precision set by user (for example, to format output to 6 sig digits)

Rewrite extended to include ConcstantG

Added FreeTermG function (multivector complement of ConstantG)

Added ExpandSortG, a sub-function to ensure that functions Expand and then sort multivectors in a consistent way

```
In[6652]:= ? GeomAlg2017June`*
```

▼ GeomAlg2017June`

BiVectorG	debug3	EvenClifG	InverseG	ReverseG
BladeG	debug4	ExpandG	LeftContractionG	RightContractionG
ClifFormatG	debug5	FreeTermG	ListToClifG	RotorG
ClifToListG	debug6	GeomPrdtBladeG	MaxDimG	ScalarPrdtG
CollectG	debug7	GeomPrdtG	nClifG	SignatureG
ComplexG	debug8	GormG	NormG	SubscriptListG
ConstantG	debug9	GradeListG	nVectorG	WedgePrdtG
ConstantToListZeroG	DotPrdtG	GradePTermG	pBladeG	
ConstantToZeroG	e	HodgeDual2G	PrdtG	
debug1	EijTermG	HodgeDualG	PseudoScalarG	
debug2	eSubscriptListG	InitializeG	QuaternionG	