Testing the improved accuracy of Excel 2010 - a joyride to Numerics

(Or: why don't you show up your complete results?)

AVt, Mar 2010

Actually I only wanted to do some tests after reading "Function Improvements in Microsoft Office Excel 2010" (Oct 2009) at Microsoft's advertising blog, which sounds quite wholehearted (especially if reading the section "Process for Implementing Changes" saying that expertise from NAG was used).

http://blogs.msdn.com/excel/archive/2009/09/10/function-improvements-in-excel-2010.aspx
http://blogs.msdn.com/excel/archive/2009/10/08/going-back-to-the-topic-of-functions-for-a-moment.aspx
http://blogs.msdn.com/excel/attachment/9905140.ashx (= the above mentioned document as pdf)

Why not simply taking some results given by Excel, feed Maple with higher precision and cross check to find expected inaccuracies?

Going that way quickly show errors.

But it is an incomplete way to find out the full story: it turns out, that this is only based on the mediocre precision at the user level allowing mere 15 significant decimal places in I/O routines.

While internally Excel's accuracy is much better. And testing that should be done regarding IEEE 754 for double precision and not wildly by comparing decimal representations. Which is not that simple and quick. Of course I only picked some special functions and do not want to check all the herd.

Testing was done on Windows 7 for Excel 2010 Beta and both as 32 bit version (so I can use can use an older Maple version on that system or older DLLs if needed – a quite typical demand of users in next future, even if the machine below would allow the full 64 bit versions).

Though I used the beta version I think that this part of Excel might be the same in the upcoming final role out.

1. The nasty limit of 15 decimal places: there is more than you can see but get.................................... 1
2. Usually deliberately ignored: double precision numbers in IEEE 754.. 3
3. The 'correct' values and measuring errors using Maple 3
4. Some Test results.. 4
   4.1 The error functions erf and erfc 4
   3.2 The cumulative normal.. 5
   3.3 The inverse cumulative normal... 7

1. The nasty limit of 15 decimal places: there is more than you can see but get

There is still a notorious problem, both for worksheet and VBA code: there is no direct way to have more than 15 decimal places. Even if one chooses better I/O formats: all the places beyond are just trailing zeros and worthless.

However one needs 17 and usually uses 18 decimal places to convert correctly between a common input and its binary value, say for entering sqrt(2) = `1.41421356237309505`, which in VBA will be cut down to `1.4142135623731`.

This is a feature (and a really bad one for those who want to do computations, may be once it was introduced to 'protect' users from trusting in all decimals they would see), which prevents to get the actual value, which Excel is working with.

There is a way to bypass it by writing `x = head(x) + tail(x)`, where head is the displayed value and tail is the difference to the actual value.

```
Function tostr(x As Double) As String
  Dim sx As String
  Dim localDecimalSeparator As String  ' to care for language settings
  'Dim localThousandsSeparator As String

  localDecimalSeparator = Application.International(xlDecimalSeparator)
  '--------------- 1.2345678901234567 8
  sx = Format(x, "+0.0000000000000000E+000;-0.0000000000000000E+000")
  sx = Replace(sx, localDecimalSeparator, ".")
  tostr = sx
End Function

Function head(x As Double) As Double
  head = CDbl(Evaluate(tostr(x)))
End Function

Function tail(x As Double) As Double
  tail = x - head(x)
End Function

Function strhead(x As Double) As String
  strhead = tostr(x)
End Function

Function strtail(x As Double) As String
  strtail = tostr(tail(x))
End Function
```

Now use this like in the following example:

```
Sub tst_headtail()
  Dim y As Double, r As Double, y0 As Double, y1 As Double

  r = 1.4142135623731 ' 15 decimals enforced by Excel
  y = Sqr(2#)         ' internal value
  y0 = head(y)
  y1 = tail(y)

' in the the debug window r and y are shown as the same decimal numbers
  Debug.Print r, y

' while Excel knows it better, they are different and r is only the head,
' it prints false and true for the following command:
  Debug.Print r = y, r = y0

' head and tail = 1.4142135623731 and -4.88498130835069E-15
  Debug.Print y0, y1

' and adding them gives the correct value: y = head + tail,
' Excel confirms it by printing 'true':
  Debug.Print y = y0 + y1

End Sub
```

So we have an artificial way to recover the missing trailing places which Excel actually uses internally and will use that for judging results.

Input can be done similar by working through such pairs.

2. Usually deliberately ignored: double precision numbers in IEEE 754

Almost certainly Excel uses C runtime libraries for numerics and comparing or judging numerical values should be done regarding that – and not by just looking at displayed results (besides the '15 Digits Limitation' different decimal numbers may stay for the same binary number).

Recall the rough basics, for which Higham "Accuracy and stability of numerical algorithms" is a good reference (and if you do not want that just take as fact: checking accuracy is done at that level):

A double precision x writes as `x = m * 2^d`, where ½ <= m < 1, m a rational number (called the mantissa or sometimes the significant), d is an integer (called the exponent), within some limits.

More precisely m is a (finite) binary fraction, m = $\sum_{j=1}^{53} \varepsilon(j) * 2\text{^}(-j)$, $\varepsilon(j)$ = 0 or 1 (yes, 53 bits).

As a rounding rule the usual "round nearest, tie even" is taken for passing from decimal to binary presentation and back.

Comparing and judging results is done for the according *nearest* IEEE numbers only:

It would not make sense to expect something else: something like $\sqrt{2}$ can only represented by such a thing, everything beyond only would talk about limitations of presenting an abstract number through a finite floating point number using IEEE.

3. The 'correct' values and measuring errors using Maple

For testing a function f write x and y = f(x) to a text file and read that from Maple to process it. Since that is done for a test range of input values it makes sense.

Testing a function f is limited by the 15 decimal shit. Thus consider x and y = f(x) through 'heads' and 'tails' for x and y. Because they stand for a IEEE double one knows that their sum can be represented exactly as such a number - which is a rational number.

This can be done in Maple, both for correct addition (choosing enough precision) as well as finding the according IEEE number (rounding mode = nearest, tie to even).

Name them X and Y.

Evaluate that in Maple for the according function F using high precision and good settings to get a floating point number (say of decimal length 60).

Then one can compute *the* (!) nearest IEEE value for Y = F(X), calling it $Y_{correct}$.

This $Y_{correct}$ is the value which one could expect as best result in a C program in double precision (on which Excel is certainly based), it is exactly the binary value written in decimals as fraction of integers. And it does not depend on any decimal I/O.

Looking for absolute and relative errors is done through that. And *not* directly through any other I/O routines of Excel, may it be in a worksheet or displayed in a debug window.

Except for f(x) ~ 0 for all judging I prefer the relative error $error_{relative} = (Y – Y_{correct}) / Y_{correct}$.

As a kind of natural scaling I want to see it in multiples of `DBL_EPSILON = 1 / 2^52`, since this is a kind of limitation in double precision.

It roughly is responsible for how many correct places one may loose in a decimal representation (if it is ~ 2^10 = 1024, then the last 3 places are worthless, if displaying 17 significant decimal places).

Or otherwise said: how many significant decimals are present (that's what one usually wants to know).

This way one even can have test protocols usable by those, who do not have Maple.

Another way is:

Feed a numerical value x to a function returning just that as double (so the function is the identity).

Take the AdressOf operator, send it to a DLL. Then within the DLL one has access to x and it automatically has exactly the same value as in Excel (the DLL just takes it from the same memory space).

Now use OpenMaple – which allows to communicate between Maple and a C program and through that IEEE numbers are correctly converted.

That would allow to compute function values in Maple with (very) high precision. After cutting down and being rounded correctly to IEEE through that communication one has the correct IEEE in the DLL and can finally send back to Excel the value and (all) kinds of errors, which could have been computed in the same way.

This sounds fine and avoids the I/O problem with 15 decimal places – but I am not sure whether Maple is ready for Win 7. And I would have to code that (with careful tests ☹). And would still have to care for roundings … (and OpenMaple needs a version for which I would violate some license agreements).

4. Some Test results

Test results are really fine for those elementary function at which I looked (but was to lazy to check powers, especially for larger integer exponents or non-rationals):

The relative error (measured in `DBL_EPSILON`) was `0` or `+-1`.

After wasting times I guessed they will rely on those implementations which they do have anyway in a MS C library and it only makes sense to test something beyond (assuming that the I/O routines are correct).

And what is mentioned in their blog and to be known as being quite bad before – the special functions, as they are used in Statistics or technical applications.

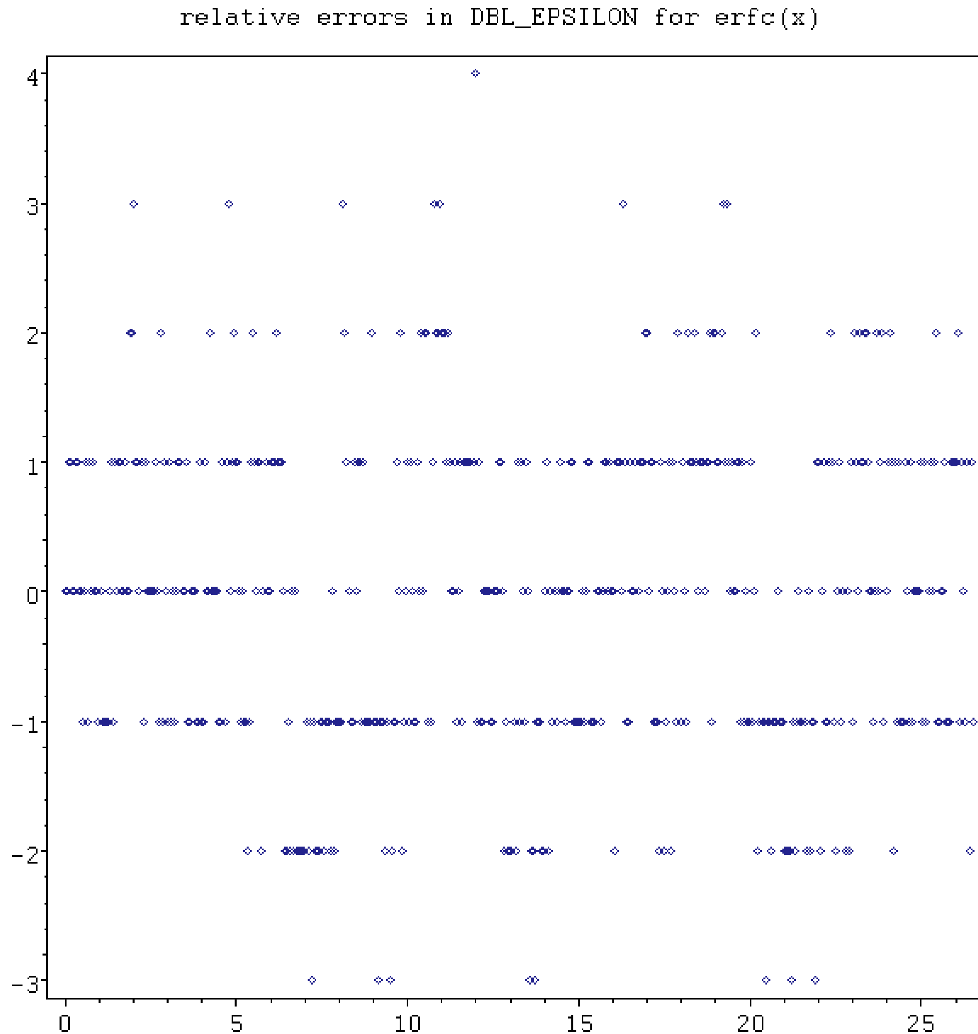
I picked up: the error function `erf`, its complement `erfc = 1 - erf` and the cumulative standard normal cdfN.

4.1 The error functions erf and erfc

Within a fixed floating point environment one has no reasonable possibility to check values very close to 1, since `1 - DBL_EPSILON / 2` is a natural limit (nevertheless I did that as well).

But one can look at input values close to 0 and that means to look at erfc on the positive axis.

Excel 2010 behaves excellent and gives relative errors below ~ `4 DBL_EPSILON` :



3.2 The cumulative normal

The cumulative normal can be reduced to the negative axis by `cdfN(x) = 1 - cdfN(-x)` where it can be expressed through the complementary error function, `cdfN(-x) = erfc(-x/sqrt(2))/2`.

Despite the results for `erfc` Excel 2010 gives that with a relative error up to ~ 1000 `DBL_EPSILON`.

Since in some cases errors are given in ULP (=unit in last place) – for example newer books on CS or Numerics occasionally do it, or the CUDA library does it in its documentation – here I do that as well.

What is it? Higham's book tells it: The last binary place in the mantissa of `f(x) = y = m * 2^d` can be `1*2^(-53)` and `ULP(y) = 2^(d - 53)` as a rule.

So it tells how many of the last binary places are (in)correct.

One has `ULP(1) = 2^(-53) = DBL_EPSILON/2` and `ULP(2) = DBL_EPSILON`, `ULP(+infinity) = 2^(+971)` and `ULP(0) = 2^(-1074)`, which is a 'subnormal' IEEE number.

The relative error in ULPs is $(f(x) - f(x)_{correct}) / ULP(f(x)_{correct})$. It is an integer.

Since the mantissa is between ½ and 1 it follows that a relative error measured in ULPs is between once and twice the relative error measured in `DBL_EPSILON`.

The largest relative error in my test occurred for `x = -36.181640625 = -18525/32768*2^6`:

The correct value (after norming to the nearest IEEE number) is

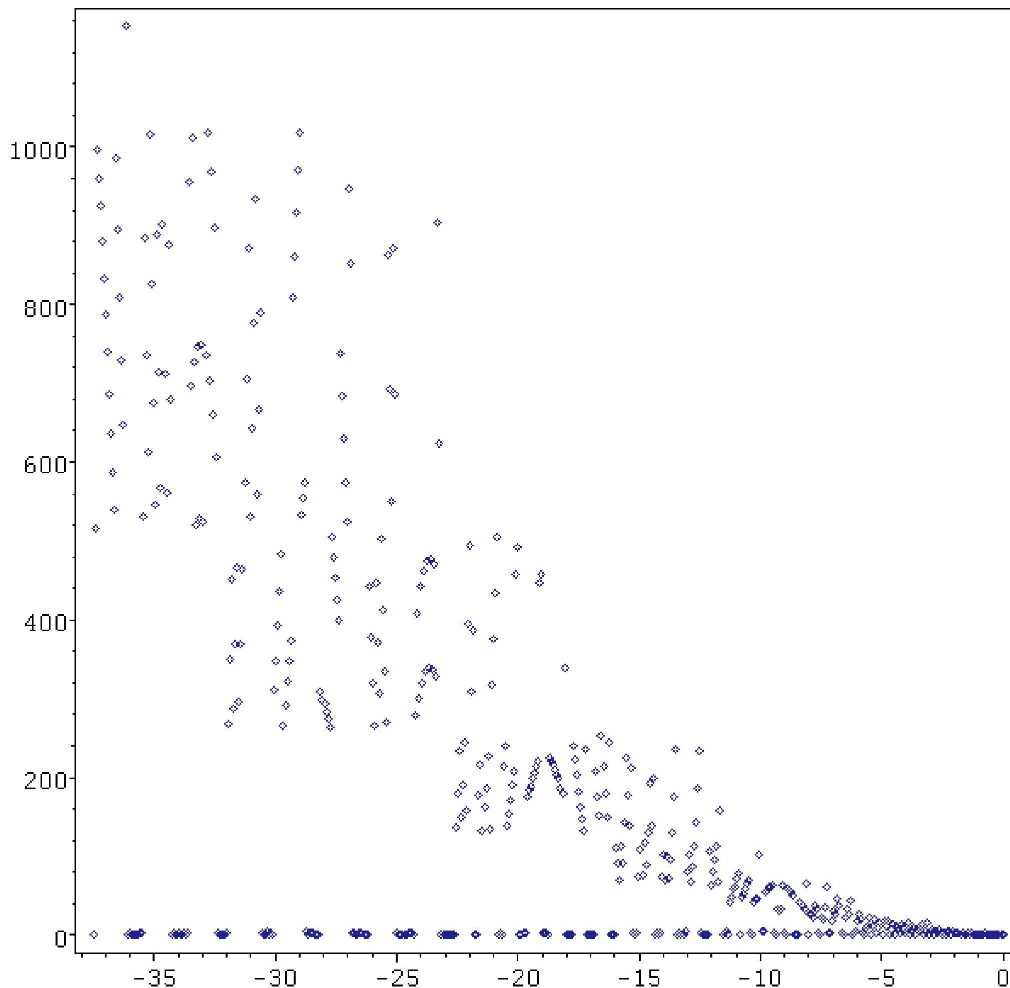
```
correct = 0.5918658087798693 38E-286 = 5073514467255131/9007199254740992*pow(2,-950)
```

while Excel returns (through head and tail as described above)

```
cdfN(x) = 0.5918658087797347 87E-286 = 2536757233626989/4503599627370496*pow(2,-950)
          123456789012^^^^^^
```

The relative error is ~ 1023.48 measured in `DBL_EPSILON` and 1153 measured in ULPs, the leading 12 decimals are correct (and beyond one has the relative error).

Since one can expect 14 – 15 decimal places to be displayed correct for a converted IEEE number without care that fits together: 1023 *`DBL_EPSILON` ~ 0.2E-12 says that the first 12 places are correct.



Using `erfc(-x/sqrt(2))/2` directly in Excel I got similar results … ☹ … *HTF is that possible*?

Grübel, grübel und studier … - it is not (directly) Excel's fault, it comes through error propagation and double precision having IEEE, since sqrt(2) is not exactly representable:

```
P = - x / sqrt(2.0) =
  = - 36.181640625 / 1.41421356237309505 =
  = - 3600667792771605/4503599627370496 * pow(2,5)
  = - 25.5842834403921771
```

While

```
p = - nearest(x) / nearest(sqrt(2)) =
  = - 7201335585543209/9007199254740992 * pow(2,5)
  = - 25.5842834403921736
```

The *absolute* error is very small, `p - P ~ - 16 DBL_EPSILON`.

But the relative error `1 - nearest(erfc(P))/nearest(erfc(p)) ~ 808 DBL_EPSILON`:

We are in an asymptotical situation for t = x / sqrt(2), thus erfc(t) ~ exp(-t^2) / t / sqrt(π) and disturbing t by 16 `DBL_EPSILON` gives the relative error of ~ `800 DBL_EPSILON` , which explains the magnitude (and the rest should follow from the slight inaccuracy of erfc).

Thus one has to have a routine for cdfN itself (and not reducing it to erfc) if one wants to avoid that.

To be fair: within applications an input x will always be a computed value, so the phenomenon will strike anyway. But having a slightly better cdfN would eliminate one unnecessary error propagation.

3.3 The inverse cumulative normal

They say Wichura's algorithm is used (for the tails), which is known to be one of the best standard algorithm, see http://www.mth.kcl.ac.uk/~shaww/web_page/papers/quantiles/Quantiles.htm for details.

For me it is a bit difficult to say what I expect here. While it is more or less clear for values y around ½, it is far from being obvious at the singularities y = 0 or y = 1. And I rarely need it.

For the latter one will stumble into all the limitations for floats close to 1, of course (i.e. representable numbers are too coarse).

So testing makes sense for `0 < y <= 1/2`.

Shooting y values and taking new Excel's invcdfN has relative errors below ~ 15 (mainly in the range `0.08 <= y <= 0.2`, which is `-1.5 <= x <= -1.0`, may be just there is the switch from erf^{-1} to Wichura's method, mentioned in the document), measuring them in `DBL_EPSILON`.

Another way: just expecting and testing for `inv_cdfN( cdfN(x) ) = x`, which depends on the quality of cdfN (hence this kind of test more or less tests the implementation of cdfN and error propagation at the same time).

If taking an improved version for cdfN coded in pure VBA directly (without using new features of Excel, having relative error of `~ 3 DBL_EPSILON` or `~ 5 ULPs`) I get `~ 3 DBL_EPSILON` or `~ 5 ULPs` for the above task and without special behaviour for x around `-1`.