A SageTeX Hypermatrix Algebra Package

Edinah K. Gnang[*], Ori Parzanchevski[†], Yuval Filmus[‡]

March 15, 2014

Abstract

We describe here a rudimentary sage [S6] implementation of the Bhattacharya-Mesner hypermatrix algebra package.

1 Introduction

The current package implements very basic features of the Bhattacharya-Mesner *hypermatrix* algebra. A hypermatrix denotes a finite set of complex numbers each of which is indexed by members of an integer cartesian product set of the form $\{0, \cdots, (n_0 - 1)\} \times \cdots \times \{0, \cdots, (n_{l-1} - 1)\}$. Such a hypermatrix is said to be of order l or simply an l-hypermatrix for short. The algebra and the spectral analysis of hypermatrices arise as a natural generalization of matrix algebra. Important hypermatrix results available in the literature are concisely surveyed in [L], the reader is also refered to [LQ] for a more detail survey on the spectral analysis of hypermatrices. The hypermatrix algebra discussed here differs from the hypermatrix algebras surveyed in [L] in the fact that the hypermatrix algebra considered here centers around the Bhattacharya-Mesner hypermatrix product operation introduced in [BM1, BM2, B] and followed up in [GER]. Although the scope of the Bhattacharya-Mesner algebra extends to hypermatrices of all finite integral orders, the package will be mostly geared towards 3-hypermatrices.

2 The Hypermatrix Sage Package

We try here to simultaneously follow precepts of the New Jersey school of experimental mathematics initiated by Doron Zeilberger [Z] and the fundamental paradigm of litterate programming pioneered by Donald Knuth[K] to discuss various computational aspects of the Bhattacharya-Mesner 3-hypermatrix algebra. We therefore present here a very rudimentary SageTeX[S6] implementation of a hypermatrix package. The proposed package is available through the source code for the current document either in the format of a LyX file or alternatively as a TeX file or an independent sage file.
Our implementation will be concerned with generic 3-hypermatrices and consequently we will often work with symbolic expressions. The implementation starts out by describing procedures which enable us to generate symbolic matrices and hypermatrices of desired size, order and with other additional properties. Throughout the package, the data structure used will be a lists.

```
def MatrixGenerate(nr, nc, c):
    """
    Generates a list of lists associated with a symbolic nr x nc
    matrix using the input character c followed by indices.

    EXAMPLES:
    ::
        sage: M = MatrixGenerate(2, 2, 'm'); M
        [[m00, m01], [m10, m11]]
```

[*]School of Mathematics, Institute for Advanced Study
[†]School of Mathematics, Institute for Advanced Study
[‡]School of Mathematics, Institute for Advanced Study

```
        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        # Setting the dimensions parameters.
        n_q_rows = nr
        n_q_cols = nc

        # Test for dimension match
        if n_q_rows > 0 and n_q_cols > 0:
            # Initialization of the hypermatrix
            q = []
            for i in range(n_q_rows):
                q.append([])
            for i in range(len(q)):
                for j in range(n_q_cols):
                    # Filling up the matrix
                    (q[i]).append(var(c+str(i)+str(j)))
            return q

        else :
            raise ValueError, "Input dimensions "+\
    str(nr)+" and "+str(nc)+" must both be non-zero positive integers."
```

in addition we implement a similar procedure for generating symbolic symmetric matrices

```
    def SymMatrixGenerate(nr, c):
        """
        Generates a list of lists associated with a symbolic nr x nc
        symmetric matrix using the input character c followed by
        indices.

        EXAMPLES:
        ::
            sage: M = SymMatrixGenerate(2, 'm'); M
            [[m00, m01], [m10, m11]]

        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        # Setting the dimensions parameters.
        n_q_rows = nr
        n_q_cols = nr

        # Test for dimension match
        if n_q_rows > 0 and n_q_cols > 0:
            # Initialization of the hypermatrix
            q = []
            for i in range(n_q_rows):
                q.append([])
            for i in range(len(q)):
                for j in range(n_q_cols):
                    # Filling up the matrix
                    (q[i]).append(var(c+str(min(i,j))+str(max(i,j))))
            return q
            else :
                raise ValueError, "Input dimensions "+\
        str(nr)+" must be a non-zero positive integers."
```

The two procedures implemented above for generating symbolic lists will typically be used in conjunction with the Sage[S6] Matrix class over symbolic rings as illustrated

$$\mathbf{M}_1 = \text{Matrix(SR,MatrixGenerate(2,3,'m'))} = \left(\begin{array}{ccc} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \end{array}\right). \tag{1}$$

$$\mathbf{M}_2 = \text{Matrix(SR,SymMatrixGenerate(2,'m'))} = \left(\begin{array}{cc} m_{00} & m_{01} \\ m_{01} & m_{11} \end{array}\right). \tag{2}$$

We implement similar procedures for generating symbolic hypermatrices of desired order and size.

```
    def HypermatrixGenerate(*args):
        """
        Generates a list of lists associated with a symbolic arbitrary
        hypematrix of order and size specified by the input.

        EXAMPLES:
        ::
            sage: M = HypermatrixGenerate(2, 2, 2, 'm'); M

         AUTHORS:
        - Edinah K. Gnang, Ori Parzanchevski and Yuval Filmus
        """
        if len(args) == 1:
            return var(args[0])
        return [apply(\
    HypermatrixGenerate,args[1:-1]+(args[-1]+str(i),)) for i in range(args[0])]
```

The procedures implemented above illustrate the use of lists for representing hypermatrices. We show bellow for convenience of the reader the output of the function call

$$\mathbf{T} = \text{HypermatrixGenerate(2, 2, 2, 't')} = \left[\left[\left[t_{000}, t_{001}\right], \left[t_{010}, t_{011}\right]\right], \left[\left[t_{100}, t_{101}\right], \left[t_{110}, t_{111}\right]\right]\right]. \tag{3}$$

In connection with the spectral decomposition of 3-hypermatrices we discuss the implemention of a procedure which generates the desired size symbolic 3-hypermatrices with entries symmetric under cyclic permutation of the hypermatrix indices.

```
    def SymHypermatrixGenerate(nr, c):
        """
        Generates a list of lists associated with a symbolic nr x nc x nd
        third order hypematrix using the input character c followed by
        indices.

        EXAMPLES:
        ::
            sage: M = SymHypermatrixGenerate(2, 'm'); M

        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        # Setting the dimensions parameters.
    n_q_rows = nr
    n_q_cols = nr
    n_q_dpts = nr

    # Test for dimension match
    if n_q_rows > 0 and n_q_cols > 0 and n_q_dpts >0:
        # Initialization of the hypermatrix
        q = []
        for i in range(n_q_rows):
            q.append([])
        for i in range(len(q)):
            for j in range(n_q_cols):
                (q[i]).append([])
        for i in range(len(q)):
            for j in range(len(q[i])):
                for k in range(n_q_dpts):
                    if i==j or i==k or j==k:
                        (q[i][j]).append(\
var(c+str(min(i,j,k))+str(i+j+k-min(i,j,k)-max(i,j,k))+str(max(i,j,k))))
                    else:
                        if i == min(i,j,k) and k == max(i,j,k):
                            (q[i][j]).append(\
var(c+str(min(i,j,k))+str(i+j+k-min(i,j,k)-max(i,j,k))+str(max(i,j,k))))
                        elif k == min(i,j,k) and j == max(i,j,k):
                            (q[i][j]).append(\
var(c+str(min(i,j,k))+str(i+j+k-min(i,j,k)-max(i,j,k))+str(max(i,j,k))))
                        elif i == max(i,j,k) and j == min(i,j,k):
                            (q[i][j]).append(\
var(c+str(min(i,j,k))+str(i+j+k-min(i,j,k)-max(i,j,k))+str(max(i,j,k))))
                        else:
                            (q[i][j]).append(\
var(c+str(i+j+k-min(i,j,k)-max(i,j,k))+str(min(i,j,k))+str(max(i,j,k))))
        return q

    else :
        raise ValueError, "Input dimensions "+\
str(nr)+" must be a non-zero positive integer."
```

We illustrate the use of the procedure by showing the output of the following function call

$$\mathbf{S} = \text{SymHypermatrixGenerate}(2, \text{'s'}) = \left[\left[\left[s_{000}, s_{001}\right], \left[s_{001}, s_{011}\right]\right], \left[\left[s_{001}, s_{011}\right], \left[s_{011}, s_{111}\right]\right]\right]. \tag{4}$$

We also implement a procedure for canonically stripping down the 3-hypermatrix (encoded as a list of list) to a simple list of symbolic variables in a similar spirit as the matrix vectorization operation.

```
    def HypermatrixVectorize(A):
        """
        Outputs our canonical vectorization of
        the input hypermatrices A.

        EXAMPLES:
        ::
            sage: M = HypermatrixVectorize(A); M
            AUTHORS:
            - Edinah K. Gnang and Ori Parzanchevski
            """
            # Setting the dimensions parameters.
            n_q_rows = len(A)
            n_q_cols = len(A[0])
            n_q_dpts = len(A[0][0])

            # Test for dimension match
            if n_q_rows>0 and n_q_cols>0 and n_q_dpts>0:
                # Initialization of the hypermatrix
                q = []
                for i in range(n_q_rows):
                    for j in range(n_q_cols):
                        for k in range(n_q_dpts):
                            q.append(A[i][j][k])
                return q

            else :
                raise ValueError, "The Dimensions non zero."
```

The implementation of the hypermatrix vectorization procedure concludes the implementation of procedure for generating and formating symbolic 3-hypermatrices.

The next part of the package will discuss the implementation of procedures which enable us to perform very basic operations on 3-hypermatrices starting with the addition operation

```
    def HypermatrixAdd(A, B):
        """
        Outputs a list of lists corresponding to the sum of
        the two input hypermatrices A, B of the same size

        EXAMPLES:
        ::
            sage: M = HypermatrixAdd(A, B); M


        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        # Setting the dimensions parameters.
        n_q_rows = len(B)
        n_q_cols = len(B[0])
        n_q_dpts = len(B[0][0])

        # Test for dimension match
        if n_q_rows==len(A) and n_q_cols==len(A[0]) and n_q_dpts==len(A[0][0]):
            # Initialization of the hypermatrix
            q = []
            for i in range(n_q_rows):
                q.append([])
            for i in range(len(q)):
                for j in range(n_q_cols):
                    (q[i]).append([])
        for i in range(len(q)):
            for j in range(len(q[i])):
                for k in range(n_q_dpts):
                    (q[i][j]).append(A[i][j][k]+B[i][j][k])
        return q

    else :
        raise ValueError, "The Dimensions of the input hypermatrices must match."
```

quite similarly we implement the 3-hypermatrix hadamard product procedure

```
def HypermatrixHadamardProduct(A, B):
    """
    Outputs a list of lists associated with the addtion of
    the two input hypermatrices A and B

    EXAMPLES:
    ::
        sage: M = HypermatrixHadamardProduct(A, B); M


    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
    # Setting the dimensions parameters.
    n_q_rows = len(A)
    n_q_cols = len(A[0])
    n_q_dpts = len(A[0][0])

    # Test for dimension match
    if n_q_rows==len(A) and n_q_cols==len(A[0]) and n_q_dpts==len(A[0][0]):
        # Initialization of the hypermatrix
        q = []
        for i in range(n_q_rows):
            q.append([])
        for i in range(len(q)):
            for j in range(n_q_cols):
                (q[i]).append([])
        for i in range(len(q)):
            for j in range(len(q[i])):
                for k in range(n_q_dpts):
                    (q[i][j]).append(A[i][j][k]*B[i][j][k])
        return q

    else :
        raise ValueError, "The Dimensions of the input hypermatrices must match."
```

We illustrate the usage of the two procedures implemented above

$$\mathbf{S}+\mathbf{T} = \text{HypermatrixAdd(SymHypermatrixGenerate(2,'s'),HypermatrixGenerate(2,2,2,'t'))} =$$

$$[[[s_{000}+t_{000}, s_{001}+t_{001}], [s_{001}+t_{010}, s_{011}+t_{011}]], [[s_{001}+t_{100}, s_{011}+t_{101}], [s_{011}+t_{110}, s_{111}+t_{111}]]] \tag{5}$$

and

$$\mathbf{S}\star\mathbf{T} = \text{HypermatrixHadamardProduct(SymHypermatrixGenerate(2,'s'),HypermatrixGenerate(2,2,2,'t'))} =$$

$$[[[s_{000}t_{000}, s_{001}t_{001}], [s_{001}t_{010}, s_{011}t_{011}]], [[s_{001}t_{100}, s_{011}t_{101}], [s_{011}t_{110}, s_{111}t_{111}]]]. \tag{6}$$

Furthermore, we implement the procedure for multiplying a 3-hypermatrix by a scalar.

```
def HypermatrixScale(A, s):
    """
    Outputs a list of lists associated with product of the
    input scalar s with the input hypermatrix A.

    EXAMPLES:
    ::
        sage: M = HypermatrixScale(A, 3); M


    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
    # Setting the dimensions parameters.
    n_q_rows = len(A)
    n_q_cols = len(A[0])
    n_q_dpts = len(A[0][0])

    # Initialization of the hypermatrix
    q = []
    for i in range(n_q_rows):
        q.append([])
    for i in range(len(q)):
        for j in range(n_q_cols):
            (q[i]).append([])
    for i in range(len(q)):
        for j in range(len(q[i])):
            for k in range(n_q_dpts):
                (q[i][j]).append(A[i][j][k]*s)
    return q
```

typically used as follows

$$3\,\mathbf{T} = \text{HypermatrixScale(HypermatrixGenerate(2,2,2,'t'),3)} =$$
$$[[[3\,t_{000}, 3\,t_{001}], [3\,t_{010}, 3\,t_{011}]], [[3\,t_{100}, 3\,t_{101}], [3\,t_{110}, 3\,t_{111}]]]. \tag{7}$$

similarly, we implement the entry-wise exponentiation bellow

```
def HypermatrixEntryExponent(A, s):
    """
    Outputs a list of lists associated with product of the
    scalar s with the hypermatrix A.

    EXAMPLES:
    ::
        sage: M = HypermatrixEntryExponent(A, 3); M


    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
```

```
        # Setting the dimensions parameters.
        n_q_rows = len(A)
        n_q_cols = len(A[0])
        n_q_dpts = len(A[0][0])

        # Initialization of the hypermatrix
        q = []
        for i in range(n_q_rows):
            q.append([])
        for i in range(len(q)):
            for j in range(n_q_cols):
                (q[i]).append([])
        for i in range(len(q)):
            for j in range(len(q[i])):
                for k in range(n_q_dpts):
                    (q[i][j]).append((A[i][j][k])^s)
        return q
```

due to the fact that the exponentiation operation is noncommutative we also implement the entry-wise exponentiation operation where the input is to be taken as basis for the exponentiation computation.

```
    def HypermatrixEntryExponentB(s, A):
        """
        Outputs a list of lists associated with product of the
        scalar s with the hypermatrix A.

        EXAMPLES:
        ::
            sage: M = HypermatrixEntryExponentB(3,A); M

        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        # Setting the dimensions parameters.
        n_q_rows = len(A)
        n_q_cols = len(A[0])
        n_q_dpts = len(A[0][0])

        # Initialization of the hypermatrix
        q = []
        for i in range(n_q_rows):
            q.append([])
        for i in range(len(q)):
            for j in range(n_q_cols):
                (q[i]).append([])
        for i in range(len(q)):
            for j in range(len(q[i])):
                for k in range(n_q_dpts):
                    (q[i][j]).append(s^(A[i][j][k]))
        return q
```

At the heart of the Mesner-Bhattacharya 3-hypermatrix algebra lies the ternary non-associative hypermatrix product operation[BM2, BM1]. We provide here a naive implementation of the Mesner-Bhattacharya 3-hypermatrix product. We may briefly recall that the product is defined for input hypermatrices $\mathbf{A}$ of dimensions $m \times k \times p$, $\mathbf{B}$ of dimensions $m \times n \times k$

and the matrix $\mathbf{C}$ of dimension $k \times n \times p$, to result into an $m \times n \times p$ hypermatrix with entries specified by

$$[\circ(\mathbf{A}, \mathbf{B}, \mathbf{C})]_{i,j,k} = \sum_{0 \le t < k} a_{itk} \, b_{ijt} \, c_{tjk} \tag{8}$$

```
def HypermatrixProduct(A, B, C):
    """
    Outputs a list of lists associated with the ternary
    non associative Bhattacharya-Mesner product of the
    input hypermatrices A, B and C.

    EXAMPLES:
    ::
        sage: M = HypermatrixProduct(A, B, C); M

    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
    # Setting the dimensions parameters.
    n_a_rows = len(A)
    n_a_cols = len(A[0])
    n_a_dpts = len(A[0][0])

    n_b_rows = len(B)
    n_b_cols = len(B[0])
    n_b_dpts = len(B[0][0])

    n_c_rows = len(C)
    n_c_cols = len(C[0])
    n_c_dpts = len(C[0][0])

    # Test for dimension match
    if n_a_rows==n_b_rows and n_b_cols==n_c_cols and n_c_dpts==n_a_dpts and \
n_a_cols==n_b_dpts and n_b_dpts==n_c_rows:
        # Initialization of the hypermatrix
        q = []
        for i in range(n_a_rows):
            q.append([])
        for i in range(len(q)):
            for j in range(n_b_cols):
                (q[i]).append([])
        for i in range(len(q)):
            for j in range(len(q[i])):
                for k in range(n_c_dpts):
                    (q[i][j]).append(\
sum([A[i][l][k]*B[i][j][l]*C[l][j][k] for l in range(n_a_cols)]))
        return q

    else :
        raise ValueError, "Hypermatrix dimension mismatch."
```

In connection with the computation of the spectral elimination ideals, we implement a slight generalization of the Mesner-Bhattacharya hypermatrix product hypermatrix product, introduced in [GER]. Recall that the 3-hypermatrix product of input hypermatrices $\mathbf{A}$ of dimensions $m \times l \times p$, $\mathbf{B}$ of dimensions $m \times n \times l$ and the matrix $\mathbf{C}$ of dimension $l \times n \times p$, with non-trivial background $\mathbf{T}$ with dimensions $l \times l \times l$ results in $m \times n \times p$ hypermatrix and in particular the m, n, p of the product is expressed by

$$\left[\circ_{\mathbf{T}}\left(\mathbf{A},\,\mathbf{B},\,\mathbf{C}\right)\right]_{mnp}=\sum_{1\le i\le l}\left(\sum_{1\le j\le l}\left(\sum_{1\le k\le l}a_{mip}\,b_{mnj}\,c_{knp}\,t_{ijk}\right)\right), \tag{9}$$

which is implemented as follows

```
def HypermatrixProductB(A, B, C, D):
    """
    Outputs a list of lists associated with the ternary
    product the input hypermatrices A, B and C with
    background hypermatrix D.

    EXAMPLES:
    ::
        sage: M = HypermatrixProductB(A, B, C, D); M

    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
    # Setting the dimensions parameters.
    n_a_rows = len(A)
    n_a_cols = len(A[0])
    n_a_dpts = len(A[0][0])

    n_b_rows = len(B)
    n_b_cols = len(B[0])
    n_b_dpts = len(B[0][0])

    n_c_rows = len(C)
    n_c_cols = len(C[0])
    n_c_dpts = len(C[0][0])

    n_d_rows = len(D)
    n_d_cols = len(D[0])
    n_d_dpts = len(D[0][0])

    # Test for dimension match
    if \
n_a_rows==n_b_rows and n_b_cols==n_c_cols and n_c_dpts==n_a_dpts and \
n_a_cols==n_b_dpts and n_b_dpts==n_c_rows and n_a_cols==n_d_rows and \
n_a_cols==n_d_cols and n_a_cols==n_d_dpts:
        # Initialization of the hypermatrix
        q = []
        for i in range(n_a_rows):
            q.append([])
        for i in range(len(q)):
            for j in range(n_b_cols):
                (q[i]).append([])
        for i in range(len(q)):
            for j in range(len(q[i])):
                for k in range(n_c_dpts):
                    (q[i][j]).append(\
sum([A[i][l0][k]*B[i][j][l1]*C[l2][j][k]*D[l0][l1][l2] for l0 in range(n_d_rows)\
    for l1 in range(n_d_cols) for l2 in range(n_d_dpts)]))
            return q

        else :
            raise ValueError, "Hypermatrix dimension mismatch."
```

We illustrate bellow, how to initialize and obtain 3-hypermatrix products either with the trivial or arbitrary background hypermatrix. The example discussed here will be for $2 \times 2 \times 2$ hypermatrices.

```
    # We put here together the seperate pieces we have implemented above.
    A = HypermatrixGenerate(2, 2, 2, 'a')
    B = HypermatrixGenerate(2, 2, 2, 'b')
    C = HypermatrixGenerate(2, 2, 2, 'c')
    T = HypermatrixGenerate(2, 2, 2, 't')
    P = HypermatrixProduct(A, B, C)
    Q = HypermatrixProductB(A, B, C, T)
```

from which we obtain that the 0,0,0 entry of the product with trivial background is given by

$$p_{000} = \left[\circ\left(\mathbf{A},\, \mathbf{B},\, \mathbf{C}\right)\right]_{0,0,0} = \mathrm{P}[0][0][0] = a_{000}b_{000}c_{000} + a_{010}b_{001}c_{100} \tag{10}$$

while the 0,0,0 entry of the product with non trivial background is given

$$q_{000} = \left[\circ_{\mathbf{T}}\left(\mathbf{A},\, \mathbf{B},\, \mathbf{C}\right)\right]_{0,0,0} = \mathrm{Q}[0][0][0] = a_{000}b_{000}c_{000}t_{000} + a_{000}b_{000}c_{100}t_{001} + a_{000}b_{001}c_{000}t_{010} + a_{000}b_{001}c_{100}t_{011} +$$
$$a_{010}b_{000}c_{000}t_{100} + a_{010}b_{000}c_{100}t_{101} + a_{010}b_{001}c_{000}t_{110} + a_{010}b_{001}c_{100}t_{111}. \tag{11}$$

We now implement the procedure which generalizes to 3-hypermatrices the notion of matrix transpose. The transpose operation for matrices consists in performing a transposition of matrix indices and this has the effect of simultaneously changing rows vectors into column vectors and column vecors into row vectors. However in the case of 3-hypermatrices there are six possible permutations which can be performed on the indices and among these permutations, the cyclic permutation form a very special subgroup, because cyclic permutations simultaneously map rows vectors to columns vectors and column vectors to depth vectors. As a result, cyclic permutations of the indices should be thought off as operations which are inherent to 3-hypermatrices while the remaining three transpositions are to be thought off as matrix operations.

```
    def HypermatrixCyclicPermute(A):
        """
        Outputs a list of lists associated with the hypermatrix
        with entries index cycliclly permuted.

        EXAMPLES:
        ::
            sage: M = HypermatrixCyclicPermute(A); M


        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        # Setting the dimensions parameters.
        n_q_rows = len(A[0])
        n_q_cols = len(A[0][0])
        n_q_dpts = len(A)

        # Initialization of the hypermatrix
        q = []
    for i in range(n_q_rows):
        q.append([])
    for i in range(len(q)):
        for j in range(n_q_cols):
            (q[i]).append([])
    for i in range(len(q)):
        for j in range(len(q[i])):
            for k in range(n_q_dpts):
                (q[i][j]).append(A[k][i][j])
    return q
```

We illustrate the hypermatrix transpose operation by starting with the 3-hypermatrix

$$\mathbf{A} = \left[\left[\left[a_{000}, a_{001}\right], \left[a_{010}, a_{011}\right]\right], \left[\left[a_{100}, a_{101}\right], \left[a_{110}, a_{111}\right]\right]\right] \tag{12}$$

and showing the result of the transposition

$$\mathbf{A}^T = \text{HypermatrixCyclicPermute(A)} = \left[\left[\left[a_{000}, a_{100}\right], \left[a_{001}, a_{101}\right]\right], \left[\left[a_{010}, a_{110}\right], \left[a_{011}, a_{111}\right]\right]\right]. \tag{13}$$

In connection with 3-hypermatrix spectral decompositions computations, we implement procedure for generating special family of 3-hypermatrices starting with Kronecker delta 3-hypermatrices. The defining properties of the Kronecker delta 3-hypermatrix can be expressed as follows

$$\mathbf{\Delta} = \left(\delta_{ijk} \geq 0\right)_{0 \leq i,j,k < n}, \quad \text{and} \quad \mathbf{\Delta} = \circ\left(\mathbf{\Delta}, \mathbf{\Delta}^{T^2}, \mathbf{\Delta}^{T}\right) \tag{14}$$

and the procedure generating Kronecker delta 3-hypermatrices is implemented as follows

```
    def HypermatrixKroneckerDelta(nr):
        """
        Generates a list of lists associated with the nr x nr x nr
        Kronecker Delta hypermatrix.

        EXAMPLES:
        ::
            sage: M = HypermatrixKroneckerDelta(2); M


        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        # Setting the dimensions parameters.
        n_q_rows = nr
        n_q_cols = nr
        n_q_dpts = nr

        # Test for dimension match
        if n_q_rows > 0 and n_q_cols > 0 and n_q_dpts >0:
            # Initialization of the hypermatrix
            q = []
            for i in range(n_q_rows):
                q.append([])
            for i in range(len(q)):
                for j in range(n_q_cols):
                    (q[i]).append([])
            for i in range(len(q)):
                    for j in range(len(q[i])):
                        for k in range(n_q_dpts):
                            if i==j and i==k:
                                (q[i][j]).append(1)
                            else:
                                (q[i][j]).append(0)
                return q

            else :
                raise ValueError, "Input dimensions "+\
        str(nr)+" must be a non-zero positive integer."
```

Furthermore for some particular numerical routines we implement procedures for initializing hypermatrices so as to have all entries either equal to zero or equal to one

```
    def HypermatrixGenerateAllOne(*args):
        """
        Generates a list of lists associated with the nr x nr x nr
        all one hypermatrix.

        EXAMPLES:
        ::
            sage: M = HypermatrixGenerateAllOne(2,2,2); M


        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        if len(args) == 1:
            return [1 for i in range(args[0])]
        return [apply(HypermatrixGenerateAllOne, args[1:] ) for i in range(args[0])]
```

for initializing all entries to zero we have

```
    def HypermatrixGenerateAllZero(*args):
        """
        Generates a list of lists associated with the nr x nr x nr
        all zero hypermatrix.

        EXAMPLES:
        ::
            sage: M = HypermatrixGenerateAllZero(2,2,2); M


        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        if len(args) == 1:
            return [0 for i in range(args[0])]
        return [apply(HypermatrixGenerateAllZero, args[1:] ) for i in range(args[0])]
```

More interestingly, we implement procedures for generating 3-hypermatrices with binary entries which correspond to the 3-hypermatrix analogue of permutation matrices. Permutation 3-hypermatrices by analogy to permutation matrices effect some prescribed permutations of row slices or column slices or alternatively the depth slices of some specified hypermatrices. The permutation is effected by performing the appropriate sequence hypermatrix products. The procedure which we implement here for generating permutation 3-hypermatrix takes as input a list of integer in the range 0 to $(n-1)$ inclusively whose particular order in the list specify the desired transposition. The procedure outputs the corresponding transposition 3-hypermatrix. The output 3-hypermatrix will be of dimension $n \times n \times n$. We recall from [GER] that permutation hypermatrices corresponding to some transposition $\sigma \in S_n$ is expressed by

$$\mathbf{P}_\sigma = \sum_{1 \le k \le n} \circ \left(\mathbf{1}_{n \times n \times n},\, \mathbf{1}_{n \times n \times n},\, \mathbf{e}_k \otimes \mathbf{e}_k \otimes \mathbf{e}_{\sigma(k)} \right) \tag{15}$$

```
def HypermatrixPermutation(s):
    """
    Generates a list of lists associated with the permutation
    hypermatrix deduced from sigma.

    EXAMPLES:
    ::
        sage: M = HypermatrixPermutation([0,2,1]); M


    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
    n = len(s)
    # Setting the dimensions parameters.
    n_q_rows = n
    n_q_cols = n
    n_q_dpts = n

    # Test for dimension match
    if n_q_rows > 0 and n_q_cols > 0 and n_q_dpts >0:
        # Initialization of the hypermatrix
        q = []
        T = HypermatrixKroneckerDelta(n)
        U = HypermatrixGenerateAllOne(n,n,n)
        Id= HypermatrixProduct(U,U,T)
        Id= HypermatrixCyclicPermute(Id)
        for i in range(n):
            q.append(Id[s[i]])
        return HypermatrixCyclicPermute(HypermatrixCyclicPermute(q))

    else :
        raise ValueError, "Input dimensions "+\
str(n)+" must be a non-zero positive integer."
```

It is important to note that because of the associativity symmetry breaking, it is important to express the permutations as product of transpositions. We also illustrate how the 3-hypermatrix product effects some desired transposition to the appropriate 3-hypermatrix slices.

```
# the code writen here is merely to put together the peices we have implemented so far.
# Generic Symbolic hypermatrix
A   = HypermatrixGenerateAllZero(3,3,3)
Tmp = HypermatrixGenerate(3, 3, 3, 'a')
for i in range(2):
    for j in range(3):
            for k in range(3):
                A[i][j][k]=Tmp[i][j][k]

    # Initialization of the hypermatrix and it's cyclic permutations
    P  = HypermatrixPermutation([1,0,2])
    Pt =HypermatrixCyclicPermute(P)
    Ptt=HypermatrixCyclicPermute(HypermatrixCyclicPermute(P))

    # Effecting the permutation of ...
    # row slices
    Ar = HypermatrixProduct(Pt,Ptt,A)
    # column slice
    Ac = HypermatrixProduct(A,P,Pt)
    # and depth slices
    Ad = HypermatrixProduct(P,A,Ptt)
```

It follows from the lines of code written above that starting from the $3 \times 3 \times 3$ symbolic 3-hypermatrix

$$\begin{aligned}\mathbf{A} = [&[[a_{000}, a_{001}, a_{002}], [a_{010}, a_{011}, a_{012}], [a_{020}, a_{021}, a_{022}]],\\ &[[a_{100}, a_{101}, a_{102}], [a_{110}, a_{111}, a_{112}], [a_{120}, a_{121}, a_{122}]]]\end{aligned} \tag{16}$$

and for performing the transposition $[1, 0, 2]$, we produced the permutation hypermatrix

$$\mathbf{P}_{[1,0,2]} = [[[0, 1, 0], [1, 0, 0], [0, 0, 1]], [[0, 1, 0], [1, 0, 0], [0, 0, 1]], [[0, 1, 0], [1, 0, 0], [0, 0, 1]]]. \tag{17}$$

In order to effect the transposition to the row slices of $\mathbf{A}$ we compute the product

$$\begin{aligned}\circ\left(\mathbf{P}_{[1,0,2]}^{T}, \mathbf{P}_{[1,0,2]}^{T^2}, \mathbf{A}\right) = [&[[a_{100}, a_{101}, a_{102}], [a_{110}, a_{111}, a_{112}], [a_{120}, a_{121}, a_{122}]],\\ &[[a_{000}, a_{001}, a_{002}], [a_{010}, a_{011}, a_{012}], [a_{020}, a_{021}, a_{022}]]].\end{aligned} \tag{18}$$

furthermore in order to effect the transposition to the column slices of $\mathbf{A}$ we compute the product

$$\begin{aligned}\circ\left(\mathbf{A}, \mathbf{P}_{[1,0,2]}, \mathbf{P}_{[1,0,2]}^{T}\right) = [&[[a_{010}, a_{011}, a_{012}], [a_{000}, a_{001}, a_{002}], [a_{020}, a_{021}, a_{022}]],\\ &[[a_{110}, a_{111}, a_{112}], [a_{100}, a_{101}, a_{102}], [a_{120}, a_{121}, a_{122}]]].\end{aligned} \tag{19}$$

finally in order to effect the same transposition to the depth slices of $\mathbf{A}$ we compute the product

$$\begin{aligned}\circ\left(\mathbf{P}_{[1,0,2]}, \mathbf{A}, \mathbf{P}_{[1,0,2]}^{T^2}\right) = [&[[a_{001}, a_{000}, a_{002}], [a_{011}, a_{010}, a_{012}], [a_{021}, a_{020}, a_{022}]],\\ &[[a_{101}, a_{100}, a_{102}], [a_{111}, a_{110}, a_{112}], [a_{121}, a_{120}, a_{122}]]].\end{aligned} \tag{20}$$

We now implement a procedure for generating 3-hypermatrix analog of diagonal martrices. We recall that just as for matrices the diagonal 3-hypermatrices are slight variation of the identity permutation 3-hypermatrix and their defining equality is expressed by

$$\mathbf{D}^{\star^3} = \circ\left(\mathbf{D}^{T}, \mathbf{D}^{T^2}, \mathbf{D}\right) \tag{21}$$

where $\mathbf{D}^{\star^3}$ denotes the Hadamard cube power of $\mathbf{D}$. The procedure that we implement here for generating a diagonal 3-hypermatrix, takes as input a symmetric generic $n \times n$ symbolic matrix and outputs a $n \times n \times n$ 3-hypermatrix satisfying the defining equation

```
    def DiagonalHypermatrix(Mtrx):
        """
        Outputs a diagonal third order hypermatrix
        constructed using the input square matrix
    to enforce the symmetry constraint we will
    only take entry from the lower triangular
    part of the input matrix.

     EXAMPLES:
    ::
        sage: var('a00, a11, a01')
        sage: Mtrx = Matrix(Sr,[[a00,a01],[a01,a11]])
        sage: d = DiagonalHypermatrix(Mtrx)

    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
    # Initialization of the dimensions
    n = min(Mtrx.nrows(),Mtrx.ncols())
    n_d_rows = n
    n_d_cols = n
    n_d_dpts = n

    # Initialization of the identity permutations hypermatrix
    D = HypermatrixPermutation(range(n))

    # Filling up the entries of the hypermatrix.
    for i in range(n_d_rows):
        for j in range(n_d_cols):
            for k in range(n_d_dpts):
                if (D[i][j][k] != 0):
                    D[i][j][k] = Mtrx[min(i,k),max(i,k)]
    return D
```

We illustrate with the following few lines of codes how to generate a diagonal $2\times2\times2$ hypermatrices and verify their defining identity

```
# Generating a diagonal hypermatrices
Mtrx = Matrix(SR,MatrixGenerate(2, 3,"lambda"))
D  = DiagonalHypermatrix(Mtrx)
Dt = HypermatrixCyclicPermute(D)
Dtt= HypermatrixCyclicPermute(HypermatrixCyclicPermute(D))
Dc = HypermatrixProduct(Dt,Dtt,D)
```

hence

$$\mathbf{D} = [[[\lambda_{00},0],[0,\lambda_{01}]],[[\lambda_{01},0],[0,\lambda_{11}]]] \tag{22}$$

and we observe that

$$\circ\left(\mathbf{D}^{T},\mathbf{D}^{T^2},\mathbf{D}\right) = \text{HypermatrixProduct(Dt, Dtt, D)} = \left[\left[\left[\lambda_{00}^3,0\right],\left[0,\lambda_{01}^3\right]\right],\left[\left[\lambda_{01}^3,0\right],\left[0,\lambda_{11}^3\right]\right]\right] \tag{23}$$

and incidentally has the same entries as the hypermatrix $\mathbf{D}^{\star^3}$

$$\mathbf{D}^{\star^3} = \text{HypermatrixEntryExponent(D, 3)} = \left[\left[\left[\lambda_{00}^3,0\right],\left[0,\lambda_{01}^3\right]\right],\left[\left[\lambda_{01}^3,0\right],\left[0,\lambda_{11}^3\right]\right]\right] \tag{24}$$

We now implement procedures which enables us to constrast 2×2 , $2\times2\times2$, and so on type hypermatrices wich are orthogonal in the sense introduced in [GER].

```
def Orthogonal2x2x2Hypermatrix(t):
    """
    Outputs an orthogonal third order hypermatrix
    of size 2 by 2 by 2.

     EXAMPLES:
    ::
        sage: t=var('t')
        sage: Orthogonal2x2x2Hypermatrix(t)

    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
    return [[[cos(t)^(2/3),sin(t)^(2/3)],[sin(t)^(2/3), cos(t)^(2/3)]],\
[[-sin(t)^(2/3),cos(t)^(2/3)],[sin(t)^(2/3),sin(t)^(2/3)]]]
```

we also present here a parametrization of a subset of $3 \times 3 \times 3$ orthogonal hypermatrix bellow

```
def Orthogonal3x3x3Hypermatrix(t1,t2):
    """
    Outputs an orthogonal third order hypermatrix
    of size 3 by 3 by 3.

     EXAMPLES:
    ::
        sage: t1,t2=var('t1,t2')
        sage: Orthogonal3x3x3Hypermatrix(t1,t2)

    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
    c1=cos(t1)^(2/3)
    s1=sin(t1)^(2/3)
    c2=cos(t2)^(2/3)
    s2=sin(t2)^(2/3)
    return [[[c1,s1*c2,0],[s1*c2,s1*s2,0],[s1*s2,exp(-I*2*pi/3)*c1,0]],\
[[s1*s2,c1,exp(-I*2*pi/3)*s1*c2],[exp(I*2*pi/3)*c1,s1*c2,s1*s2],\
[s1*c2,s1*s2,c1]],[[0,s1*s2,c1],[0,c1,s1*c2],[0,exp(I*2*pi/3)*s1*c2,s1*s2]]]
```

The use of the procedures for generating orthogonal hypermatrices are illustrated bellow

```
theta = var('theta')
Q  = Orthogonal2x2x2Hypermatrix(theta)
Qt = HypermatrixCyclicPermute(Q)
Qtt= HypermatrixCyclicPermute(HypermatrixCyclicPermute(Q))
```

Expressing $2 \times 2 \times 2$ orthogonal hypermatrices in term of the free parameter θ we obtain

$$\mathbf{Q}(\theta) = \left[\left[\left[\cos(\theta)^{\frac{2}{3}}, \sin(\theta)^{\frac{2}{3}}\right], \left[\sin(\theta)^{\frac{2}{3}}, \cos(\theta)^{\frac{2}{3}}\right]\right], \left[\left[-\sin(\theta)^{\frac{2}{3}}, \cos(\theta)^{\frac{2}{3}}\right], \left[\sin(\theta)^{\frac{2}{3}}, \sin(\theta)^{\frac{2}{3}}\right]\right]\right] \tag{25}$$

$$\circ\left(\mathbf{Q}(\theta), [\mathbf{Q}(\theta)]^{T^2}, [\mathbf{Q}(\theta)]^{T}\right) = \text{HypermatrixProduct(Q, Qtt, Qt)} =$$

$$\left[\left[\left[\cos(\theta)^2 + \sin(\theta)^2, 0\right], [0,0]\right], \left[[0,0], \left[0, \cos(\theta)^2 + \sin(\theta)^2\right]\right]\right] \tag{26}$$

We also illustrate the output of the procedure implemented above for generating parametrization for $3 \times 3 \times 3$ orthogonal hypermatrices

```
    # Defining the Parametrization Variables
    theta1,theta2=var('theta1,theta2')
    c1=cos(theta1)^(2/3)
    s1=sin(theta1)^(2/3)
    c2=cos(theta2)^(2/3)
    s2=sin(theta2)^(2/3)

    # Parametrization of a orthogonal hypermatrix
    U  = Orthogonal3x3x3Hypermatrix(theta1,theta2)
    Ut = HypermatrixCyclicPermute(U)
    Utt= HypermatrixCyclicPermute(HypermatrixCyclicPermute(U))
    UUttUt = HypermatrixProduct(U,Utt,Ut)
    for i in range(3):
        for j in range(3):
            for k in range(3):
                UUttUt[i][j][k] = (UUttUt[i][j][k]).simplify_exp()
```

We verify that the obtained $3 \times 3 \times 3$ hypermatrix is indeed orthogonal via the following computation

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{0,0,0} = \cos\left(\theta_2\right)^2\sin\left(\theta_1\right)^2 + \sin\left(\theta_1\right)^2\sin\left(\theta_2\right)^2 + \cos\left(\theta_1\right)^2 \tag{27}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{1,1,1} = \cos\left(\theta_2\right)^2\sin\left(\theta_1\right)^2 + \sin\left(\theta_1\right)^2\sin\left(\theta_2\right)^2 + \cos\left(\theta_1\right)^2 \tag{28}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{2,2,2} = \cos\left(\theta_2\right)^2\sin\left(\theta_1\right)^2 + \sin\left(\theta_1\right)^2\sin\left(\theta_2\right)^2 + \cos\left(\theta_1\right)^2 \tag{29}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{0,0,1} = 0 \tag{30}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{0,0,2} = 0 \tag{31}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{1,1,2} = 0 \tag{32}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{1,1,0} = 0 \tag{33}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{2,2,0} = 0 \tag{34}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{2,2,1} = 0 \tag{35}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{0,1,2} = 0 \tag{36}$$

$$\left[\circ\left(\mathbf{U},\,\mathbf{U}^{T^2},\,\mathbf{U}^{T}\right)\right]_{1,0,2} = 0 \tag{37}$$

In the remaining part of the package, we implement functions which relates to genralizations to hypermatrices of the classic Cayley-Hamilton theorem and to the notion of hypermatrix inversion. We first start by implementing the function which creates a list of hypermatrices corresponding to all the possible product composition of the input hypermatrix **A**.

```
def HypermatrixCayleyHamiltonList(A, n):
    """
    Outpts a list of hypermatrices of all product
    composition of order n from which it follows
    that n must be odd.

     EXAMPLES:
    ::
        sage: A = HypermatrixGenerate(2,2,2,'a')
        sage: L = HypermatrixCayleyHamiltonList(A,3)

    AUTHORS:
    - Edinah K. Gnang and Ori Parzanchevski
    """
    if n == 1:
        return [A]
    else:
        gu = []
        for i in range(1,n,2):
            for j in range(1,n-i,2):
                gu = gu + [HypermatrixProduct(g1,g2,g3) \
for g1 in HypermatrixCayleyHamiltonList(A,i) \
for g2 in HypermatrixCayleyHamiltonList(A,j) \
for g3 in HypermatrixCayleyHamiltonList(A,n-(i+j))]
        return gu
```

It then becomes possible to establish that the dimension of the span of hypermatrix composition powers is maximal for generic $2 \times 2 \times 2$ and $3 \times 3 \times 3$ hypermatrices. The lines of code bellow computes hypermatrix compositions and stacks the resulting hypermatrices into a square matrix and computing the determinant in order to assert that the matrix is full rank.

```
# Initializing an orthogonal hypermatrix
A = Orthogonal2x2x2Hypermatrix(e/pi)

# Initialization of the list
L = HypermatrixCayleyHamiltonList(A,1)+HypermatrixCayleyHamiltonList(A,3)+\
HypermatrixCayleyHamiltonList(A,5)+HypermatrixCayleyHamiltonList(A,7)

# Initializing the index variables
Indx = Set(range(len(L))).subsets(8)

# Initialization of the of the matrix
M = Matrix(RR,identity_matrix(8,8))
cnt = 0
for index in Indx:
    if cnt < 10:
        M = M*Matrix(RR,[HypermatrixVectorize(L[i]) for i in index])
        cnt= cnt+1
    else:
        break

# Defining the Parametrization Variables
c1=cos(e/pi)^(2/3)
s1=sin(e/pi)^(2/3)
c2=cos(pi/e)^(2/3)
```

```
s2=sin(pi/e)^(2/3)

# Defining the unitary hypermatrices
U=[[[c1,s1*c2,0],[s1*c2,s1*s2,0],[s1*s2, exp(-I*2*pi/3)*c1,0]],\
[[s1*s2,c1,exp(-I*2*pi/3)*s1*c2],\
[exp(I*2*pi/3)*c1,s1*c2,s1*s2],[s1*c2,s1*s2,c1]],\
[[0,s1*s2,c1],[0,c1,s1*c2],[0,exp(I*2*pi/3)*s1*c2,s1*s2]]]

Lu = HypermatrixCayleyHamiltonList(U,1)+HypermatrixCayleyHamiltonList(U,3)+\
HypermatrixCayleyHamiltonList(U,5)+HypermatrixCayleyHamiltonList(U,7)+\
HypermatrixCayleyHamiltonList(U,9)

# Initializing the index variables
Indxu = Set(range(len(Lu))).subsets(27)

# Initialization of the of the matrix
Mu = Matrix(CC,identity_matrix(27,27))
cntu = 0
for index in Indxu:
    if cntu < 5:
        Mu = Mu*Matrix(CC,[HypermatrixVectorize(Lu[i]) for i in index])
        cntu = cntu+1
    else:
        break
```

The determinant of resulting matrix is given by

$$\det \mathbf{M} = 2.36118324143482 \times 10^{21}, \tag{38}$$

furthermore for $3 \times 3 \times 3$ we have that

$$\det \mathbf{M}' = 1.67255588988980 \times 10^{43} i \tag{39}$$

The very last piece of the current package corresponds to the hypermatrix pseudo-inversion procedure. The routine that we implement here will be predominantly numerical. The notions of hypermatrix inverse pairs was first proposed in the work of Battacharya and Mesner in [BM2], we follow up by implementing here numerical routine for the computation of pseudo-inverse pairs for $2 \times 2 \times 2$ hypermatrices.

We first start by implementing a constraint formator procedure which formats a list of linear constraints into a system of linear equation in the canonical form $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, the constraint formator will be curcial for formating the linear constraints which arise from the hypermatrix inversion constraints.

```
def ConstraintFormator(CnstrLst, VrbLst):
    """
    Takes as input a List of linear constraints
    and a list of variables and outputs matrix
    and the right hand side vector associate
    with the matrix formulation of the constraints.

    EXAMPLES:
    ::
        sage: x, y = var('x,y')
        sage: CnstrLst = [x+y==1, x-y==2]
        sage: VrbLst = [x, y]
        sage: [A,b] = ConstraintFormator(CnstrLst, VrbLst)

    AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        # Initializing the Matrix
        A=Matrix(CC,len(CnstrLst),len(VrbLst),zero_matrix(len(CnstrLst),len(VrbLst)))
        b=vector(CC, [eq.rhs() for eq in CnstrLst]).column()
        for r in range(len(CnstrLst)):
            for c in range(len(VrbLst)):
                A[r,c] = diff((CnstrLst[r]).lhs(),VrbLst[c])
        return [A,b]

    def ConstraintFormatorII(CnstrLst, VrbLst):
        """
        Takes as input a List of linear constraints
        and a list of variables and outputs matrix
        and the right hand side vector associate
        with the matrix formulation of the constraints.

        EXAMPLES:
        ::
            sage: x, y = var('x,y')
            sage: CnstrLst = [x+y==1, x-y==2]
            sage: VrbLst = [x, y]
            sage: [A,b] = ConstraintFormator(CnstrLst, VrbLst)

        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        # Initializing the Matrix
        A=Matrix(SR,len(CnstrLst),len(VrbLst),zero_matrix(len(CnstrLst),len(VrbLst)))
        b=vector(SR, [eq.rhs() for eq in CnstrLst]).column()
        for r in range(len(CnstrLst)):
            for c in range(len(VrbLst)):
                A[r,c] = diff((CnstrLst[r]).lhs(),VrbLst[c])
        return [A,b]
```

Having implemented the constraint formator we are now ready to implement the 3-hypermatrix pseudo-inversion subroutine. The following equation constitutes the defining property of an inverse pairs for 3-hypermatrices $\mathbf{A}$ and $\mathbf{B}$

$$\forall\,\mathbf{M} \in \mathbb{C}^{n\times n\times n}, \quad \mathbf{M} = \circ\left(\circ\left(\mathbf{M},\mathbf{A},\mathbf{B}\right),\mathbf{U},\mathbf{V}\right) \tag{40}$$

consequently as suggested by the equality above, the ordered pair of hypermatrices $(\mathbf{U},\mathbf{V})$ is said to denote inverse pairs associated with the ordered hypermatrix pair $(\mathbf{A},\mathbf{B})$. in the case where no such hypermatrices pairs exist for the pair $(\mathbf{A},\,\mathbf{B})$ we say that the pair $(\mathbf{A},\mathbf{B})$ is non invertible and in such case we may compute a pseudo-inverse inverse pair as follows.

```
    def HypermatrixPseudoInversePairs(A,B):
        """
         Outputs the pseudo inverse pairs associated with the input pairs of matrices

        EXAMPLES:
        ::
            sage: A1=[[[0.1631135370902057,0.11600112072013125],[0.9823708115400902,0.39605960486710756]]\
    ,[[0.061860929755424676,0.2325542810173995],[0.39111210957450926,0.2019809359102137]]]
            sage: A2=[[[0.15508921433883183,0.17820377184410963],[0.48648171594508205,0.015680176360820 64]]\
    ,[[0.8250247759993575,0.1938307874191597],[0.23867299119274843,0.3935578730402869]]]
```

```
            sage: [B1,B2]=HypermatrixPseudoInversePairs(A1,A2)

        AUTHORS:
        - Edinah K. Gnang and Ori Parzanchevski
        """
        sz = len(A)

        # Initializing the list of linear constraints
        CnstrLst = []

        # Initilizing the variable list
        Vrbls  = [var('ln_al'+str(i)+str(j)+str(k)) \
    for i in range(sz) for j in range(sz) for k in range(sz)]+\
    [var('ln_bt'+str(i)+str(j)+str(k)) for i in range(sz) for j in range(sz) \
    for k in range(sz)]

        for m in range(sz):
            for p in range(sz):
                for n in range(sz):
                    V=Matrix(CC, sz, sz, [(A[m][k1][k0])*(B[k0][k1][p]) \
    for k0 in range(sz) for k1 in range(sz)]).inverse()
                    CnstrLst=CnstrLst+[\
    var('ln_al'+str(m)+str(n)+str(k1))+var('ln_bt'+str(k1)+str(n)+str(p))==\
    ln(V[k1,n])  for k1 in range(sz)]
        [A,b]=ConstraintFormator(CnstrLst,Vrbls)

        # Importing the Numerical Python package
        # for computing the matrix pseudo inverse
        import numpy

        sln = matrix(numpy.linalg.pinv(A))*b
        R1 = HypermatrixGenerateAllZero(sz,sz,sz)
        for i in range(sz):
            for j in range(sz):
                for k in range(sz):
                    R1[i][j][k] = exp(sln[i*sz^2+j*sz^1+k*sz^0,0])
        R2 = HypermatrixGenerateAllZero(sz, sz, sz)
        for i in range(sz):
            for j in range(sz):
                for k in range(sz):
                    R2[i][j][k] = exp(sln[sz^3+i*sz^2+j*sz^1+k*sz^0,0])
        return [R1,R2]
```

To illustrate how the procedure implemented above are used, we compute for the inverse pair corresponding to the hypermatrix pair $(\mathbf{A}_1, \mathbf{A}_2)$ specified bellow

```
    # Building from the example mentioned in the implementation
    # we consider the hypermatrices
    A1=[[[0.1631135370902057,0.11600112072013125],\
    [0.9823708115400902,0.39605960486710756]],\
    [[0.061860929755424676,0.2325542810173995],\
    [0.39111210957450926,0.2019809359102137]]]

    A2=[[[0.15508921433883183,0.17820377184410963],\
        [0.48648171594508205,0.01568017636082064]],\
        [[0.8250247759993575,0.1938307874191597],\
        [0.23867299119274843,0.3935578730402869]]]

        # Numerical computation of the hypermatrix inverse pairs
        [B1,B2]=HypermatrixPseudoInversePairs(A1,A2)

        # To appreciate how good the numerical approximation of the
        # inverse pair is we generate the generic symbolic hypermatrix M
        M0 = HypermatrixGenerate(2,2,2,'m')

        # We would want to compare the symbolic hypermatrix M to the product
        M1 = HypermatrixProduct(M0,A1,A2)
        M2 = HypermatrixProduct(M1,B1,B2)
```

unfortunately the hypermatrix pair $(\mathbf{A}_1,\ \mathbf{A}_2)$ (chosen here randomly above) admits no inverse pair and hence starting from the generic symbolic hypermatrix $\mathbf{M}$

$$\mathbf{M} = [[[m_{000}, m_{001}], [m_{010}, m_{011}]], [[m_{100}, m_{101}], [m_{110}, m_{111}]]] \tag{41}$$

we illustrate the error induced by the pseudo-inversion by comparing to $\mathbf{M}$ the 3-hypermatrix product computation $\circ(\circ(\mathbf{M}, \mathbf{A}_1, \mathbf{A}_2), \mathbf{B}$ with entries given by

$$[\circ(\circ(\mathbf{M}, \mathbf{A}_1, \mathbf{A}_2), \mathbf{B}_1, \mathbf{B}_2)]_{0,0,0} = \left(0.644368723513728 + 1.40512601554121 \times 10^{-15} i\right) m_{000} + \left(7.49400541621981 \times 10^{-16} + 3.33066907387547 \times 10^{-16} i\right) m_{010} \tag{42}$$

$$[\circ(\circ(\mathbf{M}, \mathbf{A}_1, \mathbf{A}_2), \mathbf{B}_1, \mathbf{B}_2)]_{0,0,1} = \left(1.55190648383277 - 1.24900090270330 \times 10^{-15} i\right) m_{001} + \left(3.55271367880050 \times 10^{-15} - 1.11022302462516 \times 10^{-16} i\right) m_{011} \tag{43}$$

$$[\circ(\circ(\mathbf{M}, \mathbf{A}_1, \mathbf{A}_2), \mathbf{B}_1, \mathbf{B}_2)]_{0,1,0} = -\left(1.94289029309402 \times 10^{-16} + 9.71445146547012 \times 10^{-17} i\right) m_{000} + \left(0.644368723513727 + 1.40165656858926 \times 10^{-15} i\right) m_{010} \tag{44}$$

$$[\circ(\circ(\mathbf{M}, \mathbf{A}_1, \mathbf{A}_2), \mathbf{B}_1, \mathbf{B}_2)]_{0,1,1} = \left(5.55111512312578 \times 10^{-17} + 1.24900090270330 \times 10^{-16} i\right) m_{001} + \left(1.55190648383277 + 1.33226762955019 \times 10^{-15} i\right) m_{011} \tag{45}$$

$$[\circ(\circ(\mathbf{M}, \mathbf{A}_1, \mathbf{A}_2), \mathbf{B}_1, \mathbf{B}_2)]_{1,0,0} = \left(1.55190648383277 + 1.78156100982818 \times 10^{-15} i\right) m_{100} + \left(4.38538094726937 \times 10^{-15} + 5.55111512312578 \times 10^{-16} i\right) m_{110} \tag{46}$$

$$[\circ(\circ(\mathbf{M}, \mathbf{A}_1, \mathbf{A}_2), \mathbf{B}_1, \mathbf{B}_2)]_{1,0,1} = \left(0.644368723513724 - 1.38777878078145 \times 10^{-16} i\right) m_{101} + \left(0.644368723513724 - 1.38777878078145 \times 10^{-16} i\right) m_{101} \tag{47}$$

$$[\circ(\circ(\mathbf{M}, \mathbf{A}_1, \mathbf{A}_2), \mathbf{B}_1, \mathbf{B}_2)]_{1,1,0} = \left(1.38777878078145 \times 10^{-17} - 2.15105711021124 \times 10^{-16} i\right) m_{100} + \left(1.55190648383277 - 2.72351585728359 \times 10^{-16} i\right) m_{110} \tag{48}$$

$$[\circ(\circ(\mathbf{M}, \mathbf{A}_1, \mathbf{A}_2), \mathbf{B}_1, \mathbf{B}_2)]_{1,1,1} = -\left(2.77555756156289 \times 10^{-17} - 2.01227923213310 \times 10^{-16} i\right) m_{101} + \left(0.644368723513726 + 1.22124532708767 \times 10^{-15} i\right) m_{111} \tag{49}$$

2.1 Hypermatrix class

As a summary for the hypermatrix package for the convenience of the user we encapsulate all the pieces into a single class all the precedures implemented above. This is particularly useful for the purpose of setting up computer experiments with the proposed hypermatrix package.

```
class HM:
    """HM class"""
    def __init__(self,*args):
# Single argument class constructor specification.
        if len(args) == 1:
            inp = args[0]
            if type(inp)==type(Matrix(SR,2,1,[var('x'),var('y')])) or \
type(inp)==type(Matrix(RR,2,1,[1,2])) or type(inp)==type(Matrix(CC,2,1,[1,1])):
                self.hm=DiagonalHypermatrix(inp)
            elif type(inp) == list:
                self.hm = inp
            else:
                raise ValueError, \
"Expected either a list or and an object of type Matrix"
            return
        # Two or more arguments class constructor
        s = args[-1]
        dims = args[:-1]
        if s == 'one':
            self.hm = apply(HypermatrixGenerateAllOne, dims)
        elif s == 'zero':
            self.hm = apply(HypermatrixGenerateAllZero, dims)
        elif s == 'ortho':
            if len(dims) == 1:
                self.hm=Orthogonal2x2x2Hypermatrix(dims[0])
            elif len(dims) == 2:
                self.hm=Orthogonal3x3x3Hypermatrix(dims[0],dims[1])
            else:
                raise ValueError,\
"ortho not supported for order %d tensors" % len(dims)
        elif s == 'perm':
            self.hm=HypermatrixPermutation(dims[0])
        elif s == 'kronecker':
            self.hm=HypermatrixKroneckerDelta(dims[0])
        elif s == 'sym':
            if len(dims) == 2:
                self.hm=SymHypermatrixGenerate(dims[0],dims[1])
            else :
                raise ValueError,\
"kronecker not supported for order %d tensors" % len(dims)
        else:
            self.hm=apply(HypermatrixGenerate, args)

    def __repr__(self):
        return `self.hm`

    def __add__(self, other):
        return GeneralHypermatrixAdd(self,other)
```

```
def __neg__(self):
    return GeneralHypermatrixScale(self.hm,-1)

def __sub__(self, other):
    return GeneralHypermatrixAdd(self, GeneralHypermatrixScale(other,-1))

def __mul__(self, other):
    if other.__class__.__name__=='HM':
        return HM(GeneralHypermatrixHadamardProduct(self,other))
    elif other.__class__.__name__=='tuple':
        # This function takes a a list as intput
        l = other
        return GeneralHypermatrixProduct(self,*l)
    else:
        return GeneralHypermatrixScale(self,other)

def __rmul__(self, a):
    return self*a

def __getitem__(self,i):
    if i.__class__.__name__=='tuple':
        tmp = self.hm
        for j in i:
            tmp = tmp[j]
        return tmp

def __setitem__(self, i, v):
    if   i.__class__.__name__=='tuple':
        tmp = self.hm
        while len(i)>1:
            tmp = tmp[i[0]]
            i = i[1:]
        tmp[i[0]] = v

def __call__(self, *inpts):
    # This function takes a a list as intput
    return GeneralHypermatrixProduct(self, *inpts)

def hprod(self,*inpts):
    # This function takes a a list as intput
    return GeneralHypermatrixProduct(self,*inpts)

def hprod3b(self, b, c, t):
    return HM(HypermatrixProductB(self.hm, b.hm, c.hm, t.hm))

def elementwise_product(self,B):
    return GeneralHypermatrixHadamardProduct(self,B)

def elementwise_exponent(self,s):
    return GeneralHypermatrixExponent(self,s)

def elementwise_base_exponent(self,s):
    return GeneralHypermatrixBaseExponent(self,s)
```

```
def transpose(self, i=1):
    t = Integer(mod(i, self.order()))
    A = self
    for i in range(t):
        A = GeneralHypermatrixCyclicPermute(A)
    return A

def nrows(self):
    return len(self.hm)

def ncols(self):
    return len(self.hm[0])

def ndpts(self):
    return len(self.hm[0][0])

def n(self,i):
    tmp = self.listHM()
    for j in range(i):
        tmp = tmp[0]
    return len(tmp)

def list(self):
    lst = []
    l = [self.n(i) for i in range(self.order())]
    # Main loop canonicaly listing the elements
    for i in range(prod(l)):
        entry = [mod(i,l[0])]
        sm = Integer(mod(i,l[0]))
        for k in range(len(l)-1):
            entry.append(Integer(mod(Integer((i-sm)/prod(l[0:k+1])),l[k+1])))
            sm = sm+prod(l[0:k+1])*entry[len(entry)-1]
        lst.append(self[tuple(entry)])
    return lst

def listHM(self):
    return self.hm

def cayley_hamilton_list(self,n):
    tmp = HypermatrixCayleyHamiltonList(self.hm,n)
    return [HM(h) for h in tmp]

def cayley_hamilton_mtrx(self,itr,bnd):
    tmp = []
    for i in range(itr):
        tmp = tmp + HypermatrixCayleyHamiltonList(self.hm, 2*i+1)
    return Matrix([HM(h).list() for h in tmp[0:bnd]])

def order(self):
    cnt = 0
    H = self.listHM()
    while type(H) == type([]):
        H = H[0]
```

```
            cnt = cnt+1
        return cnt
```

We implement also some additional auxiliary special functions specifically used by the class for dealing with hypermatrices of order greater then 3. We start by implementing a general hypermatrix product operation which incorporate matrix and 3-hypermatrix products as special cases and therfore captures the full Bhattacharya-Mesner algebra.

```
def GeneralHypermatrixProduct(*args):
    # Initialization of the list specifying the dimensions of the output
    l = [(args[i]).n(i) for i in range(len(args))]
    # Initializing the input for generating a symbolic hypermatrix
    inpts = l+['zero']
    # Initialization of the hypermatrix
    Rh = HM(*inpts)
    # Main loop performing the assignement
    for i in range(\
prod([(args[j]).n(Integer(mod(j+1,len(args)))) for j in range(len(args))])):
        entry = [mod(i,l[0])]
        sm = Integer(mod(i,l[0]))
        for k in range(len(l)-1):
            entry.append(Integer(mod(Integer((i-sm)/prod(l[0:k+1])),l[k+1])))
            sm = sm+prod(l[0:k+1])*entry[len(entry)-1]
        if len(args)<2:
            raise ValueError, "The number of operands must be >= 2"
        elif len(args) >= 2:
            Rh[tuple(entry)]=sum(\
[prod([args[s][tuple(entry[0:Integer(mod(s+1,len(args)))]+\
[t]+entry[Integer(mod(s+2,len(args))):])] for s in range(len(args)-2)]+\
[args[len(args)-2][tuple(entry[0:len(args)-1]+[t])]]+\
[args[len(args)-1][tuple([t]+entry[1:])]]) for t in range((args[0]).n(1))])
    return Rh
```

We also implement the more generally cyclic action on arbitrary order hypermatrices.

```
def GeneralHypermatrixCyclicPermute(A):
    # Initialization of the list specifying the dimensions of the output
    l = [A.n(i) for i in range(A.order())]
    l = l[1:]+[l[0]]
    # Initializing the input for generating a symbolic hypermatrix
    inpts = l+['r']
    # Initialization of the hypermatrix
    Rh = HM(*inpts)
    # Main loop performing the transposition of the entries
    for i in range(prod(l)):
        entry = [mod(i,l[0])]
        sm = Integer(mod(i,l[0]))
        for k in range(len(l)-1):
            entry.append(Integer(mod(Integer((i-sm)/prod(l[0:k+1])),l[k+1])))
            sm = sm+prod(l[0:k+1])*entry[len(entry)-1]
        # Performing the transpose
        Rh[tuple(entry)]=A[tuple([entry[len(entry)-1]]+entry[:len(entry)-1])]
    return Rh
```

similarly the scaling function will be given by

```
def GeneralHypermatrixScale(A,s):
    # Initialization of the list specifying the dimensions of the output
    l = [A.n(i) for i in range(A.order())]
    # Initializing the input for generating a symbolic hypermatrix
    inpts = l+['r']
    # Initialization of the hypermatrix
    Rh = HM(*inpts)
    # Main loop performing the transposition of the entries
    for i in range(prod(l)):
        entry = [mod(i,l[0])]
        sm = Integer(mod(i,l[0]))
        for k in range(len(l)-1):
            entry.append(Integer(mod(Integer((i-sm)/prod(l[0:k+1])),l[k+1])))
            sm = sm+prod(l[0:k+1])*entry[len(entry)-1]
        # Performing the computation
        Rh[tuple(entry)]=s*A[tuple(entry)]
    return Rh
```

the function which compute the exponentiation of the entries for arbitrary order hypermatrices is given bellow

```
def GeneralHypermatrixExponent(A,s):
    # Initialization of the list specifying the dimensions of the output
    l = [A.n(i) for i in range(A.order())]
    # Initializing the input for generating a symbolic hypermatrix
    inpts = l+['r']
    # Initialization of the hypermatrix
    Rh = HM(*inpts)
    # Main loop performing the transposition of the entries
    for i in range(prod(l)):
        entry = [mod(i,l[0])]
        sm = Integer(mod(i,l[0]))
        for k in range(len(l)-1):
            entry.append(Integer(mod(Integer((i-sm)/prod(l[0:k+1])),l[k+1])))
            sm = sm+prod(l[0:k+1])*entry[len(entry)-1]
        # Performing computation
        Rh[tuple(entry)]=(A[tuple(entry)])^s
    return Rh
```

for accounting for the non commutativeity of the exponentiation operation we implement the other function

```
def GeneralHypermatrixBaseExponent(A,s):
# Initialization of the list specifying the dimensions of the output
    l = [A.n(i) for i in range(A.order())]
    # Initializing the input for generating a symbolic hypermatrix
    inpts = l+['r']
    # Initialization of the hypermatrix
    Rh = HM(*inpts)
    # Main loop performing the transposition of the entries
    for i in range(prod(l)):
        entry = [mod(i,l[0])]
        sm = Integer(mod(i,l[0]))
        for k in range(len(l)-1):
            entry.append(Integer(mod(Integer((i-sm)/prod(l[0:k+1])),l[k+1])))
            sm = sm+prod(l[0:k+1])*entry[len(entry)-1]
            # Performing computation
            Rh[tuple(entry)]=s^(A[tuple(entry)])
        return Rh
```

Similarly arbitrary order hypermatrix addition is implemented bellow as

```
    def GeneralHypermatrixAdd(A,B):
        # Initialization of the list specifying the dimensions of the output
        l = [A.n(i) for i in range(A.order())]
        s = [B.n(i) for i in range(B.order())]
        # Testing the dimensions
        x = var('x')
        if(sum([l[i]*x^i for i in range(len(l))])==sum(\
    [s[i]*x^i for i in range(len(s))])):
            # Initializing the input for generating a symbolic hypermatrix
            inpts = l+['r']
            # Initialization of the hypermatrix
            Rh = HM(*inpts)
            # Main loop performing the transposition of the entries
            for i in range(prod(l)):
                entry = [mod(i,l[0])]
                sm = Integer(mod(i,l[0]))
                for k in range(len(l)-1):
                    entry.append(Integer(mod(Integer((i-sm)/prod(l[0:k+1])),l[k+1])))
                    sm = sm+prod(l[0:k+1])*entry[len(entry)-1]
                Rh[tuple(entry)]=A[tuple(entry)]+B[tuple(entry)]
            return Rh
        else:
            raise ValueError,\
    "The Dimensions of the input hypermatrices must match."
```

quite similarly the Hadamard product is given by

```
    def GeneralHypermatrixHadamardProduct(A,B):
    # Initialization of the list specifying the dimensions of the output
        l = [A.n(i) for i in range(A.order())]
        s = [B.n(i) for i in range(B.order())]
        # Testing the dimensions
        x = var('x')
        if(sum([l[i]*x^i for i in range(len(l))])==sum(\
    [s[i]*x^i for i in range(len(s))])):
            # Initializing the input for generating a symbolic hypermatrix
            inpts = l+['r']
            # Initialization of the hypermatrix
            Rh = HM(*inpts)
            # Main loop performing the transposition of the entries
            for i in range(prod(l)):
                entry = [mod(i,l[0])]
                sm = Integer(mod(i,l[0]))
                for k in range(len(l)-1):
                    entry.append(Integer(mod(Integer((i-sm)/prod(l[0:k+1])),l[k+1])))
                    sm = sm+prod(l[0:k+1])*entry[len(entry)-1]
                Rh[tuple(entry)]=A[tuple(entry)]+B[tuple(entry)]
        return Rh
    else:
        raise ValueError,\
"The Dimensions of the input hypermatrices must match."
```

As a way of illustrating the existence of arbitrary order hypermatrices we implement a procedure for parametrizing (albeit somewhat redundantly in the number of variables) hypermatrices of the size $2 \times 2 \times 2 \times \cdots \times 2$. and therefore providing a constructif proof of existence of arbitrary order orthogonal hypermatrices.

```
def GeneralOrthogonalHypermatrix(od):
    # Initializing the hypermatrix
    Q = apply(HM,[2 for i in range(od)]+['q'])

    # Initilizing the list of variable
    VrbLst = Q.list()

    # Reinitializing of Q by exponentiation
    Q = Q.elementwise_base_exponent(e)

    # Computing the product
    Eq = apply(GeneralHypermatrixProduct,[Q.transpose(j) for j in range(od,0,-1)])

    # Writting up the constraints
    LeQ = (Set(Eq.list())).list()

    # Removing the normalization constraints
    LeQ.remove(e^(od*var('q'+''.join(['0' for i in range(od)])))+\
e^(od*var('q01'+''.join(['0' for i in range(od-2)]))))
    LeQ.remove( e^(od*var('q10'+''.join(['1' for i in range(od-2)])))+\
e^(od*var('q'+''.join(['1' for i in range(od)]))))

    # Filling up the linear constraints
    CnstrLst= []
    for f in LeQ:
        CnstrLst.append(\
ln((f.operands())[0]).simplify_exp()-I*pi-ln((f.operands())[1]).simplify_exp()==0)

    # Directly solving the constraints
    Sl = solve(CnstrLst,VrbLst)

    # Main loop performing the substitution of the entries
    Lr = [var('r'+str(i)) for i in range(1,2^od+1)]
    l = [Q.n(i) for i in range(Q.order())]
    for i in range(prod(l)):
        # Turning the index i into an hypermatrix array location
        # using the decimal encoding trick
        entry = [mod(i,l[0])]
        sm = Integer(mod(i,l[0]))
        for k in range(len(l)-1):
            entry.append(Integer(mod(Integer((i-sm)/prod(l[0:k+1])),l[k+1])))
            sm = sm+prod(l[0:k+1])*entry[len(entry)-1]
        Q[tuple(entry)]=Q[tuple(entry)].subs(\
dict(map(lambda eq: (eq.lhs(),eq.rhs()), Sl[0]))).simplify_exp()
    return Q
```

Acknowledgments

We would like to thank the IAS for providing excellent working conditions. This material is based upon work supported by the National Science Foundation under agreements Princeton University Prime Award No. CCF-0832797 and Sub-contract No. 00001583. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[B] P. Bhattacharya, “A new 3-D transform using a ternary product,” IEEE Trans. on Signal Processing 43(12):3081-3084, 1995.

[BM1] D. M. Mesner and P. Bhattacharya, “Association schemes on triples and a ternary algebra,” J. Comb. Theory, ser. A, vol. 55, pp. 204-234, 1990.

[BM2] D. M. Mesner and P. Bhattacharya, “A ternary algebra arising from an association scheme on triples,” J. Algebra. vol. 164, no. 1 pp. 595-613, 1994.

[GER] E. K. Gnang, A. Elgammal and V. Retakh, “A Spectral Theory for Tensors” Annales de la faculté des sciences de Toulouse Sér. 6, 20 no. 4, p. 801-841, 2011.

[K] D. E. Knuth, “Literate Programming” (Stanford, California: Center for the Study of Language and Information, 1992)

[L] L.-H. Lim, "Tensors and hypermatrices," in: L. Hogben (Ed.), Handbook of Linear Algebra, 2nd Ed., CRC Press, Boca Raton, FL, 2013.

[LQ] L. Qi, “The Spectral Theory of Tensors”, preprint arXiv:1201.3424 [math.SP], 2012.

[S6] W. A. Stein et al., *Sage Mathematics Software (Version 6.0)*, The Sage Development Team, (2013) , `http://www.sagemath.org`.

[Z] D. Zeilberger, "[Contemporary Pure] Math Is Far Less Than the Sum of Its [Too Numerous] Parts", Opinion. Vol. 60 Iss. 10, Notices of the AMS. December 2013