Appendix C

Some details of Matrix.xla(m)

C.1 Matrix nomenclature

For the sake of notational compactness, we will denote a square *diagonal* matrix by **D** with elements d_{ii}, a square *tridiagonal* matrix by **T** with elements t_{ij} where $| j - i | \le 1$, most other *square* matrices by **S**, *rectangular* matrices by **R**, and all matrix elements by m_{ij}. A *vector* will be shown as **v**, with elements v_i, and a *scalar* as s. Particular values are denoted by x when real, and by z when complex. All optional parameters are shown in straight brackets, [ ]. All matrices, vectors, and scalars are assumed to be real, except when specified otherwise. All matrices are restricted to two dimensions, and vectors to one dimension. Table C.1 briefly explains some matrix terms that will be used in subsequent tables.

With some functions, the user is given the integer option *Int* of applying integer arithmetic. When a matrix only contains integer elements, selecting integer arithmetic may avoid most round-off problems. On the other hand, the range of integer arithmetic is limited, so that overflow errors may result if the matrix is large and/or contains large numbers. Another common option it *Tiny*, which defines the absolute value of quantities that can be regarded as most likely resulting from round-off errors, and are therefore set to zero. When not activated, the routine will use its user-definable default value.

Condition	of a matrix: ratio of its largest to smallest singular value
Diagonal	of a square matrix: the set of terms m_{ij} where $i = j$
Diagonal matrix **D**	square matrix with $m_{ij} = 0$ for all off-diagonal elements $i \ne j$.
Decomposition	or factorization: writing a matrix as the product of two or more special matrices
False	as optional parameter: False = 0
First lower subdiagonal	of a square matrix: the set of terms m_{ij} where $j = i+1$
First upper subdiagonal	of a square matrix: the set of terms m_{ij} where $j = i-1$
Inverse square matrix $\mathbf{S}^{-1}$	square matrix that satisfies $\mathbf{S}^{-1}\,\mathbf{S} = \mathbf{S}\,\mathbf{S}^{-1} = \mathbf{I}$
Hermitean matrix	a square matrix for which $\mathbf{S}^{*\mathrm{T}} = \mathbf{S}$ where $\mathbf{S}^{*}$ denotes the complex conjugate of **S**; all symmetric real matrices are Hermitian
Hessenberg matrix **H**	a square matrix with $m_{ij} = 0$ for $j = i+k$, $k > 1$
Lower triangular matrix **L**	a square matrix with only 0's below its diagonal
Order	of a square matrix: its number of rows or columns
Orthogonal matrix	a real, square matrix with the property $\mathbf{S}^{-1} = \mathbf{S}^{\mathrm{T}}$
Rank	order of largest nonsingular square submatrix of a matrix
Rectangular matrix **R**	a matrix with (in general) an unequal number of rows and columns
Square matrix **S**	a matrix with an equal number of rows and columns
Subdiagonal	the set of terms m_{ij} where $i = j \pm k$ where k is an integer
Symmetric matrix	a square matrix **S** with all $m_{ij} = m_{ji}$, hence $\mathbf{S} = \mathbf{S}^{\mathrm{T}}$
Toeplitz matrix	a square matrix with constant elements on each diagonal parallel to the main diagonal
Transpose $\mathbf{R}^{\mathrm{T}}$	matrix after interchanging its rows and columns
Triangular matrix **T**	matrix with non-zero terms only on its diagonal and first upper and lower subdiagonals
True	as optional parameter: True = 1
Uniform matrix	repeats its elements on its diagonal and each subdiagonal
Unit matrix **I**	square matrix of arbitrary dimension $m \times m$ with 1's on its diagonal, and 0's above and below it
Upper triangular matrix **U**	a square matrix with only 0's below its diagonal. (Exceptions: the upper triangular matrix **R** in QR decomposition; the orthogonal matrix **U** in singular value decomposition.)

Table C-1: The nomenclature used

C.2 Functions for basic matrix operations

C.2.1 Functions with a scalar output

Entering the functions listed below does not require the use of Ctrl∪Shift∪Enter.

MAbs(**R**)	Absolute value of **R**	$\sqrt{\sum_{i,j} m_{ij}^2}$
MCond(**R**)	Condition number κ of a matrix computed using singular value decomposition	κ
MpCond(**R**)	$-\log_{10}$ of matrix condition number computed using singular value decomposition	$\mathrm{p}\kappa = -\log(\kappa)$
MDet(**S** [,Int] [,*Tiny*])	Determinant of a square matrix **S** Similar to Excel's =MDETERM(**S**). Because of rounding errors, both MDET and MDETERM can yield (often different) non-zero answers For a singular matrix. When all elements of **S** are integer, and Integer is set to True, MDET uses integer mode. Defaults: Integer = False, *Tiny* = 0.	det[**S**]
MRank(**R**)	Rank of a matrix	
MTrace(**S**)	Trace of a square matrix	$\mathrm{tr}(\mathbf{S}) = \sum_i m_{ii}$

C.2.2 Basic matrix functions

Entering the following functions requires the use of Ctrl∪Shift∪Enter

MAdd($\mathbf{R}_1$,$\mathbf{R}_2$)	Addition of two matrices equivalent to Excel's =$\mathbf{R}_1$+$\mathbf{R}_2$, as in =B2:D5+F2:H5.	$\mathbf{R}_1+\mathbf{R}_2$
MSub($\mathbf{R}_1$,$\mathbf{R}_2$)	Subtraction of two matrices Equivalent to Excel's = $\mathbf{R}_1$–$\mathbf{R}_2$, as in =B2:D5–F2:H5.	$\mathbf{R}_1-\mathbf{R}_2$
MT(**R**)	Transpose of a matrix equivalent to Excel's function TRANSPOSE	$\mathbf{R}^T$
MMult($\mathbf{R}_1$,$\mathbf{R}_2$)	Product of two matrices Excel's function is listed here for the sake of completeness	$\mathbf{R}_1\ \mathbf{R}_2$
MProd($\mathbf{R}_1$,$\mathbf{R}_2$,$\mathbf{R}_3$,...)	Product of two or more matrices Pay attention to the dimensions, as the function MProd does *not* check them.	$\mathbf{R}_1\mathbf{R}_2\mathbf{R}_3$...
MMultS(**R**,*s*)	Product of a matrix and a scalar equivalent to Excel's scalar multiplication, as in =3.21*B2:G9.	$s\mathbf{R} = \mathbf{R}s$
MPow(**S**,*n*)	$\mathbf{S}^n = \mathbf{S}\ \mathbf{S}\ \mathbf{S} \ldots \mathbf{S}$ (*n* terms)	$\mathbf{S}^n$
MInv(**S** [,Int] [,*Tiny*])	Inverse of **S** similar to Excel's =MINVERSE(M). Because of rounding errors, both M_INV(M) and MINVERSE(M) can yield (different) non-zero element values for a singular matrix. When Integer is set to True, integer mode is used. Any result smaller in absolute magnitude than *Tiny* is set to zero. Defaults: Integer = False, *Tiny* = 0.	$\mathbf{S}^{-1}$
MExp(**S** [,Algo] [,*n*])	Matrix exponential Uses Padé approximation (the default, Algo = "P"), otherwise the power method. The default stops when convergence is reached. When *n* is specified, the resulting error can be obtained with =MExpErr(**S**, *n*)	$e^S = \sum_{n=0}^{\infty} \frac{S^n}{n!}$
MExpErr (**S** ,*n*)	Error term in matrix exponential	

C.2.3 Vector functions

ProdScal($\mathbf{v}_1$,$\mathbf{v}_2$)	Scalar product of two vectors	$\mathbf{v}_1 \bullet \mathbf{v}_2$
ProdVect($\mathbf{v}_1$,$\mathbf{v}_2$)	Vector product of two vectors	$\mathbf{v}_1\ \mathbf{v}_2$
VectAngle($\mathbf{v}_1$,$\mathbf{v}_2$)	Angle between two vectors	$\arccos\left(\dfrac{\mathbf{v}_1 \bullet \mathbf{v}_2}{\lvert\mathbf{v}_1\rvert \cdot \lvert\mathbf{v}_2\rvert}\right)$

C.3: More sophisticated matrix functions

Diagonal or tridiagonal square matrices occur quite frequently in practical problems. When such matrices are of high orders, they can take up a large amount of space, even though most of it will be occupied by zeros. It is then often convenient to store and display $m\times m$ diagonal matrices **D** in compact notation as single $m\times 1$ column vectors, and tridiagonal matrices **T** as $m\times 3$ rectangular matrices. A number of special instructions are provided for this space-saving approach. Don't confuse compact notation with sparse notation, as used in connection with sparse matrices, see Table C.10.3.

MDetPar(**S**)	Determinant of **S** containing one symbolic parameter *k* Used with Ctrl␣Shift␣Enter yields vector, otherwise output shown as text string.	det[**S**]
MDet3(**T**)	Determinant of **T** in $n\times 3$ format There is no need to use Ctrl␣Shift␣Enter, because the output is a scalar.	det[**T**]
MMult3(**T**,**R**)	Multiplies a tridiagonal matrix in tricolumnar format with a rectangular or square matrix **R,** or even a vector **v**.	**T R**
MMultTpz (**S**,**v**)	Multiplies a Toeplitz matrix in compact (columnar) format and a vector **v**. For a Toeplitz matrix of order $2n+1$, **v** must be $n\times 1$	
MBAB($\mathbf{S}_1$,$\mathbf{S}_2$)	Similarity transform	$\mathbf{S}_1^{-1}\,\mathbf{S}_2\,\mathbf{S}_1$
MBlock(**S**)	Transforms reducible, sparse square matrix into block-partitioned form	
MBlockPerm(**S**)	The permutation matrix for MBlock	
MDiag(**v**)	Convert vector **v** into **D**	$m_{ii} = v_i$
MDiagExtr(**S** [,*d*])	Extract the diagonal of **S** $d = 1$ for the diagonal, $i = j$ (the default), $d = 2$ for the first lower subdiagonal, $i = j+1$.	

C.4: Functions for matrix factorization

The terms matrix *factorization* and matrix *decomposition* refer to the same operations, in which a given matrix is expressed as the product of two or more special matrices. This approach is often used to facilitate finding the required solution. The differences between the various available approaches reflect their general applicability, numerical efficiency, tolerance of ill-conditioning, etc.

SVDD(**R**)	Yields **D** of $\mathbf{R} = \mathbf{U}^{\mathrm{T}}\,\mathbf{D}\,\mathbf{V}$ The central result of singular value decomposition, providing the singular values σ_i as well as easy routes to matrix rank *r* and condition number κ. When **R** is Hermitian, the σ_i are the absolute values of its eigenfunctions. Note: the traditional symbol **U** here does not imply an upper triangular matrix.	**D**
SVDU(**R**)	Yields **U** of $\mathbf{R} = \mathbf{U}^{\mathrm{T}}\,\mathbf{D}\,\mathbf{V}$	**U**
SVDV(**R**)	Yields **V** of $\mathbf{R} = \mathbf{U}^{\mathrm{T}}\,\mathbf{D}\,\mathbf{V}$	**V**
MCholesky(**S**)	Cholesky decomposition of a symmetric matrix **M** into a lower triangular square matrix **L** and its transpose $\mathbf{L}^{\mathrm{T}}$	$\mathbf{S} = \mathbf{L}\,\mathbf{L}^{-1}$

MLU(**S** [,pivot])	LU decomposition into a lower (**L**) and upper (**U**) triangular square matrix. The optional pivot (the default) activates partial pivoting	**S** = **L U**
MOrthoGS(**R**)	Modified Gram-Schmidt orthogonalization	
MQH(**S**,**v**)	decomposition of **S** with vector **b** **Q** is orthogonal, **H** is Hessenberg. If **S** is symmetric, **H** is tridiagonal	$\mathbf{S} = \mathbf{Q}\,\mathbf{H}\,\mathbf{Q}^T$
MQR(**R**)	QR decomposition **Q** is orthogonal, **R** is upper triangular	**A** = **Q R**
MHessenberg(**S**)	Converts **S** into its Hessenberg form **H**	
MChar(**S**, *x*)	Computes characteristic matrix at real value *x* If *x* complex, use MCharC(**S**, *z*)	
MCharPoly(**S**)	Computes characteristic polynomial of **S** Can often be combined with PolyRoots(P)	
PolyRoots(P)	Finds all roots of a polynomial P	
PolyRootsQR(P)	Finds all roots of a polynomial P using the QR algorithm	
MNorm(**R** or **v** [,Norm])	Finds the matrix or vector norm For matrix **R**: Norm: 0 (default) = Frobenius, 1 = max. abs. column sum, 2 = Euclidian norm, 3 = max. abs. row sum. For vector **v**: Norm: 1 = max. sum, 2 = Euclidian norm, 3 (default) = max. abs. value	
MPerm(**p**)	generates a permutation matrix from a permutation vector **p**	
MCmp(**v**)	Companion matrix of a monic polynomial *P* where **v** contains the coefficients of *P*	
MCovar(**R**)	covariance matrix similar to Excel's COVAR(a_i, a_j)	$c_{ij} = \frac{\sum_{k=1}^{m}(m_{ki} - m_{i,av})(m_{kj} - m_{j,av})}{m}$
MCorr(**R**)	correlation matrix (i.e., normalized covariance)	

$$r_{ij} = \frac{m\sum_{k=1}^{m}(m_{ki} - m_{i,av})(m_{kj} - m_{j,av})}{\sqrt{\sum_{k=1}^{m}(m_{ki} - m_{i,av})}\sqrt{\sum_{k=1}^{m}(m_{kj} - m_{j,av})}}$$

MExtract(**R**, *row*, *column*)	Creates a submatrix of **R** by extracting a specified *row* and *column*	
MMopUp(**R** [,*ErrMin*])	Eliminates round-off errors from **R** by replacing by zero all elements $|a_{ij}| < ErrMin$ (default 10^{-15})	
MRot(*m*, *theta*, *p*, *q*)	Creates orthogonal matrix of order *m* that rotates by angle *theta* in *p*,*q* plane $p \neq q, p \leq m, q \leq m$	

C.5 Eigenvalues & eigenvectors

The German word "eigen" in this context is best translated as "particular to": eigenvalues and eigenvectors of a matrix are scalars and vectors that are *particular to that matrix*. They are only defined for square matrices.

C.5.1: For general square matrices

MEigenvalJacobi(**S** [,*MaxIter*]) Jacobi sequence of orthogonality transforms
MaxIter (default 100) is the max. # of iterations

MEigenvalMax(**S** [,*MaxIter*]) Finds maximum |eigenvalue by using the iterative power method
MaxIter (default 1000) is the max. # of iterations

MEigenvecPow(**S** [,*Norm*] [,*MaxIter*]) Approximates eigenvalues for diagonizable **S**
by using the power method. Normalizes eigenvector if *Norm* = True; default = False
MaxIter (default 1000) is the max. # of iterations

MEigenvalQR(**S**) Approximates the eigenvalues of **S** by QR decomposition
Yields an $n\times1$ array, or $n\times2$ for complex eigenvalues

MQRIter(**S**[,*MaxIter*]) Iterative diagonalization of **M** to yield its eigenvalues
based on QR decomposition *MaxIter* (default = 100) sets the max. # of iterations

MEigenvec(**S**, **eval** [,*MaxErr*]) Computes eigenvector of **S** for a given eigenvalue(s) in vector **eval**

MEigenvecInv(**S**, **eval**) Computes eigenvectors for a given vector **eval** by inverse iteration

MEigenvecJacobi(**S**[,*MaxIter*]) Orthogonal similarity transforms of a symmetric matrix **S**
MaxIter (default = 100) sets the max. # of iterations

MEigenvectMax(**S** [,*Norm*] [,*MaxIter*]) Yields eigenvector for dominant eigenvalue
(i.e., with max. absolute value). Normalizes eigenvector if *Norm* = True; default = False

MEigenvecPow(**S** [,*Norm*] [,*MaxIter*]) Yields real eigenvectors for diagonizable **S**
using the power method. Normalizes eigenvector if *Norm* = True; default = False.
MaxIter (default 1000) is the max. # of iterations

MRotJacobi(**S**) Jacobi orthogonal rotation of symmetric **S**

MEigenSortJacobi(**eval**, **evec** [,*n*]) Sorts eigenvectors by value of |eigenvalue|
Optional *n* specifies number of eigenvectors shown

MNormalize(**R** [,Norm] [*Tiny*]) Normalize real matrix **R**
Norm specifies normalizing denominator: 1 = $|v_{min}|$,
2 (default) = $|v|$, 3 = $|v_{max}|$; *Tiny* default = 2×10^{-14}

C.5.2: For tridiagonal matrices

MEigenvalQL(**T** [,*MaxIter*]) Approximates eigenvalues of tridiago nal symmetric matrix
using the QL algorithm accepts **T** in either regular or compact format.
MaxIter (default 200) is the max. # of iterations

MEigenvalTTpz(n, a, b, c) Computes eigenvalues for a tridiagonal
Toeplitz matrix with elements a, b, c
All eigenvalues are real if $ac > 0$, complex if $ac < 0$

MEigenvecT(**T**, eigenvalues [,*MaxErr*]) Approximates eigenvectors for given eigenvalue(s) of **T**
Accepts **T** in either square or compact format

C.6 Linear system solvers

Linear system solvers solve a system of simultaneous linear equations in one single user operation. Int = True uses integer computation, otherwise use False (default). *Tiny* sets the minimum absolute round-off error that will be replaced by 0 (default: 10^{-15}).

SysLin (**S**, **x** [,Integer] [,*Tiny*]) Gauss-Jordan solution of linear system
M is the matrix of independent (control) parameters, **x** is the unknown coefficient vector or matrix

SysLinIterG (**S**, **x**, $\mathbf{x}_0$ [,*MaxIter*] [,*w*]) Iterative Gauss-Seidel solution of linear system
using relaxation **M** is the matrix of independent (control) parameters, **x** is the unknown coefficient vector or matrix, $\mathbf{x}_0$ its starting value, *MaxIter* (default = 200) is the max # of iteration (*MaxIter* = 1 can be used for step-by-step use), *w* (default = 1) is the relaxation factor

SysLinIterJ (**S**, **x**, $\mathbf{x}_0$ [,*MaxIter*] [,*w*]) Iterative Jacobi solution of linear system
S is the matrix of independent (control) parameters, **x** is the unknown coefficient vector or matrix, $\mathbf{x}_0$ its starting value, *MaxIter* (default = 200) is the max # of iteration (*MaxIter* = 1 for step-by-step use).

SysLinT (**T**, **x** [,*Type*] [,*Tiny*]) Solution of triangular linear system
by forward or backward substitution. **T** is either **U** (upper) or **L** (lower) diagonal; the optional (i.e., unnecessary) *Type* specifies **U** or **L**.

SysLin3 (**T**3, **x** [,Integer] [,*Tiny*]) SysLin for tridiagonal matrix **T**3
where **T**3 is in compact notation

SysLinTpz (**S**, **v**) Solves a Toeplitz linear system by Levinson's method

SysLinSing (**S** or **R** [,**x**] [,*MaxErr*]) Linear system analysis of a singular system
The matrix can be square ($m\times m$) or rectangular ($m\times n$, where $m<n$, i.e., for an underdetermined system). When **x** is not specified, it is taken as **0**. *MaxErr* (default = 10^{-13}) sets the relative precision. For degenerate (multiplicitous) eigenvalues a larger error tolerance may be needed, such as *MaxErr* = 10^{-10}. A system without solution returns a question mark.

TraLin (**R** ,**X** [,**B**]) Linear transformation ` $\mathbf{Y} = \mathbf{RX} + \mathbf{B}$
R is $m\times n$; **X** is $n\times p$, **B** is $m\times p$, and **Y** is $m\times p$. Also works when $p = 1$, in which case **X**, **B**, and **Y** are vectors.

C.7 Functions for complex matrices

There are many physical phenomena that are best described in terms of matrix algebra with complex rather than real numbers. For example, the concept of a dielectric permittivity ε of a medium can be extended from strictly transparent media to (partially or completely) light-absorbing ones by considering ε as a complex quantity. Electrical networks containing phase-shifting components are conveniently described in terms of complex quantities such as admittance and impedance. Likewise, the linear (i.e., small-amplitude) response of an electrochemical interface is most completely described in terms of Rangarajan's matrix model (*J. Electroanal. Chem*. 55 (1974) 297-374), which includes complex quantities reflecting the time lags of mass transport and interfacial capacitance. Modern quantum theory uses complex wave functions.

The Excel functions involving complex quantities, as listed in Appendix A.5, only use the character string format. The matrix operations involving complex functions listed below allow the user, through the optional instruction parameter *c*, to select one of three notational formats. These formats are *c* = 1: split; *c* = 2: interlaced, and *c* = 3: character string. Figs. C.7.1 and C.7.2 illustrate these for when the real and imaginary components are integer or non-integer respectively.

In the split format each complex *entity* (scalar, vector, matrix) is displayed with its real components, and to its immediate right with its imaginary components. In the interlaced format, each complex *number* is represented in two adjacent cells on the same row. In the text string format, the numbers are displayed as character strings listing both the real and imaginary component, as in the Excel-supplied functions for complex numbers. In the latter case, the results may have to be decoded with =IMREAL() or =IMAGINARY(). These three ways of representing complex numbers are illustrated in Fig. C.7.2. The default mode is 1, the split format.

	A	B	C	D	E	F	G	H	I
1									
2	***Split format (default)***								
3	-2	8	0	0	5	7	=MDetC(A3:F5)		
4	3	4	6	5	-4	-2	9	-103	
5	-2	9	8	6	2	6			
6									
7	***Interlaced format***								
8	-2	0	8	5	0	7	=MDetC(A8:F10,2)		
9	3	5	4	-4	6	-2	9	-103	
10	-2	6	9	2	8	6			
11									
12	***Text string format***								
13	-2	8+5j	7j			=MDetC(A13:C15,3)			
14	3+5j	4-4j	6-2j				9-103j		
15	-2+6j	9+2j	8+6j						

Fig C.7.1: The three ways to display complex quantities: (1) "split", as entire quantities with real and imaginary components, the default mode; (2) "interlaced", in which each individual element is shown with its two components adjacent to each other; and (3) "string", as text strings. The matrix and its determinant contain only integer and imaginary components, in which case the text string format is often the more compact.

	A	B	C	D	E	F	G	H
1								
2	***Split format (default)***							
3	-2.42750	8.17185	0.04820	0	4.53980	6.84630		=MDetC(A3:F5)
4	2.65938	4.07577	6.33463	5.28370	-4.32580	-1.68270	-20.96548	-78.61323
5	-2.36394	9.49214	7.88308	6.49700	2.01150	5.62860		
6								
7	***Interlaced format***							
8	-2.42750	0	8.17185	4.53980	0.04820	6.84630		=MDetC(A8:F10,2)
9	2.65938	5.28370	4.07577	-4.32580	6.33463	-1.68270	-20.96548	-78.61323
10	-2.36394	6.49700	9.49214	2.01150	7.88308	5.62860		
11								
12	***Text string format***							
13	-2.4275	8.17185+4.5398j	0.0482+6.8463j					=MDetC(A13:C15,3)
14	2.65938+5.2837j	4.07577-4.3258j	6.33463-1.6827j					-20.9654787597-78.6132259685j
15	-2.36394+6.497j	9.49214+2.0115j	7.88308+5.6286j					

Fig C.7.2: The three ways to display complex quantities, when the numbers are not restricted to integers, in which case the text string format may require much wider columns.

MCplx (**R**$_1$, **R**$_2$ [,*c*])	Convert two real matrices **M** into one complex matrix **C**	$\mathbf{C} = \mathbf{R}_1 + i\mathbf{R}_2$
MAddC (**C**$_1$, **C**$_2$ [,*c*])	Add two complex matrices	$\mathbf{C}_1 + \mathbf{C}_2$
MSubC (**C**$_1$, **C**$_2$ [,*c*])	Subtract two complex matrices	$\mathbf{C}_1 + \mathbf{C}_2$
MAbsC (**C**[,*c*])	Absolute value of a complex vector	
MDetC (**C**)	Determinant of a complex square matrix **C**	Det(**C**)
MInvC (**C** [,*c*])	Invert of a complex square matrix	$\mathbf{C}^{-1}$
MMultC (**C**$_1$, **C**$_2$ [,*c*])	Product of two complex matrices $\mathbf{C}_1\,\mathbf{C}_2$	
MPowC (**C**$_1$, **C**$_2$, **C**$_{3, \ldots}$ [,*c*])	Product of two or more complex matrices	$\mathbf{C}_1\,\mathbf{C}_2\,\mathbf{C}_3 \ldots$
MMultSC(**C**, *s* [,*c*])	Product of a complex matrix **C** and scalar *s*	$s\,\mathbf{C} = \mathbf{C}\,s$
MTC (**C** [,*c*])	Transpose of a complex matrix **C**	$\mathbf{C}^{T}$

MTH (**C** [,c]) Hermitian (conjugate, adjoint) transpose of **C** $\mathbf{C}^{H} = \mathbf{C}^{*T} = \mathbf{C}^{T*}$

ProdScaleC ($\mathbf{v_1}$, $\mathbf{v_2}$) Scalar product of complex vectors $\mathbf{v}_1 \bullet \mathbf{v}_2$

MNormalize (**C** [,Norm] [,c] [*Tiny*]) Normalize complex matrix **C**
Norm specifies normalizing denominator: 1 = $|v_{min}|$, 2 (default) = $|v|$, 3 = $|v_{max}|$; *Tiny* default = 2×10^{-14}

MCharC (**C**, z [,c]) Compute characteristic matrix of **C** at value z
M and/or z can be real or complex

MCharPolyC (**C**, [,c]) Compute the characteristic polynomial

PolyRootsQRC (**p**, [,c]) Find all roots of a complex vector **p** of polynomial coefficients
using the QR algorithm

MEigenvalQRC (**C** [,c]) Approximates the eigenvalues of a complex square matrix **C**
using QR decomposition

MEigenvecC (**C** [,c]) Compute complex eigenvector of **C** for given complex eigenvalue(s)

MEigenvecInvC (**C**, eigenvalues [,c]) Compute eigenvector of **C** for given eigenvalue(s)
by inverse iteration

SysLinC (**C**, **x** [,c]) Gauss-Jordan solution of complex linear system.
C: vector or matrix of independent parameters, **x**: is the unknown coefficient vector or matrix

C.8 Matrix generators

The following is a collection of routines for generating various types of matrices. It starts with the simplest, the identity matrix, and includes not only a number of named matrices but, also, routines to generate custom-ordered matrices, such as matrices with a given set of eigenvalues or with a given amount of sparsity. Often used option: Int = True (default) creates an integer matrix, otherwise use False.

MIde(m) Generates the identity matrix **I** of order m, i.e., $\mathbf{I}_{m\times m}$

MRnd(m [,n] [,Type] [,Int] [,*AMax*] [,*AMin*] [,*sparse*])
Generates a random $m\times n$ matrix (default: $n = m$). Type specifies the type of matrix: All (default) fills all cells, Sym generates a symmetrical matrix, Dia a diagonal one, Trd a tridiagonal, Tlw a tridiagonal lower, Tup a tridiagonal upper, and SymTrd a symmetrical tridiagonal matrix. *AMax* and *AMin* specify the maximum and minimum element values. *Sparse* accepts values from 0 to 1: 0 (default) for filled, 1 for very sparse.

MRndEig(**v** [,Int]) Creates a random real matrix for a given vector **v** of eigenvalues

MRndEigSym(**v**) Creates a symmetrical random real matrix for a given vector **v** of eigenvalues

MRndRank(m [,*Rank*] [,*Det*] [,Int]) Creates a square real matrix with a given value of *Rank* or *Determinant*. If Rank < m, Det = 0.

MRndSym(m [,*Rank*] [,*Det*] [,Int]) Creates a square real symmetrical matrix of dimension $m\times m$ with a given value of *Rank* or *Determinant*. If Rank < m, Det = 0.

MHilbert(m) Creates the $m\times m$ Hilbert matrix
The Hilbert matrix is ill-conditioned; its elements $h_{ij} = 1/(i+j+1)$ are shown in decimal form

MHilbertInv(m) Creates the $m\times m$ inverse Hilbert matrix
The elements of the inverse Hilbert matrix are all integer

MHouseholder(**x**) Creates the Householder matrix of vector **x**

MTartaglia(m) Creates the $m\times m$ Tartaglia (or Pascal) matrix
Element values: $m_{i1} = m_{1j} = 1$; for $i > 1, j > 1$: $m_{ij} = m_{i-1,j}+m_{i,j-1}$

MVandermonde(**x**) Creates the Vandermonde matrix **X** of vector **x**, as used in, e.g., the least squares formalism

C.9 Miscellaneous functions

C.9.1 Linear least squares routines

RegrL(**y**, **x** [,*Intercept*]) Linear least squares based on svd

Equivalent to post-Excel2002 LinEst. **y**: $N\times1$ vector of dependent variables, **x**: $N\times1$ vector or $N\times m$ matrix of independent parameters for the monovariate and multivariate case respectively. *Intercept* = a_0 when specified; default leaves a_0 unspecified. First output column: coefficients a_i; 2nd output column: standard deviations s_i.

RegrP(*Order*, **y**, **x** [,*Intercept*]) Linear least squares polynomial fit

based on svd, equivalent to post-Excel2002 LinEst. *Order* is the polynomial order, **y** the $N\times1$ vector of dependent variables, **x** the $N\times1$ vector of the independent parameter x. Powers of x are generated internally. *Intercept* = a_0 when specified; default leaves a_0 unspecified. Output: 1st column: coefficients a_i; 2nd column: standard deviations s_i.

RegrCir(**x**, **y**) Least squares fit to a circle through all points (x_i,y_i), yields radius and x,y coordinates of circle center, with standard deviations

C.9.2 Optimization routine

Simplex(**y**, constraints [,optimum]) Simplex optimization

$\mathbf{y} = a_0 + a_1x_1 + a_2x_2 + ...$, as $1\times m$ vector of the coefficients a_0, a_1, a_2, ...
constraints: <, >, = ; optimum: 1 (default) maximum, 0 minimum

C.9.3 Step-by-step demonstration

GJStep(**S** [,Type] [,Integer] [,*Tiny*]) Step-by-step (didactic) tracing of Gauss-Jordan elimination

leading to either diagonal (Type = D) or triangular (Type = T) reduction. Integer = True conserves integer values, default = False. *Tiny* sets minimum round-off error; default = 2×10^{-15}. Copy & paste for the next step.

C.9.4 Economic optimization routines

MLeontInv (**S**,**v**) Inverts the Leontief matrix encountered in economic input-output analysis

VarimaxIndex (**F** [,row-norm]) Varimax index for given factor loading matrix **F**.

Row-normalization: False (default) or True

VarimaxRot (**F** [,row-norm] [,*MaxErr*] [,*MaxIter*]) Orthogonal rotation of factor loading matrix **F** in Kaiser's Varimax model.

Row-normalization: False (default) or True; *MaxErr* default = 10^{-4}; *MaxIter* default = 500.

C.9.5 Minimum path routines

PathFloyd(**G**) Computes the matrix of shortest-path pairs from an adjacency matrix **G**

PathMin(**G**) Shows vectors of shortest paths

C.9.6 Routine for electrical circuit admittance

MAdm(**B**) Creates an admittance matrix from a 3- or 4-column wide branch matrix **B**

(two columns for the nodes, and 1 or 2 columns for the admittance of the individual circuit elements

C.10: Matrix macros

The Matrix Toolbar provides access to a set of matrix-related macros through three menu headings: Selector, Generator, and Macros. Below we will briefly describe each one of these.

C.10.1 The Selector tool

The Selector tool can be used to select different parts of a matrix. Start with identifying a matrix (when that matrix is bordered by empty cells, just clicking on a single cell of that matrix will do), and then use the choices presented in the Selector dialog box. In other words, click on a cell in a matrix, click on Selector, click on a choice, such as Triang. low, again click on the Selector, then on the Paster (at the bottom of the Selector menu), select a starting cell, and click OK. You will see the lower triangular part of the selected matrix appear, starting at the selected starting cell. The available choices are listed in Table C.10.1. You can even arrange for diverse output formats through the Target range selector. When you do not specify a matrix ahead of time, click on Selector, and its dialog box will give you entry to the Selector choices.

Selector choice	**Brief description**
Full	the entire matrix
Triang. low	the lower triangle, including the diagonal
Triang. up	the upper triangle, including the diagonal
Diag. 1st	the (main) diagonal, from top-left to bottom-right
Diag. 2nd	the anti-diagonal, running from top-right to bottom-left
Tridiag. 1st	the tridiagonal, from top-left to bottom-right
Tridiag. 2nd	the anti-tridiagonal, from top-right to bottom-left
Subtriang. low	the lower triangle minus the diagonal
Subtriang. up	the upper triangle minus the diagonal
Adjoint	the matrix minus the row and column of the chosen cell

Table C.10.1: The choices offered in the Selector dialog box.

As its default, the Selector dialog box will copy the selected matrix parts as is, at your option leaving the unselected cells empty or filling them with zeros. By using its Target range you can also choose different output formats, such as vertical, horizontal, diagonal, transposed, etc. For the Adjoint output, also set the Target range at Adjoint.

C.10.2 The Generator tool

The Generator tool allows you to create matrices to your specifications. Apart from its four generators of specific matrices (Hilbert, inverse Hilbert, Tartaglia, and Toeplitz) of user-selectable order, it contains four random matrix generators, which are marvelous learning and teaching tools, especially when combined with some of the matrix functions described in the earlier sections to monitor their performance. Table C.10.2 lists the various choices available.

Generator choice	Brief description
Random	generates random matrices of user-selected dimensions, minimum and maximum element values, format (full, triangular, tridiagonal, integer, symmetric), and numerical resolution.
Rank/Determinant	generates random square matrices of user-selected order and determinant (the default, if rank = order) or rank (if det = 0).
Eigenvalues	generates random square matrices with user-selected eigenvalues.
Hilbert	generates the Hilbert matrix of given order.
Hilbert inverse	generates the inverse Hilbert matrix of given order.
Tartaglia	generates the Tartaglia matrix of given order.
Toeplitz	generates the Toeplitz matrix of given order.
Sparse	generates sparse square matrices of user-selected order, minimum and maximum element values, dominance factor, filling factor, and spreading factor. One can specify integer and/or symmetrical output, and regular (square) or sparse output display. In the latter case, all non-zero elements m_{ij} are listed in three adjacent columns as i, j, and m_{ij}.

Table C.10.2: The choices offered in the Generator dialog box.

C.10.3 The Macros tool

The Macros tool provides easy access to a number of macros. Many of these macros duplicate matrix functions already described in appendices B.2 to B.8, but the sparse matrix operations contains some additional features. The choices given in the Macros dialog box are listed in Table B.10.3. Some matrices can be selected by simply pointing to one cell of that matrix, and by then clicking on the smart selector icon, labeled with a rectangle. This method works only when the matrix in question is surrounded by empty cells and/or the spreadsheet border.

Macro choice	Brief description
Matrix operations	reproduces the most often used matrix functions
Complex matrix operations	duplicates many of the functions of section 9.7
Sparse matrix operations	applies the most common matrix operations to sparse matrices in sparse matrix format (i.e., in three adjacent columns: i, j, m_{ij}), thereby greatly facilitating handling large sparse matrices on the spreadsheet. It includes an efficient ADSOR (adaptive successive over-relaxation) Gauss-Seidel method.
Eigen-solving	provides eigenvalues, eigenvectors, the characteristic matrix, and the characteristic polynomial for a square (real, real tridiagonal, complex) matrix
Gauss step-by-step	a macro form of GJ_Step
Graph	includes Shortest Path and Draw
Methods	Clean-up and Round

Table C.10.3: The choices offered in the Macros dialog box.

Appendix D

XN extended-precision functions & macros

Here we list the major instructions available at present with XN.xla(m) version 6051. The further down the list, the sparser the annotations. A more complete listing is available once you have installed XN.xla(m), and its Toolbar, which can be toggled on and off by clicking on the XN purple book icon featuring an **X**. Because this software is still developing and growing; whenever information provided here differs from the documentation provided with your installed version, consider the latter as authoritative. For a quick guide on the format used, also consult the Paste (Insert) Function window by clicking on its icon, $\boldsymbol{f_x}$. Note that numbers displayed by Excel are usually stored as their binary approximations; when they are text strings, they are shown within quotation marks " inside the function argument, or as '*a* = .

For the list of available functions click on the Help button of the XN Toolbar, click on Help-on-line, which will open up the Xnumbers version 6.0 Help file. For the most recent list of functions, which includes the many recent updates from John Beyers, click on "changes to version 6.0" at the end of its first paragraph. For the older functions, use its Index of Functions or other items in its Contents. When in doubt, try them out!

In the list below, items shown within straight brackets [ ] are optional. The letter *D* is used as an abbreviation for DgtMax; I recommend a value of 35 (roughly quintuple precision) to 50, as usually sufficient for final 15-decimal accuracy yet still very fast. The value of *D* = 35 is used here unless otherwise specified. As long as you avoid degrading its performance by mixing in double-precision operations, XN functions and macros with *D* = 35 pass all NIST StRD linear and nonlinear least squares tests with flying colors. Whether you will find p*E* = 15 or 'merely' p*E* ≥ 14 may well depend on how you read in the data files. When you import test data, and then let a VBA routine read them from Excel, it will read the *stored* data, which are binary *approximations* of the data shown on the screen, see section 11.14. Instead, copy them literally and place them between quotation marks. In the same vein, be careful with your input arguments. Instead of 1/3 use xDiv(3,10), replace 0.317 by "0.317", for -2 substitute xNeg(2) or "-2", etc., e you may degrade the accuracy of your output.

To change the default *D*-value, use the XN Toolbar, select X-Edit ⇨ Configuration, and enter the desired value in the Default digits window. For 32-bit systems, the current *D*-values range from *D* ≤ 630 for XN.xla(m)6051-7A or -7M, to *D* ≤ 4030 for XN.xla(m)6051-13A or -13M. For best accuracy and speed, stay at least two packets (14 decimals for –7A and –7M, 26 decimals for –13A and –13M) below the upper edges of these ranges. Using a *D*-value much larger than needed merely slows you down.

D.1 Numerical constants

The brackets are required, even when empty, in which case *D* assumes its default value, here set to 35.

xPi([*D*]) **π, the ratio of circumference to diameter of a circle** π

xPi() = 3.14159265358979323846264338327950029 when default *D* is 35;
xPi(58) = 3.141592653589793238462643383279502884197169399375105820975;
xPi(600) = 3.141592653589793238462643383279502884197169399375105820974
94459230781640628620899862803482534211706798214808651328230664709384
46095505822317253594081284811174502841027019385211055596446229489549
30381964428810975665933446128475648233786783165271201909145648566923
46034861045432664821339360726024914127372458700660631558817488152092
09628292540917153643678925903600113305305488204665213841469519415116
09433057270365759591953092186117381932611793105118548074462379962749
56735188575272489122793818301194912983367336244065664308602139494639
522473719070217986094370277053921717629317675238467481846766940513.

x2Pi([*D*])	2π xPi(50) = 6.28318530717958647692528676655900576839433879875 02; xPi(5) = 6.2832; xPi() = 6.2831853071795864769252867665590058.	2π
xPi2([*D*])	π/2 xPi2(50) = 1.57079632679489661923132169163975144209858469968 76.	π/2
xPi4([*D*])	π/4 xPi4(50) = 0.78539816339744830961566084581987572104929234984 378.	π/4
xE([*D*])	**e, the base of the natural logarithm** xE() = 2.71828182845904523536028747135266 25 when the default *D* is 35	*e*
xEu([*D*]), **xGm**([*D*])	**γ, Euler's gamma** xEu(42) = xGm(42) = 0.577215664901532860606512090082402431042159.	γ
xLn2([*D*])	**Natural logarithm of 2** xLn2(50) = 0.69314718055994530941723212145817656807550013436026.	ln(2)
xLn10([*D*])	**Natural logarithm of 10** xLn10(50) = 2.30258509299404568401799145468436420760110148862 88.	ln(10)
xRad5([*D*])	**Square root of 5** xRad5(50) = 2.23606797749978969640917366873127623544061835961 15.	√(5)
xRad12([*D*])	**Square root of 12** xRad12(50) = 3.46410161513775458705489268301174473388561050762 08.	√(12)

D.2 Basic mathematical operations

xAbs(*a*)	**Absolute value** Do not enter *D* in this instruction. xAbs("-1.2345") = 1.2345; xAbs("-1234567890.0987654321") = 1234567890.0987654321; xCos(xPi()) = -1 so that xAbs(xCos(xPi())) = 1.	|*a*|
xIncr(*a*)	**Increment *a* by 1** e.g., xIncr(xPi()) = 4.1415926535897932384626433832795029 and xIncr(xPi(28)) = 4.1415926535897932384626433833 for (π + 1), where xPi([*D*]) has an optional *D*, while xIncr(xPi(),28) yields #VALUE! because it incorrectly specifies *D* for xIncr(), which cannot handle it.	*a*+1
xAdd(*a*,*b*[,*D*])	**Addition** e.g., Add(xPi(),xE()) = 5.8598744820488384738229308546321654, xAdd(xPi(),xE(),21) = 5.85987448204883847382 for (π + *e*) with 35 (the default used here) or 21 decimals respectively.	*a*+*b*
xSum(A[,*D*])	**Summation of terms in a cell range** Ignores empty cells as well as cells containing text. Example: Place the instruction =xPi() in cell B3, =xIncr(B3) in B4, and copy this down to B8. In cell B10 then place the instruction =xSum(B3:B8), which will yield 33.849555921538759430775860299677017. In cell B11 verify that you get the same answer with =xAdd(15,xMult(6,xPi())) for (1+2+3+4+5) + 6 π.	Σ a_i
xNeg(*a*)	**Negation** Do *not* use –*a* because it will convert the result to double precision. Instead, always use xNeg instead of a minus sign in XN, otherwise you will revert to double precision. Using quotation marks surrounding a fractional number uses it as shown, xNeg("-1234567890.0987654321") = 1234567890.0987654321 whereas xNeg(-1234567890.0987654321) = 1234567890.098759889602661133 uses the value stored by Excel approxi- mating the 15-decimal number -1234567890.09876 in binary notation. No such distortion (but still truncation to 15 decimals) occurs with integers: xNeg(-12345678900987654321) = 123456789009876.	–*a*

xSub(*a*,*b*[,*D*]) **Subtraction** *a–b*

equivalent to xAdd(*a*, xNeg(*b*)). Do *not* use xAdd(*a*,–*b*) because the notation –*b* will make the result double precision. Example: (π – *e*) ⇨ xSub(xPi(),xE()) = 0.42331082513074800310235591192684O4. Also: xSub("1.00000000000000000012345678","1.00000000000000000023456789") = -1.1111111E-17, with all leading zeroes automatically deleted. And note: xSub(1.2345678901234," 1.2345678901234") = 6.9057879591E-17 illustrates the distortion due to decimal-to-binary conversion.

xMult(*a*,*b*[,*D*]) **Multiplication** *a* × *b*

e.g., xMult(6, Pi()) = 18.849555921538759430775860299677017, and xMult(6,Pi(42),42) = 18.849555921538759430775860299677017305183.

xProd(*a*[,*D*]) **Multiplication of components of a cell range** $\Pi\ a_i$

Ignores empty cells as well as cells containing text. The range can be a column, a row, or a rectangular array, but *not* an enumeration of comma-separated cell values or cell addresses..

xInv(*a*) **Inversion** 1/*a*

When *a* = 0, xInv(*a*) yields "?". Example: 1/9 in 42-decimal precision is xInv(9,42) = 0.111111111111111111111111111111111111111111

xDiv(*a*,*b*[,*D*]) **Division** *a*/*b*

or: xMult(*a*, xInv(*b*)). When *b* = 0, xDiv(*a*,*b*) yields "?". xDiv(7,9,42) = 0.777777777777777777777777777777777777777778

xDivInt(*a*,*b*) **Integer division** int (*a*/*b*)

xDivInt(*a*,0) ⇨ "?". xDivInt(7,9) = 0; xDivInt(13,7) = 1; xDivInt(-13,7) = -2.

xPow(*a*,*p*[,*D*]) **Power** a^p

where *a* can be positive or negative, and with integer or noninteger powers *p*
xPow(xPi(),xNeg("2.7"),21) = 4.54668999316115830687E-2
but xPow(xPi(),xNeg(2.7),21) = 4.54668999316115738232E-2; and watch this:
xPow(xNeg(xPi());xNeg("2.7"),21) = -2.67247732472589436167E-2
-3.67834947262189055211E-2j because $-\pi^{-2.7}$ has a complex root.

xPow2(*p* [,*D*]) **Power of 2** 2^p

where the power *p* can be positive or negative, integer or noninteger.
e.g. xPow2(xNeg("400.3") = 3.14552206294614155070350912629303O1E-121;
xMult(xPow2(xNeg("400.3")),xPow2("400.3"),34) = 1.

xExp(*p*[,*D*]) **Exponential** e^p

xExp(80) = 55406223843935100525711733958316613, xExp(800) = 2.7263745721125665673647795463672698E+347 and xExp(800,14) = 2.7263745721126E+347. The latter two cannot be read by Excel or reduced to double precision, because Excel cannot store numbers beyond E308.

xExpa(*p*[,*a*][,*D*]) **Arbitrary power** a^p

Note the *unusual argument order*: power first, then the value raised to it:
xExpa(3,7) = 343 = 7^3. When *a* is unspecified, *a* = 10: xExpa(3) = 1000;
xExpa(3,xPi()) = π^3 = 31.006276680299820175476315067101396,
xExpa(xNeg(3),xPi()) = $(-\pi)^3$ = -3.2251534433199492956082745196133453E-2;
watch the commas: xExpa("3.01",17) = 5054.1863831357180932094218872658106 but xExpa("3.01",,17) = 1023.2929922807541 and xExpa(3.01,,17) = 1023.2929922807536;
xExpa(xNeg("3.01")) = 31.363254111413810434877685894955175;
xExpa(xNeg("3.01"),xNeg(xPi()),21) = $(-\pi)^{-3.01}$ = -3.18844465707427014412E-2).

xExpBase(*a*,*x*[,*D*]) **Arbitrary power** a^x

Arbitrary power of any base. Similar to xExpa(*x*,*a*[,*D*]) but *a* not optional.

xSqr(a[,D]) **Square root of *a*** $\sqrt{(a)}$

xSqr("4.7") = 2.16794833886787899418989624480732099 = √ (4.7),
xSqr("4.7",50) = 2.167948338867879941898962448073209935826865748722.

xSqrPi(a[,D]) **Square root of *a* times π** $\sqrt{(a\pi)}$

for $a \geq 0$. If a is omitted, a = 1. xSqrPi(,21) = 1.7724538509055160273 = √π ,
xSqrPi("4.7",21) = 3.84258838179059041156 = √ (4.7 π) to 21 decimals.

xRoot(a[,b][,D]) **Arbitrary root** $a^{1/b}$

b need not be an integer; default: b = 2. xRoot(9) = 3 = √2 , as is xRoot(9,2),
but xRoot(2,9) = 1.0800597388923061698729308312885969, and xRoot(2,,9)
= 1.41421356; xRoot(78,9) = 1.6226794404526244307856240252218919 = $78^{1/9}$,
while xRoot(78,"9.0001") = 1.6226707127436371883687249182251982.

xLn(a[,D]) **Natural logarithm** ln a

xLn(11,50) = 2.3978952727983705440619435779651292998217068539374.

xLog(a[,base][,D]) **General logarithm** $\log_n a$, $\log a$

Optional base must be positive; default = 10. Analogous to Excel's LOG(a [,base])
where LOG(4,2) = 2 = $\log_2(4)$ and LOG(4) = 0.60206.. = $\log_{10}(4)$, XN uses
xLog(30,3) = 3.0959032742893846042965675220214013 =$\log_3(30)$ at $D_{default}$ = 35,
and xLog(30,,35) = xLog(30) = 1.4771212547196624372950279032551153 =$\log_{10}(30)$

D.3 Trigonometric and related operations

All angles are assumed to be in radians. The prefix ar stands for area, the prefix arc for arc.

xSin(α [,D]) **Sine** sin α

xSin(0.5,50) = 0.4794255386042030002732879352155713880818033679406;
xSin(xPi()) = -1.5802830600624894179025055407692184E-35;
xSin(xPi(46),46) = 3.7510582097494459230781640628620899862803482534E-46;
xSin(xSub(xPi(),0.00000001)) = 1.0000000000000000042558941617530493E-8;
xSin(xSub(xPi(),"0.00000001")) = 9.9999999999999999833333333175305036E-9.

xCos(α [,D]) **Cosine** cos α

xCos("0.5",50) = 0.87758256189037271611628158260382965199164519710974
and xCos(0.5,50) = 0.87758256189037271611628158260382965199164519710974,
because 0.5 = ½ is exactly convertible into binary notation, as are 0.75, 0.625, etc.;
xCos(xPi2(),50) = 4.2098584699687552910487472296153908203143104499314E-35.
xCos(xPi2(50),50)=-4.7089512527703846091796856895500685982587328941466E-50.

xTan(α [,D]) **Tangent** tan α

xTan(0.5,50) = 0.54630248984379051325517946578028538329755172017979.

xASin(a[,D]) **Inverse sine** arcsin a

$|a| \leq 1$; xASin(1) = 1.5707963267948966192313216916397514;
xASin(xNeg(1),48) = -1.57079632679489661923132169163975144209858469969.

xACos(a[,D]) **Inverse cosine** arccos a

$|a| \leq 1$; xACos(0,48) = 1.57079632679489661923132169163975144209858469969.

xATan(a[,D]) **Inverse tangent** arctan a

xATan(1,50) = 0.78539816339744830961566084581987572104929234984378.

xATan2(a, b[,D]) **Inverse tangent of quotient *a*/*b*** arctan (a/b)

xATan2(3,4,50) = 0.64350110879328438680280922871732263804151059111531;
note that the order of a and b is reversed from that used in Excel's ATAN2.

xSinH(a[,D]) **Hyperbolic sine** sinh a

sinh a = ($e^x - e^{-x}$) / 2; xSinH(3) = 10.017874927409901898974593619465828.

xCosH(a[,D]) **Hyperbolic cosine** $\cosh a$

$\cosh a = (e^x + e^{-x}) / 2$; xCosH(0.3) = 1.04533851412886048164445463383234 57 but xCosH(xDiv(3,10)) = xCosH("0.3") = 1.0453385141288604850253090463229121.

xTanH(a[,D]) **Hyperbolic tangent** $\tanh a$

$\tanh a = (e^x - e^{-x}) / (e^x + e^{-x})$; xTanH("0.1",28) = 9.966799462495581711830508368E-2

xASinH(a[,D]) **Inverse hyperbolic sine** $\operatorname{arsinh} a$

$\operatorname{arsinh} a = \ln [a+\sqrt{(a^2+1)}]$; xASinH("0.1",28) = 0.09983407889920756332730312 47

xACosH(a[,D]) **Inverse hyperbolic cosine** $\operatorname{arcosh} a$

$\operatorname{arcosh} a = \ln [a+\sqrt{(a^2-1)}]$, $a > 1$;

xATanH(a[,D]) **Inverse hyperbolic tangent** $\operatorname{artanh} a$

$\operatorname{artanh} a = ½ \ln [(1+a)/(1-a)]$; xATanH(0.1,28) = 0.1003353477310755862429135451; xATanH("0.1",28) = 0.1003353477310755806357265521.

xAngleC(a[,D]) **Complement of angle α** $\pi/2-\alpha$

xAngleC(0.25,21) = 1.320796326794896619231321691639 7514; xSub(xPi2(21),0.25,21) = 1.3207963267948966192313216916397514.

xDegrees(a[,D]) **Converts radians into degrees** radians→degrees

xDegrees(xPi4()) = 45; xdegrees(xMult(4,xPi()),28) = 720.

xRadians(a[,D]) **Converts degrees into radians** degrees→radians

xRadians(180) = 3.1415926535897932384626433832795029 = xPi()

xAdjPi(a[,D]) **Adjusted angle, in radians, between –π and +π**

xAdjPi(xMult(5.75,xPi()),21) = -2.35619449019234492885 = xMult(3,xNeg(xPi4()),21)

xAdj2Pi(a[,D]) **Adjusted angle, in radians, between 0 and 2π**

xAdj2Pi(xMult(6.75,xPi()),21) = 2.35619449019234492885 = xMult(3,xPi4(),21)

D.4 Statistical operations

A is an array of numbers a_i in a contiguous row, column, or block.

xMean(A[,D]) **Mean** $(\sum_{i=1}^{n} a_i)/n$

xMean(1,3,4,10) = xMean({1,3,4,10},21) = xMean(C14:C17,21) = 4.5 when C14:C17 contains 1, 3, 4, and 10 respectively.

xMedian(A) **Median**

xMedian(1,3,4,10) = xMean(C14:C17,21) = 3.5 when C14:C17 contains 1, 3, 4, and 10 respectively. Do *not* specify D.

xGMean(A[,D]) **Geometric mean** $\sqrt[n]{a_1 \times a_2 \times \cdots \times a_n}$

xGMean({1,3,4,10},21) = xGMean(C14:C17,21) = 3.30975091964687310503 when C14:C17 contains 1, 3, 4, and 10 respectively; A must be an array or a named range.

xHMean(A[,D]) **Harmonic mean** $n \Big/ \sum_{i=1}^{n} \left(\frac{1}{a_i} \right)$

xHMean({1,3,4,10},21) = xHMean(C14:C17,21) = 2.37623762376237623762 when C14:C17 contains 1, 3, 4, and 10 respectively; A must be an array or a named range.

xQMean(A[,*D*]) **Quadratic mean** $(\sum_{i=1}^{n} a_i^2)/n$

xQMean({1,3,4,10},21) = xQMean(C14:C17,21) = 5.6124860801609l207838 when C14:C17 contains 1, 3, 4, and 10 respectively; A must be an array or a named range.

xStDev(A[,*D*]) **Standard deviation** $\sqrt{\frac{\sum_{i=1}^{n}(a_i - a_{av})^2}{n-1}}$

xStDev({3.1,3.2,3.3},21) = xStDev(B3:B5,21) = 9.9999999999999998667732E-2 when B3:B5 contains 3.1, 3.2, and 3.3 respectively; xStDev({"3.1","3.2","3.3"},21) = 0.1

xStDevP(A[,*D*]) **Population standard deviation** $\sqrt{\frac{\sum_{i=1}^{n}(a_i - a_{av})^2}{n}}$

xStDevP({3.1,3.2,3.3},21) = xStDevP(B3:B5,21) = 0.081649658092772494494 when B3:B5 contains 3.1, 3.2, and 3.3 respectively; xStDev({"3.1","3.2","3.3"},21) = 8.1649658092772603273 2E-2.

xVar(A[,*D*]) **(Sample) variance** $\frac{\sum_{i=1}^{n}(a_i - a_{av})^2}{n-1}$

xVar({3.1,3.2,3.3},21) = xVar(B3:B5,21) = 9.9999999999999997335465E-3 when B3:B5 contains 3.1, 3.2, and 3.3 respectively; xVar({"3.1","3.2","3.3"},21) = 0.01

xVarP(A[,*D*]) **Population variance** $\frac{\sum_{i=1}^{n}(a_i - a_{av})^2}{n}$

xVarP({3.1,3.2,3.3},21) = xVarP(B3:B5,21) = 6.6666666666666664489031E-3 when B3:B5 contains 3.1, 3.2, and 3.3 respectively; xVar({"3.1","3.2","3.3"},21) = 6.6666666666666666666667E-3.

xFact(*n*[,*D*]) **Factorial** $n!$

For *n* a positive integer; if not integer, *n* is rounded down to the next integer.
xFact(27) = 10888869450418352160768000000,
xFact(28) = 3.0488834461171386050150 4E+29,
xFact(1E7) = 1.2024234005159034561401534879443076E+65657059,
xFact(xFact(25)) = 3.5679279579588489448587652949509 ×
E+38400096332207799837905 2338.

xFact2(*n*[,*D*]) **Double factorial**

n odd: $(2n-1)!! = \prod_{i=1}^{n}(2i-1) = \frac{(2n)!}{n!2^k}$; *n* even: $(2n)!! = \prod_{i=1}^{n}(2i) = n!2^k$

xFact(27) = 10888869450418352160768000000,
xFact(28) = 3.0488834461171386050150 4E+29,
xMult(xFact2(27),xFact2(28)) = 3.0488834461171386050150 4E+29 = xFact(28).

xComb(*n*,*m*[,*D*]) **Binomial coefficient** $\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$

xComb(20,10) = 184756,
xComb(200,100) = 9.0548514656103281165404177077484164E+58,
xComb(2000,1000,45) = 2.0481516269894897143351625029808250 4439642489E+600
or, displayed in its full 600-decimal glory, as xComb(2000,1000, *D*) with $D \geq 601$.

xComb_Big(n,m[,D]) **Binomial coefficient for large numbers** $\binom{n}{m} = \dfrac{n!}{m!\,(n-m)!}$

xComb_Big(10000000,9000000,28) = 1.093540446065167765202685186E+1411814

xCorrel(A,B[,D]) **Correlation coefficient** $r_{AB} = \dfrac{v_{AB}}{s_A\ s_B}$

xCorrel({1,2,3,4,5,6},{7,5,8,6,9,7},21) = 0.377964473009227227215;
xDiv(xCovar(A12:A17,A19:A24),xMult(xStDevP(A12:A17),xStDevP(A19:A24)),21)
= 0.377964473009227227215, see (2.10.2), when A12:A17 = {1,2,3,4,5,6} etc.
xCorrel({1,2,3,4,5,6},{3,4,5,6,7,8},21) = 1. *A* and *B* are data sets, addressed either as a listing of their individual values (see the above examples) or by reference to their spreadsheet addresses ranges. Note: in this book we deal with physical laws, and a correlation coefficient $|r_{xy}| \leq 0.9$ is usually considered insignificant. However, in the social sciences, where there are often many complicating factors, and $|r_{xy}| = 0.9$ may be viewed as highly significant. It all depends on the context.

xCovar(n,m[,D]) **Covariance** $\dfrac{1}{N}\sum_{k=1}^{N}(a_{i,k} - a_{i,av})(a_{j,k} - a_{j,av})$

xCovar({1,2,3,4,5,6},{7,5,8,6,9,7}) = 0.83333333333333333333333333333333333,
xCorrel({1,2,3,4,5,6},{1,2.1,3,4,5,6}) = 0.99979520559482815691603169600455 99.

xStatis(A[,D]) **Univariate statistical summary of a data range *A***

Yields five parameters in row format (deposit the instruction with block-enter): number of data *N*; their mean; sample standard deviation; population standard deviation; and autocorrelation with lag 1 = $\Sigma_1^{n-1}\{(x_i - x_{av})\,(x_{i+1} - x_{av})\} / \Sigma_1^{n}\{(x_i - x_{av})^2$.
xStatis({1,2,3,4,5,6},18) = {6, 3.5, 1.87082869338697069, 1.70782512765993306, 0.5}.

xRand([,D]) **Random number between 0 and 1** $U(0, 1)$

xRand() = 0.36884713172912601715122290811538286.

xRandD(a,b[,D]) **Random number between *a* and *b*** $U(a, b)$

xRandD(4.1,4.3) = 4.19716314077377293399589081019878 38.
Note that *a* can be smaller or larger than *b*.

xRandI(a,b[,D]) **Random integer between *a* and *b***

xRandI(4.2,-11.3) = -2; *a* can be smaller or larger than *b*, and neither needs to be integer.

D.5 Least squares functions

xIntercept(**y**,**x**[,D]) **Intercept of least squares straight line with *y*-axis** a_0

xSlope(**y**,**x**[,D]) **Slope of least squares straight line** a_1

xRegLinCoef(**y**,**x**[,D][,*intercept*]) **Least squares coefficients** a_0 through a_p

y is the vector of *n* dependent variables; **x** is the vector of *n* (or the matrix of $n \times m$) independent variables; intercept forces the *y*-intercept through *y* = intercept for *x* = 0. The output yields the least squares coefficients, in row format.

xRegLinCov(**y**,**x**,**coef** [,D][,intercept]) **Least squares covariance matrix** CM

y is the vector of *n* dependent variables, **x** is the vector of *n* (or the matrix of $n \times m$) independent variables, coef refers to the output of xRegLinCoef, and intercept forces the *y*-intercept through *y* = intercept for *x* = 0. The output yields the covariance matrix.

xRegLinErr(y, x, coef [,*D*] [,*intercept*]) Standard deviations of LS coefficients s_0 through s_p

y is the vector of *n* dependent variables; **x** is the vector of *n* (or the matrix of *n*×*m*) independent variables; and intercept forces the *y*-intercept through *y* = intercept for *x* = 0. The output yields the standard deviations of the coefficients, in row format.

xRegLinEval(coef,*x* [,*D*]) Evaluating a least squares fit at a specified *x*-value

Coef refers to the output of xRegLinCoef, and *x* is the specific value at which the fitting function is to be evaluated.

xRegLinStat(y, x, coef [,*D*] [,*intercept*]) More statistical least squares information r^2 and s_f

y is the vector of *n* dependent variables, **x** is the vector of *n* (or the matrix of *n*×*m*) independent variables, coef refers to the output of xRegLinCoef, and intercept forces the *y*-intercept through *y* = intercept for *x* = 0. Outputs r^2 and s_f in row format.

xRegPolyCoef(y, x, *degree* [,*D*] [,*intercept*]) Least squares coefficients a_0 through a_p

y is the vector of *n* dependent variables; **x** is the vector of *n* independent variables; *degree* is the highest polynomial order; and intercept forces the *y*-intercept through *y* = intercept for *x* = 0. The default, intercept = TRUE, is to include a_0 in the analysis. In default mode (*D* = 35), xRegPolyCoef(B3:B84,C3:C84,10) aces the NIST LLS test Filip.dat (see exercise 11.13.3) provided that (1) the *y*-values in B3:B84, and the *x*-values in C3:C84, are in string format, i.e., preceded by an apostrophe, either manually or, faster, with the instruction xCStr(xRoundR(*number*,15)), and (2) the output data z are copied with the instruction = xCDbl(xRoundR((*address*,15)) where *number* is an input value read from the spreadsheet, and *address* an output result displayed there. If (1) and/or (2) are disregarded, the output may 'only' agree to p*E* = 14.0 instead of to p*E* = 15. Use a block-enter; the output is in row format.

xRegPolyErr(y, x, *degree*, coef [,*D*] [,*intercept*]) Standard deviations of LS coefficients s_0 through s_p

y is the vector of *n* dependent variables; **x** is the vector of *n* independent variables; *degree* is the highest polynomial order; and the optional intercept forces the *y*-intercept through *y* = intercept for *x* = 0. Do *not* forget to enter the coefficients from xRegPolyCoef! The default, intercept = TRUE, is to include a_0 in the analysis. The output yields the standard deviations *s* of the coefficients.

xRegPolyStat(y, x, *degree*, coef [,*D*] [,*intercept*]) More statistical least squares information r^2 and s_f

y is the vector of *n* dependent variables; **x** is the vector of *n* independent variables; *degree* is the highest polynomial order; and the optional intercept forces the *y*-intercept through *y* = intercept for *x* = 0. Do *not* forget to enter the coefficients from xRegPolyCoef! The default, intercept = TRUE, is to include a_0 in the analysis. The output yields r^2 and the standard deviations s_f of the over-all fit of the model function to the data.

xRegrL(y, x [,*D*] [,*intercept*] [,*ε*] [,tol]) Least squares coefficients obtained by SVD a_0 through a_p

This function uses SVD rather than the traditional pseudo-inverse; **y** is the vector of *n* dependent variables; **x** is the vector of *n* (or the matrix of *n*×*m*) independent variables; and intercept forces the *y*-intercept through *y* = intercept for *x* = 0; *ε* is the resolution (default: 10^{-D}); tol (for tolerance, default: 0) specifies the largest absolute value that should be considerd round-off error and therefore can be set to 0 (similar to *Tiny*).

xRegrLC(y, x [,*cf*] [,*D*] [,*intercept*] [,*ε*] [,tol]) Least squares coefficients of complex data by SVD

The extension of xRegrL to complex data. *cf* defines the complex format used; default = 1 for split format.

D.6 Statistical functions

Note: even though their names have the prefix x, the functions xGamma, xGammaLn, xGammaLog, xGammaQ and xBeta *used to be* double precision. John Beyers has now converted them to fully extended precision. If you have used them earlier in programs that plotted their output, make sure to use them now within an x CDbl() command so that their outputs will still be read properly by the graph. While Excel's functions treat numerical strings as numbers, Excel's graphs do not recognize such strings as valid input data.

xGamma(x[,D]) **Gamma function** $\Gamma(x)$

$\Gamma(n) = \pm\infty$ for n a non-positive integer, $\Gamma(n) = (n-1)!$, $\Gamma(1/2) = \sqrt{\pi}$.
xGamma(-101.01,50) = 1.01316813059536869258112405851033723855160984E-158,
xGamma(0.5,40) = xSqr(xPi(40),40) = 1.772453850905516027298167483341145182798,
xGamma(1000,50) = 4.0238726007709377354370243392300398571937486421071E+2564.
xGamma(0.000000001) = 999999999.42278427380593167581398533 for $D_{default}$ = 35.

xGammaLn(x[,D]) **Natural logarithm of the gamma function** $\ln\Gamma(x)$

xGammaLn(0.5,70) = xLn(xGamma(0.5,70),70) = xLn(xSqr(xPi(70),70),70) =
0.5723649429247000870717136756765293558236474064576557857568115357360689.

xGammaLog(x[,D]) **10-based logarithm of the gamma function** $\log\Gamma(x)$

xGammaLog(1000,70) = xLog(xGamma(1000,70),,70) = xLog(xFact(999,70),,70) =
2567.604644222132848771423057804523691677114513162463461310044207289183.

xGammaQ(x_1,x_2[,D]) **Ratio of two gamma functions** $\Gamma(x_1)/\Gamma(x_2)$

xGammaQ(0.5,1000,25) = 4.404845845680923991421408E-2565, xGammaQ(0.5,1000,65) =
xDiv(xGamma(0.5,65),xGamma(1000,65),65) = xDiv(xSqr(xPi(70),70),xFact(999,65),65) =
4.4048458456809239914214080519445322777708010072456291610680796307E-2568.

xBeta(x,y[,D] **Complete Beta function** $B(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}\,dt$

where B(x, y) = $\Gamma(x)\,\Gamma(y) / \Gamma(x+y)$, provides an easy check on the function.
Let B1, B2, etc are cell addresses, then for x = 1.2, y = 3.4 and $x+y$ = 4.6 we have
B1: xGamma("1.2",50) = 0.91816874239976061064095165518583040068682199965868,
B2: xGamma("3.4",50) = 2.9812064268103329717913686054439211818356413783808,
B3: xGamma("4.6",50) = 13.381285870932449355274522094100253203034374722681,
B4: xDiv(xMult(B1,B2,50),B3,40) = 0.2045581106435018057463802648086835068269,
finally in B5: xBeta("1.2","3.4") = 0.204558110643501805746380264808868351, and
xBeta("1.2","3.4",50) = 2.0455811064350180574638026480868350682689657512436E-1.

xZeta(x[,D]) **Riemann zeta function** $\zeta(x)$

$\zeta(x) = 0$ for x a negative even integer, $\pm\infty$ for $x = 1$: xZeta(1) = #VALUE!,
xZeta(-101,50) = -7.2612008803606716303677281510706847232235031164793E+78,
xZeta(-9,60) = -7.57575757575757575757575757575757575757575757575757575757576E-3,
xZeta(0.999999) = -999999.42275565224980209723357694814, xZeta(0,600)= 0.5,
xZeta(1.000001) = 1000000.57729800435533, xZeta(50,21) = 1.00000000000000088818

D.7 Statistical distributions

type = 0 or FALSE (default) for the probability density f; type = 1 or TRUE for the corresponding cumulative distribution F.

xNormal(x,μ,σ [,type] [,D]) **Normal distribution** $f(x,\mu,\sigma) = \dfrac{\exp[-(x-\mu)^2/(2\sigma^2)]}{\sigma\sqrt{2\pi}}$

Extended-precision version of Excel's NORMDIST:
xNormal(-1000,7,0.5,0,40) = 1.354506334060962146056106217684345437524E-880792,
xNormal(-10,7,0.5,,48)=7.58105280018573627361442359669669459333212880463E-252,
xNormal(0,7,0.5) = 2.1932131187779426125067829785218123E-43,
xNormal(10,7,0.5)=xNormal(10,7,0.5,0)=1.2151765699646570973992615481363651E-8,
xNormal(45000,7,0.5) = 1.0582474958611311359386964761976518E-1306737131;
xNormal(-9876,7,0.5,1) = 9.6676869595648374723957510755689408E-84838294,
xNormal(-10,7,0.5,1,45)=1.1138987855743793865819505555930236035018809E-253,
xNormal(3,7,0.5,1) = 6.2209605742717841235159951725881884E-16,
xNormal(12,7,0.5,1,50) = 0.999999999999999999999999238014697583947393402665675,
xNormal(14.6,7,0.5,1,50) = 1.

xNormalS$(z[,\text{type}][,D])$ **Standard normal distribution** $f(z)=\dfrac{\exp[-z^2/2]}{\sqrt{2\pi}}$

Extended-precision version of NORM.S.DIST of Excel 2010, with zero mean and unit st. dev.: xNormalS(-1000,0,16) = xNormal(-1000,,16) = 2.290648437187064E-2171486, xNormalS(-10,0) = xNormal(-10,,1) = 7.69459862670641934633903358004187 72E-23, xNormalS(0,0) = xNormal(0) = 0.398942280401432677939946059934381 87, xNormalS(10,0) = xNormal(10) = 7.6945986267064193463390335800418772E-23, xNormalS(1000,0,21) = xNormal(1000,,21) = 2.29064843718706368675E-217148; xNormalS(-1000, 1,21) = 7.0452236580171781353528161610508209E-217586, xNormalS(-10, 1) = 7.6198530241605260659733432515993084E-24, xNormalS(0, 1) = 0.5, xNormalS(10,1) = 0.9999999999999999999999923801469 7584, xNormalS(1000, ,1) = 1.

xBinomial$(k,n,p[,type][,D])$ **Binomial distribution** $f(k,n,p)=\dfrac{n!\,p^k(1-p)^{n-k}}{k!(n-k)!}$

$k>0$, $n>0$, $p\neq 1$; typically, k and n are integer, with $k\leq n$, and $0\leq p\leq 1$. xBinomial(10,8,0.7) = xBinomial(10,8,0.7,0) = 0.31386138777777748573834678592351 65, xBinomial(100,80,0.7) = 9.2763880344798178286504175936858 76E-6, xBinomial(100,80,0.5) = 8.2718061255302767487140869206996285E-25; xBinomial(100,80,0.7,1) = 1.4483734111111107223573231024739338.

xLogistic$(x,\mu,s[,type][,D])$ **Logistic distribution** $f(x,\mu,s)=\dfrac{\exp[-(x-\mu)/s]}{s\left[1-\exp[-(x-\mu)/s\right]^2}$

$s>0$. xLogistic(1,1,0.5) = xLogistic(1,1,0.5,0) = 0.5, xLogistic(0.1,1,0.5) = 0.24345868057417078321107988837114377, xLogistic(0.5,1,0.5) = 0.39322386648296370507484946717181805, xLogistic(2,1,0.5) = 0.20998717080701303469724836952085072, xLogistic(10,1,0.5) = 3.0459958561616145970616505932086991E-8; xLogistic(0.1,1,0.5,1) = 2.2648142878370235005474413358600984E-2, xLogistic(0.5,1,0.5,1) = 0.14973849934787756480856989948056052, xLogistic(1,1,0.5,1) = 0.38079707797788244405972914130239 68, xLogistic(2,1,0.5,1) = 0.76159415595576488811945828260479359, xLogistic(10,1,0.5,1) = 0.88079706274790293129938019689418799.

xLogNorm$(x,\mu,\sigma[,type][,D])$ **Lognormal distribution** $f(x,\mu,\sigma)=\dfrac{\exp[-(\ln x-\mu)^2/(2\sigma^2)]}{x\sigma\sqrt{2\pi}}$

$x>0$. xLogNorm(0.1,1,0.5,,28) = 2.680284603881915428668405659E-9, xLogNorm(1,1,0.5,,20) = xLogNorm(1,1,0.5,0,20) = 0.1079819330263761039, xLogNorm(10,1,0.5) = 2.6802846038819136119445616856654493E-3; xLogNorm(0.01,1,0.5,1) = 1.8139777883515688426389318351031983E-29, xLogNorm(1,1,0.5,1) = 2.2750131948179207200282637166533437E-2, xLogNorm(50,1,0.5,1) = 0.99999999712801243139612132792027695.

xMaxwell$(x,a[,type][,D])$ **Maxwell distribution** $f(x,a)=4x^2e^{-ax^2}\sqrt{a^3/\pi}$

Note that different authors define this distribution differently; here we use $a=m/2kT$ where m is mass, k is the Boltzmann constant, and T the absolute temperature. xMaxwell(0.02,1,,28) = xMaxwell(0.02,1,1,28) = 9.023423245495783540059193 24E-4, xMaxwell(1,1) = 0.83021499484118940668053649888267473, xMaxwell(5,1) = 7.8354332655086676541216841613105858E-10; xMaxwell(0.02,1,1) = 6.0165781054863134065267945144474112E-6, xMaxwell(1,1,1) = 0.42759329552912016600095238564127189, xMaxwell(5,1,1) = 0.99999999992010820755048528860859481.

xRayleigh$(x, \sigma[,type][,D])$ **Rayleigh distribution** $f(x,\sigma) = xe^{-x^2/2\sigma^2}/\sigma^2$

$x \geq 0$. xRayleigh(0.01,1,,60) = xRayleigh(0.01,1,0,60) = 9.99950001249979187740640069032792883187110465298047034833696E-3, xRayleigh(1,1) = 0.60653065971263342360379953499118045, xRayleigh(10,1) = 1.9287498479639177830173428165270126E-21; xRayleigh(0.01,1,1) = 4.9998750020833075000834902025581322E-5, xRayleigh(1,1,1) 0.39346934028736657639620046500881955, xRayleigh(10,1,1) = 0.99999999999999999999980712501520361.

xWeibull$((x,k,\lambda[,type][,D])$ **Weibull distribution** $f(x,k,\lambda) = \frac{kx^{k-1}}{\lambda^k} e^{-(x/\lambda)^k}$

$x \geq 0$. xWeibull(0.01,1,0.5,,60) = xWeibull (0.01,1,0.5,0,60) = 1.96039734661351060362544885658056190613736410084899623650741, xWeibull(1,1,0.5) = 0.27067056647322538378799898994496881, xWeibull(10,1,0.5) = 4.1223072448771156559318807603116 42E-9; xWeibull(0.01,1,0.5,1) = 1.9801326693244698187275571709719047E-2, xWeibull(1,1,0.5,1) = 0.86466471676338730810600050502751 56, xWeibull(10,1,0.5,1) = 0.99999999793884637756144217203405962.

D.8 Operations with complex numbers

Use the Configuration dialog box (under the X-Edit button on the XN toolbox) to select either i or j for √(–1). Here we will use j. Complex numbers will be denoted by $z = a + j\,b$, and must be defined in terms of their separate, real and imaginary components, a and b. The notation has been simplified by allowing single-cell or split formatting of both input and output, simply by highlighting a single cell or specifying two (horizontally or vertically) adjacent cells, see Fig. 11.12.6. Here we will use (except for the first three functions) the default (1, horizontally split) format for both input and output. (Note that this simplified notation applies only to operations on individual complex numbers, as considered in this section; for arrays of complex numbers this short notation would be ambiguous, and *cf* must be specified when it differs from the chosen default.)

In B1 we have used =xCplx(3,4) to place 3+4j, and in E1 likewise =xCplx("5.6","7.8") to deposit 5.6+7.8j. The complex numbers $z_1 = 3 + 4\,j$ and $z_2 = 5.6 + 7.8j$ are stored as strings in row 2: as ′3 in B2, ′4 in C2, ′5.6 in E2, and ′7.8 in F2. They are also stored as regular spreadsheet numbers in row 3, i.e.,as 3 in B3, as 4 in C3, as 5.6 in E3, and as 7.8 in F3. All examples will assume $D = 35$ unless otherwise indicated. Array output in adjacent cells will be shown as separated by a comma, and must of course be entered with the block enter combination Ctrl◡Shift◡Enter.

xCplx$(z[,D])$ **Converts Re(z) and Im(z) into a complex single-cell format**

xCplx(3,4) = 3+4j; xCplx("5.6","7.8") = 5.6+7.8j

xReal$(z[,D])$ **Real part of a single-cell complex number** $a = \mathrm{Re}(a + jb)$

xReal(xCplx(3,4)) = 3

xImag$(z[,D])$ **Imaginary part of a single-cell complex number** $b = \mathrm{Im}(a + jb)$

xImag(xCplx(3,4)) = 4

xCplxAbs$(z[,D])$ **Absolute value of single-cell format** $|z| = |a + jb| = \sqrt{a^2 + b^2}$

xCplxAbs(B1) = xCplxAbs(B2:C2) = xCplxAbs(xCplx(3,4)) = 5

xCplxArg$(z[,D])$ **Complex argument** $\arg(z) = \arctan(b/a)$

xCplxArg(B1,70) = xCplxArg(B2:C2,70) = xCplxArg(B2:C2,70) = 0.9272952180016122324285124629224288040570741085722405276218661774403957

xCplxNeg$(z[,D])$ **Negation** $-z = -(a + j\,b) = -a - j\,b$

xCplxNeg(B1) = –3–4j; xCplxNeg(B2:C2) = xCplxNeg(B3:C3) = –3, –4

xCplxConj$(z[,D])$ **Conjugate** $z^* = a - j\,b$

xCplxConj(B1) = 3–4j; xCplxConj(B2:C2) = xCplxConj(B3:C3) = 3, –4

xCplxAdd$(z_1,z_2[,D])$ **Addition** $z_1+z_2 = (a_1 + a_2) + j\,(b_1 + b_2)$

xCplxAdd(B1,E1) = 8.6+11.8j; xCplxAdd(B2:C2,E2:F2) = 8.6, 11.8;
xCplxAdd(B3:C3,E3:F3,21) = 8.59999999999999964473, 11.7999999999999998224

xCplxSub$(z_1,z_2[,D])$ **Subtraction** $z_1-z_2 = (a_1 - a_2) + j\,(b_1 - b_2)$

xCplxSub(B1,E1) = -2.6-3.8j; xCplxSub(B2:C2,E2:F2) = -2.6, -3.8;
xCplxSub(B3:C3,E3:F3,21) = -2.59999999999999964473, -3.79999999999999982236

xCplxMult$(z_1,z_2[,D])$ **Multiplication** $z_1\,z_2 = (a_1a_2-b_1b_2) + j\,(a_1b_2+a_2b_1)$

xCplxMult(B1,E1) = -14.4+45.8j; xCplxMult(2:C2,E2:F2) = -14.4, 45.8;
xCplxMult(B3:C3,E3:F3,21) = -14.4000000000000003553, 45.7999999999999998046

xCplxPow$(z,n[,D])$ **Integer power** $z^n = \sqrt{a^2+b^2}\ \exp[n \arctan(a/b)]$

xCplxPow(B1,2) = -7+24j; xCplxPow(B2:C2,2) = xCplxPow(B3:C3,2) = -7, 24

xCplxRoot$(z,n\,[,D])$ **Integer root** $z^{1/n} = \sqrt[n]{a+jb}$

xCplxRoot(B1,2) = xCplxRoot(B3:C3,2) = 2+j, –2–j; xCplxRoot(E1,2) =
xCplxRoot(E2:F2,2,21) = 2.75699864955772539922+1.41458175927131251328j in one cell, and -2.75699864955772539922-1.41458175927131251328j in the next; likewise, xCplxRoot(E3:F3,2,21) = 2.75699864955772533513+1.41458175927131251395j in one cell, and -2.75699864955772533513-1.41458175927131251395j in the next.

xCplxSqr$(z\,[,D])$ **Square root** $z^{½} = \sqrt{a+jb}$

xCplxSqr(B1) = 2+j; xCplxSqr(B2:C2) = xCplxSqr(B3:C3) = 2, 1
xCplxSqr(E1,19) = xCplxSqr(E2:F2,19) = 2.756998649557725399, 1.414581759271312513
xCplxSqr(E2:F2,19) = 2.756998649557725335, 1.414581759271312514

xCplxDiv$(z_1,z_2[,D])$ **Division** $z_1/z_2 = \dfrac{(a_1a_2+b_2b_2)+j(-a_1b_2+a_2b_1)}{a_2^2+b_2^2}$

xCplxDiv(B2:C2,E2:F2,21) = 0.520607375271149674 62, -1.08459869848156182213E-2
xCplxDiv(B3:C3,E3:F3,21) = 0.520607375271149693469, -1.08459869848156286485E-2

xCplxInv$(z[,D])$ **Inversion** $1/z = \dfrac{1}{a+jb} = \dfrac{a-jb}{a^2+b^2}$

xCplxInv(B1) = 0.12-0.16j when placed in one cell; when placed in two cells,
xCplxInv(B1) = xCplxInv(B2:C2) = xCplxInv(B3:C3)= 0.12, –0.16;
xCplxInv(E2:F2,21) = 0.0607375271149674 62039, -8.45986984815618221258E-2
xCplxInv(E3:F3,21) = 6.07375271149674626325E-2, -8.45986984815618263928E-2

xCplxExp$(z[,D])$ **Exponential** $e^z = e^a \cos(a) + j\,e^b \cos(b)$

xCplxExp(E1,28)=14.5909705439244867111507082 5+270.0324895489463602631116766j
xCplxExp(E2:F2,21) = 14.5909705439244867112, 270.032489548946360263
xCplxExp(E3:F3,21) = 14.5909705439245294948, 270.032489548946261736

xCplxLn$(z[,D])$ **Natural logarithm** $\ln z$

In one cell: xCplxLn(E1,21) = 2.26198006528127407189+0.948125538037829317382j,
in two: xCplxLn(E1,70) = xCplxLn(E2:F2,70) =
2.261980065281274071885982930024169450064511264424455256333274238956,
0.9481255380378293173815983411752882151513212835055453722109185788 09796;
xCplxLn(E3:F3,25) = 2.2619800652812740352799 31, 0.9481255380378293366479415

xCplxLog$(z,b[,D])$ **Logarithm to base *b*** $\log_b(z) = \ln(z)/\ln(b)$

Careful: xCplxLog(E1,,21) = 0.982365460526814654246+0.411765689321380975201j which assumes that the non-specified base is 10, whereas xCplxLog(E1,21) = 0.74296711932683140068942681618616545+0.31142011840346337425331216516 15121j for $\log_{21}(z)$ with the default number of decimals, here 35. If you need the 10-based log, use:

xCplxLog10(z[,D]) **10-based logarithm** $\log(z) = \log_{10}(z) = \ln(z) / \ln(10)$

In two cells: xCplxLog10(E1,25) = xCplxLog10(E2:F2,70) =
0.9823654605268146701442103566059571819809685627552493938363525724175293,
0.4117656893213809668342048131500706165852505219562462786489498073138795;
xCplxLog10(E3:F3,25) = 0.982365460526814654246404, 0.4117656893213809752014713

xCplxLog2(z[,D]) **2-based logarithm** $\log_2(z) = \ln(z) / \ln(2)$

xCplxLog2(E2:F2,25)= 3.263347422770987814603177, 1.367856011867356597452088
xCplxLog2(E3:F3,25) =3.263347422770987761791808, 1.367856011867356625247546

xCplxSin(z [,D]) **Sine** sin (z)

xCplxSin(E2:F2,25)= -770.335431725789249221414, 946.4236495468643587804233
xCplxSin(E3:F4,25)= -770.3354317257894486197362, 946.4236495468639169837239

xCplxCos(z[,D]) **Cosine** cos (z)

xCplxCos(E2:F2,25)= 946.4239673233332876174362, 770.3351730737666499652134
xCplxCos(E3:F4,25)= 946.4239673233328458207013, 770.3351730737668493633767

xCplxTan(z[,D]) **Tangent** tan (z)

xCplxTan(E2:F2,24) = -3.2877408328165508373533E-7, 0.999999931837917456442433
xCplxTan(E3:F4,24) = -3.2877408328165524897145E-7, 0.999999931837917456442642

xCplxASin(z[,D]) **Inverse sine** arcsin(z)

xCplxASin(E2:F2,24)= 0.620108349818012666322386, 2.95600293720697536127987
xCplxASin(E3:F4,24)= 0.620108349818012646903013, 2.95600293720697532483704

xCplxACos(z[,D]) **Inverse cosine** arccos (z)

xCplxACos(E2:F2,24)= 0.950687976976883952908935, -2.95600293720697536127987
xCplxACos(E3:F4,24)= 0.950687976976883972328309, -2.95600293720697532483704

xCplxATan(z[,D]) **Inverse tangent** arctan (z)

xCplxATan(E2:F2,24) = 1.50969874144921909210512, 8.44859768081672965961273E-2
xCplxATan(E3:F4,24) = 1.50969874144921909146551, 8.44859768081673008713949E-2

xCplxSinH(z[,D]) **Hyperbolic sine** sinh (z)

xCplxSinH(E2:F2,24) = 7.29538551206624025612712, 135.018091013076278249296
xCplxSinH(E3:F4,24) = 7.29538551206626164758967, 135.018091013076228986589

xCplxCosH(z[,D]) **Hyperbolic cosine** cosh (z)

xCplxCosH(E2:F2,24) = 7.29558503185824645502359, 135.014398535870082013816
xCplxCosH(E3:F4,24) = 7.29558503185826784721294, 135.014398535870032749833

xCplxTanH(z [,D]) **Hyperbolic tangent** tanh (z)

xCplxTanH(E2:F2,24) = 1.00002718952482972149739, 2.94696926173800019499848E-6
xCplxTanH(E3:F4,24) = 1.00002718952482972151566, 2.94696926173801194879248E-6

xCplxASinH(z[,D]) **Inverse hyperbolic sine** arsinh (z)

xCplxASinH(E2:F2,24) = 2.95426910101325167773266, 0.945549735665370431458319
xCplxASinH(E3:F4,24) = 2.95426910101325164096493, 0.945549735665370450568323

xCplxACosH(z[,D]) **Inverse hyperbolic cosine** arcosh (z)

xCplxACosH(E2:F2,24) = 2.95600293720697536127987, 0.950687976976883952908935
xCplxACosH(E3:F4,24) = 2.95600293720697532483704, 0.950687976976883972328309

xCplxATanH(z[,D]) **Inverse hyperbolic tangent** artanh (z)

xCplxATanH(E2:F2,24)=6.03776070460713078765599E-2, 1.48608980485008744950066
xCplxATanH(E3:F4,24)=6.03776070460713084243571E-2, 1.48608980485008744524278

xCplxPolar(z [,D]) **Convert to polar** $z = \rho e^{j\theta}$

xCplxPolar(E1,,35) = xCplxPolar(E2:F2,,35) = xCplxPolar(E2:F2) =
9.60208310732624309126871450256650, 0.94812553803782931738159834117528822
xCplxPolar(E3:F4,,25) = 9.602083107326242739774361, 0.9481255380378293366479415

xCplxRect(z[,D]) **Convert to rectangular** $z = \rho\{\cos(\theta) + j\sin(\theta)\}$

xCplcRect(xCplxPolar(E1,,35),21) = 5.6, 7.8000000000000000000000000000000001,
xCplcRect(xCplxPolar(E2:F2,35),21) = 5.6, 7.8000000000000000000000000000000001,
xCplcRect(xCplxPolar(E3:F3,21),21) = xCplcRect(xCplxPolar(E3:F3,500),21) =
5.5999999999999999964472863212, 7.7999999999999999982236431606.

D.9 Matrix and vector operations

D.9.1 Standard operations

We denote vectors as **v** with elements v_i. Matrices are either square real **S**, rectangular real **R** or (square or rectangular) complex **C**, all with elements m_{ij}. *cf* denotes the *c*omplex *f*ormat used: 1 for split (= default), 2 for interspersed, 3 for Excel's string format. The number of rows of a vector or matrix is indicated by r, the number of colums by c. Absolute element values m_{ij} smaller than ε are set to zero as probable rounding errors; the default value for ε is 1E–D. Noninteger numbers should be placed between quotation marks when their exact rather than their Excel-stored values are to be used. We will use the compact matrix notation $\{m_{11}, m_{12}, \ldots; m_{21}, m_{22}, \ldots; m_{31}, m_{32}, \ldots; \ldots\}$ to denote a matrix with elements m_{ij} where commas separate individual elements in the same row, and semicolons separate different rows. $D_{default} = 35$.

xMAbs(**R**[,D]) **Absolute value of a real matrix** $\|\mathbf{R}\| = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}(m_{i,j})^2}$

xMAbs({1,2;"3.1",-4}) = 5.5326304774492214410001161638167525;
xMAbs({1,2; 3.1,-4}) = 5.5326304774492214907658309046178264

xMAbsC(**C**[,cf][,D]) **Absolute value of a complex matrix** $\|\mathbf{C}\| = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}(m_{i,j})^2}$

xMAbsC({1,2,0,-3; "3.1",-4,-1,0;6,5,2,1}) = 10.325211862233142574713865204941196;
xMAbsC({1,2,0,-3; 3.1,-4,-1,0; 6,5,2,1}) = 10.325211862233142601380176151460634

xMAdd($\mathbf{R}_1$,$\mathbf{R}_2$[,D]) **Addition of two real matrices** $\mathbf{R}_1 + \mathbf{R}_2$

$\mathbf{R}_1$ and $\mathbf{R}_2$ must have the same size $m\times n$, i.e., $c_1 = c_2$ *and* $r_1 = r_2$.

xMAddC($\mathbf{R}_1$,$\mathbf{R}_2$[,cf][,D]) **Addition of two complex matrices** $\mathbf{R}_1 + \mathbf{R}_2$

$\mathbf{R}_1$ and $\mathbf{R}_2$ must have the same size $m\times n$.

xMSub($\mathbf{R}_1$,$\mathbf{R}_2$[,D]) **Subtraction of two real matrices** $\mathbf{R}_1 - \mathbf{R}_2$

$\mathbf{R}_1$ and $\mathbf{R}_2$ must have the same size $m\times n$.

xMSubC($\mathbf{R}_1$,$\mathbf{R}_2$[,cf][,D]) **Subtraction of two complex matrices** $\mathbf{R}_1 - \mathbf{R}_2$

$\mathbf{R}_1$ and $\mathbf{R}_2$ must have the same size $m\times n$.

xProdScal($\mathbf{v}_1$,$\mathbf{v}_2$[,D]) **Scalar product of two vectors (or matrices)** $\mathbf{v}_1 \bullet \mathbf{v}_2$

$\mathbf{v}_1$ and $\mathbf{v}_2$ must have the same size m. The scalar product is zero if $\mathbf{v}_1$ and $\mathbf{v}_2$ are perpendicular. This function can also be applied to two matrices $\mathbf{R}_1$ and $\mathbf{R}_2$ where $c_1 = r_2$ in which case xProdScal($\mathbf{R}_1$,$\mathbf{R}_2$) yields the product $\mathbf{R}_1^{\mathrm{T}}\,\mathbf{R}_2$

xProdScalC($\mathbf{v}_1$,$\mathbf{v}_2$[,cf][,D]) **Complex scalar product of two vectors** $\mathbf{v}_1 \bullet \mathbf{v}_2$

$\mathbf{v}_1$ and $\mathbf{v}_2$ must have the same size m. The scalar product is zero if $\mathbf{v}_1$ and $\mathbf{v}_2$ are perpendicular.

xProdVect($\mathbf{v}_1$,$\mathbf{v}_2$[,D]) **Vector product** $\mathbf{v}_1 \times \mathbf{v}_2$

$\mathbf{v}_1$ and $\mathbf{v}_2$ must have the same size m.

xMMult($\mathbf{R}_1$,$\mathbf{R}_2$[,D]) **Multiplication of two real matrices** $\mathbf{R}_1\,\mathbf{R}_2$

When $\mathbf{R}_1$ is $m\times p$, $\mathbf{R}_2$ must be $p\times n$, i.e., $c_1 = r_2$.

xMMultC(C₁,C₂[,*cf*][,*D*]) Multiplication of two complex matrices — $\mathbf{C}_1\ \mathbf{C}_2$

When $\mathbf{R}_1$ is $m\times p$, $\mathbf{R}_2$ must be $p\times n$, i.e., $c_1 = r_2$.

xMMultS(R,*a*[,*D*]) Multiplication of a real scalar *a* and a real matrix R — $a\,\mathbf{R}$

Note the order of terms in the argument: first the matrix **R**, then the scalar *a*, regardless of the matrix size. *a***R** will have the size of **R**.

xMMultSC(R,*z*[,*cf*][,*D*]) Multiplication of a complex scalar *z* and a complex matrix C — $z\,\mathbf{C}$

Note he order of terms in the argument: first the matrix **C**, first, then the scalar *z*, regardless of The matrix size. This order is the reverse of that in the function name. *z***C** will have the size of **C**.

xMPow(S,*n*[,*D*]) Integral power of a square, real matrix S — $\mathbf{S}^n$

n must be a positive integer.

xMPowC(C,*n*[,*cf*][,*D*]) Integral power of a complex matrix C — $\mathbf{C}^n$

n must be a positive integer.

xMInv(S[,*D*]) Inversion of a square real matrix — $\mathbf{S}^{-1}$

Uses Gauss-Jordan diagonalization with partial pivoting.

xMInvC(S[,*cf*][,*D*]) Inversion of a square complex matrix — $\mathbf{C}^{-1}$

Uses Gauss-Jordan diagonalization with partial pivoting.

xMDivS(R,*a*[,*D*]) Division of a real matrix R by a real scalar *a* — $\mathbf{R}\,/\,a$

Note the order of terms in the argument: first the matrix **R**, then the scalar *a*, regardless of the matrix size. **R** / *a* will have the size of **R**.

xMPseudoInv(R[,*D*]) Pseudo-inverse of a rectangular real matrix — $\mathbf{R}^+ = \mathbf{V}\,\boldsymbol{\Sigma}^{-1}\,\mathbf{U}^{\mathrm{T}}$

based on SVD. When **R** is $m\times n$, $\mathbf{R}^+$ is $n\times m$. When R is square and nonsingular, its pseudoinverse is equal to its inverse. Uses Gauss-Jordan diagonalization with partial pivoting.

xMPseudoInvC(C[,*cf*][,*D*]) Pseudo-inverse of a complex matrix — $\mathbf{C}^+ = \mathbf{V}\,\boldsymbol{\Sigma}^{-1}\,\mathbf{U}^{\mathrm{H}}$

based on SVD. When **R** is $m\times n$, $\mathbf{R}^+$ is $n\times m$. When R is square and nonsingular, its pseudoinverse is equal to its inverse. Uses Gauss-Jordan diagonalization with partial pivoting.

xMExp(S[,*n*][,*D*]) Exponentiation of a square real matrix — $e^{\mathbf{S}}$

$\mathrm{Exp}(\mathbf{S}) = 1 + \mathbf{S} + \mathbf{S}^2/2 + \mathbf{S}^3/6 + \mathbf{S}^4/24 + \ldots + \mathbf{S}^n/n!$

When *n* is deleted, the series is continues until it converges.

xMExpC(S[,*n*][,*cf*][,*D*]) Exponentiation of a square complex matrix — $e^{\mathbf{C}}$

$\mathrm{Exp}(\mathbf{S}) = 1 + \mathbf{S} + \mathbf{S}^2/2 + \mathbf{S}^3/6 + \mathbf{S}^4/24 + \ldots + \mathbf{S}^n/n!$

When *n* is deleted, the series is continues until it converges.

xMExpErr(S,*n* [,*D*]) Error term in xMExp — $\left\|\mathbf{S}^n/n!\right\|$

Note that *n* is required.

xMExpErrC(C,*n*[,*cf*][,*D*]) Error term in xMExpC — $\left\|\mathbf{C}^n/n!\right\|$

Note that *n* is required.

xMMopUp(S[,*errMin*][,*cf*][,*D*]) Cleans up matrix errors close to zero

Replaces matrix elements smaller than ErrMin or ε by 0.

D.9.2 More sophisticated matrix operations

xMDet(S[,*D*]) Determinant of a square real matrix — $|\mathbf{S}|$

Uses Gauss- Jordan diagonalization with partial pivoting. Returns "?" when **S** is singular.

xMDet({1,2;"3.1",-4}) = -10.2

xMDet({1,2; 3.1,-4}) = -10.2000000000000017763568394

xMDetC(C[,*D*]) Determinant of a square complex matrix — $|\mathbf{C}|$

xMDetC({1,2,0,-3;"3.1",-4,1,7}) = -13.2, 14.3; xMDetC({1,2,0,-3;3.1,-4,1,7}) = -13.2000000000000017763568394, 14.3000000000000026645352591

xMCond(**R**[,*D*]) **Condition number of a real matrix** κ

Based on SVD. xMCond({1,2;"3.1",-4}) = 2.6191817659615200272394889923128097;
xMCond({1,2;3.1,-4}) = 2.6191817659615200292582110124456817

xMCondC(**C**[Cformat,*D*,ε,tol]) **Condition number of a complex matrix** κ

Based on SVD. xMCondC({1,2,0,-3;"3.1",-4,1,7},,21) = 4.37608205969300766727, 4.37608205969300766727; xMCondC({1,2,0,-3; 3.1,-4,1,7},,21) = 4.37608205969300761817, 4.37608205969300761817

xMpCond(**R**[Cformat,*D*,ε,tol]) **–log$_{10}$ of the condition number of a real matrix** $-\log_{10}(\kappa)$

xMpCond({1,2;"3.1",-4}) = -0.41816563863710134091248426474409013;
xMpCond({1,2; 3.1,-4}) = -0.41816563863710134124721469286252258

xMpCondC(**C**[Cformat,*D*,ε,tol]) **–log$_{10}$ of the condition number of a complex matrix** $-\log_{10}(\kappa)$

xMpCondC({1,2,0,-3;"3.1",-4,1,7},,2) = xMpCondC({1,2,0,-3;3.1,-4,1,7},,2) = -0.64

xMNormalize(**R**[,normtype][,tiny][,*D*]) **Normalize a real matrix** $v_i \Big/ \sqrt{\sum v_i^2}$

Normtype: all nonzero vertical vectors normalized; default = 2 for Euclidean norm.

$\mathbf{R} = \begin{bmatrix} 3 & 6.1 \\ 4 & -5 \end{bmatrix}$, xMNormalize(R,,,21) = $\begin{bmatrix} 0.6 & 0.773392104960212657067 \\ 0.8 & -0.633927954885420247631 \end{bmatrix}$

xMNormalizeC(**C**[,normtype][,Cformat][,tiny][,*D*]) **Normalize a complex matrix**

Normtype: all nonzero vertical vectors normalized; default = 2 for Euclidean norm.

$\mathbf{C} = \begin{bmatrix} 3 & 6.1 & -7 & 0 \\ 4 & -5 & 9 & 8 \end{bmatrix}$, xMNormalize(R,,,9) = $\begin{bmatrix} 0.6 & 0.773392105 & -0.6139406135 & 0 \\ 0.8 & -0.633927955 & 0.789352217 & 1 \end{bmatrix}$

xMT(**R**) **Transpose a real matrix** $\mathbf{R}^{\mathrm{T}}$

$\mathbf{R} = \begin{bmatrix} 3 & 6.1 \\ 4 & -5 \end{bmatrix}$, xMT(R) = $\begin{bmatrix} 3 & 4 \\ 6.1 & -5 \end{bmatrix}$, do ***not*** specify *D*.

xMTC(**C**) **Transpose a complex matrix** $\mathbf{C}^{\mathrm{T}}$

$\mathbf{C} = \begin{bmatrix} 3 & 6.1 & -7 & 0 \\ 4 & -5 & 9 & 8 \end{bmatrix}$, xMTC(C) = $\begin{bmatrix} 3 & 4 & -7 & 9 \\ 6.1 & -5 & 0 & 8 \end{bmatrix}$, do ***not*** specify *D*.

xMTH(**C**) **Hermitean (conjugate, adjoint) transpose a complex matrix** $\mathbf{C}^{\mathrm{H}}$

$\mathbf{C} = \begin{bmatrix} 3 & 6.1 & -7 & 0 \\ 4 & -5 & 9 & 8 \end{bmatrix}$, xMTH(C) = $\begin{bmatrix} 3 & 4 & 7 & -9 \\ 6.1 & -5 & 0 & -8 \end{bmatrix}$, do *not* specify *D*.

D.9.3 Matrix decompositions

xMLU(**S**[,Pivot][,*D*]) **LU decomposition** using Crout's algorithm **L U**

Returns the Lower and Upper triangular matrices that satisfy **S** = **L U** or, when Pivot is True, **S** = **P L U** where **P** is the permutation matrix. If Pivot = False, the first diagonal element of **S** cannot be zero.

xMCholesky(**S**[,*D*]) **LL decomposition** $\mathbf{L}\,\mathbf{L}^{\mathrm{T}}$

Cholesky decomposition of a square matrix.

xSysLin(**A**,**B**[,*D*]) **Solves simultaneous real linear equations** $\mathbf{X} = \mathbf{A}^{-1}\,\mathbf{B}$

Uses the Gauss-Jordan diagonalization; **A**, **X** and **B** must be real; **A** must be *m*×*m*; **X** and **B** must both be *m*×1 or *m*×*n*. Solves **A X** = **B** to yield $\mathbf{X} = \mathbf{A}^{-1}\,\mathbf{A}\,\mathbf{X} = \mathbf{A}^{-1}\,\mathbf{B}$.

xSysLinC(**A**,**B**[,*D*]) **Solves simultaneous complex linear equations** $\mathbf{X} = \mathbf{A}^{-1}\,\mathbf{B}$

Equivalent to xSysLin for complex arrays. **A**, **X** and **B** must be complex; **A** must be *m*×*m*; **X** and **B** must be *m*×1 or *m*×*n*. Solves **A X** = **B** to yield $\mathbf{X} = \mathbf{A}^{-1}\,\mathbf{A}\,\mathbf{C} = \mathbf{A}^{-1}\,\mathbf{B}$.

xGaussJordan(**M**,*n*,*m*,Det, Algo, *D*) **Gauss-Jordan elimination**

Uses partial pivoting.

xSVDD(**R**[,D][,ε]) **Matrix Σ from SVD of a real rectangular matrix R** **Σ**

SVD used in "compact" format; when **R** is $m\times n$, and $p = \min(m,n)$, **Σ** is $p\times p$.
$|\varepsilon|$ is the ignored rounding error; default: $|\varepsilon| \le 1\text{E}{-}D$.

xSVDDC(**C**[,c][,D][,ε]) **Matrix Σ from SVD of a complex rectangular matrix C** **Σ**

SVD used in "compact" format; when **C** is $m\times n$, and $p = \min(m,n)$, **Σ** is $p\times p$.
Default format: $c = 1$ (split). $|\varepsilon|$ is the ignored rounding error; default: $|\varepsilon| \le 1\text{E}{-}D$.

xSVDU(**R**[,D][,ε]) **Matrix U from SVD of a real rectangular matrix R** **U**

SVD used in "compact" format; when **R** is $m\times n$, and $p = \min(m,n)$, **U** is $n\times p$.
$|\varepsilon|$ is the ignored rounding error; default: $|\varepsilon| \le 1\text{E}{-}D$.

xSVDUC(**C**[,c][,D][,ε]) **Matrix U from SVD of a complex rectangular matrix C** **U**

SVD used in "compact" format; when **C** is $m\times n$, and $p = \min(m,n)$, **U** is $n\times p$.
Default format: $c = 1$ (split). $|\varepsilon|$ is the ignored rounding error; default: $|\varepsilon| \le 1\text{E}{-}D$.

xSVDV(**R**[,D][,ε]) **Matrix V from SVD of a real rectangular matrix R** **V**

SVD used in "compact" format; when **R** is $m\times n$, and $p = \min(m,n)$, **V** is $m\times p$.
$|\varepsilon|$ is the ignored rounding error; default: $|\varepsilon| \le 1\text{E}{-}D$.

xSVDVC(**C**[,c][,D][,ε]) **Matrix V from SVD of a complex rectangular matrix C** **V**

SVD used in "compact" format; when **C** is $m\times n$, and $p = \min(m,n)$, **V** is $m\times p$.
Default format: $c = 1$ (split). $|\varepsilon|$ is the ignored rounding error; default: $|\varepsilon| \le 1\text{E}{-}D$.

D.10 Miscellaneous functions

D.10.1 Manipulating numbers

xCStr(x[,D]) **Converts a number *x* from double precision to string format**

Ignores $D_{default}$; when D is deleted, as many digits as needed (up to Digits_Limit) are displayed.
xCStr(1) = 1; xCStr(0.1) = 0.1000000000000000055511151231257827021181583404541015625;
xCStr("1.1") = 1.1; xCStr(1.1) = 1.100000000000000088817841970012523233890533447265625;
xCStr("4.1") = 4.1; xCStr(4.1) = 4.0999999999999996447286321199499070644378662109375.
When B2 holds the number 4.1, xCStr(B2) = xCStr(4.1), see above, but xCStr(""&B2&"") = 4.1,
i.e., the stored, binary value of x is read unless its spreadsheet value is selected with double quotes.
D can be used to limit the output: xCStr(B2,20) = xCStr(4.1,20) = 4.0999999999999996447.

xDec(a) **Decimal part of number *a***

xDec(2.99) = 0.99; xDec(–2.99) = –0.99.

xTrunc(a) **Truncation**

xTrunc(2.99) = 2; xTrunc(–2.99) = –2; xTrunc(a) + xDec(a) = a.

xRound(a,[d][,D]) **Round**

Rounds a to d decimal places; default: $d = 0$. If least significant digit is 5, rounds it away
from zero. xRound(1.5) = 2; xRound(2.5) = 3; xRound(–1.5) = –2; xRound(–2.5) = –3.

vRoundR(a[,s][,D]) **Relative round**

Uses *unbiased* (banker's) relative rounding. Rounds the mantissa of a to s ***s***ignificant
digits, while leaving its exponent alone. Note: the default (with s unspecified) is 15.

xRoundR(a[,s][,D]) **Relative round**

Uses standard rounding to round the mantissa of a to s ***s***ignificant digits,
while leaving its exponent alone. Note: the default (with s unspecified) is 15.

xInt(a) **Integer part**

Rounds down: xInt(2.99) = 2; xInt(–2.99) = –3. Warning: in general, for $a < 0$, xInt(a) + xDec(a) $\neq a$.

xComp(a[,b]) **Comparison of value of *a* with *b***

xComp(a, b) = 1 for $a > b$, xComp(a, b) = 0 for $a = b$,
xComp(a, b) = –1 for $a < b$. The default assumes that $b = 0$.

xComp1(a) **Comparison of absolute value of *a* with 1**

xComp1(a) = 1 for $|a| > 1$, xComp1(a) = 0 for $|a| = 1$, xComp1(a) = –1 for $|a| < 1$.

xDgt(*a*) **Digit count**

xDgt(–2.99) = 3; xDgt(–0.00299) = 6.

xDgtS(*a*) **Significant digit count**

Treats all trailing zeros as not significant: xDgtS(1234000) = 4;
xDgtS(1.234) = 28 (counting significant digits in corresponding string number);
xDgtS("–0.0029900") = 3; xDgtS(–0.0029900) = 28.

xCDbl(*a*) **Converts from extended to double precision**

Converts an extended precision numerical string into a double precision number.
Example: xPi() = 3.1415926535897932384626433832795029; xCDbl(xPi()) = 3.1415927 with up to 15 digits depending on the cell formatting.

x2Dbl(*a*) **Converts from extended to double precision**

Slower but in rare cases more precise version of xCDbl.

D.10.2 Formatting instructions

xFormat(*a*[,*Digit_Sep*]) **Format**

formats a string ′*a* in comma-separated groups of *Digit_Sep*; default: *Digit_Sep* = 6.
For *a* = ′1234567.89012345, xFormat(*a*) = 1,234567.890123,45 and xFormat(*a*,3) = 1,234,567.890,123,45; when *a* = 1234567.89012345, a spreadsheet number, the result will reflect the stored value: xFormat(*a*) = 1,234,567.890,123,449,964,448,809,624.

xUnformat(*a*) **Unformat**

Removes formatting commas from *a*

xSplit(*a*) **Splits scientific notation over two cells**

Converts a number into scientific notation, spread over two adjacent cells.
xSplit(*a*) = {1.23456699999999999941758246258, 89} for *a* = 1.234567E+89;
xSplit(*a*) = {1.234567, 896} for *a* = 1234567E890 or *a* = 1234567E+890;
xSplit(*a*) = {1.234567, -884} for *a* = 1234567E-890.

xMantissa(*a*) **Mantissa of *a* in scientific format**

Yields the mantissa of a numerical string *a*, e.g., xMantissa(a) = -123.4567 for *a* = ′-1.234567E-890 but -1.23456000000000000004997855423 for *a* = –1.234567E-890.

xExponent(*a*) **Exponent of *a* in scientific format**

Yields the exponent of a numerical string *a* or a number in the cell, e.g., xExponend(*a*) = -890 for either *a* = –1.234567E-890 or *a* = ′-1.234567E-890 because the exponent is always integer.

xCvExp(*mant*[,*exp*]) **Converts scientific notation into mantissa and exponent**

=xCvExp(-123.456,789) yields -1.23456000000000000030695446185E+791, and
=xCvExp(-0.0000123456,0) generates -1.23455999999999999916351148266E-5,
in both cases showing decimal-to-binary conversion errors. You can avoid these by setting *exp* to zero: =xCvExp("-0.0000123456",0) leads to –1.23456E-5.

D.10.3 Logical functions

x_And(*a*,*b*) **Boolean logic AND** AND(*a*,*b*)

x_And(*a*,*b*) = True only when *a* ≠ 0 (or FALSE) *and b* ≠ 0 (or FALSE);
a blank cell does not count as 0 (or FALSE).

x_Or(*a*,*b*) **Boolean logic OR** OR(*a*,*b*)

x_Or(*a*,*b*) = True when *a* ≠ 0 (or FALSE)*or b* ≠ 0 (or FALSE)*or both*,
a blank cell doesn't count.

x_If (*a*,*b*) **Boolean logic IF** IF()

x_If(*a*,*b*,*c*) = *b* when *a* = 1 or TRUE, x_If(*a*,*b*,*c*) = *c* when *a* = 0 or FALSE

x_Not (*a*) **Boolean logic NOT** NOT(*a*)

x_Not(*a*) = True when *a* = 0 (or FALSE). Non-zero numbers and strings evaluate as True.

D.10.4 Polynomial functions

xPolyTerms (*poly* [,*D*]) **Extract the coefficients of a polynomial**

When *poly* is, e.g., 'x^5-2.1+3*x^3+4*x^2 in cell B2, xPolyTerms(B2) = {–2.1, 0, 4, 3, 0, 1}

xPoly (*a*,*coef* [,*D*]) **Evaluate a polynomial at *x***

When the polynomial is defined by its coefficients *coef* in, e.g., B4:G4 as {–2.1, 0, 4, 3, 0, 1}, xPoly(3,B4:G4) = –374.3.

xPolyAdd(*poly*1,*poly*2 [,*D*]) **Adds two polynomials in *x***

The polynomials are *poly*1 and *poly*2. Block-enter their coefficients in the same order. Missing coefficients will be interpreted as zero. If enumerated in the argument, use ,, to indicate a missing coefficient.

xPolySub (*poly*1,*poly*2 [,*D*]) **Subtracts two polynomials in *x***

The polynomials are *poly*1 and *poly*2. Block-enter their coefficients in the same order. Missing coefficients will be interpreted as zero. If enumerated in the argument, use ,, to indicate a missing coefficient.

xPolyMult (*poly*1,*poly*2 [,*D*]) **Multiplies two polynomials in *x***

The polynomials are *poly*1 and *poly*2. Block-enter their coefficients in the same order. Missing coefficients will be interpreted as zero. If enumerated in the argument, use ,, to indicate a missing coefficient. Assign space in the highlighted area for the higher-order cross-terms.

xPolyDiv (*a* [,*D*]) **Divides two polynomials in *x***

The polynomials are *poly*1 and *poly*2. Block-enter their coefficients in the same order. Missing coefficients will be interpreted as zero. If enumerated in the argument, use ,, to indicate a missing coefficient.

xPolyRem (*a* [,*D*]) **The remainder of polynomial division**

The polynomials are *poly*1 and *poly*2. Block-enter their coefficients in the same order. Missing coefficients will be interpreted as zero. If enumerated in the argument, use ,, to indicate a missing coefficient.

D.10.5 Integer operations

xPowMod (*a*,*p* [,*D*]) **Modular power** $a^p \bmod m$

Returns the remainder of the integer division a^p, i.e., $a^p - m\,(a^p \setminus m)$, e.g., xPowMod(10,3,7) = 6 because 10^3 = 1000 = 142*7 + 6 where 142*7 = 994. Useful for finding the remainders of divisions of very large integers, as in xPow(12,34567) = 1.14322609302954137911815317255379 44E+37304 with more than 3700 decimals, yet xPowMod(12,34567,89) = 52. This is the remainder of dividing 12^{34567} by 89 despite the fact that XN-version used, XN6051–7A, cannot hold more than 630 decimals.

xDivMod (*a*,*b*,*m*) **Modular division** $(a/b) \bmod m$

where *a* and *b* are integers, and *m* is a positive prime integer; otherwise the function returns “?”. Example: xPow(12,3939393) = 1.1127850718610753473503619921808241E+4251319, i.e., it is a number with more than 4 million digits! While XN cannot perform the regular division of such a giant number by the prime number 3001, it can find xPowMod(12,3939393,3001) = 2758.

D.10.6 Getting (& setting) XN configuration information

Here are a number of functions that allow you to read or “get” configuration settings, and to define or “set” them. Since each Get function has a corresponding set counterpart, only the former are listed here; these Get instructions must be followed by empty argument brackets to identify them as functions. A corresponding Set function must have a replacement value as its argument, and is meant for use within a VBA function or macro.

GetDigitsLimit() **Specifies the current DigitsLimit**

For XN.xla605 the function =GetDigitsLimit() yields 630, its largest allowed *D*-value.

GetExcelAppVer() **Specifies the current version of Excel used**

For Excel97: =GetExcelAppVer() yields 8, 9 for 2000, 10 for 2002, 11 for 2003, 12 for 2007, and 14 for 2010.

GetxBase() **Specifies the current packet size**

For XN.xla605 the function =GetxBase() yields the value 7.

GetXnArgSep() **Specifies the current VBA argument separator**

In the US, =GetXnArgSep() should yield a comma.

GetXnCaseSen() **Specifies the current case sensitivity**

If case-insensitive (the default), =GetXnCaseSen() yields FALSE; if case-sensitive, TRUE.

GetXnConfigStatus() **Specifies the current configuration settings**

Needs a 19 rows high, 2 columns wide array to list the names and values of all 19 configuration settings.

GetXnDecSep() **Specifies the current VBA decimal separator**

In the US, =GetXnDecSep() should yield a period.

GetXnDefaultDigits() **Specifies the currently selected default *D*-value**

For the examples in this table, =GetXnDefaultDigits() should yield 35.

GetXnDefCStr() **Specifies the current default value for default Dbl2Str digits**

=GetXnDefCStr() yields 0 for vCStr, 15 to 28 for dCStr, 29 to Digits_Limit for xCStr.

GetXnSMPAdj() **Specifies the Digit Max Adjustment of the Simulated Machine Precision**

For 7-digit packets, the recommended value is 2 × 7 = 14 decimals.

GetXnAddAdj() **Specifies the current Digit Max Adjustment for xAdd**

The recommended value is 0 decimals for all versions of XN.

GetXnDivAdj() **Specifies the current Digit Max Adjustment for xDiv**

The recommended value is 0 decimals for all versions of XN.

GetXnMultAdj() **Specifies the current Digit Max Adjustment for xMult**

The recommended value is 2 packets for all versions of XN.

D.11 The Math Parser and related functions

The Math Parser can evaluate many formulas *f* written in quasi-algebra, as a function of the specified parameter Values. It thereby brings an aspect of symbolic calculus to numerical computation. Its formulas resemble those in Excel's VBA, as a function of the parameter Values. The Math Parser performs two functions: it first "parses" the formula, then evaluates its value. Its extended precision implementations xEval and xEvall, as implemented by John Beyers, uses the original parser developed for double precision expressions, but with XN for value evaluation. This can be especially helpful because writing complicated mathematical expressions in XN can be error-prone, a complication readily avoided by using xEval or xEvall. The Help-on-Line entry xEval (see the XN Toolbar under Help) gives many clear examples. xEvall uses a sophisticated search for the value labels which makes it about ten times slower than xEval; its use is therefore not recommended.

xEval(*f*, Values, [,*D*] [,Angle] [,*Tiny*] [,IntSwapFix]) **Evaluates quasi-algebraic formulas**

xEvall(*f*, Values, [,*D*] [,Angle] [,*Tiny*] [,IntSwapFix]) **xEval using top labels if present**

xEval assigns the parameter values in the order in which they are listed under Values. *D* = 0 will use the faster double-precision mode; *D* = -1 specifies quadruple precision in the Variant Decimal mode. Leaving *D* unspecified will use the value of Default Digits specified in the XN Toolbar under X-Edit ⇨ Configuration.
Angle provides a choice between the default rad(ians), deg(ree), and grad(s). *Tiny* defines the minimum absolute value that will be considered to be different from zero; for the optional IntSwapFix see the Help-on-Line file.

The formula *f* and its values can be fully specified in the argument, as in xEval("1/x^2+5*x*y+7*sqr(y)",{"2","3"},28) = 42.37435565298214105469212439, or the formula and/or its parameter values can be read from specified spreadsheet cells, as in =xEval("1/x^2+5*x*y+7*sqr(y)",I2:I3,28) or =xEval(I4,{"2","3"},28 or

=xEval(B4,B2:B3,28), which all give the same result when cell B4 contains the formula 1/x^2+5*x*y +7*sqr(y), and cells B2 and B3 the values 2 and 3 respectively..

For further details about the Math Parser see section 8.16 and, especially, the Help-on-Line entry on xEval. As described there, several functions can also use its quasi-algebraic code, such as the integration functions Integr(), Integr_2D, etc.
Here are two extended precision functions that use the Math Parser: xGrad and xJacobi.

xGrad(Values, *f* [,**x**] [,*D*] [,Labels]) **Gradient of a multivariate function *f***

$$\begin{bmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \\ \vdots \\ \partial f/\partial x_m \end{bmatrix}$$

Approximates the gradient of a single function *f* of several variables, by default called x, y, z, and t in this order, as evaluated at the parameter Values in that same order, using 5-point expressions for the derivative. If you want to use other variable name Labels in your function, specify them as Labels and count your commas, see below.

You can specify the Values and the formula for *f* directly into the expression, as in =xGrad({-1,2,3,7},("(x+2*y-3*z^2)/LN(t)")), or read them from the spreadsheet, as in =xGrad(B2:B5,B6), when B2:B5 contain the values –1, 2, 3, and 7 respectively, and cell B6 the formula (x+2*y-3z^2)/ln(t).

In both cases you will get $\begin{bmatrix} 0.51389834\ 2369750693\ 0446493893\ 7018939 \\ 0.51389834\ 2369750693\ 0446493893\ 7018939 \\ -9.25017016\ 2655512474\ 8036890086\ 63409 \\ 0.90545659\ 2995580001\ 2161419931\ 8047098 \end{bmatrix}$

Also in both cases, the expression must be written in Math Parser format. The Values, either enumerated or taken from B2:B5, must be in the order *x,y,z,t*. If you use other names, e.g., a, b, c, and d, then these must be defined in Labels as =xGrad(B2:B55,B6,,,A2:A5) where A2:A5 contains a, b, c, and d respectively.

xJacobian(Values, **f,** [,**x**] [,*D*] [,Labels] [,MaxPrec]) **Jacobian of *f***

$$\begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_m \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_m \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & \partial f_n/\partial x_2 & \cdots & \partial f_n/\partial x_m \end{bmatrix}$$

Approximates the Jacobian of a vector **f** of *n* functions *f* , each of *m* variables, by default called x, y, z, and t in this order, as evaluated at the parameter Values listed in that same order, using 5-point expressions for the derivative. If you want to use other variable name Labels in your function, specify them as Labels and keep track of the commas.

As with xGrad you can specify the Values and the formula for *f* directly into the expression or, as is usually more convenient, read them from the spreadsheet, as in =xGrad(B2:B4,B5:B7), where B2:B5 contain the specific Values at which the function formulas (in Math Parser format) in B5:B7 must be evaluated:

$$\mathbf{J} = \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_m \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_m \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & \partial f_n/\partial x_2 & \cdots & \partial f_n/\partial x_m \end{bmatrix} \text{ for } \mathbf{f} = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_m) \\ f_2(x_1, x_2, \ldots, x_m) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_m) \end{bmatrix} \text{ and } \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}.$$

Subject index

Courier font identifies VBA instructons. Numbers refer to pages; italic numbers indicate the starting page of section(s) primarily devoted to that topic. Excel functions are shown in caps, VBA. Matrix & XN functions in lower case.

D

J

L

M

P

Q

R

S

W

X

Z