TESTING SOFTWARE FOR SPECIAL FUNCTIONS

Allan MacLeod
University of the West of Scotland

Ceud Mìle Fàilte à Alba

Many thanks to N. I. S. T. and the U.S. Government for financial support in attending this Conference.

20^{th} Century
method for computing special functions:

goto

Abramowitz & Stegun
or Gradshteyn & Ryzhik
or Other

find formula
code formula

21^{st} Century

method for computing special functions:

goto Google

enter function name

download available code in language wanted

Hundreds of languages available.

Algol 60, Simple, IMP, Fortran 4, Algol 68, Basic, Fortran 77, Matlab, Pascal, C, Fortran 90, Ada, C++, Java, Mathematica, Pari, Python

are the languages in which I personally have written programs.

High Quality software for special functions needs:

A deep knowledge of the underlying mathematics of the function being computed.

A deep knowledge of floating-point arithmetic, especially the effects of overflow and underflow.

A deep knowledge of the programming language being used and the possible compilers that might be used.

Testing such software needs all of these and **cunning**.

Two quotes from Edsger Dijkstra (ALGOL 60 and shortest-path algorithm)

Programming is one of the most difficult branches of applied mathematics: the poorer mathematicians had better remain pure mathematicians

Program Testing can be used to show the presence of bugs but never to show their absence

If Code is Perfect **then** No Bugs Present

has logical negation

If Bugs Found **then** Program Not Perfect

Sadly, mainly people think logic goes

If No Bugs Found **then** Code is Perfect

Basic Problem

Let $f(x)$ be the function under consideration, and $F(x)$ the result returned by a software module designed to compute the function.

Several aspects of testing software

1. The **accuracy** of the code. That is, assess the behaviour of the function

$$e(x) = \frac{f(x) - F(x)}{f(x)}$$

where we assume that x is not a zero of the function.

2. The **efficiency** of the code. Exponential Integral $E_1(x)$ can be computed millions of times in ONE Birch/Swinnerton-Dyer conjecture computation.

3. The **robustness** of the code - does it prevent underflow and especially overflow. Different languages and compilers do different things for underflow and overflow.

4. The **documentation** of the code - how easy is the code to use - especially for non-experts.

For example, the HELP facility for Excel 2010 has the following description

The n-th order modified Bessel function of the variable x is:

$$K_n(x) = \frac{p}{2} i^{n+1} \left[J_n(ix) + iY_n(ix)\right]$$

where Jn and Yn are the J and Y Bessel functions, respectively.

Accuracy Testing Method 1

Compare against results from higher-precision computations.

1. How are higher-precision results computed? If we use use same code bumped-up to higher precision, all we are testing is numerical stability **NOT** accuracy. If we use another code, how do we know this comparison code is any good?

2. What if code being tested is already in the highest available precision? For example, several user environments have only one precision.

Accuracy Testing Method 2

Developed by Jim Cody at Argonne National Lab as extension of the **elefunt** elementary-function test software.

Use functional identities to test performance

$$\Gamma(2x) = \frac{1}{\sqrt{\pi}}\, 2^{2x-1}\, \Gamma(x)\, \Gamma(x + 1/2)$$

Compare LHS to RHS.

Reduce possible sources of error to lowest level.

Argument purification:

Half=0.5

Y=random

X=Y*Half

Z=X+Half

X=Z-Half

Y=X+X

Accuracy Testing Method 3

Taylor Series

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \dots$$

where the derivatives can be easily calculated eg. error function, normal distribution function, sine integral

If $f(x) = Si(x)$, we have $f'(x) = \sin(x)/x$, and

$$f^{(n+1)}(x) + \frac{n}{x} f^{(n)}(x) = \frac{\sigma_n(x)}{x}$$

where $\sigma_1 = \cos x$, $\sigma_2 = -\sin x$, $\sigma_3 = -\cos x$, $\sigma_4 = \sin x$, $\sigma_5 = \sigma_1$, $\sigma_6 = \sigma_2, \dots$.

Codes for these derivatives developed by Walter Gautschi.

Accuracy Testing Method 4

Table-based tests, based on the ideas of Liu and Tang originally applied to elementary functions.

$$\begin{aligned} J_0(a+h) &= J_0(a) + hJ_0'(a) + \dots \\ &= J_0(a) - hJ_1(a) + \dots \end{aligned}$$

$$J_0(a+h) = J_{01} + (J_{02} - hJ_{11}) - (hJ_{12} + R_N)$$

where $J_0(a) = J_{01} + J_{02}$ for example, with J_{01} accurate to 12 bits (say) and J_{02} accurate to 23 bits for single precision tests. We thus get extended accuracy for these control values, which are computed in multiple-precision beforehand.

Example 1: Normal distribution function

$$P(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-t^2/2)\; dt$$

$$P(x) = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

Use Taylor-series approach to develop tests.

Craig in 1984 published (in Journal of Quality Technology) a code based on the identity

$$\operatorname{erf}(v) \approx \frac{2}{\pi}\left(\frac{v}{5} + \sum_{n=1}^{37} \frac{\exp(-n^2/25)\,\sin(2nv/5)}{n}\right)$$

for $|v| \leq 5\pi/2$.

The tests showed very poor results in certain regions.

```
      DOUBLE PRECISION FUNCTION DNML(X)
      DOUBLE PRECISION X,Y,S,RN,ZERO,ONE,ERF,SQRT2,PI
      DATA SQRT2,ONE/1.414213562373095,1.D0/
      DATA PI,ZERO/3.141592653589793,0.D0/
      Y=X/SQRT2
      IF(X.LT.ZERO) Y=-Y
      S=ZERO
      DO 1 N=1,37
      RN=DFLOAT(N)
      S=S+DEXP(-RN*RN/25)/N*DSIN(2*N*Y/5)
1     CONTINUE
      S=S+Y/5
      ERF=2*S/PI
      DNML=(ONE+ERF)/2
      IF(X.LT.ZERO) DNML=(ONE-ERF)/2
      IF(X.LT.-8.3D0) DNML=ZERO
      IF(X.GT.8.3D0) DNML=ONE
      RETURN
      END
```

Example 2: Sine Integral

$$Si(x) = \int_0^x \frac{\sin t}{t} dt$$

For large $|x|$, use

$$Si(x) = \frac{\pi}{2} - fi(x)\cos x - gi(x)\sin x \qquad x > 0,$$

and $Si(-x) = -Si(x)$.

Taylor series test showed big errors in **fnlib** code for large negative x. Code was, with absx=$|x|$,

```
call r9sifg (absx, f, g)
cosx = cos (absx)
si = pi2 - f*cosx - g*sin(x)
if (x.lt.0.0) si = -si
```

Code should be (roughly)

```
call r9sifg (absx, f, g)
sinx = sin (absx)
si = pi2 - f*cos(x) - g*sinx
if (x.lt.0.0) si = -si
```

This error pointed out in MacLeod(1996), but code unchanged as of Sunday 27 March 2011!!!!

Example 3: Excel Functions

Statistical distribution functions in all of Excel 97, Excel 2003, Excel 2007 heavily criticized by McCullough et al.

Excel 2010 documents described improvements in Special Functions after discussions with external groups such as Nag.

No Gamma function - just $\ln \Gamma(x)$.

$$\begin{aligned}\ln \Gamma(2x) = -0.5 \ln \pi + (2x-1)\ln 2 + \ln \Gamma(x) \\ + \ln \Gamma(x+1/2)\end{aligned}$$

Results

	$(1.3125, 1.625)$	$(3.5, 5.0)$
Excel 2003	5.4E-11	1.2E-11
Excel 2007	5.4E-11	1.2E-11
Excel 2010	2.8E-15	4.0E-16

Special Function Software in the 21^{st} Century

Most high-quality software originally written in $1960 - 1985$ in Fortran.

Several original developers now retired or dead! Need young blood.

Computers completely different beasts nowadays - PCs dominate and RAM cheap. For example, integer factorisations are now being done on networks of PS3 games consoles.

New software needed to use new facilities - function domains can be divided into far more subregions.

Enormous growth in languages eg Python. Should we continue to use Fortran as standard? Perhaps use Matlab as a standard meta-language.