

Volume

1

FOXES TEAM

Reference Guide for Matrix.xla

Matrices and Linear Algebra

REFERENCE GUIDE FOR MATRIX.XLA

Matrices and Linear Algebra

Index

About Matrix.xla	6
Matrix.xla	6
Array functions	7
What is an array function	7
How to insert an array function	7
How to get help on line	11
MATRIX installation	12
How to install	12
Update to a new version	13
How to uninstall	13
About complex matrix formats	14
Functions Reference	16
Function MAbs(V)	17
Function MAbsC(V, [Cformat])	17
Function MAdd(A, B)	17
Function MAddC(A, B, [Cformat])	18
Function MBAB(A, B)	18
Function MDet(Mat, Mat, [IMode], [Tiny])	18
Function MDetC (Mat, [Cformat])	19
Function MDet3(Mat3)	20
Function MDetPar(Mat)	20
Function MInv(Mat, [IMode], [Tiny])	21
Function MPseudInv (A)	21
Function MPow(A, n)	22
Function MPowC(A, n, [Cformat])	22
Function MExp(A, [Algo], [n])	22
Function MExpErr(A, n)	22
Function MProd(A, B, ...)	23
Function MMultS(Mat, k)	23
Function MMultSC(Mat, scalar, [Cformat])	24
Function MMult3(Mat3, Mat)	24

Function MMultTpz(tpz, v)	25
Function MSub(A1, A2)	25
Function MSubC(A1, A2, [Cformat])	26
Function MTrace(Mat)	26
Function MDiag(Diag)	26
Function MDiagExtr(Mat, [Diag])	26
Function MT(Mat)	27
Function MTC(Mat, [Cformat])	27
Function MTH(Mat, [Cformat])	27
Function MRank(A)	28
Function MIde(n)	28
Function ProdScal(v1, v2)	28
Function ProdVect(v1, v2)	29
Function VectAngle(v1, v2)	29
Function MEigenvalJacobi(Mat, Optional MaxLoops)	30
<i>Eigenvalues problem with Jacobi, step by step</i>	31
Function MRotJacobi(Mat)	32
Function MBlock(Mat)	32
Function MBlockPerm(Mat)	33
Function MEigenvalQR(Mat)	34
Function MEigenvalQRC(Mat, [Cformat])	35
Function MEigenvec(A, Eigenvalues, [MaxErr])	35
Function MEigenvecC(A, Eigenvalue, [MaxErr])	36
Function MEigenvecJacobi(Mat, Optional MaxLoops)	37
Function MEigenSortJacobi(EigvalM, EigvectM, [num])	37
Function MEigenvalQL(Mat3, [IterMax])	37
Function MEigenvalTTPz(n, a, b, c)	38
Function MEigenvecT(Mat3, Eigenvalues, [MaxErr])	39
Function MChar(A, x)	40
Function MCharC(A, z, [Cformat])	40
Function MCharPoly(Mat)	41
Function MCharPolyC(Mat, [CFormat])	41
Function PolyRoots(poly)	42
Function MEigenvalMax(Mat, [IterMax])	42
<i>A global localization method for real eigenvalues</i>	42
Function MEigenvecMax(Mat, [Norm], [IterMax])	43
Function MEigenvalPow(Mat, [IterMax])	44
Function MEigenvecPow(Mat, [Norm], [IterMax])	44

Function MEigenvecInv(Mat, Eigenvalue)	44
Function MEigenvecInvC(Mat, Eigenvalue, [CFormat])	45
<i>About perturbed eigenvalues</i>	45
Matrix Generator	47
Function GJstep(Mat, [Typ], [IntValue], [tiny])	50
Function SysLin(A, b, [IMode], [Tiny])	52
Function SysLin3(Mat3, y)	53
Function SysLinIterG(A, b, x0, [Nmax], [w])	53
Function SysLinT(Mat, b, [typ], [tiny])	54
Function SysLinIterJ(A, b, x0, [Nmax])	55
Function SysLinSing(A, [b], [MaxErr])	55
Function SysLinTpz(A, b)	57
Function TraLin(A, x, [B])	58
<i>Matrix Geometric action</i>	58
Function MOrthoGS(A)	59
<i>Gram-Schmidt Orthogonalization</i>	60
Function MCholesky(A)	61
Function MLU(A, optional Pivot)	61
Function MQR(Mat)	63
Function MQRiter(Mat, [MaxLoops])	64
Function MQH(A, b)	65
Function MExtract(A, i_pivot, j_pivot)	65
Function PathFloyd(G)	66
Function PathMin(G)	66
<i>Graphs theory recalls</i>	66
<i>Shortest path</i>	69
Singular Value Decomposition -	71
Condition Number	72
Function MMopUp(M, [ErrMin])	74
Function MCovar(A)	74
Function MCorr(A)	74
Function RegrL(Y, X, [Intcpt])	75
Function RegrP(Degree, Y, X, [Intcpt])	77
Function RegrCir(X, Y)	78
Function MCmp(Coeff)	79
Function MCplx([Ar], [Ai], [Cformat])	79
Function PolyRootsQR(poly)	80
Function PolyRootsQRC(Coefficients)	80
Function MRot(n, theta, p, q)	81
Function VarimaxRot(FL, [Normal], [MaxErr], [MaxIter])	82

Function VarimaxIndex(Mat, [Normal])	83
Function MNormalize(Mat, [NormType], [Tiny])	83
Function MNormalizeC(Mat, [NormType], [Cformat], [Tiny]).....	83
Function MNorm(v, [NORM])	84
Function MMultC(M1, M2, [Cformat]).....	85
Function MInvC(A, [Cformat])	86
Function ProdScaC(v1, v2,)	86
Function SysLinC(A, b, [Cformat])	87
Function Simplex(Funct, Constraint, [Opt]).....	88
Function MPerm(Permutations)	90
Function MHessenberg(Mat).....	90
Function MAdm(Branch)	91
<i>Linear Electric Network</i>	91
<i>Thermal Network</i>	92
Function MLeontInv(ExTab, Tot).....	94
<i>Input Output Analysis</i>	94
Matrix Tool	95
The Matrix toolbar	95
<i>Selector tool</i>	95
<i>Matrix Generator</i>	98
Macro stuff.....	102
<i>Matrix operations</i>	102
<i>Sparse Matrix operations</i>	103
<i>Complex Matrix operations</i>	104
<i>Macro Gauss-step-by-step</i>	105
<i>Macro Shortest-Path</i>	106
<i>Macro Draw Graph</i>	106
<i>Macro Block reduction</i>	107
References	108
Credits	109

About Matrix.xla

Matrix.xla

Matrix.xla is an Excel add-in 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 a symmetric matrix with Jacobi algorithm. Jacobi's rotation matrix. Eigenvalues with the QR and QL algorithm. Characteristic polynomials. Polynomial roots with the QR algorithm. Eigenvectors for real and complex matrices

Generation of a random matrix with given eigenvalues or of given Rank or Determinant. Generation of useful matrices: Hilbert's, Householder's, Tartaglia's. Vandermonde's

Solving a system of linear equations: Linear System with iterative methods: Gauss-Seidel and Jacobi algorithms. Gauss-Jordan algorithm, step by step. Singular Linear System.

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

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

Why Matrix.xla has the same functions as Excel?

Yes, The same functions, such as determinant, inversion, multiplication, and transpose, are in both Excel and Matrix.xla. They perform the same tasks. And in many case they return the same values. But they are not exchangeable in every situation.

Matrix.xla
algorithms
are open

The main difference lies in the algorithms used; in other words, in the way in which 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 calculations. Their main difference is that the Matrix.xla Inversion function uses the popular Gauss-Jordan algorithm - explained in many books and web pages - while the Excel built-in functions are written in inaccessible, proprietary code. In a few other cases we have simply created 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 an "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 use the keys sequence CTRL+SHIFT+ENTER;. The sequence must be used just after inserting the function parameters. Keep down both CTRL and SHIFT keys (the order of depressing them doesn't matter) and then press ENTER.

If you miss this sequence or use only the ENTER key, the function only 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 that you have to solve a 3x3 linear system. $\mathbf{Ax} = \mathbf{b}$

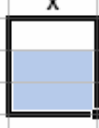
$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 3 & 4 \end{bmatrix} \quad b = \begin{bmatrix} 4 \\ 2 \\ 3 \end{bmatrix}$$

The function **SysLin** returns the solution \mathbf{x} ; but to see all the three values you must preselect the area where you want to insert these values.

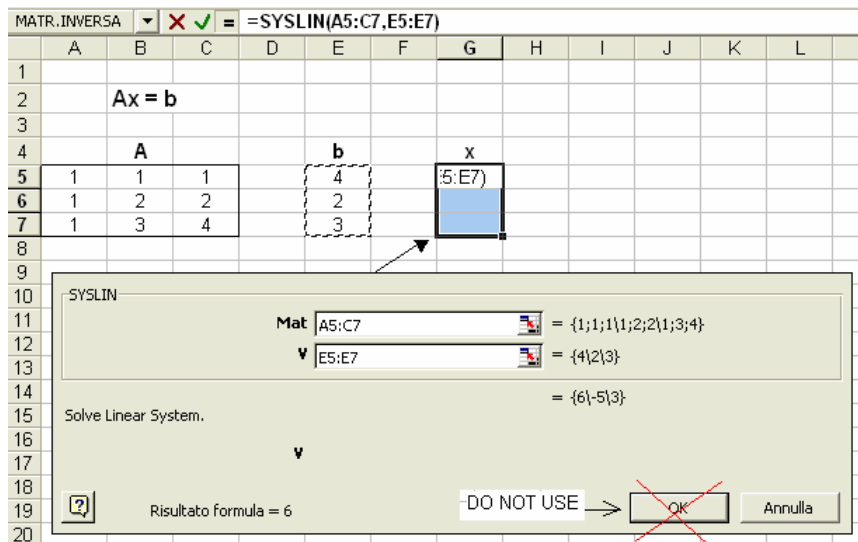
Now insert the function by menu, or by clicking

on the icon 

Select the area where you want to paste the result

	G5			=				
	A	B	C	D	E	F	G	H
1								
2		$\mathbf{Ax} = \mathbf{b}$						
3								
4		A			b		x	
5	1	1	1		4			
6	1	2	2		2			
7	1	3	4		3			
8								
9								

Select A5:C7 as the area of matrix **A** "A5:C7", and E5:E7 as 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 values will fill all the cells that you have selected.

	G5		= {=SYSLIN(A5:C7,E5:E7)}					
	A	B	C	D	E	F	G	H
1								
2								
3								
4								
5	1	1	1		4		6	
6	1	2	2		2		-5	
7	1	3	4		3		3	
8								
9								

The solution appears in the selected area

Note that Excel shows the function within two curly brackets { }. These symbols indicate that the function returns an array. You cannot insert them by yourself.

The output of an array function acts as a single unit. No part of the output array, can be modified or deleted. To modify or delete the output of an array function, you must select all the array 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 MAdd() function of Matrix.xla but we can also directly use 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; then write "+" and 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.

	A	B	C	D	E	F
1						
2						
3						
4		1	-2		1	0
5		2	1		0	1
6						
7						
8		=B4:C5+E4:F5				
9						
10						

- 4) Press and hold down CTRL + SHIFT
- 5) Press ENTER.

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

	A	B	C	D	E	F
3						
4		1	-2		1	0
5		2	1		0	1
6						
7						
8		2	-2			
9		2	2			
10						
11						

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

Let's see another useful examples

Linear combination of two vectors

The following example shows how to calculate the linear combination of two vectors

	A	B	C	D	E	F	G
10		v1		v2		v3	
11		1		0		34	
12	34	-2	22	1		-46	
13		4		-1		114	
14							
15							
16							
17							

Formula: $\{=A12*B11:B13+C12*D11:D13\}$

Scalar product of two vectors

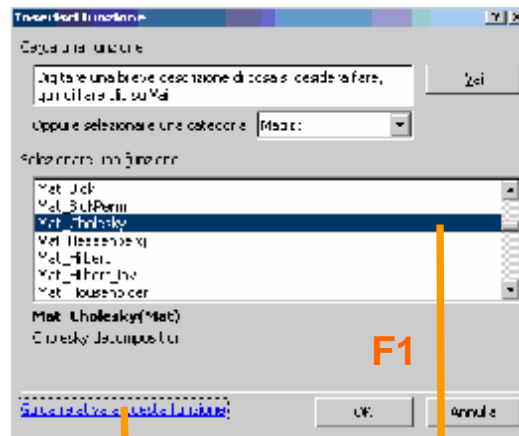
The following example shows how to calculate the scalar product of two vectors

	A	B	C	D	E	F	G
1							
2			v1		v2		v1*v2
3			3		2		-22
4			5		-4		
5			-1		8		
6							
7							
8							

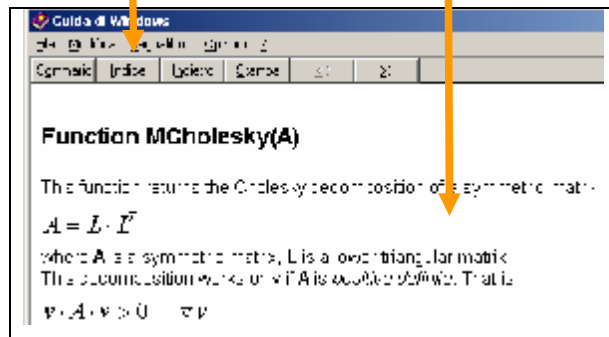
Formula: $\{=SUM(C3:C5*E3:E5)\}$

How to get help on line

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



Note that all the functions of this add-in appear under the 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 add-in for Excel is a zip file composed of the following files:

- MATRIX.XLA Excel add-in file
- MATRIX.HLP Help file
- MATRIX.CSV Function information (only for XNUMBERS add-in)
- FUNCUSTOMIZE.DLL¹ Dynamic Library for add-in

How to install

Unzip and place all the above files in a folder of your choice. The add-in 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 add-in:


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

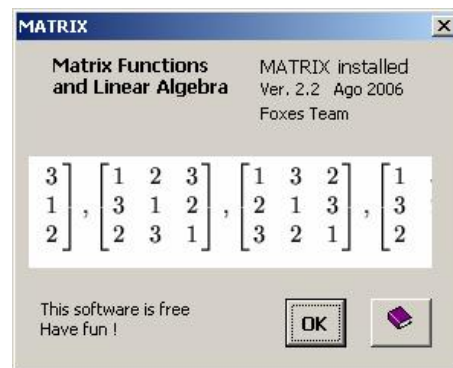
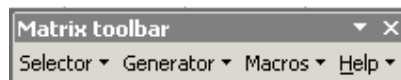
After the first installation, **matrix.xla** will be added to the Add-in list manager


When Excel starts, all add-ins checked in the Add-in 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 the first time you activate Matrix.xla in the Add-in Manager dialog box. Afterwards, when Excel automatically loads Matrix.xla, this popup remains hidden.

The Matrix Icon  is added to the main menu bar. By clicking on it, the Matrix Toolbar appears



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

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

Update to a new version

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

Do not keep two different versions on your PC, and, in general, never load two different versions, because Excel would make a mess of it.

How to uninstall

This package never alters your system files

If you want to uninstall this package, simply delete its folder. Once you have canceled the Matrix.xla file, remove the corresponding entry in the Add-in Manager list, by following these steps:

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

About complex matrix formats

Matrix.xla supports 3 different complex matrix formats: 1) split, 2) interlaced, and 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. This is the default format.

In the second format, the complex values are written as two adjacent cells, so that each individual matrix element occupies two adjacent cells. The number of columns is the same as in the first format, but the values are interlaced, so that each 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 text string "a + bi" or "a + bj"; therefore the square matrix is still square. This is the most compact and intuitive format for integer values. For non-integer values the matrix may become illegible. We must also point out that these elements, being strings, cannot be formatted with the standard tools of Excel, but must be converted back to numbers with the Excel commands IMREAL and IMAGINARY.

WHITE PAGE

Functions Reference

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

GJstep	MEigenvalQL	MNorm	PathMin
MAbs	MEigenvalQR	MNormalize	PolyRoots
MAbsC	MEigenvalQRC	MOrthoGS	PolyRootsQR
MAdd	MEigenvalTtpz	MpCond	PolyRootsQRC
MAddC	MEigenvec	MPerm	ProdScal
MAdm	MEigenvecC	MPow	ProdScalC
Mat_Hessemberg	MEigenvecInv	MPowC	ProdVect
MBAB	MEigenvecInvC	MProd	RegrCir
MBlock	MEigenvecJacobi	MPseudoinv	RegrL
MBlockPerm	MEigenvecMax	MQH	RegrP
MChar	MEigenvecPow	MQR	Simplex
MCharC	MEigenvecT	MQRiter	SVDD
MCharPoly	MExp	MRank	SVDU
MCharPolyC	MExpErr	MRnd	SVDV
MCholesky	MExtract	MRndEig	SysLin
MCmp	MHilbert	MRndEigSym	SysLin3
MCond	MHilbertInv	MRndRank	SysLinC
MCorr	MHouseholder	MRndSym	SysLinIterG
MCovar	MIde	MRot	SysLinIterJ
MCplx	MInv	MRotJacobi	SysLinSing
MDet	MInvC	MSub	SysLinT
MDet3	MLeontInv	MSubC	SysLinTtpz
MDetC	MLU	MT	TraLin
MDetPar	MMopUp	MTartaglia	VarimaxIndex
Mdiag	MMult3	MTC	VarimaxRot
MdiagExtr	MMultC	MTH	VectAngle
MEigenvalJacobi	MMultS	MTrace	
MEigenvalMax	MMultsC	MVandermonde	
MEigenvalPow	MMultTtpz	PathFloyd	

Function MAbs(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 MAbsC(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
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 =MAbsC(D3:G4)					4 =MAbsC(I3:J4, 3)		
7											
8	8.718	=MAbsC(A3:B6)									

See [About complex matrix format](#)

Function MAdd(A, B)

Returns the sum of two matrices

$$C = A + B$$

according to the definition:

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

For example, the sum of (2 x 2) matrices is

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

Note: EXCEL has a simply way to perform the addition of two arrays. For details see [How to insert an array function...](#)

Function MAddC(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

The optional parameter *Cformat* sets the complex input/output format (default = 1)

G2		fx		{=MAddC(A2:B3,D2:E3,3)}					
	A	B	C	D	E	F	G	H	
1	A			B			A + B		
2	2 + j	-1		8 + 2j	1		10+3j	0	
3	0	3 + j		2 + j	j		2+j	3+2j	

Function MBAB(A, B)

Returns the product:

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

This operation is also called the "*similarity transform*" of matrix **A** by matrix **B**

Similarity transforms play a crucial role in the computation of eigenvalues, because they leave the eigenvalues of the matrix **A** unchanged.

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

I2		=		{=MBAB(A2:C4,E2:G4)}							
	A	B	C	D	E	F	G	H	I	J	K
1		A				B			B ⁻¹ A B		
2	0	1	0		1	1	1		1	0	0
3	0	-1	2		1	2	3		0	2	0
4	3	-9	7		1	3	6		0	0	3

Function MDet(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.

Use IMODE only with integer matrices of moderate size.

Tiny (default is 0) sets the minimum round-off error; any value with an absolute value less than Tiny will be set to zero.

For n = 1

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

For n = 2

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

For n = 3

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

Clearly, the computation of the determinant of a large matrix is one of the most tedious of all Math. Fortunately we have 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		MDet			EXCEL
3	-507	2025	-987		Integer	Float		-6.868E-10
4	245	-987	553		0	1.5039E-09		
5								

Function MDetC (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

The optional parameter Cformat sets the complex input/output format (default = 1)

A complex split or interlaced matrix must always have an even number of columns

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

	A	B	C	D	E	F	G	H	I
1									
2	1	2	3	0	1	-1			
3	-1	1	2	1	0	1		-11	15
4	2	0	-1	2	-1	0			
5									
6	1	0	2	1	3	-1			
7	-1	1	1	0	2	1		-11	15
8	2	2	0	-1	-1	0			
9									

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.

	A	B	C	D	E
9					
10	1	2+j	3-j		
11	-1+j	1	2+j		9+15j
12	2-2j	-i	-1		
13					
14					

The last example shows the rectangular *string* format

Function MDet3(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 handle only the three diagonals

Example: to find the determinant of a 18x18 tridiagonal matrix we pass to the function only 54 values (the first cell of the column **a** and the last of the column **c** are always 0) instead of 324 values.

	A	B	C	D	E
1	Matrix A (18 x 18)				
2	a	b	c		determinant(A)
3	0	2	-1		849612816
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		

=MDet3(A3:C20)

Function MDetPar(Mat)

Computes the parametric determinant D(k) of a (n x n) matrix containing a parameter k
This function returns the polynomial string D(k) or its vector coefficients depending on the range selected.
If you have selected one cell the function returns a string; if you have selected a vertical range, the function returns a vector. In that last case you have to insert the function with the ctrl+shift+enter key sequence.

The function accepts one parameter.

Any matrix element can be a linear function of k:

$$a_{ij} = q_{ij} + m_{ij} \cdot k$$

The maximum degree of the polynomial D(k) is 9.

	A	B	C	D	E	F	G	H	I	J
1										
2		k	2	0	1	0		126		
3		-2	2k	4	0	0		-170		
4		-1	-2	1	-4	2		32		
5		0	3	3	0	2				
6		0	0	1	3	1				
7										
8										
9										
10										

=MDetPar(B2:F6)

=MDetPar(B2:F6)

Function MInv(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 for integer matrices but also more limited because it may easily reach the overflow error. Use IMODE only with integer matrices of low size.

Tiny (default is 0) sets the minimum round-off error; any absolute value less than Tiny will be set to zero. If by this chosen criterion the matrix is singular, the function returns "singular".

If the matrix is not square the function returns "?"

Example: the following matrix is singular but only the MInv 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	{=MInv(A2:C4)}				{=MInv(A2:C4,TRUE)}		

Function MPseudInv (A)

Computes the Moore-Penrose pseudo-inverse of a (n x m) matrix

Def: the minimum-norm least squares solution x to a linear system

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

is the vector

$$x = (A^T A)^{-1} A^T b$$

The matrix A^+ is called the pseudo-inverse of A

If the matrix A has dimension (n x m), its pseudo-inverse has dimension (m x n)

One of the most important applications of SVD decomposition is

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

	A	B	C	D	E	F	G	H	I	J
1										
2		1	3	5						
3		1	-4	2						
4		0	-11	2						
5		1	-18	0						
6		2	0	1						
7		3	0	3						
8										

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

Function MPow(A, n)

Returns the integer power of a square matrix

$$B = A^n = \overbrace{A \cdot A \cdot A \dots A}^n$$

Function MPowC(A, n, [Cformat])

Returns the integer power of a complex square matrix

Use CTRL+SHIFT+ENTER to insert this function

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

Optional parameter *Cformat* sets the complex format of input/output (default = 1)

	A	B	C	D	E
1	A		n	A^n	
2	2	-j	6	-171-56j	-40+40j
3	4-j	-1		-200-120j	-51+64j
4					
5	{=MPow(A2:B3,C2,3)}				

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

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

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

This function uses two alternative algorithms: the first one uses the popular power series

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

For n sufficiently large, the error becomes negligible, and the sum approximates the exponential matrix function. The parameter "n" fixes the max term of the series. If omitted the expansion continues until convergence is reached; this means that the norm of the nth matrix term becomes less than Err = 1E-15.

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

When using this function without n, especially for a large matrix, the evaluation time can be very long. The second, more efficient, method uses the Padé approximation¹. It is recommendable especially for large matrices.

We can switch the algorithm by the optional parameter *Algo*. If "P" (default) the function uses the Padé approximation, otherwise it uses the power series

Function MExpErr(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

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

¹ This routine was developed by Gregory Klein, who kindly consented to add it to this package

Function MProd(A, B, ...)

Returns the product of two or more matrices

$$C = A \cdot B$$

As known, the product is defined as:

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

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, the product is also a square (n x n) matrix.

Matrix multiplication is associative. Thus:

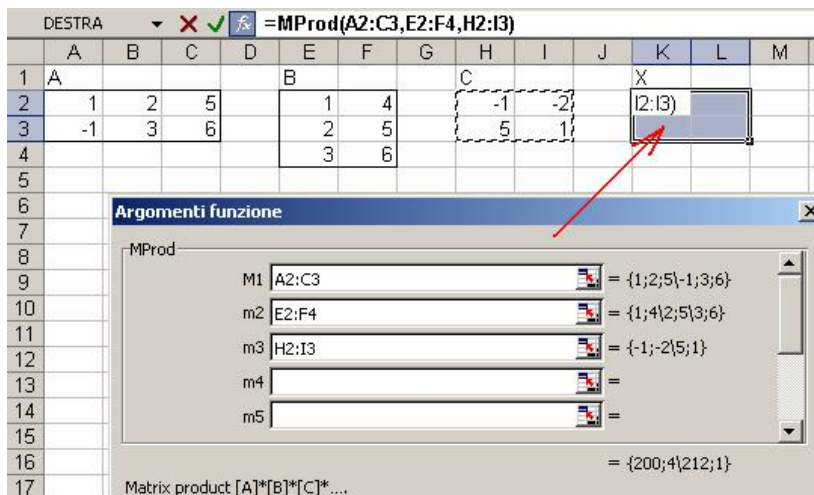
$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

But, generally, it is not commutative:

$$A \cdot B \neq B \cdot A$$

The function **MProd** can perform the product of several matrices, even with different dimensions.

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



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

Function MMultS(Mat, k)

Multiplies a matrix by a scalar

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

It can be nested in other function. For example, if the range A1:B2 contains the 2x2 matrix [1, 2, / -3, 8] MDet(MMultS(A1:B2; 3)) returns the determinant 126 of the matrix [3, 6, / -9, 24]

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

Function MMultSC(Mat, scalar, [Cformat])

Multiplies a complex matrix by a complex 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 a 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 input/output format (default = 1)

See [About complex matrix format](#)

Example. Performs the following complex multiplication

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

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	{=MMultSC(A2:F4, C6:D6)}					
12						

	L	M	N	O
	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	
	{=MMultSC(L2:N4, N6, 3)}			

This function also multiplies a complex vector and a complex number

...or a real vector and 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	{=MMultSC(A2:B4,D2E2)}							

	A	B	C	D	E	F	G	H
9								
	u			k			k u	
10	1		2+4j		2+4j			
11	2				4+8j			
12	4				8+16j			

Function MMult3(Mat3, Mat)

This function performs the multiplication of a tridiagonal 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)

The 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 and a vector because pre-2007 Excel cannot manage matrices larger than 256 columns wide.

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

	A	B	C	D	E	F	G
1	Matrix A (16 x 16)						
2	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
19							
20	{=MMult3(A3:C18, E3:E18)}						
21							

$$A \cdot x =$$

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

Note how compact and efficient this input is. This is true especially for large matrices

Function MMultTpz(tpz, v)

Perform the multiplication between a Toeplitz matrix and a vector

Parameter *Tpz* is a Toeplitz matrix written in the compact vector form (2n-1). See [Toeplitz matrices](#) for further details

Parameter *b* is the vector (n) of constant terms

Of course this function came in handy when the Toeplitz matrix is written a compact form. Otherwise you can use the M_MULT or the built-in MMULT Excel function as well

	A	B	C	D	E	F	G	H	I
1		Tpz		x		b			
2		-2		10		133			
3		-1		22		335			
4		-3		30		494			
5		2		45		409			
6		5		50		208			
7		4							
8		3							
9		-1							
10		-1							

Function MSub(A1, A2)

Returns the difference of two matrices

$$C = A_1 - A_2$$

Tip: EXCEL has a simply way to perform the multiplication of an array by a scalar. For details see "[How to insert an array function...](#)"

Function MSubC(A1, A2, [Cformat])

Returns the difference 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 input/output format (default = 1)

Example.

G2		=MSubC(A2:B3,D2:E3,3)							
	A	B	C	D	E	F	G	H	
1	A			B			A + B		
2	2 + j	-1		8 + 2j	1		-6-j	-2	
3	0	3 + j		2 + j	j		-2-j	3	

Function MTrace(Mat)

Returns the trace of a square matrix, i.e., the sum of the elements of the first diagonal

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

Example.

G2		=MTrace(A2:C4)							
	A	B	C	D	E	F	G	H	
1		A							
2	0	1	0				trace(A) =	6	
3	0	-1	2						
4	3	-9	7						

Function MDiag(Diag)

Returns to standard (expanded) form the diagonal matrix having the vector "Diag" as its diagonal

Example.

	A	B	C	D	E	F	G	H	
1	Diagonal								
2	1		1	0	0	0	0		
3	2		0	2	0	0	0		
4	3		0	0	3	0	0		
5	4		0	0	0	4	0		
6	5		0	0	0	0	5		

(=MDiag(A2:A6))

Function MDiagExtr(Mat, [Diag])

This function extracts the diagonals of a matrix¹

The optional parameter *Diag* selects the diagonal to extract.

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

Example.

¹ (Thanks to an idea of Giacomo Bruzzo)

	A	B	C	D	E	F	G
1						Diag 1	Diag 2
2	1	0.02	0.1	0.5		1	0.5
3	-0.5	-3	-0.2	0.3		-3	-0.2
4	0.2	0.4	6	0.02		6	0.4
5	0.7	0.1	0.2	4		4	0.7
6							
7							
8							
9							

Formulas shown in the image:

- `{=MDiagExtr(A2:D5)}` (pointing to cell F2)
- `{=MDiagExtr(A2:D5;2)}` (pointing to cell G2)

Function MT(Mat)

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

	A	B	C	D	E	F	G
1							
2	0	1	0		0	0	3
3	0	-1	2		1	-1	-9
4	3	-9	7		0	2	7

Formula shown: `=MT(A2:C4)`

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

Function MTC(Mat, [Cformat])

Returns the transpose of a complex matrix, that is 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 input/output format (default = 1)
 Use CTRL+SHIFT+ENTER to insert this function
 Example. Transpose the following (3 x 2) complex matrix

	A	B	C	D	E	F	G	H
1								
2		1	4	0	-1			
3	A =	2	3	2	6			
4		0	0	0	1			
5								
6								
7	A ^T =	1	2	0	0	2	0	
8		4	3	0	-1	6	1	

Formula shown: `{=MTC(B2:E4)}`

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

	A	B	C	D	E	F	G	H
1								
2		1	4	0	-1			
3	A =	2	3	2	6			
4		0	0	0	1			
5								
6								
7	A ^H =	1	2	0	0	-2	0	
8		4	3	0	1	-6	-1	

(=MTH(B2:E4))

Function MRank(A)

Returns the rank of a given matrix¹

Examples:

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2		1	-8	1		1	-8	-5		1	1	1
3		-8	64	-8		-6	49	40		2	3	0
4		3	-24	3		-4	32	20		4	-1	1
5												
6		Det	Rank			Det	Rank			Det	Rank	
7		0	1			0	2			-13	3	
8												
9		=MDet(B2:D4)				=MRank(B2:D4)						
10												

	A	B	C	D	E	F
1		A (3 x 4)				
2		1	5	9	-9	
3		-4	-19	-39	34	
4		-9	-45	-81	81	
5						
6		Det	Rank			
7		?	2	:=MRank(B2:E4)		

When Det = 0 the Rank is always less than 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 MIde(n)

Returns the identity square matrix.

This function is useful in nested expressions

Example: Compute the matrix $A - \lambda I$ for the parameter $\lambda = 1$

	E10							
		A	B	C	D	E	F	G
9			A		λ		A - λ I	
10		15	13	22	1	14	13	22
11		-6	-4	-10		-6	-5	-10
12		-4	-4	-5		-4	-4	-5

Note that we have used the power array arithmetic of Excel

But we could use the following nested expression:

`{=MAdd(A10:C12, D10*MIde(3))}`

Function ProdScal(v1, v2)

Returns the scalar product of two vectors

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = \sum v_{1,i} \cdot v_{2,i}$$

Note that if \mathbf{V}_1 and \mathbf{V}_2 are the same vectors, this function returns the square of its module.

¹ (Thanks to the original routine developed by Bernard Wagner.)

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

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

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

	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.

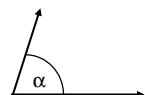
Vectors can be in vertical or horizontal format as well.

	A	B	C	D	E	F
1		v1	v2		v1 x v2	
2		1	2		0	
3		3	6		0	
4		-2	-4		0	
5						
6		{=ProdVect(B2:B4;C2:C4)}				
7						

Function VectAngle(v1, v2)

Computes the angle between the two vectors V_1 , V_2 .

The angle is defined as:



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

Example.

	A	B	C	D	E	F	G
1							
2		v1	v2	angle	v1 =	1	1
3		1	1	0.685	v2 =	-1	1
4		1	2		angle =	1.571	
5		-1	0				
6		=VectAngle(A3:A5,B3:B5)					
7							
8							
9							

Function MEigenvalJacobi(Mat, Optional MaxLoops)

This function performs the Jacobi 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 number of steps of the sequence.

The Jacobi algorithm can be used to find both eigenvalues and eigenvectors of symmetric matrices

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

$$A_2 = P_2^T \cdot A_1 \cdot P_2 = P_2^T P_1^T A P_1 P_2$$

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

For n sufficiently large, the sequence $\{A_i\}$ converges to a diagonal matrix, thus

$$\lim_{n \rightarrow \infty} A_n = [I]$$

while the matrix

$$U_n = P_n P_{n-1} \dots P_3 P_2 P_1$$

converges to the eigenvectors of A.

All these matrices **A**, **U**, **P** can be obtained by the functions: **MEigenvalJacobi**, **MEigenvecJacobi**, **MRotJacobi**

Example: Solve the eigenvalue problem of the following symmetric matrix

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

	A	B	C	D	E	F	G
1	Jacobi iterative method for diagonalization of symmetric matrices						
2	MaxLoop=	10					
3	5	2	0				
4	2	6	2				
5	0	2	7				
6	Eigenvalues			Eigenvectors			
7	3	8.46E-18	1.11E-16	0.666667	0.333333	-0.666667	
8	-1.26E-29	9	-1.11E-16	-0.666667	0.666667	-0.333333	
9	4.92E-34	1.47E-20	6	0.333333	0.666667	0.666667	

As we can see, the Jacobi method has found all Eigenvalues and Eigenvectors with few iterations (10 loops). The function **MEigenvalJacobi** returns in the range A7:C9 the diagonal matrix of the eigenvalues. Note that elements beyond the first diagonal have an error of less than 1E-15.

At the right side - in the range D7:F9 - the function **MEigenvecJacobi** returns the orthogonal matrix of the eigenvectors.

Compare with the exact solution

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

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

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

You can use the standard matrix inversion and multiplication functions or the **MBAB** function also in this package, as you like. Note that - only in this case - the matrix inversion is very simple, because:

$$U^{-1} = U^T$$

Eigenvalues problem with Jacobi, step by step

Suppose you want to study each step of the Jacobi method. The functions MEigenvalJacobi, MEigenvecJacobi and MRotJacobi are useful if you set the parameter MaxLoop=1. In this case, they return the first step of Jacobi's iteration. Here is how they work.

A1 = MEigenvecJacobi(A)
 A2 = MEigenvecJacobi(A1)
 A3 = MEigenvecJacobi(A2)

 A10 = MEigenvecJacobi(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

In order to quickly obtain the above iterations follow these simple steps:

- At the first time, insert in range A7:C9 the function MEigenvalJacobi(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 next range A11, and paste (CTRL+V)
- Copy it (CTRL+C)
- Select the next range A15, and paste again (CTRL+V), etc.

By the sequence of copying and pasting- you can perform all Jacobi iterations, as you like. In the middle, we see the Jacobi's rotation matrices sequence. We can easily obtain these it by the function MRotJacobi() in the same way as the eigenvalues matrix

P1 = MRotJacobi(A), P2 = MRotJacobi(A1), P3 = MRotJacobi(A2), etc.

This function searches for the max absolute values of off-diagonal elements, 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 MRotJacobi(Mat)

This function returns Jacobi 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

$$A_1 = P_1^T A \cdot P_1, \quad \text{Where: } P_1 = \text{MRotJacobi}(A)$$

For further details see Function MEigenvalJacobi(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 just these elements zero.

The rotation matrix, in that case is

1	0	0	0	0
0	1	0	0	0
0	0	$\cos(\alpha)$	0	$\sin(\alpha)$
0	0	0	1	0
0	0	$-\sin(\alpha)$	0	$\cos(\alpha)$

where the angle α is given by the formula:

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

	A	B	C	D	E	F	G	H	I	J	K
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						

`{=MRotJacobi(A2:E6)}`

`{=MBAB(A2:E6,G2:K6)}`

Function MBlock(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.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1															
2		3	1	7	0	1	0								
3		0	1	0	3	1	2								
4		3	5	3	1	0	1								
5		2	-3	0	1	1	1								
6		0	2	1	0	1	0								
7		1	0	1	2	-9	1								
8															
9		Permutation													
10		?	?	?	?	?	?								

Irreducible

{=MBlockPerm(B2:G7)} ..

Also this matrix has several zeros (one more than the matrix of the previous example). But in this case does not exist any permutation that transform it into a block-partitioned matrix. Therefore the function returns "?".

We say that the matrix is "irreducible"

Function MEigenvalQR(Mat)

Approximates real or complex eigenvalues of a real matrix by the QR method¹, returns an (n x 2) array

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

	A	B	C	D	E	F	G	H	I
1						λ re	λ im		
2	9	1	-1			7	0		
3	1	6	-5			8	4		
4	4	3	8			8	-4		
5									
6									
7									

{=MEigenvalQR(A2:C4)}

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

	A	B	C	D	E	F	G	H	I	J	K
1	Matrix 8 x 8									Eigenvalues (QR)	
2	2.75	1.5	1.25	1	0.75	0.5	0.25	0		1	
3	1.5	3.25	1	0.75	0.5	0.25	0	-0.25		8	
4	1.25	1	3.75	0.5	0.25	0	-0.25	-0.5		7	
5	1	0.75	0.5	4.25	0	-0.25	-0.5	-0.75		2	
6	0.75	0.5	0.25	0	4.75	-0.5	-0.75	-1		6	
7	0.5	0.25	0	-0.25	-0.5	5.25	-1	-1.25		4	
8	0.25	0	-0.25	-0.5	-0.75	-1	5.75	-1.5		3	
9	0	-0.25	-0.5	-0.75	-1	-1.25	-1.5	6.25		5	
10											
11											
12											

{=MEigenvalQR(A1:H8)}

¹ This function uses a reduction of the EISPACK FORTRAN HQR and ELMHES subroutines (April 1983) HQR and ELMHES are the translation of the ALGOL procedure. NUM. MATH. 14, 219-231(1970) by Martin, Peters, and Wilkinson.

Function MEigenvalQRC(Mat, [Cformat])

Approximates real or complex eigenvalues of a real matrix by the QR method ¹

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

Optional parameter *Cformat* sets the complex input/output format (default = 1)

See [About complex matrix format](#)

Example. Find the eigenvalues of the following complex matrix.

	A	B	C	D	E	F	G	H	I	J	K
1										λ re	λ im
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											
8											

{=MEigenvalQRC(A2:H5)}

The matrix could also be passed in compact string format

	A	B	C	D	E	F	G
8						λ re	λ im
9	4+3j	2-4j	4+5j	5-4j		4	0
10	1+2j	2	1+2j	2-j		1	3
11	-2+4j	4+2j	-2+2j	2+6j		0	-2
12	3-3j	-3-3j	3-3j	1-3j		0	1
13							
14							
15							

{=MEigenvalQRC(A9:D12,3)}

Note that the result is always in split format

Function MEigenvec(A, Eigenvalues, [MaxErr])

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

$$A \mathbf{x} = \lambda \cdot \mathbf{x}$$

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		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

{=MEigenvec(A2:C4,E2:E4)}

¹ This function uses a reduction of the EISPACK FORTRAN COMQR and CORTH subroutines (April 1983)
COMQR is a translation of the ALGOL procedure
MATH. 12, 369-376(1968) by Martin and Wilkinson.

Function MEigenvecC(A, Eigenvalue, [MaxErr])

This function returns the complex eigenvector associates with 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:

	A	B	C	D	E	F
1	Matrix A			Complex Eigenvalues		
2					real	im.
3	9	-6	7		2	0
4	1	4	1		5	1
5	-3	4	-1		5	-1
6						
7	Complex Eigenvectors					
8	real	im	real	im	real	im
9	-1	0	-2	1	-2	-1
10	-0	0	-0	1	-0	-1
11	1	0	1	0	1	0
12						
13	{=MEigenvecC(A3:C5;E3:F3)}					
14	{=MEigenvecC(A3:C5;E4:F4)}					
15	{=MEigenvecC(A3:C5;E5:F5)}					
16						
17						

The given matrix has 3 eigenvalues:

$$2, 5+j, 5-j$$

For each eigenvalue, the function **MEigenvecC** 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

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

	A	B	C	D	E	F	G	H	I	J
1	Matrix				Eigenvalue		Eigenvector			
2	-1	5	3	1	5	-1	1	-1		
3	-1	8	5	-3			0	-2		
4	{=MEigenvecC(A2:D3,F2:G2)}									
5										

Thus, the eigenvector for $I = 5-j$ is

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

Function MEigenvecJacobi(Mat, Optional MaxLoops)

This function performs Jacobi 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 U_n that transforms to a diagonal form the symmetric matrix A , for n sufficiently high

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

The matrix U_n is composed by eigenvectors of A

E2				=	{=MEigenvecJacobi(A2:C4)}		
	A	B	C	D	E	F	G
1		A			eigenvectors		
2	0	1	0		0.894	-0.446	0.031
3	1	-1	2		0.428	0.873	0.233
4	0	2	7		-0.131	-0.195	0.972

For further details see Function MEigenvalJacobi

Function MEigenSortJacobi(EigvalM, EigvectM, [num])

This function¹ sorts 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

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	Symmetric matrix					Eigenvalues diagonal matrix					Eigenvector unitary matrix					Eigenvector sorted matrix			
2	2	0	1	2		0.25	0	0	0		0.66	-0.41	0.45	0.44		0.44	0.45	-0.41	0.66
3	0	1	0	1		0	1	0	0		0.52	0.82	-0.19	0.17		0.17	-0.19	0.82	0.52
4	1	0	2	0		0	0	2.55	0		-0.38	0.41	0.82	0.11		0.11	0.82	0.41	-0.38
5	2	1	0	5		0	0	0	6.2		-0.39	0	-0.29	0.88		0.88	-0.29	0	-0.39
6																			
7	{=MEigenvalJacobi(A2:D5)}					{=MEigenvecJacobi(A2:D5)}					{=MEigenSortJacobi(F2:I5,K2:N5)}								

Function MEigenvalQL(Mat3, [IterMax])

This function returns the real eigenvalues of a tridiagonal symmetric matrix. It works also for an asymmetrical tridiagonal matrix having all real eigenvalues.

The optional parameter *Itermax* sets the max number of iteration allowed (default *Itermax* =200).

This function uses the efficient QL algorithm

If the matrix does not have all-real eigenvalues, this function returns "?"

This function accepts tridiagonal matrices in both square (n x n) and (n x 3) rectangular formats.

¹ This function appears thanks to the courtesy of Carlos Aya

	A	B	C	D	E	F	G
9	a	b	c				Eigenvalues
10	0	10	2				9.1715729
11	2	14	0				14.828427
12	3	36	6				37.065367
13	4	12	8				20.545443
14	7	15	0				5.3891907
15							
16	{=MEigenvalQL(A10:C14)}						

	A	B	C	D	E	F	G
1	Tridiagonal matrix						Eigenvalues
2	10	2	0	0	0		9.1715729
3	2	14	0	0	0		14.828427
4	0	3	36	6	0		37.065367
5	0	0	4	12	8		20.545443
6	0	0	0	7	15		5.3891907
7							
8	{=MEigenvalQL(A10:C14)}						

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 to each other, in a short interval

$$0.8 < I_k < 1.2$$

Other algorithms have difficulty in this pathological case.
On the contrary the QL algorithm works fine, giving a high general accuracy.

	A	B	C	D	E
1	a	b	c		Eigenvalues
2	0	1	0.1		0.846791111
3	0.2	1	0.1		0.871442478
4	0.1	1	0.1		0.826794919
5	0.1	1	0.1		0.812061476
6	0.1	1	0.1		0.803038449
7	0.1	1	0.1		0.8
8	0.1	1	0.1		0.9
9	0.1	1	0.1		0.931595971
10	0.1	1	0.1		0.965270364
11	0.1	1	0.1		1
12	0.1	1	0.1		1.034729636
13	0.1	1	0.1		1.068404029
14	0.1	1	0.1		1.1
15	0.1	1	0.1		1.128557522
16	0.1	1	0.1		1.153208889
17	0.1	1	0.1		1.173205081
18	0.1	1	0.1		1.187938524
19	0.1	1	0.2		1.196961551
20	0.1	1	0		1.2
21					
22	{=MEigenvalQL(A2:C20)}				

Function MEigenvalTTPz(n, a, b, c)

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

b	c	0	0	...
a	b	c	0	...
0	a	b	c	...
0	0	a	b	...
...

It has been demonstrated that:

- 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 toeplitz tridiagonal matrices, there is a nice close formula giving all eigenvalues for any size of the matrix dimension. See chapter "Tridiagonal toeplitz matrix".

Example :

find the eigenvalues of the 40 x 40 tridiagonal toeplitz 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	{=MEigenvalTtpz(A2,B2,C2,D2)}				0.204721	0
5					0.245973	0
6					0.303388	0

Find the eigenvalues of the 40 x 40 tridiagonal toeplitz 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	{=MEigenvalTtpz(A2,B2,C2,D2)}				3	-2.79528
5					3	-2.75403
6					3	-2.69661

Function MEigenvecT(Mat3, Eigenvalues, [MaxErr])

This function returns the eigenvector associated with the given eigenvalue of a tridiagonal matrix **A**

$$\mathbf{A} \mathbf{x} = \mathbf{I} \cdot \mathbf{x}$$

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	{=MEigenvecT(A2:C20,D2)}				
23					

Function MChar(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 {=MChar(A2:C4,D2)}						
	A	B	C	D	E	F	G	I
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 MCharC(A, z, [Cformat])

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

$$C = A - zI$$

where **A** can be a real or complex square matrix; z can be a 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 input/output format (default = 1)

A complex split or interlaced matrix must have always an even number of columns

Example.: Compute the matrix $A - I I$ for $I = 1 - 5j$

where **A** is the complex matrix

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

	A	B	C	D	E	F	G	H	I	J
1				A						
2		3	-1	0	6	-2	0		λ	
3		1	2	1	2	4	2		1	-5
4		-1	0	5	-2	0	10			
5										
6				A - λ I						
7		2	-1	0	11	-2	0		Determinant	
8		1	1	1	2	9	2		-906	-1370
9		-1	0	4	-2	0	15			
10										
11		{=MCharC(B2:G4,3:J3)}							{=MDetC(B7:G9)}	
12										

Example.: Compute the matrix $A - I I$ for $I = 1 + 2j$

where **A** is the real matrix

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

	A	B	C	D	E	F	G	H	I	J
1		A					A - λ I			
2	3	-1	0		2	-1	0	-2	0	0
3	1	2	1		1	1	1	0	-2	0
4	-1	0	5		1	0	4	0	0	-2
5										
6	λ =	1	2		{=MCharC(A2:C4,B6:C6)}					
7										

Function MCharPoly(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 nth degree and n+1 coefficients. As known, 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.

	A	B	C	D	E	F
1		A			degree	coeff
2	5	2	0		0	162
3	2	6	2		1	-99
4	0	2	7		2	18
5					3	-1
6						
7						

{=MCharPoly(A2:C4)}

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 through the characteristic polynomial is in general less efficient than other decomposition methods (QR, Jacobi), but becomes a good choice for low-dimension matrices (typically < 6°) and for complex eigenvalues. See also Function PolyRoots(Coefficients, [ErrMax])

Function MCharPolyC(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 nth degree and n+1 coefficients. As known, 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 input/output format (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											
8											

{=MCharPolyC(A2:H5)}

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 PolyRoots(poly)

This function returns all roots of a given polynomial

poly can be an array of (n+1) coefficients [a0, a1, a2...] or a string like "a0+a1x+a2x^2+..."

This function uses the Siljak+Ruffini algorithm for finding the roots of an nth degree polynomial

For high accuracy or for stiff polynomials you can find more suitable rootfinder routines in the Xnumbers.xla add-in

If the given polynomial is the characteristic polynomial of a matrix (returned, for example, by MCharPoly) this function returns the eigenvalues of the matrix itself.

Computing eigenvalues through the characteristic polynomial is in general less efficient than other decomposition methods (QR, Jacoby), but becomes a good choice for low-dimension matrices (typically < 6°) and/or for complex eigenvalues.

Example: find the eigenvalues of the following matrix

	A	B	C	D	E	F	G
1	Matrix A				coeff	eigenvalues	
2					52	real	im
3	9	-6	7		-46	2	0
4	1	4	1		12	5	1
5	-3	4	-1		-1	5	-1
6							
7							
8							
9							
10							

Formula boxes in the image:

- Cell D7: `{=MCharPoly(A3:C5)}`
- Cell E7: `{=PolyRoots(E2:E5)}`

Note: we can also use the nested function: `{=PolyRoots(MCharPoly(A))}`

Function MEigenvalMax(Mat, [IterMax])

Returns the dominant eigenvalue of a matrix.

A dominant eigenvalue, if it 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 iterative method

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

$$|I_1| > |I_2| > \dots |I_n|$$

Starting with a generic vector x_0 , we have:

$$y_k = A^k x_0 \quad \lim_{k \rightarrow \infty} \frac{y_k^T y_{k+1}}{y_k^T y_k} = I_1 \quad \lim_{k \rightarrow \infty} \frac{y_k}{|y_k|} = u_1$$

Tip. This algorithm is started with a random generic vector. Often it converges, but sometimes 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.

	A	B	C	D	E	F	G	H
1	Matrix 4x4					λ max	C	R
2	10	8	-5	2		16.24	4.5	11.74
3	8	4	3	-2				
4	-5	3	6	0				
5	2	-2	0	-2				
6								
7								

=F2-G2

=MTrace(A2:D5)/4

=MEigenvalMax(A2:D5)

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

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

We find the dominant eigenvalues λ_1 by the function MEigenvalMax

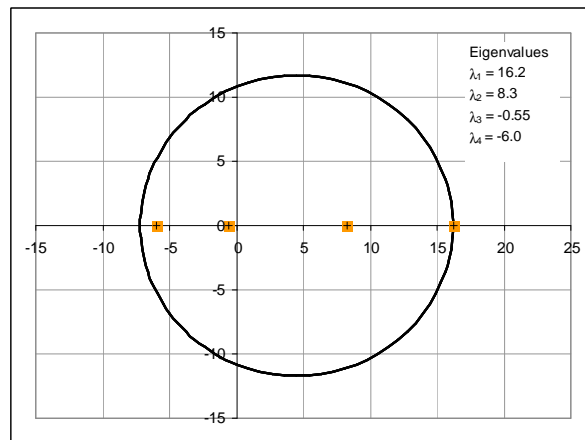
The radius can be found by the

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

We have found the center

$C = (4.5 ; 0)$ with a radius $R = 11.7$.

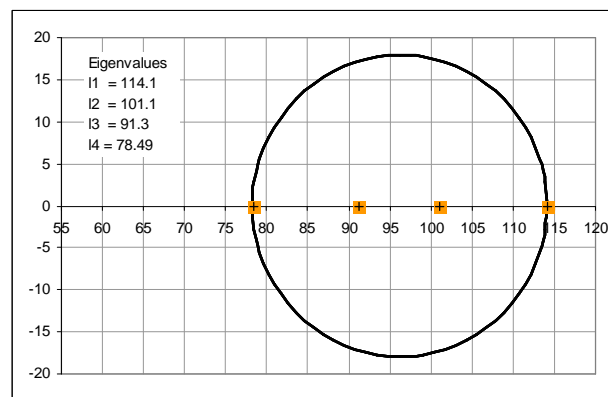
If we plot the circle and the roots, we observe a general good result



This method works also for asymmetric 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 MEigenvecMax(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 iterative power method
For further details see the function MatEigenvalue_Max

Function MEigenvalPow(Mat, [IterMax])

This function returns all real eigenvalues of a given diagonalizable 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 iterative power method. This algorithm works also for asymmetric matrices having real eigenvalues

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	{=MEigenvecpow(A2:D5)}								
7						5	4	3	2
8	{=MEigenvalpow(A2:D5)}								
9									

We have used the functions MEigenvalPow and MEigenvecPow. We can see the small values instead of 0. This is due to the round-off errors. If you want to clean the matrix, remove these round-off errors with the function MMopUp

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

This function returns all eigenvectors of a given diagonalizable 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 iterative power method. This algorithm works also for asymmetric matrices having real eigenvalues

See also function MEigenvalPow

	A	B	C	D	E	F	G
1	600	-600	0		0.273	0.7394	0.8133
2	-400	1200	-800		-0.6941	-0.4485	0.5275
3	0	-600	1500		0.6661	-0.5021	0.2455
4							
5	{=MEigenvecpow(A1:C3,TRUE)}						
6							

Function MEigenvecInv(Mat, Eigenvalue)

This function returns the eigenvector associated with an eigenvalue of a matrix **A** using the inverse iterative algorithm

$$\mathbf{A} \mathbf{x} = \lambda \cdot \mathbf{x}$$

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 errors, because it is more stable than the singular system resolution.

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

	A	B	C	D	E	F	G	H	I	J
1	Matrix				Eigenvalues			Eigenvectors		
2	4	-2	1		3			0.5774	0.7071	0
3	3	-1	1		2			0.5774	0.7071	0.4472
4	4	-4	3		1			0.5774	0	0.8944
5										
6	{=MEigenvecInv(A2:C4,E2:E4)}							1	1	0
7								1	1	0.5
8	{=MNormalize(H2:J4,3)}							1	0	1
9										

Function MEigenvecInvC(Mat, Eigenvalue, [CFormat])

This function returns the eigenvector associated to a complex eigenvalue of a complex matrix **A** by the inverse iteration algorithm

$$\mathbf{A} \mathbf{x} = \lambda \cdot \mathbf{x}$$

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 errors, because it is more stable than the singular system resolution of the function MEigenvecC .

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

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

	A	B	C	D	E	F	G	H	I	J	K
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	{=MEigenvecInvC(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

Often 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 to each other, having only a 4% of difference. In this

case a small error in the eigenvalues could cause a large error in the eigenvectors. In this situation the inverse iterative algorithm is handy. It has great stability. Let's see this example
First of all we define a sensitivity coefficient for measuring the instability.

Instability Sensitivity

$$S_{u,l} = \frac{\sum |u_i - u_i^*|}{\|u\|} \cdot \frac{|I|}{|I - I^*|} = \frac{\sum |\Delta u_i|}{\|u\|} \cdot \frac{|I|}{|\Delta I|}$$

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 MEigenvec() function while the last one is used by the MEigenvec_inv function.

Singular linear system method		Iterative inverse method	
λ	$\Delta \lambda$	λ	$\Delta \lambda$
100	0.0001	100.3	0.3
u	$ \Delta u $	u	$ \Delta u $
1	0.99981665	1.00000000	0
12	12.00028336	12.00000085	7.12E-08
7	7.00005834	7.00000046	6.58E-08
-3	-3.00003333	-3.00000020	6.58E-08
-1	-1.00000000	-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 substantial (0.3%). On the other hand, the first method computes a sufficiently accurate eigenvector only if the eigenvalue error is very small (0.0001%). Note that for larger errors the first method fails, returning the null vector.

On the contrary, the iterative inverse algorithm tolerates a large amount of error in the eigenvalue. This can be shown by the instability factor

Singular linear system method				Iterative inverse method			
λ	$\Delta \lambda$	$ u $	$\Sigma \Delta u $	λ	$\Delta \lambda$	$ u $	$\Sigma \Delta u $
100.0001	0.0001	14.28	0.000226	100.3	0.3	14.28	2.69E-07
S1 = 15.85				S2 = 6.3E-06			

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

Matrix Generator

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

Function MRnd(n, [m], [Typ], [MatInteger], [Amax], [Amin], [Sparse])

Function MRndEig(Eigenvalues, [MatInteger])

Function MRndEigSym(Eigenvalues)

Function MRndRank(n, [Rank], [Det], [MatInteger])

Function MRndSym(n, [Rank], [Det], [MatInteger])

Function MHilbert(n)

Function MHilbertInv(n)

Function MHouseholder(x)

Function MTartaglia(n)

Function MVandermonde(x)

Function MRnd(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				symmetric				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	=MRnd(4)				=MRnd(4,,"sym")				=MRnd(4,,"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	=MRnd(4,,"tlw")				=MRnd(4,,"tup")				=MRnd(4,,"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		=MRnd(5,,"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		=MRnd(5,,"trd",,2,1)				

Note: The generation is random; it's means that each time that you recalculate this function, you get different values. If you don't want change the values, preserve them with Edit > PasteSpecial > Values.

Function MRndEig(Eigenvalues, [MatInteger])

Returns a general real matrix with a given set of eigenvalues

Function MRndEigSym(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					symmetric 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			{=MRndEig(A2:A5)}					{=MRndEigSym(A2:A5)}			

Function MRndRank(n, [Rank], [Det], [MatInteger])

Returns a real matrix with a given Rank or Determinant

Note: if Rank < max dimension then always Det = 0

Function MRndSym(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 MRndUni(n, [MatInteger])

Returns an unitary matrix (Det=1)

Function MHilbert(n)

Returns the (n x n) Hilbert's matrix

Note: this matrix is always decimal

Function MHilbertInv(n)

Returns the inverse of the (n x n) Hilbert's matrix.

Note: this matrix is always integer

Hilbert matrices are strongly ill-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		{=MHilbert(4)}					{=MHilbertInv(4)}			
7										

Function MHouseholder(x)Returns the Householder matrix of a given vector $\mathbf{x} = (x_1, x_2, \dots, x_n)$ 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

	A	B	C	D	E	F	G	H
18								
19		2		0.765	-0.12	-0.24	0	0.588
20		1		-0.12	0.941	-0.12	0	0.294
21		2		-0.24	-0.12	0.765	0	0.588
22		0		0	0	0	1	0
23		-5		0.588	0.294	0.588	0	-0.47
24								
25								
26				{=MHouseholder(B19:B23)}				

Function MTartaglia(n)

Returns the Tartaglia matrix of order n

This kind of matrix (also called Pascal matrix) is ill-conditioned and is useful for testing of algorithms

In the example below we see a (5 x 5) matrix

	C	D	E	F	G	H
	1	1	1	1	1	1
	1	2	3	4	5	6
	1	3	6	10	15	21
	1	4	10	20	35	56
	1	5	15	35	70	126
	1	6	21	56	126	252

{=MTartaglia(6)}

Definition: Tartaglia's matrix is defined as

$$a_{1j} = 1$$

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

Function MVandermonde(x)Returns the Vandermonde matrix of a given vector $\mathbf{x} = (x_1, x_2, \dots, x_n)$

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

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

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4$$

We can find the coefficients $\mathbf{a} = (a_0, a_1, a_2, a_3, a_4)$ solving the linear system $\mathbf{V} \mathbf{a} = \mathbf{y}$ Where \mathbf{V} is the Vandermonde's matrix

x	y	Vandermonde matrix						
-2	600	1	-2	4	-8	16	a0 =	460
-1	521	1	-1	1	-1	1	a1 =	-50
1	423	1	1	1	1	1	a2 =	12
4	516	1	4	16	64	256	a3 =	1
6	808	1	6	36	216	1296	a4 =	0

{=MVandermonde(A2:A6)} {SysLin(D2:H6,B2:B6)}

Function GJstep(Mat, [Typ], [IntValue], [tiny])

This function, also available in macro version, has been developed for its 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" for Diagonal or "T" for Triangular; the default is D.

Optional parameter *IntValue* = TRUE forces the function to conserve integer values through all steps. Default is FALSE.

Default is FALSE.

Optional parameter *tiny* set the minimum round-off error. (default 2E-15)

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

$$A = \begin{bmatrix} 0 & -10 & 3 \\ 2 & 1 & 1 \\ 4 & 0 & 5 \end{bmatrix} \quad b = \begin{bmatrix} 105 \\ 17 \\ 91 \end{bmatrix}$$

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

	A	B	C	D	E
1	0	-10	3	105	
2	2	1	1	17	
3	4	0	5	91	
4					
5					
6					
7					
8					

	A	B	C	D
1	0	-10	3	105
2	2	1	1	17
3	4	0	5	91
4				
5	4	0	5	91
6	2	1	1	17
7	0	-10	3	105
8	={GJstep(A1:D3)}			

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 with CTRL+C

Move to the cell A9, and give the command CTRL+V. The next step will be performed. Continuing in this way we can visualize each step of the elimination algorithm

	A	B	C	D
1	0	-10	3	105
2	2	1	1	17
3	4	0	5	91
4				
5	4	0	5	91
6	2	1	1	17
7	0	-10	3	105
8	={GJstep(A1:D3)}			
9	4	0	5	91
10	0	1	-1.5	-28.5
11	0	-10	3	105
12	={GJstep(A5:D7)}			

	A	B	C	D
4				
5	4	0	5	91
6	2	1	1	17
7	0	-10	3	105
8	={GJstep(A1:D3)}			
9	4	0	5	91
10	0	1	-1.5	-28.5
11	0	-10	3	105
12	={GJstep(A5:D7)}			
13	4	0	5	91
14	0	-10	3	105
15	0	1	-1.5	-28.5
16	={GJstep(A9:D11)}			

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

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

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

Example. Invert the 3x3 matrix by the Gauss-Jordan method

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 6 \end{bmatrix}$$

For inversion add the 3x3 identity matrix adjacent to the right.

	A	B	C	D	E	F
1						
2	1	1	1	1	0	0
3	1	2	3	0	1	0
4	1	3	6	0	0	1
5	{=GJstep(A2:F4,,True)}					
6	1	1	1	1	0	0
7	0	1	2	-1	1	0
8	1	3	6	0	0	1
9	{=GJstep(A6:F8,,True)}					
10	1	1	1	1	0	0
11	0	1	2	-1	1	0
12	0	2	5	-1	0	1

13	{=GJstep(A10:F12,,True)}					
14	1	1	1	1	0	0
15	0	2	5	-1	0	1
16	0	1	2	-1	1	0
17	{=GJstep(A14:F16,,True)}					
18	2	0	-3	3	0	-1
19	0	2	5	-1	0	1
20	0	1	2	-1	1	0
21	{=GJstep(A18:F20,,True)}					
22	2	0	-3	3	0	-1
23	0	2	5	-1	0	1
24	0	0	-1	-1	2	-1

25	{=GJstep(A22:F24,,True)}					
26	-2	0	0	-6	6	-2
27	0	2	5	-1	0	1
28	0	0	-1	-1	2	-1
29	{=GJstep(A26:F28,,True)}					
30	-2	0	0	-6	6	-2
31	0	2	0	-6	10	-4
32	0	0	-1	-1	2	-1
33	{=GJstep(A30:F32,,True)}					
34	1	-0	-0	3	-3	1
35	0	1	0	-3	5	-2
36	0	0	1	1	-2	1

Thus, the inverse matrix is

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

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.

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad \text{or} \quad \mathbf{A} \mathbf{X} = \mathbf{B}$$

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

The optional parameters: IMode (default False) sets the floating point (False) or integer arithmetic (True). Integer computation is intrinsically more accurate when the original matrix A contains only integer elements, but it is also more limited because it may easily reach the overflow limit. Therefore, use IMode only with integer matrices of low size.

The optional parameter Tiny (default is 1E-15) 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 square the function returns "?"

As known, the above linear equation has only one solution if - and only if -, $\text{Det}(\mathbf{A}) \neq 0$

Otherwise the solutions can be infinite or even non-existing. In that case the system is called "singular".

Example.

	A	B	C	D	E	F	G	H	I
1	Linear System $\mathbf{A} \mathbf{x} = \mathbf{b}$								
2									
3		1	5	0	-2		-256		134
4		2	-1	10	3		2366		2
5		1	7	4	2		1148		150
6		4	1	0	1		738		200
7									
8									
9									

{=SysLin(B3:E6,G3:G6)}

The parameter **b** can also be a matrix of m columns. In that case, SysLin simultaneously solves a set of m systems.

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

Example.

	A	B	C	D	E	F	G	H	I	J
1	A					B			X	
2		3	-2	0	1		9	5	2	1
3		4	5	-1	6		8	-3	-1	-1
4		4	0	1	1		10	6	1	2
5		0	-1	-1	1		1	-1	1	0
6										
7										
8										

{=SysLin(A2:D5,B2:G5,TRUE)}

Note that in this case we have set the IMode = True, that forces the integer exact algorithm. In floating point the approximate average error of the solution would be about 1E-14

Function SysLin3(Mat3, y)

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:

$$\mathbf{A} \mathbf{x} = \mathbf{y}$$

A is the system square matrix (n x n)

x is the unknown vector (n x 1)

y is the constant terms vector (n x 1)

As known, the above linear equation has only one solution if - and only if -, $\det(\mathbf{A}) \neq 0$

Otherwise the solutions can be infinite or even non-existing. 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's see how to solve a 16 x 16 tridiagonal linear system $\mathbf{A} \mathbf{x} = \mathbf{y}$

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 us to solve systems larger than 256 x 256 (the max square matrix in a pre-2007 Excel worksheet)

	A	B	C	D	E	F	G
1	Matrix A (16 x 16)						
2	a	b	c		y		x
3	0	2	-1		0		1
4	-1	3	2		11		2
5	-1	3	0		7		3
6	2	4	1		27		4
7	2	5	2		45		5
8	1	6	-1		34		6
9	-1	7	5		83		7
10	-1	9	-1		56		8
11	-1	7	-1		45		9
12	-1	5	-1		30		10
13	-1	3	0		23		11
14	2	2	1		59		12
15	2	2	0		50		13
16	-2	2	-1		-13		14
17	-1	2	3		64		15
18	-1	2	0		17		16
19							
20							
21							
22	{=SysLin3(A3:C18,E3:E18)}						

Function SysLinIterG(A, b, x0, [Nmax], [w])

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

$$\mathbf{A} \mathbf{x} = \mathbf{b}$$

Parameter **A** is the system matrix (range n x n)

Parameter **b** is the system vector (range n x 1)

Parameter **x₀** is the starting approximate solution vector (range n x 1)

Parameter **Nmax** is the max step allowed (default = 200)

Parameter **w** is the relaxation factor (default = 1)

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

In the example below the 20th iteration step of this iterative method is shown.

As we can see, the values approximate the exact solution [4, -3, 5]. Precision increases with an increasing number of steps (of course, for a 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								
8								

↓ ↓ ↓ ↑

{=SysLinIterG(A3:C5, D3:D5, G3:G5, H2)}

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

```

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

```

In the example below we see the trace of the iteration values

	B	C	D	E	F	G	H	I
	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	{=SysLinIterG(\$B\$4:\$D\$6;\$E\$4:\$E\$6;B10:D10)}				
	2,51111	-1,1683	3,60317	{=SysLinIterG(\$B\$4:\$D\$6;\$E\$4:\$E\$6;B11:D11)}				
	4,7709	-3,977	5,74039	{=SysLinIterG(\$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 extrapolation formula, also called as "*square delta*" extrapolation, or by tuning the relaxaction factor w.

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

This function solves a triangular linear system by the forward and backward substitution algorithms.

$$\mathbf{A} \mathbf{x} = \mathbf{b}$$

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 -, $\det(\mathbf{A}) \neq 0$

Otherwise the solutions can be infinite or even non-existing. 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 from solving for the 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 1E-15) sets the minimum round-off error; any absolute value 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
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											
12											

{=SysLinT(A2:G8, I2:I8)}

Function SysLinIterJ(A, b, x0, [Nmax])

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

$$\mathbf{A} \mathbf{x} = \mathbf{b}$$

Parameter **A** is the system matrix (range n x n)

Parameter **b** is the system vector (range n x 1)

Parameter **x₀** 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 SysLinIterG function.

For further details see Function SysLinIterG

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

Singular linear system can have infinitely many solutions or even none. This happens when $\text{DET}(\mathbf{A}) = 0$. In that case the following matrix equations:

$$\mathbf{A} \mathbf{x} = 0 \quad \text{or} \quad \mathbf{A} \mathbf{x} = \mathbf{b}$$

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

$$\mathbf{y} = \mathbf{C} \mathbf{x} + \mathbf{d}$$

where **C** is the transformation matrix and **d** is the known vector; **C** is a square matrix having the same columns of **A**, and **d** the same dimension 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 "?"

The optional parameter MaxErr sets the relative precision level; elements lower than this level are forced to zero. Default is 1E-13.

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

Example: Solve the following system

	A	B	C	D	E	F	G	H	I
1		A		b					
2	1	0	-8	3		0	0	8	3
3	-1	1	10	2		0	0	-2	5
4	0	1	2	5		0	0	1	0
5									
6	{=SysLinSing(A2:C4;D2:D4)}								
7									

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

$$y = Cx$$

	A	B	C	D	E	F
1		rank = 2				
2		1	8	10		
3	A =	0	1	7		$Ax = 0$
4		-2	-16	-20		
5						
6		0	0	46		
7	C =	0	0	-7		$y = Cx$
8		0	0	1		
9						
10						
11						
12						

Looking at the first diagonal of the matrix **C** we discover a 1 in column 3, row 3. This means that there is only one independent variable; all the others are dependent variables. This means also that the rank of the 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 numerical solution of the given system
Example, setting $x_3 = 1$ we have $y = [46, -7, 1]^T$, $x_3 = -1$ we have $y = [-94, 14, -2]^T$ and so on.

Example 2: Find the solution of the following homogeneous system

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

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 the matrix is 1

	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		
9						

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

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 y_1, y_2, y_3 with the usually variables x, y, z , we get:

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

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

$$\mathbf{A} \mathbf{x} = \mathbf{b}$$

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

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

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{d}$$

	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,E2:E5)}			
13					

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

Function SysLinTpz(A, b)

Solves a Toeplitz linear system by Levinson method

Parameter **A** is a Toeplitz matrix that can be written as a common square matrix (n x n) or also in a compact and efficient vector form (2n-1). See [Toeplitz matrices](#) for further details

Parameter **b** is the vector (n) of constant terms

Example:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1																	
2		Tpz		b		x	Toeplitz matrix					b		x			
3		-2		133		10	5	4	3	-1	-1	133		10			
4		-1		335		22	2	5	4	3	-1	335		22			
5		-3		494		30	-3	2	5	4	3	494		30			
6		2		409		45	-1	-3	2	5	4	409		45			
7		5		208		50	-2	-1	-3	2	5	208		50			
8		4															
9		3															
10		-1															
11		-1															

The SysLinTpz function saves more than 700% of the elaboration time. It is suitable for large matrices. But of course there is also a drawback: not all Toeplitz linear system can be computed. Sometime the algorithm fails and returns "?" even if the Toeplitz matrix is not singular. When this happens we have to come back to the SysLin function

Levinson's method has been demonstrated to be successful when the matrix is diagonal dominant

Function TraLin(A, x, [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

	A	B	C	D	E	F	G	H	I
1		A			x		b		y
2	1	2	4		1		9		6
3	2	3	-1		2		-1		9
4	-1	0	1		-2		2		-1
5									
6									
7									

Formula bar: {=TraLin(A2:C4,E2:E4,G2:G4)}

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 transformations have a useful geometric interpretation¹.

Take a point $x(x_1, x_2)$ of the plane and compute the linear transform $y = [A]x$, where **A** (2 x 2) is a matrix, $y(y_1, y_2)$ a point. We wonder if there is a geometrical relation between the points x and y .

The relation exists, and becomes evident once we perform the transformation of the points belonging to the unitary circle.

In Excel we can easily generate the unitary circle pattern with the formula

$$x = (\cos(k \cdot \Delta\alpha), \sin(k \cdot \Delta\alpha)) \quad \text{for } k = 1, 2 \dots N \quad \text{where } \Delta\alpha = 2\pi/N$$

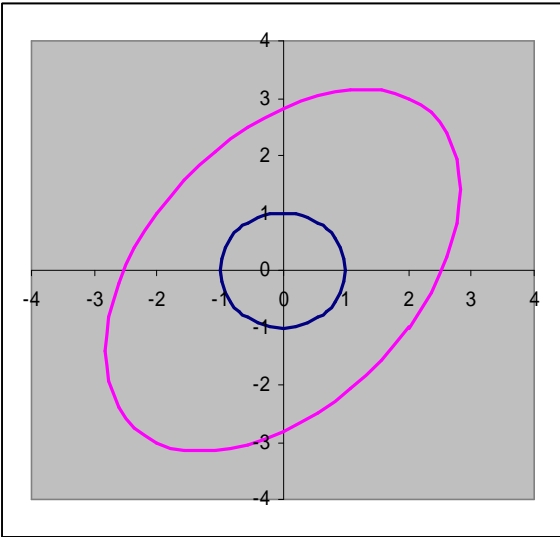
If x is a row-vector it is useful to have the dual Linear Transform for row-vectors

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

Here is a possible arrangement.

¹ 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			2	2			
3	A =		-1	3			
4							
5	N =	32	$\Delta\alpha =$	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	



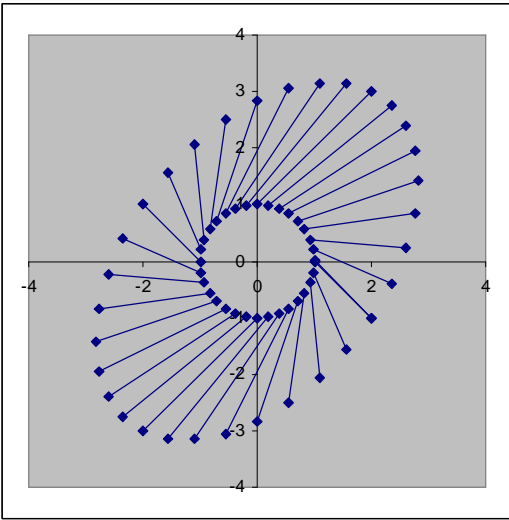
The blue line corresponds to the unitary circle points; the pink line belongs to the transformed y points. Thus, the circle has been transformed into a centred ellipse; if we also add $b \neq 0$, we get a translated ellipse. Each point of the unitary circle has been projected on the ellipse.

We have to point out that the projection is not radial but it happens in a very strange way. Look at the image to the right. It shows how a point on the circle moves to the ellipse by 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 name – in higher dimension

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



Function MOrthoGS(A)

This function performs orthogonalization with the modified Gram-Schmidt method
Argument **A** is a matrix (m x n) containing n independent vectors.
This function returns the orthogonal matrix **U** = [u₁, u₂, ..u_n]

$$U = (v_1, v_2, v_3, \dots, v_n)$$

Where:

$$v_i \bullet v_j = \begin{cases} 1 & \Leftrightarrow i = j \\ 0 & \Leftrightarrow i \neq j \end{cases}$$

In this example, we determine an orthogonal basis for the subspace generated by the base-vectors

$$[\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3] = \begin{bmatrix} 1 & 2 & -1 \\ 1 & -1 & 2 \\ 1 & -1 & 2 \\ -1 & 1 & 1 \end{bmatrix}$$

	A	B	C	D	E	F	G	H	I	J	K
1	Base				Gram-Schmidt				QR factoriz.		
2	1	2	-1		0.5	0.866	0		0.5	0.866	-0
3	1	-1	2		0.5	-0.289	0.408		0.5	-0.289	0.408
4	1	-1	2		0.5	-0.289	0.408		0.5	-0.289	0.408
5	-1	1	1		-0.5	0.289	0.816		-0.5	0.289	0.816
6					{=MOrthoGS(A2:C5)}				{=MQR(A2:C5)}		
7											
8											

Note that we can build the orthogonal basis by two different processes: using the Gram-Schmidt algorithm or the QR factorization.

Gram-Schmidt Orthogonalization

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

$$(v_1, v_2, v_3, \dots, v_n)$$

The orthogonal basis U is build with the following iterative algorithm

For $k = 1, 2, 3 \dots n$

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

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

Developing this algorithm, we see that the vector k is built from all $k-1$ previous vectors
At the end, all vectors of the bases U will be orthogonal and normalized.

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

The so-called "modified Gram-Schmidt" algorithm, shown here, is a more stable variant

For $k = 1, 2, 3 \dots n$

$$q_k = v_k^{(k)} / \|v_k^{(k)}\|$$

For $j = k+1, \dots n$

$$v_j^{(k+1)} = v_j^{(k)} - (v_j^{(k)} \bullet q_k) q_k$$

Function MCholesky(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:

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

or, in other words, when the eigenvalues of **A** are all-positive. This function always returns a matrix.

Inspecting the diagonal elements of the returned matrix we can discover if the matrix is positive definite: if the diagonal elements are all positive then the matrix **A** is also positive definite.

Example - Determine if the given matrices are positive definite

A			B		
3	1	2	5	2	1
1	6	4	2	2	3
2	4	7	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 the eigenvalues are all positive.

On the contrary, the decomposition of the matrix **B** shows a negative number at position a_{33} ; then we can say that **B** is not positive definite and at least one eigenvalues is negative.

	A	B	C	D	E	F	G
1	Cholesky Decomposition						
2							
3		A				B	
4	3	1	2		5	2	1
5	1	6	4		2	2	3
6	2	4	7		1	3	1
7		L				L	
8	1.7321	0	0		2.2361	0	0
9	0.5774	2.3805	0		0.8944	1.0954	0
10	1.1547	1.4003	1.9251		0.4472	2.3735	-4.8333
11							
12							
13	{=MCholesky(A3:C5)}				this element is negative :		
14							

This decomposition is useful also for solving the so-called "*generalized eigenproblem*"

Function MLU(A, optional Pivot)

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

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

Where **L** is a lower triangular matrix, and **U** is an 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					{=MLU(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					{=MLU(A7:C9, False)}							{=MMULT(E7:G9,H7:J9)}			

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

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
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					{=MLU(A20:C22,FALSE)}									

As we can see, in that case, the matrix product **B** is different from the given matrix **A**. We will see how to solve the problem by introducing a permutation matrix.

LU and Linear System. The LU decomposition is often used to solve linear system

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \mathbf{L} \mathbf{U} \mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \mathbf{L} (\mathbf{U} \mathbf{x}) = \mathbf{b}$$

The original system is now split into two simpler systems.

$$\mathbf{L} \mathbf{y} = \mathbf{b} \quad (1) \quad \mathbf{U} \mathbf{x} = \mathbf{y} \quad (2)$$

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.

$$y_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} y_j \right) \quad \text{For } i = 1, 2, \dots, N$$

Forward substitution. For lower triangular system like $\mathbf{L} \mathbf{y} = \mathbf{b}$

$$x_i = \frac{1}{b_{ii}} \left(y_i - \sum_{j=i+1}^N b_{ij} x_j \right) \quad \text{For } i = N, N-1, \dots, 2, 1$$

Backward substitution. For upper triangular system like $\mathbf{U} \mathbf{x} = \mathbf{y}$

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

When pivoting is activate the right decomposition formula is $\mathbf{A} = \mathbf{P} \mathbf{L} \mathbf{U}$, where **P** is a permutation matrix. This function can return also the permutation matrix in the last n columns. Globally, the output of MLU 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**

	A	B	C	D	E	F	G	H	I	J	K	L
1		A			L			U			P	
2	4	1	1	1	0	0	8	4	1	0	0	1
3	8	4	1	-0.5	1	0	0	5	-3.5	1	0	0
4	-4	3	-4	0.5	-0.2	1	0	0	-0.2	0	1	0
5												
6												
7												

{=MLU(A2:C4)}

Example: find the factorization of matrix **A** with $a_{11} = 0$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1		A				L			U			P				PLU	
2	0	1	2		1	0	0	2	4	7	0	0	1		0	1	2
3	1	6	4		0.5	1	0	0	4	0.5	0	1	0		1	6	4
4	2	4	7		0	0.25	1	0	0	1.875	1	0	0		2	4	7
5																	
6																	
7																	

=MLU(A2:C4) **=MPROD(K2:M4;E2:G4;H2:J4)**

Note that the permutation matrix is returned in the last 3 columns just for practical reason. Often the user is not interested in the matrix **P** (for example if the pivot = false then **P** = identity matrix). Selecting only the first 6 columns and pasting the function **MLU**, the **P** matrix stays hidden. Remember that, in the matrix product **PLU**, the permutation matrix must always be the first (see image).

Function MQR(Mat)

This function performs the QR decomposition of a square ($n \times n$) matrix

$$A = Q \cdot R$$

A is a rectangular ($m \times n$) matrix¹, with $m \geq n$.

Q is an orthogonal ($m \times n$) matrix

R is an upper triangular ($n \times n$) matrix.

This function returns a matrix ($m \times (n + n)$), where the first ($m \times n$) block is **Q** and the first n rows of the second ($m \times n$) block is **R**. The last $m - n$ rows of the second block are all zero.

The QR decomposition forms the basis of an efficient method for calculating the eigenvalues. See also the function **MEigenvalQR(Mat, Optional MaxLoops, Optional Acc)**

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

	A	B	C	D	E	F
1	-2	-4	-6			
2	1	3	2			
3	2	2	5			
4						
5	-0.66667	0.33333	0.66667	3	5	8
6	0.33333	-0.66667	0.66667	0	-2	-2.2E-16
7	0.66667	0.66667	0.33333	0	2.2E-16	-1

{=MQR(A1:C3)}

¹ Thanks to Ola Mårtensson for the rectangular version of QR algorithm

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

	A	B	C	D	E	F	G	H	I	J
1	Matrix A (5 x 3)				Matrix Q (5 x 3)			Matrix R (3 x 3)		
2	1	2	-1		-0.32	0.646	-0.68	-3.16	-2.85	1E-16
3	2	1	-1		-0.63	-0.47	-0.17	0	1.703	1.174
4	-1	-1	-1		0.316	-0.059	-0.36	0	9E-17	2.573
5	0	1	2		0	0.587	0.509	0	0	0
6	2	2	1		-0.63	0.117	0.335	0	0	0
7										
8	{=MQR(A1:C5)}									

Function MQRiter(Mat, [MaxLoops])

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

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

If the matrix **A** has: $|\lambda_1| > |\lambda_2| > |\lambda_3| > \dots > |\lambda_n|$, then the sequence converges to the diagonal matrix of eigenvalues

$$\lim_{n \rightarrow \infty} A_{n+1} = [I]$$

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.

	A	B	C	D	E	F	G
1							
2	1	3	2		10.15901	1.504833	-0.197191
3	2	9	1		4.97E-63	2.349349	0.834621
4	3	2	1		2.16E-82	-2.31E-19	-1.508356
5							
6	{=MQRiter(A2:C4)}				10.15901	1.504833	-0.197191
7					0	2.349349	0.834621
8					0	0	-1.508356

Function MQH(A, b)

This function performs the **QH** decomposition of the square matrix **A** with the vector **b**. The function returns a matrix (n x 2n), where the first (n x n) block contains **Q** and the second (n x n) block contains **H**.

$$A = Q H Q^T$$

where **Q** is an orthogonal matrix and **H** is an Hessenberg matrix.

If **A** is symmetric, **H** is a tridiagonal matrix.

This function uses the Arnoldi-Lanczos algorithm.

	A	B	C	D	E	F	G	H	I	J
1		A				b				
2	2	3	-1	6		1				
3	5	0	1	2		0				
4	1	4	8	-1		1				
5	-1	1	3	3		0				
6										
7	Matrix Q				Matrix H					
8	0.71	-0.47	-0.27	0.45	5	5.67	0.28	2.97		
9	0	0.71	-0.6	0.37	6	1.56	-0.47	-1.8		
10	0.71	0.47	0.27	-0.45	0	4.19	0.88	-3.9		
11	0	0.24	0.7	0.67	0	0	3.1	5.56		

Function MExtract(A, i_pivot, j_pivot)

Returns the sub-matrix extracted from **A** by eliminating one row and one column.

i_pivot = row to eliminate

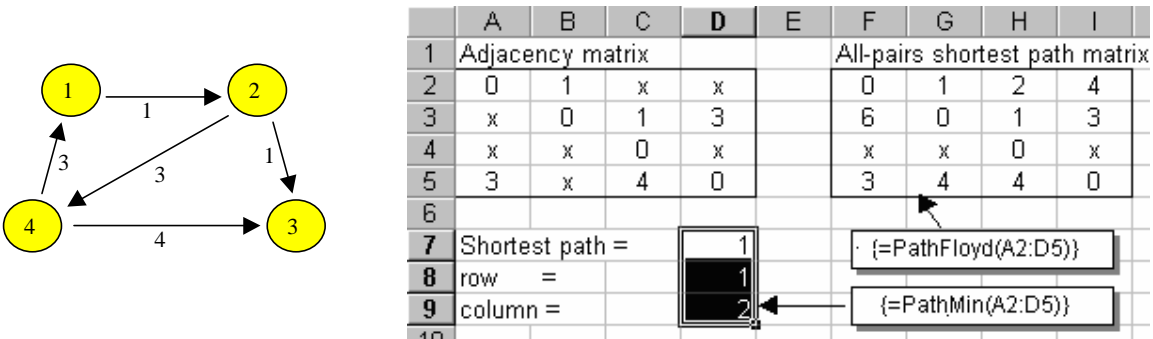
j_pivot = column to eliminate

	A	B	C	D	E	F	G	H	I
1									
2		1	8	10	1		i pivot=	2	
3		0	1	7	-4		j pivot=	3	
4		-2	-1	-20	2				
5		4	-1	2	3				
6									
7		1	8	1					
8		-2	-1	2					
9		4	-1	3					

Function PathFloyd(G)

This function, now available also in macro version, returns the matrix of all shortest-path pairs 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 paths between all pairs of vertices in a graph that can be represented as an adjacency matrix **[G]** in which each element a_{ij} - called node - represents the "distance" between element i and j . If there is no link between two nodes, we leave the cell blank or we can set any non-numeric symbol you like: for example "x"
This function uses *Floyd sequential algorithm*

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



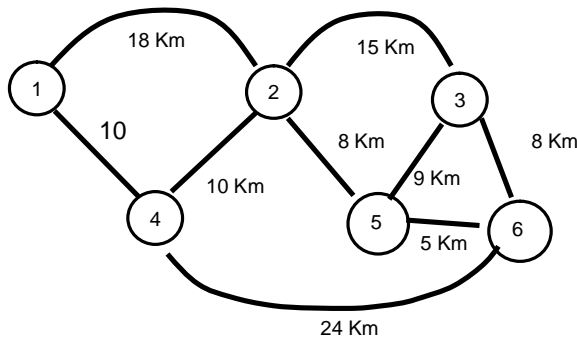
Function PathMin(G)

Returns a vector containing the shortest paths of a graph; the row and column of the cell
This function uses the PathFloyd() function to find the all-pairs shortest paths of the given graph **G**

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

Graphs theory recalls

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



The map can be expressed in 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 the 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 the matrix.

The above "adjacent matrix" reports only the direct distances between each pair of cities.

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

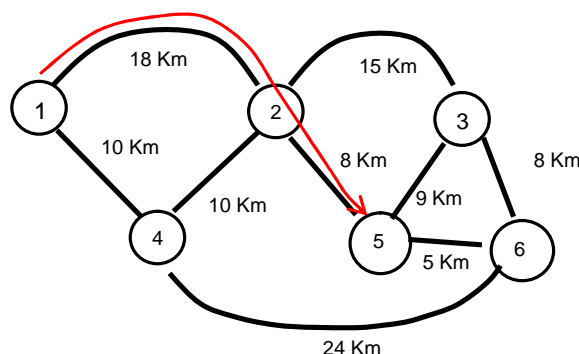
We can repeat this search for any other pair, and find the shortest path for all pairs of cities. But it will be tedious. The Floyd algorithm automates just this task. Applying this algorithm to the above matrix we get the following matrix

	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 pair of cities

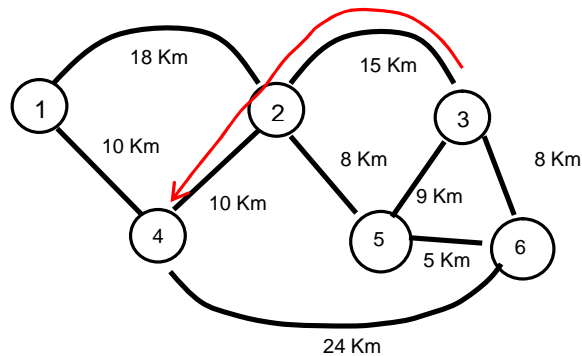
For example the shortest distance between city 1 and city 5 is 26 Km

	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

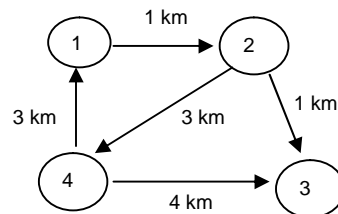
	city 1	city 2	city 3	city 4	city 5	city 6
city 1	0	18	33	10	26	31
city 2	18	0	15	10	8	13
city 3	33	15	0	25	9	8
city 4	10	10	25	0	18	23
city 5	26	8	9	18	0	5
city 6	31	13	8	23	5	0



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

The problem is more difficult if the paths are "oriented"; for example if one or more ways are only one-directional

Let see this example



The adjacent matrix is built in the same way; the only difference is that in this case the matrix is asymmetric.

For example between the node 1 and node 2 there is a direct path of 1 km, but the reverse is not true

	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

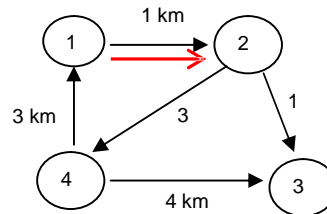
Applying the Floyd algorithm we get the following matrix

	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

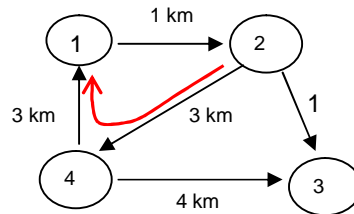
Reading this matrix is simple:

To go from node 1 to the node 2 there's the shortest path of 1 km; to return 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 no path to reach any other nodes. The row of node 3 has all "x"s (meaning no paths) except for itself. But it can be reached from all other nodes.

Let's see how use this array function in Excel

Shortest path

First of all, write the adjacent matrix (we have also shown column and row headers but they are dispensable)

P19									
	A	B	C	D	E	F	G	H	I
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		
8									
9									
10									

Now choose the site of the shortest-path matrix; that is the matrix returned by function PathFloyd. It must be inserted 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 "ctrl+shift+enter" keys sequence
That is:

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

The solution values fill all the cells that you have selected.

Note that Excel shows the function surrounded by two braces { }

These symbols mean that the function returns an array.

The matrix returned is the shortest-path matrix

B10		{=PathFloyd(B2:G7)}					
	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
8							
9							
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

Singular Value Decomposition -

Function SVDU(A)

Function SVDD(A)

Function SVDV(A)

Singular Value Decomposition of a (n x m) matrix **A** provides three matrices, **U**, **D**, **V** performing the following decomposition¹:

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

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.

Example.

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

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

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

Example. Find the SVD decomposition for the given matrix

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		A				U			D			V	
2	1	1	-5		0.57	0.77	-0.3	7.95	0	0	-0.3	0.92	0.29
3	2	0	1		-0.2	0.45	0.88	0	3.29	0	0.37	-0.2	0.91
4	3	-3	5		-0.8	0.45	-0.4	0	0	1	-0.9	-0.4	0.3
5													
6					{=SVDU(A2:C4)}			{=SVDD(A2:C4)}			{=SVDV(A2:C4)}		

From the **D** matrix of singular values we get the max and min values to compute the condition number m^2 , used to measure the ill-conditioning of a matrix. In fact, we have:

$$m = 7.95 / 1 = 7.95$$

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

¹ Some authors give a different definition for SVD decomposition, but the main concept is the same.

² With 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												

Sometimes 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 greater than a small value, usually $1\text{E-}13$). For this we need only the singular values matrix returned by the function SVDD. Let's see.

H2		=SVDD(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 values different from zero.

Nomenclature: The matrices returned by the SVD are sometimes called

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

So the decomposition for a matrix A can be written as¹

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

Condition Number

This number is conventionally used to indicate how **ill-conditioned** a matrix is

Formally the condition number of a matrix is defined by the SVD decomposition as the ratio of the largest element to the smallest element of diagonal matrix

Given the SVD decomposition:

$$A = UDV^T \quad \text{where:} \quad D = \text{diag}[d_1, d_2 \dots d_n]$$

The condition number is defined as: $k = \max_i(d_i) / \min_j(d_j)$ $i = 1 \dots n$, $j = 1 \dots n$

¹ For further details see in Internet the web pages “Introduction to the Singular Value Decomposition” by Todd Will, UW-La Crosse, Wisconsin, 1999 and the web pages “Matrices Geometry & Mathematic” by Bill Davis and Jerry Uhl

A matrix is ill-conditioned if its condition number is very large. For 32bit double precision arithmetic this number is about 1E12. It can be easily calculated from the D matrix returned by the function SVDD

Hilbert matrix				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

In this case, the 4x4 Hilbert matrix has $c \approx 15514$

The following functions return directly the condition number of a matrix and its decimal logarithm

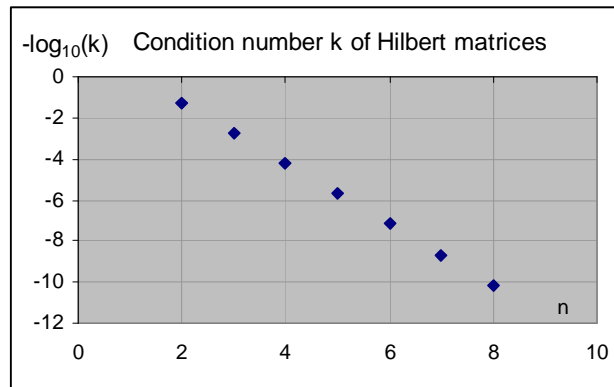
Function MCond(A)

Function MpCond(A)

$MCond = \kappa$

$MpCond = -\log_{10}(\kappa)$

The following graph shows the $p\kappa = -\log_{10}(\kappa)$ of Hilbert matrices of increasing order



Function MMopUp(M, [ErrMin])

This function eliminates all round-off errors from a matrix. Each element that has an absolute value 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		

{=MMopUp(A1:D3)}
improves the reading

As we can see, MMopUp improves the readability

Function MCovar(A)

Returns the covariance matrix (m x m) of a given matrix (n x m)

The (column) covariance is defined by the following formulas:

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

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

See also the matrix correlation function MCorr()

This function is similar to the Excel built-in function COVAR for two variables.

Function MCorr(A)

Returns the correlation matrix (m x m) of a given matrix (n x m)

The correlation matrix is definite by the following formulas:

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

Where:

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

$$S_i = \frac{1}{n} \sqrt{\sum_{k=1}^n (a_{ki} - \bar{a}_i)^2} \quad , \text{ for } i=1 \dots m$$

Note. Correlation matrix has always diagonal =1

See also the matrix covariance function MCovar

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 S_{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 u_i with the following simple formulas: $u_{ij} = \frac{x_{ij} - \bar{x}_i}{S_{xi}}$

	A	B	C	D	E	F	G	H	I	J
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		{=MCovar(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		{=MCovar(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	{=MCorr(A3:C12)}		

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:

$$\text{Covariance(Normalized data)} \circ \text{Correlation(Row data)}$$

The function MCorr() 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

Function RegrL(Y, X, [Intcpt])

Computes the multivariate linear regression with the SVD method¹.

Parameter Y is the (n x 1) vector of the dependent variable.

Parameter X is a list of independent variables. It may be a (n x 1) vector for monovariate regression or a (n x m) matrix for multivariate regression.

Parameter Intcpt calculates the intercept $y(0) = a_0$. Default is True

The function returns a (m+1 x 2) array containing the coefficients of the linear regression in the first column and the standard deviation in the second column..

¹ REGRL is maintained only for compatibility. From Excel 2003, the built-in function LINEST is accurate as REGRL.

Example - find the linear regression of 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 is: $y = a_0 + a_1 x$

	A	B	C	D	E	F	G	H
1	x	y	y^2			Coefficients		
2	1	5.12	6.53		a0	5.4981319	{=REGRL(B2:B15,A2:A15)}	
3	2	6.61	7.55		a1	1.0272967		
4	3	8.55	8.58					
5	4	10.07	9.61	Observations		14		
6	5	11.35	10.63	sum of the squared residual		7.2399732	=SUMX2MY2(B2:B15,C2:C15)	
7	6	12.47	11.66	residual standard deviation		0.7767439	=SQRT(F6/(F5-2))	
8	7	13.48	12.69	R-Squared		0.9707274	=VAR(C2:C15)/VAR(B2:B15)	
9	8	14.41	13.72					
10	9	15.27	14.74					
11	10	16.07	15.77	...				
12	11	16.82	16.80				
13	12	17.54	17.83	=F\$2+\$F\$3*A13				
14	13	18.22	18.85	=F\$2+\$F\$3*A14				
15	14	18.86	19.88	=F\$2+\$F\$3*A15				

In Matrix.xla there is no specific function for regression statistics but they can be computed using the standard statistic functions and the formula as shown in the above worksheet arrangement

The function RegrL can also return the standard deviation of the estimate. For that simply select an array of two columns before inserting RegrL.

	A	B	C	D	E	F	G
1	x1	x2	y			Coefficients	Stand. Dev
2	1	100	880		a0	998.1584083	2.633286811
3	2	100	960		a1	76.61291905	1.712538205
4	3	156	928		a2	-1.934587081	0.032771082
5	4	156	1000				
6	5	310	780				
7	6	312	856				

{=REGRL(C2:C7,A2:B7)}

Setting intcpt = False , forces the regression intercept to zero.

	A	B	C	D	E	F	G
1	x1	x2	y			Coefficients	Stand. Dev
2	1	50	10		a0	0	0
3	2	100	10		a1	56.05175211	3.790300371
4	3	156	4		a2	-1.071680315	0.067000456
5	4	156	54				
6	5	310	-50				
7	6	366	-60				

{=REGRL(C2:C7,A2:B7, False)}

Function RegrP(Degree, Y, X, [Intcpt])

Computes the polynomial regression $f(x)$ of a dataset of points $[x_i, y_i]$.

$$f(x) = a_0 + a_1x + a_2x^2 + \dots a_mx^m$$

Parameter Degree set the degree m of the polynomial.

Parameter Y is a $(n \times 1)$ vector of the dependent variable.

Parameter X is a $(n \times 1)$ vector of the independent variable.

Parameter Intcpt calculates the intercept $y(0) = a_0$. Default is True

The function returns a $(m+1 \times 2)$ array containing the coefficients $[a_0, a_1, a_2, \dots a_m]$ of the linear regression in the first column and the standard deviation in the second column..

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 coefficients		
2	0	1.54		a0		
3	0.5	155.5555556		a1		
4	1	132.39375		a2		
5	1.5	211.357125		a3		
6	2	230		a4		
7	2.5	302.337375		a5		
8	3	333		a6		
9	3.5	274.54375				
10	4	152.5				
11	4.5	378.1734375				
12	5	575				
13						
14						

The function RegrP can also return the standard deviation of the estimate. For that simply select an array of two columns before inserting RegrP.

	A	B	C	D	E	F	G
1	x	y			Coefficient Stand. Dev		
2	-2	-8		a0	2.108289	0.266952	
3	-1	-1		a1	2.952529	0.106831	
4	2	20		a2	0.949731	0.04257	
5	6	271		a3	1.005582	0.003444	
6	9	839					
7	10	1132					
8							
9							

NIST Certification Test

The following table gives the LRE results from RegrL and RegrP on the NIST linear regression data set

StRD Datasets	Difficulty	Model of class	min	avg	max
Norris	low	Linear	12.3	13.3	14.4
Pontius	low	Quadratic	11.7	13.3	14.9
NoInt1	low	Linear	14.7	14.9	15.0
Filip	high	Polynomial	7.4	7.4	7.5
Longley	high	Multilinear	11.7	12.3	12.7
Wampler1	high	Polynomial	9.7	11.3	13.9
Wampler2	high	Polynomial	13.1	13.7	15.0
Wampler3	high	Polynomial	9.5	11.3	14.0
Wampler4	high	Polynomial	9.5	11.2	13.8
Wampler5	high	Polynomial	7.2	8.9	11.6

The average LRE for all datasets is about 11.8.

Function RegrCir(X, Y)

It computes the LS circular regression of a dataset [xi, yi].

It returns a (3 x 2) array containing the radius R and the center coordinate (Xc, Yc) of the best fitting circle in the sense of the least squares.

$$(x - X_c)^2 + (y - Y_c)^2 = R^2$$

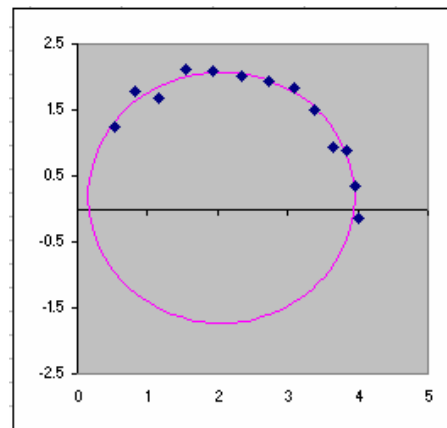
The second column contains the standard deviations of the estimates.

R	R Stand. Dev.
Xc	Xc Stand. Dev.
Yc	Yc Stand. Dev.

Example.

	A	B	C	D	E
1	x	y			
2	4	-0.13613			
3	3.96013	0.348796			
4	3.84212	0.882689			
5	3.65067	0.941187			
6	3.39341	1.503458			
7	3.0806	1.830088			
8	2.72472	1.924248			
9	2.33993	2.008011			
10	1.9416	2.091774			
11	1.5456	2.125867			
12	1.16771	1.669812			
13	0.823	1.78767			
14	0.52521	1.238593			

{=RegrCir(A4:A16, B4:B16)}



In the above example the best fitting circle has the parameters:

$$R = 1.90 (\pm 0.13), X_c = 2.05 (\pm 0.05), Y_c = 0.18 (\pm 0.09)$$

Of course when the dataset (xi, yi) contains exactly 3 points, the function RegrCir returns the exact interpolating circle.

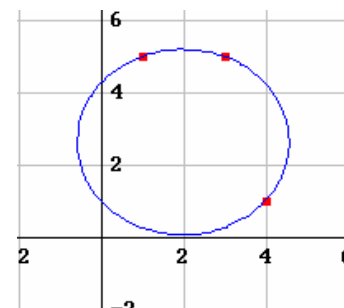
Example. Find the circle interpolating the points A(1,5), B(3,5), C(4,1)

	A	B	C	D	E
1	x	y			
2	1	5			
3	3	5			
4	4	1			
5					
6					

Circle Parameters

R = 2.576941
Xc = 2
Yc = 2.625

{=RegrCir(A2:A4,B2:B4)}



Compare with the exact result

$$(x - 2)^2 + (y - 21/8)^2 = 425/64$$

Function MCmp(Coeff)

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

$$a(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} + x^n$$

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

Parameter Coeff is the complete coefficient vector. If $a_n \neq 1$, the function performs the normalization before generating the companion matrix

Example:

	A	B	C	D	E	F	G	H																									
1	Polynomial =		50+24x+18x^2+7x^3+2x^5+x^5																														
2	Coefficients																																
3	a0	50	<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>-50</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>-24</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>-18</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>-7</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>-2</td></tr></table>						0	0	0	0	-50	1	0	0	0	-24	0	1	0	0	-18	0	0	1	0	-7	0	0	0	1	-2
0	0	0							0	-50																							
1	0	0							0	-24																							
0	1	0							0	-18																							
0	0	1							0	-7																							
0	0	0	1	-2																													
4	a1	24																															
5	a2	18																															
6	a3	7																															
7	a4	2																															
8	a5	1																															
9																																	
10																																	

{=MCmp(B3:B8)}

{=MCmp(B3:B8)}

	A	B	C	D	E	F	G	H																									
1	Polynomial =		100+67x+22x^2-8x^3+3x^5+2x^5																														
2	Coefficients																																
3	a0	100	<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>-50</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>-33.5</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>-11</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>4</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>-1.5</td></tr></table>						0	0	0	0	-50	1	0	0	0	-33.5	0	1	0	0	-11	0	0	1	0	4	0	0	0	1	-1.5
0	0	0							0	-50																							
1	0	0							0	-33.5																							
0	1	0							0	-11																							
0	0	1							0	4																							
0	0	0	1	-1.5																													
4	a1	67																															
5	a2	22																															
6	a3	-8																															
7	a4	3																															
8	a5	2																															
9																																	
10																																	

{=MCmp(B3:B8)}

Function MCplx([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 as

$$\text{MCplx}(\text{Ar}) = [A_r] + j[0]$$

A purely imaginary matrix can be written as

$$\text{MCplx}(\text{Ai}) = [0] + j[A_i] \quad (\text{remember the comma before the 2nd argument})$$

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

Optional parameter Cformat sets the complex input/output format (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 MMultC. If we have to multiply a real matrix by a complex vector, we can use the MMultC 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 MCplx function and then pass the result to MMultC. 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	{=MMultC(MCplx(A2:D5),F2:G5)}									

Function PolyRootsQR(poly)

This function returns all roots of a given polynomial

Poly can be an array of (n+1) coefficients [a0, a1, a2...] or a string like "a0+a1x+a2x^2+..."

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 is suitable for solving a polynomial up to 10th degree with a good accuracy (1E-9 – 1E-12)

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

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

	A	B	C	D	E	F
1						
2	TERMS					
3	degree	coeff.		x re	x imm	
4	0	240		-2	1	
5	1	-68		-2	-1	
6	2	-190		-1	1	
7	3	-76		-1	-1	
8	4	79		1	0	
9	5	28		2	0	
10	6	-10		3	0	
11	7	-4		4	0	
12	8	1				
13						
14	{=Poly_Roots_QR(B4:B12)}					
15						

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

Function PolyRootsQRC(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 for solving polynomials up to the 10th degree

This function uses the QR algorithm. The process consists of iteratively applying the QR decomposition to the complex companion matrix.

Example. Find all the roots of the following polynomial

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

	A	B	C	D	E	F	G
13			Coefficients			Roots	
14			re	im		re	im
15		a0	8	24		0	-2
16		a1	-22	-2		0	1
17		a2	9	7		1	3
18		a3	-5	-2		4	0
19		a4	1	0			
20							
21			{=PolyRootsQRC(C15:D19)}				

Function MRot(n, theta, p, q)

Returns the orthogonal matrix (n x n) that performs the planar rotation over the plane defined by axes p and q

Parameter *theta* sets the angle of rotation in radians

Parameters *p* and *q* are the columns of the rotation and must be: $p \neq q$ and $p \leq n$ and $q \leq n$

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

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

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

Where: $c = \cos(\theta)$. $s = \sin(\theta)$

Note that all rotation matrices have determinant = 1

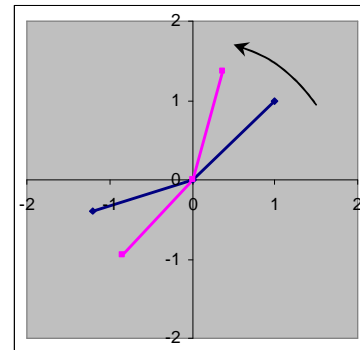
Example. Given two vectors in \mathbf{R}^2 ($\mathbf{v}_1, \mathbf{v}_2$) , find the same vectors after a rotation of 30° deg

The transformation formula is:

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} \cos(q) & -\sin(q) \\ \sin(q) & \cos(q) \end{bmatrix} \cdot \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix}$$

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	{=MRot(2,E6;1;2)}			30	0.524		{=MPROD(D2:E3;A2:B3)}		



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

This function computes the orthogonal rotation for a Factor 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 factors m, at this release, can only be 2 or 3.

Optional parameter Normal = True/False chooses the "Varimax normalized criterion". That is, it 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 than 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* x *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 where *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 widely used, by now, in many primarily (and expensive) statistical tools, but because it is very rare in freeware software, we have added it 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 (most 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						
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

Factor Loading is a very large topic and its explanation is out of the scope of this tutorial. See the specialized statistical literature.

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

Varimax, for a (p x k) matrix **A** (i.e., with p rows and k columns), is defined as follows:

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

where:

$$b_{ij} = \begin{cases} a_{ij} & \Rightarrow \text{normal} = \text{false} \\ \frac{a_{ij}}{|a_i|} & \Rightarrow \text{normal} = \text{true} \end{cases}$$

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

Performs the normalization of a real (n x m) matrix.

The optional parameter *Normtype* indicates what normalization is performed

The optional parameter *Tiny* sets the minimum error level (default 2E-14)

$u_i = \frac{v_i}{ v_{\min} }$	Normtype = 1. All vector's components are scaled to the non-zero minimum of the absolute values
$u_i = \frac{v_i}{ v }$	Normtype = 2 (default). All non-zero vectors are length = 1
$u_i = \frac{v_i}{ v_{\max} }$	Normtype = 3. All vector's components are scaled to the maximum 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 MNormalizeC(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 input/output format (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 MNorm(v, [NORM])

Returns 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	$\ A\ _2 = \sqrt{r(A^T A)}$
Norm = 3 (also infinite)	Maximum absolute row sum	$\ A\ _3 = \max_i \left(\sum_j a_{ij} \right)$

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

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	lnorm				
2	6	2	3		0	22.113	=MNorm(\$A\$2:\$C\$4,E2)			
3	-7	-9	5		1	29	=MNorm(\$A\$2:\$C\$4,E3)			
4	9	-7	-9		2	16.833	=MNorm(\$A\$2:\$C\$4,E4)			
5	7	-5	0		3	25	=MNorm(\$A\$2:\$C\$4,E5)			

Function MMultC(M1, M2, [Cformat])

Performs a complex matrix multiplication.

If the dimension of the matrix M_1 is $(n \times m)$ and M_2 is $(m \times p)$, then the product is a matrix $(n \times p)$

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

Optional parameter *Cformat* sets the complex input/output format (default = 1)

$$M_1 = A + j B$$

$$M_2 = C + j D$$

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

Examples:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1														
2														
3														
4														
5														
6														
7														
8														
9														
10														
11														

Matrix multiplication of two 3x3 complex matrices

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												

Imaginary part must always be provided even if it is entirely null .
We see that we have added to the original real matrix A2:C4 the "0" range D2:F4
In this example the 3x3 matrix is complex and the 3x1 vector is complex

	A	B	C	D	E	F	G	H	I
1									
2									
3									
4									
5									
6									
7									

One can avoid to insert directly the imaginary part if is zero by the function **MCplx(re, im)**
In this example the 3x3 matrix is real and the 3x1 vector is complex

	A	B	C	D	E	F	G	H	I	J	K
1											
2											
3											
4											
5											
6											
7											

In this example the 3x3 matrix is complex and the 3x1 vector is real

	A	B	C	D	E	F	G	H
5								
6								
7								
8								

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

Function MInvC(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 input/output format (default = 1)

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

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Matrix A							Inverse of A					
2	Real			Imm				Real			Imm		
3	1	2	0	0	-1	3		0.892	-0.11	0.021	0.095	0.095	-0.67
4	-1	3	-1	0	2	-1		0.249	0.249	0.029	-0.07	-0.07	-0.14
5	0	-1	4	0	-2	0		0.095	0.095	0.328	0.108	0.108	-0.02
6													
7													
8													

{=MInvC(A3:F5)}

	A	B	C	D	E	F	G
1	1	1+2i	2+10i		10+i	-2+6i	-3-2i
2	1+i	3i	-5+14i		9-3i	8i	-3-2i
3	1+i	5i	-8+20i		-2+2i	-1-2i	1
4							

{=MInvC(A1:C3, 3)}

Function ProdScalC(v1, v2,)

Returns the scalar product of two complex vectors

$$\mathbf{a} \bullet \mathbf{b} = \sum_k (a_{re} + ia_{im})_k \cdot (b_{re} - ib_{im})_k$$

	A	B	C	D	E	F	G
1	vector a		vector b			a • b	
2	re	im...	re	im		re	im
3	2	1	-1	0		-6	9
4	3	2	0	1			
5	4	5	1	-2			
6							
7							

{=ProdScalC(A3:B5, C3:D5)}

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

In string format we can write complex numbers as string "a+ib" and need not insert zero real or imaginary components.

Look at the same example. Note the third optional parameter cformat = 3

	A	B	C	D	E
1	vector a	vector b		a • b	
2	2+i	-1		-6+9i	
3	3+2i	i			
4	4+5i	1-2i			
5					

=ProdScalC(A2:A4, B2:B4, 3)

Function SysLinC(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 input/output format (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 -, $\text{Det}(\mathbf{A}) \neq 0$

Example - solve the following complex 3x3 linear system

$$\begin{cases} x_1 + (2-i)x_2 + 3ix_3 = 5+10i \\ -x_1 + (3+2i)x_2 - (1+i)x_3 = -11-i \\ -(1+2i)x_2 + 4x_3 = 11-3i \end{cases} \quad \begin{bmatrix} 1 & 2-i & 3i \\ -1 & 3+2i & -1-i \\ 0 & -1-2i & 4 \end{bmatrix} \cdot \mathbf{x} = \begin{bmatrix} 5+10i \\ -11-i \\ 11-3i \end{bmatrix}$$

	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			Im			Real	Im	Real	Im		
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												

=SysLinC(A4:F6, H4:I6)

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

	A	B	C	D	E	F	G
1							
2	1	2-j	3j		5+10j		3+j
3	-1	3+2j	-1-j		-11-j		-1+j
4	0	-1-2j	4		11-3j		2-j
5							
6							

=SysLinC(A2:C4, E2:E4, 3)

Function Simplex(Funct, Constraint, [Opt])

This function performs the linear optimization with the Simplex method

Funct is the array (1 x n) containing the coefficients of the linear function to optimize

Constraint is the array (m x n+2) containing the coefficients of m linear constraints and the type of constraints (“<”, “>”, “=”)

Opt sets 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

$$z = a_1x_1 + a_2x_2 + \dots a_nx_n$$

With the following constraints:

$$x_i \geq 0$$

and for j=1 to m

$$b_{j1}x_1 + b_{j2}x_2 + \dots b_{jn}x_n \leq c_j$$

This function accepts the constraint symbols: “<”, “>”, and “=”

This function returns:

If an optimal solution exists – that is: all constraints are satisfied and the function is maximized – then it returns the solution vector and, in the last cell, the corresponding function value

If the constraints region is unbounded – that is, if the region is not closed - a finite solution cannot exist and the function return “inf”. Typically this happens when the constraints are insufficient

If the constraints region is bounded, but the solution doesn't exist, the function returns “?”. Typically this happens when you add too many constraints

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

Now lets see how it works.

Example: find the maximum of the function:

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

With the following constraints:

$$40x + 25y \leq 1000$$

$$35x + 25y \leq 980$$

$$25x + 35y \leq 875$$

	A	B	C	D	E	F	G
1	Linear Optimization of $f(x,y) = 1.2x + 1.4y$						
2	f(x,y) to maximize			constraints			
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 whether you write “<” or “<=” for the constraint symbols

The solution is about:

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

This function accept also mixed constraint symbols

Let's see this example

Maximize $z = x_1 + x_2 + 3x_3 - 0.5x_4$

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

$$x_1 + 2x_3 \leq 10$$

$$2x_2 - 7x_4 \leq 0$$

$$x_2 - x_3 + 2x_4 \geq 10$$

$$x_1 + x_2 + x_3 + x_4 = 9$$

	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	constraint	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 MPerm(Permutations)

Returns the permutations matrix. It consists of a sequence of n integer unitary vectors.
The parameter is a vector indicating the sequence.
Any permutations matrix can be indicated by a sequence vector, such as:

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

	A	B	C	D	E	F	G	H	I	J	K
1	Perm.		permutation matrix								
2	3		0	0	1	0					
3	4		0	0	0	1					
4	1		1	0	0	0					
5	2		0	1	0	0					

`{=MPerm(A2:A5)}`

Function MHessenberg(Mat)

Returns the Hessenberg form of a square matrix
As known a matrix is in Hessenberg 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																

`{=MHessenberg(A1:G7)}`

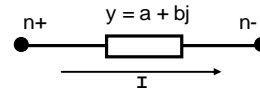
Function MAdm(Branch)

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 number of 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 \mathbf{V} is the vector of nodal voltage, \mathbf{I} is the vector of nodal current and $[\mathbf{Y}]$ is the admittance matrix.

$$[\mathbf{Y}] \cdot \mathbf{V} = \mathbf{I}$$

If $N+1$ is the number of nodes, then the matrix dimension will be $(N \times N)$.

(usually the references nodes is set at $\mathbf{V} = 0$)

\mathbf{V} , \mathbf{I} , and $[\mathbf{Y}]$ are in general complex

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

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

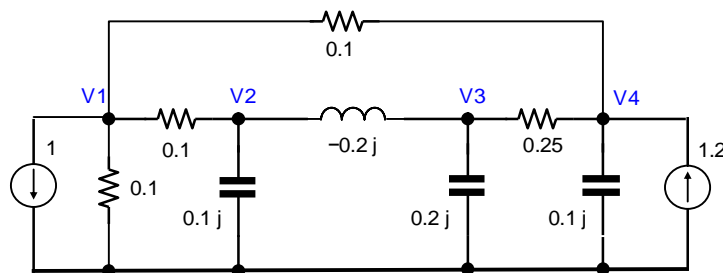
Linear Electric Network

Nodal Analysis is widely used for Electric Networks. A passive linear network is composed of four basic components: Resistors, Inductors, Capacitors, and Current Sources. 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 $y = j \omega C$
Inductor		Value L (henry)	Admittance $y = -j 1/(\omega L)$
Current source		Value I (ampere)	$I = I_{re} + j I_{im}$

Where : $\omega = 2 \pi f$ (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			{=MAdm(A3:D10)}								{=SysLinC(F2:M5,G7:H10)}				

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

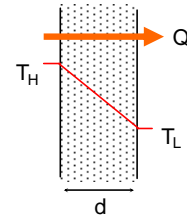
Thermal Network

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 Conductive Heat Transfer.

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

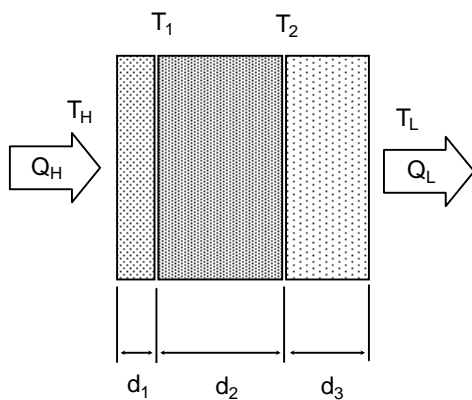
$$Q = -\frac{k}{d} \cdot (T_H - T_L) = -g(T_H - T_L)$$



Thus, the network equations are the same as for 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:

$$T_H = 400 \text{ } ^\circ \text{C}$$

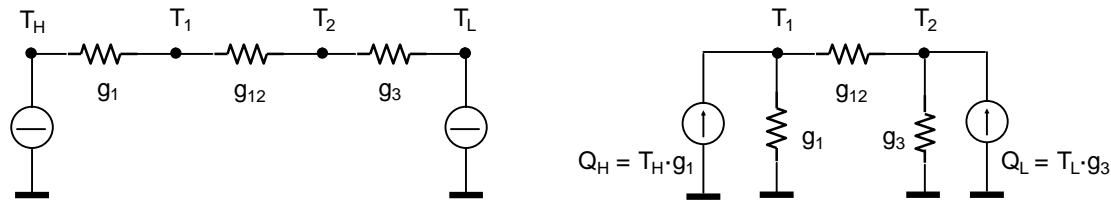
$$T_L = 20 \text{ } ^\circ \text{C}$$

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

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

Internal temperature T_1 and T_2 are unknowns

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



Where thermal conductance are:

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

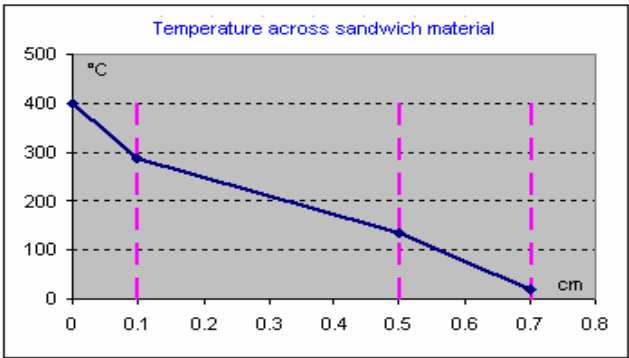
A spreadsheet calculus can be arranged as the following

	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					{=MAdm(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 temperatures T_1 and T_2 we can easily draw the thermal profile across the material



Function MLeontInv(ExTab, Tot)

Returns the inverse of Leontief matrix of 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 economic 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 the interdependence of industries. Assuming EX the *Exchange table*

$$EX = [x_{ij}]$$

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

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

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

Leontief matrix is .

$$L = (I - A)$$

If \mathbf{d} is the *Final Demand* vector, the production \mathbf{x} is given by the following formula.

$$\mathbf{x} = (I - A)^{-1} \mathbf{d} = L^{-1} \mathbf{d}$$

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: agriculture = 45, manufactory = 74, services = 130.

Supply	Purchasing sectors			total
sectors	Agriculture	Manufact.	Services	output
Agriculture	34.69	4.92	5.62	85
Manufact.	5.28	61.82	22.99	163
Services	10.45	25.95	42.03	219


Sectors	demand
Agriculture	45
Manufact.	74
Services	130

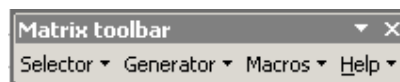
	A	B	C	D	E	F
1	Interindustry exchange matrix					
2	Supply	Purchasing sectors			total	external
3	sectors	Agriculture	Manufact.	Services	output	demand
4	Agriculture	34.69	4.92	5.62	85	45
5	Manufact.	5.28	61.82	22.99	163	74
6	Services	10.45	25.95	42.03	219	130
7						
8		Inverse Leontief matrix			Output	
9		1.71411	0.10067	0.06751	93.36	9.8%
10		0.22307	1.67962	0.22528	163.62	0.4%
11		0.30473	0.34622	1.29215	207.31	-5.3%
12						
13		{=MLeontInv(B4:D6,E4:E6)}			{=MMULT(B9:D11,F4:F6)}	
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 by 9.8% for agriculture, and by 0.4% for manufacturing, while decreased by -5.3% for services


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 display it by clicking on the Matrix icon .



(Matrix.xla v.2.x)

 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, adjoint, etc. Simply select any cell into the matrix and choose the menu item that you want.

<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>0</td><td>3</td><td>6</td><td>2</td><td>1</td><td>0</td></tr><tr><td>-1</td><td>4</td><td>9</td><td>0</td><td>3</td><td>-6</td></tr><tr><td>-2</td><td>5</td><td>12</td><td>-2</td><td>7</td><td>-1</td></tr><tr><td>-3</td><td>6</td><td>15</td><td>-4</td><td>11</td><td>-2</td></tr><tr><td>-4</td><td>7</td><td>18</td><td>-6</td><td>15</td><td>8</td></tr></table>	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	<table><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td></td><td>0</td><td>3</td><td>6</td><td>2</td><td>1</td><td>0</td></tr><tr><td></td><td>-1</td><td>4</td><td>9</td><td>0</td><td>3</td><td>-6</td></tr><tr><td></td><td>-2</td><td>5</td><td>12</td><td>-2</td><td>7</td><td>-1</td></tr><tr><td></td><td>-3</td><td>6</td><td>15</td><td>-4</td><td>11</td><td>-2</td></tr><tr><td></td><td>-4</td><td>7</td><td>18</td><td>-6</td><td>15</td><td>8</td></tr></table>		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	<table><tr><td>4</td><td>8</td><td>7</td><td>1</td><td>-6</td><td>4</td></tr><tr><td>10</td><td>3</td><td>9</td><td>7</td><td>-7</td><td>3</td></tr><tr><td>-4</td><td>9</td><td>-2</td><td>3</td><td>1</td><td>8</td></tr><tr><td>-9</td><td>-8</td><td>5</td><td>-3</td><td>-4</td><td>3</td></tr><tr><td>-10</td><td>4</td><td>-7</td><td>4</td><td>-10</td><td>-9</td></tr><tr><td>-5</td><td>-2</td><td>-1</td><td>5</td><td>1</td><td>0</td></tr></table>	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						
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	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																																																																																																																				
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																																																																																																																					
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>0</td><td>3</td><td>6</td><td>2</td><td>1</td><td>0</td></tr><tr><td>-1</td><td>4</td><td>9</td><td>0</td><td>3</td><td>-6</td></tr><tr><td>-2</td><td>5</td><td>12</td><td>-2</td><td>7</td><td>-1</td></tr><tr><td>-3</td><td>6</td><td>15</td><td>-4</td><td>11</td><td>-2</td></tr><tr><td>-4</td><td>7</td><td>18</td><td>-6</td><td>15</td><td>8</td></tr></table>	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	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>0</td><td>3</td><td>6</td><td>2</td><td>1</td><td>0</td></tr><tr><td>-1</td><td>4</td><td>9</td><td>0</td><td>3</td><td>-6</td></tr><tr><td>-2</td><td>5</td><td>12</td><td>-2</td><td>7</td><td>-1</td></tr><tr><td>-3</td><td>6</td><td>15</td><td>-4</td><td>11</td><td>-2</td></tr><tr><td>-4</td><td>7</td><td>18</td><td>-6</td><td>15</td><td>8</td></tr></table>	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	<table><tr><td>4</td><td>8</td><td>7</td><td>1</td><td>-6</td><td>4</td></tr><tr><td>10</td><td>3</td><td>9</td><td>7</td><td>-7</td><td>3</td></tr><tr><td>-4</td><td>9</td><td>-2</td><td>3</td><td>1</td><td>8</td></tr><tr><td>-9</td><td>-8</td><td>5</td><td>-3</td><td>-4</td><td>3</td></tr><tr><td>-10</td><td>4</td><td>-7</td><td>4</td><td>-10</td><td>-9</td></tr><tr><td>-5</td><td>-2</td><td>-1</td><td>5</td><td>1</td><td>0</td></tr></table>	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												
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	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																																																																																																																					
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																																																																																																																					
<table><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td></td><td>0</td><td>3</td><td>6</td><td>2</td><td>1</td><td>0</td></tr><tr><td></td><td>-1</td><td>4</td><td>9</td><td>0</td><td>3</td><td>-6</td></tr><tr><td></td><td>-2</td><td>5</td><td>12</td><td>-2</td><td>7</td><td>-1</td></tr><tr><td></td><td>-3</td><td>6</td><td>15</td><td>-4</td><td>11</td><td>-2</td></tr><tr><td></td><td>-4</td><td>7</td><td>18</td><td>-6</td><td>15</td><td>8</td></tr></table>		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	<table><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td></td><td>0</td><td>3</td><td>6</td><td>2</td><td>1</td><td>0</td></tr><tr><td></td><td>-1</td><td>4</td><td>9</td><td>0</td><td>3</td><td>-6</td></tr><tr><td></td><td>-2</td><td>5</td><td>12</td><td>-2</td><td>7</td><td>-1</td></tr><tr><td></td><td>-3</td><td>6</td><td>15</td><td>-4</td><td>11</td><td>-2</td></tr><tr><td></td><td>-4</td><td>7</td><td>18</td><td>-6</td><td>15</td><td>8</td></tr></table>		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	<table><tr><td>4</td><td>8</td><td>7</td><td>1</td><td>-6</td><td>4</td></tr><tr><td>10</td><td>3</td><td>9</td><td>7</td><td>-7</td><td>3</td></tr><tr><td>-4</td><td>9</td><td>-2</td><td>3</td><td>1</td><td>8</td></tr><tr><td>-9</td><td>-8</td><td>5</td><td>-3</td><td>-4</td><td>3</td></tr><tr><td>-10</td><td>4</td><td>-7</td><td>4</td><td>-10</td><td>-9</td></tr><tr><td>-5</td><td>-2</td><td>-1</td><td>5</td><td>1</td><td>0</td></tr></table>	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
	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	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																																																																																																																				
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																																																																																																																					

Automatically the “*Selector tool*” works on the max area bordered by empty cells 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

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 calls the *Paster tool*

The dialog box titled "Matrix scraps selector" has a close button (X) in the top right. It contains three input fields: "First cell" with the text "\$X\$12", "Last cell" which is empty, and "Option" with a dropdown menu showing "Triang. upper". To the right of these fields are four buttons: "Select", "Paste", "Help", and a "Triang. upper" dropdown menu.

Scraps Paster tool.

The selection of matrix parts is obtained by a multi-range selection. Excel cannot copy it. If you try to give the usual sequence CTRL+C you will obtain an error message.

For this task, this smart little macro for pasting the range or multi-range that you have previous selected comes in handy.

Select *Paster* from the menu. This 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.

The dialog box titled "Matrix scrap paster" has a close button (X) in the top right. It contains a "Copy selected cells into range starting from:" label above an empty input field. Below this is a "Fill unselected cells with:" section with two radio buttons: "nothing" (selected) and "zero". To the right is a "Target range" section with several radio buttons: "As is" (selected), "vertical", "horizontal", "diagonal", "transpose", "flip horiz.", "flip vert.", and "Adjoint". At the bottom are three buttons: "OK", "Help", and "Exit".

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 elements of another one, you can use this simple option, and the result will be similar to the following:

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

Change the target range: Normally the range is copied as is. But sometimes we need to rearrange the geometry of the target 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".

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

Sometimes 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. By giving your OK, you will generate the matrix to the left

1	1	0	0	0	0	0
3	0	3	0	0	0	0
9	0	0	9	0	0	0
-2	0	0	0	-2	0	0
11	0	0	0	0	11	0
8	0	0	0	0	0	8

Or we can extract an adjoint sub-matrix. For example, select the a33 element and choose the menu *Selector/adjoint*. Then 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

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

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.

-10	-8	7	-4
-1	4	7	4
-5	-6	2	-5
1	-8	1	-8
7	-10	0	-11
13	-12	-1	-14

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

	A	B	C	D	E	F
1						
2		-10	-8	7	-4	
3		-1	4	7	4	
4		-5	-6	2	-5	
5		1	-8	1	-8	
6						

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

	A	B	C	D	E	F
1						
2		-10	-1	-5	1	
3		-8	4	-6	-8	
4		7	7	2	1	
5		-4	4	-5	-8	
6						

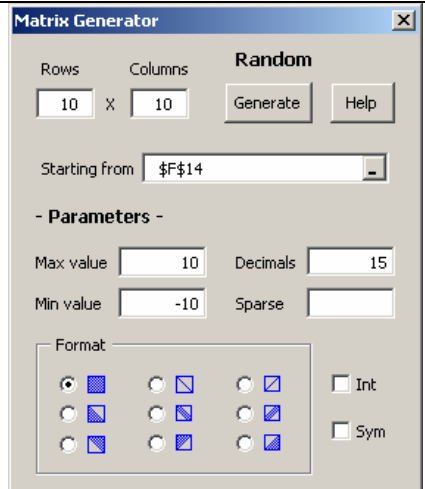
Matrix Generator

This smart macro can generate different kind of matrices

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

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

Random matrix with given format



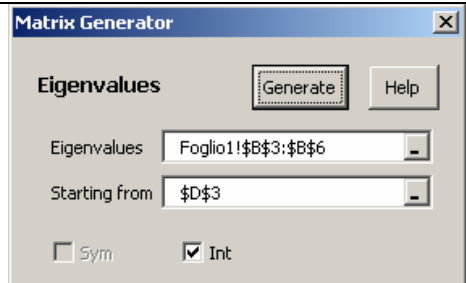
Parameters:
Random numbers **x** are generated with the following constrains:

Max value: upper limit of **x**
Min value: lower limit of **x**
Decimals: sets the number of decimals of **x**
Int checkbox: **x** as integers

Sym checkbox: the matrix will be symmetric
Sparse: index between 0 and 1: the number 0 (default) means full matrix, 1 very sparse matrix

Starting from: top-left matrix corner

Random matrix with given 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

	<p>Generate a square matrix with given determinant or rank</p> <p><i>Dimension:</i> matrix dimension <i>Rank:</i> 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. <i>Starting from:</i> top-left matrix corner <i>Determinant:</i> sets the determinant of the random matrix <i>Sym:</i> generate a symmetric random matrix <i>Int:</i> generate a random matrix with integer values</p>
--	---

Tartaglia's matrix

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

Hilbert's matrix

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

Sparse matrix

	<p>Parameters: Random sparse matrix [aij] is generated with the following constraints:</p> <p><i>Max: value:</i> upper limit of aij <i>Min: value:</i> lower limit of aij <i>Dim:</i> matrix dimension (n x n) <i>Dom:</i> Dominance factor D, with $0 < D < 1$ <i>Fill:</i> Filling factor F, with $0 < F < 1$ <i>Spread:</i> Spreading factor S, with $0 < S < 1$ <i>Sym:</i> check it for symmetric matrix <i>Int:</i> check it for integer matrix. <i>Starting from:</i> left-top matrix corner</p> <p>Output format <i>Coordinates:</i> generates a (k x 3) matrix in sparse coordinates format: [i, j, aij] <i>Square:</i> generates a square sparse matrix [aij]</p>
--	---

There are several way to arrange a sparse matrix in order to save space, computation efficiency, etc. In this add-in we use the (old but simple) Coordinate format

The **Coordinate format** of sparse matrix consists a 3 columns array where the 1st and 2nd columns contains the indexes "i" and "j", while the 3rd column contains the value aij.

Of course, the space saving is valid only for sparse matrices with $F < 1/3$

i	j	aij
1	1	a ₁₁
1	2	a ₁₂
1	3	a ₁₃
2	1	a ₂₁
...

Filling factor

The filling factor measures how "dense" a matrix is . In this document, it is defined as

$$F = 1 - \frac{N_{zero}}{N_{Tot}}$$

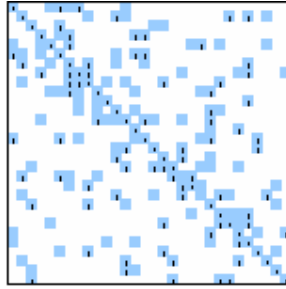
N_{zero} = number of zero elements
 N_{tot} = total of matrix elements

The factor is between 0 and 1; 1 means dense matrix , 0 empty matrix

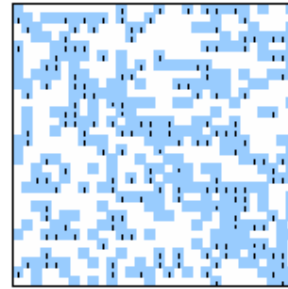
We see that the coordinate format of a square matrix is useful only if $F < 1/3$

We note also that for a simple (n x n) diagonal square matrix, $F = 1/n$

The following pictures show 2 random sparse matrices having different filling factor



(30 x 30) $F = 0.3$



(30 x 30) $F = 0.6$

Usually large sparse matrices in applied science have a factor F less then 0.2 (20%)

Dominance Factor

The dominance factor measures how much a matrix is "diagonal dominant" In this document, it is defined as

$$D = \frac{1}{n} \sum_{i=1}^n D_i$$

where the single row dominance factor D_i , for a not empty row, is defined as

$$D_i = \frac{|a_{ii}|}{\sum_{j=1}^n |a_{ij}|} = \frac{d_i}{d_i + S_i} \quad d_i = |a_{ii}| \quad S_i = \sum_{j=1, j \neq i}^n |a_{ij}|$$

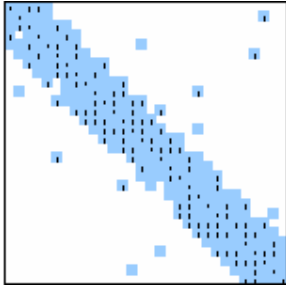
The row dominance factor D_i is always between 0 and 1;

Case	Description
$D_i = 0$	The diagonal element is zero: $a_{ii} = 0$
$0 < D_i < 0.5$	The row is dominated: $d_i < S_i$
$D_i = 0.5$	The row is indifferent: $d_i = S_i$
$0.5 < D_i < 1$	The row is dominant: $d_i > S_i$
$D_i = 1$	The row contains only the diagonal element: $S_i = 0$

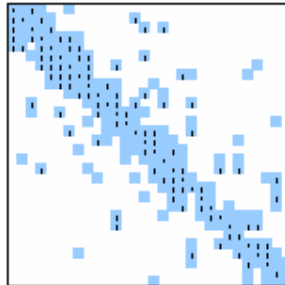
Spreading Factor

This factor is used for spreading the elements around the first diagonal. This has no definition and is used only in this macro. $S = 0$ means no spreading at all, that is a band-diagonal matrix;

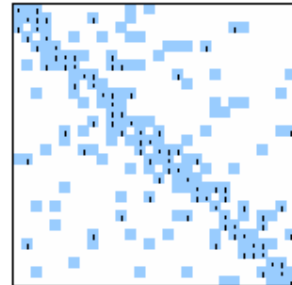
$S = 1$ means large spreading.



(30 x 30) $S = 0.05$



(30 x 30) $S = 0.2$



(30 x 30) $S = 0.5$

Macro stuff.

The menu **>macros > Matrix operations** contains several macros performing useful tasks.

Macro versus Function

In this Matrix package there are worksheet functions that perform the same tasks of the macros. The reason for also having macros is that macros are more suitable for large matrices and heavy long computation.

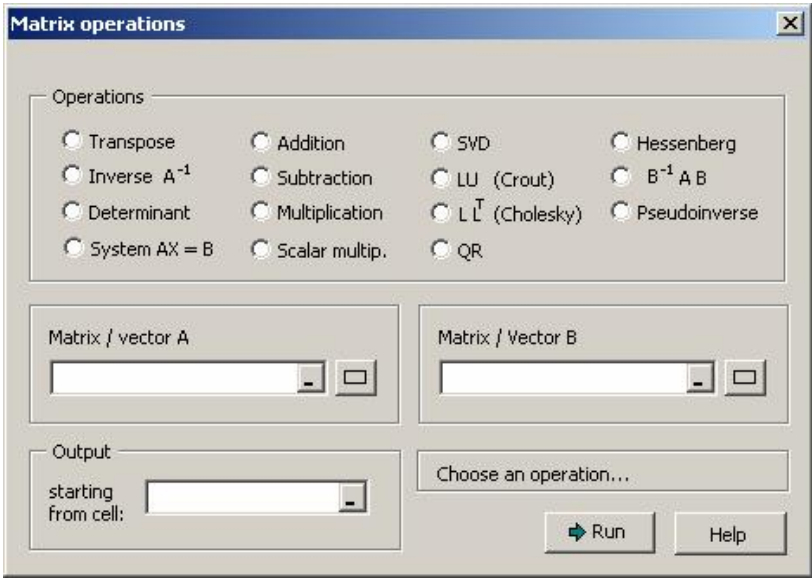
On the other hand, functions are suitable for automatic recalculation, but Excel becomes very slow for large worksheet full of active functions. You should see when it is convenient to use either macros or functions.

 Available macros are:


Matrix operations	Real Matrix operation
Complex matrix operations	Complex Matrix operation
Sparse matrix operations	Sparse linear solving, vector product and other stuff
Eigen-solving	Eigenvalues / Eigenvector for real and complex matrices
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

Matrix operations

This macro performs several common matrix operations like: addition, subtraction, multiplication, inversion, scalar multiplication, etc.



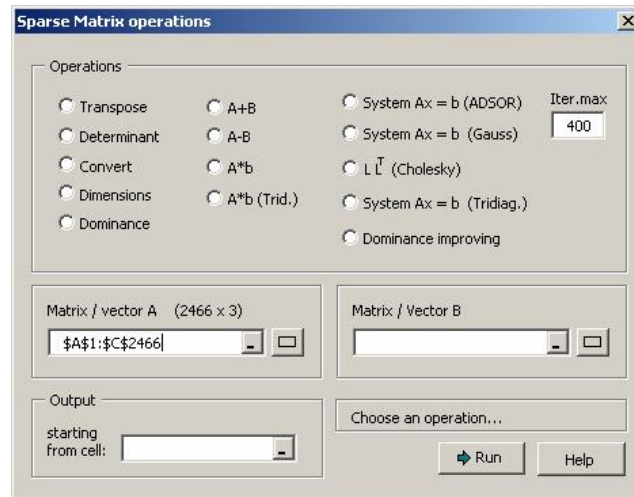
Using this macro is quite simple. Select the matrix (or select any cell inside the matrix) and start the macro from its menu. Some tasks need only one matrix, and some others require two matrices. Choose the operations that you want to perform. If the operation needs a second parameter - matrix or vector -, select it in the B field. Then, choose a starting point for output. Click "run", wait, and check the result. That's all.

 The smart selector is useful for selecting large matrices. Simply select a cell inside the matrix and click on it. The entire matrix will automatically be selected. This tool works if there are only empty cells around the matrix.

Sparse Matrix operations

This macro solves sparse linear systems and performs some common matrix operations like: transposition, addition, subtraction, vector multiplication, determinant

Some tasks need only one matrix and some others require two matrices



Sparse matrices require a special coordinates format (see Random sparse matrix). Matrix dimension must be greater than 3.

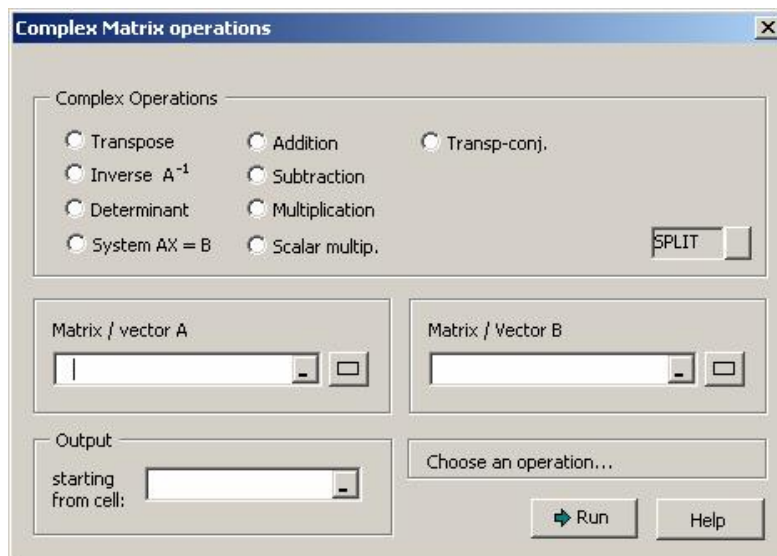
The Linear System macros accept both sparse and standard square matrix formats

Task	Description
Transpose	Returns the transpose. Input A is in sparse format
Determinant	Computes the determinant. Input A is in sparse format
Convert	Converts a matrix in sparse coordinates format and vice versa. Input A accepts both standard or sparse format
Dimensions	Returns the true dimensions (n x m) and the filling factor of a sparse matrix (see Random sparse matrix). Input A is in sparse format.
Dominance	Returns the row dominance factor of a matrix (see Random sparse matrix). Input A accepts both standard or sparse format
A+B	Returns the addition of two sparse matrices. Inputs A and B are in sparse format
A-B	Returns the subtraction of two sparse matrices. Inputs A and B are in sparse format
A*b	Returns the product of a sparse matrix A for a vector b . Input A is in sparse format
A*b (Trid.)	Returns the product of a sparse tridiagonal matrix A for a vector b . Input A is in tridiagonal format (see Function MMult3)
Linear System Ax = b (ADSOR)	Solves a sparse linear system using the iterative ADSOR algorithm (Adaptive-SOR). Input A accepts both standard or sparse format. The Iterative algorithm always converges if the matrix is diagonal dominant. It is usually faster than other deterministic methods (Gauss. LR, LL). The iterations limit (default 400) can be modified by the top-right field
Linear System	Solves a sparse linear system using the iterative Gauss algorithm with partial pivot

Ax = b (Gauss)	and back-substitution. Input A accepts both standard or sparse format.
LL (Cholesky)	Returns the Cholesky factorization of a sparse symmetric matrix
Linear System Ax = b (Trid.)	Solves a tridiagonal linear system using the iterative Gauss algorithm with partial pivot and back-substitution. Input A is in tridiagonal format (see Function MMult3) ..
Dominance improving	Tries to improve the dominance of a sparse matrix by rows exchanging. Input A accepts both standard or sparse format. Useful for sparse linear solving Ax = b . In that case, the macro also accepts the vector b

Complex Matrix operations

This macro performs several common matrix operations like: addition, subtraction, multiplication, inversion, scalar multiplication, etc. in complex arithmetic.



The macros accept matrices in 3 different formats: 1) split, 2) interlaced, 3) string.
(see About complex matrix format)

Choose the adapt format that you want simply by clicking repeatedly on the little button at the right

Macro Gauss-step-by-step

This macro performs, step by step, the reduction of the given matrix into a triangular or diagonal form by the Gauss and Gauss-Jordan algorithms.

It works like the Function GJstep except that it traces all multiplication coefficients as well all the swap actions.

3	3	2	1	0	0	2
2	2	-1	0	1	0	-3
3	2	-1	0	0	1	
Det(A1) = -3 Det(A)						

coefficients

Multiply the 1st row for 2
Multiply the 2nd row for -3
Add the two rows and substitute the result to the 2nd row

3	3	2	1	0	0	
0	0	7	2	-3	0	< swap
0	1	3	1	0	-1	< swap

Exchange the 2nd row with the 3rd row

Using this macro is very easy. Select the matrix that you want to reduce and start the macro at the menu:

>macros > Gauss step by step

The reduction steps are traced below the original matrix.

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

3	3	2	1	0	0
2	2	-1	0	1	0
3	2	-1	0	0	1
matrix to invert			Identity matrix		

Then select the 3x6 range and start the macro

	A	B	C	D	E	F	G	H
1								
2		3	3	2	1	0	0	
3		2	2	-1	0	1	0	
4		3	2	-1	0	0	1	
5								

Matrix Reduction step-by-step

Matrix reduction to triangular or diagonal form with Gauss or Gauss-Jordan algorithm

Matrix:

Reduction type

☒ Diagonal

☐ Triangular

Pivoting

☒ always

☐ only for zero

Options

☐ Integer

Errmax:

☐ last step only

Run

?

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

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

The integer check box sets the reduction method in integer exact mode (only for integer matrices).

Errmax sets the round-off error

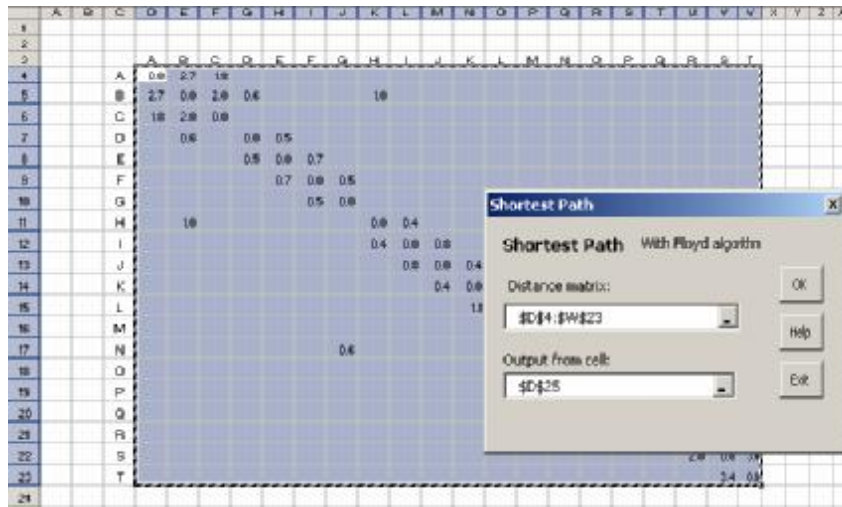
Last step check box writes only the last step. Useful for large matrices

Macro Shortest-Path

This macro generates the shortest-path matrix from a given distance-matrix. It works like the function **PathFloyd** 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

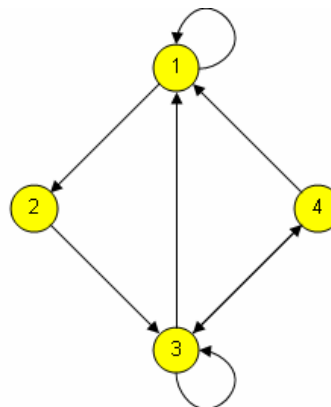
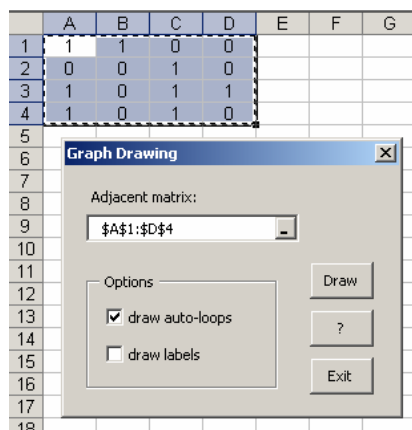


For default, the output matrix begins at cell D25, just below the input matrix.

Macro Draw Graph

This useful stuff draws a simple graph from its adjacent matrix. (There is now [an](#) 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 MBlock and MBlockPerm except that it is better adapted for large matrices.

Using this macro is very easy. Select the matrix and start the macro at the menu:

>macros > Block reduction

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1														
2		-1	0	0	1	0	1	0	0					
3		-1	1	0	1	1	1	0	1					
4		-1	1	-1	1	1	1	1	1					
5		1	0	0	1	0	1	0	0					
6		-1	1	0	1	1	1	0	1					
7		1	0	0	1	0	1	0	0					
8		-1	1	-1	1	1	1	1	1					
9		-1	1	0	1	1	1	0	1					
10														
11		-1	1	1	0	0	0	0	0					
12		1	1	1	0	0	0	0	0					
13		1	1	1	0	0	0	0	0					
14		-1	1	1	1	1	1	0	0					
15		-1	1	1	1	1	1	0	0					
16		-1	1	1	1	1	1	0	0					
17		-1	1	1	1	1	1	1	-1					
18		-1	1	1	1	1	1	1	-1					
19														
20		1	6	4	8	5	2	7	3					
21														

input
sparse

block
partitioned

permutation

References

In this chapter we have collected, listed for subject, all the documents that we have consulted without regarding their source (paper books, electronics books, html web pages, pdf notes, etc.). For all internet documents we have used the web engine GOOGLE .

"LAPACK -- Linear Algebra PACKage" 3.0, Update: May 31, 2000

"Numerical Analysis" F. Sheid, McGraw-Hill Book Company, New-York, 1968

"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

"Matrix Analysis and Applied Linear Algebra", C. D. Mayer, Siam, 2000

"Nonlinear regression", Gordon K. Smyth, John Wiley & Sons, 2002, Vol. 3, pp 1405-1411

"Linear Algebra" vol 2 of Handbook for Automatic Computation, Wilkinson, Martin, and Peterson, 1971

"Linear Algebra" Jim Hefferon, Saint Michael's College, Colchester (Vermont), July 2001.

"Linear Algebra - Answers to Exercises" Jim Hefferon, Saint Michael's College, Colchester (Vermont), July 2001

"Calcolo Numerico" Giovanni Gheri, Università degli Studi di Pisa, 2002

"Introduction to the Singular Value Decomposition" by Todd Will, UW-La Crosse, University of Wisconsin, 1999

"Calcul matriciel et équation linéaires", Jean Debord, Limoges Cedex, 2003

"Leontief Input-Output Modelling" Addison-Wesley, Pearson Education

"Computational Linear Algebra with Models", Gareth Williams, (Boston: Allyn and Bacon, 1978), pp. 123-127.

"Scalar, Vectors & Matrices", J. Walt Oler, Texas Teach University, 1980

EISPACK Guide, "Matrix Eigensystem Routines", Smith B. T. at al. 1976

"Numerical Methods that usually work", F. S. Acton, The Mathematical Association of America, 1990

"Analysis Numerical Methods", E. Isaacson, H. B. Keller, Wiley & Sons, 1966

"Calculo Numérico", Neide M. B. Franco, 2002

"Metodos Numericos" Sergio R. De Freitas, 2000

"Numerical Mathematics in Scientific Computation", G. Dahlquist, Å. Björck, vol II.

"Advanced Excel for scientific data analysis" , Robert de Levie, Oxford University Press, 2004

Credits

Manu thanks to:

Gregory Klein,

Bernard Wagner

and Carlos Aya

The library FunCustomize.dll, appears by courtesy of Laurent Longre (<http://longre.free.fr>)

Many thanks are also due to Prof. Robert de Levie for numerous corrections and valuable comments.

WHITE PAGE



© 2006, by Foxes Team
ITALY

Dic. 2006