Volume

2

FOXES TEAM

Tutorial on Numerical Analysis with Matrix.xla

Matrices and Linear Algebra

TUTORIAL ON NUMERICAL ANALYSIS WITH MATRIX.XLA

Matrices and Linear Algebra

Index

Chapter

1

About this tutorial

Matrix.xla

Matrix.xla is an Excel add-in that contains useful functions and macros for matrix 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, QH, 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 and macros. Of course it speaks about math and linear algebra, 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 Excel can be found ion many Internet sites.

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

Chapter

2

Linear Systems

This chapter explains how to solve linear systems of equations with the aid of many examples. They cover the major part of cases: systems with a single as well as with, infinitely many solutions, or none at all. Several algorithms are shown: Gauss-Jordan, Crout's LU factorization, SVD

Linear System

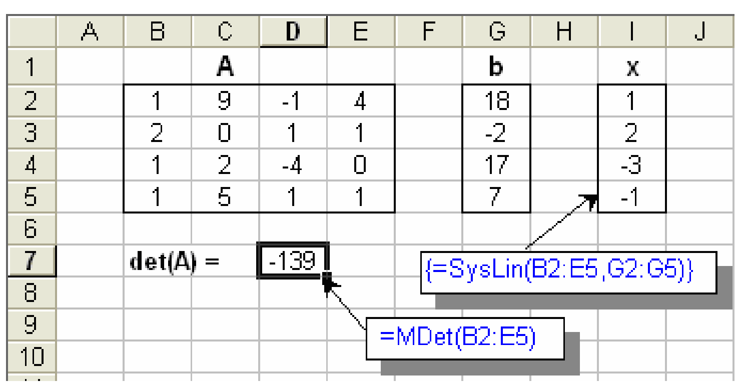
Example 1. Solve the following 4x4 linear system

A x = b

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

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

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, if **A** is non-singular. In that case we can solve the problem with the **SysLin** function.



The determinant can be computed with the MDet function or with Excel's built-in function MDETERM.

The Gauss-Jordan algorithm

The Gauss and Gauss-Jordan algorithms are probably the most popular approaches 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 equal to 1 by using a few row operations: linear combination, normalization, and exchange.

Let's see how it works

Example: The following 3x3 system has the solution $x_1 = -1$; $x_2 = 2$; $x_3 = 1$, as you can verify it by direct substitution.

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

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 ≠ 0 and pivot ≠1 then divide all first row elements by the value of the 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 substitute the second row with the difference between the second row itself and the first row multiplied by a21

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

3. Linear combination: if $a_{31} \neq 0$ then substitute 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, which 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 perform these tasks by using the power of array functions. Below is an example of the resolution of a system by the Gauss-Jordan algorithm

Note that all the rows are obtained by array operations {...}. You must insert them with 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). The formulas used for computing 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. This operation does not affect the final solution at all; it is equivalent to reordering the algebraic equations of the given system

Example: The following 3x3 system has the 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}, is zero. We cannot normalize this row
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 now has the element $a_{21} = 0$, so we simply leave that row unchanged. The linear combination isn't needed in this case

The pivoting strategy

Pivoting can always be 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: to minimize round off errors.

Pivoting can reduce round off errors

The Gaussian elimination algorithm can have a large number of operations. If we count the operations for the resolution of one system of n simultaneous equations, we will discover that it requires of the order of $n^3/3$ computer operations, i.e.., additions, subtractions, multiplications, and divisions. So, if the number of equations and unknowns doubles, the number of operations increases by a factor of 8. If n = 200, then there are more than two million operations! Certainly, one might begin to worry about the accumulation of round off errors. One method to reduce such round off errors is to avoid division by small numbers, and this is known as *row pivoting* or *partial pivoting* , the strategy of the Gaussian elimination algorithm.

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

Its 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 a numerical precision of 15 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

On the contrary, if we simply exchange the order of the algebraic equations, so that the second row becomes the first one, 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, with 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 searches the element below it for the maximum absolute value, to be used as pivot. If that maximum value is greater then that of the current pivot, then the row of the pivot and the row of the maximum value are exchanged.

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

if |a63| =max(|a33|, |a43|, |a53|, |a63|)

then rows 6 and 3 are swapped, and the old element a63 becomes the new pivot a33

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 in which to search for a maximum pivot we could exchange rows and columns. But when we swap two columns, the corresponding unknown variables are also exchanged. So, in order to rebuild the final solution in the original given sequence, we have to perform all the permutations, in reverse order, that we have made. This makes the final algorithm a bit more complicated, because we now have to store all columns permutations.

The full pivoting method extends the search area for the maximum value

For example, if the pivot is element a_{33}, then the algorithm searches for the absolute maximal value in the yellow area below and to the right of a_{33}. If a maximum value is found at a56, then rows 5 and 3 are swapped and, thereafter, columns 5 and 3 are exchanged.

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 non-integer numbers - and round off errors -, even if the solutions and coefficients of the system are integers.

Is there a way to avoid such decimal round off errors 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 approach - is very similar to the one that is sometimes 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 that we have the following two rows: the pivot row, and the row that has to be reduced.

Pivot is a_{11} = - 6

The element to set zero is a_{21} = 4

mcm = MCM(6 , 4) = 12

Multiply the first row by mcm / a_{21} = 12/4 = 3

And the second row by –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	now, add the two rows
12	9	0	30	

-6	0	5	9	<== the first row remains unchanged
0	9	10	48	<== add the first row to the second row

In this way we can reduce a row only using integer numbers
Let's see how it works, step by step, in the function **GJ_setp** of Matrix.xla.

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

Note the 3rd parameter setting the integer algorithm. If "false", the operations will use standard floating point operations using real (i.e., not necessarily integer) numbers.

Only the last step can introduce decimal numbers; the previous steps are always exact. Unfortunately, this method cannot be adopted in general, even for matrices containing only integers, because the values grow at each step and can become large enough to cause overflow

The function used are:

{=GJstep(A5:D7,,True)} inserted in the range A9:D11

{=GJstep(A9:D11,,True)} inserted in the range A13:D15

{=GJstep(A13:D15,,True)} inserted in the range A17:D19

{=GJstep(A17:D19,,True)} inserted in the range A21:D23

{=GJstep(A21:D23,,True)} inserted in the range A25:D27

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

This sequence shows how to do it.

	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						
7						
8						
9						
10						

Given a complete system matrix in range B2:E4, select the range A6:E8, just below the given matrix (leaving a free row for separation)

	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		{=GJstep(B2:B4, , TRUE)}				

Insert the array function GJstep with the CTRL+SHIFT+ENTER key sequence and the given parameter

You should see the modified matrix after the first step. 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 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 intermediate Gauss-Jordan matrices, either in floating or in integer mode (at your option).

Several ways to use the Gauss-Jordan algorithm

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

Solving a non-singular linear system

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

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 the given system; the original matrix **A** is transformed into the identity matrix.

Solving m simultaneous 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 found in the last m columns of the complete matrix

Inverse matrix computing

This problem is similar to the one above, except that the matrix **B** is the identity matrix. In fact, by 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 (3 x 6) matrix is:

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

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

Determinant computing

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

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

after which the determinant can be computed readily as

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

Linear independence checking

A linearly independent set of vectors S = {$\mathbf{v}_1$, $\mathbf{v}_2$...$\mathbf{v}_n$} is this in which no vector is a combination of the others. Gauss algorithm can evidence how many linear dependent vectors there are in a given set. For that simply perform the triangularization of the matrix in which the columns are the vectors of the set.

$$\left[\begin{pmatrix} v_{11} \\ v_{21} \\ v_{31} \\ v_{41} \end{pmatrix}, \begin{pmatrix} v_{12} \\ v_{22} \\ v_{32} \\ v_{42} \end{pmatrix}, \begin{pmatrix} v_{13} \\ v_{23} \\ v_{33} \\ v_{43} \end{pmatrix}, \begin{pmatrix} v_{14} \\ v_{24} \\ v_{34} \\ v_{44} \end{pmatrix} \right] \Rightarrow \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} \\ v_{21} & v_{22} & v_{23} & v_{24} \\ v_{31} & v_{32} & v_{33} & v_{34} \\ v_{41} & v_{42} & v_{43} & v_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The number of zero rows at the bottom of the triangularized matrix coincides with the linearly dependent vectors: i.e., one zero row, one dependent vector; two zero rows, two dependent vectors, and so on. Of course, no zero row means that the columns of the matrix form a linearly independent set.

Non-singular Linear system

The function SysLin finds the solution of a non-singular linear system using the Gauss-Jordan algorithm with full pivoting 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 is the direct application of the formula (2); the second 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 using Excel's MINVERSE and our SysLin function. In both cases we find the unitary solution (1, 1, 1, 1, 1, 1) (Note that the algebraic sum of terms in each row is equal to the corresponding constant term b)

	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 their algorithms are different. In this case both solutions are very accurate (≈1E-15) but this is not always true.

Round-off errors

Sometimes, the round-off errors decrease the obtainable maximum accuracy
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, again, 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 calculated 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 more than a thousand times greater that that in the previous example.

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

$$\mathbf{A} = \begin{bmatrix} 3877457 & -3 & -347 & -691789 & 3877457 \\ -3773001 & 0 & 34 & 46 & -3773001 \\ -286314 & 1 & 0 & -2 & -286314 \\ -377465 & -12 & 6 & 4 & -377465 \\ -1 & 0 & -6 & 0 & -1 \end{bmatrix} \qquad \begin{vmatrix} 387 \\ 194 \\ 599989 \\ 0 \\ 200001 \end{vmatrix}$$

As we can easily see by inspection, the matrix is singular because the first and last columns are equal. So there is no solution for this system. But if you try to solve this system with the MINVERSE function you will get a totally different (and clearly wrong) result. This error is particularly sneaky because, if we try to compute the determinant, we also get a wrong, non-zero result

MDETERM(A) = -0.0082

As the algorithms used by Excel and Matrix.xla are not equal, we can compute an alternative solution with SysLin and the determinant with MDet . In this case, the full pivot strategy of Gauss-Jordan is used, and gives 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	=MDet(A2:E6)	

Full pivoting or partial pivoting?

The full-pivoting strategy reduces the round-off errors, so we might expect that its accuracy is greater than with a partial-pivoting method. But this is not always true. Sometimes it can

happen that the full strategy gives an error similar to or even greater than the one obtained by the partial strategy.
In Matrix.xla we can perform the partial Gauss-Jordan algorithm using the didactic function GJstep.
Example: Solve the following linear system. The matrix is the inverse of the 6x6 Tartaglia matrix. The exact system solution is the vector [1, 2, 3, 4, 5, 6]

A =

							b
6	-15	20	-15	6	-1		0
-15	55	-85	69	-29	5		1
20	-85	146	-127	56	-10		0
-15	69	-127	117	-54	10		0
6	-29	56	-54	26	-5		0
-1	5	-10	10	-5	1		0

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 somewhat more accurate than full pivoting. Why, then, complicate the algorithm with full pivoting? The reason is that the Gauss-Jordan method, with full pivoting, is generally more stable for a large variety of matrices. Moreover, its round-off error control is more efficient. And the frequency of catastrophic mistakes, such as in the earlier comparison of MINVERSE and SysLin, is greatly reduced with a full-pivoting strategy.
Look at this example: Solve the following system

A =

						b
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 the Gauss-Jordan algorithm with either partial or full pivoting we note in this case a loss of accuracy of more than thousand times for partial pivoting.

[3] Note that, in these problems, we have not inserted the results given by Excel's MINVERSE function, because we will ignore that algorithm: in a long series of testes, we have found that its results resemble those obtained by a partially pivoting algorithm).

	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 that have large values in their right side. In that case, the round-off errors grow sharply. Full pivoting avoids this rare - but heavy - loss of accuracy.

Solution stability

Sometimes, coefficients of a linear system cannot be known exactly. Often, they derive from experimental results, and can therefore be affected by experimental errors. We are interested in investigating how the system solution changes with such errors. Many important studies have demonstrated that the solution behavior depends on the matrix of system coefficients. Some matrices tend to amplify the errors of the coefficients or the constant terms, so the solution will be very different from that of the "exact" system. When this happens we call it an "*ill-conditioned*" or "*unstable*" linear system.

Example: show that the following linear system, with the Wilson 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 some 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

A x' = **b'** **x'** = **x** + Δ**x**

Defining the system sensitivity coefficient as

S = (Δ**x** %) / (Δ**b** %) = (| Δ**x**| / | **x** |) / (| Δ**b**| / | **b** |)

We find S ≅ 400.

	A	B	C	D	E	F	G	H
1	**System perturbation**							
2	Wilson matrix							
3								
4	**A (4 x 4)**				**b**	**x**	**b'**	**x'**
5	10	7	8	7	32	1	32.1	-0.2
6	7	5	6	5	23	1	23.1	3
7	8	6	10	9	33	1	33.1	0.5
8	7	5	9	10	31	1	31.1	1.3
9								
10	**det(A) =**	1		**Δ b**	**Δ x**	**Δ b %**	**Δ x %**	**S**
11				0.1	1.3136	0.17%	66%	394.2
12								
13								
14								
15								

{=SysLin(A2:D5,E2:E5)}

{=SysLin(A2:D5,E2:E5)}

{=E5:E8+G11}

=MAbs(H5:H8)-MAbs(F5:F8)

=D11/MAbs(E5:E8)

=E11/MAbs(F5:F8)

=G11/F11

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

-0.2, 3, 0.5, 1.3

, which is completely different from the exact one,

1, 1, 1, 1

Note that Det =1

Even worse stability is found in the following linear system

A=

117	85	127	118		447
97	70	103	97		367
74	53	71	64		262
62	45	65	59		231

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

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

Note the very high sensitivity coefficient S of this problem, and the wide spread of the solution point, even for very small perturbations.
Note also that, in both problems, the determinant was unitary (Det = 1). So we cannot discover the instability simply by considering the determinant.

The Condition Number

One popular measure of instability matrix uses its eigenvalues

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

But, unfortunately, eigenvalues are not very easy to compute
A more practical index is based on the singular value decomposition (see the SVDD) function). Extracting the largest and smallest singular values of the diagonal matrix **D** we define the measure of instability, commonly called the condition number, as:

$$k = d_{max} / d_{min}$$

For the above matrix the eigenvalues are

λ=	323.98	-5.72328	-1.256573	0.000429

So we have:

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

while the SVD gives

$k = 340.9 / 0.000308 \cong 1106504$

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

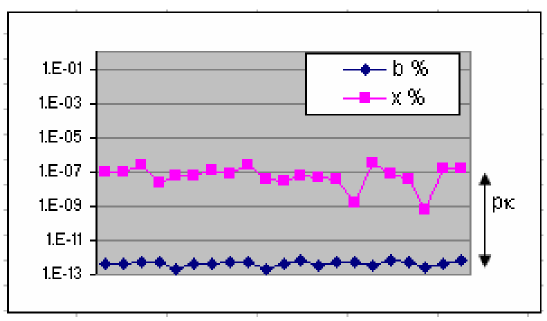
It is also possible to compute directly the condition number with **MCond** and **MpCond** functions

Apart theory, the condition number has a useful, practical meaning: in system solving, it indicates how many significant digits will be lost. See this example
Taking the last Wilson system, we perturb the vector **b** with a small random error:
$b'_i = b_i(1 + e \cdot \mathrm{R})$ where ε = 1E-12 and R is a uniform random variable between 0 and 1.
For each set of **b** values, we register the average error of the solution obtained with the formula **x** = $\mathbf{A}^{-1} \cdot \mathbf{b}$

	A	B	C	D	E	F	G	H	I
1	**A**					**b**	**Δ b %**	**x**	**Δ x %**
2	117	85	127	118		447	8.8E-13	1.000000329	3.3E-07
3	97	70	103	97		367	1.1E-13	0.999999453	5.5E-07
4	74	53	71	64		262	8.5E-13	1.000000144	1.4E-07
5	62	45	65	59		231	2.7E-13	0.999999913	8.7E-08
6									
7		pκ =	-6.044	=MpCond(A2:D5)					

As we can see the average error of **x** is about 1E-7, just 6 digits less then the precision of the **b** vector.
The precision leakage roughly corresponds to the decimal log of condition number

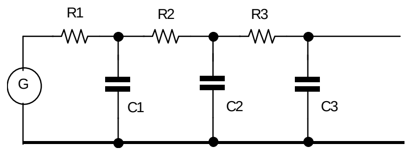
$$p\kappa = -\log_{10}(\kappa) \cong -6$$



Complex systems

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

Example - Analysis of a lattice network. For all nodes, find the voltages and phase angles, at the frequencies: f = 10, 50, 100, and 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)

We use the notation $v(t) = V\sin(w\,t + q) \Leftrightarrow \dot{V} = Ve^{jwt} = V_{re} + jV_{im}$

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

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

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

Using the worksheet, the problem can be solved by calculating, first of all, the frequency ω, the two real matrices **G** and **B**, and the input current 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		{=SysLinC(D3:I5, J3:K5)}						

SysLinC provides the vector solution in complex form; to convert it into magnitude (modulo) and phase we have used the 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 added an imaginary column to the current vector, even though the input currents are purely real quantities. Complex matrices and complex vectors must be always be specified with both their real and imaginary parts. Consequently they must always have an even numbers of columns.

In the above example there are many Excel formulas that we couldn't shown for clarity. To more fully explain 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 =			=MAbs(E8:F8)		=ATAN2(E8,F8)*180/PI()		
9	G	2.5	V	v2 =	{=SysLinC(D3:I5,J3:K5)}		=MAbs(E9:F9)		=ATAN2(E9,F9)*180/PI()		
10	f	400	Hz	v3 =			=MAbs(E10:F10)		=ATAN2(E10,F10)*180/PI()		
11		2513.3	rad/s								

See also the function MAdm for admittance matrix.

Example - Solve the following complex system

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

The system is equivalent to the following complex matrix equation

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

With the SysLinC function it is simple to find the solution of such 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					{=SysLinC(A2:F4,H2:I4)}							

About the 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 its advantages and drawbacks.

In the split format the complex matrix [**Z**] is split into two separate matrices: the first one contains the real values, and the second the imaginary values. This is the default format

In the interlaced format, each complex value is written in two adjacent cells, so that a single matrix element occupies two cells. The number of columns is the same as in 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 as, e.g, by the Xnumbers.xla add-in

The last format is the well known "*complex rectangular format*". Each element is written as a string "a+ib" so that a square matrix is still square. It appears to be the most compact and intuitive format, but this is true only for integer values. For long decimal values the matrix elements become illegible. We should also point out that the elements, being strings, cannot be formated as other Excel numbers, or even used in subsequent computations without conversion from text strings to numbers.

Determinant

In contrast to 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 matrix can be computed with the following simple rules

- When we multiply a matrix row by a number k, the determinant must be multiplied by the same number
- When we exchange two rows, the determinant changes its sign but retains its magnitude

Gaussian elimination

With these simple rules it is easy to calculate the matrix determinant. It is sufficient to track of all pivot multiplications and rows swappings performed during the Gauss-Jordan process

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

- A triangular matrix and a 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 the computation effort. This is called the Gauss algorithm or *Gaussian elimination*.

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

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

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

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

And:

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

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

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

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

R2 = R1 + 2*R2 (*)

Det(A1) = 2 Det(A)

(*)
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

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

R3 = R1 + (– 4)*R3

Det(A2) = -8 Det(A)

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

Det(A3) = 8 Det(A)

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

R3 = R2 + 3*R3

Det(A4) = 24 Det(A)

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

Det(A4) = 24 Det(A)

Det(A4) = 4*9*(-2) = -72

The final matrix A4 is triangular. So its determinant is readily computed as -72

But it is also:

$$\text{Det}(A_4) = 24\ \text{Det}(A)$$

Substituting, we have:

$$-72 = 24\ \text{Det}(A) \qquad \Rightarrow \qquad \text{Det}(A) = -24 / 72 = -3$$

Ill-conditioned matrix

Of course there are functions such as MDet in Matrix.xla and MDETERM in Excel to compute the determinant of any square matrix. Both are very fast and efficient, covering most cases. But, sometimes, they can fail because of the round-off error introduced by the finite precision of the computer. It usually happens for large matrices but, sometimes, even for small matrices. Look at this example.

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

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

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

Both functions return a very small, but non-zero value, quite different from each other.
If you repeat the calculation with another numerical routine in a 32 bit operating system you will get similar results.

The given matrix is <u>singular</u> and its determinant is 0. We can easily verify this by hand with exact fractional numbers, or by using the GJstep function with integer algorithm, as shown below.

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

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

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

In this case it was easy to analyze the matrix, but for a larger matrix do you know what would happen? Before one accepts any results - especially for large matrices, one has to perform some extra tests, such as the singular value decomposition.

Example. Compute the determinant of the following (20 x 20) sparse matrix

0	0	0	0	0	0	0	0	0	0.7	0	1.3	0	0	7.2	0	0	0	0	0
0	-0.5	0	-4	0	0	0	5.9	0	0	0	0	0	0	0	0	0	0	0	0
0.2	0	0	0	0	1.8	-0.7	0	0	0	0	0	-4.2	0	0	0	1	0	0	0
0	0	0	-4.7	0	-8.1	0	0	0	8	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	4.9	11	0	0	10.2	0	0	0	0	0	0	0	0	0
0	-10	0	0	8.4	0	0	0	0	0	6.7	0	1	0	0	0	0	0	0	2.5
0	0	0	0	0	-2.5	1	4	8	0	0	0	1	0	0	-9.6	0	0	0	0
0	0	0	9.1	0	0	0	0	5	0	0	0	1	0	0	0	0	0	0	0
0	0	8.5	0	0	0	0	0	0.5	0	4	0	1	0	-9.8	0	0	0	0	0
0	6.6	0	4.1	0	0	0	0	0	0	0	0	1	0	0	-9.6	0	0	0	3.3
0	0	0	0	0	0	0	0	4.5	0	7.9	0	1	-6	9	0	0	10.3	6.7	0
0	3.2	0	0	0	0	2.7	0	7	0	0	0	1	0	0	-5.1	0	0	0	0
0	0	-6.4	-8.6	0	0	0	0	0	0	7.7	0	1	3	0	0	0	0	0	0
0	0	-10	0	0	0	0	0	0	0	0	0	1	0	0	2	0	0	0	0
0	0	0	0	0	0	9.1	0	0	0	-1.2	0	1	0	0	3.2	0	0	0	0
0	0	1.9	0	-2.7	0	0	0	0	0	8.7	0	1	0	0	0	0	0	0	0
0	0	0	-7.6	0	0	0	0	0	0	0	0	1	2.4	0	0	0	0	0	0
-1.1	0	0	6.3	0	0	9.3	-1.6	0	0	-3.5	0	1	0	0	0	0	0	0	0
0	0	0	2.7	0	0	0	0	0	0	0	0	0	0	0	-8.7	-9.7	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	-0.9

Select the above matrix and paste it in a worksheet starting from the cell A1. Using the Excel

MINVERSE we get the determinant of about Det(**A**) $\cong$ 0.14259
If we try to change one element by a little amount - for example, the element K5 from 10.2 to 10.3 - we get a complete different result Det(**A**) $\cong$ -0.123 . Note that the determinant even changes its sign. It would be sufficient for suspecting of a large round-off error.
In fact, if we compute the determinant by the function MDet (that uses the Gauss algorithm with full-pivot) with have the result Det(A) = 0. This means that the matrix is singular.
We can check that result with the SVD algorithm. Using the function SVDD we get the singular value matrix: the minimum value, less then 1E-15, fully confirms that the matrix is singular.

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 becomes large. Laplace's expansion becomes competitive when there are rows or columns with many zeros.

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

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

where **|A** $_{ij}$**|** are the *minors*, that is the determinant of the sub matrix extracted from the original matrix eliminating the row i and the column j. The minors are taken with sign + if the sum of (i+j) is even; or with the minus sign if (I+j) is odd.

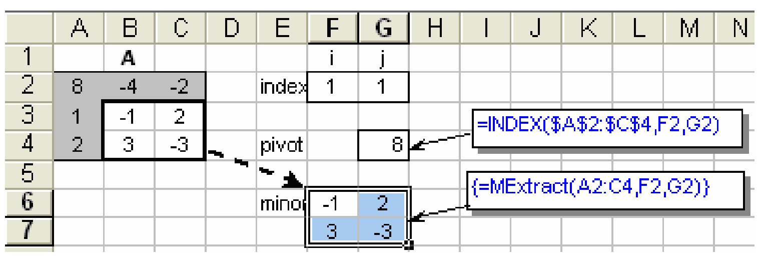
Many authors call the term: $(-1)^{(i+j)}$**|A** $_{ij}$| a *cofactor*.

Let's see how it works with an example

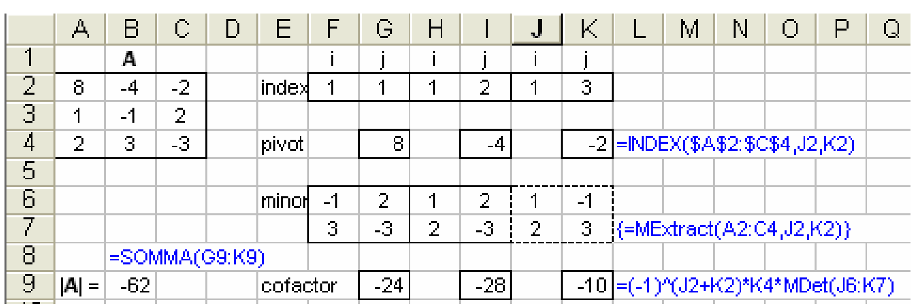
Example - Calculate the determinant of the given 3x3 matrix with the Laplace's expansion.

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

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



Completing the worksheet with the other minors and the cofactor terms we have



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

Simultaneous Linear Systems

The function SysLin can give solutions for many linear systems having the same incomplete coefficient matrix and different constant vectors.

Example: solve the following matrix equation

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

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 is the direct application of the formula (2); the second 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			{MMULT(MINVERSE(A2:D5),F2:G5)}						{=SYSLIN(A2:D5,F2:G5)}				
9													

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

Inverse matrix

Simultaneous systems solving is used to find the inverse of a matrix. In fact, if **B** is the identity matrix, we have:

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

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

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

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

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

The inverse of a Hilbert matrix is always integer. So, if any decimals appear in the result, we can be sure that they are due to round off errors, and we can consequently estimate the accuracy of the result. You can easily generate these matrices by hand or with the function MHilbert

	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	{=MHilbert(4)}			{=MINVERSE(A5:D8)}					=ROUND(F8,0)-F8					
11														

Round-off error

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

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

Applying this method to the above inverse matrix, we see that there are absolute round-off errors from 1E-13 to 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 off-diagonal values measure the errors. If we compute the mean of their absolute values we have an estimation of the round-off error.

	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			{=MInv(A2:F7)}				
11	-6E-14	1	-1E-11	3E-11	-1E-10	-1E-11							
12	-6E-14	4E-12	1	1E-10	6E-11	1E-11			{=MProd(A2:F7,H2:M7)}				
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							

The "diagonalization" accuracy measures 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 exact as, e.g., 1/3 or 1/7. But, actually, these numbers are always affected by truncation errors of about 1E-15.

Other classes of matrices, such as Tartaglia's matrices, avoid the input truncation errors, because they are always integer.

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 measuring round-off errors.

Tartaglia matrices are defined as

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

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

Here is a 6x6 Tartaglia matrix

1	1	1	1	1	1
1	2	3	4	5	6
1	3	6	10	15	21
1	4	10	20	35	56
1	5	15	35	70	126
1	6	21	56	126	252

and its inverse

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

As we can see, both matrices are integer. Any errors in the inverse matrix must be regarded as a round-off errors, and are immediately detected.
In the example below we evaluate the global accuracy of the inverse of the 6 x 6 Tartaglia matrix with two different functions

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	Tartaglia's matrix							Inverse Tartaglia's matrix				{=MInv(A3:F8)}				=ABS(ROUND(H3,0)-H3)				
2																				
3	1	1	1	1	1	1		6	-15	20	-15	6	-1		0	0	0	0	0	0
4	1	2	3	4	5	6		-15	55	-85	69	-29	5		0	0	0	0	0	0
5	1	3	6	10	15	21		20	-85	146	-127	56	-10		0	0	0	0	0	0
6	1	4	10	20	35	56		-15	69	-127	117	-54	10		0	0	0	0	0	0
7	1	5	15	35	70	126		6	-29	56	-54	26	-5		0	0	0	0	0	0
8	1	6	21	56	126	252		-1	5	-10	10	-5	1		0	0	0	0	0	0

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
10										{=MINVERSE(A12:F17)}						=ABS(ROUND(H12,0)-H12)				
11																				
12	1	1	1	1	1	1		6	-15	20	-15	6	-1		2E-13	9E-13	1E-12	2E-12	8E-13	2E-13
13	1	2	3	4	5	6		-15	55	-85	69	-29	5		8E-13	4E-12	8E-12	9E-12	4E-12	9E-13
14	1	3	6	10	15	21		20	-85	146	-127	56	-10		2E-12	9E-12	2E-11	2E-11	9E-12	2E-12
15	1	4	10	20	35	56		-15	69	-127	117	-54	10		2E-12	9E-12	2E-11	2E-11	9E-12	2E-12
16	1	5	15	35	70	126		6	-29	56	-54	26	-5		9E-13	4E-12	9E-12	8E-12	4E-12	8E-13
17	1	6	21	56	126	252		-1	5	-10	10	-5	1		2E-13	8E-13	2E-12	2E-12	8E-13	2E-13
18																				

Excel occasionally compute $\mathbf{A}^{-1}$ even if a matrix is singular. If this happens, your solution will be wrong.
Let's see this example:
Example: find the inverse of the following matrix

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

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

	A	B	C	D	E	F	G	H	I	J
1			**A**							
2		127	-507	245		-2.12E+14	-5.61E+13	-6.24E+12		-7E-10
3		-507	2025	-987		-5.61E+13	-1.49E+13	-1.65E+12		
4		245	-987	553		-6.24E+12	-1.65E+12	-1.83E+11		
5										
6					{=MINVERSE(B2:D4)}			=MDETERM(B2:D4)		

Tip. You should always examine the determinant before attempting to calculate the inverse. 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 MInv (in this case, with the integere option), or by GJstep function, or with SVD (see later).

How to avoid decimals

An inverse matrix is not always integer; usually it contains decimals. If the given matrix is integer, we can obtain the fractional expression of its inverse with this little trick

Example

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

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

Multiplying the inverse by the determinant we get a 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

A linear system

A x = b

with **A** an (n x m) matrix and with **b** = 0 we call a homogeneous linear system. . Such a system always has the trivial solution **x** = 0. But we are interested in knowing if the system also has other solutions.

Assume A is a square matrix of the following system

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

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

We note that the last row can be obtained by multiplying the first row by -2. So, having two rows that are linearly dependent, the given matrix is singular, with a zero determinant. One of the two rows can be eliminated; we choose to eliminate the last row, and obtain 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 regarded as a new independent variable. Assume, for example, z as the independent parameter; the other variables x, y can then 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 system of linear equations (1) expresses all the solutions of the given system, an infinite number of them. Geometrically speaking it is a line in the space $\mathbf{R}^3$

It can be also be regarded as a linear transformation that moves 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 this subspace is $\mathbf{R}^1$

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

If we assume, on the contrary, x as independent parameter, the other variables y, z can be expressed as functions of the "independent" parameter x

That is represented by the linear transformation at the right

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

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

This matrix transformation is useful for finding the parametric form of the linear function (mapping function)

Parametric form

The linear transformations of the above example give relations between points in space. A common form for handling this relation is the parametric form. It is 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: $\vec{D}$

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

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

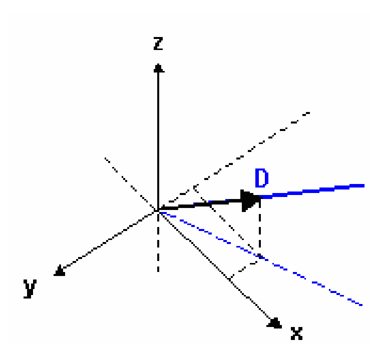
You can study the entire problem with the function **SysLinSing** of Matrix.xla. Here is an example:

	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/ MAbs(G16:G18)}			

SysLinSing solves a singular linear system, returning the transformation matrix of the solution, if one exists. The determinant is calculated only to show that the given matrix is singular. It is not used in the calculation. SysLinSing automatically detects if a matrix is singular or not. If the matrix is not singular (Det $\neq$ 0), the function returns all zeros.

From the transformation matrix we can 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 MAbs. Note that both expression must be inserted as array functions { }

In a 3D space, 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 an infinite number of solutions of the system; those solutions represent a subspace. After that we have found a solution, and we have seen that the subspace was a line and its dimension was 1.

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 this is easy only for low matrix dimensions; it becomes very difficult for high matrix dimensions.

- *The rank of a square matrix is the maximum number of independent rows (or columns) that we can find in the matrix.*

For a 3 x 3 matrix the possible cases are collected 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 **MRank** of Matrix.xla calculates the rank of a given matrix. In the following example we calculate the determinant and the rank of 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	=MRank(A2:C4)			2	=MRank(E2:G4)			1	=MRank(I2:K4)		

Note that the determinant is always 0 when the rank is less then the matrix dimension n.
Solving homogeneous systems with the given matrices, we will generate in a 3D space respectively the following subspaces: 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)				{=F16:F18/ MAbs(F16:F18)}				
21	1	=MRank(A16:C18)								
22										
23		{=SysLinSing(A16:C18)}				{=G16:G18/ MAbs(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 that the transform matrix: variables y and z have both the diagonal element 1 (a_{22} = 1 , a_{33} = 1) . These can be assumed to be independent parameters.

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

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

Eliminating both parameters we get the normal equations of the plane

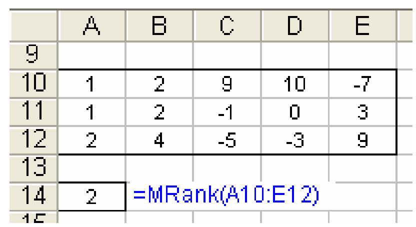
$$x = -2y + z \quad \Rightarrow \quad x + 2y - z = 0 \qquad (2)$$

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 a rectangular matrix

Differently form the determinant, the rank can be computed also for a non-square matrix. 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
9					
10	1	2	9	10	-7
11	1	2	-1	0	3
12	2	4	-5	-3	9
13					
14	2	=MRank(A10:E12)			

By inspection we see that there are 2 independent rows and 2 independent columns.
In fact, column c_2 is obtained multiplying the first column by 2; column $c_4 = c_1 + c_3$; and column $c_5 = c_2 - c_3$. So the rank is: rank = 2

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

In other words: all 3 x 3 matrices extracted from the matrix in 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

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

$$A(m \times n) = \begin{bmatrix} a_{11} & a_{12} & .... & a_{1n} \\ a_{21} & a_{22} & .... & a_{2n} \\ \multicolumn{4}{c}{.................................} \\ a_{m1} & a_{m2} & ...... & a_{mn} \end{bmatrix}$$

The matrix **A** is called the *coefficient matrix* or *incomplete matrix*

$$B(m, n+1) = \begin{bmatrix} a_{11} & a_{12} & .... & a_{1n} & b_1 \\ a_{21} & a_{22} & .... & a_{2n} & b_2 \\ \multicolumn{5}{c}{..} \\ a_{m1} & a_{m2} & .... & a_{mn} & b_3 \end{bmatrix}$$

The matrix **B** is called *complete matrix* or *augmented matrix*

If the column **b** only contains zeros, the system is called *homogeneous*

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

ROUCHÈ-CAPELLI THEOREM :

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

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

Among ranks, number of equations and number of unknowns exist important relations. The following table reviews 12 possible cases: 6 for homogeneous systems, and 6 for full system..

Homogeneous System Cases

4

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

4 This table, very clear and well-organized, is due to Marcello Pedone

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

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

Suppose that, for the linear system

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

you have gotten the following factorization

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

where **L** is lower-triangular and **U** upper-triangular. That is:

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

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

A x = **b** Þ **(LU) x** = **b** Þ **L (U x)** = **b**

Setting: **y** = **U x** we can write:

L y = **b** (3) **U x** = **y** (4)

The triangular systems (3) and (4) can now be solved with very efficient algorithms

Forward and Backward 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			R	
1	0	0	6	5	1
2	2	0	0	-1	2
-1	1	1	0	0	4

In Matrix.xla we can use the function SysLinTthat applies the efficient forward/backward algorithm to solve triangular systems.

This function has an optional parameter to switch the algorithm to the upper (Typ = "U") or lower (Typ = "L") triangular matrix. If omitted, the function automatically finds the matrix type

	A	B	C	D	E	F	G	H	I	J	K	L
1		A										
2	6	5	1					{=SysLinT(D7:F9,J7:J9)}				
3	12	8	6									
4	-6	-6	5		{=SysLinT(A7:C9,H7:H9)}							
5												
6		L			U			b		y		x
7	1	0	0	6	5	1		19		19		1
8	2	2	0	0	-1	2		46		4		2
9	-1	1	1	0	0	4		-3		12		3

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 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 MLU function. This function returns both factors in an (n x 2n) array.

But there are same things that should be pointed out. We may believe that, once we have the **LU** decomposition of **A**, we can solve as many linear systems as we want, simple changing the vector **b**. This is not completely true.

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:

	A	B	C	D	E	F	G	H	I	J
1		A				L			U	
2	0	5	4		1	0	0	-8	0	-9
3	2	4	2		-0	1	0	0	5	4
4	-8	0	-9		-0.25	0.8	1	0	0	-3.45
5										
6	{=MLU(A2:C4)}									

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

The Crout algorithm has returned the following triangular matrices:

	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	$\mathbf{y = L^{-1} b}$	$\mathbf{x = U^{-1} y}$
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 different result. Why? What's happened?

The fact is that LU algorithm does not give the exact original matrix **A,** but a new matrix **A'** that is a row permutation of the given one. This is due to the partial pivoting strategy of Crout's algorithm. You simple prove it by multiplying L and U.

So the correct factorization formula is:

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

where **P** is a permutation matrix

The process to solve the system is therefore:

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

We have shown that only the information of the two factors **L** and **U** insufficient to solve the general system. We also need the **P** matrix.

But how can we get the permutation matrix? This matrix is provided by the algorithm itself at the end of the factorization process. Most LU routines do not give us the permutation matrix, because 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 LU factorization we need, in generally, three matrices **P**, **L**, and **U**.

	A	B	C	D	E	F	G	H	I
1		**P**			**L**			**U**	
2	0	1	0	1	0	0	-8	0	-9
3	0	0	1	-0	1	0	0	5	4
4	1	0	0	-0.25	0.8	1	0	0	-3.45
5									
6	**b**		**b'**		**y**		**x**		
7	22		-35		-35		1		
8	16		22		22		2		
9	-35		16		-10.4		3		
10									
11			{=SysLinT(D2:F4,C7:C9)}						
12									
13	{=MMULT(MT(A2:C4),A7:A9)}					{=SysLinT(G2:I4,E7:E9)}			

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 by comparing the original **A** matrix with 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

A' = LU

-8	0	-9
0	5	4
2	4	2

A

0	5	4
2	4	2
-8	0	-9

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 supplied by the function MLU as the third, optional part of its output. For a 3x3 matrix you should therefore select a range of 9 (rather than the usual 6) columns to see the permutation matrix.

MLU(**A**) returns (**L** , **U** , **P**) arrays

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

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

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

If we form the matrix product **P L U** (here the MProd function is useful) we obtain finally the given original matrix. (Note that the last matrix **P** must be the first of the matrix product)

Overdetermined Linear System

If the matrix has more rows than columns, then the linear system is said to be overdetermined. Often in an overdetermined system, there is no solution **x** that satisfies all the rows exactly, but there are solutions **x** such that the residual

$$r = \mathrm{A} \cdot x - b$$

is a vector of "small values" that are within working accuracy.
The solution **x** minimizing the norm of the residual vector

$$\|r\|^2 = \sqrt{\sum_i \; r_i^2} = r^T r$$

is the least squares solution of the linear system. In this case the system has a unique solution specified by the least squares criterion.

Example. Solve the following linear system with the least squares criterion

$$\begin{cases} 2x_1 + 4x_2 & = 35 \\ x_1 + 2x_2 + x_3 & = 18 \\ -x_1 + x_2 - 2x_3 & = -8 \\ 3x_1 + 3x_3 & = 26 \\ 5x_1 - x_3 & = 33 \end{cases} \qquad \begin{bmatrix} 2 & 4 & 0 \\ 1 & 2 & 1 \\ -1 & 1 & -2 \\ 3 & 0 & 3 \\ 5 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 35 \\ 18 \\ -8 \\ 26 \\ 33 \end{bmatrix}$$

The normal equation

One way to resolve the given linear system is by transforming the rectangular system matrix into a square matrix using the so called normal equation transformation.

$$\mathbf{A} \cdot x = b \quad \Rightarrow \quad \mathbf{A}^{\mathrm{T}}\mathbf{A} \cdot x = \mathbf{A}^{\mathrm{T}} \cdot b \quad \Rightarrow \quad \mathbf{B} \cdot x = c$$

The normal matrix $\mathbf{B} = \mathbf{A}^{\mathrm{T}} \mathbf{A}$ is square and symmetric. So the last system can be resolved with the usual methods (B^{-1}, LR, LL, Gauss, etc.). The solution of the normal system is also the least squares solution of the overdetermined system

$$\mathbf{B} = \begin{bmatrix} 2 & 1 & -1 & 3 & 5 \\ 4 & 2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 3 & -1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 4 & 0 \\ 1 & 2 & 1 \\ -1 & 1 & -2 \\ 3 & 0 & 3 \\ 5 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 40 & 9 & 7 \\ 9 & 21 & 0 \\ 7 & 0 & 15 \end{bmatrix} \qquad c = \begin{bmatrix} 2 & 1 & -1 & 3 & 5 \\ 4 & 2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 3 & -1 \end{bmatrix} \cdot \begin{bmatrix} 35 \\ 18 \\ -8 \\ 26 \\ 33 \end{bmatrix} = \begin{bmatrix} 339 \\ 168 \\ 79 \end{bmatrix}$$

	A	B	C	D	E	F	G	H	I	J	K
1	Overdetermined system						Normal system				
2	A (5 x 3)				b		B (3 x 3)				c
3	2	4	0		35		40	9	7		339
4	1	2	1		18		9	21	0		168
5	-1	1	-2		-8		7	0	15		79
6	3	0	3		26						
7	5	0	-1		33						
8											
9											

{=MMULT(TRANSPOSE(A3:C7), A3:C7)}

{=MMULT(TRANSPOSE(A3:C7), E3:E7)}

Solving the normal system we find the solution [x_1 = 7, x_2 = 5, x_3 = 2]

The residual vector can be easily computed as

	A	B	C	D	E	F	G	H	I	J	K
1	Overdetermined system				{=MMULT(A3:C7, G3:G5)-E3:E7}					=MAbs(I3:I7)	
2	A (5 x 3)				b		x		r		$\|r\|^2$
3	2	4	0		35		7		-1		2.646
4	1	2	1		18		5		1		
5	-1	1	-2		-8		2		2		
6	3	0	3		26				1		
7	5	0	-1		33				0		

Note that only the last equation is exactly satisfied (residual zero).

Remark. Many books warm us of using the normal equation for solving these problems. They point out that, generally, the transformed system is worse conditioned than the original system and so the numerical solution may be error prone. This is conceptually true in general. But we should not emphasize this aspect too much. We have seen that, for systems of low-to-moderate size this method gives reasonably good solution accuracy. This method is also quick and easy to apply. Another advantage of this method is that the transformed matrix is symmetric positive definite, and thus we can adopt several efficient algorithms (e.g. Cholesky decomposition) to solve the system. Last but not least, if the original matrix is integer, the normal matrix is still integer.

QR decomposition

Another way to resolve a rectangular linear system is performing the QR factorization of the matrix system. As know, the QR factorization can be applied also to a rectangular matrix. The transformation is:

$$\mathbf{A}\cdot x = b \quad\Rightarrow\quad \mathbf{Q}\cdot\mathbf{R}\cdot x = b \quad\Rightarrow\quad \mathbf{R}\cdot x = \mathbf{Q}^{-1}\cdot b \Rightarrow \quad \mathbf{R}\cdot x = \mathbf{Q}^{T}\cdot b$$

Remember that **R** is triangular and **Q** is orthogonal and unitary so $\mathbf{Q}^{-1} = \mathbf{Q}^{T}$

The given rectangular linear system is transformed into a triangular linear system that can be solved efficiently with the back-substitution algorithm. Let's see.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		**A** (5 x 3)				**Q** (5 x 3)			**R** (3 x 3)			b	$\mathbf{Q}^T$*b
2	2	4	0		0.316	0.815	-0.015	6.325	1.423	1.107		35	53.601
3	1	2	1		0.158	0.407	0.263	0	4.356	-0.362		18	21.057
4	-1	1	-2		-0.158	0.281	-0.467	0	0	3.694		-8	7.3876
5	3	0	3		0.474	-0.155	0.655	0	0	0		26	
6	5	0	-1		0.791	-0.258	-0.533	0	0	0		33	
7													
8					{=MQR(A2:C6)}			{=MMULT(TRANSPOSE(E2:G6),L2:L6)}					

Now the (3 x 3) triangular linear system can easily be solved

O	P	Q	R	S	T	U
	R (3 x 3)			$\mathbf{Q}^T$*b		x
6.325	1.423	1.107		53.601		7
0	4.356	-0.362		21.057		5
0	0	3.694		7.3876		2
		{=SysLinT(O2:Q4,S2:S4)}				

Note that we have only used the first 3 rows of the **R** matrix returned by the QR factorization algorithm in order to set the (3 x 3) system matrix.

This method is in general more accurate and stable than the normal equation for solving large linear systems. On the other hand, the QR method requires - for m >> n - about twice as much work as the normal equation. We note also that integer values are never conserved by the QR factorization

SVD and the pseudo-inverse matrix

The most general way for solving an overdetermined linear system is

$$\mathbf{A} \cdot x = b \quad \Rightarrow \quad x = \mathbf{A}^{+} \cdot b$$

The matrix $\mathbf{A}^{+}$ is called the "pseudo-inverse" of **A** and, for a square matrix, coincides with the inverse $\mathbf{A}^{-1}$. The pseudo-inverse always exists, whether or not matrix is square or has full rank. For a rectangular matrix **A**(n x m), it is defined as

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

We note that the normal matrix ($\mathbf{A}^{T}\mathbf{A}$) appears in this definition
We can avoid computing the normal matrix directly by using the singular value decomposition

$$\mathbf{A} \cdot x = b \quad \Rightarrow \quad \mathbf{U}\mathbf{D}\mathbf{V}^{T}x = b$$

where, setting p = min(n, m), **U** is a (n x p) orthogonal[5] matrix, **V** is an (m x p) orthogonal matrix and **D** is a (p x p) diagonal matrix. For semplicity assume here n > m. In that case **D** and **V** are both square with dimension (m x m).
Multiplying both sides by $\mathbf{U}^{T}$ and remembering that: $\mathbf{U}^{T}\mathbf{U} = \mathbf{I}$, we have.

$$\mathbf{U}^{T}\mathbf{U}\mathbf{D}\mathbf{V}^{T}x = \mathbf{U}^{T}b \quad \Rightarrow \quad \mathbf{D}\mathbf{V}^{T}x = \mathbf{U}^{T}b$$

The matrix $\mathbf{D}\mathbf{V}^{T}$ is square so, taking its inverse, we have.

$$\mathbf{D}\mathbf{V}^{T}x = \mathbf{U}^{T}b \quad \Rightarrow \quad x = \left(\mathbf{D}\mathbf{V}^{T}\right)^{-1}\mathbf{U}^{T}b \quad \Rightarrow \quad x = \left(\mathbf{V}^{T}\right)^{-1}\mathbf{D}^{-1}\mathbf{U}^{T}b$$

Because $\mathbf{V}^{T} = \mathbf{V}^{-1}$, we have finally:

$$x = \mathbf{V}\,\mathbf{D}^{-1}\mathbf{U}^{T}b \quad \Rightarrow \quad \mathrm{x} = \mathbf{A}^{+}b$$

Therefore, the pseudo-inverse can be computed by the following stable formula

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

In Matrix.xla this computation is performed by the function MPseudoinv

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		**A** (5 x 3)			b				**A**+ (3 x 5)				x
2	2	4	0		35		0.009	-0.01	-0.02	0.049	0.166		7
3	1	2	1		18		0.187	0.099	0.054	-0.02	-0.07		5
4	-1	1	-2		-8		-0	0.071	-0.13	0.177	-0.14		2
5	3	0	3		26		{=MPseudoinv(A2:C6)}						
6	5	0	-1		33								
7									{=MMULT(G2:K4,E2:E6)}				
8													

Note that the pseudo-inverse of a (5 x 3) matrix is a (3 x 5) matrix. The solution is the product of the pseudo-inverse and the vector **b**

[5] The terms orthogonal here implies the concept of *column-orthogonal*: A matrix **A** (n x m), with n ≥ m, having all its columns mutually orthogonal is called column-orthogonal matrix.

Underdetermined Linear System

If the matrix has less rows than columns, then the linear system is said to be underdetermined.

If the rank of the incomplete and the complete matrix is equal then there are infinite solutions **x** that satisfie the given system

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

In that case the matrix equations $\mathbf{A}\,\mathbf{x} = 0$ or $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ define an implicit *Linear Function* - also called *Linear Transformation* - between the vector spaces, that can be put in the following explicit form

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{d} \qquad (1)$$

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

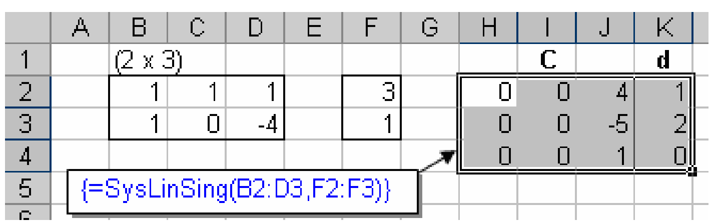
Example. Find the solutions (if any) of the following (2 x 3) system

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

The given rectangular system can be conceptually transformed into a square singular system simply adding a zero row (for example, at the bottom)

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -4 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \end{bmatrix} \quad \Leftrightarrow \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -4 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 0 \end{bmatrix}$$

The rank of this system is 2, therefore there are infinite solution that can be put in the form (1)
The solutions, in that case, can be easily find by hand or by SysLinSing



	A	B	C	D	E	F	G	H	I	J	K
1		(2 x 3)							C		d
2		1	1	1		3		0	0	4	1
3		1	0	-4		1		0	0	-5	2
4								0	0	1	0
5		{=SysLinSing(B2:D3,F2:F3)}									

Note that is not necessary to add the zero row because the function automatically does it.
The solutions can be written, after the substitution $x_3 = t$, as:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 4 \\ 0 & 0 & -5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ t \end{bmatrix} + \cdot \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \quad \Leftrightarrow \quad \begin{cases} y_1 = 4t + 1 \\ y_2 = -5t + 2 \\ y_3 = t \end{cases}$$

Note that the parametric form is not unique: substituting, for example the expression $x_3 = (t - 1)/4$, we get another parametric form representing the same subspace.

Example. Find the solutions (if any) of the following (3 x 4) system

$$\begin{cases} x_1 + 2x_2 + 3x_3 + 3x_4 = 4 \\ 2x_1 + 4x_2 + x_3 + 3x_4 = 5 \\ 3x_1 + 6x_2 + x_3 + 4x_4 = 7 \end{cases}$$

The rank r of the matrix **A** and the augmented matrix [**A**, **b**] are equal

	A	B	C	D	E	F	G	H	I	J
1		**A**			**b**					
2	1	2	2	3	4		2	=MRank(A2:D4)		
3	2	4	1	3	5					
4	3	6	1	4	7		2	=MRank(A2:E4)		

Therefore the system has surely infinite solutions. The matrix [**C**, **d**] returned by SysLinSing is

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
14			**A**				**b**				**C**			**d**
15		1	2	2	3		4			0	-2	0	-1	2
16		2	4	1	3		5			0	1	0	0	0
17		3	6	1	4		7			0	0	0	-1	1
18		0	0	0	0		0			0	0	0	1	0
19														
20		{=SysLinSing(B15:E18,G15:G18)}												

that can be written in parametric form, after the substitution $x_2 = t$, $x_4 = s$, as:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 & -2 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ t \\ x_3 \\ s \end{bmatrix} + \begin{bmatrix} 2 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \Leftrightarrow \quad \begin{cases} y_1 = -2t - s + 2 \\ y_2 = t \\ y_3 = -s \\ y_4 = s \end{cases}$$

Note that, as m = 4 , r = 2, the subspace generates by the solutions has dimension: m - r = 2; and therefore there are two parameters in the solution set.

Example. Find the solutions (if any) of the following (3 x 4) system

$$\begin{cases} x_1 + 2x_2 + 3x_3 + 3x_4 = 4 \\ 2x_1 + 4x_2 + x_3 + 3x_4 = 5 \\ 3x_1 + 6x_2 + x_3 + 4x_4 = 0 \end{cases}$$

	A	B	C	D	E	F	G	H	I	J
1		**A**			**b**					
2	1	2	2	3	4		2	=MRank(A2:D4)		
3	2	4	1	3	5					
4	3	6	1	4	0		3	=MRank(A2:E4)		

This system, apparently very similar to the above one, cannot be solved because the rank r of the matrix **A** and the augmented matrix [**A**, **b**] are different.
In this case SysLinSing would return "?"

Minimum module solution

As we have seen, an undetermined linear system have generally infinite solutions. We wander if, among the infinite solutions, there is one having the minimum module. For homogenous systems this solution surely exists because the trivial solution x = 0 has the minimum module.
The non-homogeneous case is more interesting. Recalling the example

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

we have found in a previous example that all its solutions can be represented by the following parametric equation.

$$x = [4t-1, -5t+2, t]^T$$

Computing the square module of **x**:

$$|x|^2 = x_1^2 + x_2^2 + x_3^2 = (4t-1)^2(-5t+2)^2 + t^2 = 42\,t^2 - 12\,t + 5$$

Taking its derivative, and solving, we have:

$$\frac{d}{dt}(42\,t^2 - 12\,t + 5) = 0 \qquad \Rightarrow \; 84t - 12 = 0 \qquad \Rightarrow \; t = \frac{1}{7}$$

Thus for t = 1/7 the solution x* = [11/7, 9/7, 1/7] has minimum module
For large systems this method becomes quite difficult, but we can obtain the minimal module solution of the undetermined system **A x** = **b** in a very quick way by the following matrix equation

$$\mathbf{x} = \mathbf{A}^{T}\,\mathbf{C}^{-1}\,\mathbf{b}$$

where the square matrix **C** = (**A**·**A**^{T})

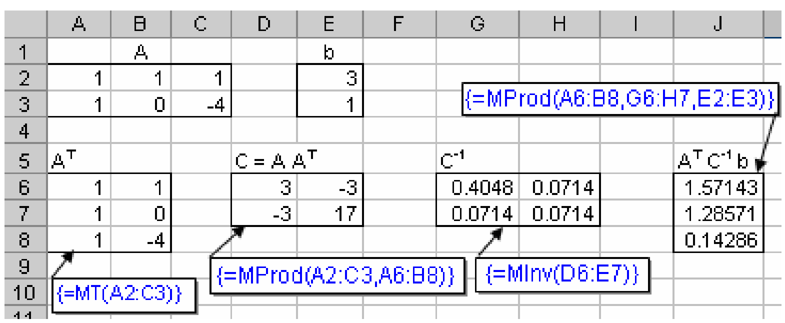
In the previous example we have

$$C = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -4 \end{bmatrix}^T \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -4 \end{bmatrix} = \begin{bmatrix} 3 & -3 \\ -3 & 17 \end{bmatrix} \quad \Rightarrow \quad C^{-1} = \frac{1}{42}\begin{bmatrix} 17 & 3 \\ 3 & 3 \end{bmatrix}$$

and the final solution is

$$x^* = \frac{1}{42}\begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 1 & -4 \end{bmatrix} \cdot \begin{bmatrix} 17 & 3 \\ 3 & 3 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 1 \end{bmatrix} = \frac{1}{7}\begin{bmatrix} 11 \\ 9 \\ 1 \end{bmatrix}$$

The following worksheet shows a possible solution arrangement



	A	B	C	D	E	F	G	H	I	J
1		A			b					
2	1	1	1		3					
3	1	0	-4		1					
4										
5	A^T			C = A A^T			C^{-1}			A^T C^{-1} b
6	1	1		3	-3		0.4048	0.0714		1.57143
7	1	0		-3	17		0.0714	0.0714		1.28571
8	1	-4								0.14286
9										
10										

Parametric Linear System

Sometimes the system matrix may contain a parameter, for example "k", and we may have to study the system solutions as a function of this parameter.
Generally speaking this is not a truly numerical problem, and the matrix cannot be inverted or factorized with the usual numerical methods. This problem can be solved using symbolic computational systems or, alternatively, by hand.
The function MDetPar in Matrix.xla computes the parametric determinant for matrices of low dimension. It returns the determinant D(k) as a polynomial in the variable k.
Then, with the aid of the Cramer's rule, we can obtain the solutions of the parametric system in the form of polynomials fractions.

Cramer's rule

Given a linear system: **[A] x = b** ⇔ [$\mathbf{a}_1$, $\mathbf{a}_2$, ... $\mathbf{a}_n$] **x** = **b**
The single element x_i of the solution vector **x**, can be found taking the fraction of the determinants of two matrices: the first matrix is obtained from the system matrix replacing the column $\mathbf{a}_i$ with the vector **b**; the second matrix is the system matrix itself.
That is, in formulas:

$$D_i = \det[\mathbf{a}_1, \mathbf{a}_2 ... \mathbf{a}_{i-1}, \mathbf{b}, \mathbf{a}_{i+1} ... \mathbf{a}_n]$$

$$D = \det[\mathbf{a}_1, \mathbf{a}_2 ... \mathbf{a}_{i-1}, \mathbf{a}_i, \mathbf{a}_{i+1} ... \mathbf{a}_n]$$

$$x_i = D_i / D$$

Repeating for i = 1, 2...n, we find the solution vector.
Example. Solve the following system containing the real parameter k

$$\begin{cases} k \cdot x_1 + 2x_2 + x_3 = 7 \\ 5x_1 + x_2 - k \cdot x_3 = 7 \\ 3x_1 + k \cdot x_2 + 3x_3 = 12 \end{cases} \Leftrightarrow \begin{bmatrix} k & 2 & 1 \\ 5 & 1 & -k \\ 3 & k & 3 \end{bmatrix} \quad \begin{bmatrix} 7 \\ 7 \\ 12 \end{bmatrix}$$

For the first, we build the 4 matrices and compute theirs determinants

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
6																
7		k	2	1		7	2	1		k	7	1		k	2	7
8		5	1	-k		7	1	-k		5	7	-k		5	1	7
9		3	k	3		12	k	3		3	12	3		3	k	12
10																
11		D =	-33+2k+k^3			D1 =	-33-17k+7k^2			D2 =	-66+12k^2			D3 =	-99+47k-7k^2	
12																
13		=MDetPar(B7:D9)				=MDetPar(F7:H9)				=MDetPar(J7:L9)				=MDetPar(N7:P9)		

The zeros of the determinant

D = $-33+2k+k^3$

can be found by the PolyRoots function.
Two roots are complex, and one is real: k = 3.

	A	B	C	D
11		D =	-33+2k+k^3	
12		{PolyRoots(C11)}		
13		3	0	
14		-1.5	-2.96	
15		-1.5	2.958	

Thus the system has solutions for k ≠ 3, that are:

$$x_1 = \frac{-33-17k+7k^2}{-33+2k+k^3}, \quad x_2 = \frac{-66+12k^2}{-33+2k+k^3} \quad x_3 = \frac{-99+47k-7k^2}{-33+2k+k^3}$$

Block-Triangular Form

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

$$\begin{bmatrix} 1 & 2 & 1 & 0 & 0 & 0 \\ 2 & 1 & 5 & 0 & 0 & 0 \\ 1 & -1 & 3 & 0 & 0 & 0 \\ -6 & 5 & 3 & 1 & 1 & 2 \\ 1 & -3 & 2 & 1 & -1 & -2 \\ -9 & 7 & 1 & 1 & 2 & 1 \end{bmatrix} \qquad \begin{bmatrix} \mathbf{A_1} & \mathbf{0} \\ \mathbf{A_{21}} & \mathbf{A_2} \end{bmatrix}$$

The block-triangular form saves a lot of computational effort for many important problems of linear algebra: linear system, determinants, 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

A x = **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

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

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

Computing the determinant

Determinant computing also takes advantage of the block-triangular form
For example, the determinant of the following (6 x 6) matrix is given by the product of the determinants of the two (3 x 3) matrices $\mathbf{A}_1$ and $\mathbf{A}_2$.

1	2	1	0	0	0		
2	1	5	0	0	0		
1	-1	3	0	0	0	=	18
-6	5	3	1	1	2		
1	-3	2	1	-1	-2		
-9	7	1	1	2	1		

1	2	1					
2	1	5	=	3			
1	-1	3					
			1	1	2		
			1	-1	-2	=	6
			1	2	1		

Permutations

Differently form the other factorization algorithms (Gauss, LR, etc.), the block-triangular reduction uses only permutations of rows and columns. Formally a permutation can be treated as a similarity transformation. For example, given a (6 x 6) matrix, exchanging rows 2 and 5, followed by exchanging 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. Matrix multiplication is a very expensive task that should be avoided whenever possible; we use instead the direct exchange of rows and columns or, even better, the exchange of their indices.

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

Eigenvalue Problems

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

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

A

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

$\mathbf{A}_1$				$\mathbf{A}_2$			λ_1	λ_2
-15	0	-16					1	
10	2	11					2	
8	0	9					-7	
			3	0	-4			-1
			2	5	4			3
			-6	-6	-1			5

The set of eigenvalues of the (6 x 6) matrix **A** is the sum of the eigenvalue set of $\mathbf{A}_1$ [1 , 2 , -7] and the eigenvalue 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 schemes having blocks with mutually different dimensions. At last, all blocks can have

unitary dimension as in a triangular matrix.
Below are shown some examples of block-triangular matrices (blocks are yellow)

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

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

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

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

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

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

Remark. The effort of 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 it is 3; in the third matrix the dimension is 1, showing the best-effort reduction that would be possible.
On the contrary, the last two matrices give a quite poor effort reduction.

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 on the zero elements. So only sparse matrices could be block-partitioned. But this is not sufficient. It depends also on the configuration of the zeros in 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 have been developed in the past for solving these problems. A very popular one is the Flow-Graph method.

Matrix Flow-Graph

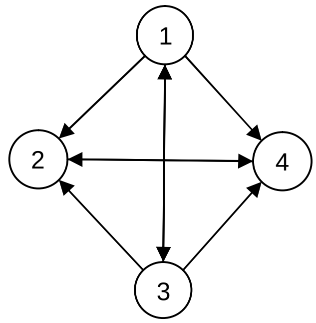
Following this method, we draw the graph of the given matrix following these simple rules:

- the graph consists of *nodes* and *branches*
- the number of 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 all 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 (4 x 4) matrix **A**

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:

node 1 is linked to nodes 2, 3, 4;
node 2 is linked to node 4;
node 3 is linked to nodes 1, 2, 4;
node 4 is linked to node 2.

We observe that from node 2 there is no path linking to node 1 or to node 3
The same happens if we start from node 4
It is sufficient to say that the graph is not strongly connected

Flow-Graph rule. If it is always possible for each node to find a path going through all other nodes, then we say that the graph is <u>strongly connected</u>

An important theorem of Graph Theory states that if the flow-graph G(**A**) is <u>strongly connected,</u> then the associated matrix is <u>not reducible</u> to block-triangular form, and vice versa.
On the contrary, if the flow-graph G(**A**) is <u>not strongly connected</u> then there always exists a permutation matrix **P** that reduces the associated matrix to block-triangular form. Synthetically:

G(**A**) strongly connected	$\Leftrightarrow$	matrix **A** irreducible
G(**A**) not strongly connected	$\Leftrightarrow$	matrix **A** block reducible

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

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

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

A

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 of searching for and testing all possible permutations grows sharply. For example, it requires much work for matrices like the following one. For this reason the flow-graph method becomes practically useless for matrices of dimension (7 x 7) or higher

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 can be applied also to medium-to-large matrices. It consists of an iterative process having as its main goal to group zeros near the upper-right corner of the matrix using only rows and columns exchanges.
This algorithm was first implemented as an automatic program, but thanks to its simplicity it can also

be performed by hand, at least, for low-to-moderately dimensioned matrices.
Let's see how it works

Given, e.g., the (6 x 6) matrix shown just above, we begin by initializing the permutation vector

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

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

The main goal is to bring to the upper triangular (grey) area) the largest possible number of zeros.
Let's begin to search all non-zero elements above 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 neighboring element a_{15} and so on till we have reached 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 non-zero element is a_{15} ;
Let's find, if exists, the first zero on the same row, beginning from left to right.
The first 0 is the element a_{12}. We shall exchange columns 2 and 5 and, thereafter, rows 2 and 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

P

1	5	3	4	2	6
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 the right is a_{14}. The first 0, starting from left, is a_{13}.
Thus we permute 3 and 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

P

1	2	4	3	5	6
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 a (6 x 6) matrix to block-triangular form. We have to emphasize that we worked only by hand. This method also keeps a good efficiency with larger matrices.

Let's have a look at 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 ≠ 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: a_{14}
The first element = 0, from left, is: a_{13}.
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: a_{13}
The first element = 0, from left, is: a_{21}.
So the pivot columns are 1 and 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 resulting 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 strongly connected.

The Score Function

The matrices used up to now had all zero elements completely filled moved into the upper-triangle area. Now let's see what happens if the matrix has more zeros than those strictly necessary for block partitioning (spurious zeros). In that case not all permutations will be useful for grouping zeros. Some of them will be useless, and some others even worse. Thus, it is necessary to measure the goodness of each permutation. By simple inspection it is easy to select the "good" permutations from "bad" permutations. But in an 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 (B) the permutation, 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 it will be negative or null.

x					
x	x				
x	x	x			
x	x	x	x		
x	x	x	x	x	
x	x	x	x	x	x

The zeros do not all have the same weight: the zeros nearest to the upper-right corner have a higher weight, because a matrix filled with zeros close to the upper-right corner is better than 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 from this concept, the weight function *w*(i,j) is arbitrary. One function that we have tested with good result is the following

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

Weight function for an (n x n) matrix
.

For each recognized permutation, the algorithm measures the score. If positive, the permutation is performed, otherwise the permutation is rejected and the algorithm continues to find a new permutation. After some loops the disposition of zeros 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.

Some 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

$P^TA\ P$

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 the fact that the permutation is much faster then any other arithmetic operation in floating point, we can guess the high speed of this algorithm

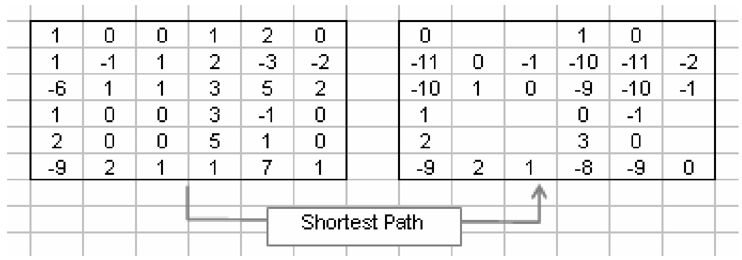
In Excel, with Matrix.xla, it is very easy to study the matrix permutations.
A simple arrangement of (6 x 6) matrices is shown in the following example. We have used the function MPerm . When you change the permutation numbers, also the permutation matrix changes and, consequently the final, transformed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

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

The Shortest Path algorithm

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

Example. Say if the given matrix is reducible

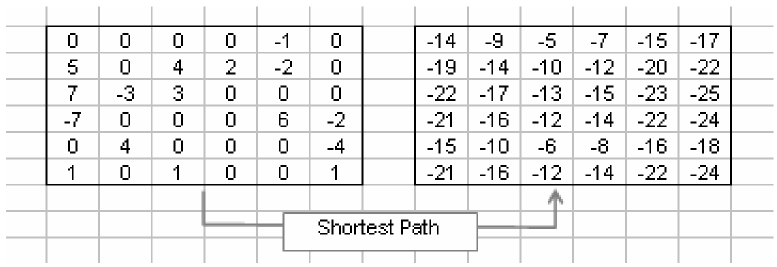


1	0	0	1	2	0
1	-1	1	2	-3	-2
-6	1	1	3	5	2
1	0	0	3	-1	0
2	0	0	5	1	0
-9	2	1	1	7	1

0			1	0	
-11	0	-1	-10	-11	-2
-10	1	0	-9	-10	-1
1			0	-1	
2			3	0	
-9	2	1	-8	-9	0

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 node 2 from node 1. This is sufficient for saying that the given matrix is not strongly connected and thus, reducible.

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



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

-14	-9	-5	-7	-15	-17
-19	-14	-10	-12	-20	-22
-22	-17	-13	-15	-23	-25
-21	-16	-12	-14	-22	-24
-15	-10	-6	-8	-16	-18
-21	-16	-12	-14	-22	-24

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

Limits in matrix computation

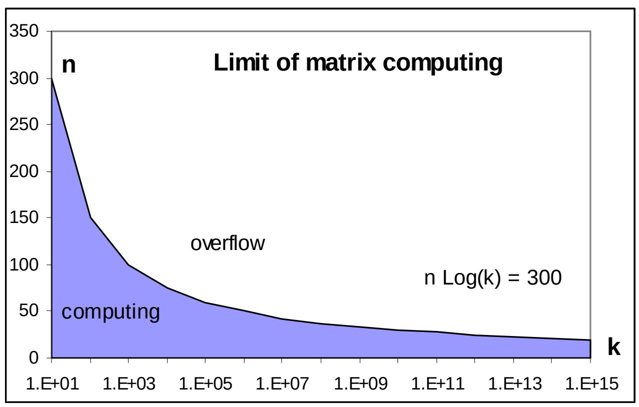
One recurrent question about matrix computation is: - what is the maximum dimension for a matrix operation, for example for the determinant, or for inversion?
Well, the right answer should be: it depends. Many factors, such as hardware configuration, algorithm, software code, operating system and - of course - the matrix itself, contribute to limit the maximum dimension. One sure thing is that the limit is not fixed at all.
In the past, the main limitation was memory and evaluation speed, but nowadays these factors no longer constitute 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:

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

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:

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

If we plot this relation for all points (k, n) we have the area for computing (blue area in the graph below). On the other hand, the dangerous error area is the remaining (white) area



How does it work?
Simple. If you have to compute the determinant of a (80 x 80) matrix having values no larger 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 (80 x 80) matrix having values up to 1E+7, the point (1E+7, 80) falls within the white area; so you will probably get an overflow error
From this graph we see that matrices of dimension (25 x 25) or less, can be evaluated for all values, while matrices of size (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 that are not ill-conditioned. If the matrix is ill-conditioned you could get an overflow/underflow error even for low-to-/moderate matrix dimensions. Fortunately, there are also special kinds of matrices that can be evaluated even if the constraint (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, especially for large matrices, of the round-off errors. They are very tricky and difficult to detect. Sometime the result of inverting a large matrix is taken as valid even if it is completely wrong!

Sparse Linear Systems

We have seen that finite arithmetic and memory storage both limit the maximum dimension of the matrix, and thus the associated linear system. In pre-2007 Excel, for example, the absolute maximum dimension for a linear system would be about (250 x 250). This limitation is due to the maximum number of the spreadsheet columns. But rarely we can solve such large systems because with 15 digits finite arithmetic the round-off errors often overwhelms the results.
There is a situation that allows one to successfully solve larger systems, of dimension greater than 250. It happens when the systems matrix is sparse. A system of linear equations is called sparse if only relatively few of its matrix elements [a_{ij}] are nonzero. If we store only these values, we can save a large amount of storage. For example, a (300 x 300) matrix with 10% nonzero elements requires only 9,000 cells of storage, just about the same as a dense (95 x 95) matrix.
Of course we have to choose a new arrangement to store these values. In the past, several ingenious and efficient schemes, tightly related to the hardware/software of the machine, were developed for this purpose. Here we adopt the *sparse coordinate format (or Yale scheme)*
This scheme is surely not one of the most efficientones, but it is conceptually simple, compact and adaptable to a spreadsheet implementation.

Specifically, the first 2 columns contain the integer coordinates while the last column contains the element values.
The sparse matrix of the previous example requires 9000 rows and 3 columns for a total of 27.000 cells.
We note that this array can easily be arranged in a spreadsheet while, on the contrary, its associated (300 x 300) standard matrix cannot be written, except with Excel 2007.

i	j	aij
1	1	a_{11}
1	2	a_{12}
1	3	a_{13}
2	1	a_{21}
…	…	…

The coordinate text format provides a simple and portable method to exchange sparse matrices. Any language or computer system that understands ASCII text can read this file format with a simple read loop. This makes these data accessible not only to users in the Fortran community, but also to developers using C, C++, Pascal, or Basic environments.

Filling factor and matrix dimension

The filling factor measures how much "dense" a matrix is. In this paper, the filling factor is defined as

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

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

There is a simple relation between the factor F and the maximum dimension of the system that can be solved in Excel. Remembering that the maximum number of rows of the pre-2007 spreadsheet are 2^{16}, we have

$$F \cdot N^2 \le 2^{16} \qquad \Rightarrow \qquad N \le \frac{2^8}{\sqrt{F}}$$

The corresponding limit in Excel 2007, with 2^{20} rows, is a factor of 100 larger.

The relation of N_{max} versus the filling factor shows that, for sparse matrices having

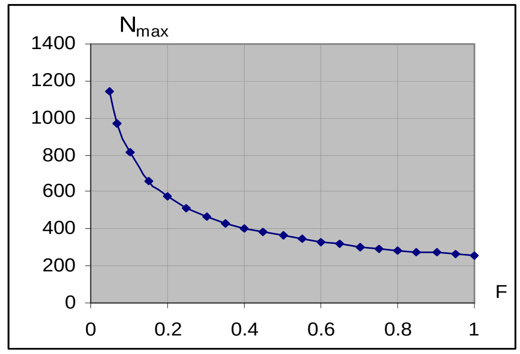
$0.1 < F < 0.4$

the max pre-2007 dimension of the system matrix is about

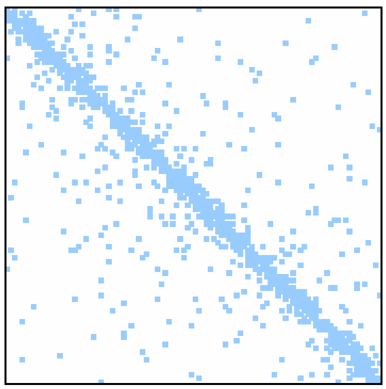
$400 < N_{max} < 800$

That is a great improvement with respect to the standard matrix format

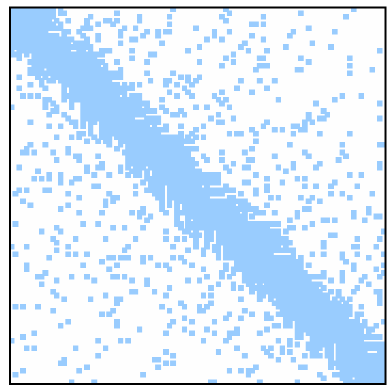
F	N
0.1	810
0.2	572
0.3	467
0.4	405
0.5	362



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



(100 x 100) F = 0.1



(100 x 100) F = 0.3

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

The dominance factor

Storing a matrix system does not automatically mean "solving" the system. As we have seen in the previous chapters, the round off errors may overwhelm the final result if the matrix is badly conditioned. For very large linear system the results can be acceptable only if the system matrix is well conditioned. It has been demonstrated that this happens for *row-diagonal dominant* matrices.
A matrix is called row diagonal dominant if each diagonal absolute element $|a_{ii}|$ is greater then the sum of the other absolute elements of the corresponding row. That is, in formula form:

$$|a_{ii}| > \sum_{j=1,\ j\neq i}^{n} |a_{ij}| \qquad \text{for } i = 1, 2 \ldots n$$

This criterion guarantees the convergence of iterative algorithms such as those of Gauss-Seidel and Jacobi. Moreover, it assures the complete Cholesky LL^T factorization, and a general good behavior against the propagation of round-off error. The row dominance criterion is sufficient but not necessary. That means that also non-dominant matrices may converge with a reasonable accuracy. On the other hand, there are matrices satisfying this criterion but in practice converging very slowly.
For these reasons it is convenient to define a *row dominance factor* measuring how much a matrix is "diagonal dominant". In this paper, it is defined, for a non-empty row, as

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

The row dominance factor D_i is always between 0 and 1

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

Therefore, the above criterion can be simply expressed as: $D_i > 0.5$ for i = 1, 2...n
With all due caution, we define the statistics D, D_m, D_M

$$D = \frac{1}{n}\sum_{i=1}^{n} D_i \qquad D_m = \min\{D_i\} \qquad D_M = \max\{D_i\}$$

These are a kind of matrix dominance factors summarizing the global dominance behaviors of the matrix itself. Note that D can be greater than 0.5 even if some rows are less than 0.5 or even 0.

Algorithms for sparse systems

Now we examine the algorithms suitable for solving large sparse systems: they can be direct and iterative algorithms.

Direct algorithm

Most direct system-solving algorithms operate a transformation on the system matrix and thus change the number of the zero elements. Unfortunately, none of these algorithm maintains the initial filling factor.

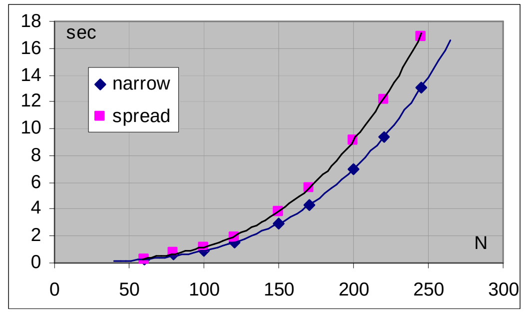
For example, starting with a (30 x 30) sparse matrix with F = 15 %, the average behavior of the most popular factorization algorithms are shows in this table. Clearly, we should give our preference to those algorithms that minimize the filling factor.

Algorithm	Final matrix
Gauss	F = 23%
LR	F = 46%
LL^T (Cholesky)	F = 23%
QR	F = 75%

The Gauss algorithm with partial pivot and back substitution still appears to be the right choice for a general system. For symmetric dominant systems, the Cholesky factorization is preferable for its efficiency
Those algorithms have a computational effort proportional to n^3, where n is the dimension of the linear system.
The following graph shows two typical factorization-time curves[6] performed by the Gauss algorithm for solving sparse linear systems having F = 20% with increasing dimension.

[6] Pentium 4, 1.8 GHz, 256 MB RAM

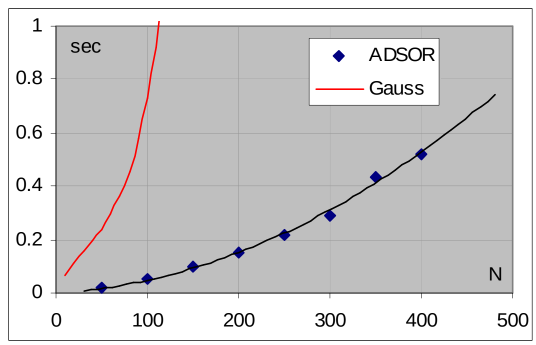
The time is measured in seconds.
The upper curve is obtained for sparse system matrices that are uniformly distributed, while the lowest curve is obtained for matrices concentrated around the first diagonal. As we can see, at the same dimension, the latter save more than 30% of the factorization time.
For symmetric sparse matrices the Cholesky factorization saves even more than 50%.



Iterative algorithms

But the truly strong reduction of effort is exhibited by iterative algorithms like the Successive Relaxation Gauss-Seidel algorithm or, better yet, the ADSOR method (Adaptive Successive Over-Relaxation).
When the system matrix is well-conditioned, for example for a diagonal dominant matrix, these methods converge to the solution with the best accuracy possible, in very few iterations, typically less then 100 steps. Unfortunately, not all sparse systems can be solved by an iterative procedure. But when they can, the time savings in factorization are remarkable
The following graph shows the factorization time of a direct method and an iterative method for diagonally dominant sparse linear systems (F = 20%) of increasing dimension

n	Time (sec) Gauss	Time (sec) ADSOR
50	0.22	0.02
100	0.6	0.05
150	1.7	0.09
200	4.2	0.15
250	8.6	0.22
300	15.6	0.31
350	25.7	0.41
400	39.6	0.53



We see that the factorization time remains less then one second even for very large systems. How can we justify this brilliant result? There are three facts:

1) Iterative algorithms operate in a very straightforward way, using only matrix-vector multiplications; for sparse matrices, this operation is very efficient, requiring only $F \cdot n^2$ elementary operations (multiplications + additions).

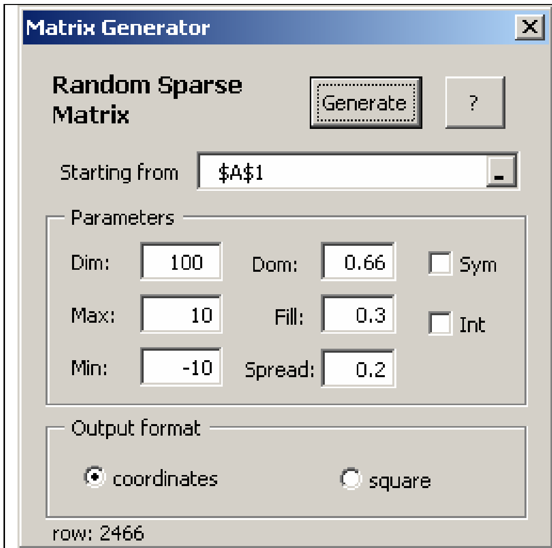
2) Iterative algorithms do no transform the system matrix, so its sparse factor F does not increase along the iterative process.

3) The number of steps Ns required for converging to a fixed precision is substantlially independent of the dimension; it mostly depends on the dominance factor of the matrix and, for the ADSOR algorithm, is usually less then 50-100.

The factorization time Ti of an iterative algorithm is proportional to the number of operations for each step, that is $Ti \approx Ns \cdot F \cdot n^2$. The elaboration time Td of a direct method is proportional to n^3 , i.e., $Td \approx n^3$.
Therefore the efficiency gain defined as G = Td / Ti will be: $G \approx n / (Ns \cdot F)$.
That gain is directly proportional to the dimension and inversely proportional to the filling factor. Example, for a real large sparse matrix of n = 400, with F = 20%, the gain G = 75.
The gain reaches more than 150 if F is less then 10%.

Sparse Matrix Generator

Of course, sparse matrices come from problems, and should not be generated. However sometimes we need to generate a sparse matrix for algorithm testing, time measuring, etc. On the internet there are some resources that can generate many type of matrices, including sparse matrices[7]. Matrix.xla also has a little tool for generating sparse matrices.



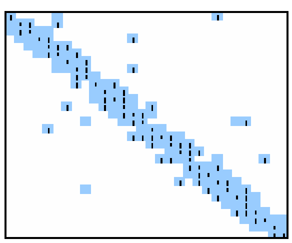
Parameters:
Random sparse matrix [aij] is generated with the following constraints:

Max: value: upper limit of aij
Min: value: lower limit of aij
Dim: matrix dimension (n x n)
Dom: Dominance factor D, with 0 < D < 1
Fill: Filling factor F, with 0 < F < 1
Spread: Spreading factor S, with 0 < S < 1
Sym: check it for symmetric matrix
Int: check it for integer matrix.
Starting from: left-top matrix corner

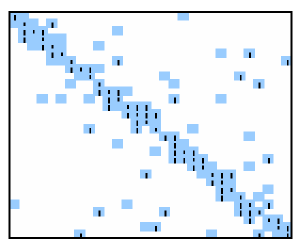
Output format
Coordinates: generates a (k x 3) matrix in sparse coordinate format: [i, j, aij]
Square: generates a square matrix [aij]

This macro can output a matrix in standard or coordinate format. Of course the coordinate format is the only possible one on pre-2007 Excel for matrices greater then (256 x 256).

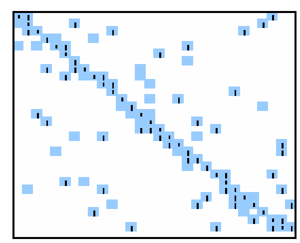
Here are some patterns generated for different parameters F and S



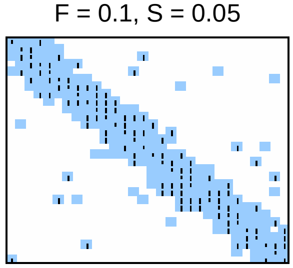
F = 0.1, S = 0.05



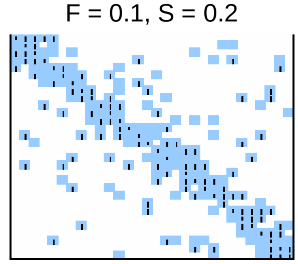
F = 0.1, S = 0.2



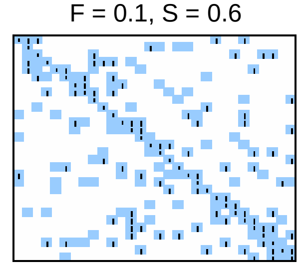
F = 0.1, S = 0.6



F = 0.3, S = 0.05



F = 0.3, S = 0.2

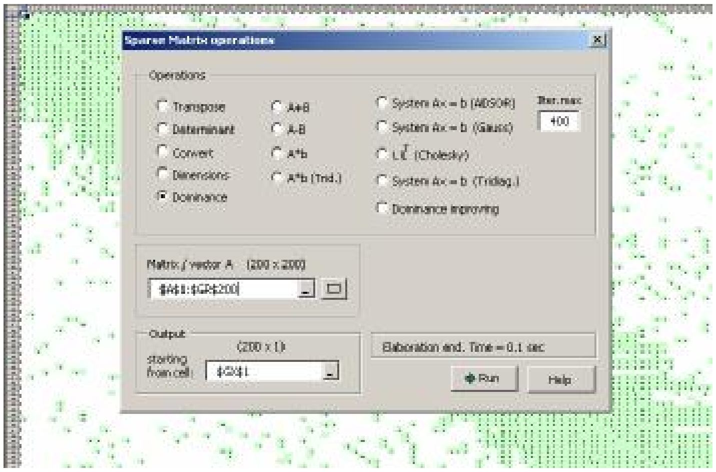


F = 0.3, S = 0.6

[7] NIST MatrixMarket has one of the most useful and complete tools, called "Deli", for generating a wide range of matrices with several output formats: http://math.nist.gov/MatrixMarket/deli/Random/

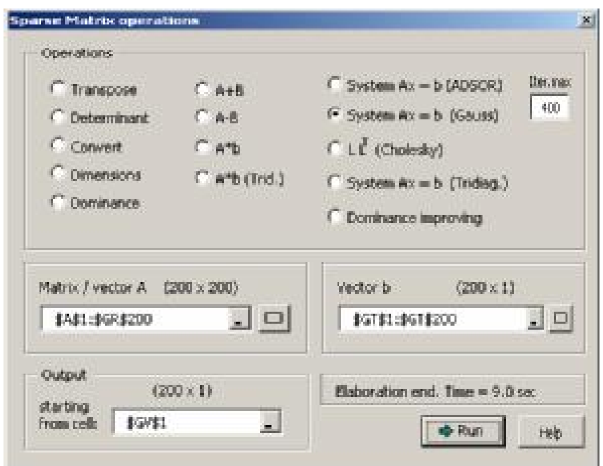
How to solve sparse linear systems

Assume that you have to solve a sparse (200 x 200) system **A** x = b, where the system matrix "**A**" is in the range A1: GR200 and the vector "b" is in the range GT1:GT200
First we analyze the dominance. Select one cell inside the matrix, for example A1; call the macro "Macros > Sparse matrix Operations..." from the menu, and select the operation "*Dominance*"



The macro returns the dominance factors of each row and the average, the max and the min of all dominance factors.
In this case we have obtained
$D_{min} > 0.5$ with an average of $D = 0.66$.
This indicates that the system is diagonally dominant and well-conditioned. We can use both iterative and direct methods

Select one cell inside the matrix, for example A1; call the macro "Sparse matrix Operations..." from the menu, and select the operation "*System (Gauss)*"



The input A matrix is already filled with the system matrix. Move the cursor inside the field "vector b" and select the range GT1:GT200.

Tip: You can select only the first cell GT1 and then click the smart selector at the right: the correct range will now be selected automatically. But make sure that the vector b is surrounded by empty cells.
Then choose the output range, and click "Run"

After a while (9 seconds in this example), the macro returns the solution vector of the system with a very high global relative accuracy (1E-14) .

Now we solve the same problem with the iterative algorithm ADSOR. The procedure is the same as above, except that we have to set the iteration limit (the default is 400).
This algorithm returns the vector solution and, in addition, the number of iterations performed, the average relative error, and the relaxation factor used. In this example, only 0.5 sec and 20 iterations are been necessary to reach an accuracy of about 3E-15.

As we can see the factorization time is much shorter than with the direct Gauss method. The Gauss method should be utilized only when the sparse matrix is not dominant, or the diagonal has some nonzero elements.

How to get the true dimensions

When the system is very large we necessarily have to adopt the coordinate format. For example, assume to have the matrix system **A** in the first three columns in the range A1:C14779. The coordinate format does not show directly the dimensions of the matrix. To avoid errors it is necessary to get the dimensions of a sparse matrix written in coordinate form, i.e (rows x columns).

For that, it is convenient to use the macro task "*Dimension*", which searches for the maximum number of rows and columns; in addition it returns the filling factor of the matrix itself

	A	B	C	D	E	F	G
1	1	1	51.883		Rows	Columns	F
2	1	2	-1.257		300	300	0.1642
3	1	3	1.035				
4	1	4	0.51				
5	1	5	-0.027				

How to analyze the dominance

Before solving a large system we have to analyze the conditioning of the system matrix in order to choose the algorithm and to understand if there is a chance of obtaining an acceptable result. If the matrix is diagonally dominant ($D_{min} > 0.5$), iterative algorithms converge to the solution. The dominance assures also an accurate result.

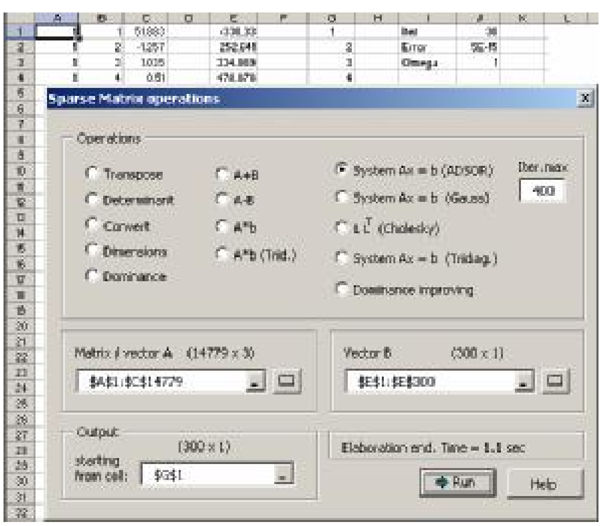
For that, use the macro task "*Dominance*", that computes the dominance factor Di of each ith row and, in addition, computes the statistics: average, max, and min.

A	B	C	D	E	F	G
1	1	51.883		0.6639	avg	0.6573
1	2	-1.257		0.6897	max	0.7241
1	3	1.035		0.7017	min	0.4832
1	4	0.51		0.6594		
1	5	-0.027		0.6479		

A matrix is totally row-dominant if $D_{min} > 0.5$.
In this example we have a $0.48 < D_{min} < 0.5$, so the matrix is not totally row-dominant.
Because $D_{min} > 0$, all rows have diagonal nonzero elements and this is the only necessary condition for using the iterative ADSOR method. The total dominance is a sufficient condition but it is not necessary; the ADSOR algorithm can often converge also for "quasi-dominant" matrices.

Solving Sparse System in coordinate format

Assume to have the system matrix in the range A1:C14779 and the vector "b" in the range D1:D300. Select one cell inside the matrix, for example A1. Call the macro "*Sparse matrix Operations...*" from the menu, and select the operation "*System (ADSOR)*"



The input A matrix is already filled with the system matrix A1:C14779. Move the cursor to the field "vector b" and select the range D1:D300.

The time for solving a (300 x 300) system is about 1 sec

The macro outputs the solution vector plus some useful information, such as the number of iterations, the estimated relative error, and the relaxation factor.

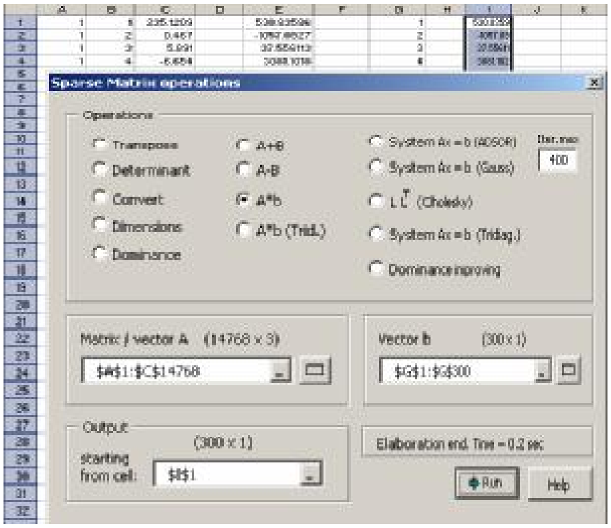
Note that the Gauss algorithm would need about 30 sec to solve this system
Note also that this system cannot be solved directly in pre-2007 Excel

How to check the result

A quick way for testing a linear system solution is to compute the residuals vector:

$$\mathbf{r} = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}$$

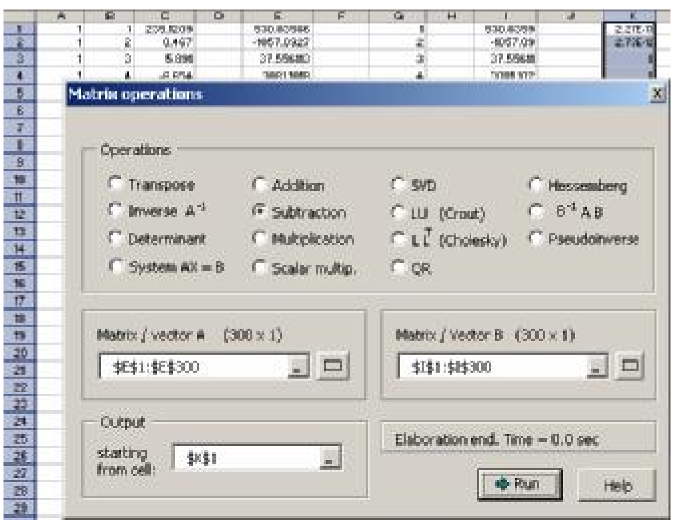
We have to point out that a low residuals vector does not automatically mean an accurate solution, but it is always a good and cheap test.



In the previous example we have the sparse matrix A in A1:C14779 , the b vector in D1:D300 and the **x** solution in G1:G300. First of all, we form the product **A·x** , putting the result in the range I1:I300. For this task we call the macro "Sparse matrix operations", selecting the product operation A*b.

The matrix-vector product is a very fast operation on sparse matrices.

After that, we compute the residual vector **r** as the difference between the **b** vector and the product **A*b**.



We can compute the difference between two vectors simply by selecting the range K1:K300 and inserting the array function {=E1:E300-I1:I300} with the ctrl+shift+enter keys sequence.

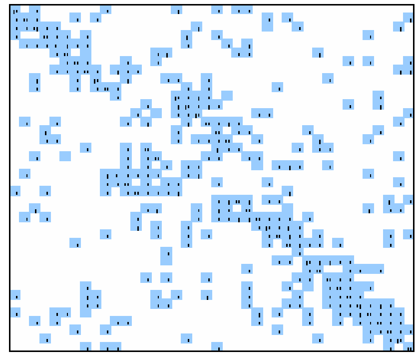
Or, alternatively, by using the macro "Matrix operations", selecting the "*subtraction*" task

The result is in the range K1:K300

The relative residual error can be computed as Erres = | **r** | / | **b** |
The norm can be computed with the MAbs function or with the Excel formula =SQRT(SUMSQ(K1:K300))

Solving Sparse System with Gauss

Often the linear system cannot be solved with the fast ADSOR algorithm. This happens, for example, when the system matrix has some zeros on the first diagonal, or has a low dominance factor. In these cases we have to go back to the Gauss reduction algorithm, adapted for sparse matrices



For example, assume to have a system with some diagonal zero elements.
The dominance factor analysis gives us the following factors

Davg =	0.295
Dmax =	0.4
Dmin =	0

The presence of zero diagonal elements is revealed by $D_{min} = 0$.

In that case we cannot adopt ADSOR and we have to use the Gauss algorithm.
Always remember to check the result because, in that case, the round-off error may completely obscure the solution obtained.

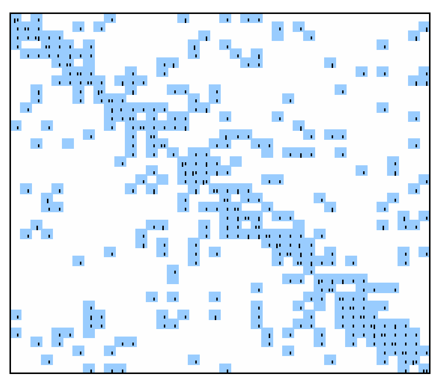
How to improve the dominance

In some cases the dominance of a linear system can be improved simply reordering the equations.
For example, the following system is not diagonal dominant

$$\begin{cases} 11x_1 + 3x_2 + 7x_3 = 22 \\ 4x_2 - 9x_3 + 15x_4 = -52 \\ -8x_1 + x_2 + 21x_3 + 2x_4 = 29 \\ x_1 + 20x_2 + 7x_3 - 4x_4 = 3 \end{cases}$$

11	3	7	0	22
0	0	-9	15	-48
-8	1	21	2	29
1	20	7	-4	3

But it becomes diagonal dominant simply exchanging the 2nd and 4th equations.
Of course for large system the manual rows exchanging is prohibitive. For this task comes useful the macro "*Dominance improving*". Starting from the system matrix **A** and the vector **b**, the macro tries to improve the dominance by rows exchanging and returns a new system matrix **A** and a new vector **b**.



Using the system of the above example we get the following new matrix. The dominance factors are now:

Davg =	0.4
Dmax =	0.45
Dmin =	0.33

As we can see the average dominance is improved but the best result is that no zero element appears in the diagonal (Dmin > 0).

That system can be efficiently solved with iterative algorithms.
Note that ADSOR con converge to the solution even if the system is not row-dominant

The relaxation parameter w

The convergence of iterative methods can be improved introducing a relaxation parameter ω Thus, the iteration schema, called SOR (Successive Over-Relaxation) can be modified as:

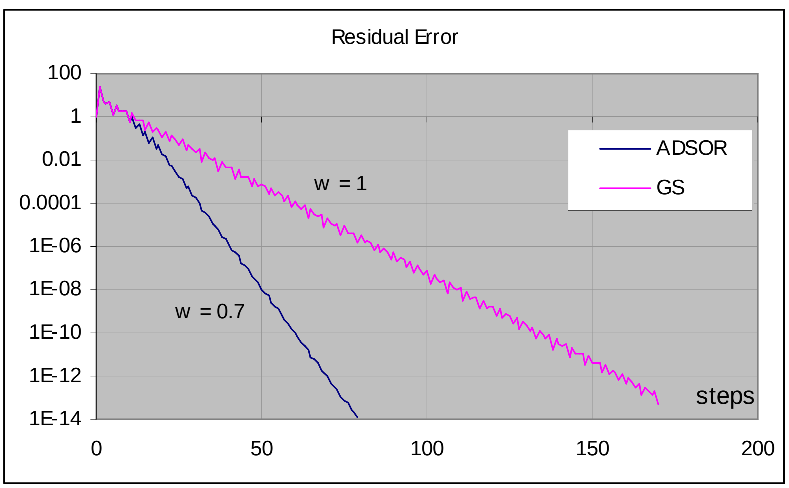
$$\mathbf{x}^{(k+1)} = (1 - \omega)\,\mathbf{x}^{(k)} + \omega\,\mathbf{x}_{GS}^{(k+1)}$$

where $\mathbf{x}_{GS}$ is the vector generated by the Gauss-Seidel algorithm. Usually is $0 < \omega < 2$. Generally, it is not simple to find the adaptive parameter for the fastest convergence.
In the ADSOR (ADaptive Successive Over-Relaxation) the parameter is chosen by the algorithm itself.

Example. Appling the ADSOR algorithm to the following system, we have the solution with an error of less then 1E-14, in about 80 iterations. We note also that this result is reached with the relaxation parameter $\omega = 0.7$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1															
2				A					b		x				
3		65	63	12	6	0	5		676		1		Iter	80	
4		-90	108	-80	6	0	-10		-940		2		Error	5E-15	
5		-89	-77	100	-1	-27	-10		-1364		3		Omega	0.7	
6		-8	0	45	45	-2	0		265		4				
7		-1	-9	0	-9	21	-1		301		21				
8		-6	0	0	100	-1	-92		-7447		85				
9															

If we repeat the calculation using the Gauss-Seidel algorithm ($\omega = 1$) we need about twice as many iterations.
The following graph shows the accelerating effect of the relaxation parameter



How to solve tridiagonal systems

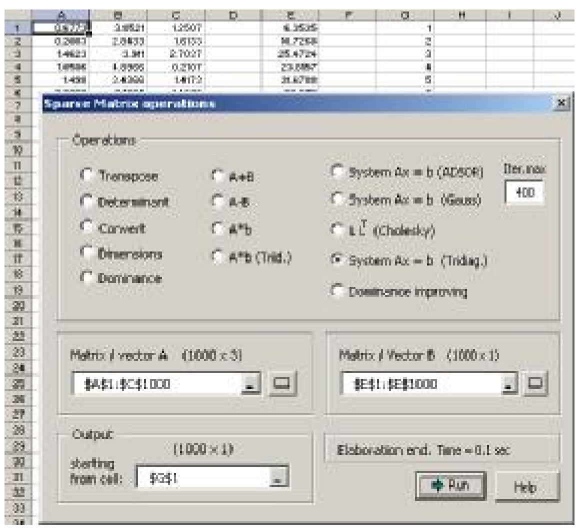
Tridiagonal systems are a subclass of sparse systems. Thanks to their particular structure they can be efficiently written in a very compact 3-column format.

- The first column contains the lower subdiagonal;
- The second column contains the diagonal
- The third column contains the upper subdiagonal

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
13																					
14		3	-1	0	0	0	0	0	0	0	0		1			0	3	-1		1	
15		-10	4	-1	0	0	0	0	0	0	0		-5			-10	4	-1		-5	
16		0	-6	4	2	0	0	0	0	0	0		8			-6	4	2		8	
17		0	0	3	9	10	0	0	0	0	0		95			3	9	10		95	
18		0	0	0	-7	-1	3	0	0	0	0		-15			-7	-1	3		-15	
19		0	0	0	0	-6	-8	-7	0	0	0		-127			-6	-8	-7		-127	
20		0	0	0	0	0	-7	10	-8	0	0		-36			-7	10	-8		-36	
21		0	0	0	0	0	0	-10	10	4	0		46			-10	10	4		46	
22		0	0	0	0	0	0	0	10	7	-1		133			10	7	-1		133	
23		0	0	0	0	0	0	0	0	-2	2		2			-2	2	0		2	
24																					

The space saving is evident. Note that the first element of the first column and the last element of the third column do not really exist. Usually they are set to zero, but their values are irrelevant because the macro does not read them.

Large linear tridiagonal systems can be solved efficiently using the macro "*Sparse Matrix Operation*"
In this example the system matrix is contained in A1:C1000, and the **b** vector is in E1:E1000
As we can see, only 0.1 sec is sufficient for solving a (1000x 1000) system. Usually the accuracy is very high for a dominant system.



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 of solving the following matrix equation

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

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

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

This homogeneous system can have non-trivial solutions if its determinant is zero. That is:

$$|A - l\ I| = 0 \qquad (3)$$

Characteristic Polynomial

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

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

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

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

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

For a (3x3) matrix, the system (2) becomes:

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

and its characteristic equation (3) becomes

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

With a larger matrix the difficulty of 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 with the function MCharPoly.

Roots of the characteristic polynomial

Apart from the 2nd degree case, finding the roots of a polynomial needs numerical approximation methods. Matrix.xla has the function **PolyRoots** that finds all roots - real or complex - of a given real polynomial, using the *Siljak+Ruffini* methods. This function is suitable for general polynomials up to 6^{th} or 7^{th} degree. When possible, the function uses the Ruffini method for finding small integer roots.

There is also the function **PolyRootsQR** for finding all polynomial roots. It uses the efficient QR algorithm and it is adapted for polynomials up to 10^{th} or 12^{th} degree.

For complex polynomials there is the similar function PolyRootsQRC

Case of symmetric matrix

Symmetric matrices play a fundamental role in numerical analysis. They have a feature of great importance: <u>Their eigenvalues are all real</u>. Or, in other words, its characteristic polynomial has only <u>real roots</u>. Another important reason for using symmetric matrices is that there are many straightforward, efficient, and also accurate algorithms for solving their eigen-systems; this is much more complicated for asymmetric matrices.

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

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

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

Below are the first such 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* - states 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 by substituting the variable λ with the matrix **A**. Let's see how to test this statement with a practical example in Excel.
Given the following (3 x 3) matrix

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

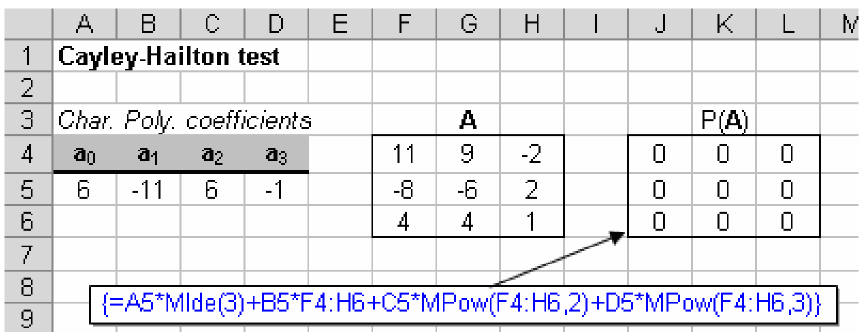
Its characteristic polynomial is:

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

After substituting **A** for λ 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 MPow

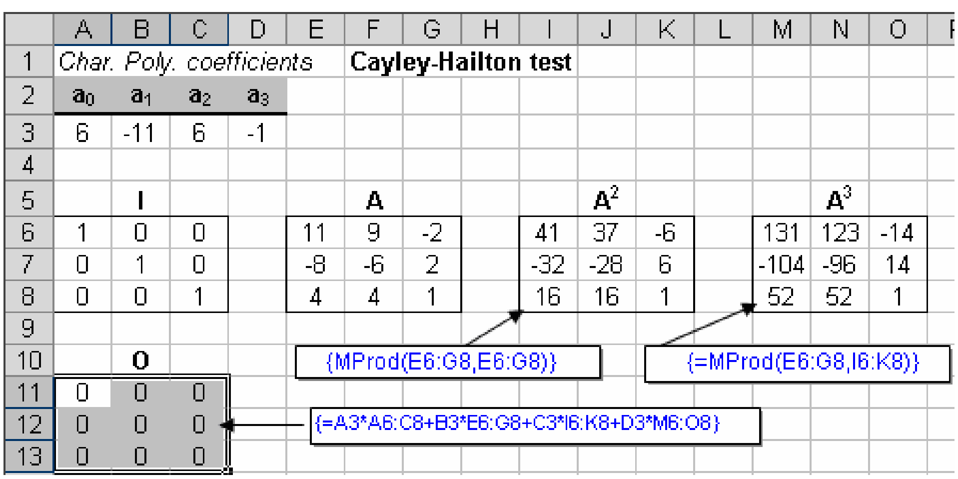


	A	B	C	D	E	F	G	H	I	J	K	L
1	**Cayley-Hailton test**											
2												
3	*Char. Poly. coefficients*						**A**				P(**A**)	
4	a_0	a_1	a_2	a_3		11	9	-2		0	0	0
5	6	-11	6	-1		-8	-6	2		0	0	0
6						4	4	1		0	0	0

{=A5*MIde(3)+B5*F4:H6+C5*MPow(F4:H6,2)+D5*MPow(F4:H6,3)}

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

Of course it is also possible to compute the matrix powers $\mathbf{A}^2$, $\mathbf{A}^3$ with the matrix product.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	*Char. Poly. coefficients*					**Cayley-Hailton test**									
2	a_0	a_1	a_2	a_3											
3	6	-11	6	-1											
4															
5		**I**				**A**				$\mathbf{A}^2$				$\mathbf{A}^3$	
6	1	0	0		11	9	-2		41	37	-6		131	123	-14
7	0	1	0		-8	-6	2		-32	-28	6		-104	-96	14
8	0	0	1		4	4	1		16	16	1		52	52	1
9															
10		**O**													
11	0	0	0												
12	0	0	0												
13	0	0	0												

{MProd(E6:G8,E6:G8)}

{=MProd(E6:G8,I6:K8)}

{=A3*A6:C8+B3*E6:G8+C3*I6:K8+D3*M6:O8}

Eigenvectors

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

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

Normally for each real eigenvalues with multiplicity one, 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 to every math student, and it is very popular. For this reasons it is explained in this chapter, despite its intrinsic 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.

The step-by-step method, is composed of the following steps:

1. Compute the coefficients of the characteristic polynomial
2. Find their 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 associated eigenvector $\mathbf{x}_i$ by solving the homogeneous system

Let's see how it works with some examples

Example - Simple eigenvalues

Find all eigenvalues and associated eigenvectors of the following matrix

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

For task 1) we use the function MathCharPoly; for task 2) we use the function PolyRoots; task 3) is performed with the MIde function which returns the identity matrix.Finally, task 4) uses the function SysLinSing to find a solution of the singular system.

	A	B	C	D	E	F	G	H	I	J	K
1	A				coeff	eigenvalues					
2	-4	14	-6		42	real	imm				
3	-8	19	-8		-41	2	0		{=PolyRoots(E2:E5)}		
4	-5	10	-3		12	3	0				
5					-1	7	0				
6	{=MCharPoly(A2:C4)}										
7											
8	A -λ I	for	λ = 2		A -λ I	for	λ = 3		A -λ I	for	λ = 7
9	-6	14	-6		-7	14	-6		-11	14	-6
10	-8	17	-8		-8	16	-8		-8	12	-8
11	-5	10	-5		-5	10	-6		-5	10	-10
12											
13	{=A2:C4-F3*MIde(3)}				{=A2:C4-F4*MIde(3)}				{=A2:C4-F5*MIde(3)}		
14											
15											
16	0	0	-1		0	2	0		0	0	2
17	0	0	-0		0	1	0		0	0	2
18	0	0	1		0	-0	0		0	0	1
19											
20	{=SysLinSing(A9:C11)}				{=SysLinSing(E9:G11)}				{=SysLinSing(I9:K11)}		

For the given matrix, we have found the eigenvalues and eigenvectors at the right

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

Eigenvalues	
$\lambda 1$	2
$\lambda 2$	3
$\lambda 3$	7

Example - How to check an eigenvector

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

$$u_i = A\,x_i \;\Rightarrow\; u_i = l_i\; x$$

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

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

Eigenvectors are not unique. It is easy to prove that any multiple of an eigenvector is also an eigenvector. This 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

By convention, mathematicians take the eigenvector with norm 1, that is: | **x** | =1.
In that case it is called the *eigenversor*.
Following this rule the eigenvector matrix becomes as we can see at right

	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/ MAbs(A51:A53)}					
56							

Sometimes, in order to avoid floating numbers, we normalize only the smallest value of the vector; for that, we divide all values by the GCD
The SysLinSing function adopts this solution. If you want to get the eigenversors you have to do it manually.

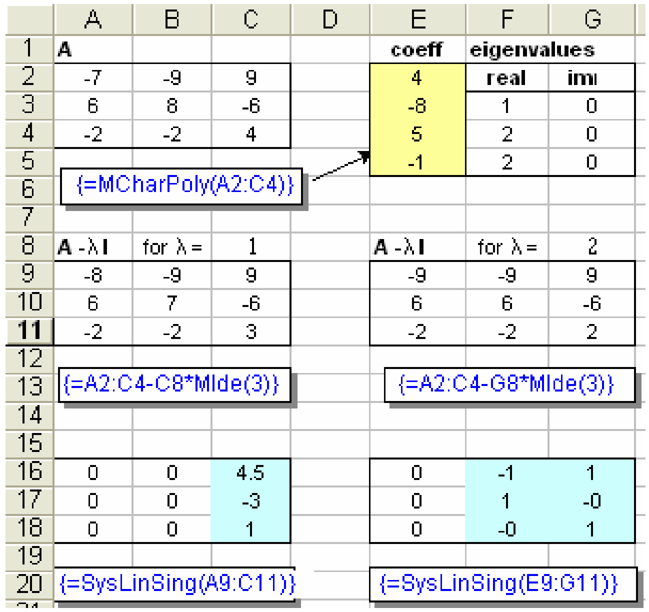
Example - Eigenvalues with multiplicity
Find all eigenvalues and associated eigenvectors of the following matrix

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

For the given matrix we have found two roots:

$$\lambda = 1 \;,\; m = 1$$
$$\lambda = 2 \;,\; m. = 2$$

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



	A	B	C	D	E	F	G
1	**A**				**coeff**	**eigenvalues**	
2	-7	-9	9		4	**real**	**imm**
3	6	8	-6		-8	1	0
4	-2	-2	4		5	2	0
5					-1	2	0
6	{=MCharPoly(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*MIde(3)}				{=A2:C4-G8*MIde(3)}		
14							
15							
16	0	0	4.5		0	-1	1
17	0	0	-3		0	1	-0
18	0	0	1		0	-0	1
19							
20	{=SysLinSing(A9:C11)}				{=SysLinSing(E9:G11)}		

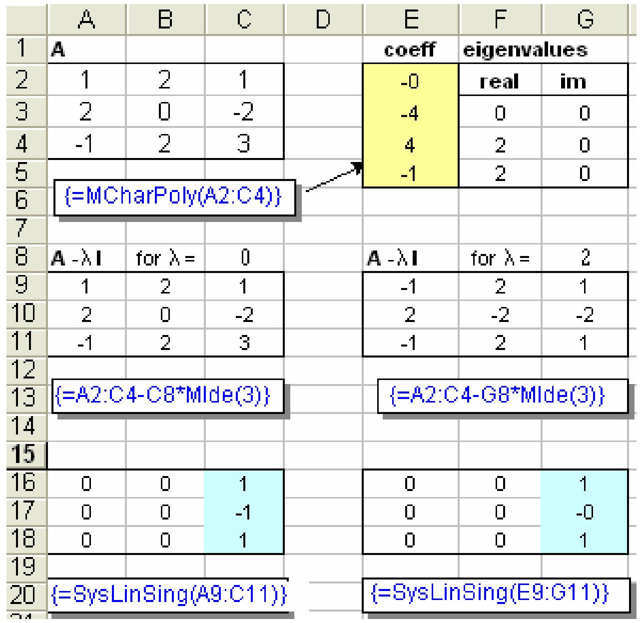
Tip: The accuracy of multiple roots is in general lower than that of a singular root. For this reason, the SysLinSing function sometimes cannot return any solution. In those cases, try to set the SysLinSing parameter MaxError to less then 1E-15, depending on the eigenvalue accuracy (usually for a root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

In the above example the number of eigenvectors corresponds exactly to the eigenvalue multiplicity. But this is always valid? Does the eigenvalu multiplicity gives the dimension of the eigenvector subspace? Unfortunately not. There are cases in which the multiplicity doesn't' t correspond to the associated eigenvectors.
Lets' see the following example.

Example - Eigenvalues with multiplicity not corresponding to the number of eigenvectors

Find all eigenvalues and associated eigenvectors of the following matrix

1	2	1
2	0	-2
-1	2	3



	A	B	C	D	E	F	G
1	A				coeff	eigenvalues	
2	1	2	1		-0	real	im
3	2	0	-2		-4	0	0
4	-1	2	3		4	2	0
5					-1	2	0
6	{=MCharPoly(A2:C4)}						
7							
8	A -λI	for λ =	0		A -λI	for λ =	2
9	1	2	1		-1	2	1
10	2	0	-2		2	-2	-2
11	-1	2	3		-1	2	1
12							
13	{=A2:C4-C8*MIde(3)}				{=A2:C4-G8*MIde(3)}		
14							
15							
16	0	0	1		0	0	1
17	0	0	-1		0	0	-0
18	0	0	1		0	0	1
19							
20	{=SysLinSing(A9:C11)}				{=SysLinSing(E9:G11)}		

For the given matrix the characteristic polynomial is:

$$-l^3 + 4l^2 - 4l$$

That has two roots:

λ = 0 , m = 1

λ = 2 , m. = 2

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

Example - Complex Eigenvalues

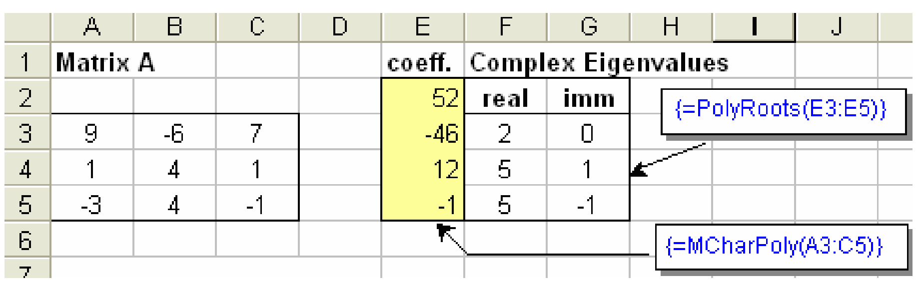
Sometimes it happens that not all roots of the characteristic polynomial are real. In that case, the eigenvectors associated with these complex eigenvalues are complex too.

Find all eigenvalues and associated eigenvectors of the following matrix

A =

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

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



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

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

Matrix.xla does not contain a SysLinSing for solving a complex singular system, but we can derive a real system from the original complex one:

Separating both eigenvalues and eigenvectors in their real and imaginary parts:

$$l = l_{re} + j\,l_{im} \qquad x = x_{re} + jx_{im}$$

the homogeneous linear system, becomes

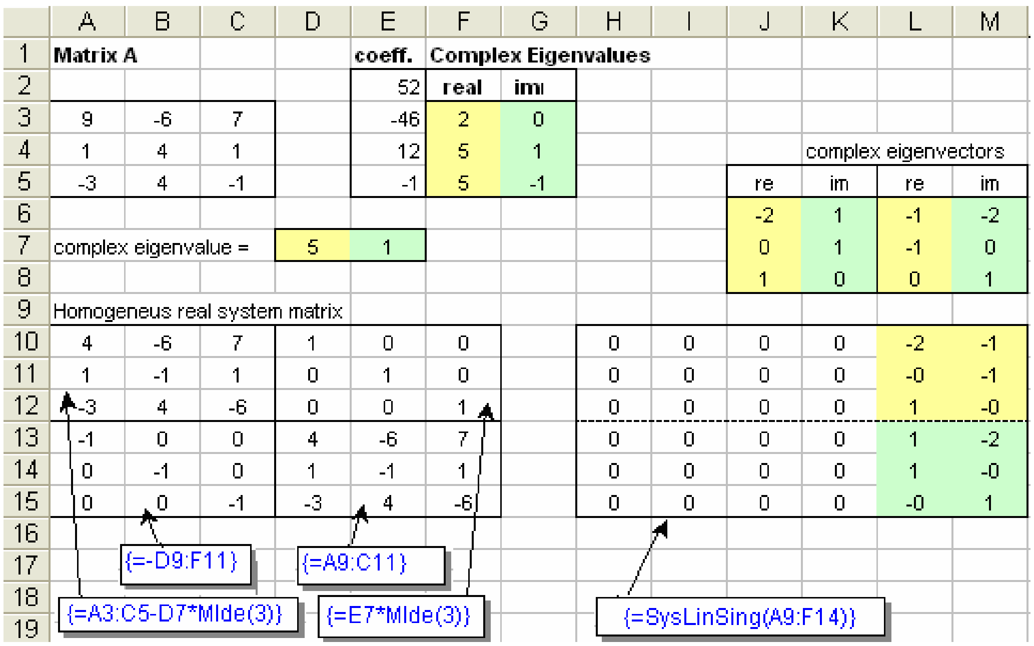
$$(A - lI)x = 0 \quad \Rightarrow \quad (A - (l_{re} + j\,l_{im})I)(x_{re} + jx_{im}) = 0$$

Rearranging: $$((A - l_{re}I)x_{re} + l_{im}I\,x_{im}) + j(-\,l_{im}I\,x_{re} + (A - l_{re}I)x_{im}) = 0$$

The above complex equation is equivalent to the following homogeneous system

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

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 conceptually divided in two parts: the upper part contains the real parts of the eigenvectors; the lower part holds the imaginary parts of the same eigenvectors.

Substituting the conjugated eigenvalues we find conjugated eigenvectors.

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

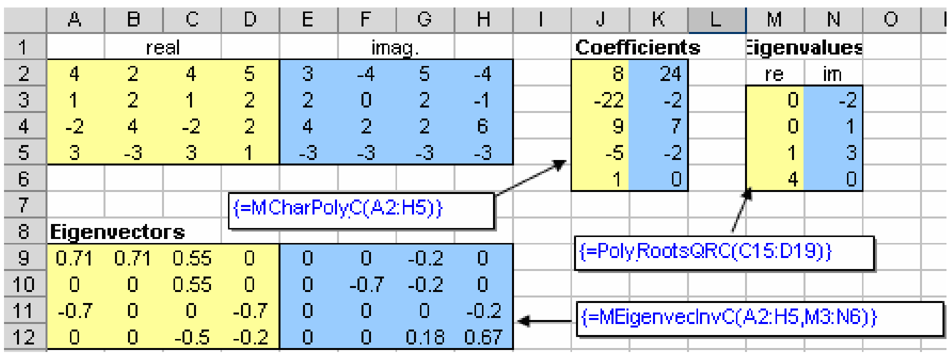
Example - Complex Matrix

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:

- MCharPolyC - computes the complex coefficient of the characteristic polynomial
- PolyRootsQRC - computes the roots of a complex polynomial
- MEigenvecInvC - 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.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		real				imag.				Coefficients			Eigenvalues		
2	4	2	4	5	3	-4	5	-4		8	24		re	im	
3	1	2	1	2	2	0	2	-1		-22	-2		0	-2	
4	-2	4	-2	2	4	2	2	6		9	7		0	1	
5	3	-3	3	1	-3	-3	-3	-3		-5	-2		1	3	
6										1	0		4	0	
7				{=MCharPolyC(A2:H5)}											
8	Eigenvectors														
9	0.71	0.71	0.55	0	0	0	-0.2	0		{=PolyRootsQRC(C15:D19)}					
10	0	0	0.55	0	0	-0.7	-0.2	0							
11	-0.7	0	0	-0.7	0	0	0	-0.2		{=MEigenvecInvC(A2:H5,M3:N6)}					
12	0	0	-0.5	-0.2	0	0	0.18	0.67							

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

Example - How to check a complex eigenvector

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

$$\mathbf{A} = \begin{bmatrix} 9 & -6 & 7 \\ 1 & 4 & 1 \\ -3 & 4 & -1 \end{bmatrix} \qquad \lambda = 5+j$$

xre	xim
-1	-2
-1	0
0	1

The test can be arranged as in the following worksheet

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

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

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

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

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

There is also another way to directly compute the eigenvector of a given eigenvalue: the functions MEigenvec and MEigenvecC of Matrix.xla return the eigenvector associated with their eigenvalues; the first function works for real eigenvalues, and the second for complex eigenvalues. See the chapter "*Function Reference*" of Vol. 2 for details

In the following arrangement we have used MEigenvecC for calculating the associated eigenvectors, and MMultsC for obtaining the complex scalar product

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Matrix							Eigenvalue			Eigenvector			A*u			λ * u	
2	9	-6	7	0	0	0		5	1		-1	-2		-3	-11		-3	-11
3	1	4	1	0	0	0					-1	0		-5	-1		-5	-1
4	-3	4	-1	0	0	0					0	1		-1	5		-1	5

{=MEigenvecC(A2:F4,H2:I2)}

{=MMultSC(K2:L4,H2:I2)}

Of course the final result is equivalent

Similarity Transformation

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

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

We say: ***C** is the similarity transform 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 as those of **C**, for any matrix **B**

It can be easily demonstrated 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 we can use the function **MBAB** of Matrix.xla. But, of course we can use, the standard formula as well.

=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**			{=MBAB(A3:C5,E3:G5)}				**coeff**	**eigenvalues**	
8	30	real	im						30	real	im.
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 coefficients of the characteristic polynomial we have used the function MCharPoly

Factorization methods

The heart of many eigensystem routines is to perform a sequence of similarity transformations until the resulting matrix is nearly diagonal within a 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

$$D = \begin{bmatrix} l_1 & 0 & 0 \\ 0 & l_2 & 0 \\ 0 & 0 & l_3 \end{bmatrix}$$

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 criteria. 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 spoken about the general method for resolving 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. Unfortunately, this method can be used only for matrices with low dimensions. When the matrix size is larger than 3, this method becomes quite tedious, long, and inefficient.

To overcome this, many algorithms have been developed. Generally, they calculate all eigenvalues and eigenvectors by efficient iterative methods. The price is that those methods are not general but are specialized for particular types of matrix classes. Very efficient algorithms exist for the symmetric matrix class, but the same algorithms 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; their ranges of application are summarized in the following table

Method	Real eigensystem		Complex eigensystem	
	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, highly efficient algorithms for tridiagonal and Toeplitz matrices.

Jacobi transformation of symmetric matrix

For real symmetric matrices, Jacobi's method is convergent, and gives both eigenvalues and eigenvectors. It consists of a sequence of orthogonal similarity transformations, each of them – called a Jacobi rotation - is just a plane rotation that annihilates one of the off-diagonal elements.

Referring to the paragraph "Factorization methods", this method gives us two matrices:

D (eigenvalues) and **U** (eigenvectors), being:

$$\lim_{n\to\infty} A_{n-1} = \begin{bmatrix} l_1 & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & l_n \end{bmatrix} \qquad\qquad \lim_{n\to\infty} P_1 P_2 \dots P_{n-1} P_n = U$$

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

	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			{=MEigenvalJacobi(B2:F6)}							{=MEigenvecJacobi(B2:F6)}								

We note how clean this method is. Just plain and straightforward! By default, both functions use 100 iterations to reach this highly accurate result. Sometimes, for larger matrices, you may need to increase this limit, otherwise you may have to accept a lower precision.

Tip. Jacobi's algorithm returns eigenvalues in the main diagonal. If you like to extract them in a vector, the function MDiagExtr comes in handy.

	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	{=MDiagExtr(A18:E22)}						

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

Each step of Jacobi's rotation method makes zero the two highest off-diagonal values. At subsequent steps these zeros cannot be preserved, but the off diagonal elements 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
1	A						matrix at step:			1	
2	9	-26	-14	36	24		9	-26	-27	36	7.69
3	-26	14	-4	46	14		-26	14	-13	46	7.36
4	-14	-4	-6	-6	-54		-27	-13	45.6	-1.6	0
5	36	46	-6	19	-4		36	46	-1.6	19	-7
6	24	14	-54	-4	-11		7.69	7.36	0	-7	-63
7							{=MEigenvalJacobi(A2:E6, H1)}				
8											
9							matrix at step:			4	
10	Jacobi rotation method						37.4	-3.5	-18	6.92	0.89
11							-3.5	-62	-0	2.52	12.5
12							-18	-0	49.8	-10	-2.5
13							6.92	2.52	-10	62.6	-0
14							0.89	12.5	-2.5	-0	-63
15							{=MEigenvalJacobi(A2:E6, J9)}				

	L	M	N	O	P	Q
1		matrix at step:			2	
2		9	-44	-27	8.26	7.69
3		-44	-30	-8	0	10.2
4		-27	-8	45.6	-9.8	0
5		8.26	0	-9.8	62.6	-0
6		7.69	10.2	0	-0	-63
7		{=MEigenvalJacobi(A2:E6, P1)}				
8						
9		matrix at step:			5	
10		24.9	-2.8	0	-0.4	-0.7
11		-2.8	-62	2.02	2.52	12.5
12		-0	2.02	62.3	-13	-2.5
13		-0.4	2.52	-13	62.6	-0
14		-0.7	12.5	-2.5	-0	-63
15		{=MEigenvalJacobi(A2:E6, P9)}				

	R	S	T	U	V	W
1		matrix at step:			3	
2		37.4	-0	-18	6.92	0.89
3		-0	-58	-21	4.51	12.7
4		-18	-21	45.6	-9.8	0
5		6.92	4.51	-9.8	62.6	-0
6		0.89	12.7	0	-0	-63
7		{=MEigenvalJacobi(A2:E6, V1)}				
8						
9		matrix at step:			6	
10		24.9	-2.8	-0.3	-0.3	-0.7
11		-2.8	-62	3.21	0.38	12.5
12		-0.3	3.21	49.9	-0	-1.8
13		-0.3	0.38	0	75	1.74
14		-0.7	12.5	-1.8	1.74	-63
15		{=MEigenvalJacobi(A2:E6, V9)}				

For a symmetric matrix, convergence is always guaranteed. In our example, after 15 steps, we have an average diagonalization error of only 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							{=MEigenvalJacobi(A2:E6,H1)}				

Orthogonal matrices

The eigenvector 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:

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

In other words, the scalar product of a vector with itself must be 1; for any other vector it 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 int

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: 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 matrices, encountered in the previous chapters, have always |det| =1, but they are never orthogonal.

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

ProdScal

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 scalar cross product of each pair of columns with the help of the function ProdScal. But this will tedious for a large matrix. It is faster to use the identity **U U**^{T} = **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)						verify orthonormalization						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							{=MProd(A2:E6, MT(A2:E6))}						{=MMopUp(G2:K6)}				
8	1	-0															
9			=ProdScal($A2:$A6,B2:B6)														
10																	
11			=ProdScal($A2:$A6,A2:A6)														

Tip. Often, a matrix product generates round-off errors, as in this case. We can sweep them up with the function MMopUp

Eigenvalues with the QR factorization method

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

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

This factorization is always possible; you can easily perform such factorization in Matrix.xla with the function MQR .

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 then repeat steps 2 and 3

We have the iterative process, starting with A:

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

If the eigenvalues all have distinct absolute values:

$$|\lambda_1| > |\lambda_2| > |\lambda_3| > \ldots > |\lambda_n|$$

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 MQRiter 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 MQRiter to perform the first 10 steps of the QR algorithm. The convergence to the diagonal form is evident, and becomes closer 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		=MQRiter(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 MQRiter 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		=MQRiter(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

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 $\lambda_1 = 9$, $\lambda_2 = -9$, $\lambda_3 = 18$. Applying 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		=MEigenvalQR(A2:C4)				=MEigenvalJacobi(A2:C4)			
7									

In this simple case QR fails (we note the two -9 off-diagonal elements). It was not able to find the two opposite eigenvalues = ± 9 , but it has found only the 18 one. Note that, under the same conditions, the Jacobi algorithm finds all the eigenvalues, exactly.

Real and complex eigenvalues with the QR method

Starting from the simple QR method shown above, a more general QR algorithm was developed with important improvements - shifting for rapid convergence, Hessenberg reduction, etc. The result is a very robust and efficient general QR algorithm[8] that can find complex and real eigenvalues of any real matrix.

This task is performed by the function MEigenvalQR 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 MEigenvalQR to find all eigenvalues in a very straightforward way

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

{=MEigenvalQR(A1:H8)}

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

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

{=MatEigenvalue_QR(A2:F7)}

Note how clean, easy and fast is the eigenvalue computation, even in this case

[8] Matrix.xla uses the routines HQR and ELMHES derived from the Fortran 77 EISPACK library

Complex eigenvalues of a complex matrix with the QR method

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

Example. Find the eigenvalues of the following matrix

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

	A	B	C	D	E	F	G
1	real		im			eigenvalues	
2	5	-77	-14	-19		51	68
3	50	80	-4	-20		34	-34
4							
5							

{=MEigenvalQRC(A2:D3)}

L	M	N	O	P	Q
	Matrix			re	im
	5-14j	-77-19j		34	34
	50-4j	80-20j		51	-68

{=MEigenvalQRC(M2:N3,3)}

This function accepts also the compact rectangular input 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

$$A - l\ I$$

For this task the functions MCharC and MDetC are useful

	A	B	C	D	E	F	G	H	I	J
1										
2	2	-1	3	4	3	1		λ =	4	-2
3	14	11	-7	-2	-3	1				
4	-6	-3	11	-2	-1	7		Det =	0	0
5										
6										
7										

{=MDetC(MCharC(A2:F4,I2:J2))}

When the matrix size becomes larger, round-off errors may mask the final result, and the eigenvalue check may be not so easy and straightforward.

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

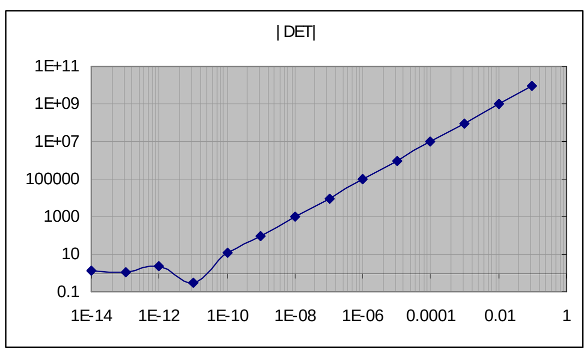
	A	B	C	D	E	F	G	H	I	J
1	**A**									
2	4569	-9128	-9136	-4556	-4484	9008	-9024	-4348	-9464	-9840
3	2004	-4003	-4016	-1996	-1960	3952	-3976	-1936	-4200	-4356
4	68	-136	-127	-76	-76	148	-128	-40	-124	-104
5	-556	1112	1112	569	552	-1112	1104	512	1144	1172
6	316	-632	-632	-316	-299	632	-624	-304	-648	-684
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(MChar(A2:J11,A14))						=MDetC(MChar(A2:J11,A14),TRUE)			

If we compute the determinant of the matrix $\mathbf{A} - \lambda\,\mathbf{I}$, 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, with the function MDet with the parameter IMode = True, we get the correct result

Note that, in general, we can have non-integer matrices or we can have non-integer eigenvalues, so we can not always use the trick of 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 corresponding absolute value of the 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



How to find polynomial roots with eigenvalues

In a previous example we have shown how to compute eigenvalues by polynomial roots. Sometimes the contrary happens: we have to find polynomial roots by eigenvalue 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 as 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 MCmp

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

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

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

	A	B	C	D	E	F	G	H	I	J
1	Poly coef			Companion matrix					Eigenvalues	
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		{=MCmpn(B2:B6)}				{=MEigenvalQR(D2:G5)}				
9										

Eigenvalues are also the roots of the given polynomial.

Rootfinder with QR algorithm for real and complex polynomials

The QR method is so robust and efficient that it is implemented in the rootfinder function PolyRootsQR and PolyRootsQRC 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			{=PolyRootsQR(B2:B13)}				{=PolyRootsQR(G2:G12)}		

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, that it has : good stability for all roots configurations and avoids the disastrous accuracy loss, characteristic of other rootfinding algorithms.

The function PolyRootsQRC works in a similar way 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
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		{=PolyRootsQRC(B3:C10)}				

The power method

The power method can find the dominant real eigenvalue - the eigenvalue that has the highest absolute value - and its associated eigenvector of a real matrix. This ancient method, still very popular, has some advantages:

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

With the matrix reduction method it can iteratively find all real eigenvalues and eigenvectors
But let us begin to understand the heart of the algorithm:
For the sake of simplicity we will assume a 3x3 matrix with 3 independent eigenvectors $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ and a dominant eigenvalue λ_1, i.e., $|\lambda_1| > |\lambda_2| > |\lambda_3|$. Take an arbitrary vector $\mathbf{v}_0$ - called the starting vector - and 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:

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

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

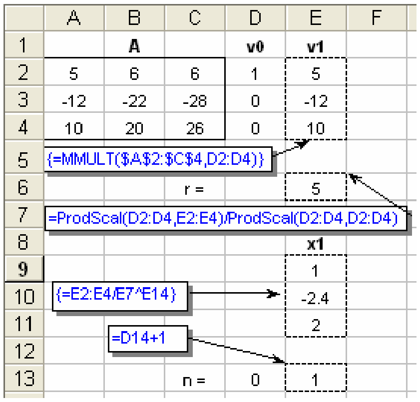
$$\lim_{n\to\infty} r = l_1 \qquad \Rightarrow \qquad \lim_{n\to\infty} v_n (l_1)^{-n} = x_1$$

We shall see how it works in a practical case
Example - Analyze the convergence of the power method for the following matrix

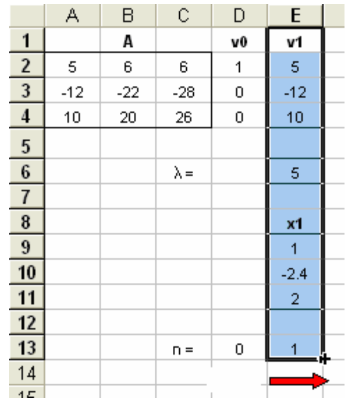
-1	2	-2
-2	-6	3
-2	-4	1

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

Let's see how to arrange the worksheet. First of all, insert the formulas as indicated to the left; then, select the appropriate range and drag it to the right to iterate the formulas.
Assume the starting vector to be $\mathbf{v}_0 = (1, 0, 0)$



Insert the formulas in column E



Select the range E1:E13 and drag it 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 associated 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 a higher number of steps. To avoid this, the algorithm is modified by inserting a vector-rescaling routine after a fixed number 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; the magnitude is the main thing.
Note also that the Rayleigh ratio is not affected by rescaling

Finding non-dominant eigenvalues. Once the dominant eigenvalue λ_1 and its associated eigenvector $\mathbf{x}_1$ are found, we may want to continue to compute the remaining eigenvalues.
Compute the normalized value 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 the eigenvalues: 0, λ_2, λ_3 . Now, the dominant eigenvalues of $\mathbf{A}_1$ is λ_2
Therefore we can apply the power method once more.

Example - reduce the matrix **A** of the previous example with the eigenvalue λ_1 = –3 and eigenvector $\mathbf{x}_1$ = (0, 1, 1). Repeat the power method to find the dominant eigenvector λ_2

	A	B	C	D	E	F	G	H	I	J	K
1		A				u u^T				A1	
2	-1	2	-2		0	0	0		-1	2	-2
3	-2	-6	3		0	0.5	0.5		-2	-4.5	4.5
4	-2	-4	1		0	0.5	0.5		-2	-2.5	2.5
5											
6	λ1	x1	u1								
7	-3	0	0								
8		1	0.707								
9		1	0.707								

{=MMULT(C7:C9), MT(C7:C9))}

{=B7:B9 / MAbs(B7:B9)}

{=A2:C4-A7*E2:G4}

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 λ_2 = –2 and its associated 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 method* + *matrix reduction* can be iterated for all eigenvalues. We have to realize that, since the computed eigenvalues are approximations, round-off errors will be introduced in the next iteration 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 accumulates as the process continues.

Does the power 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 it does not have n linearly independent eigenvectors. Simple, of course, but 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 the max iteration limit may have to be extended

Eigensystems with the power method

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

- MEigenvecPow returns all eigenvectors
- MatEigenvalues_pow returns all eigenvalues

Just simple and straightforward. Let's see

Example - solve the eigenproblem for the following symmetric matrix

	A	B	C	D	E	F	G	H	I	J	K
1	A (5 x 5)					eigenvectors (power)					eigenvalues (power)
2	2.6	1.2	0.8	0.4	0	-0.6667	-0.6667	-0.6667	-0.6667	1	5
3	1.2	2.8	0.4	0	-0.4	-0.6667	-0.6667	-0.6667	1	-0.6667	4
4	0.8	0.4	3	-0.4	-0.8	-0.6667	-0.6667	1	-0.6667	-0.6667	3
5	0.4	0	-0.4	3.2	-1.2	-0.6667	1	-0.6667	-0.6667	-0.6667	2
6	0	-0.4	-0.8	-1.2	3.4	1	-0.6667	-0.6667	-0.6667	-0.6667	1
7											
8				{=MEigenvecPow(A2:E6)}					{=MEigenvalPow(A2:E6)}		
9											

The function MEigenvecPow 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				{=MEigenvecPow(A2:E6,TRUE)}							{=MMULT(F2:J6,TRANSPOSE(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.

Example: solve the eigenproblem for the following asymmetric 6x6 matrix.

	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
							{=MEigenvecPow(A2:F7)}					{=MEigenvalPow(A2:F7)}		

This matrix has eigenvalues –1, 3, –6, 9, 12, –15

The power method works also for asymmetric matrices. In this example we have left the round-off errors to give an idea of the general accuracy. Eigenvalue errors are shown in the last column.

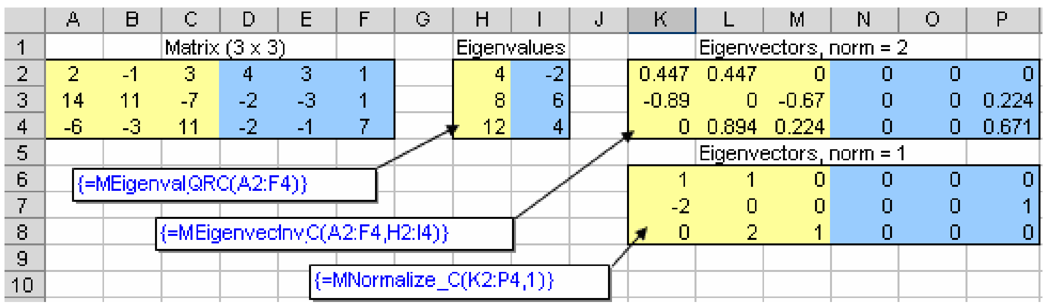
Complex Eigensystems

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
- MEigenvecInvC returns all distinct eigenvectors by inverse iteration
- MEigenvecC returns the eigenvectors of associated eigenvalues

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

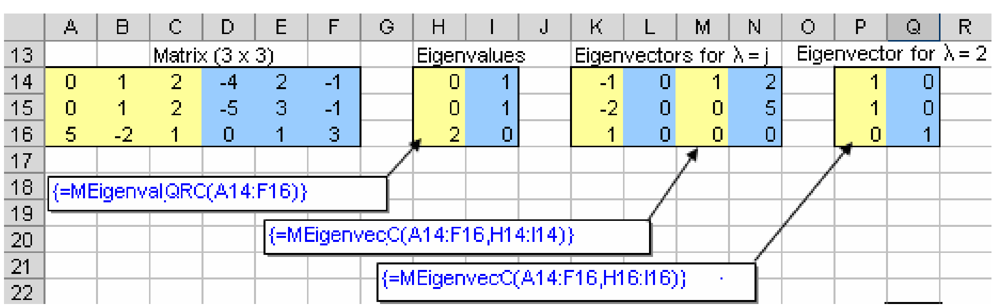


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1			Matrix (3 x 3)					Eigenvalues				Eigenvectors, norm = 2				
2	2	-1	3	4	3	1		4	-2		0.447	0.447	0	0	0	0
3	14	11	-7	-2	-3	1		8	6		-0.89	0	-0.67	0	0	0.224
4	-6	-3	11	-2	-1	7		12	4		0	0.894	0.224	0	0	0.671
5												Eigenvectors, norm = 1				
6	{=MEigenvalQRC(A2:F4)}										1	1	0	0	0	0
7											-2	0	0	0	0	1
8			{=MEigenvecInvC(A2:F4,H2:I4)}								0	2	1	0	0	0
9																
10						{=MNormalize_C(K2:P4,1)}										

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 MEigenvecInvC have always unit absolute magnitude (norm = 2). For changing the normalization type we can use the function MNormalizeC.

When the eigenvalues are not all distinct we cannot use the inverse iteration but instead should use the singular system method performed by the MEigenvecC
Example. The following matrix has only 2 distinct eigenvalues: 2, and j



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
13			Matrix (3 x 3)					Eigenvalues			Eigenvectors for λ = j				Eigenvector for λ = 2			
14	0	1	2	-4	2	-1		0	1		-1	0	1	2		1	0	
15	0	1	2	-5	3	-1		0	1		-2	0	0	5		1	0	
16	5	-2	1	0	1	3		2	0		1	0	0	0		0	1	
17																		
18	{=MEigenvalQRC(A14:F16)}																	
19																		
20					{=MEigenvecC(A14:F16,H14:I14)}													
21								{=MEigenvecC(A14:F16,H16:I16)}										
22																		

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

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

But, before testing, we show how to arrange the eigenvector matrix in order to avoid non-integers. This is not essential, but it helps the visual inspection.

First of all, we begin with eliminating round off error by using the function MMopUp

	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					{=MMopUp(A2:F7,1E-13)}							
10												

Now, for each column, we choose the *pivot*, that is, the absolute minimum value, except the zeros. Multipling each pivot by the corresponding eigenvector we obtain a new integer vector that it is still an eigenvector

	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**										{=H2:H7/H9}	
9	0.1429	-0.25	-0.5	1	1	0.0385	{=G2:G7/G9}					
10												

	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 by its eigenvector matrix: **A U**.

The matrix on the right is obtained by multiplying each eigenvector $\mathbf{u}_i$ for its corresponding eigenvalue.

Because the two matrices are identical, the eigensystem (eigenvectors + eigenvalues) is correct.

How to generate a random symmetric matrix with given eigenvalues

Many times, 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) vector, **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) Choose a random vector **v**, like for example:

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

Build the associated *Householder* matrix **H**

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

Set the diagonal matrix **D**

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

Perform the similarity transformation 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 as **H**.
The resulting matrix **A** has the wanted eigenvalues = (1, 2, 4)
If we want to avoid fractional numbers we can multiply the matrix **A** by 9 and 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 multiples of 9; thus 9, 18, 36

As we can see, this method is general, and can be very useful in many cases: for testing algorithms, formulas, subroutines, etc.

In the add-in Matrix.xla, there are functions for generating Householder matrices and performing the Similarity Transform.

	A	B	C	D	E	F	G	H	I	J	K	L
1	**Random Symmetric matrix with given eigenvalues**											
2	**V**			**D**			**H**				**A**	
3	2		1	0	0	-0.333	-0.667	0.667		2.778	0.222	1.111
4	1		0	2	0	-0.667	0.667	0.333		0.222	1.778	0.889
5	-1		0	0	4	0.667	0.333	0.667		1.111	0.889	2.444
6												
7	Seed: random vector			eigenvalues setting		{=MHouseholder(A3:A5)}						
8												
9										{=MBAB(C3:E5;F3:H5)}		

All these actions are performed by the function MRndEigSym

Eigenvalues of a tridiagonal matrix

Tridiagonal matrices are very common in practical numerical computation. These matrices can be handled with all methods shown before, but there are dedicated algorithms, more efficient and faster, to solve those specialized eigenvalues problem. We have to consider that many times a problem involving tridiagonal matrices has a quite large dimension. Also, the storage of a tridiagonal matrix should be considered. A general full 30 x 30 matrix requires 900 cells, but for a tridiagonal one with the same dimension we need to store only 90 cells, saving more than 90%. Clearly, paying particular attention to storage is quite important.

Matrix.xla contains the following specialized functions applicable to tridiagonal matrices:

- MEigenvalQL — finds all real eigenvalues with the QL algorithm
- MEigenvecT — computes the eigenvector of a real eigenvalue
- MatEigenvalTTpz — finds all eigenvalues for a toeplitz tridiagonal matrix

All these function accept the matrix either in standard (n x n) form or in compact (n x 3) form

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

For tridiagonal matrices there are several useful lemmas that help us to find the eigenvalues

One rule says that:

If all "perpendicular couples" of elements have the same sign, than the matrix has only 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	{=MEigenvalQL(A2:C16)}						{=MEigenvecT(A2:C16,E2:E5)}			
19										

Note that the eigenvectors returned by MEigenvecT are not normalized. Use for this task the MNormalize function.

Eigenvalues of a tridiagonal Toeplitz matrix)

In numeric calculus it is common to encounter symmetric, tridiagonal, toeplitz matrices like the following. For this kind, there is a nice close formula giving all eigenvalues for matrices of any 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:

$$l_k = a + 2b \cdot \cos\left(\frac{kp}{n+1}\right)$$

where k = 1, 2… n

We make the following observations:

- All eigenvalues are real and distinct when the matrix is symmetric
- All eigenvalues are symmetric around the point "a"
- For n odd there exists the trivial eigenvalue $\lambda = a$
- All roots lie inside the interval $a-2b < \lambda_\kappa < a+2b$

Also the eigenvector matrix can be written in a compact closed 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 the n x n dimension n x n, the elements of the eigenvectors matrix are:

$$u_{ik} = \sin\left(i \cdot k \frac{p}{n+1}\right)$$

where i = 1, 2… n , k = 1, 2… n

The unsymmetrical tridiagonal toeplitz 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 the dimension n x n, and bc > 0 , the eigenvalues are:

$$l_k = a + 2\sqrt{bc} \cdot \cos\left(\frac{kp}{n+1}\right)$$

where k = 1, 2… n

All roots lie within the interval:

$$a - 2\sqrt{bc} < l_k < a + 2\sqrt{bc}$$

2) The sub-diagonals have different sign. In that case we can demonstrate that all roots are complex conjugate for n even; for n odd there 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 the dimension n x n, and bc < 0 , the eigenvalues are complex:

$$l_k = a + i \cdot 2\sqrt{-bc} \cdot \cos\left(\frac{kp}{n+1}\right) = a + i d_k$$

where k = 1, 2… n

All roots lie inside the segment:

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

Eigenvectors can be computed by the following iterative algorithm

$$x_k = l_k - a$$

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

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 toeplitz 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 in the lower and upper triangles have the same signs, so that all eigenvalues are real and distinct.
They can be obtained by the following closed formula:

$$l_k = a + 2\sqrt{bc}\cdot\cos\left(\frac{kp}{n+1}\right)$$

for k = 1, 2, …8   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 within the interval (a – 4, a + 4) = (6, 14)

Example Find all eigenvalues of the following tridiagonal toeplitz 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 signs, and that the dimension n is odd, so that all eigenvalues are complex conjugate except one real, trivial root at $\lambda = 10$.
The eigenvalues can be obtained from the following closed formula:

$$l_k = a + i\cdot 2\sqrt{-bc}\cdot\cos\left(\frac{kp}{n+1}\right) = a + id_k$$

for k = 1, 2, …7   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 toeplitz 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 signs, and the dimension n is even, so that no real eigenvalues exist, and all eigenvalues are complex conjugate.
They can be obtained by the following closed formula:

$$l_k = a + i \cdot 2\sqrt{-bc} \cdot \cos\left(\frac{kp}{n+1}\right) = a + id_k$$

for k = 1, 2, … 8   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 toeplitz 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 closed formula:

$$l_k = a + 2b \cdot \cos\left(\frac{kp}{n+1}\right)$$

for k = 1, 2, …8   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 in the interval (a – 2, a + 2) = (–4, 0)
We observe that they are all negative
The eigenvector matrix can be obtained in a very fast way using the formula

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

That gives the following approximate eigenvector 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.

Generalized eigen problem

The 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, is called a generalized eigen problem.

Equivalent asymmetric 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 when **A** and **B** are.

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

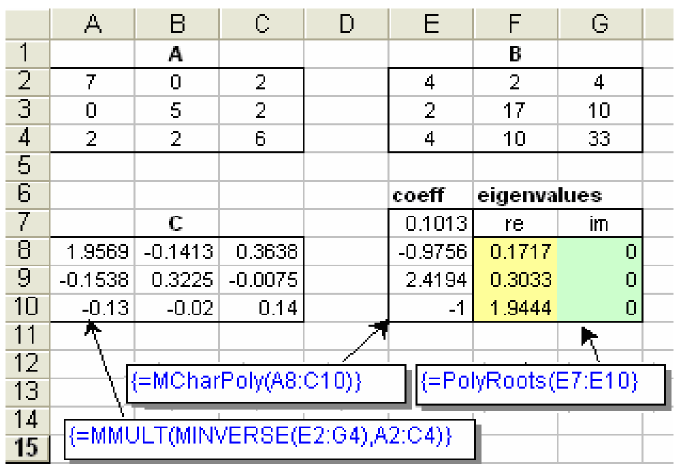
A

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 calculated the matrix **C** = **B**$^{-1}$ **A**



	A	B	C	D	E	F	G
1		**A**				**B**	
2	7	0	2		4	2	4
3	0	5	2		2	17	10
4	2	2	6		4	10	33
5							
6					**coeff**	**eigenvalues**	
7		**C**			0.1013	re	im
8	1.9569	-0.1413	0.3638		-0.9756	0.1717	0
9	-0.1538	0.3225	-0.0075		2.4194	0.3033	0
10	-0.13	-0.02	0.14		-1	1.9444	0

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 approximated its roots with the function PolyRoots. 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

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

where **A** and **B** are both symmetric matrices and **B** is positive definite.
In the previous paragraph we have seen how to transform this problem into a standard eigenproblem by setting $\mathbf{C} = \mathbf{B}^{-1}\mathbf{A}$. But **C** is not symmetric. Many algorithms only work well for symmetric matrices. By contrast, there is no equally satisfactory algorithm for the asymmetric case. So, it is better to convert the problem into a symmetrical matrix, by the Cholesky's decomposition

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

Where **L** is a triangular matrix.
Substituting (2) into (1) and multiplying the equation by $\mathbf{L}^{-1}$, we get:

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

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

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

After setting the auxiliary matrix: **W** equal to $\mathbf{L}^{-1}$, and the auxiliary vector **d** to $\mathbf{L}^{T}\,\mathbf{x}$, we have

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

Equation (3) is the new eigen problem where $\mathbf{D} = \mathbf{W\,A\,W}^{T}$ is symmetric
Eigenvalues of problem (3) are equivalent to (1) while the original eigenvectors **x** can be obtained from the eigenvectors **d** by the following formula:

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

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

Matrix.xla contains everything you need to solve generalized eigen problems: Cholesky decomposition can be done by the function MCholesky; eigenvectors and eigenvalues of symmetric matrices can be calculated with Jacoby iterative rotations performed by the two functions MEigenvalJacobi and MEigenvecJacobi.
Thus, let's see how to arrange a worksheet for solving a generalized eigen-problem, assuming the matrices **A** and **B** of the previous example. The following worksheet contains 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
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									{=MCholesky(E2:G4)}		

	A	B	C	D	E	F	G	H	I	J	K
7		**W**				$\mathbf{W}^{T}$				**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	{=MInv(I2:K4)}				{=MT(A8:C10)}		{=MProd(A8:C10, A2:C4, E8:G10)}				

.

	A	B	C	D	E	F	G	H	I	J	K
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	{=MEigenvalJacobi(I8:K10)}				{=MMopUp(A14:C16)}				{=MDiagExtr(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	{=MEigenvecJacobi(I8:K10)}				{=MMULT(E8:G10,A20:C22)}				{=E20:E22 / MAbs(E20:E22)}		

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 $\mathbf{L}^{-1}$ is quite simple: just take the inverse of each diagonal element.

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

Example - How to get mode shapes and frequencies for a structure with multiple degrees of freedom [9]

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 this 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. One can work around this problem by converting the problem to a symmetric one using the Cholesky 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 a 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):

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

So here is what you do:

Starting with **k** and **m**, make **L** ; then **W** ; and then **D**.

[9] This example comes from a true problem proposed to me by Douglas C. Stahl of the Architectural Engineering and Building Construction of the 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.

	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 the functions matEigenvalue_jacobi and matEigenvector_jacobi contained in the add-in MATRIX. Allow for a number of iterations larger 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[10] shows how to solve a larger torsion system with 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

$$\mathbf{M} = \begin{bmatrix} 115.2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 15.8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.35 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.35 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.35 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.35 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 9.21 \end{bmatrix}$$

[10] Thanks to Anthony Garcia

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 by 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 sides of equation (1) by 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** can be computed easily 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 the L matrix; but also in this case, it is very easy to compute the inverse, as

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

[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 MProd
Note that $W^T = W$ because W is diagonal.

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

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 by the factor 10^6, we finally have 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 Jacobi algorithm or by the inverse iteration
Here we have used the function MEigenvecJacobi

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

Multiplying the **V**d matrix by the auxiliary **W** matrix we find 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 MNormalize

WHITE PAGE

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EISPACK Guide, “Matrix Eigensystem Routines”, Smith B. T. at al. 1976

"Numerical Methods that usually work", F. S. Acton, The Mathematical Association of America, 1990

"Analysis Nunerical Methods", E. Isaacson, H. B. Keller, Wiley & Sons, 1966

"Calculo Numérico", Neide M. B. Franco, 2002

"Metodos Numericos" Sergio R. De Freitas, 2000

"Numerical Mathematics in Scientific Computation", G. Dahlquist, Å. Björck, vol II.

"Advanced Excel for scientific data analysis" , Robert de Levie, Oxford University Press, 2004

WHITE PAGE

Dic 2006