Volume

1

FOXES TEAM

Tutorial of Numerical Analysis with Matrix.xla

Matrices and Linear Algebra

TUTORIAL OF NUMERICAL ANALYSIS FOR MATRIX.XLA

Matrices and Linear Algebra

ITALY
leovlp@libero.it

5. Edition
1 Printing: June 2005

Index

Chapter

1

About this tutorial

MATRIX.XLA

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

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

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

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

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

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

The main purpose of this document is to show how to work with matrices and vectors in Excel and how to use matrices for solving linear systems. This tutorial is written with the aim to teach how to use the Matrix.xla functions. Of course it speaks about math and the 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.

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

Chapter

2

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

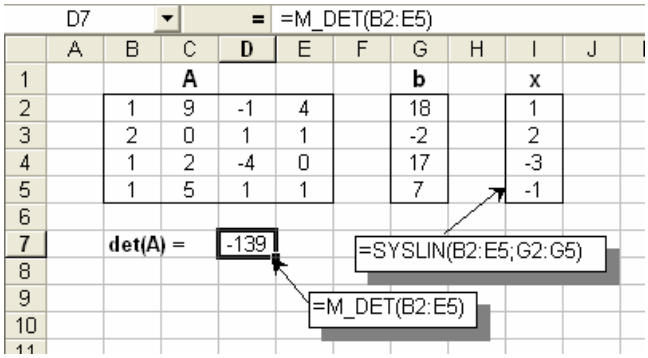
Example 1. Solve the following 4x4 linear system

A x = b

Where **A** and **b** are:

$$\begin{vmatrix} 1 & 9 & -1 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 2 & -4 & 0 \\ 1 & 5 & 1 & 1 \end{vmatrix} \quad \begin{vmatrix} 18 \\ -2 \\ 17 \\ 7 \end{vmatrix}$$

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 **A** is not zero. That is, **A** is non-singular. In that case we can solve the problem with the **SYSLIN** function.



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 triangular[1] or diagonal[2] 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.

$$\begin{cases} 4x_1 + x_2 = -2 \\ -2x_1 - 2x_2 + x_3 = -1 \\ x_1 - 2x_2 + 2x_3 = -3 \end{cases} \Leftrightarrow$$

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

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.

[1] Properly called Gauss algorithm
[2] 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	-0.8333	{=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 the second row has now 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 of 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 simply 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 now 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 current pivot; if the max value is greater then the current pivot, then the row of the pivot 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 a_{33} only the element a_{33}, a_{43}, a_{53}, a_{63} (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 a_{63} 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 pivot we could exchange rows and columns. But when we swap two columns the corresponding unknown variables are also exchanged. So, to rebuild the final solution in the original given sequence, we have to perform all the permutations, in reverse order, that we have done. This makes the final algorithm a bit more 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

The 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 the solutions and coefficients of the system are integer.

There is a way to avoid such decimal round off error and preserve the global 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 manually by 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 pivot row and the row that has 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	<== pivot row; multiply for 2
4	3	0	10	<== for reducing; multiply for 3

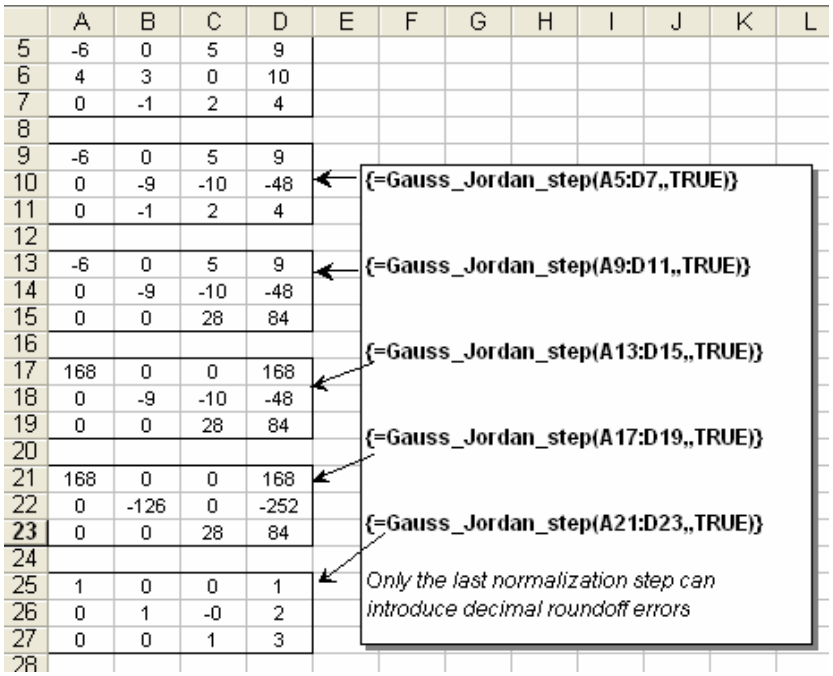
-12	0	10	18
12	9	0	30

now, add the two rows

-6	0	5	9	<== the first row remain unchanged
0	9	10	48	<== substitutes the result to the second row

As we have seen, 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.

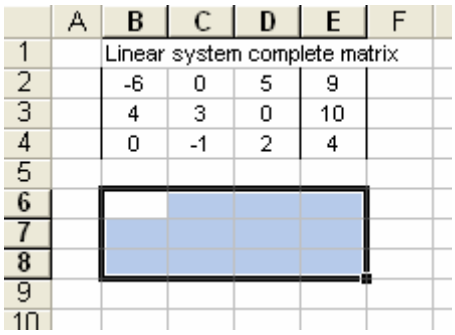


Note the 3rd parameter setting the integer algorithm. If "false", the operations will perform in the standard decimal way.

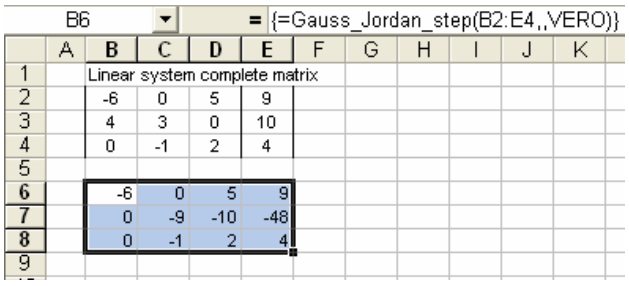
Only the last step could introduce decimal numbers; the previous steps are always exact. Unfortunately, this method cannot be adopted in general because the values grow up at each step and they can became too large (overflow error)

Tip The above example can be quickly 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 to do.

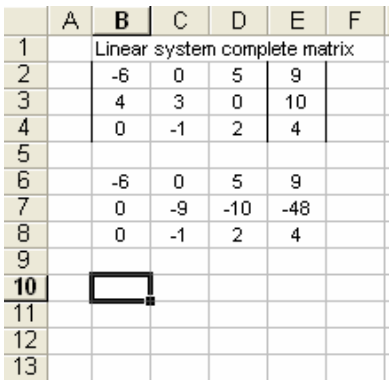


Given a complete system matrix in range B2:E4, select the range A6:E8, just below the given matrix



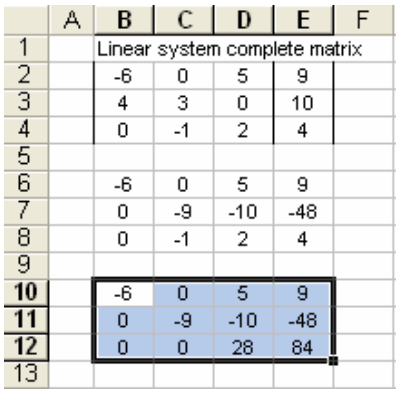
Insert the array function Gauss_Jordan_step with the CTRL+SHIFT+ENTER key sequence and the given parameter ("VERO" means "TRUE" in English)

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



	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 can get all the Gauss-Jordan step-matrices, 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 same 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

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

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

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

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 the 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

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

Non Singular Linear system

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

Example: solve the following matrix equation

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

The solution is

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

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 **A** (6 x 6) and **b** (6 x 1)

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

47.7
-20.5
-3
-47
100.1
-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 A x = b**						**b**	**x**	**A^{-1} 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 theirs algorithms are different. In this case both the solutions are very accurate (≈1E-15) but this is not always true.

Round-off errors

Many times, the round-off errors can decrease the maximum accuracy obtained
Look at the following system:

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

387
194
599989
0
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	1.000000000000010	1.09E-14	1.000000000000000	3.55E-15
6	-100000	200000	0	0	100001	200001	0.999999999999989	1.1E-14	1.000000000000040	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]

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

Let's see how both algorithms - full and partial pivoting - work[3].

	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	1.00000000000001	8.7E-15	1.00000000000006	6E-14
3	-15	55	-85	69	-29	5	1	2.00000000000003	2.9E-14	2.00000000000020	2E-13
4	20	-85	146	-127	56	-10	0	3.00000000000008	7.6E-14	3.00000000000045	4.5E-13
5	-15	69	-127	117	-54	10	0	4.00000000000017	1.7E-13	4.00000000000086	8.6E-13
6	6	-29	56	-54	26	-5	0	5.00000000000036	3.6E-13	5.00000000000148	1.5E-12
7	-1	5	-10	10	-5	1	0	6.00000000000066	6.6E-13	6.00000000000232	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.

Look at this example: Solve the following system

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

A =

1	-3	-9	-1	38800000012	38800000000
7	-1	12300000045	1	0	12300000052
0	1	0	-2	2	1
23	-12	6	4	1	22
2	0	-6	0	-1	-5

Solving with Gauss-Jordan algorithm with both partial and full pivoting we note in this case a lack age 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	12300000045	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 heavy - 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 **x** = (1,1,1,1); now give same perturbations to the constant terms. For simplicity we give

b' = **b** + Δ**b** with **b** = 0.1

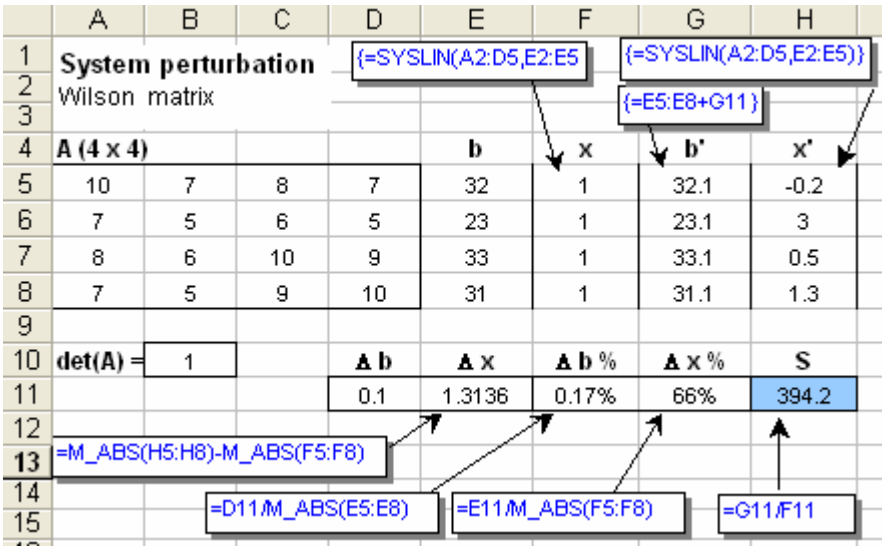
The solution of the perturbed system is now

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

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 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

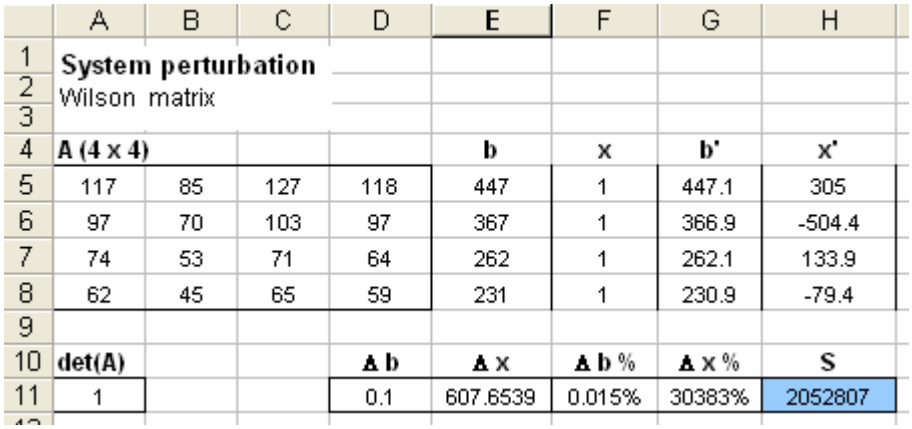
Completely different from the exact one

1, 1, 1, 1

Note that Det =1

Even worst for the stability of the following linear system

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



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

λ=	323.98	-5.72328	-1.256573	0.000429

So the instabilty factor will be:

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

While the SVD decomposition gives

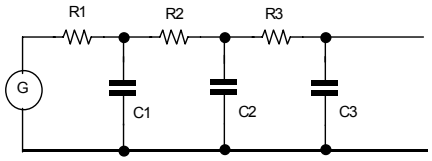
340.9215	0	0	0
0	7.879412	0	0
0	0	1.208233	0
0	0	0	0.000308

$S_D = 340.9 / 0.000308 \cong 1106504$

Complex systems

Complex systems are very common in applied science. Matrix.xla has a dedicated function SYSLIN_C to solve them.
We shell to learn how it works with a practically example from the Network theory.

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



Components values

R1 = 100 Ω C1 = 1.5 μF
R2 = 120 Ω C1 = 2.2 μF
R2 = 120 Ω C1 = 2.2 μF
G = 2.5 sin(2π f t)

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

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

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

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

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

Using 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).

	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		**Node voltages**			**Modulo**	**phase**		
8	C3	2.2E-06	F	v1 =	1.39368	-0.658		1.54122	-25.3		
9	G	2.5	V	v2 =	0.36378	-0.8172		0.89452	-66.0		
10	f	400	Hz	v3 =	-0.1239	-0.735		0.74537	-99.6		
11	ω	2513.274	rad/s		{=SYSLIN_C(D3:I5,J3:K5)}						
12											

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

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

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		**Node voltages**		**Modulo**		**phase**		
8	C3	2.2E-06	F	v1 =	{=SYSLIN_C(D3:I5,J3:K5)}		=M_ABS(E8:F8)		=ATAN2(E8,F8)*180/PI()		
9	G	2.5	V	v2 =			=M_ABS(E9:F9)		=ATAN2(E9,F9)*180/PI()		
10	f	400	Hz	v3 =			=M_ABS(E10:F10)		=ATAN2(E10,F10)*180/PI()		
11		2513.3	rad/s								

See also the function Mat_Adm for admittance matrix.

Example - Solve the following complex system

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

The system is equivalent to the following complex matrix equation

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

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 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 "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 other Excel numbers.

Determinant

Differently from the solution of linear system, the 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 rules

- 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

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

A1=	4	1	0	1	R2 = R1 + 2*R2 (*)
	0	-3	2	2	Det(A1) = 2 Det(A)
	1	-2	2		
A2=	4	1	0	1	R3 = R1 + (– 4)*R3
	0	-3	2		Det(A2) = -8 Det(A)
	0	9	-8	-4	
A3=	4	1	0		
	0	9	-8	< swap	Det(A3) = 8 Det(A)
	0	-3	2	< swap	
A4=	4	1	0		R3 = R2 + 3*R3
	0	9	-8	1	Det(A4) = 24 Det(A)
	0	0	-2	3	
A4=	4	1	0		Det(A4) = 24 Det(A)
	0	9	-8		Det(A4) = 4*9*(-2) = -72
	0	0	-2		

(*)
The formula
R2 = R1 + 2*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

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

But it is also:

$$Det(A_4) = 24\ Det(A)$$

Substituting, we have:

$$-72 = 24\ Det(A) \quad \Rightarrow \quad Det(A) = -24 / 72 = -3$$

Hill-conditioned matrix

Of course there are functions such as M_DET in Matrix.xla and 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 precision of the computer. It may happen 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

127	-507	245	< swap
-507	2025	-987	< swap
245	-987	553	

Det(A1) = -1 Det(A)

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

Det(A2) = 507 Det(A)

-507	2025	-987	-245
0	-126	1134	
245	-987	553	-507

Det(A3) = -257049 Det(A)

-507	2025	-987	
0	-126	1134	< swap
0	4284	-38556	< swap

Det(A4) = 257049 Det(A)

-507	2025	-987	-476
0	4284	-38556	225
0	-126	1134	

Det(A5) = -122355324 Det(A)

241332	0	-8205288	
0	4284	-38556	1
0	-126	1134	34

Det(A6) = -4160081016 Det(A)

241332	0	-8205288
0	4284	-38556
0	0	0

Det(A6) = 0 ==> Det(A) = 0

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

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. Laplace's 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 $|\mathbf{A}_{ij}|$ 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.

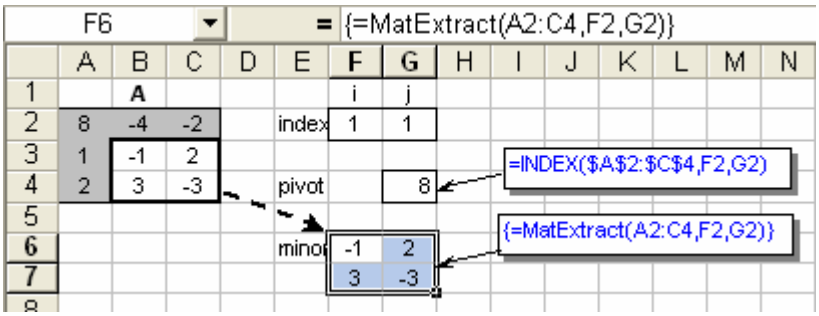
Many authors call *cofactor* the term: $(-1)^{(i+j)}|\mathbf{A}_{ij}|$.

Let's see how it 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



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	=INDEX($A$2:$C$4,J2,K2)					
5																	
6					minor	-1	2	1	2	1	-1						
7						3	-3	2	-3	2	3	{=MatExtract(A2:C4,J2,K2)}					
8		=SOMMA(G9:K9)															
9	|A| =	-62			cofactor		-24		-28		-10	=(-1)^(J2+K2)*K4*M_DET(J6:K7)					
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

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

Where:

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

The solution is

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

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:

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

You have the function **MINVERSE** in Excel or the function **M_INV** in Matrix.xla to invert a square matrix.

Example: find the inverse of the 4 x 4 Hilbert matrix

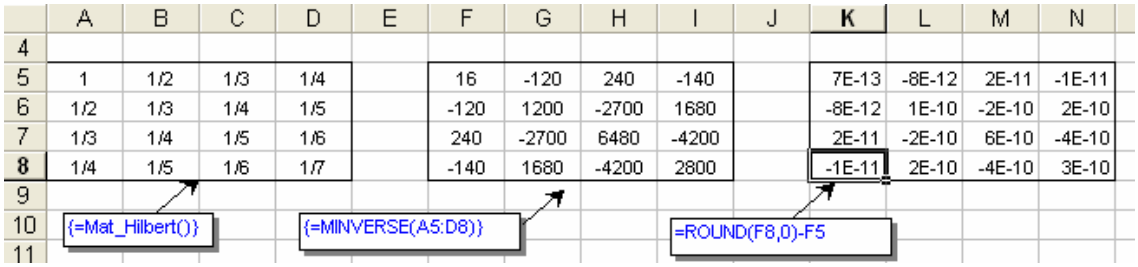
$$\begin{vmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{vmatrix}$$

Hilbert matrices are a known class of hill-conditioned matrices,
very easy to generate:

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

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														
11														

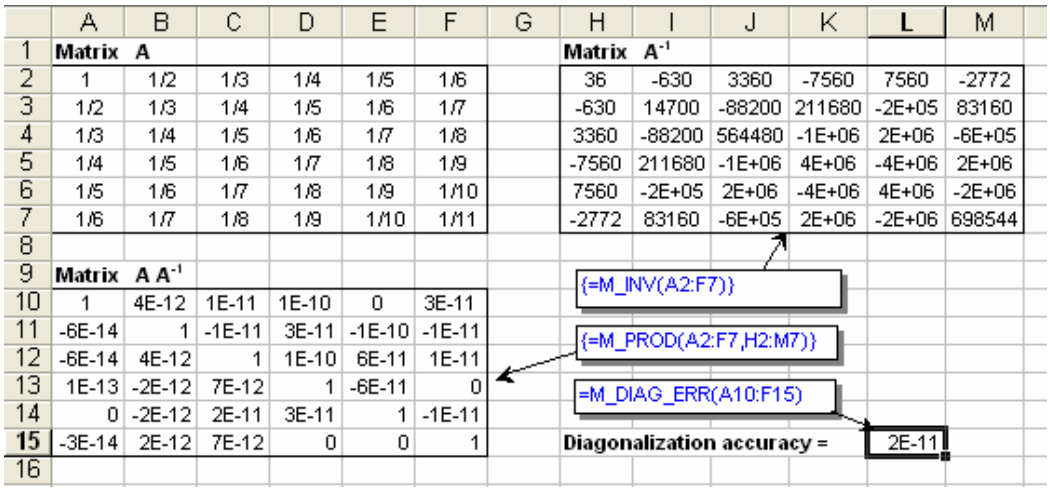
Round-off error

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 measures 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							
11	-6E-14	1	-1E-11	3E-11	-1E-10	-1E-11							
12	-6E-14	4E-12	1	1E-10	6E-11	1E-11							
13	1E-13	-2E-12	7E-12	1	-6E-11	0							
14	0	-2E-12	2E-11	3E-11	1	-1E-11							
15	-3E-14	2E-12	7E-12	0	0	1		**Diagonalization accuracy =**				2E-11	
16													

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

Global error = Input matrix error + Inversion + 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 about 1E-15.

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

Tartaglia's matrices

Tartaglia's matrices are very useful because they are easy to generate but - this is very important - the matrix and its inverse are always integer. This comes in handy for testing the algorithm round-off error.

They are defined with:

$a_{1j} = 1$ for j =1 ...n (all 1 in the first row)
$a_{i1} = 1$ for i =1 ...n (all 1 in the first column)

$$a_{ij} = \Sigma_j \, a_{(i-1)\,j} \quad \text{for } j = 2 \,..\, n$$

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 taht 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))										
11													

{=M_INV(A2:F7)}

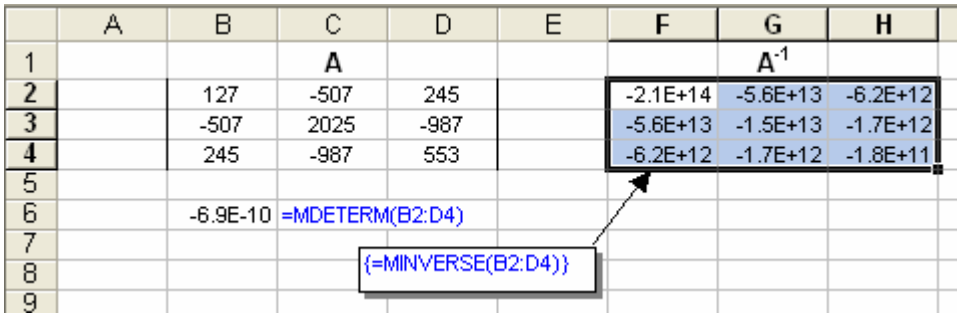
Sometimes however, Excel produces errors. Excel rounds numbers and will occasionally compute $\mathbf{A}^{-1}$ even if a matrix has a determinant equal to zero. If this happens, your solution will be wrong.

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 a 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

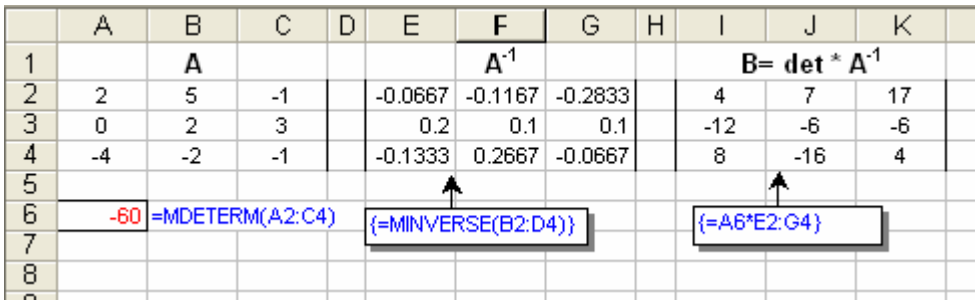


Tip. You should always examine the determinant. If the determinant is close to zero, you should try to verify the solution with other methods. For instance, you can always try to solve the inverse by the function M_INV (with integere option), or by Gauss_Jordan_Step function, or with SVD decomposition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

How to avoid decimal numbers

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



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 **b** = 0

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

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

Assume 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 linear dependent, the given matrix has determinant = 0; that is singular. One of the two rows can be eliminate; we choose to eliminate the last row obtaining the following system

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

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

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

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 $\mathbf{R^1}$

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

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

If we assume, on the contrary, x as independent parameter, the other variables y, z can be express as function of the 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 is a line with the direction vector: $\vec{D}$

given by

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

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

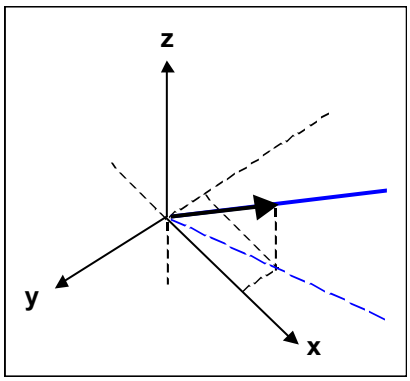
You can study the entire problem by the function **SYSLINSING** of Matrix.xla. Here the example:

	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					{=G16:G18/M_ABS(G16:G18)}				
23									

SYSLINSING solve a linear singular system, returning the transformation matrix, if exists, of the solution. 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 ≠ 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 norm of the vector we have used the function M_ABS. Note that both expression must be insert as array functions { }

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



Rank and Subspace

In the above example we have seen that if the matrix of a homogeneous system is singular, then there are infinite solutions of the system; those solutions, in a RCO represent a subspace. After that we have find the solution, and 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 the system? The answer is yes, knowing the rank of the matrices. But we have to say that 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 matrix 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 solving its homogeneous system.

	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)								
21	1	=M_RANK(A16:C18)								
22										
23		{=SYSLINSING(A16:C18)}								

{=F16:F18/M_ABS(F16:F18)}

{=G16:G18/M_ABS(G16:G18)}

Consequently, the transformation matrix has two columns indicating that the subspace has 2 dimensions, thus is 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) . The 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 equations of the plane

$$x = -2y + z \quad \Rightarrow \quad x + 2y - z = 0 \qquad (2)$$

The linear equation (2) express 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 form the determinant, the rank can be computed also for rectangular matrix. Its definition is:

- *Rank of a rectangular matrix is the max number of independent rows (or 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 2 independent rows and 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 always given by: rank = 2

One popular theorem - due to Kroneker - 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 singulars

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 are 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

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

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

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

$$B(m, n+1) = \begin{bmatrix} a_{11} & a_{12} & ..... & a_{1n} & b_1 \\ a_{21} & a_{22} & ..... & a_{2n} & b_2 \\ ..... & & & & \\ a_{m1} & a_{m2} & ..... & a_{mn} & b_3 \end{bmatrix}$$

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

ROUCHÈ-CAPELLI THEOREM.

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

That is: rank(**A**) = rank(**B**) $\Leftrightarrow$ $\exists$ **x** solution

Note: This rule is always valid for homogeneous systems that are always the **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	rank(A)= m =n	Trivial solution (0,0,..0)	$\begin{cases} 2x+3y=0 \\ x-3y=0 \end{cases} \quad S(0,0)$
2	rank(A)= m <n	∞^{n-m} solutions + trivial solution	$\begin{cases} 2x+3y+z=0 \\ x-3y+2z=0 \end{cases} \quad S\left(-z,\frac{1}{6}z,z\right)(\infty^1 soluzioni) + S(0,0,0)$
3	rank(A)< m <n	∞^{n-r} solutions + trivial solution	$\begin{cases} 2x+3y+z=0 \\ 4x+6y+2z=0 \end{cases} \quad S\left(\frac{-3y-z}{2},y,z\right)(\infty^2 soluzioni) + S(0,0,0)$
4	rank(A)< m =n	∞^{n-r} solutions + trivial solution	$\begin{cases} x-2y=0 \\ 2x-4y=0 \end{cases} \quad S(2y,y)(\infty^1 soluzioni) + S(0,0)$
5	rank(A)= n <m	Trivial solution (0,0,..0)	$\begin{cases} x+2y=0 \\ 3x-2y=0 \\ x-y=0 \end{cases} \quad S(0,0)$
6	rank(A)<n <m	∞^{n-r} solutions + trivial solution	$\begin{cases} x+2y=0 \\ 2x+4y=0 \\ 3x+6y=0 \end{cases} \quad S(-2y,y)(\infty^1 soluzioni) + S(0,0)$

Non Homogeneous System Cases

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

Triangular Linear Systems

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

Triangular factorization

Having the linear system

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

Suppose you have got the following factorization

$$\mathbf{A = LU} \qquad (2)$$

Where **L** is lower-triangular and **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 split the linear system (1) into two systems:

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

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

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

Forward and Back substitutions

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

Let' see how it works

Having the following factorization **LU** = **A**, solve the linear system **A x** = **b**

A

6	5	1
12	8	6
-6	-6	5

b

19
46
-3

L

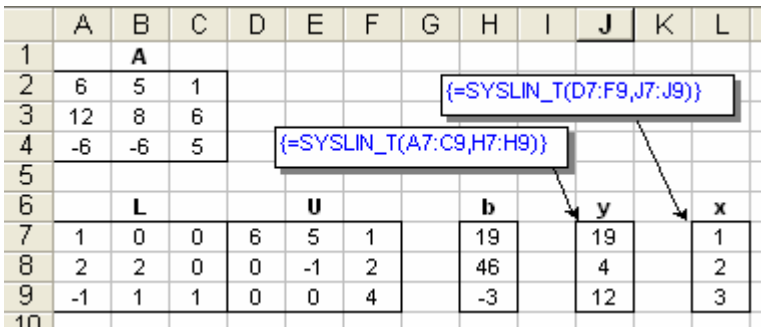
1	0	0
2	2	0
-1	1	1

R

6	5	1
0	-1	2
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 omitted, the function tries to detect by itself the matrix type



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. This is a very efficient and popular method to solve linear systems and to invert matrices. In Matrix.xla this algorithm is performed by the **Mat_LU** function. This function returns both factors in a (n x 2n) array.

But there are same 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 as many linear system as we want, simple changing the vector b. This is not completely true and it may induce in wrong results.

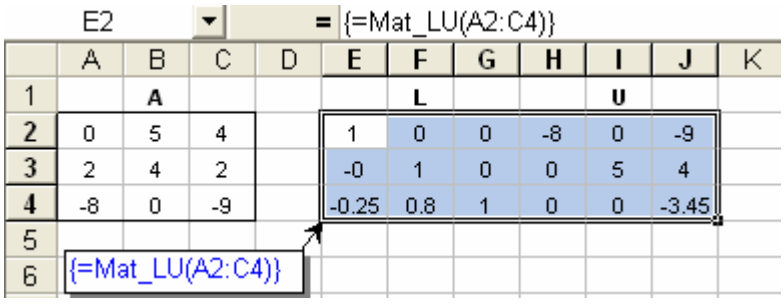
Look at this example..

A x = b where:

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

b
22
16
-35

If we compute the LU factorization we have:



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 (3) **U x = y** (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 **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 **A** but a new matrix **A'** that is a rows permutation of the given one. This is due to the partial pivoting strategy of the Crout's algorithm. You simple prove it by multiplying L and U.

So the right factorization formula would be:

$$\mathbf{A = PLU}$$

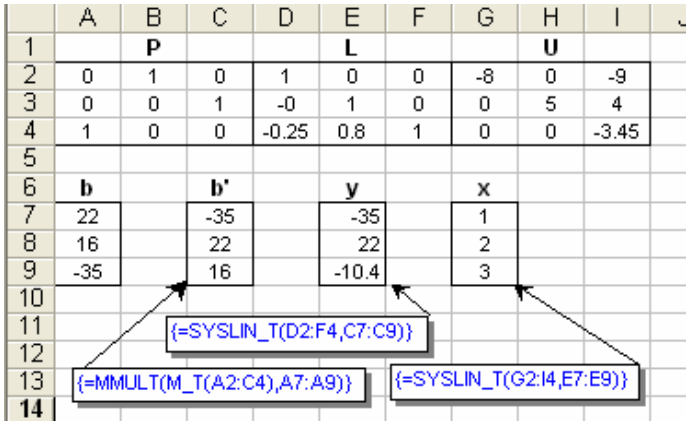
Where **P** is a permutation matrix

The process to solve the system is therefore:

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

We have shown that only the information of the two factors **L** and **U** are no sufficient to solve the general system. We must complete it with the **P** matrix.

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



The original system is broken into two triangular systems

A x = b

b' = P^T b

L y = b'

U x = y

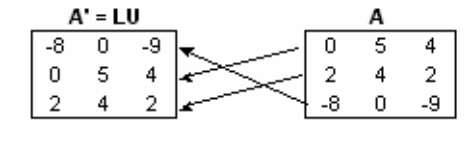
The permutation matrix can be obtained comparing the original **A** matrix and the matrix obtained from the product **A' = LU**. Let' see how.

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

$$u_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad , \quad u_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad , \quad u_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

We examine now the matrix rows of the two matrices **A'** and **A**.

The row 1 of A' comes from row 3 of A, ⇒ p1 = u3
The row 2 of A' comes from row 1 of A, ⇒ p2 = u1
The row 3 of A' comes from row 2 of A, ⇒ p3 = u2



So the permutation matrix will be:

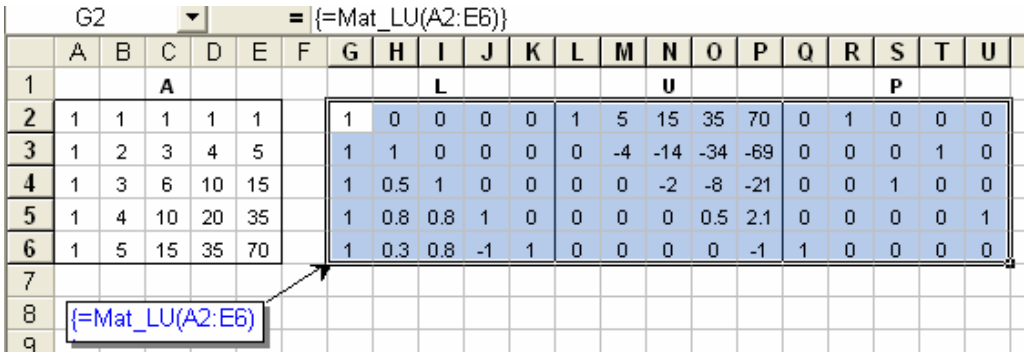
$$P = (p_1\ ,\ p_2\ ,\ p_3) = (u_3\ ,u_1\ ,u_2) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Clearly this process can be very tedious for larger matrices. Fortunately the permutation matrix is supply by the function Mat_LU as the 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**) arrays

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

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



	A	B	C	D	E
1					
2	1	1	1	1	1
3	1	2	3	4	5
4	1	3	6	10	15
5	1	4	10	20	35
6	1	5	15	35	70

	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1			L					U					P		
2	1	0	0	0	0	1	5	15	35	70	0	1	0	0	0
3	1	1	0	0	0	0	-4	-14	-34	-69	0	0	0	1	0
4	1	0.5	1	0	0	0	0	-2	-8	-21	0	0	1	0	0
5	1	0.8	0.8	1	0	0	0	0	0.5	2.1	0	0	0	0	1
6	1	0.3	0.8	-1	1	0	0	0	0	-1	1	0	0	0	0

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 **P** must be the first matrix of the product)

Block-Triangular Form

Square sparse matrices, thus matrices 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

$\mathbf{A}_1$	**0**
$\mathbf{A}_{21}$	$\mathbf{A}_2$

The block-triangular form saves a lot of computation effort for many important problems of linear algebra: linear system, determinant, eigenvalues, 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

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

$$\begin{bmatrix} 1 & 2 & 1 & 0 & 0 & 0 \\ 2 & 1 & 5 & 0 & 0 & 0 \\ 1 & -1 & 3 & 0 & 0 & 0 \\ -6 & 5 & 3 & 1 & 1 & 2 \\ 1 & -3 & 2 & 1 & -1 & -2 \\ -9 & 7 & 1 & 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix}$$

It could be written as

$$\mathbf{A}_1\,\mathbf{x}_1 = \mathbf{b}_1$$

$$\mathbf{A}_2\,\mathbf{x}_2 = \mathbf{b}_2 - \mathbf{c}_2$$

where the vector $\mathbf{c}_2$ is given by: $\mathbf{c}_2 = \mathbf{A}_{21}\,\mathbf{x}_1$

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

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 5 \\ 1 & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 2 \\ 1 & -1 & -2 \\ 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_4 \\ b_5 \\ b_6 \end{bmatrix} - \begin{bmatrix} -6 & 5 & 3 \\ 1 & -3 & 2 \\ -9 & 7 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

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 $\mathbf{A}_2$.

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

= 18

1	2	1
2	1	5
1	-1	3

= 3

1	1	2
1	-1	-2
1	2	1

= 6

Permutations

Differently form 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, exchanging the rows 2 and 5, followed by exchanging the columns 2 and 5, can be formally (but only formally!) written as.

$\mathbf{B} = \mathbf{P}^T \mathbf{A}\, \mathbf{P}$, where the permutation matrix is $\mathbf{P} = (e_1, e_5, e_3, e_4, e_2, e_6)$

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

P

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

$\mathbf{P}^T\mathbf{A}\,\mathbf{P}$

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

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

Note that the similarity transform keeps the original eigenvalues. Consequently the eigenvalues of the matrix **A** are the same of the matrix **B**

Eigenvalues Problem

The eigenvalue problem takes advantage from the block-triangular form.
For example, the following matrix (6 x 6) **A** has the eigenvalues:

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

A

-15	0	-16	0	0	0
10	2	11	0	0	0
8	0	9	0	0	0
1	3	5	3	0	-4
2	6	1	2	5	4
-4	9	-3	-6	-6	-1

λ

-7
-1
1
2
3
5

$\mathbf{A}_1$

-15	0	-16
10	2	11
8	0	9

$\mathbf{A}_2$

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

λ_1

1
2
-7

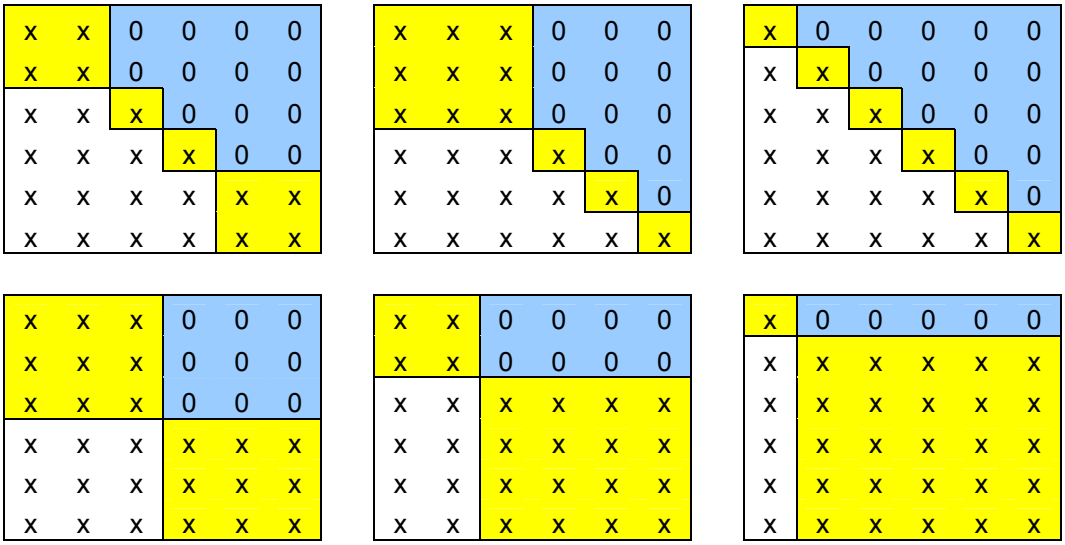
λ_2

-1
3
5

The eigenvalues set of the (6 x 6) matrix **A** is the sum of the eigenvalues set of $\mathbf{A}_1$ [1 , 2 , -7] and the eigenvalues set of $\mathbf{A}_2$ [-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 form; but there are many other schemas having blocks with different dimension each others. At last, all the blocks can have unitary dimension as in a triangular matrix.
Just below there are same example of block-triangular matrices (blocks are yellow)



Remark. The effort reduction is high when the dimension of the maximum block is low. In the first matrix the dimension of the maximum block is 2; in the second matrix is 3; in the third matrix the dimension is 1, showing the best effort reduction that it would be possible.
On the contrary, the last two 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 could 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

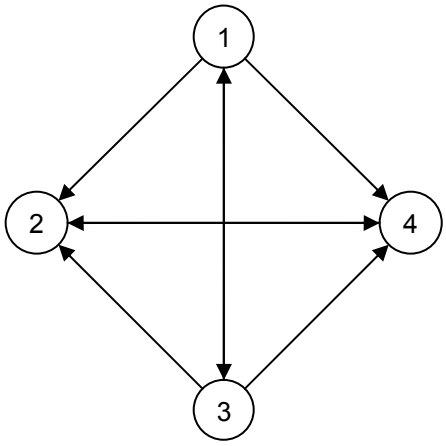
Following this method, we draw the graph of the given matrix following these simple rules:

- 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(**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(**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(**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(**A**) strong connected $\Leftrightarrow$ matrix **A** irreducible

G(**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 $\mathbf{P} = [\, e_2, e_4, e_1, e_3 \,]$, the similarity transform gives a block-triangular form

$$\mathbf{B} = \mathbf{P}^T \mathbf{A} \mathbf{P}$$

A

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

P

0	0	1	0
1	0	0	0
0	0	0	1
0	1	0	0

P^T A P

-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

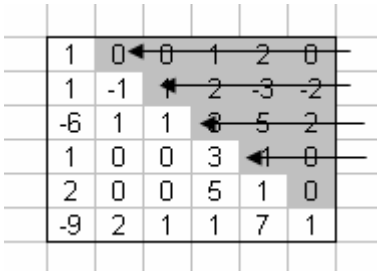
1	0	0	1	2	0
1	-1	1	2	-3	-2
-6	1	1	3	5	2
1	0	0	3	-1	0
2	0	0	5	1	0
-9	2	1	1	7	1

The score-algorithm

In this chapter we shall introduce a heuristic 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



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 firs zero on the same row, beginning from left to right.
The first 0 is the element a_{12}. We shell exchange the columns 2 e 5 and after, the rows 2 e 5

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

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

1 5 3 4 2 6

P

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

$P^TA\,P$

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

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

3 4

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

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:

A

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

1 2 4 3 5 6

P

1	0	0	0	0	0
0	1	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	0	0	1	0
0	0	0	0	0	1

$P^TA\,P$

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

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 ≠ 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 ≠ 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.

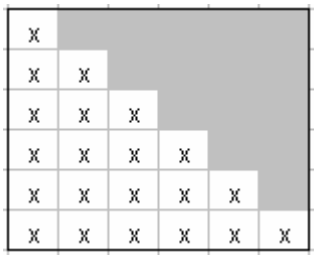
The Score-Function

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. Same of them will be useless and same 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.

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

$$w(i,j) = \begin{cases} 0 & \Leftrightarrow \ a_{ij} \neq 0 \\ (n-i+1)^2 \cdot j^2 & \Leftrightarrow \ a_{ij} = 0 \end{cases}$$

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

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	0	3	0	2	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

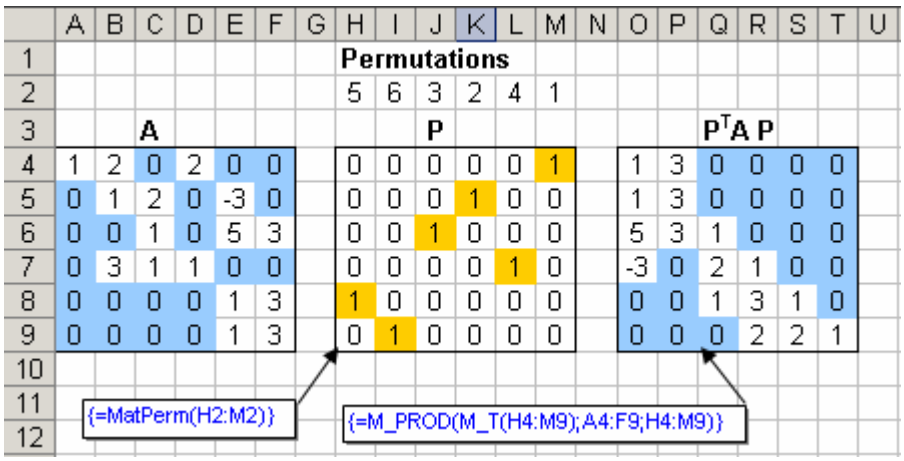
$P^T A P$

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

P = [e17, e19, e3, e12, e16, e6, e8, e1, e11, e20, e2, e10, e13, e14, e18, e15, e4, e7, e9, e5]
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 then any other arithmetic operation in floating point, we can guess the high speed of this algorithm

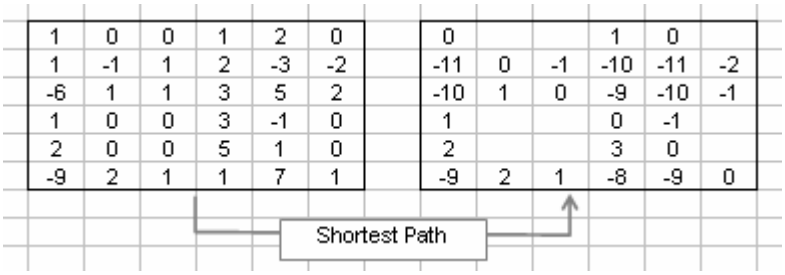
In Excel, with Matrix.xla, is very easy to study the matrix permutations.
A simple arrangement for (6 x 6) matrices is shown in the following example. We have used the function **MatPerm** . When you change the permutation numbers, also the permutation matrix changes an, consequently the final transformed matrix



The Shortest-Path-algorithm

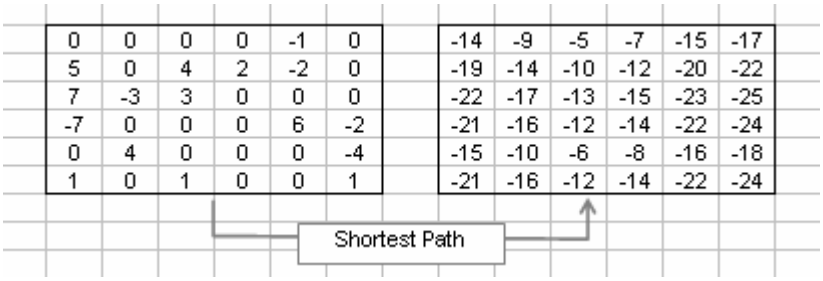
The above algorithm does not say if the matrix is irreducible. For that comes in handy the shortest-path matrix, built by the Floyd's algorithm. In Matrix.xla you can perform this by the function **Path_Floyd** or by the macro "**Macros>Shortest Path**"

Example. Say if the given matrix is reducible



The shortest-path matrix show the presence of empty elements. For example, the element a_{12} is null, meaning that there is no path reaching the node 2 from the node 1. This is sufficient for saying that the given matrix is not strong connected and thus, reducible.

Example. Prove that, on the contrary, the following matrix is irreducible



The shortest-path matrix is dense, meaning that every node can be reached from any other. For definition, the given matrix is strong connected and thus, irreducible

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 code, 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 x n) with its elements a_{ij} randomly distributed from -k to k. With this hypothesis the determinant grows roughly as:

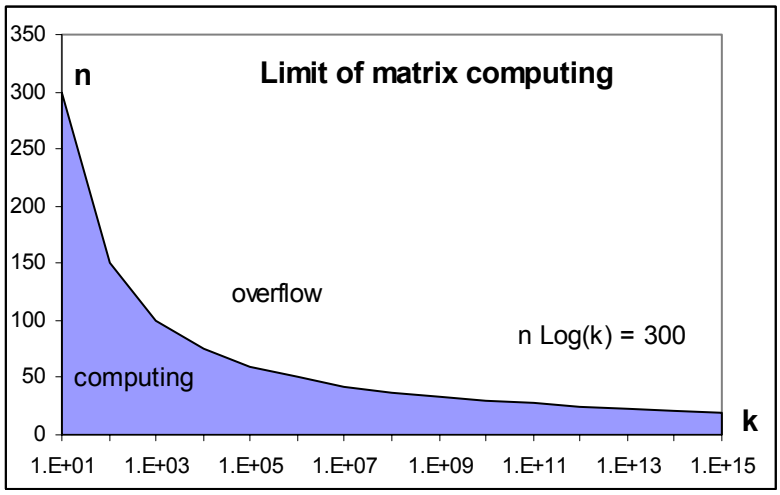
$$\text{Log}(|D|) \cong n\,\text{Log}(k) + 0.0027 \cdot n^2 \cong n\,\text{Log}(k)$$

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 1E+300, 1E-300. So if we want to avoid the overflow/underflow error, we must constrain:

$$300 \geq n\,\text{Log}(k) \qquad (1)$$

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 x 80) having values no more than 1000, the point (1000, 80) falls into the blue area; so you will be able to performs this operation.
On the contrary, If you have a matrix (80 x 80) having values up to 1E+7, the point (1E+7, 80) falls into white area; so you will probably get an overflow error

From this graph we see that matrices (25 x 25) or less, can be elaborated for all values, while matrices (100 x 100) or more can be computed only if their values are less that 1000

Of course this result is valid only for generic dense matrices not ill-conditioned. If the matrix is ill-conditioned you could get an overflow/underflow error even for low/moderate dimension. 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. 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 << 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

WHITE PAGE

Chapter

3

Eigen-problems

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

Eigen-problems

Eigenvalues and Eigenvectors

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

$$A\,x = \lambda\,x \qquad (1)$$

Where **A** is n x n matrix and the unknowns are λ and **x**, respectively called *eigenvalue* and *eigenvector*[4]. Rearranging the equation (1) we have:

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

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

$$\left|A - \lambda\, I\right| = 0 \qquad (3)$$

Characteristic Polynomial

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

For a (2x2) matrix, the system (2) becomes:

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

[4] 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) matrix, the 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 the polynomial coefficients using the Newton-Girard recursive formulas. In Matrix.xla we can get these coefficients 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 real polynomial, using the *Lin-Bairstow* method. This function is suitable for general polynomials up to 6-7 degree. When possible, the function uses the Ruffini's method for finding small integer roots.

There is also the function **Poly_Roots_QR** for finding all polynomial roots. It uses the efficient QR algorithm and it is adapt for polynomial up to 10-12 degree.

For complex polynomials there is the similar function **Poly_Roots_QRC**

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 eigen-system solution; much more complicated, instead, for asymmetric matrices.

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

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

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

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

2	0
0	1

eigenvalues: 1, 2

7/3	2/3	0
2/3	6/3	-2/3
0	-2/3	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

8/3	4/3	3/3	2/3	1/3	0
4/3	9/3	2/3	1/3	0	-1/3
3/3	2/3	10/3	0	-1/3	-2/3
2/3	1/3	0	11/3	-2/3	-3/3
1/3	0	-1/3	-2/3	12/3	-4/3
0	-1/3	-2/3	-3/3	-4/3	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(λ) an important theorem, known as *Cayley-Hamilton's theorem* - stats that the any square matrix **A** verifies its characteristic polynomial. That is, in formula:

P(**A**) = **O** (where O is the null matrix)

The above matrix 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

A =

11	9	-2
-8	-6	2
4	4	1

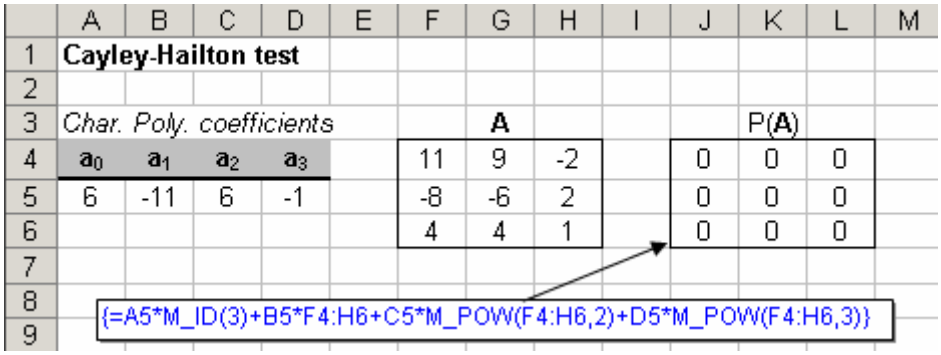
Its characteristic polynomial is:

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

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

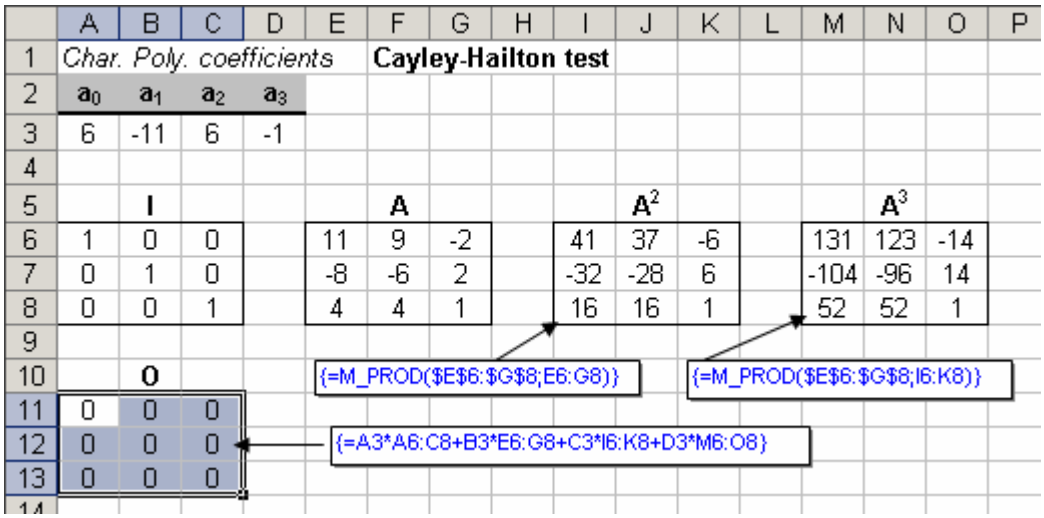
$$P(A) = 6 \cdot I - 11 \cdot A + 6 \cdot A^2 - A^3$$

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_POW**



Note that we have inserted the P(**A**) formula as an array function {=….}

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



Eigenvectors

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

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

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 explained in this chapter, 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 it works with same examples

Example - Simple eigenvalues

Find all eigenvalues and associated eigenvectors of the following matrix

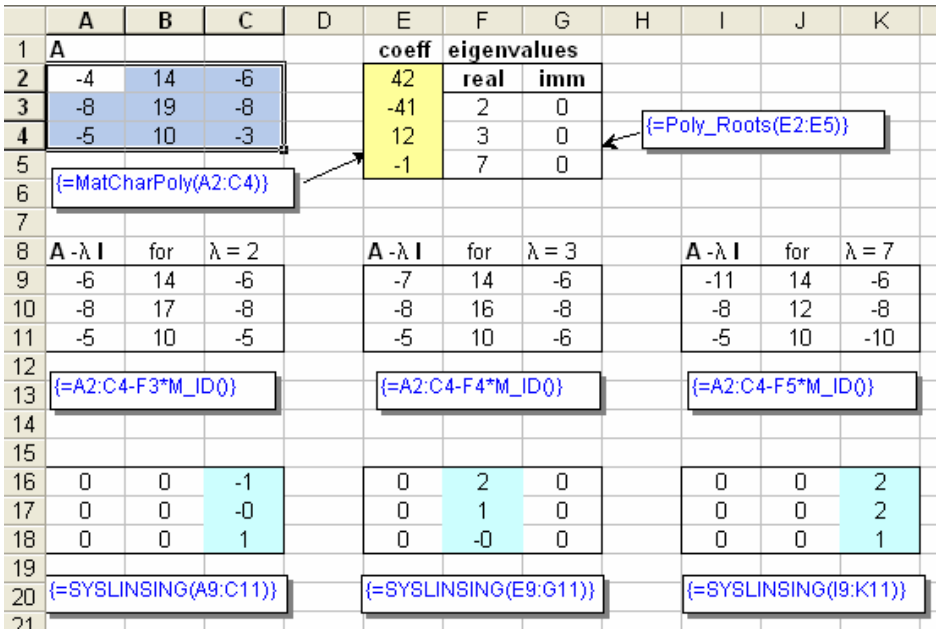
A =

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

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



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

Eigenvalues	
λ1	2
λ2	3
λ3	7

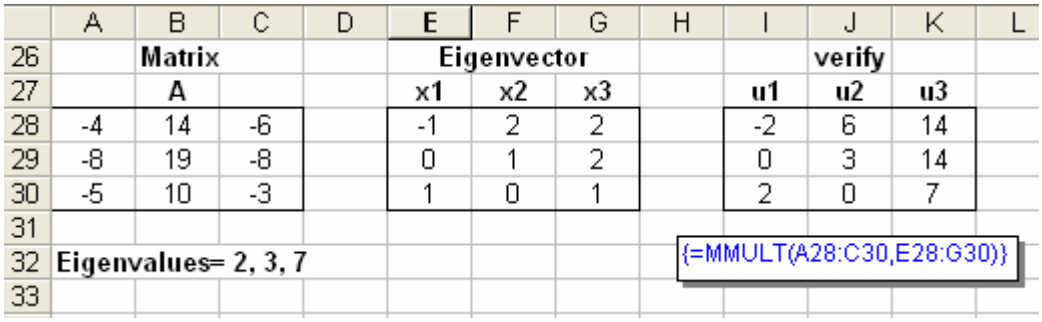
Eigenvector		
x1	**x2**	**x3**
-1	2	2
0	1	2
1	0	1

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 bellow



Eigenvectors are not unique. It can be easy prove 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 …						
-4	14	-6		-0.04	-0.5	-1	-2	-3	-4	-5
-8	19	-8	λ =2	0	0	0	0	0	0	0
-5	10	-3		0.04	0.5	1	2	3	4	5

For convention, mathematicians use to take the eigenvector with norm 1, that is: | **x** | =1. In that case it is called *eigenversor*.

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

Sometime, in order to avoid decimals numbers, we normalize only the smallest value of the vector; for that, we divides all values for the GCD

	A	B	C	D	E	F	G
49	Eigenvector				Eigenversor		
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 eigenversors you have to do it 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:

λ = 1 , m = 1

λ = 2 , m. = 2

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

	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							
13	{=A2:C4-C8*M_ID()}				{=A2:C4-F4*M_ID()}		
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:C1				{=SYSLINSING(E9:G11)}		
21							

Tip: accuracy of multiple roots is in general lower than the one of the singular roots. For this reason, sometimes, the SYSLINSING function cannot return any solution. In those cases, try to set the SYSLINSING parameter MaxError less then 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 tree row multiple each other's. So its rank is 1 and its solution generate 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' t correspond to the associate eigenvectors.
Lets' see the following example.

Example - Eigenvalues with multiplicity not corresponding to eigenvectors

Find all eigenvalues and associated eigenvectors of the following matrix

A =

1	2	1
2	0	-2
-1	2	3

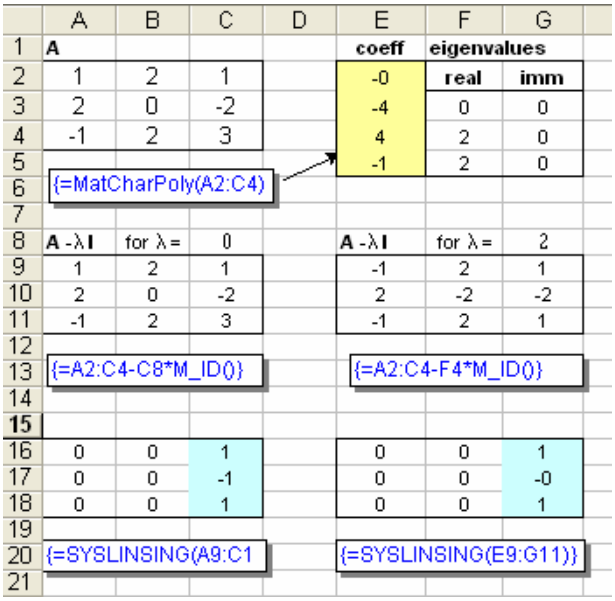
For the given matrix the characteristic polynomial is:

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

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.

Example - Complex Eigenvalues

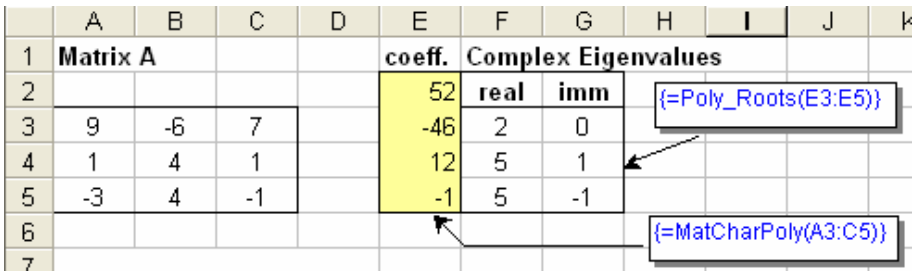
Sometime happens that not all roots of the characteristic polynomial are real. In that case the eigenvectors associate at complex eigenvalues are complex too.

Find all eigenvalues and associated eigenvectors of the following matrix

A =

9	-6	7
1	4	1
-3	4	-1

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



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:

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

The homogeneous linear system, becomes

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

Rearranging:

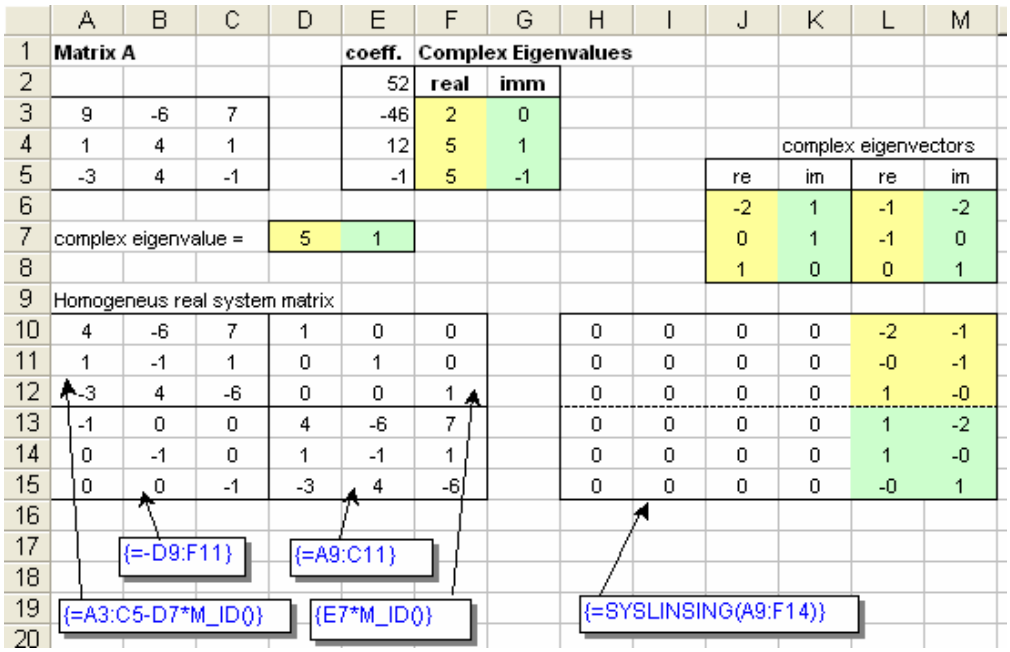
$$\left((A-\lambda_{re}I)x_{re} + \lambda_{im}I\,x_{im}\right) + j\left(-\lambda_{im}I\,x_{re} + (A-\lambda_{re}I)x_{im}\right) = 0$$

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.



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.

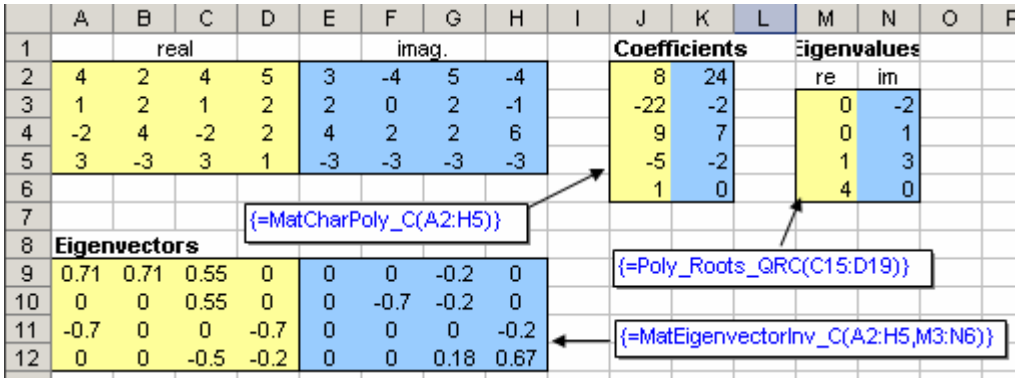
Example - Complex Matrix

Matrix.xla has several functions developed for solving the eigen-problem for complex matrices of moderate dimension.
Following the step-by-step method previous seen, we need the following functions:

- MatCharPoly_C - computes the complex coefficient of the characteristic polynomial
- Poly_Roots_QRC - computes the roots of a complex polynomial
- MatEigenvectorInv_C - computes the eigenvectors of a complex matrix

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

A possible arrangement is shown in the following worksheet.



Note that the given matrix has distinct eigenvalues: 2 real and 2 complex
This means that its eigenvector are distinct and we can use the inverse iteration algorithm for finding them. Note also that, in general, at a real eigenvalue does not correspond to a real eigenvector. Curiously the only real eigenvector corresponds to the imaginary eigenvalues $\lambda = -2j$

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

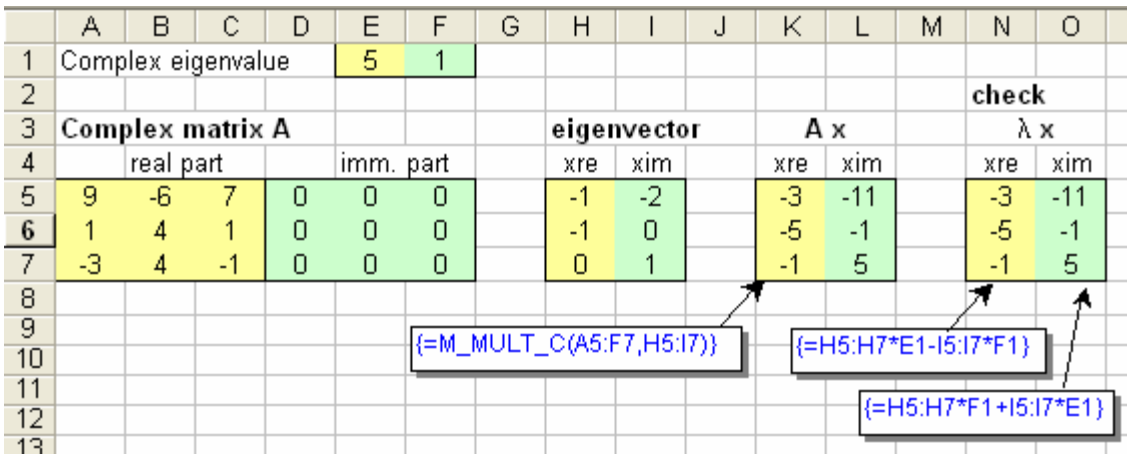
A =

9	-6	7
1	4	1
-3	4	-1

$\lambda = 5+j$

xre	xim
-1	-2
-1	0
0	1

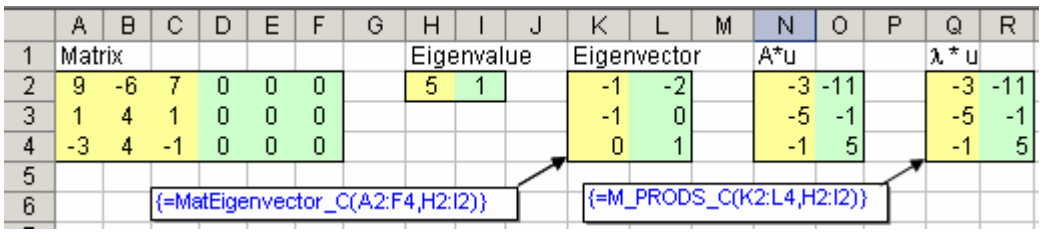
The test can be arrange as in the following worksheet



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 their eigenvalues; the first function works for real eigenvalues and the second one for complex. See the chapter "*Functions References*" of the Vol. 2 for details

In the following arrangement we have used the MatEigenvector_C for calculating the eigenvectors associated and the M_PRODS_C for obtaining the complex scalar product



Of course the final result is equivalent

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 e 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, remembering 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

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

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

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

To prove that eigenvalues are the same it is sufficient that the characteristic polynomials of A and B are equals. For computing the transformed matrix it 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 $B^{-1}AB$**		
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		similarity transformation				-31	2	0
10	10	3	0		eigenvalues unchanged				10	3	0
11	-1	5	0						-1	5	0

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.

$\mathbf{A}_1 = (\mathbf{P}_1)^{-1}\,\mathbf{A}\,(\mathbf{P}_1)$

$\mathbf{A}_2 = (\mathbf{P}_2)^{-1}\,\mathbf{A}_1\,(\mathbf{P}_2)$

$\mathbf{A}_3 = (\mathbf{P}_3)^{-1}\,\mathbf{A}_2\,(\mathbf{P}_3)$

.....

$\mathbf{A}_n = (\mathbf{P}_n)^{-1}\,\mathbf{A}_{n-1}\,(\mathbf{P}_n)$

$A_{n-1} \xrightarrow{n\to\infty} D$ Where D is diagonal

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

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.

Eigen-problems versus resolution methods

In the above paragraph we have spoke about the general method to resolve eigen-problems. 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 matrices. 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 that those methods are no more general but they are specialized for matrix class type. Very efficient algorithms exist for the symmetric matrix class, but the same algorithm cannot work, for example, with complex eigenvalues matrices. So, for a specific eigen-problem, 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

	Real eigensystem		**Complex eigensystem**	
Method	**Symmetric real matrix**	**Real matrix**	**Real matrix**	**Complex matrix**
Jacoby	**yes**	no	no	no
QR factorization	**yes**	**yes**	**yes**	**yes**
Power	**yes**	**yes**	no	no
Characteristic polynomial	**yes**	**yes**	**yes**	**yes**
Inverse iteration	**yes**	**yes**	**yes**	**yes**
Singular system	**yes**	**yes**	**yes**	**yes**

There are also special efficient algorithm for tridiagonal and Toeplitz, matrices.

Jacobi's transformation of symmetric matrix

For 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 – called 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:

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

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

A =

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

	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)}

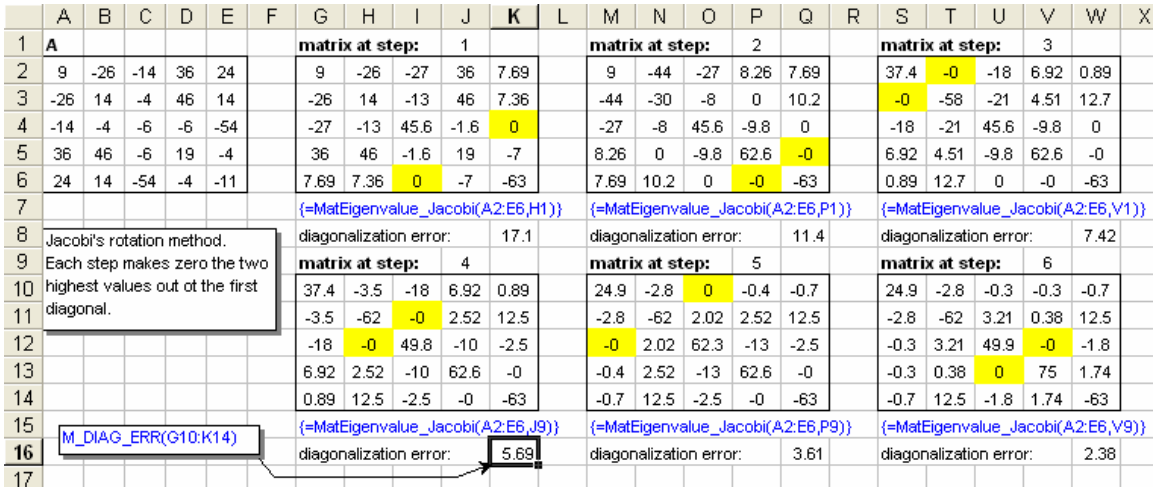
We can note the high clean of this method. Just plain and straight! By default, both functions use 100 iterations to reach this high accurate result. Sometime, for larger matrices, may be need to increase this limit or you have to accept less precision.

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	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)}						

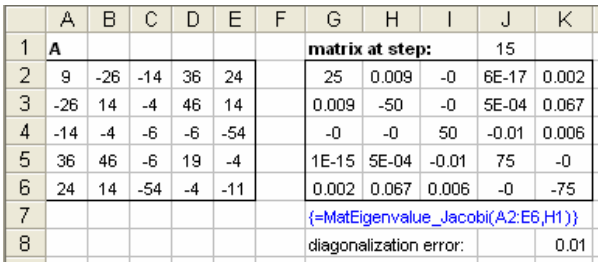
Example - Compute the first steps A1, A2, ... A6 of the Jacobi's algorithm and study the convergence of the previous example

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 lower and lower 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			**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						**matrix at step:**			4			**matrix at step:**			5			**matrix at step:**			6		
10	highest values out ot the first						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	diagonal.						-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 is always guaranteed. In our example, after 15 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

Orthogonal matrices

The eigenvectors matrix returned by the Jordan algorithm is "orthogonal" with each vector having norm 1; that is, an "orthonormal" matrix

Indicating the scalar product with the symbol $\bullet$, 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.

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

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

Pay attention that the second statement is not invertible. 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 the 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 pair of columns with the help of the function ProdScal. But it will tedious for large matrix. It is faster using 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)**						**hortonormalization 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																	

=ProdScal($A2:$A6,B2:B6)

=ProdScal($A2:$A6,A2:A6)

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

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**:

$\mathbf{A} = \mathbf{Q}\,\mathbf{R}$ where **Q** is orthonormal and **R** is triangular upper

This factorization is always possible; you can easy make such factorization in Matrix.xla with the function Mat_QR .

This method applies the following 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:

$$\mathbf{A} = \mathbf{Q}\,\mathbf{R} \Rightarrow \mathbf{A}_1 = \mathbf{R}\,\mathbf{Q}$$
$$\mathbf{A}_1 = \mathbf{Q}_1\,\mathbf{R}_1 \Rightarrow \mathbf{A}_2 = \mathbf{R}_1\,\mathbf{Q}_1$$
$$\mathbf{A}_2 = \mathbf{Q}_2\,\mathbf{R}_2 \Rightarrow \mathbf{A}_3 = \mathbf{R}_2\,\mathbf{Q}_2$$
$$\ldots\ldots\ldots\ldots\ldots \quad \ldots\ldots\ldots\ldots$$
$$\mathbf{A}_p = \mathbf{Q}_p\,\mathbf{R}_p \Rightarrow \mathbf{A}_{p+1} = \mathbf{R}_p\,\mathbf{Q}_p$$

If the eigenvalues are all distinct in modulo: $|\lambda_1| > |\lambda_2| > |\lambda_3| > \ldots > |\lambda_n|$ and **A** is symmetric; then the matrix $\mathbf{A}_p$ 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 Mat_QR_iter for performing the first 10 step of the QR algorithm. The convergence to the diagonal form is evident and becomes more close after 100 iterations.
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

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.

Example - calculate the first 10 and 100 steps of the QR algorithm for the following asymmetric 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 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 after 100 iterations.
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 (3x3) matrix has the eigenvalues $\lambda_1 = 9$, $\lambda_2 = -9$, $\lambda_3 = 18$. Apply the QR method we get.

5	-8	-10
-8	11	-2
-10	-2	2

	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 the Jacobi's algorithm finds 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 algorithm[5] being able to find complex an real eigenvalues of any real matrix.

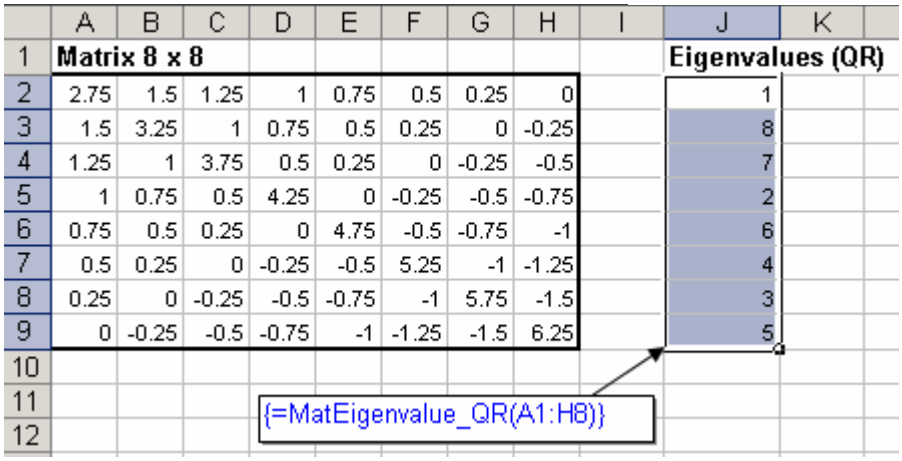
This task is performed by the function MatEigenvalue_QR of matrix.xla

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

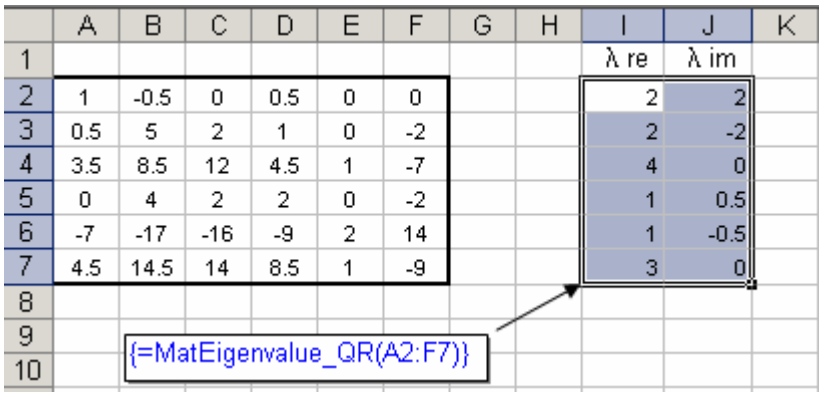


The function can also return complex eigenvalues. Let's see this example

This matrix has 2 real and 4 complex conjugate eigenvalues

3 , 4 , $2 \pm 2j$, $1 \pm 0.5j$

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



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

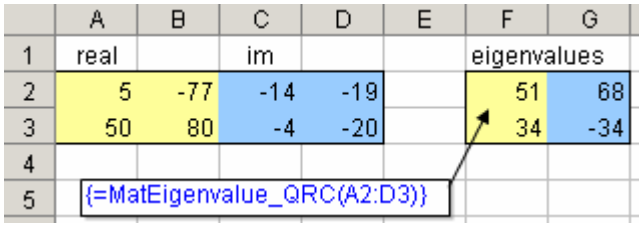
[5] Matrix.xla use the routine HQR and ELMHES derived from Fortran 77 EISPACK library

Complex eigenvalues of complex matrix with QR method

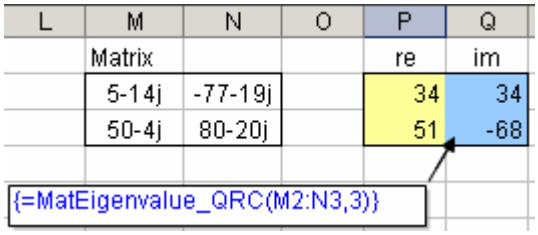
The function MatEigenvalue_QRC performs the complex implementation of the QR algorithm for a general complex matrix

Example. Find the eigenvalues of the following matrix

$$A=\begin{bmatrix}5-14j & -77-19j\\ 50-4j & 80-20j\end{bmatrix}=\begin{bmatrix}5 & -77\\ 50 & 80\end{bmatrix}+j\begin{bmatrix}-14 & -19\\ -4 & -20\end{bmatrix}$$



This function accept also the compact rectangular format "a+bj"



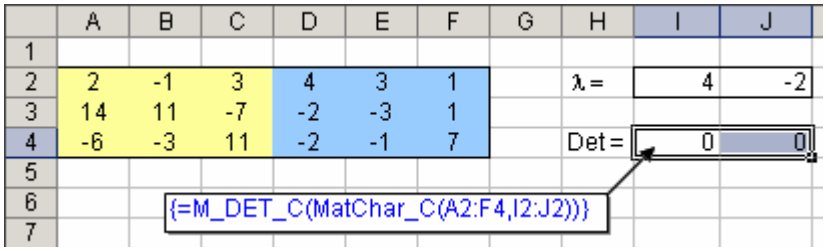
Note that the roots are always returned in split format

How to test complex eigenvalues

This test is conceptually very easy. We have only to compute the determinant of the characteristic matrix

$$A-\lambda I$$

For this task, comes useful the functions MatChar_C and M_DET_C



When the matrix size becomes higher, the round-off errors may mask the final result and the eigenvalue check may be not so easy and straight.
Just to give you an idea of the problem, let's see the following example

Example. Given the following (10 x 10) real matrix, prove that 1 is an eigenvalue

4569	-9128	-9136	-4556	-4484	9008	-9024	-4348	-9464	-9840
2004	-4003	-4016	-1996	-1960	3952	-3976	-1936	-4200	-4356
68	-136	-127	-76	-76	148	-128	-40	-124	-104
-556	1112	1112	569	552	-1112	1104	512	1144	1172
316	-632	-632	-316	-299	632	-624	-304	-648	-684
-284	568	568	284	284	-547	576	268	580	648
84	-168	-168	-84	-84	168	-143	-84	-176	-164
144	-288	-288	-144	-144	288	-288	-115	-296	-304
-72	144	144	72	72	-144	144	72	177	152
-36	72	72	36	36	-72	72	36	72	109

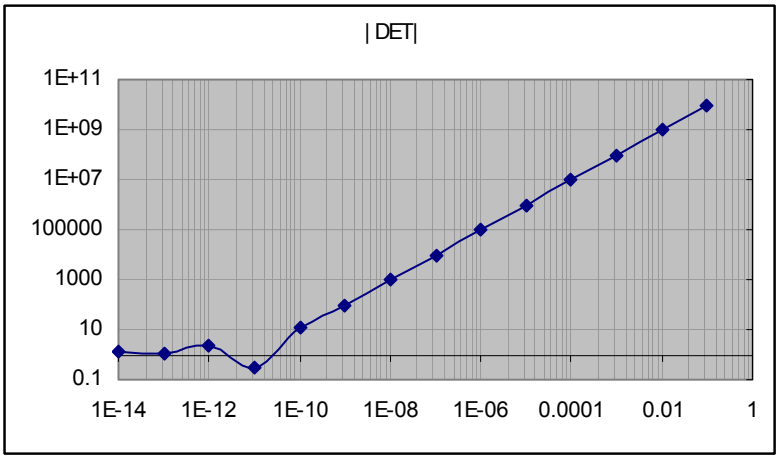
We can arrange a worksheet test like that

	A	B	C	D	E	F	G	H	I	J
1	**A**									
2	4569	-9128	-9136	-4556	-4484	9008	-9024	-4348	-9464	-9840
3	2004	-4003	-4016	-1996	-1960	3952	-3976	-1936	-4200	-4356
4	68	-136	-127	-76	-76	148	-128	-40	-124	-104
5	-556	1112	1112	569	552	-1112	1104	512	1144	1172
6	316	-632	-632	-316	-299	632	-624	-304	-648	-684
7	-284	568	568	284	284	-547	576	268	580	648
8	84	-168	-168	-84	-84	168	-143	-84	-176	-164
9	144	-288	-288	-144	-144	288	-288	-115	-296	-304
10	-72	144	144	72	72	-144	144	72	177	152
11	-36	72	72	36	36	-72	72	36	72	109
12										
13	λ	DET(**A** - λ **I**)		decimal		integer mode				
14	1			1.325		0				
15										
16	=MDETERM(MatChar(A2:J11,A14))									
17										
18		=M_DET(MatChar(A2:J11,A14),VERO)								

If we compute the determinant of the matrix **A** – λ **I** , we see, surprisingly, that it is much more than zero. What is wrong?
The fact is that we have computed the determinant with 15 digits floating point arithmetic and the round-off errors have masked the final true result
If we repeat the computation in integer mode, for example, by the function M_DET with the parameter IMODE = true, we get the correct result

Note that, in general, we can have decimal matrices or we can have decimal eigenvalues, so we can use the trick of the exact integer computing.
Perturbed eigenvalue method. In that case we should study the behavior of the determinant around the given eigenvalue. We can add random little increment ε to the eigenvalue, registering the correspondent absolute determinant. With the aid of the above functions, this process becomes quite handy. For example, giving incremental steps from 1E-14 to 0.1, we can easily get the following table and plot

λ	ε	| DET|
1	1E-14	1.322093
1	1E-13	1.1254013
1	1E-12	2.4366796
1	1E-11	0.316329
1	1E-10	11.011358
1	1E-09	84.180014
1	1E-08	956.28487
1	1E-07	9512.5687
1	1E-06	95119.334
1	0.00001	951267.55
1	0.0001	9512010.6
1	0.001	95059557
1	0.01	944559562
1	0.1	8.859E+09



How to find polynomial root with eigenvalues

In a previous example we have shown how to compute eigenvalues by 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 polynomial. The *companion matrix* is what we need. It can be easily built by hand or - even better - by the function **MatCmp**

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

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

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				{=MatCmpn(B2:B6)}					{=MatEigenvalue_QR(D2:G5)}			
9												

Eigenvalues are also the roots of the given polynomial.

Rootfinder with QR algorithm for real and complex polynomial

The QR method is so robust and efficient that it is implemented in the rootfinder function **Poly_Roots_QR** and **Poly_Roots_QRC** of Matrix.xla

Thanks to its efficiency, it is especially adapt for higher degree polynomial. Let' 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	2998800	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 left 11th degree polynomial all roots are real. The right 10th degree polynomial 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 QR algorithm 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 lack, characteristic of other rootfinder algorithms.

The function Poly_Roots_QRC works similar but for complex polynomials.
Example. find the roots of the following polynomial

$$x^7 - 2\cdot x^6 - 2\cdot x^5 + 6\cdot x^4 - 13\cdot x^3 + 4\cdot x^2 + 2\cdot x - 20 \; + \; i\cdot(2\cdot x^6 - 6\cdot x^4 + 8\cdot x^3 + 4\cdot x^2 - 8\cdot x)$$

	A	B	C	D	E	F	G
1		**coefficients**			**Roots**		
2	**degree**	**re**	**im**		**re**	**im**	
3	a0	-20	0		-2	0	
4	a1	2	-8		-1	0	
5	a2	4	4		0	1	
6	a3	-13	8		1	1	
7	a4	6	-6		1	-1	
8	a5	-2	0		1	-2	
9	a6	-2	2		2	-1	
10	a7	1	0				
11							
12		{=Poly_Roots_QRC(B3:C10)}					

Powers' method

Powers' method can found the dominant[6] 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 an asymmetric matrices
- It has an important didactic meaning

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

But shell 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:

$$v_1 = Av_0 \qquad \Rightarrow \qquad r = \frac{v_0^T v_1}{v_0^T v_0}$$

Iterating, we have:

$$v_2 = Av_1 \quad \Rightarrow \quad r = \frac{v_1^T v_2}{v_1^T v_1} \qquad \ldots\ldots\ldots \qquad v_{n+1} = Av_n \quad \Rightarrow \quad r = \frac{v_n^T v_{n+1}}{v_n^T v_n}$$

Under certain conditions, the ratio converges to the dominant eigenvalue for n >> 1 and the associate eigenvector can be obtained by the formulas:

$$\lim_{n\to\infty} r = \lambda_1 \qquad \Rightarrow \qquad \lim_{n\to\infty} v_n(\lambda_1)^{-n} = x_1$$

We shell see how it works in a practical case

Example - Analyze the convergence of the power's method for the following matrix

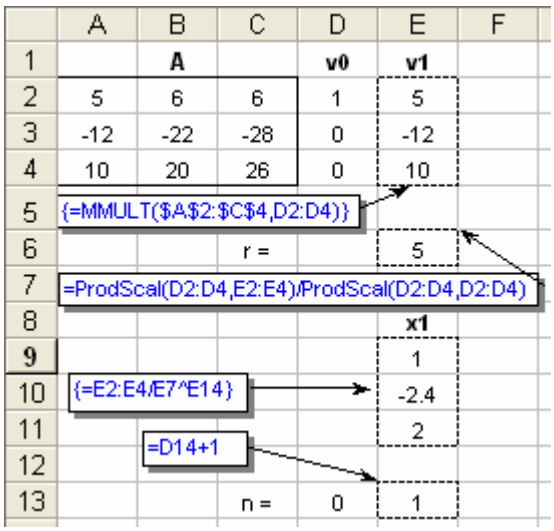
-1	2	-2
-2	-6	3
-2	-4	1

The matrix has three separate eigenvalues:
$\lambda_1 = -3$, $\lambda_2 = -2$, $\lambda_3 = -1$

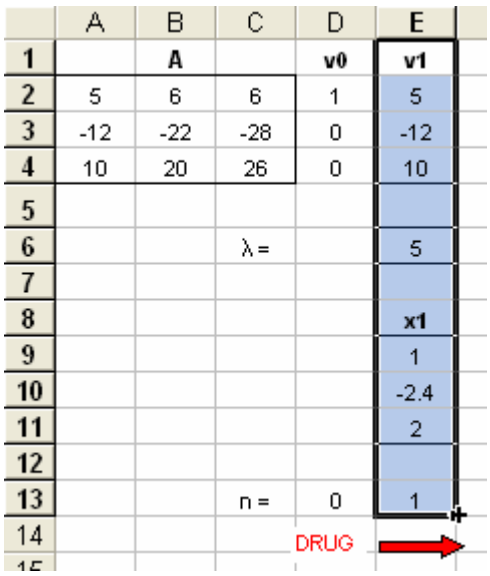
Let's see how to arrange the worksheet. First of all, insert the formulas as indicate to the left; than, select the appropriate range and drug to the right to iterate the formulas.

Assume the starting vector to be $\mathbf{v}_0 = (1, 0, 0)$

[6] The eigenvalue that has the highest absolute value is the "*dominant eigenvalue*"



Insert the formulas in the column E



Select the range E1:E13 and drug 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 the 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	⇒	-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

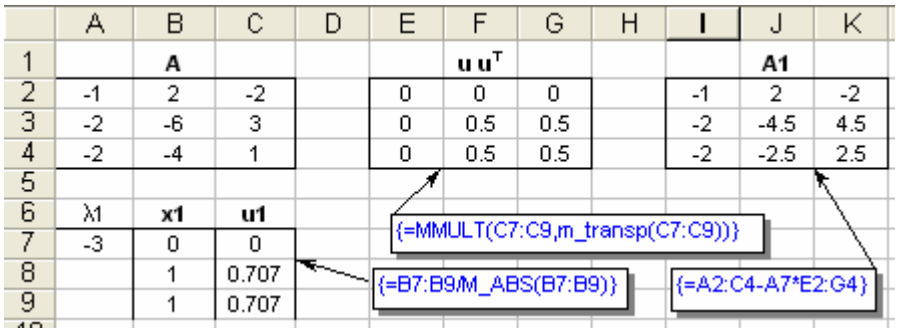
Finding non-dominant eigenvalues. Once the dominant eigenvalue λ_1 and its associate eigenvector $\mathbf{x}_1$ are found, we may want to continue to compute the eigenvalues remaining.

Compute the normalized of **x** and the new matrix $\mathbf{A}_1$:

$$\mathbf{u}_1 = \mathbf{x}_1 / |\mathbf{x}_1| \qquad \Rightarrow \qquad \mathbf{A}_1 = \mathbf{A} - \lambda_1 \mathbf{u}\mathbf{u}^T$$

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 **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



The matrix $\mathbf{A}_1$ is the new reduced matrix. It should have all the eigenvalues of the original matrix **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 x = (-2, 0.25, -0.75) is slow but evident. After 25 steps the error is less than about 1E-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, round-off errors will be introduced in the next iterate steps; 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 the power method

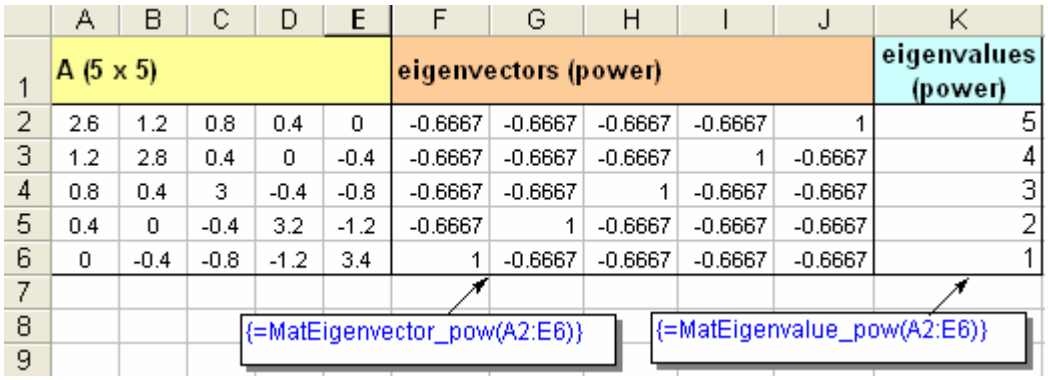
In Matrix.xla the power method is implemented by two main functions:

- 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



The function MatEigenvector_pow has a second parameter: *Norm*. If TRUE, the function returns normalized eigenvectors (default FALSE).

	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 is there any 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
The 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)}		

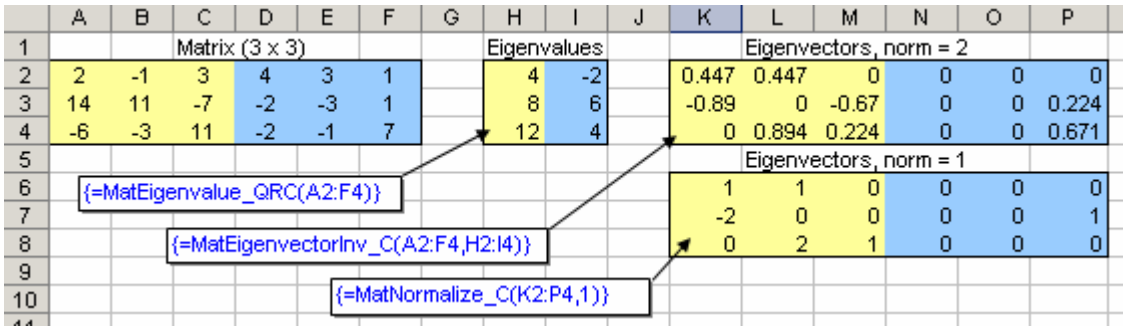
Complex Eigensystems

In Matrix.xla the eigen-problem of a general complex matrix is solved with the aid of the following main functions:

- MatEigenvalues_QRC returns all the eigenvalues by the complex QR algorithm
- MatEigenvectorInv_C returns all distinct eigenvectors by the inverse iteration
- MatEigenvector_C returns the eigenvectors of associated eigenvalue

Example 1. Find eigenvalues and eigenvectors of the following complex matrix

2+4j	-1+3j	3+j
14-2j	11-3j	-7+j
-6-2j	-3-j	11+7j

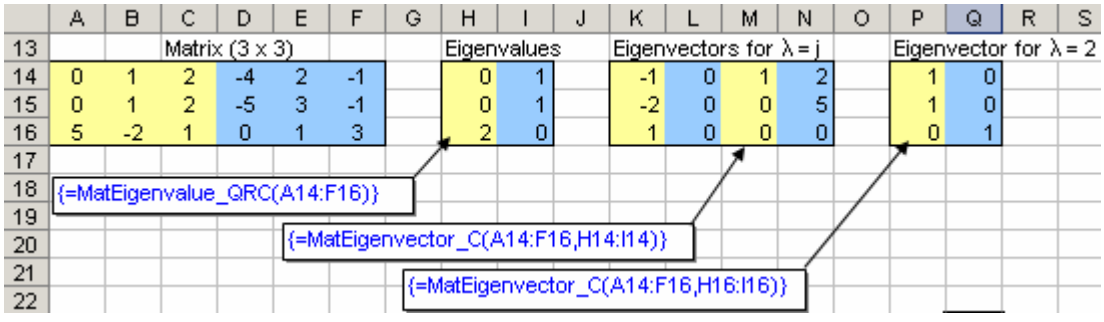


In this case the eigenvalues are all distinct, therefore we can quickly obtain the associated eigenvectors by the inverse iteration algorithm

Note that the eigenvectors returned by the function MatEigenvectorInv_C have always unitary module (norm = 2). For changing the normalization type we can use the function MatNormalize_C.

When the eigenvalues are not all distinct we cannot use the inverse iteration but the singular system method performed by the MatEigenvector_C

Example. The following matrix has only 2 distinct eigenvalues: 2, j



Note that the eigenvalue λ = j with multiplicity = 2 has associated 2 eigenvectors returned in a (3 x 4) array. The eigenvalue λ = 2 has associated one eigenvector returned in the last (3 x 2) array

How to validate an eigen-system

Example - Check the real eigen-system of the previous example

In order to test an eigenvector matrix **U** of a given matrix **A,** we can use the definition

$$\mathbf{A\,U} = (\lambda_1 \mathbf{u}_1, \lambda_2 \mathbf{u}_2, \ldots \lambda_6 \mathbf{u}_6)$$

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 shell begin with eliminating round off error by the function MatMopUp

	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												

{=MatMopUp(A2:F7,1E-13)}

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												

{=G2:G7/G9}

{=H2:H7/H9}

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)}						{=G3:G8*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 the two matrices are identical, the eigensystem (eigenvectors + eigenvalues) is correct.

How to generate a random symmetric matrix with given eigenvalues

Many time, for testing algorithms, we need a symmetric matrix with known eigenvalues
For building this test matrix, the following simple method can be useful

- First, we generate a random (n x 1) vectors, **v**
- Then we generate the *Householder* matrix **H** with the vector **v**
- We create a diagonal (n x n) matrix **D** with the eigenvalues that we want to obtain.
- Finally we make a *Similarity Transformation* of matrix **D** by the matrix **W**.

The result is a symmetric matrix with the given eigenvalues.

Example: Suppose we want a (3 x 3) random symmetric matrix with eigenvalues = (1, 2, 4)
We chose a random vector **v**, like for example:

$$v = \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}$$

We create the associate *Householder* matrix **H**

$$H = I - 2\frac{v \cdot v^T}{\|v\|^2} = \begin{bmatrix} -1/3 & -2/3 & 2/3 \\ -2/3 & 2/3 & 1/3 \\ 2/3 & 1/3 & 2/3 \end{bmatrix}$$

We create the diagonal matrix **D**

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

We make the *Similarity Transform* of **D** by **H**

$$A = H^{-1}A \cdot H = \begin{bmatrix} 25/9 & 2/9 & 10/9 \\ 2/9 & 16/9 & 8/9 \\ 10/9 & 8/9 & 22/9 \end{bmatrix}$$

Note that, in this case, the inverse of **H** is the same of **H**.
The result matrix **A** has the wanted eigenvalues = (1, 2, 4)

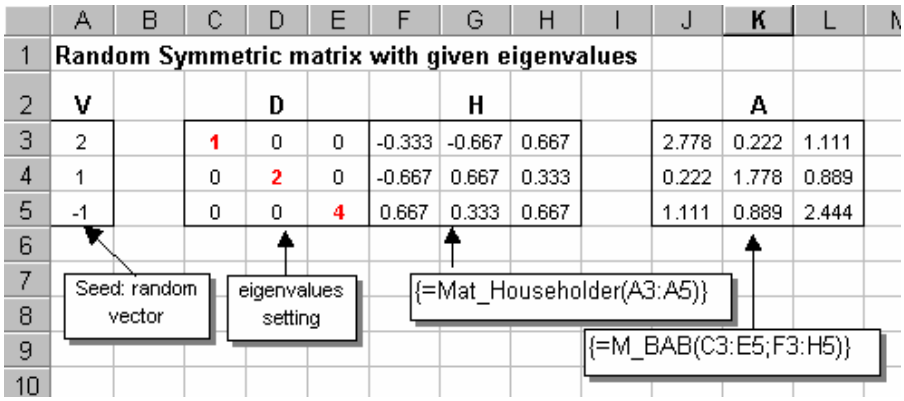
If we want to avoid fractional numbers we can multiply the matrix **A** for 9 and we get a new symmetric matrix **B**

$$B = 9 \cdot A = \begin{bmatrix} 25 & 2 & 10 \\ 2 & 16 & 8 \\ 10 & 8 & 22 \end{bmatrix}$$

The eigenvalues of **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.



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_3U 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		
0	5	-0.5
-2	5	-1
-1	5	-1
-1	5	-2
-1	4	-0.5
-1	3	-1
-5	-5	-1
-0.5	6	-1
-1	5	-1
-0.5	9	-1
-1	2	-1
-1	-9	-1
-1	10	-1
-2	-4	-1
-1	-8	0

15 x 15 tridiagona matrix in standard form															
5	-0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-2	5	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	-1	5	-1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	-1	5	-2	0	0	0	0	0	0	0	0	0	0	0
0	0	0	-1	4	-0.5	0	0	0	0	0	0	0	0	0	0
0	0	0	0	-1	3	-1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	-5	-5	-1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	-0.5	6	-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	-1	5	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	-0.5	9	-1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	-1	2	-1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	-1	-9	-1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	-1	10	-1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	-2	-4	-1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-8	0

In the compact form we store only the diagonal and sub-diagonals values
For tridiagonal matrices there are several useful lemmas that help us to discover the kind of eigenvalues

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 example we have computed 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 eigenvectors returned by MatEigenvector3 are not normalized. Use for this task the MatNormalize function.

Eigenvalues of tridiagonal Toeplitz matrix (tridiagonal uniform)

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 x n dimension, eigenvalues are:

$$\lambda_k = a + 2b \cdot \cos\left(\frac{k\pi}{n+1}\right)$$

where *k* = 1, 2… 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 $a-2b < \lambda_\kappa < a+2b$

Also the eigenvectors matrix can be written in a closed compact form.

$$U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ u_{21} & u_{22} & \dots & u_{1n} \\ \dots & \dots & \dots & \dots \\ u_{n1} & u_{n2} & \dots & u_{nn} \end{bmatrix}$$

If the symmetric matrix has n x 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… n , k = 1, 2… n

The unsymmetrical tridiagonal uniform case can be led back to the above one.
We distingue 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 x 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… 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 x 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… n

All roots lie on the segment:

$$re(\lambda_k) = a \qquad -2\sqrt{-bc} < im(\lambda_k) < 2\sqrt{-bc}$$

Eigenvectors can be computed by the following iterative algorithm

$$x_k = \lambda_k - a$$

$$u_{ik} = \frac{1}{b}\left(x_k \cdot u_{(i-1)k} - c \cdot u_{(i-2)k}\right)$$

Where : *k* = 1, 2… n , *i* = 1, 2… n

$$u_{1k} = 1 \ , \ u_{2k} = \frac{1}{b}x_k$$

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 values of the sub-diagonals lower and upper 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, …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 sub-diagonal 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, …7 and where a = 10, b = 2, c = –1, n= 7

Giving the following 7 eigenvalues.

	real	im
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 sub-diagonal 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, … 8 and where a = 1, b = 1, c = –1, n= 8

Giving the following 8 eigenvalues.

	real	im
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, …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

$$u_{ij} = \sin\left(i \cdot j \frac{\pi}{n+1}\right) \qquad U = \begin{bmatrix} \sin(\alpha) & \sin(2\alpha) & \ldots & \sin(8\alpha) \\ \sin(2\alpha) & \sin(4\alpha) & \ldots & \sin(16\alpha) \\ \ldots & \ldots & \ldots & \ldots \\ \sin(8\alpha) & \sin(16\alpha) & \ldots & \sin(64\alpha) \end{bmatrix}$$

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. Eigen-problems can lead to very large different solutions and, we must say that, they represent 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 eigen-system 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 shell 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			18	
3	-8	11	-2		81	-9	0			-3.332	
4	-10	-2	2		18	9	0			4.061	
5					-1	18	0				
6											
7					Eigenvalues (Jacobi)				Eigenvalues (QR)		
8					18	0	1E-31		18	-4E-16	4E-16
9					9E-16	9	8E-22		1E-43	8E-13	-9
10					3E-16	7E-16	-9		-4E-44	-9	-8E-13
11											
12					right !				wrong !		
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 the Numeric Calculus!

Generalized Eigen-problem

Given the following matrix equation

$$\mathbf{A}\,\mathbf{x} = \lambda\,\mathbf{B}\,\mathbf{x} \qquad (1)$$

Where **A** and **B** are both matrices symmetric and **B** is positive definite, is said a generalized eigen-problem.

Equivalent non symmetric problem

This problem is equivalent to:

$$(\mathbf{B}^{-1}\mathbf{A})\,\mathbf{x} = \lambda\,\mathbf{x} \quad \Rightarrow \quad \mathbf{C}\,\mathbf{x} = \lambda\,\mathbf{x} \qquad (2)$$

In generally **C** is not symmetric even **A** and **B** they are.

Example: transform a generalized eigen-problem into a standard eigen-problem, where the matrices **A** and **B** are

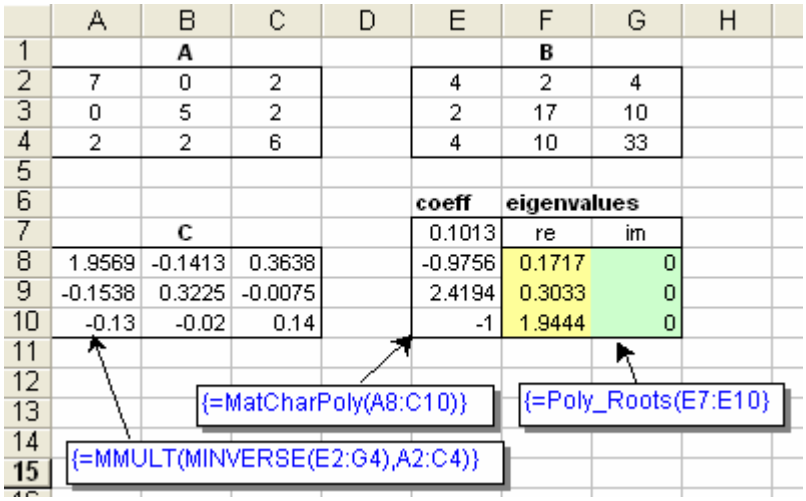
A

7	0	2
0	5	2
2	2	6

B

4	2	4
2	17	10
4	10	33

In the following worksheet we have calculate the matrix $\mathbf{C} = \mathbf{B}^{-1}\,\mathbf{A}$



As we can see, the matrix **C** is not symmetric even if **A** and **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:

$$\lambda_1 = 0.1717 \qquad \lambda_2 = 0.3033 \qquad \lambda_3 = 1.9444$$

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

$$\mathbf{A}\,\mathbf{x} = \lambda\,\mathbf{B}\,\mathbf{x} \qquad (1)$$

Where **A** and **B** are both symmetric matrices and **B** is positive definite. It is called "generalized eigen-problem".

In the previous paragraph we have learnt how to transform this problem into a standard eigen-problem setting $\mathbf{C} = \mathbf{B}^{-1}\mathbf{A}$. But **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

$$\mathbf{B} = \mathbf{L}\,\mathbf{L}^{T} \qquad (2)$$

Where **L** is a triangular matrix.

Substituting (2) into (1) and multiplying the equation by $\mathbf{L}^{-1}$, we get:

$$\mathbf{L}^{-1}\,\mathbf{A}\,\mathbf{x} = \lambda\,(\mathbf{L}^{-1}\mathbf{L})\,\mathbf{L}^{T}\,\mathbf{x} \qquad \Rightarrow \qquad \mathbf{L}^{-1}\,\mathbf{A}\,\mathbf{x} = \lambda\,\mathbf{L}^{T}\,\mathbf{x}$$

And, because $\mathbf{I} = (\mathbf{L}^{T})^{-1}\,\mathbf{L}^{T} = (\mathbf{L}^{-1})^{T}\,\mathbf{L}^{T}$, we can write:

$$\mathbf{L}^{-1}\,\mathbf{A}\,(\mathbf{L}^{-1})^{T}\,\mathbf{L}^{T}\,\mathbf{x} = \lambda\,\mathbf{L}^{T}\,\mathbf{x} \qquad \Rightarrow \qquad \mathbf{L}^{-1}\,\mathbf{A}\,\mathbf{x} = \lambda\,\mathbf{L}^{T}\,\mathbf{x}$$

Set the auxiliary matrix: $\mathbf{W} = \mathbf{L}^{-1}$ and the auxiliary vector $\mathbf{d} = \mathbf{L}^{T}\,\mathbf{x}$ we have

$$\mathbf{W}\,\mathbf{A}\,\mathbf{W}^{T}\,\mathbf{d} = \lambda\,\mathbf{d} \qquad \Rightarrow \qquad \mathbf{D}\,\mathbf{d} = \lambda\,\mathbf{d} \qquad (3)$$

The equation (3) is the new eigen-problem 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 **x** can be obtained from eigenvectors **d** by the following formula:

$$\mathbf{d} = \mathbf{L}^{T}\,\mathbf{x} \qquad \Rightarrow \quad \mathbf{x} = (\mathbf{L}^{T})^{-1}\,\mathbf{d} \qquad \Rightarrow \qquad \mathbf{x} = (\mathbf{L}^{-1})^{T}\,\mathbf{d}$$

$$\mathbf{x} = \mathbf{W}^{T}\,\mathbf{d}$$

That is, eigenvectors of (1) can be obtained by multiplying eigenvectors of (3) for the auxiliary matrix **W**.

In Matrix.xla there is everything you need to solve generalized eigen-problem: 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 eigen-problem, assuming the matrices **A** and **B** of the previous example

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**				**eigenversor 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 = λ 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 = λ d
eigenvectors
x = W^T d
normalized eigenvectors
u = x / |x|

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.

$$B = \begin{bmatrix} b_{11} & 0 & 0 \\ 0 & b_{22} & 0 \\ 0 & 0 & b_{33} \end{bmatrix} \qquad L = \begin{bmatrix} \sqrt{b_{11}} & 0 & 0 \\ 0 & \sqrt{b_{22}} & 0 \\ 0 & 0 & \sqrt{b_{33}} \end{bmatrix} \qquad L^{-1} = \begin{bmatrix} \frac{1}{\sqrt{b_{11}}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{b_{22}}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{b_{33}}} \end{bmatrix}$$

Let's see a practical example.

Example - How to get mode shapes and frequencies for a multi-degree of freedom structure [7]

Example 1 - Our problem is an example of the "generalized" eigenproblem:

$$\mathbf{k}\,\phi = \omega^2\,\mathbf{m}\,\phi \qquad (1)$$

Where **k** and **m** are both symmetric positive definite matrices. In the specific case they were:

Stiffness matrix **k**:

600	-600	0
-600	1800	-1200
0	-1200	3000

Mass matrix **m**:

1	0	0
0	1.5	0
0	0	2

This problem is equivalent to a "standard" eigenproblem:

$$(\mathbf{m}^{-1}\cdot\mathbf{k})\,\phi = \omega^2\,\phi \qquad \Rightarrow \qquad \mathbf{C}\,\phi = \omega^2\,\phi$$

The problem is that **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

$$\mathbf{m} = \mathbf{L}\,\mathbf{L}^{\mathbf{T}}$$

Where **L** is a triangular matrix. In a case like ours where **m** is diagonal the **L** matrix is also diagonal with each term of **L** being the square root of the corresponding term in **m**. Define new matrix **W** as:

$$\mathbf{W} = \mathbf{L}^{-1}$$

Multiplying equation (1) by **W**, one gets:

$$\mathbf{W}\,\mathbf{k}\,\mathbf{W}^{\mathbf{T}}\,(\mathbf{L}^{\mathbf{T}}\,\phi) = \omega^2\,(\mathbf{L}^{\mathbf{T}}\,\phi)$$

Or, more concisely

$$\mathbf{D}\,\mathbf{v} = \omega^2\,\mathbf{v} \qquad (2)$$

Where

$$\mathbf{D} = \mathbf{W}\,\mathbf{k}\,\mathbf{W}^{\mathbf{T}} \qquad (3)$$

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):

---

[7] 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.

$$\phi = (L^T)^{-1} v = W v \qquad (4)$$

So here's what you do:
Starting with **k** and **m**, make **L** ; 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								

Example 2 - Seven inertia torsion system

This example[8] shows how to solve a more larger torsion system with a good accuracy. Assume to have the following torsion system equation

$$\mathbf{K}\,\phi = \omega^2\,\mathbf{M}\,\phi \qquad (1)$$

Where the matrices [K] and [M] are

[8] Thanks to Anthony Garcia

[M]=

115.2	0	0	0	0	0	0
0	15.8	0	0	0	0	0
0	0	1.35	0	0	0	0
0	0	0	1.35	0	0	0
0	0	0	0	1.35	0	0
0	0	0	0	0	1.35	0
0	0	0	0	0	0	9.21

[K]=

9400000	-9400000	0	0	0	0	0
-9400000	24400000	-15000000	0	0	0	0
0	-15000000	49000000	-34000000	0	0	0
0	0	-34000000	68000000	-34000000	0	0
0	0	0	-34000000	68000000	-34000000	0
0	0	0	0	-34000000	106000000	-72000000
0	0	0	0	0	-72000000	72000000

Tip. Scaling the given matrix for a suitable factor may increase the computing accuracy of several orders. In this case we divide the [K[matrix for a factor 10^6. The eigenvalues are proportionally scaled by the same factor. In fact, multiplying both terms of the equation (1) for the same scaling factor, we have:

$$10^{-6}\,\mathbf{K}\,\phi = 10^{-6}\,\omega^2\,\mathbf{M}\,\phi$$

$$\mathbf{K'}\,\phi = \lambda\,\mathbf{M}\,\phi$$

where $\mathbf{K'} = 10^{-6}\,\mathbf{K}$ and $\omega^2 = 10^6\,\lambda$

[K'] =

9.4	-9.4	0	0	0	0	0
-9.4	24.4	-15	0	0	0	0
0	-15	49	-34	0	0	0
0	0	-34	68	-34	0	0
0	0	0	-34	68	-34	0
0	0	0	0	-34	106	-72
0	0	0	0	0	-72	72

the Cholesky factorization of **M**; it can be easily computed because it is a diagonal matrix

$$\mathbf{L} = [\,(m_{11})^{1/2}\,,\,(m_{22})^{1/2}\,,\,....\,(m_{77})^{1/2}\,]$$

[L]=Choleski Decomposition of [J]=(sqrt(Jii))

10.7331	0	0	0	0	0	0
0	3.97492	0	0	0	0	0
0	0	1.1619	0	0	0	0
0	0	0	1.1619	0	0	0
0	0	0	0	1.1619	0	0
0	0	0	0	0	1.1619	0
0	0	0	0	0	0	3.0348

The auxiliary matrix is the inverse of L matrix; but also in this case, it is very easy to compute the inverse, being

$$\mathbf{W} = \mathbf{L}^{-1} = [\,1/L_{11}\,,\,1/L_{22}\,,\,....\,1/L_{22}\,]$$

$[W]=[L]^{-1}$

0.09317	0	0	0	0	0	0
0	0.25158	0	0	0	0	0
0	0	0.86066	0	0	0	0
0	0	0	0.86066	0	0	0
0	0	0	0	0.86066	0	0
0	0	0	0	0	0.86066	0
0	0	0	0	0	0	0.32951

Now we compute the matrix $[D]=[W][K'][W]^T$ by the function M_PROD
Note that, being [W] diagonal, is $[W]^T = [W]$

$[D]=[W][K'][W]^T$

0.0816	-0.2203	0	0	0	0	0
-0.2203	1.5443	-3.2478	0	0	0	0
0	-3.2478	36.2963	-25.185	0	0	0
0	0	-25.185	50.3704	-25.185	0	0
0	0	0	-25.185	50.3704	-25.185	0
0	0	0	0	-25.185	78.5185	-20.419
0	0	0	0	0	-20.419	7.81759

Applying the Jacoby algorithm or, even better, the QL algorithm, to the symmetric tridiagonal matrix [D], we get all its real eigenvalues. Multiplying them for the factor 10^6, we have finally the eigenvalues of the given torsion system

Eigenvalues by Jacobi method

0	0	0	0	0	0	0
0	1.29553	0	0	0	0	0
0	0	10.2229	0	0	0	0
0	0	0	74.4626	0	0	0
0	0	0	0	37.7008	0	0
0	0	0	0	0	101.039	0
0	0	0	0	0	0	0.27776

Eigenvalues

0
1295533.38
10222899.63
74462588.17
37700787.71
101039475.2
277762.0792

Eigenvectors [Vd] of D by Jacobi Method

0.8895	-0.1516	0.0045	-0.0001	-0.0003	0.0000	-0.4311
0.3294	0.8351	-0.2073	0.0204	0.0582	0.0042	0.3838
0.0963	0.0742	0.5536	-0.4581	-0.6483	-0.1288	0.1789
0.0963	-0.0045	0.5998	0.6916	0.0286	0.3305	0.2064
0.0963	-0.0830	0.4026	-0.2035	0.6627	-0.5362	0.2316
0.0963	-0.1573	0.0420	-0.4969	0.3048	0.7482	0.2542
0.2515	-0.4924	-0.3562	0.1522	-0.2082	-0.1639	0.6885

The eigenvectors of [D] may be computed by the Jacoby algorithm or by the inverse iteration
Here we have used the function MatEigenvector_Jacobi

Eigenvectors Va of A obtained by [Va] = [W][Vd]

0.0829	-0.0141	0.0004	0.0000	0.0000	0.0000	-0.0402
0.0829	0.2101	-0.0521	0.0051	0.0147	0.0011	0.0966
0.0829	0.0639	0.4764	-0.3943	-0.5580	-0.1108	0.1540
0.0829	-0.0039	0.5163	0.5952	0.0247	0.2845	0.1776
0.0829	-0.0715	0.3465	-0.1751	0.5704	-0.4615	0.1993
0.0829	-0.1354	0.0361	-0.4277	0.2623	0.6439	0.2188
0.0829	-0.1623	-0.1174	0.0502	-0.0686	-0.0540	0.2269

Multiply the [Vd] matrix for the auxiliary [W] matrix we have the eigenvectors of the given system

Normalized Eigenvectors [Va]

0.3780	-0.0451	0.0005	0.0000	0.0000	0.0000	-0.0887
0.3780	0.6704	-0.0656	0.0060	0.0174	0.0012	0.2131
0.3780	0.2039	0.5996	-0.4628	-0.6617	-0.1303	0.3399
0.3780	-0.0124	0.6497	0.6986	0.0292	0.3344	0.3921
0.3780	-0.2281	0.4361	-0.2055	0.6764	-0.5424	0.4399
0.3780	-0.4320	0.0455	-0.5020	0.3111	0.7569	0.4829
0.3780	-0.5178	-0.1477	0.0589	-0.0814	-0.0635	0.5007

That can be normalized as we like by the function MatNormalize

Chapter

4

Linear regression

Recalls

Generally, the multivariate linear regression function is:

$$y = a_0 + a_1 x_1 + a_2 x_2 + ... a_m x_m$$

Where:

$$[a_0, a_1, a_2 ... a_m]$$

Is the coefficients vector of regression, which can be found by the following procedure
Make the following variables substitution:

$$X_i = x_i - \bar{x} \qquad \text{for i} = 1..\text{m}$$

$$Y = y - \bar{y}$$

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:

$$\begin{bmatrix} X_{11} & ... & X_{1m} \\ ... & ... & ... \\ X_{nm} & ... & X_{nm} \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ ... \\ a_m \end{bmatrix} = \begin{bmatrix} Y_1 \\ ... \\ Y_m \end{bmatrix}$$

That is, in compact form:

$$[X] \cdot a = Y$$

We solve this singular system with SVD method, obtaining the following 3 matrices

$$[X] = [U] \cdot [D] \cdot [V]^T = [U] \cdot \begin{bmatrix} d_{11} & 0 & 0 \\ ... & ... & ... \\ 0 & 0 & d_{mm} \end{bmatrix} \cdot [V]^T$$

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

$$a_0 = \overline{Y} - \sum_{i=1}^{m} a_i \overline{X}_i$$

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

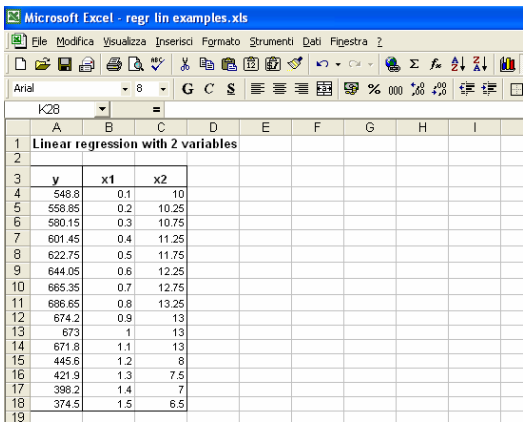
The function (model) is

$$f(x_1, x_2) = a_0 + a_1 x_1 + a_2 x_2$$

We use the linear regression to find the coefficients a_0, a_1, a_2

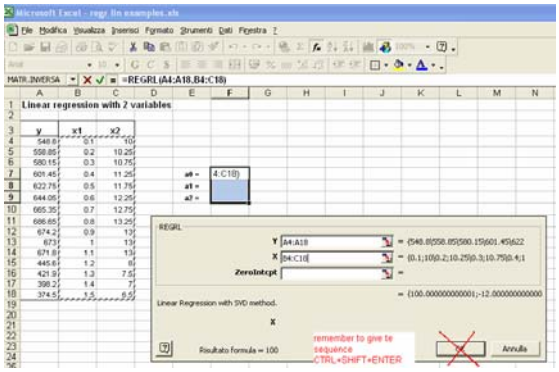
Arrange the table data as in the following worksheet

Select the range where you want to paste the coefficients, For example the range F7:F9



	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 f_x and insert the **REGRL** function



Remember: give the CTRL+SHIFT+ENTER sequence to enter this function

F7 = {=REGRL(A4:A18,B4:C18)}

	A	B	C	D	E	F
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 =	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			

The solution is a_0= 100, a_1 = -12 , a_2= 45, giving the regression function:

$$f(x_1, x_2) = 100 - 12\,x_1 + 45\,x_2$$

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

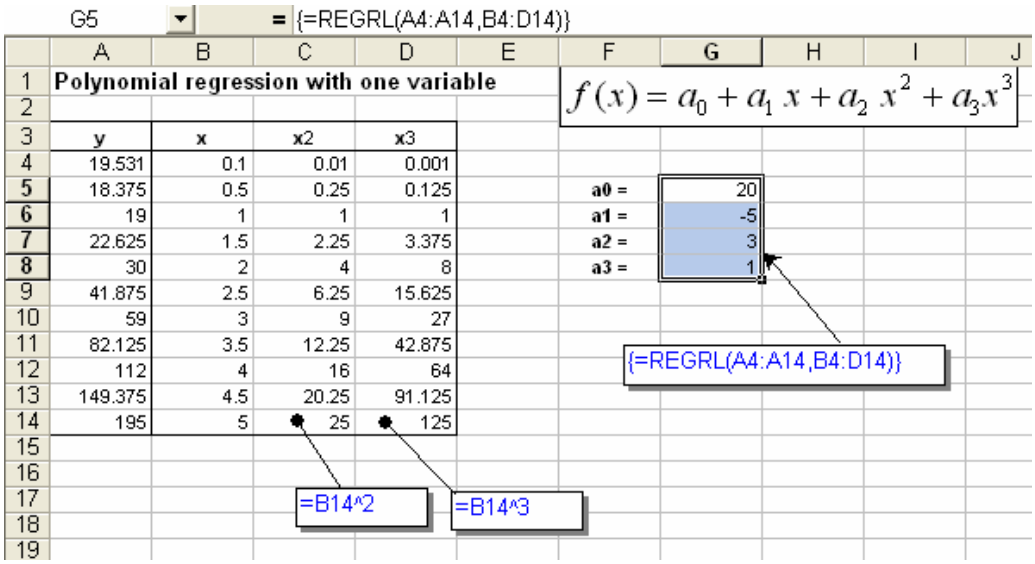
y	x
19.531	0.1
18.375	0.5
19	1
22.625	1.5
30	2
41.875	2.5
59	3
82.125	3.5
112	4
149.375	4.5
195	5

The function (model) is

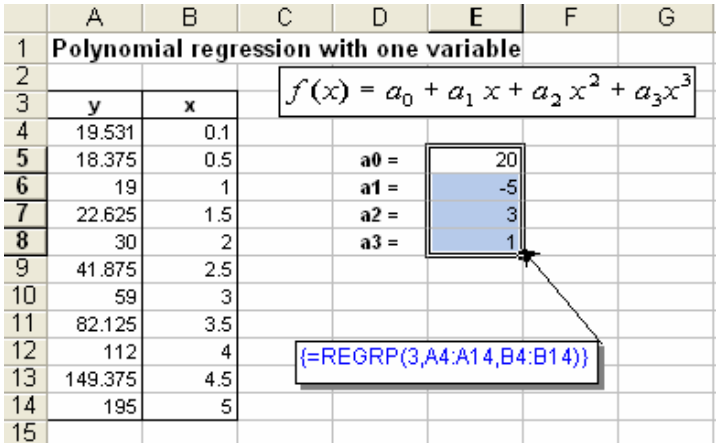
$$f(x) = a_0 + a_1\,x + a_2\,x^2 + a_3 x^3$$

Copy the above table in a worksheet and add two other column $\mathbf{x}^2$ and $\mathbf{x}^3$

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



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



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 2nd 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							

{=REGRL(A4:A10,B4:F10)}

First of all, we easily calculate the 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

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

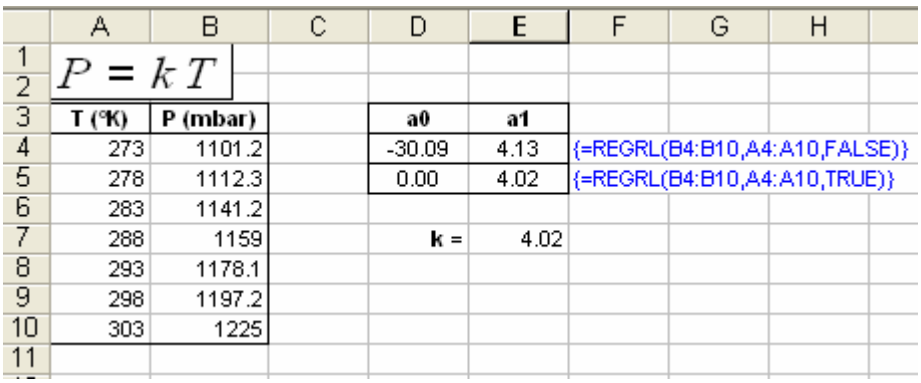
Example: A test of a gas-barometer has given the following experimental result. Find the barometer constant k = P / T (mbar / °K)

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



	A	B	C	D	E	F	G	H
1	$P = kT$							
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 (a0 ≅ -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). We might then ask ourselves what mathematically formula best describes the relationships (if any). As seen in the previous section, the technique used is the least square linear regression.
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

Same 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 the advantages that they can be treated with the known linear regression formulas. This technique, however, is only possible 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 that we have 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). We calculate the sum of squared residual.

$$ssr = \sum \left(y_i - f(x_i, a_1, a_2)\right)^2$$

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 $y = y_0\, e^{k\,x}$

Taking the logarithm of both sides, we have

$$\ln(y) = \ln\left(y_0\, e^{k\,x}\right) \quad \Rightarrow \quad \ln(y) = \ln(y_0) + kx$$

Setting the new variable z = ln(y), the equation became linear in x and z, with the parameters z_0 and k.

$$z = \ln(y) \quad \Rightarrow \quad z = z_0 + kx$$

$$y_0 = \exp(z_0)$$

The original parameter y_0 can be found by the simple formula

Reassuming, to calculate an exponential regression of data set (x_i, y_i), we have to made:

1. Convert the 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

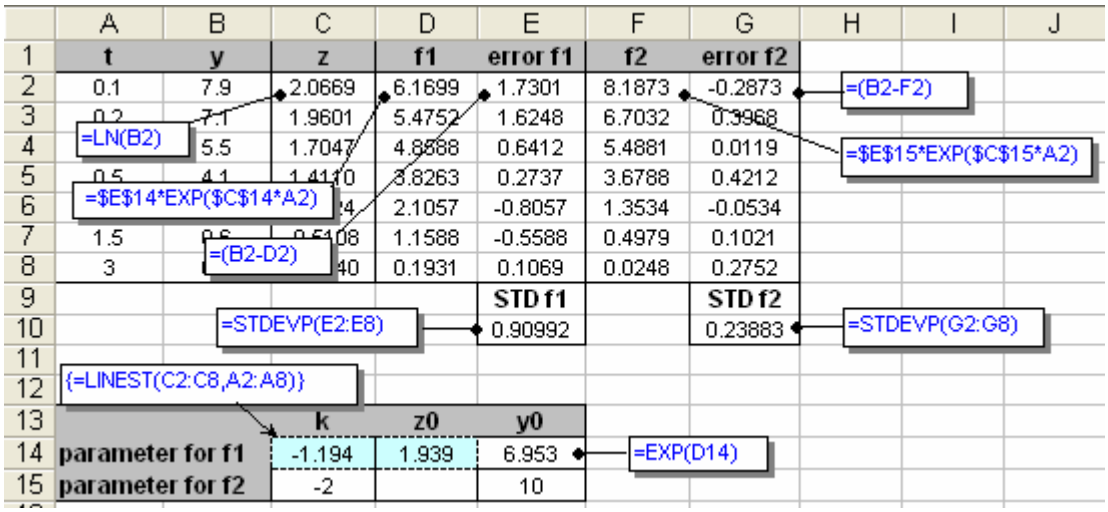
t	y
0.1	7.9
0.2	7.1
0.3	5.5
0.5	4.1
1	1.3
1.5	0.6
3	0.3

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 "z_0". With the EXP function we have computed the "y_0" parameter
Then we have calculated the estimated values of the column "f_1", and the value of residues "error f_1". At the bottom, we have computed the standard deviation of the residues for estimating the standard error.



As we see the exponential parameters found by linear regression are:

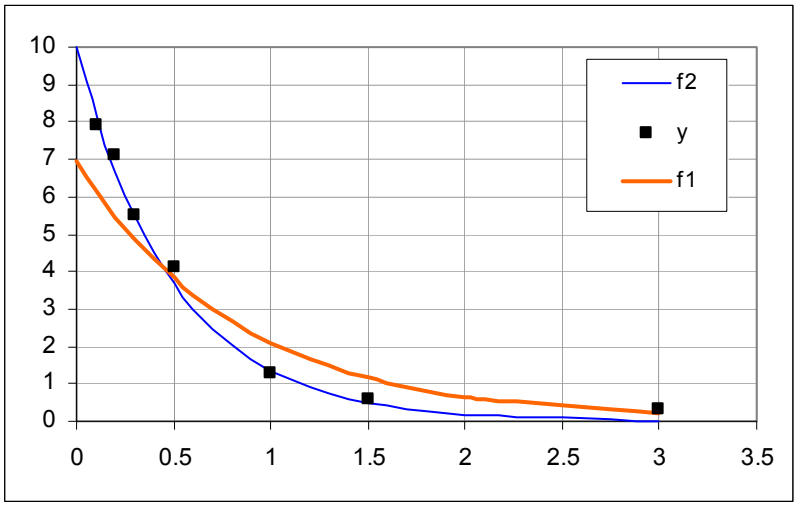
k	y_0	Standard error
-1.194	6.953	0.91

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 see that there is another couple of parameter - differently from the ones given by linear regression, that are better from the point of view of the least squares criterion.

This can also be seen, at the first sight, by the following graph



The curve obtained by the linearized regression f_1 is not sure the best fit for the original data set

From the above example a question raises: 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 linearized method gives result sufficiently close to the best regression.

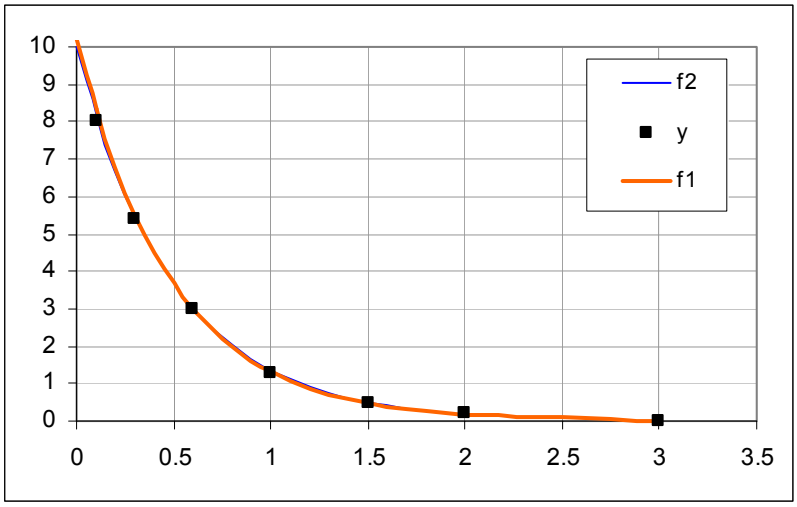
Let's repeat the exponential regression with this data set

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	z0	y0
-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 the data set (x_i, y_i) into a new data set (t_i, y_i) , where $t_i = \text{Ln}(x_i)$
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 the LINEST built-in function, finding "b_0" and "b_1". Then we have calculated the estimated values in the column "f_1", and the value of residues "error f_1". 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_1 = 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			
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			
5	2.5	10.33		10.278	0.052	10.163	0.167			
6	3			12.118	0.232	11.986	0.364			
7	4			.021	0.169	14.863	0.327			
8	5	16.68		273	-0.593	17.094	-0.414			
9					STD f1		STD f2			
10					0.5091		0.5107			
11										
12	{=LINEST(C2:C8,A2:A8)}									
13			b1	b0						
14	parameter for f1		10.091	1.032						
15	parameter for f2		10	1						

=(B2-F2)

=$D$15+$C$15*LN(A2)

=LN(A2)

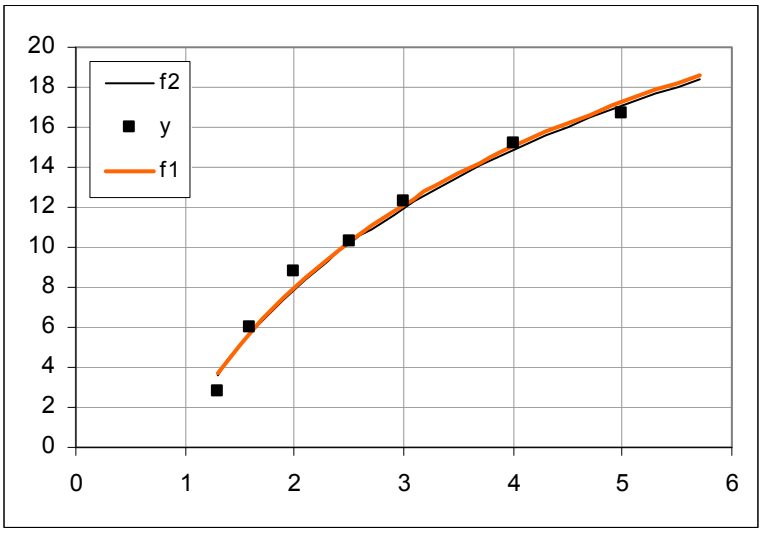
=$D$14+$C$14*LN(A2)

=(B2-D2)

=STDEVP(E2:E8)

=STDEVP(G2:G8)

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 the 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

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

$f(x) = (b_0 + b_1 x)^{-1}$

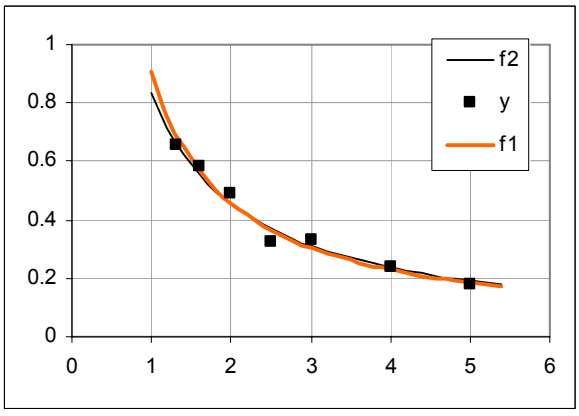
Parameters to determine are: b_0 and b_1

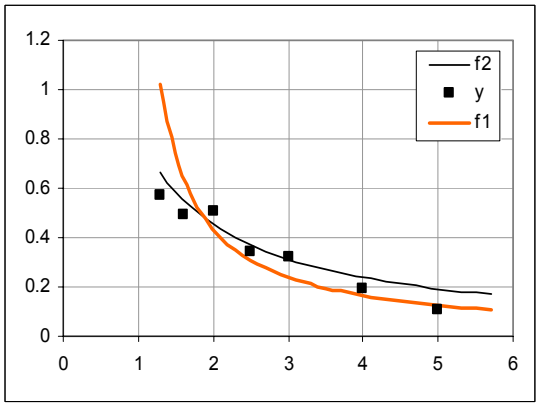
	A	B	C	D	E	F	G
1	x	y	z	f1	error f1	f2	error f2
2	1.3	0.66	1.521	0.698	-0.041	0.667	-0.009
3	1.6	0.58	1.718	0.569	0.013	0.556	0.026
4	2	0.49	2.032	0.456	0.036	0.455	0.038
5	2.5	0.32	3.107	0.365	-0.043	0.370	-0.048
6	3	0.33	3.026	0.305	0.026	0.313	0.018
7	4	0.24	4.242	0.229	0.007	0.238	-0.002
8	5	0.18	5.580	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			

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 then 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

$$\ln(y) = \ln\left(a\, x^{\alpha}\right) \quad \Rightarrow \quad \ln(y) = \ln(a) + \alpha \ln(x)$$

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 the 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 the 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^{α}

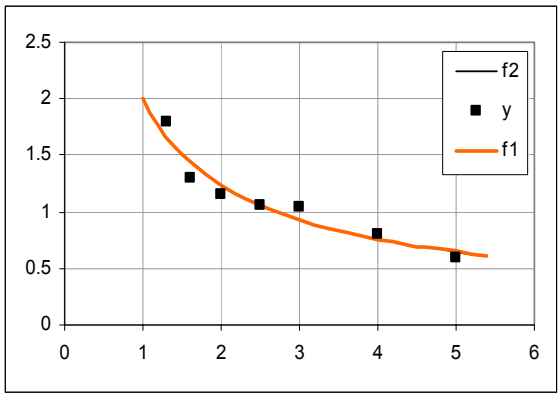
Parameters to determine are:
a and α

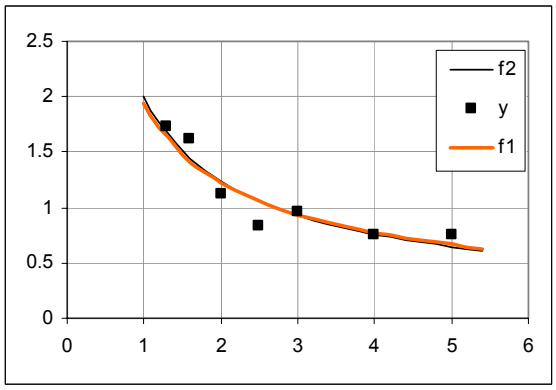
	A	B	C	D	E	F	G	H
1	x	y	t	z	f1	error f1	f2	error f2
2	1.3	1.79	0.262	0.583	1.664	0.127	1.664	0.127
3	1.6	1.30	0.470	0.266	1.440	-0.135	1.439	-0.134
4	2	1.16	0.693	0.145	1.233	-0.076	1.231	-0.075
5	2.5	1.06	0.916	0.057	1.055	0.003	1.053	0.006
6	3	1.04	1.099	0.037	0.929	0.109	0.927	0.111
7	4	0.80	1.386	-0.217	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						STD f1		STD f2
10						0.0899		0.0898
11								
12								
13			α	z0	a			
14	parameter for f1		-0.697	0.692	1.99771			
15	parameter for f2		-0.7		2			

=LN(A8) =LN(B8) =$E$14*A8^$C$14 {=LINEST(D2:D8,C2:C8)} =LN(B8)

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.

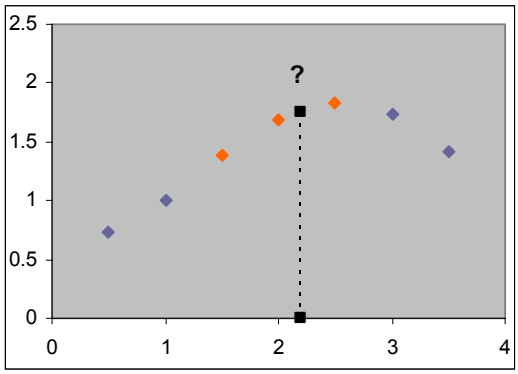
Chapter

5

Interpolation

Recalls

Given a set of values of an unknown function $y_i = f(x_i)$ calculates at N points x_1, x_2, … 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 between the 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 xi points are often equispaced. Sometime the points may be random.



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 some common interpolation methods: the 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 they use the same algorithms.

Why to interpolate?

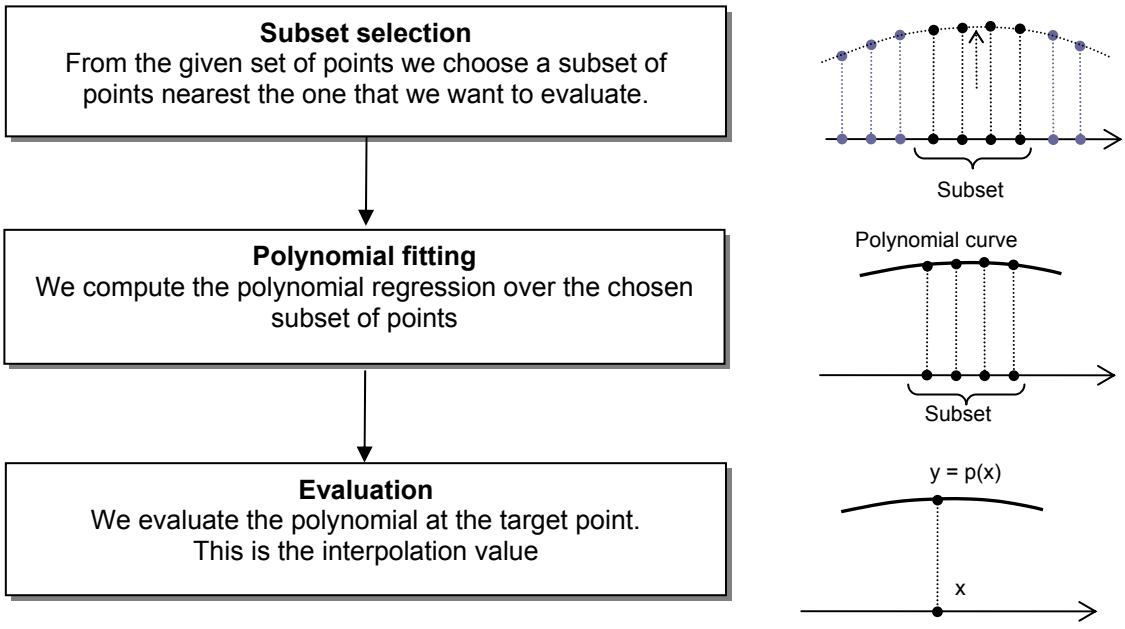
Many different problems can take advantage from the power of interpolation methods. The most common is the so called *sub tabulation* problem. It happens when we want

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 task, the data smoothing and regularization, may 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 shell 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.

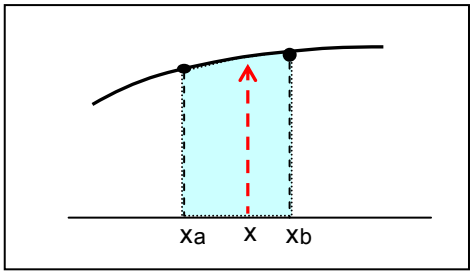


The second step is common with the modeling problem, but we still emphasized the fact that here the polynomial is used only to evaluate the y value at point x. Changing the point x, also the subset and, consequently, the polynomial may change. So there would be many polynomial formulas covering the entire 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

$$y(x) = \frac{y_b - y_a}{x_b - x_a} \cdot (x - x_a) + y_a$$

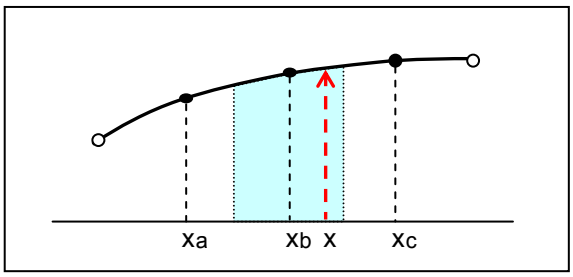
Or, as well, by the normalized formula

$$y(x) = y_b \cdot t + y_a \cdot (1 - t) \quad , \quad t = \frac{x - x_a}{x_b - x_a}$$

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

$$\frac{x_a + x_b}{2} < x \leq \frac{x_b + x_c}{2} \qquad (1a)$$

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

$$y = a_0 + a_1\, x + a_2\, x^2 \qquad (2a)$$

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 Wandermonde's matrix that can be easily obtained by the function **Mat_Vandermonde** and the solution can be find 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

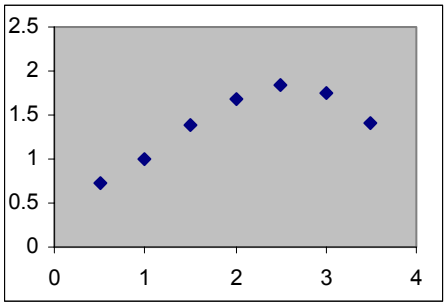
	A	B	C	D	E	F	G	H	I
10				{=Mat_Vandermonde(A13:A15)}					
11	Knots								
12	x	y		Vandermonde matrix				parabolic coeff.	
13	1.5	1.37791		1	1.5	2.25		a0 =	-0.6146
14	2	1.69285		1	2	4		a1 =	1.852176
15	2.5	1.83318		1	2.5	6.25		a2 =	-0.34923
16									
17				x =	2.2		{=SYSLIN(D13:F15;B13:B15)}		
18				y =	1.76994	=I13+I14*E17+I15*E17^2			
19									
20				y =	1.76994	=Interpolate(E17;A13:B15;2)			
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 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..

Example. Sub tabulate the following table for with step $\Delta x = 0.1$ between 0.5 and 3.5

Knots

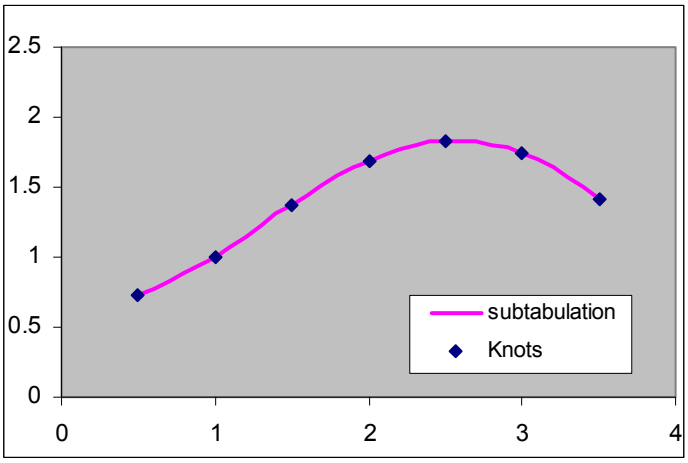
x	y
0.5	0.73008
1	1
1.5	1.37791
2	1.69285
2.5	1.83318
3	1.74207
3.5	1.41966



	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)}				1.2343
41					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 the sub-tabulation.

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 the interpolation range.

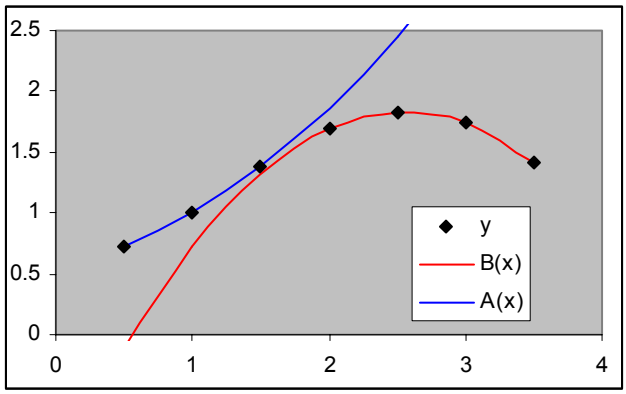
The coefficients of the interpolation polynomials can be computed by the **REGRP** function

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$5*A9^2	
12							
13	{=REGRP(2;B7:B9;A7:A9)}					=$D$7+$D$8*A9+$D$9*A9^2	

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, 31.5]



From the graph we understand why we need many parabolas in order to obtain a good interpolation accuracy for all points of the range.
The first parabola A(x) is used to interpolate the points nearest the 1st and 2nd knots. At the end, we use the parabola B(x) to better interpolate points nearest the 6th an 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 parabola works badly at the end of the range.

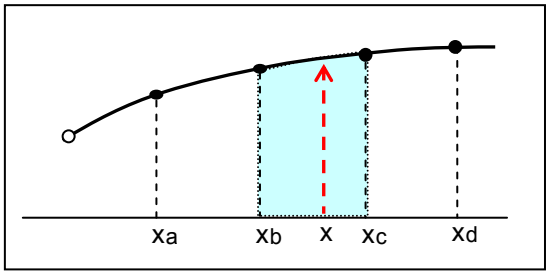
Cubic interpolation

We may choose 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 expect 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: *Degree* = *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 Wandermonde'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

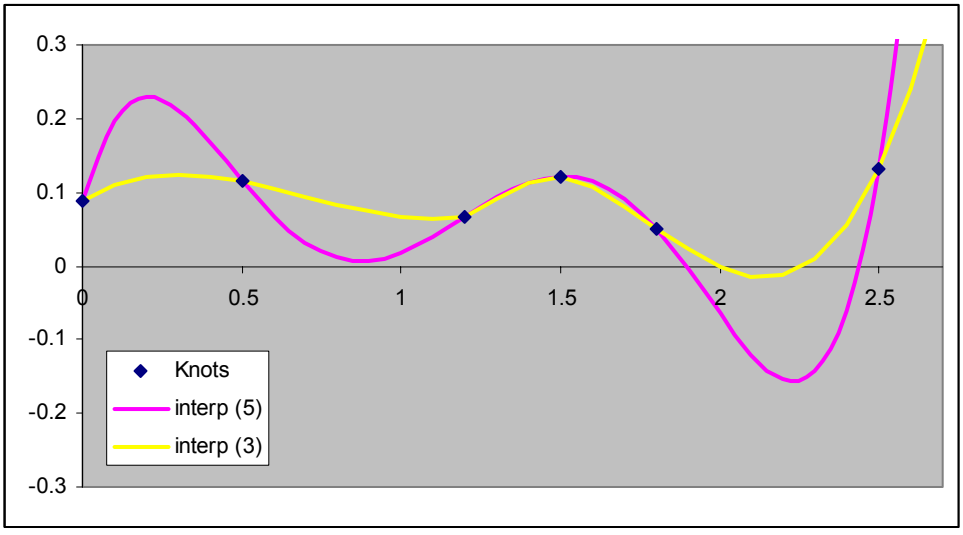
x	y
0	0.0887048
0.5	0.11539
1.2	0.0659381
1.5	0.121927
1.8	0.0513094
2.5	0.1306148

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, especially at the boundaries 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]. Cleary, 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 interpolate values near the range limits [0 , 2.5] may be largely inaccurate

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.

$$f(t) = \begin{cases} 1 & , \quad t < t_0 \\ e^{-k(t-t_0)} & , \quad t \geq t_0 \end{cases}$$

Where:

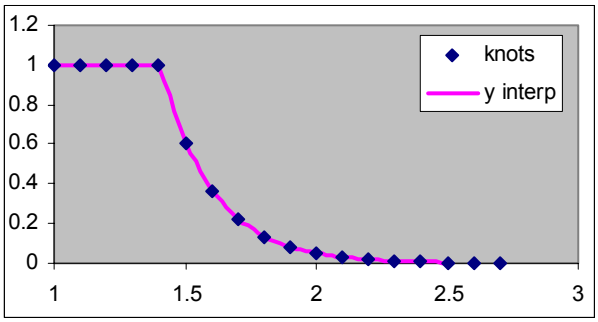
$t_0 = 1.4$, $k = 5$

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

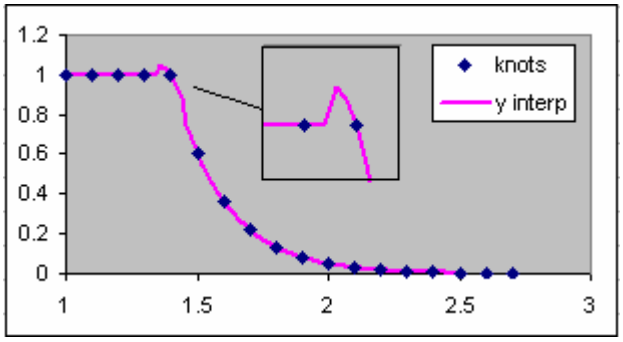
Linear interpolation.

The fit appears very good. The interpolate curve seems to follow the original trend of the points



Parabolic interpolation

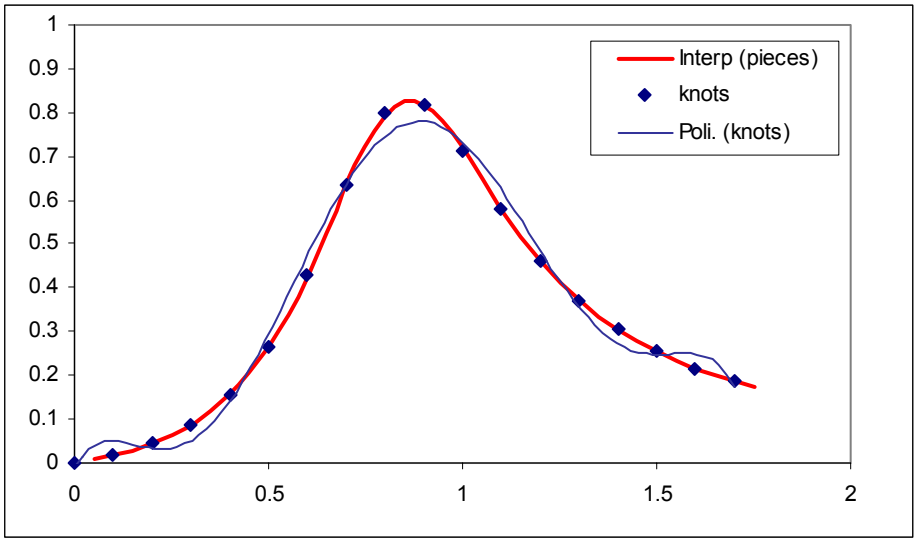
In this case we note a peak near the corner at x = 1.4



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 intepolation 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 (not 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. This leads to an over determined linear system that can be solved with the *Least Squares* method.

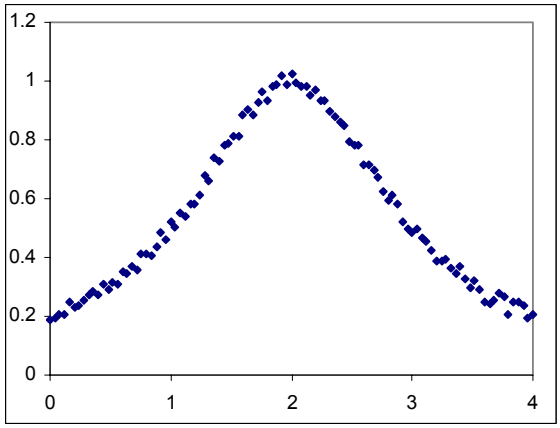
For example, for a parabolic interpolation (that require at least 3 points) we can choose the nearest 6, or 10 points. The number varies from case to case, and it is correlated to the error of to the points: great errors involve large subset of points and vice versa.

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.

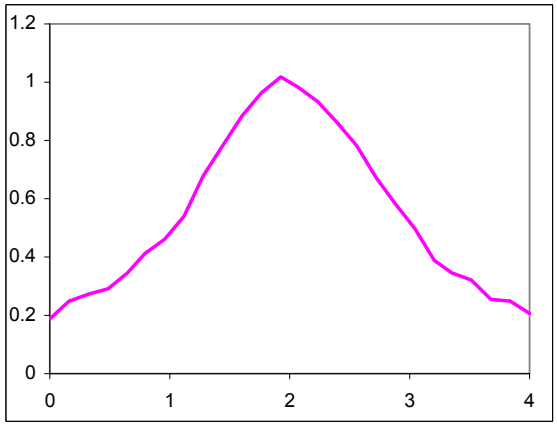
The function Interpolate(x, knots, [degree], [points]) has a 4th optional parameter setting the number of the points to choose for the regression. If omitted, the function assumes the number equal to the polynomial degree plus one (exact interpolation)

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.

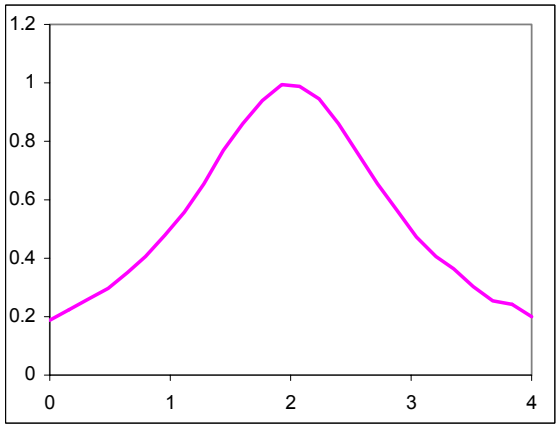
Row data points



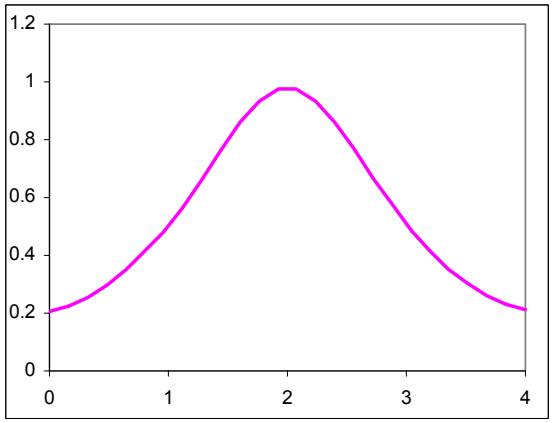
Degree = 2 , Points = 3 (exact interpolation)



Degree = 2 , Points = 8



Degree = 2 , Points = 40

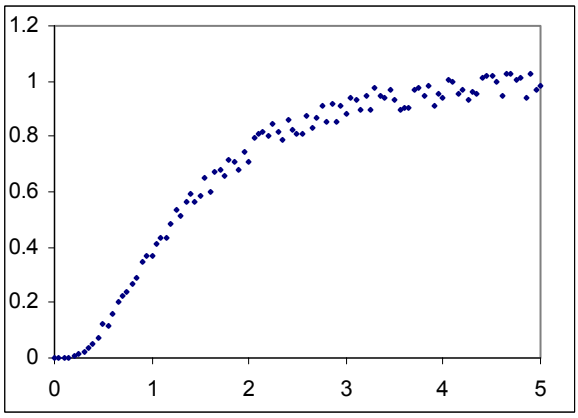


As we can see the “smoothing” effect is quite evident. We put in evidence that a larger number of points give a more smoothing curve but, on the other hand, shows the tendency to lose the original trend. We have to choose a right compromise.

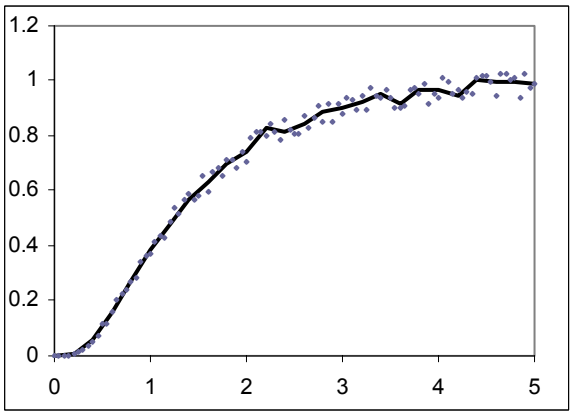
Let’s see the following example

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.

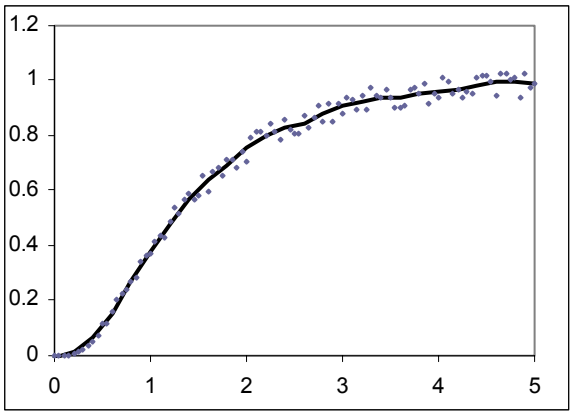
Row data points



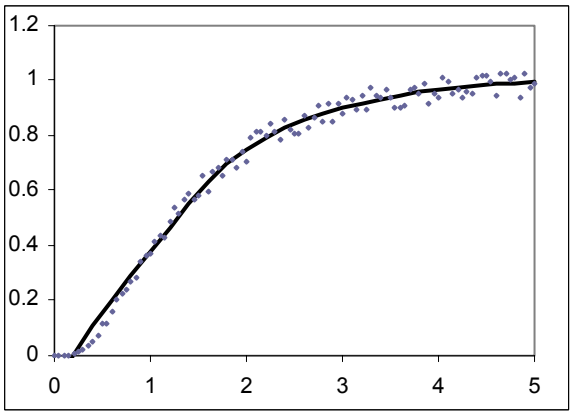
Degree = 2 , Points = 8



Degree = 2 , Points = 20



Degree = 2 , Points = 50

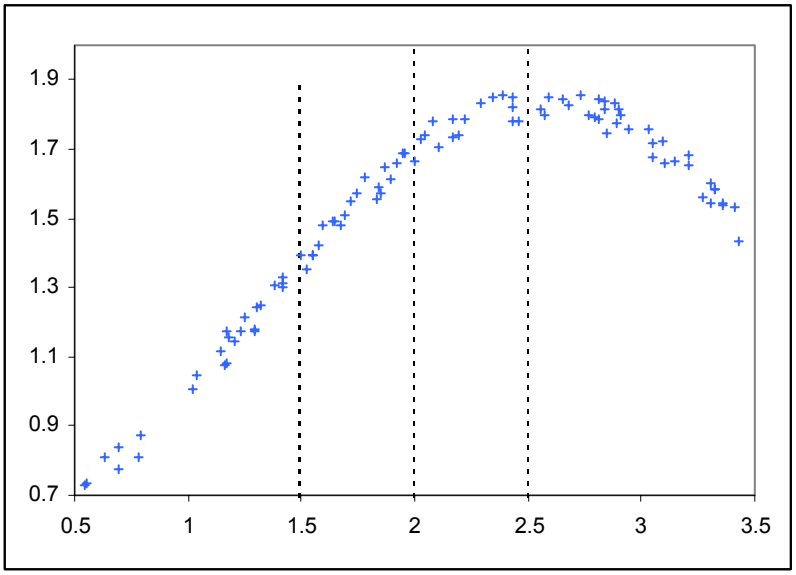


As we can see the 2nd piecewise regression with 20 points seems the best compromise. The 1st curve is not smoothing enough, while the 3rd curve is very smooth but near the origin it shows a different trend respect to the original points The 2nd curve follow better the local "knee" effect near the origin 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 follow 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

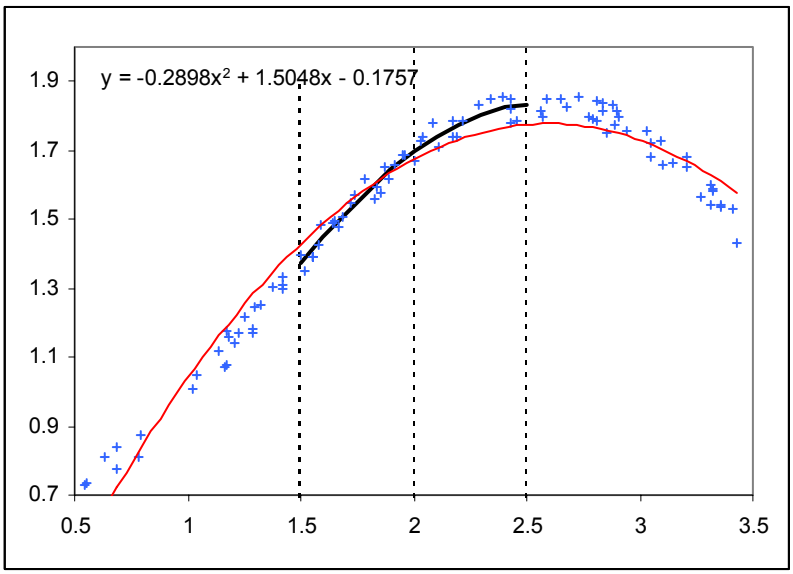


This scattering plot was extracted from an experimental dataset. The points are heavy perturbed by random noise.

We are interested in two problems:

1) find a smooth curve that best follow the trend of the points.

2) find a closed simplest formula (if it exists) that best fit the points



The read light continue curve is the parabola, 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

WHITE PAGE

References

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

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

[3] "Numerical Recipes in FORTRAN 77- The Art of Scientific Computing" - 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software

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

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

[6] “Linear Algebra” Jim Hefferon, Saint Michael’s College, Colchester (Vermont), July 2001.

[7] “Linear Algebra - Answers to Exercises” Jim Hefferon, Saint Michael’s College, Colchester (Vermont), July 2001

[8] “Calcolo Numerico” Giovanni Gheri, Università degli Studi di Pisa, 2002

[9] “Introduction to the Singular Value Decomposition” by Todd Will, UW-La Crosse, University of Wisconsin, 1999

[10] “Calcul matriciel et équation linéaires”, Jean Debord, Limoges Cedex, 2003

[11] “Leontief Input-Output Modelling” Addison-Wesley, Pearson Education

[12] “Computational Linear Algebra with Models”, Gareth Williams, (Boston: Allyn and Bacon, 1978), pp. 123-127.

[13] “Scalar, Vectors & Matrices”, J. Walt Oler, Texas Teach University, 1980

WHITE PAGE

Analytical Index

ITALY
leovlp@libero.it

5. Edition
5 Printing: June. 2005