Linear Algebra Paper

Ryan Porter

December 2022

Contents

1 Introduction

The purpose of this project is to create an Object-Oriented implementation of basic Exterior Algebra operations, specifically the wedge product, inner product, and Hodge Star. This paper will give a brief overview of Exterior algebra as well as each of these products, then expand on the specific MatLab implementations. Particular focus is placed on the Linear Algebra required to understand and encode Change of Basis propagation between vector spaces of different grades, as well as computing each product.

2 Exterior Algebra and Multivectors

2.1 Overview

Exterior Algebra is an algebra of objects called Multivectors, which are extensions of the vector objects taught in Calculus III and below.

Consider an n-dimensional vector space $V^N \in \mathbb{R}^N$, with the elements of V^N as column vectors. Within V^N, we can define a set of n linearly independent vectors to act a basis $\{e_1, e_2, ...e_n\}$, from which we can then define any other vector $v \in V^N$ as a linear combination of our basis vectors.

2.2 Multivectors

In the language of Exterior Algebra, this base N-dimensional vector space V^N is related to $N-1$ other vector spaces through the bilinear and antisymmetric Wedge product, which is explained in depth later. These vector spaces are formed of elements called multivectors, also known as k-vectors. The integer value 'k' denotes the grade of a multivector, and represents the number of basis vectors that have been wedged together to form the new basis of the multivector space. The vectors we have used up to this point are known as 1-vectors.

Aside from having different basis objects, multivector spaces work the same as vector spaces, in that vectors are denoted by linear combinations of the basis multivectors. Addition and scalar multiplications are also defined the same way for multivectors. One thing to note is due to the properties of the wedge product, a multivector of grade k in dimension n has $\binom{n}{x}$ basis multivectors of grade k.

This multivector basis is written with indices increasing from left to right. For example, the bi-vector basis in $\mathbb{R}^N$ is the following:

$$\{e_1 \wedge e_2,\, e_1 \wedge e_3 \,...\, e_1 \wedge e_N,\, e_2 \wedge e_3 \,...\, e_{N-1} \wedge e_N\} \tag{1}$$

$$\{e_i \wedge e_j\} \quad \text{for} \quad 1 \leqslant i < j \leqslant N \tag{2}$$

(2) can be modified to show the basis for any grade k in dimension N.

2.3 Change of Basis

Given a new set of basis 1-vectors $\{e'_1, e'_2, ...e'_n\}$ written in the current basis, we can construct a nxn matrix P whose columns are $e'_1, e'_2, ...e'_n$. This matrix has the following property

$$P\, v' = v \tag{3}$$

Since we want to transform v into v', we can left multiply (3) by P^{-1} to get

$$v' = P^{-1} v \tag{4}$$

We call P the change of basis matrix for 1-vectors. For higher grades of multivectors, the change of basis matrix P^k must be computed using P. The size of P^k is $\binom{n}{k}\mathrm{x}\binom{n}{k}$.

Since multivectors of higher grades are formed by wedging multivectors of lower grades together, we can figure out the components of P^k by wedging k 1-forms together under a change of basis, and identifying how the components in the resulting k-form are transformed. We can start with computing the change of basis matrix for 2-vectors. Consider the two basis 1-vectors e'_a and e'_b under the change of basis matrix P(4)

$$e'_a = \sum_{i=1}^{N} \left(P^{-1}_{i\,a} e_i\right), \quad e'_b = \sum_{j=1}^{N} \left(P^{-1}_{j\,b} e_j\right) \tag{5}$$

$$e'_a \wedge e'_b = \sum_{i=1}^{N}\sum_{j=1}^{N} \left(P^{-1}_{i\,a}\, P^{-1}_{j\,b}\right) e_i \wedge e_j \tag{6}$$

By the properties of the wedge product, if $i = j$ then $e'_i \wedge e'_j = 0$. Alternatively if $i \neq j$ then $e_i \wedge e_j = -e_i \wedge e_j$. In light of this we can use (7) as the following:

$$e'_a \wedge e'_b = \sum_{i=1}^{N-1}\sum_{j=i+1}^{N} \left(P^{-1}_{i\,a}\, P^{-1}_{j\,b} - P^{-1}_{j\,a}\, P^{-1}_{i\,b}\right) e_i \wedge e_j \tag{7}$$

This way, each permutation of $e_i \wedge e_j$ occurs only once, and it becomes easier to decipher the summation. Astute readers will recognize the quantity $P^{-1}_{i\,a}\, P^{-1}_{j\,b} - P^{-1}_{j\,a}\, P^{-1}_{i\,b}$ as the determinant of the 2x2 matrix:

$$\begin{vmatrix} P^{-1}_{i\,a} & P^{-1}_{j\,a} \\ P^{-1}_{i\,b} & P^{-1}_{j\,b} \end{vmatrix} \tag{8}$$

We can solidify the relationship between the entries in P^k and determinants of the submatrices of P by computing the wedge of three basis 1-vectors under the change of basis matrix P.

$$e'_a \wedge e'_b \wedge e'_c = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{l=1}^{N} \left(P^{-1}_{i\,a}\, P^{-1}_{j\,b}\, P^{-1}_{l\,c} \right) e_i \wedge e_j \wedge e_l \tag{9}$$

$$e'_a \wedge e'_b \wedge e'_c = \sum_{i=1}^{N-2} \sum_{j=i+1}^{N-1} \sum_{l=i+1}^{N} P^{-1}_{i\,a} \left(P^{-1}_{j\,b}\, P^{-1}_{l\,c} - P^{-1}_{l\,b}\, P^{-1}_{j\,c} \right)$$

$$-P^{-1}_{j\,a} \left(P^{-1}_{i\,b}\, P^{-1}_{l\,c} - P^{-1}_{l\,b}\, P^{-1}_{i\,c} \right) + P^{-1}_{l\,a} \left(P^{-1}_{i\,b}\, P^{-1}_{j\,c} - P^{-1}_{j\,b}\, P^{-1}_{i\,c} \right) e_i \wedge e_j \wedge e_l \tag{10}$$

Which is exactly the determinant of the 3x3 matrix:

$$\begin{vmatrix} P^{-1}_{i\,a} & P^{-1}_{j\,a} & P^{-1}_{l\,a} \\ P^{-1}_{i\,b} & P^{-1}_{j\,b} & P^{-1}_{l\,b} \\ P^{-1}_{i\,c} & P^{-1}_{j\,c} & P^{-1}_{l\,c} \end{vmatrix} \tag{11}$$

Let us define Rk_i as the index of a i-th basis k-vector. The basis vectors $e_1 \wedge e_2$, $e_1 \wedge e_3$, and $e_2 \wedge e_3$ in $\mathbb{R}^3$ would therefore have indices $R2_1$, $R2_2$, $R2_3$ respectively. Additionally, we can use this index to get the permutation of basis 1-vectors for a k-vector basis in the following way:

$$\text{perm}(Rk_i) = [\text{Indices of the basis vectors wedged to form the i-th k-vector basis}]$$

Here is an example in $\mathbb{R}^3$:

$$\text{perm}(R2_1) = [1\ 2], \quad \text{perm}(R2_2) = [1\ 3], \quad \text{perm}(R2_3) = [2\ 3]$$

We can now define the entries of P^k as follows:

$$P^k_{a\,b} = det(P([\text{perm}(Rk_a); \text{perm}(Rk_b)]))$$

where $P([\text{perm}(Rk_a); \text{perm}(Rk_b)])$ is the submatrix of P constructed using rows from $\text{perm}(Rk_a)$ and columns from $\text{perm}(Rk_b)$.

Now that we have constructed P^k for each grade k in dimension N, to transform a k-vector we simply perform the same matrix multiplication as (4) using P^k and our k-vector.

2.4 Covectors and Contravectors

Consider the following simple physics question: Given a constant force vector F and a displacement vector s, find the work done. In this case, work is

$$W = F \cdot s \tag{12}$$

Consider the same force and displacement vectors, but under a change of basis. Now, force is $P^{-1}F$ and displacement is $P^{-1}s$. The work calculated is now

$$W = P^{-1}F \cdot P^{-1}s \tag{13}$$

which is clearly not equivalent to (12). It is helpful then to define new objects through which we can ensure that necessary computations, such as work, are invariant with regards to changes in basis. We do this by separating multivectors into two categories: covectors which change with the basis, and contravectors with change inversely with the basis. Since the multivectors we have been dealing with before for the change of basis computations are transformed by P^{-1}, they are classified as contravectors. We can now define force to be a covector. Since the most natural application of force is as a dot product with displacement and other contravectors, we will define covectors as rows, and replace the dot product with standard matrix multiplication. This means our change of basis computation of a covector F for a change of basis matrix P is

$$F_p = F\,P \tag{14}$$

When using these covector and contravector definitions, if we again consider (12), we calculate work to be

$$W = FP\,P^{-1}s = F\,s \tag{15}$$

Remembering that F is a row vector, we find that this is the same result as (12), therefore work is now invariant to changes in basis.

In continuing forward, multivector operations are assumed to work identically between covectors and contravectors, except where otherwise stated. Covectors will be denoted by subscript, and contravectors will be denoted by superscript, as shown in the basis and components of (6) respectively

2.5 Implementation

The basic functionality of the Exterior Algebra class should be the following

1. Validate a 1-vector basis
2. Construct bases for higher grade multivectors from an input 1-vector basis

3. Create multivectors of different grades, validating component array sizes
4. Change basis, which should propagate to multivectors and change their components, so that their evaluations remain the unchanged

The Exterior Algebra class will have the following properties and constructor.

```
properties (SetAccess = private, GetAccess = public)
    gradeBasis
    gradeMetric
    permuteBasis
    dimension
    hodgeArr
end
properties (SetAccess = public, GetAccess = public)
end

methods (Access = public)
    %Input Basis and Metric are for grade 1 vector space
    function self = ExteriorAlgebra(Dim, Basis, Metric)
        self.dimension = Dim;
        if(self.dimension > 32)
            error("ERROR:\nDimension %s is too large. Currently " + ...
                "only supports up to Dim 32", self.dimension);
        end

        if(nargin < 2)
            Basis = eye(Dim);
        else
            Basis = ExteriorAlgebra.CheckBasis(Basis, Dim);
        end

        if(nargin < 3)
            Metric = eye(Dim);
        else
            Metric = ExteriorAlgebra.CheckMetric(Metric, Dim);
        end

        self.permuteBasis = ExteriorAlgebra.CreateGradeBasisPermute(self.dimension);
        self.gradeBasis = self.CreateGradeBasis(Basis);
        self.gradeMetric = self.CreateGradeMetrics(Metric);
        self.hodgeArr = self.CreateHodgeArrays();
    end
```

The properties of Exterior Algebra are set to get only, which is important to

ensure that Multivector components stay consistent with the Exterior Algebra basis. Properties should in general be kept private, and should be edited by methods within the class, such as the ChangeBasis method that will be discussed in 2.5.4. The gradeBasis property is set by the CreateGradeBasis method, which takes in a 1-vector basis and returns a cell array containing all bases for all grades within the Exterior Algebra, and is covered in 2.5.2. The permuteBasis property stores arrays with the index elements of each multivector basis Rk_i. The function to do this is below:

```
function kbasis = CreateGradeBasisPermute(dim)
    kbasis = cell([dim 1]);
    for i=1:dim
        kbasis{i} = nchoosek(1:dim,i);
    end
end
```

Metrics and Hodge arrays will be explained in future sections.

2.5.1 Validation

For a dimension N, the 1-vector basis written as a matrix b should have the following properties:

1. b should be a NxN matrix
2. The columns of b should be linearly independent

To check item (2), one approach would be to calculate the determinant of b, and check if it equals zero. This is computationally costly, so instead we will check the rank of b, which will equal N if the basis is linearly independent.

The following code takes in a matrix b for a dimension dim and validates whether or not b can be used as a basis:

```
function basis = CheckBasis(nbasis, dim)
    if(issparse(nbasis))
        nbasis = full(nbasis);
    end
    basis = nbasis;

    s1 = size(basis,1);
    s2 = size(basis,2);

    if(s1 ~= dim)
        error("ERROR:\nBasis size should match dimension\n" + ...
            "Current matrix is [%d by %d]",s1, s2)
    end
```

```
    if(s1 ~= s2)
        error("ERROR:\nPlease use a square matrix as your basis\n" + ...
            "Current matrix is [%d by %d]",s1, s2)
    end

    if(rank(basis) ~= dim)
        error("ERROR:\nBasis must have rank = dimension\n" + ...
            "Current matrix has rank %d and dim %d",rank(basis), s1)
    end
end
```

2.5.2 Higher-grade Bases

The steps to calculate higher-grade bases are outlined in section 2.3. Given a 1-vector basis b, the following code can calculate all higher grade multivector bases:

```
function basis = CreateGradeBasis(self, b)
basis = cell([self.dimension 1]);
basis{1} = b;

for i=2:self.dimension
    iBasis = self.permuteBasis{i};
    n = size(iBasis,1);
    Q = zeros(n);
    for x=1:n
        for y = 1:n
            %Find P_xy Here
            subMatrix = zeros(i);
            pRow = iBasis(x,:);
            pCol = iBasis(y,:);
            for x1 = 1:i
                for y1 = 1:i
                    subMatrix(x1,y1) = b(pRow(x1),pCol(y1));
                end
            end
            Q(x,y) = det(subMatrix);
        end
    end
    basis{i} = Q;
end
```

2.5.3 Multivector Creation

To create a multivector, we need to know the grade, the components, and the type (covector or contravector). The only validation that needs to be done is on the

components, to ensure their size matches $\binom{n}{k}$. Below is both the Multivector class as well as the Exerior Algebra function to create Multivectors.

```
classdef Multivector < handle
    properties (SetAccess = private, GetAccess = public)
        components
        degree
        dimension
        type
    end

    methods
        function self = Multivector(ExAlgebra, Degree, Components, Type)
            assert(isa(ExAlgebra,"ExteriorAlgebra"), ...
                "Ensure first argument is of type ExteriorAlgebra")

            s = size(ExAlgebra.permuteBasis{Degree},1);
            c = size(Components);
            if((c(1) ~= s|c(2)~=1) & ((c(1) ~= 1|c(2)~=s)))
                error("ERROR:\nComponents size for Degree %d in Dimension %d" + ...
                    " should be %d",Degree, ExAlgebra.dimension, s)
            end

            self.components = Components;
            if(nargin < 4)
                Type = MultivectorType.Contravector;
            end

            if(Type == MultivectorType.Covector)
                if(~isrow(self.components))
                    self.components = self.components';
                end
            elseif(Type == MultivectorType.Contravector)
                if(isrow(self.components))
                    self.components = self.components';
                end
            else
                error("ERROR:\nLabel Multivector as " + ...
                    "MultivectorType.Covector or MultivectorType.Contravector")
            end

            addlistener(ExAlgebra, 'ChangeBasisEvent', @self.basisChangeCallback);
            self.degree = Degree;
            self.dimension = ExAlgebra.dimension;
```

```
            self.type = Type;
        end
    end
end

function obj = CreateMultivectorCurrentBasis(self, Degree, Components, Type)
    assert(isa(Type,"MultivectorType"), ...
        "Ensure third argument is of type MultivectorType")

    %Check Components Length matches permuteBasis{Degree} length
    s = size(self.permuteBasis{Degree},1);
    c = size(Components);
    if((c(1) ~= s|c(2)~=1) & ((c(1) ~= 1|c(2)~=s)))
        error("ERROR:\nComponents size for Degree %d in Dimension %d" + ...
            " should be %d",Degree,self.dimension,s)
    end

    obj = Multivector(self, Degree, Components, Type);
end
```

The line "addlistener(ExAlgebra, 'ChangeBasisEvent', @self.basisChangeCallback);" in the Multivector constructor will be explained in the next section.

2.5.4 Change of Basis and Propagation

When implementing Change of Basis, given an input matrix nb perhaps the most logical thing to do is to simply use the function defined in 2.5.2 to calculate the set of newBasis, which can then be stored and used to compute new Multivector components. However, this means our newBasis would be in terms of the previous Basis matrix, whose data has been overriden and therefore is meaningless. Instead, we will have the input matrix nb be interpreted as the standard basis representation of our new Basis 1-vectors. To compute change of basis now we should use the current basis to convert Multivectors into the standard basis, then use the new basis to convert Multivectors into the new basis. Below is this Exterior Algebra method for making the change of basis, as well as the event that the method triggers.

```
function ChangeBasis(self, newBasis)
    newBasis = ExteriorAlgebra.CheckBasis(newBasis, self.dimension);

    newGradeBasis = self.CreateGradeBasis(newBasis);

    notify(self, 'ChangeBasisEvent', ChangeBasisData(self.gradeBasis,
                                                     newGradeBasis));
    self.gradeBasis = newGradeBasis;
    self.CreateHodgeArrays();
```

```
end


events
    ChangeBasisEvent
end
```

Here is the data class that passes change of basis data to Multivectors.

```
classdef (ConstructOnLoad) ChangeBasisData < event.EventData
   properties
       oldBasis
       newBasis
   end
   methods
       function eventData = ChangeBasisData(oldBasis,newBasis)
           eventData.oldBasis = oldBasis;
           eventData.newBasis = newBasis;
      end
   end
end
```

We also need to add the callback function that will be called when the 'ChangeBasisEvent' is notified. This code snippet is added to the methods section of Multivector

```
function basisChangeCallback(self, ~, event)
    if(self.type == MultivectorType.Covector)
        self.components = (self.components/event.oldBasis{self.degree})*
                                              event.newBasis{self.degree};
    elseif(self.type == MultivectorType.Contravector)
        self.components = event.newBasis{self.degree}\
                     (event.oldBasis{self.degree}*self.components);
    end
end
```

Note the difference in how multivector components change between covectors and contravectors. Since covectors change with the basis, we convert them to standard basis by right multiplying by inv(oldBasis), which here is implemented as 'self.components/event.oldBasisself.degree)'. From there we compute the new components by multiplying by newBasis. Contravectors change inversely with the basis, therefore the operations are reversed. These dual transformations can be combined into one matrix by matrix multiplication in Exterior Algebra before being sent to Multivectors, however I preferred the visual representation of changing to and from standard basis. Both implementations would work fine, and combining the matrix transformations beforehand will perform better as the number of concurrent

Multivectors increases.

3 Wedge Product

3.1 Overview

As mentioned previously, the wedge product is a bilinear antisymmetric operation that takes two multivectors of grade i and j respectively and creates a new multivector of grade $k = i + j$. The antisymmetric property manifests itself in the following ways. Since we defined previously in (2) that the basis 1-vectors wedged to form a k-vector are written in ascending order by index, given a k-vector of wedged 1-vectors $Ve_1 \wedge e_2, \wedge ... \wedge e_k$ shuffled randomly, we would need to reorder them in order to find the true component V with regards to the basis vectors. Here is an example for a 4-vector.

$$Ve_1 \wedge e_4 \wedge e_3 \wedge e_2 = -Ve_1 \wedge e_3 \wedge e_4 \wedge e_2 = Ve_1 \wedge e_3 \wedge e_2 \wedge e_4 = -Ve_1 \wedge e_2 \wedge e_3 \wedge e_4 \quad (16)$$

This is important in the wedge product, where wedged 1-vectors are often shuffled right after computation, and need to be readjusted. In general, given f flips needed to rearrange the 1-vectors, the component needs to be adjusted by $(-1)^f$.

3.2 Implementation

The wedge product is only defined on multivectors of the same type (covariant or contravariant), so we do a quick check at the beginning of the function. Additionally, any multivector formed by the wedge product with grade higher than N is identically zero. This is due to anti-symmetry, which leads to the identity $e_a \wedge e_a = 0$. Since in a N dimensional space there are exactly N basis 1-vectors, any wedge involving more than N 1-vectors contains duplicate 1-vectors and is identically 0. We can check for this at the beginning of the function and warn the user if this happens.

To find the sign of the resulting wedge, we can use the formula $(-1)^f$. This requires us to find the number of flips, which can be obtained via the MatLab sort function. The sort function returns a permutation sequence that shows how the elements were swapped. We can permute the rows of an identity matrix with this permutation sequence and take the derivative. One of the properties defining the determinant is: If two rows of a matrix A are interchanged to produce matrix B, then $det(B) = -det(A)$. Since the determinant of identity matrix is 1, our result will be either -1 or 1 depending on the number of flips. The wedge product function is below:

```
function mv3 = Wedge(self, mv1, mv2)
```

```
    assert(isa(mv1,"Multivector"), ...
        "Wedge takes Multivectors as inputs")
    assert(isa(mv2,"Multivector"), ...
        "Wedge takes Multivectors as inputs")
    assert(mv1.type == mv2.type, ...
        "Ensure Multivector types match")

    newDegree = mv1.degree + mv2.degree;
    if(newDegree > self.dimension)
        mv3 = 0;
        warning("Wedge Product Resulted in Multivector Degree %d, returning zero.", new
        return
    end

    xb = self.permuteBasis{mv1.degree};
    yb = self.permuteBasis{mv2.degree};
    zb = self.permuteBasis{newDegree};

    x = mv1.components;
    y = mv2.components;
    z = zeros(size(zb,1),1);

    I = eye(newDegree);

    for n = 1:size(xb,1)
        for m=1:size(yb,1)
            [bnm, perm] = sort([xb(n,:) yb(m,:)]);
            if prod(diff(bnm)) ~= 0
                [member,k] = ismember(bnm,zb,'rows');
                if member
                    z(k) = z(k)+x(n)*y(m)*det(I(:,perm));
                end
            end
        end
    end
    mv3 = Multivector(self, newDegree, z, mv1.type);
end
```

4 Inner Product

4.1 Overview

In order for us to accomplish meaningful computations with connections and applications to the real world, we must define geometry, which is something not native to vector spaces. This can be done through the introduction of the inner product. The inner product is defined as a symmetric bi-linear tensor that takes in two multivectors X,Y, and returns a scalar $\langle X, Y \rangle$. This product defines geometry through the following equality

$$\langle X, Y \rangle = |X|\,|Y|\,cos(\theta) \tag{17}$$

where $|X|$ is the length of multivectors X defined by the inner product. From (17) we can define length and angle in generality as

$$|X| = \sqrt{\langle X, X \rangle} \tag{18}$$

$$\theta = cos^{-}1\left(\frac{\langle X, Y \rangle}{|X||Y|}\right) \tag{19}$$

We can represent the inner product tensor as a real valued NxN matrix M, such that the following is true

$$\langle X, Y \rangle = X'\,M\,Y \tag{20}$$

The dot product, commonly taught in undergraduate physics and math classes, is a special case of this inner product tensor, where M is the identity matrix. This reduces the computation of $\langle X, Y \rangle$ to the simple matrix multiplication of $X'\,Y$.

4.2 Higher Grade Metrics

The inner product above has only been defined for 1-vectors. We can define an inner product on 2-vectors in the following way: Given 2-vectors $X = e_a \wedge e_b$ and $Y = e_c \wedge e_d$, the inner product $\langle X, Y \rangle$ is:

$$\langle X, Y \rangle = det\left(\begin{vmatrix} \langle e_a, e_c \rangle & \langle e_a, e_d \rangle \\ \langle e_b, e_c \rangle & \langle e_b, e_d \rangle \end{vmatrix}\right) \tag{21}$$

One must be careful when parsing this notation. X and Y are 2-vectors and are fundamentally different objects than 1-vectors such as e_a. The inner product $\langle X, Y \rangle$ is therefore a different, albeit related operation than the inner product $\langle e_a, e_d \rangle$. However, the concepts behind both are identical, since both are symmetric bi-linear tensors and can be represented by a matrix called the metric. The 2-vector metric will be of the size $\binom{n}{2}$x$\binom{n}{2}$, and in general the k-vector metric will be of size $\binom{n}{k}$x$\binom{n}{k}$.

The elements of the k-grade metric matrix are defined in the same way as the elements of the k-grade basis matrix in 2.3:

$$M^k_{a\,b} = det(M([\text{perm}(Rk_a);\, \text{perm}(Rk_b)]))$$

where $M([\text{perm}(Rk_a);\, \text{perm}(Rk_b)])$ is the submatrix of M constructed using rows from $\text{perm}(Rk_a)$ and columns from $\text{perm}(Rk_b)$.

4.3 Change of Basis

Complications arise when changing the basis of our multivector spaces. To illustrate this, consider (20) under a change of basis. Let X and Y be contravariant multivectors changed by a change of basis matrix P. Our inner product becomes the following:

$$\left\langle P^{-1}X, P^{-1}Y \right\rangle = X^T (P^{-1})^T\, M\, P^{-1} Y \tag{22}$$

This is not the same answer as (20), which is a major problem, as the inner product and definitions of geometry should be invariant to basis. This can be solved by letting the metric M change with the basis alongside multivectors. Looking at (22), we can see that if X and Y are contravariant multivectors under a change of basis matrix P, M should become M' such that $M' = P^T\, M\, P$. Recomputing (22) with this new metric we find:

$$\left\langle P^{-1}X, P^{-1}Y \right\rangle = X^T (P^{-1})^T\, (P^T\, M\, P)\, P^{-1} Y = X'\, M\, Y \tag{23}$$

Our inner product is now invariant to changes in basis! We can quickly compute how the metric should change if X and Y are covariant multivectors.

$$\langle XP, YP \rangle = XP\, M\, P^T Y^T \tag{24}$$

M' should therefore equal $P^{-1}\, M\, (P^{-1})^T$. We verify this:

$$\langle XP, YP \rangle = XP\, (P^{-1}\, M\, (P^{-1})^T)\, P^T Y^T = X\, M\, Y^T \tag{25}$$

The calculations here show that in performing the inner product, we are essentially transforming our input multivectors into their standard basis components, and computing the inner product with the original metric. This informs us that we should seek to store the standard basis metric for each grade, and use the corresponding pre-computed grade basis matrix to transform it when calculating the inner product.

4.4 Implementation

4.4.1 Validation

There are some restrictions on the metric matrix. The 1-vector metric must be NxN to allow for vector multiplication. The metric must also be symmetric, as

$< X, Y >=< Y, X >$. Here is the MatLab function that validates an input metric matrix m.

```
function metric = CheckMetric(m, dim)
    if(issparse(m))
        m = full(m);
    end
    metric = m;
    assert(size(metric,2) == size(metric,1), "Please use a square matrix as " + ...
        "your metric\nCurrent matrix is [%d by %d]",size(metric,1), size(metric,2))
    assert(size(metric,1) == dim, "Metric must have size = dimension\n" + ...
            "Current matrix has size %d, Algebra dim is %d",size(metric,1),dim)
    assert(issymmetric(metric), "Metric must be symmetric.")
end
```

4.4.2 Higher-Grade Metrics

The code for generating higher-grade metrics is essentially identical to the code for generating higher-grade basis, with one exception. Because the metric is symmetrical, we only need to compute the upper triangular entries of M^k, and copy the values into the lower triangular entries. The code for this function is below:

```
function metric = CreateGradeMetrics(self, m)
    metric = cell([self.dimension 1]);
    metric{1} = m;
    for k=2:self.dimension
        kbasis = self.permuteBasis{k};
        s2 = size(kbasis,1);
        kmetric = zeros(s2);
        for x=1:s2
            for y=x:s2
                a = kbasis(x,:);
                b = kbasis(y,:);
                submatrix = zeros(k);
                for i=1:k
                    for j=1:k
                        submatrix(i,j) = m(a(i),b(j));
                    end
                end
                d = det(submatrix);
                kmetric(x,y) = d;
                kmetric(y,x) = d;
            end
        end
```

```
        metric{k} = kmetric;
    end
end
```

4.4.3 Inner Product

The inner product is only defined on multivectors of the same grade. Therefore, we will need to do an additional round of validation to ensure the two input multivectors are of the same grade. For the inner product computations, we need to first adjust the metric by the change of basis matrix based on the types of the input multivectors (covariant or contravariant). From there the computation is a simple vector matrix vector multiplication. The functions for adjusting the metric and calculating the inner product are below:

```
function m = GetMetric(self, grade, mv1Type, mv2Type)
    m = self.gradeMetric{grade};
    Q = self.gradeBasis{grade};
    if mv1Type == MultivectorType.Covector
        m = Q\m;
    else
        m = Q'*m;
    end

    if mv2Type == MultivectorType.Covector
        m = m/(Q');
    else
        m = m*Q;
    end
end


function result = InnerProduct(self, mv1, mv2)
    assert(isa(mv1,"Multivector"), ...
        "InnerProduct takes Multivectors as inputs")
    assert(isa(mv2,"Multivector"), ...
        "InnerProduct takes Multivectors as inputs")
    assert(mv2.degree == mv1.degree, ...
        "Multivector degrees do not match")

    x = mv1.components;
    y = mv2.components;
    m = self.GetMetric(mv1.degree, mv1.type, mv2.type);

    n = size(m,1);
```

```
    result = 0;

    if ~isrow(x)
        x = x';
    end
    if isrow(y)
        y = y';
    end

    result = x*m*y;
end
```

5 Hodge Star

5.1 Overview

The Hodge star is a linear mapping defined by the following relation. Given a contravariant multivector X of grade k, define a multivector Y of grade $N-k$ with components $[Y^1\ Y^2\ ...\ Y^{\binom{N}{(N-k)}}]$. It is interesting and useful to note that $\binom{N}{(N-k)} = \binom{N}{k}$, therefore the number of basis multivectors in (k)-vectors and $(N-k)$-vectors are equal. The Hodge star of X is defined by the following equality:

$$X \wedge Y = \langle \star(X), Y \rangle\, e_1 \wedge e_2 ... \wedge e_N \tag{26}$$

For the rest of this section, we will define the following quantities:

1. $k2 = N - k$
2. $s = \binom{N}{(N-k)} = \binom{N}{k}$

Since the Hodge star is linear, we can focus on X being individual basis k-vectors. The quantity $X \wedge Y$ always results in exactly one component of Y surviving, since there is only one combination of the k basis and the $k2$ basis that does not go to zero in the wedge product. Given that X is the basis k-vector with index Rk_i, the index of the surviving Y $k2$-vector is $a = Rk2_{s-i}$. Therefore, the left side of 26 becomes:

$$X \wedge Y = (-1)^f\, Y^a\, e_1 \wedge e_2 ... \wedge e_N \tag{27}$$

We can equate like coefficients on both sides of (26) now. This leads us to the equality:

$$(-1)^f\, Y^a = \langle \star(X), Y \rangle \tag{28}$$

Recall that the inner product is a tensor and can be represented by a matrix M. (28) now becomes:

$$(-1)^f\, Y^a = (\star(X))^T\, M\, Y \tag{29}$$

Consider that $(\star(X))^T M Y$ results in the sum:

$$\sum_{i=1}^{N}\sum_{j=1}^{N} X^i M_{ij} Y^j$$

We know that we want the sums $\sum_{i=1}^{N} \star(X)^i M_{ij} Y^j = 0$ when $j \neq a$, and $\sum_{i=1}^{N} \star(X)^i M_{ij} Y^j = Y^j$ when $j = a$. We can create an equivalent expression from this observation:

$$(-1)^f I_{:,a} = \star(X)^T M \tag{30}$$

Where $I_{:,a}$ is the a-th column vector of the identity matrix. This is a trivially solvable system for $\star(X)$.

5.2 Change of Basis

Since the Hodge star depends on the inner product, the Hodge star values above must be recalculated whenever the basis changes. This the same process as above, but now using the changed metric/inner product.

5.3 Implementation

We can compute the $\star(X)$ values above and construct a matrix HodgeArray, using $\star(X)$ values as the rows for covariant version and the columns for the contravariant version, with one for each grade basis. With these matrices we can now find the Hodge star of any k-vector by multiplying it with the corresponding grade HodgeArray. The method for finding $\star(X)$ values is the same as described above. The functions to generate the HodgeArray matrices and to find the Hodge star are below:

```
function hodge = CreateHodgeArrays(self)
    %Hodge star is invariant of basis. These hodge arrays are
    %calculated from the metric in standard basis

    %Hodge cells is 2xn
    % first row is covariant hodge values
    % second row is contravariant hodge values
    % new index after hodge is n-i+1
    dim = self.dimension;
    hodge = cell(2,dim-1);
    options = 1:dim;

    for k = 1:dim-1
        basis = self.permuteBasis{k};
        n = size(basis,1);
```

```
        hCo = zeros(n);
        hContra = zeros(n);

        mCo = self.GetMetric(dim-k, MultivectorType.Covector,
                                    MultivectorType.Covector);
        mContra = self.GetMetric(dim-k, MultivectorType.Contravector,
                                        MultivectorType.Contravector);

        I = eye(dim);
        for i=1:n
            starIndex = n-i+1;
            sb = setdiff(options, basis(i,:));
            [~, perm] = sort([basis(i,:) sb]);
            %Hodge star indexes are reversed of normal index
            sign = det(I(:,perm));

            YVector = zeros(1,n);
            YVector(starIndex) = sign;
            hCo(i,:) = (YVector/mCo);
            hContra(:,i) = (YVector/mContra);
        end
        hodge{1, k} = hCo;
        hodge{2, k} = hContra;
    end
end


function result = HodgeStar(self, mv)
    newDegree = self.dimension - mv.degree;
    %Go Component-wise, multiply
    if(mv.type == MultivectorType.Covector)
        h = self.hodgeArr{1, mv.degree};
        cStar = mv.components*h;
        result = self.CreateMultivectorCurrentBasis(newDegree, cStar, mv.type);
    else
        h = self.hodgeArr{2, mv.degree};
        cStar = h*mv.components;
        result = self.CreateMultivectorCurrentBasis(newDegree, cStar, mv.type);
    end
end
```