Multiprecision Gamma function G(x)

Leonardo Volpi, April, 2007

Scope

This note explains how to develop routines for computing gamma function for real values with multiprecision. They use the freeware multiprecision ActiveX library Xnumbers.dll for VB 6, but the algorithms could be implemented as well in C/C++ using, for example, the MPFR multiprecision library.

Remark about gamma function

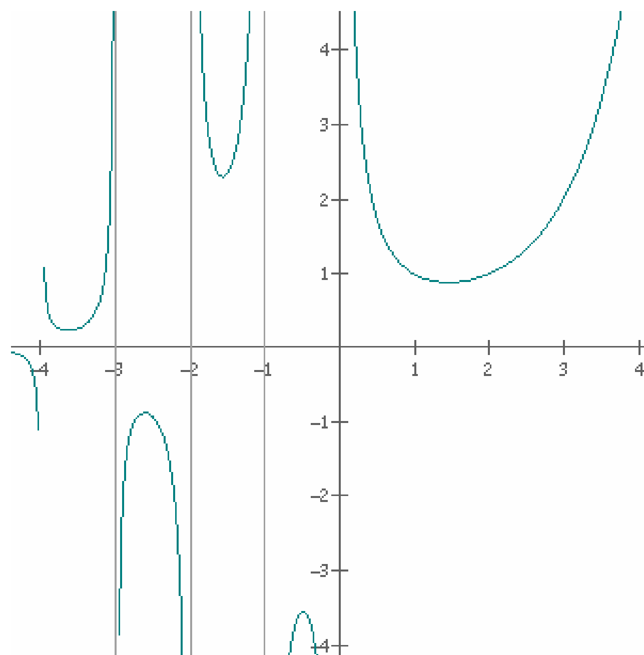
The gamma function can be defined as a definite integral (Euler's integral form).

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \qquad (1)$$

The extension to negative values, $x < 0$, can be done by the following identity (reflection identity)

$$\Gamma(x)\Gamma(-x) = -\frac{p}{x \sin(p\ x)} \qquad (2)$$

That is valid for any x except 0 and integer values. The gamma function, as shown in the following graph, is defined everywhere except at 0, -1, -2, -3, ...



The following recurrent relations are always valid

$$\Gamma(x+1) = x\Gamma(x) \qquad (3)$$

$$\Gamma(n+u) = \Gamma(u) \prod_{i=1}^{n} (n-i+u) \qquad (4)$$

where $n = [x] - 1$, and $u = x - n$

The reduction formula (4), can be used for computing Γ(x) for any value x if we know the value Γ(u) with $1 < u < 2$.
Another identity formula, for $x \neq 0$, states

$$\Gamma(x) = \Gamma(1+x)/x \qquad (5)$$

This formula is useful for computing the gamma function with small positive argument. In fact when $x \to 0^+$, $\Gamma(x) \to \infty$ and all the approximating formulas leads to precision loss. With the formula (4), on the contrary, we lead again the evaluation to the well posed interval [1, 2]

Spouge formula

This algorithm computes the gamma function for |x| < 171 with variable precision up to 180 digits. It uses the Spouge's formula, rewritten here in way that avoid large oscillation of the coefficients

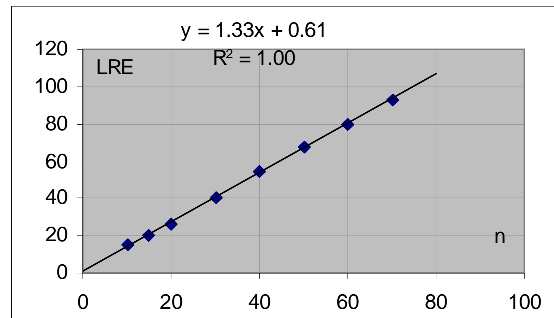
$$\Gamma(1+x) = (x+n)^{x+0.5} e^{-x} \left[b_0 + \sum_{i=1}^{n-1} \frac{b_i}{x+i} \right] \qquad (6)$$

where the n coefficient bi are:

$$b_0 = e^{-n}\sqrt{2p}\,, \qquad b_i = \frac{(-1)^{i-1}(n-i)^{i-0.5}e^{-i}}{(i-1)!} \qquad (7)$$

The number n is choosen by the precision wanted; with $0 < x < 1$ it can be estimated by the following simple formula $n = [0.75 \cdot \text{digits}]+1$

n	LRE
10	15
15	20
20	26
30	40
40	54
50	68
60	80
70	93



For example, if we want to implement a gamma function having a precision of 15 digits in the interval $0< x < 1$, we have to choose n = 12.
The explicit coefficients, calculated with the formulas (7) are shown in table at the right.
Note that, using the reduction formula (4), the reflection formula (2) and the expansion formula (5), we can compute the function for any argument value.

```
b(0) =  1.54012564101896E-05
b(1) =  1.2201180744513
b(2) = -4.279677428117
b(3) =  6.04912880669547
b(4) = -4.42064638634256
b(5) =  1.78343599950919
b(6) = -0.393444736450643
b(7) =  4.42497843410595E-02
b(8) = -2.18103956172808E-03
b(9) =  3.47824207712617E-05
b(10) = -9.05893423645743E-08
b(11) =  4.60254100260297E-12
```

The following code computes the Spouge series coefficients for any value of n.
The parameter DgtMax set the arithmetic accuracy, that shoul be set about 20% more that the precision required for the gamma function; that is DgtMax = 1.2·digits.

```
'computes the n-th Spouge's coefficients in multiprecision (Digits_Max <= 200)
Private Sub Spouge_coeffx(b, n, DgtMax)
Dim i&, p, e, y, s, u
Dim MP As New Xnumbers
With MP
```

```
    .DigitsMax = DgtMax
    ReDim b(n)
    b(0) = .xmult(.xSqr(.x2pi()), .xexp(-n))
    p = 1
    e = .xE()
    For i = 1 To n - 1
        y = .xdiv((n - i), e)
        u = .xpow(y, i)
        s = .xSqr(n - i)
        b(i) = .xdiv(u, s)
        If i > 2 Then
            p = .xmult(p, (i - 1))
            b(i) = .xdiv(b(i), p)
        End If
        If i Mod 2 = 0 Then b(i) = .xneg(b(i))
    Next i
    b(n) = 0
End With
Set MP = Nothing
End Sub
```

The statement "`Dim MP As New Xnumbers`" at the beginning of the subroutine, declares and creates a new instance of the multiprecision library, Of course, because it works correctly, you have to link this code with the xnumbers.dll library (see xnumbers.dll reference for further details).

Now let's see how Spouge's method work practically. The following code implement the gamma function $y = \Gamma(x)$

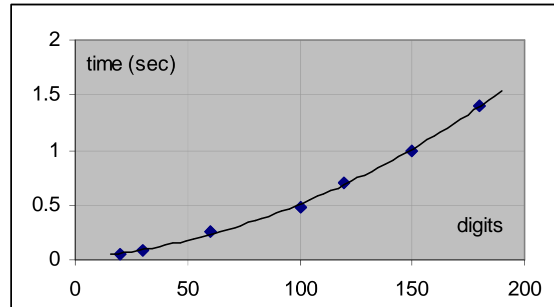
```
'Multiprecision Gamma function ' 15 <= Digits_Max <= 180
'with Spouge formula
'accepts also negative values x<0, with x <> 0, -1, -2, -3...
Sub mp_gamma1(x, y, Optional Digits_Max = 15)
Dim DgtMax&, n&, t, xa, m&, z
Dim MP As New Xnumbers
'input check
If Int(x) = x And x <= 0 Then
    y = "?"
    Exit Sub
End If
'arithmetic accuracy estimation
DgtMax = Int(Digits_Max * 1.2)
If DgtMax > 200 Then DgtMax = 200
'Spouge's serie terms estimation
n = Int(Digits_Max * 0.75) + 1
With MP
.DigitsMax = DgtMax
xa = .xAbs(x)
'input value reduction
If CDbl(xa) >= 1 Then
    m = Int(xa) - 1
    z = .xSub(xa, m + 1)
Else
    z = xa
End If
'compute (z+n)^(z+0.5)*exp(-z)
t = .xmult(.xExpBase(.xadd(z, n), .xadd(z, 0.5)), .xexp(-z))
'compute the Spouge's serie coefficients
Spouge_coeffx b, n, DgtMax
s = b(0)
For i = 1 To n - 1
    s = .xadd(s, .xdiv(b(i), .xadd(z, i)))
Next i
t = .xmult(t, s)
'recursive productory
If CDbl(xa) >= 1 Then
    p = 1
    For i = 1 To m
```

```
        p = .xmult(p, .xSub(xa, i))
    Next i
    t = .xmult(t, p)
Else
    t = .xdiv(t, xa)
End If
'negative argument transform
If x < 0 Then
    '-pi/(x*sin(pi*x))/t
    p = .xmult(xa, .xSin(.xmult(.xpi(), xa)))
    t = .xneg(.xdiv(.xdiv(.xpi(), p), t))
End If
y = .xroundr(t, Digits_Max)
End With
Set MP = Nothing
End Sub
```

For example, here are the first 60 digits of gamma function for x = 1.74

$\Gamma(1.74)$ = 0.916826025151838603000657014812211836876760872759860492036195

Of course, all that accuaracy has a cost. The routine spend about 0.25 sec[1] for reaching this result and the time increases with the accuracy as shown in the graph.
We see that a precision of 30 digits need less then 0.1 sec but it takes more that 1 sec to reach a precision better that 150 digits !



Lanczos formula

This algorithm computes the gamma function for |x| < 171 with quadruple precision (30 digits). It uses the efficient Lanczos's formula

$$\Gamma(1+x) = \frac{e^g}{\sqrt{2p}} \cdot \left(\frac{x+g+n}{e} \right)^{x+0.5} \left[c_0 + \sum_{i=1}^{n-1} \frac{c_i}{x+i} \right] \qquad (8)$$

The parameter g set the global precision. For 30 digits g = 22 and n = 22.
The Lanczos series coefficients, approximated at with 40 digits are enclosed as string in the following routine. Remember that if you system is set with comma as decimal point, you have to replace consequently the period "." with comma "," in all coefficients.

```
'Computes the gamma function in extended precision with the Lanczos formula
'accuracy better then 1E-30 for 1E-6 < |x| < 170
'accepts also negative values x<0, with x <> 0, -1, -2, -3...
Sub mp_gamma2(x, y)
Dim expo, mant1, mant2, w, z, a, b, t
Dim G_, e_, Coef, s, i&, e1, tiny
Dim MP As New Xnumbers
'input check
If Int(x) = x And x <= 0 Then
    y = "?"
    Exit Sub
```

[1] The elaboration time is always referred to a Pentium PC with 1.2 GHz, 256 RAM, if not differently indicated

```
End If
'
With MP
.DigitsMax = 40
tiny = 0.8
G_ = 22
e_ = .xe()
q = .xAbs(x)
If CDbl(q) <= tiny Then  'small value
    z = q
Else
    z = .xSub(q, 1)
End If
b = .xadd(z, 0.5)
a = .xdiv(.xadd(z, G_ + 0.5), e_)  '(z+g+0.5)/e
''compute a^b
t = .xExpBase(a, b)
''now compute the Lanczos series
'' S= 1/w*[c0+c1/(z+1)+c2/(z+2)...c21/(z+21)]
w = .xdiv(.xexp(G_), .xsqr(.x2pi()))     'W = exp(g)/sqrt(2Pi)

'Lanczos coefficients
    Coef = Array("1", _
    "4.0221721696794202128382998278369337394 11E9", _
    "-2.9619500913893162799596454573159445754 60E10", _
    "9.8561883837468292561475489495630998598 78E10", _
    "-1.9602333787164181801822801492985820583 60E11", _
    "2.5973081393828707518841777621085815006 53E11", _
    "-2.4196328071290056137367686176888269674 62E11", _
    "1.6304146799891162137272154713698284794 65E11", _
    "-8.0539612125058664200081672924950039324 04E10", _
    "2.9255133231690388479886902304379131446 07E10", _
    "-7.7709332821232008629095410119727894108 40E9", _
    "1.4888841222625153136702126099979860402 60E9", _
    "-2.0115153315934808483040728946060597398 97E8", _
    "1.8531720691432275113724090647287779403 33E7", _
    "-1.1099114783471324312185914728905910242 14E6", _
    "4.0373708884448177681170664570979260893 69E4", _
    "-808.13071783026187595727390670068500246 54", _
    "7.6717510097986206590140097003383685677 46", _
    "-0.0272512049703047679713633932695809296 3569", _
    "2.3980517596888325749345480396864521929 37E-5", _
    "-2.2981682391174769389519106424208056888 44E-9", _
    "2.9931150883621645672085432206454681268 24E-15")  'for G_ = 22

s = Coef(0)
For i = 1 To UBound(Coef)
    s = .xadd(s, .xdiv(Coef(i), .xadd(z, i)))
Next
s = .xdiv(s, w)
''finally compute the gamma function S*a^b
t = .xmult(s, t)
If q <= tiny Then t = .xdiv(t, q)  'small value

'negative value transformation
If x < 0 Then
    tmp = .xmult(-q, .xsin(.xmult(.xPi(), q)))
    t = .xdiv(.xPi(), .xmult(t, tmp))
End If
y = .xRoundr(t, 30)
End With
Set MP = Nothing
End Sub
```

Thanks to its efficiency, this algorithm is quite faster, at the same precision, then the previous one. It does not need the recursive reduction formula. It takes less then 0.03 sec for all values x 1E-6 < |x| < 170 , with an accuracy of 30 significant digits in the entire range.

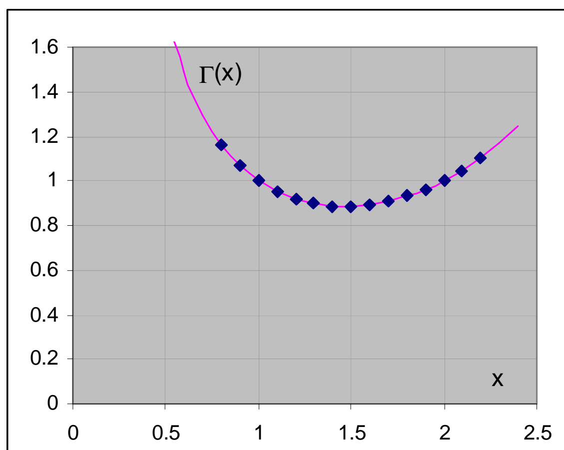
On the other hand the computing of the coefficients is not simple as in the sponge algorithm and they must be recalcuated from scratch when we design a new routine for a different precision range.

Continue Fraction formula

This algorithm also computes the gamma function for |x| < 171 with quadruple precision (30 digits). But it uses the continue fraction interpolation method

$$\Gamma(x) = a_0 + \cfrac{x - x_0}{a_1 + \cfrac{x - x_1}{a_2 + \cfrac{x - x_2}{a_3 + \dots}}} \tag{9}$$

where $[x_0, x_1, \dots x_n]$ are the knots of interpolation and $[a_0, a_1, \dots a_n]$ are the coefficients
The number n+1 of knots, as we can easily imagine, gives the accuracy of the algorithm
Conceptually speaking this methods is much more simple then the other methods.
Choosing a set of n+1 knots of gamma values containing the range that we want to interpolate, for example $1 < x < 2$, we simply compute the coefficients of the continue fraction interpolation. The values of the gamma function can be taken by other source (math programs, tables, etc.).
In that case we have chosen a set of 15 points, between 0.8 and 2.2 with fixed step $h = 0.1$.
Experimental tests shows that this set assures a precision better then 16 digits for all values $1 < x < 2$



x	G(x)	coefficients
0.8	1.16422971372530	1.16422971372530
0.9	1.06862870211932	-1.04601403604546
1	1	-0.582097976
1.1	0.95135076986687	3.17518934563744
1.2	0.91816874239976	-1.34468149377593
1.3	0.89747069630628	4.79683190289953
1.4	0.88726381750308	0.21626283144713
1.5	0.88622692545276	-9.94193351153482
1.6	0.89351534928769	0.88662503123056
1.7	0.90863873285329	22.33231143961830
1.8	0.93138377098024	-0.11395642996203
1.9	0.96176583190739	-21.66305100677670
2	1	-0.699929198
2.1	1.04648584685356	-8.13904972307284
2.2	1.10180249087971	0.63091282561672

Also the Lanczos formula requires about 15 coefficients in order to reach the same accuracy but we have to consider that it works for a wide range |x| < 171, while the continue fraction interpolation works only in the range $1 < x < 2$.
In order to cover the range |x| < 171 , the continue fraction algorithm needs the recurrent reduction formula (4), that generally takes more time. Lancozos formula is surely more efficient but we have to consider, thus, that the continue fraction interpolation, involving only elementary arithmetic operations, is more simple and light then the Lanczos formula. We hope this compensates the time spent in the reduction formula.
Another advantage of the continue fraction interpolation algorithm regards the software coding that is quite compact and straight. Just for example the following routine implements the gamma function in double precision (15 digits) for |x| < 171.

```
'returns the gamma function in double precision (1E-15)
'accepts arguments |x| < 171 and x<0, with x <> 0, -1, -2, -3...
```

```
Function gamma(x)
Dim coeff, y#, knot, n&, i&, z#, m&, pi_#, xa#, p#

'argument check
If Int(x) = x And x <= 0 Then
    gamma = "?"
    Exit Function
End If
xa = Abs(x)
If xa >= 1 Then
    m = Int(xa - 1)
    z = xa - m
Else
    z = 1 + xa
End If
coeff = Array( _
  1.1642297137253, -1.04601403604546, -0.58209797628432, _
  3.17518934563744, -1.34468149377593, 4.79683190289953, _
  0.21626283144713, -9.94193351153482, 0.88662503123056, _
  22.3323114396183, -0.11395642996203, -21.6630510067767, _
  -0.69992919755029, -8.13904972307284, 0.63091282561672)

n = UBound(coeff)
ReDim knot(n)
For i = 0 To n: knot(i) = 0.8 + i * 0.1: Next
'continue fraction expansion
y = coeff(n)
For i = n - 1 To 0 Step -1
    y = coeff(i) + (z - knot(i)) / y
Next i
If xa >= 1 Then
    'recursive productory
    p = 1
    For i = 1 To m
        p = p * (xa - i)
    Next i
    y = y * p
Else
    y = y / xa
End If
If x < 0 Then
    'negative argument transform
    pi_ = 4 * Atn(1)
    y = -pi_ / (xa * Sin(pi_ * xa)) / y
End If
gamma = y
End Function
```

Thanks to its high speed, it is implemented as VBA macro function for Excel spreadsheet. It takes less then 0.1 ms sec for computing the gamma function for any argument in the range |x| < 171 with a good accuracy (error less then 1E-15).
This is remarkable because, this routine does not require any other external library

The following subroutine performs the same task but with 30 digts precision and it requires the Xnumbers.dll multiprecision library.

```
'Computes the gamma function in extended precision with the continue fraction
'accuracy better then 1E-30 for 1E-6 < |x| < 170
'accepts also negative values x<0, with x <> 0, -1, -2, -3...
Sub mp_gamma3(x, y)
Dim coeff, knot, n&, i&, z, m&, pi_, xa, p
Dim MP As New Xnumbers
'argument check
If Int(x) = x And x <= 0 Then
    y = "?"
```

```
    Exit Sub
End If
With MP
.DigitsMax = 40
xa = .xAbs(x)
If CDbl(xa) >= 1 Then
    m = Int(xa - 1)
    z = .xSub(xa, m)
Else
    z = .xAdd(1, xa)
End If
ReDim coeff(30)
'continue fraction coefficients
coeff(0) = "1.298055332647557785681171179152811 61"
coeff(1) = "-0.6883389716251620054994656470110554"
coeff(2) = "-0.8488660503934381846778310070455584"
coeff(3) = "3.256306683356039724373498708692452"
coeff(4) = "-1.765801460293971794708728558029223"
coeff(5) = "0.9962945362919307743350858489130241"
coeff(6) = "0.9460372099388620804735105394100844"
coeff(7) = "-9.354089354904358458998281947554 44"
coeff(8) = "0.6996356373174796357342960205828161"
coeff(9) = "-32.80049800288017580219606918385201"
coeff(10) = "-2.777692198875000987076415316495102E-2"
coeff(11) = "44.51368982912936020271161609644044"
coeff(12) = "-0.4770600420414227024396313381876586"
coeff(13) = "-38.66150551259325654936431331 63229"
coeff(14) = "0.1200458585737961309958674510605318"
coeff(15) = "44.83280894989578002282996116905182"
coeff(16) = "0.3134583802232278954871577520874977"
coeff(17) = "15.19801830038985082731789493648723"
coeff(18) = "-0.3246252892846237884276265628630774"
coeff(19) = "-80.57807362771244935429833231256949"
coeff(20) = "0.1944279417341480622104345393432769"
coeff(21) = "-67.92489578269287391774863426300384"
coeff(22) = "-3.703382652715721261377634290764893E-2"
coeff(23) = "159.3181730883064622570179505405649"
coeff(24) = "-0.1465541847205154017487323991154874"
coeff(25) = "-163.8478601551737280120002585418295"
coeff(26) = "4.384943427103193281324627347834941E-2"
coeff(27) = "18.74151458434201630012609148665023"
coeff(28) = "4.052522336907972391802000916514176E-2"
coeff(29) = "-57.26360570499208965219767272527122"
coeff(30) = "-1.298859127309160565387005917257284E-2"
n = UBound(coeff)
ReDim knot(n)
For i = 0 To n: knot(i) = 0.7 + i * 0.05: Next
'continue fraction expansion
y = coeff(n)
For i = n - 1 To 0 Step -1
    y = .xAdd(coeff(i), .xDiv(.xSub(z, knot(i)), y))
Next i
If CDbl(xa) >= 1 Then
    'recursive productory
    p = 1
    For i = 1 To m
        p = .xMult(p, .xSub(xa, i))
    Next i
    y = .xMult(y, p)
Else
    y = .xDiv(y, xa)
End If
If x < 0 Then
    'negative argument transform
    pi_ = 4 * Atn(1)
    y = -pi_ / (xa * Sin(pi_ * xa)) / y
End If
y = .xRoundr(y, 30)
End With
End Sub
```

This routine takes about 0.03 sec for computing 30 digits of the gamma function for $|x| < 60$. and less then 0.06 sec for $60 < |x| < 170$. As we can see, for moderate value of x, this simple algorithm is competitive with the more sophisticated algorithms.

Software

All the code shows here is available in the module mpgamma.bas contained in the same package of this document

Bibliography

"An Analysis of the Lanczos Gamma Approximation", by Glendom Ralph Pugh, Thesis, University of British Columbia, July 2004.

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

"Lanczos Implementation of the Gamma Function" by Paul Godfrey, Intersil Corp, 2001

"A Yet Another Shot at Gamma", Zdzislaw Meglicki , 2001-02-26, Web page, http://beige.ucs.indiana.edu

"Calculators and the Gamma Function", Viktor T. Toth, 2002, 2002, Web page, www.rskey.org

"Handbook of Mathematical Functions", M. Abramowitz and I. Stegun, Dover, 1964

"Compute Gamma(x) (and x factorial) to arbitrary high precisions", Kenneth Wilder, Web page, http://oldmill.uchicago.edu/~wilder/Code/hpgamma