

---

UNIVERSITY COLLEGE, DUBLIN

SCHOOL OF MATHEMATICAL SCIENCES

Master of Computational Science Degree 2006-2007

NUMERICAL ALGORITHMS

*Dr Derek O'Connor*

---

**Lab Exercise No. 2 : EVALUATING ELEMENTARY FUNCTIONS.**

OUT: Wed 20 Sep 2006

IN : Wed 27 Sep 2006

---

## 1 PURPOSE

The purpose of this exercise is to

- Gain more experience with the MATLAB system.
- Write simple but non-trivial MATLAB functions that use loops.
- Demonstrate the difficulties in evaluating the elementary functions.

## 2 THE ELEMENTARY FUNCTIONS

There are four groups of elementary functions :

1. **Reciprocal-Square Root** :  $1/x, \sqrt{x}, 1/\sqrt{x}$
2. **Exponential** :  $e^x, \log_e x, 2^x, \log_2 x$
3. **Circular** :  $\sin x, \sin^{-1} x, \cos x, \cos^{-1} x, \tan x, \tan^{-1} x$
4. **Hyperbolic** :  $\sinh x, \sinh^{-1} x, \cosh x, \cosh^{-1} x, \tanh x, \tanh^{-1} x$

Most, if not all, of these functions are implemented in silicon on the standard floating point processors. This means that algorithms for calculating these functions, using only  $\{+, -, *, /\}$ , are 'hard-wired' into the processor.

## 3 EXERCISE

1. Write a MATLAB function MySin(x,n) to calculate  $\sin(x)$  using the Taylor-MacLaurin series

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

2. Write a MATLAB function to test this function for  $x = 0, \pm 10^0, \pm 10^1, \pm 10^2, \pm 10^5$ . Use  $n = 10, 100, 1000$  for each value of  $x$ . Compare your results with the built-in MATLAB  $\sin(x)$  function, i.e., calculate the relative error for each test

$$\text{Rel. Error} = \frac{|\text{MySin}(x, n) - \sin(x)|}{|\sin(x)|}.$$

## 4 NOTES

### 4.1 Summing a Sequence

Many numerical calculations have the form

$$S = \sum_{k=0}^n T_k = T_0 + T_1 + T_2 + \cdots + T_n.$$

A program to sum  $n$  terms of a sequence has the general *recurrence* form :

$$S_{k+1} := S_k + T_k.$$

The sum is built up by adding the  $k$ th term of the sequence,  $T_k$ , to the  $k$ th partial sum  $S_k = \sum_{i=0}^k T_i$ . Hence the sum for  $\sin(x)$  above must be re-written in recurrence form first and then translated into program statements (the  $k$ s are dropped).

If the term  $T_k$  is complicated then it should be written in recurrence (or update) form. That is,  $T_k$  is calculated as an update of  $T_{k-1}$ .

Be careful of the boundary conditions, i.e., initial and final values of  $S$  and  $T$ .

### 4.2 Writing Test Functions

Part 2 of this exercise requires you to test  $\text{MySin}(x, n)$  for various pairs of values of  $(x, n)$ . The simplest way to do this is to type in  $\text{MySin}(x, n)$  at the command line for each pair. This is fine if you want to test the function for one or two pairs and see that the function actually works.

However, command-line testing is also the worst way because there are 27 different  $(x, n)$  pairs and you get a huge mess of command-line entries and responses which you have to sift through and record — assuming you have made no errors.

Once you are satisfied that the function is working properly then the systematic testing should be done by a function written to do all tests and summarize the results. In other words, get the computer to do all the mundane repetitive work.

## 5 SOLUTION

### 5.1 Implementation

```

%=====
function sum = MySin1(x,n)
%
% MySin1(x,n) is a naive implementation of Sin(x)
% using the first n+1 terms of the Taylor series.
% Derek O'Connor, Sep 2006.
%
    sum = x;
    term = x;
    for k = 1:n
        term = - (x^2/((2*k+3)*(2*k+2)))*term;
        sum = sum + term;
    end;

```

### 5.2 Programming Notes

Note the following points about this MATLAB function :

- Programming Style : (1) There is a *preamble* that explains the purpose of the function, along with author details etc. When you type `help MySin` in MATLAB you will see this preamble. (2) The statements within *loops are indented* 3 spaces. This makes it easy to read.
- Note the starting values for `sum` and `term` : these represent the case  $n = 0$ . Note that this gives the correct answer for  $x = 0$ .
- The statements `term = -(x^2/((2*k+3)*(2*k+2)))*term` and `sum = sum + term` implement the mathematical statements

$$T_{k+1} = -\frac{x^2}{(2k+3)(2k+1)}T_k \quad \text{m and } S_{k+1} = S_k + T_{k+1}.$$

These two statements need some explanation. We have the two successive sums

$$S_k = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots + (-1)^k \frac{x^{2k+1}}{(2k+1)!} \quad (1)$$

$$S_{k+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots + (-1)^k \frac{x^{2k+1}}{(2k+1)!} + (-1)^{k+1} \frac{x^{2(k+1)+1}}{(2(k+1)+1)!}. \quad (2)$$

It is obvious where the recursive statement  $S_{k+1} = S_k + T_{k+1}$  comes from. At this point the naïve or impatient would stop and write

$$S_{k+1} = S_k + T_{k+1} = S_{k+1} = S_k + (-1)^{k+1} \frac{x^{2(k+1)+1}}{(2(k+1)+1)!},$$

and implement it in MATLAB as `sum = sum + sign*x^(2*k+3)/factorial(2*k+3)`

This statement requires  $(2k + 2)$  mults for the numerator,  $(2k + 2)$  mults for the denominator and 1 div. Thus we have a total of  $\sum_{k=1}^n (4k + 5) \text{ ops} = 4(n + 1)n/2 + 5n = 2n^2 + 7n = O(n^2)$  ops for the sum of  $n$  terms.

The  $2n^2$  term should make us suspicious : summing  $n$  terms rarely requires more than  $O(n)$  ops. How can we reduce the number of ops in the  $T_{k+1}$  term? — we need to get a recursive update form for it. Notice that we were forced to write  $S_{k+1}$  in such a form. How else would you write in MATLAB the sum of  $n$  terms when you do not know a specific value for  $n$ ?

We need to write  $T_{k+1}$  in a recursive update form such as  $T_{k+1} = T_k + t_k$  or  $T_{k+1} = t_k T_k$  or some other form. It is obvious we need the second form. We have

$$T_k = (-1)^k \frac{x^{2k+1}}{(2k+1)!} \quad \text{and} \quad T_{k+1} = (-1)^{k+1} \frac{x^{2(k+1)+1}}{(2(k+1)+1)!}.$$

Let us take each part of these expressions in turn : sign, numerator, and denominator.

- Sign.  $(-1)^{k+1} = (-1)^k (-1)^1 = -(-1)^k$ , i.e.,  $\text{Sign}_{k+1} = -\text{Sign}_k$ .
- Numerator.  $x^{2(k+1)+1} = x^{2k+3} = x^2 x^{2k+1}$ , i.e.,  $\text{Num}_{k+1} = x^2 \text{Num}_k$ .
- Denominator.

$$\begin{aligned} (2(k+1)+1)! = (2k+3)! &= (2k+3)(2k+2)(2k+1)!, \quad \text{i.e.,} \\ \text{Den}_{k+1} &= (2k+3)(2k+2)\text{Den}_k. \end{aligned}$$

Putting the three parts together we get

$$T_{k+1} = t_k T_k = -\frac{x^2}{(2k+3)(2k+2)} T_k.$$

The MATLAB statement `term = - (x^2/((2*k+3)*(2*k+2)))*term` requires a total of 5 mults + 2 adds + 1 div = 8ops per term. Thus we have a total of  $\sum_{k=1}^n 8 \text{ ops} = 8n \text{ ops}$  for the sum of  $n$  terms.

Some implemented this in the naïve  $O(n^2)$  way. Apart from being wasteful, this implementation (translation of mathematics into computational mathematics) is *overflow-prone*. For example, in MATLAB we have  $10^{170}/170! \approx 1.378 \times 10^{-137}$  while  $10^{171}/171! = 0$ ; it should be  $\approx 8.0579 \times 10^{-139}$  which is larger than the smallest f.p. number by a factor of  $10^{169}$ . MATLAB's `factorial(171)` overflows and gives the answer `inf`. Because it correctly implements IEEE arithmetic, it sets  $10^{171}/171! = 10^{171}/\infty = 0$ . If you type `>help factorial` in MATLAB you will see that it is accurate for  $n < 21$ . Although the magnitude will be correct, only the first 15 digits will be accurate because double precision has about 15 decimal digits accuracy.

### 5.3 Testing

Assuming that MATLAB's `sin(x)` function is exact to machine precision, the relative error was calculated for each test pair  $(x, n)$  :

$$E_{\text{rel}} = \frac{|\text{MySin}(x,n) - \sin(x)|}{|\sin(x)|}.$$

The results are shown in Table 1.

Table 1: Relative Error MySin1(x)

$n$	10	100	1000
-10000	6.4923e+084	NaN	NaN
-100	4.3609e+020	8.11e+022	8.0747e+022
-10	2.225	2.1629	2.1629
-1	0.13037	0.13037	0.13037
0	0	0	0
1	0.13037	0.13037	0.13037
10	2.225	2.1629	2.1629
100	4.3609e+020	8.11e+022	8.0747e+022
10000	6.4923e+084	NaN	NaN

### 5.4 Analysis

The results in Table 1 show that the series expansion of  $\sin(x)$  is good around  $x = 0$  only. Outside a narrow range about 0, the expansion, although mathematically exact, is computationally useless. Once more, this shows the difference between ideal mathematics and computational mathematics.

Why is the evaluation of  $\sin(x)$  so inaccurate outside the narrow band around 0? One of the reasons is that  $\sin(x)$  is *ill-conditioned* at points outside this range. From the Class Notes, Chapter 2, we have the following definition :

$$\text{cond}(f(x)) = \frac{|f(x + \delta x) - f(x)|/|f(x)|}{|\delta x/x|} = \frac{|x|}{|f(x)|} \frac{|f(x + \delta x) - f(x)|}{|\delta x|} \approx \frac{|xf'(x)|}{|f(x)|},$$

for small  $\delta x$ .

Applying the definition above to the problem of evaluating  $\sin(x)$  we have

$$\text{cond}(\sin(x)) = \frac{|x \cos(x)|}{|\sin(x)|} = \frac{|x|}{|\tan(x)|}$$

It is obvious that the problem becomes more ill-conditioned as  $|x|$  increases. Worse still, as we can see in Figure 1, the condition goes to  $\infty$  at  $x = \pm k\pi, k = 1, 2, \dots$  because of  $1/|\tan(x)|$ .

If we could restrict the range of  $x$  to  $[-\pi/2, \pi/2]$  then the problem of evaluating  $\sin(x)$  would not be ill-conditioned.

### 5.5 Argument Reduction

If we look at the graph of  $\sin(x)$  and  $\cos(x)$  we notice that they have the following properties :

1.  $\sin(x)$  and  $\cos(x)$  are *periodic* with period  $2\pi$ , i.e.,  $\sin(x) = \sin(x + 2k\pi), k = \pm 1, \pm 2, \dots$
2.  $\sin(x)$  is *odd*, i.e.,  $\sin(-x) = -\sin(x)$ , and  $\cos(x)$  is *even*, i.e.,  $\cos(-x) = \cos(x)$
3.  $\sin(x) = \cos(x - \pi/2) = \cos(\pi/2 - x)$  and  $\cos(x) = \sin(x + \pi/2) = \sin(\pi/2 - x)$

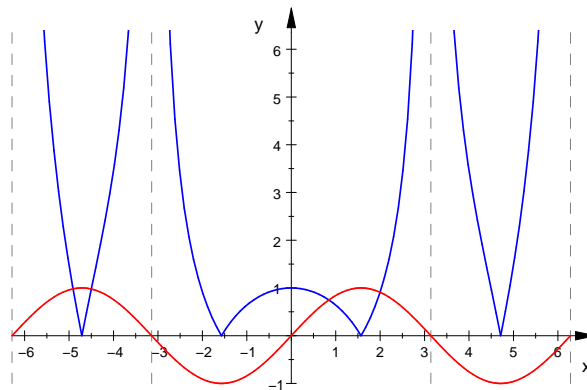


Figure 1: Condition of  $\sin(x)$ .

These properties allow us to reduce the range of  $x$  :

$$\text{Let } x = N \left( \frac{\pi}{2} \right) + r,$$

where  $N$  is the closest integer to  $x(2/\pi)$  and  $|r| \leq \pi/4$  is the *reduced argument*. We then get the value of  $\sin(x)$  by computing either  $\sin(r)$ ,  $-\sin(r)$ ,  $\cos(r)$ , or  $-\cos(r)$ , depending on the value of  $N \bmod 4$ . We calculate  $r$  with the MATLAB statements<sup>1</sup>

$$N = \text{round}((2/\text{pi})*x) \text{ and } r = x - N*(\text{pi}/2)$$

Let us look at some values of  $x$ ,  $r$ ,  $N$ , etc., to see how this works.<sup>2</sup>

Table 2: Range Reduction

$x$	$(2/\text{pi}) * x$	$N$	$\text{mod}(N,4)$	$r$	$\sin(r)$	$\cos(r)$	$\sin(x)$	$\cos(x)$
-10.0000	-6.3662	-6	2	-0.5752	-5.4402e-1	8.3907e-1	5.4402e-1	-8.3907e-1
-7.7778	-4.9515	-5	3	0.0762	7.6130e-2	9.9710e-1	-9.9710e-1	7.6130e-2
-5.5556	-3.5368	-4	0	0.7276	6.6510e-1	7.4675e-1	6.6510e-1	7.4675e-1
-3.3333	-2.1221	-2	2	-0.1917	-1.9057e-1	9.8167e-1	1.9057e-1	-9.8167e-1
-1.1111	-0.7073	-1	3	0.4596	4.4367e-1	8.9619e-1	-8.9619e-1	4.4367e-1
1.1111	0.7073	1	1	-0.4596	-4.4367e-1	8.9619e-1	8.9619e-1	4.4367e-1
3.3333	2.1221	2	2	0.1917	1.9057e-1	9.8167e-1	-1.9057e-1	-9.8167e-1
5.5556	3.5368	4	0	-0.7276	-6.6510e-1	7.4675e-1	-6.6510e-1	7.4675e-1
7.7778	4.9515	5	1	-0.0762	-7.6130e-2	9.9710e-1	9.9710e-1	7.6130e-2
10.0000	6.3662	6	2	0.5752	5.4402e-1	8.3907e-1	-5.4402e-1	-8.3907e-1

First of all, notice that  $|r| \leq \pi/4 = 0.785398163$ . Looking at the values of  $N \bmod 4$  and  $\sin$  and  $\cos$  we see that the following rule holds :

<sup>1</sup>In what follows it is important to distinguish between  $\pi$ , the transcendental number, and  $\text{pi}$ , the computer approximation to  $\pi$ .

<sup>2</sup>We are using MATLAB'S built-in  $\sin$  and  $\cos$  functions here.

```

Calculate both  $\sin(r)$  and  $\cos(r)$ . Then
if  $N \bmod 4 = 0$  then return  $+\sin(r)$ 
if  $N \bmod 4 = 1$  then return  $+\cos(r)$ 
if  $N \bmod 4 = 2$  then return  $-\sin(r)$ 
if  $N \bmod 4 = 3$  then return  $-\cos(r)$ 
    
```

The implementation of these ideas is shown in the function MySin2(x) below.

```

%=====
function sum = MySin2(x)
r = x;
N = 0;
%---- Range reduction x -> r in [-pi/4,pi/4] ----
if abs(x) > pi/4 % reduce range if |x| > pi/4
    N = round(x*(2/pi));
    r = x-N*(pi/2);
end;
ssum = r;  sterm = r;
csum = 1;  cterm = 1;
sum = 0;
%---- Evaluation of sin(r) and cos(r) -----
k= 1;
while (ssum+sterm) ~= ssum
    sterm = - r*r/((2*k+1)*(2*k))*sterm;
    ssum = ssum + sterm;
    cterm = - r*r/((2*k)*(2*k-1))*cterm;
    csum = csum + cterm;
    k = k + 1;
end;
%----- Adjustment -----
if mod(N,4) == 0
    sum = ssum;
elseif mod(N,4) == 1
    sum = csum;
elseif mod(N,4) == 2
    sum = -ssum;
else
    sum = -csum;
end;
    
```

Notice that this code does not use a fixed number of terms  $n$ . Instead, it iterates while the following condition is true

```
while (ssum+sterm) ~= ssum
```

which gives a result to full precision. [WHY?]

Table 3: Relative Error MySin2(x)

x	-10000	-100	-10	-1	0	1	10	100	10000
MySin2(x)	6.8155e-011	6.5776e-015	4.0815e-016	0	0	0	4.0815e-016	6.5776e-015	6.8155e-011

## 5.6 Mini-Max Polynomial Approximation

The Taylor-Maclaurin series for  $\sin(x)$  converges for all  $x$ , but computationally it is not very useful. It is both inaccurate and inefficient. There are many other polynomials that we could use. Those that *minimize the maximum deviation* of the polynomial from the original function are often used in function evaluation.

Let  $f(x)$  be continuous on  $[a, b]$ . We want a polynomial  $p_n(x)$  of degree  $n$  or less that minimizes  $\max |f(x) - p(x)|$  on the interval  $[a, b]$ , or

$$\min_{p_n(x)} \max_{x \in [a, b]} |f(x) - p(x)|.$$

For example, the minimax 3rd degree polynomial for  $e^x$  on  $[-1, 1]$  is

$$p_3^*(x) = 0.994579 + 0.995668x + 0.542973x^2 + 0.179533x^3,$$

which is close to the Taylor-Maclaurin 3rd degree polynomial  $1 + x + x^2/2 + x^3/6$ . Despite the closeness of the coefficients, the behavior of the two polynomials is quite different, as shown in Figure 2.

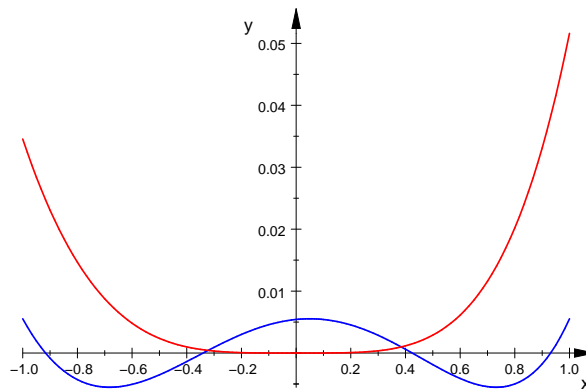


Figure 2: Errors of 3rd Degree Minimax (blue) & Taylor Approximations of  $e^x$

In the third version  $\text{MySin3}(x)$  we use both range reduction and minimax polynomials to represent  $\sin(x)$  and  $\cos(x)$  on  $[-\pi/4, \pi/4]$ . The coefficients for these polynomials are from the FDLIBM math library. (Sun Micros)



Table 4: Relative Error MySin2(x) and MySin3(x)

$x$	MySin2(x)	MySin3(x)
-9.42477796076938	1	1
-7.33038285837618	0	1.28197512425571e-016
-5.23598775598299	2.56395024851142e-016	1.28197512425571e-016
-3.14159265358979	1	1
-1.0471975511966	1.28197512425571e-016	0
1.0471975511966	0	0
3.14159265358979	1	1
5.23598775598299	1.28197512425571e-016	1.28197512425571e-016
7.33038285837618	1.28197512425571e-016	1.28197512425571e-016
9.42477796076938	1	1

There is nothing wrong with MATLAB's  $\sin(\pi)$  because MATLAB's  $\pi = 3.14159265358979 \neq \pi$  and so MATLAB's  $\sin(\pi)$  cannot possibly equal 0. It does indicate that MySin2( $\pi$ ) and MySin3( $\pi$ ) are not accurate. Read Cleve Moler's essay below for an explanation of this 'anomaly'.

Further testing of MySin2(x) and MySin3(x) show that they are completely wrong for  $|x| > 10^{15}$ , whereas MATLAB's  $\sin(x)$  goes wrong for  $|x| > 10^{22}$ . To see how difficult it is to get an accurate implementation of  $\sin(x)$  for large  $x$ , read Ng's article at

<http://www.derekroconnor.net/Software/Ng--ArgReduction.pdf>

### 5.6.1 Timing Comparisons

A simple timing test was run to compare MySin2( $\pi$ ) and MySin3( $\pi$ ) with MATLAB's  $\sin(x)$  function. A random vector  $x$  of length  $N_{\text{samp}}$  was generated with each  $x_i \in [-2\pi, 2\pi]$  and the time taken for each function to calculate  $\sin(x)$  of the  $n$  elements of  $x$  was recorded.

Table 5: Timings of MySin2(x), MySin3(x) and  $\sin(x)$

$N_{\text{samp}}$	$T_2$ secs	No. per sec	$T_3$ secs	No. per sec	$T_m$ secs	No. per sec
$10^6$	34	29,610	38	26,359	0.4	2,557,200

These timings show that MATLAB's  $\sin(x)$  is 100 times faster than our implementations. This is further proof that numerical software writing, at least for critical functions, is best left to the experts.

The code for MySin(x), which is complicated, may seem to be too elaborate for such a simple function. Nonetheless, code very similar to MySin(x) is implemented 'in silicon' on Intel's processors. This was one of the reasons for doing this exercise : to see what goes on inside a standard processor.

Cleve's Corner

**The Tetragamma Function and Numerical Craftsmanship**

MATLAB's special mathematical functions rely on skills from another era.

We've just added the code for the *tetragamma* function to MATLAB; it will be in the next release. I suspect that most of you have never heard of the tetragamma function. I hadn't, until my colleagues working on the Statistics Toolbox needed the *digamma* function to compute maximum likelihood fits to binomial probability distributions.

You may be familiar with the gamma function. It's defined by an infinite integral, but it is known best as the continuous function that interpolates the integer factorial function  $\Gamma(n) = (n-1)!$

The digamma function, which is also called the *psi* function, is the logarithmic derivative of the gamma function  $\Psi(x) = d[\ln\Gamma(x)]/dx = \Gamma'(x)/\Gamma(x)$ .

If you take higher order derivatives, you get functions named trigamma, tetragamma, pentagramma, and so on.

All of this is explained in chapter 6 of A&S, the definitive reference for these matters, *Handbook of Mathematical Functions*, by Milton Abramowitz and Irene Stegun. A&S was first published by the National Bureau of Standards in 1964. A Web- and CD-based successor to A&S, the Digital Library of Mathematical Functions, is under development at the National Institute for Standards and Technology, see <http://dlmf.nist.gov/>.

Much of A&S is devoted to tables that we don't need anymore, except to check the results computed by MATLAB and other mathematical software. But the algorithms we use in modern software for special mathematical functions are derived from techniques used in hand computation over 50 years ago. There are too few people today who know how to write robust software for computing these functions.

A predecessor to Abramowitz and Stegun is an amazing book by Jahnke and Emde, *Tables of Higher Functions*, published in Germany in 1909. (Reprinted by Dover in 1945 and by McGraw Hill in 1960.) Jahnke and Emde, and their students, computed tables of dozens of different functions. They also created, again by hand, intricate 2- and 3-dimensional graphs showing the behavior of these functions in the complex plane. I've included two of my favorite graphs here. One shows the complex valued gamma function. The height of the surface is the modulus, or absolute value, and the contour lines are the modulus and phase. Since  $\Gamma(n) = \Gamma(n+1)/n$ , you can see that this function has poles at the negative integers. The other graph shows the *Hankel* function, or complex valued Bessel function. I've made MATLAB versions of these graphs and added color and lighting, but it's not so easy to recreate the labeling and other nuances of the hand-drawn work.

To appreciate the subtleties involved in the computation of special functions, consider a more basic and familiar function,  $\sin(x)$ . You might think you could use the power series

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

After all, this series converges for all  $x$ . But it turns out to be both inefficient and inaccurate. Here is a toy MATLAB function implementing the power series approach.

```
function s = sin(x)
    s = 0;
    t = x;
    n = 1;
    while s+t ~= s;
        s = s + t;
        t = -x^2/((n+1)*(n+2)).*t;
        n = n + 2;
    end
```

Note that there is no  $x^n$  or factorial(n). Each term  $t$  is computed by multiplying the previous term by the appropriate factor. The number of terms required is determined by roundoff error. Eventually,  $t$  becomes so small that it is numerically negligible compared to the accumulated sums.

This function works pretty well if  $x$  is small enough. Take  $x = \pi/4 = .785398\dots$  In this case, the series starts with

$$0.785398\dots - 0.080746\dots + 0.002490\dots - 0.000037\dots$$

The first term is the largest. It takes just nine terms to compute  $\sin(x) = .707107\dots$  to full double precision accuracy.

Now try  $x = 7.5\pi = 23.5619\dots$  Since is an odd multiple of  $\pi/2$ ,  $\sin(x)$  should be  $-1$ . But the power series starts out with  $23.5619\dots - 2180.13\dots + 605165.9\dots - 799922\dots$

The terms get too big before they start to get small. The biggest term is  $23/23!$ , which is larger than  $10^9$ . We lose 9 of the 16 digits available with double precision floating point arithmetic. The final computed sum is  $-0.99999997360709$ . If that weren't bad enough, our function takes a long time because it uses 47 terms.

Of course, it gets worse. The power series always converges, but as increases, some of the intermediate terms become so large that the computed sum eventually loses all accuracy. The loss of accuracy results from the magnitude of the terms, not their number, even though that increases as well. Fortunately, we can make use of an important property of  $\sin(x)$ , its periodicity,  $\sin(x + 2\pi) = \sin(x)$ .

So, if  $|x| > \pi$ , subtract an integer multiple of  $2\pi$  to bring it into the range where the power series has acceptable numerical properties. This is called *argument reduction*. We can actually combine periodicity with other trig identities to reduce the argument to the interval  $|x| < \pi/4$ . On the reduced interval,  $\sin(x)$  is approximated by a polynomial of degree 13 with only odd powers of

$$\sin(x) \approx x - c_1x^3 + c_2x^5 + c_6x^{13} = p(x).$$

The six coefficients are close to, but not exactly equal to, the power series coefficients

$$1/3!, 1/5!, \dots, 1/13!$$

They minimize the maximum relative error,  $|(\sin(x) - p(x))/\sin(x)|$ , over the interval. Six terms are enough to make this approximation error less than  $2^{-52}$ , which is the roundoff error involved when all the terms, and the sum, are less than one.

The resulting algorithm for approximating  $\sin(x)$ , as well as other trig functions,  $\exp(x)$  and  $\log(x)$ , is carefully implemented in `fdlibm`, a *Freely Distributable Math Library*, developed at Sun Microsystems by K. C. Ng and others, (see [www.netlib.org/fdlibm](http://www.netlib.org/fdlibm)). We use `fdlibm` in MATLAB for all the machines we support. This ensures uniform behavior, particularly for exceptional conditions.

Intel microprocessor architecture includes instructions for evaluating polynomial approximations to  $\sin(x)$  and  $\cos(x)$  on a reduced interval, but MATLAB doesn't use them because the Microsoft optimizing C compiler does not provide a complete argument reduction.

This brings up a delicate and controversial topic: what is the correct value of the MATLAB expression  $\sin(\pi)$ ?

We all agree that  $\sin(\pi)$  is zero. But the quantity that MATLAB denotes by `pi` is a floating point number that is not exactly equal to the theoretical mathematical quantity denoted by  $\pi$ . It turns out that  $\pi$  is about 1/4 of the way between `pi` and the next larger floating point number, `pi + 2 * eps`. In fact,  $\pi - \text{pi} \approx 1.224610^{-16}$ .

Near  $\pi$ ,  $\sin(x)$  is very nearly linear, with a slope of  $-1$ . So  $\sin(\text{pi})$  should be about  $1.2246e - 16$ . The exact value reveals information about the value of  $\pi$  used internally by the math library for argument reduction. Many people expect  $\sin(\text{pi})$  to be zero, but then we have to worry about  $\sin(k * \text{pi})$  for increasingly large integer  $k$ .

This approach to computing  $\sin(x)$ , except for the details about , carries over to other mathematical functions. The general outline is:

- Transform the argument to one or more intervals where the function can be accurately approximated.
- Evaluate a polynomial or rational approximation for the transformed argument.

- If necessary, adjust the result to account for the initial argument transformation.

The argument reduction formulae for the log and exponential functions allow the reduction to an interval near  $x = 1$ .

$$\begin{aligned}\exp(kx) &= (\exp(x))^k. \\ \log(2^k x) &= k \log(2) + \log(x).\end{aligned}$$

The algorithm for the real gamma function shows the complexity of the more specialized algorithms.

1. If  $0 \leq x < \text{eps}$ ,  $\Gamma(x) \approx 1/x$ .
2. If  $\text{eps} \leq x < 1$ , use  $\Gamma(x) = \Gamma(x+1)/x$ .
3. If  $1 \leq x < 2$ , compute a rational approximation.
4. If  $2 \leq x < 12$ , use  $\Gamma(x+1) = x\Gamma(x)$  several times.
5. If  $12 \leq x$ , use an approximation related to Sterling's asymptotic series,

$$\log(\Gamma(x)) \approx (x - \frac{1}{2}) \log(x) - x + \frac{1}{2} \log(2\pi) + \frac{c_1}{x} + \frac{c_2}{x^2} + \dots$$

The decisions that go into these algorithm designs—the choice of reduction formulae and interval, the nature and derivation of the approximations—involve skills that few mathematicians have mastered. The algorithms that MATLAB uses for gamma functions, Bessel functions, error functions, Airy functions, and the like are based on Fortran codes written 20 or 30 years ago by W. J. Cody at Argonne National Labs and Don Amos of Sandia National Labs. (You can see code in the MATLAB directory `.\toolbox\matlab\specfun`.) Cody and Amos are now retired. I hope we have people around today to fill their shoes.

©1994-2004 The MathWorks, Inc. - Trademarks - Privacy Policy

## 6 COMMENTS

Most of you did a good job programming in MATLAB and the L<sup>A</sup>T<sub>E</sub>X ed reports were fairly good also. Some of you attempted argument reduction, which is a good idea, as we have seen above. However, this defeated one of the purposes of this exercise which was to see how bad a Taylor series can be when used to evaluate a function.

Those who used argument reduction did not explain why it is important, given that the Taylor series for  $\sin(x)$  converges for all  $x$ . The primary purpose of argument reduction is not to speed up convergence or increase efficiency. It is to confine the evaluation of  $\sin(x)$  to a range where its condition is small and so reduce the error. See Figure 1.

### 6.1 Programming Notes

1. Some had too many comments. These were often just stating the obvious, e.g.,  
`k = 1:n % k gets all values from 1 to n. Thanks! I'd never have guessed that.`
2. Generally, use space around the operators `= + -` to enhance readability.
3. If variable names are chosen carefully then programs become self-documenting, and very few comments are needed
4. Many professional software writers strive to keep each function or procedure within one page. This is a good rule because it allows the reader (and the writer) to see everything at once.
5. If you find that you have written a complicated piece of code to do a simple task then : *Stop. Think. Re-write.*