Recursive vectors

Gonzalo Reynaga García
February 2019

This article describes a new operation with different kind of vectors called recursive vectors that look interesting by the simplicity and it is easy to be expanded to N dimensions and works like a scalar complex number, that It could be useful for machine learning. It is implemented as built-in operation in Math-c version 1.2.0.

Introduction

The basic operation is just expanding the operation of 2 complex numbers to N dimensions of power of 2. It splits a N dimension vector in "real" part and "imaginary" part, and those parts then split it in "real" and "imaginary" parts and so on, until you reach a scalar part.
For instance, It can be represented as:
[1 3 4 5 6 7 4 6]   to   [[1+3i 4+5i] [6+7i 4+6i]]

Implementation of this operations Math-c/Octave/Matlab are in Appendix.
Note that all the operations with recursive vectors("revectors" for short) are commutative and associative.

Sum and subtraction

There no special operation, each element is added or subtracted as any vector.

Multiplication

The same way the multiplication of complex scalars, also is done in a recursive way.

$$(a + bi) \cdot (c + di) = (a \cdot c - b \cdot d) + (b \cdot c + a \cdot d)i$$

Re() = "Real" part
Im() = "Imaginary" part

The multiplication of V and U:

$$V = [Re(V) \quad Im(V)]$$
$$U = [Re(U) \quad Im(U)]$$

$$V\text{^}U = [(Re(V) \cdot Re(U) - Im(V) \cdot Im(U)) \quad (Im(V) \cdot Re(U) + Re(V) \cdot Im(U))]$$

The notation for the multiplication of recursive vectors will be the wedge symbol "^".

Division

It is very similar as the division of complex scalars:

$$\frac{1}{(a+bi)} = \frac{a-b}{a^2+b^2}$$

The inverse of recursive vector V:

$$V = [Re(V) \quad Im(V)]$$

$$\frac{1}{V} = [Re(V) \quad -Im(V)] \cdot \left(\frac{1}{[Re(V) \cdot Re(V) \quad Im(V) \cdot Im(V)]}\right)$$

The vector V divided by V is:

$$\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \frac{V}{V}$$

Conjugate

The conjugate of the vector V is the recursive conjugate of:

$$\bar{V} = [\overline{Re(V)} \quad -\overline{Im(V)}]$$

Matrix

As the multiplication and division operations are defined and has the associative and commutative properties, we can solve matrices of recursive vectors in the same way it's solved for matrices in scalars.

$$R = \begin{bmatrix} U_{0,0} & \cdots & U_{0,N} \\ \vdots & \ddots & \vdots \\ U_{N,0} & \cdots & U_{N,N} \end{bmatrix} ^\wedge \begin{bmatrix} V_0 \\ \vdots \\ V_N \end{bmatrix}$$

Notation

The notation in this document to define recursive vector of scalars

$$[1] = [1\ 0\ 0 \ldots 0\ 0]$$
$$[1_1] = [0\ 1\ 0 \ldots 0\ 0]$$
$$[1_2] = [0\ 0\ 1 \ldots 0\ 0]$$

And:

$$[\overrightarrow{1}] = [1\ 1\ 1 \ldots 1\ 1]$$

Dot product

Given a vector and the dot product result, there is infinity vectors that could have the same result. For instance, for the vector V and dot product scalar α shown in figure 1, any vector perpendicular will have the same dot product result.

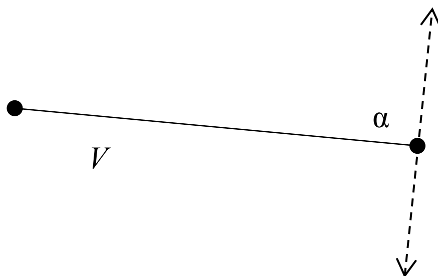

Figure 1. vector with dot product α

The dot product of U and V is equal to scalar part of recursive vector product of V and U:

$$scalar(\overline{V}\,\hat{}\,U) = scalar(V\,\hat{}\,\overline{U}) = dot(V, U)$$

The infinite vectors U given V and α can be obtained from any value x we set in the result of the multiplication:

$$\begin{bmatrix} \alpha \\ x_1 \\ \vdots \\ x_N \end{bmatrix} = V\,\hat{}\,\overline{U}$$

Suppose the following problem, we have V *and* U and vectors, we want a function *f(V)*=[1], and a function *f(U)*=[0.8]. We can create linear equations and creating the following matrix system and setting 0 for the other scalar values of the result.
Note that the linear equation scalar part is equal to dot product plus a scalar.

$$[1] = V\,\hat{}\,X_0 + X_1$$
$$[0.8] = U\,\hat{}\,X_0 + X_1$$

In matrix form:

$$\begin{bmatrix} [1] \\ [0.8] \end{bmatrix} = \begin{bmatrix} V & [1] \\ U & [1] \end{bmatrix} \hat{} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix}$$

So the result is:

$$\begin{bmatrix} [1] \\ [0.8] \end{bmatrix} \hat{} \begin{bmatrix} V & [1] \\ U & [1] \end{bmatrix}^{-1} = \begin{bmatrix} X_0 \\ X_1 \end{bmatrix}$$

The final function will be $F(R) = X_0\,\hat{}\,R + X_1$

It can be seen is has a the from *"ax + b"* and can be extended to any polynomial degree.

Split method

But also, the function $f(V)=[1]$ and $f(U)=[0.8]$ could be solved in the following way.
We can split V in 2 vectors, in real part and imaginary part to create the matrix system reducing the dimension of the vectors(smaller vectors).

$$[1] = V_r\hat{}X_0 + V_i\hat{}X_1$$
$$[0.8] = U_r\hat{}X_0 + U_i\hat{}X_1$$

In matrix form:

$$\begin{bmatrix}[1]\\ [0.8]\end{bmatrix} = \begin{bmatrix}V_r & V_i\\ U_r & U_i\end{bmatrix} \hat{} \begin{bmatrix}X_0\\ X_1\end{bmatrix}$$

If we require multiples desired values, we can continue splitting the vector until reach the maximum dimension of the vector.
This is shown as an example in other article where can used for classification problem.

Scalar Matrix to recursive vector matrix

The scalar matrices can be transformed to recursive vector matrices.
In a 2x2 matrix, the recursive vector is just a complex number $a+ib$.
Given a scalar 2x2 matrix:

$$\begin{bmatrix} r_0 \\ r_1 \end{bmatrix} = \begin{bmatrix} m_{0,0} & m_{0,1} \\ m_{1,0} & m_{1,1} \end{bmatrix} \cdot \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \qquad \text{Equation.1}$$

We can create complex numbers in a horizontal or vertical way and give the same result.
Let's define the complex numbers:

$$R = r_0 + r_1 i$$
$$W = w_0 + w_1 i$$

In vertical:

$$M_{v0} = m_{0,0} + m_{1,0} i$$
$$M_{v1} = m_{0,1} + m_{1,1} i$$

In horizontal:

$$M_{h0} = m_{0,0} + m_{0,1} i$$
$$M_{h1} = m_{1,0} + m_{1,1} i$$

We can transform the Equation 1 to:

Using vertical complex

$$R = \left(\frac{M_{v0} - M_{v1} i}{2}\right) \wedge W + \left(\frac{M_{v0} + M_{v1} i}{2}\right) \wedge \bar{W}$$

To obtain conjugate R

$$\bar{R} = \left(\frac{\bar{M}_{v0} - \bar{M}_{v1} i}{2}\right) \wedge W + \left(\frac{\bar{M}_{v0} + \bar{M}_{v1} i}{2}\right) \wedge \bar{W}$$

We can merge these two equations to obtain the matrix:

$$\begin{bmatrix} R \\ \bar{R} \end{bmatrix} = \begin{bmatrix} (M_{v0} - M_{v1} i)/2 & (M_{v0} + M_{v1} i)/2 \\ (\bar{M}_{v0} - \bar{M}_{v1} i)/2 & (\bar{M}_{v0} + \bar{M}_{v1} i)/2 \end{bmatrix} \wedge \begin{bmatrix} W \\ \bar{W} \end{bmatrix}$$

With horizontal complex numbers, we can obtain:

$$\begin{bmatrix} R \\ \bar{R} \end{bmatrix} = \begin{bmatrix} (\bar{M}_{h0} + \bar{M}_{h1} i)/2 & (M_{h0} + M_{h1} i)/2 \\ (\bar{M}_{h0} - \bar{M}_{h1} i)/2 & (M_{h0} - M_{h1} i)/2 \end{bmatrix} \wedge \begin{bmatrix} W \\ \bar{W} \end{bmatrix}$$

In a similar way also can be done with 4x4 matrices, the appendix B shows how the rows and columns can be merged to create the recursive vector in a 4x4 matrix.

Linear regression

The approximation of points to a line could be represented by the multiplication of a complex number. And with the recursive vector, it could expand the same equation to N dimensions but with the loss of precision comparing with linear regression of classic matrix/vectors.
With input values of one dimension(plus "bias"):

$$X_c = [x_0 + i; \quad x_1 + i; \quad \dots \quad x_k + i]$$
$$Y_c = [y_0 \quad y_1 \quad \cdots \quad y_k]$$

Approximate to function:

$$Y = X \hat{} A$$

$$X^T \hat{} Y = X^T \hat{} X \hat{} A$$
$$\frac{X^T \hat{} Y}{X^T \hat{} X} = A$$

Note that $X^T \hat{} X$ is a vector, not a matrix, and the division of recursive vectors are commutative.
If A is a vector of scalar or a vector of recursive vectors, then:

$$(X^T \cdot X)^{-1} \cdot X^T \cdot Y = A$$

Conclusion

Analyzing the recursive vector multiplication, in the scalar give us a value of how similar a vector is with other, just like the dot product, but the other values give us information of how different it is and use the vectors as any scalar without taking account if has N dimensions.

I think this operation could be very useful for areas of classification and machine learning due the simplicity of the operation and I think it is possible to solve machine learning problems in a more deterministic way.

Contact:

Any questions or suggestions, you can contact me at gonzaloreynaga@yahoo.com

Appendix A

Algebra code implementation

Vectorize/unbracket

These functions transform the regular array into a recursive vector format to be able to multiply/divide.

Octave/Matlab:

```
>> a = [1 5 3 10 9 6 5 8];
>> a=vectorize(a)
a =
{
  [1,1] =
  {
    [1,1] =  1 + 5i
    [1,2] =   3 + 10i
  }

  [1,2] =
  {
    [1,1] =  9 + 6i
    [1,2] =  5 + 8i
  }

}
>> a=unbracket(a)
a =
    1    5    3   10    9    6    5    8
>>
```

vectorize function:

```
function retval = vectorize (a)
  vSize = max(size(a));

  Ra = a(1:vSize/2);
  Ia = a(vSize/2+1:vSize);

  if vSize == 2
    retval = a(1) + 1i*a(2);
    return;
  endif

  Ra = vectorize(Ra);
  Ia = vectorize(Ia);

  retval = { Ra Ia };
  return;
endfunction
```

unbracket function:

```
function retval = unbracket (a)
  vSize = max(size(a));
  if vSize == 1
    retval = [real(a) imag(a)];
    return;
  end
  vectorarray = [];
  for i=1:2
    vectorarray = [vectorarray unbracket(a{i})];
  endfor
  retval = vectorarray;
  return;
endfunction
```

Math-c:

```
>>> a = [1 5 3 10 9 6 5 8];
>>> a=vectorize(a)
a = [[1+5i  3+10i]  [9+6i  5+8i]]
>>> unbracket(a)
ans = [1  5  3  10  9  6  5  8]
```

vectorize function:

```
function vectorize(a)
    vSize = max(size(a))(0);
    Ra = a(0:(vSize/2-1));
    Ia = a((vSize/2):vSize-1);

    if vSize == 2
        if isArray(a(0)) == 0
            return a(0) + 1i*a(1);
        end
        Ra = Ra(0);
        Ia = Ia(0);
    end
    Ra = vectorize(Ra);
    Ia = vectorize(Ia);
    return [Ra Ia];
end
```

unbracket function:

```
function unbracket(a)
    vSize = max(size(a))(0);
    if max(size(a(0)))(0) == 1
        return [real(a(0)) imag(a(0)) real(a(1)) imag(a(1))];
    end
    vectorArray = [];
    for i=0:vSize-1
        vectorArray = vectorArray <-> unbracket(a(i));
    end
    return vectorArray;
end
```

Addition and subtraction.

Octave/Matlab:

Only if it's added/subtracted after the "vectorize" function.

Add

```
>> a = [1 5 3 10 9 6 5 8];
>> b = [1 6 2  6 8 6 9 3];
>> a = vectorize(a);
>> b = vectorize(b);
>> addV(a,b)
```

Substraction:

```
>> a = [1 5 3 10 9 6 5 8];
>> b = [1 6 2  6 8 6 9 3];
>> a = vectorize(a);
>> b = vectorize(b);
>> subV(a,b)
```

addV function:

```
function retval = addV(v,u)
  if iscell(v) == 1
    retval = cellfun(@plus, v , u ,'UniformOutput', false);
    return;
  end
  retval = v + u;
endfunction
```

subV function:

```
function retval = subV(v,u)
  if iscell(v) == 1
    retval = cellfun(@minus, v , u ,'UniformOutput', false);
    return;
  end
  retval = v - u;
endfunction
```

Math-c:

Add

```
>>> a = [1 5 3 10 9 6 5 8];
>>> b = [1 6 2  6 8 6 9 3];
>>> a = vectorize(a);
>>> b = vectorize(b);
>>> a + b
```

Substraction:

```
>>> a = [1 5 3 10 9 6 5 8];
>>> b = [1 6 2  6 8 6 9 3];
>>> a = vectorize(a);
>>> b = vectorize(b);
>>> a - b
```

Multiplication

The multiplication of "a" and "b" vectors:

Octave/Matlab:

```
>> a = [1 5 3 10 9 6 5 8];
>> b = [1 6 2  6 8 6 9 3];

>> a = vectorize(a);
>> b = vectorize(b);

>> vxV(a,b)
ans =
{
  [1,1] =
  {
    [1,1] =  10 - 42i
    [1,2] = -140 - 131i
  }

  [1,2] =
  {
    [1,1] =  -8 - 39i
    [1,2] = -103 + 250i
  }

}
```

vxV function:

```
function retval = vxV (a, b)
  vSize = max(size(a));
  if vSize == 1
    retval = a * b;
    return;
  end
  Ra = a{1};
  Ia = a{2};

  Rb = b{1};
  Ib = b{2};

  if iscell(Ra) == 1
    Rq = cellfun(@minus, vxV( Ra,Rb ) , vxV( Ia,Ib ) ,'UniformOutput', false);
    Iq = cellfun(@plus, vxV( Ia,Rb ) , vxV( Ra,Ib ) ,'UniformOutput', false);
  else
    Rq = vxV(Ra,Rb) - vxV(Ia,Ib);
    Iq = vxV(Ia,Rb) + vxV(Ra,Ib);
  endif

  retval = {Rq Iq};
endfunction
```

Math-c:

The multiplication of recursive vectors is already a built-in operation with the operator "^"

```
>>> a = [1 5 3 10 9 6 5 8];
>>> b = [1 6 2  6 8 6 9 3];

>>> a = vectorize(a);
>>> b = vectorize(b);
>>> a^b
ans = [[10-42i  -140-131i]  [-8-39i  -103+250i]]
```

vxV function:

```
function vxV(a,b)
    vSize = max(size(a))(0);
    if vSize == 1
        if isArray(a(0)) == 1
            error("vxV(a,b)-> array has size 1x1\n");
        end
        //return scalar product
        return a*b;
    end

    Ra = a(0);
    Ia = a(1);

    Rb = b(0);
    Ib = b(1);

    Rq = vxV(Ra,Rb) - vxV(Ia,Ib);
    Iq = vxV(Ia,Rb) + vxV(Ra,Ib);
    //return product
    return [Rq Iq];
end
```

Division(inverse)

Octave/Matlab:

```
>> a = [6 7 5 8 2 10 2 7];

>> a = vectorize(a);
>> invV(a)
ans =
{
  [1,1] =
  {
    [1,1] =  0.012014 - 0.032301i
    [1,2] = -0.0038357 + 0.0277321i
  }

  [1,2] =
  {
    [1,1] = -0.017017 + 0.021477i
    [1,2] =  0.019948 - 0.024582i
  }

}

>> vxV(invV(a),a)
ans =
{
  [1,1] =
  {
    [1,1] =  1.0000e+00 + 2.7756e-17i
    [1,2] =  0.0000e+00 - 1.3878e-17i
  }

  [1,2] =
  {
    [1,1] =  0.0000e+00 + 2.4286e-17i
    [1,2] = -5.5511e-17 - 3.4694e-17i
  }

}
```

invV function:

```
function retval = invV(v)

  len = max(size(v));
  if len == 1
    retval = 1.0/v;
    return;
  endif

  if iscell(v{1}) == 1
    detv = invV(cellfun(@plus, vxV( v{1}, v{1} ) , vxV( v{2}, v{2} ) ,'UniformOutput', false));

    Rr = vxV(detv, v{1});
    Ri = vxV(detv, cellfun(@uminus, v{2} ,'UniformOutput', false));
  else
    detv = invV(v{1}*v{1} + v{2}*v{2});

    Rr = vxV(detv, v{1});
    Ri = vxV(detv, -v{2});
  end

  retval = {Rr Ri};
endfunction
```

Math-c:

The inverse of recursive vectors is a built-in function “inv()”

```
>>> a = vectorize(a);
>>> inv(a)
ans = [[0.0120141-0.0323009i  -0.00383569+0.0277321i]  [-0.0170166+0.0214773i
  0.0199476-0.0245818i]]
>>> inv(a)^a
ans = [[1+2.77556e-17i  -1.38778e-17i]  [2.42861e-17i  -5.55112e-17-3.46945e-17i]]
```

invV function:

```
function invV(v)
    len = max(size(v))(0);
    if(len == 1 )
        return 1/v;
    end
    detv  = invV( v(0)^v(0) + v(1)^v(1) );
    Rr = detv ^ v(0);
    Ri = detv ^ (-v(1));
    return [Rr Ri];
end
```

Appendix B

In this appendix will shows some scripts in Math-c where the recursive vectors are calculated from scalar matrices.

Note: In the program menu set the ”*Math library folder*” from “*Math-c->prefererences…”* to be able to find the “*import*” files.

Matrix 2x2

File **new2x2.mc**

```
#import newmatrix/newAlgebra.mc

clc()
clear()

print("\nClassic algebra\n");

Mc = randi(10,2,2);
vc = randi(10,2,1);
yc = Mc*vc

print("Classic determinant\n");
det(Mc)


print("\nCreating vectors\n");
M = quadMerge( Mc(0,0) + 1i * Mc(0,1) , Mc(1,0) + 1i * Mc(1,1) );
v= vectorMerge(vc(0),vc(1));

print("Dot vector multiplication\n");
dotV(M,v)

print("Matrix x new Vector Multiplication\n");
M2b = [M(0) M(1);cjV(M(1)) cjV(M(0))];
MxV(M2b,v)

print("Determinant with vectors\n");
MxDet(M2b)
```

Output:

```
Classic algebra
yc = [26 ;
      30]
Classic determinant
ans = 6

Creating vectors
Dot vector multiplication
ans = 26+30i
Matrix x new Vector Multiplication
ans = [26+30i  26-30i]
Determinant with vectors
ans = 6
```

Matrix 4x4

File **new4x4.mc**

```
#import newmatrix/newAlgebra.mc


clc();
clear();

print("\nClassic algebra\n");

//Mc = randi(10,4,4);
//vc = randi(10,4,1);
Mc = [ 4  0  8  9 ; ...
       8  3  3  10 ; ...
       2  2  3  2 ; ...
       6  0  6  10 ]
vc = [ 4 ; 9 ; 1 ; 2 ]
yc = Mc * vc


print("\nCreating vectors\n");
M00 = quadMerge( Mc(0,0) + 1i * Mc(0,1) , Mc(1,0) + 1i * Mc(1,1) );
M10 = quadMerge( Mc(2,0) + 1i * Mc(2,1) , Mc(3,0) + 1i * Mc(3,1) );

M01 = quadMerge( Mc(0,2) + 1i * Mc(0,3) , Mc(1,2) + 1i * Mc(1,3) );
M11 = quadMerge( Mc(2,2) + 1i * Mc(2,3) , Mc(3,2) + 1i * Mc(3,3) );

v1= vectorMerge( vc(0), vc(1) );
v2= vectorMerge( vc(2), vc(3) );

print("\nMerge\n");

Mr0 = quadMerge( [ M00(0) M01(0) ], [ M10(0) M11(0) ] )
vr0 = vectorMerge( [ v1(0) ], [ v2(0) ] )
merge1 = Mr0(0)^ vr0(0) + Mr0(1)^ vr0(1)

Mr1 = quadMerge( [ M00(1) M01(1) ], [ M10(1) M11(1) ] )
vr1 = vectorMerge( [ v1(1) ], [ v2(1) ] )
merge2 = Mr1(0)^vr1(0) + Mr1(1)^vr1(1)
print("\nThe result is equals to yc\n");
result = merge1 + merge2
```

Output:

```
Classic algebra
Mc = [4  0  8  9 ;
      8  3  3  10 ;
      2  2  3  2 ;
      6  0  6  10]
vc = [4 ;
      9 ;
      1 ;
      2]
yc = [42 ;
      82 ;
      33 ;
      50]

Creating vectors

Merge
Mr0 = [[5+3i  -4+2.5i]  [-1.5+1i  5-0.5i]]
vr0 = [[4+9i  1+2i]  [4+9i  -1-2i]]
merge1 = [-7+62.5i  -11.5+32i]
Mr1 = [[-1.5+4i  1-1i]  [2  5i]]
vr1 = [[4-9i  1-2i]  [4-9i  -1+2i]]
merge2 = [49+19.5i  44.5+18i]

The result is equals to yc
result = [42+82i  33+50i]
```

File **newAlgebra.mc**

```
////////////////////////////////////////////////////
//argument must be a vector that contains vector(s) to vectorize
function checkPower2(arg)
            if size(arg)(1) == 2
                        [a,b] = arg;
                        //Check size power 2/////////////////////////
                        vSize = max(size(a))(0);
                        if (1 != min(size(a))(0) )||(1 != min(size(b))(0) )
                                    error("vm(a,b)-> vector must size 1xD\n");
                                    return 0;
                        end
                        if (vSize != max(size(b))(0)) || (2^ceil(log2(vSize)) != 2^log2(vSize))
                                    if vSize < max(size(b))(0)
                                                len = 2^ceil( log2(max(size(b)))(0) );
                                    else
                                                len = 2^ceil( log2(max(size(a)))(0) );
                                    end
                                    b = resize(b,[1 len]);
                                    a = resize(a,[1 len]);
                                    a = vectorize(a);
                                    b = vectorize(b);
                        end
                        return [a b];
                        //////////////////////////////////////////
            else
                        a = arg;
                        vSize = max(size(a))(0);
                        if 1 != min(size(a))(0)
                                    error("vm(a,b)-> vector must size 1xD\n");
                                    return 0;
                        end
                        if 2^ceil(log2(vSize)) != 2^log2(vSize)
                                    len = 2^ceil( log2( vSize ));
                                    a = resize(a,[1 len]);
                                    a = vectorize(a);
                        end
                        return a;
            end
            return 0;
end


////////////////////////////////////////////////////
//vectorize an array of numbers WITHOUT
//imaginary part(scalars were not merged)
//example: vectorize([3 4]) = 3+4i
function vectorize(a)
            vSize = max(size(a))(0);

            Ra = a(0:(vSize/2-1));
            Ia = a((vSize/2):vSize-1);

            if vSize == 2
                        if isArray(a(0)) == 0
                                    return a(0) + 1i*a(1);
                        end
                        Ra = Ra(0);
                        Ia = Ia(0);
            end

            Ra = vectorize(Ra);
            Ia = vectorize(Ia);
            return [Ra Ia];
end



////////////////////////////////////////////////////
//remove the brackets to convert it to an array of numbers.
function unbracket(a)
            vSize = max(size(a))(0);
            if max(size(a(0)))(0) == 1
                        return [real(a(0)) imag(a(0)) real(a(1)) imag(a(1))];
            end
            vectorArray = [];
            for i=0:vSize-1
                        vectorArray = vectorArray <-> unbracket(a(i));
            end
            return vectorArray;
end



////////////////////////////////////////
// multiply vector by imaginary number
// a is vector
function vxI(a)
            vSize = max(size(a))(0);
            if vSize == 1
                        if (isArray(a(0)) == 1)
                                    error("mvI(a)-> has an array size 1x1\n");
                                    return 0;
                        end
                        //return scalar product
                        return 1i*a;
   end
            //return product times "i"
            return [-a(1) a(0)];
end


////////////////////////////////////////
// Recursive conjugate of vector
function cjV(v)
            vSize = max(size(v))(0);
            if(min(size(vSize))(0) != 1)
                        print("cjv -> error\n");
                        return 0;
            end
            if(vSize == 1 )
                        return v';
            end
            Rr = cjV(v(0));
            Ri = -cjV(v(1));
            return [Rr Ri];
end
```

```
/////////////////////////////////
// Multiply vector "a" and "b"
// return Gonz vector
function vxV(a,b)
            ////////////////////////
            //[a b] = checkPower2([a,b]);
            ////////////////////////
            vSize = max(size(a))(0);
            if vSize == 1
                        if isArray(a(0)) == 1
                                    error("vxV(a,b)-> array has size 1x1\n");
                        end
                        //return scalar product
                        return a*b;
   end

            Ra = a(0);
            Ia = a(1);

            Rb = b(0);
            Ib = b(1);

            Rq = vxV(Ra,Rb) - vxV(Ia,Ib);
            Iq = vxV(Ia,Rb) + vxV(Ra,Ib);
            //return product
            return [Rq Iq];
end


/////////////////////////////////////
// Determinant Matrix "m"
// "m" is size 2x2 that contains vectors,
// all vector must have the same dimension
function MxDet(m)
            mSize = size(m);
            if mSize != [2 2]
                        error("MxV(m,a)-> Matrix must be 2x2 size\n");
                        return 0;
   end

            R1 = vxV(m(0,0),m(1,1)) - vxV(m(0,1),m(1,0));
            //return determinant
            return R1;
end


/////////////////////////////////////
// Multiply Matrix "m" by vector "a"
// "m" is size 2x2 that contains vectors,
// all vector must have the same dimension
function MxV(m,a)
            mSize = size(m);
            if mSize != [2 2]
                        error("MxV(m,a)-> Matrix must be 2x2 size\n");
                        return 0;
   end

            R1 = vxV(m(0,0),a(0)) + vxV(m(0,1),a(1));
            R2 = vxV(m(1,0),a(0)) + vxV(m(1,1),a(1));
            //return product
            return [R1 R2];
end


//////////////////////////////////
//Multiply dot product
function dotV(a,b)
            len = max(size(a))(0);
            if len == 1
                        if 1 != max(size(b))(0)
                                    error("dotV(a,b)-> array {b} size\n");
                        end
                        return real(a)*real(b) + imag(a)*imag(b);
            end
            if(len!= 2) || (2 != max(size(b))(0))
                        a = vectorize(a);
                        b = vectorize(b);
            end
            Rr = vxV(a(0), b(0)) + vxV(a(1),b(1));
            return Rr;
end


//////////////////////////////////
//Vector inverse
function invV(v)
            len = max(size(v))(0);
            if(len == 1 )
                        return 1/v;
            end
            detv  = invV( vxV(v(0),v(0))+ vxV(v(1),v(1)) );
            Rr = vxV(detv,  v(0));
            Ri = vxV(detv, -v(1));
            return [Rr Ri];
end



////////////////////////////////////////////////
//Vectorize an 2x1 vector of elements and merge it
//if top and bottom are real numbers, no array notation is set,
//otherwise must be enclosed in brackets.
function vectorMerge(top, bottom)
            len = max(size(top(0)))(0);
            if len != max(size(bottom(0)))(0)
                        error("vectorMerge-> elements must have same dimension\n");
                        return 0;
            end
            if          isArray(top) == 0
                        //the elements doesnt have imaginary numbers when scalars
                        m0v = top  + 1i * bottom;
                        return      [m0v m0v'];
            end

            m0v = [top(0)  bottom(0)];
            R1 = m0v;
            //One level conjugate
            R2 = [m0v(0) -m0v(1)];
            return      [R1 R2];
end
```

```
////////////////////////////////////////////////
//Vectorize an 2x2 matrix of elements and merge it
//into a vector, the elements can be vector too.
//scalars, is 0 or 1, 1 if only contains scalar
//without imaginary numbers
//scalar must be 1 for the first merge
//return new merged vector
function quadMerge(top, bottom)
            len1 = max(size(top(0)))(0);
            len2 = max(size(bottom(0)))(0);
            if len1 != len2
                        error("quadMerge-> elements must have same dimension\n");
                        return 0;
            end
            if          isArray(top) == 0
                        m0v = real(top)  + 1i * real(bottom);
                        m1v = imag(top) + 1i * imag(bottom);
            else
                        m0v = [top(0)  bottom(0)];
                        m1v = [top(1)  bottom(1)];
            end
            v1=(m0v - vxI(m1v))/2;
            v2=(m0v + vxI(m1v))/2;
            return [v1 v2];
end
```