Programming assignment #3: Machine Precision and Roundoff Errors

Date assigned: February 15, 2013
Date due: February 22, 2013 (before class)

Lab # 3

In the memo, document your results and discussions and include all necessary tables and source codes that demonstrate your work.

1. Introduction

In the lecture we studied how floating numbers are represented in the computer and how roundoff error can occur. Consider the following subtractions:

123456789.12345678 - 123456789.12345677

and

0.12345678 - 0.12345677

The results of both calculations should be 1E-8. However, try it in your MATLAB, you can see the first calculation give you 0 while the second subtraction gives the desired answer. Thus, you see that when you subtracting two large numbers you results won’t be accurate. In this lab experiment, we want to study how it would lead to disasters if your algorithm does not consider this.

Consider the following Taylor series expansion to evaluate the exponential function:

\[ e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \]

The following MATLAB code is an implementation of this technique:

```matlab
function y=exp1(x)
% Evaluate exp(x) using the following expression
% exp(x)=1+x+x^2/2!+x^3/3! + ...
%
% this calculation is done in single precision.
% y=1.0; % variable to save the sum
yold=0; % sum from the previous iteration
term=single(1.0); % set the term variable as single precision number
i=1; % iteration counter

% loop until the sum does not change any more
while y~=yold
```

```matlab
    y=y+term;
    yold=y;
    term=term*x/(i+1);
    i=i+1;
end
```

```matlab
end
```
term=term*x/i;  % in MATLAB a double precision number and a single
% precision number results a single precision number
yold=y;         % save the sum
y=y+term;       % update the sum by adding one more term
i=i+1;          % increase the iteration counter
fprintf('term #%d = %e, sum = %e
',i,term,y);
end
end

While we mentioned that MATLAB by default uses IEEE double precision (i.e., using 64-bit for a float number) in its internal calculations, we can change the variables to be single precision (i.e., using 32-bit for a float number). In the above code, the variable ‘term’, which represents the next term in the Taylor series, is set to be single precision, and thus causing all the related variables to be single.

2. Problems
   a. Implement the above code in MATLAB and calculate the $e^{15}$ and $e^{-15}$ using the above function. Compare the results with the ones from the MATLAB build-in function ‘exp’.

   \[
   \begin{align*}
   \exp(-15) &= 3.0590e-007 \\
   \exp(15) &= 3.2690e+006 \\
   \exp1(-15) &= 0.021234 \\
   \exp1(15) &= 3.2690e+006
   \end{align*}
   \]

   b. Change the line ‘term=single(1.0);’ to ‘term=1.0;’ in the code so that that calculation is done in double precision. Recalculate $e^{15}$ and $e^{-15}$, and compare with the single precision results and the MATLAB build-in function results. Discuss what you notice.

   \[
   \begin{align*}
   \exp1(-15) &= 3.0591e-007 \\
   \exp1(15) &= 3.2690e+006
   \end{align*}
   \]

   Discussion: double precision version of the code gives results very similar to MATLAB build-in function.

   c. Evaluate $e^{-10}, e^{-11}, \ldots, e^{-25}$ using the double precision version of the above code, and compare the results with the MATLAB build-in exp function by calculating the relative error. Make a table that summarizes the results and discuss what you notice.

   ```matlab
   lab3.m
   for i=-10:-1:-25
       y1=exp1(i);
       y2=exp(i);
       fprintf('%d, %e, %e, %5.2f%%
', i, y1, y2, abs((y2-y1)/y2)*100);
   end
   octave-3.2.4.exe:28> lab3a
   -10, 4.539993e-005, 4.539993e-005, 0.00%
   ```
Discussion: The double precision version of the function \texttt{exp1} gives very accurate results until \(x = -18\). The error become significant for \(x = -19\) and beyond.

\textbf{d.} Use the single precision program shown above again to calculate \(\text{exp}(-10)\) and inspect the output that shows the value of each term in the series and the sum. Explain why the results are off significantly comparing to the true value.

Results from single precision version:

\texttt{term #2 = -1.000000e+001, sum = -9.000000e+000}
\texttt{term #3 = 5.000000e+001, sum = 4.100000e+001}
\texttt{term #4 = -1.666667e+002, sum = -1.256667e+002}
\texttt{term #5 = 4.166667e+002, sum = 2.910000e+002}
\texttt{term #6 = -8.333334e+002, sum = -5.423334e+002}
\texttt{term #7 = 1.388889e+003, sum = 8.465557e+002}
\texttt{term #8 = -1.984127e+003, sum = -1.137572e+003}
\texttt{term #9 = 2.480159e+003, sum = 1.342587e+003}
\texttt{term #10 = -2.755732e+003, sum = -1.413145e+003}
\texttt{term #11 = 2.755732e+003, sum = 1.342587e+003}
\texttt{term #12 = -2.505211e+003, sum = -1.162624e+003}
\texttt{term #13 = 2.087676e+003, sum = 9.250522e+002}
\texttt{term #14 = -1.605905e+003, sum = -6.808523e+002}
\texttt{term #15 = 1.147075e+003, sum = 4.662223e+002}
\texttt{term #16 = -7.647164e+002, sum = -2.984941e+002}
\texttt{term #17 = 4.779478e+002, sum = 1.794536e+002}
\texttt{term #18 = -2.811458e+002, sum = -1.016921e+002}
\texttt{term #19 = 1.561921e+002, sum = 5.449994e+001}
\texttt{term #20 = -8.220636e+001, sum = -2.770642e+001}
\texttt{term #21 = 4.110318e+001, sum = 1.339676e+001}
\texttt{term #22 = -1.957294e+001, sum = -6.176184e+000}
\texttt{term #23 = 8.896792e+000, sum = 2.720609e+000}
\texttt{term #24 = -3.868171e+000, sum = -1.147562e+000}
\texttt{term #25 = 1.611738e+000, sum = 4.641758e-001}
\texttt{term #26 = -6.446951e-001, sum = -1.805193e-001}
\texttt{term #27 = 2.479596e-001, sum = 6.744036e-002}
\texttt{term #28 = -9.183690e-002, sum = -2.439654e-002}
\texttt{term #29 = 3.279889e-002, sum = 8.402355e-003}
Discussion: In single precision representation of the float numbers, the numbers are accurate to at most seven decimal digits (This can be tested by assign a=single(123456789.123456789) and look at what a is). The true result of $\exp(-10)$ is $4.54e-5$, or 0.0000454. Some of the terms are much larger than this number, for example, term 11 is 2755.732. The algorithm used in this example leads to alternative positive and negative terms and adding a large positive number to a large negative number leads to significant round off errors. Still use term 11 as our example: Since the sum of terms 1 to 10 is -1413.144, thus the sum from 1 to 11 terms is only accurate to 3 decimal places after the decimal point, or 0.001, which is much larger than the final result of 0.0000454. This type of round off error accumulates and leads to significant error in the final result.

e. Improve the example single precision code so that it gives reasonable for both positive and negative x. (Hint: recall the relationship: $e^x=1/e^{-x}$). Demonstrate that it now works for $\exp(-10)$ by comparing your results with the results from the MATLAB build-in function.

ex1_mod.m:
function y=ex1_mod(x)
% Evaluate $\exp(x)$ using the following expression
% $\exp(x)=1+x+x^2/2!+x^3/3! + ...$
% This calculation is done in single precision. For $x<0$, $\exp(x)=1/\exp(-x)$ is used.
% y=1.0; % variable to save the sum
yold=0; % sum from the previous iteration
term=single(1.0); % set the term variable as single precision number
i=1; % iteration counter
% check if x is greater than zero
if x<0
    isign=-1;
    x=x*(-1);
else
    isign=1;
end

% loop until the sum does not change any more
while y~=yold
    term=term*x/i;  % in MATLAB a double precision number and a single
                   % precision number results a single precision number
    yold=y;         % save the sum
    y=y+term;       % update the sum by adding one more term
    i=i+1;          % increase the iteration counter
    % fprintf('term %d = %e, sum = %e\n',i,term,y);
end

% calculate 1/exp(x) if x is negative originally
if isign==-1
    y=1/y;
end

end

octave-3.2.4.exe:50> exp(-10)
an = 4.5400e-005
octave-3.2.4.exe:51> exp1_mod(-10)
an = 4.5400e-005
octave-3.2.4.exe:52>

3. Challenge Problem

Write a MATLAB function that evaluates the error function using the following equation:

\[
\text{erf}(x) = \frac{2}{\sqrt\pi} \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{n!(2n+1)}
\]

Include as many terms as necessary so that the relative change in the results by adding another term is less than 0.001%. Use single precision in the calculation and don’t forget the constant coefficient \(2/\sqrt\pi\). Evaluate \(\text{erf}(0.5), \text{erf}(1.0), \text{erf}(5)\) and \(\text{erf}(10)\) and compare the results with the results from the MATLAB build-in function ‘erf’.

erf1.m:

function [y,i]=erf1(x)
% evaluate the error function based on the following series:
% erf(x)=2/sqrt(pi)*sum((-1)^n*x^(2n+1)/(n!(2n+1)))
term=single(x);     % set the term variable as single precision number
y=term;     % variable to save the sum
i=1;        % iteration counter
ea=100;    % relative error
yold=0;     % sum from the previous iteration

while ea>0.001
    yold=y;         % save the sum
    term=term*x*x*(-1)*(2*i-1)/(i*(2*i+1));
    y=y+term;       % update the sum by adding one more term
    ea=abs((y-yold)/y)*100;
    i=i+1;          % increase the iteration counter
end
y=y*2/sqrt(pi);

lab3c.m:

```matlab
for i=[0.5 1.0 5 10]
    fprintf('erf1(%3.1f)=%f, erf(%3.1f)=%f, relative error=%5.2f%%\n',... 
        i,erf1(i), i,erf(i), abs((erf1(i)-erf(i))/erf(i))*100);
end
```

octave-3.2.4.exe:72> lab3c
erf1(0.5)=0.520500, erf(0.5)=0.520500, relative error= 0.00%
erf1(1.0)=0.842701, erf(1.0)=0.842701, relative error= 0.00%
erf1(2.0)=0.995322, erf(2.0)=0.995322, relative error= 0.00%
erf1(5.0)=-89.190865, erf(5.0)=1.000000, relative error=9019.09%
erf1(10.0)=Inf, erf(10.0)=1.000000, relative error= Inf%
```