MtxVec Expression parser

Version 5.3, © Dew Research 2018

The purpose of the MtxVec expression parser is to allow the users of MtxVec and of the add-on packages to expose more features of the underlying math libraries to their customers.

Getting Started

The syntax supported by the expression parser is very similar to Matlab/Scilab with some exceptions:

1.) The arrays and matrices are 0-based
2.) Division of two integers returns an integer for per element operator: 5./6 = 0, 5/.6 = 0
3.) The concatenation like [1, 2] requires comma between consecutive elements
4.) Consecutive elements are stored in the same row. The first index to access matrix elements is row and the second is column. (row-major ordering).
5.) Many functions and constants are available with aliases: Tan(), tan(); False, false; etc.. (the parser is case-sensitive).
6.) Memory management is to be “auto-reference-counted”, where memory is freed at the end of the function unless the variable is reused before. (To be, because function declaration is not yet possible).
7.) For most operators you can use the ANSI C like syntax. Some operators are also duplicated like for example (“not” and “ ~ ”) and then (“or” and “|”) and (“and” and “&”)…
8.) Precedence of the colon operator is more than +,-,/,* and allows additions 2:3+1 = 3:4 or for example 2:3 + 3:5 = 5:8, but brackets are needed for 2:(n-1).
9.) Built-in types include: string, integer, double, complex, double vector, double matrix, integer vector, integer matrix, boolean, Boolean vector, boolean matrix, custom object. Double matrix and vector can also hold complex values.
10.) Variables are statically typed. Once the type of the variable is set, the value can change, but the type not anymore.
11.) Integer type is 64bit and there will be an error reported in case of "Integer overflow" or "Division by zero".

The type of the variable is determined from the type of the first value assigned. A variable can be undefined with a call to undefine(myvar1, myvar2,…). Once undefined, it can again receive a new type.

To define a real value:

```
a = 0.0
```

To define a 64bit integer:

```
a = 0
```

Conversion is possible:

```
b = Boolean(a), I = integer(a), d = double(a)
```

cast to integer is the same as:

```
I = TruncToInt(a)
```

To declare an integer vector:

```
a = [1,2,3,4]
```

To declare a double vector:

```
a = [1.0,2,3,4]
```

at least one parameter needs to be a double. To declare complex vector:

```
a = [1+1i, 2,3,4]
```

The imaginary part even if 1 needs to be always preceded with a number. To separate rows use the semicolon:

```
A = [1+1i, 2; 3, 4]  //2x2 complex value matrix
```

To define an if-clause:

```
If b == 2
   a = 2
end
```

To define an If-else clause

```
If b == 2
   a = 2
else
   a = 3
end;
```

Semicolons can be present or absent. They don't affect the flow of the code. The flow control keywords like "if", "else", "end", "continue", "break", "while", "for" need to be on their own lines. The while loop:

```
while (b == 2)
   a = 2
end
```

The "condition" can be in brackets or not. And the for-loop:

```
for k = 1:10
   j = j + 1
end;
```

The for-loop has one more flavor:

```
j = 0;
a = [4,5,6]
for k = a
   j = j + k
end; //computes the sum of values in a and returns 15
```

where the iterator will hold the value of the vector element from the current iteration. Comments need to be preceded with "//". Functions with multiple results, can be written like this:

```
(mag, phase) = CartToPolar(FFT([0:511]))
```

And of course, the most powerful features are index selectors:

```
A(0:2,2:3) = B(2:4;5:6); //to copy sub-matrix
```

```
a = [1,0,3,4];
mask = a <> 0;
d = a(mask); //perform gather operator by mask
d = d + 2; //modify the selected consecutive values with vector math
a(mask) = d; //scatter them back out
```

You can also omit the second parameter in the range operator:

```
a(2:) = b;
```

This will be executed as:

```
a(2:(2+length(b)-1)) = b;
```

To copy entire contents of "b" in to "a" starting at index 2.

Example for “break” or “continue”:

```
for k = a   //on its own line
   if k > 2   //on its own line
      continue //on its own line
   end    //on its own line
   j = j + k
end; //on its own line
```

To copy values to a string grid like object for display, it is possible to define:

```
grid1var := TExprGridVariable.Create;
expr.DefineCustomValue('sheet1', grid1var);
```

Then it is possible to write string values with implicit conversion to string like this:

```
sheet1(1,1) = "Test";
sheet1(0:2,1) = vector(0:2);
```

And of course to read values also:

```
a = sheet1(0,0)
b = sheet1(0:2,0:3) //matrix 3x4 (convert from string to double)
```

String operations include concatenation:

```
s = s + " " + "test"
```

```
s = ["Testing = ", k] //all items in the list are converted to string
```

Additionally, there are several string replace and compare functions available: StrToSample, StrToCplx, CplxToStr, SampleToStr, FormatSample, ReplaceStr, CompareStr and Pos.

Some functions accept type as parameter:

```
b = sheet1(0:2,0:3, "integer") //returns integer matrix
```

Even though the "integer" type is a parameter and the string there could be a variable, the type evaluation is static and happens only once at first run. The line is therefore locked to return integer matrices only.

Possible automatic type promotions in expressions:

A	B	Math operation
Integer	Double	b = double(a)
Bool	Integer	b = integer(a)

Double	Complex	a = Cplx(a)

In general, all type conversions need to be written explicitly. This is to prevent hidden, automatic, repetitive and costly conversions from integer to double or similar within loops. Below are several tables showing the type combinations supported by the assign operator when dealing with indexes and ranges.

Types

Type	Purpose
Undefined	State of variable until first value is assigned. The type is then set to the type of the assigned value.
Double	Real number in double precision or single precision (depends on library build)
Range	Two or three integer or real values in format 1:10 or 10:-1:0
String	"Something"
Complex	Real and Imaginary part in double precision (struct).
Vector	Double precision array. Assumed orientation is single row, but can also be accepted as a column by some routines. Vector can hold also complex numbers.
Matrix	Double precision 2D matrix. Matrix can also hold complex numbers. Rows are stored consecutively.
Integer	Integer number
Integer vector	1D array of 32bit integers, 16bit integers or 8bit unsigned integers. Assumed orientation is single row, but can also be accepted as a column by some routines.
Integer matrix	2D array of 32bit integers, 16bit integers or 8bit unsigned integers. Rows are stored consecutively.
Boolean	Can be True or False. Storage format is 32bit integer.
Boolean vector	1D array of 32bit boolean values. Assumed orientation is single row, but can also be accepted as a column by some routines.
Boolean matrix	2D array of 32bit boolean values. Rows are stored consecutively.
Custom	Used for arbitrary object types. Custom value typed object cannot be assigned to other variables with the assign operator.

Operators

Operator	Priority	Description

`!x`
`~x`
`not x`
`Not x`
	10	Logical operator when applied to booleans and bitwise not operator when applicable to integers.

`x % y` `x mod y`	10	Remainder operator applicable to integers and real numbers
`x div y`	30	Strictly integer division only operator.

`x != y`
`x <> y`
`x ~= y`
	55	Not equal to operator

`x & y` `x and y`	70	Logical "and" when applied to Booleans and bitwise "and" when applied to integers
`x >> y` `x shr y`	45	Bit shift integer, integer vector or integer matrix x by y bits right
`x << y` `x shl y`	45	Bit shift integer, integer vector or integer matrix x by y bits left
`x or y` `x | y`	80	Logical "or" when applied to Booleans and bitwise "or" when applied to integers
`x xor y`	70	Logical "xor" when applied to Booleans and bitwise "xor" when applied to integers
`x = y`	200	Assignment operator
`x == y`	55	Equal operator returns boolean or mask of Booleans
`x'`	10	Unary transpose (adjungate) vector/matrix operator
`x * y`	30	Matrix multiplication operator
`x *. y` `x .* y`	30	Multiplication by element ignoring dimensions except for Length.
`x + y`	40	Add with matching dimensions. String concatenation.
`x +. y` `x .+ y`	40	Add elements ignoring dimension except for Length.
`x - y`	40	Subtract with matching dimensions.
`x -. y` `x .- y`	40	Subtract elements ignoring dimension except for Length.
`+x`	10	Plus sign
`-x`	10	Minus sign
`x / y`	30	Matrix division operator. Integer division returns real number.
`x /. y` `x ./ y`	30	Division by element ignoring dimensions except for Length. Integer division returns integer.
`A \ y`	30	Back division operator for matrix operations. x = A^(-1)*y is written as x = A\y when A is matrix and y is matrix or vector. Coming from equation Ax = y, where "x" is the solution of the linear system.
`x:y` `x:step:y`	15	Range operator. Can be with integers or real numbers. X represents start and y is the final value. If the step is not specified it is assumed to be 1. Step can be positive or negative.
`x < y`	50	less than operator
`x <= y`	50	less than or equal to operator
`x > y`	50	greater than operator
`x >= y`	50	greater than or equal to operator
`x ^ y`	20	power operator

Assign operator with single parameter a (b) = c

Supported types and operations when **a** is **Vector**:

A	b	c	Math operation
Vector	Integer	Integer	a[b] = c
Vector	Integer	Double	a[b] = c
Vector	Integer	Complex	a[b] = c
Vector	Integer	Bool	a[b] = integer(c)
Vector	Range (i:j) or (i:j:k)	Integer	a[i:j] = c, scatter
Vector	Range (i:j) or (i:j:k)	Double	a[i:j] = c, scatter
Vector	Range (i:j) or (i:j:k)	Complex	a[i:j] = c, scatter
Vector	Range (i:j) or (i:j:k)	Bool	a[i:j] = Integer(c), scatter
Vector	Range (i:j) or (i:j:k)	Vector	a[i:j] = c, copy or scatter
Vector	Integer vector	Integer	a[b] = c, scatter by indices
Vector	Integer vector	Double	a[b] = c, scatter by indices
Vector	Integer vector	Complex	a[b] = c, scatter by indices
Vector	Integer vector	Bool	a[b] = c, scatter by indices
Vector	Integer vector	Vector	a[b] = c, scatter by indices
Vector	Boolean vector	Integer	a[b] = c, scatter by mask
Vector	Boolean vector	Double	a[b] = c, scatter by mask
Vector	Boolean vector	Complex	a[b] = c, scatter by mask
Vector	Boolean vector	Bool	a[b] = c, scatter by mask
Vector	Boolean vector	Vector	a[b] = c, scatter by mask

Supported types and operations when **a** is **Integer Vector**:

a	b	c	Math operation
Integer Vector	Integer	Integer	a[b] = c
Integer Vector	Integer	Bool	a[b] = integer(c)
Integer Vector	Range (i:j) or (i:j:k)	Integer	a[i:j] = c, scatter
Integer Vector	Range (i:j) or (i:j:k)	Bool	a[i:j] = Integer(c), scatter
Integer Vector	Range (i:j) or (i:j:k)	Integer Vector	a[i:j] = c, copy or scatter
Integer Vector	Range (i:j) or (i:j:k)	Boolean Vector	a[i:j] = c, copy or scatter
Integer Vector	Integer vector	Integer	a[b] = c, scatter by indices
Integer Vector	Integer vector	Bool	a[b] = c, scatter by indices
Integer Vector	Integer vector	Integer Vector	a[b] = c, scatter by indices
Integer Vector	Integer vector	Boolean Vector	a[b] = c, scatter by indices
Integer Vector	Boolean vector	Integer	a[b] = c, scatter by mask
Integer Vector	Boolean vector	Bool	a[b] = c, scatter by mask

Integer Vector	Boolean vector	Integer Vector	a[b] = c, scatter by mask
Integer Vector	Boolean vector	Boolean Vector	a[b] = c, scatter by mask

Supported types and operations when **a** is **Boolean Vector**:

a	**b**	**c**	**Math operation**
Boolean Vector	Integer	Integer	a[b] = (c <> 0)
Boolean Vector	Integer	Bool	a[b] = c
Boolean Vector	Range (i:j) or (i:j:k)	Integer	a[i:j] = (c <> 0), scatter
Boolean Vector	Range (i:j) or (i:j:k)	Bool	a[i:j] = c, scatter
Boolean Vector	Range (i:j) or (i:j:k)	Boolean Vector	a[i:j] = c, copy or scatter
Boolean Vector	Integer vector	Integer	a[b] = (c <> 0), scatter by indices
Boolean Vector	Integer vector	Bool	a[b] = c, scatter by indices
Boolean Vector	Integer vector	Boolean Vector	a[b] = c, scatter by indices
Boolean Vector	Boolean vector	Integer	a[b] = (c <> 0), scatter by mask
Boolean Vector	Boolean vector	Bool	a[b] = c, scatter by mask
Boolean Vector	Boolean vector	Boolean Vector	a[b] = c, scatter by mask

Supported types and operations when treating **a** as **"flattened" 1D Matrix**:
Index = Cols*Row + Col

A	**b**	**c**	**Math operation**
Matrix	Integer	Integer	a[b] = c
Matrix	Integer	Double	a[b] = c
Matrix	Integer	Complex	a[b] = c
Matrix	Integer	Bool	a[b] = integer(c)
Matrix	Range (i:j) or (i:j:k)	Integer	a[i:j] = c, scatter
Matrix	Range (i:j) or (i:j:k)	Double	a[i:j] = c, scatter
Matrix	Range (i:j) or (i:j:k)	Complex	a[i:j] = c, scatter
Matrix	Range (i:j) or (i:j:k)	Bool	a[i:j] = Integer(c), scatter
Matrix	Range (i:j) or (i:j:k)	Vector	a[i:j] = c, copy or scatter
Matrix	Integer vector	Integer	a[b] = c, scatter by indices
Matrix	Integer vector	Double	a[b] = c, scatter by indices
Matrix	Integer vector	Complex	a[b] = integer(c), scatter
Matrix	Integer vector	Bool	a[b] = c, scatter by indices
Matrix	Integer vector	Vector	a[b] = c, scatter by indices
Matrix	Boolean vector	Integer	a[b] = c, scatter by mask
Matrix	Boolean vector	Double	a[b] = c, scatter by mask
Matrix	Boolean vector	Complex	a[b] = c, scatter by mask
Matrix	Boolean vector	Bool	a[b] = Integer(c), scatter
Matrix	Boolean vector	Vector	a[b] = c, scatter by mask
Matrix	Boolean matrix	Vector	a[b] = c, scatter by mask
Matrix	Boolean matrix	Integer	a[b] = c, scatter by mask
Matrix	Boolean matrix	Double	a[b] = c, scatter by mask

Matrix	Boolean matrix	Complex	a[b] = c, scatter by mask

Supported types and operations when treating **a** as **"flattened" 1D integer Matrix**:
Index = Cols*Row + Col

A	**b**	**c**	**Math operation**
Integer Matrix	Integer	Integer	a[b] = c
Integer Matrix	Integer	Bool	a[b] = integer(c)
Integer Matrix	Range (i:j) or (i:j:k)	Integer	a[i:j] = c, scatter
Integer Matrix	Range (i:j) or (i:j:k)	Bool	a[i:j] = Integer(c), scatter
Integer Matrix	Range (i:j) or (i:j:k)	Integer Vector	a[i:j] = c, copy or scatter
Integer Matrix	Integer vector	Integer	a[b] = c, scatter by indices
Integer Matrix	Integer vector	Bool	a[b] = c, scatter by indices
Integer Matrix	Integer vector	Integer Vector	a[b] = c, scatter by indices
Integer Matrix	Boolean vector	Integer	a[b] = c, scatter by mask
Integer Matrix	Boolean vector	Bool	a[b] = integer(c), scatter
Integer Matrix	Boolean vector	Integer Vector	a[b] = c, scatter by mask
Integer Matrix	Boolean matrix	Integer Vector	a[b] = c, scatter by mask
Integer Matrix	Boolean matrix	Integer	a[b] = c, scatter by mask
Integer Matrix	Boolean matrix	Bool	a[b] = Integer(c), scatter

Supported types and operations when treating **a** as **"flattened" 1D Boolean Matrix**:
Index = Cols*Row + Col

A	**b**	**c**	**Math operation**
Boolean Matrix	Integer	Integer	a[b] = c <> 0
Boolean Matrix	Integer	Bool	a[b] = c
Boolean Matrix	Range (i:j) or (i:j:k)	Integer	a[i:j] = c <> 0, scatter
Boolean Matrix	Range (i:j) or (i:j:k)	Bool	a[i:j] = c, scatter
Boolean Matrix	Range (i:j) or (i:j:k)	Boolean Vector	a[i:j] = c, copy or scatter
Boolean Matrix	Integer vector	Integer	a[b] = c <> 0, scatter by indices
Boolean Matrix	Integer vector	Bool	a[b] = c, scatter by indices
Boolean Matrix	Integer vector	Boolean Vector	a[b] = c, scatter by indices
Boolean Matrix	Boolean vector	Integer	a[b] = c <> 0, scatter by mask
Boolean Matrix	Boolean vector	Bool	a[b] = c, scatter by mask
Boolean Matrix	Boolean vector	Boolean Vector	a[b] = c, scatter by mask
Boolean Matrix	Boolean matrix	Boolean vector	a[b] = c, scatter by mask
Boolean Matrix	Boolean matrix	Integer	a[b] = c <> 0, scatter by mask

Assign operator with two parameters a (b, c) = d

Supported types and operations when treating **a** as **2D Matrix:**

b	**c**	**d**	**Math operation**
Range (i:j) or integer	Range (k:n) or integer	Integer	a[i:j, k:n] = c, scatter
Range (i:j) or integer	Range (k:n) or integer	Double	a[i:j, k:n] = c, scatter
Range (i:j) or integer	Range (k:n) or integer	Complex	a[i:j, k:n] = c, scatter
Range (i:j) or integer	Range (k:n) or integer	Bool	a[i:j, k:n] = Integer(c), scatter
Range (i:j) or integer	Range (k:n) or integer	Vector	a[i:j, k:n] = c, copy
Range (i:j) or integer	Range (k:n) or integer	Matrix	a[i:j, k:n] = c, copy

Supported types and operations when treating **a** as **2D Integer Matrix:**

b	**C**	**d**	**Math operation**
Range (i:j) or integer	Range (k:n) or integer	Integer	a[i:j, k:n] = d, scatter
Range (i:j) or integer	Range (k:n) or integer	Bool	a[i:j, k:n] = Integer(d), scatter
Range (i:j) or integer	Range (k:n) or integer	Integer Vector	a[i:j, k:n] = d, copy
Range (i:j) or integer	Range (k:n) or integer	Integer Matrix	a[i:j, k:n] = d, copy

Supported types and operations when treating **a** as **2D Boolean Matrix:**

B	**c**	**D**	**Math operation**
Range (i:j) or integer	Range (k:n) or integer	Integer	a[i:j, k:n] = d <> 0, scatter
Range (i:j) or integer	Range (k:n) or integer	Bool	a[i:j, k:n] = d, scatter
Range (i:j) or integer	Range (k:n) or integer	Boolean Vector	a[i:j, k:n] = d, copy
Range (i:j) or integer	Range (k:n) or integer	Boolean Matrix	a[i:j, k:n] = d, copy

String handling functions

`CompareStr(a, b)`: returns 0 if both strings match with case sensitivity.
`CompareText(a, b)`: returns 0 if both strings match with case insensitivity.
`CplxToStr(x)`: converts x from complex to string.

`FormatCplx(x, reFormat, imFormat)`:
converts x from string according to format. Examples of good format values:

`" 0.###;-0.###", "+0.###i;-0.###i"`

`FormatSample(x, format)`: converts x from string according to format.
`Pos(SubStr, Str)`: returns -1 if substring is not found and otherwise 0-based index position.

`Pos(SubStr, Str, Offset)`:
returns -1 if substring is not found and otherwise 0-based index position. The search starts at offset.

`ReplaceStr(x, AFromText, AToText)`:
case sensitive replaces FromText to ToText within x.

`ReplaceText(x, AFromText, AToText)`:
case insensitive replaces FromText to ToText within x.

`SampleToStr(x)`: converts x from double to string.
`IntToStr(x)`: converts x from integer to string.

File handling functions

`a = csvRead("C:\Work\File.txt")`:
returns matrix read from the comma delimited text file.

`a = csvRead("C:\Work\File.txt", "integer")`:
returns integer, boolean or double/complex matrix read from the text file delimited with commas.

`a = csvRead("C:\Work\File.txt", "integer", Delimiter)`:
returns integer, boolean or double/complex matrix read from the text file delimited with the Delimiter char.

`csvWrite("C:\Work\File.txt", a)`: writes variable a, which must be vector or matrix, to comma delimited text file.

`csvWrite("C:\Work\File.txt", a, Delimiter)`: writes variable a, which must be vector or matrix, to comma delimited text file.

`DirectoryCreate("C:\Work\File.txt")`: returns True, if the directory was created.

`DirectoryDelete("C:\Work\File.txt")`: returns True, if the directory was deleted.

`DirectoryExists("C:\Work\File.txt")`: returns True, if the directory exists.

`FileClose(aHandle)`: returns aHandle for the file in the path in parameter. To close the file call fileclose.

`FileCopy("Src.dat", "Dst.dat")`: returns True, if the file was copied.

`FileDelete("C:\Work\File.txt")`: returns True, if file was deleted.
`FileExists("C:\Work\File.txt")`: returns True, if the file exists.
`FileMove("Src.dat", "Dst.dat")`: returns True, if the file was moved.

`(aHandle) = FileOpen("C:\Work\File.dat")`: returns aHandle for the file in the path in parameter. To close the file call FileClose.

`Pos = FilePosition(aHandle)`: returns the position Pos within the file designated with aHandle. Use OpenFile to obtain a file handle.

`a = FileRead(aHandle, "double")`: reads a single double value from files current position with aHandle and stores it in a. Use OpenFile to obtain a file handle and CloseFile to close it.

`a = FileRead(aHandle, "double", Count)`: reads Count double value from files current position with aHandle and stores it in to vector a. Use OpenFile to obtain a file handle and CloseFile to close it.

`FileSetPosition(aHandle, Pos)`:
seek to position Pos within the file designated with aHandle. Use OpenFile to obtain a file handle.

`size = FileSize(aHandle)`:
returns the Size of the file designated with aHandle. Use OpenFile to obtain a file handle.

`FileWrite(aHandle, a)`:
write contents of variable a to file designated with aHandle. Use OpenFile to obtain a file handle.

General math functions

All functions accept real and complex values as input where applicable. The input/result can be individual value, vector or matrix:

`Abs, Sin, Sinh, Sec, Sech, Cos, Cosh, Csc, Csch, Tan, Tanh, Cot, Coth,`

`ArcSin, ArcSinh, ArcSec, ArcSech, ArcCos, ArcCosh, ArcCsc, ArcCsch, ArcTan, ArcTan(,), ArcTanh, ArcCot, ArcCoth,`

`Exp, Exp2, Exp10, Ln, Log, Log2, Log10, LogN, Cbrt, Sqrt, Sqr, Cis, Expj, Conj, DegToRad, RadToDeg, Imag, Real, Pow, Power, IntPower, Flip,`

Additionally:

```
IsInf, IsNan, IsInfNan, Lcm, Gcd, Pythag, Min, Max, Rem, Sgn, Cplx
```

The following aliases are applicable:

```
Ln(x) for Log(x)
Expj(x) for Cis(x)
Integer(x) for TruncToInt(x)
IntPow for IntPower
Pow for Power
IsNanInf for IsInfNan
```

Rounding

`Ceil(x)`: the smallest integer greater than or equal to 'x'
`Floor(x)`: the biggest integer less than or equal to 'x'
`Frac(x)`: returns fractional part of a real number.
`Round(x)`: rounds 'x' value to the nearest whole number and returns double
`RoundToInt(x)`: rounds 'x' value to the nearest whole number and returns integer
`Trunc(x)`: truncates 'x' value towards zero and returns double
`TruncToInt(x)`: truncates 'x' value towards zero and returns integer

Vector specific functions

`AutoCorrBiased(x, Lag)`: returns auto-correlation of x.
`AutoCorrNormal(x, Lag)`: returns auto-correlation of x.
`AutoCorrUnbiased(x, Lag)`: returns auto-correlation of x.
`CumSum(x)`:
returns cumulative sum of data in vector x = [1,2,..] => y = [1,1+2,1+2+3,..] or columns in the matrix.

`DCT(x)`: returns discrete cosine transform of x
`Dct(x)`: returns discrete cosine transform of x
`Difference(x, lag)`: returns vector of differences between elements lag a part.
`FFT(x)`: returns 1D FFT of vector or of rows of matrices.
`IDCT(x)`: returns inverse discrete cosine transform of x
`IFFT(x)`: returns 1D Inverse FFT of vector or of rows of matrices.
`IFFTToReal(x)`: returns 1D Inverse real FFT of vector or rows of matrices.
`Integrate(x)`: Integrate all elements in x once.
`Integrate(x, init)`:
Integrate all elements in x using initial values in Init vector Length(Init) times.

`Kron(vec1, vec2)`: returns the Kronecker product between Vec1 and Vec2.
`k = Kurtosis(x)`: computes the fourth moment from the data in x.
`k = Kurtosis(x, mean, stddev)`:
computes the fourth moment from data in x and stddev(x) and mean(x).

`Length(x)`: element count in vector/matrix x
`Max(x)`: largest element in x
`Mean(x)`: average value of elements in x
`Median(x)`: returns median value of data in vector x.

`Min(x)` : smallest element in x
`Norm(x)` : squared norm of complex value
`Norm1(x)` : returns the 1-norm of the matrix x.
`NormC(x)` : returns C-Norm of data in x.
`NormFro(x)` : returns the Frobenius norm of the matrix x.
`NormInf(x)` : returns the Infinity norm of the matrix x.
`NormL1(x)` : returns L1-Norm of data in x.
`NormL2(x)` : returns L2-Norm of data in x.
`Ones(Len)` : returns a vector of ones Len in size.
`x = PolarToCart(amplt, phase)` : converts polar coordinate to cartesian coordinate
`PolyCoeff(Roots)` : returns polynom coefficients from Roots.
`PolyEval(Values, Coeff)` : returns polynom defined with Coeff evaluated at Values.
`PolyRoots(Coeff)` : returns polynom roots from coefficients in Coeff.
`Product(x)` : product of elements in x
`Reverse(x)` : reverses indexes in vector x
`Rms(x)` : root mean square of elements in x
`Rotate(x,offset)` : rotates the data in x by offset left or right
`SortAscend(x)` : sorts values in vector x or values in rows of matrix x.
`SortDescend(x)` : sorts values in vector x or values in rows of matrix x.
`Sum(x)` : sum of elements in x
`SumOfSqr(x)` : returns sum of squares of x.
`ThreshBottom(x, Bottom)` : returns Bottom, if x < Bottom. Bottom must be a scalar.
`ThreshTop(x, Top)` : returns Top, if x > Top. Top must be a scalar.
`Zeros(Len)` : returns a vector of zeros Len in size.

Matrix specific functions

Most vector functions work also on matrices on the per row basis or have a matrix specific variant. The following is additionally available:

`Cols(x)` : column count of matrix x
`CumSum(x)` :
returns cumulative sum of data in vector x = [1,2,..] => y = [1,1+2,1+2+3,..] or columns in the matrix.

`Eig(x)` : returns eigevalues (d) of x
`Eig(x, l, r)` :
returns eigevalues (d) and stores left eigenvectors in l and right eigenvectors in r.

`Eye(r,c)` : returns non-square matrix size r x c of zeros with ones on the main diagonal.
`FFT2D(x)` : returns 2D FFT of matrix x.
`IFFT2D(x)` : returns 2D Inverse FFT of matrix x.
`IFFT2DToReal(x)` : returns 2D Inverse real FFT of matrix x.
`Hankel(firstColumn)` : returns hankel matrix with first column vector specified as parameter.
`LQR(x, Q, R)` : returns the LQR composition. The L is returned as function result and Q and R are stored in to the parameters.
`MtxIntPower(x, n)` : returns matrix x raised to integer power of n.
`MtxPower(x, r)` : returns matrix x raised to arbitrary power r.
`MtxSqrt(x, n)` : returns square root of the matrix.
`Ones(r,c)` : returns a vector or a matrix of ones r x c in size.

`Rotate90(x)`: returns the matrix x rotated by 90 degrees clockwise.
`Rows(x)`: row count of matrix x
`S = SVD(x)`: returns singular values of matrix x.
`S = SVD(x, u, v)`: returns singular values of matrix x and U and V values in parameters..

`X = SVDSolve(a,b,s)`:
returns the svd solution of a linear system AX = B. Singular values are returned in s. Function returns solution x.

`X = SVDSolve(a,b,s, tol)`: returns the svd solution of a linear system AX = B. Singular values are returned in s. Rejection tolerance is specified with tol. Function returns solution x.

`ZScore(x)`: returns zscore of diagonal elements of matrix x.
`Zeros(r,c)`: returns a vector or a matrix of zeros r x c in size.