

Xnum - Multiprecision Floating Point Library for Scilab

v.1.0, Sept. 2007

Introduction

Scilab is a very nice freeware program for quick and interactive computations with vectors and matrices, real and complex numbers. In general, Scilab is a numerical matrix system..

scilab-4.1

Copyright (c) 1989-2006
Consortium Scilab (INRIA, ENPC)



Usually the global precision of Scilab is sufficiently high for the most part of applications. On a Window 32 bit PC, the basic structure is the IEEE-754 which allows for a 64 bit standard floating point double precision number. The later has a mantissa of 53 bits (one is the implied bit) which becomes equivalent to 15.4 digits. Scilab reports a maximum of 16 digits.

Sometime, however, it is necessary to increase the standard precision up to 30, 60 or more significant digits. For that scope, Scilab provides a set of tools (macros, interface, etc.) for interfacing the most important multiprecision library written in Fortran or C++. But this is theory; practically it is not so easy as we could wait.

For those that cannot waste time with all these stuffs, it comes in handy this easy library.

Features

- Xnum¹ is a library for real numbers in floating point up to 200 significant digits.
- It works for real number (no complex).
- Multiprecision numbers are written as strings (called x-numbers in this document).
Examples: "-1.27636600000488477", "1.27636600000488477E-023", etc
- The library accepts in input also standard numbers
- The output is always in string format.
- Of course, the operation with x-numbers must be performed by the multiprecision functions (called x-functions in this document).
- Xnum contains more than 40 x-functions covering the basic arithmetic operations and some transcendental and trigonometric functions

How to install it

Xnum is composed by 3 simple files:

- **xnum_loader.sce** installer and loader scrip
- **xnum.sci** interfacing user functions
- **xnum.dll** multiprecision arithmetic

Installing Xnum is very simple. Copy all the above files in a directory that you want, for example C:\scilab_lib. Than, start Scilab and make this directory the current directory by the menu command *File / Change Directory*. This is only for starting the script **xnum_loader.sce** by the menu-command *File / Exec*. Once the script is gone, Xnum.dll is linked and the x-functions are loaded into Scilab, ready for a multiprecision session.

¹ The core of this library is derived from xnumber.dll Active-X, ver. v. 1.1.6 - March. 2006, Foxes Team

Multiprecision Functions

All functions work in real domain

For default, all functions use 30 digits precision (dgtmax = 30).

Xnum does not make overloading. All the operations are always performed by x-functions

```
xmmul(a,b,[dgtmax]) // matrix product
xminv(a,[dgtmax]) // matrix inverse
xmul(a,b,[dgtmax]) // product (single, dot, scalar)
xadd(a,b,[dgtmax]) // addition (single, dot, scalar)
xsub(a,b,[dgtmax]) // subtraction (single, dot, scalar)
xdiv(a,b,[dgtmax]) // division (single, dot, scalar)
xmod(a,b,[dgtmax]) // modulo
xsysl(a,b,[dgtmax]) // linear system a x = b
xroundr(x,d) // round relative
xround(x,d) // round
xneg(x) // change sign
xfloor(x) // integer part
xdet(x,[dgtmax]) // matrix determinant
xabs(x) // absolute
xsqrt(x,[dgtmax]) // square root
xroot(x,n,[dgtmax]) // n-th root
xpow(x,n,[dgtmax]) // dot integer power
xmpow(x,n,[dgtmax]) // matrix integer power
xexp(x,[dgtmax]) // exponential e^x
xlog(x,[dgtmax]) // logarithm in base "e" log(x)
xlogb(x,b,[dgtmax]) // logarithm in any base "b" log_b(x)
xexpb(b,x,[dgtmax]) // exponential in any base "b" b^x
xpi_([dgtmax]) // pi constant
xe_([dgtmax]) // e constant
xsin(x,[dgtmax]) // sin
xcos(x,[dgtmax]) // cos
xtan(x,[dgtmax]) // tan
xasin(x,[dgtmax]) // inverse sin
xacos(x,[dgtmax]) // inverse cos
xatan(x,[dgtmax]) // inverse tan
xgcd(x,[dgtmax]) // GCD
xlcm(x,[dgtmax]) // LCM
xcomb(n,k,[dgtmax]) // combinations C(n,k)
xfact(n,[dgtmax]) // factorial n!
xlength(x) // significant digits
xcomp(a,[b]) // compares two numbers a b (default b = 0): returns 1, 0, -1
dblstr(x) // converts a number from double to string
strdbl(x) // converts a number from string to double
```

Scalar, and dot matrix operation

The x-functions operate either with scalar or matrix input

For example the basic functions, such as xdiv(a,b), require two operands and can perform the following 4 different operations depending on the type of its operands.

1 st operand	2 nd operand	result of xdiv(a,b)
scalar: a	scalar: b	a/b
scalar: a	matrix: $\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$	$a/\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a/b_{11} & a/b_{12} \\ a/b_{21} & a/b_{22} \end{bmatrix}$
matrix: $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$	scalar: b	$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}/b = \begin{bmatrix} a_{11}/b & a_{12}/b \\ a_{21}/b & a_{22}/b \end{bmatrix}$
matrix: $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$	matrix: $\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$	$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}/\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}/b_{11} & a_{12}/b_{12} \\ a_{21}/b_{21} & a_{22}/b_{22} \end{bmatrix}$

As we can see, the function works similar to the dot "./" Scilab operator.

Note that in the last case, the size of both matrices must be the same.

With the same concept work all other x-functions:

$$\text{xadd} \Leftrightarrow ".+" , \quad \text{xsub} \Leftrightarrow ".-" , \quad \text{xmul} \Leftrightarrow ".*" .$$

1 st operand	2 nd operand	result of xmul (a,b)
scalar: a	scalar: b	$a \cdot b$
scalar: a	matrix: $\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$	$a \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a \cdot b_{11} & a \cdot b_{12} \\ a \cdot b_{21} & a \cdot b_{22} \end{bmatrix}$
matrix: $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$	scalar: b	$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot b = \begin{bmatrix} a_{11} \cdot b & a_{12} \cdot b \\ a_{21} \cdot b & a_{22} \cdot b \end{bmatrix}$
matrix: $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$	matrix: $\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$	$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \cdot b_{11} & a_{12} \cdot b_{12} \\ a_{21} \cdot b_{21} & a_{22} \cdot b_{22} \end{bmatrix}$

The well known matrix multiplication is performed by xmmul(a,b)¹, another x-function.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

¹ The "m" prefix stands for matrix and it avoids confusion with the dot-multiplication xmul(a,b).


```
-->xroundr(C,29)
ans =

!1          3.2D-31  !
!
!-1.7D-31  1        !
```

It is also possible to round to 15 significant digits and convert in real at the same time using the function `evstr(x)`

```
-->evstr(C)
ans =

  1.          3.200D-31
- 1.700D-31  1.
```

Using multiprecision in Scilab

Example. Solve the linear system $\mathbf{A} \mathbf{x} = \mathbf{b}$ where \mathbf{A} is the following matrix

```
A =  
  
 1.   1.   1.   1.   1.   1.   1.   1.  
 1.   2.   3.   4.   5.   6.   7.   8.  
 1.   4.   9.  16.  25.  36.  49.  64.  
 1.   8.  27.  64.  125.  216.  343.  512.  
 1.  16.  81.  256.  625.  1296.  2401.  4096.  
 1.  32.  243.  1024.  3125.  7776.  16807.  32768.  
 1.  64.  729.  4096.  15625.  46656.  117649.  262144.  
 1.  128.  2187.  16384.  78125.  279936.  823543.  2097152.
```

and \mathbf{b} is the vector

```
b' = [ 8. 36. 204. 1296. 8772. 61776. 446964. 3297456. ]
```

Both \mathbf{A} and \mathbf{b} can be generated by the following short script

```
// Vandermonde system  
n=8;  
x=1;  
A=zeros(n,n);  
for i=1:n  
  for j=1:n  
    A(i,j)=(x+j-1)^(i-1);  
  end,end  
  
b=zeros(n,1);  
for i=1:n  
  b(i)=sum(A(i,1:$));  
end,
```

Note that the vector \mathbf{b} is the sum of each matrix row; therefore the solution \mathbf{x} is the unitary vector. The matrix \mathbf{A} belongs to the Vandermonde matrices family and therefore very difficult to solve. If we try to solve the linear system we get the following result

```
->x=A\b  
warning  
matrix is close to singular or badly scaled. rcond = 6.7540D-10  
computing least squares solution. (see lsq)  
  
x =  
  
 1.0714285701  
 0.66266707408  
 1.75471707083  
 0.  
 1.82902936055  
 0.57539941262  
 1.12343216586  
 0.98436828424
```

Scilab, rightly, warns us that the matrix is badly scaled having a low condition factor (6.7E-10) and therefore the result may have a large error. As, in fact, it happens: the \mathbf{x} vector is quite different from the expected unitary vector. We can measure the global error by the formula

```
stdev(1-x) = 0.60610565491
```

Scilab also suggests to try the function `lsq(A,b)` that solve the system with least squares method
Let's try it.

```
-->x1=lsq(A,b)
x1 =

    0.99997105729
    1.00013668224
    0.99969420294
    1.00040517919
    0.99966409546
    1.00017203856
    0.99994998814
    1.0000063336
```

As we can see the solution `x1` is much more accurate than the previous one. The error is now

```
stdev(1-x1) = 0.00024558157
```

The least squares method exhibits an error of about $1E-4$, that is thousand times more accurate than the previous solution.

Now let's solve the same linear system using the multiprecision function `xsysl(A,b)`

```
x2= xsysl(A,b);
y=evstr(x2);

y'= [1, 1, 1, 1, 1, 1, 1, 1]
```

After the real conversion the solution looks perfect; numerically speaking, it means that the error is "squeezed" under $1E-16$

Example. Solve the linear system $\mathbf{A} \mathbf{x} = \mathbf{b}$ where \mathbf{A} is the (10x10) Hilbert matrix and \mathbf{b} is the sum of the rows. Let's use the following script for generating both \mathbf{A} and \mathbf{b}

```
//Hilbert system
n=10;
A=zeros(n,n);
for i=1:n
for j=1:n
    A(i,j)=1/(i+j-1);
end,end

b=zeros(n,1);
for i=1:n
    b(i)=sum(A(i,1:$));
end,
```

As known, Hilbert matrices are "terrible" from the point of view of numeric calculus. They tend greatly to amplify round-off error and the solution are affected by large error even with moderate matrix size.

If we try to solve the linear system by \mathbf{A} / \mathbf{b} we found an error of about 0.8. Practically the error overcomes the solution itself.

Fortunately, the least square solution works much better with a limited error of about $5.3E-5$ that it is a very good result considering the difficulty and the size of the matrix

And multiprecision? If we try to solve the system with `xsysl(A,b)` we have a global accuracy of about $1E-3$. Not bad, but very far from that we have expected. Why? What happened?

The fact is that we have used in input an approximated (10x10) Hilbert matrix rounded to 15 significant digits. The input round-off error has been amplified in the same way of those generated

The x-functions can be used inside the body of user-macros and user-function as any other built-in function

Example. Implement a function computing the continue fraction expansion of square root

$$x \in \mathbf{N}, \quad \sqrt{x} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\dots}}} \quad \sqrt{x} = [a_0 \ a_1 \ a_2 \ a_3 \ \dots \ a_m]$$

As known, the expansion is infinite with period m , where $a_m = 2a_0$

A possible implementation is the following code

```
//
// multi-precision continue fraction
// expansion of square root x^(1/2)
//
function v= sqrtcf(x, dgtmax)
// input check
[lhs,rhs] = argn()
if rhs < 2 then dgtmax = 30; end //default set
// initialize
n= 150;
v= zeros(n);
d= xsqrt(x, dgtmax);
// loop begins
for i=1:n
    v(i)= evstr(xfloor(d)); // v(i)= [d]
    d= xdiv(1, xsub(d, v(i), dgtmax), dgtmax); // d= 1/(d-v(i))
    if(v(i) == 2*v(1)) then break; end
end
v=v' // out transpose
if i >= n then warning("period too long"); end
endfunction
```

Let's use now the function sqrtcf(x, [dgtmax]). After loaded in Scilab we give the command

```
-->sqrtcf(2)
ans =

    1.    2.
```

the period is $m = 1$ and the continue fraction expansion is $\sqrt{2} = 1 + [2, 2, 2, 2, \dots]$

```
-->sqrtcf(31)
ans =

    5.    1.    1.    3.    5.    3.    1.    1.    10.
```

the period in this case is $m = 8$ and the expansion is $\sqrt{31} = 5 + [1, 1, 3, 5, 3, 1, 1, 10, \dots]$

As we can see, the period changes with the integer number under the root; some numbers, such as 2, 5, 10 have a short period while others, for example 31, 43, 46, have longer period.

When the period becomes longer ($m > 20$) it is necessary to increment also the precision.

For example 5771 has $m = 32$, requiring $dgtmax = 40$, while 5773 has a very long period, $m = 88$, but needs a precision of more than 100 digits for reaching this result

X-Functions

xmmul(a,b,[dgtmax]) // matrix product

performs the matrix product of (n x p) matrix by a (p x m) matrix: the result is a (n x m) matrix

```
A =
!7904383488      58791264791104   !
!               !
!1007006299349337 735409593821447   !

-->b=['2410677402'; '70372826']

-->C=xmmul(A,b)
C =

!4156362366115551678080      !
!                               !
!2479320180897821374381696   !
```

xminv(a,[dgtmax]) // matrix inverse

computes the matrix inverse of a square non singular matrix

```
A =
    19873.  - 2845.
   - 3773.   1040.

-->C=xminv(A)
C =

!1.04693753155283486020112273984D-4  2.86397815121905305506941749503D-4   !
!                                       !
!3.79816856398927493032580393981D-4  2.00055668889898915161316463546D-3   !
```

xmul(a,b,[dgtmax]) // product (dot, scalar)

computes the dot product of two matrices of identical size.
It is the multiprecision version of Scilab dot operation ". *"

```
A =
    126635.    38.4995
   - 0.013556    98745.

B =
    3465.    26.
   0.5995886 - 49.5996

-->xmul(A,B)
ans =

!438790275      1000.987      !
!               !
!-0.0081280230616 -4897712.502 !
```

input a or b may be also scalar.

```
A =
    126635.    38.4995
   - 0.013556    98745.

-->b=1000;

-->xmul(A,b)
ans =

!126635000    38499.5    !
!              !
!-13.556     98745000    !
```

input may also be both scalar

```
-->xmul(1277374775, 5.3123456789)
ans =

6785856366.3071097475
```

xadd(a,b,[dgtmax]) // addition (dot, scalar)

computes the dot addition of two matrices of identical size
It is the multiprecision version of Scilab dot operation ". +"

```
A =
!10000000000    701583371424    !
!              !
!7841016360300100000    -254803968    !

B =
    100.    234.
   6010.   - 48.

-->xadd(A,B)
ans =

!100000000100    701583371658    !
!              !
!7841016360300106010    -254804016    !
```

xsub(a,b,[dgtmax]) // subtraction (dot, scalar)

computes the dot subtraction of two matrices of identical size
It is the multiprecision version of Scilab dot operation ". -"

```
-->xsub(A,B)
ans =

!99999999900    701583371190    !
!              !
!7841016360300093990    -254803920    !
```



```
!1198652 2205083 !
!
!4024311 2464783 !
```

xsysl(A,b,[dgtmax]) // linear system

Solves a determined linear system $Ax = b$ or $AX = B$

```
A =
!82343975 43050332 -123456 !
!
!79088020 48637407 19873636 !
!
!31234569 -81874798 27935103 !

b =
!53917646612577913 !
!
!59878000807780528 !
!
!57796322403148133 !

-->xsysl(A,b)
ans =

!781000015 !
!
!-240003356 !
!
!492280170 !
```

This function solves also a set of linear systems. In this case the right term **B** is a matrix

```
A =
!82343975 43050332 -123456 !
!
!79088020 48637407 19873636 !
!
!31234569 -81874798 27935103 !

B =
!63272004443 247525749 8665024276 !
!
!61474885836 157769516 8375302434 !
!
!27588338073 -18036705 2276773817 !
```

```
-->xsysl(A,B)
ans =

!781 3 100 !
!
!-24 0 10 !
!
!44 -4 -1 !
```

Note that the solution **X** has always the size of **B**.

xneg(x) // change sign

changes sign -x to an string number

```
-->xneg(1.8533310853102241853331085310224)
ans =

-1.8533310853102241853331085310224
```

xabs(x) // absolute

returns the absolute of a x-number

```
-->xabs ('-123.4999999999999377288')
ans =

123.4999999999999377288
```

xroundr(x,d) // round relative

rounds a number to "d" significant digits

xround(x,d) // round

rounds a number to "d" decimals

This example shows the difference between decimal rounding and relative rounding

```
x='0.00001234567690'

xroundr(x,1) = 0.00001          xround(x,1) = 0
xroundr(x,2) = 0.000012        xround(x,2) = 0
xroundr(x,3) = 0.0000123       xround(x,3) = 0
xroundr(x,4) = 0.00001235      xround(x,4) = 0
xroundr(x,5) = 0.000012346     xround(x,5) = 0.00001
xroundr(x,6) = 0.0000123457    xround(x,6) = 0.000012
xroundr(x,7) = 0.00001234568   xround(x,7) = 0.0000123
xroundr(x,8) = 0.000012345677  xround(x,8) = 0.00001235
```

xfloor(x) // integer part

returns the [x] (also called integer part of x), that is the highest integer less or equal to x

xdet(x,[dgtmax]) // matrix determinant

returns the determinant of a square matrix

```
A =

 82343975.    43050332.   - 123456.
 79088020.    48637407.    19873636.
 123456.      1874798.     27935103.

-->xdet(A)
ans =

13787542451261211659759
```

If the matrix is integer, the determinant can be used for computing its inverse in exact rational mode. For example

```
B =

 - 8.   - 10.   3.
 - 3.    37.   13.
 - 1.    2.    1.

-->d=det(B)
d =

 105.
```

```
-->C= xminv(B);
-->xmul(C,d)
ans =
!11   16  -241  !
!      !      !
!-10  -5   95   !
!      !      !
!31   26  -326  !
```

therefore the exact inverse of **B** is

$$\mathbf{B}^{-1} = \begin{bmatrix} \frac{11}{105} & \frac{16}{105} & \frac{-241}{105} \\ \frac{-10}{105} & \frac{-5}{105} & \frac{95}{105} \\ \frac{31}{105} & \frac{26}{105} & \frac{-326}{105} \end{bmatrix}$$

xsqrt(x,[dgtmax]) // square root

computes the square root of a positive number \sqrt{x}

```
-->xsqrt(2)
ans =
1.41421356237309504880168872421
-->Q=[2;3;4;-1];
-->xsqrt(Q)
ans =
!1.41421356237309504880168872421  !
!                                !
!1.73205080756887729352744634151  !
!                                !
!2                                !
!                                !
!?                                !
```

Note that xsqrt works only in the real domain: xsqrt(-1) returns "?".

xroot(x,n,[dgtmax]) // n-th root

Compute the n^{th} root of a number $\sqrt[n]{x}$ with "n" positive integer

```
Q=[2;-8];
-->xroot(Q,3)
ans =
!1.25992104989487316476721060728  !
!                                !
!-2                                !
```

xpow(x,n,[dgtmax]) // dot integer power

returns the n^{th} integer power of x. It is the multi-precision version of " $x.^n$ ".

```
v=[128, 34; 972, -10];
-->xpow(v,8)
ans =
!72057594037927936      1785793904896  !
!
!796764763524754885902336  100000000  !
```

xmpow(A,n,[dgtmax]) // matrix integer power

returns the n^{th} integer power of a square matrix: $A^n = A A A \dots A$
 It is the multi-precision version of A^n

```
A=[128, 34; 972, -10];
A =
    128.    34.
    972.   -10.
-->xmpow(A,8)
ans =
!11569418282584869376  1478519300221822720  !
!
!42268257641635637760  5568369358155118336  !
```

xexp(x,[dgtmax]) // exponential

returns the exponential of a number e^x in base "e"

```
-->v=[-1;1;2;20];
-->xexp(v)
ans =
!0.367879441171442321595523770161  !
!
!2.71828182845904523536028747135  !
!
!7.38905609893065022723042746058  !
!
!485165195.409790277969106830542  !
```

xlog(x,[dgtmax]) // natural logarithm

returns the natural logarithm of a positive number $\log_e(x)$

```
-->xlog(2)
ans =
0.693147180559945309417232121458
```

xlogb(x,b,[dgtmax]) // logarithm in any base

returns the logarithm of a positive number x in any base $\log_b(x)$, with $b > 0$

```
-->xlogb(0.5, 10)
ans =
```

```
-0.301029995663981195213738894724
```

xexpb(b,x,[dgtmax]) // exponential in any base

returns the exponential in any base b^x , with $b > 0$
It is equivalent to the Scilab operation b^x .

```
-->xexpb(2.5, 0.7) // 2.5^0.7
ans =
1.89914448233093468677454218283
```

```
-->e=[0.7 ; 1.4; 10.5];
-->y=xexpb(2,e)
!1.62450479271247104521941876555 !
!
!2.63901582154578851874800394246 !
!
!1448.15468787004932997292925359 !
```

Note that when x is integer this function is numerically equivalent to `xpow` (but from the point of view of the efficiency `xexpb` is quite expensive).

xpi_([dgtmax]) // pi constant

returns the first "dgtmax" digits of the constant π

```
-->xpi_(80)
ans =
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089
```

xe_([dgtmax]) // e constant

returns the first "dgtmax" digits of the constant e

```
-->xe_(80)
ans =
2.7182818284590452353602874713526624977572470936999595749669676277240766303535475
```

Example. Compute the Ramanujan number $e^{\pi\sqrt{163}}$ with 30 digits

```
// Ramanujan number e^[pi*163^(1/2)]
-->xexp(xmul(xpi_(), xsqrt(163)))
ans =
262537412640768743.999999999972
```

xsin(x,[dgtmax]) // sin

computes the sine of x , in radiant

```
-->xsin(2.5)
ans =
0.598472144103956494051854702186
```

xcos(x,[dgtmax]) // cos

computes the cosine of x, in radiant

```
-->xcos(-1.4)
ans =
0.169967142900240938616748035204
```

xtan(x,[dgtmax]) // tan

computes the tangent of x, in radiant

```
-->xtan(1.5)
ans =
14.101419947171719387646083652
```

xasin(x,[dgtmax]) // inverse sin

computes the inverse-sine of x with $|x| < 1$.
The inverse-sine is defined between $-\pi/2 \leq \text{asin} \leq \pi/2$

```
-->xasin(0.6)
ans =
0.643501108793284386802809228717
```

xacos(x,[dgtmax]) // inverse cos

computes the inverse-cosine of x with $|x| < 1$.
The inverse-cosine is defined between $0 \leq \text{acos} \leq \pi$

```
-->xacos(0.6)
ans =
0.927295218001612232428512462922
```

xatan(x,[dgtmax]) // inverse tan

computes the inverse-tan of x .
The inverse-tan is defined between $-\pi/2 < \text{atan} < \pi/2$

```
-->xatan(24)
ans =
1.52915374769630819537114362694
```

xgcd(x,[dgtmax]) // GCD

computes the Greatest-Common-Divisor of a set of number

```
-->x=['2042628727295599881984', '87766497'];
-->xcgcd(x)
ans =
9751833
```

Note that **x** elements must be written as string to avoid round-off error

xlcm(x,[dgtmax]) // LCM

computes the Least-Common-Multiple of a set of number

```
-->x=[133518, 209826283, 560947];
-->xlcm(x)
ans =
15715258725626593518
```

xcomb(n,k,[dgtmax]) // combinations

computes the combinations (also called binomial coefficients) of n objects in k classes
Input n, k are positive integer.

```
-->xcomb(100,50)
ans =
100891344545564193334812497256
```

xfact(n,[dgtmax]) // factorial

computes the factorial n! of a positive integer n.

```
-->xfact(45)
ans =
1.19622220865480194561963161496D56
```

note that 45! is an integer with 56 digits. Because it exceeds the 30 digits (default), the result is automatically approximated and converted in exponential format. If you want to see the exact number, extend the precision to 57 or more digits

```
-->xfact(45,60)
ans =
119622220865480194561963161495657715064383733760000000000
```

xlength(x) // significant digits

counts the significant digits of a number
For example, the number of significant digits of 45! is 47

xcomp(a,[b]) // compares two numbers a b

compares two numbers and returns -1, 0, 1

$$\text{xcomp}(a,b) = \begin{cases} +1 & a > b \\ 0 & a = b \\ -1 & a < b \end{cases}$$

if b is omitted, the function assumes b = 0 returning the sign of a, that is:

$$\text{xcomp}(a) = \text{sgn}(a)$$

```
-->x=['0.000125530077', '-0.002661777', '0', '0.000000000000000000123']
-->xcomp(x)
ans =
!1 -1 0 1 !
```

dblstr(x) // converts a number from double to string

Converts a double into a floating point x-number with 15 significant digits

```
A =
    0.8422018 - 0.1100917 0.0385321
    0.8366972 0.1651376 - 0.1577982
    - 0.7816514 0.0825688 0.1211009
-->dblstr(A)
ans =
!8.42201834862385D-01 -1.10091743119266D-01 3.85321100917431D-02 !
!
!8.36697247706422D-01 1.65137614678899D-01 -1.57798165137615D-01 !
!
!-7.81651376146789D-01 8.25688073394495D-02 1.21100917431193D-01 !
```

note that `dblstr(x)` is independent from the current format setting

strdbl(x) // converts a number from string to double

Converts x-number into double. The result is approximated to 15 significant digits

```
B =
!-96712345678901230004 0.0002553729925416790188263391 !
-->strdbl(B)
ans =
- 9.671D+19 0.0002554
```

XNUM.DLL

XNUM.DLL contains the core of multi-precision arithmetic functions.

Usually you do not need to call it directly because the interface functions contained in xnum.sci perform all the work. This note explains only general interfacing concepts useful if you want to write your own multiprecision macros.

The dll must be linked to Scilab with the command `link('XNUM.DLL','xdispatcher2','c');` `xdispatcher2` is the only function providing the interface between Scilab and the internal library arithmetic functions

```
-----
function xdispatcher2
input:
    op      integer      operation code
    dgtmax  integer      maximum number of significant digits
    dm[9]   integer      dimensions of the matrices a1,a2,a3 [n1,m1,k1,n2,m2,k2,n3,m3,k3]
    a1[k1]  integer      1st integer matrix formatted as vector of length k1
    a2[k2]  integer      2nd integer matrix formatted as vector of length k2
output:
    a3[k3]  integer      3rd integer matrix formatted as vector of length k3
    msg     string       error message
-----
```

The operation code specifies which operation is requested according to the following list
Code Operation

```
-----
101  .x
102  .+
103  .-
104  ./
105  matrix product A*B
106  linear system AX = B
107  compare A<=>B ?
108  logarithm in any base "b"  log(x,b)
109  exponential in any base "b"  b^x
110  combinations C(n,k)
111  modulo
99   matrix inverse
98   -x  change sign
97   |x| absolute
96   [x] floor
95   rounding
94   rounding relative
93   () spare
92   integer power x^n (additional parameter n)
91   matrix integer power A^n (additional parameter n)
90   n-root A^(1/n) (additional parameter n)
89   exponential e^x
88   logarithm in base "e" ln(x)
87   sin(x)
86   cos(x)
85   tan(x)
84   asin(x)
83   acos(x)
82   atan(x)
81   factorial n!
39   matrix determinant
38   MCD GCD
37   MCM LCM
36   sum
35   mean
34   var (additional parameter n=0,1 switches between sample and pop. stdev)
33   stdev (additional parameter n=0,1 switches between sample and pop. stdev)
-----
```

I/O Matrices are passed as vector of ASCII code. The code = 44 (comma) separates each number.

Example. A = | 234 78 |
 | -50 8.5 |

is converted as

Char =	2	3	4	,	-	5	0	,	7	8	,	8	.	5
ASCII =	50	51	52	44	45	53	48	44	55	56	44	56	46	53

This corresponds to the vector: `V = [50 51 52 44 45 53 48 44 55 56 44 56 46 53]` (length=14)

Note that matrix are written and accessed for column order. `A(i,j) = V(i + j*n)`

The conversion task is performed in Scilab by the function `mat_ascii_chain(a)` of `xnum.sci`

The vector `Dm[]` contains information relate to the size of I/O matrices
Input: `A(na, ma)`, `B(nb, mb)`. Output: matrix `C(nc, mc)`

`Dm[1]` = rows of 1st input matrix
`Dm[2]` = columns of 1st input matrix
`Dm[3]` = length of 1st input vector
`Dm[4]` = rows of 2nd input matrix (or additional input parameter `n`)
`Dm[5]` = columns of 2nd input matrix
`Dm[6]` = length of 2nd input vector
`Dm[7]` = rows of output matrix
`Dm[8]` = columns of output matrix
`Dm[9]` = length of output vector

For operation requiring only one input matrix the length `Dm[6]` is set to 0
For operation requiring only one input matrix, `Dm[4]` is used for passing additional parameter such as the integer power "`n`" of the operation `A^n`

Error or warning messages are come back by parameter `msg` (max 80 char).

The output vector must be converted back from ASCII code to string matrix.
This conversion task is performed in Scilab by the function `mat_ascii_back(v, n, m)` of `xnum.sci`
The length of output vector depends by the output matrix size (`n x m`) and by the set significant digits (`dgtmax`). The formula for estimating the length is: $K = (dgtmax + 1) * n * m$

Examples of operation calling

```
// operation = 105 matrix product
```

```
Example.      A = | 234   78 |   b = | -3.5 |  
              | -50   8.5 |       |  0.5 |
```

```
-->av= mat_ascii_chain(a);  
-->bv= mat_ascii_chain(b);  
-->ka=size(av,2);  
-->kb=size(bv,2);  
-->kc=36*size(a,1)*size(b,2);  
-->dm=[size(a,1), size(a,2), ka, size(b,1), size(b,2), kb, size(a,1),size(b,2),kc];
```

The content of vectors `av`, `bv` and `dm` should be:

```
av =[50 51 52 44 45 53 48 44 55 56 44 56 46 53]  
bv =[ 45 51 46 53 44 48 46 53]  
dm =[2, 2, 14, 2, 1, 8, 2, 1, 72]
```

```
[cv,s]= call('xdispatcher2', 105 , 1 , 'i' , 30, 2, 'i', dm, 3 , 'i' , av , 4 , 'i', bv , 5 , 'i',  
            'out', [1 , dm(9)] , 6 , 'i', [1 , 80] , 7 , 'c' );
```



© 2007, by Foxes Team
ITALY

Sept 2007