



FOXES TEAM

Tutorial of Numerical Analysis with Matrix.xla

Matrices and Linear Algebra

TUTORIAL OF NUMERICAL ANALYSIS FOR MATRIX.XLA

Matrices and Linear Algebra

© 2004, by Foxes Team
Piombino, ITALY
leovlp@libero.it

4. Edition
4 Printing: May 2004

Index

About this Tutorial	8
MATRIX.XLA	8
Why Matrix.xla has same functions that are also in Excel?	8
Array functions	10
What is an array function	10
How to insert an array function	10
System solution	10
Adding two matrices	12
How to get help on line	14
MATRIX installation.....	15
How to install	15
How to uninstall	17
Linear System	19
Gauss-Jordan algorithm	20
The pivoting strategy	21
Integer calculation	23
Several ways for using the Gauss-Jordan algorithm	26
Solving linear system (non singular)	26
Solving simultaneously m linear systems	26
Inverse matrix computing	26
Determinant computing	27
Linear non singular system	28
Round-off errors	28
Full pivoting or partial pivoting?	30
Solution stability	31
Complex systems	34
About complex matrix format	36
Determinant	37
Gaussian elimination	37
Hill-conditioned matrix	38
Laplace's expansion	39
Simultaneous Linear Systems	41
Inverse matrix	41
Round-off error	42
How to avoid decimal number	44
Homogeneous and Singular Linear Systems	45
Parametric form	46
Rank and Subspace	47
General Case - Rouché-Capelli Theorem	49
Homogeneous System Cases	50

Non Homogeneous System Cases.....	51
Triangular Linear Systems.....	52
Triangular factorization.....	52
Forward and Back substitutions.....	52
LU factorization.....	53
Block-Triangular Form.....	56
Linear system solving.....	56
Determinant computing.....	57
Permutations.....	57
Eigenvalues Problem.....	57
Several kinds of block-triangular form.....	58
Permutation matrices.....	58
Matrix Flow-Graph.....	59
The score-algorithm.....	60
Eigenproblems.....	68
Eigenvalues.....	68
Characteristic Polynomial.....	68
Roots of characteristic polynomial.....	69
Case of symmetric matrix.....	69
Example – How to check the Cayley-Hamilton theorem.....	71
Eigenvectors.....	72
Step-by-step method.....	72
Example - Simple eigenvalues.....	72
Example - How to check an eigenvector.....	73
Example - Eigenvalues with multiplicity.....	74
Example - Eigenvalues with multiplicity not corresponding to eigenvectors.....	75
Example - Complex Eigenvalues.....	75
Example - How to check a complex eigenvector.....	77
Similarity Transformation.....	78
Factorization methods.....	79
Eigensystems versus resolution methods.....	79
Jacobi's transformation of symmetric matrix.....	80
Orthogonal matrices.....	81
Eigenvalues with QR factorization method.....	82
Real and complex eigenvalues with QR method.....	84
How to find polynomial root with eigenvalues.....	86
Powers' method.....	88
Eigensystems with power's method.....	91
How to validate an eigensystem.....	93
How to generate a random symmetric matrix with given eigenvalues.....	94
Eigenvalues of tridiagonal matrix.....	95
Eigenvalues of tridiagonal uniform matrix.....	97
Why so many different methods?.....	102
Generalized Eigenproblem.....	103
Equivalent non symmetric problem.....	103
Equivalent symmetric problem.....	104
Case diagonal matrix.....	105
Example - How to get mode shapes and frequencies for a multi-degree of freedom structure.....	106
Linear regression.....	108
Recalls.....	108
Linear Regression models.....	109
Linear model: $a_0 + a_1 x_1 + a_2 x_2$	109
Polynomial model: $a_0 + a_1 x + a_2 x^2 + a_3 x^3$	110
Two variables polynomial model: $a_0 + a_1 x + a_2 y + a_3 xy + a_4 x^2 + a_5 y^2$	112

Linear model with fixed intercept: $k x$	112
Non linear regression - Transformable linear models	114
Quasi linear model.....	114
Exponential curve fit: $y_0 e^{kx}$	114
Logarithmic curve fit: $b_0 + b_1 \ln(x)$	117
Rational curve fit: $(b_0 + b_1 x)^{-1}$	118
Power curve fit: $a x^\alpha$	120
Interpolation.....	122
Recalls	122
Why to interpolate?.....	122
Piecewise polynomial interpolation schema.....	123
Linear Interpolation.....	124
Parabolic Interpolation.....	124
Cubic interpolation.....	127
Instability of higher interpolation degree	128
Piecewise polynomial regression schema.....	130
Piecewise regression versus global regression	132
Matrix Tool	134
Matrix tool menubar	134
Selector tool.....	134
Scraps Paster tool.	135
Matrix Generator.....	137
Macros stuff.	139
Macro Gauss-step-by-step	139
Macro Shortest-Path.....	140
Macro Draw Graph	141
Macro Block reduction	142
Limits in matrix computation	143
Functions Reference.....	145
Function M_ABS(V).....	146
Function M_ADD(Mat1, Mat2).....	146
Function M_BAB(A, B).....	146
Function M_DET(Mat, Mat, [IMode], [Tiny])	147
Function M_DET_C (Mat, [Cformat]).....	147
Function M_DET3(Mat3)	148
Function M_INV(Mat, [IMode], [Tiny]).....	148
Function M_POW(Mat, n).....	149
Function M_EXP(A, [Algo], [n]).....	149
Function M_EXP_ERR(A, n)	150
Function M_PROD(Mat1, Mat2, ...).....	150
Function M_PROD_S(Mat, k).....	151
Function M_MULT3(Mat3, Mat).....	152
Function M_SUB(Mat1, Mat2)	152
Function M_TRAC(Mat).....	153

Function M_DIAG(Diag)	153
Function MatDiagExtr(Mat, [Diag])	153
Function M_T(Mat)	154
Function M_RANK(A)	154
Function M_DIAG_ERR(A).....	154
Function M_TRIA_ERR(A)	155
Function M_ID()	155
Function ProdScal(v1, v2)	155
Function ProdVect(v1, v2)	156
Function MatEigenvalue_Jacobi(Mat, Optional MaxLoops).....	156
<i>Eigenvalues problem with Jacobi step by step</i>	158
Function MatRotation_Jacobi(Mat).....	159
Function Mat_Block(Mat,).....	160
Function Mat_BlockParm(Mat,)	161
Function MatEigenvalue_QR(Mat)	162
Function MatEigenvector(A, Eigenvalues, [MaxErr]).....	163
Function MatEigenvector_C(A, Eigenvalue, [MaxErr]).....	163
Function MatEigenvector_Jacobi(Mat, Optional MaxLoops).....	164
Function MatEigenvalue_QL(Mat3, [IterMax]).....	164
Function MatEigenvalue_TridUni(n, a, b, c)	166
Function MatEigenvector3(Mat3, Eigenvalues, [MaxErr])	166
Function MatCharPoly(Mat).....	167
Function Poly_Roots(Coefficients, [ErrMax])	167
Function MatEigenvalue_max(Mat, [IterMax]).....	168
<i>A global localization method for real eigenvalues</i>	169
Function MatEigenvector_max(Mat, [Norm], [IterMax]).....	170
Function MatEigenvalue_pow(Mat, [IterMax])	170
Function MatEigenvector_pow(Mat, [Norm], [IterMax]).....	171
Function MatEigenvector_inv(Mat, Eigenvalue).....	171
<i>About perturbed eigenvalues</i>	172
Matrices Generator	174
Function Gauss_Jordan_step(Mat, [Typ], [IntValue]).....	177
Function SYSLIN(A, b, [IMode], [Tiny])	178
Function SYSLIN3(Mat3, v).....	179
Function SYSLIN_ITER_G(A, b, X0, Optional Nmax).....	180
Function SYSLIN_T(Mat, b, [typ], [tiny])	181
Function SYSLIN_ITER_J(Mat, U, X0, Optional Nmax).....	181
Function SYSLINSING(A, [b], [MaxErr]).....	182
Function TRASFLIN(A, x, Optional B).....	184

<i>Matrix Geometric action</i>	184
Function Gram_Schmidt(A)	186
<i>Gram-Schmidt's Orthonormalization</i>	186
<i>Double step Gram-Schmidt method</i>	187
Function Mat_Cholesky(A)	188
Function Mat_LU(A, optional Pivot).....	188
Function Mat_QR(Mat)	190
Function Mat_QR_iter(Mat, [MaxLoops])	191
Function MatExtract(A, i_pivot, j_pivot)	192
Function MatOrtNorm(A)	192
Function Path_Floyd(G)	193
Function Path_Min(G).....	193
<i>Graphs theory recalls</i>	193
<i>Shortest path</i>	196
Function SVD - Singular Value Decomposition	198
Function MatMopUp(M, [ErrMin])	200
Function MatCovar(A).....	200
Function MatCorr(A)	200
Function REGRL(Y, X, [ZeroIntcpt])	202
Function REGRP(degree, f, x, [ZeroIntcpt])	203
Function Interpolate(x, Knots, [Degree], [Points])	203
Function MatCmpn(Coeff)	204
Function Poly_Roots_QR(Coefficients).....	205
Function MatRot(n, teta, p, q).....	205
Conditioned Number	206
Function VarimaxRot(FL, [Normal], [MaxErr], [MaxIter])	207
Function VarimaxIndex(Mat, [Normal]).....	208
Function MatNormalize(Mat, [NormType], [Tiny])	208
Function MatNorm(v, [NORM]).....	209
Function M_MULT_C(Mat1, Mat2, [Cformat])	210
Function M_INV_C(A, [Cformat]).....	211
Function ProdScal_C(v1, v2,).....	211
Function SYSLIN_C(A, B, [Cformat])	212
Function Simplex(Funct, Constrain, [Opt])	213
Function RRMS(v1, [v2])	215
Function MatPerm(Permutations).....	215
Function Mat_Hessemberg(Mat).....	216
Function Mat_Adm(Branch).....	216
<i>Linear Electric Network</i>	217
<i>Thermal Network</i>	218

TUTORIAL FOR MATRIX.XLA

Function Mat_Leontief(ExTab, Tot).....	220
<i>Input Output Analysis</i>	220
References	222

About this tutorial

MATRIX.XLA

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

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

Eigenvalues and Eigenvectors of symmetric matrix with Jacobi algorithm, Jacobi's rotation matrix. Eigenvalues with QR algorithm, Characteristic polynomial, Polynomial roots with QR algorithm

Generate random matrix with given eigenvalues and random matrix with given Rank or Determinant, Several useful matrix - Hilbert's, Householder's, Tartaglia's, Vandermonde's - are supported

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

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

The main purpose of this document is to show how to work with matrices and vectors in Excel and to use matrices and vectors for solving linear systems. This tutorial is written with the aim to teach how to use better all matrix.xla functions. Of course it speaks about math and linear algebra fundamental results but this is not a math book. You rarely find here theorems and demonstrations. You can find, on the contrary, many examples that explain, step by step, how to reach the result that you need. Just straight and easy. And, of course, we speak about Microsoft Excel but this is not a tutorial for Excel. Tips and tricks for this program can be found in many Internet sites.

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

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

Matrix.xla algorithms are open

The main difference is into the algorithms used; or in other words, in the way that the functions are implemented. In Matrix.xla all the algorithms are open and the user can verify, if he wants, how each function works. Numerically speaking, the function that perform matrix inversion in Excel and in Matrix.xla, for example, can give different results, especially in high accuracy calculation.

The main difference is that Matrix.xla Inversion function uses the popular Gauss-Jordan

TUTORIAL FOR MATRIX.XLA

algorithm -explained in many books and sites - while the Excel built-in functions are code-proprietary. In other few cases we have simply create new functions to avoid the original verbose names (MTRANSPOSE(), or MATR.TRASPOSTA () in Italian version, are substituted by the more handy M_T)

I thank all those who suggested me to write this tutorial and - indeed - who encouraged me. I am grateful to all those who will provide constructive criticisms.

Leonardo Volpi

May. 2004

Array functions

What is an array function

A function that returns multiple values is called "array function". Matrix.xla contains lots of these functions. All functions that return a matrix are array functions. Inversion, multiplication, sum, vector product, etc. are examples of array functions. On the contrary, Norma 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 it, you must give the keys sequence CTRL+SHIFT+ENTER; otherwise you will get only the value of the first cell. The sequence must be given just after inserting the function parameters. Keep down both key CTRL and SHIFT (do not care the order) and then press ENTER.

How to insert an array function

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

System solution


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

$$Ax = b$$

Where:

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

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

Now insert the function by menu or by icon as well 

	G5		=						
	A	B	C	D	E	F	G	H	
1									
2		Ax = b							
3									
4		A			b		x		
5	1	1	1		4				
6	1	2	2		2				
7	1	3	4		3				
8									
9									
10									
11									

Select the area you want to paste the result x

TUTORIAL FOR MATRIX.XLA

The screenshot shows an Excel spreadsheet with the following data:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1																	
2		Ax = b															
3																	
4		A				b	x										
5	1	1	1		4		=										
6	1	2	2		2												
7	1	3	4		3												
8																	
9																	
10	Select the area you want to paste the result x																
11																	
12																	
13																	
14																	
15																	
16																	
17																	
18																	
19																	
20																	

The 'Incolla funzione' (Paste Function) dialog box is open, showing the 'SYSLIN' function selected under the 'Matematiche e trig.' category. The dialog also shows a list of other functions and the description 'Solve Linear System.'

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

The screenshot shows an Excel spreadsheet with the following data:

	A	B	C	D	E	F	G	H	I	J	K	L	M
1													
2		Ax = b											
3													
4		A			b		x						
5	1	1	1		4		5:E7)						
6	1	2	2		2								
7	1	3	4		3								

The formula bar shows: **MATR.INVERSA** \times \checkmark **=** **=SYSLIN(A5:C7,E5:E7)**

A dialog box titled "SYSLIN" is open, showing the following information:

Mat A5:C7 = {1;1;1\1;2;2\1;3;4}

v E5:E7 = {4\2\3}

= {6\ -5\ 3}

Solve Linear System.

v

Resultato formula = 6

Buttons: OK, Annulla

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

That is:

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

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

	A	B	C	D	E	F	G	H	I
1									
2		Ax = b							
3									
4		A			b		x		
5	1	1	1		4		6		
6	1	2	2		2		-5		
7	1	3	4		3		3		
8									
9									
10									
11									
12									

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

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

Adding two matrices

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

Example - Adding two matrices

$$\begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

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

In order to perform this summation follow these steps.

- 1) Enter the matrices into the spreadsheet.
- 2) Select empty cells so that a 2×2 range is highlighted.
- 3) Write a formula that adds the two ranges. Either write =B4:C5+E4:F5 directly or write "=", then select the first matrix; after, write "+" and then select the second matrix. Do not press <Enter>. At this point the spreadsheet should look something like the figure below. Note that the entire range B8:C9 is selected.

TUTORIAL FOR MATRIX.XLA

	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 the 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.

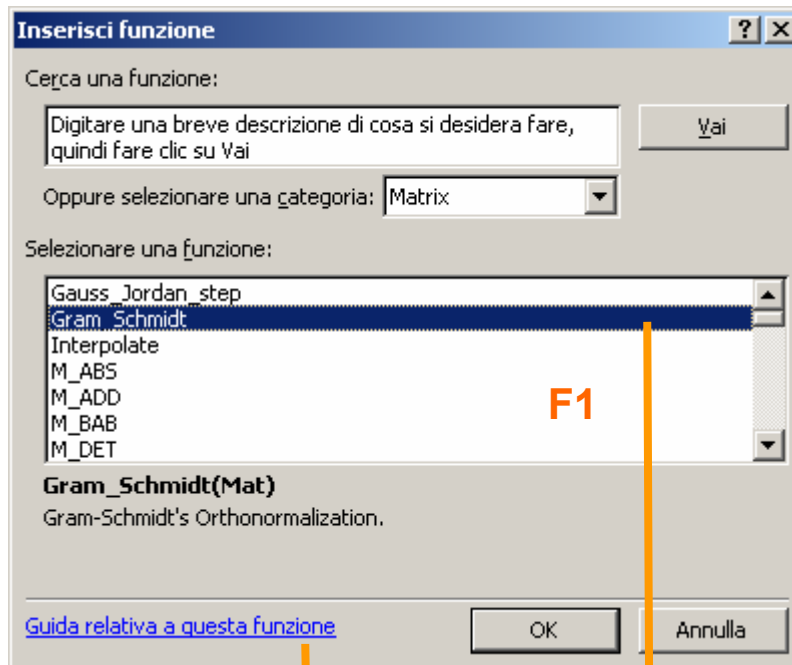
Let's see this example that show how to calculate the linear combination of two vectors

	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		{=A12*B11:B13+C12*D11:D13}					
16							
17							

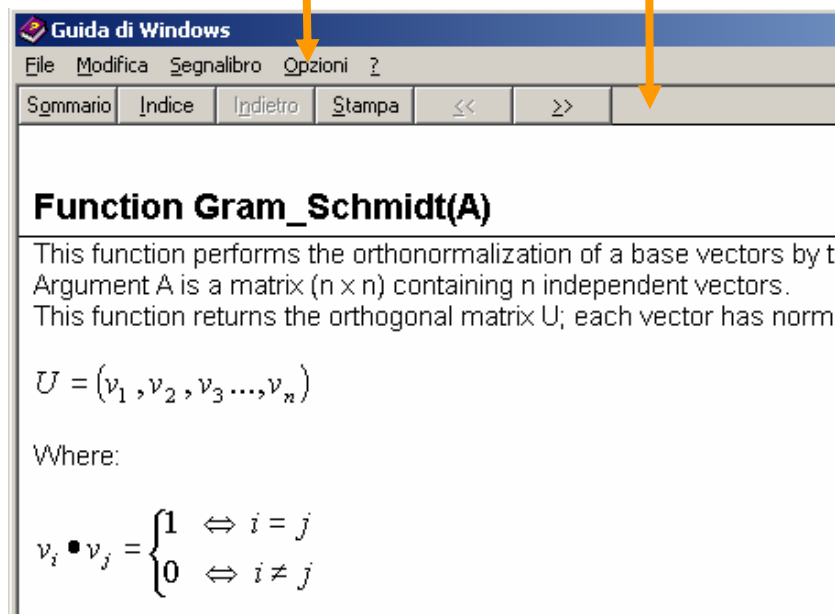
Fine, inst'it?

How to get help on line

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



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



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

MATRIX installation

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

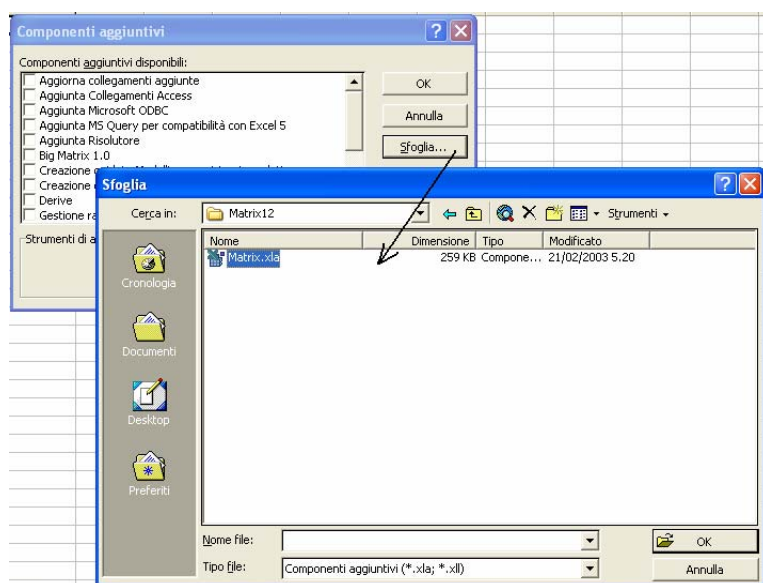
- MATRIX.XLA Excel addin file
- MATRIX.HLP Help file
- MATRIX.CSV Functions information(only for XNUMBERS addin)
- FUNCUSTOMIZE.DLL¹ Dynamic Library for addin setting

How to install

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

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

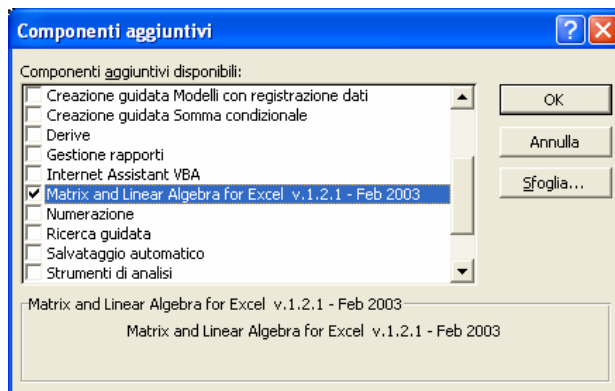
- 1) Open Excel
- 2) From the Excel menu toolbar select "Tools" and then select "Add-ins"..
- 3) Once in the Addins Manager, browse for "**matrix.xla**" and select it
- 4) Click OK



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

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

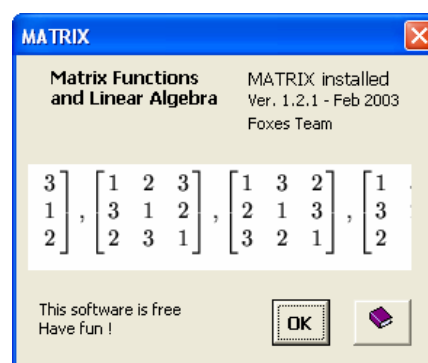
After the first installation, **matrix.xla** will be add to the Addin' list manager




When Excel starts, all addins checked in the Addins Manager will be automatically loaded


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

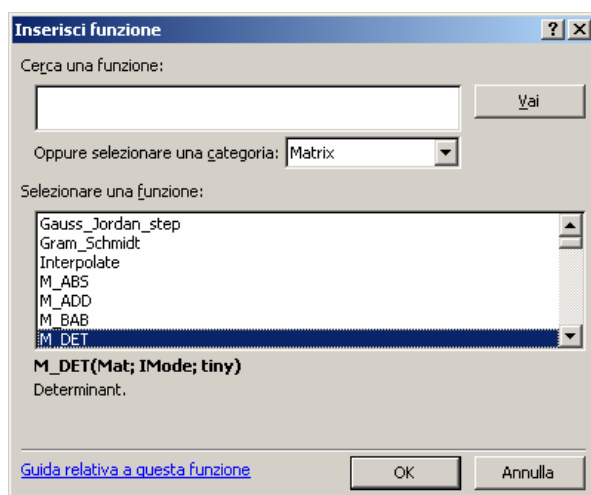
If all goes OK you should see the welcome popup of matrix.xla. This appears only when you select "on" the check box of the Addin Manager. When Excel automatically loads Matrix.xla, this popup remains hidden.



The Matrix Icon  is added to the menu bar. From this button you can start the *Matrix Tool* menu bar



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



How to uninstall

This package never alter your system files

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

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

WHITE PAGE

Linear Systems

This chapter explains how to solve linear system problems, with the aid of many examples. They cover the most part of cases: systems with single, infinity and none solution. Several algorithms are shown: Gauss-Jordan, Crout's LU factorization, SVD decomposition

Linear System

Example 1. Solve the following 4x4 linear system

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

Where \mathbf{A} and \mathbf{b} are:

1	9	-1	4
2	0	1	1
1	2	-4	0
1	5	1	1

18
-2
17
7

Square matrix. If the number of unknowns and the number of equations are the same, the system has surely one solution if the determinant of the matrix \mathbf{A} is not zero. That is, \mathbf{A} is non-singular. In that case we can solve the problem with the **SYSLIN** function.

D7		=M_DET(B2:E5)									
	A	B	C	D	E	F	G	H	I	J	
1			A				b		x		
2		1	9	-1	4		18		1		
3		2	0	1	1		-2		2		
4		1	2	-4	0		17		-3		
5		1	5	1	1		7		-1		
6											
7											
8											
9											
10											
11											

det(A) = -139

=SYSLIN(B2:E5;G2:G5)

=M_DET(B2:E5)

The determinant can be computed with M_DET() function or with the built-in function MDETERM (Excel USA version) as well.

Gauss-Jordan algorithm

Gauss-Jordan is probably the most popular algorithm for solving linear systems. Functions SYSLIN and SYSLINSING of Matrix.xla use this method with pivoting strategy. Ancient, solid, efficient and - last but not least - elegant.

The main goal of this algorithm is to reduce the matrix **A** of the system **A x = b** to a triangular² or diagonal³ matrix with all diagonal elements = 1 by few kind of row operations: linear combination; normalization, exchange.

Let's see how it works

Example: The following 3x3 system has solution ($x_1 = -1$; $x_2 = 2$; $x_3 = 1$). We can verify it by direct substitution.

$$\begin{cases} 4x_1 + x_2 = -2 \\ -2x_1 - 2x_2 + x_3 = -1 \\ x_1 - 2x_2 + 2x_3 = -3 \end{cases} \Leftrightarrow \begin{array}{ccc|c} 4 & 1 & 0 & -2 \\ -2 & -2 & 1 & -1 \\ 1 & -2 & 2 & -3 \end{array}$$

Let's begin to build the complete matrix (3x4) with the matrix coefficients and the constant vector (gray) as shown on the right. Our goal is to reduce the matrix coefficients to the identity matrix.

Choose the first diagonal element a_{11} ; it is called the "pivot" element

1. Normalization step: if pivot $\neq 0$ and pivot $\neq 1$ then we divide all first row for pivot = 4.

1	0.25	0	-0.5
-2	-2	1	-1
1	-2	2	-3

2. Linear combination: if $a_{21} \neq 0$ then substitutes the second row with the difference between the second row itself and the first row multiplied by a_{21}

1	0.25	0	-0.5
0	-1.5	1	-2
1	-2	2	-3

3. Linear combination: if $a_{31} \neq 0$ then substitutes the second row with the difference between the second row itself and the first row

1	0.25	0	-0.5
0	-1.5	1	-2
0	-2.25	2	-2.5

As we can see the first column has all zeros except for the diagonal element that is 1. Repeating the process for the second column - with pivot a_{22} - and for the third column - with pivot a_{33} - we will perform the matrix diagonalization; the last column will contain, at the end, the solution of the given system

In Excel, we can do these tasks by using the power of array functions. Below there is an example of Excel resolution of a system by Gauss-Jordan algorithm

Note that all the rows are obtained by array operations {...}. You must insert them by the CTRL+SHIFT+ENTER key sequence.

² Properly called Gauss algorithm

³ Properly called Gauss-Jordan algorithm

	A	B	C	D	E
1	4	1	0	-2	
2	-2	-2	1	-1	
3	1	-2	2	-3	
4					
5	1	0.25	0	-0.5	{=A1:D1/A1}
6	0	-1.5	1	-2	{=A2:D2-A2*A5:D5}
7	0	-2.25	2	-2.5	{=A3:D3-A3*A5:D5}
8					
9	1	0	0.16667	-0.83333	{=A5:D5-B5*A10:D10}
10	0	1	-0.6667	1.33333	{=A6:D6/B6}
11	0	0	0.5	0.5	{=A7:D7-B7*A10:D10}
12					
13	1	0	0	-1	{=A9:D9-C9*A15:D15}
14	0	1	0	2	{=A10:D10-C10*A15:D15}
15	0	0	1	1	{=A11:D11/C11}

We see in the last column the solution (-1 ; 2 ; 1). Formulas for each row are shown on the right

Swap rows If one pivot is zero we cannot normalize the corresponding row. In that case we will swap the row with another row that has no zero in the same position. Also this operation does not affect the final solution at all; it is equivalent to reorder the algebraic equation of the given system

Example: The following 3x3 system has solution ($x_1 = 5$; $x_2 = -3$; $x_3 = 7$)

	A	B	C	D	E
1	0	1	0	-3	
2	-2	-2	1	3	
3	1	-2	2	25	
4					
5	1	1	-0.5	-1.5	{=A2:D2/A2}
6	0	1	0	-3	{=A1:D1}
7	0	-3	2.5	26.5	{=A3:D3-A3*A5:D5}
8					
9	1	0	-0.5	1.5	{=A5:D5-B5*A10:D10}
10	0	1	0	-3	{=A6:D6/B6}
11	0	0	2.5	17.5	{=A7:D7-B7*A10:D10}
12					
13	1	0	0	5	{=A9:D9-C9*A15:D15}
14	0	1	0	-3	{=A10:D10}
15	0	0	1	7	{=A11:D11/C11}

Note that the first pivot $a_{11}=0$. We cannot normalize this row and in this case the algorithm could not start

In this case we swap the first row with the second one. Now the new pivot is -2 and the normalization can be done.

Note that now the second row has the element $a_{21} = 0$; so we simply leave the row unchanged. The linear combination doesn't need in this case

The pivoting strategy

Pivoting can be always performed. In the above example we have exchanged one zero pivot with any other non-zero pivot in order to continue the Gauss algorithm. But there is another reason for which the pivoting method is adopted: the round off error minimization.

**Pivoting
reduce round
off error**

The Gaussian elimination algorithm can have a large number of operations. If we count the operations for one system resolution, we will discover that there are order $n^3/3$ operations. So, if the number of unknowns doubles, the number of operations increases by a factor of 8. If $n = 200$, then there are approximately 8/3 million operations! Certainly, one might begin to worry

about the accumulation of round off error. One method to reduce the round off error is to avoid division by small numbers, and this is known as *row pivoting* or *partial pivoting* strategy of the Gaussian elimination algorithm.

Let's see the following remarkable example of a 2x2 system

Solution is $(x_1; x_2) = (1; 1)$ as we can easily verify by substitution

1	987654321	987654322
123456789	-1	123456788

If we apply the Gauss-Jordan algorithm, with 15 precision digits, we have:

	A	B	C	D
1	1	987654321	987654322	
2	123456789	-1	123456788	
3				
4	1	987654321	987654322	{=A1:C1/A1}
5	0	-1.2193E+17	-1.21933E+17	{=A2:C2-A2*A4:C4}
6				
7	1	0	0.999999762	{=A4:C4-B4*A8:C8}
8	0	1	1	{=A5:C5/B5}

The pivot = 1

The solution has an error of about 1E-7

While, on the contrary, if we exchange the order of algebraic equations, we have

	A	B	C	D
1	123456789	-1	123456788	
2	1	987654321	987654322	
3				
4	1	-8.1E-09	0.999999992	{=A1:C1/A1}
5	0	987654321	987654321	{=A2:C2-A2*A4:C4}
6				
7	1	0	1	{=A4:C4-B4*A8:C8}
8	0	1	1	{=A5:C5/B5}

Pivot = 123456789 >> 1

The solution is much better having an error of less than 1E-15

As we can see, this little trick can improve the general accuracy.

The standard Gauss-Jordan algorithm always search for the max absolute value into the element under the actual pivot; if the max value is greater then actual pivot, then the actual pivot row and the row of the max value are exchanged.

Not all elements, thus, can be used as pivot exchange. In the matrix to the right we could use as pivot a33 only the element a33, a43, a53, a63 (yellow cells).
For example:

if $|a_{63}| = \max(|a_{33}|, |a_{43}|, |a_{53}|, |a_{63}|)$

Then the row 6 and 3 are swapped and the old element a63 becomes the new pivot 33

1	a12	a13	a14	a15	a16
0	1	a23	a24	a25	a26
0	0	a33	a34	a35	a36
0	0	a43	a44	a45	a46
0	0	a53	a54	a55	a56
0	0	a63	a64	a65	a66

Full Pivoting

In order to extend the area where to search the max value for pivot we could exchange rows and columns. But when we swap two columns the corresponding unknown variables are also exchanged. So, to get the final solution in the original given sequence, we have to perform a permutation in the reverse order that we have done. This makes the final algorithm process a bit complicate because we have to store all columns permutations performed.

The full pivoting method extends the search area for max value

For example, if the pivot is the element 33, then the algorithm searches for the absolute max value into the yellow area. If max value is found at a56, then the row 5 and 3 are swapped and then, the column 5 and 3 are swapped.

Unknown x5 and x3 are permuted

1	a12	a13	a14	a15	a16
0	1	a23	a24	a25	a26
0	0	a33	a34	a35	a36
0	0	a43	a44	a45	a46
0	0	a53	a54	a55	a56
0	0	a63	a64	a65	a66

Functions SYSLIN and SYSLINSING of matrix.xla use the Gauss-Jordan algorithm with full pivoting strategy

Integer calculation

In the above examples we have seen that the Gauss elimination steps introduce decimal numbers - with round off error -, even if solutions and coefficients of the system are all integer.

There is a way to avoid such decimal round off error and preserve the all accuracy? The answer is yes, but in general, only for integer matrices.

This method - a variant of the original Gauss-Jordan - is very similar to the one that is performed by hand from students. It is based on the "minimum common multiple" MCM (also LCM Least Common Multiple) and it is conceptually very simple

Assume to have the following two rows: the first one is the pivot row and the second one to be reduced.

Pivot is $a_{11} = -6$;

Element to set zero is $a_{21} = 4$;

$mcm = MCM(6, 4) = 12$

Multiply the first one for $mcm / a_{21} = 12/4 = 3$

And the second one for $-mcm / a_{11} = -12/(-6) = 2$

-6	0	5	9
4	3	0	10

<== pivot row; multiply for 2

<== to be reduced; multiply for 3

-12	0	10	18
12	9	0	30

now, add the two rows

-6	0	5	9
0	9	10	48

<== the first row remain unvaried

<== substitutes the result to the second row

As we can see, we can reduce a row without introducing decimal numbers

Let's see how it works step by step by the function **gauss_jordan_setp()** of matrix.xla.

	A	B	C	D	E	F	G	H	I	J	K	L
5	-6	0	5	9								
6	4	3	0	10								
7	0	-1	2	4								
8												
9	-6	0	5	9								
10	0	-9	-10	-48								
11	0	-1	2	4								
12												
13	-6	0	5	9								
14	0	-9	-10	-48								
15	0	0	28	84								
16												
17	168	0	0	168								
18	0	-9	-10	-48								
19	0	0	28	84								
20												
21	168	0	0	168								
22	0	-126	0	-252								
23	0	0	28	84								
24												
25	1	0	0	1								
26	0	1	-0	2								
27	0	0	1	3								
28												

{=Gauss_Jordan_step(A5:D7,,TRUE)}
 {=Gauss_Jordan_step(A9:D11,,TRUE)}
 {=Gauss_Jordan_step(A13:D15,,TRUE)}
 {=Gauss_Jordan_step(A17:D19,,TRUE)}
 {=Gauss_Jordan_step(A21:D23,,TRUE)}
 Only the last normalization step can introduce decimal roundoff errors

Note that the 3° parameter setting the integer algorithm, if False, the operation will perform in the standard decimal way.

Only the last step could introduce decimal numbers; the previous steps are always exact. Unfortunately, this method cannot be adopted in general because numbers grow up each step and they can became too large (overflow error)

Tip The above example can be fast reproduced. After inserting the function in the range A9:D11, give the CTRL+C to copy the range still selected; select the cell A13 and give CTRL+V to paste the new matrix; repeat this simple step still you reach the final identity 3x3 matrix; the solution will be in the last column.

This sequence shows how you do.

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

Given a complete system matrix in range B2:E4, select the range A6:E8, just below the given matrix

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

Insert the array function Gauss_Jordan_step with the CTRL+SHIFT+ENTER key sequence and the given parameter ("VERO" means "TRUE" in English language)

You should see the first step matrix. Leave the selected range and give the copy command (CTRL+C)

TUTORIAL FOR MATRIX.XLA

	A	B	C	D	E	F
1		Linear system complete matrix				
2		-6	0	5	9	
3		4	3	0	10	
4		0	-1	2	4	
5						
6		-6	0	5	9	
7		0	-9	-10	-48	
8		0	-1	2	4	
9						
10						
11						
12						
13						

Select the cell B10, under the 1st step matrix. Make sure that the range below is empty.

	A	B	C	D	E	F
1		Linear system complete matrix				
2		-6	0	5	9	
3		4	3	0	10	
4		0	-1	2	4	
5						
6		-6	0	5	9	
7		0	-9	-10	-48	
8		0	-1	2	4	
9						
10		-6	0	5	9	
11		0	-9	-10	-48	
12		0	0	28	84	
13						

Now, simply give the paste command (CTRL+V) and the 2nd step matrix will appear

Repeating the above steps you could get all the Gauss-Jordan step matrix, with row pivot algorithm either in decimal or in integer mode,

Several ways for using the Gauss-Jordan algorithm

The reduction matrix method can be used in several useful ways. Here some basic cases:

Solving linear system (non singular)

$$Ax = b \Rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

The complete matrix (3 x 4) is

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & x_2 \\ 0 & 0 & 1 & x_3 \end{bmatrix}$$

At the end, the last column is the solution of a given system; the original matrix A is transformed into the identity matrix.

Solving simultaneously m linear systems

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

The complete matrix (3 x 3+m) is:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_{11} & b_{12} & b_{1m} \\ a_{21} & a_{22} & a_{23} & b_{21} & b_{22} & \dots & b_{2m} \\ a_{31} & a_{32} & a_{33} & b_{31} & b_{32} & b_{3m} \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & x_{11} & x_{12} & x_{1m} \\ 0 & 1 & 0 & x_{21} & x_{22} & \dots & x_{2m} \\ 0 & 0 & 1 & x_{31} & x_{32} & x_{3m} \end{bmatrix}$$

At the end, the solutions of the m system are the last m column of the complete matrix

Inverse matrix computing

This problem is similar to the above one, except that the matrix B is the identity matrix
In fact, for definition:

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

$$AX = I \Leftrightarrow X = A^{-1}$$

$$AX = I \Rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The complete matrix (3 x 6) is:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \overbrace{x_{11} & x_{12} & x_{13}}^{A^{-1}} \\ 0 & 1 & 0 & x_{21} & x_{22} & x_{23} \\ 0 & 0 & 1 & x_{31} & x_{32} & x_{33} \end{bmatrix}$$

At the end, the inverse matrix is into the 3 last columns of the complete matrix

Determinant computing

For this scope we need only reduce the given matrix to a triangular form.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ 0 & t_{22} & t_{23} \\ 0 & 0 & t_{33} \end{bmatrix}$$

So the determinant can be easy computed by the following

$$Det(A) = t_{11} \cdot t_{22} \cdot t_{33}$$

Linear non singular system

The function SYSLIN finds the solution of non-singular linear system with Gauss-Jordan algorithm with full pivot strategy.

Example: solve the following matrix equation

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (1)$$

The solution is

$$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b} \quad (2)$$

You can get the numerical solution in two different ways. The first one is the direct application of the formula (2); the second one is the resolution of the simultaneous linear system (1)

Example: Find the solution of the linear system having the following \mathbf{A} (6 x 6) and \mathbf{b} (6 x 1)

-10	93	6.7	5	-47	0	47.7
-0.5	-28	1	7	0	0	-20.5
0	0	1	8	35	-47	-3
45	0	-13	3	-23	-59	-47
65	0.1	3	32	0	0	100.1
-7	4	-1.5	-1	0	4.9	-0.6

We solve this linear system with both methods: by Excel MINVERSE() and SYSLIN() function. We found the unitary solution (1, 1, 1, 1, 1, 1) (Note that the sum of each row is equal to the constant terms)

	A	B	C	D	E	F	G	H	I
1	Linear System $\mathbf{A} \mathbf{x} = \mathbf{b}$						\mathbf{b}	\mathbf{x}	$\mathbf{A}^{-1} \mathbf{b}$
2	-10	93	6.7	5	-47	0	47.7	1.0000000000000000	1.0000000000000000
3	-0.5	-28	1	7	0	0	-20.5	1.0000000000000000	1.0000000000000000
4	0	0	1	8	35	-47	-3	1.0000000000000000	1.0000000000000000
5	45	0	-13	3	-23	-59	-47	1.0000000000000000	0.9999999999999980
6	65	0.1	3	32	0	0	100.1	1.0000000000000000	1.0000000000000000
7	-7	4	-1.5	-1	0	4.9	-0.6	1.0000000000000000	1.0000000000000000
8									
9									
10							{=SYSLIN(A2:F7,G2:G7)}		{=MMULT(MINVERSE(A2:F7),G2:G7)}
11									

Note also that the methods give similar - but not equal - results, because their algorithms are different. In this case both the solutions are very accurate ($\approx 1\text{E-}15$) but this is not always true.

Round-off errors

Many times, the round-off errors can decrease the max accuracy obtained

Look at the following system:

-151	386	-78	-4	234	387
-76	194	-39	-2	117	194
-299994	599988	3	-2	299994	599989
2	-4	0	2	0	0
-100000	200000	0	0	100001	200001

The exact solution is the unitary solution (1, 1, 1, 1, 1, 1).

In order to measure the error, we use the following formula

$=ABS(x-ROUND(x, 0))$ where x is one approximate solution value

The total error is calculate with

$=AVERAGE(H2:H6)$

total error for SYSLIN function

$=AVERAGE(J2:J6)$

total error for MINVERSE function

	A	B	C	D	E	F	G	H	I	J
1	A					b	x (SYSLIN)	err	x (MINVERSE)	err
2	-151	386	-78	-4	234	387	0.999999999990788	9.21E-12	1.000000000000000	0
3	-76	194	-39	-2	117	194	0.999999999995399	4.6E-12	1.000000000000000	0
4	-299994	599988	3	-2	299994	599989	0.999999999995034	4.97E-12	0.999999999941792	5.82E-11
5	2	-4	0	2	0	0	1.0000000000000010	1.09E-14	1.000000000000000	3.55E-15
6	-100000	200000	0	0	100001	200001	0.999999999999989	1.1E-14	1.0000000000000040	3.95E-14
7								3.76E-12		1.17E-11
8										
9							$\{=SYSLIN(A2:F7,G2:G7)\}$		$\{=MMULT(MINVERSE(A2:F7),G2:G7)\}$	

As we can see the total errors of these solutions are thousand times greater that the one of the previous example.

Sometimes, round-off errors are so strong that can give totally wrong results. Look at this example.

A =	3877457	-3	-347	-691789	3877457
	-3773001	0	34	46	-3773001
	-286314	1	0	-2	-286314
	-377465	-12	6	4	-377465
	-1	0	-6	0	-1
					387
					194
					599989
					0
					200001

As we can easily see by inspection, the matrix is singular having the first and last column equal. So there is no solution for this system. But if you try to solve this system with MINVERSE() function you will get a wrong totally different result

This error is particularly sneaky because if we try to compute the determinant we get a finite (wrong) result

$$MDETERM(A) = -0.0082$$

As we have told, the algorithm used by Excel and matrix.xla are not equal. So we can try to compute the solution by SYSLIN() and the determinant by M_DET(). In this case the full pivot strategy of Gauss-Jordan works fine and give us the right answer.

	A	B	C	D	E	F	G	H
1	A					b	x (SYSLIN)	x (MINVERSE)
2	3877457	-3	-347	-691789	3877457	3185319	?	-2.77862E+16
3	-3773001	0	34	46	-3773001	-3772922	?	-1.10853903
4	-286314	1	0	-2	-286314	-286315	?	0.152519883
5	-377465	-12	6	4	-377465	-377444	?	1.0003915
6	-1	0	-6	0	-1	-5	?	2.77862E+16
7								
8						-0.0082097	=MDETERM(A2:E6)	
9						0	=M_DET(A2:E6)	

Full pivoting or partial pivoting?

The strategy of full-pivot reduces the round-off errors, so we could aspect that its accuracy is greater than partial-pivot strategy. But this is not always true. Sometime can happen that the full strategy gives an error similar or even greater then the one obtained by partial strategy.

In matrix.xla we can perform the partial gauss-Jordan algorithm using the didactic function Gauss_Jorda_step().

Example: Solve the following linear system. The matrix is the inverse of the 6x6 Tartaglia's matrix. The exact system solution is the vector [1, 2, 3, 4, 5, 6]

$$A = \begin{bmatrix} 6 & -15 & 20 & -15 & 6 & -1 \\ -15 & 55 & -85 & 69 & -29 & 5 \\ 20 & -85 & 146 & -127 & 56 & -10 \\ -15 & 69 & -127 & 117 & -54 & 10 \\ 6 & -29 & 56 & -54 & 26 & -5 \\ -1 & 5 & -10 & 10 & -5 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Let's see how both algorithms - full and partial pivoting - work⁴.

	A	B	C	D	E	F	G	H	I	J	K
1	A						b	x (full-pivot)	err	x (partial-pivot)	err
2	6	-15	20	-15	6	-1	0	1.000000000000001	8.7E-15	1.000000000000006	6E-14
3	-15	55	-85	69	-29	5	1	2.000000000000003	2.9E-14	2.000000000000020	2E-13
4	20	-85	146	-127	56	-10	0	3.000000000000008	7.6E-14	3.000000000000045	4.5E-13
5	-15	69	-127	117	-54	10	0	4.000000000000017	1.7E-13	4.000000000000086	8.6E-13
6	6	-29	56	-54	26	-5	0	5.000000000000036	3.6E-13	5.000000000000148	1.5E-12
7	-1	5	-10	10	-5	1	0	6.000000000000066	6.6E-13	6.000000000000232	2.3E-12
8									2.2E-13		8.9E-13

As we can see, in that problem, partial pivoting is even more accurate (but not too much) than full pivoting.

Then, why we complicate the algorithm with the full pivoting? The reason is that the Gauss-Jordan with full pivoting is generally more reliable for a large type of matrices. The round-off error control is more efficient. Disastrous mistakes are greatly reduced with full-pivot strategy.

⁴ Note that in these problems we have not inserted the results given by the MINVERSE() Excel function, because we ignore its algorithm in detail but, from a long series of testes, we have found that it works similar to the partial-pivot algorithm

Look at this example: Solve the following system

$$A = \begin{bmatrix} 1 & -3 & -9 & -1 & 38800000012 \\ 7 & -1 & 123000000045 & 1 & 0 \\ 0 & 1 & 0 & -2 & 2 \\ 23 & -12 & 6 & 4 & 1 \\ 2 & 0 & -6 & 0 & -1 \end{bmatrix} \begin{bmatrix} 38800000000 \\ 12300000052 \\ 1 \\ 22 \\ -5 \end{bmatrix}$$

Solving with Gauss-Jordan algorithm with both partial and full pivoting we note in this case a lack of accuracy more than thousand times for the first solution.

	A	B	C	D	E	F
1	A					b
2	1	-3	-9	-1	38800000012	38800000000
3	7	-1	123000000045	1	0	12300000052
4	0	1	0	-2	2	1
5	23	-12	6	4	1	22
6	2	0	-6	0	-1	-5
7						
8	partial pivot	error		full pivot	error	
9	1.0000019073	1.91E-06		1.0000000000	2.22E-16	
10	1.0000019073	1.91E-06		1.0000000000	7.77E-16	
11	1.0000000000	8.88E-16		1.0000000000	0.00E+00	
12	1.0000000000	0.00E+00		1.0000000000	5.55E-16	
13	1.0000000000	0.00E+00		1.0000000000	0.00E+00	
14		7.63E-07			3.11E-16	

We can observe that in general, partial pivoting becomes inefficient for matrices having large values in the right side. In that case the round-off errors grow sharply; full pivoting avoids this rare - but deep - accuracy loss.

Solution stability

Many times, coefficients of a linear system cannot be known exactly. Often, they derived from experimental results, measures, etc. So they can be affected by several errors. We are interested to investigate how the system solution changes with these errors. Many important studies has demonstrated that the solution behavior depends by the system coefficients matrix. Same matrices tend to amplify the errors of the coefficients or the constant terms, so the solution will be very different from the one of the "exact" system. When it happens we say "*hill-conditioned*" or "*unstable*" linear system.

Example: show that the following linear system with the Wilson's matrix, is very unstable

$$\begin{cases} 10x_1 + 7x_2 + 8x_3 + 7x_4 = 32 \\ 7x_1 + 5x_2 + 6x_3 + 5x_4 = 23 \\ 8x_1 + 6x_2 + 10x_3 + 9x_4 = 33 \\ 7x_1 + 5x_2 + 9x_3 + 10x_4 = 31 \end{cases}$$

10	7	8	7	32
7	5	6	5	23
8	6	10	9	33
7	5	9	10	31

The solution of the exact system is $\mathbf{x} = (1,1,1,1)$; now give same perturbations to the constant terms. For simplicity we give

TUTORIAL FOR MATRIX.XLA

$$\mathbf{b}' = \mathbf{b} + \Delta \mathbf{b} \quad \text{with } \mathbf{b} = 0.1$$

The solution of the perturbed system is now

$$\mathbf{A} \mathbf{x}' = \mathbf{b}' \quad \mathbf{x}' = \mathbf{x} + \Delta \mathbf{x}$$

Defining the system sensitivity coefficient as

$$S = (\Delta \mathbf{x} \%) / (\Delta \mathbf{b} \%) = (|\Delta \mathbf{x}| / |\mathbf{x}|) / (|\Delta \mathbf{b}| / |\mathbf{b}|)$$

We have $S \cong 400$.

	A	B	C	D	E	F	G	H
1	System perturbation			{=SYSLIN(A2:D5,E2:E5)}		{=SYSLIN(A2:D5,E2:E5)}		
2	Wilson matrix						{=E5:E8+G11}	
3								
4	A (4 x 4)				b	x	b'	x'
5	10	7	8	7	32	1	32.1	-0.2
6	7	5	6	5	23	1	23.1	3
7	8	6	10	9	33	1	33.1	0.5
8	7	5	9	10	31	1	31.1	1.3
9								
10	det(A) =			Δ b	Δ x	Δ b %	Δ x %	S
11	1			0.1	1.3136	0.17%	66%	394.2
12								
13	{=M_ABS(H5:H8)-M_ABS(F5:F8)}							
14								
15	{=D11/M_ABS(E5:E8)}							
	{=E11/M_ABS(F5:F8)}							
	{=G11/F11}							

A high value of S means high instability. In fact in this system for a small perturbation of about 0.2% of the constant terms we have the solution

-0.2, 3, 0.5, 1.3

Completely different from the exact one

1, 1, 1, 1

Note that Det =1

Even worst for the stability of the

following linear system

$$\mathbf{A} = \begin{bmatrix} 117 & 85 & 127 & 118 \\ 97 & 70 & 103 & 97 \\ 74 & 53 & 71 & 64 \\ 62 & 45 & 65 & 59 \end{bmatrix} \quad \begin{bmatrix} 447 \\ 367 \\ 262 \\ 231 \end{bmatrix}$$

	A	B	C	D	E	F	G	H
1	System perturbation							
2	Wilson matrix							
3								
4	A (4 x 4)				b	x	b'	x'
5	117	85	127	118	447	1	447.1	305
6	97	70	103	97	367	1	366.9	-504.4
7	74	53	71	64	262	1	262.1	133.9
8	62	45	65	59	231	1	230.9	-79.4
9								
10	det(A)			Δ b	Δ x	Δ b %	Δ x %	S
11	1			0.1	607.6539	0.015%	30383%	2052807

For a very small perturbation of about 0.01% of the constant term, the system solution values are moved far away from the point (1, 1, 1, 1)

Note the very high sensitivity

coefficient S of this problem and the wide spread of the solution point even for very small perturbations.

Note also that in both problems the determinant was unitary (Det = 1). So we cannot discover the instability simply detecting the determinant.

One popular factor for instability matrix uses its eigenvalues

$$S_{\lambda} = |\lambda|_{\max} / |\lambda|_{\min}$$

But, unfortunately, eigenvalues are not very easy to compute

So another practically index references the SVD decomposition (see SVD_D () function).

Extracting the d_{\max} and d_{\min} singular values of the diagonal matrix D we define the instability factor as:

$$S_D = d_{\max} / d_{\min}$$

For the above matrix the eigenvalues are

$\lambda =$	323.98	-5.72328	-1.256573	0.000429
-------------	--------	----------	-----------	----------

So the instability factor will be:

$$S_{\lambda} = 324.0 / 0.000429 \cong 754861$$

While the SVD decomposition gives

$$S_D = 340.9 / 0.000308 \cong 1106504$$

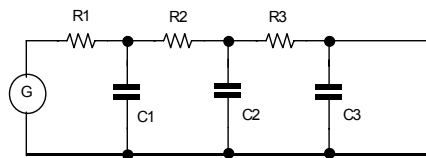
340.9215	0	0	0
0	7.879412	0	0
0	0	1.208233	0
0	0	0	0.000308

Complex systems

Complex systems are very common in applied science. Matrix.xla has just added a dedicated function SYSLIN_C() to solve them.

We shall learn how it works with a practically example from Network Theory.

Example - Analysis of the Lattice network. Find voltages and phases at the nodes of the following network, for frequency $f = 10, 50, 100, 400$ Hz.



Components values

$R1 = 100 \, \Omega$ $C1 = 1.5 \, \mu F$
 $R2 = 120 \, \Omega$ $C2 = 2.2 \, \mu F$
 $R3 = 120 \, \Omega$ $C3 = 2.2 \, \mu F$
 $G = 2.5 \sin(2\pi f t)$

As known, using the notation $v(t) = V \sin(\omega t + \theta) \Leftrightarrow \dot{V} = V e^{j\omega t} = V_{re} + j V_{im}$

The Nodal Analysis provides the solution by the following complex matrix equation

$$[Y] \dot{V} = \dot{I} \quad (1)$$

Where: $[Y] = [G] + j[B]$, $\dot{I} = I_{re} + j I_{im}$, $\dot{V} = V_{re} + j V_{im}$

The real matrix **G** and **B** are called respectively *conductance* and *susceptance*; they form the complex matrix *admittance* **Y**. These matrices depend on the $\omega = 2 \pi f$ frequency

In the worksheet the problem can be solved, first of all, calculating the frequency ω , the two real matrices **G** and **B** and the currents input vector; then, we build the complex system (1) with the SYSLIN_C function.

	A	B	C	D	E	F	G	H	I	J	K
1	Lattice Network Analysis										
2	Components			conductance matrix			susceptance matrix			Currents	
3	R1	100	ohm	0.01833	-0.0083	0	3.8E-03	0	0	0.025	0
4	R2	120	ohm	-0.0083	0.01667	-0.0083	0	5.5E-03	0	0	0
5	R3	120	ohm	0	-0.0083	0.00833	0	0	5.5E-03	0	0
6	C1	1.5E-06	F								
7	C2	2.2E-06	F								
8	C3	2.2E-06	F								
9	G	2.5	V	v1 =	1.39368	-0.658			1.54122	-25.3	
10	f	400	Hz	v2 =	0.36378	-0.8172			0.89452	-66.0	
11	ω	2513.274	rad/s	v3 =	-0.1239	-0.735			0.74537	-99.6	
12					{=SYSLIN_C(D3:I5,J3:K5)}						

SYSLIN_C provides the vector solution in complex form; to convert it in magnitude and phase we have used the following well-known formulas

$$|V| = \sqrt{(V_{re})^2 + (V_{im})^2} \quad , \quad \angle V = \text{atan}\left(\frac{V_{im}}{V_{re}}\right)$$

Note that we have to add the imaginary column at the current vector, even if it is pure real; complex matrices and complex vectors must be always definite with real and imaginary parts. They must have always an even numbers of columns.

In the above example there are many Excel formulas that we couldn't shown for clarity. To reply the example, copy the following formulas (in blue) in your worksheet.

	A	B	C	D	E	F	G	H	I	J	K
1	Lattice Network Analysis										
2	Components			conductance matrix			susceptance matrix			Currents	
3	R1	100	ohm	=1/B3+1/B4	=-1/B4	0	=B11*B6	0	0	=B9/B3	0
4	R2	120	ohm	=-1/B4	=1/B4+1/B5	=-1/B5	0	=B11*B7	0	0	0
5	R3	120	ohm	0	=-1/B5	=1/B5	0	0	=B11*B8	0	0
6	C1	1.5E-06	F								
7	C2	2.2E-06	F								
8	C3	2.2E-06	F	v1 =	Node voltages {=SYSLIN_C(D3:I5,J3:K5)}		Modulo		phase		
9	G	2.5	V	v2 =			=M_ABS(E8:F8)		=ATAN2(E8,F8)*180/PI()		
10	f	400	Hz	v3 =			=M_ABS(E9:F9)		=ATAN2(E9,F9)*180/PI()		
11		2513.3	rad/s				=M_ABS(E10:F10)		=ATAN2(E10,F10)*180/PI()		

See also Function Mat_Adm() for admittance matrix.

Example - Solve the following complex system

$$\begin{cases} (1+i\sqrt{2})x + (1-i\sqrt{2})y - z = -1+i \\ -\sqrt{2}x + y + (\sqrt{2}+i)z = \sqrt{2}-2 \\ x + y + z = \sqrt{2}-1-i \end{cases}$$

The system is equivalent to the following complex matrix equation

$$\begin{bmatrix} (1+i\sqrt{2}) & (1-i\sqrt{2}) & -1 \\ -\sqrt{2} & 1 & (\sqrt{2}+i) \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -1+i \\ \sqrt{2}-2 \\ \sqrt{2}-1-i \end{bmatrix}$$

With SYSLIN_C() function is simple to find the solution of a complex matrix system. We have only to separate the real and imaginary parts.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Complex matrix						Constant			solution		
2	1	1	-1	1.4142	-1.414	0	-1	1		1.4142	-2E-16	
3	-1.414	1	1.4142	0	0	1	-0.586	0		1E-16	-1	
4	1	1	1	0	0	0	2.4142	-1		1	-1E-16	
5												
6							{=SYSLIN_C(A2:F4,H2:I4)}					

About complex matrix format

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

1) Split format

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

2) Interlaced format

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

3) String format

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

Each format has advantages and drawbacks.

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

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

This format is useful when elements are returned by complex functions (for example by `xnumbers.xla` `addin`)

The last format is the well known "*complex rectangular format*". Each element is written as a string

$z = a+ib$ so the matrix is still squared. Apparently is the most compact and intuitive format but this is true only for integer values. For long decimal values the matrix becomes illegible. We have also to point out that the elements, being strings, cannot be format as the other Excel numbers.

Determinant

Differently from the solution of linear system, matrix determinant changes with the reduction operations of the Gauss-Jordan algorithm. In fact the final reduced matrix is the identity matrix that has always determinant = 1. But the determinant of the original given matrix can be computed with the following simple rule

- When we multiply a matrix row for a number k , thus the determinant is multiply for the same number
- When we swap two rows, thus the determinant change the sign

Gaussian elimination

With these simple rules it's easy to calculate the matrix determinant. It is sufficient to keep trace of all pivot multiplications and rows swapping performed during the Gauss-Jordan process

There also another rule, useful to reduce the computing effort.

- Triangular matrix and diagonal matrix with the same diagonal have the same determinant

So, in order to compute the determinant, we can reduce the given matrix to a triangular matrix instead of a diagonal one, saving half of computation effort. This is called the Gauss algorithm or *Gaussian elimination*.

The determinant of a diagonal matrix is the product of all elements

$$\det(A) = \det \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix} = a_{11} \cdot a_{22} \cdot a_{33}$$

And also:

$$\det(A) = \det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{21} \\ 0 & 0 & a_{33} \end{bmatrix} = a_{11} \cdot a_{22} \cdot a_{33}$$

And also:

$$\det(A) = \det \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11} \cdot a_{22} \cdot a_{33}$$

The example below shows how to compute, step by step, the determinant with the Gauss algorithm

$$A = \begin{vmatrix} 4 & 1 & 0 \\ -2 & -2 & 1 \\ 1 & -2 & 2 \end{vmatrix} \quad \text{Det}(A) = ?$$

TUTORIAL FOR MATRIX.XLA

$$A1 = \begin{vmatrix} 4 & 1 & 0 \\ 0 & -3 & 2 \\ 1 & -2 & 2 \end{vmatrix} \begin{matrix} 1 \\ 2 \\ \end{matrix}$$

$$R2 = R1 + 2 \cdot R2 \quad (*)$$

$$\text{Det}(A1) = 2 \text{Det}(A)$$

(*)
The formula
 $R2 = R1 + 2 \cdot R2$
Is a compact way for describing the following operations:

- 1) Multiply the 2nd row for 2.
- 2) Add the 2nd row and the 1st row
- 3) substitute the result to the 2nd row

$$A2 = \begin{vmatrix} 4 & 1 & 0 \\ 0 & -3 & 2 \\ 0 & 9 & -8 \end{vmatrix} \begin{matrix} 1 \\ 1 \\ -4 \end{matrix}$$

$$R3 = R1 + (-4) \cdot R2$$

$$\text{Det}(A2) = -8 \text{Det}(A)$$

$$A3 = \begin{vmatrix} 4 & 1 & 0 \\ 0 & 9 & -8 \\ 0 & -3 & 2 \end{vmatrix} \begin{matrix} 1 \\ 1 \\ 1 \end{matrix}$$

< swap
< swap

$$\text{Det}(A3) = 8 \text{Det}(A)$$

$$A4 = \begin{vmatrix} 4 & 1 & 0 \\ 0 & 9 & -8 \\ 0 & 0 & -2 \end{vmatrix} \begin{matrix} 1 \\ 1 \\ 3 \end{matrix}$$

$$R3 = R2 + 3 \cdot R3$$

$$\text{Det}(A4) = 24 \text{Det}(A)$$

$$A4 = \begin{vmatrix} 4 & 1 & 0 \\ 0 & 9 & -8 \\ 0 & 0 & -2 \end{vmatrix}$$

$$\text{Det}(A4) = 24 \text{Det}(A)$$

$$\text{Det}(A4) = 4 \cdot 9 \cdot (-2) = -72$$

The final matrix A4 is triangular. So its determinant - easy to compute - is -72

But it is also:

$$\text{Det}(A4) = 24 \text{Det}(A)$$

Substituting, we have:

$$-72 = 24 \text{Det}(A) \quad \Rightarrow \quad \text{Det}(A) = -24 / 24 = -1$$

Hill-conditioned matrix

Of course there are functions M_DET() in Matrix.xla and also MDETERM() in Excel to obtain quickly the determinant of any square matrix. Both are very fast and efficient, covering the most part of cases. But, sometime, they can fail because of the round-off error introduced by the finite numeric arithmetic of the computer. It happens for large matrices, or even for small matrices (hill- conditioned matrices). Look at this example.

Compute the determinant of this simple (3 x 3) matrix

127	-507	245
-507	2025	-987
245	-987	553

Both functions return a very small, but finite value, quite different each other.

	A	B	C	D	E
1					
2		127	-507	245	
3		-507	2025	-987	
4		245	-987	553	
5					
6		1.504E-09	=M_DET(B2:D4)		
7		-6.868E-10	=MDETERM(B2:D4)		
8					

If you repeat the calculus with other numerical routine in 32 bit OS you will get similar results. Is there any reason for suspecting this result? Yes, there is, because this result is completely wrong!

In fact, the determinant is 0; the given matrix is singular. You can easily compute by hand with exact fractional numbers. If you are lazy use the Gauss_Jordan_step function with integer algorithm

$$\begin{vmatrix} 127 & -507 & 245 \\ -507 & 2025 & -987 \\ 245 & -987 & 553 \end{vmatrix} \begin{matrix} < \text{swap} \\ < \text{swap} \\ \end{matrix}$$

$$\text{Det}(A1) = -1 \text{ Det}(A)$$

$$\begin{vmatrix} -507 & 2025 & -987 \\ 127 & -507 & 245 \\ 245 & -987 & 553 \end{vmatrix} \begin{matrix} -127 \\ -507 \\ \end{matrix}$$

$$\text{Det}(A2) = 507 \text{ Det}(A)$$

$$\begin{vmatrix} -507 & 2025 & -987 \\ 0 & -126 & 1134 \\ 245 & -987 & 553 \end{vmatrix} \begin{matrix} -245 \\ -507 \\ \end{matrix}$$

$$\text{Det}(A3) = -257049 \text{ Det}(A)$$

$$\begin{vmatrix} -507 & 2025 & -987 \\ 0 & -126 & 1134 \\ 0 & 4284 & -38556 \end{vmatrix} \begin{matrix} < \text{swap} \\ < \text{swap} \\ \end{matrix}$$

$$\text{Det}(A4) = 257049 \text{ Det}(A)$$

$$\begin{vmatrix} -507 & 2025 & -987 \\ 0 & 4284 & -38556 \\ 0 & -126 & 1134 \end{vmatrix} \begin{matrix} -476 \\ 225 \\ \end{matrix}$$

$$\text{Det}(A5) = -122355324 \text{ Det}(A)$$

$$\begin{vmatrix} 241332 & 0 & -8205288 \\ 0 & 4284 & -38556 \\ 0 & -126 & 1134 \end{vmatrix} \begin{matrix} 1 \\ 34 \\ \end{matrix}$$

$$\text{Det}(A6) = -4160081016 \text{ Det}(A)$$

$$\begin{vmatrix} 241332 & 0 & -8205288 \\ 0 & 4284 & -38556 \\ 0 & 0 & 0 \end{vmatrix}$$

The last row is all zero. This means that the matrix is singular and its determinant is zero.

$$\text{Det}(A6) = 0 \implies \text{Det}(A) = 0$$

In this case it was easy to analyze the matrix, but for a larger matrix (50 x 50) do you know what would happen? Before to accept any results - specially for large matrices -we have to do same extra tests, like for example the SVD decomposition.

Laplace's expansion

Expansion by minors is another technique for computing the determinant of a given square matrix. Although efficient for small matrices (practically for $n = 2, 3$), techniques such as Gaussian elimination are much more efficient when the matrix size becomes large. Laplacian expansion becomes competitive when there are rows or columns with many zeros.

The expansion formula is applied to any line (row or column) of the matrix. The choice is arbitrary. For example, the expansion along the first row of a 3x3 matrix becomes.

$$|A| = \sum_{j=1}^n (-1)^{(1+j)} |A_{1j}| a_{1j} = |A_{11}| a_{11} - |A_{12}| a_{12} + |A_{13}| a_{13}$$

Where $|A_{ij}|$ are the *minors*, that is the determinant of the sub matrix extracted from the original matrix eliminating the row i and the column j . The minors are taken with sign $+$ if the sum of $(i+j)$ is even; on the contrary if odd.

Many authors call *cofactor* the term: $(-1)^{(i+j)}|A_{ij}|$.

Let's see how works with an example

Example - Calculate the determinant of the given 3x3 matrix with the Laplace's expansion.

8	-4	-2
1	-1	2
2	3	-3

We use the function **MatExtract()** to get the 2x2 minor sub matrix; we use also the INDEX function to get the a_{ij} element

F6		= {=MatExtract(A2:C4,F2,G2)}													
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	
1	A					i	j								
2	8	-4	-2		index	1	1								
3	1	-1	2												
4	2	3	-3		pivot		8								
5															
6					minor	-1	2								
7						3	-3								

=INDEX(\$A\$2:\$C\$4,F2,G2)

{=MatExtract(A2:C4,F2,G2)}

Completing the worksheet with the others minor and the cofactor terms we have

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1		A				i	j	i	j	i	j						
2	8	-4	-2		index	1	1	1	2	1	3						
3	1	-1	2														
4	2	3	-3		pivot		8		-4		-2						
5																	
6					minor	-1	2	1	2	1	-1						
7						3	-3	2	-3	2	3						
8																	
9	A =	-62			cofactor	-24		-28		-10							
10																	

Tip. We can use the arbitrary of the row (or column) expansion in order to minimize the computing. Usually we choose the row or column with the most zeros (if any).

Simultaneous Linear Systems

The function SYSLIN can give solutions of many linear systems having the same incomplete coefficients matrix and different constant vectors.

Example: solve the following matrix equation

$$\mathbf{A} \mathbf{X} = \mathbf{B} \quad (1)$$

Where:

$$\mathbf{A} = \begin{vmatrix} 1 & 3 & -4 & 9 \\ 2 & 3 & 5 & 1 \\ 2 & -1 & 4 & 10 \\ 0 & -1 & 1 & 0 \end{vmatrix} \quad \mathbf{B} = \begin{vmatrix} 59 & -19 \\ 3 & 20 \\ 58 & 24 \\ -1 & 6 \end{vmatrix}$$

The solution is

$$\mathbf{X} = \mathbf{A}^{-1} \mathbf{B} \quad (2)$$

You can get the numerical solution in two different ways. The first one is the direct application of the formula (2); the second one is the resolution of the simultaneous linear system (1)

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	A (4 x 4)					B (4 x 2)			Solution X			Solution X	
2	1	3	-4	9		59	-19		1	3		1	3
3	2	3	5	1		3	20		2E-14	-2		-2E-14	-2
4	2	-1	4	10		58	24		-1	4		-1	4
5	0	-1	1	0		-1	6		6	0		6	-2E-14
6													
7													
8													
9													

{MMULT(MINVERSE(A2:D5),F2:G5)}

{=SYSLIN(A2:D5,F2:G5)}

From the point of view of the accuracy both methods are substantially the same; for the efficiency, the second one is better, especially for larger matrices

Inverse matrix

Computing of the inverse matrix is an especially application of the simultaneous systems resolution.

In fact, if **B** is the identity matrix, we have:

$$\mathbf{A} \mathbf{X} = \mathbf{I} \quad \Rightarrow \quad \mathbf{X} = \mathbf{A}^{-1} \mathbf{I} = \mathbf{A}^{-1}$$

You have the **MINVERSE()** Excel or the **M_INV()** Matrix.xla functions to invert a given square matrix.

Example: find the inverse of the 4 x 4 Hilbert matrix
Hilbert matrices are a note class of hill-conditioned matrices, very easy to generate:

$$a(i, j) = 1/(i+j-1)$$

1	1/2	1/3	1/4
1/2	1/3	1/4	1/5
1/3	1/4	1/5	1/6
1/4	1/5	1/6	1/7

Inverse of Hilbert matrices are always integer. So, if same decimals appear in the result, we can be sure that they are due to errors round off and we can valuate consequently the accuracy of the result. You can easily generate these matrices by hand or also by the function Mat_Hilbert()

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
4														
5	1	1/2	1/3	1/4		16	-120	240	-140		7E-13	-8E-12	2E-11	-1E-11
6	1/2	1/3	1/4	1/5		-120	1200	-2700	1680		-8E-12	1E-10	-2E-10	2E-10
7	1/3	1/4	1/5	1/6		240	-2700	6480	-4200		2E-11	-2E-10	6E-10	-4E-10
8	1/4	1/5	1/6	1/7		-140	1680	-4200	2800		-1E-11	2E-10	-4E-10	3E-10
9														
10	{=Mat_Hilbert() }				{=MINVERSE(A5:D8) }				=ROUND(F8,0)-F5					
11														

Round-off error

As you can see, Excel hides the round-off error and the result seems to be exact. But there is not the true. In order to show the error without format the cells with 10 or more decimal we can use this simple trick. Extract only the round-off error from each a_{ij} value by the following formula:

$$E = \text{ROUND}(a_{ij}, 0) - a_{ij}$$

Applying this method to the above inverse matrix, we see that there are absolute round-off errors from 1E-13 till 1E-10.

There is another method to estimate the accuracy of the inverse matrix: multiplying the given matrix by its approximate inverse we get a "near" identity matrix. The values out of the first diagonal measure the errors. If we compute the mean of the absolute values we have an estimation of the round-off error. The M_DIAG_ERR() automates this task.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Matrix A							Matrix A ⁻¹					
2	1	1/2	1/3	1/4	1/5	1/6		36	-630	3360	-7560	7560	-2772
3	1/2	1/3	1/4	1/5	1/6	1/7		-630	14700	-88200	211680	-2E+05	83160
4	1/3	1/4	1/5	1/6	1/7	1/8		3360	-88200	564480	-1E+06	2E+06	-6E+05
5	1/4	1/5	1/6	1/7	1/8	1/9		-7560	211680	-1E+06	4E+06	-4E+06	2E+06
6	1/5	1/6	1/7	1/8	1/9	1/10		7560	-2E+05	2E+06	-4E+06	4E+06	-2E+06
7	1/6	1/7	1/8	1/9	1/10	1/11		-2772	83160	-6E+05	2E+06	-2E+06	698544
8													
9	Matrix A A ⁻¹												
10	1	4E-12	1E-11	1E-10	0	3E-11		{=M_INV(A2:F7) }					
11	-6E-14	1	-1E-11	3E-11	-1E-10	-1E-11		{=M_PROD(A2:F7,H2:M7) }					
12	-6E-14	4E-12	1	1E-10	6E-11	1E-11		{=M_DIAG_ERR(A10:F15) }					
13	1E-13	-2E-12	7E-12	1	-6E-11	0		Diagonalization accuracy =					
14	0	-2E-12	2E-11	3E-11	1	-1E-11		2E-11					
15	-3E-14	2E-12	7E-12	0	0	1							
16													

The diagonalization accuracy measure the global error due to the following three step:

$$\text{Global error} = \text{Input matrix error} + \text{Inversion} + \text{multiplication}$$

The first step needs an explanation. Excel can show fractional number as exacts like for example 1/3 or 1/7. But really, these numbers are always affected by the truncation error of 1E-15

Other class of matrices - Tartaglia's - can eliminate the input truncation error because both matrix and its inverse are always integers.

Tartaglia's matrices

This class of matrices is very useful because they can be easy to generate but - this is very important - both Tartaglia's matrix and its inverse have always integer values, very useful for testing the round-off error.

They are defined with:

$$a_{1j} = 1 \quad \text{for } j = 1 \dots n \quad (\text{all 1 in the first row})$$

$$a_{i1} = 1 \quad \text{for } i = 1 \dots n \quad (\text{all 1 in the first column})$$

$$a_{ij} = \sum_j a_{(i-1)j} \quad \text{for } j = 2 \dots n$$

Here is a 6x6 Tartaglia's matrix

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

and its inverse

6	-15	20	-15	6	-1
-15	55	-85	69	-29	5
20	-85	146	-127	56	-10
-15	69	-127	117	-54	10
6	-29	56	-54	26	-5
-1	5	-10	10	-5	1

As we can see both matrices are integer. Any round-off error of the inverse matrix must be regard as a round-off error and immediately detected.

In the example below we evaluate the global accuracy of the inverse of 6 x 6 Tartaglia's matrix

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Tartaglia's matrix							inverse Tartaglia's matrix					
2	1	1	1	1	1	1		6	-15	20	-15	6	-1
3	1	2	3	4	5	6		-15	55	-85	69	-29	5
4	1	3	6	10	15	21		20	-85	146	-127	56	-10
5	1	4	10	20	35	56		-15	69	-127	117	-54	10
6	1	5	15	35	70	126		6	-29	56	-54	26	-5
7	1	6	21	56	126	252		-1	5	-10	10	-5	1
8													
9	Accuracy												
10	1.54639E-13	=M_DIAG_ERR(M_PROD(A2:F7,H2:M7))							{=M_INV(A2:F7)}				
11													

Sometimes however, Excel produces errors. Excel rounds numbers and will occasionally compute A^{-1} even if a matrix has a determinant equal to zero. Also, if a determinant is close to zero, Excel may internally round numbers so that the inverse is incorrect. If this happens, your solution will be wrong.

Let's see this example:

Example: find the inverse of the following matrix

127	-507	245
-507	2025	-987
245	-987	553

As we have seen in the previous example, the given matrix is singular. So, its inverse doesn't exist. However, if we try to compute the inverse we have the following result

	A	B	C	D	E	F	G	H
1			A				A ⁻¹	
2		127	-507	245		-2.1E+14	-5.6E+13	-6.2E+12
3		-507	2025	-987		-5.6E+13	-1.5E+13	-1.7E+12
4		245	-987	553		-6.2E+12	-1.7E+12	-1.8E+11
5								
6		-6.9E-10	=MDETERM(B2:D4)					
7			{=MINVERSE(B2:D4)}					
8								
9								

You should always examine the determinant. If the determinant is close to zero, you should try to verify the solution with other methods. For instance, you can always try to solve the inverse with integer option of M_INV function, or by Gauss_Jordan_Step function, or with SVD decomposition (see after)

How to avoid decimal number

Not always the inverse matrix is integer; many times it has many decimal numbers. If the given matrix is integer, we can obtain the fractional expression of its inverse with this little trick

Example

	A	B	C	D	E	F	G	H	I	J	K
1		A				A ⁻¹			B= det * A ⁻¹		
2	2	5	-1		-0.0667	-0.1167	-0.2833		4	7	17
3	0	2	3		0.2	0.1	0.1		-12	-6	-6
4	-4	-2	-1		-0.1333	0.2667	-0.0667		8	-16	4
5											
6		-60	=MDETERM(A2:C4)			{=MINVERSE(B2:D4)}			{=A6*E2:G4}		
7											
8											

Note the compact format of the matrix multiplication by scalar {A6*E2:G4} of last matrix

Multiplying the inverse for the determinant we get the matrix B of integer values. Thus, the inverse can be put in the following fractional form

$$A^{-1} = \frac{1}{\det(A)} B = -\frac{1}{60} \begin{bmatrix} 4 & 7 & 17 \\ -12 & -6 & -6 \\ 8 & -16 & 4 \end{bmatrix}$$

Homogeneous and Singular Linear Systems

Given a linear system with $\mathbf{b} = 0$

$$\mathbf{A} \mathbf{x} = \mathbf{0}$$

And \mathbf{A} ($n \times m$) matrix, we say a homogeneous linear system; this class of system always have the trivial solution $\mathbf{x} = 0$. But we are interested to know if the system has also other solutions.

Assume \mathbf{A} is a square matrix

$$\begin{cases} x + 2y - z = 0 \\ -x + 4y + 5z = 0 \\ -2x - 4y + 2z = 0 \end{cases}$$

1	2	-1
-1	4	5
-2	-4	2

We note that the last row can be obtained multiply the first one by -2. So, having two rows multiply each other, the given matrix has determinant = 0; that is singular. One of the two rows can be eliminate; we choose to eliminate the last row and the system becomes

$$\begin{cases} x + 2y - z = 0 \\ -x + 4y + 5z = 0 \end{cases}$$

One of the three variables can be freely chosen and it can be regard as a new independent variable. Set, for example, z as independent parameter; the other variables x, y can be express as function of independent parameter " z "

$$\begin{cases} x + 2y = z \\ -x + 4y = -5z \end{cases} \quad \begin{cases} x = \frac{7}{3}z \\ y = -\frac{2}{3}z \end{cases} \quad (1)$$

The linear equation's system (1) expresses all the infinite solutions of the given system. Geometrically speaking it is a line in the space \mathbf{R}^3

It can be also regard as a linear transformation that move a generic point $P(x, y, z)$ of the space into another point $P'(x, y, z)$ of the subspace. In this case the subspace is a line and the dimension of the subspace is \mathbf{R}^1

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}' = \begin{bmatrix} 0 & 0 & \frac{7}{3} \\ 0 & 0 & -\frac{2}{3} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{cases} 2y - z = -x \\ 4y + 5z = x \end{cases} \quad \begin{cases} y = -\frac{2}{7}x \\ z = \frac{3}{7}x \end{cases}$$

If we set on the contrary, x as independent parameter, the other variables y, z can be express as function of independent parameter " x "

That is represented by the linear transformation at the right

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}' = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{2}{7} & 0 & 0 \\ \frac{3}{7} & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The Matrix transformation is useful to find the parametric form of the linear function (mapping function)

Parametric form

The linear transformations of the above example give relations between points of the space. A common form for handling this relation is the parametric form. It easy to pass from the transformation matrix to its parametric form

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 & 0 & \frac{7}{3} \\ 0 & 0 & -\frac{2}{3} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Having the transformation matrix, we search for the variable that has 1 in the diagonal element, "z" in this case. Setting $z = t$, and performing the multiplication, we have the parametric function

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -\frac{7}{3}t \\ -\frac{2}{3}t \\ t \end{bmatrix}$$

Geometrically specking the parametric function at the right is a line with direction vector: \vec{D} given by

$$\vec{D} = \begin{bmatrix} -\frac{7}{3} \\ -\frac{2}{3} \\ 1 \end{bmatrix} \cdot \frac{1}{\sqrt{(\frac{7}{3})^2 + (\frac{2}{3})^2 + 1}} = \begin{bmatrix} -7 \\ -2 \\ 3 \end{bmatrix} \cdot \frac{1}{\sqrt{62}} \cong \begin{bmatrix} 0.889 \\ -0.256 \\ 0.381 \end{bmatrix}$$

Note that $\sqrt{62}$ is the norm of the first vector

You can study this entire problem by the **SYSLINSING()** matrix.xla function. Here the example:

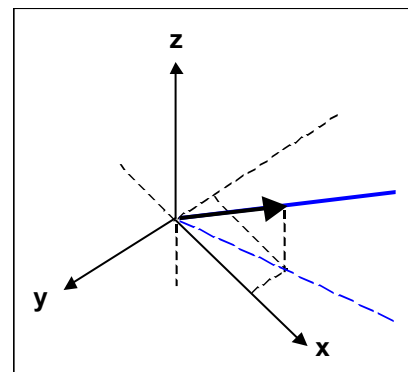
	A	B	C	D	E	F	G	H	I
15		A				B			Direction
16	1	2	-1		0	0	2.3333		0.889
17	-1	4	5		0	0	-0.667		-0.254
18	-2	-4	2		0	0	1		0.381
19									
20	0	=MDETERM(A16:C18)			{=SYSLINSING(A16:C18)}				
21									
22									
23									

Note: In the original image, the formulas are shown as array functions: {=SYSLINSING(A16:C18)} and {=G16:G18/M_ABS(G16:G18)}.

SYSLINSING solve linear singular system, returning the transformation matrix, if exists, of the solution just explained in the previous paragraph. The determinant is calculated only to show that the given matrix is singular. It is not used into calculation. SYSLINSING detects automatically if a matrix is singular or not. If the matrix is not singular (Det \neq 0) the function returns all zeros.

From the transformation matrix we have extract the direction vector by normalization of the third column of matrix B; to get the Norma of the vector we have used the function M_ABS() of matrix.xla. Note that both expression must be insert as array functions { }

In a RCO xyz, the function represents a line passing for the origin, having direction the vector D, as drawn in the figure.



Rank and Subspace

In the above example we have seen that if the matrix of a homogeneous system is singular, then exist infinite solutions of the system; that solution, in a RCO represent a subspace. After that we have find the solution, we have seen that the subspace was a line and its dimension was 1.

Well, is there a way to know the dimension of the subspace without resolving all the system? The answer is yes, but it is easy for low matrix dimension, very difficult for higher matrix dimensions.

- Rank of a square matrix is the max number of independent rows (or columns) that we find in the matrix.

For a 3 x 3 matrix the possible cases are reassume in the following table

Independent rows	Rank	Linear System Solution	Subspace
3	3	0	Null
2	2	∞^1	Line
1	1	∞^2	Plane

The function **M_RANK()** of matrix.xla calculates the rank of a given matrix. In the following example we calculate the determinant and rank for three different matrices

	A	B	C	D	E	F	G	H	I	J	K	L
1		A				B				C		
2	2	2	-1		1	2	-1		1	2	-1	
3	-1	4	5		-1	4	5		-6	-12	6	
4	-2	-4	2		-2	-4	2		-2	-4	2	
5												
6	28	=MDETERM(A2:C4)			0	=MDETERM(E2:G4)			0	=MDETERM(I2:K4)		
7	3	=M_RANK(A2:C4)			2	=M_RANK(E2:G4)			1	=M_RANK(I2:K4)		
8												

Note that the determinant is always 0 when the rank is less then the max dimension

Solving homogeneous systems with the given matrices, we will generate in a 3D space respectively the following subspace: a null space, a line, and a plane.

Let's test the last matrix

	A	B	C	D	E	F	G	H	I	J
15		A				B			Direction	
16	1	2	-1		0	-2	1		-0.894	0.7071
17	-6	-12	6		0	1	-0		0.4472	0
18	-2	-4	2		0	-0	1		0	0.7071
19										
20	0	=MDETERM(A16:C18)				{=F16:F18/M_ABS(F16:F18)}				
21	1	=M_RANK(A16:C18)				{=G16:G18/M_ABS(G16:G18)}				
22		{=SYSLINSING(A16:C18)}								
23										

Consequently, the transform matrix has two columns indicating that the subspace has 2 dimensions, thus a plane.

In order to get the parametric form of the plane we observe the transform matrix: variable y and z have both the diagonal element 1 ($a_{22} = 1$, $a_{33} = 1$). They can be assumed as independent parameters.

Let $y = t$ and $z = s$, we have

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 & -2 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -2t + s \\ t \\ s \end{bmatrix}$$

Eliminating both parameters we get the normal plane equation of the $x = -2y + z \Rightarrow x + 2y - z = 0$ (2)

The linear equation (2) expresses all the infinite solutions of the given system. Geometrically speaking it is a plane in the space \mathbf{R}^3

Rank for rectangular matrix

Differently from the determinant, the rank can be computed also for rectangular matrix. Its definition is:

- *Rank of a rectangular matrix is the min between the max number of independent rows and the max number of independent columns*

Example: find the rank of the following 3 x 5 matrix

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

	A	B	C	D	E	F
9						
10	1	2	9	10	-7	
11	1	2	-1	0	3	
12	2	4	-5	-3	9	
13						
14	2	=M_RANK(A10:E12)				
15						

By inspection we see that there are 3 independent rows and only 2 independent columns. In fact column c2 is obtained multiplying the first for 2; the column c4 = c1 + c3; column c5 = c2 - c3

So the rank is given by: $\text{rank} = \min(2, 3) = 2$

One popular theorem -due to Kronecker - says that if the rank = r, then all the square sub-matrices (p x p) extracted from the given matrix, having $p > r$, are all singular

In other way all matrices 3 x 3 extracted from the matrix of the above example have determinant = 0. You can enjoy finding yourself all the 10 matrices of 3 dimensions. Here 5 of them.

1	2	9
1	2	-1
2	4	-5

1	9	10
1	-1	0
2	-5	-3

2	9	-7
2	-1	3
4	-5	9

1	2	10
1	2	0
2	4	-3

1	9	-7
1	-1	3
2	-5	9

General Case - Rouché-Capelli Theorem

Given a linear system of m equations and n unknowns

$$(1) \quad \begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n} = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n} = b_2 \\ a_{31}x_1 + a_{32}x_2 + \dots + a_{3n} = b_3 \\ \dots\dots\dots \\ \dots\dots\dots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn} = b_{m1} \end{cases}$$

$$A(m \times n) = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

The matrix A is called *coefficients' matrix* or *incomplete matrix*

$$B(m, n+1) = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{bmatrix}$$

The matrix B is called *complete matrix*

If the column b is zero, the system is called *homogeneous*

In order to know if the system (1) has solutions is valid the following fundamental theorem

ROUCHÉ-CAPELLI THEOREM.

A linear system has solutions if, and only if, the ranks of matrices A and B are equals

That is: $\text{rank}(\mathbf{A}) = \text{rank}(\mathbf{B}) \Leftrightarrow \exists \mathbf{x}$ solution

Note: This rule is always valid for homogeneous systems that are always the $\mathbf{x} = 0$ solution (trivial solution)

Among ranks of the matrices, number of equations and number of unknowns exist important relations. The following table reassumes 12 possible cases: 6 for homogeneous system and 6 for full system. This table, very clear and well organized, is due to Marcello Pedone.

Homogeneous System Cases

Case	Rank of incomplete matrix A	Non homogeneous system solution	Example
1	$\text{rank}(A) = m = n$	Trivial solution (0,0,...0)	$\begin{cases} 2x + 3y = 0 \\ x - 3y = 0 \end{cases} S(0,0)$
2	$\text{rank}(A) = m < n$	∞^{n-m} solutions + trivial solution	$\begin{cases} 2x + 3y + z = 0 \\ x - 3y + 2z = 0 \end{cases} S\left(-z, \frac{1}{6}z, z\right) (\infty^1 \text{ soluzioni})$ $+ S(0,0,0)$
3	$\text{rank}(A) < m < n$	∞^{n-r} solutions + trivial solution	$\begin{cases} 2x + 3y + z = 0 \\ 4x + 6y + 2z = 0 \end{cases} S\left(\frac{-3y-z}{2}, y, z\right) (\infty^2 \text{ soluzioni})$ $+ S(0,0,0)$
4	$\text{rank}(A) < m = n$	∞^{n-r} solutions + trivial solution	$\begin{cases} x - 2y = 0 \\ 2x - 4y = 0 \end{cases} S(2y, y) (\infty^1 \text{ soluzioni})$ $+ S(0,0)$
5	$\text{rank}(A) = n < m$	Trivial solution (0,0,...0)	$\begin{cases} x + 2y = 0 \\ 3x - 2y = 0 \\ x - y = 0 \end{cases} S(0,0)$
6	$\text{rank}(A) < n < m$	∞^{n-r} solutions + trivial solution	$\begin{cases} x + 2y = 0 \\ 2x + 4y = 0 \\ 3x + 6y = 0 \end{cases} S(-2y, y) (\infty^1 \text{ soluzioni})$ $+ S(0,0)$

Non Homogeneous System Cases

Case	Rank of incomplete matrix A	Non homogeneous system solution	Example
1	$\text{rank}(A) = m = n$	One solution	$\begin{cases} 2x + 3y = 2 \\ x - 3y = 1 \end{cases} \quad S(1,0)$
2	$\text{rank}(A) = m < n$	∞^{n-m} solutions	$\begin{cases} 2x + 3y + z = 1 \\ x - 3y + 2z = 2 \end{cases} \quad S\left(1 - z, \frac{z-1}{6}, z\right) (\infty^1 \text{ soluzioni})$
3	$\text{rank}(A) < m < n$	∞^{n-r} solutions If $r(B) = r(A)$	$1) \begin{cases} 2x + 3y + z = 1 \\ 4x + 6y + 2z = 2 \end{cases} \quad S\left(\frac{1-3y-z}{2}, y, z\right) (\infty^2 \text{ soluzioni})$ $2) \begin{cases} 2x + 3y + z = 1 \\ 4x + 6y + 2z = 0 \end{cases} \quad \text{incompatibile } r(A) \neq r(B)$
4	$\text{rank}(A) < m = n$	∞^{n-r} solutions se $r(B) = r(A)$	$1) \begin{cases} x - 2y = 1 \\ 2x - 4y = 2 \end{cases} \quad S(1 - 2y, y) (\infty^1 \text{ soluzioni})$ $2) \begin{cases} x - 2y = 1 \\ 2x - 4y = 1 \end{cases} \quad \text{incompatibile } r(A) \neq r(B)$
5	$\text{rank}(A) = n < m$	One solution If $r(B) = r(A)$	$1) \begin{cases} x + 2y = 1 \\ 3x - 2y = 0 \\ 2x + 4y = 2 \end{cases} \quad S\left(\frac{1}{4}, \frac{3}{8}\right)$ $2) \begin{cases} x + 2y = 1 \\ 3x - 2y = 0 \\ x - y = 1 \end{cases} \quad \text{incompatibile } r(A) \neq r(B)$
6	$\text{rank}(A) < n < m$	∞^{n-r} solutions If $r(B) = r(A)$	$1) \begin{cases} x + 2y = 1 \\ 2x + 4y = 2 \\ 3x + 6y = 3 \end{cases} \quad S(1 - 2y, y) (\infty^1 \text{ soluzioni})$ $2) \begin{cases} x + 2y = 1 \\ 2x + 4y = 0 \\ 3x + 6y = 3 \end{cases} \quad \text{incompatibile } r(A) \neq r(B)$

Triangular Linear Systems

Solving triangular linear system is simple and exist very efficient algorithms for this task. So, many methods try to decompose the full system into one or two triangular systems by factorization algorithms.

Triangular factorization

Having the linear system

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (1)$$

Suppose you have got the following factorization

$$\mathbf{A} = \mathbf{L} \mathbf{U} \quad (2)$$

Where \mathbf{L} is lower-triangular lower and \mathbf{U} upper-triangular. That is:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix}$$

In that case, we can divide the linear system (1) into two systems:

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

Setting: $\mathbf{y} = \mathbf{U} \mathbf{x}$ we can write:

$$\mathbf{L} \mathbf{y} = \mathbf{b} \quad (3) \qquad \mathbf{U} \mathbf{x} = \mathbf{y} \quad (4)$$

Forward and Back substitutions

The method proceeds in two steps: first we solve the lower-triangular system (3) with forward-substitution algorithm; then, with the vector \mathbf{y} used as constant terms, we solve the upper-triangular system (4) with the back-substitutions algorithm. Both algorithms are very fast.

Let's see how works

Having the following factorization $\mathbf{L} \mathbf{U} = \mathbf{A}$, solve the linear system $\mathbf{A} \mathbf{x} = \mathbf{b}$

\mathbf{A}			\mathbf{b}		\mathbf{L}			\mathbf{R}		
6	5	1	19		1	0	0	6	5	1
12	8	6	46		2	2	0	0	-1	2
-6	-6	5	-3		-1	1	1	0	0	4

In Matrix.xla we can use the function **SYSLIN_T()** that applies the efficient back/forward algorithm to solve triangular systems.

This function has an optional parameter to switch the algorithm for upper (Typ = "U") or lower (Typ = "L") triangular matrix. If missed, the function tries to detect by itself the matrix type

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

The original system is broken into two triangular systems

$$A x = b$$

$$L y = b$$

$$U x = y$$

We can prove that the vector $x = (1, 2, 3)$ is the solution of the original system $A x = b$

LU factorization

This method, based on the Crout's factorization algorithm, splits a square matrix into two triangular matrices factors. This method is useful and very popular to solve linear system and also to invert matrices. In Matrix.xla this algorithm is performed by the **Mat_LU()** function. This function returns both factors in an array (n x 2n).

But there are some things that it is better to point out. Many authors emphasizes the fact that the **LU** Crout's factorization is independent from the constant vector **b** of a system, getting to understand that once we have the **LU** decomposition of **A** we can solve with as many linear system we want, simple change the vector b. It is not completely true and may induce in wrong results.

Look at this example. Find the solution of the following linear system with the LU factorization.

$$A x = b$$

where:

	A	b
0	5	4
2	4	2
-8	0	-9

If we compute the LU factorization we have:

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

Note that you must select (3x6) cells if you want to get the factorization of a (3x3) matrix

The Crout's algorithm has returned the following triangular

L			U		
1	0	0	-8	0	-9
-0	1	0	0	5	4
-0.25	0.8	1	0	0	-3.45

Now solve the system (3) and (4) in order to have the final solution

$$L y = b \quad (3)$$

$$U x = y \quad (4)$$

We have

b	y = L⁻¹ b	x = U⁻¹ y
22	22	-16.54348
16	16	-6.608696
-35	-42.3	12.26087

The exact solution of the original system (1) is $\mathbf{x} = (1, 2, 3)$, but the LU method has given a wrong result. Why? What's happened?

The fact is - and too many authors omit this, this algorithm do not give the exact original matrix \mathbf{A} but a new matrix \mathbf{A}' being a rows permutation of the given one. This is due to the partial pivoting strategy of Crout's algorithm. You simple prove it by multiplying \mathbf{L} and \mathbf{U} .

So the right factorization formula would be:

$$\mathbf{A} = \mathbf{PLU}$$

Where \mathbf{P} is a permutation matrix

The process to solve the system is therefore:

$$\mathbf{b}' = \mathbf{P}^T \mathbf{b} \quad (5) \quad \mathbf{L} \mathbf{y} = \mathbf{b}' \quad (6) \quad \mathbf{U} \mathbf{x} = \mathbf{y} \quad (7)$$

We have shown that the information of only two factors \mathbf{L} and \mathbf{U} are no sufficient to solve any system. We must complete it with the information of \mathbf{P} matrix.

But how can we get the permutation matrix? This matrix must be providing by the algorithm itself at the end of the elaboration process. Usually the most part of routines for LU system solving do not give us the permutation matrix because the formula (5) is applied directly to the vector \mathbf{b} passed to the routines. But the concept is substantially the same: for solving a system with LU factorization we need, in generally, three matrices $\mathbf{P} \mathbf{L} \mathbf{U}$.

	A	B	C	D	E	F	G	H	I	J
1										
2	0	1	0	1	0	0	-8	0	-9	
3	0	0	1	-0	1	0	0	5	4	
4	1	0	0	-0.25	0.8	1	0	0	-3.45	
5										
6										
7	b		b'		y		x			
8	22		-35		-35		1			
9	16		22		22		2			
10	-35		16		-10.4		3			
11										
12										
13										
14										

The original system is broken into two triangular systems

$$\begin{aligned} \mathbf{A} \mathbf{x} &= \mathbf{b} \\ \mathbf{b}' &= \mathbf{P}^T \mathbf{b} \\ \mathbf{L} \mathbf{y} &= \mathbf{b}' \\ \mathbf{U} \mathbf{x} &= \mathbf{y} \end{aligned}$$

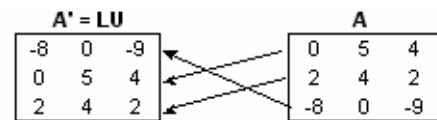
The permutation matrix can be get comparing the original \mathbf{A} matrix and the one obtained from the product $\mathbf{A}' = \mathbf{LU}$. Let's see how.

The base vectors \mathbf{u}_1 , \mathbf{u}_3 , \mathbf{u}_3 are:

$$\mathbf{u}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{u}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{u}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

We examine now the matrix rows of the two matrices \mathbf{A}' and \mathbf{A} .

The row 1 of A' comes from row 3 of A , $\Rightarrow p_1 = u_3$
 The row 2 of A' comes from row 1 of A , $\Rightarrow p_2 = u_1$
 The row 3 of A' comes from row 2 of A , $\Rightarrow p_3 = u_2$



So the permutation matrix will be:

$$P = (p_1, p_2, p_3) = (u_3, u_1, u_2) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Clearly this process can be very tedious just for a bit larger matrices. Fortunately the permutation matrix is supply by Mat_LU function as third side of its output. For a 3x3 matrix you have to select a range of 9 columns to see the permutation matrix.

Mat_LU(**A**) returns (**L** , **U** , **P**) array

That gives the decomposition **A = P L U** .

Example - Perform the exact LU decomposition for a 5x5 Tartaglia's matrix

G2		= {=Mat_LU(A2:E6)}																			
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1			A						L					U					P		
2	1	1	1	1	1		1	0	0	0	0	1	5	15	35	70	0	1	0	0	0
3	1	2	3	4	5		1	1	0	0	0	0	-4	-14	-34	-69	0	0	0	1	0
4	1	3	6	10	15		1	0.5	1	0	0	0	0	-2	-8	-21	0	0	1	0	0
5	1	4	10	20	35		1	0.8	0.8	1	0	0	0	0	0.5	2.1	0	0	0	0	1
6	1	5	15	35	70		1	0.3	0.8	-1	1	0	0	0	0	-1	1	0	0	0	0
7																					
8	{=Mat_LU(A2:E6)}																				
9																					

If we perform the matrix product **P L U** (it is useful the **M_PROD()** function) we obtain finally the given original matrix. (Note that the **P** must be the first of product)

Block-Triangular Form

Square sparse matrices, thus with several zero elements, can be, under certain conditions, put in the useful form called “block-triangular” (also called “Jordan’s form”) by simple permutations of rows and columns

1	2	1	0	0	0
2	1	5	0	0	0
1	-1	3	0	0	0
-6	5	3	1	1	2
1	-3	2	1	-1	-2
-9	7	1	1	2	1

A_1	0
A_{21}	A_2

The block-triangular form allows a good saving of computation effort for many important problems of linear algebra: linear system solving, determinant computing, eigenvalues problems, etc.

We have to point out that each of these tasks has a computing cost that grows approximately with N^3 . Thus, reducing for example the dimension to $N/2$, the effort will decrease 8 times. Clearly it's a great advantage.

Linear system solving

For example, the following (6 x 6) linear system

$$A x = b$$

1	2	1	0	0	0	x_1	b_1
2	1	5	0	0	0	x_2	b_2
1	-1	3	0	0	0	x_3	b_3
-6	5	3	1	1	2	x_4	b_4
1	-3	2	1	-1	-2	x_5	b_5
-9	7	1	1	2	1	x_6	b_6

It could be written as

$$A_1 x_1 = b_1$$

$$A_2 x_2 = b_2 - c_2$$

where the vector c_2 is given by: $c_2 = A_{21} x_1$

Practically, the original system (6 x 6) is split into two (3 x 3) sub-systems

1	2	1	x_1	b_1
2	1	5	x_2	b_2
1	-1	3	x_3	b_3

1	1	2	x_4	b_4	-	-6	5	3	x_1
1	-1	-2	x_5	b_5	-	1	-3	2	x_2
1	2	1	x_6	b_6	-	-9	7	1	x_3

Determinant computing

Determinant computing also takes advantage from the block-triangular form

For example, the determinant of the following (6 x 6) is given by the determinants product of the two matrices (3 x 3) \mathbf{A}_1 and \mathbf{A}_2 .

$$\begin{bmatrix} 1 & 2 & 1 & 0 & 0 & 0 \\ 2 & 1 & 5 & 0 & 0 & 0 \\ 1 & -1 & 3 & 0 & 0 & 0 \\ -6 & 5 & 3 & 1 & 1 & 2 \\ 1 & -3 & 2 & 1 & -1 & -2 \\ -9 & 7 & 1 & 1 & 2 & 1 \end{bmatrix} = 18 \quad \begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 5 \\ 1 & -1 & 3 \end{bmatrix} = 3 \quad \begin{bmatrix} 1 & 1 & 2 \\ 1 & -1 & -2 \\ 1 & 2 & 1 \end{bmatrix} = 6$$

Permutations

Differently from the other factorization algorithms (Gauss, LR, etc.) the block-triangular reduction use only permutations of rows and columns. From the point of view of the linear algebra a permutation can be treated as a similarity transformation.

For example, given a (6 x 6) matrix, the exchange between the rows 2 and 5, followed by the exchange between the columns 2 and 5, can be formally (but only formally) written as.

$$\mathbf{B} = \mathbf{P}^T \mathbf{A} \mathbf{P}, \quad \text{where the permutation matrix is } \mathbf{P} = (\mathbf{e}_1, \mathbf{e}_5, \mathbf{e}_3, \mathbf{e}_4, \mathbf{e}_2, \mathbf{e}_6)$$

$$\begin{array}{c} \mathbf{A} \\ \begin{bmatrix} 1 & 0 & 0 & 1 & 2 & 0 \\ 1 & -1 & 1 & 2 & -3 & -2 \\ -6 & 1 & 1 & 3 & 5 & 2 \\ 1 & 0 & 0 & 3 & -1 & 0 \\ 2 & 0 & 0 & 5 & 1 & 0 \\ -9 & 2 & 1 & 1 & 7 & 1 \end{bmatrix} \end{array} \quad \begin{array}{c} \mathbf{P} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \quad \begin{array}{c} \mathbf{P}^T \mathbf{A} \mathbf{P} \\ \begin{bmatrix} 1 & 2 & 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 5 & 0 & 0 \\ -6 & 5 & 1 & 3 & 1 & 2 \\ 1 & -1 & 0 & 3 & 0 & 0 \\ 1 & -3 & 1 & 2 & -1 & -2 \\ -9 & 7 & 1 & 1 & 2 & 1 \end{bmatrix} \end{array}$$

Remark. From the point of view of the numeric calculus the matrix multiplication is a very expensive task that we should be avoided; we use instead the direct exchange of the rows and columns or, even better, the exchange of the indices.

We have to point out that similarity transform keeps the original eigenvalues. Consequently the eigenvalues of the matrix \mathbf{A} are the same of the matrix \mathbf{B}

Eigenvalues Problem

The eigenvalue problem also takes advantage from the block-triangular form.

For example, the following matrix (6 x 6) \mathbf{A} has the eigenvalues:

$$\lambda = [-7, -1, 1, 2, 3, 5]$$

A						λ	A ₁			A ₂			λ_1	λ_2
-15	0	-16	0	0	0	-7	-15	0	-16				1	
10	2	11	0	0	0	-1	10	2	11				2	
8	0	9	0	0	0	1	8	0	9				-7	
1	3	5	3	0	-4	2				3	0	-4		-1
2	6	1	2	5	4	3				2	5	4		3
-4	9	-3	-6	-6	-1	5				-6	-6	-1		5

The eigenvalues set of the (6 x 6) matrix A is the sum of the eigenvalues set of A_1 [1 , 2 , -7] and the eigenvalues set of A_2 [-1 , 3 , 5].

Several kinds of block-triangular form

Up to now the matrices that we have seen are only one kind of block-triangular; but there are many other block-triangular schemas having blocks with different dimension each others. At last, all the blocks can have unitary dimension as in lower-triangular matrix.

Just below there are same example of block-triangular matrices (blocks are yellow)

x	x	0	0	0	0
x	x	0	0	0	0
x	x	x	0	0	0
x	x	x	x	0	0
x	x	x	x	x	x
x	x	x	x	x	x

x	x	x	0	0	0
x	x	x	0	0	0
x	x	x	0	0	0
x	x	x	x	0	0
x	x	x	x	x	0
x	x	x	x	x	x

x	0	0	0	0	0
x	x	0	0	0	0
x	x	x	0	0	0
x	x	x	x	0	0
x	x	x	x	x	0
x	x	x	x	x	x

x	x	x	0	0	0
x	x	x	0	0	0
x	x	x	0	0	0
x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x

x	x	0	0	0	0
x	x	0	0	0	0
x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x

x	0	0	0	0	0
x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x

Remark. The effort reduction is high when the dimension of the max block is low. In the first matrix the dimension of the max block is 2; in the second matrix is 3; in the third matrix the dimension of the max block is 1, obtaining the best effort reduction that it is possible.

On the contrary, the two last matrices give an effort reduction quite poor.

Permutation matrices

Is it always possible to transform a square matrix into a block-triangular form? Unfortunately not.

The chance for block-triangular reduction depends of course by the zero elements. So only sparse matrices can be block-partitioned. But this is not sufficient. It depends also by the zeros configuration of the matrix.

Two important problems arise:

1. To detect if a matrix can be reduced to a block-triangular form
2. To obtain the permutation matrix P

Several methods are developed in the past for solving these problems. One very popular is the Flow-Graph method.

Matrix Flow-Graph

Following this method, we draw the graph of the given matrix following these simple rules:

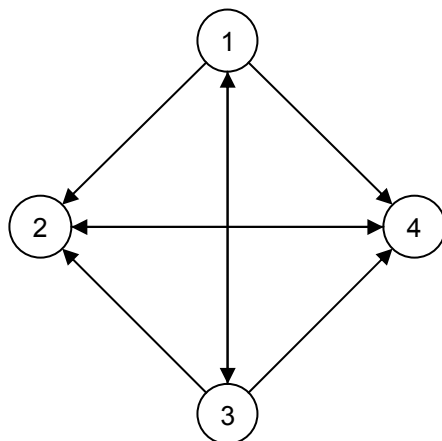
- the graph consists of *nodes* and *branches*
- the number of the nodes is equal to the dimension of the matrix
- the nodes, numbered from 1 to N, represent the elements of the first diagonal a_{ii}
- for each elements $a_{ij} \neq 0$ we draw an oriented branch (arrow) from node-i to node-j

Complicated? Not really. Let's have a look at this example.

Given the matrix **A** (4 x 4):

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

The flow-graph $G(\mathbf{A})$ associated, looks like the following (see the macro *Graph Draw* for automatic drawing)



Where

The node 1 is linked to the nodes 2, 3, 4.

The node 2 is linked to the node 4

The node 3 is linked to the nodes 1, 2, 4

The node 4 is linked to the node 2

We observe that from the node 2 there is not any path linking the node 1 or the node 3

Similarly happens if we start from the node 4

This is sufficient to say that the graph is not strong connected

Flow-Graph rule. If is always possible for each node to find a path going through all other nodes, then we say that the graph is **strong connected**

An important theorem of the Graph Theory states that if the flow-graph $G(\mathbf{A})$ is **strong connected** then the associate matrix is **not reducible** to the block-triangular form and vice versa.

On the contrary, if the flow-graph $G(\mathbf{A})$ is **not strong connected** then it always exists a permutation matrix **P** that reduces the associate matrix to the block-triangular form. Synthetically:

$G(\mathbf{A})$ strong connected	\Leftrightarrow	matrix A irreducible
$G(\mathbf{A})$ not strong connected	\Leftrightarrow	matrix A block reducible

This method is quite elegant and very important in the Graph theory. But from the point of view of the practical calculus it has several drawbacks:

- it becomes laborious for larger matrices
- the software coding is quite complicated
- it does not provide the permutation matrix **P**

In the above example, we observe that for $P = [e_2, e_4, e_1, e_3]$, the similarity transform gives a block-triangular form

$$B = P^T A P$$

A	P	$P^T A P$																																																
<table><tr><td>4</td><td>2</td><td>3</td><td>1</td></tr><tr><td>0</td><td>-1</td><td>0</td><td>1</td></tr><tr><td>3</td><td>1</td><td>-1</td><td>2</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	4	2	3	1	0	-1	0	1	3	1	-1	2	0	1	0	1	<table><tr><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	<table><tr><td>-1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>2</td><td>1</td><td>4</td><td>3</td></tr><tr><td>1</td><td>2</td><td>3</td><td>-1</td></tr></table>	-1	1	0	0	1	1	0	0	2	1	4	3	1	2	3	-1
4	2	3	1																																															
0	-1	0	1																																															
3	1	-1	2																																															
0	1	0	1																																															
0	0	1	0																																															
1	0	0	0																																															
0	0	0	1																																															
0	1	0	0																																															
-1	1	0	0																																															
1	1	0	0																																															
2	1	4	3																																															
1	2	3	-1																																															

For matrices larger than (4 x 4) the effort for searching and testing all possible permutations grows sharply. For example, it requires a heavy work for matrices like the following one. For this reasons the flow-graph method becomes practically useless for matrices of (7 x 7) or more

1	0	0	1	2	0
1	-1	1	2	-3	-2
-6	1	1	3	5	2
1	0	0	3	-1	0
2	0	0	5	1	0
-9	2	1	1	7	1

The score-algorithm

(A strange, working, algorithm)

In this chapter we shall introduce a technique for efficiently reducing a sparse matrix to a block-triangular form. The method is both simple and very efficient, and it can be applied also to medium-large matrices. It consists of an iterative process having the main goal to group zeros near the upper-right corner of the matrix using only rows and columns exchanges.

This algorithm was first ideated as automatic program, but thanks to its simplicity it can be also performed by hand, at least, for low-moderate matrices.

Let's see how it works

Giving for example the (6 x 6) matrix just shown above, we begin to initialize the permutation vector

1	2	3	4	5	6
e_1	e_2	e_3	e_4	e_5	e_6

1	0	0	1	2	0
1	-1	1	2	-3	-2
-6	1	1	3	5	2
1	0	0	3	-1	0
2	0	0	5	1	0
-9	2	1	1	7	1

The main goal is to take to the upper triangular area (grey area) the most possible zeros.

Let's begin to search all elements not zero over the first diagonal. The searching must start from the first row and from right to left: thus from the element a_{16} ; if zero, we jump to the near element a_{15} and so on till to a_{12} .

Then we repeat along the second row, from a_{26} to a_{23} .

And so on till the last row

	2		5		
1	0	0	1	2	0
1	-1	1	2	-3	-2
-6	1	1	3	5	2
1	0	0	3	-1	0
2	0	0	5	1	0
-9	2	1	1	7	1

In this example, the first element not zero is a_{15} ;

Let's search, if exists, the first zero on the same row, beginning from left to right.

The first 0 is the element a_{12} . We shall exchange the columns 2 e 5 and after, the rows 2 e 5

TUTORIAL FOR MATRIX.XLA

After the permutation (2, 5), the matrix will be the following:

A						1	5	3	4	2	6	P					
1	0	0	1	2	0	1	0	0	0	0	0	1	0	0	0	0	0
1	-1	1	2	-3	-2	0	0	0	0	1	0	0	0	0	0	0	0
-6	1	1	3	5	2	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	3	-1	0	0	0	0	1	0	0	0	0	0	0	0	0
2	0	0	5	1	0	0	1	0	0	0	0	0	0	0	0	0	0
-9	2	1	1	7	1	0	0	0	0	0	0	0	0	0	0	0	1
P ^T AP						1	2	0	1	0	0	1	0	0	0	0	0
2	1	0	5	0	0	0	1	0	0	0	0	0	0	0	0	0	0
-6	5	1	3	1	2	0	0	1	0	0	0	0	0	0	0	0	0
1	-1	0	3	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	-3	1	2	-1	-2	0	1	0	0	0	0	0	0	0	0	0	0
-9	7	1	1	2	1	0	0	0	0	0	0	0	0	0	0	0	1

We observe the zero grouping close to the upper-right corner.

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

Now the first non-zero element starting from right is a_{14} . The first 0, starting from left, is a_{13} . Thus we permute 3 e 4

After permutation 3, 4 we have:

						1	2	4	3	5	6						
A						P						P ^T AP					
1	2	0	1	0	0	1	0	0	0	0	0	1	2	1	0	0	0
2	1	0	5	0	0	0	1	0	0	0	0	2	1	5	0	0	0
-6	5	1	3	1	2	0	0	0	1	0	0	1	-1	3	0	0	0
1	-1	0	3	0	0	0	0	1	0	0	0	-6	5	3	1	1	2
1	-3	1	2	-1	-2	0	0	0	0	1	0	1	-3	2	1	-1	-2
-9	7	1	1	2	1	0	0	0	0	0	1	-9	7	1	1	2	1

All zeros are now positioned in the upper-triangular area. The matrix is partitioned in two (3 x 3) blocks. The process ends.

The finally permutation matrix is

1	2	3	4	5	6
e1	e5	e4	e3	e2	e6

As shown, with only 2 permutations we were able to reduce in block-triangular form a (6 x 6) matrix. We have to put in evidence that we are worked only by hand. This method keeps a good efficiency also with larger matrices.

Let's have a look to another example.

Reduce, if possible, the following (6 x 6) matrix

↓	↓				
3	1	-1	1	-5	2
0	-1	0	1	0	0
5	1	1	2	-3	4
0	0	0	1	0	0
1	1	7	-9	13	1
0	1	0	-6	0	1

The first element $\neq 0$, from right, is: a_{16}
The first element $= 0$, from left, is: a_{21} .
So the pivot columns are 1 and 6

		⇓	⇓		
1	1	0	-6	0	0
0	-1	0	1	0	0
4	1	1	2	-3	5
0	0	0	1	0	0
1	1	7	-9	13	1
2	1	-1	1	-5	3

The first element $\neq 0$, from right, is: a14
 The first element = 0, from left, is: a13.
 So the pivot columns are 3 and 4

		⇓	⇓		
1	1	-6	0	0	0
0	-1	1	0	0	0
0	0	1	0	0	0
4	1	2	1	-3	5
1	1	-9	7	13	1
2	1	1	-1	-5	3

The first element $\neq 0$, from right, is: a13
 The first element = 0, from left, is: a21.
 So the pivot columns are 1 e 3.

Finally we get the block-triangular matrix.

1	0	0	0	0	0
1	-1	0	0	0	0
-6	1	1	0	0	0
2	1	4	1	-3	5
-9	1	1	7	13	1
1	1	2	-1	-5	3

The matrix has been block-partitioned:
 There are 3 blocks (1 x 1) and one block (3 x 3)

We observe that this algorithm does not provide any information about the success of the process. It simply stops itself when there are no more elements to permute. At the end of the process, if the result matrix is in block-triangular form, then the original matrix is reducible. Otherwise, it means that the original matrix is irreducible and its flow-graph is strong connected.

The Score-Function

The matrices used up to now had all zero elements completely filled into the upper-triangle area. Now let's see what happens if the matrix has more zeros than those tightly necessary for block partitioning (spurious zeros). In that case not all permutations will be useful for grouping zeros. Some of them will be useless and some others even will go zeros away from the upper-right corner. Thus, it is necessary to measure the goodness of each permutation. By a simple inspection it is easy to select the "good" permutations from "bad" permutations. But in automatic process it is necessary to choose a function for evaluating the permutation goodness: the *score-function* is the measure adopted in this algorithm.

The score function counts the zeros in the upper triangle area (grey) before (A) and after the permutation (B) returning the difference.

$$score = \sum_B w(i, j) - \sum_A w(i, j)$$

The score will be positive if the permutation will be advantageous otherwise will be negative or null.

×					
×	×				
×	×	×			
×	×	×	×		
×	×	×	×	×	
×	×	×	×	×	×

The zeros have not the same weight: the zeros nearest to the upper-right corner have a higher weight, because the matrix filled with zeros close to the upper-right corner is better than the one with zeros close to the first diagonal.

TUTORIAL FOR MATRIX.XLA

x	x	0	x	0	0
x	x	x	x	0	0
x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x

better

x	x	x	x	x	x
x	x	0	x	x	x
x	x	x	0	x	x
x	x	x	x	0	x
x	x	x	x	x	0
x	x	x	x	x	x

worse

Apart this concept, the weigh function $w(i,j)$ is arbitrary. One function that we have tested with good result is the following

$$w(i,j) = \begin{cases} 0 & \Leftrightarrow a_{ij} \neq 0 \\ (n-i+1)^2 \cdot j^2 & \Leftrightarrow a_{ij} = 0 \end{cases} \quad \text{Weight function for a matrix (n x n)}$$

For each permutation recognized, the algorithm measures the score; if positive the permutation is performed, otherwise the permutation is rejected and the algorithm continue to find a new permutation. After same loops the zeros disposition will reach the maximum score possible; every other attempt of permutation will produce a negative or null score. So the algorithm will stop the process.

Same examples

Now let's see the algorithm in practical cases

A

1	2	0	2	0	0
0	1	2	0	-3	0
0	0	1	0	5	3
0	3	1	1	0	0
0	0	0	0	1	3
0	0	0	0	1	3

P^T A P

1	3	0	0	0	0
1	3	0	0	0	0
5	3	1	0	0	0
-3	0	2	1	0	0
0	0	1	3	1	0
0	0	0	2	2	1

P = [e5, e6, e3, e2, e4, e1]

Accepted permutations = 6

Rejected permutations = 4

A

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

P^T A P

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

TUTORIAL FOR MATRIX.XLA

$P = [e7, e3, e10, e1, e8, e2, e4, e6, e9, e5]$

Accepted permutations = 9

Rejected permutations = 10

A									
1	0	1	0	1	0	1	6	0	1
1	1	0	1	1	1	1	1	1	0
0	0	1	0	0	0	0	0	0	0
1	0	0	1	1	4	1	1	0	1
0	0	1	0	1	0	5	0	0	0
1	0	1	0	0	1	1	1	0	0
0	0	1	0	0	0	1	0	0	0
0	0	1	0	4	0	1	3	0	0
1	0	1	3	0	4	1	1	1	1
0	0	0	0	0	0	1	4	0	1

$P^T A P$									
1	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0
1	5	1	0	0	0	0	0	0	0
1	1	4	3	0	0	0	0	0	0
0	1	0	4	1	0	0	0	0	0
1	1	1	6	1	1	0	0	0	0
1	1	0	1	0	1	1	0	0	0
0	1	1	1	1	1	4	1	0	0
1	1	0	1	1	1	4	3	1	0
0	1	1	1	0	1	1	1	1	1

$P = [e3, e7, e5, e8, e10, e1, e6, e4, e9, e2]$

Accepted permutations = 7

Rejected permutations = 1

A

3	0	8	0	0	3	0	3	0	0	0	6	0	0	0	14	8	0	7	0
4	4	0	0	0	6	0	6	0	0	3	9	0	0	0	20	0	0	10	4
0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	3	0	2	0	0
0	0	17	10	10	0	10	0	10	0	0	15	0	10	10	0	17	10	16	0
4	9	16	9	9	11	9	11	9	9	8	14	9	9	9	30	0	9	15	9
0	0	0	0	0	1	0	0	0	0	0	4	0	0	0	10	0	0	20	0
0	0	20	20	0	0	13	20	0	20	12	0	0	13	13	38	20	13	0	13
0	0	0	0	0	2	0	2	0	0	0	20	0	0	0	0	7	0	6	0
4	11	18	0	20	13	11	13	11	11	10	16	11	11	11	34	18	0	17	11
20	5	0	0	0	7	0	0	0	5	0	0	0	0	0	0	1	0	11	5
4	0	9	0	0	0	0	4	0	0	1	0	0	0	0	20	0	0	8	0
0	0	4	0	0	0	0	0	0	0	0	2	0	0	0	0	4	0	3	0
4	6	13	0	0	8	0	8	0	6	5	11	6	0	0	0	0	0	12	6
0	7	14	0	0	9	0	9	0	7	20	12	7	7	0	0	0	0	13	0
4	0	19	12	12	14	12	14	12	0	0	17	0	12	12	36	19	12	18	0
0	0	5	0	0	0	0	0	0	0	0	3	0	0	0	8	5	0	4	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
4	8	15	0	0	10	0	10	0	8	7	13	8	8	0	0	0	8	14	8
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	1	0
4	0	10	0	0	5	0	5	0	0	2	8	0	0	0	18	0	0	0	3

$P^T A P$

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	3	4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	4	5	3	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	20	0	4	10	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	6	0	20	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0
8	7	8	6	14	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0
9	8	9	0	20	0	4	4	1	0	0	0	0	0	0	0	0	0	0	0
10	0	10	8	18	5	5	4	2	3	0	0	0	0	0	0	0	0	0	0
11	0	10	0	9	20	6	6	4	3	4	4	0	0	0	0	0	0	0	0
12	11	0	0	7	0	20	0	5	5	5	5	0	0	0	0	0	0	0	0
13	12	13	11	0	8	8	4	5	6	6	6	6	0	0	0	0	0	0	0
14	13	14	12	0	9	9	0	20	0	7	7	7	7	0	0	0	0	0	0
15	14	15	13	0	10	10	4	7	8	8	8	8	8	8	0	0	0	0	0
16	19	18	19	17	36	14	14	4	0	0	0	0	0	12	12	12	12	12	12
17	16	17	15	0	0	0	0	0	0	0	0	0	0	10	10	10	10	10	10
18	20	0	20	0	38	0	20	0	12	13	0	20	0	13	13	13	20	13	0
19	18	17	18	16	34	13	13	4	10	11	11	11	11	11	0	11	0	11	11
20	0	15	16	14	30	11	11	4	8	9	9	9	9	9	9	9	9	9	9

$P = [e_{17}, e_{19}, e_3, e_{12}, e_{16}, e_6, e_8, e_1, e_{11}, e_{20}, e_2, e_{10}, e_{13}, e_{14}, e_{18}, e_{15}, e_4, e_7, e_9, e_5]$

Accepted permutations = 18

Rejected permutations = 237

As we can see, also for larger matrices the number of permutations remains quite limited. Regarding this and that the permutation is much faster than any other arithmetic operation in floating point, we can guess the high speed of this algorithm

TUTORIAL FOR MATRIX.XLA

In Excel, with matrix.xla, is very easy to study the matrix permutations.

A simple arrangement for (6 x 6) matrices is shown in the following example. We have used the function **MatPerm**. When you change the permutation numbers, also the permutation matrix changes an, consequently the final transform matrix

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	
1								Permutations														
2								5	6	3	2	4	1									
3		A								P							P ^T A P					
4	1	2	0	2	0	0		0	0	0	0	0	1			1	3	0	0	0	0	
5	0	1	2	0	-3	0		0	0	0	1	0	0			1	3	0	0	0	0	
6	0	0	1	0	5	3		0	0	1	0	0	0			5	3	1	0	0	0	
7	0	3	1	1	0	0		0	0	0	0	1	0			-3	0	2	1	0	0	
8	0	0	0	0	1	3		1	0	0	0	0	0			0	0	1	3	1	0	
9	0	0	0	0	1	3		0	1	0	0	0	0			0	0	0	2	2	1	
10																						
11	{=MatPerm(H2:M2)}																					
12								{=M_PROD(M_T(H4:M9);A4:F9;H4:M9)}														

WHITE PAGE

Eigenproblems

This chapter explains how to solve common problems involving eigenvalues and eigenvectors systems, with the aid of many examples and different methods.

Eigenproblems

Eigenvalues

These problems are very common in math, physics, engineering, etc. Usually they consist to solve the following matrix equation

$$A x = \lambda x \quad (1)$$

Where **A** is n x n matrix and the unknowns are λ and **x**, respectively called *eigenvalue* and *eigenvector*⁵. Rearranging the equation (1) we have:

$$(A - \lambda I)x = 0 \quad (2)$$

This homogeneous system can have no-trivial solutions if its determinant is zero. That is:

$$|A - \lambda I| = 0 \quad (3)$$

Characteristic Polynomial

The left side of (3) is an n degree polynomial in λ , – called *characteristic polynomial* - whose roots are the eigenvalues of matrix **A**.

For **A** 2x2, the matrix of system (2) becomes:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} - \lambda & a_{12} \\ a_{21} & a_{22} - \lambda \end{bmatrix}$$

⁵ In lingua italiana, λ e **x** sono conosciuti come *autovalore* e *autovettore*

Computing the determinant we have the equation (3) in expanded form

$$\lambda^2 - (a_{11} + a_{22})\lambda + \det(A) = 0$$

For **A** 3x3, the matrix of system (2) becomes:

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

And its characteristic equation (3) becomes

$$-\lambda^3 + (a_{11} + a_{22} + a_{33})\lambda^2 - (a_{11}a_{22} - a_{12}a_{21} + a_{11}a_{33} - a_{13}a_{31} + a_{22}a_{33} - a_{23}a_{32})\lambda + \det(A) = 0$$

With larger matrix the difficulty for computing the characteristic polynomial grows sharply; fortunately there is a very efficient way to compute it using the Newton-Girard recursive formulas. In Matrix.xla we can get the characteristic polynomial coefficients for any square matrix by the function **MatCharPoly()**.

Roots of characteristic polynomial

Apart the 2nd degree case only, finding roots of a polynomial need numerical approximated methods. Matrix.xla has the function **Poly_Roots()** that finds all roots - real or complex - of a given polynomial, using the *Lin-Bairstow* method. This function is suitable for general polynomials up to 6-7 degree.

From ver. 1.4 there is also the function **Poly_Roots_QR** for finding all polynomial roots. It uses the QR algorithm and it is adapt for polynomial up to 12-13 degree.

Case of symmetric matrix

Symmetric matrix plays a fundamental role in numeric analysis. It has a great important feature. Its eigenvalues are all-real. Or, in other words, its characteristic polynomial has only real roots. Another important reason for using symmetric matrices is that there are many straight, efficient and also accurate algorithms for the eigensystem solution: much more complicate for asymmetric matrices.

Tip. There is a close formula for generating an n x n symmetric matrix having the first n natural numbers as eigenvalues

$$a_{ii} = \frac{(i+2)n - 4i + 2}{n}$$

$$a_{ij} = 2 \cdot \frac{n+1-i-j}{n} \quad i \neq j$$

Below there are the first matrices for n=2, 3, 4, 5, 6, 8

TUTORIAL FOR MATRIX.XLA

2	0
0	1

eigenvalues: 1, 2

$\frac{7}{3}$	$\frac{2}{3}$	0
$\frac{2}{3}$	$\frac{6}{3}$	$-\frac{2}{3}$
0	$-\frac{2}{3}$	$\frac{5}{3}$

eigenvalues: 1, 2, 3

2.5	1	0.5	0
1	2.5	0	-0.5
0.5	0	2.5	-1
0	-0.5	-1	2.5

eigenvalues: 1, 2, 3, 4

2.6	1.2	0.8	0.4	0
1.2	2.8	0.4	0	-0.4
0.8	0.4	3	-0.4	-0.8
0.4	0	-0.4	3.2	-1.2
0	-0.4	-0.8	-1.2	3.4

eigenvalues: 1, 2, 3, 4, 5

$\frac{8}{3}$	$\frac{4}{3}$	$\frac{3}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	0
$\frac{4}{3}$	$\frac{9}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	0	$-\frac{1}{3}$
$\frac{3}{3}$	$\frac{2}{3}$	$\frac{10}{3}$	0	$-\frac{1}{3}$	$-\frac{2}{3}$
$\frac{2}{3}$	$\frac{1}{3}$	0	$\frac{11}{3}$	$-\frac{2}{3}$	$-\frac{3}{3}$
$\frac{1}{3}$	0	$-\frac{1}{3}$	$-\frac{2}{3}$	$\frac{12}{3}$	$-\frac{4}{3}$
0	$-\frac{1}{3}$	$-\frac{2}{3}$	$-\frac{3}{3}$	$-\frac{4}{3}$	$\frac{13}{3}$

eigenvalues: 1, 2, 3, 4, 5, 6

2.75	1.5	1.25	1	0.75	0.5	0.25	0
1.5	3.25	1	0.75	0.5	0.25	0	-0.25
1.25	1	3.75	0.5	0.25	0	-0.25	-0.5
1	0.75	0.5	4.25	0	-0.25	-0.5	-0.75
0.75	0.5	0.25	0	4.75	-0.5	-0.75	-1
0.5	0.25	0	-0.25	-0.5	5.25	-1	-1.25
0.25	0	-0.25	-0.5	-0.75	-1	5.75	-1.5
0	-0.25	-0.5	-0.75	-1	-1.25	-1.5	6.25

eigenvalues: 1, 2, 3, 4, 5, 6, 7, 8

Example – How to check the Cayley-Hamilton theorem

Regarding the characteristic polynomial $P(\lambda)$ an important theorem, known as *Cayley-Hamilton's theorem* - states that the any square matrix **A** verifies its characteristic polynomial. That is In formula:

$$P(\mathbf{A}) = \mathbf{O} \quad (\text{where } \mathbf{O} \text{ is the null matrix})$$

The above matricial equation can be formally obtained substituting the variable λ with the matrix **A**. Let's see how to test this statement with a practical example in Excel. Given the following (3 x 3) matrix

$$\mathbf{A} = \begin{bmatrix} 11 & 9 & -2 \\ -8 & -6 & 2 \\ 4 & 4 & 1 \end{bmatrix}$$

Having its characterist polynomial

$$P(\lambda) = 6 - 11\lambda + 6\lambda^2 - \lambda^3$$

After substituting λ with **A**, we have

$$P(\mathbf{A}) = 6 \cdot \mathbf{I} - 11 \cdot \mathbf{A} + 6 \cdot \mathbf{A}^2 - \mathbf{A}^3$$

Evaluating this formula by hand is quite tedious, but it is very easy in Excel. Let's see the following spreadsheet arrangement using the function **M_POWER**

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Cayley-Hamilton test												
2													
3	Char. Poly. coefficients					A				P(A)			
4	a₀	a₁	a₂	a₃		11	9	-2		0	0	0	
5	6	-11	6	-1		-8	-6	2		0	0	0	
6						4	4	1		0	0	0	
7													
8	{=A5*M_ID0+B5*F4:H6+C5*M_POWER(F4:H6;2)+D5*M_POWER(F4:H6;3)}												
9													

Note that we have inserte the $P(\mathbf{A})$ formula as an array funtion {=...}

Of course it is also possible to perform the matrix powers with the matrix product.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Char. Poly. coefficients					Cayley-Hamilton test										
2	a₀	a₁	a₂	a₃												
3	6	-11	6	-1												
4																
5		I				A				A²				A³		
6	1	0	0			11	9	-2		41	37	-6		131	123	-14
7	0	1	0			-8	-6	2		-32	-28	6		-104	-96	14
8	0	0	1			4	4	1		16	16	1		52	52	1
9																
10		O				{=M_PROD(\$E\$6:\$G\$8;E6:G8)}										
11	0	0	0													
12	0	0	0			{=A3*A6:C8+B3*E6:G8+C3*I6:K8+D3*M6:O8}										
13	0	0	0													
14																

Eigenvectors

Logically speaking, once we have found an eigenvalue we can solve the homogeneous system (2) in order to find the associate eigenvector.

$$(A - \lambda_i I)x_i = 0 \Rightarrow x_i$$

Normally for each real eigenvalues having one multiplicity, there is only one eigenvector. For multiplicity 2, we will find two eigenvectors or even only one.

Step-by-step method

The method explained above is general and is valid for all kind of matrices. It is known by every math student and it is very popular. For this reasons is detailed in this document, despite his intrinsically inefficiency. As we can see in the following paragraphs, there are other methods that can compute both eigenvectors and eigenvalues at the same time in a very efficient and fast way. They are suitable for larger matrices, while the step-by-step method can be applied to matrices of low dimension (usually from 2x2 , up to 5x5).

But didactically speaking this method is still valid and it can help when other methods fail or raise doubts.

Let's see how works with same examples

Example - Simple eigenvalues

Find all eigenvalues and associated eigenvectors of the following matrix

$$\mathbf{A} = \begin{vmatrix} -4 & 14 & -6 \\ -8 & 19 & -8 \\ -5 & 10 & -3 \end{vmatrix}$$

Reassuming the step-by-step method, we have to:

1. Compute the characteristic polynomial's coefficients
2. Find its roots, that is the matrix eigenvalues λ_i
3. For each root λ_i build the matrix $\mathbf{A} - \lambda_i \mathbf{I}$
4. Find the associate eigenvector \mathbf{x}_i solving the homogeneous system

For task 1) we use the MathCharPoly() function; for task 2) we use the Poly_Roots() function; task 3) are performed with M_ID() function that return the identity matrix; finally task 4) use the SYSLINSING() to find a solution of the singular system.

TUTORIAL FOR MATRIX.XLA

	A	B	C	D	E	F	G	H	I	J	K
1	A				coeff eigenvalues						
2	-4	14	-6		42	real	imm				
3	-8	19	-8		-41	2	0				
4	-5	10	-3		12	3	0				
5					-1	7	0				
6	{=MatCharPoly(A2:C4)}										
7											
8	A - λ I for λ = 2				A - λ I for λ = 3				A - λ I for λ = 7		
9	-6	14	-6		-7	14	-6		-11	14	-6
10	-8	17	-8		-8	16	-8		-8	12	-8
11	-5	10	-5		-5	10	-5		-5	10	-10
12											
13	{=A2:C4-F3*M_ID0}				{=A2:C4-F4*M_ID0}				{=A2:C4-F5*M_ID0}		
14											
15											
16	0	0	-1		0	2	0		0	0	2
17	0	0	-0		0	1	0		0	0	2
18	0	0	1		0	-0	0		0	0	1
19											
20	{=SYSLINSING(A9:C11)}				{=SYSLINSING(E9:G11)}				{=SYSLINSING(I9:K11)}		
21											

For the given matrix, we have found the following eigenvalues and eigenvectors

Eigenvalues	
λ ₁	2
λ ₂	3
λ ₃	7

Eigenvector		
x1	x2	x3
-1	2	2
0	1	2
1	0	1

Example - How to check an eigenvector

Once we have found the eigenvectors, we can easily verify them by simple matrix multiplication.

$$u_i = A x_i \Rightarrow u_i = \lambda_i x$$

If **x** is an eigenvector, the vector **u** must be exactly a λ multiple of the vector **x**, as we can see in the worksheet below

	A	B	C	D	E	F	G	H	I	J	K	L
26	Matrix				Eigenvector				verify			
27	A				x1	x2	x3		u1	u2	u3	
28	-4	14	-6		-1	2	2		-2	6	14	
29	-8	19	-8		0	1	2		0	3	14	
30	-5	10	-3		1	0	1		2	0	7	
31												
32	Eigenvalues= 2, 3, 7								{=MMULT(A28:C30,E28:G30)}			
33												

Eigenvectors are not unique. It can be easily proved that any multiple of an eigenvector is an eigenvector too. It means that if $(-1, 0, -1)$ is an eigenvector, other possible eigenvectors are:

Matrix	Eigenvalue	Eigenvectors ...																														
<table><tr><td>-4</td><td>14</td><td>-6</td></tr><tr><td>-8</td><td>19</td><td>-8</td></tr><tr><td>-5</td><td>10</td><td>-3</td></tr></table>	-4	14	-6	-8	19	-8	-5	10	-3	$\lambda = 2$	<table><tr><td>-0.04</td><td>-0.5</td><td>-1</td><td>-2</td><td>-3</td><td>-4</td><td>-5</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0.04</td><td>0.5</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	-0.04	-0.5	-1	-2	-3	-4	-5	0	0	0	0	0	0	0	0.04	0.5	1	2	3	4	5
-4	14	-6																														
-8	19	-8																														
-5	10	-3																														
-0.04	-0.5	-1	-2	-3	-4	-5																										
0	0	0	0	0	0	0																										
0.04	0.5	1	2	3	4	5																										

For convention, mathematicians use to take the eigenvector with Norm 1, that is: $|\mathbf{x}| = 1$. In that case it is called *eigenvector*.

Following this rule the eigenvectors matrix becomes as we can see at the right

Sometimes, in order to avoid decimal numbers, we normalize only the smallest value of the vector; for that, we divide all values for the GCD

	A	B	C	D	E	F	G
49	Eigenvector				Eigenvector		
50	x1	x2	x3		u1	u2	u3
51	-1	2	2		-0.70711	0.89443	0.66667
52	0	1	2		0	0.44721	0.66667
53	1	0	1		0.70711	0	0.33333
54							
55	{=A51:A53/M_ABS(A51:A53)}						
56							

The SYSLINSING function adopts this solution. If you want to get the eigenvectors you have to do manually.

Example - Eigenvalues with multiplicity

Find all eigenvalues and associated eigenvectors of the following matrix

A =

-7	-9	9
6	8	-6
-2	-2	4

For the given matrix we have found two roots:

$$\lambda = 1, m = 1$$

$$\lambda = 2, m = 2$$

With the eigenvalue with multiplicity = 1, we get one eigenvector; while with the second one with multiplicity = 2, we get two eigenvectors

	A	B	C	D	E	F	G
1	A				coeff	eigenvalues	
2	-7	-9	9		4	real	imm
3	6	8	-6		-8	1	0
4	-2	-2	4		5	2	0
5					-1	2	0
6	{=MatCharPoly(A2:C4)}						
7							
8	A - λI for λ = 1				A - λI for λ = 2		
9	-8	-9	9		-9	-9	9
10	6	7	-6		6	6	-6
11	-2	-2	3		-2	-2	2
12	{=A2:C4-C8*M_ID0}				{=A2:C4-F4*M_ID0}		
13							
14							
15							
16	0	0	4.5		0	-1	1
17	0	0	-3		0	1	-0
18	0	0	1		0	-0	1
19							
20	{=SYSLINSING(A9:C11)}				{=SYSLINSING(E9:G11)}		
21							

Tip: accuracy of multiple roots is in general less than other singular roots. For this reason, sometimes, the SYSLINSING function cannot return any solution. In those cases, try to set the SYSLINSING parameter MaxError less than 1E-15, depending from the eigenvalue accuracy (usually for a root with $m = 2$, we set MaxError = 1E-10)

We note that the matrix obtained with eigenvalue=2 has three rows multiple of each other's. So its rank is 1 and its solution generates a subspace of $3 - 1 = 2$ dimension.

(See for better details the *Rouché-Capelli Theorem* in the previous chapter)

But this is always valid? The multiplicity gives the dimension of the eigenvector subspace? Unfortunately no. There are cases in which the multiplicity doesn't correspond to the associated eigenvectors.

Let's see the following example.

Example - Eigenvalues with multiplicity not corresponding to eigenvectors

Find all eigenvalues and associated eigenvectors of the following matrix

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

For the given matrix the characteristic polynomial is:

$$-\lambda^3 + 4\lambda^2 - 4\lambda$$

That has two roots:

$$\lambda = 0, m = 1$$

$$\lambda = 2, m = 2$$

With the eigenvalue with multiplicity = 1, we get one eigenvector; with the second one with multiplicity = 2, we get only one eigenvector, not twice.

	A	B	C	D	E	F	G
1	A				coeff	eigenvalues	
2	1	2	1		-0	real	imm
3	2	0	-2		-4	0	0
4	-1	2	3		4	2	0
5					-1	2	0
6	{=MatCharPoly(A2:C4)}						
7							
8	A - λI	for λ = 0			A - λI	for λ = 2	
9	1	2	1		-1	2	1
10	2	0	-2		2	-2	-2
11	-1	2	3		-1	2	1
12							
13	{=A2:C4-C8*M_ID0}				{=A2:C4-F4*M_ID0}		
14							
15							
16	0	0	1		0	0	1
17	0	0	-1		0	0	-0
18	0	0	1		0	0	1
19							
20	{=SYSLINSING(A9:C1)}				{=SYSLINSING(E9:G11)}		
21							

Example - Complex Eigenvalues

Sometimes happens that not all roots of the characteristic polynomial are real. In that case the eigenvectors associated at complex eigenvalues are complex too.

Find all eigenvalues and associated eigenvectors of the following matrix

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

The characteristic polynomial is: $-\lambda^3 + 12\lambda^2 - 46\lambda + 50$

	A	B	C	D	E	F	G	H	I	J	K
1	Matrix A				coeff.	Complex Eigenvalues					
2					52	real	imm				
3	9	-6	7		-46	2	0			{=Poly_Roots(E3:E5)}	
4	1	4	1		12	5	1				
5	-3	4	-1		-1	5	-1				
6										{=MatCharPoly(A3:C5)}	
7											

The eigenvalues are $\lambda_1 = 2$, $\lambda_2 = 5 + j$, $\lambda_3 = 5 - j$

In Matrix.xla there is not a SYSLINSING for solve singular complex system, but we can derive a real system from the original complex one:

Separating both eigenvalue and eigenvector in their real and imaginary parts:

$$\lambda = \lambda_{re} + j\lambda_{im} \quad x = x_{re} + jx_{im}$$

The homogeneous linear system, becomes

$$(A - \lambda I)x = 0 \Rightarrow (A - (\lambda_{re} + j\lambda_{im})I)(x_{re} + jx_{im}) = 0$$

Rearranging:

$$((A - \lambda_{re}I)x_{re} + \lambda_{im}I x_{im}) + j(-\lambda_{im}I x_{re} + (A - \lambda_{re}I)x_{im}) = 0$$

The above complex equation is equivalent to the following homogeneous system

$$\begin{cases} (A - \lambda_{re}I)x_{re} + \lambda_{im}I x_{im} = 0 \\ -\lambda_{im}I x_{re} + (A - \lambda_{re}I)x_{im} = 0 \end{cases} \Rightarrow \begin{bmatrix} (A - \lambda_{re}I) & \lambda_{im}I \\ -\lambda_{im}I & (A - \lambda_{re}I) \end{bmatrix} \cdot \begin{bmatrix} x_{re} \\ x_{im} \end{bmatrix} = 0$$

Let's see how to arrange a solution in Excel

The 6 x 6 homogeneous system matrix is built in four 3x3 sub-matrices.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Matrix A				coeff.	Complex Eigenvalues							
2					52	real	imm						
3	9	-6	7		-46	2	0						
4	1	4	1		12	5	1						
5	-3	4	-1		-1	5	-1						
6													
7	complex eigenvalue =				5	1							
8													
9	Homogeneous real system matrix												
10	4	-6	7	1	0	0		0	0	0	0	-2	-1
11	1	-1	1	0	1	0		0	0	0	0	-0	-1
12	-3	4	-6	0	0	1		0	0	0	0	1	-0
13	-1	0	0	4	-6	7		0	0	0	0	1	-2
14	0	-1	0	1	-1	1		0	0	0	0	1	-0
15	0	0	-1	-3	4	-6		0	0	0	0	-0	1
16													
17													
18													
19													
20													

The solution of the homogeneous system returned by SYSLINSING is conceptual divided in two parts: the upper contains the real part of the eigenvectors; the lower there is the imaginary parts of the same eigenvectors.

Substituting the conjugate eigenvalues we find conjugate eigenvectors.

The case of real eigenvalue 2 is the same of the above example, so we do not repeat the process. Rather we want to show here how to arrange a check for complex eigenvectors.

Example - How to check a complex eigenvector

Given the matrix **A** and one of theirs eigenvalue λ , prove that the vector **x** is an eigenvector

$$\mathbf{A} = \begin{bmatrix} 9 & -6 & 7 \\ 1 & 4 & 1 \\ -3 & 4 & -1 \end{bmatrix} \quad \lambda = 5+j \quad \begin{bmatrix} x_{re} & x_{im} \\ -1 & -2 \\ -1 & 0 \\ 0 & 1 \end{bmatrix}$$

The test can be arrange as in the following worksheet

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Complex eigenvalue				5	1									
2															
3	Complex matrix A				eigenvector				A x				check		
4	real part			imm. part				xre	xim		xre	xim		xre	xim
5	9	-6	7	0	0	0		-1	-2		-3	-11		-3	-11
6	1	4	1	0	0	0		-1	0		-5	-1		-5	-1
7	-3	4	-1	0	0	0		0	1		-1	5		-1	5
8															
9															
10															
11															
12															
13															

{=M_MULT_C(A5:F7,H5:I7)}

{=H5:H7*E1-I5:I7*F1}

{=H5:H7*F1+I5:I7*E1}

We have used the function for complex matrix multiplication **M_MAT_C()** of Matrix.xla. Note that we have to insert the imaginary part of the matrix because those complex functions always request both parts: real and imaginary.

There is also another way to compute directly the eigenvector of a given eigenvalue: the functions **MatEigenvector()** and **MatEigenvector_C()** of Matrix.xla return the eigenvector associate to an eigenvalue; the first function works for real eigenvalues and the second one for complex. See the chapter "Functions References" for details

Similarity Transformation

This linear transformation is very important because it leave eigenvalues unchanged. Let's see how it works. Giving a square matrix **A** and a second square matrix **B** we generate a third matrix **C** by the formula:

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

We say: **C** is similarity transformed of **A** by matrix **B**

Similarity transformations play a crucial role in the computation of eigenvalues because they leave the eigenvalues of a matrix unchanged. Thus, eigenvalues of **A** are the same of those of **C**, for any matrix **B**

It can be easily demonstrate that $\det(\mathbf{C} - \lambda \mathbf{I}) = \det(\mathbf{A} - \lambda \mathbf{I})$

In fact, keeping in mind that $\mathbf{I} = \mathbf{B}^{-1} \mathbf{B}$, we can write:

$$\det(\mathbf{C} - \lambda \mathbf{I}) = \det(\mathbf{B}^{-1} \mathbf{A} \mathbf{B} - \lambda \mathbf{I}) = \det(\mathbf{B}^{-1} \mathbf{A} \mathbf{B} - \lambda \mathbf{B}^{-1} \mathbf{B})$$

But, rearranging, we have

$$\begin{aligned} \det(\mathbf{B}^{-1} \mathbf{A} \mathbf{B} - \lambda \mathbf{B}^{-1} \mathbf{B}) &= \det(\mathbf{B}^{-1} (\mathbf{A} \mathbf{B} - \lambda \mathbf{B})) = \det(\mathbf{B}^{-1} (\mathbf{A} - \lambda \mathbf{I}) \mathbf{B}) = \\ &= \det(\mathbf{B}^{-1}) \det(\mathbf{A} - \lambda \mathbf{I}) \det(\mathbf{B}) = \det(\mathbf{A} - \lambda \mathbf{I}) \det(\mathbf{B}^{-1}) \det(\mathbf{B}) = \det(\mathbf{A} - \lambda \mathbf{I}) \end{aligned}$$

Example - verify that the similarity-transformed matrix of **A** by the matrix **B** has the same eigenvalues.

In order that eigenvalues are the same is sufficient that the characteristic polynomial are equals. For computing the transformed matrix we is useful the function **M_BAB()** of Matrix.xla. But, of course we can use, as well, the standard formula

`=MMULT(MMULT(MINVERSE(E3:G5),A3:C5),E3:G5)`

	A	B	C	D	E	F	G	H	I	J	K
2	Matrix A				Matrix B				Matrix B ⁻¹ A B		
3	1	-2	6		1	2	0		7.571	-4	-0.57
4	1	4	-3		-2	1	-1		1.714	2	-1.71
5	-2	-4	5		1	0	-1		-3.43	4	0.429
6											
7	coeff eigenvalues				{=M_BAB(A3:C5,E3:G5)}				coeff eigenvalues		
8	30	real	imm						30	real	imm
9	-31	2	0						-31	2	0
10	10	3	0						10	3	0
11	-1	5	0						-1	5	0
12											

similarity transformation
eigenvalues unchanged

For computing the characteristic polynomial coefficients we have used the function **MatCharPoly()**

Factorization methods

The heart of many eigensystem routines is to perform a sequence of similarity transformation until the result matrix is nearly diagonal with small error.

$$\begin{aligned}
 \mathbf{A}_1 &= (\mathbf{P}_1)^{-1} \mathbf{A} (\mathbf{P}_1) \\
 \mathbf{A}_2 &= (\mathbf{P}_2)^{-1} \mathbf{A}_1 (\mathbf{P}_2) \\
 \mathbf{A}_3 &= (\mathbf{P}_3)^{-1} \mathbf{A}_2 (\mathbf{P}_3) \\
 &\dots\dots\dots \\
 \mathbf{A}_n &= (\mathbf{P}_n)^{-1} \mathbf{A}_{n-1} (\mathbf{P}_n)
 \end{aligned}
 \qquad
 \mathbf{A}_{n-1} \xrightarrow{n \rightarrow \infty} \mathbf{D} \quad \text{Where D is diagonal}$$

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix}$$

Eigenvalues of a diagonal matrix are simply the diagonal elements; but, because they are equal to the matrix A for the similarity property, we have found also the eigenvalues of the matrix A. We found this strategy in algorithms such as Jacobi' iterative rotations, QR factorization, etc.

Note: This iterative method does not converge for all matrices. There are several convergence criterions. One of the most popular says that convergence is guaranteed for the class of symmetric matrices.

Eigensystems versus resolution methods

In the above paragraph we have spoke about the general method to resolve eigensystems. It starts form the characteristic polynomial and builds the solutions step by step. It is valid for any kind of matrix, with real or complex eigenvalues. That fact is that this method can be used only for low dimension matrix. When the matrix size is higher than 3, this method becomes quite tedious, long and inefficient.

To overcome this, many algorithms have been developed. Generally, they calculate all eigenvalues and eigenvector by efficient iterative methods. The price is those methods are no more general but they are specialized for a sort of matrix class. Very efficient algorithms exist for the symmetric matrix class, but the same algorithm cannot work, for example, with complex eigenvalues matrices. So, in front of an eigensystem, we have to analyze which method can be applied.

Matrix.xla offers several different methods; the range of application is synthesized in the following table

Eigenproblems	Symmetric real matrix	Non symmetric real matrix	
Methods	Real eigenvalues	Real eigenvalues	Complex eigenvalues
Jacoby	yes	no	no
QR factorization	yes	yes	yes
Powers	yes	yes	no
Characteristic Polyn ⁶ .	yes	Yes	yes

There are also special algorithm for tridiagonal matrices, more efficient for large dimension.

⁶ "Characteristic polynomial" is also called "step-by-step" method, or "classical" method

Jacobi's transformation of symmetric matrix

For all real symmetric matrices Jacobi method is convergent and gives both eigenvalues and eigenvectors. It consists of a sequence of orthogonal similarity transformation, each of them - *Jacobi's rotation* - is just a plane rotation that annihilate one of the elements out of the first diagonal.

Referring to the paragraph "Factorization methods", this method gives us two matrices: **D** (eigenvalues) and **U** (eigenvectors), being:

$$\lim_{n \rightarrow \infty} A_{n-1} = \begin{bmatrix} \lambda_1 & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & \lambda_n \end{bmatrix} \quad \lim_{n \rightarrow \infty} P_1 P_2 \dots P_{n-1} P_n = U$$

Example - Solve the eigenproblem for the following 5x5 symmetric matrix

$$A = \begin{bmatrix} 9 & -26 & -14 & 36 & 24 \\ -26 & 14 & -4 & 46 & 14 \\ -14 & -4 & -6 & -6 & -54 \\ 36 & 46 & -6 & 19 & -4 \\ 24 & 14 & -54 & -4 & -11 \end{bmatrix}$$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1		Matrix						eigenvalues (Jacobi)						eigenvectors (jacobi)				
2		9	-26	-14	36	24		25	0	0	-0	0		0.6	0.4	-0	0.4	-0
3		-26	14	-4	46	14		0	-50	0	0	0		-0	0.4	0.6	0.4	-0
4		-14	-4	-6	-6	-54		0	0	50	-0	-0		0.4	0.6	0.4	-0	0.4
5		36	46	-6	19	-4		0	-0	0	75	0		0.4	-0	0.4	0.6	0.4
6		24	14	-54	-4	-11		-0	0	0	0	-75		-0	0.4	-0	0.4	0.6
7																		
8																		
9																		

{=MatEigenvalue_Jacobi(B2:F6)}

{=MatEigenvector_Jacobi(B2:F6)}

We can note the high clean of this method. Just plain and straight! By default, both functions use 100 iterations to reach this high accurate result. Sometime, for larger matrices, may be need to increase this limit or you have to accept less precision.

Tip. Jacobi's algorithm returns eigenvalues in the first diagonal. If you like to extract them in a vector, comes in handy the function **MatDiagExtr()**

	A	B	C	D	E	F	G
17							
18		eigenvalues (Jacobi)					
18		25	0	0	-0	0	25
19		0	-50	0	0	0	-50
20		0	0	50	-0	-0	50
21		0	-0	0	75	0	75
22		-0	0	0	0	-75	-75
23							
24							

{=MatDiagExtr(A18:E22)}

Example - Compute the first steps A1, A2, ... A6 of the Jacobi's algorithm and study the convergence of the previous example

Each step of the Jacobi's rotation method makes zero the two highest values out of the first diagonal. At next steps the zeros cannot be preserved but they are getting low step by step. The diagonalization error indicates this convergence, slow but inexorable, to zero

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
1	A						matrix at step: 1					1		matrix at step: 2						matrix at step: 3				
2	9	-26	-14	36	24		9	-26	-27	36	7.69		9	-44	-27	8.26	7.69		37.4	-0	-18	6.92	0.89	
3	-26	14	-4	46	14		-26	14	-13	46	7.36		-44	-30	-8	0	10.2		-0	-58	-21	4.51	12.7	
4	-14	-4	-6	-6	-54		-27	-13	45.6	-1.6	0		-27	-8	45.6	-9.8	0		-18	-21	45.6	-9.8	0	
5	36	46	-6	19	-4		36	46	-1.6	19	-7		8.26	0	-9.8	62.6	-0		6.92	4.51	-9.8	62.6	-0	
6	24	14	-54	-4	-11		7.69	7.36	0	-7	-63		7.69	10.2	0	-0	-63		0.89	12.7	0	-0	-63	
7							{=MatEigenvalue_Jacobi(A2:E6,H1)}						{=MatEigenvalue_Jacobi(A2:E6,P1)}						{=MatEigenvalue_Jacobi(A2:E6,V1)}					
8	Jacobi's rotation method.						diagonalization error:					17.1	diagonalization error:					11.4	diagonalization error:					7.42
9	Each step makes zero the two highest values out of the first diagonal.						matrix at step: 4						matrix at step: 5						matrix at step: 6					
10							37.4	-3.5	-18	6.92	0.89		24.9	-2.8	0	-0.4	-0.7		24.9	-2.8	-0.3	-0.3	-0.7	
11							-3.5	-62	-0	2.52	12.5		-2.8	-62	2.02	2.52	12.5		-2.8	-62	3.21	0.38	12.5	
12							-18	-0	49.8	-10	-2.5		-0	2.02	62.3	-13	-2.5		-0.3	3.21	49.9	-0	-1.8	
13							6.92	2.52	-10	62.6	-0		-0.4	2.52	-13	62.6	-0		-0.3	0.38	0	75	1.74	
14							0.89	12.5	-2.5	-0	-63		-0.7	12.5	-2.5	-0	-63		-0.7	12.5	-1.8	1.74	-63	
15							{=MatEigenvalue_Jacobi(A2:E6,J9)}						{=MatEigenvalue_Jacobi(A2:E6,P9)}						{=MatEigenvalue_Jacobi(A2:E6,V9)}					
16	M_DIAG_ERR(G10:K14)						diagonalization error:					5.69	diagonalization error:					3.61	diagonalization error:					2.38
17																								

For symmetric matrix the convergence always guaranteed. In our example at steps we have an average diagonalization error of about 0.01

	A	B	C	D	E	F	G	H	I	J	K	
1	A						matrix at step:				15	
2	9	-26	-14	36	24		25	0.009	-0	6E-17	0.002	
3	-26	14	-4	46	14		0.009	-50	-0	5E-04	0.067	
4	-14	-4	-6	-6	-54		-0	-0	50	-0.01	0.006	
5	36	46	-6	19	-4		1E-15	5E-04	-0.01	75	-0	
6	24	14	-54	-4	-11		0.002	0.067	0.006	-0	-75	
7							{=MatEigenvalue_Jacobi(A2:E6,H1)}					
8							diagonalization error:					0.01

is 15

Orthogonal matrices

The eigenvectors matrix returned by Jordan algorithm is *orthogonal* with each vector having Norma 1; that is, an *orthonormal* matrix

Indicating the scalar product with • symbol, the normal and orthogonal conditions are:

$$x_i \bullet x_j = \delta_{ij} = \begin{cases} 1 & \Rightarrow i = j \\ 0 & \Rightarrow i \neq j \end{cases}$$

In other words, the scalar product of a vector for itself must be 1; for any other different vector must be 0. (δ_{ij} is called Kroneker's symbol)

$$\mathbf{x}_{11} \bullet \mathbf{x}_{11} = |\mathbf{x}_{11}|^2 = 1$$

$$\mathbf{x}_{11} \bullet \mathbf{x}_{12} = \mathbf{x}_{11} \bullet \mathbf{x}_{13} = \mathbf{x}_{11} \bullet \mathbf{x}_{14} = 0$$

Orthogonal matrices have also other interesting features.

$$\text{If } \mathbf{U} \text{ is orthogonal, we have } \Leftrightarrow \mathbf{U}^{-1} = \mathbf{U}^T$$

$$\text{If } \mathbf{U} \text{ is also orthonormal; we have } \Rightarrow |\det(\mathbf{U})| = 1$$

Pay attention that the second statement is not invertible; that is, there are matrices with $\det = 1$ that are not orthogonal at all.

$$\det \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} = 1$$

The matrix at the left, for example, has $\det = 1$ (unitary) but is not orthogonal. Also all Tartaglia's matrices, seen in the previous chapters, have always $|\det|=1$ but they are never orthogonal.

Example - verify the orthogonality of the eigenvectors matrix of the above example

0.6	0.4	-0.4	0.4	-0.4
-0.4	0.4	0.6	0.4	-0.4
0.4	0.6	0.4	-0.4	0.4
0.4	-0.4	0.4	0.6	0.4
-0.4	0.4	-0.4	0.4	0.6

To verify, we can calculate the cross scalar product of each column with help of the **ProdScal()** function. But it will be tedious for large matrix. It is faster to use the identity $\mathbf{U} \mathbf{U}^T = \mathbf{I}$, as shown in the above worksheet.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	eigenvectors (jacobi)						horthonormalization verify						mop-up				
2	0.6	0.4	-0.4	0.4	-0.4		1	1E-16	-0	-0	1E-16		1	0	0	0	0
3	-0.4	0.4	0.6	0.4	-0.4		1E-16	1	-0	-0	1E-16		0	1	0	0	0
4	0.4	0.6	0.4	-0.4	0.4		-0	-0	1	1E-16	-0		0	0	1	0	0
5	0.4	-0.4	0.4	0.6	0.4		-0	-0	1E-16	1	-0		0	0	0	1	0
6	-0.4	0.4	-0.4	0.4	0.6		1E-16	1E-16	-0	-0	1		0	0	0	0	1
7							{=MMULT(A2:E6,M_TRANSP(A2:E6))}						{=MatMopUp(G2:K6)}				
8	1	-0															
9																	
10																	
11																	
12																	

Many times the matrix product generates the round-off error as in this case. We can sweep them with the **MatMopUp()** function

Eigenvalues with QR factorization method

Another popular algorithm to find all eigenvalues of a matrix is the *QR factorization method*. The heart is the following factorization of a matrix **A**:

$$\mathbf{A} = \mathbf{Q} \mathbf{R} \quad \text{where } \mathbf{Q} \text{ is orthonormal and } \mathbf{R} \text{ is triangular upper}$$

This factorization is always possible; you can easily make such factorization in Matrix.xla with the **Mat_QR()** function

The method follows the steps:

1. Factorize the given matrix $\mathbf{A} = \mathbf{Q} \mathbf{R}$
2. Multiply the two factors **R** and **Q** obtaining a new matrix $\mathbf{A}_1 = \mathbf{R} \mathbf{Q}$
3. Factorize the new matrix $\mathbf{A}_1 = \mathbf{Q} \mathbf{R}$ and repeat the step 2 an 3

We have the iterative process, starting with A:

$$\begin{array}{lll}
 \mathbf{A} = \mathbf{Q} \mathbf{R} & \Rightarrow & \mathbf{A}_1 = \mathbf{R} \mathbf{Q} \\
 \mathbf{A}_1 = \mathbf{Q}_1 \mathbf{R}_1 & \Rightarrow & \mathbf{A}_2 = \mathbf{R}_1 \mathbf{Q}_1 \\
 \mathbf{A}_2 = \mathbf{Q}_2 \mathbf{R}_2 & \Rightarrow & \mathbf{A}_3 = \mathbf{R}_2 \mathbf{Q}_2 \\
 \dots\dots\dots & & \dots\dots\dots \\
 \mathbf{A}_p = \mathbf{Q}_p \mathbf{R}_p & \Rightarrow & \mathbf{A}_{p+1} = \mathbf{R}_p \mathbf{Q}_p
 \end{array}$$

If the eigenvalues are all distinct in modulo: $|\lambda_1| > |\lambda_2| > |\lambda_3| > \dots > |\lambda_n|$ and **A** is symmetric; then the matrix **A_p** converges to diagonal form, where the elements are the eigenvalues of **A**

With the function **Mat_QR_iter** it is very easy to test how this process works.

Example - calculate the first 10 and 100 steps of the QR algorithm for the following symmetric matrix having the eigenvalues 1, 2, 3, 4, 5

2.6	1.2	0.8	0.4	0
1.2	2.8	0.4	0	-0.4
0.8	0.4	3	-0.4	-0.8
0.4	0	-0.4	3.2	-1.2
0	-0.4	-0.8	-1.2	3.4

We use the function of Matrix.xla **Mat_QR_iter()** for performing the first 10 step of the QR algorithm. The convergence at the diagonal form is evident and becomes more close with 100 iteration. Note the eigenvalues 1, 2, 3, 4, 5 appearing in the diagonal

	A	B	C	D	E	F	G	H	I	J	K
1	Matrix 5 x 5						iteration	10			
2	2.6	1.2	0.8	0.4	0		4.9885	0.0003	0.012	0.1061	4E-16
3	1.2	2.8	0.4	0	-0.4		0.0003	2	1E-06	1E-05	-1E-03
4	0.8	0.4	3	-0.4	-0.8		0.012	1E-06	3.0001	0.0006	-3E-05
5	0.4	0	-0.4	3.2	-1.2		0.1061	1E-05	0.0006	4.0114	-3E-06
6	0	-0.4	-0.8	-1.2	3.4		-1E-22	-1E-03	-3E-05	-3E-06	1
7											
8							iteration	100			
9	=Mat_QR_iter(A2:E6;H1)						5	-2E-10	-4E-16	1E-17	5E-16
10							-2E-10	4	2E-13	-8E-14	-1E-16
11							8E-23	2E-13	3	-2E-16	-2E-16
12							8E-24	-8E-14	7E-17	2	2E-16
13							9E-69	-3E-44	5E-47	-8E-31	1
14											

When the given matrix is not symmetric the method works the same; only the final matrix is triangular instead of diagonal. See the following example.

TUTORIAL FOR MATRIX.XLA

Example - calculate the first 10 and 100 steps of the QR algorithm for the following unsymmetric matrix having the eigenvalues 1, 2, 3, 4, 5

5	-3	-1	3	-7
7	-5	-1	9	-13
-4	4	3	-4	8
-1	1	0	3	2
-4	4	0	-4	9

We use the function of Matrix.xla **Mat_QR_iter()** for performing the first 10 step of the QR algorithm. The convergence at the triangular form is evident and becomes more close with 100 iteration. Note the eigenvalues 1, 2, 3, 4, 5 appearing in the diagonal

	A	B	C	D	E	F	G	H	I	J	K
1	Matrix 5 x 5						iteration	10			
2	5	-3	-1	3	-7		5.0013	-3.612	4.0641	4.1655	-22.47
3	7	-5	-1	9	-13		-0.019	2.0228	-0.52	1.1034	-1.727
4	-4	4	3	-4	8		0.0112	-0.013	3.026	-0.473	-1.425
5	-1	1	0	3	2		-0.03	0.0363	-0.07	3.95	-8.349
6	-4	4	0	-4	9		5E-08	-7E-08	1E-07	9E-08	1
7											
8							iteration	100			
9	=Mat_QR_iter(A2:E6;H1)						5	-0.192	6.1432	2.9294	-22.18
10							-8E-11	4	0.8081	1.2053	-7.612
11							3E-23	-9E-13	3	-0.192	-5.31
12							-6E-25	2E-14	1E-14	2	-1.134
13							7E-71	4E-60	4E-60	-6E-46	1
14											

Does the QR method always converge? There are cases - very rare indeed - where the algorithm fails. This happens for example when the eigenvalues are equal and opposite. Let's see this example

Example - The following matrix 3x3 has eigenvalues $\lambda_1 = 9$, $\lambda_2 = -9$, $\lambda_3 = 18$. Apply the QR method.

	A	B	C	D	E	F	G	H	I
1	Matrix 3 x 3			Eigenvalues (QR)			Eigenvalues (Jacobi)		
2	5	-8	-10	18	-4E-16	4E-16	18	0	1E-31
3	-8	11	-2	1E-43	8E-13	-9	9E-16	9	8E-22
4	-10	-2	2	-4E-44	-9	-8E-13	3E-16	7E-16	-9
5									
6	=MatEigenvalue_QR(A2:C4)						=MatEigenvalue_Jacobi(A2:C4)		
7									

In this simple case QR fails (we note the two -9 out of the diagonal). It was not able to find the two opposite eigenvalues $= \pm 9$, but it has found only the 18 one. Note that in the same condition Jacobi's algorithm has found exactly all the eigenvalues.

Real and complex eigenvalues with QR method

Starting from the simple QR method shown above, a more general QR algorithm was developed with important improvement - shifting for rapid convergence, Hessember reduction,

etc. The result is a very robust and efficient QR general algorithm⁷ being able to find complex and real eigenvalues of any real matrix.

This task can now be performed with the function **MatEigenvalue_QR** of matrix.xla

Example: find all eigenvalues of the given symmetric matrix

2.75	1.5	1.25	1	0.75	0.5	0.25	0
1.5	3.25	1	0.75	0.5	0.25	0	-0.25
1.25	1	3.75	0.5	0.25	0	-0.25	-0.5
1	0.75	0.5	4.25	0	-0.25	-0.5	-0.75
0.75	0.5	0.25	0	4.75	-0.5	-0.75	-1
0.5	0.25	0	-0.25	-0.5	5.25	-1	-1.25
0.25	0	-0.25	-0.5	-0.75	-1	5.75	-1.5
0	-0.25	-0.5	-0.75	-1	-1.25	-1.5	6.25

As previous shown, this matrix has the first 8 natural eigenvalues
1, 2, 3, 4, ... 8

We use the MatEigenvalue_QR to find all eigenvalues in a very straight way

	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											

{=MatEigenvalue_QR(A1:H8)}

The function can also return complex eigenvalues. Let's see this example

1	-0.5	0	0.5	0	0
0.5	5	2	1	0	-2
3.5	8.5	12	4.5	1	-7
0	4	2	2	0	-2
-7	-17	-16	-9	2	14
4.5	14.5	14	8.5	1	-9

This matrix has 2 real and 4 complex conjugate eigenvalues

⁷ Matrix.xla use the routine HQR and ELMHES derived from Fortran 77 EISPACK library

	A	B	C	D	E	F	G	H	I	J	K
1									λ re	λ im	
2	1	-0.5	0	0.5	0	0			2	2	
3	0.5	5	2	1	0	-2			2	-2	
4	3.5	8.5	12	4.5	1	-7			4	0	
5	0	4	2	2	0	-2			1	0.5	
6	-7	-17	-16	-9	2	14			1	-0.5	
7	4.5	14.5	14	8.5	1	-9			3	0	
8											
9											
10											

Formula Bar: `{=MatEigenvalue_QR(A2:F7)}`

Note how clean, easy and fast is the eigenvalues computation also in this case

How to find polynomial root with eigenvalues

In the previous example we have show how to compute eigenvalues by finding polynomial roots. Sometime it happens the contrary: we have to find polynomial roots by eigenvalues methods.

Example - Find all the roots of the given 4th degree polynomial

$$x^4 + 7x^3 - 41x^2 - 147x + 540$$

We need to get a matrix having its characteristic polynomial the given one. The *companion matrix* is what we need. It can be easily built by hand or - even better - by the function **MatCmpn()**

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

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

When we have the matrix, we can apply a method to find the eigenvalues. Being the matrix asymmetric, we choose the QR method.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Poly coef			Companion matrix					Eigenvalues = roots			
2	a0	540		0	0	0	-540		-5	0		
3	a1	-147		1	0	0	147		3	0		
4	a2	-41		0	1	0	41		-9	0		
5	a3	7		0	0	1	-7		4	0		
6	a4	1										
7												
8												
9												

Formula Bar 1: `{=MatCmpn(B2:B6)}`

Formula Bar 2: `{=MatEigenvalue_QR(D2:G5)}`

Eigenvalues are also the roots of the given polynomial.

This method is so robust and efficient that it is implemented in the rootfinder **Poly_Roots_QR** function of matrix.xla

Thanks to its efficiency, it is especially adapted for higher polynomials. Let's see this example

	A	B	C	D	E	F	G	H	I
1	Degree	Coefficients	Z re	Z im		Degree	Coefficients	Z re	Z im
2	a0	-39916800	1	0		a0	29988800	3	1
3	a1	120543840	2	0		a1	-6866160	3	-1
4	a2	-150917976	3	0		a2	7080548	3	1
5	a3	105258076	4	0		a3	-4321632	3	-1
6	a4	-45995730	5	0		a4	1725716	4	1
7	a5	13339535	6	0		a5	-470296	4	-1
8	a6	-2637558	7	0		a6	88445	5.99997	0
9	a7	357423	8	0		a7	-11318	6.00003	0
10	a8	-32670	9	0		a8	942	6.99998	0
11	a9	1925	10	0		a9	-46	7.00002	0
12	a10	-66	11	0		a10	1		
13	a11	1							
14			{=Poly_Roots_QR(B2:B13)}					{=Poly_Roots_QR(G2:G12)}	
15									

In the first 11th degree polynomial all roots are real. The second one of 10th degree has both complex and real roots with double multiplicity. In the first case the general accuracy is about 1E-9; in the second one is about 1E-6. Even in this difficult case the function returns a sufficient approximation of all the roots

It is the main advantage of this method: to have a good stability for all roots configurations avoiding the disastrous accuracy lackage characteristic of other rootfinder algorithms.

Powers' method

Powers' method can find the dominant⁸ real eigenvalue and its associate eigenvector of a real matrix. An ancient method, but still very popular, having same advantages:

- It is conceptually simple in its first proposition;
- It is robust;
- It works with both real symmetric and asymmetric matrices
- It has an important didactic meaning

With the matrix reduction method it can find iteratively all real eigenvalues and eigenvectors

But shall we begin to understand the heart of the algorithm:

We suppose a 3x3 matrix (for simplicity) with 3 independent eigenvectors \mathbf{x}_1 , \mathbf{x}_2 , \mathbf{x}_3 and a dominant eigenvalue λ_1 , being: $|\lambda_1| > |\lambda_2| > |\lambda_3|$

Taken an arbitrary vector \mathbf{v}_0 - called starting vector calculate the Rayleigh quotient (ratio) with the formulas:

$$\mathbf{v}_1 = A\mathbf{v}_0 \quad \Rightarrow \quad r = \frac{\mathbf{v}_0^T \mathbf{v}_1}{\mathbf{v}_0^T \mathbf{v}_0}$$

Iterating, we have:

$$\mathbf{v}_2 = A\mathbf{v}_1 \quad \Rightarrow \quad r = \frac{\mathbf{v}_1^T \mathbf{v}_2}{\mathbf{v}_1^T \mathbf{v}_1} \quad \dots\dots\dots \quad \mathbf{v}_{n+1} = A\mathbf{v}_n \quad \Rightarrow \quad r = \frac{\mathbf{v}_n^T \mathbf{v}_{n+1}}{\mathbf{v}_n^T \mathbf{v}_n}$$

Under certain conditions, the ratio converges to the dominant eigenvalue for $n \gg 1$ and the associate eigenvector can be obtained by the formulas:

$$\lim_{n \rightarrow \infty} r = \lambda_1 \quad \Rightarrow \quad \lim_{n \rightarrow \infty} \mathbf{v}_n (\lambda_1)^{-n} = \mathbf{x}_1$$

We shall see how it works in a practical case

Example - Analyze the convergence of the power's method for the following matrix

-1	2	-2
-2	-6	3
-2	-4	1

The matrix has three separate eigenvalues:

$$\lambda_1 = -3, \lambda_2 = -2, \lambda_3 = -1$$

Let's see how to arrange the worksheet. First of all, insert the formulas as indicated to the left; then, select the appropriate range and drag to the right to iterate the formulas.

Assume the starting vector to be $\mathbf{v}_0 = (1, 0, 0)$

⁸ The eigenvalue that has the highest absolute value is the "dominant eigenvalue"

TUTORIAL FOR MATRIX.XLA

	A	B	C	D	E	F
1		A		v0	v1	
2	5	6	6	1	5	
3	-12	-22	-28	0	-12	
4	10	20	26	0	10	
5	{=MMULT(\$A\$2:\$C\$4,D2:D4)}					
6			r =		5	
7	{=ProdScal(D2:D4,E2:E4)/ProdScal(D2:D4,D2:D4)}					
8					x1	
9					1	
10	{=E2:E4/E7*E14}				-2.4	
11					2	
12						
13			n =	0	1	

	A	B	C	D	E
1		A		v0	v1
2	5	6	6	1	5
3	-12	-22	-28	0	-12
4	10	20	26	0	10
5					
6			λ =		5
7					
8					x1
9					1
10					-2.4
11					2
12					
13			n =	0	1
14					
15					

Insert the formulas in the column E

Select the range E1:E13 and drag to right

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		A		v0	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
2	-1	2	-2	1	-1	1	-1	1	-1	1	-1	1	-1	1
3	-2	-6	3	0	-2	8	-26	80	-242	728	-2186	6560	-19682	59048
4	-2	-4	1	0	-2	8	-26	80	-242	728	-2186	6560	-19682	59048
5														
6			r =		-1	-3.667	-3.233	-3.075	-3.025	-3.008	-3.003	-3.001	-3.0003	-3
7														
8					x1	x2	x3	x4	x5	x6	x7	x8	x9	x10
9					1	0.074	0.03	0.011	0.004	0.001	5E-04	2E-04	5E-05	2E-05
10					2	0.595	0.77	0.894	0.956	0.982	0.993	0.997	0.999	1
11					2	0.595	0.77	0.894	0.956	0.982	0.993	0.997	0.999	1
12														
13			n =	0	1	2	3	4	5	6	7	8	9	10

As we can observe, the convergence to dominant eigenvalue $\lambda_1 = -3$ and its associate eigenvector $x = (0, 1, 1)$ is slow but evident.

Rescaling. We note also a first drawback of this method; the values of vector v become larger step after step. This could cause an overflow error for higher steps. To avoid this, the algorithm is modified inserting a vector rescaling routine after a fixed amount of steps.

v9	v10		v9	v10
-1	1	\Rightarrow	-1E-04	1E-04
-19682	59048	rescaling	-1.968	5.905
-19682	59048	dividing for 10000	-1.968	5.905

The value of the rescaling factor is not very important; only the magnitude is the main thing.

Note also that the Rayleigh's ratio is not affected by rescaling

Finding non-dominant eigenvalues. Once the dominant eigenvalue λ_1 and its associate eigenvector \mathbf{x}_1 are found, we may want to continue to compute the eigenvalues remaining. Compute the normalized of \mathbf{x} and the new matrix \mathbf{A}_1 :

$$\mathbf{u}_1 = \mathbf{x}_1 / |\mathbf{x}_1| \quad \Rightarrow \quad \mathbf{A}_1 = \mathbf{A} - \lambda_1 \mathbf{u} \mathbf{u}^T$$

The matrix \mathbf{A}_1 has eigenvalues: 0, λ_2 , λ_3 . Now, the dominant eigenvalues of \mathbf{A}_1 is λ_2 . Therefore we can apply the power's method once more.

Example - reduce the matrix \mathbf{A} of the previous example with the eigenvalue $\lambda_1 = -3$ and eigenvector $\mathbf{x}_1 = (0, 1, 1)$. Repeat the power's method to find the dominant eigenvector λ_2

	A	B	C	D	E	F	G	H	I	J	K
1		A				u u^T				A1	
2	-1	2	-2		0	0	0		-1	2	-2
3	-2	-6	3		0	0.5	0.5		-2	-4.5	4.5
4	-2	-4	1		0	0.5	0.5		-2	-2.5	2.5
5											
6	λ_1	x1	u1								
7	-3	0	0								
8		1	0.707								
9		1	0.707								

Formulas shown in the image:

- $\{=MMULT(C7:C9,m_transp(C7:C9))\}$ (for $\mathbf{u u}^T$)
- $\{=B7:B9/M_ABS(B7:B9)\}$ (for \mathbf{u}_1)
- $\{=A2:C4-A7*E2:G4\}$ (for \mathbf{A}_1)

The matrix \mathbf{A}_1 is the new reduced matrix. It should have all the eigenvalues of the original matrix \mathbf{A} , except λ_1 . Let's see. Repeating the power method we will find its dominant eigenvalues. Choosing (0, 1, 0) for starting vector, we have something like this:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		A		v0	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
2	-1	2	-2	0	2	-6	14	-30	62	-126	254	-510	1022	-2046
3	-2	-4.5	4.5	1	-4.5	5	-6	8	-12	20	-36	68	-132	260
4	-2	-2.5	2.5	0	-2.5	1	2	-8	20	-44	92	-188	380	-764
5														
6			r =		-4.5	-1.213	-1.806	-2.051	-2.058	-2.036	-2.019	-2.01	-2.005	-2.003
7														
8					x1	x2	x3	x4	x5	x6	x7	x8	x9	x10
9					-0.444	-4.077	-2.375	-1.696	-1.678	-1.771	-1.857	-1.915	-1.9517	-1.973
10					1	3.398	1.018	0.452	0.325	0.281	0.263	0.255	0.2521	0.251
11					0.556	0.68	-0.339	-0.452	-0.541	-0.619	-0.672	-0.706	-0.7257	-0.737
12														
13			n =	0	1	2	3	4	5	6	7	8	9	10

As we can observe, the convergence to dominant eigenvalue $\lambda_2 = -2$ and its associate eigenvector $\mathbf{x} = (-2, 0.25, -0.75)$ is slow but evident. After 25 steps the error is less than about $1\text{E}-6$

The process *Power's method + matrix reduction* can be iterated for all eigenvalues. We have to pay attention that since the eigenvalues computing is approximated, some error will be introduced in the next iterate step; the last eigenvalue could be affected by a considerable round-off error. In general, the matrix reduction (or matrix deflation) method becomes more inaccurate as we calculate more eigenvalues, because round-off error is introduced in each result and it accumulates itself as the process continues.

Does the power's method always converge? Although it has worked well in the above examples, we must say that there are cases in which the method may fail. There are basically three cases:

- The matrix **A** is not diagonalizable; that means that has n linearly independent eigenvectors. Simple, but, of course, it is not easy to tell by just looking at **A** how many eigenvectors there are.
- The matrix **A** has complex eigenvalues
- The matrix **A** does not have a very dominant eigenvalue. In that case the convergence is so slow that often the max iteration limit has exceeded

Eigensystems with power's method

In Matrix.xla the power method is implemented by two main functions:

- MatEigenvector_pow() returns all eigenvectors
- MatEigenvalues_pow() returns all eigenvalues

Just simple and straight. Let's see

Example - solve the eigenproblem for the following symmetric matrix

2.6	1.2	0.8	0.4	0
1.2	2.8	0.4	0	-0.4
0.8	0.4	3	-0.4	-0.8
0.4	0	-0.4	3.2	-1.2
0	-0.4	-0.8	-1.2	3.4

	A	B	C	D	E	F	G	H	I	J	K
1	A (5 x 5)					eigenvectors (power)					eigenvalues (power)
2	2.6	1.2	0.8	0.4	0	-0.6667	-0.6667	-0.6667	-0.6667	1	5
3	1.2	2.8	0.4	0	-0.4	-0.6667	-0.6667	-0.6667	1	-0.6667	4
4	0.8	0.4	3	-0.4	-0.8	-0.6667	-0.6667	1	-0.6667	-0.6667	3
5	0.4	0	-0.4	3.2	-1.2	-0.6667	1	-0.6667	-0.6667	-0.6667	2
6	0	-0.4	-0.8	-1.2	3.4	1	-0.6667	-0.6667	-0.6667	-0.6667	1
7											
8											
9											

The function MatEigenvector_pow() has a second parameter: *Norm*. If TRUE, the function returns all normalized eigenvectors | (default FALSE)

TUTORIAL FOR MATRIX.XLA

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	A (5 x 5)					U eigenvectors (power)					orthogonality test $I = U U^T$				
2	2.6	1.2	0.8	0.4	0	-0.4	-0.4	-0.4	-0.4	0.6	1	0	0	0	0
3	1.2	2.8	0.4	0	-0.4	-0.4	-0.4	-0.4	0.6	-0.4	0	1	0	0	0
4	0.8	0.4	3	-0.4	-0.8	-0.4	-0.4	0.6	-0.4	-0.4	0	0	1	0	-0
5	0.4	0	-0.4	3.2	-1.2	-0.4	0.6	-0.4	-0.4	-0.4	0	0	0	1	0
6	0	-0.4	-0.8	-1.2	3.4	0.6	-0.4	-0.4	-0.4	-0.4	0	0	-0	0	1
7															
8						{=MatEigenvector_pow(A2:E6, TRUE)}					{=MMULT(F2:J6,m_transp(F2:J6))}				
9															

Because of the symmetry, the eigenvector matrix **U** is also orthogonal. To prove it, simple check the relation $I = U U^T$ as shown it the above worksheet.

Convergence. Why there is no more the starting vector in these functions? Well, this algorithm is started with a random generic vector. Many times it converges, but some times not. So if one of these functions returns the error "limit iterations exceeded", do not worry. Simply, re-try it or try to increase the parameter *ITERMAX* (default 1000).

Example: solve the eigenproblem for the following asymmetric 6x6 matrix.

-62	-65	-121	-41	95	26
-43	-40	-77	-13	40	-98
17	17	28	-13	-23	88
16	16	32	25	-22	-64
-26	-26	-52	-26	38	26
-28	-28	-56	-28	28	13

This matrix has eigenvalues -1, 3, -6, 9, 12, -15
 Power's method can works also for asymmetric matrix.
 In this case we have left the round-off errors to give an idea of the general accuracy.
 Eigenvalues errors are shown in the last column.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	A (6x6)						Eigenvector						eigen values error	
2	-62	-65	-121	-41	95	26	0.286	-0.25	-0.5	1	1	0.0385	-15	5.3E-15
3	-43	-40	-77	-13	40	-98	1	1	-1	1	-1	1	12	1.1E-14
4	17	17	28	-13	-23	88	-0.714	-0.75	1	-1	-2E-14	-0.731	9	3.6E-14
5	16	16	32	25	-22	-64	0.286	0.5	-0.5	-2E-14	1E-14	0.3846	-6	7.1E-15
6	-26	-26	-52	-26	38	26	2E-15	-0.25	-2E-14	6E-15	-5E-15	-0.077	3	0
7	-28	-28	-56	-28	28	13	0.143	1E-15	-7E-16	-2E-15	-1E-15	0.0769	-1	1.8E-13
8							{=MatEigenvector_pow(A2:F7)}					{=MatEigenvalue_pow(A2:F7)}		
9														

How to validate an eigensystem

Example - Check the eigensystem of the previous example

In order to test an eigenvector matrix **U** of a given matrix **A**, we can use the definition

$$\mathbf{A} \mathbf{U} = (\lambda_1 \mathbf{u}_1, \lambda_2 \mathbf{u}_2, \dots, \lambda_6 \mathbf{u}_6)$$

But before testing, we show how to arrange the eigenvector matrix for avoiding decimals. This is not essential, but it helps the visual inspection.

First of all we shall begin with eliminating round off error by the MatMopUp() function with ErrMin = 1E-13

	A	B	C	D	E	F	G	H	I	J	K	L
1	Eigenvectors						Eigenvectors (mop-up)					
2	0.28571	-0.25	-0.5	1	1	0.03846	0.28571	-0.25	-0.5	1	1	0.03846
3	1	1	-1	1	-1	1	1	1	-1	1	-1	1
4	-0.7143	-0.75	1	-1	-2E-14	-0.7308	-0.7143	-0.75	1	-1	0	-0.7308
5	0.28571	0.5	-0.5	-2E-14	1.2E-14	0.38462	0.28571	0.5	-0.5	0	0	0.38462
6	1.8E-15	-0.25	-2E-14	6.1E-15	-5E-15	-0.0769	0	-0.25	0	0	0	-0.0769
7	0.14286	9.8E-16	-7E-16	-2E-15	-1E-15	0.07692	0.14286	0	0	0	0	0.07692
8												
9												
10												

Now, for each column, we choose the *pivot*, that is, the absolute minimum value, except the zero.

	G	H	I	J	K	L	M	N	O	P	Q	R
1	Eigenvectors (mop-up)						Integer eigenvectors					
2	0.2857	-0.25	-0.5	1	1	0.0385	2	1	1	1	1	1
3	1	1	-1	1	-1	1	7	-4	2	1	-1	26
4	-0.714	-0.75	1	-1	0	-0.731	-5	3	-2	-1	0	-19
5	0.2857	0.5	-0.5	0	0	0.3846	2	-2	1	0	0	10
6	0	-0.25	0	0	0	-0.077	0	1	0	0	0	-2
7	0.1429	0	0	0	0	0.0769	1	0	0	0	0	2
8	Pivot											
9	0.1429	-0.25	-0.5	1	1	0.0385						
10												

Multiply each pivot for the corresponding eigenvector we obtain a new integer vector that it is still an eigenvector

	A	B	C	D	E	F	G	H	I	J	K	L
1							eigenvectors					
2	A (6x6)						u1	u2	u3	u4	u5	u6
3	-62	-65	-121	-41	95	26	2	1	1	1	1	1
4	-43	-40	-77	-13	40	-98	7	-4	2	1	-1	26
5	17	17	28	-13	-23	88	-5	3	-2	-1	0	-19
6	16	16	32	25	-22	-64	2	-2	1	0	0	10
7	-26	-26	-52	-26	38	26	0	1	0	0	0	-2
8	-28	-28	-56	-28	28	13	1	0	0	0	0	2
9							eigenvalues					
10							λ1	λ2	λ3	λ4	λ5	λ6
11	AU						-15	12	9	-6	3	-1
12	-30	12	9	-6	3	-1	-30	12	9	-6	3	-1
13	-105	-48	18	-6	-3	-26	-105	-48	18	-6	-3	-26
14	75	36	-18	6	0	19	75	36	-18	6	0	19
15	-30	-24	9	0	0	-10	-30	-24	9	0	0	-10
16	-0	12	-0	-0	-0	2	0	12	0	0	0	2
17	-15	0	-0	-0	-0	-2	-15	0	0	0	0	-2
18	{=MMULT(A3:F8,G3:L8)}						{=G8:H8*G11}					
19							{=H3:H8*H11}					

The matrix on the left is obtained by multiplying the original matrix for its eigenvectors matrix: **A U**.

The matrix on the right is obtained by multiplying each eigenvectors \mathbf{u}_i for its corresponding eigenvalues.

Because of the two matrices are identical, the eigensystem (eigenvectors + eigenvalues) is correct.

How to generate a random symmetric matrix with given eigenvalues

Many time, in order to test algorithm, we need symmetric a matrix with known eigenvalues. For building this test matrix, the following simple method can be useful.

- First, we generate a random $(n \times 1)$ vectors, \mathbf{v}
- Then we generate the *Householder* matrix \mathbf{H} with the vector \mathbf{V}
- We create a diagonal matrix \mathbf{D} ($n \times n$) with the eigenvalues that we want to obtain.
- Finally we make a *Similarity Transformation* of matrix \mathbf{D} by the matrix \mathbf{W} .

The result is a symmetric matrix with the given eigenvalues.

Example: Suppose we want a (3×3) random symmetric matrix with eigenvalues = (1, 2, 4). We chose a random vector \mathbf{v} , like for example:

$$\mathbf{v} = \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}$$

Then we create the associate *Householder* matrix \mathbf{H}

$$\mathbf{H} = \mathbf{I} - 2 \frac{\mathbf{v} \cdot \mathbf{v}^T}{\|\mathbf{v}\|^2} = \begin{bmatrix} -1/3 & -2/3 & 2/3 \\ -2/3 & 2/3 & 1/3 \\ 2/3 & 1/3 & 2/3 \end{bmatrix}$$

We create the diagonal matrix \mathbf{D}

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

Then, we make the *Similarity Transform* of \mathbf{D} by \mathbf{H}

$$\mathbf{A} = \mathbf{H}^{-1} \mathbf{D} \mathbf{H} = \begin{bmatrix} 25/9 & 2/9 & 10/9 \\ 2/9 & 16/9 & 8/9 \\ 10/9 & 8/9 & 22/9 \end{bmatrix}$$

Note that, in this case, the inverse of \mathbf{H} is the same of \mathbf{H} .

The result matrix \mathbf{A} has the wanted eigenvalues = (1, 2, 4)

If we want to avoid fractional numbers we can multiply the matrix \mathbf{A} for 9 and we get a new symmetric matrix \mathbf{B}

$$\mathbf{B} = 9 \cdot \mathbf{A} = \begin{bmatrix} 25 & 2 & 10 \\ 2 & 16 & 8 \\ 10 & 8 & 22 \end{bmatrix}$$

The eigenvalues of \mathbf{B} are now multiply for 9; thus 9, 18, 36

As we can see this method is general and can be very useful in many cases: for testing algorithm, formulas, subroutine, etc.

In the addin Matrix.xla, there are functions for generating Householder matrices and performing Similarity Transform.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Random Symmetric matrix with given eigenvalues												
2	V			D			H				A		
3	2		1	0	0	-0.333	-0.667	0.667			2.778	0.222	1.111
4	1		0	2	0	-0.667	0.667	0.333			0.222	1.778	0.889
5	-1		0	0	4	0.667	0.333	0.667			1.111	0.889	2.444
6													
7													
8													
9													
10													

Seed: random vector

eigenvalues setting

{=Mat_Householder(A3:A5)}

{=M_BAB(C3:E5;F3:H5)}

All this process is performed by the function MatRndEigSym()

Eigenvalues of tridiagonal matrix

Tridiagonal matrices are very common in practical numerical computation. These matrices can be worked with all methods show before, but there are specialized dedicated algorithms more efficient and faster to solve eigenvalues problem. We have to consider that many times problem involving tridiagonal matrices have a quite larger dimension. Also the storage of a tridiagonal matrix should be made with suitable cure. A general full matrix 30 x 30 requires 900 cells, but for a tridiagonal one with the same dimension we need to store only 90 cells, saving more than 90%. Clearly a particularly attention is quite suitable.

In matrix.xla there are the following specialized functions:

- MatEigenvalue_QL finds all real eigenvalues with the QL algorithm
- MatEigenvector3 computes the eigenvector of a real eigenvalue
- MatEigenvalue_TridUni finds all eigenvalues for a uniform tridiagonal matrix

All these function accept the matrix in standard form (n x n) or in compact form (n x 3)

15 x 15 compact form			15 x 15 tridiagonal matrix in standard form														
0	5	-0.5	5	-0.5	0	0	0	0	0	0	0	0	0	0	0	0	0
-2	5	-1	-2	5	-1	0	0	0	0	0	0	0	0	0	0	0	0
-1	5	-1	0	-1	5	-1	0	0	0	0	0	0	0	0	0	0	0
-1	5	-2	0	0	-1	5	-2	0	0	0	0	0	0	0	0	0	0
-1	4	-0.5	0	0	0	-1	4	-0.5	0	0	0	0	0	0	0	0	0
-1	3	-1	0	0	0	0	-1	3	-1	0	0	0	0	0	0	0	0
-5	-5	-1	0	0	0	0	0	-5	-5	-1	0	0	0	0	0	0	0
-0.5	6	-1	0	0	0	0	0	-0.5	6	-1	0	0	0	0	0	0	0
-1	5	-1	0	0	0	0	0	0	-1	5	-1	0	0	0	0	0	0
-0.5	9	-1	0	0	0	0	0	0	-0.5	9	-1	0	0	0	0	0	0
-1	2	-1	0	0	0	0	0	0	0	-1	2	-1	0	0	0	0	0
-1	-9	-1	0	0	0	0	0	0	0	0	-1	-9	-1	0	0	0	0
-1	10	-1	0	0	0	0	0	0	0	0	0	-1	10	-1	0	0	0
-2	-4	-1	0	0	0	0	0	0	0	0	0	0	-2	-4	-1	0	0
-1	-8	0	0	0	0	0	0	0	0	0	0	0	0	-1	-8	0	0

In the compact form we store only the diagonal and sub-diagonals values in the same columns
For tridiagonal matrices there are several useful lemmas that help us to discover the kind of eigenvalues

TUTORIAL FOR MATRIX.XLA

One rule says that:

If each “perpendicular couple” of elements have the same sign, than the matrix has are all real eigenvalues
(The condition is sufficient.)

So we can apply the fast QL algorithm to calculate all 15 eigenvalues of the given matrix

5	-0.5	0	0	0	0
-2	5	-1	0	0	0
0	-1	5	-1	0	0
0	0	-1	5	-2	0
0	0	0	-1	4	-0.5
0	0	0	0	-1	3

In the following sheet we have compute all eigenvalues and the first 4 eigenvectors with a very good approximation (about 1E-14)

	A	B	C	D	E	F	G	H	I	J
1	15 x 15 compact form				Eigenvalues		v1	v2	v3	v4
2	0	5	-0.5		6.028785629		1259874	460586	-1224.81	-599424
3	-2	5	-1		6.787165094		-2592281	-1646287	3932.92	-147623
4	-1	5	-1		6.605526487		147153	2021015	-3864.79	1180671
5	-1	5	-2		4.876862645		2440892	-1965600	2272.11	293007.8
6	-1	4	-0.5		3.897733747		-1329154	745918	108.432	-572295
7	-1	3	-1		4.333252289		511352.2	-226795	-5109.26	417633.3
8	-5	-5	-1		3.443766566		-219622.5	112992	18313.1	-211545
9	-0.5	6	-1		2.533039289		-134591.8	-197878	-186987	1234.356
10	-1	5	-1		1.944973107		113685.6	99266.9	104069	107158.9
11	-0.5	9	-1		9.265043844		17633.7	20471.8	19901.4	11960.9
12	-1	2	-1		10.19392847		-4449.258	-4332.68	-4381.1	-4262.98
13	-1	-9	-1		-5.62654539		291.4016	269.423	275.809	303.1183
14	-1	10	-1		-3.898173049		69.84584	79.2558	76.9496	56.65307
15	-2	-4	-1		-9.14373019		-14.02879	-14.7872	-14.6055	-12.8769
16	-1	-8	0		-8.24162854		1	1	1	1
17										
18	{=MatEigenvalue_QL(A2:C16)}						{=MatEigenvector3(A2:C16;E2:E5)}			
19										

Note that the eigenvector returned by MatEigenvector3 is not normalized. Use for this task the MatNormalize function.

Eigenvalues of tridiagonal uniform matrix

In numeric calculus is common to encounter symmetric, tridiagonal, uniform matrices like the following. For this kind, there is a nice close formula giving all eigenvalues for any size of the matrix dimension.

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

If the symmetric matrix has $n \times n$ dimension, eigenvalues are:

$$\lambda_k = a + 2b \cdot \cos\left(\frac{k\pi}{n+1}\right)$$

where $k = 1, 2, \dots, n$

We can do the following observations:

- All eigenvalues are real and distinct being the matrix symmetric
- All eigenvalues are symmetric around the point "a"
- For n odd exists the trivial eigenvalues $\lambda = a$
- All roots lie into the interval $a - 2b < \lambda_k < a + 2b$

Also the eigenvectors matrix can be written in a closed compact form.

$$U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ u_{21} & u_{22} & \dots & u_{2n} \\ \dots & \dots & \dots & \dots \\ u_{n1} & u_{n2} & \dots & u_{nn} \end{bmatrix}$$

If the symmetric matrix has $n \times n$ dimension, the elements of the eigenvectors matrix are:

$$u_{ik} = \sin\left(i \cdot k \frac{\pi}{n+1}\right)$$

Where $i = 1, 2, \dots, n$, $k = 1, 2, \dots, n$

The unsymmetrical tridiagonal uniform case can be led back to the above one.

We distinguish two cases:

1) The sub-diagonals have the same sign. In that case we can demonstrate that all roots are real and distinct.

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

If the matrix has $n \times n$ dimension, and $bc > 0$, the eigenvalues are:

$$\lambda_k = a + 2\sqrt{bc} \cdot \cos\left(\frac{k\pi}{n+1}\right)$$

where $k = 1, 2, \dots, n$

All roots lie on the interval:

$$a - 2\sqrt{bc} < \lambda_k < a + 2\sqrt{bc}$$

2) The sub-diagonals have different sign. In that case we can demonstrate that all root are complex conjugate for n even; for n odd exists only one real root $\lambda = a$.

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

If the matrix has $n \times n$ dimension, and $bc < 0$, the eigenvalues are complex:

$$\lambda_k = a + i \cdot 2\sqrt{-bc} \cdot \cos\left(\frac{k\pi}{n+1}\right) = a + i\delta_k$$

where $k = 1, 2, \dots, n$

All roots lie on the segment:

$$re(\lambda_k) = a \quad -2\sqrt{-bc} < im(\lambda_k) < 2\sqrt{-bc}$$

Eigenvectors can be computed by the following iterative algorithm

$$x_k = \lambda_k - a$$

Where : $k = 1, 2, \dots, n$, $i = 1, 2, \dots, n$

$$u_{ik} = \frac{1}{b} (x_k \cdot u_{(i-1)k} - c \cdot u_{(i-2)k})$$

$$u_{1k} = 1, \quad u_{2k} = \frac{1}{b} x_k$$

Example

Find all eigenvalues of the following tridiagonal uniform 8 x 8 matrix

10	1	0	0	0	0	0	0
4	10	1	0	0	0	0	0
0	4	10	1	0	0	0	0
0	0	4	10	1	0	0	0
0	0	0	4	10	1	0	0
0	0	0	0	4	10	1	0
0	0	0	0	0	4	10	1
0	0	0	0	0	0	4	10

We observe that the subdiagonal values have the same sign so all eigenvalues are real and distinct. They can be obtained by the following close formula:

$$\lambda_k = a + 2\sqrt{bc} \cdot \cos\left(\frac{k\pi}{n+1}\right)$$

for $k = 1, 2, \dots, 8$ and where $a = 10, b = 1, c = 4, n = 8$

Giving the following 8 eigenvalues

λ_1	13.7587704831436
λ_2	13.0641777724759
λ_3	12
λ_4	10.6945927106677
λ_5	9.30540728933228
λ_6	8
λ_7	6.93582222752409
λ_8	6.24122951685637

All eigenvalues are contained into the interval $(a - 4, a + 4) = (6, 14)$

Example

Find all eigenvalues of the following tridiagonal uniform 7 x 7 matrix

10	2	0	0	0	0	0
-1	10	2	0	0	0	0
0	-1	10	2	0	0	0
0	0	-1	10	2	0	0
0	0	0	-1	10	2	0
0	0	0	0	-1	10	2
0	0	0	0	0	-1	10

We observe that the subdiagonal values have different sign and the dimension n is odd, then all eigenvalues are complex conjugate except only one real trivial root $\lambda = 10$.

They can be obtained by the following close formula:

$$\lambda_k = a + i \cdot 2\sqrt{-bc} \cdot \cos\left(\frac{k\pi}{n+1}\right) = a + i\delta_k$$

for $k = 1, 2, \dots, 7$ and where $a = 10$, $b = 2$, $c = -1$, $n = 7$

Giving the following 7 eigenvalues.

	real	imm
λ_1	10	2.6131259297528
λ_2	10	2
λ_3	10	1.0823922002924
λ_4	10	0
λ_5	10	-1.0823922002924
λ_6	10	-2
λ_7	10	-2.6131259297528

Example

Find all eigenvalues of the following tridiagonal uniform 8 x 8 matrix

1	1	0	0	0	0	0	0
-1	1	1	0	0	0	0	0
0	-1	1	1	0	0	0	0
0	0	-1	1	1	0	0	0
0	0	0	-1	1	1	0	0
0	0	0	0	-1	1	1	0
0	0	0	0	0	-1	1	1
0	0	0	0	0	0	-1	1

We observe that the subdiagonal values have different sign and the dimension n is even, then no real eigenvalues exist and all eigenvalues are complex conjugate.

They can be obtained by the following close formula:

$$\lambda_k = a + i \cdot 2\sqrt{-bc} \cdot \cos\left(\frac{k\pi}{n+1}\right) = a + i\delta_k$$

for $k = 1, 2, \dots, 8$ and where $a = 1$, $b = 1$, $c = -1$, $n = 8$

Giving the following 8 eigenvalues.

	real	imm
λ_1	1	1.8793852415718
λ_2	1	1.5320888862380
λ_3	1	1
λ_4	1	0.3472963553339
λ_5	1	-0.3472963553339
λ_6	1	-1
λ_7	1	-1.5320888862380
λ_8	1	-1.8793852415718

Example

Find all eigenvalues of the following tridiagonal uniform 8 x 8 matrix

-2	1	0	0	0	0	0	0
1	-2	1	0	0	0	0	0
0	1	-2	1	0	0	0	0
0	0	1	-2	1	0	0	0
0	0	0	1	-2	1	0	0
0	0	0	0	1	-2	1	0
0	0	0	0	0	1	-2	1
0	0	0	0	0	0	1	-2

We observe that the matrix is symmetric so all eigenvalues are real and distinct. They can be obtained by the following close formula:

$$\lambda_k = a + 2b \cdot \cos\left(\frac{k\pi}{n+1}\right)$$

for $k = 1, 2, \dots, 8$ and where $a = -2$, $b = 1$, $c = 1$, $n = 8$

Giving the following 8 eigenvalues

λ_1	-0.1206147584282
λ_2	-0.4679111137620
λ_3	-1
λ_4	-1.6527036446661
λ_5	-2.34729635533386
λ_6	-3
λ_7	-3.53208888623796
λ_8	-3.87938524157182

All eigenvalues are contained into the interval $(a - 2, a + 2) = (-4, 0)$

We observe that they are all negative

The eigenvectors matrix can be obtained in a very fast way using the formula

$$u_{ij} = \sin\left(i \cdot j \cdot \frac{\pi}{n+1}\right) \quad U = \begin{bmatrix} \sin(\alpha) & \sin(2\alpha) & \dots & \sin(8\alpha) \\ \sin(2\alpha) & \sin(4\alpha) & \dots & \sin(16\alpha) \\ \dots & \dots & \dots & \dots \\ \sin(8\alpha) & \sin(16\alpha) & \dots & \sin(64\alpha) \end{bmatrix}$$

TUTORIAL FOR MATRIX.XLA

That gives the following approximate eigenvectors' matrix

0.34202	0.64279	0.86603	0.98481	0.98481	0.86603	0.64279	0.34202
0.64279	0.98481	0.86603	0.34202	-0.34202	-0.86603	-0.98481	-0.64279
0.86603	0.86603	0	-0.86603	-0.86603	0	0.86603	0.86603
0.98481	0.34202	-0.86603	-0.64279	0.64279	0.86603	-0.34202	-0.98481
0.98481	-0.34202	-0.86603	0.64279	0.64279	-0.86603	-0.34202	0.98481
0.86603	-0.86603	0	0.86603	-0.86603	0	0.86603	-0.86603
0.64279	-0.98481	0.86603	-0.34202	-0.34202	0.86603	-0.98481	0.64279
0.34202	-0.64279	0.86603	-0.98481	0.98481	-0.86603	0.64279	-0.34202

Note that the column-vectors are orthogonal.

Why so many different methods?

Well, many times we have heard this question. The fact is that numerical methods can be regarded as tools for solving specific problems. Eigenproblems can lead to very large different solutions and, we must say that, they are one of the most difficult aspects of the numerical calculus. So we need several tools to succeed in solving them. "How many screwdriver do you have?" More than one, surely. So we have not to be surprised for several different eigensystem methods. Sometime we will use one and sometime another.

Look at this example

Example - Find the eigenvalues of the following symmetric 3x3 matrix

5	-8	-10
-8	11	-2
-10	-2	2

We shall use same methods that we have studied:

- Characteristic polynomial
- Jacobi's method
- QR factorization (simple method)
- Power method

	A	B	C	D	E	F	G	H	I	J	K
1	A (3x3)				coeff roots				Eigenvalues (power)		
2	5	-8	-10		-1458	re	im		<div>18</div> <div>-3.332</div> <div>4.061</div>		
3	-8	11	-2		81	-9	0				
4	-10	-2	2		18	9	0				
5					-1	18	0				
6					Eigenvalues (Jacobi)				Eigenvalues (QR)		
7					18	0	1E-31		18	-4E-16	4E-16
8					9E-16	9	8E-22		1E-43	8E-13	-9
9					3E-16	7E-16	-9		-4E-44	-9	-8E-13
10					<div>right !</div>				<div>wrong !</div>		
11											
12											
13											

As we can see, two methods - Jacobi's method and characteristic polynomial - give us the correct solution, but the two others fail. In that case, the reason is the two eigenvalues sub dominant having the same modulo (2 and -2).

General speaking we put in evidence that same methods work fine for a certain problems class, but could fail for others. There is not a general method good for all. This is Numeric Calculus!

Generalized Eigenproblem

Given the following matrix equation

$$\mathbf{A} \mathbf{x} = \lambda \mathbf{B} \mathbf{x} \quad (1)$$

Where \mathbf{A} and \mathbf{B} are both matrices symmetric and \mathbf{B} is positive definite, is said a generalized eigenproblem.

Equivalent non symmetric problem

This problem is equivalent to:

$$(\mathbf{B}^{-1}\mathbf{A}) \mathbf{x} = \lambda \mathbf{x} \Rightarrow \mathbf{C} \mathbf{x} = \lambda \mathbf{x} \quad (2)$$

In generally \mathbf{C} is not symmetric even \mathbf{A} and \mathbf{B} they are.

Example: transform a generalized eigenproblem into a standard eigenproblem, where the matrices \mathbf{A} and \mathbf{B} are

A			B		
7	0	2	4	2	4
0	5	2	2	17	10
2	2	6	4	10	33

In the following worksheet we have calculate the matrix $\mathbf{C} = \mathbf{B}^{-1} \mathbf{A}$

	A	B	C	D	E	F	G	H
1		A				B		
2	7	0	2		4	2	4	
3	0	5	2		2	17	10	
4	2	2	6		4	10	33	
5								
6					coeff	eigenvalues		
7		C			0.1013	re	im	
8	1.9569	-0.1413	0.3638		-0.9756	0.1717	0	
9	-0.1538	0.3225	-0.0075		2.4194	0.3033	0	
10	-0.13	-0.02	0.14		-1	1.9444	0	
11								
12								
13								
14								
15								

(=MatCharPoly(A8:C10))
 (=Poly_Roots(E7:E10))
 (=MMULT(MINVERSE(E2:G4),A2:C4))

As we can see, the matrix \mathbf{C} is not symmetric even if \mathbf{A} and \mathbf{B} are both symmetric. In order to calculate the eigenvalues we have before extracted the characteristic polynomial with the function MathCharPoly(); then we have approximated its roots with the function Poly_Roots(). The approximate eigenvalues are:

$$\lambda_1 = 0.1717 \quad \lambda_2 = 0.3033 \quad \lambda_3 = 1.9444$$

To solve the eigenvectors we can now follow the step-by-step method shown in the previous examples. But, we can also transform the given generalized problem into a symmetric one. Let's see how.

Equivalent symmetric problem

Given the following matrix equation

$$\mathbf{A} \mathbf{x} = \lambda \mathbf{B} \mathbf{x} \quad (1)$$

Where \mathbf{A} and \mathbf{B} are both matrices symmetric and \mathbf{B} is positive definite, is said a generalized eigenproblem.

In the previous paragraph we have learnt how to transform this problem into a standard eigenproblem setting $\mathbf{C} = \mathbf{B}^{-1}\mathbf{A}$. But \mathbf{C} is not symmetric. Many algorithms works fine only for symmetric matrices. By contrast there is not equally satisfactory algorithms for not symmetric case.

So, it is better to recover the given problem to a symmetrical matrix, by the Cholesky's decomposition

$$\mathbf{B} = \mathbf{L} \mathbf{L}^T \quad (2)$$

Where \mathbf{L} is a triangular matrix.

Substituting (2) into (1) and multiplying the equation by \mathbf{L}^{-1} , we get:

$$\mathbf{L}^{-1} \mathbf{A} \mathbf{x} = \lambda (\mathbf{L}^{-1} \mathbf{L}) \mathbf{L}^T \mathbf{x} \quad \Rightarrow \quad \mathbf{L}^{-1} \mathbf{A} \mathbf{x} = \lambda \mathbf{L}^T \mathbf{x}$$

And, because $\mathbf{I} = (\mathbf{L}^T)^{-1} \mathbf{L}^T = (\mathbf{L}^{-1})^T \mathbf{L}^T$, we can write:

$$\mathbf{L}^{-1} \mathbf{A} (\mathbf{L}^{-1})^T \mathbf{L}^T \mathbf{x} = \lambda \mathbf{L}^T \mathbf{x} \quad \Rightarrow \quad \mathbf{L}^{-1} \mathbf{A} \mathbf{x} = \lambda \mathbf{L}^T \mathbf{x}$$

Set the auxiliary matrix: $\mathbf{W} = \mathbf{L}^{-1}$ and the auxiliary vector $\mathbf{d} = \mathbf{L}^T \mathbf{x}$ we have

$$\mathbf{W} \mathbf{A} \mathbf{W}^T \mathbf{d} = \lambda \mathbf{d} \quad \Rightarrow \quad \mathbf{D} \mathbf{d} = \lambda \mathbf{d} \quad (3)$$

The equation (3) is the new eigenproblem where $\mathbf{D} = \mathbf{W} \mathbf{A} \mathbf{W}^T$ is symmetric

Eigenvalues of the problem (3) are equivalent to (1) while the original eigenvectors \mathbf{x} can be obtained from eigenvectors \mathbf{d} by the following formula:

$$\begin{aligned} \mathbf{d} = \mathbf{L}^T \mathbf{x} & \Rightarrow \mathbf{x} = (\mathbf{L}^T)^{-1} \mathbf{d} \Rightarrow \mathbf{x} = (\mathbf{L}^{-1})^T \mathbf{d} \\ \mathbf{x} &= \mathbf{W}^T \mathbf{d} \end{aligned}$$

That is, eigenvectors of (1) can be obtained by multiplying eigenvectors of (3) for the auxiliary matrix \mathbf{W} .

In Matrix.xla there is all you need to solve any generalized eigenproblem: Cholesky's decomposition can be done by the function **Mat_Cholesky()**; eigenvectors and eigenvalues of symmetric matrix can be calculate with the Jacoby iterative rotations performed by the two functions **MatEigenvalue_Jacobi()** and **MatEigenvector_Jacobi()**.

Thus, let's see how arrange a worksheet for solving a generalized eigenproblem, assuming the matrices \mathbf{A} and \mathbf{B} of the previous example

TUTORIAL FOR MATRIX.XLA

In this worksheet there are all formulas shown before. Formulas used for each matrix are written in blue, under the matrix itself

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1		A				B				L							
2	7	0	2		4	2	4		2	0	0						
3	0	5	2		2	17	10		1	4	0						
4	2	2	6		4	10	33		2	2	5						
5									{=Mat_Cholesky(E2:G4)}								
6																	
7		W				W ^T				D							
8	0.5	0	0		0.5	-0.125	-0.15		1.75	-0.438	-0.325						
9	-0.125	0.25	0		0	0.25	-0.1		-0.438	0.4219	0.0563						
10	-0.15	-0.1	0.2		0	0	0.2		-0.325	0.0563	0.2475						
11	{=MINVERSE(I2:K4)}				{=M_TRANSP(A8:C10)}				{=M_PROD(A8:C10,A2:C4,E8:G10)}								
12																	
13	Jacobi eigenvalues of D				mop-up				eigenvalues								
14	1.9444	2E-24	3E-33		1.9444	0	0		1.9444								
15	3E-17	0.3033	7E-18		0	0.3033	0		0.3033								
16	3E-17	-5E-21	0.1717		0	0	0.1717		0.1717								
17	{=MatEigenvalue_Jacobi(I8:K10)}				{=MatMopUp(A14:C16)}				{=MatDiagExtr(E14:G16)}								
18																	
19	Jacobi eigenvectors of D				eigenvector x				eigenvector u								
20	0.9418	0.2128	0.2603		0.534	0.0358	-0.041		0.9931	0.1316	-0.209						
21	-0.278	0.9289	0.2452		-0.05	0.2625	-0.032		-0.094	0.9659	-0.166						
22	-0.19	-0.303	0.9339		-0.038	-0.061	0.1868		-0.071	-0.223	0.9637						
23	{=MatEigenvector_Jacobi(I8:K10)}				{=MMULT(E8:G10,A20:C22)}				{=E20:E22/M_ABS(E20:E22)}								

Generalized Eigen-problem
 $A x = \lambda B x$
 where A and B symmetric and B is positive definite.

 auxiliary matrix from Cholesky decomposition
 $W = L^{-1}$
 symmetric matrix
 $D = W A W^T$
 symmetric eigen-problem
 $D d = \lambda d$
 eigenvectors
 $x = W^T d$
 normalized eigenvectors
 $u = x / |x|$

Case diagonal matrix

The case in which the matrix **B** is diagonal is particularly simple because **L** is diagonal too and can be computed by a simple square root. Also the L^{-1} is quite simple: just take the inverse of each diagonal element.

$$B = \begin{bmatrix} b_{11} & 0 & 0 \\ 0 & b_{22} & 0 \\ 0 & 0 & b_{33} \end{bmatrix} \quad L = \begin{bmatrix} \sqrt{b_{11}} & 0 & 0 \\ 0 & \sqrt{b_{22}} & 0 \\ 0 & 0 & \sqrt{b_{33}} \end{bmatrix} \quad L^{-1} = \begin{bmatrix} \frac{1}{\sqrt{b_{11}}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{b_{22}}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{b_{33}}} \end{bmatrix}$$

Let's see a practical example.

Example - How to get mode shapes and frequencies for a multi-degree of freedom structure ⁹

Our problem is an example of the "generalized" eigenproblem:

$$\mathbf{k} \phi = \omega^2 \mathbf{m} \phi \quad (1)$$

Where \mathbf{k} and \mathbf{m} are both symmetric positive definite matrices. In the specific case they were:

Stiffness matrix \mathbf{k} :

600	-600	0
-600	1800	-1200
0	-1200	3000

Mass matrix \mathbf{m} :

1	0	0
0	1.5	0
0	0	2

This problem is equivalent to a "standard" eigenproblem:

$$(\mathbf{m}^{-1} \cdot \mathbf{k}) \phi = \omega^2 \phi \quad \Rightarrow \quad \mathbf{C} \phi = \omega^2 \phi$$

The problem is that \mathbf{C} is not symmetric. Many algorithms work fine only for symmetric matrices, but not very well for no symmetric matrices. One can work around the problem by converting the problem to a symmetric one using the Cholesky's decomposition

$$\mathbf{m} = \mathbf{L} \mathbf{L}^T$$

Where \mathbf{L} is a triangular matrix. In a case like ours where \mathbf{m} is diagonal the \mathbf{L} matrix is also diagonal with each term of \mathbf{L} being the square root of the corresponding term in \mathbf{m} . Define new matrix \mathbf{W} as:

$$\mathbf{W} = \mathbf{L}^{-1}$$

Multiplying equation (1) by \mathbf{W} , one gets:

$$\mathbf{W} \mathbf{k} \mathbf{W}^T (\mathbf{L}^T \phi) = \omega^2 (\mathbf{L}^T \phi)$$

Or, more concisely

$$\mathbf{D} \mathbf{v} = \omega^2 \mathbf{v} \quad (2)$$

Where

$$\mathbf{D} = \mathbf{W} \mathbf{k} \mathbf{W}^T \quad (3)$$

The eigenvalues for equation (2) are identical to those of equation (1), and the eigenvalues of equation (1) can be obtained easily from the eigenvalues of equation (2):

$$\phi = (\mathbf{L}^T)^{-1} \mathbf{v} = \mathbf{W} \mathbf{v} \quad (4)$$

⁹ This example comes from a true problem proposed to me by Douglas C. Stahl of the Architectural Engineering and Building Construction of Milwaukee School of Engineering. Because it seems to me very interesting also for other people, I decide to publish it in this tutorial, in the version arranged by Doug and me.

So here's what you do:

Starting with **k** and **m**, make **L** and then **W** and then **D**.

	A	B	C	D	E	F	G	H	I	J	K
68		stiffness matrix k :				mass matrix m :					
69		600	-600	0		1	0	0			
70		-600	1800	-1200		0	1.5	0			
71		0	-1200	3000		0	0	2			
72											
73	L				W				D		
74	1	0	0		1	0	0		600	-490	0
75	0	1.2247	0		0	0.8165	0		-489.9	1200	-692.82
76	0	0	1.4142		0	0	0.7071		0	-692.82	1500
77											

Calculate the eigenvalues and eigenvectors for **D**, with functions **matEigenvalue_jacobi** and **matEigenvector_jacobi** contained in the Add-in MATRIX. Use a number of iteration more than 40. These eigenvalues are the ones you want. These are the correct squared frequencies for our problem!

	A	B	C	D	E	F	G	H	I	J
80		maxLoops =		50						
81										
82		eigenvalues are diags of this matrix:				eigenvalues		eigenvectors of D		
83		210.88	0.00	0.00		210.88		0.743	-0.636	0.210
84		0.00	963.96	0.00		963.96		0.590	0.472	-0.655
85		0.00	0.00	2125.16		2125.16		0.317	0.610	0.726

The eigenvectors must be converted using equation 4. They are the correct mode shapes for our problem! The eigenvectors are already orthonormalized!

	A	B	C	D	E	F	G	H
1	eigenvectors of D				eigenvectors of given problem			
2	0.743	-0.636	0.210		0.743	-0.636	0.210	
3	0.590	0.472	-0.655		0.482	0.386	-0.535	
4	0.317	0.610	0.726		0.224	0.432	0.513	
5								

Linear regression

Recalls

Generally, the multivariate linear regression function is:

$$y = a_0 + a_1x_1 + a_2x_2 + \dots a_mx_m$$

Where:

$$[a_0, a_1, a_2 \dots a_m]$$

Is the coefficients vector of regression, which can be found by the following procedure
Make the following variables substitution:

$$X_i = x_i - \bar{x} \quad \text{for } i = 1..m$$

$$Y = y - \bar{y}$$

Where the right values are the average of samples **y** and **x**:

$$\bar{y} = \frac{1}{n} \sum_k y_k$$

$$\bar{x}_i = \frac{1}{n} \sum_k x_{i,k}$$

After that, the system can be writing as:

$$\begin{bmatrix} X_{11} & \dots & X_{1m} \\ \dots & \dots & \dots \\ X_{nm} & \dots & X_{nm} \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ \dots \\ a_m \end{bmatrix} = \begin{bmatrix} Y_1 \\ \dots \\ Y_m \end{bmatrix}$$

That is, in compact form:

$$[X] \cdot a = Y$$

We solve this singular system with SVD method, obtaining the following 3 matrices

$$[X] = [U] \cdot [D] \cdot [V]^T = [U] \cdot \begin{bmatrix} d_{11} & 0 & 0 \\ \dots & \dots & \dots \\ 0 & 0 & d_{mm} \end{bmatrix} \cdot [V]^T$$

Taking the inverse of the diagonal matrix D, that is trivial because:

$$[D]^{-1} = \begin{cases} 1/d_{ij} & \Rightarrow i = j \\ 0 & \Rightarrow i \neq j \end{cases}$$

The system solution is:

$$a = [V] \cdot [D]^{-1} [U]^T b$$

The final constant terms can be obtained by the following formulas

$$a_0 = \bar{Y} - \sum_{i=1}^m a_i \bar{X}_i$$

Let's see how to use the regression formulas

Linear Regression models

Linear model: $a_0 + a_1 x_1 + a_2 x_2$

Example - assume to have to find the bivariate linear function $f(x_1, x_2)$ that better approximate the following table

y	x1	x2
548.8	0.1	10
558.85	0.2	10.25
580.15	0.3	10.75
601.45	0.4	11.25
622.75	0.5	11.75
644.05	0.6	12.25
665.35	0.7	12.75
686.65	0.8	13.25
674.2	0.9	13
673	1	13
671.8	1.1	13
445.6	1.2	8
421.9	1.3	7.5
398.2	1.4	7
374.5	1.5	6.5

The function (model) that we find is

$$f(x_1, x_2) = a_0 + a_1 x_1 + a_2 x_2$$

We use the linear regression to find the coefficients a_0, a_1, a_2

Arrange the table data as in the following worksheet

Select the range where you want to paste the coefficients, For example the range F7:F9

TUTORIAL FOR MATRIX.XLA


Microsoft Excel - regr lin examples.xls

File Modifica Visualizza Inserisci Formato Strumenti Dati Finestra ?

Arial 8

1	A	B	C	D	E	F	G	H	I
2	Linear regression with 2 variables								
3	y	x1	x2						
4	548.8	0.1	10						
5	558.85	0.2	10.25						
6	580.15	0.3	10.75						
7	601.45	0.4	11.25						
8	622.75	0.5	11.75						
9	644.05	0.6	12.25						
10	665.35	0.7	12.75						
11	686.65	0.8	13.25						
12	674.2	0.9	13						
13	673	1	13						
14	671.8	1.1	13						
15	445.6	1.2	8						
16	421.9	1.3	7.5						
17	398.2	1.4	7						
18	374.5	1.5	6.5						

	A	B	C	D	E	F	G
1	Linear regression with 2 variables						
2							
3	y	x1	x2				
4	548.8	0.1	10				
5	558.85	0.2	10.25				
6	580.15	0.3	10.75				
7	601.45	0.4	11.25			a0 =	
8	622.75	0.5	11.75			a1 =	
9	644.05	0.6	12.25			a2 =	
10	665.35	0.7	12.75				
11	686.65	0.8	13.25				
12	674.2	0.9	13				
13	673	1	13				
14	671.8	1.1	13				
15	445.6	1.2	8				
16	421.9	1.3	7.5				
17	398.2	1.4	7				
18	374.5	1.5	6.5				

Now open the function wizard with  and insert the **REGRL()** function

Microsoft Excel - regr lin examples.xls

File Modifica Visualizza Inserisci Formato Strumenti Dati Finestra ?

Arial 8

MATR.INVERSA X y = REGRL(A4:A18,B4:C18)

1	A	B	C	D	E	F	G	H	I	J	K	L	M	N
2	Linear regression with 2 variables													
3	y	x1	x2											
4	548.8	0.1	10											
5	558.85	0.2	10.25											
6	580.15	0.3	10.75											
7	601.45	0.4	11.25											
8	622.75	0.5	11.75											
9	644.05	0.6	12.25											
10	665.35	0.7	12.75											
11	686.65	0.8	13.25											
12	674.2	0.9	13											
13	673	1	13											
14	671.8	1.1	13											
15	445.6	1.2	8											
16	421.9	1.3	7.5											
17	398.2	1.4	7											
18	374.5	1.5	6.5											

REGRL

Y [A4:A18] = (548.8;558.85;580.15;601.45;622.75;644.05;665.35;686.65;674.2;673;671.8;445.6;421.9;398.2;374.5)

X [B4:C18] = (0.1;0.2;0.3;0.4;0.5;0.6;0.7;0.8;0.9;1;1.1;1.2;1.3;1.4;1.5)

Zerosidegap = (100.000000000000;-12.000000000000;45.000000000000)

Linear Regression with SVD method.

Result formula = 100

remember to give the sequence CTRL+SHIFT+ENTER

F7 = (=REGRL(A4:A18,B4:C18))

1	A	B	C	D	E	F
2	Linear regression with 2 variables					
3	y	x1	x2			
4	548.8	0.1	10			
5	558.85	0.2	10.25			
6	580.15	0.3	10.75			
7	601.45	0.4	11.25			a0 = 100
8	622.75	0.5	11.75			a1 = -12
9	644.05	0.6	12.25			a2 = 45
10	665.35	0.7	12.75			
11	686.65	0.8	13.25			
12	674.2	0.9	13			
13	673	1	13			
14	671.8	1.1	13			
15	445.6	1.2	8			
16	421.9	1.3	7.5			
17	398.2	1.4	7			
18	374.5	1.5	6.5			

Remember: give the CTRL+SHIFT+ENTER sequence to enter this function

The solution is $a_0 = 100$, $a_1 = -12$, $a_2 = 45$, giving the regression function:

$$f(x_1, x_2) = 100 - 12x_1 + 45x_2$$

Polynomial model: $a_0 + a_1 x + a_2 x^2 + a_3 x^3$

Example: assume to have to find the 3 degree polynomial that better approximate the following table

y	x
19.531	0.1
18.375	0.5
19	1
22.625	1.5
30	2
41.875	2.5
59	3
82.125	3.5
112	4
149.375	4.5
195	5

The function (model) that we find is

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

Copy the above table in a worksheet and add two other column x^2 and x^3
 The new values are compute by the x column; now select the range G5:G8 and insert the REGRL() function, giving the correct parameter: y = range A4:A14 and x = B4:D14

G5	={REGRL(A4:A14,B4:D14)}									
	A	B	C	D	E	F	G	H	I	J
1	Polynomial regression with one variable					$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$				
2										
3	y	x	x2	x3						
4	19.531	0.1	0.01	0.001						
5	18.375	0.5	0.25	0.125		a0 =	20			
6	19	1	1	1		a1 =	-5			
7	22.625	1.5	2.25	3.375		a2 =	3			
8	30	2	4	8		a3 =	1			
9	41.875	2.5	6.25	15.625						
10	59	3	9	27						
11	82.125	3.5	12.25	42.875						
12	112	4	16	64						
13	149.375	4.5	20.25	91.125						
14	195	5	25	125						
15										
16										
17			=B14^2	=B14^3						
18										
19										

Polynomial regression for one variable can be made in a more compact way with the function **REGRP()**. This function computes by itself the power of x and you do not need this job by hand.

Let's see how to solve the previous example with REGRP ()

	A	B	C	D	E	F	G
1	Polynomial regression with one variable						
2							
3	y	x	$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$				
4	19.531	0.1					
5	18.375	0.5					
6	19	1					
7	22.625	1.5					
8	30	2					
9	41.875	2.5					
10	59	3					
11	82.125	3.5					
12	112	4					
13	149.375	4.5					
14	195	5					
15							

Saving time and space is clear. It will be even more evident for higher degree polynomial

Two variables polynomial model: $a_0 + a_1 x + a_2 y + a_3 xy + a_4 x^2 + a_5 y^2$

Example: assume to have to find the 2 degree bivariate (x, y) polynomial that better approximate the following table

f(x,y)	x	y
1338.09	13.5	0.2
1342.41	10.5	1.4
1351.14	7.5	2.3
1407.91	12	4.9
1503.51	12.4	8.5
1442.41	-1.5	3.4
1507.54	-4.5	3.3

The function (model) that we find is

$$f(x, y) = a_0 + a_1 x + a_2 y + a_3 xy + a_4 x^2 + a_5 y^2$$

Having five parameters: $a_0, a_1, a_2, a_3, a_4, a_5$

	A	B	C	D	E	F	G
1	Multipolynomial regression						
2							
3	f(x,y)	x	y	xy	x^2	y^2	
4	1338.09	13.5	0.2	2.7	182.25	0.04	
5	1342.41	10.5	1.4	14.7	110.25	1.96	
6	1351.14	7.5	2.3	17.25	56.25	5.29	
7	1407.91	12	4.9	58.8	144	24.01	
8	1503.51	12.4	8.5	105.4	153.76	72.25	
9	1442.41	-1.5	3.4	-5.1	2.25	11.56	
10	1507.54	-4.5	3.3	-14.85	20.25	10.89	
11							
12	a0	a1	a2	a3	a4	a5	
13	1450	-22	-13	2	1	1	
14							
15							
16							
17							

First of all, we easily calculate the remain variables: xy, x^2, y^2 putting them in the adjacent columns D, E, F

Then we can calculate the unknown parameters with linear regression.

Note that, in this case we have put the results in a horizontal vector. REGRL can have both outputs: horizontal or vertical vector

`{=REGRL(A4:A10,B4:F10)}`

Linear model with fixed intercept: $k x$

Sometime we have to fix or even eliminate (fix to 0) the intercept value of a regression model

Example: A test of a gas-barometer has given the following experimental result. Find the barometer constant $k = P / T$ (mbar / °K)

T (°K)	P (mbar)
273	1101.2
278	1112.3
283	1141.2
288	1159
293	1178.1
298	1197.2
303	1221.3

The function model that we find is

$$P = kT$$

This model implies that for $T=0 \Rightarrow P=0$

TUTORIAL FOR MATRIX.XLA

	A	B	C	D	E	F	G	H	
1	$P = k T$								
2									
3	T (°K)	P (mbar)		a0	a1				
4	273	1101.2		-30.09	4.13	{=REGRL(B4:B10,A4:A10,FALSE)}			
5	278	1112.3		0.00	4.02	{=REGRL(B4:B10,A4:A10,TRUE)}			
6	283	1141.2							
7	288	1159		k =	4.02				
8	293	1178.1							
9	298	1197.2							
10	303	1225							
11									

Using the linear regression with free intercept we find a coefficient $k = 4.13$ but we note also a spurious constant terms not negligible ($a_0 \cong -30$)

Using the linear regression with fixed intercept to zero, we have a coefficient $k = 4.02$ that it is more close to the original, not perturbed, model ($k=4$)

Non linear regression - Transformable linear models

When investigating the relationship between two variables, we usually make experimental observations to take paired values of the variables (x_i, y_i). The investigator might then ask what mathematically formula best describes the relationships (if any) As seen in the previous section the technique used is the linear regression by the method of least square. But many times the model that we must chose is intrinsically not linear. Exponential, logarithmic and rational model are the most common (exponential decay, pollution, etc.).

Quasi linear model

Sam simple nonlinear model can be converted into linear model by variables transformation.

- Exponential $y = y_0 e^{kx}$
- Logarithmic $y = b_0 + b_1 \ln(x)$
- Rational $y = (b_0 + b_1 x)^{-1}$
- Power $y = a x^\alpha$

Transformable linear models have some advantages in that the linearized form can be treated with the linear regression formulas that we know. This technique, however, is only a possibility for the simplest of nonlinear model.

There is another important drawback that we must point out. The models obtained by transformation are only an approximation of the non-linear least squares model. We explain better this concept, not much explained by many authors. After we transformed a nonlinear model into a linear one we apply the linear regression formulas to get the unknown parameters, for example (a_1, a_2). If we calculate the sum of squared residual (ssr)

$$ssr = \sum (y_i - f(x_i, a_1, a_2))^2$$

Even if we have calculated (a_1, a_2) with the linear Least Square regression method is the ssr true the least? The answer is in generally negative. In other words, it could be other different couple of parameter (a_1, a_2) that minimized ssr. It is not guaranteed that the parameters given by linear regression are the best. In the following examples we show this trap.

Exponential curve fit: $y_0 e^{kx}$

The model, having two parameter y_0 and k , is $y = y_0 e^{kx}$

Taking the logarithm of both sides, we have

$$\ln(y) = \ln(y_0 e^{kx}) \Rightarrow \ln(y) = \ln(y_0) + kx$$

Setting the new variable $z = \ln(y)$, the equation became linear in x and z , with the parameters z_0 and k .

$$z = \ln(y) \Rightarrow z = z_0 + kx$$

The original parameter y_0 can be found by the simple formula $y_0 = \exp(z_0)$

Reassuming, to calculate an exponential regression of data set (x_i, y_i) , we have to made:

1. Convert al data set (x_i, y_i) into a new data set (x_i, z_i) , where $z_i = \ln(y_i)$
2. Apply the linear regression to find z_0 and k
3. Convert the z_0 into y_0 by the formula $y_0 = \exp(z_0)$

Let's see with an example. The data set is in the following table

t	y
0.1	7.9
0.2	7.1
0.3	5.5
0.5	4.1
1	1.3
1.5	0.6
3	0.3

An experimental test has given the table at the left.
We search the exponential decay model

$$f(t) = y_0 \exp(kt)$$

Parameters to determine are: y_0 and k

The worksheet below show the results and the formulas used. The arrangement should be clear: we have computed the "z" column; then, we have performed the linear regression of (x, z) with **LINEST()** built-in function, finding "k" and "z0". With the exp() function we have computed the "y0" parameter

Then we have calculated the estimated values column "f1", and the one of residues "error f1". At the bottom, we have computed the standard deviation of the residues for estimating the standard error.

	A	B	C	D	E	F	G	H	I	J
1	t	y	z	f1	error f1	f2	error f2			
2	0.1	7.9	2.0669	6.1699	1.7301	8.1873	-0.2873	= (B2-F2)		
3	0.2	7.1	1.9601	5.4752	1.6248	6.7032	0.3968			
4	=LN(B2)	5.5	1.7047	4.8888	0.6412	5.4881	0.0119	= \$E\$15*EXP(\$C\$15*A2)		
5	0.5	4.1	1.4110	3.8263	0.2737	3.6788	0.4212			
6	= \$E\$14*EXP(\$C\$14*A2)			2.1057	-0.8057	1.3534	-0.0534			
7	1.5	0.6	0.5408	1.1588	-0.5588	0.4979	0.1021			
8	3			0.1931	0.1069	0.0248	0.2752			
9										
10										
11										
12										
13										
14	parameter for f1		k	z0	y0					
15	parameter for f2		-1.194	1.939	6.953					
16			-2		10					

As we see the exponential parameters found by linear regression are:

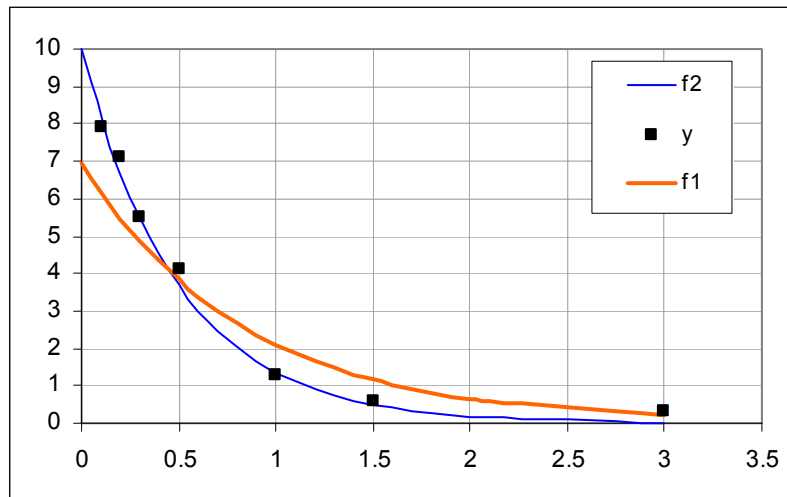
k	y_0	Standard error
-1.194	6.953	0.91

Adjacent to the previous one we have repeat the computing of the error standard for another regression "f2" obtained with another parameters couple: $k = -2$, $y_0 = 10$
Note that this couple is not given by linear regression; we say that we have known these value by another nonlinear method, a topic out of the subject of this document.

k	y ₀	Standard error
-2	10	0.24

As we can see, the standard error is quite lower than the one given by the linear regression. So we have shown that there is another couple of parameter - differently from the ones given by linear regression, that are better with the least squares criterion.

This can also be seen, at the first sight, by the following graph



The curve obtained by the linearized regression is not sure the best fit for the original data set

From the above example a question raises: why and when can we use this linearized method? The answer is: it depends by the data set. If we have a data set with many equispaced samples and with a low level of noise, the linearization method gives result sufficiently close to the best regression.

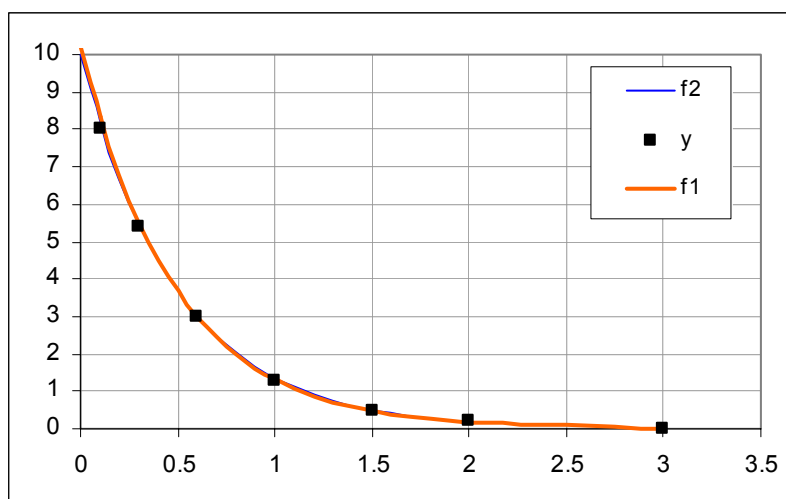
Let's repeat the exponential regression with this data set

t	y
0.1	8
0.3	5.4
0.6	3
1	1.3
1.5	0.5
2	0.2
3	0.02

For this data set, we can obtain the following parameter more close to the best model

k	z ₀	y ₀
-2.041	2.323	10.204

The better approximation it is evident in the following graph



As we can see the curve obtained by linear regression fits much better the given data set.

Logarithmic curve fit: $b_0 + b_1 \ln(x)$

The model, having two parameter b_0 and b_1 , is $y = b_0 + b_1 \ln(x)$

Substituting: $t = \ln(x)$ we have $t = \ln(x) \Rightarrow y = b_0 + b_1 t$

Thus the original parameters b_0 and b_1 remain unchanged.

Reassuming, to calculate the logarithmic regression of data set (x_i, y_i) , we have to made:

1. Convert al data set (x_i, y_i) into a new data set (t_i, y_i) , where $t_i = \ln(x_i)$
2. Apply the linear regression to find b_0 , b_1

Let's see with an example. The data set is in the following table

t	y
1.3	2.83
1.6	5.98
2	8.81
2.5	10.33
3	12.35
4	15.19
5	16.68

An experimental test has given the table at the left.
We search the logarithmic curve for best fitting

$$f(x) = b_0 + b_1 \ln(x)$$

Parameters to determine are: b_0 and b_1

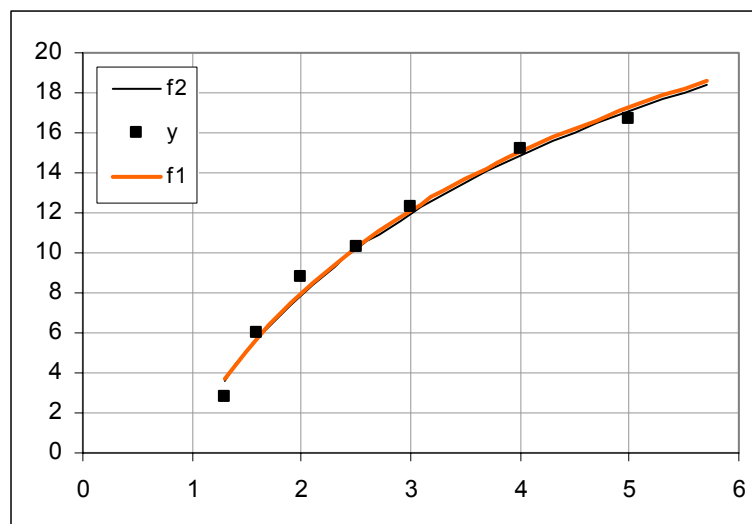
The worksheet below show the results and the formulas used. We have computed the "t" column; then, we have performed the linear regression of (t, y) with **LINEST()** built-in function, finding "b0" and "b0". Then we have calculated the estimated values column "f1", and the one of residues "error f1". At the bottom, we have computed the standard deviation of the residues for estimating the standard error.

We have made the same for two other parameter $b_0 = 1$ and $b_0 = 10$.

We can observe that, in this case, the linear regression has produced a true best-fit solution

	A	B	C	D	E	F	G	H	I	J
1	x	y	t	f1	error f1	f2	error f2			
2	1.3	2.83	0.262	3.679	-0.849	3.624	-0.794	= (B2-F2)		
3	1.6	5.98	0.470	5.775	0.205	5.700	0.280			
4	2	8.81	0.693	8.026	0.784	7.931	0.879	= \$D\$15+\$C\$15*LN(A2)		
5	2.5	10.33	=LN(A2)	10.278	0.052	10.163	0.167			
6	3	12.35	=LN(A2)	12.118	0.232	11.986	0.364			
7	4	14.35	=LN(A2)	14.021	0.169	14.863	0.327			
8	5	16.68	=LN(A2)	15.973	-0.593	17.094	-0.414			
9			= (B2-D2)		STD f1		STD f2			
10			=STDEV(P(E2:E8))		0.5091		0.5107	=STDEV(P(G2:G8))		
11										
12	{=LINEST(C2:C8,A2:A8)}									
13			b1	b0						
14	parameter for f1		10.091	1.032						
15	parameter for f2		10	1						

The graph below confirms the best fit



Rational curve fit: $(b_0 + b_1 x)^{-1}$

The model, having two parameter b_0 and b_1 , is

$$y = \frac{1}{b_0 + b_1 x}$$

Substituting: $z = 1/y$ we have

$$\frac{1}{y} = b_0 + b_1 x \Rightarrow z = b_0 + b_1 x$$

Thus the original parameters b_0 and b_1 remain unchanged.

Reassuming, to calculate the rational regression of data set (x_i, y_i) , we have to made:

3. Convert al data set (x_i, y_i) into a new data set (x_i, z_i) , where $z_i = 1/y_i$
4. Apply the linear regression to find b_0 , b_1

Let's see with an example. Two data sets are in the following tables. Find the best fit for linear rational models

TUTORIAL FOR MATRIX.XLA

x	y
1.3	0.66
1.6	0.58
2	0.49
2.5	0.32
3	0.33
4	0.24
5	0.18

x	y
1.3	0.57
1.6	0.49
2	0.51
2.5	0.34
3	0.32
4	0.19
5	0.11

An experimental test has given the table at the left.
We search the rational curve for best fitting

$$f(x) = (b_0 + b_1 x)^{-1}$$

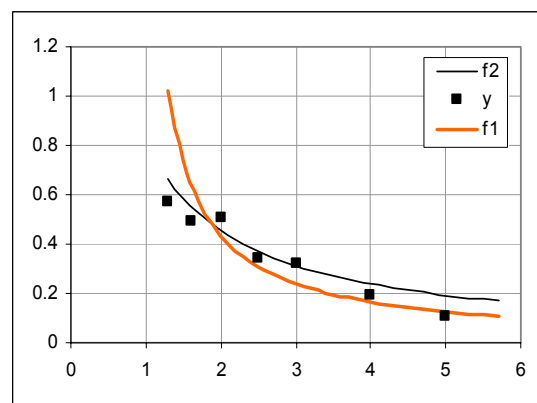
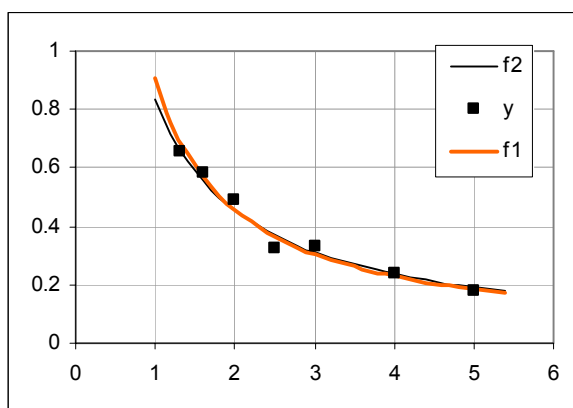
Parameters to determine are: b_0 and b_1

	A	B	C	D	E	F	G
1	x	y	z	f1	error f1	f2	error f2
2	1.3	0.66	1.521	0.698	-0.041	0.667	-0.009
3	1.6	0.58	1.718	0.569	0.013	0.556	0.026
4	2	0.49	2.032	0.456	0.036	0.455	0.038
5	2.5	0.32	3.107	0.365	-0.043	0.370	-0.048
6	3	0.33	3.026	0.305	0.026	0.313	0.018
7	4	0.24	4.242	0.229	0.007	0.238	-0.002
8	5	0.18	5.580	0.183	-0.004	0.192	-0.013
9					STD f1		STD f2
10		=1/B8			0.0287		0.0268
11							
12							
13			b1	b0			
14	parameter for f1		1.088	0.018			
15	parameter for f2		1	0.2			
16							

The worksheet arrangement is similar to the previous one. Only the column "z", where we have compute the inverse $z = 1/y$, is changed

Repeating the rational regression for the two tables we have found the parameters

	b1	b0
1st table	1.08	0.018
2nd table	1.878	-1.465



As we can see, the first example approximates much better than the second one. Note also that in the second left plot, data set has a larger random noise

Power curve fit: $a x^\alpha$

The model, having two parameter b_0 and b_1 , is $y = a x^\alpha$

Taking the logarithm of both sides, we have

$$\ln(y) = \ln(a x^\alpha) \Rightarrow \ln(y) = \ln(a) + \alpha \ln(x)$$

Setting the new variables $z = \ln(y)$, $t = \ln(x)$ and setting $z_0 = \ln(a)$ the equation became linear in t and z , with the parameters z_0 and α .

$$z = z_0 + \alpha t$$

The original parameters a can be computed by: $a = \exp(z_0)$

Reassuming, to calculate the logarithmic regression of data set (x_i, y_i) , we have to made:

1. Convert al data set (x_i, y_i) into a new data set (t_i, z_i) , where $t_i = \ln(x_i)$, $z_i = \ln(y_i)$
2. Apply the linear regression to find α , z_0
3. Calculate the original parameter $a = \exp(z_0)$

Let's see with an example. Two data sets are in the following tables. Find the best fit for power models

x	y
1.3	1.79
1.6	1.30
2	1.16
2.5	1.06
3	1.04
4	0.80
5	0.60

x	y
1.3	1.73
1.6	1.62
2	1.12
2.5	0.84
3	0.97
4	0.75
5	0.75

An experimental test has given the tables at the left.

We search the power curve for best fitting

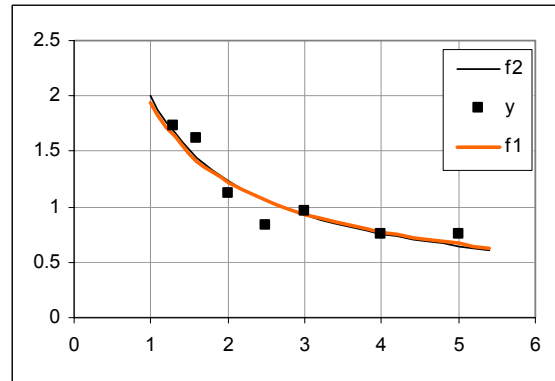
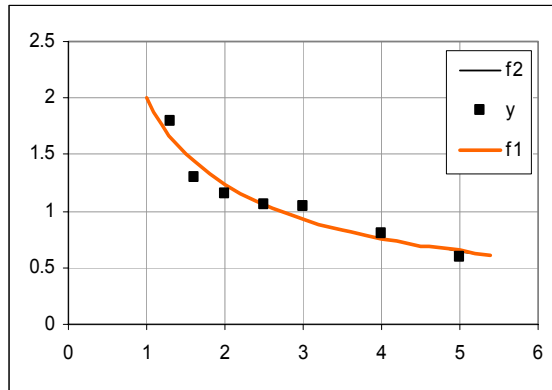
$$f(x) = a x^\alpha$$

Parameters to determine are: a and α

	A	B	C	D	E	F	G	H
1	x	y	t	z	f1	error f1	f2	error f2
2	1.3	1.79	0.262	0.583	1.664	0.127	1.664	0.127
3	1.6	1.30	0.470	0.266	1.440	-0.135	1.439	-0.134
4	2	1.16	0.693	0.145	1.233	-0.076	1.231	-0.075
5	2.5	1.06	0.916	0.057	1.055	0.003	1.053	0.006
6	3	1.04	1.099	0.037	0.929	0.109	0.927	0.111
7	4	0.80	1.386	-0.217	0.761	0.044	0.758	0.047
8	5	0.60	1.609	-0.511	0.651	-0.051	0.648	-0.048
9	=LN(A8)	=LN(B8)	=LN(A8)	=LN(B8)				
10						STD f1		STD f2
11						0.0899		0.0898
12	{=LINEST(D2:D8,C2:C8)}							
13			α	z_0	a			
14	parameter for f1		-0.697	0.692	1.99771			
15	parameter for f2		-0.7		2			

Repeating the power regression for the two tables we have found the parameters

	α	a
1st table	-0.697	1.997
2nd table	1.878	-1.465



We can note the relative high insensibility to the random perturbation of this kind of regression. In fact the power regression is one of the robust and reliable methods.

Interpolation

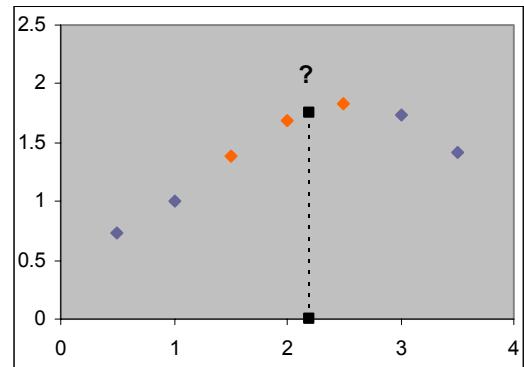
Recalls

Given a set of values of an unknown function $y_i = f(x_i)$ calculates at N points x_1, x_2, \dots, x_N , we want to estimate the value $f(x)$ at an arbitrary point x , being $x_1 < x < x_N$. This process of estimating the outcomes in between sampled data points is called **interpolation**.

The idea is that the points lie on an underlying but unknown curve. The problem is to be able to estimate the values of the curve at any position between the known points.

Of course, we cannot calculate directly the $f(x)$. For example the points $f(x_i)$ might result from physical experimental measurements or from long, heavy calculation.

In many engineering applications, data sampled are usually discrete and the analytic expression for $f(x)$ is not always well known. Usually we can only decide the time and the method of the sampling: so the x_i points are often equally spaced, but not necessarily.



Many interpolation methods exist to solve this diffuse and basic problem: linear, polynomial, trigonometric, rational, Hermite, spline, continue fraction, Padè, Chebychef, Bezier, Spline, regression, piecewise, etc. Many of them are suitable for a special kind of data. The scope of this section is limited to discussing some common interpolation methods: *linear and polynomial interpolation*.

We have to point out an important concept. Interpolation is related to, but distinct from, the function modeling (or function approximation, or fitting) that consist of finding an easy, approximate model to have a better understanding of the physical phenomenon, and a more analytically controllable function fitting the field data. This task was explained and developed in the previous chapter about “*linear regression*”

In interpolation problems, on the contrary, we do not cure of the model. We are only focused to obtain the most accurate function value using only the given points data set. Many people confuse these concepts, - interpolation and approximation – because, after all, they use the same algorithms.

Why to interpolate?

Many different problems can take advantage from the power of interpolation methods. The most common is the so called *sub tabulation* problem. It happens when we want to generate a larger table of function values (x_i, y_i) starting from an accurate but more limited table. The interpolation line should pass between the original points (knots). This is a

typical problem of function plotting when values are calculated by computer simulation programs.

An important task is the inverse of sub tabulation: the *data smoothing* or *data mop-up*. It comes when we have many sample points affected by a considerable error noise and we want to obtain a table with less, but more accurate values. The interpolation line does not cross the given points. This situation is very common in many engineering applications where data came from experimental measurements

Other problem is the *data regularization*; sometime is not possible to obtain a regular equispaced grid of points. This happens for example in random algorithms like the Monte Carlo method. In this case we can use the interpolation method to extract from the random table another table with equispaced points.

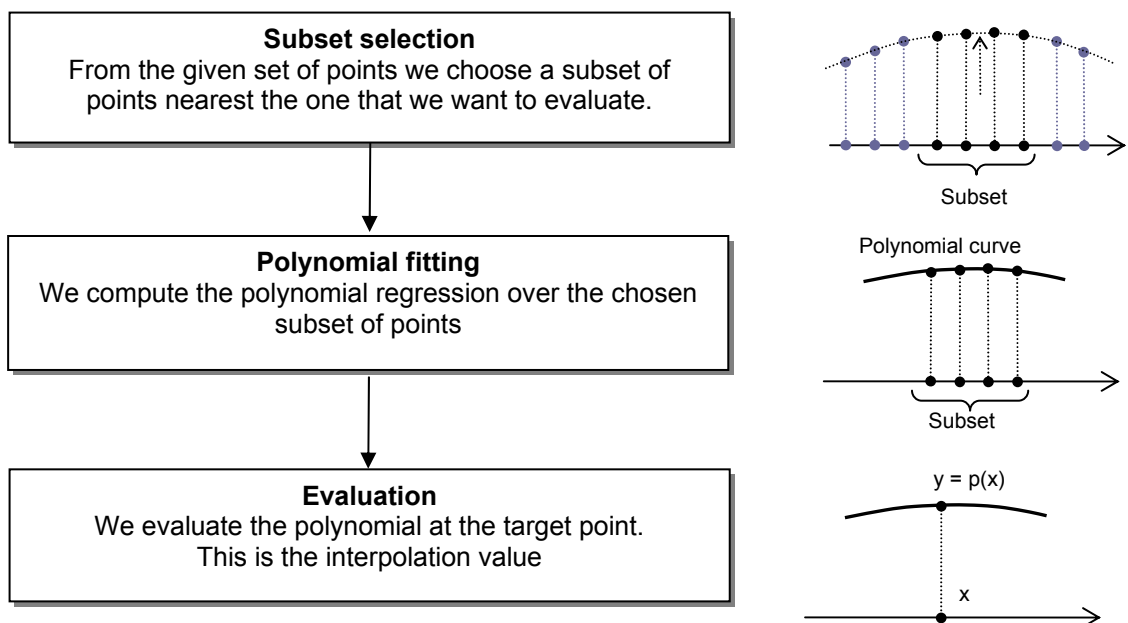
Usually these problems, for example the data smoothing and regularization, can happen at the same time.

Task	Source Data points	Scope
Sub tabulation	Few accurate points (knots)	To obtain more extra points between the knots (finer table)
Smoothing	Many approximate points	To obtain more “clean” points following a more smooth curve
Regularization	Random x_i points	Equispaced x_i points

Piecewise polynomial interpolation schema

Explained the scope of the interpolation, it remains to decide how to interpolate. Well, we have to say that there are lots of different interpolation schemes. Many of them, very ingenious, are dedicated to particular class of given points. It is out of the scope of this document to illustrate all of them.

We shall discuss here a popular interpolation method called *piecewise polynomial interpolation*, a method sufficiently general to approximate large classes of function that we may find in practice. It is also conceptually simple and didactically important.



The second step is common with the modeling problem, but we still emphasized the fact that here the polynomial is used only to evaluate the y value at point x. Changing the point x, also the subset and, consequently, the polynomial may change. So there would be many polynomial formulas covering the entire range of points. This is the main difference from the modeling problem where the goal is to find the simplest unique formula that approximate all range of points.

Linear Interpolation

This is the simplest interpolation. It assumes a straight line between 2 knots to calculate the value y for x between

$$x_a < x \leq x_b$$

It is always possible to find two knots satisfying the above constrain.

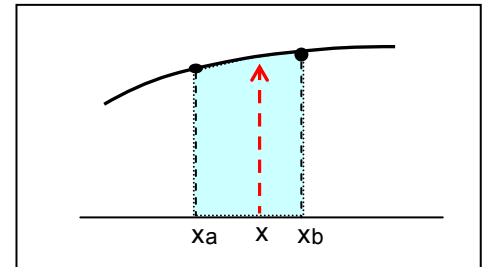
The interpolation value is obtained by the linear formula

$$y(x) = \frac{y_b - y_a}{x_b - x_a} \cdot (x - x_a) + y_a$$

Or, as well, by the normalized formula

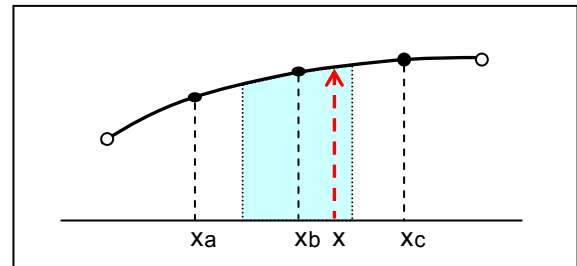
$$y(x) = y_b \cdot t + y_a \cdot (1 - t) \quad , \quad t = \frac{x - x_a}{x_b - x_a}$$

In spite of its moderate accuracy this method has several advantages that make it still very popular



Parabolic Interpolation

We want to find the value y at given x = 2.2 between a set of given points (x_i, y_i). If we want to interpolate the value y with a parabolic polynomial (2nd degree) we need at least 3 points. Let's begin to choose the 3 points nearest to the point x, (x_a, x_b, x_c), having the corresponding function values (y_a, y_b, y_c). The points a, b, c are called **knots** of the interpolation



Knots	x	y
a	1.5	1.37791
b	2	1.69285
c	2.5	1.83318

Note that the point x = 2.2 must be nearest to the central knots x_b. It assures the highest accuracy of interpolation. This condition is expressed by the following formula

$$\frac{x_a + x_b}{2} < x \leq \frac{x_b + x_c}{2} \quad (1a)$$

If the above condition is not true, we simply shift the points selection to right or left till the conditions is satisfied. Apart for the first and the last segment, we can always choose a subset that satisfies this relation

Now we have to compute the parabolic polynomial crossing the 3 knots. There are many formulas and methods to build such polynomial. We choose here the method of the linear system because it is simple and has more didactic diffusion.

The equations of the linear system can be built by the generic parabolic polynomial formula

$$y = a_0 + a_1 x + a_2 x^2 \quad (2a)$$

Substituting the values of the knots, we have 3 linear equations in the unknown (a_0 , a_1 , a_2) coefficients, which can be rearranged in matrix form

$$\begin{cases} y_a = a_0 + a_1 x_a + a_2 x_a^2 \\ y_b = a_0 + a_1 x_b + a_2 x_b^2 \\ y_c = a_0 + a_1 x_c + a_2 x_c^2 \end{cases} \Rightarrow \begin{bmatrix} 1 & x_a & x_a^2 \\ 1 & x_b & x_b^2 \\ 1 & x_c & x_c^2 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} y_a \\ y_b \\ y_c \end{bmatrix}$$

The system matrix is the Vandermonde's matrix that can be easily obtained by the function **Mat_Vandermonde()** and the solution can be found with the **SYSLIN()** function

After we have found the polynomial coefficients, we can compute the interpolate value y at the point $x = 2.2$, by the formula (2a)

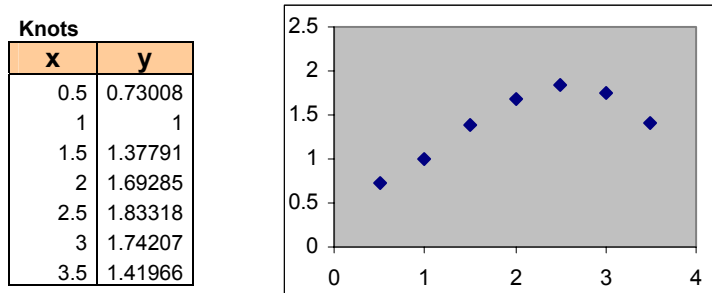
A possible solution arrangement is shown in the following worksheet

	A	B	C	D	E	F	G	H	I
10									
11	Knots			{=Mat_Vandermonde(A13:A15)}					
12	x	y		Vandermonde matrix				parabolic coeff.	
13	1.5	1.37791		1	1.5	2.25		a0 =	-0.6146
14	2	1.69285		1	2	4		a1 =	1.852176
15	2.5	1.83318		1	2.5	6.25		a2 =	-0.34923
16									
17				x =	2.2		{=SYSLIN(D13:F15;B13:B15)}		
18				y =	1.76994		=I13+I14*E17+I15*E17^2		
19							=Interpolate(E17;A13:B15;2)		
20				y =	1.76994				
21									

Of course we have shown in detail all the process to explain better the interpolation method but, in matrix.xla it exists the function **Interpolate(x, knots, [degree])** that just performs all this calculus giving the same result.

This function came in handy overall when we have to interpolate many values over a larger set of knots. In this case we have to repeat the above process for any point to interpolate.

Example. Sub tabulate the following table for with step $\Delta x = 0.1$ between 0.5 and 3.5

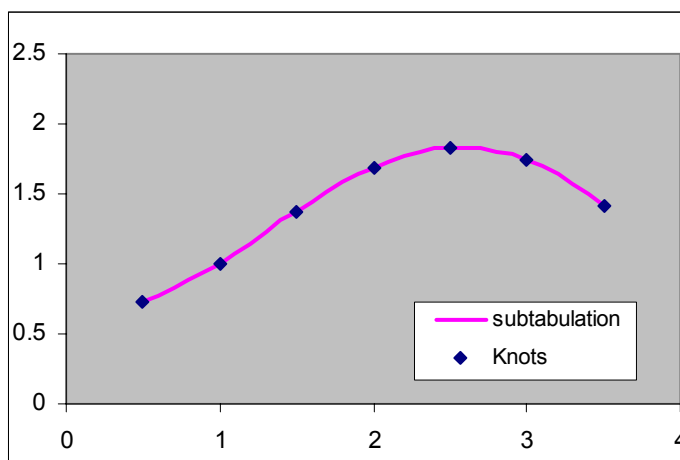


TUTORIAL FOR MATRIX.XLA

	A	B	C	D	E
30	Knots			subtabulation	
31	x	y		x	y
32	0.5	0.73008		0.5	0.73008
33	1	1		0.6	0.77542
34	1.5	1.37791		0.7	0.82509
35	2	1.69285		0.8	0.87907
36	2.5	1.83318		0.9	0.93738
37	3	1.74207		1	1
38	3.5	1.41966		1.1	1.06694
39				1.2	1.13821
40	{=Interpolate(D32:D62,A32:B38,2)}				
41				1.4	1.30736

The **interpolate(x,knots, degree)** function accepts also a vector of value x. In this case it returns a vector of values y(x) and must be insert with the CTRL+SHIFT+ENTER sequence. This is the fast way to perform a subtabulation.

Note from the graph how good is the interpolation fitting



How many polynomials have been necessary to sub tabulate the above function? Well we can say that the function Interpolate build one polynomial for any consecutive set of 3 knots; in the case we need 5 parabolic polynomials to cover all the interpolation range.

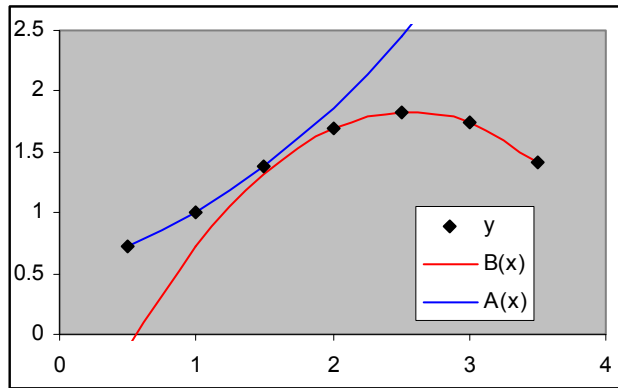
The coefficients of the interpolation polynomials can be computed by the **REGRP(degree, f, x)** function

In the following example we compute and plot the parabolic polynomials with the first and the last 3-set of knots

	A	B	C	D	E	F	G
1	Knots						
2	x	y		A		A(x)	B(x)
3	0.5	0.73008		0.56814		0.73008	-0.11541
4	1	1		0.21589		1	0.71869
5	1.5	1.37791		0.21597		1.37791	1.32149
6	2	1.69285		B		1.8638	1.69299
7	2.5	1.83318		-1.18082		2.45768	1.83318
8	3	1.74207		2.36211		3.15955	1.74207
9	3.5	1.41966		-0.46261		3.9694	1.41966
10							
11	{=REGRP(2,B3:B5,A3:A5)}					=D\$3+D\$4*A9+D\$5*A9^2	
12	{=REGRP(2,B7:B9,A7:A9)}					=D\$7+D\$8*A9+D\$9*A9^2	
13							

The 1st parabolic polynomial A(x) cross the knots at x = [0.5, 1, 1.5]

The 2nd parabolic polynomial B(x) cross the knots at x = [2.5, 3, 3.5]



From the graph we understand why we need many paraboles in order to obtain a good interpolation accuracy for all points of the range.

The first parabole A(x) is used to interpolate the points nearest the 1st and 2nd knots. At the end, we use the parabole B(x) to better interpolate points nearest the 6th and 7th knots. Incidentally we note that B(x) works good also for points near the 5th and 4th knots, but it is all unable to approximate the points near the first knots. On the other hand, the first parabole works badly at the end of the range.

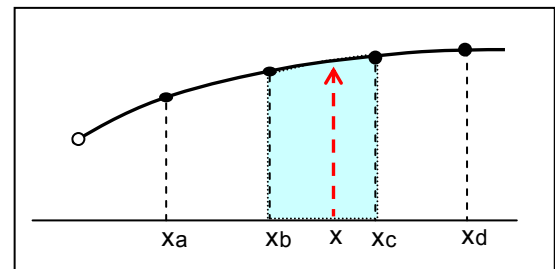
Cubic interpolation

We can choose also a cubic interpolation polynomial providing the necessary 4 knots subset.

The abscissa x should lie between the 2nd and the 3rd knots. We have to choose the 4 knots subset that satisfies to the relation

$$x_b < x \leq x_b$$

Apart for the first and the last segment, we can always choose a subset that satisfies this relation



Cubic interpolation is often chosen for its high accuracy.

The function `interpolate(x, knots, [degree])` has a third optional parameter for setting the degree of the interpolation polynomial: if omitted, the function assumes the default degree = 2 (parabolic interpolation).

Example. Repeat the interpolation at the point $x = 2.2$ with linear, parabolic and cubic polynomial for the above set of knots. Compare the error with the exact expected value given by the formula

$$f(x) = 1 + \ln(x) \cdot \sin(4x/5)$$

Comparing the error with the exact value for $x = 2.2$, we note that the cubic error is lower about 3 times than the parabolic and about 20 times than the linear interpolation one

	A	B	C	D	E	F	G
1	Knots						
2	x	y		x =	2.2		
3	1.5	1.3779093		y (linear) =	1.748983297	=Interpolate(E2;A3:B6;1;1)	
4	2	1.6928516		y (parabolic)=	1.769936871	=Interpolate(E2;A3:B6;1;2)	
5	2.5	1.8331808		y (cubic) =	1.773119055	=Interpolate(E2;A3:B6;1;3)	
6	3	1.7420722		y (expected) =	1.774386800		
7				error (linear) =	0.02540		
8				error (parab.) =	0.00445		
9				error (cubic) =	0.00127		
10							

It is evident from this example the superiority of the highest degree polynomial. But is this always true? Unfortunately not. Often, high degree does not means high interpolation accuracy. Let's read the following subject.

Instability of higher interpolation degree

For the piecewise interpolation method there is any conceptual limit in the degree of the interpolation polynomial. We can choose any degree we like providing the necessary knots subset; for 2 degree we need 3 knots; for 3 degree, 4 knots; and so on.

Generally: $\text{Degree} = \text{Knots} - 1$

On the other hand, we'll see that there are also many other things suggesting not to exceed with the degree interpolation.

One first reason concerns the Wandermode's matrix, which, increasing the dimension, is getting sharply hill-conditioned. Its solution becomes error affected that vanish the accuracy of the final result. But there is a deeper, hidden aspect: interpolation with high degree polynomials is getting unstable, especially for knots perturbed by noise, error measurements, etc.

Example. Perform the sub tabulation from 0 to 2.5 with step $\Delta x = 0.1$ of the following table with polynomial of 3rd degree and 5th degree

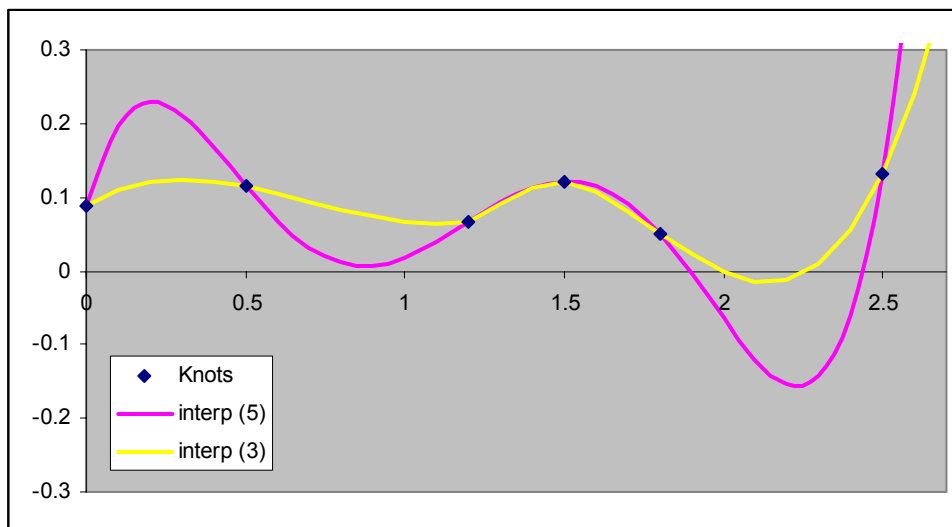
Knots	
x	y
0	0.0887048
0.5	0.11539
1.2	0.0659381
1.5	0.121927
1.8	0.0513094
2.5	0.1306148

This problem can be easily solved by the interpolate() function. The plots of the interpolation curves are shown in the following graph.

As we can see, the highest polynomial have larger oscillations between the knots, expecially at the extreme of the range.

On the contrary, the cubic polynomial seems to follow better the trend of the given samples, avoiding the instability over the critical segments [0, 0.5] and [2, 2.5]. Clearly, there are good reasons to keep low the interpolation polynomial degree .

Generally, cubic polynomial is the best compromise.



Both polynomials seem to agree at the middle of the range

This suggest that the best accuracy should be at the middle of the interpolation range. On the contrary, the extreme interpolate values would be always discharge, if possible.

Note that in the above example the knots are not equispaced. This condition takes to increase the oscillating phenomena for high interpolation degree. On the contrary, a uniform equispaced grid reduces the instability.

There is another case where it is convenient to keep low the interpolation degree: when the knots show abrupt changes of direction, due to same non-linearity of the system under observation. In this case, high degree interpolations may shown unwanted overshooting, or dumped peaks.

Example. Assume to have sampled in the range [1, 2.7] with the step $\Delta t = 0.1$ the following function.

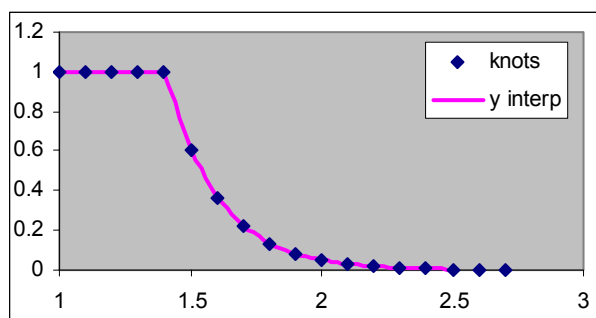
$$f(t) = \begin{cases} 1 & , \quad t < t_0 \\ e^{-k(t-t_0)} & , \quad t \geq t_0 \end{cases} \quad \text{Where:} \\ t_0 = 1.4 \quad , \quad k = 5$$

Let's perform the sub tabulation with $\Delta t = 0.2$ and with linear and parabolic interpolation

x	knots
1	1
1.1	1
1.2	1
1.3	1
1.4	1
1.5	0.606531
1.6	0.367879
1.7	0.223130
1.8	0.135335
1.9	0.082085
2	0.049787
2.1	0.030197
2.2	0.018316
2.3	0.011109
2.4	0.006738
2.5	0.004087
2.6	0.002479
2.7	0.001503

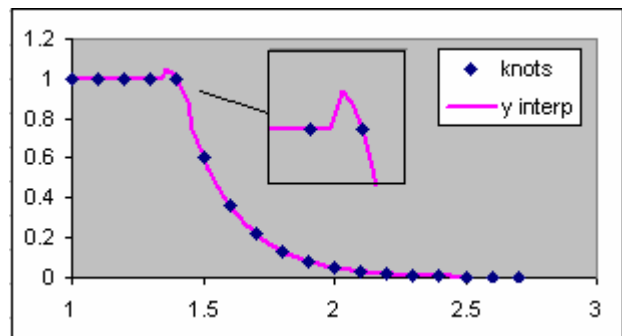
Linear interpolation.

The fit appears very good.
The interpolate curve seems to follow the original trend of the points



Parabolic interpolation

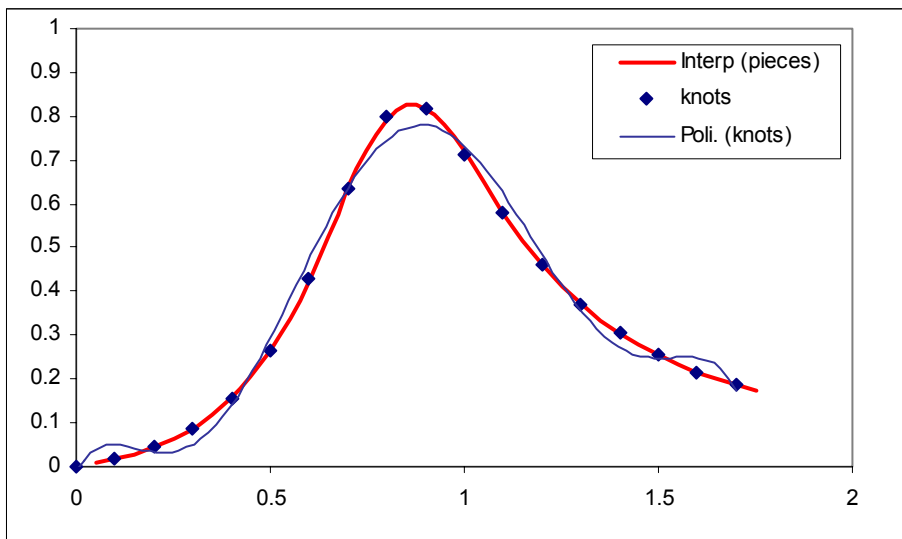
In this case we note a peak near the corner at $x = 1.4$



The parabolic interpolation shows a behavior that is not correlated to the original points. It is only due to the degree of the interpolate polynomial. As we can see, surprisingly, the best fitting, in that case, is obtained with the simplest linear interpolation. We can repeat, if we like, the interpolation with several different degree. The peak appears more dumped, but the result is substantially the same: the linear interpolation is better.

Example. Sample the following function in the range [0, 1.7] with $\Delta x = 0.1$ and plot the parabolic interpolation.

$$y = \frac{x}{1 + 10 \cdot (x - 0.8)^2}$$



The graph shows the interpolation fitting for every knot.

We have add also the global regression with 6th degree polynomial (light blu line)

As we can see the piecewise parabolic interpolation is much more accurate.

Piecewise polynomial regression schema

Many times the knots of the interpolation are affected by errors due to different sources: noise, measurement, errors calculus, etc. Often the samples occurs in random, not equispaced, grid. In this situation the exact interpolation shown in the above paragraphs may give bad results. The solution could be the polynomial regression with low-moderate degree, over many points. We hope to extract few accurate points (no worse, at least) from a set of many approximate points.

This method follows the same schema of the piecewise exact interpolation except that the subset of points chosen are more then the exact Degree+1 number. This leads to an over determined linear system that can be solved with the *Least Squares* method.

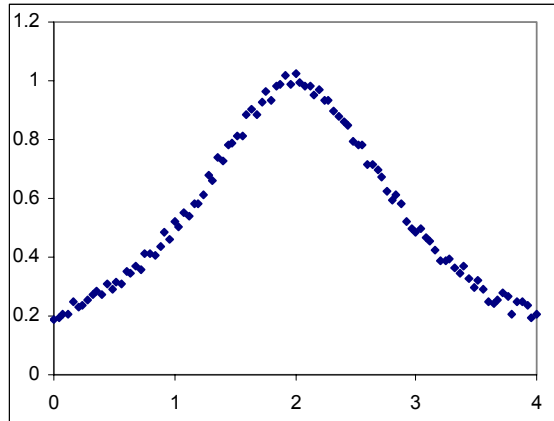
For example, for a parabolic interpolation (that require at least 3 points) we can choose the nearest 6, or 10 points. The number varies from case to case, and it is correlated to the error of to the points: great errors involve large subset of points and vice versa.

One important consequence is that the interpolation curve does not cross for any points anymore. But this is no more a problem because in this case we have assumed that every point is affected by error. Conceptually speaking there is no difference from the interpolate points and original points; they are all error prone.

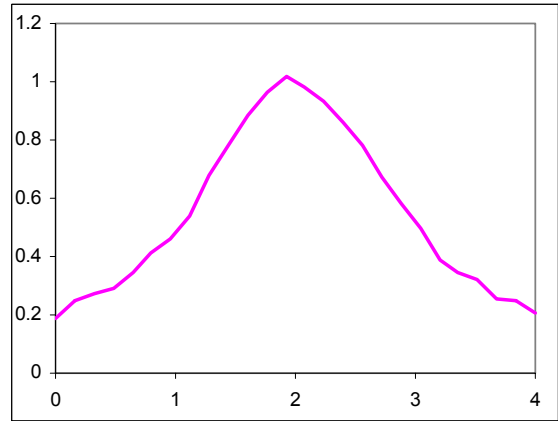
The function **Interpolate(x, knots, [degree], [points])** has a 4th optional parameter setting the number of the points to choose for the regression. If omitted, the function assumes the number equal to the polynomial degree plus one (exact interpolation)

Example. Assume to have obtained 100 samples from a measurement instrument affected by an evident noise. We plot the parabolic interpolate function obtained with 3 points (exact interpolation), 8 points and 40 points.

Row data points



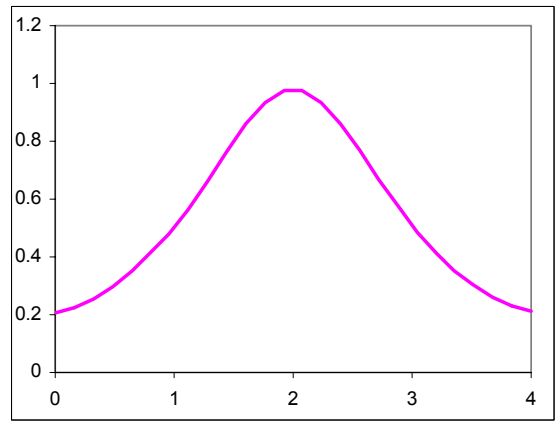
Degree = 2 , Points = 3 (exact interpolation)



Degree = 2 , Points = 8



Degree = 2 , Points = 40

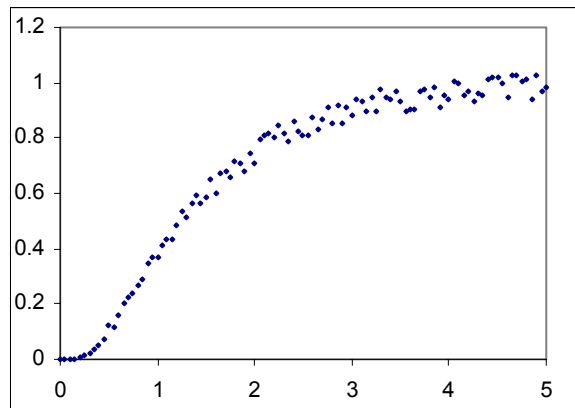


As we can see the “smoothing” effect is quite evident. We put in evidence that a larger number of points give a more smoothing curve but, on the other hand, shows the tendency to lose the original trend. We have to choose a right compromise.

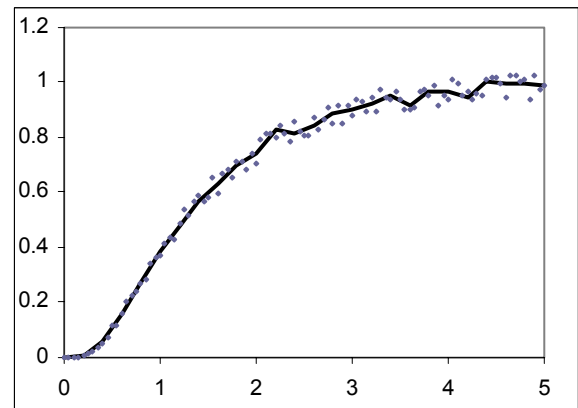
Let's see the following example

Example. Assume to have obtained 100 samples from a measurement instrument affected by an evident noise. We plot the parabolic interpolate function obtained with 3 points (exact interpolation), 8 points and 40 points.

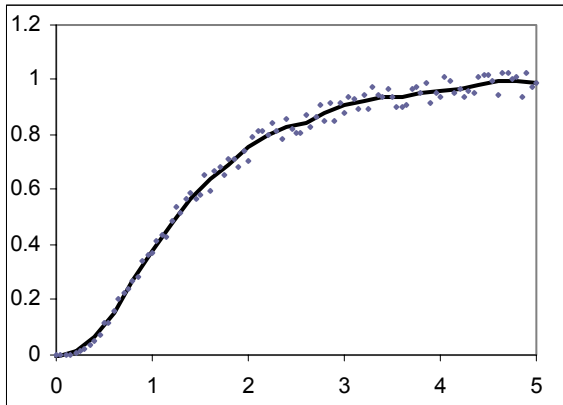
Row data points



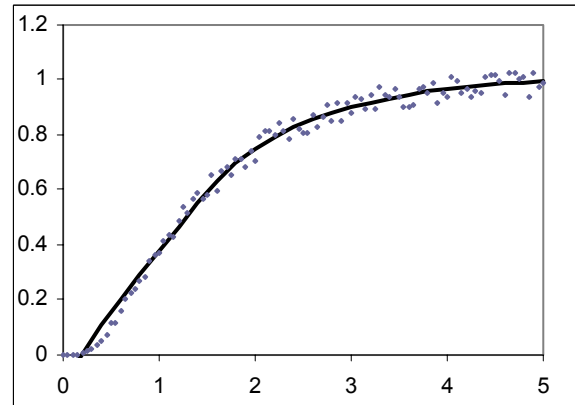
Degree = 2 , Points = 8



Degree = 2 , Points = 20



Degree = 2 , Points = 50



As we can see the 2nd piecewise regression with 20 points seems the best compromise. The 1st curve is not smoothing enough, while the 3rd curve is very smooth but near the origin it shows a different trend respect to the original points. The 2nd curve follows better the local “knee” effect near the origin of the original data.

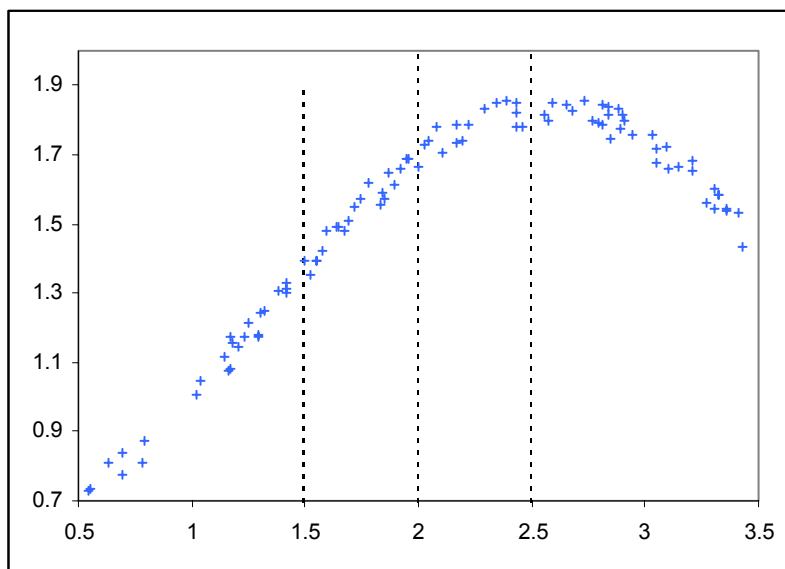
Piecewise regression versus global regression

We call “*global regression*” when we perform a polynomial regression using all points of the dataset, to distinguish from the “*piecewise regression*” that occurs when we use only a subset of the given points. As discussed in the previous sections, the two processes have different scopes even if they use the same method.

The scope of the first one is to understand which mathematical law (model) has generated the points of the dataset. We want to find a function that best approximates globally the given points.

The scope of the second regression is to find the best approximate curve that locally follows the given points, with the highest accuracy possible, no matter which formula is used.

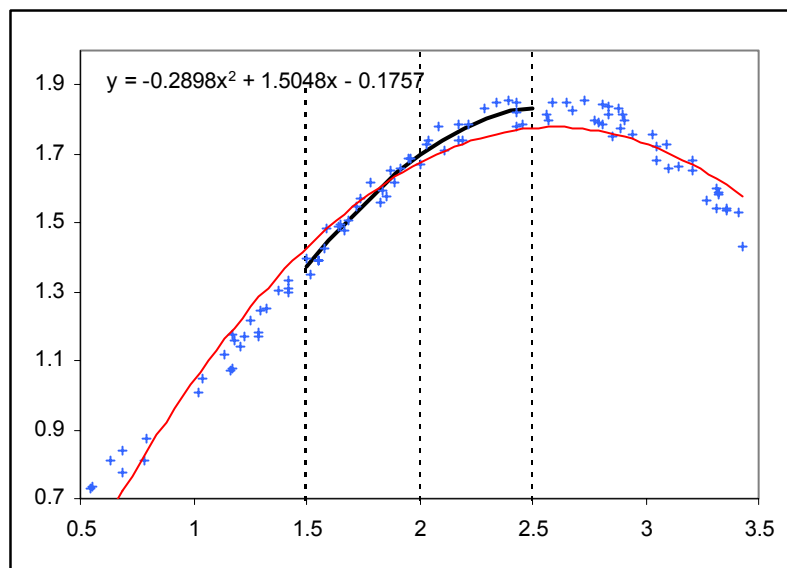
The difference from *global* and *local* fitting is shown in the following example



This scattering plot was extracted from an experimental dataset. The points are heavy perturbed by random noise.

We are interested in two problems:

- 1) find a smooth curve that best follows the trend of the points.
- 2) find a closed simplest formula (if it exists) that best fits the points




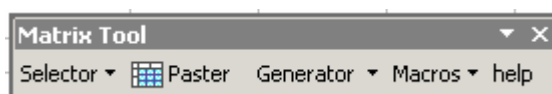
The read light continue curve is the parabole, obtained from the global regression, which equation is shown at the top of the graph. This is our math "model" of the experimental points

The tick dark curve is obtained by the parabolic regression taking the points falling in the strip between 1.5 and 2.5. In this range, this curve follows better the hazy trend of the points

Matrix Tool

Matrix tool menubar

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



matrix.xla v.1.6

Menu macros:

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

Selector tool.

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

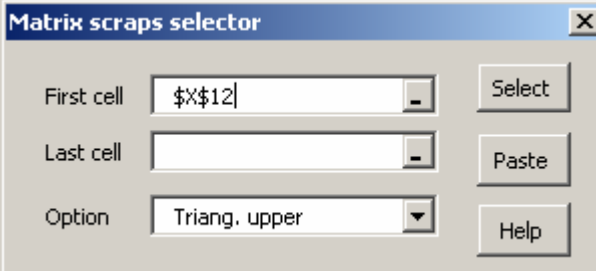
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>0</td><td>3</td><td>6</td><td>2</td><td>1</td><td>0</td></tr><tr><td>-1</td><td>4</td><td>9</td><td>0</td><td>3</td><td>-6</td></tr><tr><td>-2</td><td>5</td><td>12</td><td>-2</td><td>7</td><td>-1</td></tr><tr><td>-3</td><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></td><td>4</td><td>8</td><td>7</td><td>1</td><td>-6</td><td>4</td></tr><tr><td></td><td>10</td><td>3</td><td>9</td><td>7</td><td>-7</td><td>3</td></tr><tr><td></td><td>-4</td><td>9</td><td>-2</td><td>3</td><td>1</td><td>8</td></tr><tr><td></td><td>-9</td><td>-8</td><td>5</td><td>-3</td><td>-4</td><td>3</td></tr><tr><td></td><td>-10</td><td>4</td><td>-7</td><td>4</td><td>-10</td><td>-9</td></tr><tr><td></td><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 cell that usually correspond to the full matrix. If you want to restrict the area simply select the sub-matrix that you want before starting the *Selector* macro

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

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

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

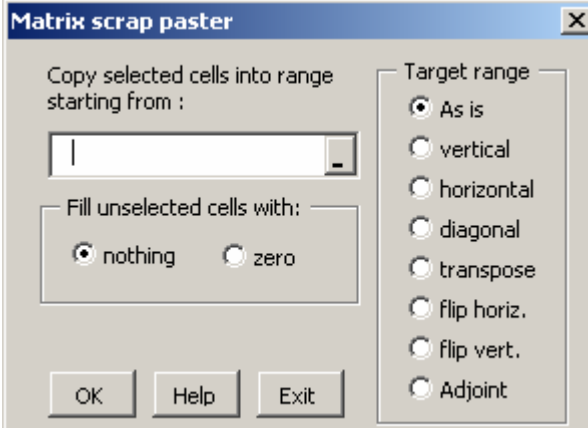


The dialog box titled "Matrix scraps selector" has a close button (X) in the top right. It contains three input fields: "First cell" with the text "\$X\$12", "Last cell" which is empty, and "Option" with a dropdown menu showing "Triang. upper". To the right of each input field is a button: "Select" for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

Scraps Paster tool.

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

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



The dialog box titled "Matrix scrap paster" has a close button (X) in the top right. It is divided into two main sections. The left section, "Copy selected cells into range starting from :", has a text input field. Below it, "Fill unselected cells with:" has two radio buttons: "nothing" (selected) and "zero". The right section, "Target range", has a group box containing several radio button options: "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 element of another one, you can use this simply option and the result will be similar to the following:

1	2	3	4	5	6	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 sometime we need to reformat the geometry of the given range. This happens, for example when we want to extract the diagonal elements from a given matrix and to convert it in a vertical vector.

In this case, after you have selected the diagonal, check the option *vertical*

The diagonal element will be... "verticalized". Fine, isn't?

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

Some time we need the inverse of this transformation: from a vertical vector, we have to build the diagonal matrix having the vector elements on its diagonal.

For that select the vector that contains the elements. Start the *Scraps Paster* tool and check the *zero* and *diagonal* option.

Giving OK, you will generate the matrix to the left

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 submatrix.

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	4	8	1	-6	4
10	3	9	7	-7	3	10	3	7	-7	3
-4	9	-2	3	1	8	-9	-8	-3	-4	3
-9	-8	5	-3	-4	3	-10	4	4	-10	-9
-10	4	-7	4	-10	-9	-5	-2	5	1	0
-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	13	-12	-1	-14
-1	4	7	4	7	-10	0	-11
-5	-6	2	-5	1	-8	1	-8
1	-8	1	-8	-5	-6	2	-5
7	-10	0	-11	-1	4	7	4
13	-12	-1	-14	-10	-8	7	-4

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

	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 the B2 as target corner and *Transpose* as option. The result will be the transpose 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	Generate random matrices with the following parameter: dimension, max e min values, format: full, triangular, tridiagonal, symmetric, decimals number.
Rank / Determinant	Generate random matrices with given rank or determinant
Eigenvalues	Generate random matrices having given eigenvalues
Hilbert	Generate Hilbert's matrices of given dimension
Tartaglia	Generate Tartaglia's matrices of given dimension

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

Random matrix with given format

Parameters:

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

Max value: upper limit of **x**

Min value: lower limit of **x**

Decimals: fix the decimals of **x**

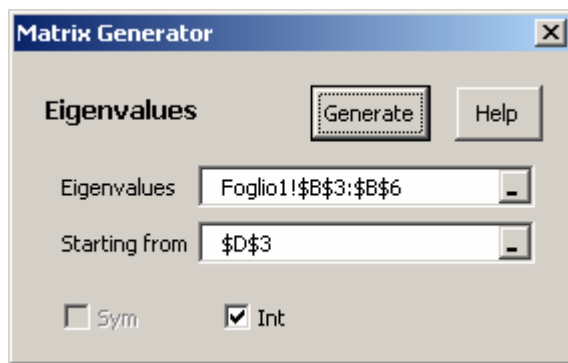
Int checkbox: numbers **x** are integers

Sym checkbox: the matrix will be symmetric

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

Starting from: top-left matrix corner

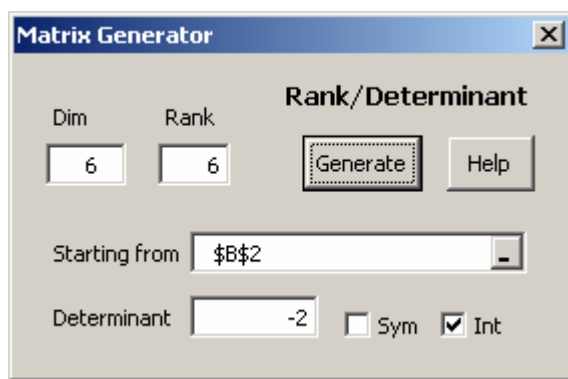
Random matrix with given eigenvalues



Eigenvalues: vertical range containing the real eigenvalues.

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

Random matrix with given determinant or rank



Generate a square matrix with given determinant or rank

Dimension: matrix dimension

Rank: rank of the matrix. For default is set equal to the dimension. If it is less than the dimension, the determinant is automatically set to 0.

Starting from: top-left matrix corner

Determinant: sets the determinant of the random matrix

Sym: generate a symmetric random matrix
Int: generate a random matrix with integer values

Tartaglia's matrix

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

Hilbert's matrix

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

Macros stuff.

This menu contains several macros performing usefull tasks.

Macro versus Function

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



Macros available are:

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
Singular values	Singular Values Decomposition

Macro Gauss-step-by-step

This macro performs all steps to reduce a given matrix into a triangular or diagonal form by the Gauss and Gauss-Jordan algorithm.

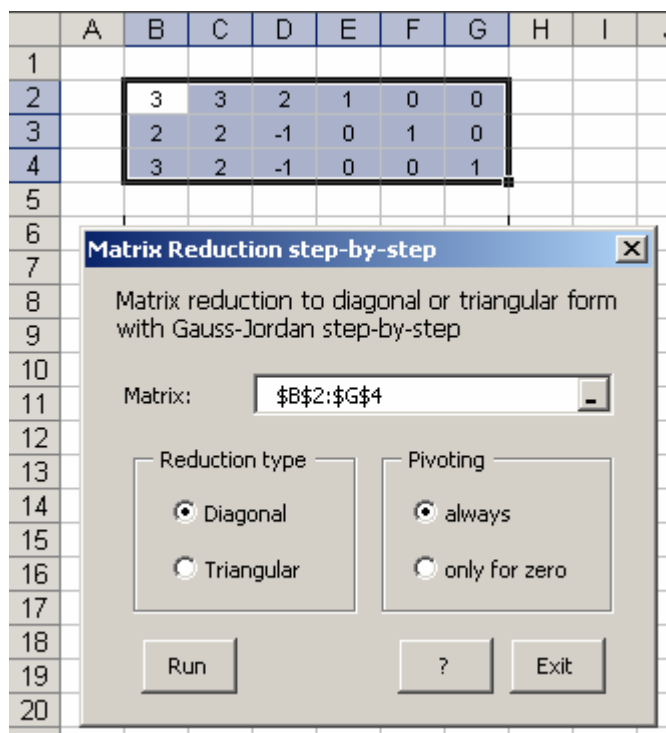
This macro works like the function **Gauss_Jordan_step** except that it traces all multiplication coefficients as well all the swap actions.

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

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

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



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

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

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

The process are traced below the original matrix.

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

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

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

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

...as well the exchange action

Of course also the determinant changes sign.

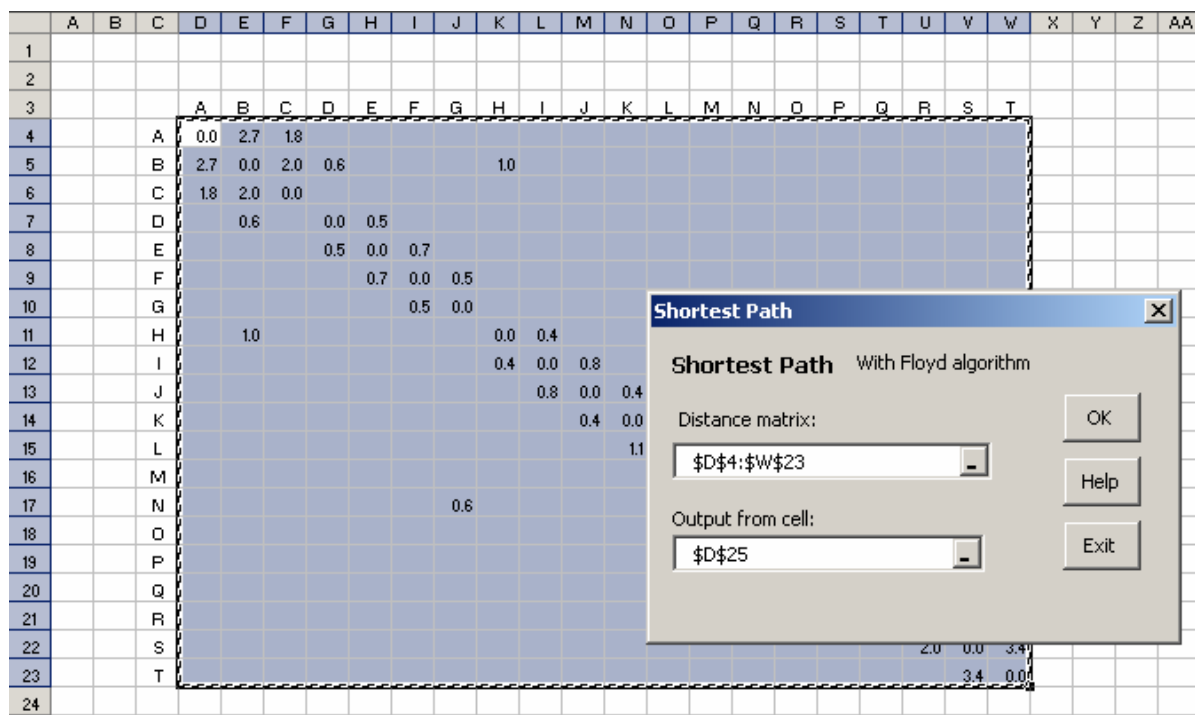
For further examples see the chapter *Gauss-Jordan algorithm*.

Macro Shortest-Path

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

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

In the example below we see a 20 x 20 distance matrix



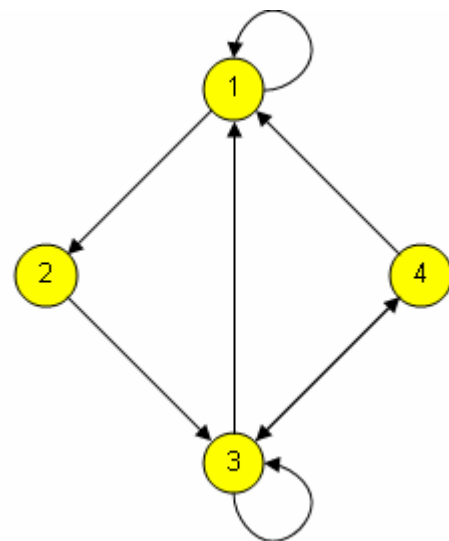
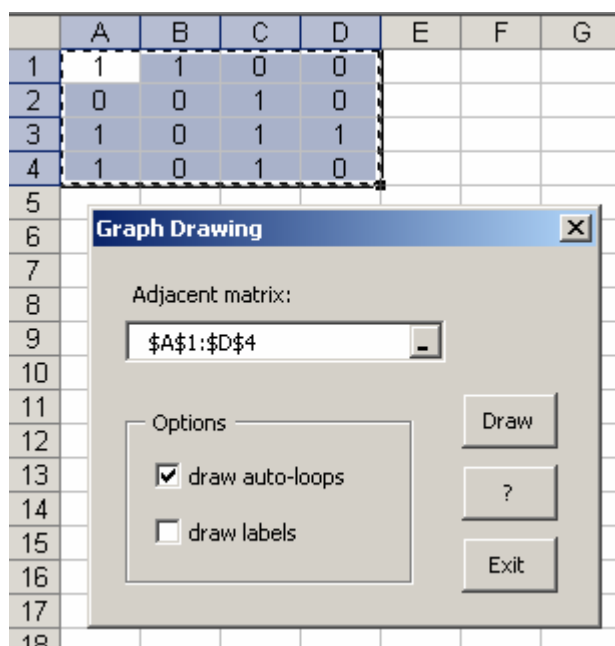
For default, the output matrix begin from cell D5, just below the input matrix.

Macro Draw Graph

This usefull stuff draws a simple graph from its adjacent matrix. (There is now equivalent function for this macro)

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

>macros > Graph > Draw



Macro Block reduction

This macro transforms a square sparse matrix into a block-partioned form using the score-algorithm. This macro works like the functions **Mat_Blok** and **Mat_BlokPerm** except it is more adapt for large matrices.

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

>macros > Block reduction

	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

Limits in matrix computation

One recurrent question about matrices computation is: - what is the max dimension for a matrix operation, for example the determinant, or inversion? -

Well, the right answer should be: it depends. Many factors, such as hardware configuration, algorithm, software coding, operating system and - of course - the matrix itself, contribute to limit the max dimension. One sure thing is that the limit is not fixed at all.

In the past, the main limitation was memory and elaboration speed, but nowadays these factors are not more a limit.

We can say that, for the standard PC, the main limitation is due to the 32-bit arithmetic and to the matrix itself.

Suppose you have a dense matrix ($n \times n$) with its elements a_{ij} randomly distributed from $-k$ to k . With this hypothesis the determinant grows roughly as:

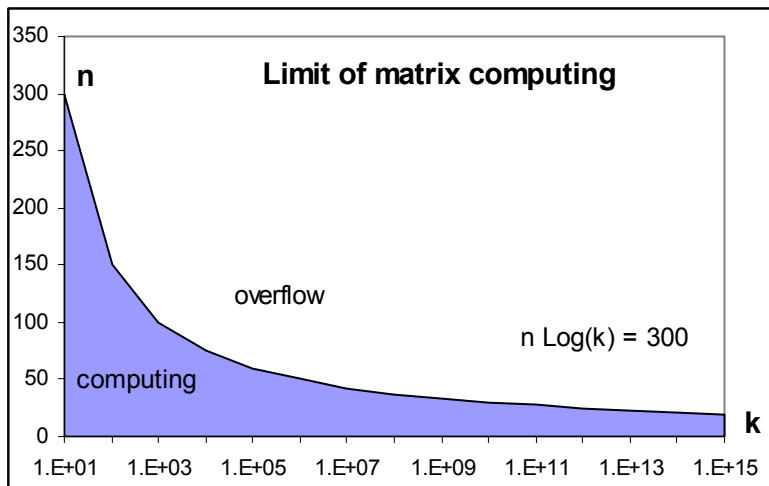
$$\text{Log}(|D|) \cong n \text{Log}(k) + 0.0027 \cdot n^2 \cong n \text{Log}(k)$$

where Log is decimal logarithm, n is the dimension of the matrix, k its max value

In 32 bit double precision the max value allowed is about $1\text{E}+300$, $1\text{E}-300$. So if we want to avoid the overflow/underflow error, we must constrain:

$$300 \geq n \text{Log}(k) \quad (1)$$

If we plot this relation for all points (k, n) we have the area for computing (blue area in the graph below). On the contrary, the dangerous error area is the remain (white) area



How does it work?

Simple. If you have to compute the determinant of a matrix (80×80) having values no more than 1000, the point $(1000, 80)$ falls into the blue area; so you will be able to perform this operation.

On the contrary, if you have a matrix (80×80) having values up to $1\text{E}+7$, the point $(1\text{E}+7, 80)$ falls into the white area; so you will probably get an overflow error.

From this graph we see that matrices (25×25) or less, can be elaborated for all values, while matrices (100×100) or more can be computed only if their values are less than 1000.

Of course this result is valid only for generic dense matrices no ill-conditioned. If the matrix is ill-conditioned you could get an overflow/underflow error even for low/moderate values. Fortunately, there are also special kind of matrices that can be elaborated even if the constrain (1) is false. We speak about diagonal, tridiagonal, sparse, block matrices etc.

We have to say that, avoiding the overflow error is not sufficient to get a good result. We have to take care, specially for large matrices, to the round-off errors. They are quite lay and very difficult to detect also. Very often the result of large matrix inversion is take good even if it is completely wrong.

If you think that this errors regard only large matrices, have a look to the following example:

Compute the numeric inverse of this simple (3 x 3) matrix

127	-507	245
-507	2025	-987
245	-987	553

If you use a in 32-bit standard precision program on your PC, the answer probably looks like the following:

-2.121E+14	-5.614E+13	-6.238E+12
-5.614E+13	-1.486E+13	-1.651E+12
-6.238E+12	-1.651E+12	-1.835E+11

And the determinant? You probably get a results near to $DET = -6.867E-10$

If you repeat the calculus with other programs you get similar results. Is there any reasons for suspecting this results? Yes, because this result is completely wrong !.

In fact, the exact determinant is 0, the given matrix is singular and its inverse, simply does not exist (you can easily compute by hand with exact fractional numbers. If you are lazy see Step-by-step matrix inversion with Gauss-Jordan algorithm)

In this case it was easy to analyze the matrix, but for a larger matrix (50 x 50) do you know what would happen? Before to accept any results - specially for large matrices -we have to do same extra test. In the example above we have to examine the SVD decomposition, that gives the following diagonal matrix:

2646.049	0	0
0	58.9513	0
0	0	4.87038E-14

The last element is very near to the machine accuracy $1E-15$, if we get the ratio between the lowest and the highest value we have:

$$m = 4.87038E-14 / 2646.049 = 1.8406E-17 \ll 1E-15$$

The ratio is more less than machine accuracy , so we have to conclude that the matrix D, "numerically specking" has one zero on the diagonal meaning that the given matrix is singular

Functions Reference

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

Gauss_Jordan_step	Mat_Hilbert	MatPerm
Gram_Schmidt	Mat_Householder	MatRnd
Interpolate	Mat_Leontief	MatRndEig
M_ABS	Mat_LU	MatRndEigSym
M_ADD	Mat_QR	MatRndRank
M_BAB	Mat_QR_iter	MatRndSim
M_DET	Mat_Tartaglia	MatRot
M_DET_C	Mat_Vandermonde	MatRotation_Jacobi
M_DET3	MatCharPoly	Path_Floyd
M_DIAG	MatCmpn	Path_Min
M_DIAG_ERR	MatCorr	Poly_Roots
M_EXP	MatCovar	Poly_Roots_QR
M_EXP_ERR	MatDiagExtr	ProdScal
M_ID	MatEigenvalue_Jacobi	ProdScal_C
M_INV	MatEigenvalue_max	ProdVect
M_INV_C	MatEigenvalue_pow	REGRL
M_MULT_C	MatEigenvalue_QL	REGRP
M_MULT3	MatEigenvalue_QR	RRMS
M_POW	MatEigenvalue_TridUni	Simplex
M_PROD	MatEigenvector	SVD
M_PROD_S	MatEigenvector_C	SYSLIN
M_RANK	MatEigenvector_inv	SYSLIN_C
M_SUB	MatEigenvector_Jacobi	SYSLIN_ITER_G
M_T	MatEigenvector_max	SYSLIN_ITER_J
M_TRAC	MatEigenvector_pow	SYSLIN_T
M_TRIA_ERR	MatEigenvector3	SYSLIN3
Mat_Adm	MatExtract	SYSLINSING
Mat_Block	MatMopUp	TRASFLIN
Mat_BlockParm	MatNorm	VarimaxIndex
Mat_Cholesky	MatNormalize	VarimaxRot
Mat_Hessemberg	MatOrtNorm	

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

Returns the determinant of a square matrix (n x n).

Optional parameters are:

IMODE switch (True/False) sets the floating point (False) or integer computing (True). Default is false. Integer computing is intrinsically more accurate but also more limited because of overflow error. Use it only with integer matrices of moderate size.

Optional parameter **Tiny** (default is 0) sets the minimum round-off error; any value in absolute less than Tiny will be set to 0.

For a n = 1

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

For a n= 2

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

For a n= 3

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

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

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

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

Function M_DET_C (Mat, [Cformat])

This function computes the determinant of a complex matrix.

The argument Mat is an array (n x n) or (n x 2n), depending of the format parameter

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

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

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

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

	A	B	C	D	E	F	G	H	I	J
1										
2	1	2	3	0	1	-1				
3	-1	1	2	1	0	1				
4	2	0	-1	2	-1	0				
5										
6	1	0	2	1	3	-1				
7	-1	1	1	0	2	1				
8	2	2	0	-1	-1	0				
9										
10	1	2+i	3-i							
11	-1+i	1	2+i							
12	2+2i	-i	-1							
13										

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

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

The last example shows the rectangular *string* format

Function M_DET3(Mat3)

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

A triangular matrix is:

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

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

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

	A	B	C	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		
21					

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

Returns the matrix inverse of a given square matrix:

Optional parameters are:

IMODE switch (True/False) sets the floating point (False) or integer computing (True). Default is False. Integer computing is intrinsically more accurate but also more limited because of overflow error. Use only with integer matrices of moderate size.

Optional parameter **Tiny** (default is 0) sets the minimum round-off error; any value in absolute less than Tiny will be set to 0.

If the matrix is singular the function returns "singular"

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

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

For definition:

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

Use CTRL+SHIFT+ENTER to insert this function

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

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

Function M_POW(Mat, n)

Returns the integer power of a given square matrix

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

Use CTRL+SHIFT+ENTER to insert this function

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

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

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

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

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

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

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

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

The second one, more efficient, uses the Padè approximation. It is recommendable especially for larger matrices. Gregory Klein, who kindly consented to add it to this package, originally developed this nice routine.

You can switch the algorithm used by the optional parameter *Algo*. If "P" (default) the function uses the Padè approximation; else uses the power's series

Use the CTRL+SHIFT+ENTER key sequence to insert it

Function M_EXP_ERR(A, n)

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

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

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

Function M_PROD(Mat1, Mat2, ...)

Returns the product of two or more matrices

$$C = A \cdot B$$

Use CTRL+SHIFT+ENTER to insert this function

As know the product is defined:

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

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

Dimension rule: If Mat1 is (n x m) and Mat2 is (m x p), then the product is a matrix (n x p)

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

Note: If Mat1 and Mat2 are square (n x n) matrices, also the product is a square (n x n) matrix.

Writing out the product explicitly

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

Where:

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

Matrix multiplication is associative. Thus:

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

But, generally, is not commutative:

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

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

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

The screenshot shows an Excel spreadsheet with the following data:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	A				B			C			X					
2	1	2	5		1	4		-1	-2		:I3)					
3	-1	3	6		2	5		5	1							
4					3	6		.								

The formula bar shows: `=M_PROD(A2:C3;E2:F4;H2:I3)`

The M_PROD function dialog box is open, showing the following matrices and their dimensions:

- Mat: A2:C3 = {1;2;5;-1;3;6}
- ... E2:F4 = {1;4;2;5;3;6}
- ... H2:I3 = {-1;-2;5;1}
- ... =

The result of the multiplication is shown in cell K2 as {200;4;212;1}.

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

Function M_PROD_S(Mat, k)

Returns a matrix multiplied for a scalar

For example:

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

This simple function can be also used as nested argument

For example

If the range A1:B2 contains a matrix 2x2 [1 , 2, / -3 , 8]

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

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

Function M_MULT3(Mat3, Mat)

This function performs the multiplication of a 3-diagonal matrix and a vector or a rectangular matrix

Mat3 is a (n x 3) array

Mat can be a vector (n x 1) or even a rectangular matrix (n x m)

Result is a vector (n x1) or a matrix (n x m)

Here is an example of a 5x5 tridiagonal matrix

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

Example

The diagonal and sub diagonals are passed to the function as vertical vector

This function is useful when you have to multiply a tridiagonal matrix larger than 256 x 256 for a vector because Excel cannot manage matrices larger than 256 columns.

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

	A	B	C	D	E	F	G	H
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	{=M_MULT3(A3:C18,E3:E18)}							
21								
22								

Function M_SUB(Mat1, Mat2)

Returns the different of two matrices

$$C = A_1 - A_2$$

Excel can perform different of matrices in a very efficient way by array-arithmetic (see example below). The benefit of this function comes when we have to subtract matrices inside another function. For example when we have to compare two matrices.

Function M_TRAC(Mat)

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

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

		G2		=	=M_TRAC(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 M_DIAG(Diag)

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

Example:

		C1		=	{=M_DIAG(A1:A6)}					
	A	B	C	D	E	F	G	H	I	
1	-1		-1	0	0	0	0	0		
2	-2		0	-2	0	0	0	0		
3	-3		0	0	-3	0	0	0		
4	-4		0	0	0	-4	0	0		
5	-5		0	0	0	0	-5	0		
6	-6		0	0	0	0	0	-6		
7			{=M_DIAG(A1:A6)}							
8										

Function MatDiagExtr(Mat, [Diag])

This function extracts the diagonals of a matrix

The optional parameter Diag sets the diagonal to extract.

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

(Thanks to Giacomo Bruzzo' idea)

	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							

{=MatDiagExtr(A2:D5)}

{=MatDiagExtr(A2:D5;2)}

Useful isn't it?

Function M_T(Mat)

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

E2				= {=M_T(A2:C4)}			
	A	B	C	D	E	F	G
1	A				A ^T		
2	0	1	0		0	0	3
3	0	-1	2		1	-1	-9
4	3	-9	7		0	2	7
5							

This function is identical to TRANSPOSE() Excel built-in function
Use CTRL+SHIFT+ENTER to insert this function

Function M_RANK(A)

Returns the rank of a given matrix

It computes the sub-space of $Ax = 0$, using the SYSLINSING function: then counts null column-vectors of sub-space.

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

Examples:

[illegible]

	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	=M_RANK(B2:E4)		

When $\text{Det} = 0$ 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 3×4 matrix the maximum rank is 3.

Function M_DIAG_ERR(A)

Diagonalization error of a given square matrix

This function computes and returns the mean of the absolute values out of the first diagonal

It is useful to measure the "distance" from the diagonal form of a square matrix

	A	B	C	D
1	How long is this matrix from the			
2	diagonal form?			
3	1	8.99E-10	9.20E-10	
4	2.23E-10	2	-3.40E-09	
5	-1.00E-09	5.23E-10	3	
6				
7	Diagonalization error		M_DIAG_ERR(B3:D5)	
8	1.16E-09			
9				

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

Function M_TRIA_ERR(A)

Returns the "triangularization" error of a given square triangular matrix

This function computes and returns the minimum of the mean of the absolute values of the upper and lower element out of the first diagonal

It is useful to measure the "distance" from the triangular form of a square matrix

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

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

	A	B	C	D	E	F
1	1	1E-05	-6E-05			
2	-1	2	3E-05		3.5E-05	
3	-6	2	3			
4						
5	8	1	0.5			
6	-2E-06	-3	-0.25		1E-06	
7	0	1E-06	10			
8						

Formulas:

- Cell D4: `=M_TRIA_ERR(A1:C3)`
- Cell D7: `=M_TRIA_ERR(A5:C7)`

Function M_ID()

Returns the identity square matrix into the cells that you have selected

This function can also be used in nested expression like for example $\mathbf{A} - \lambda \mathbf{I}$

B2 **=M_ID()**

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

Formula: `{=M_ID()}`

	A	B	C	D	E	F	G	H
1								
2		2	-5	3		-1	-5	3
3		-1	-12	-1		-1	-15	-1
4		0	4	7		0	4	4
5								
6		$\lambda =$	3					
7								
8								

Formulas:

- Cell H4: `{=B2:D4-C6*M_ID()}`

Use CTRL+SHIFT+ENTER to insert this function

Function ProdScal(v1, v2)

Returns the scalar product of two vectors

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

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

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

Note that if V1 and V2 are perpendicular, the scalar product is zero. In fact another definition of scalar product is:

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

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

Vectors can be in vertical or horizontal format as well.

Function ProdVect(v1, v2)

Returns the vector product of two vectors of 3 dimension

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

Use CTRL+SHIFT+ENTER to insert this function

Note that if V1 and V2 are parallels, the vector product is the null vector.

	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						

Vectors can be in vertical or horizontal form as well.

Function MatEigenvalue_Jacobi(Mat, Optional MaxLoops)

This function performs the Jacobi's sequence of orthogonal similarity transformation and returns the last matrix of the sequence. It works only for symmetric matrices.

The optional parameter MaxLoops (default=100) sets the max steps of the sequence.

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

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

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

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

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

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

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:

MatEigenvalue_Jacobi
MatEigenvector_Jacobi
MatRotation_Jacobi

Use CTRL+SHIFT+ENTER to insert this function

Example: Solve the eigenvalues 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	{=MatEigenvalue_Jacobi(A3:C5;B2)}			
4	2	6	2	{=MatEigenvector_Jacobi(A3:C5;B2)}			
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's method has found all Eigenvalues and Eigenvectors with few iterations (10 loops)

The function **MatEigenvalue_Jacobi()** returns in range A7:C9 the diagonal matrix of the eigenvalues.

Note that elements out of the first diagonal have an error less than 10^{-16} .

At the right side - in the range D7:F9 - the function MatEigenvector_Jacobi() returns the orthogonal matrix of the eigenvectors

Comparing with the exact solution

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

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

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

You can use the standard matrix inversion and multiplication functions or the **M_BAB()** function also in this package, as you like.

Note that - only in this case - the inversion of matrix is very simple, because:

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

Eigenvalues problem with Jacobi step by step

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

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

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

Each matrix is one step of Jacobi's Iterative method

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

To obtain quickly the above iterations follow these simple rules:

At the first time, insert in range A7:C9 the function **MatEigenvalue_Jacobi(A3:A7)**.

Give the "magic" key sequence CTRL+SHIFT+ENTER to paste an array function

Leave the range A7:C9 selected and copy it (CTRL+C)

Select the following range A11 and paste it (CTRL+V)

Copy it (CTRL+C)

Select the following range A15 and paste it (CTRL+V)

And so on.

By the sequence -copying and pasting- you can perform all Jacobi's iterations, as you like.

In the middle, we see the Jacobi's rotation matrices sequence. We can easily obtain it by the function *MatRotation_Jacobi()* in the same way as eigenvalues matrix

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

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

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

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

$$U_1 = P_1$$

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

Function MatRotation_Jacobi(Mat)

This function returns the Jacobi's orthogonal rotation matrix of a given symmetric matrix.

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

$$A_1 = P_1^T A \cdot P_1 \quad \text{Where:} \quad P1 = \text{MatRotation_Jacobi}(A)$$

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

Example - find the rotation matrix that makes zero the highest no-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

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

The rotation matrix, in that case is

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

Where the angle α is given by the formula:

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

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3												

Function **Mat_BlockParm(Mat,)**

Returns the permutation matrix that transforms a sparse square matrix ($n \times n$) into a block-partioned matrix . Under certain conditions, a square matrix can be transformed into a block-partioned form (also called block-triangular form) by similarity transformation.

$$B = P^T A P$$

where P is a permutation matrix ($n \times n$).

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

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

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
1																							
2																							
3																							
4																							
5																							
6																							
7																							
8																							
9																							
10																							
11																							
12																							
13																							

A

1	0	0	1	2	0
1	-1	1	2	-3	-2
-6	1	1	3	5	2
1	0	0	3	-1	0
2	0	0	5	1	0
-9	2	1	1	7	1

`{=Mat_BlokPerm(B3:G8)}`

P

1	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	0	0	0	0	1

`{=MatPerm(I3:I8)}`

P^TAP

1	2	1	0	0	0
2	1	5	0	0	0
1	-1	3	0	0	0
-6	5	3	1	1	2
1	-3	2	1	-1	-2
-9	7	1	1	2	1

`{=M_PROD(M_T(K3:P8);B3:G8;K3:P8)}`

Note that not all matrices can be transformed in block-triangular form. If the transformation fails the function returns “?”. This usually happens if the matrix is irreducible. A dense matrix without zero is, of course, always irreducible. But this happens also for many sparse matrices like, for example the following matrix.

The screenshot shows a spreadsheet with the following data:

	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															
10		?	?	?	?	?	?								
11															

Annotations in the image:

- A blue box labeled "Irriducible matrix" points to the 7x7 matrix in cells B2:G7.
- A blue box labeled "Permuatation" (sic) points to the row of question marks in cells A10:G10.
- A formula bar shows the formula: `=Mat_BlokPerm(B2:G7))`.

Irreducible matrix

Function MatEigenvalue_QR(Mat)

This function performs the diagonal reduction of a given matrix with the generalized QR method, and returns the approximate eigenvalues real or complex.

It returns a n x 2 array

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

F2		fx {=MatEigenvalue_QR(A2:C4)}								
	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										

{=MatEigenvalue_QR(A2:C4)}

References

This function uses a reduction of the LAPACK FORTRAN HQR and ELMHES subroutines (April 1983)

HQR IS A TRANSLATION OF THE ALGOL PROCEDURE

NUM. MATH. 14, 219-231(1970) BY MARTIN, PETERS, AND WILKINSON.

HANDBOOK FOR AUTO. COMP., VOL.II-LINEAR ALGEBRA, 359-371(1971).

ELMHES IS A TRANSLATION OF THE ALGOL PROCEDURE,

NUM. MATH. 12, 349-368(1968) BY MARTIN AND WILKINSON.

HANDBOOK FOR AUTO. COMP., VOL.II-LINEAR ALGEBRA, 339-358(1971).

Example.

Find all eigenvalues of the following symmetric matrix. Being symmetric there are only n real distinct eigenvalues. So the function return only an array n x 1

J2		fx {=MatEigenvalue_QR(A2:H9)}									
	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											

{=MatEigenvalue_QR(A1:H8)}

Function MatEigenvector(A, Eigenvalues, [MaxErr])

This function returns the eigenvector of associate eigenvalue of a matrix A (n x n)

$$Av = \lambda v$$

If Eigenvalues is a single value, the function returns a (n x 1) vector. Otherwise if Eigenvalues is a vector of all eigenvalues of matrix A, the function returns a matrix (n x n) of eigenvector.

Note: the eigenvector 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 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

For complex eigenvalues see: Function MatEigenvector_C(Mat, Eigenvalues, [MaxErr])

Function MatEigenvector_C(A, Eigenvalue, [MaxErr])

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

$$Av = \lambda v$$

The function returns an array of two columns (n x 2) if the eigenvalue is simple: the first column contains the real part, the second column contains 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 eigenvalues are affected by an error. In that case the MaxErr should be proportionally adapted. Otherwise the result may be a NULL matrix.

Look at this example:

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

The given matrix has 3 eigenvalues:

2 , 5+j, 5-j

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

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

Note also that for conjugate eigenvalues will get also conjugate eigenvectors

For real eigenvalues see also Function MatEigenvector(Mat, Eigenvalues)

Function MatEigenvector_Jacobi(Mat, Optional MaxLoops)

This function performs the Jacobi's sequence of orthogonal similarity transformation and returns the last orthogonal matrix of the sequence. 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

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

The matrix U_n is composed by the eigenvectors of **A**

E2	= {=MatEigenvector_Jacobi(A2:C4)}							
	A	B	C	D	E	F	G	H
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 MatEigenvalue_Jacobi(Mat, Optional MaxLoops)

Function MatEigenvalue_QL(Mat3, [IterMax])

Return all real eigenvalues of a tridiagonal symmetric matrix. It works also for unsymmetrical tridiagonal matrix having all eigenvalues real

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

This function use the efficient QL algorithm

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

TUTORIAL FOR MATRIX.XLA

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

Note that the rectangular form (n x 3) is very usefull for large matrices.

	A	B	C	D	E	F	G	H
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	{=MatEigenvalue_QL(A10:C14)}							
8								
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	{=MatEigenvalue_QL(A10:C14)}							
16								
17								

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

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

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

$$0.8 < \lambda_k < 1.2$$

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

	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	{=MatEigenvalue_QL(A2:C20)}				
22					
23					

Function MatEigenvalue_TridUni(n, a, b, c)

Return all eigenvalues of a tridiagonal uniform matrix $n \times n$.

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

It was be demonstrated that for these matrices:

- with n even - all eigenvalues are real if $a*c > 0$; all eigenvalues are complex otherwise.
- with 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 uniform tridiagonal matrix, there is a nice close formula giving all eigenvalues for any size of the matrix dimension. See chapter "Tridiagonal uniform mayrix".

Example :

find the eigenvalues of the 40×40 tridiagonal uniform matrix having $a = 1$, $b = 3$, $c = 2$.
Because $ac = 2 > 0$, then all eigenvalues are real.

F2	A	B	C	D	E	F	G	H	I	J
1	n	a	b	c		Eigenvalues				
2	40	1	3	2		0.179872043				
3						0.204720843				
4						0.245973452				
5						0.303387784				
6						6909				
7						0.46526103				
8						0.559770000				

Function MatEigenvector3(Mat3, Eigenvalues, [MaxErr])

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

$$Av = \lambda v$$

If *Eigenvalues* is a single value, the function returns a $(n \times 1)$ vector. Otherwise if the parameter *Eigenvalues* is a vector of all eigenvalues of matrix A, the function returns a matrix $(n \times n)$ 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 $(n \times n)$ matrices and $(n \times 3)$ rectangular matrices. The second form is useful for large matrices.

Example.

Given the 19×19 tridiagonal matrix having eigenvalue $L = 1$, find its associate eigenvector

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

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

Function MatCharPoly(Mat)

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

	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						

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

See also Function Poly_Roots(Coefficients, [ErrMax])

Function Poly_Roots(Coefficients, [ErrMax])

This function returns all roots of a given polynomial

Coefficients parameter is an array of n+1 polynomial coefficients

ErrMax optional parameter sets the max error for roots approximation (default = 1E-13)

This function use the *Lin-Bairstow* algorithm for factoring a polynomial into a square polynomial and a n-2 degree polynomial

The process is applied recursively to the n-2 polynomial, and so on, until the reduce polynomial degree becomes 2° or 1°.

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

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

Example: find all eigenvalues of the following matrix

	A	B	C	D	E	F	G
1	Matrix A				coeff	eigenvalues	
2					52	real	imm
3	9	-6	7		-46	2	0
4	1	4	1		12	5	1
5	-3	4	-1		-1	5	-1
6							
7					{=MatCharPoly(A3:C5)}		
8							
9					{=Poly_Roots(E2:E5)}		
10							

Note: we can also use it in nested functions like:

=Poly_Roots(MatCharPoly(A))

Function MatEigenvalue_max(Mat, [IterMax])

Returns the dominant eigenvalue of a matrix.

Dominant eigenvalue, if exists, is the one with the maximum absolute value.

Optional parameter: *IterMax* sets the maximum number of iterations allowed (default 1000).

This function uses the power's iterative method

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

$$|\lambda_1| > |\lambda_2| > \dots > |\lambda_n|$$

Starting with a generic vector x_0 , we have:

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

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

A global localization method for real eigenvalues

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

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

The matrix is symmetric so all its eigenvalues are real.

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

We can use the **M_TRAC()** function

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

We find the dominant eigenvalues λ_1
by the function MatEigenvalue_max()
The radius can be found by the formula

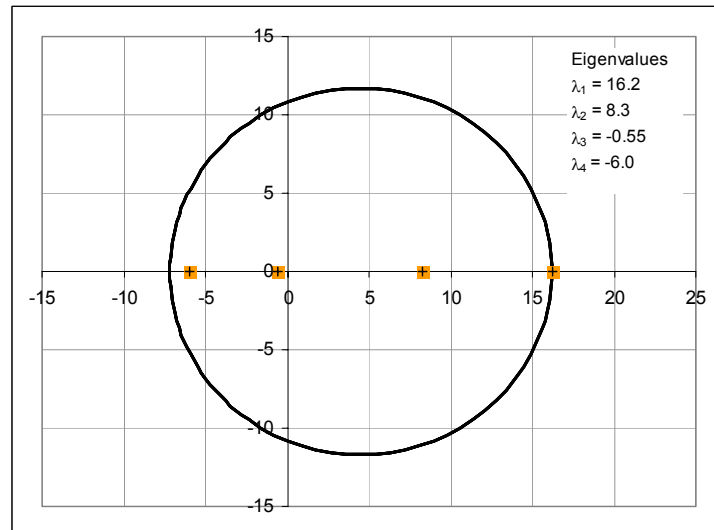
$$r = |\lambda_1| - c$$

	A	B	C	D	E	F	G	H	I
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									

Formulas shown in the image:

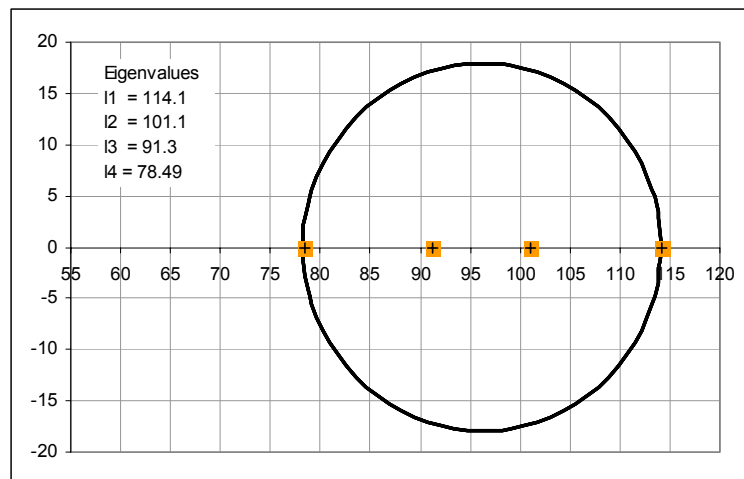
- =F2-G2** (for R)
- =M_TRAC(A2:D5)/4** (for C)
- =MatEigenvalue_max(A2:D5)** (for λ max)

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



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

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



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

Returns the dominant eigenvector of matrix **Mat**

Dominant eigenvector is related to the dominant eigenvalue.

Dominant eigenvalue, if exists, is the one with the maximum absolute value.

Optional parameters are:

IterMax sets the maximum number of iterations allowed (default 1000).

Norm: if TRUE, the function returns a normalized vector $|v|=1$ (default FALSE)

Remark: This function uses the power's iterative method

For further details see function MatEigenvalue_Max

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

Function MatEigenvalue_pow(Mat, [IterMax])

This function returns all eigenvalues of a given matrix **Mat**.

Optional parameters are:

IterMax sets the maximum number of iterations allowed (default 1000).

Norm: if TRUE, the function returns a normalized vector $|v|=1$ (default FALSE)

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

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

Example: find all eigenvalues and eigenvectors of the given matrix

	A	B	C	D	E	F	G	H	I
1									
2	-8	12	23	15		1	-1	-1	1
3	-3	7	7	6		0	-1	1	0.5
4	-8	8	19	9		1	8E-15	-1	0.5
5	6	-6	-12	-4		-0.667	-2E-15	1E-14	-0.5
6	{=MatEigenvector_pow(A2:D5)}								
7						5	4	3	2
8	{=MatEigenvalue_pow(A2:D5)}								
9									

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

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

This function returns all eigenvectors of a given matrix **Mat**.

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)

Remark: This function uses the power's iterative method. This algorithm works also for non-symmetric matrices with low-moderate dimension; it is less efficient then QR factorization. For symmetric matrices use the QR method.

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

See also function MatEigenvalue_pow

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

{=MatEigenvector_pow(A1:C3;TRUE)}

Function MatEigenvector_inv(Mat, Eigenvalue)

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

$$Av = \lambda v$$

If Eigenvalues is a single value, the function returns a (n x 1) vector. Otherwise if Eigenvalues is a vector of all eigenvalues of matrix A, the function returns a matrix (n x n) of eigenvector.

The eigenvector returned is normalized with norm=2 .

This method is adapt for eigenvalues affected by large error, because it is more stable than the singular system resolution.

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

Matrix A

94	-7	6	-19	9
-32	69	34	-64	26
-22	-22	124	-45	17
10	10	-10	123	-9
4	4	-4	8	100

Eigenvalues

100
101
102
103
104

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		matrix							eigenvalues		u1	u2	u3	u4	u5
2		94	-7	6	-19	9		λ_1	100		-0.07	-0.707	0.4082	0.7559	0
3		-32	69	34	-64	26		λ_2	101		-0.84	0.7071	-0.816	0.378	0.7698
4		-22	-22	124	-45	17		λ_3	102		-0.49	0	-0.408	0.378	0.5774
5		10	10	-10	123	-9		λ_4	103		0.21	0	0	-0.378	-0.192
6		4	4	-4	8	100		λ_5	104		0.07	0	0	0	-0.192
7															
8															
9															
10															
11															
12															
13															
14															
15															

{=MatEigenvector_inv(C2:G6;A2:A6)}

{=MatNormalize(C9:G13;2)}

About perturbed eigenvalues

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

Instability Sensitivity

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

Where:

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

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

		Singular linear system method		Iterative inverse method	
λ		λ	Δ λ	λ	Δ λ
100		100.0001	0.0001	100.3	0.3
u		u	Δ u	u	Δ u
1 12 7 -3 -1	1	0.99981665	1.83E-04	1.00000000	0
	12	12.00028336	2.36E-05	12.00000085	7.12E-08
	7	7.00005834	8.33E-06	7.00000046	6.58E-08
	-3	-3.00003333	1.11E-05	-3.00000020	6.58E-08
	-1	-1.00000000	0	-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 more heavy (0.3%). On the other hand the first method computes a sufficiently accurate eigenvector only if the eigenvalue error is very little (0.0001%). Note that for higher errors the first method fails, returning the null vector.

On the contrary the iterative inverse algorithm tollerates large amount of error in eigenvalue. This can be viewed by the instability factor

Singular linear system method

λ	$\Delta \lambda$	$ u $	$\Sigma \Delta u $
100.0001	0.0001	14.28	0.000226

S1 = 15.85

Iterative inverse method

λ	$\Delta \lambda$	$ u $	$\Sigma \Delta u $
100.3	0.3	14.28	2.69E-07

S2 = 6.3E-06

As we can see the difference is quite evident. In that case of hill-conditioned matrix (eigenvalues very close each others) the instabity factor exhibited by the iterative inverse algorithm is less then 1E-6 times the other one. Clearly this is a good reason for use it.

Matrices Generator

This is a set of useful tools to generate several types of matrices

Function [MatRnd](#)(Optional Typ, Optional MatInteger, Optional Amax, Optional Amin, Optional Sparse

Function [MatRndEig](#)(Eigenvalues, Optional MatInteger)

Function [MatRndEigSym](#)(Eigenvalues)

Function [MatRndRank](#)(Optional Rank, Optional det, Optional MatInteger)

Function [MatRndSim](#)(Optional Rank, Optional det, Optional MatInteger)

Function [MatRndUni](#)(Optional MatInteger)

Function [Mat_Hilbert](#)()

Function [Mat_Householder](#)(x)

Function [Mat_Tartaglia](#)()

Function [Mat_Vandermonde](#)(x)

These tools are also available in Macro version.

Function [MatRnd](#)(Optional Typ, Optional MatInteger, Optional Amax, Optional Amin, Optional Sparse

Generate a random matrix in the range that you have selected

Parameters are:

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

MatInteger = True (default) as Integer, False as Decimal

Amax = max number allowed

Amin = min number allowed

Sparse = coefficient from 0 to 1 - 0 mean no sparse, 1 mean 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	=MatRnd("ALL")				=MatRnd("SYM")				=MatRnd("DIA")			
7	triangular lower				triangular upper				tridiagonal			
8	4	0	0	0	7	7	-9	-2	-4	2	0	0
9	10	-6	0	0	0	10	-5	-10	10	2	-8	0
10	-4	4	9	0	0	0	-8	7	0	4	-5	9
11	-1	2	6	9	0	0	0	5	0	0	3	9
12	=MatRnd("TLW")				=MatRnd("TUP")				=MatRnd("TRD")			

	A	B	C	D	E	F	G
1							
2		-1	-1	0	1	0	
3		-1	-1	0	-1	-1	
4		0	0	-1	0	-1	
5		1	-1	0	-1	-1	
6		0	-1	-1	-1	-1	
7		{=MatRnd("SYM",1,-1)}					
8							
9		2	2	0	0	0	
10		2	1	1	0	0	
11		0	2	2	1	0	
12		0	0	2	2	2	
13		0	0	0	2	1	
14		{=MatRnd("TRD",2,1)}					
15							

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

Function [MatRndEig](#)(Eigenvalues, Optional MatInteger)

Returns a matrix with a given set of eigenvalues

MatInteger = True (default) as Integer, False as Decimal

Function MatRndEigSym(Eigenvalues)

Returns a symmetric 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			{=MatRndEig(A2:A5)}					{=MatRndEigSym(A2:A5)}			
7											

Function MatRndRank(Optional Rank, Optional det, Optional MatInteger)

Returns a matrix with a given Rank or Determinant

MatInteger = True (default) as Integer, False as Decimal

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

Function MatRndSim(Optional Rank, Optional det, Optional MatInteger)

Returns a symmetric matrix with a given Rank or Determinant

MatInteger = True (default) as Integer, False as Decimal

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

Function MatRndUni(Optional MatInteger)

Returns an unitary matrix (Det=1)

MatInteger = True (default) as Integer, False as Decimal

Function Mat_Hilbert()

This kind of matrix is a strongly hill-conditioned and is useful to test some algorithms

In the example below we see a (4 x 4) Hilbert matrix

	B2	{=Mat_Hilbert()}					
	A	B	C	D	E	F	G
1							
2		1	1/2	1/3	1/4	1/5	
3		1/2	1/3	1/4	1/5	1/6	
4		1/3	1/4	1/5	1/6	1/7	
5		1/4	1/5	1/6	1/7	1/8	
6		1/5	1/6	1/7	1/8	1/9	
7							

Function Mat_Householder(x)

Returns the Householder matrix 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

An unusual application of Householder matrix is shown at the following topic: "How to generate a random symmetric matrix with given eigenvalues"

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

This function, also available in macro version, is realized for didactic scope. It can trace, step by step, the diagonal reduction or triangular reduction of a matrix by the Gauss-Jordan algorithm.

Optional parameter *Typ* can be "D" (default) for Diagonal or "T" for Triangular

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

The argument *Mat* is the complete matrix (n x m) of the linear system

Remember that for a linear system:

$$Ax = b$$

A is the system square matrix (n x n)

x is the unknown vector (n x 1)

b is the vector of constant terms (n x 1)

$$C = [A, b]$$

C is the complete matrix of the system

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

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

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

	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	E
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	{=Gauss_Jordan_step(A1:D3)}				
9					

As we can see, the algorithm has exchanged rows 1 and 3, because the pivot a_{11} was 0

Now, with the range A5:D7 still selected, copy the active selection by CTRL+C

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

	A	B	C	D	E
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	{=Gauss_Jordan_step(A1:D3)}				
9	4	0	5	91	
10	0	1	-1.5	-28.5	
11	0	-10	3	105	
12	{=Gauss_Jordan_step(A5:D7)}				
13					

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

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

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

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

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

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

This function solves a linear system by the Gauss-Jordan algorithm.

The argument **A** is the complete matrix (n x m) of the linear system

The argument **b** is the constant vector (n x 1) of the linear system

Optional parameters:**IMODE** switch (default False) sets the floating point (False) or integer computing (True). Integer computing is intrinsically more accurate but also more limited because of overflow error. Use only with integer matrices of moderate size.

Optional parameter **Tiny** (default is 0) sets the minimum round-off error; any value in absolute less than Tiny will be set to 0.

If the matrix is singular the function returns "singular"

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

If non-singular, returns a vector or a matrix solution of a given system.

Remember that for a linear system:

$$Ax = b$$

A is the system square matrix (n x n)

x is the unknown vector (n x 1)

b is the vector of constant terms (n x 1)

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

Otherwise the solutions can be infinite or even nothing. In that case the system is called "singular". See Function SYSLINSING(Mat, Optional V)

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

	A	B	C	D	E	F	G	H	I	J
1	Linear System $Ax = b$									
2		A					b		x	
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										
10										

Formula bar: `{=SYSLIN(B3:E6,G3:G6)}`

Parameter **b** can also be a matrix of m columns. In that case SYSLIN solves simultaneously a set of m system. For further details see: Several ways for using the Gauss-Jordan algorithm

Function SYSLIN3(Mat3, v)

This function solves a tridiagonal linear system.

Returns the vector solution, if Mat3 is not singular.

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:

$$Ax = v$$

A is the system square matrix (n x n)

x is the unknown vector (n x 1)

v is the constant terms vector (n x 1)

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

Otherwise the solutions can be infinite or even nothing. In that case the system is called "singular".

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

Example - let's see how to solve a 16 x 16 tridiagonal linear system $Ax = y$

We pass to the function only 46 values (the first cell of **a** and the last of **c** are always 0) instead of 256 values.

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

G3 `= {=SYSLIN3(A3:C18,E3:E18)}`

	A	B	C	D	E	F	G	H	I
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									

Formula bar: `{=SYSLIN3(A3:C18,E3:E18)}`

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

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

$$[A] \cdot x = b$$

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.

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

As we can see, the values approximate the exact solution [4, -3, 5]. Precision increase with steps (of course, for convergent matrix)

	A	B	C	D	E	F	G	H
1	Linear system resolution with iterative methods							
2						Step =>	0	20
3	6	-1	2	37		X1 =>	0	3,999984082
4	2	-7	6	59		X2 =>	0	-2,999979881
5	-1	3	5	12		X3 =>	0	4,999984745
6								
7								
8								

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

x1 = SYSLIN_ITER_G(A, b, x0)

x2 = SYSLIN_ITER_G(A, b, x1)

x3 = SYSLIN_ITER_G(A, b, x2)

.....

x20 = SYSLIN_ITER_G(A, b, x19)

In the example below we see the trace of iteration values

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

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

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

This function solves a triangular linear system by the forward and backward substitutions algorithms. Returns a vector or a matrix solution of a given system, if not singular. The argument **Mat** is the complete matrix (n x n) of the linear system. Remember that for a linear system:

$$Ax = b$$

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

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 nothing. In that case the system is called "singular".

The parameter **b** can be also a matrix **B** (n x m). In that case the function returns a matrix solution **X** of the multiple linear system.

Parameter **typ** = "U" or "L" switches the function to solve for upper-triangular (back substitutions) or lower-triangular system (forward substitutions); if omitted the function automatically detects the type of the system.

Optional parameter **Tiny** (default is 0) sets the minimum round-off error; any value in absolute less than Tiny will be set to 0.

Example of (7 x 7) system

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

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

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

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

$$[A] \cdot x = b$$

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

Parameter x_0 is the starting approximate solution vector (range $n \times 1$)

Parameter **Nmax** is the max step allowed (default = 1)

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

This function is similar to the SYSLIN_ITER_G function.

For further details see Function SYSLIN_ITER_G(Mat, U, X0, Optional Nmax)

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

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

$$Ax = b$$

$$Ax = 0$$

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

$$y = Cx + d$$

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

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

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

This version solves also systems where n° of equations is less than n° of variables; that is: **A** is a rectangular matrix ($n \times m$) 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									

C d

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

See also: Function TRASFLIN(Mat, x, Optional B)

Example 1: Find the solution of the following homogeneous system

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

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

$$y = Cx$$

Where matrix C can be obtained by the function **SYSLINSING()**

	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		=SYSLINSING(B2:D4)				
12						

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

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

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

Example 2: Find the solution of the following homogeneous system

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

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

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

	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						

It is easy to prove that this linear function is a plane in R^3 . 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

$$Ax = b$$

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

As we can see, the rank of the given system is 2; so there is one independent variable x_3 .

The solutions can be writing as:

$$y = Cx + d$$

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

	A	B	C	D	E	F
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						
13						

{=SYSLINSING(B3:D5;E3:E5)}

Function TRASFLIN(A, x, Optional B)

This function performs the Linear Transformation

$$y = Ax + b$$

Where:

A is the matrix (n x m) of transformation

b is the known vector (n x 1); default is the null vector

x is the vector of independent variables (m x 1)

y is the vector of dependent variables (n x 1)

	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									

{=TRASFLIN(A2:C4;E2:E4;G2:G4)}

This function accepts also all matrices; in that case the matrix transformation is

$$Y = AX + B$$

Where:

A is the matrix (n x m) of transformation

B is the known matrix (n x p); default is the null vector

X is the matrix of independent variables (m x p)

Y is the matrix of dependent variables (n x p)

Matrix Geometric action

Linear transformation have a useful geometric interpretation¹⁰.

¹⁰ A wonderful, smart, cool, geometric description has shown 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 who think that Linear Algebra cannot be amusing. Don't miss them.

Take a point \mathbf{x} (x_1, x_2) of the plane and then compute the linear transform $\mathbf{y} = [\mathbf{A}] \mathbf{x}$, where \mathbf{A} (2×2) matrix; the point \mathbf{y} (y_1, y_2). We wonder if there is a geometrical relation between the point \mathbf{x} and \mathbf{y} . The relation exist and became evident if we perform the transformation of the points belong to the unitary circle.

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

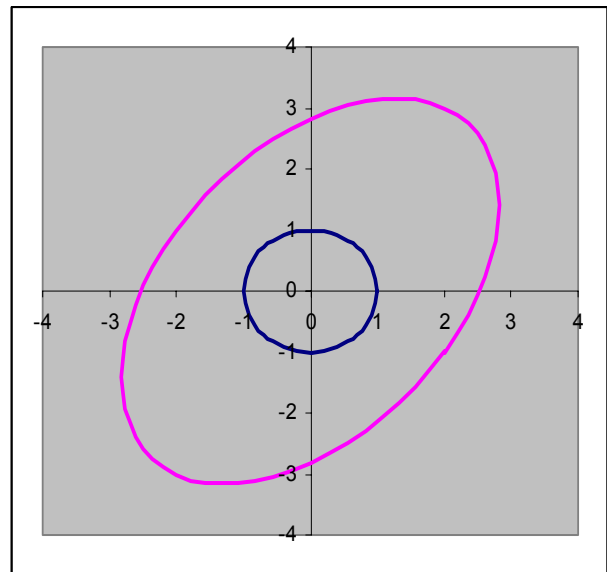
$$\mathbf{x} = (\cos(k \cdot \Delta\alpha), \sin(k \cdot \Delta\alpha)) \quad \text{for } k = 1, 2 \dots N \quad \text{where } \Delta\alpha = 2\pi/N$$

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

$$\mathbf{y}^T = \mathbf{x}^T \mathbf{A}^T + \mathbf{b}^T$$

Here a possible arrangement.

	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	1.16	0.38	0.92	2.61	2.38	
15	8	1.37	0.2	0.98	2.35	2.75	
16	9	1.57	0	1	2	3	



The blu line correspond to the unitary circle points; the other line belong to the transformed \mathbf{y} points. Thus, the circle has been transformed into a centered ellipse; if we add also $\mathbf{b} \neq 0$, we get a translated ellipse. Each point of unitary circle has been projected on the ellipse.

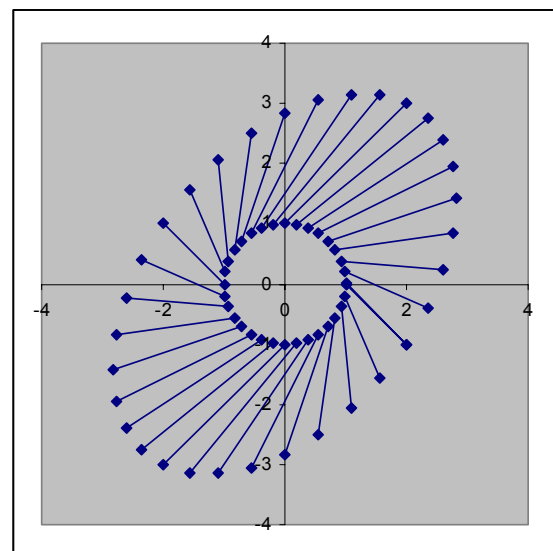
We have to point out that the projection is not radial but it happens in a very strange way. Look at the imagine to the right. It shows how a point on the circle move on the ellipse, befor and after the linear transformation.

It seems as if the circle would be dilatated (stretched) and then rotated, like a "whirlpool".

Other dimensions

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

Space	Original pattern	Transf. pattern
\mathbb{R}^2	circle	\Rightarrow ellipse
\mathbb{R}^3	sphere	\Rightarrow ellipsoid
\mathbb{R}^n	hypersphere	\Rightarrow hyperellipse



Function Gram_Schmidt(A)

This function performs the orthonormalization of a base vectors by the Gram-Schmidt's method. Argument **A** is a matrix (n x n) containing n independent vectors.

This function returns the orthogonal matrix **U**; each vector has $|v_i| = 1$ and $v_i \perp v_j$

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

This function is very sensible to the round-off errors.

For larger matrices see Function MatOrtNorm(Mat)

	A	B	C	D	E	F	G	H	I	J	K
1		A			Orthonormalization				check		
2	1	2	4		0.408	0.436	0.802		1	-0	-0
3	2	3	-1		0.816	0.218	-0.53		-0	1	-0
4	-1	0	1		-0.41	0.873	-0.27		-0	-0	1
5											
6											
7											
8											

(=Gram_Schmidt(A2:C4)) (=MMULT(E2:G4,M_TRANSP(E2:G4)))

Gram-Schmidt's Orthonormalization

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

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

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

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

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

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

Developing this algorithm, we see that the vector k is build from all k-1 previous vectors

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

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

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

At the end, all vectors of the bases **U** will be orthogonal and normalized.

This process is performed by the function Gram_Schmidt(Mat)

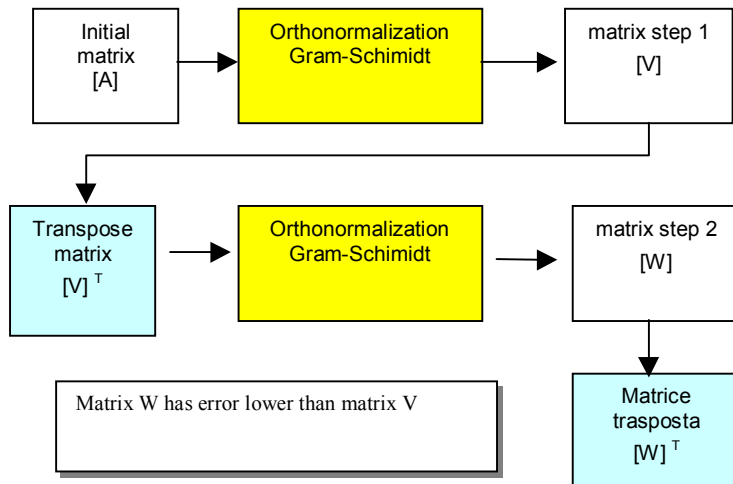
This method is very straightforward, but it's also very sensible to the round-off errors. This happens because the error propagates itself along the vectors from 1 to n

Double step Gram-Schmidt method

One method to reduce the error propagation is the following

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

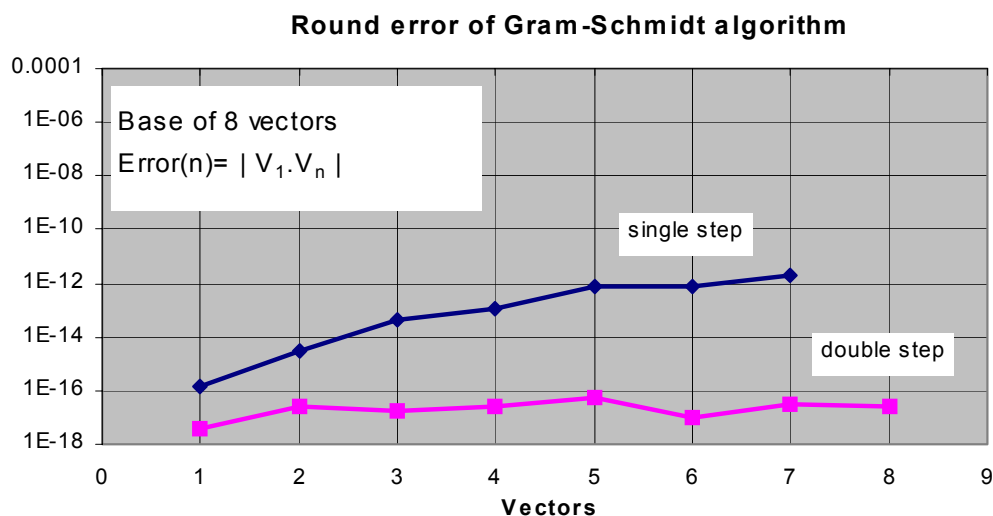
The following flow-chart illustrated better how this method works



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

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

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



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

Function Mat_Cholesky(A)

This function returns the Cholesky decomposition of a symmetric matrix

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

Where A is a symmetric matrix, L is a lower triangular matrix

This decomposition works only if A is *positive definite*. That is:

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

Or, that the eigenvalues of A is all-positive. This function returns always a matrix. Inspecting the diagonal element of this result and if they are all positive, then also the matrix A is positive definite.

Example - Say 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 all its eigenvalues are positive.

On the contrary, the decomposition of the matrix **B** shows a negative number at the position 33; then we can say that **B** is not positive definite and some of its eigenvalues are negative.

	A	B	C	D	E	F	G
1	Cholesky Decomposition						
2							
3							
4	A				B		
5	3	1	2		5	2	1
6	1	6	4		2	2	3
7	2	4	7		1	3	1
8	L				L		
9	1.7321	0	0		2.2361	0	0
10	0.5774	2.3805	0		0.8944	1.0954	0
11	1.1547	1.4003	1.9251		0.4472	2.3735	-4.8333
12							
13	{=Mat_Cholesky(A3:C5)}						
14							
15							
16							

this element is
negative =>
Matrix indefinite

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

Function Mat_LU(A, optional Pivot)

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

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

Where **L** is a lower triangular matrix, and **U** is upper triangular matrix

If the square matrix has (n x n) dimension, this function returns a matrix (n x 2n) where the first n columns are the matrix **L** and the last n columns are the matrix **U**.

Parameter *Pivot* (default=TRUE) activates the partial pivoting.

Note: that if partial pivot is active (TRUE) the LU decomposition can refer to a permutation of **A**.

TUTORIAL FOR MATRIX.XLA

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

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

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

The LU decomposition are often used to solve linear system

$$\mathbf{A} \mathbf{x} = \mathbf{b} \Rightarrow \mathbf{LU} \mathbf{x} = \mathbf{b} \Rightarrow \mathbf{L}(\mathbf{U} \mathbf{x}) = \mathbf{b}$$

The original system is now split into two simpler systems.

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

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

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.

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

For $i = 1, 2, \dots, N$

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

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

For $i = N, N-1, \dots, 2, 1$

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 Mat_LU function will be:

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

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

D2													
	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													

Function Mat_QR(Mat)

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

$$A = Q \cdot R$$

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

In this version¹¹ **Mat_QR()** can factors also a rectangular matrix A (n x m). It returns a matrix (m x (n + n)), where the first (m x n) block is **Q** and the first n rows of the second (m x n) block is **R**. The last m – n rows of the second block are all zero.

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

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

$$\begin{bmatrix} -2 & -4 & -6 \\ 1 & 3 & 2 \\ 2 & 2 & 5 \end{bmatrix} = \begin{bmatrix} -2/3 & 1/3 & 2/3 \\ 1/3 & -2/3 & 2/3 \\ 2/3 & 2/3 & 1/3 \end{bmatrix} \cdot \begin{bmatrix} 3 & 5 & 8 \\ 0 & -2 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

	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
8						

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

¹¹ Thanks to Ola Mårtensson

	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	{=Mat_QR(A1:C5)}									
8										
9										

Function Mat_QR_iter(Mat, [MaxLoops])

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

$$A = QR \Rightarrow A_1 = RQ$$

$$A_1 = Q_1 R_1 \Rightarrow A_2 = R_1 Q_1$$

$$A_2 = Q_2 R_2 \Rightarrow A_3 = R_2 Q_2$$

$$A_n = Q_n R_n \Rightarrow A_{n+1} = R_n Q_n$$

If the matrix **A** has:

$$|\lambda_2| > \dots > |\lambda_n|$$

Then the sequence converges to the diagonal matrix of eigenvalues

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

If the matrix is not symmetric the process gives a triangular matrix where the diagonal elements are still the eigenvalues.

Optional parameter *MaxLoops* (default 100) sets the max iteration allowed.

Example.

	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	{=Mat_QR_iter(A2:C4)}				10.15901	1.504833	-0.197191
7					0	2.349349	0.834621
8	{=MatMopUp(E2:G4)}				0	0	-1.508356
9							

Function MatExtract(A, i_pivot, j_pivot)

Returns the sub-matrix extract from A by eliminating one row and one column

i_pivot = row to eliminate

j_pivot = column to eliminate

	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			{=MatExtract(A1:D4;G1;G2)}		
8		-2	-1	2					
9		4	-1	3					

Function MatOrtNorm(A)

This function performs the orthogonalization with the double-step Gram-Schmidt algorithm

Argument **A** is a matrix (n x n) containing n independent vectors.

This function returns the orthogonal matrix U; each vector has norm = 1

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

See also Function Gram_Schmidt(Mat)

Example - Orthonormalize a given matrix with both Gram-Schmidt methods (single e double step)

	A	B	C	D	E	F	G	H	I	J
1	Gram-Schmidt's Orthonormalization									
2										
3	1	0	0	3						
4	5	26	12	5						
5	2	2	2	7						
6	1	5	7	27						
7										
8	Gram-Schmidt						Double Gram-Schmidt			
9	0.1796	-0.496	-0.108	0.8427			0.1796	-0.496	-0.108	0.8427
10	0.898	0.396	-0.191	0.0172			0.898	0.396	-0.191	0.0172
11	0.3592	-0.771	0.0437	-0.525			0.3592	-0.771	0.0437	-0.525
12	0.1796	0.0571	0.9747	0.1204			0.1796	0.0571	0.9747	0.1204
13										
14		-3E-16	-1E-16	-9E-16				-1E-17	-3E-17	3E-18
15										
16	{=ProdScal(\$A9:\$A12;B9:B12)}						Scalar product: V ₁ *V ₂ , V ₁ *V ₃ , V ₁ *V ₄			
17										

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

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

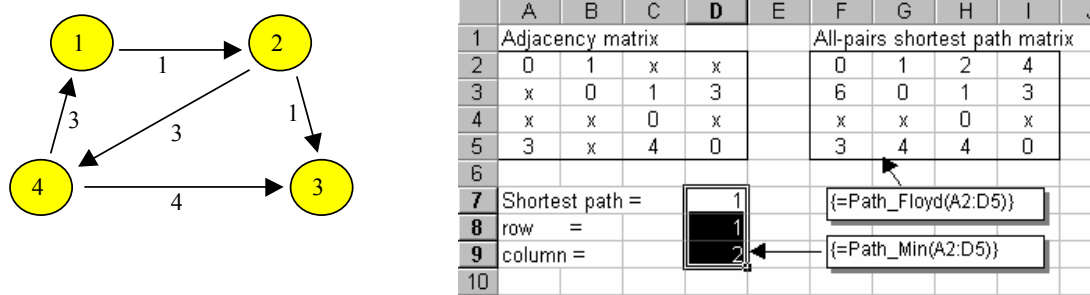
Function Path_Floyd(G)

This function, now available also in macro version, returns the matrix of all pairs shortest-path of a graph. This is an important problem in Graph-Theory and has applications in several different fields: transportation, electronics, space syntax analysis, etc.

The all-pairs shortest-path problem involves finding the shortest path between all pairs of vertices in a graph that can be represented as an adjacency matrix [G] in which each element a_{ij} - called node - represents the "distance" between element i and j . If there is not a link between two nodes, we leave the cell blank or we can set any not numeric symbol you like: for example "x"

This function uses the *Floyd's sequential algorithm*

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



See also the function Path_Min()

Function Path_Min(G)

Returns a vector contains the shortest path of a graph; the row and column of the cell

This function uses the Path_Floyd() function to find the all-pairs shortest path of the given graph G

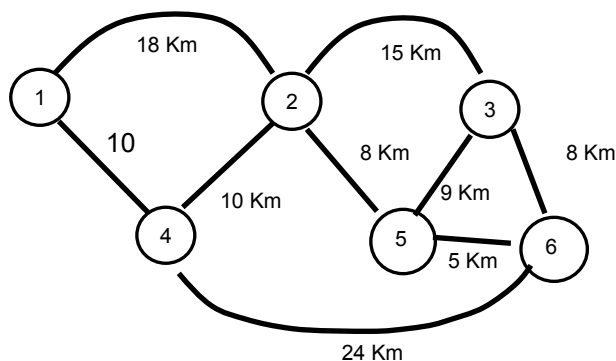
Path_Min(G) => [path_min, i_min ; j_min]

This function return an array; give the CTRL+SHIFT+ENTER keys sequence

If you only want the first value press simply the ENTER key

Graphs theory recalls

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



The map can be reassumed into the following matrix

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

In the cell 1,2 we fill the distance between city 1 and city 2 , that is 18 Km;
 In the cell 1,3 we fill "x" because there is not a direct way between city 1 and city 3.
 In 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 matrix.

The above matrix - "adjacent matrix" - reports only the direct distance between each couple of city.
 But we can join, for example, city 1 and city 3 in several different paths:
 city 1 - city 2- city 3 = 18 + 15 = 33 Km
 city 1 - city 4 - city 6 - city 3 = 10 + 24 + 8 = 42 Km etc.

The first one is the shortest distance path for city 1 and city 3

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

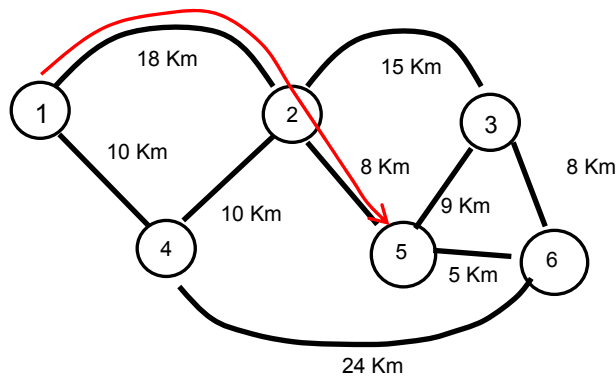
	city 1	city 2	city 3	city 4	city 5	city 6
city 1	0	18	33	10	26	31
city 2	18	0	15	10	8	13
city 3	33	15	0	25	9	8
city 4	10	10	25	0	18	23
city 5	26	8	9	18	0	5
city 6	31	13	8	23	5	0

This matrix reports the shortest distance between each couple of city

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

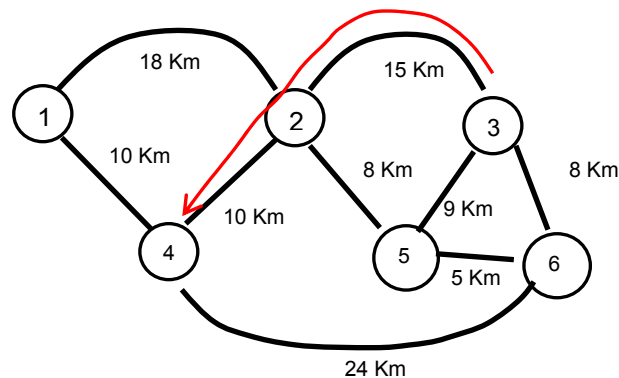
	city 1	city 2	city 3	city 4	city 5	city 6
city 1	0	18	33	10	26	31
city 2	18	0	15	10	8	13
city 3	33	15	0	25	9	8
city 4	10	10	25	0	18	23
city 5	26	8	9	18	0	5
city 6	31	13	8	23	5	0

TUTORIAL FOR MATRIX.XLA



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

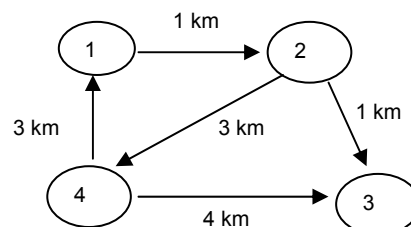
	city 1	city 2	city 3	city 4	city 5	city 6
city 1	0	18	33	10	26	31
city 2	18	0	15	10	8	13
city 3	33	15	0	25	9	8
city 4	10	10	25	0	18	23
city 5	26	8	9	18	0	5
city 6	31	13	8	23	5	0



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

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

Let see this example



The adjacent matrix is built in the same way; the only different is that in this case is not symmetric.

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

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

Applying the Floyd algorithm we get the following matrix

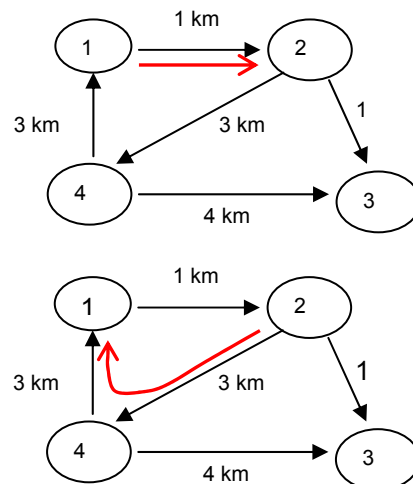
	node 1	node 2	node 3	node 4
node 1	0	1	2	4
node 2	6	0	1	3
node 3	x	x	0	x
node 4	3	4	4	0

Reading this matrix is simple:

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

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

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



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

Let's see how use this array function in Excel

Shortest path

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

TUTORIAL FOR MATRIX.XLA

	A	B	C	D	E	F	G	H	I	J	K
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											
11											
12											

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

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
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														
11														
12														
13														
14														
15														
16														

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

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

All the solution's values fill all the cells that you have selected.

Note that Excel shows the function around two braces { }

These symbols mean that the function return an array.

The matrix returned is the shortest-path matrix

	A	B	C	D	E	F	G	H
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	
16								

Function SVD - Singular Value Decomposition**Function SVD_U(A)****Function SVD_D(A)****Function SVD_V(A)**

Singular Value Decomposition of a matrix **A** (n x m) provides 3 matrices, **U**, **D**, **V** performing the following decomposition¹²:

Where: $p = \min(n, m)$

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

U is an orthogonal matrix (n x p)

D is a square diagonal matrix (p x p)

V is an orthogonal matrix (m x p)

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

For example

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

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

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

Example. Find the SVD decomposition for the given matrix

	A	B	C	D	E	F	G	H	I	J	K	L
1		A			U			D			V	
2	1	8	2	-0.878	-0.379	-0.293	9.361	0	0	-0.234	0.772	-0.59
3	3	4	-1	-0.47	0.801	0.372	0	3.219	0	-0.961	-0.092	0.261
4	-1	1	1	-0.093	-0.464	0.881	0	0	0.1	-0.147	-0.628	-0.764
5				{=SVD_U(A2:C4)}			{=SVD_D(A2:C4)}			{=SVD_V(A2:C4)}		
6												

From the **D** matrix of singular values we get the max and min values to compute the condition number m^{13} , used to measure the ill-conditioning of a matrix. In fact, we have:

$$m = 9.361 / 0.1 = 93.61$$

¹² Some authors give a different definition for SVD decomposition, but the main concept is the same.

¹³ Respect to the norm 2

TUTORIAL FOR MATRIX.XLA

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

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

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

H2		fx {=SVD_D(B2:F6)}										
	A	B	C	D	E	F	G	H	I	J	K	L
1												
2		1	10	7	9	-2		121	0	0	0	0
3		4	37	25	36	-7		0	1.68	0	0	0
4		-8	-64	-43	-62	12		0	0	0.55	0	0
5		-2	-20	-14	-18	4		0	0	0	0	0
6		-1	-10	-7	-9	2		0	0	0	0	0
7												

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

Denomination. The matrix returned by the singular value decomposition are usually called: U (hanger), D (stretcher), V (aligner). So the decomposition for a matrix A can be written¹⁴

$$(any\ matrix) = (hanger)(stretcher)(aligner)$$

¹⁴ For further details see the "Introduction to the Singular Value Decomposition" by Todd Will, UW-La Crosse, Wisconsin, 1999 and "Matrices Geometry & Mathematica" by Bill Davis and Jerry Uhl

Function MatMopUp(M, [ErrMin])

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

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

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

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

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

Function MatCovar(A)

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

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

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

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

See also the matrix correlation function MatCorr()

This function is similar to COVAR() built-in function for two variables.

Function MatCorr(A)

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

The correlation matrix is definite by the following formulas:

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

Where:

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

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

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

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

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

There are three variables x1, x2, x3 and 10 data observations. The matrix will be 3 x 3.
In the first columns A, B, C we have arranged the row data (orientation is not important).
In the last row we have calculate the statistics: average \bar{x}_i and standard deviation σ_{xi} for each column.

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

We have calculate each column u_i with the following simple formulas: $u_{ij} = \frac{x_{ij} - \bar{x}_i}{\sigma_{xi}}$

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

At the right we have calculated the covariance matrices for both row and normalized data: They are always symmetric.

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

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

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

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

TUTORIAL FOR MATRIX.XLA

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

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

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

In the correlation matrix we discover a relative high level of correlation for

$$a_{23} = a_{32} = 0.78 \text{ (the max is 1)}$$

This means that between variable 2 and 3 there is a tight relation.

On the contrary, we note a weak dependence between variable 1 and 3

Function REGRL(Y, X, [ZeroIntcpt])

Computes the multivariate linear regression with the SVD method in multi precision arithmetic. Parameter **Y** is a vector (n x 1) of dependent variable values. Parameter **X** is a list of independent variable values. It may be a vector for monovariate regression (n x 1) or a matrix for multivariate regression (n x m).

Parameter **ZeroIntcpt**, if present, forces the Y intercept to zero: $Y(0) = 0$

The function returns the coefficients vector of linear regression. For monovariate regression, it returns two coefficients [a0, a1]; the first one is the intercept of Y-axis, the second one is the slope.

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

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

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

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

var(y) var(y^) r2

17.663 17.145 0.9707

=VARP(C3:C16) =VARP(B3:B16) =F6/E6

=F\$3+\$G\$3*A16

The coefficient r2, between 0 and 1, measures how good is the fit (0 bad - 1 perfect)

In Matrix.xla there is not a specific function for that, but it can be easily computed by the standard statistic functions and using the formula

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

Function REGRP(degree, f, x, [ZeroIntcpt])

Computes the polynomial regression of a set of data points [xi, f(xi)].

Parameter **degree** set the degree of the polynomial.

Parameter **f** is a vector (n x 1) of dependent variable values. Parameter **X** is a vector (n x 1) of independent variable values.

Parameter **ZeroIntcpt**, if present, forces the intercept to zero: f(0)= 0

The function returns the coefficients vector of linear regression [a0, a1, a2, ...am].

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

Example - given a table of (x, y) point, find the 6-th degree polynomial that approximates better the data, using the REGRP() function

	A	B	C	D	E	F
1	x	y		polynomial coefficients		
2	0	134		a0	134	
3	0.2	153.355584		a1	97	
4	0.5	182.296875		a2	-1	
5	0.8	211.857024		a3	-1	
6	1	233		a4	2	
7	1.5	302.984375		a5	1	
8	2	444		a6	1	
9	2.5	774.546875				
10	3	1523				
11	3.5	3081.984375				
12	4	6074				
13						
14						

[=REGRP(6,B2:B12,A2:A12)]

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

It returns the polynomial interpolation of a set of data points [xi, f(xi)].

Parameter **x** is the point to interpolate. It can be also a vector of interpolation points.

Parameter **knots** is a matrix (n x 2) of data point [xi, f(xi)].

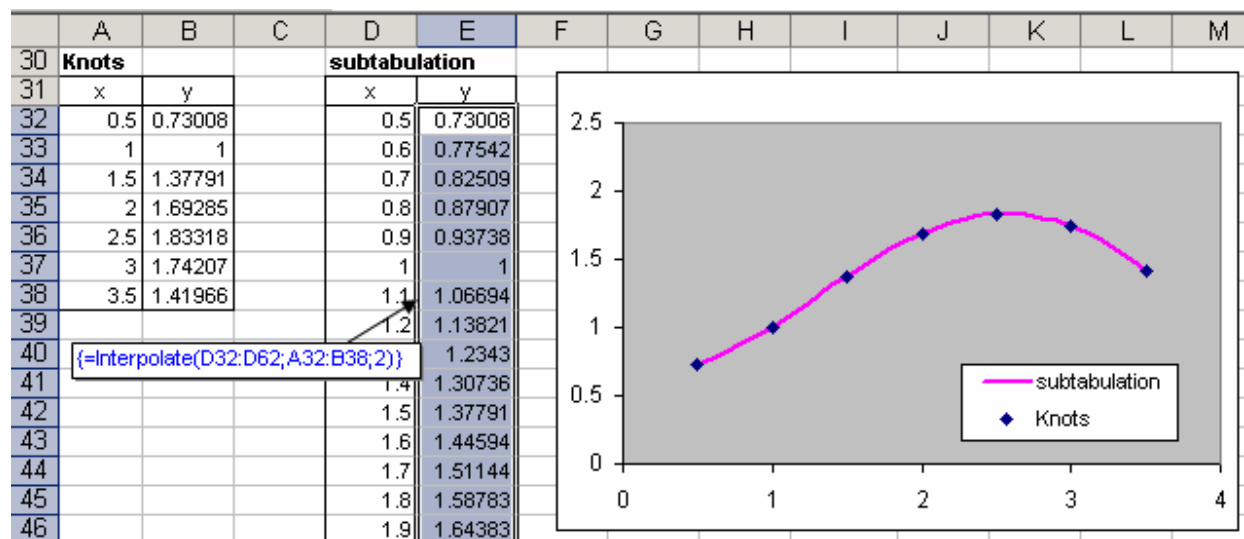
Parameter **degree**, set the degree of interpolation polynomial (default degree = 2)

Parameter **points**, set the number of points subset for the polynomial regression . if omittes the function assumes points = degree +1

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

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

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



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

Function MatCmpn(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 coefficients vector. If an $< > 1$, all coefficients are normalized before generating the companion matrix

Example:

	A	B	C	D	E	F	G	H	I	J
1	Polynomial =			50+24x+18x^2+7x^3+2x^4+x^5						
2	Coefficients			Companion Matrix						
3	a0	50		0	0	0	0	-50		
4	a1	24		1	0	0	0	-24		
5	a2	18		0	1	0	0	-18		
6	a3	7		0	0	1	0	-7		
7	a4	2		0	0	0	1	-2		
8	a5	1								
9										
10	{=MatCmpn(B3:B8)}									

Function Poly_Roots_QR(Coefficients)

This function returns all roots of a given polynomial

Coefficients parameter is an array of n+1 polynomial coefficients

This function use the *QR algorithm*. The process consists of finding the eigenvalues of a companion matrix with the given polynomial coefficients.

This process is very fast, robust and stable but may not be converging under certain conditions. If the function cannot find a root it returns "?". Usually it suitable to solve polynomial up to 10- 12 ° with a good accuracy (1E-9 – 1E-12)

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

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

D4		fx {=Poly_Roots_QR(B4:B12)}				
	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 MatRot(n, teta, p, q)

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

Parameter *teta* sets the angle of rotation in radiant

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

Example: In 3D space, all the 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 $\det = 1$

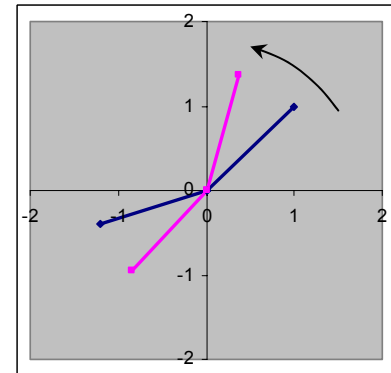
Example. Given two vectors in \mathbf{R}^2 ($\mathbf{v}_1, \mathbf{v}_2$), found the same vectors after a rotation of 30° deg

The transformation formula is :

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

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	{=MatRot(2,E6,1,2)}			30	0.524		{=M_PROD(D2:E3,A2:B3)}		
7									



Conditioned Number

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

Formally the conditioned number of a matrix is defined by the SVD decomposition as the ratio of the largest (in magnitude) element to the smallest element of diagonal matrix

Given the SVD decomposition:

$$A = UDV^T \quad \text{Where:} \quad D = \text{diag}[d_{11}, d_{22}, d_{33}, \dots, d_{nn}]$$

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

A matrix is ill-conditioned if its conditioned number is very large. For a 32bit double precision arithmetic this number is about $1E12$.

In this package there is not a specific function that returns this number, but it can be easily calculate by the D matrix returned by the function SVD_D()

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

Conditioned number = 15514

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

This function computes the orthogonal rotation for a Factors Loading matrix using the Kaiser's Varimax method for 2D and 3D factors

Parameter **FL** is the Factor Loading matrix to rotate (n x m). The number of factor m, at this release, must be only 2 or 3.

Optional parameter **Normal** = True/False chooses the "Varimax normalized criterion". That is, indicates if the matrix of loading is to be row normalized before rotation (default = False)

Optional parameter **MaxErr** set sets the accuracy required (default = 10^{-4}). The algorithm stops when the absolute difference of two consecutive Varimax values is less of MaxErr

Optional parameter **MaxIter** sets the maximum number of iterations allowed (default=500)

Algorithm

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

For Varimax criterion definition see Varimax Index

This function is diffused, by now, in the almost principal (and expensive) statistics tools, because, on the contrary, it is very rare in freeware software, we have added to our add-in.

Let's see how it works with one popular example

Example 2D- Initial Factors Matrix

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

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

Rotate Factors Matrix: method Varimax

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

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

Function VarimaxIndex(Mat, [Normal])

Returns the Varimax value of a given Factor matrix **Mat**

Varimax is a popular criterion Kaiser (1958) to perform orthogonal rotation of Factors Loading matrices. Usually, the rotation stops when Varimax is maximized

Optional parameter Normal = True/False indicates if the matrix is to be row normalized before computing

Formula

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

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

Where:

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

See also Varimax Rotation

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

Returns the normalized vectors of the matrix. *Mat* is an array (n x m)

The optional parameter *Normtype* indicate what normalization is performed

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

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

Example - Normalize the following 3x3 matrix

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

Function MatNorm(v, [NORM])

Return the Norm of a matrix or vector

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

The Norms returned, depending by the parameter *Norm*, are:

For vectors

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

For matrices

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

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

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

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

Function M_MULT_C(Mat1, Mat2, [Cformat])

Performs the complex matrix multiplication.

The argument Mat1 and Mat2 are array (n x n) or a (n x 2n), depending by the format parameter

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

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

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

$$\mathbf{M}_1 = \mathbf{A} + j \mathbf{B}$$

$$\mathbf{M}_2 = \mathbf{C} + j \mathbf{D}$$

Where **A**, **B**, **C**, **D** are real matrices: **A** and **C** are the real parts and **B** and **D**

Example:

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

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1			Matrix A							Matrix B				
2		Real			Imm					Real			Imm	
3	1	2	0	0	-1	3		1	2	0	0	0	1	
4	-1	3	-1	0	2	-1		0	-1	-1	-1	0	-1	
5	0	-1	4	0	-2	0		0	-1	1	0	4	0	
6														
7			Matrix C											
8		Real			Imm									
9	0	-12	-3	-2	-2	3								
10	1	0	-2	-3	-5	-7								
11	-2	-3	3	1	18	3								

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

{=M_MULT_C(A3:F5;H3:M5)}

Complex matrix multiplication C=A*B

The calculus can be also performed in the handy string rectangular format

We have only to set the Cformat parameter to 3

	A	B	C	D	E	F	G	H	I
1			A					B	
2		1	2-i	3i		1	2	i	
3		-1	3+2i	-1-i		-i	-1	-1-i	
4		0	-1-2i	4		0	-1+4i	1	
5									
6			C						
7		-2j	-12-2j	-3+3j					
8		1-3j	-5j	-2-7j					
9		-2+j	-3+18j	3+3j					
10									

{=M_MULT_C(B2:D4;F2:H4;3)}

Function M_INV_C(A, [Cformat])

Complex matrix inversion

The argument A is the array (n x n) or a (n x 2n), depending by the format parameter

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

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

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

	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													

{=M_INV_C(A3:F5)}

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

Function ProdScal_C(v1, v2,)

Returns the scalar product of two complex vectors

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

	A	B	C	D	E	F	G
1	vector a		vector b			a • b	
2	re	imm	re	imm		re	imm
3	2	1	-1	0		-6	9
4	3	2	0	1			
5	4	5	1	-2			
6							
7							
8							

{=ProdScal_C(A3:B5;C3:D5)}

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

In string format we can write complex numbers as string "a+ib".

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

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

=ProdScal_C(A2:A4;B2:B4;3)

Function SYSLIN_C(A, B, [Cformat])

This function solves a complex linear system by the Gauss-Jordan algorithm.

Returns a vector or a matrix solution of a given system, if not singular.

The argument **A** and **B** are array (n x n) or (n x 2n), depending by the format parameter

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

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

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

Remember that for a linear system:

$$Ax = b$$

A is the system complex square matrix (n x 2*n)

x is the unknown complex vector (n x 2)

b is the known complex vector (n x 2)

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

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			Imm			Real		Imm		Real		Imm	
4	1	2	0	0	-1	3	5	10	3	1				
5	-1	3	-1	0	2	-1	-11	-1	-1	1				
6	0	-1	4	0	-2	0	11	-3	2	-1				
7														
8														
9														

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

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

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

Function Simplex(Funct, Constrain, [Opt])

This function perform the linear optimization with the Simplex method

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

Constrain is the array (m x n+2) containing the coefficients of m linear constrain and the type of constrain (" $<$ ", " $>$ ", " $=$ ")

Opt set the optimization type: 1 (default) for maximization, 0 for minimization.

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

Maximize the function z

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

With the following constrains:

$$x_i \geq 0$$

and for j=1 to m

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

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

This function returns:

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

If the constrains region is unbounded – that is, if the region is not closed - a finite solution cannot exist and the function return "**inf**". Typically it happens when the constrains are insufficient

If the constrains region is bounded, but the solution doesn't exist, the function returns "?". Typically it happens when you add too many constrains

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

Now lets see how it works.

Example: find the maximum of the function:

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

With the following constrains:

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

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

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

TUTORIAL FOR MATRIX.XLA

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

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

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

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

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

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	constrain	x1	x2	x3	x4		
7		1	0	2	0	<	10
8		0	2	0	-7	<	0
9		0	1	-1	2	>	0.5
10		1	1	1	1	=	9
11							
12	solution	x1	x2	x3	x4	max	
13		0	3.325	4.725	0.95	17.025	
14							
15	{=Simplex(B4:E4;B7:G10)}						
16							

Function RRMS(v1, [v2])

This function computes the root mean squares of the regression residuals.

The argument v1 and v2 are vectors (n x 1) or (1 x n)

If the second vector is omitted the function returns simply the root mean squares of the vector

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

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

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

C6			=	=RRMS(A2:C4,D2:F4)					
	A	B	C	D	E	F	G	H	I
1		A			A^			A-A^	
2	7	-21	35	7	-21	35	-2E-12	1E-11	-2.7E-11
3	-21	91	-175	-21	91	-175	6E-12	-3E-11	6.9E-11
4	35	-175	371	35	-175	371	-7E-12	3E-11	-5.1E-11
5								{=(A2:C4-D2:F4)}	
6	Difference RMS :		3E-11	=RRMS(A2:C4,D2:F4)					
7									

Function MatPerm(Permutations)

Returns the permutations matrix. It consists of sequence of n unitary vectors (versors):

The parameter is a vector indicating the sequence.

For a space of 4x4 matrices the versors are

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

Permutations matrices are indicated by a sequence vector, like for example:

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

C2		= {=MatPerm(A2:A5)}										
	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						
6												

Function Mat_Hessemberg(Mat)

Returns the Hessemberg form of a square matrix

As known a matrix is in Hessemberg form if all values under the lower subdiagonal are zero.

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

Example

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

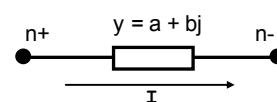
Function Mat_Adm(Branch)

Returns the Admittance Matrix of a Linear Passive Network Graph.

Branch is a list of 3 columns giving the basic information of each branch:

node+, node-, admittance .

The number of rows must be equal to the branches of the graph



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

Nodal Analysis gives the following equation to solve the linear passive network, where V is the vector of nodal voltage, I is the vector of nodal current and [Y] is the admittance matrix.

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

(usually the references nodes is set at V = 0)

V, I, [Y] are in general complexes





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

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

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

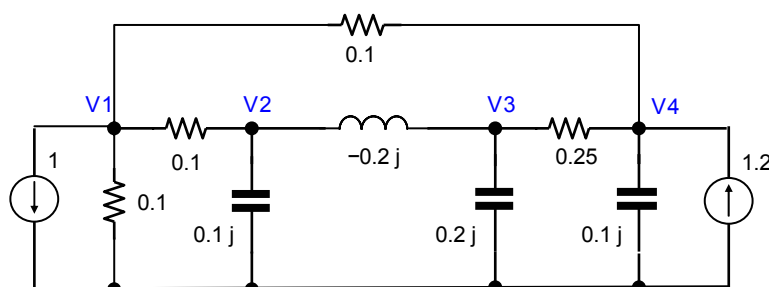
Linear Electric Network

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

Resistor		Value R (ohm)	Admittance $y = 1/R$
Capacitor		Value C (farad)	Admittance $y = j \omega C$
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			{=Mat_Adm(A3:D10)}								{=SYSLIN_C(F2:M5;G7:H10)}				

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

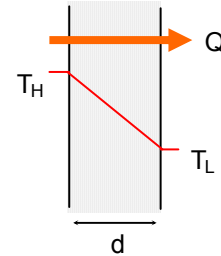
Thermal Network

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

One-Dimensional Conduction Heat Transfer.

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

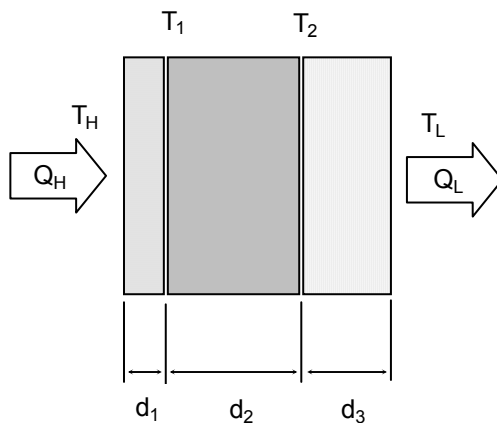
$$Q = -\frac{k}{d} \cdot (T_H - T_L) = -g(T_H - T_L)$$



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

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

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



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

Where:

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

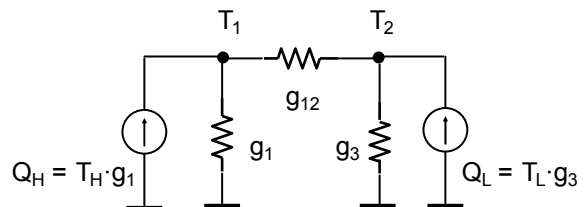
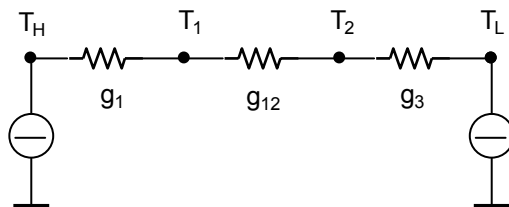
$$T_L = 20 \text{ } ^\circ \text{C}$$

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

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

Internal temperature T_1 and T_2 are unknowns

The thermal networks equivalent 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 conductances are:

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

TUTORIAL FOR MATRIX.XLA

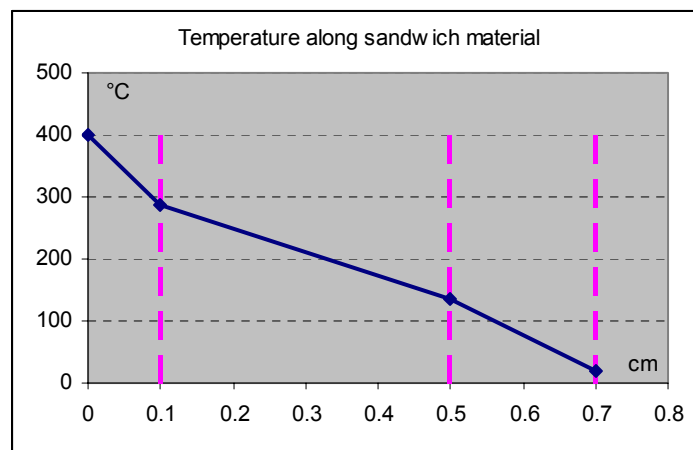
A spreadsheet calculus can be arranged as the following

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

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

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

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



Function Mat_Leontief(ExTab, Tot)

Returns the Leontief inverse matrix of the Input-Output Analysis Theory.

Parameter *ExTab* is the interindustry exchange table (or IO-table). This table lists the value of the goods produced by each economy sector and how much of that output is used by each sector.

Parameter *Tot* is the total production vector

Input Output Analysis

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

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

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

The *Leontief inverse matrix* is .

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

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

$$D = L \cdot X$$

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

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

Sectors	demand
Agriculture	45
Manufact.	74
Services	130

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

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

.

References

- [1] "LAPACK -- Linear Algebra PACKage" 3.0, Update: May 31, 2000
- [2] "Numerical Analysis" F. Sheid, McGraw-Hill Book Company, New-York, 1968
- [3] "Numerical Recipes in FORTRAN 77- The Art of Scientific Computing" - 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software
- [4] "Nonlinear regression", Gordon K. Smyth, John Wiley & Sons, 2002, Vol. 3, pp 1405-1411
- [5] "Linear Algebra" vol 2 of Handobook for Automatic Computation, Wilkinson, Martin, and Peterson, 1971
- [6] "Linear Algebra" Jim Hefferon, Saint Michael's College, Colchester (Vermont), July 2001.
- [7] "Linear Algebra - Answers to Exercises" Jim Hefferon, Saint Michael's College, Colchester (Vermont), July 2001
- [8] "Calcolo Numerico" Giovanni Gheri, Università degli Studi di Pisa, 2002
- [9] "Introduction to the Singular Value Decomposition" by Todd Will, UW-La Crosse, University of Wisconsin, 1999
- [10] "Calcul matriciel et équation linéaires", Jean Debord, Limoges Cedex, 2003
- [11] "Leontief Input-Output Modelling" Addison-Wesley, Pearson Education
- [12] "Computational Linear Algebra with Models", Gareth Williams, (Boston: Allyn and Bacon, 1978), pp. 123-127.
- [13] "Scalar, Vectors & Matrices", J. Walt Oler, Texas Teach Univerity, 1980

WHITE PAGE

Analytical Index

A

ABS; 29
adjacency; 192
adjacent matrix; 193; 194; 195
Admittance; 215
AVERAGE; 29

B

block; 159; 160

C

characteristic polynomial; 68; 166
Cholesky; 104; 187
coefficients; 203
cofactor; 40
companion; 86; 203
complete matrix; 49
complex; 34; 36; 146; 209; 210; 211
complex eigenvalue; 75; 162
complex eigenvector; 162
Conduction; 217
Consumption; 219
correlation; 199
covariance; 199
Crout; 187
Cubic interpolation; 127

D

deflation; 91
Demand; 219
determinant; 37; 137; 146
diagonal; 152
dominant; 88; 90; 167; 169

E

eigenvalue; 33; 68; 155; 167; 169
eigenvalues; 137
eigenvector; 68; 155; 162; 169; 170
eigenversor; 74
error; 153
Euclidean Norm; 145
Exchange table; 219
Exponential; 114

F

Factors Loading; 206
Floyd; 192
format; 36; 146; 209; 210; 211
Full Pivoting; 23
full-pivot; 30

G

Gauss_Jorda_step; 30
Gauss-Jordan; 20; 28; 176; 211
Gauss-Seidel; 179
generalized eigenproblem; 103
Gram-Schmidt; 185; 191
graph; 159; 192
Graph; 215

H

heat; 217
Hilbert; 138; 173
hill- conditioned; 38
hill-conditioned; 31
homogeneous linear system; 45; 181
Householder; 94; 173

I

identity; 154
incomplete matrix; 49
Input Output Analysis; 219
integer matrices; 23
Interpolate; 202
interpolation; 122
inverse matrix; 41

J

Jacobi; 79; 80; 155; 157; 158; 163; 180

K

Kaiser; 207

L

least square; 114
Leontief; 219
Lin-Bairstow; 167

Linear Function; 181
 Linear Interpolation; 124
 linear regression; 201
 linear system; 19; 211
 Linear Transformation; 181; 183
 LINEST(); 115; 117
 logarithmic; 114
 LU decomposition; 187

M

M_ABS; 145
 M_ADD; 145
 M_BAB; 145
 M_BAB(); 78
 M_DET; 29; 146
 M_DET(); 19
 M_DET_C; 146
 M_DET3; 147
 M_DIAG; 152
 M_DIAG_ERR; 153
 M_EXP_ERR; 149
 M_ID; 154
 M_ID(); 72
 M_INV; 147
 M_INV_C; 210
 M_MAT_C(); 77
 M_MULT_C; 209
 M_POW; 71; 148
 M_PROD; 149
 M_PROD(); 55
 M_PROD_S; 150
 M_RANK; 153
 M_SUB; 151
 M_T; 153
 M_TRAC; 152
 M_TRAC(); 168
 M_TRIA_ERR; 154
 Macros stuff; 139
 Mat_Adm; 215
 Mat_Block; 159
 Mat_BlockParm; 160
 Mat_Cholesky; 187
 Mat_Cholesky(); 104
 Mat_Hessemberg; 215
 Mat_Leontief; 219
 Mat_LU; 187
 Mat_LU(); 53
 Mat_QR; 189
 Mat_QR(); 82
 Mat_QR_iter; 83
 MatCharPoly; 166
 MatCharPoly(); 69
 MatCmpn; 203
 MatCmpn(); 86
 MatCorr; 199
 MatCovar; 199
 MatDiagExtr; 152
 MatDiagExtr(); 80
 MatEigenvalue_Jacobi; 155

MatEigenvalue_Jacobi(); 104
 MatEigenvalue_pow; 169
 MatEigenvalue_QL; 163
 MatEigenvalue_QR; 85; 161
 MatEigenvalue_TridUni; 164
 MatEigenvalues_pow(); 91
 MatEigenvector; 162
 MatEigenvector_C; 162
 MatEigenvector_inv; 170
 MatEigenvector_Jacobi; 163
 MatEigenvector_Jacobi(); 104
 MatEigenvector_pow; 170
 MatEigenvector_pow(); 91
 MatEigenvector3; 165
 MatExtract; 191
 MatExtract(); 40
 MathCharPoly(); 103
 MatMopUp; 199
 MatMopUp(); 82
 MatNorm; 208
 MatOrtNorm; 191
 MatPerm; 66; 214
 Matrix Generator; 137
 Matrix tool; 134
 MatRnd; 173
 MatRndEig; 173
 MatRndEigSym; 173
 MatRndEigSym(); 95
 MatRndRank; 173
 MatRndSim; 173
 MatRndUni; 173
 MatRot; 204
 MatRotation_Jacobi; 158
 MDETERM; 29
 MDETERM(); 38
 menubar; 134
 minors; 40
 MINVERSE; 28; 29
 MINVERSE(); 41
 mop-up; 123

N

network; 216
 Network; 216; 217
 Newton-Girard; 69; 166
 Nodal Analysis; 216
 Norm; 208
 normalized; 207

O

optimization; 212
 orthogonal; 81; 145; 158; 189; 204; 206
 orthogonalization; 191
 orthonormal; 81

P

Parabolic Interpolation; 124
 Parametric form; 46

partial pivoting; 22
 partial-pivot; 30
 partioned; 159
 partitioned; 160
 Paster tool; 135
 Path; 192
 Path_Floyd; 192; 196
 Path_Min; 192
 permutation; 159; 160
 permutations; 66; 214
 piecewise polynomial interpolation; 123
 Pivoting; 21
 Poly_Roots; 166
 Poly_Roots(); 69; 103
 Poly_Roots_QR; 87; 204
 polynomial; 167; 203
 polynomial regression; 202
 positive definite; 187
 power; 148
 power's iterative method; 169
 Powers; 88
 ProdScal; 154
 ProdScal(); 82
 ProdScal_C; 210
 product; 149
 production; 219
 ProdVect; 155

Q

QR; 79; 82

R

random; 137
 rank; 137; 153
 Rank; 47
 rational; 114
 reduction; 91
 REGRL; 201
 REGRL(); 110
 REGRP; 202
 REGRP(); 111
 regularization; 123
 residuals; 214
 root mean squares; 214
 rotation; 158; 204; 206
 ROUCHÉ-CAPELLI; 49
 ROUND; 29
 Round-off error; 42
 round-off errors; 28; 199
 RRMS; 214

S

scalar product; 81; 154
 Selector tool; 134
 sensitivity; 32; 171
 shortest-path; 192
 similarity; 94; 145
 Simplex; 212
 Simultaneous Linear Systems; 41
 Singular linear system; 181
 Singular Value Decomposition; 197
 smoothing; 123
 stability; 32
 sub tabulation; 122
 Subspace; 47
 SVD; 33; 197; 201; 205
 SYSLIN; 10; 19; 28; 29; 177
 SYSLIN_C; 211
 SYSLIN_C(); 34; 35
 SYSLIN_ITER_G; 179
 SYSLIN_ITER_J; 180
 SYSLIN_T(); 52
 SYSLIN3; 178
 SYSLINSING; 181

T

Tartaglia; 138; 173
 Tartaglia's matrices; 43
Technology; 219
 temperature; 217
 trace; 152
 transpose; 153
 TRASFLIN; 183
 triangular; 160; 187; 189
 tridiagonal; 97
 Tridiagonal; 95

U

uniform; 95; 97
 unitary; 174; 214
 unstable; 31

V

Vandermonde; 173
 Varimax; 206; 207
 VarimaxIndex; 207
 VarimaxRot; 206
 vector product; 155



© 2004, by Foxes Team
Piombino, ITALY
leovlp@libero.it

4. Edition
4 Printing: May. 2004