Volume

2

FOXES TEAM

Reference Guide for Matrix.xla

Matrices and Linear Algebra

REFERENCE GUIDE FOR MATRIX.XLA

Matrices and Linear Algebra

ITALY
leovlp@libero.it

5. Edition
1 Printing: June 2005

Index

Chapter 1

About Matrix.xla

MATRIX.XLA

Matrix.xla is an Excel addin that contains useful functions for matrices and linear Algebra:

Norm. Matrix multiplication. Similarity transformation. Determinant. Inverse. Power. Trace. Scalar Product. Vector Product.

Eigenvalues and Eigenvectors of symmetric matrix with Jacobi algorithm. Jacobi's rotation matrix. Eigenvalues with QR and QL algorithm. Characteristic polynomial. Polynomial roots with QR algorithm. Eigenvectors for real and complex matrices

Generation of random matrix with given eigenvalues and random matrix with given Rank or Determinant. Generation of useful matrix: Hilbert's, Houseolder's, Tartaglia's. Vandermonde's

Linear System. Linear System with iterative methods: Gauss-Seidel and Jacobi algorithms. Gauss Jordan algorithm step by step. Singular Linear System.

Linear Transformation. Gram-Schmidt's Orthogonalization. Matrix factorizations: LU, QR, SVD and Cholesky decomposition.

This tutorial is divided into two parts. The first part explains with practical examples how to solve several basic topics about matrix theory and linear algebra. The second part is the reference manual of Matrix.xla

Why Matrix.xla has same functions that are also in Excel?

Yes. Same functions like determinant, inversion, multiplication, transpose, etc. are both in Excel and in Matrix.xla. They perform the same tasks. And in many case they return the same values. But they are not exchangeable in every situations.

Matrix.xla algorithms are open

The main difference is into the algorithms used; or in other words, in the way that the functions are implemented. In Matrix.xla the algorithms are open and people can verify how each function works. The function that performs matrix inversion in Excel and in Matrix.xla, for example, can give different results, especially in high accuracy calculation. The main difference is that Matrix.xla Inversion function uses the popular Gauss-Jordan algorithm -explained in many books and sites - while the Excel built-in functions are code-proprietary. In other few cases we have simply create new functions to avoid the original verbose names (MTRANSPOSE(), or MATR.TRASPOSTA () in Italian version, are substituted by the more handy MT())

Array functions

What is an array function

A function that returns multiple values is called "array function". Matrix.xla contains lots of these functions. All functions that return a matrix are array functions. Inversion, multiplication, sum, vector product, etc. are examples of array functions. On the contrary, Norm and Scalar product are scalar functions because they return only one value.

In a worksheet, an array function returns always a rectangular (n x m) range of cells. To insert this function, select before the (n x m) range where you want to insert the function, then, you must give the keys sequence CTRL+SHIFT+ENTER; The sequence must be given just after inserting the function parameters. Keep down both key CTRL and SHIFT (do not care the order) and then press ENTER.
If you miss this sequence or give only the ENTER key, the function always returns the first cell of the array

How to insert an array function

The following example explains, step-by-step, how it works

System solution

Assume to have to solve a 3x3 linear system. The solution will be a vector of 3rd dimension.

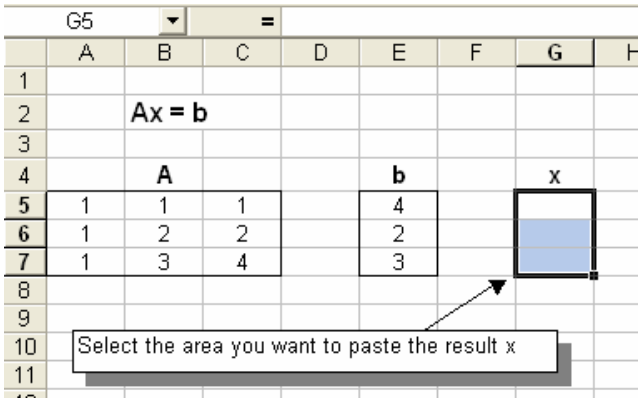
Ax = b

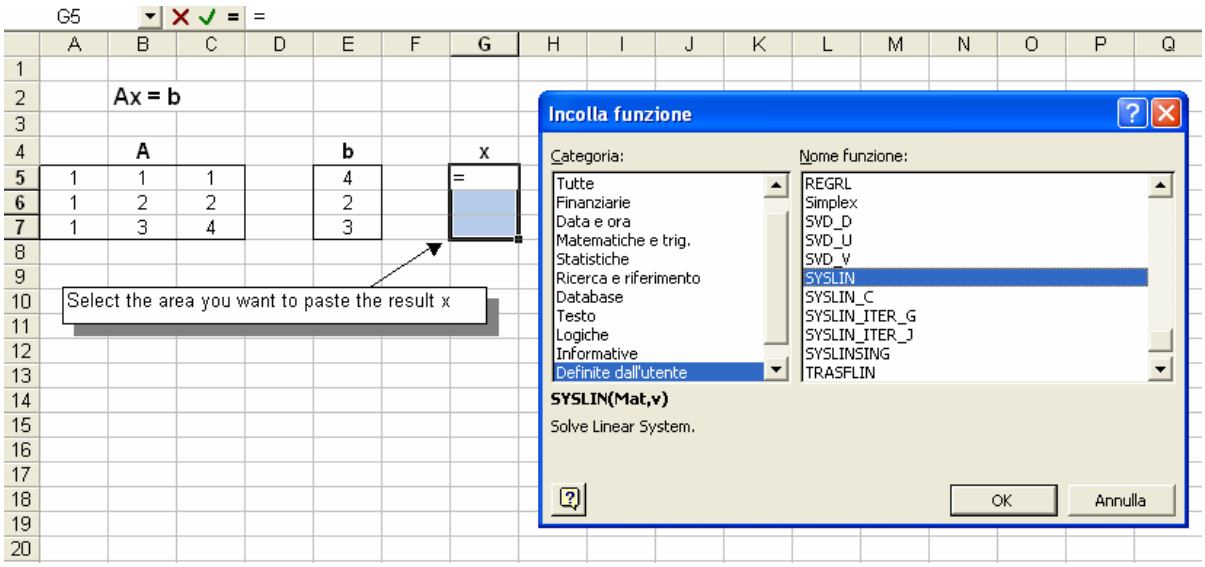
Where:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 3 & 4 \end{bmatrix} \qquad b = \begin{bmatrix} 4 \\ 2 \\ 3 \end{bmatrix}$$

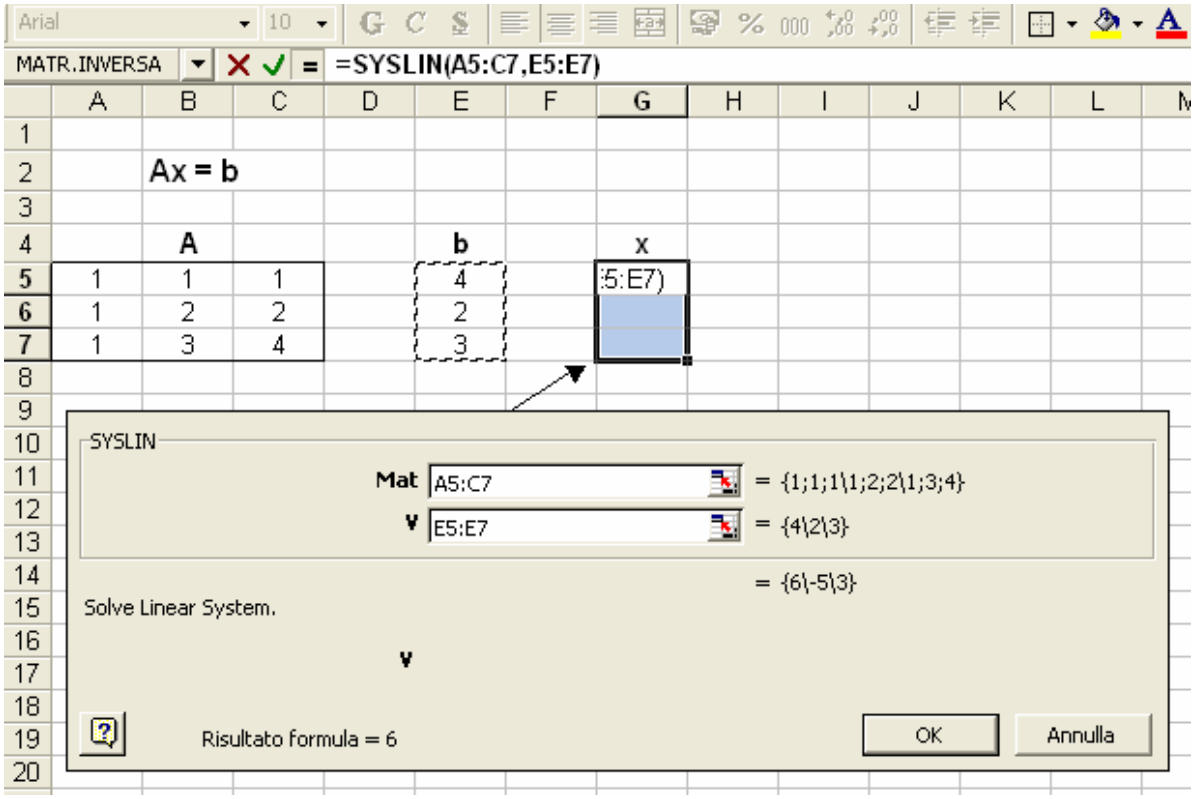
The function **SYSLIN** returns the solution **x**; but to see all the three values you must select before the area where you want to insert these values.

Now insert the function by menu or by icon  as well





Select the area of matrix **A** "A5:C7" and the constant vector **b** "E5:E7"

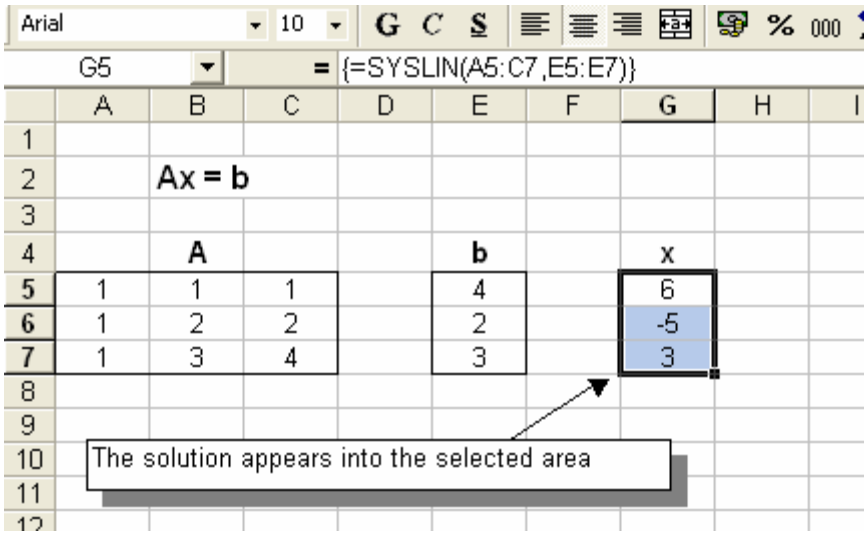


Now - **attention!** - give the "magic" keys sequence CTRL+SHIFT+ENTER

That is:

- Press and keep down the CTRL and SHIFT keys
- Press the ENTER key

All the values will fill all the cells that you have selected.



Note that Excel shows the function around two braces { }. These symbols mean that the function return an array (you cannot insert them by yourself).

An array function has several constrains. Any cell of the array, cannot be modified or deleted. To modify or delete an array function you must selected before all the array's cells.

Adding two matrices

The CTRL+SHIFT+ENTER rule is valid for any function or operation when the result is a matrix or a vector

Example - Adding two matrices $\begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

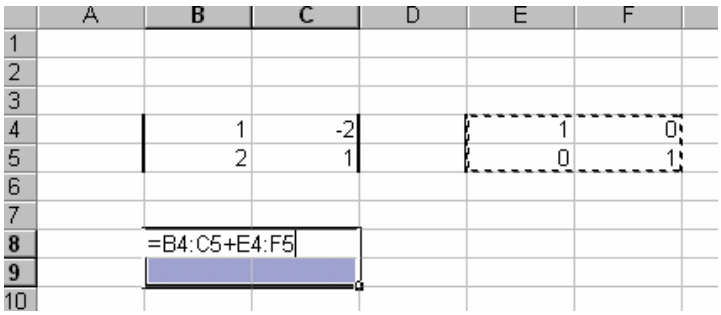
We can use the M_ADD() function of Matrix.xla but we can also use directly the addition operator "+".

In order to perform this addition, follow these steps.

1) Enter the matrices into the spreadsheet.

2) Select empty cells so that a 2 × 2 range is highlighted.

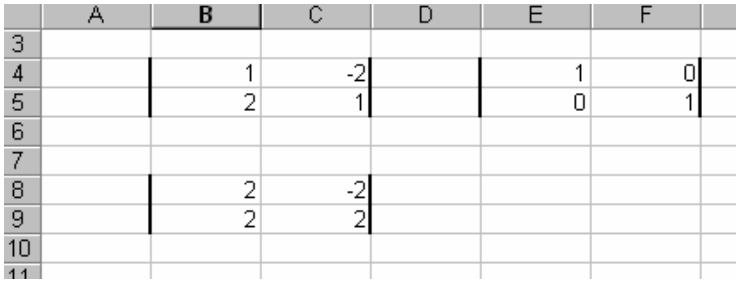
3) Write a formula that adds the two ranges. Either write =B4:C5+E4:F5 directly or write "=", then select the first matrix; after, write "+" and then select the second matrix. Do not press <Enter>. At this point the spreadsheet should look something like the figure below. Note that the entire range B8:C9 is selected.



4) Press and hold down <CTRL> + <SHIFT>

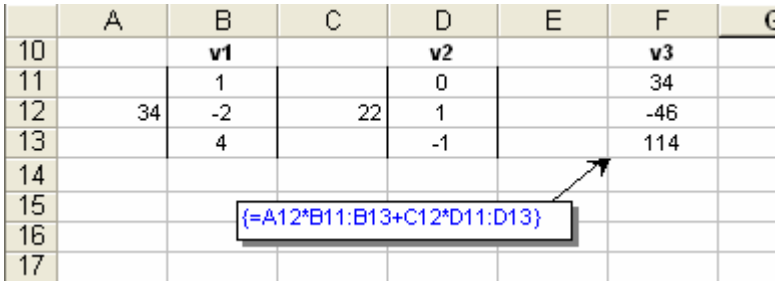
5) Press <ENTER>.

If the procedure is followed correctly, the spreadsheet should now look something like this



This trick can work also for matrix subtraction and for the scalar-matrix multiplication, but not for the matrix-matrix multiplication.

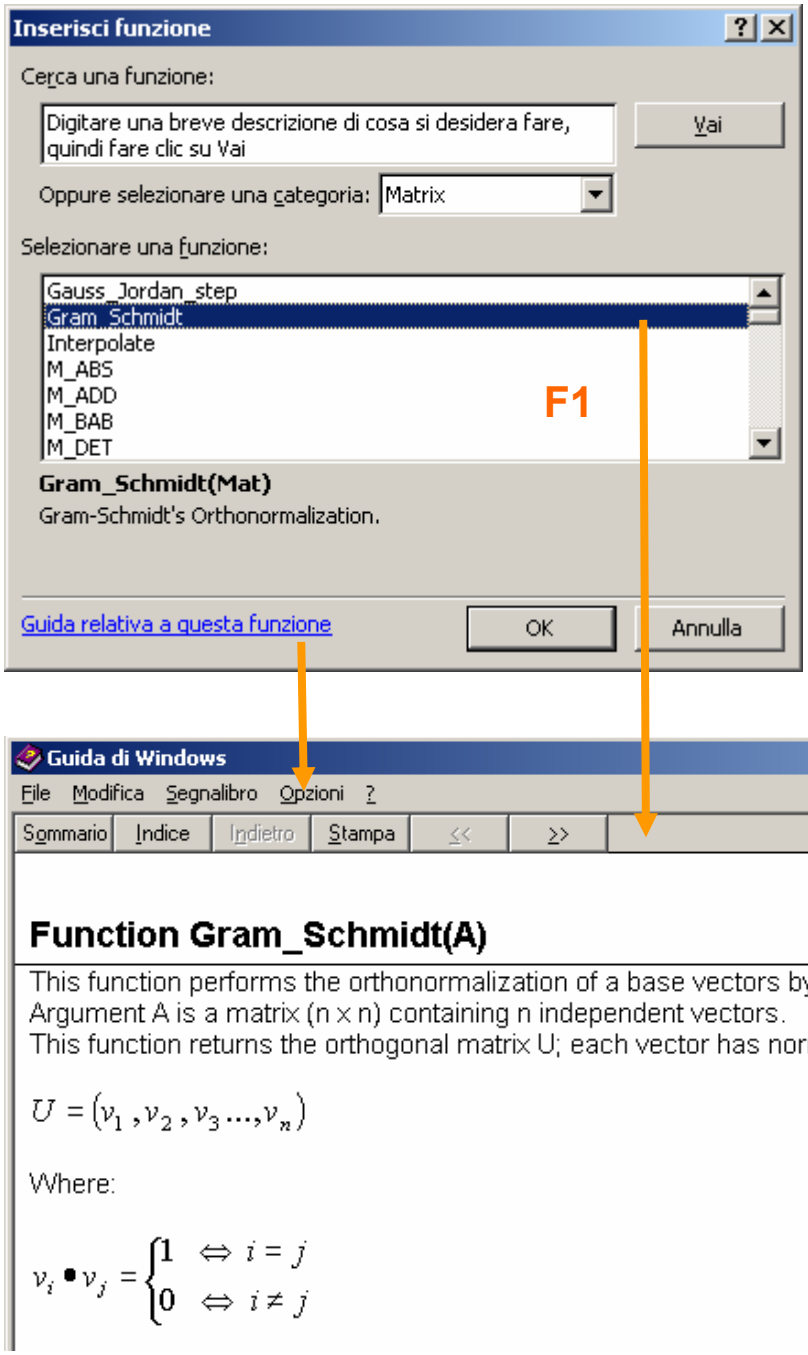
Let's see this example that show how to calculate the linear combination of two vectors



It's useful, isn't?

How to get help on line

Matrix.xla provides help on line that can be recall in the same way of any other Excel function
When you have selected the function that you need in the function wizard, press **F1** key



Note that all the functions of this addin appear under the same category "**Matrix**" in the Excel function wizard

Of course you can call the help on-line also by double clicking on the Matrix.hlp file or from the starting pop-up window or from the "**Matrix Tool**" menu bar

MATRIX installation

MATRIX addin for Excel 2000/XP is a zip file composed by two files:

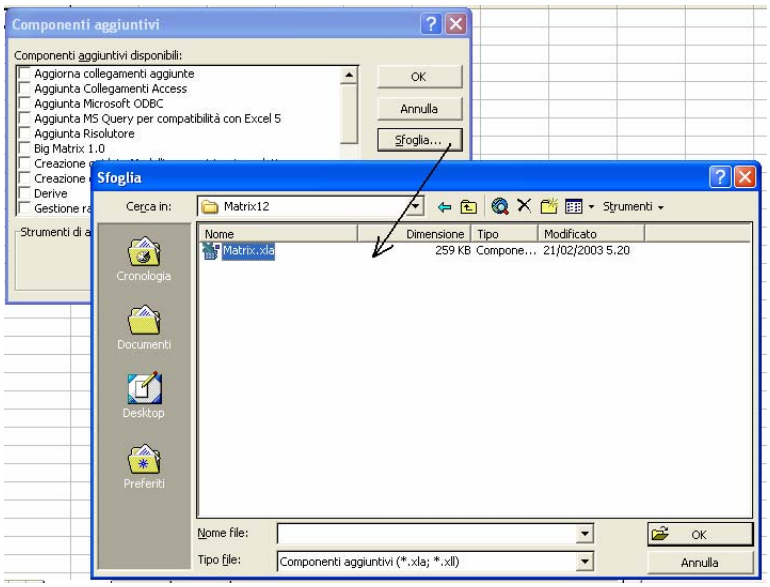
- MATRIX.XLA — Excel addin file
- MATRIX.HLP — Help file
- MATRIX.CSV — Functions information(only for XNUMBERS addin)
- FUNCUSTOMIZE.DLL[1] — Dynamic Library for addin setting

How to install

Unzip and place all the above files in a folder of your choice. The addin is contained entirely in this directory. Your system is not modified in any other way. If you want to uninstall this package, simply delete its folder - it's as simple as that!

To install, follow the usual procedure for installing an Excel addin:

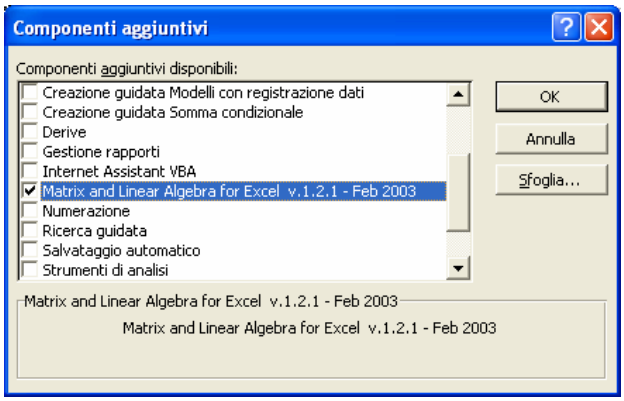
1) Open Excel
2) From the Excel menu toolbar select "Tools" and then select "Add-ins"..
3) Once in the Addins Manager, browse for "**Matrix.xla**" and select it
4) Click OK



Nella versione italiana di Excel, "Addin Manager" si chiama "Componenti aggiuntivi" e si trova nel menu **<Strumenti> < Modelli e aggiunte...>**

[1] FUNCUSTOMIZE.DLL appears by courtesy of Laurent Longre (http://longre.free.fr)

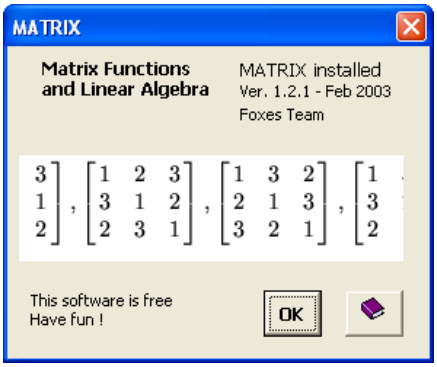
After the first installation, **matrix.xla** will be add to the Addin' list manager



When Excel starts, all addin checked in the Addin Manager will be automatically loaded

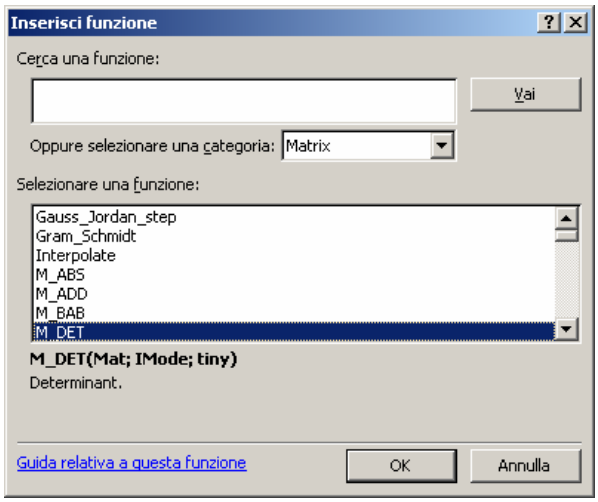
If you want to stop the automatic loading of matrix.xla simply deselect the check box before closing Excel

If all goes OK you should see the welcome popup of matrix.xla. This appears only when you select "on" the check box of the Addin Manager. When Excel automatically loads Matrix.xla, this popup remains hidden.



The Matrix Icon is added to the main menu bar. Clicking on it the Matrix Toolbar appears

The Matrix category. All the functions contained in this addin will be visible by the Excel function wizard under the *Matrix* category.



Update a new version

When you update a new version you must replace the older files with the new version.

Do not keep two different versions on your PC, and, overall, never load two different version because Excel would make mesh.

How to uninstall

This package never alter your system files

If you want to uninstall this package, simply delete its folder. Once you have cancelled the Matrix.xla file, to remove the corresponding entry in the Addin Manager list, follow these steps:

1) Open Excel
2) Select <Addin...> from the <Tools> menu.
3) Once in the Addin Manager, click on the **Matrix.xla**
4) Excel will inform you that the addin is missing and ask you if you want to remove it from the list. Give "yes".

About complex matrix format

Matrix.xla supports 3 different complex matrix formats: 1) split, 2) interlaced, 3) string

Split format

1	2	0	0	-1	3
-1	3	-1	0	2	-1
0	-1	4	0	-2	0

Interlaced format

1	0	2	-1	0	3
-1	0	3	2	-1	-1
0	0	-1	-2	4	0

String format

1	2-i	3i
-1	3+2i	-1-i
0	-1-2i	4

As we can see in the first format the complex matrix [Z] is split into two separate matrices: the first contains the real values and the second one the imaginary values. It is the default format

In the second format, the complex values are written as two adjacent cells, so a single matrix element is fitted in two cells. The columns numbers are the same of the first format but the values are interlaced one real column is followed by an imaginary column and so on.
This format is useful when the elements are returned by complex functions

The last format is the well known "*complex rectangular format*". Each element is written as a string a+ib; therefore the matrix is still squared. Apparently is the most compact and intuitive format but this is true only for integer values. For decimal values the matrix may become illegible. We have also to point out that these elements, being strings, cannot be formatted with the standard tool of Excel.

WHITE PAGE

Chapter 2

Functions Reference

This chapter lists all functions of MATRIX.XLA addin. It is the printable version of the on-line help file MATRIX.HLP

Function M_ABS(V)

Returns the absolute value ||V|| (Euclidean Norm) of a vector V

$$\|V\| = \sqrt{\sum v_i^2}$$

The parameter **V** may be also a matrix; in this case the function returns the Frobenius norm of the matrix

$$\|A\|_F = \sqrt{\sum a_{ij}^2}$$

Function M_ABS_C(V, [Cformat])

Returns the absolute value ||V|| (Euclidean Norm) of a complex vector V
This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
The parameter **V** may be also a matrix; in this case the function returns the Frobenius norm of the matrix
The optional parameter *Cformat* sets the complex input format(default = 1)

	A	B	C	D	E	F	G	H	I	J	K	L
1	vector				matrix				matrix			
2	re	im			re		im					
3	3	1		2	-1	1	0		2 + j	-1		
4	5	-2		0	3	0	1		0	3 + j		
5	0	1										
6	0	-6		4	=M_ABS_C(D3:G4)				4	=M_ABS_C(I3:J4,3)		
7												
8	8.718	=M_ABS_C(A3:B6)										
9												

See About complex matrix format

Function M_ADD(A, B)

Returns the sum of two matrices

$$C = A + B$$

For definition:

$$c_{ij} \equiv a_{ij} + b_{ij}$$

Example of sum of (2 x 2) matrices

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}.$$

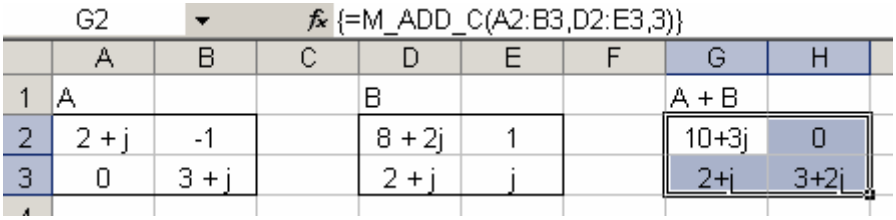
Note: EXCEL has a simply way to performs the addiction of two arrays. For details see How to insert an array function...

Function M_ADD_C(A, B, [Cformat])

Returns the sum of two complex matrices

$$C = A + B$$

This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
Optional parameter *Cformat* sets the complex format of input/output (default = 1)



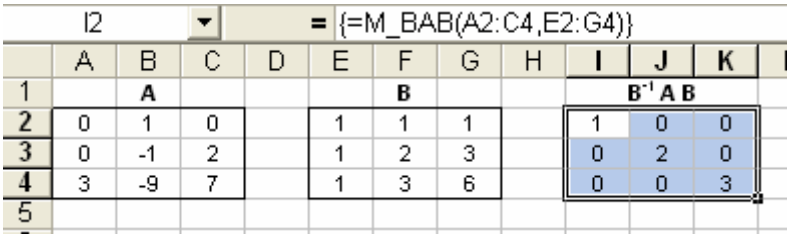
Function M_BAB(A, B)

Returns the product:

$$C = B^{-1} A \; B$$

This operation is also called "*similarity transform*" of matrix A by matrix B
Similarity transform plays a crucial role in the computation of eigenvalues, because they leave the eigenvalues of matrix A unchanged.

For real symmetric matrices, B is orthogonal. The similarity transformation is the also called "*orthogonal transform*"



Function M_DET(Mat, Mat, [IMode], [Tiny])

Returns the determinant of a square (n x n) matrix.
IMODE switch (True/False) sets the floating point (False) or integer computation (True). Default is false. Integer computation is intrinsically more accurate but also more limited because it may easily reaches the overflow error. Use IMODE only with integer matrices of moderate size.
Tiny (default is 0) sets the minimum round-off error; any value in absolute less than Tiny will be set to zero.

For a n = 1

$$\det[a_{11}] = a_{11}$$

For a n= 2

$$\det\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

For a n= 3

$$\det\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11}a_{22}a_{33} + a_{31}a_{23}a_{12} + a_{13}a_{21}a_{32} - a_{31}a_{22}a_{13} - a_{11}a_{32}a_{23} - a_{33}a_{21}a_{12}$$

Well, clearly the computation of the determinant is one of the most tedious of all Math. Fortunately that we have the Numeric Calculus...!

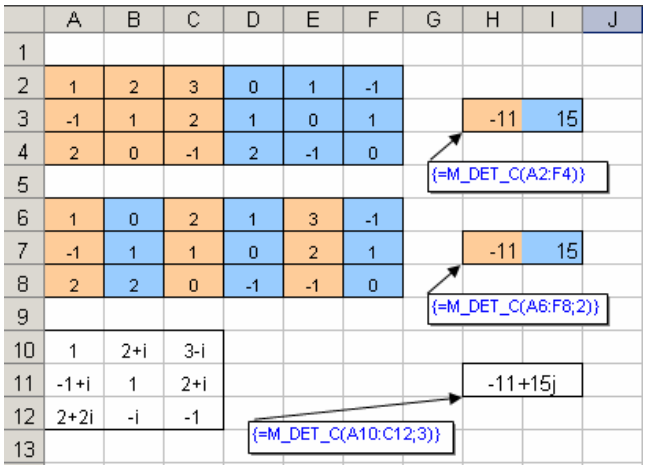
Example - The following matrix is singular but only the integer computation can give the exact answer

	A	B	C	D	E	F	G	H
1								
2	127	-507	245		M_DET			EXCEL
3	-507	2025	-987		Integer	Float		-6.868E-10
4	245	-987	553		0	1.5039E-09		
5								

Function M_DET_C (Mat, [Cformat])

This function computes the determinant of a complex matrix.
The argument Mat is an array (n x n) or (n x 2n) , depending of the format parameter
This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
The optional parameter *Cformat* sets the complex format of input/output (default = 1)
A complex (split or interlaced) matrix must have always an even number of columns

The example shows how to compute a determinant for a complex matrix written in three different formats.



The first complex matrix is in the *split* format (default): real and imaginary values are in two separated matrices.

The second example shows the same matrix in *interlaced* format: imaginary values are adjacent to real parts.

The last example shows the rectangular *string* format

Function M_DET3(Mat3)

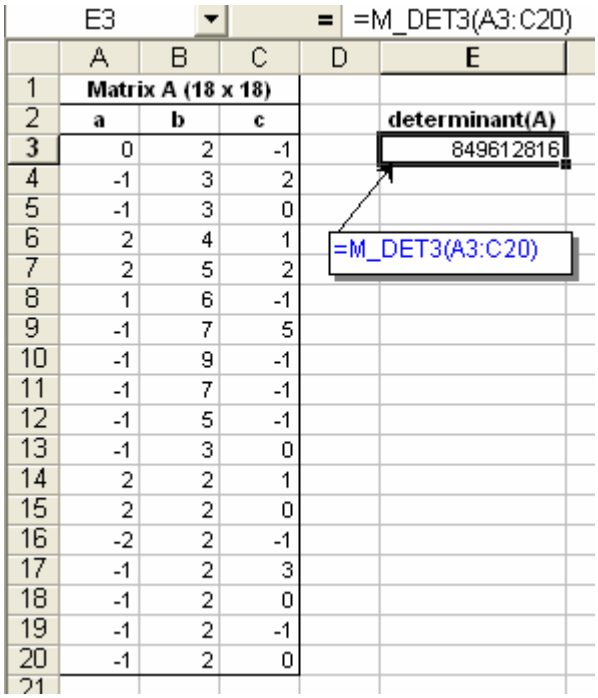
This function computes the determinant of a tridiagonal matrix.
The argument Mat3 is an (n x 3) array representing the (n x n) matrix

A triangular matrix is:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 \\ 0 & 0 & 0 & a_5 & b_5 \end{bmatrix}$$

In order to save space we can handle only the 3 diagonal vectors

Example: to find the determinant of a 20x20 tridiagonal matrix we pass to the function only 52 values (the first cell of **a** and the last of **c** are always 0) instead of 400 values.



	A	B	C
3	0	2	-1
4	-1	3	2
5	-1	3	0
6	2	4	1
7	2	5	2
8	1	6	-1
9	-1	7	5
10	-1	9	-1
11	-1	7	-1
12	-1	5	-1
13	-1	3	0
14	2	2	1
15	2	2	0
16	-2	2	-1
17	-1	2	3
18	-1	2	0
19	-1	2	-1
20	-1	2	0

Function M_INV(Mat, [IMode], [Tiny])

Returns the matrix inverse of a given square matrix

$$B = A^{-1}$$

IMODE switch (True/False) sets the floating point (False) or integer computation (True). Default is false. Integer computation is intrinsically more accurate but also more limited because it may easily reaches the overflow error. Use IMODE only with integer matrices of moderate size.
Tiny (default is 0) sets the minimum round-off error; any value in absolute less than Tiny will be set to zero.
If the matrix is singular the function returns "singular".
If the matrix is not squared the function returns "?"

Example: the following matrix is singular but only the M_INV function with integer computation can give the right answer

	A	B	C	D	E	F	G
1	127	-507	245		-2E+14	-6E+13	-6E+12
2	-507	2025	-987		-6E+13	-1E+13	-2E+12
3	245	-987	553		-6E+12	-2E+12	-2E+11
4					{=MINVERSE(A2:C4)}		
5							
6	9.7E+13	2.6E+13	2.8E+12		singular	singular	singular
7	2.6E+13	6.8E+12	7.5E+11		singular	singular	singular
8	2.8E+12	7.5E+11	8.4E+10		singular	singular	singular
9	{=M_INV(A2:C4)}				{=M_INV(A2:C4,VERO)}		

Function Mat_Pseudoinv (A)

Computes the Moore-Penrose pseudoinverse of a (n x m) matrix
Def: the minimum-norm least squares solution x to a linear system

$$Ax = b \quad \Rightarrow \quad \min_x \|Ax - b\|$$

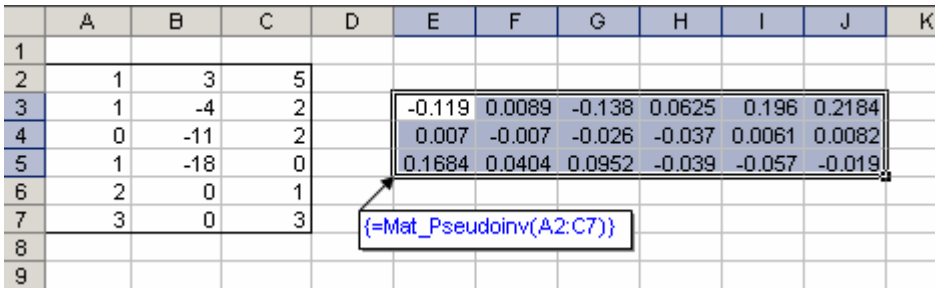
is the vector

$$x = \left(A^T A\right)^{-1} A^T = A^+ b$$

The matrix A^+ is called the pseudoinverse of A
If the matrix A has dimension (n x m), its pseudoinverse has dimension (m x n)
One of the most important application of the SVD decomposition is

$$A = U \cdot D \cdot V^T$$

$$A^+ = V \cdot D^{-1} U^T$$



Note the psudoinverse coincides with the inverse for non-singular square matrices.

Function M_POW(Mat, n)

Returns the integer power of a given square matrix

$$B = A^n = \overbrace{A \cdot A \cdot A \ldots A}^{n \text{ time}}$$

Function M_EXP(A, [Algo], [n])

This function approximates the exponential series expansion of a given square matrix **[A]**

$$e^{[A]} = I + \sum_{n=1}^{\infty} \frac{1}{n!} A^n$$

This function uses two alternative algorithm to approximates the infinite summation: the first one uses the popular power's series

$$EXP(A,n) = I + A + \tfrac{1}{2}A^2 + \tfrac{1}{6}A^3 + \ldots \tfrac{1}{n!}A^n + err$$

For n sufficiently larger, the error becomes negligible and the sum approximates the matrix exponential function. The parameter **n** fixes the max term of the series. If omitted the expansion continue until the convergence is reached; this means that the norm of the n[th] matrix term becomes less than Err = 1e−15.

$$\left\| \tfrac{1}{n!}A^n \right\| < Err$$

Take care using this function without n; especially for larger matrix, the evaluation time can be very long.

The second method, more efficient, uses the Padè approximation[2]. It is recommendable especially for larger matrices.
You can switch the algorithm by the optional parameter *Algo*. If "P" (default) the function uses the Padè approximation; else it uses the power's series

Function M_EXP_ERR(A, n)

This function returns the truncation n-th matrix term of the series expansion of a square matrix **[A].** It is useful to estimate the truncation error of the series approximation

$$EXP(A,n) = \left\| \tfrac{1}{n!}A^n \right\|$$

See also Function M_EXP(A, [n]) for matrix exponential series

Function M_PROD(A, B, ...)

Returns the product of two or more matrices

$$C = A \cdot B$$

As know the product is defined:

$$c_{ik} = a_{ij} b_{jk},$$

Where j is summed over for all possible value for ***i*** and ***k***

Dimension rule: If **A** is (n x m) and **B** is (m x p) , then the product is a matrix (n x p)

$$(n \times m)(m \times p) = (n \times p),$$

Note: If **A** and **B** are square (n x n) matrices, also the product is a square (n x n) matrix.

Writing out the product explicitly

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{bmatrix},$$

[2] This routine was developed by Gregory Klein, who kindly consented to add it to this package

Where:

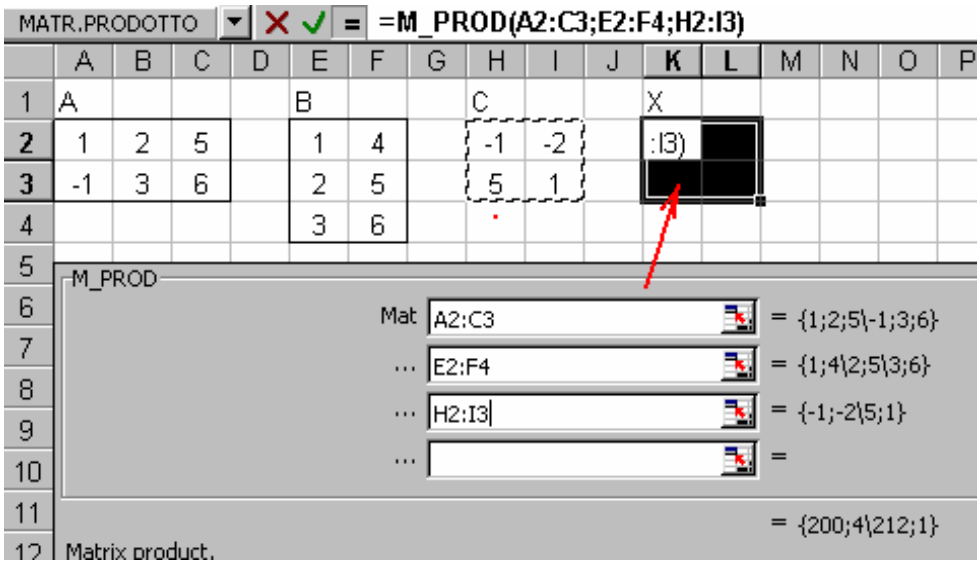
$$
\begin{aligned}
c_{11} &= a_{11}b_{11} + a_{12}b_{21} + \ldots + a_{1m}b_{m1} \\
c_{12} &= a_{11}b_{12} + a_{12}b_{22} + \ldots + a_{1m}b_{m2} \\
c_{1p} &= a_{11}b_{1p} + a_{12}b_{2p} + \ldots + a_{1m}b_{mp} \\
c_{21} &= a_{21}b_{11} + a_{22}b_{21} + \ldots + a_{2m}b_{m1} \\
c_{22} &= a_{21}b_{12} + a_{22}b_{22} + \ldots + a_{2m}b_{m2} \\
c_{2p} &= a_{21}b_{1p} + a_{22}b_{2p} + \ldots + a_{2m}b_{mp} \\
c_{n1} &= a_{n1}b_{11} + a_{n2}b_{21} + \ldots + a_{nm}b_{m1} \\
c_{n2} &= a_{n1}b_{12} + a_{n2}b_{22} + \ldots + a_{nm}b_{m2} \\
c_{np} &= a_{n1}b_{1p} + a_{n2}b_{2p} + \ldots + a_{nm}b_{mp}.
\end{aligned}
$$

Matrix multiplication is associative. Thus: $A \cdot (B \cdot C) = (A \cdot B) \cdot C$

But, generally, is not commutative: $A \cdot B \neq B \cdot A$

M_PROD() can perform the product of several matrices also with different dimensions.

$$Y = \prod_j A_j = A_1 \cdot A_2 \cdot \ldots$$



NB: If you multiply matrices with different dimension pay attention to the dimension rules above. This function does not check it. The function will return #VALUE if this rule is violate.

Function M_PRODS(Mat, k)

Returns a matrix multiplied for a scalar
For example:

$$k\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} k \cdot a_{11} & k \cdot a_{12} \\ k \cdot a_{21} & k \cdot a_{22} \end{pmatrix}$$

It can be nested in other function. For example, if the range A1:B2 contains the matrix 2x2 [1 , 2, / -3 , 8]

M_DET(M_PRODS(A1:B2; 3)) returns the determinant 126 of the matrix [3 , 6, / -9 , 24]

Note: EXCEL has a simply way to performs the multiplication of an array by a scalar. For details see How to insert an array function...

Function M_PRODS_C(Mat, scalar, [Cformat])

Performs the complex matrix multiplication for a scalar.

$$k\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} k \cdot a_{11} & k \cdot a_{12} \\ k \cdot a_{21} & k \cdot a_{22} \end{pmatrix}$$

The parameter Mat is a (n x m) complex matrix or vector
The parameter scalar can be complex or real number, in split or string format.
This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
The optional parameter *Cformat* sets the complex format of input/output (default = 1)

See About complex matrix format

Example. Performs the following complex multiplication

$$C = (2-j)\begin{bmatrix} 4+5j & 5 & -30j \\ -17+4j & -5+5j & 14 \\ -9j & -4j & 10j \end{bmatrix}$$

We can use either the split or string format

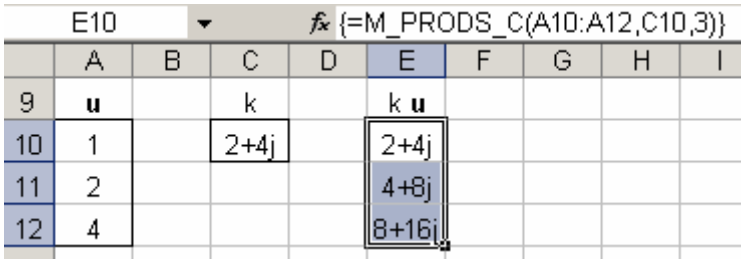
	A	B	C	D	E	F
1			Matrix A			
2	4	5	0	5	20	-30
3	-17	-4	14	4	5	0
4	0	0	0	-9	-4	10
5						
6	scalar =>		2	-1		
7						
8	13	30	-30	6	35	-60
9	-30	-3	28	25	14	-14
10	-9	-4	10	-18	-8	20
11						
12	{=M_PRODS_C(A2:F4,H2:I2)}					

	L	M	N
	Matrix A		
	4+5j	5	-30j
	-17+4j	-4+5j	14
	-9j	-4j	10j
	scalar =>		2-j
	13+6j	10-5j	-30-60j
	-30+25j	-3+14j	28-14j
	-9-18j	-4-8j	10+20j
	{=M_PRODS_C(L2:N4,N6,3)}		

This function multiplies also a complex vector for a complex number

	A	B	C	D	E	F	G	H
1	u			k			k u	
2	4	2		1	-1		6	-2
3	2	-1					1	-3
4	3	0					3	-3
5								
6	{=M_PRODS_C(A2:B4,D2:E2)}							

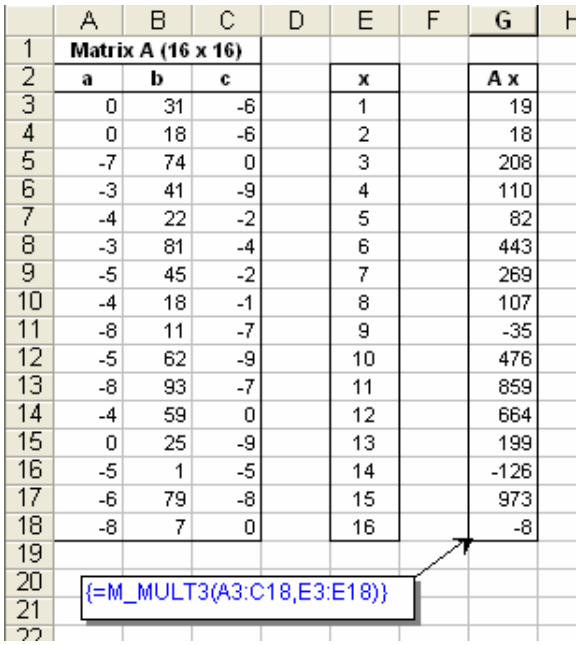
and a real vector for a complex number



Function M_MULT3(Mat3, Mat)

This function performs the multiplication of a 3-diagonal matrix and a vector or a rectangular matrix
Mat3 is a (n x 3) array
Mat can be a vector (n x 1) or even a rectangular matrix (n x m)
Result is a vector (n x 1) or a matrix (n x m)
This function is useful when you have to multiply a tridiagonal matrix larger than 256 x 256 for a vector because Excel cannot manage matrices larger than 256 columns.

Example
The diagonal and sub-diagonals are passed to the function as vertical vectors



	a	b	c	x	A x
3	0	31	-6	1	19
4	0	18	-6	2	18
5	-7	74	0	3	208
6	-3	41	-9	4	110
7	-4	22	-2	5	82
8	-3	81	-4	6	443
9	-5	45	-2	7	269
10	-4	18	-1	8	107
11	-8	11	-7	9	-35
12	-5	62	-9	10	476
13	-8	93	-7	11	859
14	-4	59	0	12	664
15	0	25	-9	13	199
16	-5	1	-5	14	-126
17	-6	79	-8	15	973
18	-8	7	0	16	-8

$$A \cdot x = \begin{bmatrix} b_1 & c_1 & 0 & \dots & 0 & 0 \\ a_1 & b_2 & c_2 & \dots & 0 & 0 \\ 0 & a_2 & b_3 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & b_{15} & c_{15} \\ 0 & 0 & 0 & \dots & a_{15} & b_{16} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_{15} \\ x_{16} \end{bmatrix}$$

Note how compact and efficient this input is. This is true overall for larger matrices

Function M_SUB(A1, A2)

Returns the different of two matrices

$$C = A_1 - A_2$$

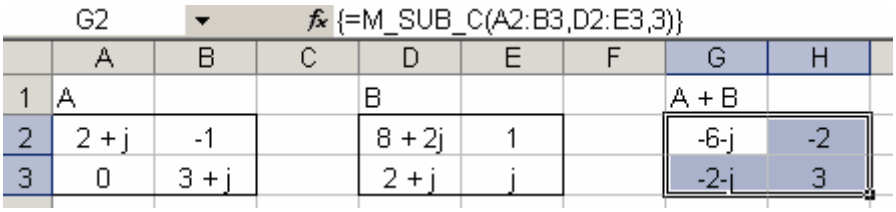
Excel can also perform matrix subtraction in a very efficient way by the array-arithmetic.

Function M_SUB_C(A1, A2, [Cformat])

Returns the different of two complex matrices

$$C = A_1 - A_2$$

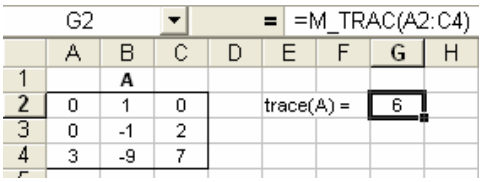
This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
The optional parameter *Cformat* sets the complex format of input/output (default = 1)



Function M_TRAC(Mat)

Returns the trace of a square matrix, thus the sum of all elements of the first diagonal

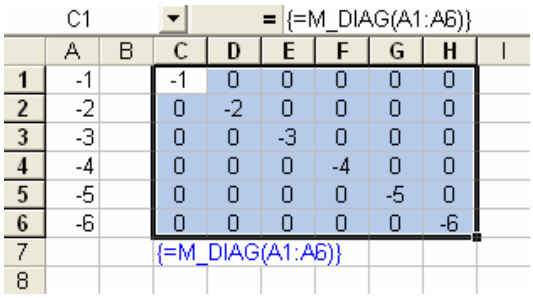
$$trace(A) = \sum a_{ii}$$



Function M_DIAG(Diag)

Returns the diagonal matrix having the vector "Diag" as the diagonal

Example:

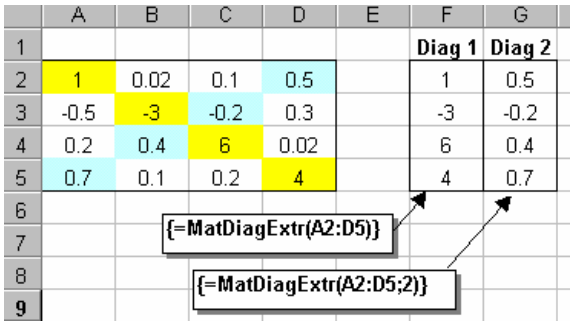


Function MatDiagExtr(Mat, [Diag])

This function extracts the diagonals of a matrix[3]
The optional parameter Diag sets the diagonal to extract.

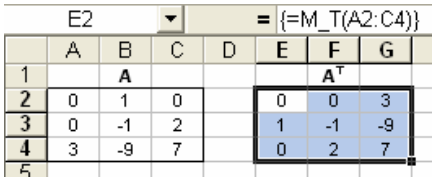
[3] (Thanks to an idea of Giacomo Bruzzo)

Diag = 1 (default) extracts the first diagonal; Diag = 2 extracts the secondary one.



Function MT(Mat)

Returns the transpose of a give matrix, thus the matrix with rows and columns exchanged



This function is identical to TRANSPOSE() Excel built-in function

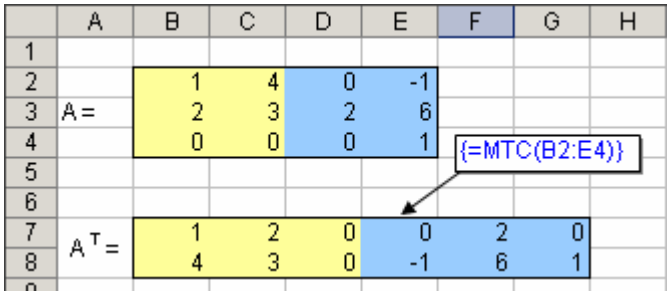
Function MTC(Mat, [Cformat])

Returns the transpose of a complex matrix, thus the matrix with rows and columns exchanged

$$A = \begin{bmatrix} 1 & 4-j \\ 2+2j & 3+6j \\ 0 & j \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 1 & 2+2j & 0 \\ 4-j & 3+6j & j \end{bmatrix}$$

This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
Optional parameter *Cformat* sets the complex format of input/output (default = 1)
Use CTRL+SHIFT+ENTER to insert this function

Example. Transpose the following (3 x 2) complex matrix

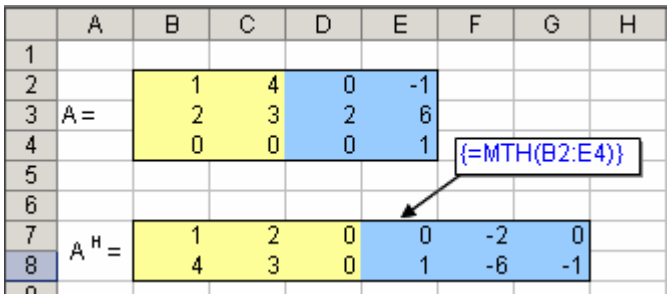


Function MTH(Mat, [Cformat])

Returns the transpose-conjugate of a complex matrix

$$A=\begin{bmatrix} 1 & 4-j \\ 2+2j & 3+6j \\ 0 & j \end{bmatrix} \Rightarrow A^H=\begin{bmatrix} 1 & 2-2j & 0 \\ 4+j & 3-6j & -j \end{bmatrix}$$

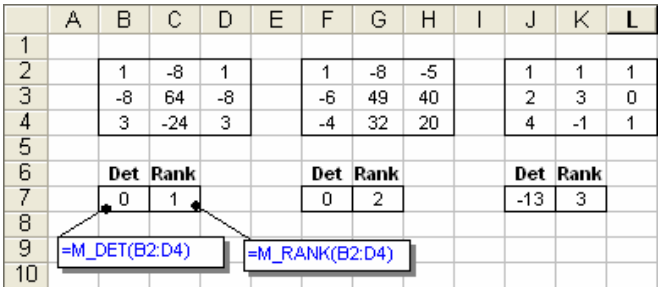
This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string

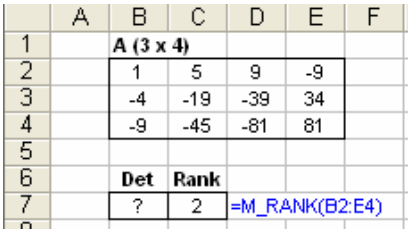


Function M_RANK(A)

Returns the rank of a given matrix[4]
It computes the sub-space of Ax = 0, using the SYSLINSING function: then counts null column-vectors of sub-space.

Examples:



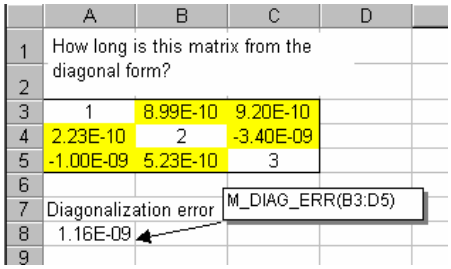


When Det = 0 the Rank is always less then the max dimension n
Differently from the determinant, rank can be computed also for rectangular matrices.
In that case, the rank can't exceed the minimum dimension; that is, for a 3x4 matrix the maximum rank is 3.

Function M_DIAG_ERR(A)

Returns the "diagonalization" error of a given square matrix
This function computes and returns the mean of the absolute values out of the first diagonal
It is useful to measure the "distance" from the diagonal form of a square matrix

[4] (Thanks to the original routine developed by Bernard Wagner.)



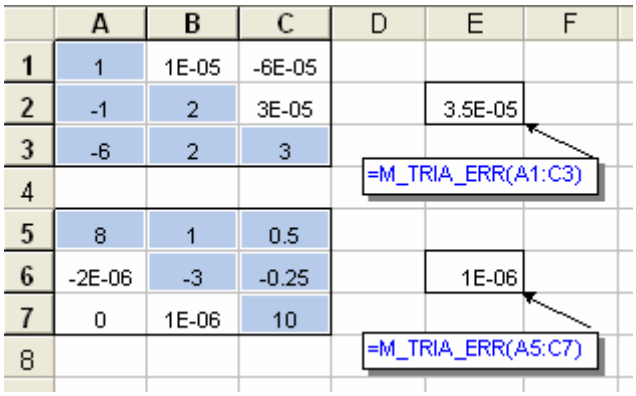
$$\text{diag error} = \frac{1}{(n^2 - n)} \sum_{i \neq j} | a_{ij} |$$

Function M_TRIA_ERR(A)

Returns the "triangularization" error of a given square triangular matrix
This function computes and returns the minimum of the mean of the absolute values of the upper and lower element out of the first diagonal
It is useful to measure the "distance" from the triangular form of a square matrix

$$err_{tria} = \frac{2}{n^2 - n} \min\left(\sum_{i>j} | a_{ij} | \ , \ \sum_{i<j} | a_{ij} | \right)$$

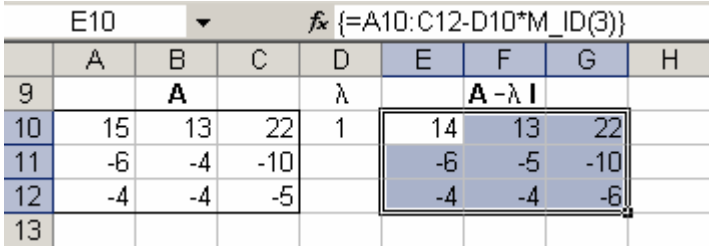
Example: A triangularization algorithm has computed the following matrices. What is the average error?



Function M_ID(n)

Returns the identity square matrix.
This function is useful in nested expression

Example: Compute the matrix **A** – λ **I** for the parameter λ = 1



Note that we have used the power array arithmetic of Excel

But we could use the following nested expression:

```
{=M_ADD(A10:C12,D10*M_ID(3))}
```

Function ProdScal(v1, v2)

Returns the scalar product of two vectors

$$V_1 \bullet V_2 = \sum v_{1,i} \cdot v_{2,i}$$

Note that if V1 and V2 are the same vectors, this function returns the square of its module.

$$V \bullet V = \sum v_i \cdot v_i = \sum v_i^2 = \|v\|^2$$

Note that if V_1 and V_2 are perpendicular, the scalar product is zero. In fact another definition of scalar product is:

$$V_1 \bullet V_2 = |V_1| \cdot |V_2| \cdot \cos(\alpha_{12})$$

	A	B	C	D	E	F	G	H	I
1		**v1**	**v2**						
2		1	2		**x1**	9.5	2	-5.5	-13
3		-2	0		**x2**	2.4	-1	-4.4	-1
4		3	-1						
5		4	6		**v1·v2**		**x1·x2**		
6					23		58		
7									
8	=ProdScal(B2:B5,C2:C5)					=ProdScal(F2:I2,F3:I3)			
9									

Vectors can be in vertical or horizontal format as well.

Function ProdVect(v1, v2)

Returns the vector product of two vectors

$$V_1 \times V_2 = \begin{bmatrix} v_{11} \\ v_{21} \\ v_{31} \end{bmatrix} \times \begin{bmatrix} v_{12} \\ v_{22} \\ v_{32} \end{bmatrix} = \begin{bmatrix} v_{21}v_{32} - v_{22}v_{31} \\ v_{12}v_{31} - v_{11}v_{32} \\ v_{11}v_{22} - v_{21}v_{12} \end{bmatrix}$$

Note that if V_1 and V_2 are parallels, the vector product is the null vector.

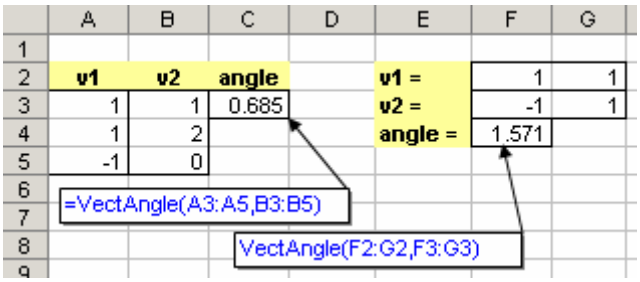
	A	B	C	D	E	F
1		V_1	V_2		$V_1 \times V_2$	
2		1	2		0	
3		3	6		0	
4		-2	-4		0	
5						
6	{=ProdVect(B2:B4;C2:C4)}					
7						

Vectors can be in vertical or horizontal form as well.

Function VectAngle(v1, v2)

Computes the angle between the two vectors V_1, V_2 .
The angle is defined as:

$$\alpha = \arccos\left(\frac{v_1 \bullet v_2}{|v_1| \cdot |v_2|}\right)$$



Function MatEigenvalue_Jacobi(Mat, Optional MaxLoops)

This function performs the Jacobi's sequence of orthogonal similarity transformation and returns the last matrix of the sequence. It works only for symmetric matrices.
The optional parameter MaxLoops (default=100) sets the max steps of the sequence.
The Jacobi's algorithm can be used to find both eigenvalues and eigenvectors of symmetric matrices

$$A_1 = P_1^T \cdot A \cdot P_1$$
$$A_2 = P_2^T \cdot A_1 \cdot P_2 = P_2^T P_1^T A P_1 P_2$$
$$A_n = P_n^T \cdot A_{n-1} \cdot P_n = P_n^T \ldots P_2^T P_1^T A P_1 P_2 \ldots P_n$$

For n sufficiently large, the sequence {Ai} converge to a diagonal matrix, thus

$$\lim_{n \to \infty} A_n = [\lambda]$$

While the matrix

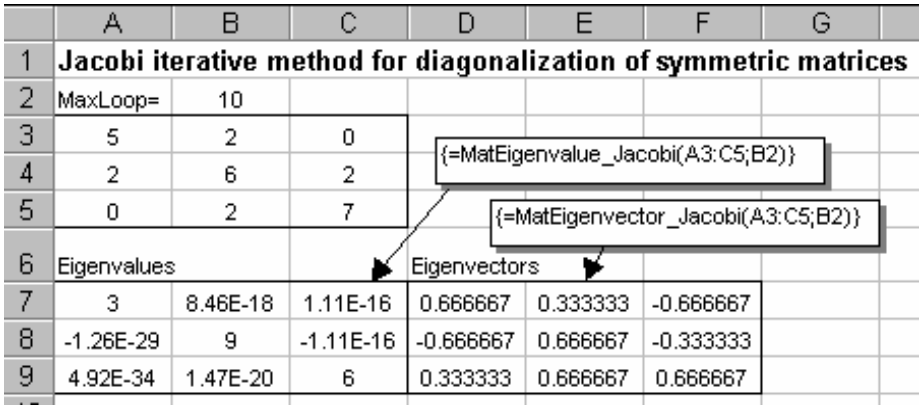
$$U_n = P_n P_{n-1} \ldots . P_3 P_2 P$$

Converges to the eigenvectors of A.

All these matrices A**, U, P** can be obtained by the functions:
MatEigenvalue_Jacobi
MatEigenvector_Jacobi
MatRotation_Jacobi

Example: Solve the eigenvalues problem of the following symmetric matrix

$$A = \begin{bmatrix} 5 & 2 & 0 \\ 2 & 6 & 2 \\ 0 & 2 & 7 \end{bmatrix}$$



As we can see, the Jacobi's method has found all Eigenvalues and Eigenvectors with few iterations (10 loops)
The function **MatEigenvalue_Jacobi()** returns in range A7:C9 the diagonal matrix of the eigenvalues. Note that elements out of the first diagonal have an error less than 10^-16.
At the right side - in the range D7:F9 - the function MatEigenvector_Jacobi() returns the orthogonal matrix of the eigenvectors.
Compare with the exact solution

$$A=\begin{bmatrix}5&2&0\\2&6&2\\0&2&7\end{bmatrix},\ \lambda=\begin{bmatrix}3&0&0\\0&6&0\\0&0&9\end{bmatrix},\ U=\begin{bmatrix}2/3&1/3&-2/3\\-2/3&2/3&-1/3\\1/3&2/3&2/3\end{bmatrix}$$

Note that you can test the approximate results by the similarity transformation

$$\lambda = U^{-1}A\cdot U$$

You can use the standard matrix inversion and multiplication functions or the **M_BAB()** function also in this package, as you like. Note that - only in this case - the inversion of matrix is very simple, because: $U^{-1}=U^{T}$

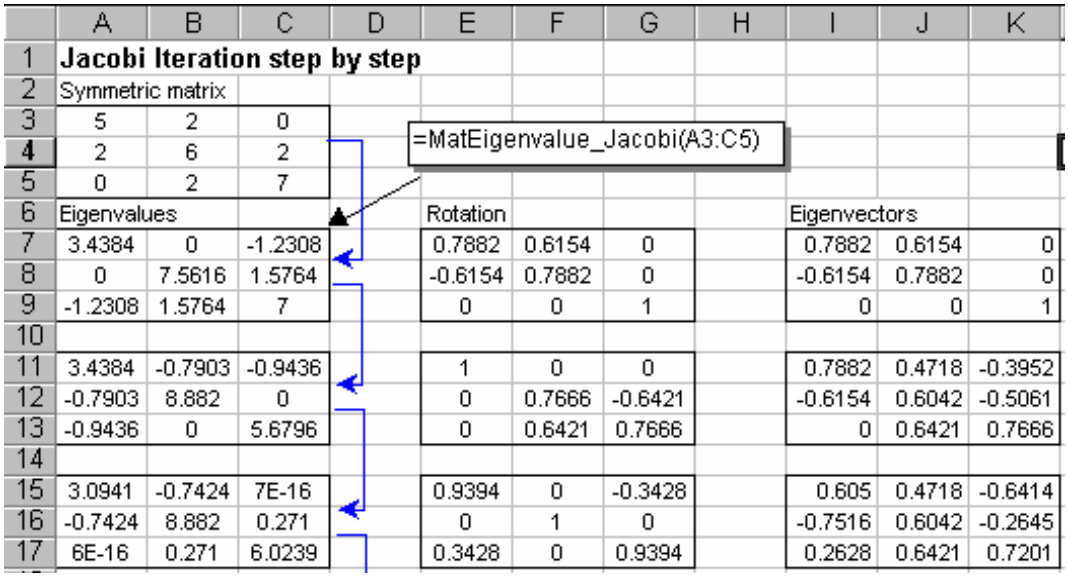
Eigenvalues problem with Jacobi step by step

Suppose you want to study about each step of the Jacobi's method. The functions ***MatEigenvalue_Jacobi()***, ***MatEigenvector_Jacobi()*** and ***MatRotation_Jacobi()*** are useful if you set the parameter MaxLoop=1. In this case, they return the first step of Jacobi's iteration. Here how they work.

A1 = MatEigenvector_Jacobi(A)
A2 = MatEigenvector_Jacobi(A1)
A3 = MatEigenvector_Jacobi(A2)
.....
A10 = MatEigenvector_Jacobi(A9)

$$A=\begin{bmatrix}5&2&0\\2&6&2\\0&2&7\end{bmatrix}$$

Each matrix is one step of Jacobi's Iterative method



	A	B	C	D	E	F	G	H	I	J	K
1	**Jacobi Iteration step by step**										
2	Symmetric matrix										
3	5	2	0								
4	2	6	2								
5	0	2	7								
6	Eigenvalues				Rotation				Eigenvectors		
7	3.4384	0	-1.2308		0.7882	0.6154	0		0.7882	0.6154	0
8	0	7.5616	1.5764		-0.6154	0.7882	0		-0.6154	0.7882	0
9	-1.2308	1.5764	7		0	0	1		0	0	1
10											
11	3.4384	-0.7903	-0.9436		1	0	0		0.7882	0.4718	-0.3952
12	-0.7903	8.882	0		0	0.7666	-0.6421		-0.6154	0.6042	-0.5061
13	-0.9436	0	5.6796		0	0.6421	0.7666		0	0.6421	0.7666
14											
15	3.0941	-0.7424	7E-16		0.9394	0	-0.3428		0.605	0.4718	-0.6414
16	-0.7424	8.882	0.271		0	1	0		-0.7516	0.6042	-0.2645
17	6E-16	0.271	6.0239		0.3428	0	0.9394		0.2628	0.6421	0.7201

To obtain quickly the above iterations follow these simple rules:
At the first time, insert in range A7:C9 the function *MatEigenvalue_Jacobi(A3:A7).*
Give the "magic" key sequence CTRL+SHIFT+ENTER to paste an array function
Leave the range A7:C9 selected and copy it (CTRL+C)
Select the following range A11 and paste it (CTRL+V)
Copy it (CTRL+C)
Select the following range A15 and paste it (CTRL+V)
And so on.
By the sequence -copying and pasting- you can perform all Jacobi's iterations, as you like.
In the middle, we see the Jacobi's rotation matrices sequence. We can easily obtain it by the function *MatRotation_Jacobi()* in the same way as eigenvalues matrix

P1 = MatRotation_Jacobi(A)
P2 = MatRotation_Jacobi(A1)
P3 = MatRotation_Jacobi(A2)
Etc.

This function search for the max absolute values out of the first diagonal and generates an orthogonal matrix in order to reduce it to zero by similarity transformation

$$A_1 = P_1^T A \cdot P_1.$$

Finally, at the right, we see the iterations of eigenvectors matrix. It can be derived from the rotation matrix by the following iterative formula:

$$U_1 = P_1$$
$$U_n = P_n \cdot U_{n-1}$$

Function MatRotation_Jacobi(Mat)

This function returns the Jacobi's orthogonal rotation matrix of a given symmetric matrix.
This function searches for the max absolute values out of the first diagonal and generates an orthogonal matrix in order to reduce it to zero by similarity transformation

$A_1 = P_1^T A \cdot P_1$. Where: P1 = MatRotation_Jacobi(A)

For further details see Function MatEigenvalue_Jacobi(Mat, Optional MaxLoops)

Example - find the rotation matrix that makes zero the highest non-diagonal element of the following symmetric matrix.

-5	-4	-1	-3	4
-4	-6	0	-2	5
-1	0	5	4	8
-3	-2	4	-5	1
4	5	8	1	-10

The highest absolute values are $a_{53} = a_{35} = 8$ (in red)
The similarity transformation with the rotation matrix will make zero just these elements.

The rotation matrix, in that case is

1	0	0	0	0
0	1	0	0	0
0	0	cos(α)	0	sin(α)
0	0	0	1	0
0	0	-sin(α)	0	cos(α)

Where the angle α is given by the formula:

$$\alpha = \frac{1}{2}\text{atan}\left(\frac{2a_{35}}{a_{55} - a_{33}}\right)$$

	A	B	C	D	E	F	G	H	I	J	K	L
1			**A**						**P**			
2	-5	-4	-1	-3	4		1	0	0	0	0	
3	-4	-6	0	-2	5		0	1	0	0	0	
4	-1	0	5	4	8		0	0	0.918	0	-0.398	
5	-3	-2	4	-5	1		0	0	0	1	0	
6	4	5	8	1	-10		0	0	0.398	0	0.918	
7			**$P^T A P$**									
8	-5	-4	0.7	-3	4.1							
9	-4	-6	2	-2	4.6							
10	0.7	2	8.5	4.1	0							
11	-3	-2	4.1	-5	-0.7							
12	4.1	4.6	0	-0.7	-13							
13												

{=MatRotation_Jacobi(A2:E6)}

{=M_BAB(A2:E6,G2:K6)}

Function Mat_Block(Mat,)

Transforms a sparse square matrix (n x n) into a block-partitioned matrix
From theory we know that, under certain conditions, a square matrix can be transformed into a block-partitioned form (also called block-triangular form) by similarity transformation.

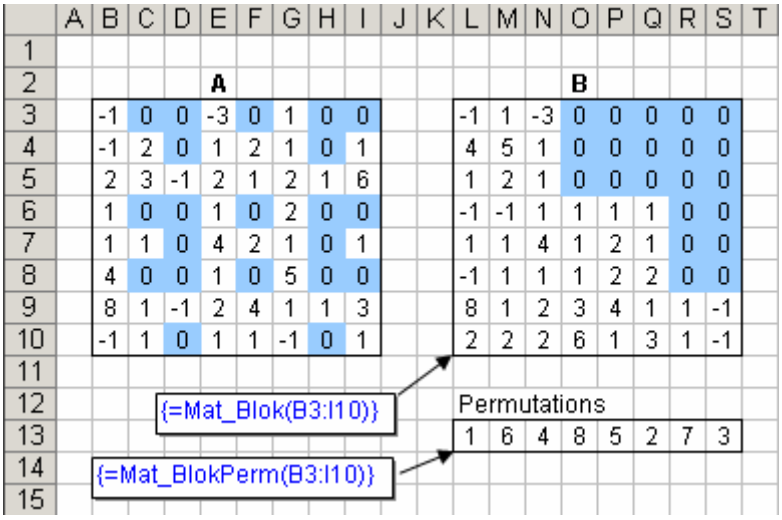
$$B = P^T A P$$

where **P** is a (n x n) permutation matrix.

For returning the permutation matrix see the function Mat_BlockPerm

Note that not all matrices can be transformed in block-triangular form. From theory we know that it can be done if, and only if, the graph associated to the matrix is not strong connected. On the contrary, if the graph is strong connected, we say that the matrix is irreducible. A dense matrix without zero elements, for example, is always irreducible.

Example:



Function Mat_BlockPerm(Mat,)

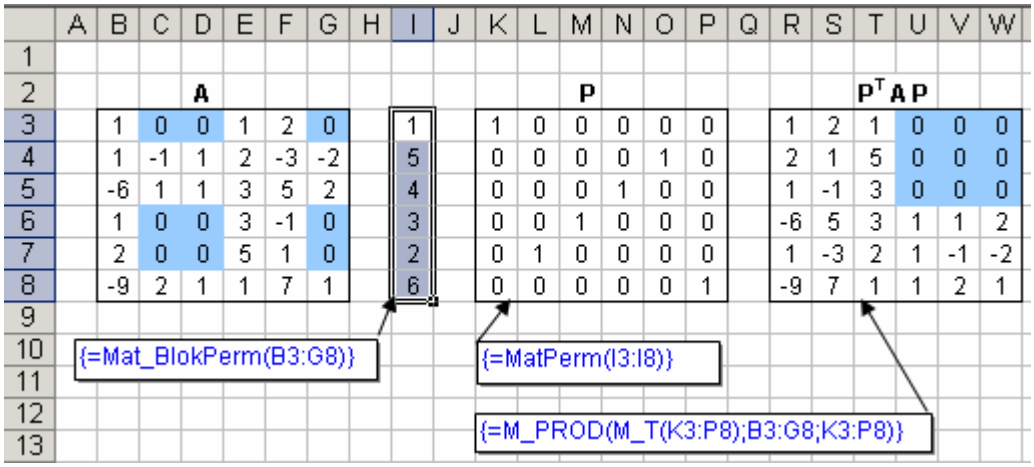
Returns the permutation matrix that trasforms a sparse square matrix (n x n) into a block-partitioned matrix. Under certain conditions, a square matrix can be transformed into a block-partitioned form (also called block-triangular form) by similarity transformation.

$$B = P^T A P$$

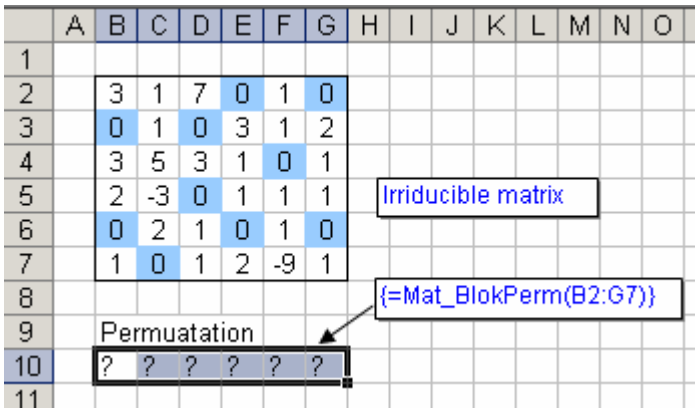
where P is a permutation matrix (n x n).

This function returns the permutation vector (n); for transforming it into a permutation matrix use the Function MatPerm

Example. Find the permutation matrix that transforms the given matrix into block triangular form



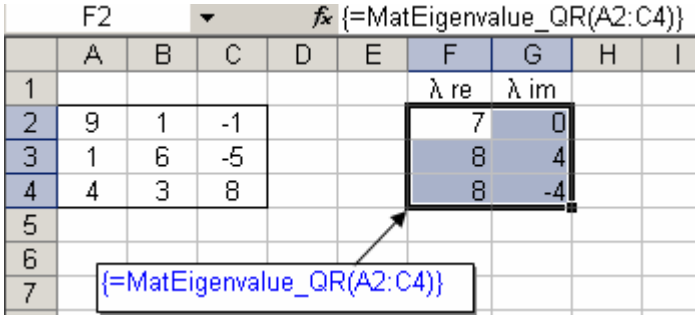
Note that not all matrices can be transformed in block-triangular form. If the transformation fails the function returns "?". This usually happens if the matrix is irreducible.



Function MatEigenvalue_QR(Mat)

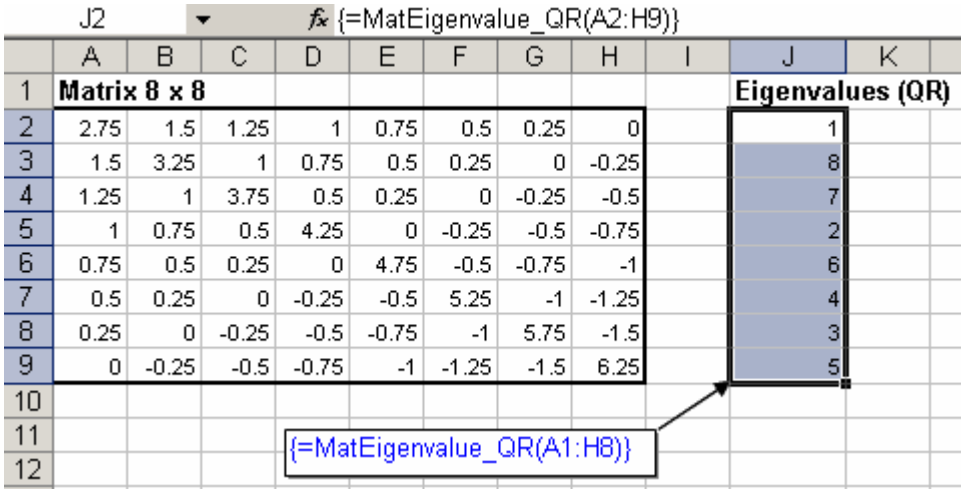
This function performs the diagonal reduction of a given matrix by the generalized QR method[5], and returns the approximate eigenvalues, real or complex.
It returns an (n x 2) array

The example below shows that the given matrix has two complex conjugate eigenvalues and only one real eigenvalue



[5] This function uses a reduction of the EISPACK FORTRAN HQR and ELMHES subroutines (April 1983)
HQR nad ELMHES are the translation of the ALGOL procedure.
NUM. MATH. 14, 219-231(1970) by Martin, Peters, and Wilkinson.

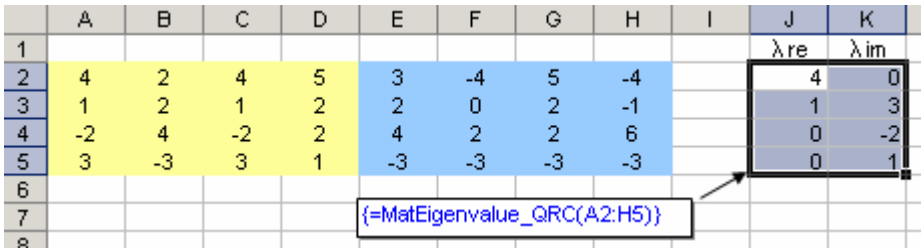
Example. Find the eigenvalues of the following symmetric matrix. Being symmetric, there are only n real distinct eigenvalues. So the function returns only an (n x 1) array



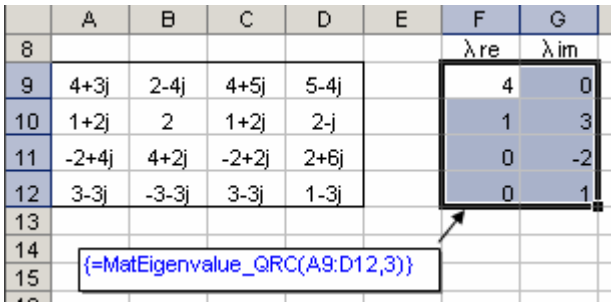
Function MatEigenvalue_QRC(Mat, [Cformat])

This function performs the diagonal reduction of a given complex matrix with complex QR method[6], and returns the approximate eigenvalues real or complex as an (n x 2) array.
This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
Optional parameter *Cformat* sets the complex format of input/output (default = 1)
See About complex matrix format

Example. Find the eigenvalues of the following complex matrix.



The matrix could be also passed in compact string format



Note that the result is always in split format

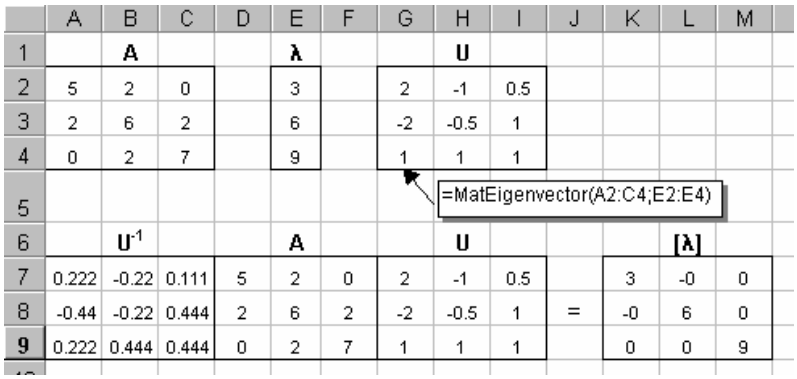
[6] This function uses a reduction of the EISPACK FORTRAN COMQR and CORTH subroutines (April 1983)
COMQR is a translation of the ALGOL procedue
MATH. 12, 369-376(1968) by Martin and Wilkinson.

Function MatEigenvector(A, Eigenvalues, [MaxErr])

This function returns the eigenvector of a matrix A (n x n) associated to the given eigenvalue

$$Av = \lambda v$$

If "Eigenvalues" is a single value, the function returns a (n x 1) vector. Otherwise if "Eigenvalues" is a vector of eigenvalues, the function returns the (n x n) matrix of eigenvectors.
The optional parameter MaxErr is useful when the eigenvalues are affected by round-off error. In that case the MaxErr should be proportionally adapted. Otherwise the result may be a NULL matrix. If omitted, the function tries to detect by itself the suitable MaxErr for the approximate eigenvalues



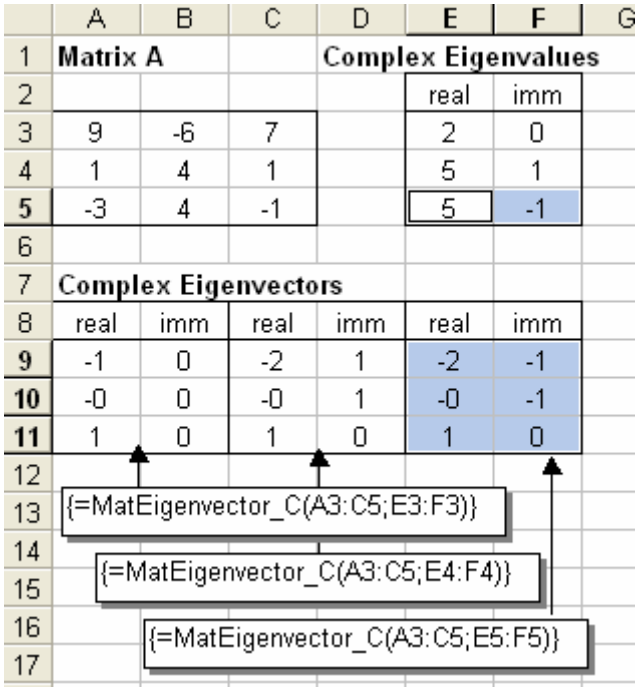
	A	B	C	D	E	F	G	H	I	J	K	L	M
1		**A**			**λ**			**U**					
2	5	2	0		3		2	-1	0.5				
3	2	6	2		6		-2	-0.5	1				
4	0	2	7		9		1	1	1				
5													
6		$\mathbf{U^{-1}}$			**A**			**U**				**[λ]**	
7	0.222	-0.22	0.111	5	2	0	2	-1	0.5		3	-0	0
8	-0.44	-0.22	0.444	2	6	2	-2	-0.5	1	=	-0	6	0
9	0.222	0.444	0.444	0	2	7	1	1	1		0	0	9

Function MatEigenvector_C(A, Eigenvalue, [MaxErr])

This function returns the complex eigenvector associates to a complex eigenvalue of a real or complex matrix A (n x n)

The function returns an array of two columns (n x 2): the first column contains the real part, the second column the imaginary part.
It returns an array of four columns (n x 4) if the eigenvalue is double. The first two columns contain the first complex eigenvector; the last two columns contain the second complex eigenvector. And so on.
The optional parameter MaxErr (default 1E-10) is useful only if your eigenvalue has an error. In that case the MaxErr should be proportionally increased (1E-8, 1E-6, etc.). Otherwise the result may be a NULL matrix.

Look at this example:



The given matrix has 3 eigenvalues:

2 , 5+j , 5–j

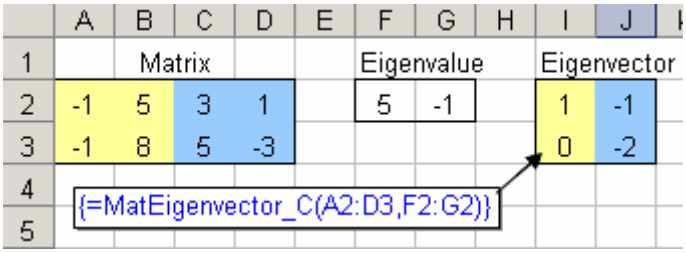
For each eigenvalue, the function **MatEigenvector_C** returns the associate eigenvector that, in general, will be complex.

Note that for the real eigenvalue 2, the function returns a real eigenvector

Note also that for conjugate eigenvalues will get also conjugate eigenvectors

The function works also for complex matrices. Example assume to have to find the eigenvector of the following matrix for the given eigenvalue

$$A = \begin{bmatrix} -1+3j & 5+j \\ -1+5j & 8-3j \end{bmatrix} \quad , \ \lambda = 5-j$$



Thus, the eigenvector for $\lambda = 5-j$ is

$$\mathrm{u} = \begin{bmatrix} 1-j \\ -2j \end{bmatrix}$$

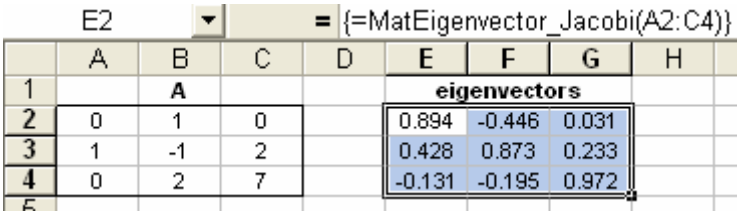
Function MatEigenvector_Jacobi(Mat, Optional MaxLoops)

This function performs the Jacobi's orthogonal similarity transformation sequence and returns the last orthogonal matrix. It works only for symmetric matrices.
The optional parameter MaxLoops (default=100) sets the max steps of the sequence.

This function returns the orthogonal matrix $\mathbf{U_n}$ that transforms to a diagonal form the symmetric matrix **A**, for n sufficiently high

$$[\lambda] \cong U_n^T A \cdot U_n$$

The matrix $\mathbf{U_n}$ is composed by eigenvectors of **A**

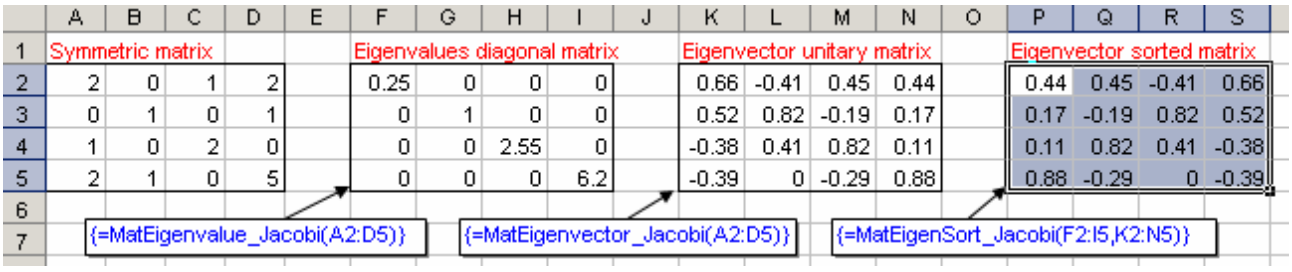


For further details see Function MatEigenvalue_Jacobi

Function MatEigenSort_Jacobi(EigvalM, EigvectM, [num])

This function[7] sorts of the eigenvalues and returns the first eigenvector associated to the absolute highest eigenvalue.
EigvalM is the diagonal eigenvalues (n x n) matrix and EigvectM is the (n x n) eigenvector unitary matrix as returned by MatEigenValue_Jacobi and MatEigenVector_Jacobi.
The optional parameter num (default num = n) sets the number of the vectors returned

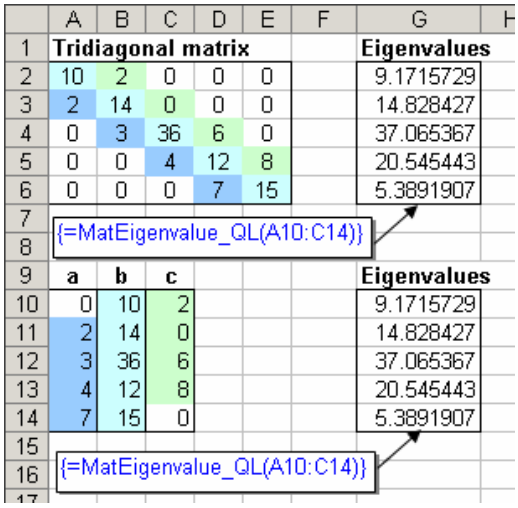
Example



[7] This function appears thanks to the courtesy of Carlos Aya

Function MatEigenvalue_QL(Mat3, [IterMax])

This function returns the real eigenvalues of a tridiagonal symmetric matrix. It works also for unsymmetrical tridiagonal matrix having its eigenvalues all real.
The optional parameter *Itermax* sets the max iteration allowed (default *Itermax* =200).
This function use the efficient QL algorithm
If the matrix has not all real eigenvalues this function returns "?"
This function accepts both tridiagonal square (n x n) matrices and (n x 3) rectangular matrices.



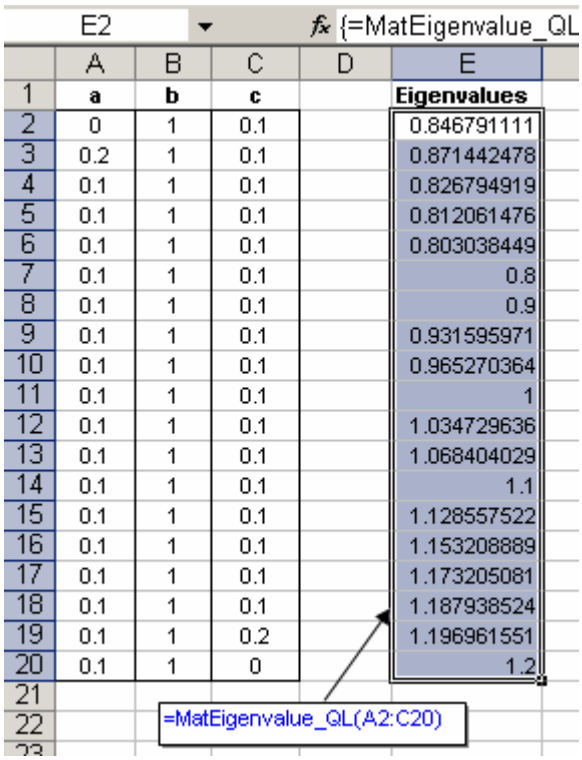
Example. Find all eigenvalues of the following 19 x 19 matrix

1	0.1	0	0	0	...	0	0	0	0
0.2	1	0.1	0	0	...	0	0	0	0
0	0.1	1	0.1	0	...	0	0	0	0
0	0	0.1	1	0.1	...	0	0	0	0
0	0	0	0.1	1	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...
0	0	0	0	0	...	1	0.1	0	0
0	0	0	0	0	...	0.1	1	0.1	0
0	0	0	0	0	...	0	0.1	1	0.2
0	0	0	0	0	...	0	0	0.1	1

Note that all 19 eigenvalues are close each other, in the short interval

$$0.8 < \lambda_k < 1.2$$

Other algorithms have difficult in this pathological case.
On the contrary the QL algorithm works fine giving an high general accuracy.



Function MatEigenvalue3U(n, a, b, c)

Returns the eigenvalues of a tridiagonal uniform (n x n) matrix.

$$\begin{bmatrix} b & c & 0 & 0 & \dots \\ a & b & c & 0 & \dots \\ 0 & a & b & c & \dots \\ 0 & 0 & a & b & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

It was been demonstrated that for these matrices:

- for n even - all eigenvalues are real if a*c>0; all eigenvalues are complex otherwise.
- for n odd - all n-1 eigenvalues are real if a*c>0; all n-1 eigenvalues are complex otherwise. The last n eigenvalue is always b.

For the uniform tridiagonal matrices, there is a nice close formula giving all eigenvalues for any size of the matrix dimension. See chapter “Tridiagonal uniform matrix”.

Example :
find the eigenvalues of the 40 x 40 tridiagonal uniform matrix having a = 1, b = 3, c = 2.
Because a*c = 2 > 0 , all eigenvalues are real.

	A	B	C	D	E	F
1	**n**	**a**	**b**	**c**	**eigenvalues**	
2	40	1	3	2	**re**	**im**
3					0.179872	0
4	{=MatEigenvalue3U(A2,B2,C2,D2)}				0.204721	0
5					0.245973	0
6					0.303388	0

find the eigenvalues of the 40 x 40 tridiagonal uniform matrix having a = 1, b = 3, c = –2.
Because a*c = –2 < 0 , all eigenvalues are complex.

	A	B	C	D	E	F
1	**n**	**a**	**b**	**c**	**eigenvalues**	
2	40	1	3	-2	**re**	**im**
3					3	-2.82013
4	{=MatEigenvalue3U(A2,B2,C2,D2)}				3	-2.79528
5					3	-2.75403
6					3	-2.69661

Function MatEigenvector3(Mat3, Eigenvalues, [MaxErr])

This function returns the eigenvector of associate eigenvalue of a tridiagonal matrix A

$$Av = \lambda v$$

If Eigenvalues is a single value, the function returns a (n x 1) vector. Otherwise if the parameter *Eigenvalues* is a vector of all eigenvalues of matrix A, the function returns a matrix (n x n) of eigenvectors. Note: the eigenvectors returned by this function are not normalized.

The optional parameter *MaxErr* is useful only if your eigenvalues are affected by an error. In that case the MaxErr should be proportionally adapted. Otherwise the result may be a NULL matrix. If omitted, the function tries to detect by itself the best error parameter

This function accepts both tridiagonal square (n x n) matrices and (n x 3) rectangular matrices.
The second form is useful for large matrices.

Example.
Given the 19 x 19 tridiagonal matrix having eigenvalue L = 1, find its associate eigenvector

1	0.1	0	0	0	...	0	0	0	0
0.2	1	0.1	0	0	...	0	0	0	0
0	0.1	1	0.1	0	...	0	0	0	0
0	0	0.1	1	0.1	...	0	0	0	0
0	0	0	0.1	1	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...
0	0	0	0	0	...	1	0.1	0	0
0	0	0	0	0	...	0.1	1	0.1	0
0	0	0	0	0	...	0	0.1	1	0.2
0	0	0	0	0	...	0	0	0.1	1

	A	B	C	D	E
1	**a**	**b**	**c**	λ	eigenvector
2	0	1	0.1	1	-1
3	0.2	1	0.1		0
4	0.1	1	0.1		2
5	0.1	1	0.1		-0
6	0.1	1	0.1		-2
7	0.1	1	0.1		0
8	0.1	1	0.1		2
9	0.1	1	0.1		-0
10	0.1	1	0.1		-2
11	0.1	1	0.1		0
12	0.1	1	0.1		2
13	0.1	1	0.1		-0
14	0.1	1	0.1		-2
15	0.1	1	0.1		0
16	0.1	1	0.1		2
17	0.1	1	0.1		-0
18	0.1	1	0.1		-2
19	0.1	1	0.2		0
20	0.1	1	0		1
21					
22	{=MatEigenvector3(A2:C20;D2)}				
23					

Function MatChar(A, x)

This function returns the characteristic matrix at the value x.

$$C = A - xI$$

where **A** is a real square matrix; x is a real number.
The determinant of **C** is the characteristic polynomial of the matrix **A**

E2 fx {=MatChar(A2:C4,D2)}

	A	B	C	D	E	F	G	H
1		**A**		λ		**A -λ I**		
2	15	13	22	1	14	13	22	
3	-6	-4	-10		-6	-5	-10	
4	-4	-4	-5		-4	-4	-6	

Function MatChar_C(A, z, [Cformat])

This function returns the complex characteristic matrix at the value z.

$$C = A - zI$$

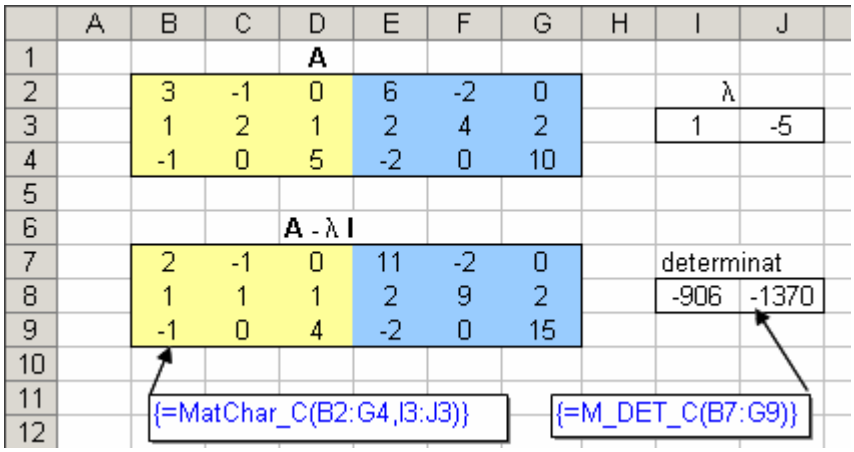
where **A** can be real or complex square matrix; z can be real or complex number.
The determinant of **C** is the characteristic polynomial of the matrix **A**

This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
The optional parameter *Cformat* sets the complex format of input/output (default = 1)

Complex (split or interlaced) matrix must have always an even number of columns

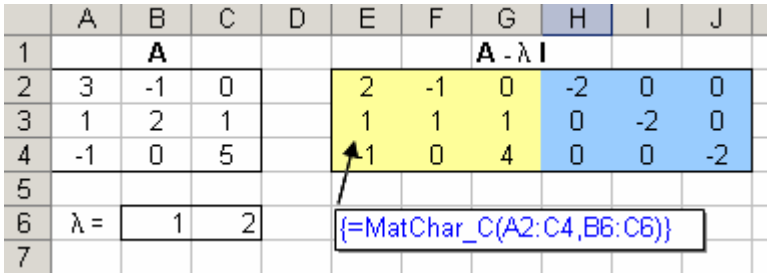
Example. Compute the matrix $A-\lambda I$ for $\lambda = 1-5j$
where **A** is the complex matrix

$$A=\begin{bmatrix} 3+6j & -1-2j & 0 \\ 1+2j & 2+4j & 1+2j \\ -1-2j & 0 & 5+10j \end{bmatrix}$$



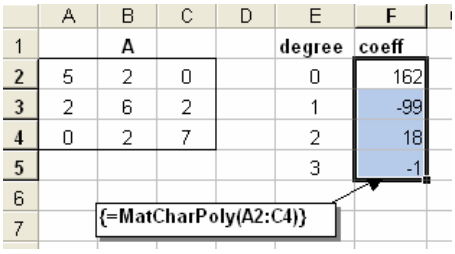
Example. Compute the matrix $A-\lambda I$ for $\lambda = 1+2j$
where **A** is the real matrix

$$A=\begin{bmatrix} 3 & -1 & 0 \\ 1 & 2 & 1 \\ -1 & 0 & 5 \end{bmatrix}$$



Function MatCharPoly(Mat)

This function returns the coefficients of the characteristic polynomial of a given matrix. If the matrix has dimension (n x n), then the polynomial has n^{th} degree and n+1 coefficients.
As know, the roots of the characteristic polynomial are the eigenvalues of the matrix and vice versa. This function uses the fast Newton-Girard formulas to find all the coefficients.



In the example the characteristic polynomial of matrix A is

$$\det(A-\lambda I) = -\lambda^3 + 18\lambda^2 - 99\lambda + 162$$

Solving this polynomial (by any method) can be another way to find eigenvalues

Note. Computing eigenvalues trough the characteristic polynomial is in general less efficient than other decomposition methods (QR, Jacoby), but became a good choose for low dimension matrices (typically < 6°) and for complex eigenvalues

See also Function Poly_Roots(Coefficients, [ErrMax])

Function MatCharPoly_C(Mat, [CFormat])

This function returns the complex coefficients of the characteristic polynomial of a given complex matrix. If the matrix has dimension (n x n), then the polynomial has n^{th} degree and n+1 coefficients.
As know, the roots of the characteristic polynomial are the eigenvalues of the matrix and vice versa.
This function uses the Newton-Girard formulas to find all the coefficients.

It supports 3 different matrix formats: 1 = split, 2 = interlaced, 3 = string
The optional parameter *Cformat* sets the complex format of input/output (default = 1)

The function always returns an array of 2 columns

	A	B	C	D	E	F	G	H	I	J	K
1		real				imag.				Coefficients	
2	4	2	4	5	3	-4	5	-4		8	24
3	1	2	1	2	2	0	2	-1		-22	-2
4	-2	4	-2	2	4	2	2	6		9	7
5	3	-3	3	1	-3	-3	-3	-3		-5	-2
6										1	0
7				{=MatCharPoly_C(A2:H5)}							
8											

As we can see the characteristic polynomial of the above complex matrix is

$$P(z) = z^4 - (5+2j)z^3 + (9+7j)z^2 - (22+2j)z + 8 + 24j$$

Function Poly_Roots(Coefficients, [ErrMax])

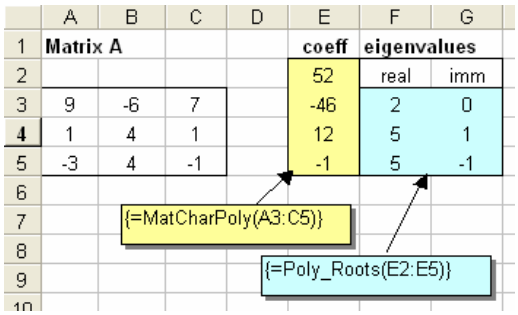
This function returns the roots of a given polynomial
Coefficients parameter is the array of n+1 coefficients
ErrMax optional parameter sets the max error for roots approximation (default = 1E-13)

This function uses the *Lin-Bairstow* algorithm for the factorization of a n^{th} degree polynomial into a square polynomial and a $(n-2)^{th}$ degree polynomial. The process is applied recursively to the $(n-2)^{th}$ polynomial, and so on, until the degree of the reduced polynomial becomes 2 or 1.

Note. This process is very fast, and robust but may not converge under certain conditions: for example if the polynomial has multiple roots. In that case we can try to reduce the accuracy by the *ErrMax* parameter, setting 1E-9, or 1E-6. For high degree polynomial you can use also the **Poly_Root_QR** function in MATRIX. For high accuracy or for stiff polynomials you can find more suitable rootfinder routines in ***XNumbers.xla*** addin

Eigenvalues. If the given polynomial is the characteristic polynomial of a matrix (returned, for example, by the **MatCharPoly()**) this function returns the eigenvalues of the matrix itself.
Computing eigenvalues trough the characteristic polynomial is in general less efficient than other decomposition methods (QR, Jacoby), but became a good choose for low dimension matrices (typically < 6°) and for complex eigenvalues.

Example: find the eigenvalues of the following matrix



Note: we can also use the nested function:

```
{=Poly_Roots(MatCharPoly(A))}
```

Function MatEigenvalue_max(Mat, [IterMax])

Returns the dominant eigenvalue of a matrix.
Dominant eigenvalue, if exists, is the one with the maximum absolute value.
Optional parameter: *IterMax* sets the maximum number of iterations allowed (default 1000).
This function uses the power's iterative method

Power's method - Given a matrix A with n eigenvalues, real and distinct, we have

$$|\lambda_1| > |\lambda_2| > \ldots |\lambda_n|$$

Starting with a generic vector $\mathbf{x}_0$, we have:

$$y_k = A^k x_0 \qquad \lim_{k\to\infty} \frac{y_k^T y_{k+1}}{y_k^T y_k} = \lambda_1 \qquad \lim_{k\to\infty} \frac{y_k}{|y_k|} = u_1$$

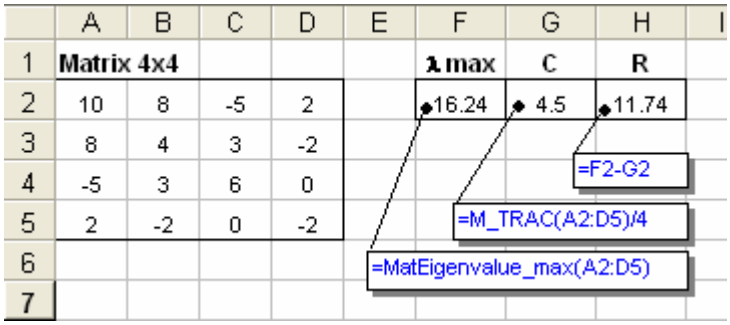
Note. This algorithm is started with a random generic vector. Many times it converges, but some times not. So if one of these functions returns the error "limit iterations exceeded", do not worry. Simply, re-try it.

A global localization method for real eigenvalues

This method is useful for finding the radius of the circle containing all real eigenvalues of a given matrix
Example. Find the circle containing all eigenvalues of the following matrix

10	8	-5	2
8	4	3	-2
-5	3	6	0
2	-2	0	-2

The matrix is symmetric so all its eigenvalues are real.



The matrix trace gives us the sum of eigenvalues, so we can get the center of the circle by:

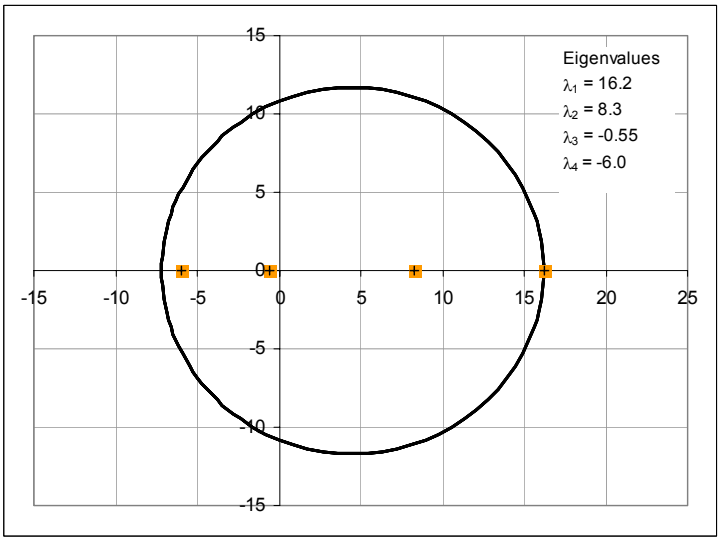
$$c = \frac{trace(A)}{n}$$

We find the dominant eigenvalues λ_1 by the function MatEigenvalue_max

The radius can be found by the formula

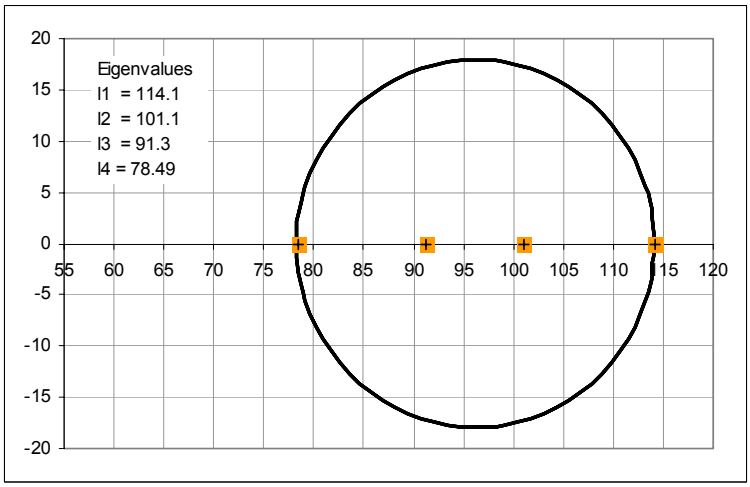
$$r = |\lambda_1| - c$$

We have found the center C = (4.5 ; 0) with R = 11.7. If we plot the circle and the roots, we observe a general good result



This method works also for non symmetric matrices, having real eigenvalues.
Example - find the circle containing all eigenvalues of the following matrix

90	-7	0	4
-5	98	0	12
-2	0	95	14
9	3	14	102



Function MatEigenvector_max(Mat, [Norm], [IterMax])

Returns the dominant eigenvector of matrix **Mat**
The dominant eigenvector is related to the dominant eigenvalue.
Optional parameters are:
IterMax: sets the maximum number of iterations allowed (default 1000).
Norm: if TRUE, the function returns a normalized vector |v|=1 (default FALSE)

Remark: This function uses the power's iterative method
For further details see function MatEigenvalue_Max

Note. This algorithm is started with a random generic vector. Many times it converges, but some times not. So if one of these functions returns the error “limit iterations exceeded”, do not worry. Simply, re-try it.

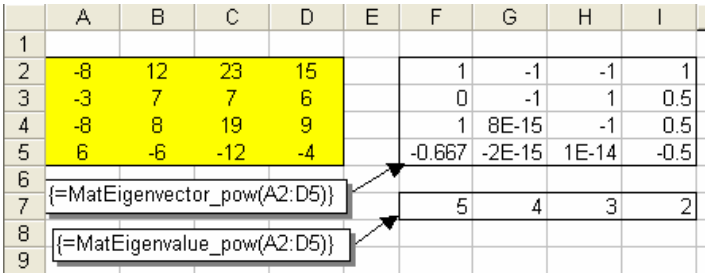
Function MatEigenvalue_pow(Mat, [IterMax])

This function returns all eigenvalues of a given matrix.
Optional parameters are:
IterMax: sets the maximum number of iterations allowed (default 1000).
Norm: if TRUE, the function returns a normalized vector |v|=1 (default FALSE)

This function uses the power’s iterative method. This algorithm works also for non-symmetric matrices with low-moderate dimension

Note. This algorithm is started with a random generic vector. Many times it converges, but some times not. So if one of these functions returns the error “limit iterations exceeded”, do not worry. Simply, re-try it.

Example: find all eigenvalues and eigenvectors of the given matrix



	A	B	C	D	E	F	G	H	I
1									
2	-8	12	23	15		1	-1	-1	1
3	-3	7	7	6		0	-1	1	0.5
4	-8	8	19	9		1	8E-15	-1	0.5
5	6	-6	-12	-4		-0.667	-2E-15	1E-14	-0.5
6									
7						5	4	3	2
8									
9									

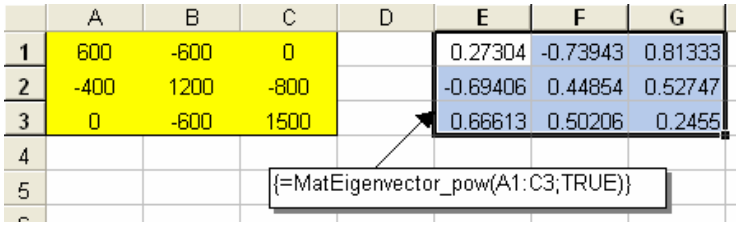
We have used the functions MatEigenvalue_pow and MatEigenvector_pow. We can see the small values instead of 0. This is due to the round-off errors. If you want to clean the matrix from these round-off errors, use the function MatMopUp

Function MatEigenvector_pow(Mat, [Norm], [IterMax])

This function returns all eigenvectors of a given matrix.
Optional parameters are:
IterMax: sets the maximum number of iterations allowed (default 1000).
Norm: if TRUE, the function returns normalized vectors |v|=1 (default FALSE)

This function uses the power’s iterative method. This algorithm works also for non-symmetric matrices with low-moderate dimension

See also function MatEigenvalue_pow



	A	B	C	D	E	F	G
1	600	-600	0		0.27304	-0.73943	0.81333
2	-400	1200	-800		-0.69406	0.44854	0.52747
3	0	-600	1500		0.66613	0.50206	0.2455
4							
5							

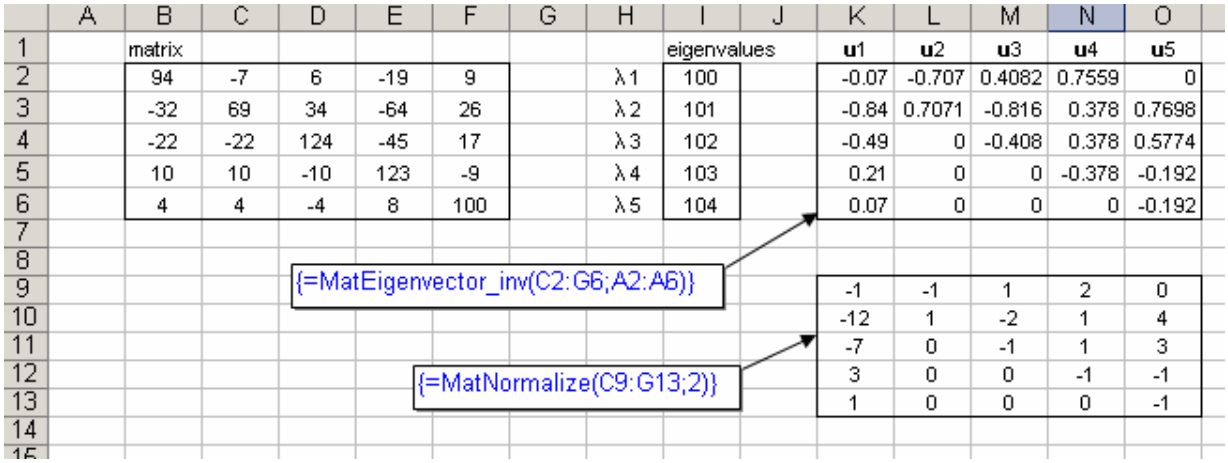
Function MatEigenvectorInv(Mat, Eigenvalue)

This function returns the eigenvector of associate eigenvalue of an (n x n) matrix **A** using the inverse iterative algorithm

$$Av = \lambda v$$

If Eigenvalues is a single value, the function returns a (n x 1) vector. Otherwise if Eigenvalues is a vector of all eigenvalues of matrix A, the function returns a matrix (n x n) of eigenvector.
The eigenvector is normalized with norm = 2 .
This method is adapted for eigenvalues affected by large error, because it is more stable than the singular system resolution.

Example. Given a matrix **A** and its eigenvalues λ_i , find the eigenvectors associated



Function MatEigenvectorInv_C(Mat, Eigenvalue, [CFormat])

This function returns the eigenvector of associate eigenvalue of a complex matrix A (n x n) by the inverse iteration algorithm

$$Av = \lambda v$$

If "Eigenvalue" is a single value, the function returns a complex vector. Otherwise if "Eigenvalues" is a complex vector, the function returns the complex matrix of the associated eigenvectors.

The inverse iteration method is adapted for eigenvalues affected by large error, because is more stable than the singular system resolution of the other function MatEigenvector_C .

Example. Find all eigenvectors of the following complex matrix, having the following eigenvalues

4+3j	2-4j	4+5j	5-4j
1+2j	2	1+2j	2-j
-2+4j	4+2j	-2+2j	2+6j
3-3j	-3-3j	3-3j	1-3j

$$\lambda = [\,4\,,\ 1+3\mathrm{j}\,,\ \mathrm{i}\,,\ -2\mathrm{j}\,]$$

	A	B	C	D	E	F	G	H	I	J	K	L
1		real				imag.				Eigenvalues		
2	4	2	4	5	3	-4	5	-4		4	0	
3	1	2	1	2	2	0	2	-1		1	3	
4	-2	4	-2	2	4	2	2	6		0	-2	
5	3	-3	3	1	-3	-3	-3	-3		0	1	
6												
7	Eigenvectors		{=MatEigenvectorInv_C(A2:H5,J2:K5)}									
8		real				imag.						
9	0	0.896	1	1	0	-0.291	0	0				
10	0	0.896	0	0	0	-0.291	0	-1				
11	-0.880	0	-1	0	-0.293	0	0	0				
12	-0.293	-0.896	0	0	0.880	0.291	0	0				

About perturbed eigenvalues

Many times we know only an approximation of the true eigenvalue. When the error is large the stability of the algorithm for finding the associate eigenvector plays a crucial role. The above example shows a critical situation because all eigenvalues are very closed each others, having only 4% of difference. In this case a little error of the eigenvalues could get large error in eigenvectors. In this situation came handy the inverse iterative algorithm. It shows a large stability. Let's see this example
First of all we define a sensitivity coefficient for measuring the instability.

Instability Sensitivity

$$S_{u,\lambda} = \frac{\sum\left|u_i - u_i^*\right|}{\|u\|} \cdot \frac{|\lambda|}{\left|\lambda - \lambda^*\right|} = \frac{\sum|\Delta u_i|}{\|u\|} \cdot \frac{|\lambda|}{|\Delta\lambda|}$$

Where:

λ = eigenvalue
λ* = perturbed eigenvalue
u = eigenvector
u* = perturbed eigenvector

Now we compare the response of two different algorithms at the perturbed eigenvalue: the singular linear system solving (traditional method) and the iterative inverse algorithm. The first one is used by MatEigenvector() function while the last one is used by the MatEigenvector_inv() function.

λ
100

Singular linear system method

λ	Δ λ
100.0001	0.0001

Iterative inverse method

λ	Δ λ
100.3	0.3

u
1
12
7
-3
-1

u	|Δ u|
0.99981665	1.83E-04
12.00028336	2.36E-05
7.00005834	8.33E-06
-3.00003333	1.11E-05
-1.00000000	0

u	|Δ u|
1.00000000	0
12.00000085	7.12E-08
7.00000046	6.58E-08
-3.00000020	6.58E-08
-1.00000007	6.58E-08

The iterative inverse algorithm returns an eigenvector affected by a very small error even if the error of the eigenvalues is heavier (0.3%). On the other hand the first method computes a sufficently accurate eigenvector only if the eigenvalue error is very little (0.0001%). Note that for higher errors the first method fails, returning the null vector.
On the contrary the iterative inverse algorithm tolerates large amount of error in eigenvalue. This can be showed by the instability factor

Singular linear system method

λ	Δ λ	| u |	Σ|Δ u|
100.0001	0.0001	14.28	0.000226

S1 = 15.85

Iterative inverse method

λ	Δ λ	| u |	Σ|Δ u|
100.3	0.3	14.28	2.69E-07

S2 = 6.3E-06

As we can see the difference is quite evident. In the last case the hill-conditioned matrix (eigenvalues very close each others), exhibits an instability factor of the iterative inverse algorithm less then 10^5 times the other one. Clearly this is a good reason for using it.

Matrices Generator

This is a set of useful tools for generating several types of matrices

Function MatRnd(n, [m], [Typ], [MatInteger], [Amax], [Amin], [Sparse])
Function MatRndEig(Eigenvalues, [MatInteger])
Function MatRndEigSym(Eigenvalues)
Function MatRndRank(n, [Rank], [Det], [MatInteger])
Function MatRndSim(n, [Rank], [Det], [MatInteger])
Function Mat_Hilbert (n)
Function Mat_Hilbert_inv(n)
Function Mat_Householder(x)
Function Mat_Tartaglia(n)
Function Mat_Vandermonde(x)

Function MatRnd(n, [m], [Typ], [MatInteger], [Amax], [Amin], [Sparse])
Generates a random matrix
Parameters are:

n = rows
m = columns (default m = n)

Typ = ALL (default) - fills all cells
SYM - symmetrical
TRD - tridiagonal
DIA - Diagonal
TLW - Triangular lower
TUP - Triangular upper
SYMTRD Symmetrical tridiagonal

MatInteger = True (default) for integer matrix, False for decimal
Amax = max number allowed
Amin = min number allowed
Sparse = decimal, from 0 to 1; 0 means no sparse (default), 1 means very sparse

	A	B	C	D	E	F	G	H	I	J	K	L
1	**full**				**symmetic**				**diagonal**			
2	-1	-7	-1	-6	9	0	-7	4	-9	0	0	0
3	-8	0	5	7	0	-6	7	5	0	7	0	0
4	-6	1	4	5	-7	7	7	-1	0	0	1	0
5	0	9	-6	2	4	5	-1	6	0	0	0	-2
6	=MatRnd("ALL")				=MatRnd("SYM")				=MatRnd("DIA")			
7	**triangular lower**				**triangular upper**				**tridiagonal**			
8	4	0	0	0	7	7	-9	-2	-4	2	0	0
9	10	-6	0	0	0	10	-5	-10	10	2	-8	0
10	-4	4	9	0	0	0	-8	7	0	4	-5	9
11	-1	2	6	9	0	0	0	5	0	0	3	9
12	=MatRnd("TLW")				=MatRnd("TUP")				=MatRnd("TRD")			

	A	B	C	D	E	F
1						
2		-1	-1	0	1	0
3		-1	-1	0	-1	-1
4		0	0	-1	0	-1
5		1	-1	0	-1	-1
6		0	-1	-1	-1	-1
7		{=MatRnd("SYM",,1,-1)}				
8						
9		2	2	0	0	0
10		2	1	1	0	0
11		0	2	2	1	0
12		0	0	2	2	2
13		0	0	0	2	1
14		{=MatRnd("TRD",,2,1)}				

Note: The generation is random; it's means that each time that you recalculate this function, you get different values

Function MatRndEig(Eigenvalues, [MatInteger])
Returns a general real matrix with a given set of eigenvalues

Function MatRndEigSym(Eigenvalues)
Returns a symmetric real matrix with a given set of eigenvalues

	A	B	C	D	E	F	G	H	I	J	K
1	**eigenvalues**		**matrix**					**symmetic matrix**			
2	1		-4	6	9	9		1.758	0.326	-0.55	0.587
3	2		-7	9	9	7		0.326	1.679	0.316	-0.29
4	3		-1	1	4	2		-0.55	0.316	2.738	0.219
5	4		3	-3	-3	1		0.587	-0.29	0.219	3.826
6			{=MatRndEig(A2:A5)}					{=MatRndEigSym(A2:A5)}			

Function MatRndRank(n, [Rank], [Det], [MatInteger])
Returns a real matrix with a given Rank or Determinant
Note: if Rank < max dimension then always Det = 0

Function MatRndSim(n, [Rank], [Det], [MatInteger])
Returns a real symmetric matrix with a given Rank or Determinant
Note: if Rank < max dimension then always Det = 0

Function MatRndUni(n, [MatInteger])
Returns an unitary matrix (Det=1)

Function Mat_Hilbert(n)
Returns the (n x n) Hilbert's matrix
Note: this matrix is always decimal

Function Mat_Hilbert_inv(n)
Returns the inverse of the (n x n) Hilbert's matrix.
Note: this matrix is always integer

Hilbert's matrices are a strongly hill-conditioned and are useful for testing algorithms
In the example below we see a (4 x 4) Hilbert matrix and its inverse.

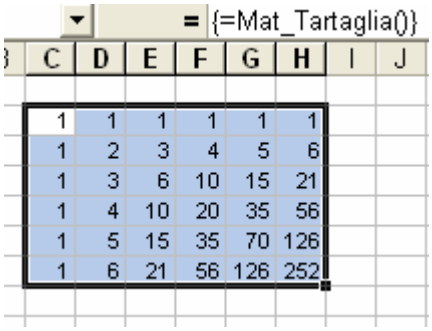
	A	B	C	D	E	F	G	H	I	J
1										
2		1	1/2	1/3	1/4		16	-120	240	-140
3		1/2	1/3	1/4	1/5		-120	1200	-2700	1680
4		1/3	1/4	1/5	1/6		240	-2700	6480	-4200
5		1/4	1/5	1/6	1/7		-140	1680	-4200	2800
6										
7		{=Mat_Hilbert(4)}					{=Mat_Hilbert_inv(4)}			

Function Mat_Householder(x)
Returns the Householder matrix of a given vector **x** = (x1, x2, ...xn) by the formula:

$$H = I - 2\frac{XX^T}{\|X\|^2}$$

This kind of matrices are used in several important algorithms as, for example, the QR decomposition

Function Mat_Tartaglia()
Returns the Tartaglia's matrix
This kind of matrix is a hill-conditioned and is useful to test same algorithms
In the example below we see a (5 x 5) matrix



Definition: Tartaglia's matrix is defined as

$$a_{1j} = 1$$

$$a_{ij} = \sum_{k=1}^{j} a_{(i-1)\,k}$$

Function Mat_Vandermonde(x)
Returns the Vandermonde's matrix of a given vector **x** = (x1, x2, ...xn)

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & {x_1}^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & {x_2}^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & {x_n}^{n-1} \end{bmatrix}.$$

This matrix is very common in many field of numeric calculus like the polynomial interpolation
Example: Find the 4th degree polynomial that fits the following table

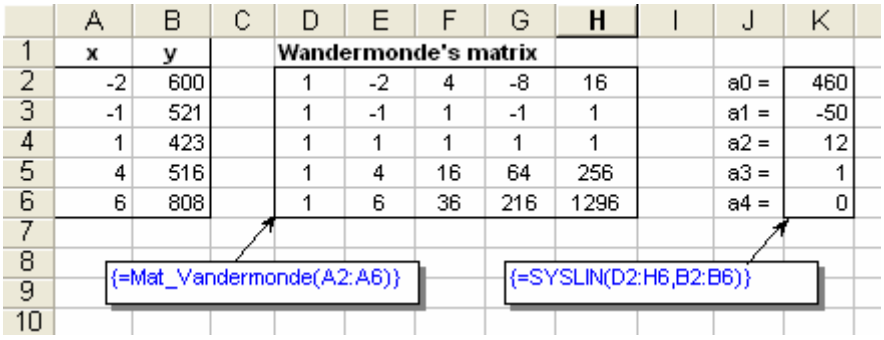
x	y
-2	600
-1	521
1	423
4	516
6	808

The generic 4th degree polynomial is

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$

We can find the coefficients a = (a_0, a_1, a_2, a_3, a_4) solving the linear system

W a = y Where **W** is the Wandermonde's matrix



Function Gauss_Jordan_step(Mat, [Typ], [IntValue])

This function, also available in macro version, has been developed for didactic scope. It can trace, step by step, the diagonal reduction or triangular reduction of a matrix by the Gauss-Jordan algorithm.
Optional parameter *Typ* can be "D" (default) for Diagonal or "T" for Triangular
Optional parameter *IntValue* = TRUE forces the function to conserve integer values through all steps. Default is FALSE.
The argument *Mat* is the complete matrix (n x m) of the linear system
Remember that for a linear system:

$$Ax = b$$

A is the system square matrix (n x n)
x is the unknown vector (n x 1)
b is the vector of constant terms (n x 1)

$$C = [A,\ b]$$

C is the complete matrix of the system

Example - Study the Gauss-Jordan algorithm for the following system

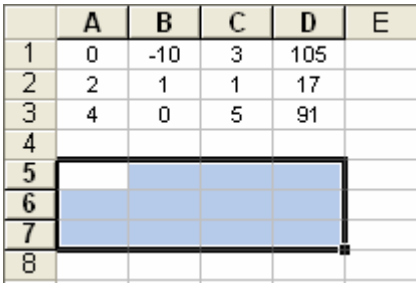
A =

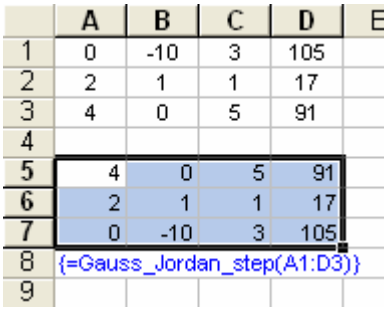
0	-10	3
2	1	1
4	0	5

b =

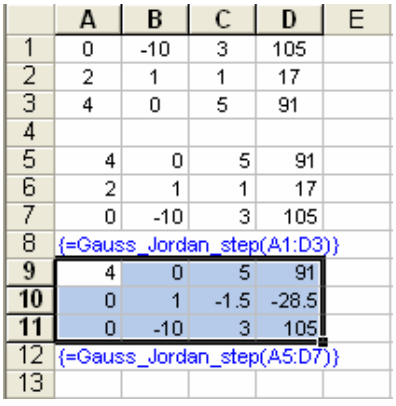
105
17
91

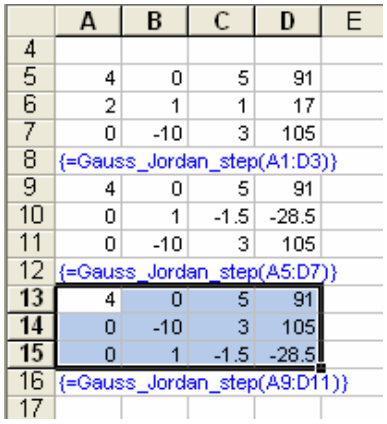
First of all, put all columns in an adjacent 3x4 matrix, example the range A1:D3. Select the cells where you want the matrix of the next step; example the range A5:D7. Insert the array-function





As we can see, the algorithm has exchanged rows 1 and 3, because the pivot a_{11} was 0
Now, with the range A5:D7 still selected, copy the active selection by CTRL+C
Move to the cell A9 give the command CTRL+V. The new step will be performed. Continuing in this way we can visualize each step of the elimination algorithm





	A	B	C	D	E
13	4	0	5	91	
14	0	-10	3	105	
15	0	1	-1.5	-28.5	
16	{=Gauss_Jordan_step(A9:D11)}				
17	4	0	5	91	
18	0	-10	3	105	
19	0	0	-1.2	-18	
20	{=Gauss_Jordan_step(A13:D15)}				
21	4	0	0	16	
22	0	-10	3	105	
23	0	0	-1.2	-18	
24	{=Gauss_Jordan_step(A17:D19)}				
25					

	A	B	C	D	E
20	{=Gauss_Jordan_step(A13:D15)}				
21	4	0	0	16	
22	0	-10	3	105	
23	0	0	-1.2	-18	
24	{=Gauss_Jordan_step(A17:D19)}				
25	4	0	0	16	
26	0	-10	0	60	
27	0	0	-1.2	-18	
28	{=Gauss_Jordan_step(A21:D23)}				
29	1	0	0	4	
30	0	1	-0	-6	
31	0	0	1	15	
32	{=Gauss_Jordan_step(A25:D27)}				
33					

The process ends when the 3x3 matrix is became an identity matrix. The system solution appears in the last column (4, -6, 15)

For further details see: Several ways for using the Gauss-Jordan algorithm

Function SYSLIN(A, b, [IMode], [Tiny])

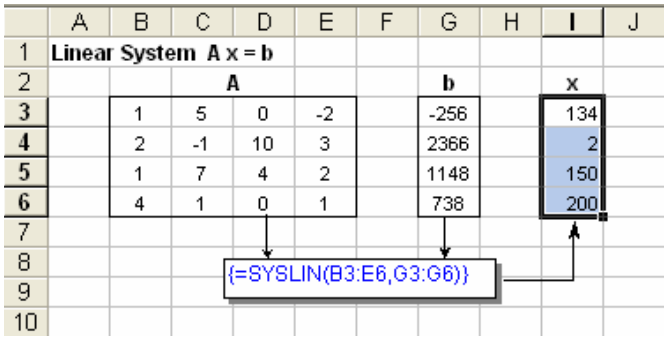
This function solves a linear system by the Gauss-Jordan algorithm.
The argument **A** is the matrix (n x n) of the linear system
The argument **b** is a (n x 1) vector or a (n x m) matrix
The optional parameters:IMODE switch (default False) sets the floating point (False) or integer computation (True). Integer computation is intrinsically more accurate but also more limited because it may easily reaches the overflow error. Use IMODE only with integer matrices of moderate size.
The optional parameter Tiny (default is 0) sets the minimum round-off error; any value in absolute less than Tiny will be set to 0.
If the matrix is singular the function returns "singular"
If the matrix is not squared the function returns "?"
If non-singular, returns the vector solution or the matrix solution of the given system.
Remember that for a linear system:

$$Ax = b$$

A is the system square matrix (n x n)
x is the unknown vector (n x 1) or the unknown matrix (n x m)
b is the vector of constant terms (n x 1) or a (n x m) matrix of constant terms

As known, the above linear equation has only one solution if - and only if -, Det(A) ≠ 0
Otherwise the solutions can be infinite or even nothing. In that case the system is called "singular". See Function SYSLINSING(Mat, Optional V)

$$AX = B \Rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_{11} & x_{12} & x_{1m} \\ x_{21} & x_{22} & \dots x_{2m} \\ x_{31} & x_{32} & x_{3m} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{1m} \\ b_{21} & b_{22} & \dots b_{2m} \\ b_{31} & b_{32} & b_{3m} \end{bmatrix}$$



Parameter **b** can also be a matrix of m columns. In that case SYSLIN solves simultaneosly a set of m systems.

Function SYSLIN3(Mat3, v)

This function solves a tridiagonal linear system.
The argument Mat3 is the array (n x 3) representing the (n x n) matrix of the linear system
Remember that for a linear system:

A x = v

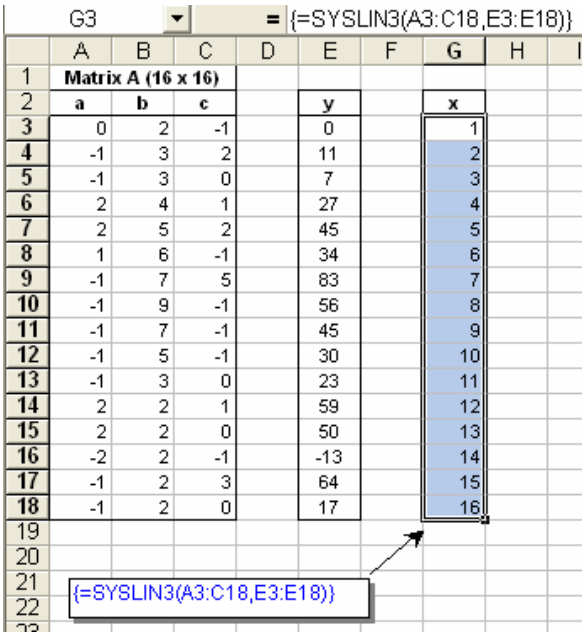
A is the system square matrix (n x n)
x is the unknown vector (n x 1)
v is the constant terms vector (n x 1)

As known, the above linear equation has only one solution if - and only if -, det(**A**) ≠ 0
Otherwise the solutions can be infinite or even nothing. In that case the system is called "singular".

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 \\ 0 & 0 & 0 & a_5 & b_5 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}$$

Example - let' se how to solve a 16 x 16 tridiagonal linear system **A x = y**
We pass to the function only 46 values (the first cell of **a** and the last of **c** are always 0) instead of 256 values.

Tip: note that this trick allows to solve systems larger that 256 x 256 (the max square matrix in Excel worksheet)



Function SYSLIN_ITER_G(A, b, X0, Optional Nmax)

This function performs the iterative Gauss-Seidel algorithm for solving a linear system and has been developed for didactic scope in order to study the convergence of the iterative process.

$$[\mathbf{A}]\cdot\mathbf{x}=\mathbf{b}$$

Parameter **A** is the system matrix (range n x n)
Parameter **b** is the system vector (range n x 1)
Parameter $\mathbf{x_0}$ is the starting approximate solution vector (range n x 1)
Parameter **Nmax** is the max step allowed (default = 1)

The function returns the vector at Nmax step; if the matrix is convergent, this vector approaches to the exact solution.

In the example below it is shown the 20th iteration step of this iterative method.
As we can see, the values approximate the exact solution [4, -3, 5]. Precision increase with steps (of course, for convergent matrix)

	A	B	C	D	E	F	G	H
1	**Linear system resolution with iterative methods**							
2						**Step =>**	0	20
3	6	-1	2	37		**X1 =>**	0	3,999984082
4	2	-7	6	59		**X2 =>**	0	-2,999979881
5	-1	3	5	12		**X3 =>**	0	4,999984745
6								
7		{=SYSLIN_ITER_G(A3:C5;D3:D5;G3:G5;H2)}						
8								

For Nmax=1, we can study the iterative method step by step

x1 = SYSLIN_ITER_G(A, b, x0)
x2 = SYSLIN_ITER_G(A, b, x1)
x3 = SYSLIN_ITER_G(A, b, x2)
.....
x20 = SYSLIN_ITER_G(A, b, x19)

In the example below we see the trace of iteration values

B	C	D	E	F	G	H	I	J
Linear system resolution with iterative methods								
6	-1	2	37					
2	-7	6	59					
-1	3	5	12					
Gauss-Seidel method								
x1	x2	x3						
0	0	0						
6,16667	-6,6667	7,63333	{=SYSLIN_ITER_G($B$4:$D$6;$E$4:$E$6;B10:D10)}					
2,51111	-1,1683	3,60317	{=SYSLIN_ITER_G($B$4:$D$6;$E$4:$E$6;B11:D11)}					
4,7709	-3,977	5,74039	{=SYSLIN_ITER_G($B$4:$D$6;$E$4:$E$6;B12:D12)}					
3,59037	-2,4824	4,60752	.....					
4,21709	-3,2744	5,20805	.....					

Usually, the convergence speed is quite low, but it can be greatly accelerate by the Aitken's extrapolation formula, also called as "*square delta extrapolation*"

Function SYSLIN_T(Mat, b, [typ], [tiny])

This function solves a triangular linear system by the forward and backward substitutions algorithms. It returns a vector or a matrix solution of a given system.
The argument Mat is the matrix (n x n) of the linear system
Remember that for a linear system:

$$Ax = b$$

A is the triangular - upper or lower - system square matrix (n x n)
x is the unknown (n x 1) vector or the (n x m) unknown matrix
b is a constant (n x 1) vector or a constant (n x m) matrix

As known, the above linear system has only one solution if - and only if -, det(**A**) <> 0
Otherwise the solutions can be infinite or even nothing. In that case the system is called "singular".

The parameter **b** can be also a (n x m) matrix **B**. In that case the function returns a matrix solution **X** of the multiple linear system
Parameter typ = "U" or "L" switches the function to solve for upper-triangular (back substitutions) or lower-triangular system (forward substitutions); if omitted the function automatically detects the type of the system.
Optional parameter Tiny (default is 0) sets the minimum round-off error; any value in absolute less than Tiny will be set to 0.

Example of (7 x 7) system

	A	B	C	D	E	F	G	H	I	J	K	L
1	**Upper-triangular matrix system (7x7)**								**b**		**x**	
2	9	-10	-5	8	2	1	-8		-6.4		1.1	
3	0	-10	4	9	-1	5	-8		-1.3		1.2	
4	0	0	8	-7	10	1	-8		3.6		1.3	
5	0	0	0	-3	-6	0	-1		-14.9		1.4	
6	0	0	0	0	-2	7	-7		-3.7		1.5	
7	0	0	0	0	0	-5	1		-6.3		1.6	
8	0	0	0	0	0	0	-7		-11.9		1.7	
9												
10												
11					{=SYSLIN_T(A2:G8,I2:I8)}							
12												

Function SYSLIN_ITER_J(Mat, U, X0, Optional Nmax)

This function performs the iterative Jacobi's algorithm for solving a linear system and was developed for didactic scope in order to study the convergence of the iterative process.

$$[\mathbf{A}] \cdot \mathbf{x} = \mathbf{b}$$

Parameter **A** is the system matrix (range n x n)
Parameter **b** is the system vector (range n x 1)
Parameter $\mathbf{x_0}$ is the starting approximate solution vector (range n x 1)
Parameter **Nmax** is the max step allowed (default = 1)

The function returns the vector at Nmax step; if the matrix is convergent, this vector is closer to the exact solution.

This function is similar to the SYSLIN_ITER_G function.
For further details see Function SYSLIN_ITER_G(Mat, U, X0, Optional Nmax)

Function SYSLINSING(A, [b], [MaxErr])

Singular linear system can have infinite solutions or even nothing. This happens when DET(A) = 0.
In that case the following equations:

$$Ax = b$$

$$Ax = 0$$

The above equation define an implicit *Linear Function* - also called *Linear Transformation* - between the vector spaces, that can be put in the following explicit form

$$y = Cx + d$$

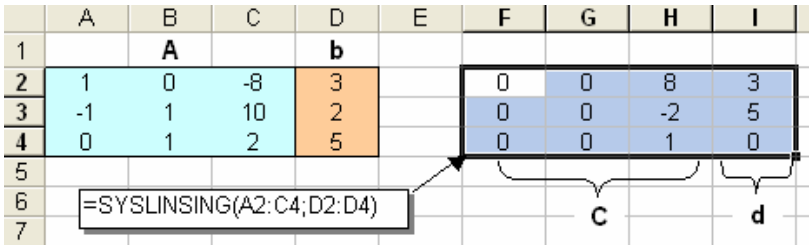
Where **C** is the transformation matrix and **d** is the known vector; **C** has the same dimension of **A**, and **d** the same of **b**

This function returns the matrix **C** in the first n columns; eventually, a last column contains the vector **d** (only if b is not missing). If the system has no solution, this function returns "?"

Optional parameter MaxErr set the relative precision level; elements lower than this level are force to zero. Default is 1E-15.

This version solves also systems where the equations number is less than the variables number; In other words, **A** is a rectangular matrix (n x m) where n < m

Example: Solve the following system



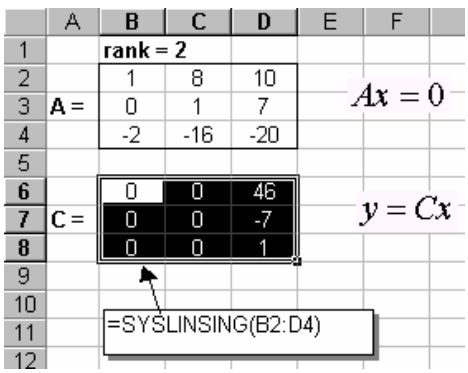
The determinant of the matrix A is 0. The system has infinite solution given by matrix C and vector d

Example 1: Find the solution of the following homogeneous system

$$\begin{bmatrix} 1 & 8 & 10 \\ 0 & 1 & 7 \\ -2 & -16 & -20 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

Because the determinant is 0, the homogeneous system has always solutions; they can be put in the following form

$$y = Cx$$



Looking at the first diagonal of the matrix **C** we discover 1 at the column 3 and row 3. This means that x_3 is the independent variable; all the others are dependent variables. This means also that the rank of matrix is 2.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 46 \\ 0 & 0 & -7 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 46x_3 \\ -7x_3 \\ x_3 \end{bmatrix}$$

Changing values to the independent variable x_3 we get all the solution of the given system

Example 2: Find the solution of the following homogeneous system

$$\begin{cases} x_1 + 3x_2 - 4x_3 = 0 \\ 13x_1 + 39x_2 - 52x_3 = 0 \\ 9x_1 + 27x_2 - 36x_3 = 0 \end{cases}$$

	I	J	K	L	M	N
1		rank = 1				
2		1	3	-4		
3	A =	13	39	-52	$Ax = 0$	
4		9	27	-36		
5						
6		0	-3	4		
7	C =	0	1	-0	$y = Cx$	
8		0	-0	1		

By inspection of the first diagonal, we see that there are 2 elements different from 0. So the independent variables are x_2, and x_3. This means that the rank of matrix is 1

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 & -3 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} -3x_2 + 4x_3 \\ x_2 \\ x_3 \end{bmatrix}$$

It is easy to prove that this linear function is a plane in R3. In fact, eliminating the variable x_2 and x_3 we get:

$$y_1 = -3y_2 + 4y_3$$

And substituting the variables y1, y2, y3 with the usually variables x, y, z, we get:

$$x + 3y - 4z = 0$$

Example 3: Find the solution of the following non-homogeneous system

$$Ax = b$$

$$\begin{bmatrix} 1 & 8 & 10 \\ 0 & 1 & 7 \\ -2 & -16 & -20 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ -4 \\ 0 \end{bmatrix}$$

	A	B	C	D	E
1		rank = 2			
2			A		b
3		1	8	10	0
4		0	1	7	-4
5		-2	-16	-20	0
6					
7			C		d
8		0	0	46	32
9		0	0	-7	-4
10		0	0	1	0
11					
12		{=SYSLINSING(B3:D5;E3:E5)}			
13					

As we can see, the rank of the given system is 2; so there is one independent variable x_3. The solutions can be writing as:

$$y = Cx + d$$

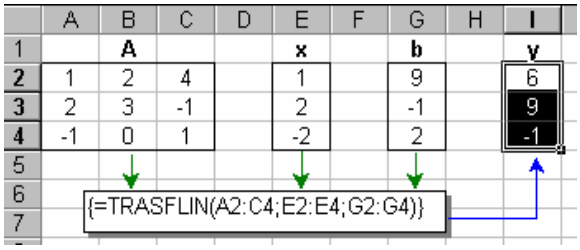
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 46 \\ 0 & 0 & -7 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 32 \\ -4 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 46x_3 + 32 \\ -7x_3 \\ x_3 \end{bmatrix}$$

Function TRASFLIN(A, x, Optional B)

This function performs the Linear Transformation

$$y = Ax + b$$

Where:
A is the (n x m) matrix of transformation
b is the (n x 1) known vector default is the null vector
x is the (m x 1) vector of independent variables
y is the (n x 1) vector of dependent variables



This function accepts also matrices for x and b; in that case the matrix transformation is

$$Y = AX + B$$

Where:
A is the matrix (n x m) of transformation
B is the known matrix (n x p); default is the null vector
X is the matrix of independent variables (m x p)
Y is the matrix of dependent variables (n x p)

Matrix Geometric action

Linear transformation have a useful geometric interpretation[8].
Take a point x (x_1, x_2) of the plane and compute the linear transform y = [A] x, where A (2 x 2) matrix; the point y (y_1, y_2) . We wonder if there is a geometrical relation between the point x and y.
The relation exist and became evident if we perform the transformation of the points belong to the unitary circle.
In Excel we can easily generate the unitary circle pattern with the formula

$x = (\cos(k \cdot \Delta\alpha), \sin(k \cdot \Delta\alpha))$ for k = 1, 2 …N where $\Delta\alpha = 2\pi/N$

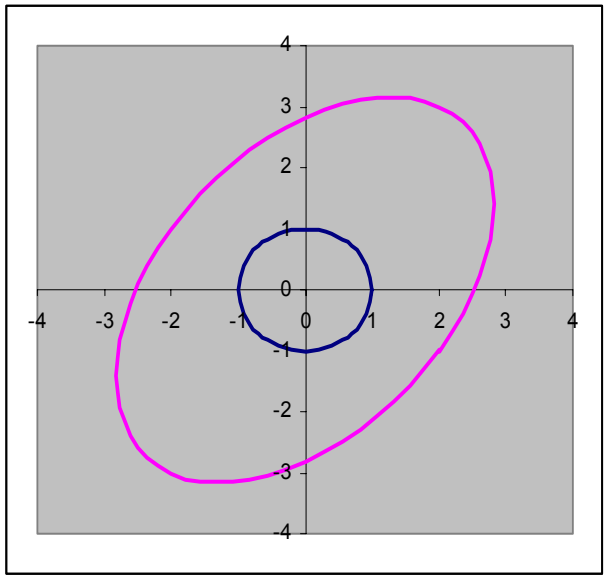
Because x is a row-vector (more adapted to form Excel list) is useful to have the dual Linear Transform for row-vectors

$$y^T = x^T A^T + b^T$$

Here a possible arrangement.

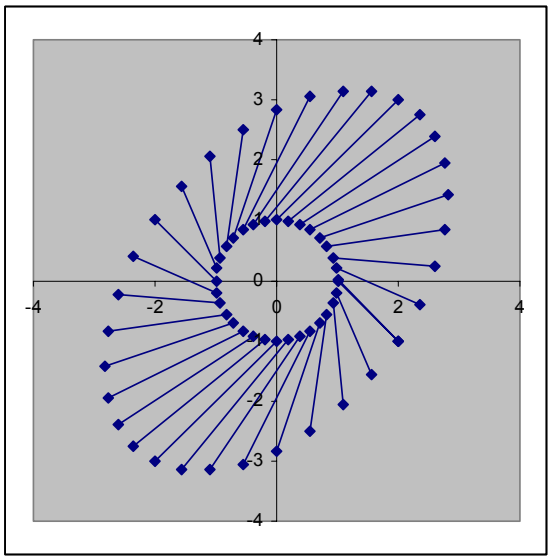
[8] A smart, cool, geometric description was developed by Todd Will at University of Wisconsin-La Crosse. I suggest to have a look at his web pages http://www.uwlax.edu/faculty/will/svd/ . . For those that think that Linear Algebra cannot be amusing. Don't miss them.

	A	B	C	D	E	F	G
1							
2	A =		2	2			
3			-1	3			
4							
5	N =	32	Δα =	0.2			
6							
7	n	t	x1	x2	y1	y2	
8	1	0	1	0	2	-1	
9	2	0.2	0.98	0.2	2.35	-0.4	
10	3	0.39	0.92	0.38	2.61	0.22	
11	4	0.59	0.83	0.56	2.77	0.84	
12	5	0.79	0.71	0.71	2.83	1.41	
13		{=MMULT(C8:D8;M_T($C$2:$D$3))}					
14	7	1.18	0.38	0.92	2.61	2.39	
15	8	1.37	0.2	0.98	2.35	2.75	
16	9	1.57	0	1	2	3	



The blue line corresponds to the unitary circle points; the pink line belongs to the transformed y points. Thus, the circle has been transformed into a centred ellipse; if we add also b ≠ 0, we get a translated ellipse. Each point of unitary circle has been projected on the ellipse.

We have to point out that the projection is not radial but it happens in a very strange way. Look at the imagine to the right. It shows how a point on the circle moves on the ellipse, before and after the linear transformation.
It seems as if the circle would be enlarged (stretched), and then rotated (like a “whirlpool”).



Other dimensions
The effect is the same – changing the denomination – in higher dimension

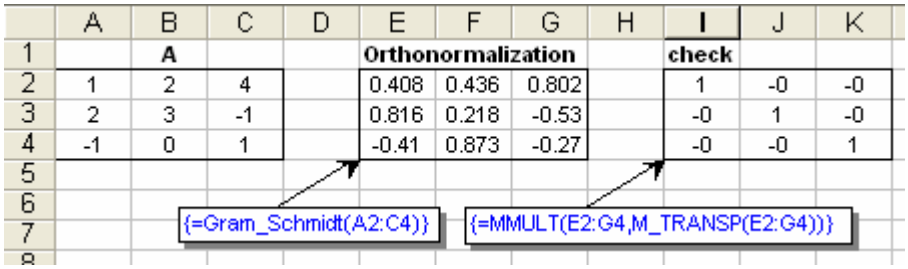
Space	Original pattern	Transf. pattern
R^2	circle	⇒ ellipse
R^3	sphere	⇒ ellipsoid
R^n	hyper-sphere	⇒ hyper-ellipse

Function Gram_Schmidt(A)

This function performs the orthonormalization of a base vectors by the Gram-Schmidt's method
Argument **A** is a (n x n) matrix containing n independent vectors.
This function returns the orthogonal matrix **U**; each vector has $|\mathbf{v}_i| = 1$ and $\mathbf{v}_i \perp \mathbf{v}_j$

$$U = (v_1, v_2, v_3 ..., v_n) \quad \text{Where} \qquad v_i \bullet v_j = \begin{cases} 1 & \Leftrightarrow \ i = j \\ 0 & \Leftrightarrow \ i \neq j \end{cases}$$

This function is very sensible to the round-off errors.
For larger matrices see Function MatOrtNorm(Mat)



Gram-Schmidt's Orthonormalization

This popular method is used to build an orthogonal-normalized base from a set of n independent vectors.

$$(v_1, v_2, v_3, ...v_n)$$

The orthogonal bases **U** is built with the following iterative algorithm

For k = 1, 2, 3...n

$$w_k = v_k - \sum_{j=1}^{k-1} (u_k \bullet u_j)\, u_j$$

$$u_k = \frac{w_k}{\|w_k\|}$$

Developing this algorithm, we see that the vector k is built with the previous k-1 vectors

$$u_1 = \frac{v_1}{\|v_1\|}$$

$$w_2 = v_2 - (v_2 \bullet u_1)\, u_1 \ \Rightarrow \ u_2 = \frac{w_2}{\|w_2\|}$$

$$w_3 = v_3 - (v_3 \bullet u_1)\, u_1 - (v_3 \bullet u_2)\, u_2 \ \Rightarrow \ u_3 = \frac{w_3}{\|w_3\|}$$

At the end, all vectors of the bases U will be orthogonal and normalized.
This process is performed by the function Gram_Schmidt(Mat)

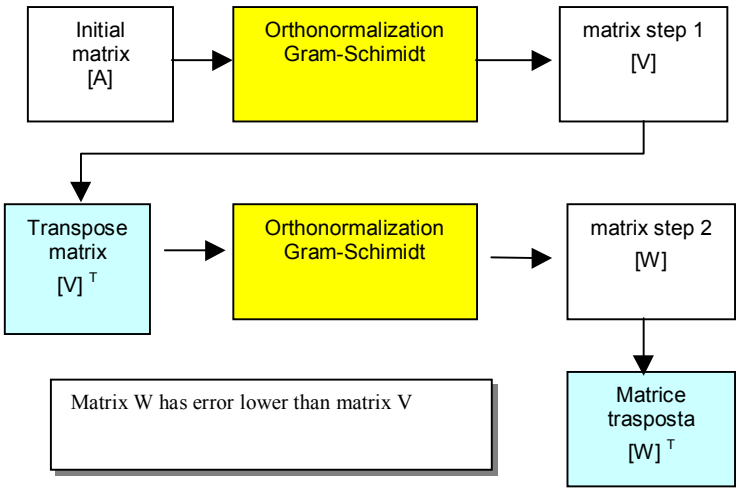
This method is very straightforward, but it's also very sensible to the round-off errors. This happens because the error propagates itself along the vectors from 1 to n

Double step Gram-Schimdt method

One method to reduce the error propagation is the following

1) First of all we apply the normal method in order to obtain a first approximate base U
2) At the second step we repeat the orthonormalization with the transpose of the bases U
3) At the end we transpose again the bases obtained from the 2° step.
This method is performed by the function Function MatOrtNorm(Mat)

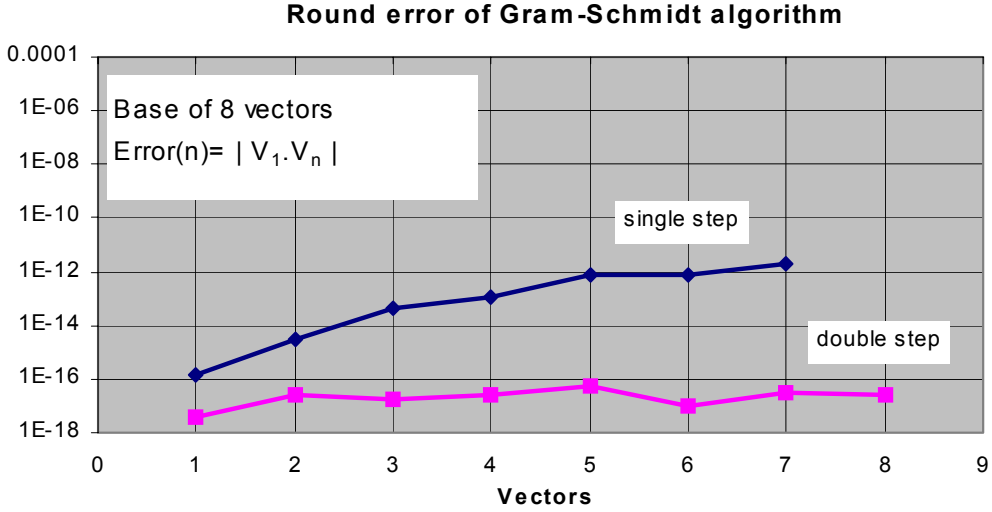
The following flow-chart illustrated better how this method works



In order to measure the round-off errors we take the scalar products between the first vectors and all the others

$$e_{12} = |u_1 \bullet u_2| \quad e_{13} = |u_1 \bullet u_3| \quad \ldots \quad e_{1n} = |u_1 \bullet u_n|$$

This graph below show this errors for a typical matrix of 8° order, orthogonalized with single and double step Gram-Schmidt method



As we can see the improving is sensible even from a (3 x 3) matrix

Function Mat_Cholesky(A)

This function returns the Cholesky decomposition of a symmetric matrix

$$A = L \cdot L^T$$

Where **A** is a symmetric matrix, L is a lower triangular matrix
This decomposition works only if A is *positive definite*. That is:

$$v \cdot A \cdot v > 0 \qquad \forall v$$

or, in other words, that the eigenvalues of **A** are all-positive. This function returns always a matrix. Inspecting the diagonal elements of the matrix returned we can discover if the matrix is positive definite: if they are all positive then also the matrix **A** is positive definite.

Example - Say if the given matrices are positive definite

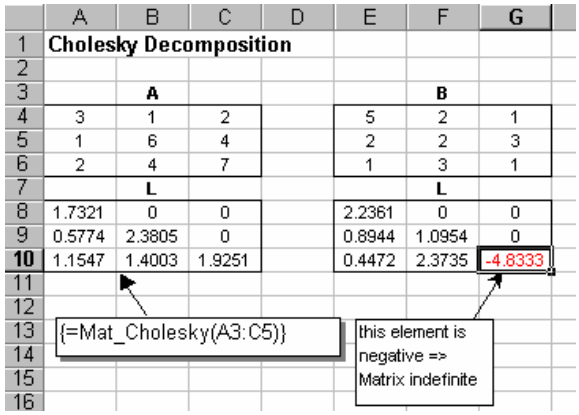
A

3	1	2
1	6	4
2	4	7

B

5	2	1
2	2	3
1	3	1

On the left, we see the decomposition of matrix **A**; the triangular matrix **L** has all diagonal elements positive; then the matrix **A** is positive definite and all its eigenvalues are positive.
On the contrary, the decomposition of the matrix **B** shows a negative number at the position 33; then we can say that **B** is not positive definite and same of its eigenvalues are negative.



This decomposition is useful also to solve the so-called "*generalized eigen-problem*"

Function Mat_LU(A, optional Pivot)

This function returns the LU decomposition of a given square matrix **A**. It uses the Crout's algorithm

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ \alpha_{21} & 1 & 0 \\ \alpha_{31} & \alpha_{23} & 1 \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & \beta_{33} \end{bmatrix}$$

Where **L** is a lower triangular matrix, and **U** is upper triangular matrix
If the square matrix has (n x n) dimension, this function returns a matrix (n x 2n) where the first n columns are the matrix **L** and the last n columns are the matrix **U**.
The parameter Pivot (default=TRUE) activates the partial pivoting.
Note: that if partial pivot is active (TRUE) the LU decomposition may refer to a permutation of **A**.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		**A**				**L**			**U**				**B = LU**		
2	1	2	4		1	0	0	2	3	-1		2	3	-1	
3	2	3	-1		-0.5	1	0	0	1.5	0.5		-1	0	1	
4	-1	0	1		0.5	0.33	1	0	0	4.33		1	2	4	
5					{=Mat_LU(A2:C4)}							{=MMULT(E2:G4,H2:J4}			
6		**A**				**L**			**U**				**B = LU**		
7	1	2	4		1	0	0	1	2	4		1	2	4	
8	2	3	-1		2	1	0	0	-1	-9		2	3	-1	
9	-1	0	1		-1	-2	1	0	0	-13		-1	0	1	
10					{=Mat_LU(A7:C9,FALSO)}							{=MMULT(E7:G9,H7:J9)}			
11															

Note: LU decomposition without pivoting does not work if the first element of diagonal of **A** is zero

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
19		**A**				**L**			**U**				**B = LU**		
20	0	1	2		1	0	0	0	1	2		0	1	2	
21	1	6	4		1	1	0	0	5	2		0	6	4	
22	2	4	7		2	0.4	1	0	0	2.2		0	4	7	
23					{=Mat_LU(A20:C22,FALSE)}										

The LU decomposition are often used to solve linear system

$$\mathbf{A\,x = b} \quad \Rightarrow \quad \mathbf{LU\,x = b} \quad \Rightarrow \quad \mathbf{L\,(Ux) = b}$$

The original system is now split into two simpler systems.

$$\mathbf{L\,y = b} \qquad (1)$$

$$\mathbf{U\,x = y} \qquad (2)$$

First of all, we solve the vector **y** from the system (1), then, substituting **y** into (2), we solve for the vector **x**. Solving a triangular system is quite simple.

$$y_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij} y_j\right)$$

For i = 1, 2, ...N

Forward substitution. For lower triangular system like L y = b

$$x_i = \frac{1}{\beta_{ii}}\left(y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j\right)$$

For i = N, N-1, ...2, 1

Backward substitution. For upper triangular system like U x = y

For a good and accurate explanation of this method see [2]

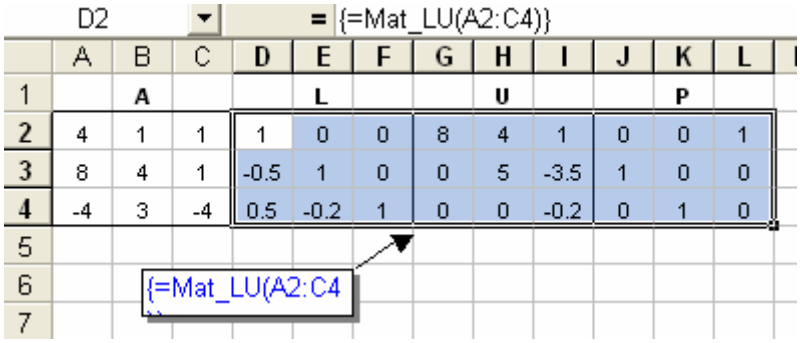
When pivoting is activate the right decomposition formula is **A = P L U** , where **P** is a permutation matrix

This function can return also the permutation matrix in the last n columns

Globally, the output of Mat_LU function will be:

Columns 1, n	Matrix **L**
Columns n+1, 2n	Matrix **U**
Columns 2n+1, 3n	Matrix **P**

Example: find the factorization of the following 3x3 matrix **A**



Function Mat_QR(Mat)

This function performs the QR decomposition of a square (n x n) matrix

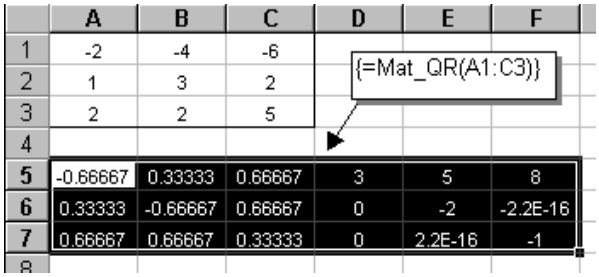
$$A = Q \cdot R$$

Where **Q** is orthogonal and **R** is upper triangular matrix

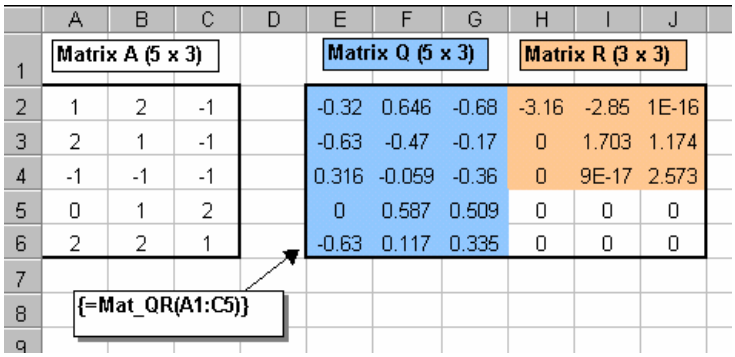
In this version[9] Mat_QR can factors also a rectangular (n x m) matrix. It returns a matrix (m x (n + n)), where the first (m x n) block is **Q** and the first n rows of the (m x n) second block is **R**. The last m – n rows of the second block are 0.

The QR decomposition is the base for an efficient method for calculating the eigenvalues. See also the function MatEigenvalue_QR(Mat, Optional MaxLoops, Optional Acc)

Example 1°: Perform the QR decomposition for the given square matrix



Example 2° Perform the QR decomposition for the given rectangular matrix



[9] Thanks to Ola Mårtensson

Function Mat_QR_iter(Mat, [MaxLoops])

This function performs the diagonalization of a symmetric matrix by the QR iterative process
The heart of this method is the QR iterated decomposition

$$A = QR \quad \Rightarrow \quad A_1 = RQ$$
$$A_1 = Q_1 R_1 \quad \Rightarrow \quad A_2 = R_1 Q_1$$
$$A_2 = Q_2 R_2 \quad \Rightarrow \quad A_3 = R_2 Q_2$$
$$A_n = Q_n R_n \quad \Rightarrow \quad A_{n+1} = R_n Q_n$$

If the matrix **A** has:

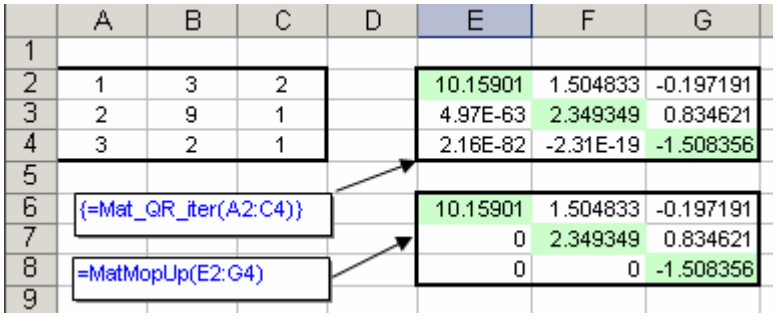
$$|\lambda_2| > \cdots > |\lambda_n|$$

Then the sequence converges to the diagonal matrix of eigenvalues

$$\lim_{n \to \infty} A_{n+1} = [\lambda]$$

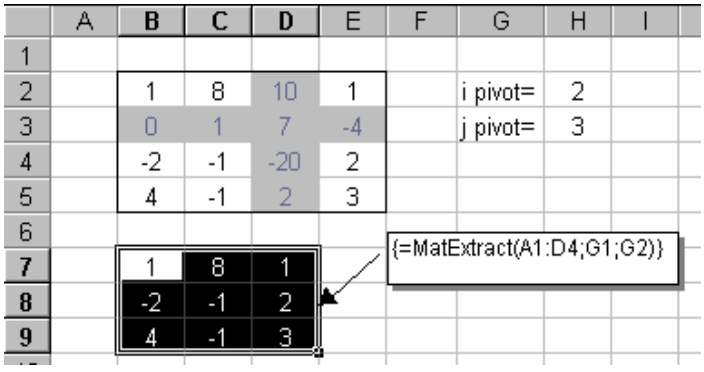
If the matrix is not symmetric the process gives a triangular matrix where the diagonal elements are still the eigenvalues.
Optional parameter *MaxLoops* (default 100) sets the max iteration allowed.

Example.



Function MatExtract(A, i_pivot, j_pivot)

Returns the sub-matrix extract from A by eliminating one row and one column
i_pivot =row to eliminate
j_pivot = column to eliminate



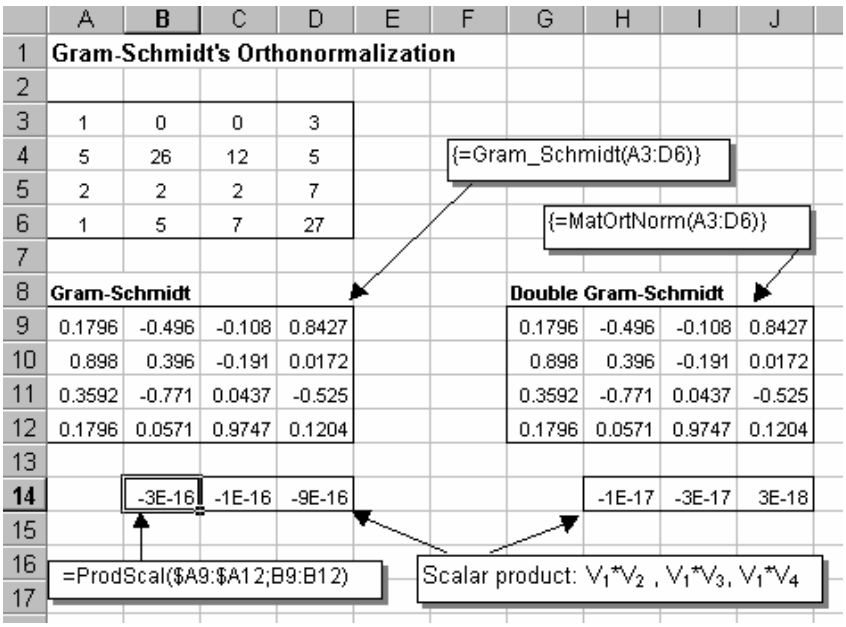
Function MatOrtNorm(A)

This function performs the orthogonalization with the double-step Gram-Schmidt algorithm
Argument **A** is a (n x n) matrix containing n independent vectors.
This function returns the orthogonal matrix U; each vector has norm = 1

$$U = (v_1, v_2, v_3 ..., v_n) \qquad \text{Where:} \qquad v_i \bullet v_j = \begin{cases} 1 & \Leftrightarrow i = j \\ 0 & \Leftrightarrow i \neq j \end{cases}$$

See also Function Gram_Schmidt(Mat)

Example - Perform the orthogonalization of the given matrix with the Gram-Schmidt methods (single e double step)



	A	B	C	D	E	F	G	H	I	J
1	**Gram-Schmidt's Orthonormalization**									
2										
3	1	0	0	3						
4	5	26	12	5						
5	2	2	2	7						
6	1	5	7	27						
7										
8	**Gram-Schmidt**						**Double Gram-Schmidt**			
9	0.1796	-0.496	-0.108	0.8427			0.1796	-0.496	-0.108	0.8427
10	0.898	0.396	-0.191	0.0172			0.898	0.396	-0.191	0.0172
11	0.3592	-0.771	0.0437	-0.525			0.3592	-0.771	0.0437	-0.525
12	0.1796	0.0571	0.9747	0.1204			0.1796	0.0571	0.9747	0.1204
13										
14		-3E-16	-1E-16	-9E-16				-1E-17	-3E-17	3E-18
15										
16										
17										

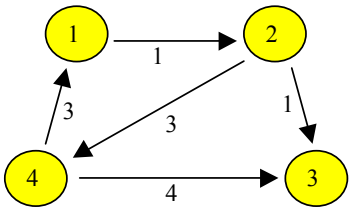
Under both the matrices we have computed the scalar product of the vectors in order to evaluate the round-off errors

As we can see double step method has errors about 10 times lower.

Function Path_Floyd(G)

This function, now available also in macro version, returns the matrix of all pairs shortest-path of a graph. This is an important problem in Graph-Theory and has applications in several different fields: transportation, electronics, space syntax analysis, etc.
The all-pairs shortest-path problem involves finding the shortest path between all pairs of vertices in a graph that can be represented as an adjacency matrix [G] in which each element aij - called node - represents the "distance" between element i and j. If there is not a link between two nodes, we leave the cell blank or we can set any not numeric symbol you like: for example "x"
This function uses the *Floyd's sequential algorithm*

Example. - A simple directed graph and its adjacency matrix G



	A	B	C	D	E	F	G	H	I	J
1	Adjacency matrix					All-pairs shortest path matrix				
2	0	1	x	x		0	1	2	4	
3	x	0	1	3		6	0	1	3	
4	x	x	0	x		x	x	0	x	
5	3	x	4	0		3	4	4	0	
6										
7	Shortest path =			1		{=Path_Floyd(A2:D5)}				
8	row =			1						
9	column =			2		{=Path_Min(A2:D5)}				
10										

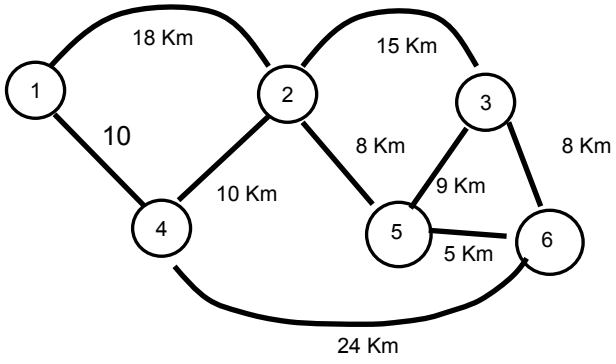
Function Path_Min(G)

Returns a vector contains the shortest path of a graph; the row and column of the cell
This function uses the Path_Floyd() function to find the all-pairs shortest path of the given graph G

Path_Min(G) => [path_min, i_min ; j_min]

Graphs theory recalls

Find the shortest distance between 6 sites drown in the following road map



The map can be reassumed into the following matrix

	city 1	city 2	city 3	city 4	city 5	city 6
city 1	0	18	x	10	x	x
city 2	18	0	15	10	8	x
city 3	x	15	0	x	9	8
city 4	10	10	x	0	x	24
city 5	x	8	9	x	0	5
city 6	x	x	8	24	5	0

In the cell 1,2 we fill the distance between city 1 and city 2 , that is 18 Km;
In the cell 1,3 we fill "x" because there is not a direct way between city 1 and city 3.
In he cell 1,4 we fill the distance between city 1 and city 4 , that is 10 Km.
And so on...

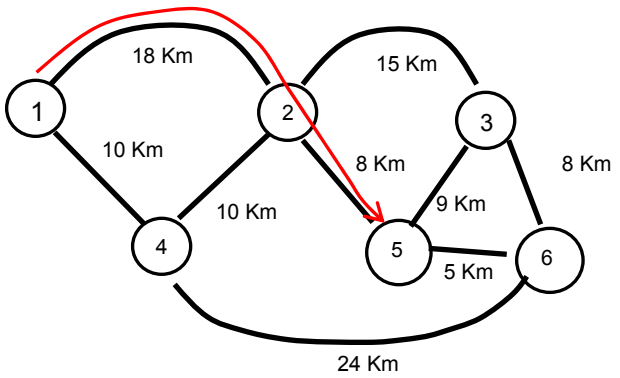
We observe that the matrix is symmetric because the distance d_{ij} is the same of d_{ji}; so we really have to compute only half matrix.
The above matrix - "adjacent matrix" - reports only the direct distance between each couple of city.
But we can join, for example, city 1 and city 3 in several different paths:
city 1 - city 2- city 3 = 18 + 15 = 33 Km
city 1 - city 4 - city 6 - city 3 = 10 + 24 + 8 = 42 Km etc.

The first one is the shortest distance path for city 1 and city 3
We can repeat this research for any other couple and find the shortest path for all couple of city. But it will tedious. The Floyd algorithm automates just this task. Applying this algorithm to the above matrix we get the following matrix

	city 1	city 2	city 3	city 4	city 5	city 6
city 1	0	18	33	10	26	31
city 2	18	0	15	10	8	13
city 3	33	15	0	25	9	8
city 4	10	10	25	0	18	23
city 5	26	8	9	18	0	5
city 6	31	13	8	23	5	0

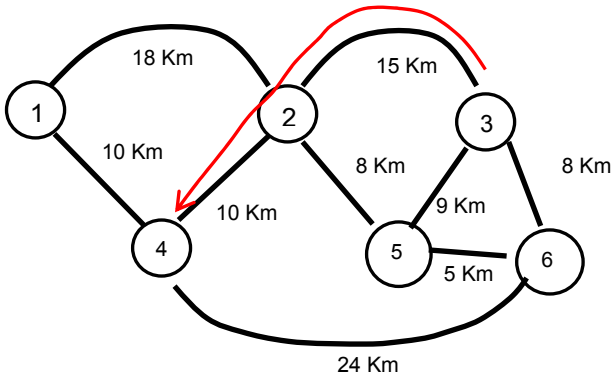
This matrix reports the shortest distance between each couple of city
For example the shortest distance between city 1 and city 5 is 26 Km

	city 1	city 2	city 3	city 4	city 5	city 6
city 1	0	18	33	10	26	31
city 2	18	0	15	10	8	13
city 3	33	15	0	25	9	8
city 4	10	10	25	0	18	23
city 5	26	8	9	18	0	5
city 6	31	13	8	23	5	0



For example the shortest distance between city 3 and city 4 is 25 Km

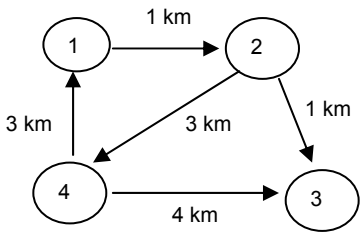
	city 1	city 2	city 3	city 4	city 5	city 6
city 1	0	18	33	10	26	31
city 2	18	0	15	10	8	13
city 3	33	15	0	25	9	8
city 4	10	10	25	0	18	23
city 5	26	8	9	18	0	5
city 6	31	13	8	23	5	0



As we can see to find the shortest paths is simple for a low set of nodes, but becomes quite heavy for larger set of nodes.

The thing are more difficult if the paths are "oriented"; for example if one or plus ways are only one direction

Let see this example



The adjacent matrix is built in the same way; the only different is that in this case is not symmetric.
For example between the node 1 and node 2 there is a direct path of 1 km, but it is not true the contrary

	node 1	node 2	node 3	node 4
node 1	0	1	x	x
node 2	x	0	1	3
node 3	x	x	0	x
node 4	3	x	4	0

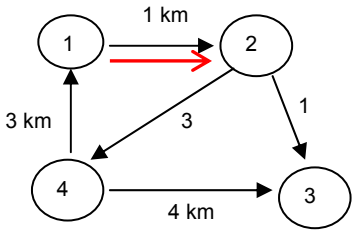
Applying the Floyd algorithm we get the following matrix

	node 1	node 2	node 3	node 4
node 1	0	1	2	4
node 2	6	0	1	3
node 3	x	x	0	x
node 4	3	4	4	0

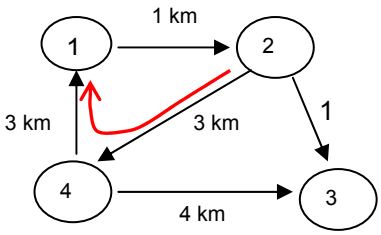
Reading this matrix is simple:

To go from node 1 to the node 2 there's the shortest path of 1 km; on the contrary, from node 2 to node 1 there's the shortest path of 6 km

	node 1	node 2	node 3	node 4
node 1	0	1	2	4
node 2	6	0	1	3
node 3	x	x	0	x
node 4	3	4	4	0



	node 1	node 2	node 3	node 4
node 1	0	1	2	4
node 2	6	0	1	3
node 3	x	x	0	x
node 4	3	4	4	0

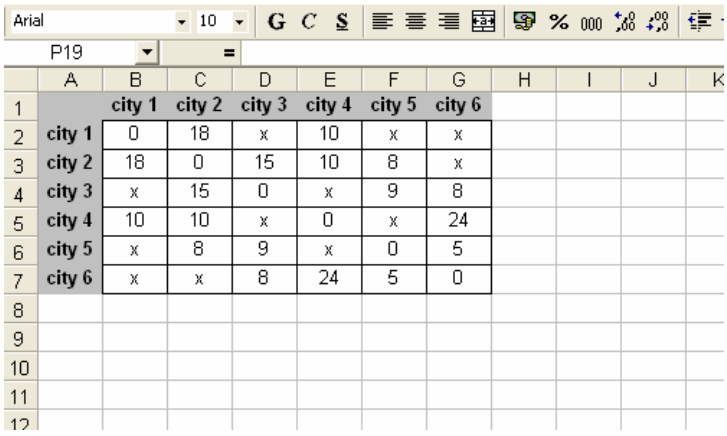


We note that from node 3 there is not any path to reach any other nodes. The row of node 3 has all "x" (means no path) except for itself. But it can be reached by all other nodes.

Let's see how use this array function in Excel

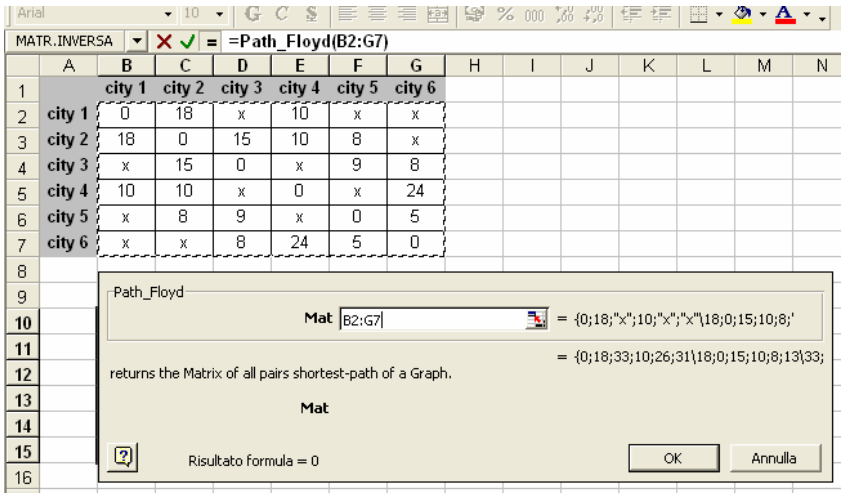
Shortest path

First of all, write the adjacent matrix (we have drown also columns and rows header but they are not indispensable)



	A	B	C	D	E	F	G
1		city 1	city 2	city 3	city 4	city 5	city 6
2	city 1	0	18	x	10	x	x
3	city 2	18	0	15	10	8	x
4	city 3	x	15	0	x	9	8
5	city 4	10	10	x	0	x	24
6	city 5	x	8	9	x	0	5
7	city 6	x	x	8	24	5	0

Now choose the site of you want to insert the shortest-path matrix; that is the matrix returned by function **Path_Floyd.** It must insert as an array function. That returns a 6 x 6 matrix
Example. Assume that you choose the area below the first matrix: select the area B10:G15 and now insert the function Path Floyd(). Now you must input the adjacent matrix; select the area B2:G7 of the first matrix

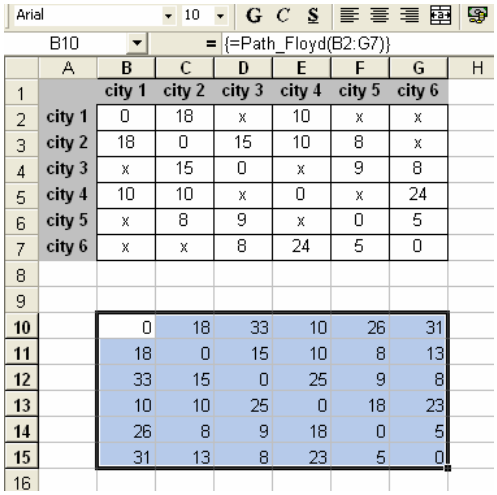


Now gives the keys sequence CTRL+SHIFT+ENTER
That is:

1. Press and keep down the CTRL and SHIFT keys
2. Press the ENTER key

All the solution's values fill all the cells that you have selected.
Note that Excel shows the function around two braces { }
These symbols mean that the function return an array.

The matrix returned is the shortest-path matrix



	A	B	C	D	E	F	G
1		city 1	city 2	city 3	city 4	city 5	city 6
2	city 1	0	18	x	10	x	x
3	city 2	18	0	15	10	8	x
4	city 3	x	15	0	x	9	8
5	city 4	10	10	x	0	x	24
6	city 5	x	8	9	x	0	5
7	city 6	x	x	8	24	5	0
10		0	18	33	10	26	31
11		18	0	15	10	8	13
12		33	15	0	25	9	8
13		10	10	25	0	18	23
14		26	8	9	18	0	5
15		31	13	8	23	5	0

Function SVD - Singular Value Decomposition

Function SVD_U(A)
Function SVD_D(A)
Function SVD_V(A)

Singular Value Decomposition of a (n x m) matrix **A** provides 3 matrices, **U**, **D**, **V** performing the following decomposition[10]:

$$A = U \cdot D \cdot V^T$$

Where: p = min(n, m)

U is an orthogonal matrix (n x p)
D is a square diagonal matrix (p x p)
V is an orthogonal matrix (m x p)

Each of the above functions returns one of the SVD matrices.

For example

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{6}}{3} & 0 \\ \frac{\sqrt{6}}{6} & \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{6}}{6} & \frac{\sqrt{2}}{2} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{-\sqrt{2}}{2} \end{bmatrix}^T$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{-\sqrt{2}}{2} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{\sqrt{6}}{3} & 0 \\ \frac{\sqrt{6}}{6} & \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{6}}{6} & \frac{\sqrt{2}}{2} \end{bmatrix}^T$$

$$\begin{bmatrix} \sqrt{3} & 1 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{6} & 0 \\ 0 & \sqrt{2} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{2} & \frac{-\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix}^T$$

Example. Find the SVD decomposition for the given matrix

	A	B	C	D	E	F	G	H	I	J	K	L
1		**A**			**U**			**D**			**V**	
2	1	8	2	-0.878	-0.379	-0.293	9.361	0	0	-0.234	0.772	-0.59
3	3	4	-1	-0.47	0.801	0.372	0	3.219	0	-0.961	-0.092	0.261
4	-1	1	1	-0.093	-0.464	0.881	0	0	0.1	-0.147	-0.628	-0.764
5				{=SVD_U(A2:C4)}			{=SVD_D(A2:C4)}			{=SVD_V(A2:C4)}		

From the **D** matrix of singular values we get the max and min values to compute the condition number m [11] **,** used to measure the ill-conditioning of a matrix. In fact, we have:

m = 9.361 / 0.1 = 93.61

The SVD decomposition of a square matrix return always square matrices of the same size, but for a rectangular matrix we should take a bit more attention to the correct dimensions.
Let's see this example

[10] Some authors give a different definition for SVD decomposition, but the main concept is the same.
[11] Respect to the norm 2

	A	B	C	D	E	F	G	H	I	J	K	L
1						U		D		V		
2		2	4			0.139	0.99	5.855	0	-0.8	0.602	
3		5	-3			-0.99	0.139	0	4.441	0.602	0.798	
4												
5		1	8			-0.86	-0.4	9.264	0	-0.25	0.969	
6		3	4			-0.5	0.774	0	2.485	-0.97	-0.25	
7		-1	1			-0.08	-0.49					
8												
9		1	8	2		-0.88	-0.48	9.321	0	-0.25	0.759	
10		3	4	-1		-0.48	0.879	0	2.849	-0.96	-0.1	
11										-0.14	-0.64	
12												

Sometime it happens that the matrix is singular or “near singular”. The SVD decomposition evidences this fact and allows computing the matrix rank in a very fast way. You have to count the singular values greater than zero (or a small value, usually 1E-13). For this scope we need only the singular values matrix returned by the function SVD_D(). Let’s see.

H2 fx {=SVD_D(B2:F6)}

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2		1	10	7	9	-2		121	0	0	0	0
3		4	37	25	36	-7		0	1.68	0	0	0
4		-8	-64	-43	-62	12		0	0	0.55	0	0
5		-2	-20	-14	-18	4		0	0	0	0	0
6		-1	-10	-7	-9	2		0	0	0	0	0
7												

In this example the true rank of the given (5x5) matrix is only 3, because there are only 3 singular value different from zero.

Denomination. The matrix returned by the SVD are usually called:

U (hanger), D (stretcher), V (aligner).

So the decomposition for a matrix A can be written[12]

(any matrix) = (hanger) x (stretcher) x (aligner)

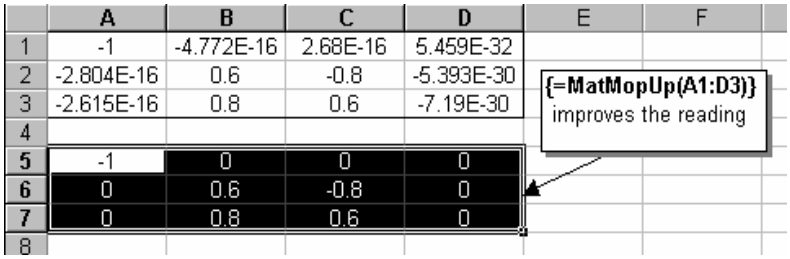
[12] For further details see the “Introduction to the Singular Value Decomposition” by Todd Will, UW-La Crosse, Wisconsin, 1999 and “Matrices Geometry & Mathematic” by Bill Davis and Jerry Uhl

Function MatMopUp(M, [ErrMin])

This function eliminates all round-off errors from a matrix. Each element that is absolute less than *ErrMin* is substituted by zero.

$$a_{ij} = \begin{cases} 0 \Rightarrow |a_{ij}| < \text{ErrMin} \\ 0 \Rightarrow \text{otherwise} \end{cases}$$

Parameter *ErrMin* is optional (default *ErrMin* = 1E-15)



	A	B	C	D	E	F
1	-1	-4.772E-16	2.68E-16	5.459E-32		
2	-2.804E-16	0.6	-0.8	-5.393E-30		
3	-2.615E-16	0.8	0.6	-7.19E-30		
4						
5	-1	0	0	0		
6	0	0.6	-0.8	0		
7	0	0.8	0.6	0		
8						

Function MatCovar(A)

Returns the covariance matrix (m x m) of a given matrix (n x m)
The (column) covariance is definite by the following formulas:

$$c_{ij} = \frac{1}{n}\sum_{k=1}^{n}(a_{ki} - \overline{a}_i)(a_{kj} - \overline{a}_j) \quad , \quad \text{for} \quad i = 1,..\text{m} \ , \ j = 1,..\text{m}$$

Where $\overline{a}_j = \frac{1}{n}\sum_{k=1}^{n} a_{kj}$, for $j = 1,..\text{m}$

See also the matrix correlation function MatCorr()
This function is similar to COVAR built-in function for two variables.

Function MatCorr(A)

Returns the correlation matrix (m x m) of a given matrix (n x m)
The correlation matrix is definite by the following formulas:

$$c_{ij} = \frac{\frac{1}{n}\sum_{k=1}^{n}(a_{ki} - \overline{a}_i)(a_{kj} - \overline{a}_j)}{\sigma_i \cdot \sigma_j} \qquad \text{, for } i = 1..m \ , \ j = 1...m$$

Where:

$$\overline{a}_i = \frac{1}{n}\sum_{k=1}^{n} a_{ki} \qquad \text{. for } i = 1 \ ...m$$

$$\sigma_i = \frac{1}{n}\sqrt{\sum_{k=1}^{n}(a_{ki} - \overline{a}_i)} \qquad \text{, for } i = 1 \ ...m$$

Note. Correlation matrix has always diagonal =1
See also the matrix covariance function MatCovar

Example - find the covariance and the correlation matrix for the following data table:

x1	7	4	6	8	8	7	5	9	7	8
x2	4	1	3	6	5	2	3	5	4	2
x3	3	8	5	1	7	9	3	8	5	2

There are three variables x_1, x_2, x_3 and 10 data observations. The matrix will be 3 x 3.
In the first columns A, B, C we have arranged the row data (orientation is not important).
In the last row we have calculate the statistics: average $\bar{x}_i$ and standard deviation σ_{xi} for each column.

In the column D, E, F we have calculated the normalized data; that is the data with average = 0 and standard dev. = 1.

We have calculated each column $\mathbf{u}_i$ with the following simple formulas: $u_{ij} = \dfrac{x_{ij} - \bar{x}_i}{\sigma_{xi}}$

	A	B	C	D	E	F	G	H	I	J	K
1	**row data**			**normalized data**				**covariance (row)**			
2	**x1**	**x2**	**x3**	**u1**	**u2**	**u3**		2.09	1.45	-0.39	
3	7	4	3	0.0692	0.3333	-0.789		1.45	2.25	-1.15	
4	4	1	8	-2.006	-1.667	1.0891		-0.39	-1.15	7.09	
5	6	3	5	-0.623	-0.333	-0.038		{=MatCovar(A3:C12)}			
6	8	6	1	0.7609	1.6667	-1.54		**covariance (norm.)**			
7	8	5	7	0.7609	1	0.7136		1	0.669	-0.1	
8	7	2	9	0.0692	-1	1.4647		0.669	1	-0.29	
9	5	3	3	-1.314	-0.333	-0.789		-0.1	-0.29	1	
10	9	5	8	1.4526	1	1.0891		{=MatCovar(D3:F12)}			
11	7	4	5	0.0692	0.3333	-0.038		**correlation (row)**			
12	8	2	2	0.7609	-1	-1.164		1	0.669	-0.1	
13								0.669	1	-0.29	
14	6.9	3.5	5.1	-3E-16	0	1E-16	= AVERAGE	-0.1	-0.29	1	
15	1.446	1.5	2.663	1	1	1	= STDEVP	{=MatCorr(A3:C12)}			
16											

At the right we have calculated the covariance matrices for both row and normalized data: They are always symmetric.
At the right-bottom side we have calculated the correlation matrix for the row data; we note that the correlation matrix of row data and the covariance matrix of normalized data are identical. That is:

Covariance(Normalized data) ≡ *Correlation(Row data)*

The function MatCorr() is useful to get the correlation matrix without performing the normalization process of the given data.

Correlation is a very powerful technique to detect hidden relations between numeric variables

Example - In an experimental test it was measured the oxygen respired by 10 persons. Of each of them it was taken his age and his weight. We want to discover if the respired oxygen depends by age or by weight or by both. The test data are in the following table

Age	44	38	40	44	44	42	47	43	38	45
Weight	85.84	89.02	75.98	81.42	73.03	68.15	77.45	81.19	81.87	87.66
Oxygen	120.94	129.47	116.55	122.92	118.57	114.73	125.37	119.20	127.10	127.52

	A	B	C	D	E	F	G
1	Age	Weight	Oxygen				
2	44	85.84	120.94		correlation matrix 3x3		
3	38	89.02	129.47		1	-0.14	-0.13
4	40	75.98	116.55		-0.14	1	0.78
5	44	81.42	122.92		-0.13	0.78	1
6	44	73.03	118.57		{=MatCorr(A2:C11)}		
7	42	68.15	114.73				
8	47	77.45	125.37				
9	43	81.19	119.20				
10	38	81.87	127.10				
11	45	87.66	127.52				
12							

In the correlation matrix we discover a relative high level of correlation for
$a_{23} = a_{32} = 0.78$ (the max is 1)

This means that between variable 2 and 3 there is a tight relation.
On the contrary, we note a weak dependence between variable 1 and 3

Function REGRL(Y, X, [ZeroIntcpt])

Computes the multivariate linear regression with the SVD method.
Parameter Y is the (n x 1) vector of the dependent variable.
Parameter X is a list of independent variable. It may be a (n x 1) vector for monovariable regression or a (n x m) matrix for multivariate regression.
Parameter ZeroIntcpt, if present, forces the Y intercept to zero: Y(0)= 0
The function returns the (m+1) coefficients of the linear regression. For monovariate regression, it returns two coefficients [a0, a1]; the first one is the intercept of Y-axis, the second one is the slope.

Example - find the linear best fit for the following data table

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14
y	5.12	6.61	8.55	10.07	11.35	12.47	13.48	14.41	15.27	16.07	16.82	17.54	18.22	18.86

The linear model for one independent variable is: $y = a_0 + a_1 x$

	A	B	C	D	E	F	G
1	Least Squares Linear Regression.						
2	x	y (samples)	y^ (best fit)		n	a0	a1
3	1	5.12	6.53		14	5.4989	1.0272
4	2	6.61	7.55				
5	3	8.55	8.58		{=REGRL(B3:B16,A3:A16)}		
6	4	10.07	9.61				
7	5	11.35	10.63				
8	6	12.47	11.66		var(y)	var(y^)	r2
9	7	13.48	12.69		17.663	17.145	0.9707
10	8	14.41	13.72				
11	9	15.27	14.74		=VARP(C3:C16)		=F6/E6
12	10	16.07	15.77				
13	11	16.82	16.80		=VARP(B3:B16)		
14	12	17.54	17.82				
15	13	18.22	18.85				
16	14	18.86	19.88		=$F$3+$G$3*A16		
17							

The coefficient r^2, between 0 and 1, measures how good is the fit (0 bad - 1 perfect)
In Matrix.xla there is not a specific function for that, but it can be easily computed by the standard statistic functions and using the formula

$$r^2 = \frac{\sum(\hat{y} - \bar{\hat{y}})^2}{\sum(y - \bar{y})^2} = \frac{\mathrm{var}(\hat{y})}{\mathrm{var}(y)}$$

Function REGRP(Degree, Y, X, [ZeroIntcpt])

Computes the polynomial regression f(x) of a dataset of points $[x_i, y_i]$.
Parameter Degree set the degree of the polynomial.
Parameter Y is a (n x 1) vector of the dependent variable.
Parameter X is a (n x1) vector of the independent variable.
Parameter ZeroIntcpt, if present, forces the intercept to zero: f(0)= 0
The function returns the coefficients of the polynomial regression $[a_0, a_1, a_2, ...a_m]$.
where m is the degree of the regression model

$$f(x) = a_0 + a_1 x + a_2 x^2 + ...a_m x^m$$

Example: Given a table of (x, y) points, find the 6th degree polynomial approximating the given data

	A	B	C	D	E	F
1	x	y		polynomial coefficents		
2	0	134		a0	134	
3	0.2	153.355584		a1	97	
4	0.5	182.296875		a2	-1	
5	0.8	211.857024		a3	-1	
6	1	233		a4	2	
7	1.5	302.984375		a5	1	
8	2	444		a6	1	
9	2.5	774.546875				
10	3	1523				
11	3.5	3081.984375	{=REGRP(6,B2:B12,A2:A12)}			
12	4	6074				
13						

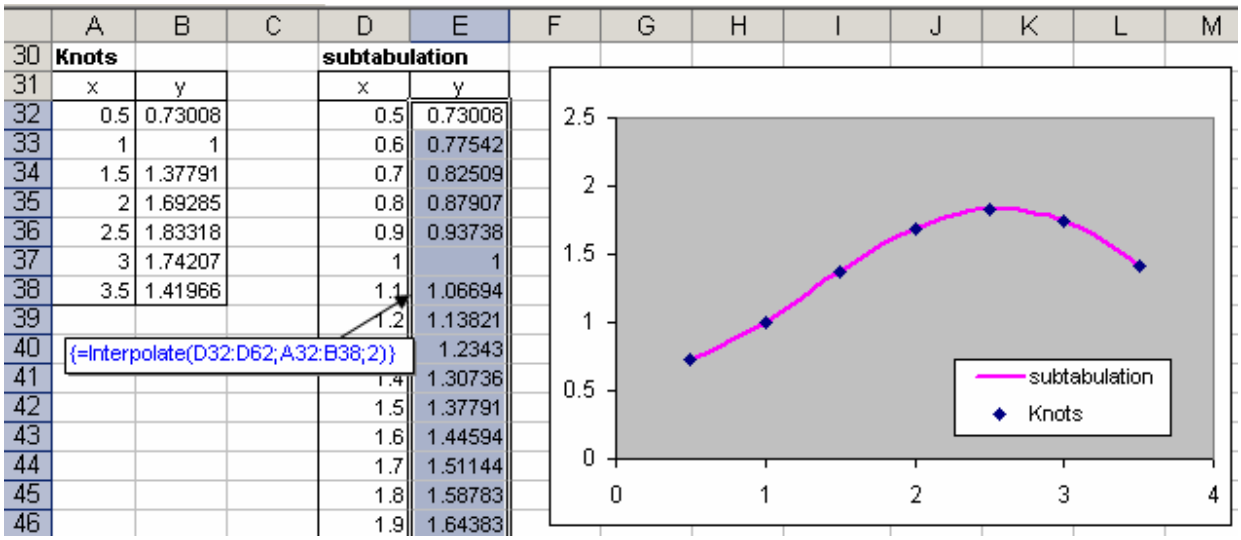
Function Interpolate(x, Knots, [Degree], [Points])

It returns the polynomial interpolation of a set of data points [xi, f(xi)].
Parameter x is the point to interpolate. It can be also a vector of interpolation points.
Parameter knots is a matrix (n x 2) of data point [xi, f(xi)].
Parameter degree, set the degree of interpolation polynomial (default degree = 2)
Parameter points, set the number of points subset for the polynomial regression. If omitted, the function assumes: points = degree +1

This function uses the *piecewise interpolation method*. Interpolation is exact if the number of points is equal to degree +1. Interpolation is approximated with the Least Squares if the number of points is greater then degree +1

The function returns interpolation value y at x point. If x is a vector, the function returns a vector. In this case you must insert the function with CTRL+SHIFT+ENTER sequence

Example. Perform the sub tabulation with step = 0.1 of a given table with 7 knots, using the parabolic interpolation (polynomial degree = 2)



This function can be used also for “data smoothing”. This problem is common when the points are derived from experimental measurements. See chapter “Interpolate” of Vol. 1.

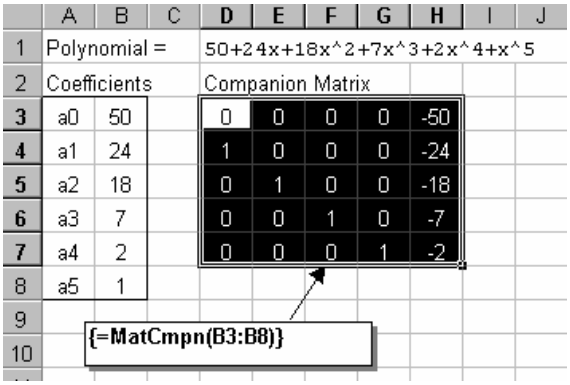
Function MatCmp(Coeff)

Returns the companion matrix of a monic polynomial, defined as:

$$a(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} + x^n$$

$$A = \begin{bmatrix} 0 & 0 & \cdots & 0 & -a_0 \\ 1 & 0 & \cdots & 0 & -a_1 \\ 0 & 1 & \cdots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & -a_{n-1} \end{bmatrix}$$

Parameter Coeff is the complete coefficients vector. If $a_n \neq 1$, the coefficients are normalized before generating the companion matrix
Example:



Function MatCplx([Ar], [Ai], [Cformat])

Converts 2 real matrices into a complex matrix
Ar is the (n x m) real part and Ai is the (n x m) imaginary part
The real or imaginary part can be omitted. The function assumes the zero-matrix for the missing part.
Example

A pure real matrix can be written into

`MatCplx(Ar)` = $[A_r]+j[0]$

A pure imaginary matrix can be written as

`MatCplx(, Ai)` = $[0]+j[A_i]$ (remember the comma before the 2nd argument)

This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
Optional parameter *Cformat* sets the complex format of input/output (default = 1)

Use CTRL+SHIFT+ENTER to insert this function

This function is useful for passing a real matrix to a complex matrix function, such as, for example the M_MULT_C. If we have to multiply a real matrix for a complex vector, we can use the M_MULT_C function; but, because this function accepts complex matrices, we have to convert the matrix **A** into a complex one (with a null imaginary part) by the MatCplx function and then pass the result to the M_MULT_C. In other words we have simply to nest the two functions like that

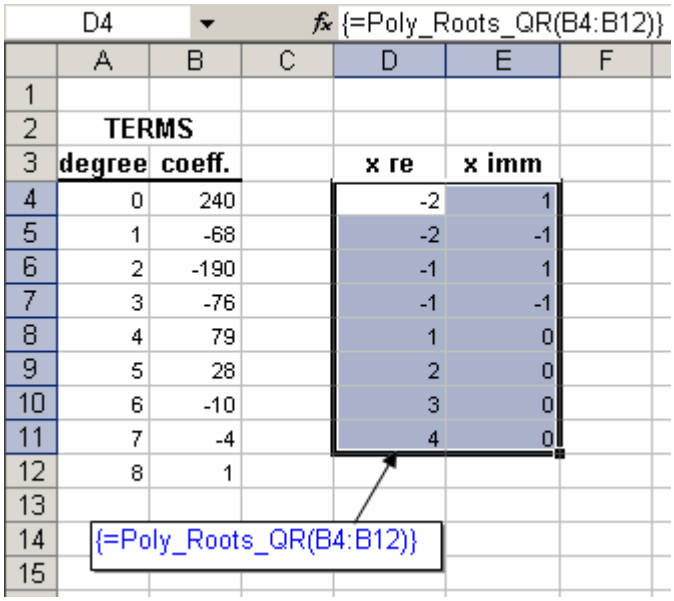
	A	B	C	D	E	F	G	H	I	J
1		A				u			A * u	
2	2	-1	3	-4		2	0		7	1
3	14	11	-7	2		0	-1		21	-11
4	-6	-3	11	2		1	0		-1	3
5	4	3	1	2		0	0		9	-3
6										
7	{=M_MULT_C(MatCplx(A2:D5),F2:G5)}									

Function Poly_Roots_QR(Coefficients)

This function returns all roots of a given polynomial
Parameter Coefficients is a (n+1 x 1) vector, where n is the degree of the polynomial
This function uses the *QR algorithm*. The process consists of finding the eigenvalues of a companion matrix with the given polynomial coefficients.
This process is very fast, robust and stable but may not be converging under certain conditions. If the function cannot find a root it returns "?". Usually it suitable to solve polynomial up to 10^{th} degree with a good accuracy (1E-9 – 1E-12)

Example: Find all roots of the following polynomial of 8 degree

$P(x) = 240 - 68x - 190x^2 - 76x^3 + 79x^4 + 28x^5 - 10x^6 - 4x^7 + x^8$



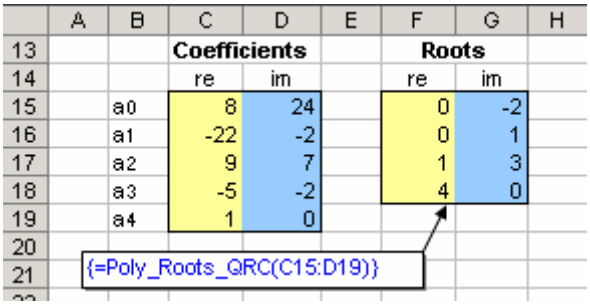
As we can see the polynomial has four real and four complex conjugate roots

Function Poly_Roots_QRC(Coefficients)

This function returns all roots of a given complex polynomial
Parameter Coefficients is a (n+1 x 2) array, where n is the degree of the polynomial
If the function cannot find a root it returns "?". Usually this function is suitable to solve polynomial up to 10 degree
This function uses the QR algorithm. The process consists of applying iteratively the QR decomposition to the complex companion matrix.

Example. Find all the roots of the following polynomial

$$P(z) = z^4 - (5+2j)z^3 + (9+7j)z^2 - (22+2j)z + 8 + 24j$$



Function MatRot(n, teta, p, q)

Returns the orthogonal matrix (n x n) that performs the planar rotation over the plane defined by axis p and q
Parameter *teta* sets the angle of rotation in radiant
Parameter *p* and *q* are the columns of the rotation and must be: p <> q and p ≤ n and p ≤ n

Example: In the 3D space, the canonical rotation matrices are

$$p=1, q=2 \qquad p=1, q=3 \qquad p=2, q=3$$

$$\begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} c & 0 & -s \\ 0 & 1 & 0 \\ s & 0 & c \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & 1 \end{bmatrix}$$

Where:

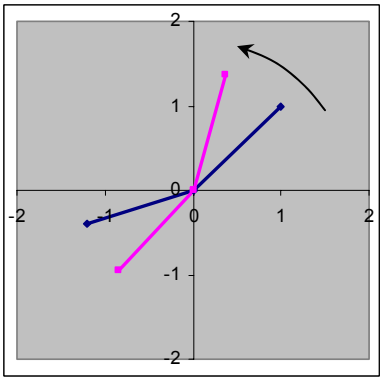
$$c=\cos(\theta)\ ,\ s=\sin(\theta)$$

Note that all rotation matrices have determinant = 1

Example. Given two vectors in $\mathbf{R}^2$ ($\mathbf{v}_1$, $\mathbf{v}_2$) , found the same vectors after a rotation of 30° deg

The transformation formula is:

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{12} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{12} \end{bmatrix}$$



That can be arranged in the following way:

	A	B	C	D	E	F	G	H	I
1	v1	v2		Rotation matrix			w1	w2	
2	1	-1.2		0.866	-0.5		0.366	-0.839	
3	1	-0.4		0.5	0.866		1.366	-0.946	
4									
5				Deg	Rad				
6	{=MatRot(2;E6;1;2)}			30	0.524		{=M_PROD(D2:E3;A2:B3)}		

Conditioned Number

This number is conventionally used to indicate how a matrix is ill-conditioned
Formally the conditioned number of a matrix is defined by the SVD decomposition as the ratio of the largest (in magnitude) element to the smallest element of diagonal matrix
Given the SVD decomposition:

$$A = UDV^T \qquad \text{where:} \qquad D = diag[d_{11}, d_{22}, d_{33}, \ldots d_{nn}]$$

The conditioned number is $c = \dfrac{|d_{ii}|_{\max}}{|d_{jj}|_{\min}}$

A matrix is ill-conditioned if its conditioned number is very large. For a 32bit double precision arithmetic this number is about 1E12. In this package there is not a specific function that returns this number, but it can be easily calculate by the D matrix returned by the function SVD_D

Matrice Hilbert				SVD_D			
1	1/2	1/3	1/4	1.5002	0	0	0
1/2	1/3	1/4	1/5	0	0.1691	0	0
1/3	1/4	1/5	1/6	0	0	0.0067	0
1/4	1/5	1/6	1/7	0	0	0	1E-04

Conditioned number = 15514

Function VarimaxRot(FL, [Normal], [MaxErr], [MaxIter])

This function computes the orthogonal rotation for a Factors Loading matrix using the Kaiser's Varimax method for 2D and 3D factors
Parameter **FL** is the Factor Loading matrix to rotate (n x m). The number of factor m, at this release, must be only 2 or 3.
Optional parameter **Normal** = True/False chooses the "Varimax normalized criterion". That is, indicates if the matrix of loading is to be row normalized before rotation (default = False)
Optional parameter MaxErr set sets the accuracy required (default = 10^-4). The algorithm stops when the absolute difference of two consecutive Varimax values is less of MaxErr
Optional parameter **MaxIter** sets the maximum number of iterations allowed (default=500)

Algorithm
The Varimax rotation procedure was first proposed by Kaiser (1958). Given a *numberOfPoints* × *numberOfDimensions* configuration **A**, the procedure tries to find an orthonormal rotation matrix **T** such that the sum of variances of the columns of **B***B is a maximum, where **B** = **AT** and * is the element wise (Hadamard) product of matrices. A direct solution for the optimal **T** is not available, except for the case when *numberOfDimensions* equals two. Kaiser suggested an iterative algorithm based on planar rotations, i.e., alternate rotations of all pairs of columns of **A**.
For Varimax criterion definition see Varimax Index

This function is diffused, by now, in the almost principal (and expensive) statistics tools, because, on the contrary, it is very rare in freeware software, we have added to our add-in.
Let's see how it works with one popular example

Example 2D- Initial Factors Matrix

	Factor 1	Factor 2
Services	0.879	-0.158
HouseValue	0.742	-0.578
Employment	0.714	0.679
School	0.713	-0.555
Population	0.625	0.766

The goal of the method is to try to maximize one factor for each variable. This will make evident which factor is dominant (more important) for each variable.

Rotate Factors Matrix: method Varimax

	A	B	C	D	E	F
1		Factor 1	Factor 2		Factor 1	Factor 2
2	Services	0.879	-0.158		0.788	0.420
3	HouseValue	0.742	-0.578		0.941	0.005
4	Employment	0.714	0.679		0.141	0.975
5	School	0.713	-0.555		0.904	0.005
6	Population	0.625	0.766		0.017	0.988
7						
8			{=VarimaxRot(B2:C6)}			
9						
10	Varimax Index	0.36			2.08	
11		=VarimaxIndex(B2:C6)			=VarimaxIndex(E2:F6)	

As we can see the varimax index is incremented after the varimax rotation method. Each variable has maximized or minimized its factors values

Function VarimaxIndex(Mat, [Normal])

Returns the Varimax value of a given Factor matrix **Mat**
Varimax is a popular criterion Kaiser (1958) to perform orthogonal rotation of Factors Loading matrices.
Usually, the rotation stops when Varimax is maximized
Optional parameter Normal = True/False indicates if the matrix is to be row normalized before computing

Formula
Varimax, for a matrix **A** with p row and k column (p x k) is defined as following (*):

$$V = \sum_{j=1}^{k}\sum_{i=1}^{p}(a_{ij})^4 - \frac{1}{p}\sum_{j=1}^{k}\left(\sum_{i=1}^{p}(a_{ij})^2\right)^2$$

Where:

$$b_{ij} = \begin{cases} a_{ij} & \Rightarrow normal = false \\ \dfrac{a_{ij}}{|\overline{a}_i|} & \Rightarrow normal = true \end{cases}$$

Function MatNormalize(Mat, [NormType], [Tiny])

Returns the normalized vectors of a real (n x m) matrix.
The optional parameter *Normtype* indicate what normalization is performed
The optional parameter *Tiny* set the minimum error level (default 2E-14)

$u_i = \dfrac{v_i}{\lvert v_{\min} \rvert}$	Normtype = 1. All vector's components are scaled to the min of the absolute values
$u_i = \dfrac{v_i}{\lvert v \rvert}$	Normtype = 2 (default). All vectors are length = 1
$u_i = \dfrac{v_i}{\lvert v_{\max} \rvert}$	Normtype = 3. All vector's components are scaled to the max of the absolute values

Example - Normalize the following 3x3 matrix

			Normalized	1		Normalized	2		Normalized	3	
2	4	-12	1	4	-12	0.37	0.87	-0.6	0.4	1	0.8
0	2	-15	0	2	-15	0	0.44	-0.75	0	0.5	1
5	1	6	2.5	1	6	0.93	0.22	0.3	1	0.25	-0.4

Function MatNormalize_C(Mat, [NormType], [Cformat], [Tiny])

Returns the normalized vectors of a complex (n x m) matrix.
This function supports 3 different complex formats: 1 = split, 2 = interlaced, 3 = string
The optional parameter *Cformat* sets the complex format of input/output (default = 1)

Example - Normalize the following complex vector

	A	B	C	D	E	F	G	H	I	J	K	
11	vector			Normalize (min)		1	Normalize (mod.)		2	Normalize (max)		3
12	2.2238	-0.234		1	0		0.4448	-0.047		0	-0.5	
13	0	0		0	0		0	0		0	0	
14	0.4682	4.4476		0	2		0.0936	0.8895		1	0	

Function MatNorm(v, [NORM])

Return the norm of a matrix or vector
Parameter v can be a vector or a matrix; optional parameter *Norm* sets the specific norm to compute (default 2 for vectors, and 0 for matrices)

The norm returned can be:

For vectors

Norm = 1	Absolute sum	$\|v\|_1 = \sum_i \| v_i \|$
Norm = 2	Euclidean norm	$\|v\|_2 = \sqrt{\sum_i v_i^2}$
Norm = 3 (also infinite)	Maximum absolute	$\|v\|_3 = \max_i(\| v_i \|)$

For matrices

Norm = 0	Frobenius norm	$\|A\|_0 = \sqrt{\sum_i \sum_j (a_{ij})^2}$
Norm = 1	Maximum absolute column sum	$\|A\|_1 = \max_j \left(\sum_i \| a_{ij} \| \right)$
Norm = 2	Euclidean norm	$\|A\|_2 = \sqrt{\rho\left(A^T A\right)}$
Norm = 3 (also infinite)	Maximum absolute row sum	$\|A\|_3 = \max_i \left(\sum_j \| a_{ij} \| \right)$

Note: the norm 2 for vectors and norm 0 for matrices give the same values of M_ABS function

Example: Find norm 0, 1, 2, 3 for the given 4x3 matrix

	A	B	C	D	E	F	G	H	I	J
1		A			type	Norm				
2	6	2	3		0	22.113	=MatNorm($A$2:$C$4,E2)			
3	-7	-9	5		1	29	=MatNorm($A$2:$C$4,E3)			
4	9	-7	-9		2	16.833	=MatNorm($A$2:$C$4,E4)			
5	7	-5	0		3	25	=MatNorm($A$2:$C$4,E5)			

Function M_MULT_C(M1, M2, [Cformat])

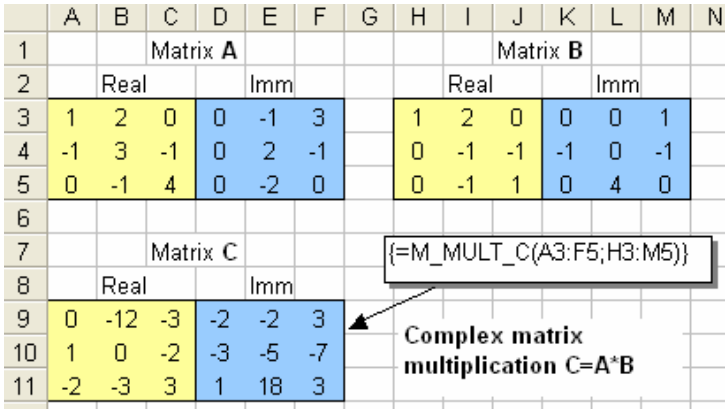
Performs the complex matrix multiplication.
If the dimension of the matrix M_1 is (n x m) and M_2 is (m x p) , then the product is a matrix (n x p)
This function now supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
Optional parameter *Cformat* sets the complex format of input/output (default = 1)

M_1 = A + j **B** **M_2 = C** + j **D**

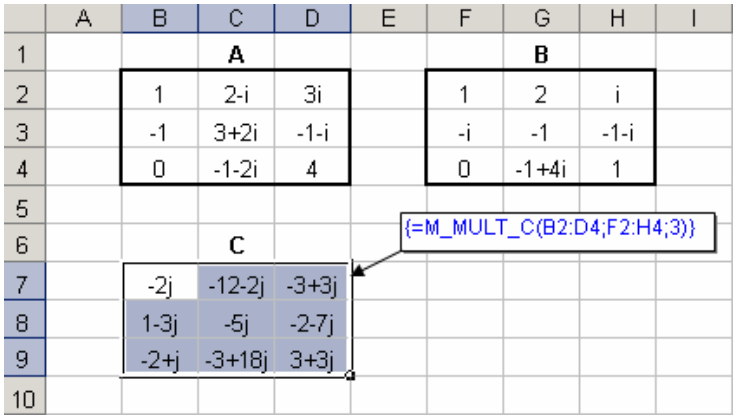
where **A**, **B**, **C**, **D** are real matrices

Example:

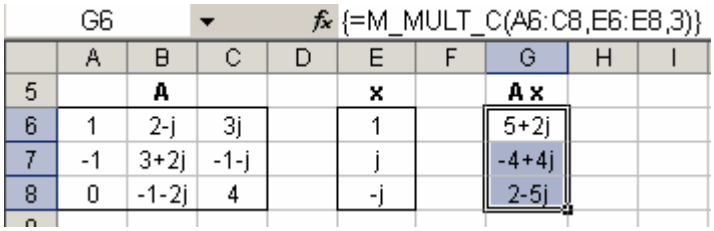
$$\mathbf{C} = \begin{bmatrix} 1 & 2-i & 3i \\ -1 & 3+2i & -1-i \\ 0 & -1-2i & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & i \\ -i & -1 & -1-i \\ 0 & -1+4i & 1 \end{bmatrix}$$



Note that the imaginary parts of matrices must be always inserted even if they are all 0 (real matrices).



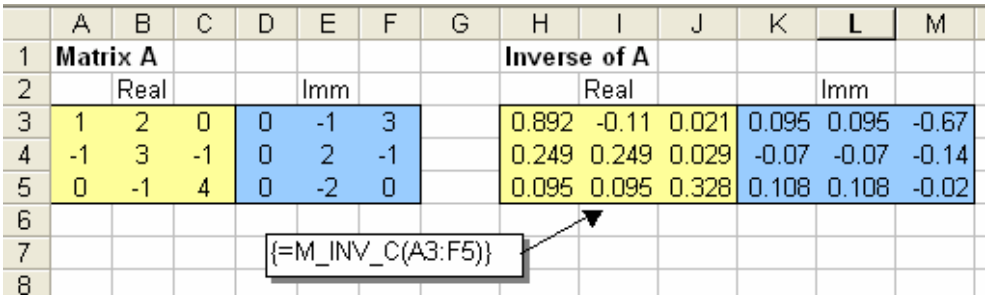
The calculus can be also performed in the handy string rectangular format. We have only to set the *Cformat* parameter to 3



Of course, this function can be used for multiplying a matrix and a vector

Function M_INV_C(A, [Cformat])

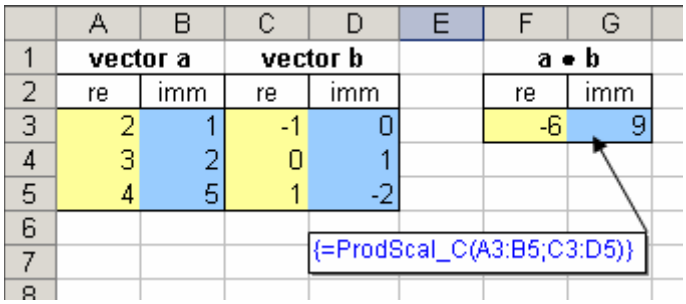
Complex matrix inversion
The complex matrix A must be square
This function supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
Optional parameter *Cformat* sets the complex format of input/output (default = 1)
Complex (split or interlaced) matrix must have always an even number of columns



Function ProdScal_C(v1, v2,)

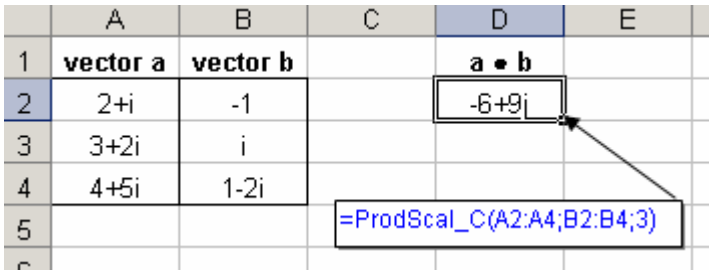
Returns the scalar product of two complex vectors

$$\vec{a} \bullet \vec{b} = \sum_k \left(a_{re} + i a_{im}\right)_k \cdot \left(b_{re} - i b_{im}\right)_k$$



Note that the imaginary parts of vectors must be always inserted even if they are all 0 (real matrices).

In string format we can write complex numbers as string "a+ib".
Look at the same example. Note the third optional parameter cformat = 3



Function SYSLIN_C(A, b, [Cformat])

This function solves a complex linear system by the Gauss-Jordan algorithm.
Returns the vector solution of the given system
This function now supports 3 different formats: 1 = split, 2 = interlaced, 3 = string
Optional parameter *Cformat* sets the complex format of input/output (default = 1)
Remember that for a linear system:

$$Ax = b$$

A is the system complex square matrix (n x n)
x is the unknown complex vector (n x 1)
b is the constant complex vector (n x 1)

As known, the above linear equation has only one solution if - and only if -, Det(**A**) <> 0

Example - solve the following complex 3x3 linear system

$$\begin{cases} x_1 + (2-i)\,x_2 + 3i\,x_3 = 5+10i \\ -x_1 + (3+2i)\,x_2 - (1+i)\,x_3 = -11-i \\ -(1+2i)\,x_2 + 4\,x_3 = 11-3i \end{cases} \qquad \begin{bmatrix} 1 & 2-i & 3i \\ -1 & 3+2i & -1-i \\ 0 & -1-2i & 4 \end{bmatrix} \cdot \mathbf{x} = \begin{bmatrix} 5+10i \\ -11-i \\ 11-3i \end{bmatrix}$$

	A	B	C	D	E	F	G	H	I	J	K	L
1	**Complex Linear System Ax=b**											
2			Matrix **A**					Vector **B**			Vector **X**	
3		Real			Imm			Real	Imm		Real	Imm
4	1	2	0	0	-1	3		5	10		3	1
5	-1	3	-1	0	2	-1		-11	-1		-1	1
6	0	-1	4	0	-2	0		11	-3		2	-1
7												
8				{=SYSLIN_C(A4:F6;H4:I6)}								
9												

We can also use directly the complex string format “a+bj”, Simply set the parameter *cformat* = 3

	A	B	C	D	E	F	G	H
1								
2	1	2-i	3i		5+10i		3+j	
3	-1	3+2i	-1-i		-11-i		-1+j	
4	0	-1-2i	4		11-3i		2-j	
5								
6		{=SYSLIN_C(A2:C4;E2:E4;3)}						
7								

Function Simplex(Funct, Constrain, [Opt])

This function perform the linear optimization with the Simplex method
Funct is the array (1 x n) containing the coefficients of the linear function to optimize
Constrain is the array (m x n+2) containing the coefficients of m linear constrain and the type of constrain (“<” , “>”, “=”)
Opt set the optimization type: 1 (default) for maximization, 0 for minimization.

A typical linear programming problem – also called linear optimization – is the following.

Maximize the function z

$$z = a_1x_1 + a_2x_2 + ...a_nx_n$$

With the following constrains:

$$x_i \geq 0$$

and for j=1 to m

$$b_{j1}x_1 + b_{j2}x_2 + ...b_{jn}x_n \leq c_1$$

This function accepts the constrains symbols: "**<**" , and also "**>**", and "**=**"

This function returns:
If an optimal solution exists – that is: all constrains are satisfied and the function is maximized – then it returns the solution vector and, in the last cell, the corresponding function value
If the constrains region is unbounded – that is, if the region is not closed - a finite solution cannot exist and the function return "**inf**". Typically it happens when the constrains are insufficient
If the constrains region is bounded, but the solution doesn't exist, the function returns "**?**". Typically it happens when you add too many constrains

Note: The columns of Constrain must be n+2, where n is the columns of the function coefficients. If this condition is not true, the function returns "**??**". Typically it happens when you select region with wrong dimensions.

Now lets see how it works.

Example: find the maximum of the function:

F(x,y)= 1.2 x + 1.4 y

With the following constrains:

40 x +25 y =< 1000
35 x + 25 y = < 980
25 x + 35 y =< 875

	A	B	C	D	E	F	G
1	**Linear Optimization of**			***f(x,y) = 1.2x + 1.4y***			
2	*f (x,y) to maximize*				*constrains*		
3	x	y		x	y		b
4	1.2	1.4		40	25	<	1000
5				35	28	<	980
6	*solution*			25	35	<	875
7	x	y	f(x,y)				
8	16.935	12.903	38.387	{=Simplex(A4:B4;D4:G6)}			

Note that it is indifferent to write "<" or "<=" for the constrain symbols
The solution is about:

x = 16.935 , y = 12.903 , f(x, y) = 38.387

This function accept also mixed constrain symbols
Let's see this example

Maximize z = x1 + x2 + 3 x3 – 0.5 x4

With all the x variables non-negative and also with:

x1 + 2 x3 <= 10
2 x2 - 7 x4 <= 0
x2 – x3 + 2 x4 >= 10
x1 + x2 + x3 +x4 = 9

	A	B	C	D	E	F	G
1	Linear Optimization of $f(x,y) = x_1 + x_2 + 3x_3 - 0.5x_4$						
2							
3	function	x1	x2	x3	x4		
4		1	1	3	-0.5		
5							
6	constrain	x1	x2	x3	x4		
7		1	0	2	0	<	10
8		0	2	0	-7	<	0
9		0	1	-1	2	>	0.5
10		1	1	1	1	=	9
11							
12	solution	x1	x2	x3	x4	max	
13		0	3.325	4.725	0.95	17.025	
14							
15	{=Simplex(B4:E4;B7:G10)}						
16							

Function RRMS(v1, [v2])

This function computes the root mean squares of the regression residuals.
The argument v1 and v2 are vectors (n x 1) or (1 x n)
If the second vector is omitted the function returns simply the root mean squares of the vector

$$rrms = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_{1i} - v_{2i})^2} \qquad rms = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_{1i})^2}$$

This function can be used to return the average difference between two matrices

Example - Two different algorithms have given the following inverse matrices. Measure the average error.

C6 = =RRMS(A2:C4,D2:F4)

	A	B	C	D	E	F	G	H	I
1		A			A^			A-A^	
2	7	-21	35	7	-21	35	-2E-12	1E-11	-2.7E-11
3	-21	91	-175	-21	91	-175	6E-12	-3E-11	6.9E-11
4	35	-175	371	35	-175	371	-7E-12	3E-11	-5.1E-11
5							{=(A2:C4-D2:F4)}		
6	Difference RMS :		3E-11	=RRMS(A2:C4,D2:F4)					
7									

Function MatPerm(Permutations)

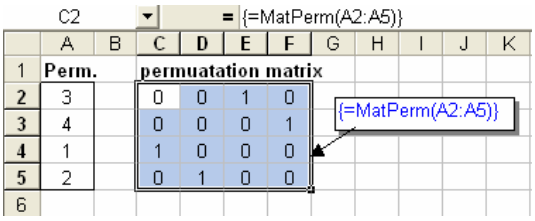
Returns the permutations matrix. It consists of sequence of n unitary vectors.
The parameter is a vector indicating the sequence.

For the 4x4 matrices space, we have

$$I=(u_1,u_2,u_3,u_4)=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

A permutations matrix is indicated by a sequence vector, like for example:

$$P(3,4,1,2)=(u_3,u_4,u_1,u_2)=\begin{bmatrix}0&0&1&0\\0&0&0&1\\1&0&0&0\\0&1&0&0\end{bmatrix}$$



Function Mat_Hessemberg(Mat)

Returns the Hessemberg form of a square matrix
As known a matrix is in Hessemberg form if all values under the lower sub-diagonal are zero.

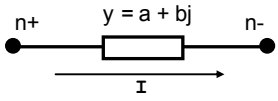
$$\begin{bmatrix}a_{11}&a_{12}&a_{13}&a_{14}&\dots\\a_{21}&a_{22}&a_{23}&a_{24}&\dots\\0&a_{32}&a_{33}&a_{34}&\dots\\0&0&a_{43}&a_{44}&\dots\\\dots&\dots&\dots&\dots&\dots\end{bmatrix}$$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	-10	-4	-3	-10	6	-3	0			-10	-0.4	-2	10.9	-11	-0.7	-3
2	9	9	-4	3	9	6	-3			9	11.2	0.92	-7	19.8	-1.6	6
3	7	-1	-2	-2	-1	6	-9			0	-15	8.89	-38	27.3	-6.3	1.33
4	1	6	4	9	-1	5	10			0	0	3.97	38.6	-46	7.53	-4.3
5	8	1	9	-1	4	1	6			0	0	0	50.6	-51	13.2	2.67
6	-5	-3	0	-5	-7	8	2			0	0	0	0	28.7	14.3	32.8
7	-1	-3	9	3	6	-2	-5			0	0	0	0	0	-2.6	1.36
8											=Mat_Hessemberg(A1:G7)					
9																

Function Mat_Adm(Branch)

Returns the Admittance Matrix of a Linear Passive Network Graph.

Branch is a list of 3 columns giving the basic information of each branch:
node+, node-, admittance .
The number of rows must be equal to the branches of the graph



A complex admittance has a real part (*conductance*) and an imaginary part (*susceptance*). In this case you have to provide a 4 columns list.

Nodal Analysis gives the following equation to solve the linear passive network, where V is the vector of nodal voltage, I is the vector of nodal current and [Y] is the admittance matrix.
If N+1 is the number of nodes, then the matrix dimension will be (N x N).
(usually the references nodes is set at V = 0)
V, I, [Y] are in general complexes

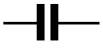
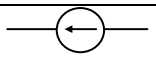
$$[Y] \cdot V = I$$

$$[Y] = \begin{cases} y_{ii} = \sum_{\substack{k=0 \\ k \neq i}}^{N} Y_{ik} \\ y_{ij} = -Y_{ij} \end{cases}$$

The function returns an (N x 2*N) array. The first N columns contain the real part and the last N columns contain the imaginary part. If all branch-admittances are real, also the matrix will be real and the function return a square (N x N) array.

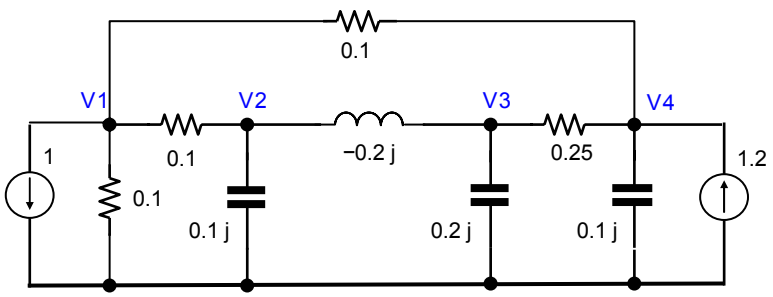
Linear Electric Network

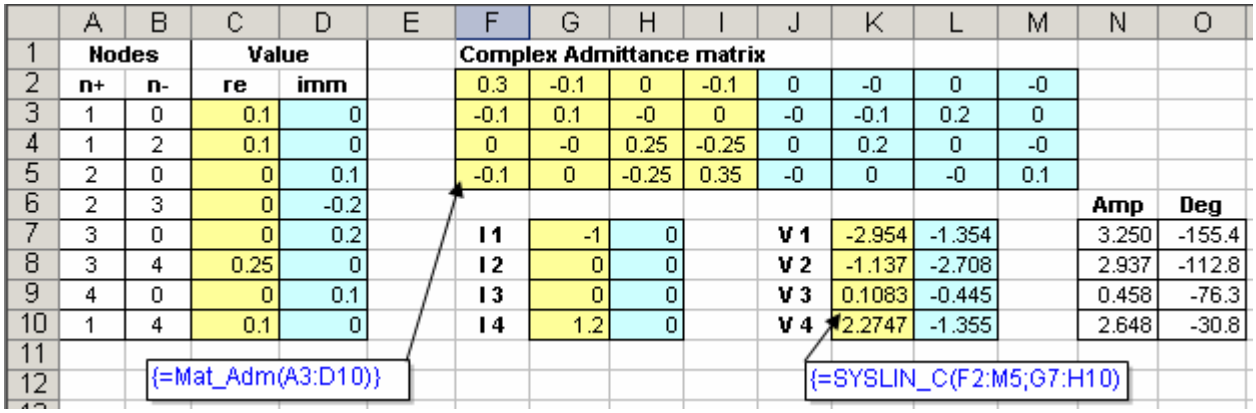
Nodal Analysis is widely used for Electric Network. A passive linear network is composed by four basic components: Resistor, Inductor, Capacitor and Current Source. In sinusoidal state, with constant frequency, the admittance branch can be derived by the following formulas

Resistor		Value R (ohm)	Admittance y = 1/R
Capacitor		Value C (farad)	Admittance y = j ω C
Inductor		Value L (henry)	Admittance y = −j 1/(ω L)
Current source		Value I (ampere)	$I = I_{re} + j\, I_{im}$

Where : ω = 2 π f (rad / s)

Example. Compute the admittance matrix of the following linear passive electric network and find the nodal voltage





	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	**Nodes**		**Value**			**Complex Admittance matrix**									
2	**n+**	**n-**	**re**	**imm**		0.3	-0.1	0	-0.1	0	-0	0	-0		
3	1	0	0.1	0		-0.1	0.1	-0	0	-0	-0.1	0.2	0		
4	1	2	0.1	0		0	-0	0.25	-0.25	0	0.2	0	-0		
5	2	0	0	0.1		-0.1	0	-0.25	0.35	-0	0	-0	0.1		
6	2	3	0	-0.2										**Amp**	**Deg**
7	3	0	0	0.2		**I 1**	-1	0		**V 1**	-2.954	-1.354		3.250	-155.4
8	3	4	0.25	0		**I 2**	0	0		**V 2**	-1.137	-2.708		2.937	-112.8
9	4	0	0	0.1		**I 3**	0	0		**V 3**	0.1083	-0.445		0.458	-76.3
10	1	4	0.1	0		**I 4**	1.2	0		**V 4**	2.2747	-1.355		2.648	-30.8
11															
12															

The network has 8 branches, mixed real and complex, and 4 nodes (the ground is the references node and is set to 0). So the network list has 8 rows and 4 columns. The independent currents generators are indicated in another list. The linear complex system can be solved by the SYSLIN_C.

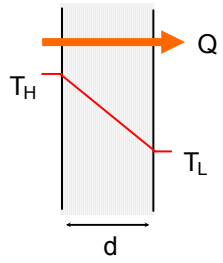
Thermal Network

There are also other networks than electrical ones, that can be solved with the same method. The same principles can be applied, for example to study one dimensional heat transfer.

One-Dimensional Conduction Heat Transfer.

The rate of conduction heat transfer through a material, having *thermal conductivity* "**k**", is proportional to the temperature gradient across the material

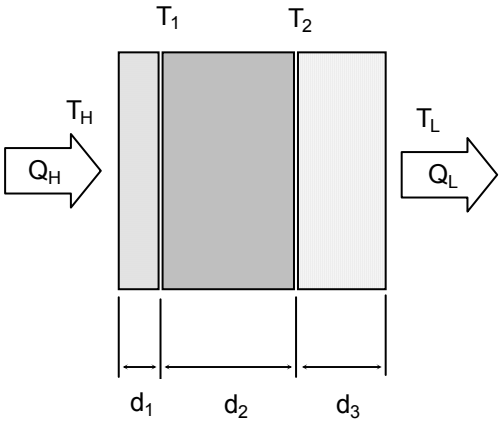
$$Q = -\frac{k}{d}\cdot(T_H - T_L) = -g(T_H - T_L)$$



Thus, the network equations are the same of the electric network after replacing:

I (ampere) electric current	⇔	Q (cal/s) rate of conduction heat
V (volt) Voltage	⇔	T (° Kelvin) temperature
g (siemens) electric conductance	⇔	g (cal/m s °K) thermal conductance

Example: Find the temperature profile through a sandwich material of 3 layers



Layer	d (cm)	K (cal /cm s °C)
1	0.1	0.04
2	0.4	0.12
3	0.2	0.08

Where:

T_H = 400 ° C

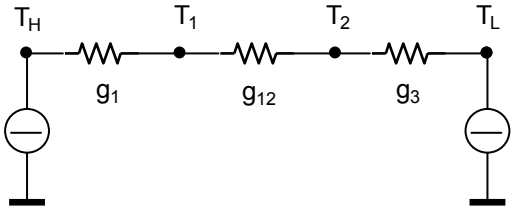
T_L = 20 ° C

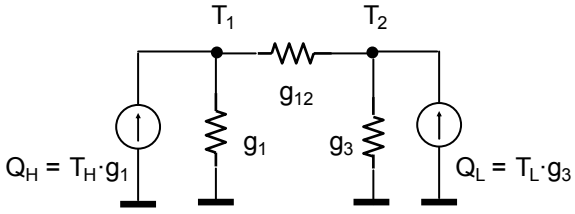
$Q_H = T_H \cdot k_1/d_1$ = 160 cal/s

$Q_L = T_L \cdot k_3/d_3$ = 8 cal/s

Internal temperature T_1 and T_2 are unknowns

The thermal networks equivalent to the above sandwich are shown in the following figures: the right one is obtained after substituting the temperature sources with theirs equivalent heat sources





Where thermal conductance are:

$$g_1 = k_1/d_1, \quad g_{12} = k_2/d_2, \quad g_3 = k_3/d_3$$

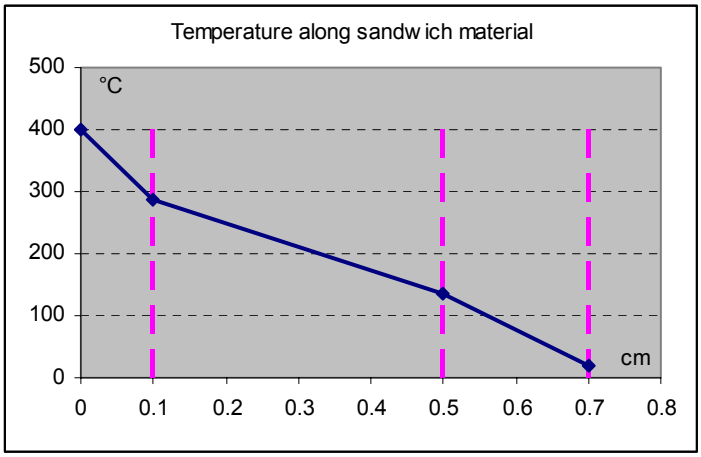
A spreadsheet calculus can be arranged as the following

	A	B	C	D	E	F	G	H
1	**Heat Transfer through a sandwich material**							
2								
3	Layer	d (cm)	K (cal /cm s °C)		TH	400	(° C)	
4	1	0.1	0.04		TL	20	(° C)	
5	2	0.4	0.12		QH	160	(cal/s)	
6	3	0.2	0.08		QL	8	(cal/s)	
7								
8	Node +	Node -	g (cal /s °C)		Admit. matrix		Q	T
9	1	0	0.4		0.7	-0.3	160	286.0
10	1	2	0.3		-0.3	0.7	8	134.0
11	2	0	0.4					
12					{=Mat_Adm(A9:C11)}			
13								

If [A] is the admittance matrix, the vector of temperature can be solved with the following formula

$T = [A]^{-1} Q$

With the internal temperature T_1 and T_2 we can easily draw the thermal profile along the material



Function Mat_Leontief(ExTab, Tot)

Returns the Leontief inverse matrix of the Input-Output Analysis Theory.
Parameter *ExTab* is the interindustry exchange table (or IO-table). This table lists the value of the goods produced by each economy sector and how much of that output is used by each sector.
Parameter *Tot* is the total production vector

Input Output Analysis

Recall theory definition. Input Output Analysis is an important branch of economics that uses linear algebra to model interdependence of industries. Assuming EX the *Exchange table*

$$EX = \lfloor x_{ij} \rfloor$$

The *Technology matrix* (or *Consuption matrix*) is

$$A = \left[\frac{x_{ij}}{X_j} \right]$$

where Xj is the total production of the j-th sector

The *Leontief inverse matrix* is .

$$L = (I - A)^{-1}$$

If D is the *Final Demand* vector the production X is given by the following formula.

$$D = L \cdot X$$

Example. Giving the following Exchange table of Goods and Services in the U.S. for 1947 (in billions of 1947 dollars), find the Leontief matrix and calculate the production for a final demand of

Supply sectors	Purchasing sectors			total output
	Agriculture	Manufact.	Services	
Agriculture	34.69	4.92	5.62	85.5
Manufact.	5.28	61.82	22.99	163
Services	10.45	25.95	42.03	219

Sectors	demand
Agriculture	45
Manufact.	74
Services	130

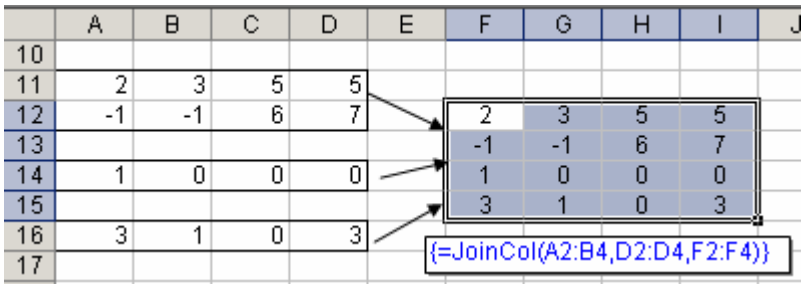
	A	B	C	D	E	F
1	**Interindustry exchange matrix**					
2	**Supply**	**Purchasing sectors**			**total**	**external**
3	**sectors**	Agriculture	Manufact.	Services	**output**	**demand**
4	Agriculture	34.69	4.92	5.62	85.5	45
5	Manufact.	5.28	61.82	22.99	163	74
6	Services	10.45	25.95	42.03	219	130
7						
8		**Multipliers Leontief matrix**			**Output**	
9		1.70698	0.10025	0.06723	92.97	8.7%
10		0.22084	1.67949	0.22519	163.49	0.3%
11		0.30169	0.34604	1.29203	207.15	-5.4%
12						
13	{=Mat_Leontief(B4:D6;F4:F6)}				{=MMULT(B9:D11;G4:G6)}	
14						

As we can see, in order to satisfy the demand of agriculture = 45, manufacturing = 74, and Services = 130, the total production should be increased of 8.7% for the agriculture, 0.3% for the manufacturing and decreased of -5.4% for services

.

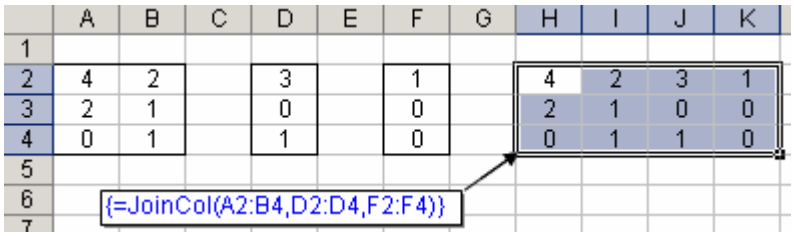
Function JoinRow(R1, R2, [R3]...)

This function builds a matrix using separated rows
R1, R2, .up to R8..are (1 x m) array having the same dimension m. They can be also rectangular (n x m) arrays having the same width.
The rows can be located in any part of the sheet



Function JoinCol(C1, C2, [C3]...)

This function builds a matrix using separated columns
C1, C2, .up to C8..are (n x 1) array having the same dimension n. They can be also rectangular (n x m) arrays having the same height.
The columns can be located in any part of the sheet

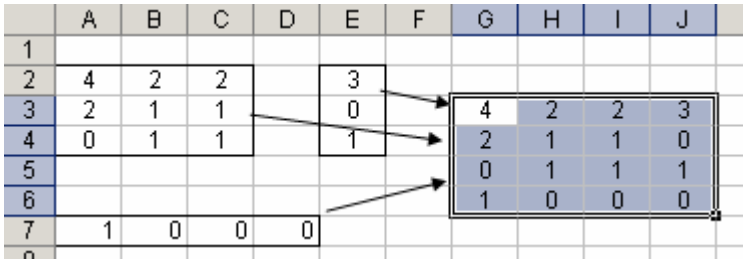


The functions JoinRow and JoinCol can be nested each other for building matrices.

Here we have used the array formula

```
{=JoinRow(JoinCol(A2:C4,E2:E4),A7:D7)}
```

for building the (4 x 4) matrix



Chapter 3

Matrix Tool

The Matrix toolbar

This floating toolbar is useful for several tasks: selecting and pasting scraps of matrices; generating different kind of matrices and several useful matrix operations. And, of course, it can be used also for recalling the Matrix help-on-line. You can show It by clicking on the Matrix icon

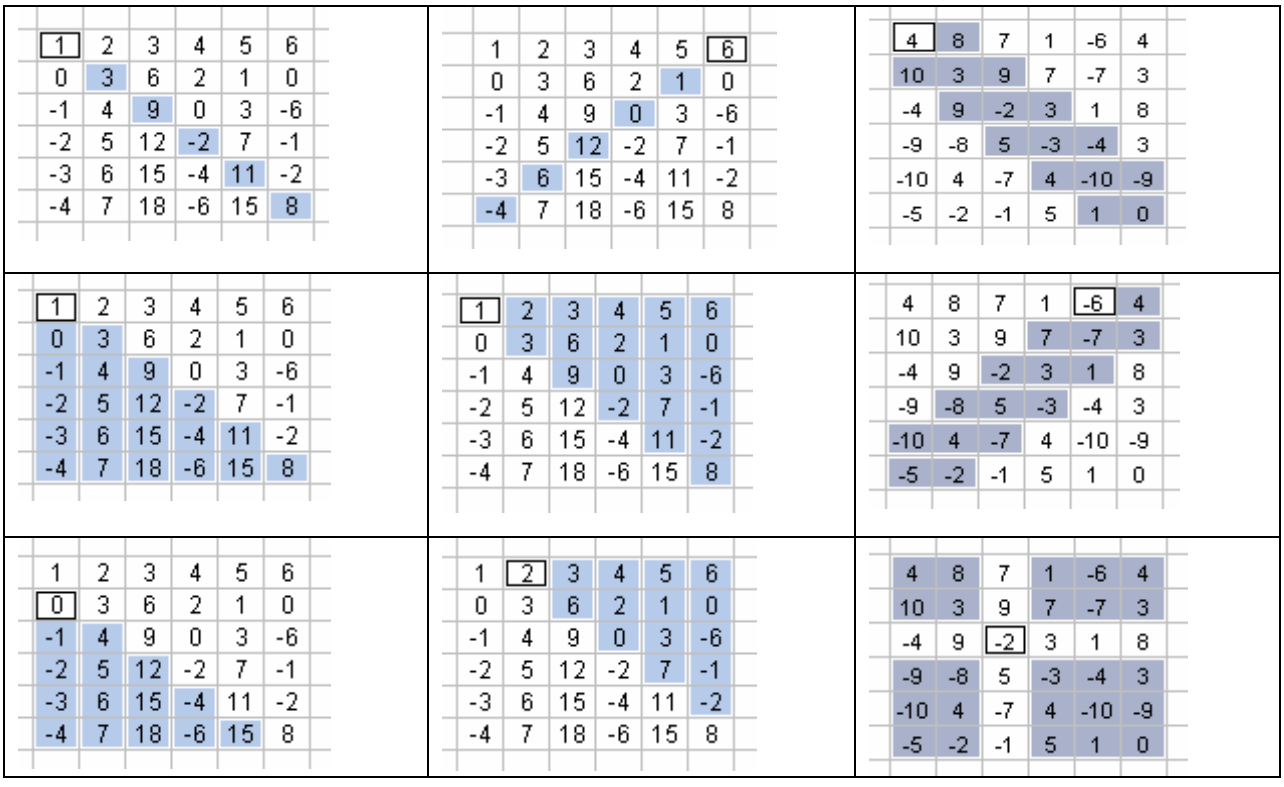
(Matrix.xla v.1.9)

Topics available are:

- **Selector tool** select matrix pieces
- **Generator tool** generate random and special matrices
- **Macros** starter for macros stuff
- **Help** call the on-line manual

Selector tool.

It can be used for selecting several different matrix formats: diagonal, triangular, tridiagonal, adjoin, etc. Simply select any cell into the matrix and choose the menu item that you want.



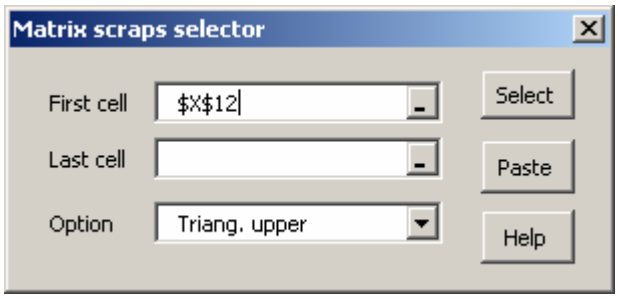
Automatically the "*Selector tool*" works on the max area bordered by empty cell that usually correspond to the full matrix. If you want to restrict the area simply select the sub-matrix that you want before starting the *Selector* macro

For example if you need to select the lower sub diagonal; simply select the sub-matrix containing the diagonal [10,9,5,4,1].
Then choose the menu item:
Selector\diagonal 1st

4	8	7	1	-6	4
10	3	9	7	-7	3
-4	9	-2	3	1	8
-9	-8	5	-3	-4	3
-10	4	-7	4	-10	-9
-5	-2	-1	5	1	0

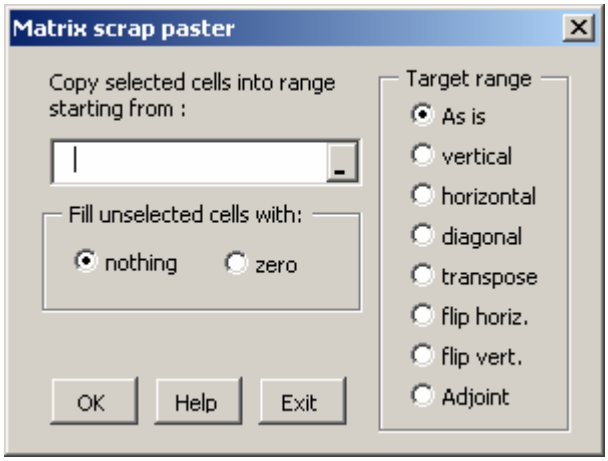
4	8	7	1	-6	4
10	3	9	7	-7	3
-4	9	-2	3	1	8
-9	-8	5	-3	-4	3
-10	4	-7	4	-10	-9
-5	-2	-1	5	1	0

If you start the macro without selecting any matrix cell, the following pop-up window appears asking you the top-left and the bottom-right corners of the range that you want to select. By the combo box you can choose the selection format that you like
The *Paste* button call the *Paster tool*



Scraps Paster tool.
The selection of matrix's parts is obtained by a multi range selection. Excel cannot copy it. If you try to give the usual sequence CTRL+C you will have an error.

For this task comes in handy this smart little macro for pasting the range or multi-range that you have previous selected.
After chosen the *Paste*r item from the menu a window pop-up appears. Simply indicate the destination top-left corner and chose OK.
That's all
The destination cells will be filled with the values of the selected cells. No format will be copied. Only plain values.



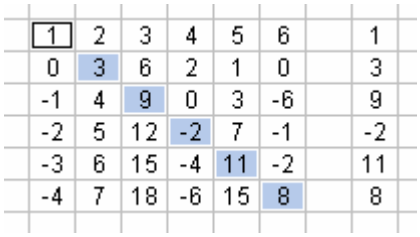
Of course this tool has other interesting options. Let's see.

Fill unselecting cells with zero: check this if you fill all other cell of the matrix with zero. This is useful to build a new matrix with a part of the original one.
Example: If you want to build a new lower triangular matrix with the element of another one, you can use this simply option and the result will be similar to the following:

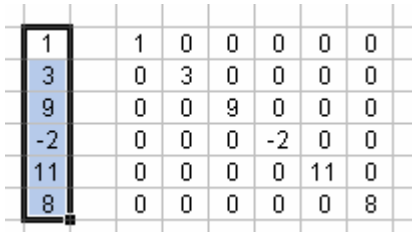
1	2	3	4	5	6
0	3	6	2	1	0
-1	4	9	0	3	-6
-2	5	12	-2	7	-1
-3	6	15	-4	11	-2
-4	7	18	-6	15	8

1	0	0	0	0	0
0	3	0	0	0	0
-1	4	9	0	0	0
-2	5	12	-2	0	0
-3	6	15	-4	11	0
-4	7	18	-6	15	8

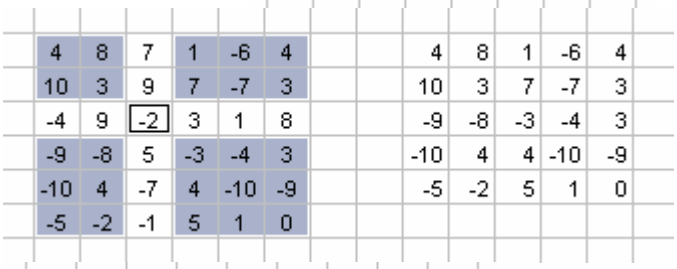
Change the target range: Normally the range is copied as is. But sometime we need to reformat the geometry of the given range. This happens, for example when we want to extract the diagonal elements from a given matrix and to convert it in a vertical vector.
In this case, after you have selected the diagonal, check the option *vertical*
The diagonal element will be... "verticalized".



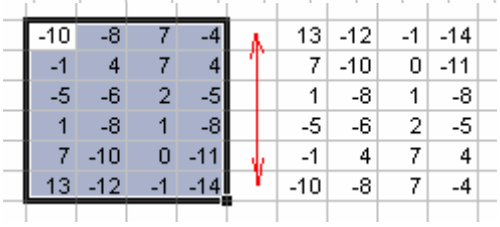
Some time we need the inverse of this transformation: from a vertical vector, we have to build the diagonal matrix having the vector elements on its diagonal.
For that select the vector that contains the elements. Start the *Scraps Paster* tool and check the *zero* and *diagonal* option. Giving OK, you will generate the matrix to the left



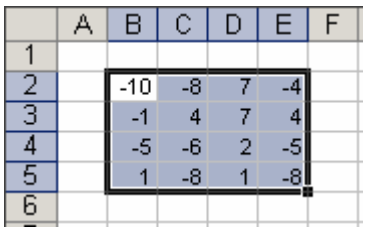
Or we can extract an adjoint sub-matrix. For example, select the a33 element and choose the menu *Selector\adjoint*. Than activate the Paster. Indicate the top-left corner and select the option "*Adjoint*". The macro will copy the selected elements rebuilding a new 5x5 matrix



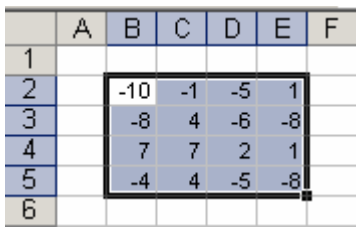
Flip. We can also invert the order of rows or columns of the target matrix
For example, select the full matrix (ctrl+shift+*) and run the "*Paster tool*", choosing the *flip vertical* option.



The matrix target can also be the same of the original one. In that case the changing will be done "on place" of the same matrix. Of course the transformation has sense only if the source and target range have the same dimensions: that is for square matrices. For example, assume you want transpose a square matrix on the same site



Select the range B2:E5 and then run the "*Paster tool*", choosing the B2 as target corner and *Transpose* as option.
The result will be the transpose matrix in the same range



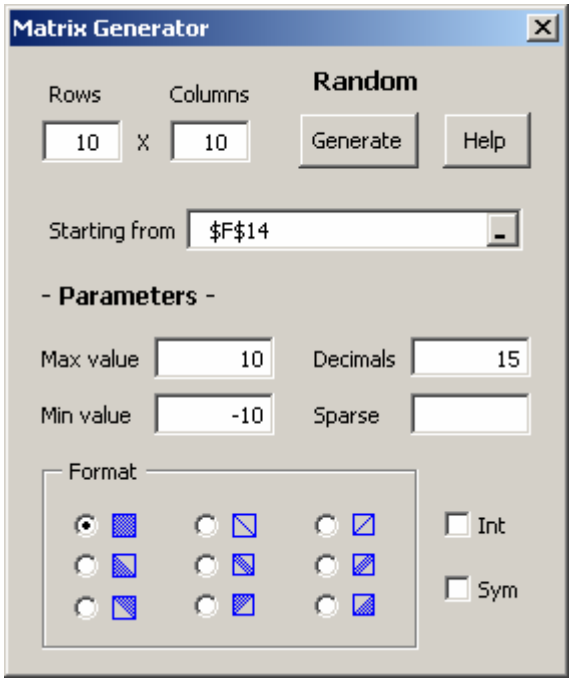
Matrix Generator

This smart macro can generate different kind of matrices

Random	Generate random matrices with the following parameter: dimension, max e min values, format: full, triangular, tridiagonal, symmetric, decimals number.
Rank / Determinant	Generate random matrices with given rank or determinant
Eigenvalues	Generate random matrices having given eigenvalues
Hilbert	Generate Hilbert's matrices
Hilbert inverse	Generate the inverse of Hilbert's matrices
Tartaglia	Generate Tartaglia's matrices
Toeplitz	Generate Toeplitz's matrices

Using this macro is quite simple: select the area that you want to fill with the matrix and then start the macro by the Matrix.xla toolbar.

Random matrix with given format



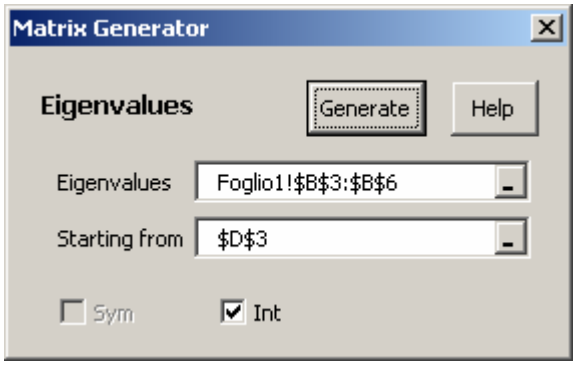
Parameters:
Random numbers **x** are generated with the following constrains:

Max value: upper limit of x
Min value: lower limit of x
Decimals: fix the decimals of x
Int checkbox: numbers x are integers

Sym checkbox: the matrix will be symmetric
Sparse: percentage (between 0 and 1) of non-zero elements to the total elements. The number 1 (default) means full matrix

Starting from: top-left matrix corner

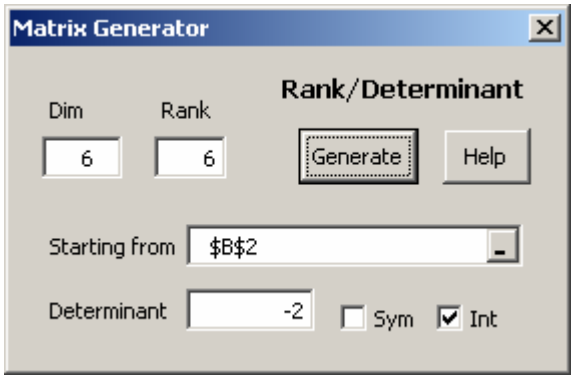
Random matrix with given eigenvalues



Eigenvalues: vertical range containing the real eigenvalues.

	A	B	C	D	E	F	G
1							
2		eigenval.					
3		1		8	6	2	10
4		2		-19	-17	-2	-40
5		3		10	10	3	21
6		4		6	6	0	16
7							

Random matrix with given determinant or rank



Generate a square matrix with given determinant or rank

Dimension: matrix dimension
Rank: rank of the matrix. For default is set equal to the dimension. If it is less than the dimension, the determinant is automatically set to 0.
Starting from: top-left matrix corner
Determinant: sets the determinant of the random matrix
Sym: generate a symmetric random matrix
Int: generate a random matrix with integer values

Tartaglia's matrix

Generate the Tartaglia's matrix with given dimension. See Function Mat_Tartaglia()

Hilbert's matrix

Generate the Hilbert's matrix with given dimension. See Function Mat_Hilbert()

Macros stuff.

This menu contains several macros performing useful tasks.

Macro versus Function
In Matrix package there are worksheet functions that perform the same tasks of the macros. The reason is that macros are more suitable for large matrices and heavy long computation.
On the other hand, functions are suitable for automatic recalculation, but we have to say that Excel becomes very slow for large worksheet full of active functions.
You should see when it is convenient using macros or functions.

Macros available are:

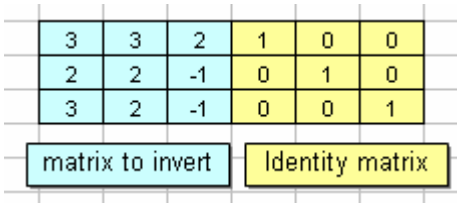
Matrix operations	Real Matrix operation
Complex matrix operations	Complex Matrix operation
Eigen-solving	Eigenvalues / Eigenvector for real and complex matricies
Gauss-step-by-step	Matrix reduction step by step with Gauss-Jordan algorithm
Shortest Path	Shortest paths matrix of a distance matrix
Draw Graph	Flow-Graph drawing of a distance matrix
Block reduction	Matrix block reduction with permutation matrix
Clean-up	Eliminate the tiny values
Round	Round values

Macro Gauss-step-by-step

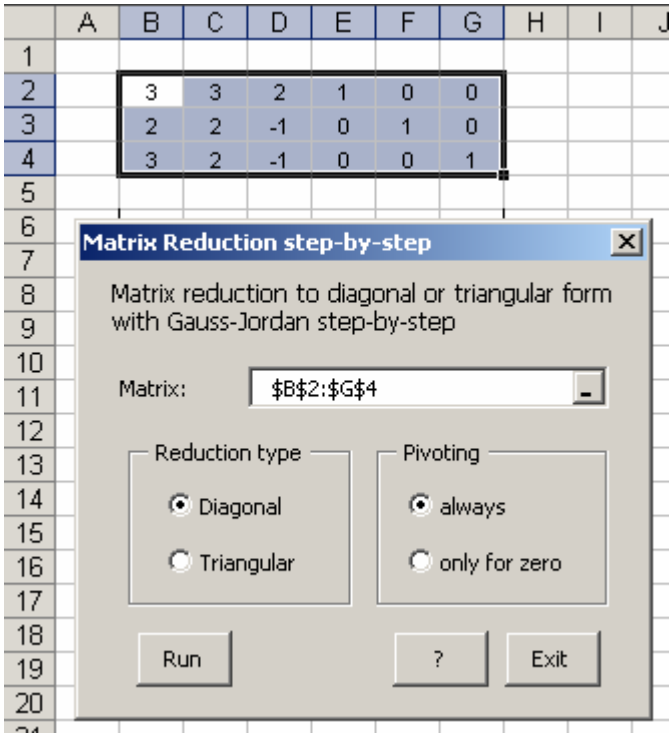
This macro performs all steps to reduce a given matrix into a triangular or diagonal form by the Gauss and Gauss-Jordan algorithm.
This macro works like the function **Gauss_Jordan_step** except that it traces all multiplication coefficients as well all the swap actions.

Using this macro is very easy. Select the matrix that you want to reduce and start the macro from the menu: **>macros > Gauss step by step**

For example, if you want to invert the following 3x3 matrix, add the 3x3 identity matrix to its right



then, select the 3x6 range and start the macro

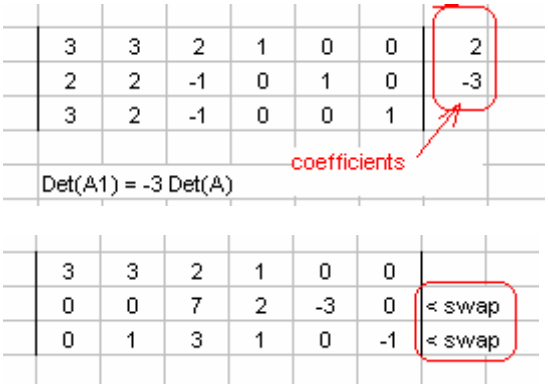


The reduction type options make the reduction to diagonal form (Gauss-Jordan) or to triangular (Gauss)

The pivoting options force the algorithm to search always the max pivot along the column (partial pivoting) or, on the contrary, only when the diagonal element is zero.

The first strategy is adopted for reducing the round off error, while the second one, more simple, is common in didactic applications

The process are traced below the original matrix.



Usually the coefficients for the linear combination are shown to the right…

Of course the determinant changes.
In this example we see that the determinant of the new matrix A1 is -3 times the one of the original matrix A

…as well the exchange action

Of course also the determinant changes sign.

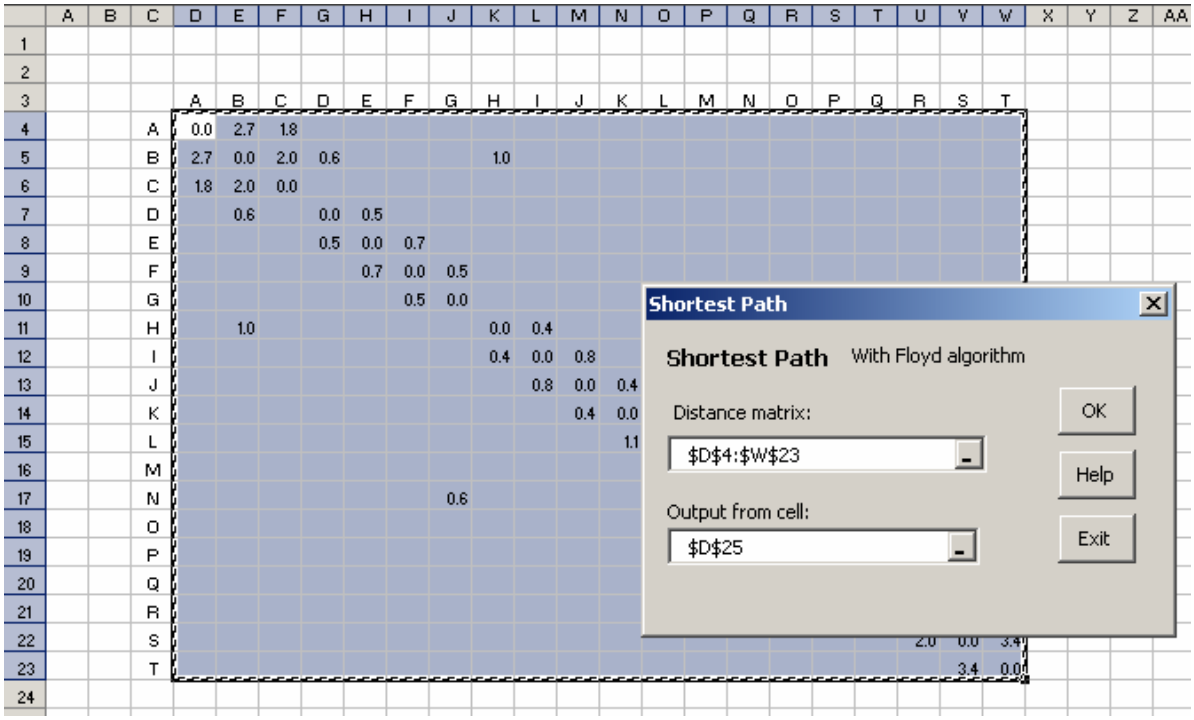
For further examples see the chapter *Gauss-Jordan algorithm*.

Macro Shortest-Path

This macro generates the shortest-path matrix from a given distance-matrix . It works like the function **Path_Floyd** except that it accepts larger matrices (up to 255 x 255)

Using this macro is very easy. Select the matrix that you want to reduce and start the macro from the menu: **>macros > Shortest path**

In the example below we see a 20 x 20 distance matrix

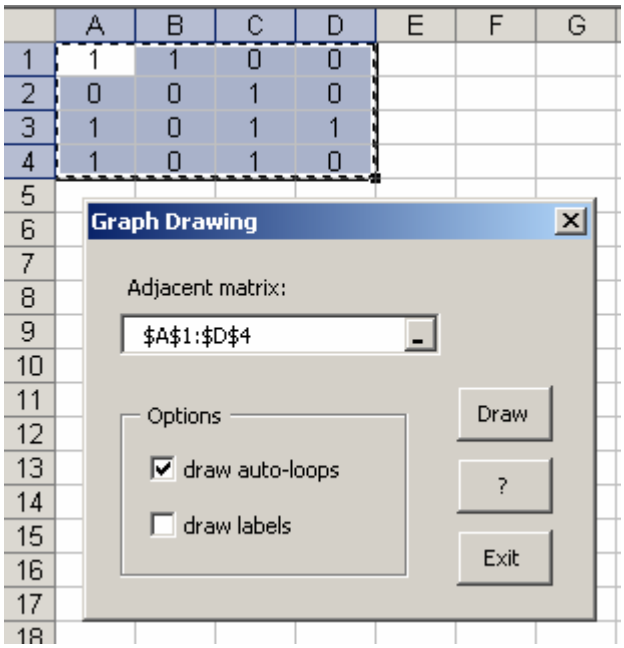


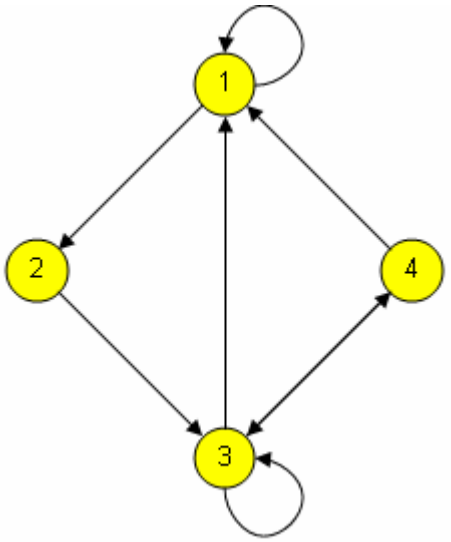
For default, the output matrix begin from cell D5, just below the input matrix.

Macro Draw Graph

This useful stuff draws a simple graph from its adjacent matrix. (There is now equivalent function for this macro). Using this macro is very easy. Select the matrix and start the macro from the menu:

>macros > Graph > Draw

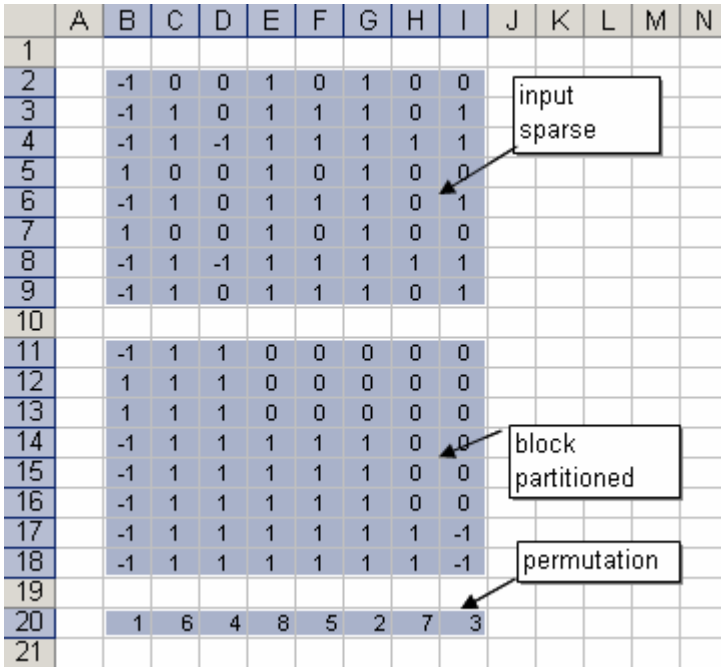




Macro Block reduction

This macro transforms a square sparse matrix into a block-partitioned form using the score-algorithm. This macro works like the functions **Mat_Blok** and **Mat_BlokPerm** except it is more adapt for large matrices.
Using this macro is very easy. Select the matrix and start the macro at the menu:

>macros > Block reduction



References

[1] "LAPACK -- Linear Algebra PACKage" 3.0, Update: May 31, 2000

[2] "Numerical Analysis" F. Sheid, McGraw-Hill Book Company, New-York, 1968

[3] "Numerical Recipes in FORTRAN 77- The Art of Scientific Computing" - 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software

[4] "Nonlinear regression", Gordon K. Smyth, John Wiley & Sons, 2002, Vol. 3, pp 1405-1411

[5] "Linear Algebra" vol 2 of Handbook for Automatic Computation, Wilkinson, Martin, and Peterson, 1971

[6] “Linear Algebra” Jim Hefferon, Saint Michael’s College, Colchester (Vermont), July 2001.

[7] “Linear Algebra - Answers to Exercises” Jim Hefferon, Saint Michael’s College, Colchester (Vermont), July 2001

[8] “Calcolo Numerico” Giovanni Gheri, Università degli Studi di Pisa, 2002

[9] “Introduction to the Singular Value Decomposition” by Todd Will, UW-La Crosse, University of Wisconsin, 1999

[10] “Calcul matriciel et équation linéaires”, Jean Debord, Limoges Cedex, 2003

[11] “Leontief Input-Output Modelling” Addison-Wesley, Pearson Education

[12] “Computational Linear Algebra with Models”, Gareth Williams, (Boston: Allyn and Bacon, 1978), pp. 123-127.

[13] “Scalar, Vectors & Matrices”, J. Walt Oler, Texas Teach University, 1980

[14] EISPACK Guide, “Matrix Eigensystem Routines”, Smith B. T. at al. 1976

WHITE PAGE

Analytical Index

ITALY
leovlp@libero.it

5. Edition
5 Printing: June. 2005