Red Matrix Object

Toomas Vooglaid
François Jouen
Xie Qingtian

Matrices with Red

A matrix is just an array of numbers. A matrix is usually shown by a capital letter (such as A, or B). Matrices are organized by rows and columns and each element of a matrix can be addressed by an index calculated from row and column position.

$$A=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$$

Unfortunately, matrix datatype! is not supported in Red language due to the complexity of the actions to be implemented for a matrix! native datatype. In Red, most of the predefined actions in native datatype are for series-like datatypes. Those actions are not really suitable for matrix.
The way we adopted is an object-oriented approach, and thus matrix is a generic Red object which allows to create and process matrices. Matrix object supports char, integer and float values. Depending on matrix type, you can use 8, 16, 32 and 64 bit-sizes. Matrix object contains a lot of methods for mathematics on matrices. Matrix object also allows to create *another mx object* with properties stored in integer! fields and data stored as vector. Matrix/methods are applied to *mx* object. *mx* constructor takes following arguments:

mType: matrix type as integer [1: char, 2: integer, 3: float]
mBits: bit-size as integer [8|16|32 for char and integer, 32|64 for float!]
mSize: matrix size as pair with COLSxROWS (e.g 3x3)
mData: matrix values as block transformed into vector for fast computation

When element address is given as pair MxN, it follows Red semantics of COLxROW, and when element address is given as two integers M and N, the order is ROW COL as conventional in math literature. In both cases, rows go left-right and columns go up-down.

Creating matrices

Several syntactic forms are supported for creating matrices:

matrix/create
With this first matrix object method, you have to use a block of values as parameter. You have also to control that the length of your data block equals to the *mx* size.
mx: matrix/create 1 8 3x3 []: Creates a 3x3 8-bit char matrix with no data.
mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]: This creates a 3x3 matrix of 32-bit integer! filled with data.

$$\mathrm{mx} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

matrix/init
This second method is a short-cut method for creating specific matrices. In this case, you don't need data block. *mx/data* field is automatically filled according to a *refinement* use.

m1: matrix/init 2 16 3x3: Creates a 3x3 integer matrix filled with 0 (no value refinement).

$$\mathrm{m1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

m2: matrix/init/value 2 16 3x3 9: Creates a 3x3 integer matrix filled with a value, here 9.

$$\mathrm{m2} = \begin{bmatrix} 9 & 9 & 9 \\ 9 & 9 & 9 \\ 9 & 9 & 9 \end{bmatrix}$$

m3: matrix/init/value/rand 2 16 3x3 255: Creates a 3x3 integer matrix filled with a random value.

$$\mathrm{m3} = \begin{bmatrix} 191 & 188 & 207 \\ 191 & 184 & 85 \\ 12 & 155 & 214 \end{bmatrix}$$

m4: matrix/init/value/rand/bias 2 16 3x3 255 -2: Creates a 3x3 integer matrix filled with a random value + a bias

$$\mathrm{m3} = \begin{bmatrix} 189 & 186 & 205 \\ 189 & 182 & 83 \\ 10 & 153 & 212 \end{bmatrix}$$

Matrix object also knows how to create *specific matrices* very useful for matrix mathematics.

matrix/scalar

m5: matrix/scalar 2 16 3x3 5: A matrix with all main diagonal entries the same (here 5), and with 0s everywhere else.

$$\mathrm{m5} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

matrix/identity

m6: matrix/identity 2 16 3x3: An identity matrix (I) has 1s on the main diagonal and 0s everywhere else.

$$\mathrm{m6} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

matrix/zero

m7: *matrix/*zero 2 16 3x3: Zero (null matrix) with 0s just everywhere.

$$\mathrm{m7} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

This is equivalent to *mx: matrix/init 2 16 3x3*

Testing matrices properties

For many math operators, preliminary control of matrices properties is required before computation. Matrix object includes such as controls.

matrix/square?
This method controls that we use a square matrix *with the same number of rows as columns* whatever the matrix size.

$$\mathrm{S} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \text{true}$$

matrix/null?
Is matrix null with *0s everywhere*?

$$\mathrm{m7} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \text{true}$$

$$\mathrm{m7} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \text{false}$$

matrix/ singular?
A square matrix that does not have a matrix inverse. A matrix is singular if its determinant is 0. For example, there are 10 singular 2×2 (0 1) matrices:

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$
$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

https://mathworld.wolfram.com/SingularMatrix.html

matrix/nonSingular?
Inverse operation.

matrix/degenerate? and **matrix/nonDegenerate?**
A square matrix whose determinant is equal to 0 (matrix/degenerate?) or not equal to 0 (matrix/nonDegenerate?).

matrix/ invertible?
In linear algebra an n-by-n square matrix A is called invertible (or nonsingular or nondegenerate), if there exists an n-by-n matrix B such that

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I}_n$$

where I_n denotes the n-by-n identity matrix and the multiplication used is ordinary matrix multiplication. If this is the case, then the matrix B is uniquely determined by A and is called the inverse of A, denoted by A^{-1}.

matrix/diagonal?
A diagonal matrix has *0s anywhere not on the main diagonal*.

$$\mathrm{mx} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

matrix/symmetric?
In a symmetric matrix, elements either side of the main diagonal are *equal*. It must be square, and is equal to its own transpose ($A = A^{T)}$.

$$A = \begin{bmatrix} 3 & 12 & 11 & 5 \\ 12 & 9 & -1 & 6 \\ 11 & -1 & 0 & 7 \\ 5 & 6 & 7 & 9 \end{bmatrix}$$

matrix/lower?
Lower triangular matrix is when all elements above the main diagonal are 0s.

$$\mathrm{l} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{bmatrix}\text{=true}$$

matrix/upper?
Upper triangular matrix is when all elements below the main diagonal are 0s.

$$\mathrm{u} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}\text{=true}$$

Rows and columns manipulation

Matrix object use a basic method for displaying mx matrix: *matrix/show*. We will use this function from now to illustrate the result of operators. In matrix object, a lot of methods allowing to manipulate both rows and columns are implemented for an easier access to matrix elements.

matrix/getCol
This function returns a 1-D matrix from column n as a vector!
matrix/show mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]

```
 [
1 2 3
4 5 6
7 8 9
]
```

Now we can get each column:

```
v: matrix/getCol mx 1 >> make vector! [1 4 7]
v: matrix/getCol mx 2 >> make vector! [2 5 8]
v: matrix/getCol mx 3 >> make vector! [3 6 9]
```

Of course, you can use the returned vector to create a new matrix with transforming the vector to a block.
matrix/show mx1: matrix/create 2 32 3x0 to-block v

```
[
   3 6 9
]
```

matrix/getRow
Similar operation can be performed on rows.

```
v: matrix/getRow mx 1 >> make vector! [1 2 3]
v: matrix/getRow mx 2 >> make vector! [4 5 6]
v: matrix/getRow mx 3 >> make vector! [7 8 9]
```

In both cases, matrices can be not square. Matrix object allows also to append, insert or remove both columns and rows. Since matrix datatype is not a real series-like datatype, we can’t use a generic action such as append, insert, or remove, and thus methods are suffixed to be easily memorized.

matrix/removeRow and **matrix/removeCol**
Both functions remove column or row according to the row or column index and return a new mx object.

mx2: matrix/removeCol mx 1
matrix/show mx2

```
[
  2 3
  5 6
  8 9
]
```

mx2: matrix/removeRow mx 2
matrix/show mx2

```
[
  1 2 3
  7 8 9
]
```

matrix/insertRow and matrix/insertCol
We can also insert rows and columns in a *mx* object. Data to be inserted are stored in a block. The length of the data block must equal to matrix M or N size. You can also use a data block with a sole value, and in this case, the value is duplicated in the *mx* object. *InsertRow* and *insertCol* modify the original *mx* object. If you want keep the *mx* unchanged, you need to make a copy before modification with *matrix/_cpy* method.

mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]
matrix/insertRow mx [-2 -1 0]
matrix/show mx

```
[
  -2 -1 0
  1 2 3
  4 5 6
  7 8 9
]
```

mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]
matrix/insertRow mx [0]
matrix/show mx

```
[
  0 0 0
  1 2 3
  4 5 6
  7 8 9
]
```

Similar insertion can be performed for columns.

mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]
matrix/insertCol mx [-1 0 1]
matrix/show mx2

```
[
  -1 1 2 3
  0 4 5 6
  1 7 8 9
]
```

mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]
mx2: matrix/insertCol mx [0]
matrix/show mx2

```
 [
  0 1 2 3
  0 4 5 6
  0 7 8 9
]
```

matrix/appendRow and matrix/appendCol
These methods are used for appending rows and columns value with the same requirement for the data length. *appendRow* and *appendCol* also modify the original *mx* object. If you want keep the *mx* unchanged, you need to make a copy before modification with *matrix/_cpy* method.
In the next example, we will apply successively *appendRow* and *appendCol* to add a row of 0s and a column of 0s.

mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]
matrix/AppendRow mx [0]
matrix/appendCol mx [0]
matrix/show mx

```
[
  1 2 3 0
  4 5 6 0
  7 8 9 0
  0 0 0 0
]
```

Naturally both insert and append operators can be chained for creating sophisticated *mx* object.

```
[
  0 0 0 0 0
  0 1 2 3 0
  0 4 5 6 0
  0 7 8 9 0
  0 0 0 0 0
]
```

matrix/switchRows and matrix/swtichCols
Both methods allow to switch either 2 rows or 2 columns inside a matrix. You have to document the first row or column to switch and the row or column with which to switch.

```
mx1: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]
matrix/show mx1
matrix/switchRows mx1 1 3
matrix/show mx1
matrix/switchCols mx1 1 3
matrix/show mx1
```

```
[
  1 2 3
  4 5 6
  7 8 9
]
[
  7 8 9
  4 5 6
  1 2 3
]
[
  9 8 7
  6 5 4
  3 2 1
]
```

matrix/augment
This method is really useful when you need to combine 2 matrices *mx1* and *mx2* into a larger one. Both matrices must be of same type and bitSize and have the same number of rows.

```
mx1: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]
matrix/show mx1
mx2: matrix/create 2 32 2x3 [1 0 0 1 1 0]
matrix/show mx2
mxa: matrix/augment mx1 mx2
matrix/show mxa
```

```
[
  1 2 3
  4 5 6
  7 8 9
]
[
  1 0
  0 1
  1 0
]
[
  1 2 3 1 0
  4 5 6 0 1
  7 8 9 1 0
]
```

matrix/split
This operator allows to extract data from a column n. If we used the *mxa* calculated above, with *matrix/split mxa 4*, we extract the first three columns.

```
[
  1 2 3
  4 5 6
  7 8 9
]
```

matrix/slice
This operator allows to extract data from a starting column and a starting row to an ending column and an ending row.

bf: [1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0]
m3: matrix/create 3 64 4x4 bf
matrix/show m3
print " Slice 2 3 2 3"
mx: matrix/slice m3 2 3 2 3
matrix/show mx
print " Slice 3 4 3 4"
mx: matrix/slice m3 3 4 3 4
matrix/show mx

```
[
  1.0 2.0 3.0 4.0
  5.0 6.0 7.0 8.0
  9.0 10.0 11.0 12.0
  13.0 14.0 15.0 16.0
]
Slice 2 3 2 3
```

```
[
   6.0 7.0
   10.0 11.0
]
 Slice 3 4 3 4
[
   11.0 12.0
   15.0 16.0
]
```

Matrix transforms

matrix/transpose
 A transpose matrix (T) is a matrix where we swap elements across the main diagonal (rows become columns): *matrix/show mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]*

```
[
   1 2 3
   4 5 6
   7 8 9
]
```

and *matrix/show matrix/transpose mx*

```
[
   1 4 7
   2 5 8
   3 6 9
]
```

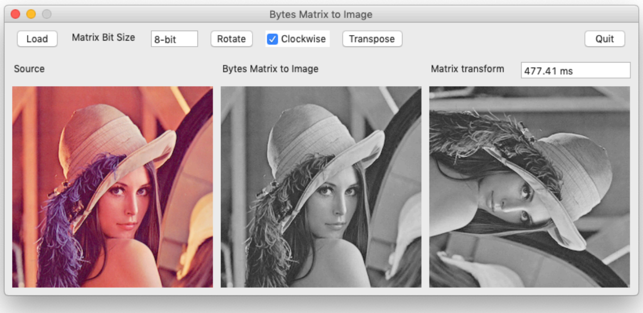
matrix/rotate
Given a matrix, positive (clockwise) or negative (counterclockwise) rotate elements in it.
matrix/show mx1: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]

```
 [
   1 2 3
   4 5 6
   7 8 9
]
```

matrix/show matrix/rotate mx1 1

```
[
   7 4 1
   8 5 2
   9 6 3
]
```

matrix/rotate is really useful for image processing.



matrix/negative

The negative of a matrix is simple: we negate each element of the matrix.

matrix/show mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]

matrix/show matrix/negative mx

```
[
   1 2 3
   4 5 6
   7 8 9
]
[
   -1 -2 -3
   -4 -5 -6
   -7 -8 -9
]
```

Matrix statistics

Matrix object includes several methods for statistics on matrices.

matrix/product
This method returns the matrix product as float value.

matrix/ sigma
Returns the sum of the matrix as a float value.

matrix/mean
Returns the mean value of the matrix as a float value

matrix/mini
Returns the minimal value of the matrix.

matrix/maxi
Returns the maximal value of the matrix.

```
matrix/show mx: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]
print [pad "Product: " 10  matrix/product mx]
print [pad "Sum: " 10 matrix/sigma mx]
print [pad "Mean: " 10 matrix/mean mx]
print [pad "Mini: " 10 matrix/mini mx]
print [pad "Maxi: " 10 matrix/maxi mx]
```

```
[
    1 2 3
    4 5 6
    7 8 9
]
Product:   362880.0
Sum:       45.0
Mean:      5.0
Mini:      1
Maxi:      9
```

Matrix computation

A lot of operators can be used for matrices computation.

matrix/addition

Add two matrices and returns a new matrix. Matrices must be similar.

```
m1: matrix/create 2 16 2x2 [3 8 4 6]
m2: matrix/create 2 16 2x2 [4 0 1 -9]
matrix/show m1
matrix/show m2
print "Addition"
mx: matrix/addition m1 m2
matrix/show mx
```

```
[
   3 8
   4 6
]
[
   4 0
   1 -9
]
Addition
[
   7 8
   5 -3
]
```

matrix/subtraction

Subtract two matrices and returns a new matrix. Matrices must be similar.

```
print "Subtraction"
mx: matrix/subtraction m1 m2
matrix/show mx
```

```
Subtraction
[
   -1 8
   3 15
]
```

matrix/standardProduct

Returns a new matrix as standard multiplication of 2 matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix.

```
m1: matrix/create 2 16 3x2 [1 2 3 4 5 6]
m2: matrix/create 2 16 2x3 [7 8 9 10 11 12]
print "Multiplication"
mx: matrix/standardProduct m1 m2
```

matrix/show mx

```
Multiplication
[
   58 64
   139 154
]
```

matrix/Hadamardproduct

This operator that takes two matrices of the same dimensions and produces another matrix of the same dimension as the operands, where each element m, n is the product of elements m, n of the original two matrices.

print "Hadamard product"
m1: matrix/create 2 16 3x2 [1 2 3 4 5 6]
m2: matrix/create 2 16 3x2 [7 8 9 10 11 12]
mx: matrix/HadamardProduct m1 m2
matrix/show mx

```
Hadamard product
[
   7 16 27
   40 55 72
]
```

matrix/KroneckerProduct

If **A** is an $r \times s$ matrix with ijth element a_{ij} for $i = 1,\ldots, r$ and $j = 1,\ldots, s$, and **B** is any $t \times v$ matrix, then the Kronecker product of **A** and **B,** denoted by **A** $\otimes$ **B,** is the $rt \times sv$ matrix formed by multiplying each a_{ij} element by the entire matrix **B.**

print "Kronecker product"
mx: matrix/KroneckerProduct m1 m2
matrix/show mx

```
Kronecker product
[
   7 8 9 14 16 18 21 24 27
   10 11 12 20 22 24 30 33 36
   28 32 36 35 40 45 42 48 54
   40 44 48 50 55 60 60 66 72
]
```

matrix/division

The division of two matrices being a mathematical nonsense, multiplying by the inverse of a matrix remains the closest thing to a division.

print "Division"
m1: matrix/create 2 16 2x2 [3 5 4 7]
m2: matrix/create 2 16 2x2 [7 -5 -4 3]
mx: matrix/division/right m1 m2
matrix/show mx

```
[
   29 50
   40 69
]
```

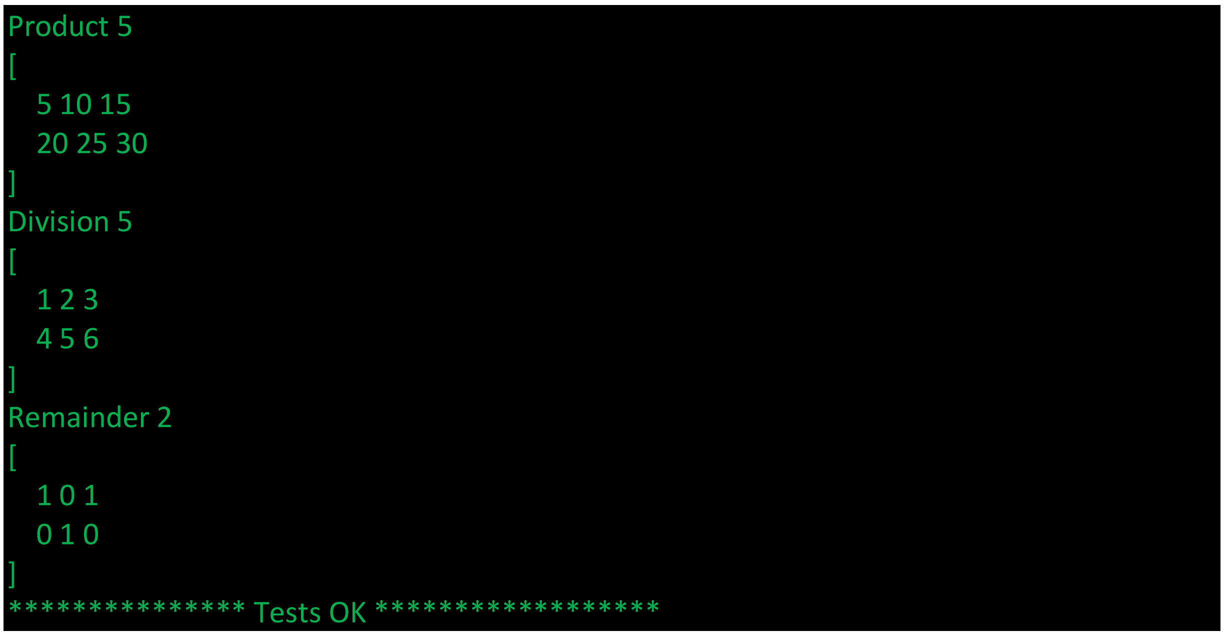
Scalar operators

You can also use scalar operators on matrices.

scalarAddition	matrix + value
scalarSubtraction	matrix - value
scalarProduct	product of scalar multiplication of the matrix
scalarDivision	matrix / value
scalarRemainder	matrix % value

```
print "*********** Scalar *****************"
mx1: matrix/create 2 16 3x2 [1 2 3 4 5 6]
print "Matrix"
matrix/show mx1
print "Addition 5"
mx2: matrix/scalarAddition mx1 5
matrix/show mx2
print "Subtraction 5"
mx1: matrix/scalarSubtraction mx2 5
matrix/show mx1
print "Product 5"
mx2: matrix/scalarProduct mx1 5
matrix/show mx2
print "Division 5"
mx1: matrix/scalarDivision mx2 5
matrix/show mx1
print "Remainder 2"
mx2: matrix/scalarRemainder mx1 2
matrix/show mx2
print "*************** Tests OK *****************"
```

```
*********** Scalar ******************
Matrix
[
  1 2 3
  4 5 6
]
Addition 5
[
  6 7 8
  9 10 11
]
Subtraction 5
[
  1 2 3
  4 5 6
]
```



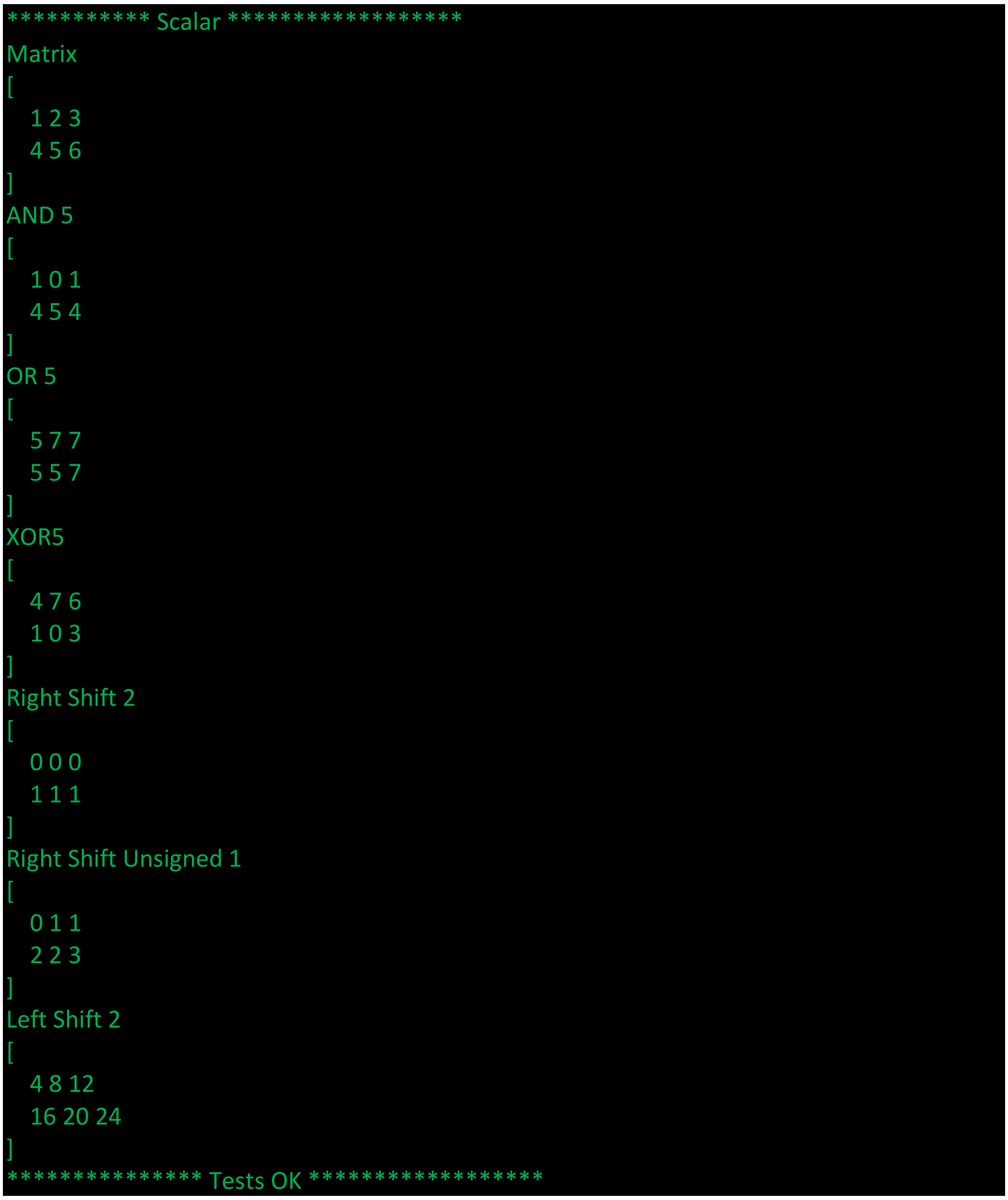
```
Product 5
[
   5 10 15
   20 25 30
]
Division 5
[
   1 2 3
   4 5 6
]
Remainder 2
[
   1 0 1
   0 1 0
]
*************** Tests OK ******************
```

Logical operators with scalars are also possible.

scalarAnd	matrix AND value
scalarOr	matrix OR value
scalarXor	matrix XOR value
scalarRightShift	matrix right shift (>>)
scalarRightShiftUnsigned	matrix right shift (unsigned >>>)
scalarLeftShift	matrix left shiht (<<)

```
print "*********** Scalar *****************"
mx1: matrix/create 2 16 3x2 [1 2 3 4 5 6]
print "Matrix"
matrix/show mx1
print "AND 5"
mx2: matrix/scalarAnd mx1 5
matrix/show mx2
print "OR 5"
mx2: matrix/scalarOr mx1 5
matrix/show mx2
print "XOR5"
mx2: matrix/scalarXor mx1 5
matrix/show mx2
print "Right Shift 2"
mx2: matrix/scalarRightShift mx1 2
matrix/show mx2
print "Right Shift Unsigned 1"
mx2: matrix/scalarRightShiftUnsigned mx1 1
matrix/show mx2
print "Left Shift 2"
mx2: matrix/scalarLeftShift mx1 2
```

```
matrix/show mx2
print "*************** Tests OK *****************"
```



```
*********** Scalar ******************
Matrix
[
   1 2 3
   4 5 6
]
AND 5
[
   1 0 1
   4 5 4
]
OR 5
[
   5 7 7
   5 5 7
]
XOR5
[
   4 7 6
   1 0 3
]
Right Shift 2
[
   0 0 0
   1 1 1
]
Right Shift Unsigned 1
[
   0 1 1
   2 2 3
]
Left Shift 2
[
   4 8 12
   16 20 24
]
*************** Tests OK *****************
```

Matrix decomposition

Matrices decomposition is useful in numerical analysis and linear algebra, and the library proposes a few methods.

matrix/getIdentity
Get (left or right) identity matrix for a given matrix. This function can be used with square and non-square matrices. You need to determine /side ['l | 'r] for non-square matrix.

```
mx1: matrix/create 2 16 3x3 [1 2 3 4 5 6 7 8 9]
print "Square Matrix"
matrix/show mx1
print "Identity"
mx2: matrix/getIdentity mx1
matrix/show mx2
mx1: matrix/create 2 16 3x4 [1 2 3 4 5 6 7 8 9 10 11 12]
print "Not Square Matrix"
matrix/show mx1
print "Identity Left"
mx2: matrix/getIdentity/side mx1 'l
matrix/show mx2
print "Identity Right"
mx2: matrix/getIdentity/side mx1 'r
matrix/show mx2
```

```
Square Matrix
[
   1 2 3
   4 5 6
   7 8 9
]
Identity
[
   1 0 0
   0 1 0
   0 0 1
]
Not Square Matrix
[
   1 2 3
   4 5 6
   7 8 9
   10 11 12
]
Identity Left
[
   1 0 0 0
```

```
  0 1 0 0
  0 0 1 0
  0 0 0 1
]
Identity Right
[
  1 0 0
  0 1 0
  0 0 1
]
```

matrix/LU
lower–upper (LU) decomposition creates a matrix as the product of a lower triangular matrix and an upper triangular matrix.
mx1: matrix/create 3 64 3x3 [2.0 -1.0 0.0 -1.0 2.0 -1.0 0.0 -1.0 2.0]
print "Matrix"
matrix/show mx1
print "LU"
mx2: matrix/LU mx1
print "L"
matrix/show mx2/1
print "U"
matrix/show mx2/2

```
Matrix
[
  2.0 -1.0 0.0
  -1.0 2.0 -1.0
  0.0 -1.0 2.0
]
LU
L
[
  1.0 0.0 0.0
  -0.5 1.0 0.0
  0.0 -0.6666666666666666 1.0
]
U
[
  2.0 -1.0 0.0
  0.0 1.5 -1.0
  0.0 0.0 1.3333333333333335
]
```