

Applied Engineering Problem Solving (CHEN.3170)

Part II: Numerical Methods and Applications

Lesson 4: Numerical Errors

The first three lessons in this course focused on developing your skills with using Matlab as a problem-solving tool. Most of the applications to date have centered on evaluation, plotting, and analysis of analytical functions or tabulated data. In real engineering analysis, however, we are often faced with a variety of additional tasks that may involve finding the roots of high order polynomials and transcendental equations, solving a system of linear or nonlinear equations, performing a statistical analysis of experimental data, or applying various curve fitting techniques to help understand the data. In addition, the need to evaluate derivatives and integrals of functions described via analytical expressions or numerical data also occurs quite frequently. And, of course, the solution of a variety of differential equations is often required for the design and analysis of real systems and processes! Over many years, a series of numerical procedures have been developed to address each of these application areas and, as you might expect, Matlab has several built-in functions for handling many of these tasks. Our goal, for the remainder of the semester, will be to highlight the basic ideas behind the various numerical algorithms used to solve these types of problems and to apply these concepts, with help from Matlab, to a variety of practical applications. Thus, the second part of this course will focus on a variety of applied numerical methods within Matlab as an essential component of a flexible toolbox for engineering problem solving.

However, before addressing each of the subject areas noted above, we need to discuss several basic concepts associated with numerical computations within the computer and the inherent errors that are associated with all numerical methods. The subject of numerical errors is discussed in some detail in Chapter 4 of your numerical methods text by Chapra. While reading Chapter 4, you should pay particular attention to the following subjects:

- Computer representation of numbers (just the basics),
- Round off error and machine precision,
- The implication of round off error in the convergence of iterative schemes,
- The Taylor series expansion and the truncation error associated with the finite approximation to infinite series, and
- The tradeoff associated with round off and truncation errors.

These subjects will also be briefly discussed in these notes, but I don't really want to get caught up in too much detail. The real goal of this lesson on numerical error is stated quite nicely with the following quote from a text by Recktenwald (pg. 188 of *Numerical Methods with Matlab*, Prentice Hall, 2000):

Perhaps it is odd to some readers to start out a discussion of numerical methods with such a depressing topic as unavoidable errors. Why should we begin with a consideration of what goes wrong before even presenting any useful algorithms? This decision is more significant than choosing between good news or bad news first. By examining the source of unavoidable numerical errors, the foundation is established for many algorithmic decisions taken to minimize the impact of these errors. It is difficult to appreciate the design of good algorithms without a fundamental understanding of the errors that are inherent in numerical calculations.

Thus, the purpose here is to make you aware of some underlying concerns that can cause problems when applying numerical methods to practical applications. As engineers and scientists, we are primarily interested in applying the methods to solve real problems, not in a detailed study of their effectiveness and robustness on certain classes of problems. However, although the details usually can be left to the mathematicians and numerical analysis experts, as users, we always need to be aware of the potential for problems because of unavoidable errors in the numerical computations. Thus, the purpose of this lesson is to identify the types of problems that can occur and to caution you to always keep a watchful eye for erroneous results due to unavoidable numerical errors.

The two types of errors we want to discuss are *round off error* and *truncation error*. Round off error is due to the fact that computers can only represent quantities with a finite number of digits -- that is, floating point arithmetic is not exact on the computer! Truncation error, on the other hand, is associated with the approximations that are usually made when attempting to represent exact mathematical expressions or operations. Since these approximations are usually related to truncating an infinite Taylor series expansion to a finite number of terms, the resulting errors are referred to as truncation errors. This lesson will elaborate a little on each of these types of errors and show how they are inter-related in many practical numerical procedures. And, of course, we are also interested in how to minimize these errors in routine analysis.

We will start by discussing round off error which, as noted above, is related to the representation of numbers within the computer hardware. To illustrate the problem, consider the following sequence of commands in Matlab:

```
>> format long e
>> format compact
>> 5 + 0.125
ans =
    5.1250000000000000e+000
>> ans + 0.125
ans =
    5.2500000000000000e+000
>> ans + 0.125
ans =
    5.3750000000000000e+000
>> 5 + 0.126
ans =
    5.1260000000000000e+000
>> ans + 0.126
ans =
    5.2520000000000001e+000
```

What has happened here? When 0.125 is continually added to the previous answer (the **ans** variable), the result is as expected. However, we only have to add 0.126 twice before an anomalous digit appears in the 15th decimal place. Clearly something is odd here!!! Where did this digit come from? Is this what it should be, or do I have a problem with my computer? Actually, my computer is just fine -- and, unfortunately, the result that was obtained above, although clearly not correct, was not really unexpected!!!

To understand what has happened, let's briefly look at how the computer stores a floating point number. Because transistor elements can be efficiently operated in an on-off manner, the natural way to store numbers is to use a binary or base 2 number system, where the on and off states correspond to the 1 and 0 digits within the base 2 system. We can easily represent a decimal number, $x.y$, by a sequence of binary digits using the expressions

$$x = \sum_{k=0} b_k 2^k \quad \text{and} \quad y = \sum_{k=1} c_k 2^{-k} \quad (1)$$

Thus, $x.y$ could be written as $\dots b_5 b_4 b_3 b_2 b_1 b_0 . c_1 c_2 c_3 \dots$ using binary notation.

It is important to realize that computers can only store a finite number of binary digits, with each digit as a single bit (which is either 0 or 1). Eight bits make a byte and, on most computers after about 2013, 8 bytes are used to store a word (i.e. the numerical value of a variable in a program, for example). In Matlab, the default arithmetic is done with 64-bit precision.

Now, at this point, I am not so interested about the details of how the information is actually stored in the computer (there are many texts that do a good job describing this if you are really interested). My focus, however, is on the fact that **there are a finite number of bits (digits) and this is what leads to the round off error mentioned above** and demonstrated in the brief Matlab sequence. Working with a full 64-bit word to illustrate the problem here is too cumbersome. Thus, we will illustrate the key ideas with a hypothetical computer that is limited to a 16-bit word for storing both the x and y values -- that is, we will write the numbers x and y in the decimal number $x.y$ using eqn. (1), where the summations will contain a maximum of 16 terms. This is done in Table 1 for the cases that include $x = 5$ and $y = 0.125$ and 0.126 .

As shown in Table 1, $x = 5$ is written as 101 in binary [sometimes written as $(101)_2$], and this can be represented exactly in a computer which has 16 bits per word. Similarly, $y = 0.125$ can be written exactly as $(0.001)_2$. However, we see that $y = 0.126$ can only be given approximately as $(0.0010000001000001)_2$. And, because of this approximate representation, any further arithmetic with this value will also give approximate results. Thus, in general, floating point arithmetic with a finite number of digits gives some round off error! This is the error associated with not being able to store all the digits required to exactly represent the numbers of interest in binary form.

Clearly, with a larger word size, more digits can be stored, and the ramifications associated with round off error decrease. Since Matlab is a numerically based computational system, it was decided to use a 64-bit word size for all numerical variables within the code. This decision was made to minimize the effects of round off in all of Matlab's numerical computations.

An important ramification of floating point arithmetic is that two real numbers that are nearly equal cannot be distinguished if their difference is less than the least significant bit in their computer representation (farthest bit to the right in the binary form). For example, if we limited the representation of $y_1 = 0.125$ and $y_2 = 0.126$ to 8 bits, then, on the computer, y_1 and y_2 would be equal -- since, as shown in Table 1, they have the same bit pattern. Fortunately, all computers use greater than 8 bits to represent a word -- and today, most computers use 64 bits!

However, even with 64-bits, this represents a finite precision limit. This limit is characterized by a number called machine epsilon, ϵ_m . It is defined precisely as the smallest floating point number, ϵ_m , such that

$$1 + \epsilon_m > 1 \quad (2)$$

If some number, δ , is less than ϵ_m , then, on the computer, we have

$$x = x + \delta$$

or, an alternate way to say this is

$$y = x - (x + \delta) = 0$$

Thus, since $\delta < \varepsilon_m$, the computer cannot distinguish between the two numbers, x and $x + \delta$.

Table 1 Binary representation of some decimal numbers.

x = 5					
k	2^k	b_k	cumulative sum		
0	1	1	1		
1	2	0	1		
2	4	1	5		
y = 0.125					
k	2^{-k}	c_k	cumulative sum	c_k	cumulative sum
1	0.5	0	0	0	0
2	0.25	0	0	0	0
3	0.125	1	0.125	1	0.125
4	0.0625			0	0.125
5	0.03125			0	0.125
6	0.015625			0	0.125
7	0.0078125			0	0.125
8	0.00390625			0	0.125
9	0.001953125			0	0.125
10	0.0009765625			1	0.1259765625
11	0.00048828125			0	0.1259765625
12	0.000244140625			0	0.1259765625
13	0.0001220703125			0	0.1259765625
14	6.103515625e-005			0	0.1259765625
15	3.0517578125e-005			0	0.1259765625
16	1.52587890625e-005			1	0.125991821289063

We can easily estimate machine epsilon, ϵ_m , on any computer by continually reducing a number (say, by a factor of two) until the condition given in eqn. (2) is no longer valid. For example, the following Matlab script file, **meps.m**,

```
%
% MEPS.M      Estimate Machine Epsilon
%
% Machine epsilon is defined as the smallest floating point number
% such that
%     1 + epsilon > 1
%
% We can estimate this quantity by continually decreasing a number until this
% condition is no longer valid. This is what is done here in this simple
% program.
%
% Note that Matlab has a built-in value of machine epsilon, called eps.
%
% Written by J. R. White, UMass-Lowell (last update: Oct. 2017)
%

clear all, close all

% initialize epsilon and continue to divide by two until condition is not met
epsilon = 1.0;
while epsilon + 1.0 > 1.0
    epsilon = epsilon/2;
end

% multiply estimate by two to be conservative
epsilon = 2*epsilon;
disp('Estimate of machine epsilon is'), disp(epsilon)

% edit Matlab's built-in value
disp(' ')
disp('Matlab''s internal value of machine epsilon is'), disp(eps)

%
% end of problem
```

gives an approximate value of 2.22e-16 as machine epsilon. Actually, since this number is so important, Matlab has a built-in variable, **eps**, that gives the value of ϵ_m for the particular computer architecture on which Matlab is running. From the last part of the above program, **eps** is given as 2.22e-16 (this is the same value we computed in our algorithm!).

Before leaving our brief discussion of the inherent errors associated with finite precision arithmetic, it is important to note that, although proper algorithm design and 64-bit arithmetic tend to minimize round off error, it is always something that you should be aware of when doing numerical computations and code development. For example, as we will see in many of the subsequent numerical methods to be studied, some form of iteration will often be needed to implement a particular procedure. The iteration scheme is stopped when some condition is met -- that is, when some measure of the error in the calculation is small. Because of round off error, we almost never ask the question “Is $x = y$?”. Instead, we ask “Is x close to y ?” and this is often implemented as follows:

```
rerr = 1;
while rerr > tol

    continue iterative calculation

    rerr = abs((x-y)/y);
end
```

where **tol** is some user-defined tolerance for the absolute relative difference between x and y .

As mentioned at the beginning of this lesson, another important type of numerical error that is inherent in most of our computational work is truncation error. However, unlike round off error, which is related to the computer hardware and software, truncation error can be controlled to some extent by the programmer. As you know, for computer implementation, we always need to discretize the quantities of interest -- and we have already seen this in the simple evaluation of a function, $f(x)$:

Continuous Form: Given $f(x)$, plot $f(x)$ vs. x .

Discrete Form: Let $x \rightarrow x_i$ and $f(x) \rightarrow f(x_i) \rightarrow f_i$, then plot f_i vs. x_i .

where the discrete form is what we implement on the computer. Although there are no new errors introduced for simple functional evaluation, **truncation error is often introduced when we approximate continuous mathematical functions and operations with a discrete algebraic representation**. This error is usually associated with the actual truncation of an infinite series expansion for the quantity of interest to a finite number of terms -- thus the term, **truncation error**.

For example, we are all familiar with the simple exponential function, $f_1(x) = e^x$ or $f_2(x) = e^{-x}$, where f_1 grows exponentially without bound for positive x , and f_2 decreases exponentially to zero as $x \rightarrow \infty$. These functions, however, are really infinite series expansions,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots = \sum_{k=1}^{\infty} \frac{x^{k-1}}{(k-1)!} \quad (3)$$

and

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \cdots = \sum_{k=1}^{\infty} \frac{(-x)^{k-1}}{(k-1)!} \quad (4)$$

where the e^x and e^{-x} representations are usually used because they are easier to write and to visualize than the actual series expansions.

One form of truncation error occurs when we simply truncate the infinite series expansion to a finite number of terms (which, of course, we must do in practice). As a short example, consider the computation of e^3 and e^{-3} using eqns. (3) and (4) truncated to 8 terms. In addition, we will only use 5 significant digits in our calculations. Doing the calculations gives:

Case 1:

$$\begin{aligned} e^3 &\approx 1 + 3 + \frac{9}{2} + \frac{27}{6} + \frac{81}{24} + \frac{243}{120} + \frac{729}{720} + \frac{2187}{5040} \\ &= 1 + 3 + 4.5 + 4.5 + 3.375 + 2.025 + 1.0125 + 0.43393 = 19.846 \end{aligned}$$

and my calculator gives $e^3 = 20.086$ -- thus, our 8-term estimate has an error of about -1.2%.

Case 2:

$$e^{-3} \approx 1 - 3 + 4.5 - 4.5 + 3.375 - 2.025 + 1.0125 - 0.43393 = -7.1430e-2$$

and the actual value is $e^{-3} = 4.9787e-2$ -- which shows that our estimate is terrible with about -243% error (we didn't even get the correct sign!!!).

The error for Case 1 is dominated by truncation error, with only a minor loss in accuracy associated with rounding the individual calculations to 5 figures. In this case, the addition of a few more terms in the series would give very accurate results. Case 2, on the other hand, has a serious case of both truncation error and round off error. In particular, notice that some of the individual terms are a factor of 100 larger than the final result. Although additional terms in the series would help considerably, we could never get 5 significant figures of accuracy, because the subtraction of nearly equal terms leads to the loss of significant digits (this is often referred to as **catastrophic cancellation**).

Notice, however, that if we compute the result for e^{-3} using the inverse of the Case 1 result, we have

$$e^{-3} = \frac{1}{e^3} = \frac{1}{19.846} = 5.0388e-2$$

which only represents an error of 1.2%. This is an example of what I mean by “proper algorithm design” as mentioned above. In our example, the computational algorithm given in the Case 2 calculation for e^{-3} is not a well-designed algorithm because of the catastrophic cancellation that occurs. Clearly a better way to compute e^{-x} would use $1/e^x$ to avoid much of the round off error associated with an infinite series expansion whose individual terms alternate in sign. Many times, however, “proper algorithm design” is not so simple, so we will leave much of the hard-core development of various numerical algorithms to the mathematicians and numerical analysis experts. In fact, that is why we will use Matlab to do many of the needed computations, since many years of experience has shown that most of Matlab’s built-in algorithms are quite efficient and robust for a wide range of applications.

The infinite series expansion for e^x and e^{-x} in eqns. (3) and (4) are examples of Taylor series expansions. Knowledge of the Taylor series is absolutely essential for the study of numerical methods and mathematical modeling. In general, it provides a means for predicting the value of a function at some point, $x_{i+1} = x_i + h$, in terms of the value of the function and its derivatives at the point x_i , where h is the step size. In addition, it also allows us to derive formulae for a variety of derivative approximations and to estimate the order of error in many approximations.

The basic formulation for the Taylor series can be written as

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{f''(x_i)}{2!}(x_{i+1} - x_i)^2 + \frac{f'''(x_i)}{3!}(x_{i+1} - x_i)^3 + \dots + \frac{f^{(n)}(x_i)}{n!}(x_{i+1} - x_i)^n + R_n \quad (5)$$

where the remainder, upon truncation after the n^{th} order term, is given by

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x_{i+1} - x_i)^{n+1} \quad \text{with } x_i < \xi < x_{i+1} \quad (6)$$

This notation is a little cumbersome and we will now go about simplifying this somewhat. First, we will simply use f_i to denote $f(x_i)$, $f'_i = f'(x_i)$, etc. and let the step size or interval width be denoted by h , where $h = x_{i+1} - x_i$. In addition, the remainder, R_n , is the truncation error upon dropping all the terms after the n^{th} order term. Since, in general, we don’t know the precise value of ξ (only its range) and we don’t know the $(n+1)^{\text{th}}$ derivative, this term is often written as

$$R_n = \alpha h^{n+1} = O(h^{n+1}) \quad (7)$$

where α is a proportionality constant (which we don't know) and the notation, $O(h^{n+1})$, says that the truncation error is "on the order of h^{n+1} " -- that is, R_n is proportional to h^{n+1} .

Note: In formal error analysis, the independent variable, x , is always normalized over the range $(0, 1)$. Therefore, h is always less than unity -- a small fraction of the total interval. Thus, since $h < 1$, $h^{n+1} < h^n$. For example, let's say that $h = 0.1$, then $h^3 = 0.001$ and $h^2 = 0.01$. Thus, truncation error on the order of h^3 is better than $O(h^2)$, and $O(h^2)$ is much better than $O(h)$, etc.. This implies that reducing h can significantly reduce R_n for a given order of approximation. Therefore, as mentioned above, we have some control over the error associated with truncating a Taylor series expansion to a finite number of terms. In particular, the two methods are:

1. Use a smaller step size, h .
 2. Use additional terms in the series expansion (i.e. a higher order approximation).
-

Now, with these notation changes, eqn. (5) can be written as

$$f_{i+1} = f_i + f_i' h + \frac{f_i'' h^2}{2!} + \frac{f_i''' h^3}{3!} + \dots + \frac{f_i^{(n)} h^n}{n!} + O(h^{n+1}) \quad (8)$$

and this is a more manageable expression for practical application. This is often referred to as a **forward** Taylor series expansion for f_{i+1} about the point x_i .

We can also take a backward step from the point x_i . Since the step size is negative, $\Delta x = x_{i-1} - x_i$, we can simply replace h in eqn. (8) with $-h$, where h is now given by $|\Delta x|$. Doing this gives

$$f_{i-1} = f_i - f_i' h + \frac{f_i'' h^2}{2!} - \frac{f_i''' h^3}{3!} + \dots + (-1)^n \frac{f_i^{(n)} h^n}{n!} + O(h^{n+1}) \quad (9)$$

This is a **backward** Taylor series for $f(x_{i-1})$ about the point x_i .

One of the most important uses of the Taylor series expansions given in eqns. (8) and (9) is the derivation of appropriate finite difference approximations for first-order and second-order derivatives at a point. In particular, we can derive several of the more common discrete derivative approximations as follows:

Forward Approximation to f_i'

Here we use eqn. (8) truncated after the 1st order term,

$$f_{i+1} = f_i + f_i' h + O(h^2)$$

and solve this for f_i' , giving

$$f_i' = \frac{f_{i+1} - f_i}{h} + O(h) \quad (10)$$

where we note that this is said to be a first-order forward approximation to $f'(x)$ at $x = x_i$ because the error in the estimate of f_i' is proportional to h . Note also that the $O(h^2)$ term in the original

equation becomes $O(h)$ upon division by h (and we also ignore the sign of this error term -- since the focus is only on the “order of the error”).

Backward Approximation to f'_i

Performing similar manipulations with eqn. (9) gives

$$f_{i-1} = f_i - f'_i h + O(h^2)$$

or

$$f'_i = \frac{f_i - f_{i-1}}{h} + O(h) \quad (11)$$

and again, we see that this is a first-order estimate using a backward differencing scheme.

Central Approximation to f'_i

This time, we start by subtracting eqn. (9) from eqn. (8), and truncate the series after the 2nd order term (note that the 2nd order terms cancel exactly), giving

$$f_{i+1} - f_{i-1} = 2f'_i h + O(h^3)$$

and solving this for f'_i gives

$$f'_i = \frac{f_{i+1} - f_{i-1}}{2h} + O(h^2) \quad (12)$$

Notice that this approximation has an error that is proportional to h^2 . Thus, this gives a more accurate estimate of f'_i for a given step size, h . However, function information on both sides of x_i is needed, so it cannot be used at the left or right boundaries of a given domain.

Central Approximation to f''_i

We can also determine discrete derivative approximations for higher-order derivatives. For example, the 2nd derivative evaluated at x_i can be obtained by simply adding eqns. (8) and (9). Doing this gives

$$f_{i+1} + f_{i-1} = 2f_i + f''_i h^2 + O(h^4)$$

and solving this for f''_i gives

$$f''_i = \frac{f_{i-1} - 2f_i + f_{i+1}}{h^2} + O(h^2) \quad (13)$$

This is said to be a 2nd order central approximation for $f''(x)$ at x_i , since the error term is proportional to h^2 .

Higher order approximations (better than 1st and 2nd order) can also be derived. However, in practice, these are more cumbersome to develop and use, and one usually applies the simple 1st and 2nd order derivative approximations for f' and f'' given here in practical problems (see examples below). If one needs to reduce the truncation error beyond that obtained with second order accuracy, then the step size is usually decreased as needed.

As a simple example, let's compute the first and second derivatives of e^x at $x = 0$ using the formulas from above. Since all derivatives of e^x are simply e^x , then, at $x = 0$, both the first and second derivatives should have the value of unity (i.e. $e^0 = 1$). A short Matlab program, **deriv_approx.m**, was written to do the actual finite difference calculations and a listing is given in Table 2. This program produces a short table of results for a series of 5 step sizes that start at $h = 0.5$ and decreases by a factor of two for each new value. The summary results are given in Table 3, which highlights two key points from the above discussion:

1. The error decreases as the step size is reduced.
2. The error in the second-order approximation is less than the error in the first-order approximation (compare column 4 with columns 2 and 3).

Also, you should note that, for the 1st order approximations where the error is $O(h)$, we should expect the error to decrease roughly by a factor of two for each new step size -- since each new h is simply one-half of the previous value. And, for the two 2nd order estimates (the last two columns in Table 3), the error should decrease by nearly a factor of 4 each time, since the error is $O(h^2)$, with h decreasing by a factor of 2 each time. As apparent from Table 3, the results obtained here match expected trends perfectly [compare (derivative value – 1) vs. step size]. Isn't it great when mathematical theory really works in practice!!!

Table 2 Listing of the deriv_approx.m program.

```
%
% DERIV_APPROX.M Calculate approximation to the derivatives of exp(x) at x = 0
%                   for different step sizes
%                   (used to generate data for Table 3 in Lesson 4 Lecture Notes)
%
% Written by J. R. White, UMass-Lowell (last update: Oct. 2017)

    clear all,    close all

%
% set different step sizes (decreasing by a factor of two)
%   h = [0.5 0.25 0.125 0.0625 0.03125]';

%
% evaluate fi-1, fi, and fi+1 for each step size (where xi = 0)
%   fm = exp(-h);           % fm = fi-1 = exp(0-h) = exp(-h)
%   f = ones(size(h));      % f = fi = exp(0) = 1
%   fp = exp(h);            % fp = fi+1 = exp(0+h) = exp(h)

%
% now form different derivative approximations
%   fpb = (f - fm)./h;      % backward approx to f'
%   fpf = (fp - f)./h;      % forward approx to f'
%   fpc = (fp - fm)/(2*h);  % central approx to f'
%   fp2c = (fm - 2*f + fp)/(h.*h); % central approx to f''

%
% create short table of results
%   fprintf('      h      fpb      fpf      fpc      fp2c \n')
%   for n = 1:size(h)
%       fprintf(' %8.5f %8.5f %8.5f %8.5f %8.5f \n', ...
%           h(n), fpb(n), fpf(n), fpc(n), fp2c(n));
%   end

%
% end of problem
```

Table 3 Approximate derivatives for e^x at $x = 0$ (from deriv_approx.m).

step size, h	$f_i' = \frac{f_i - f_{i-1}}{h}$	$f_i' = \frac{f_{i+1} - f_i}{h}$	$f_i' = \frac{f_{i+1} - f_{i-1}}{2h}$	$f_i'' = \frac{f_{i-1} - 2f_i + f_{i+1}}{h^2}$
0.50000	0.78694	1.29744	1.04219	1.02101
0.25000	0.88480	1.13610	1.01045	1.00522
0.12500	0.94002	1.06519	1.00261	1.00130
0.06250	0.96939	1.03191	1.00065	1.00033
0.03125	0.98454	1.01579	1.00016	1.00008

Well, the last point I want to make in this lesson is associated with the relationship between round off error and truncation error. Remember that round off error occurs because of the finite number of digits used for storing floating point numbers in the computer. Every time a computation is performed some small error is introduced and the round off error continues to accumulate as more and more computations are performed. Also, as just noted, truncation error is directly related to the step size -- that is, as the step size decreases, the truncation error decreases (usually $\varepsilon = \alpha h^n$, where the effective n is usually between 1 and 2).

However, there is usually a tradeoff that can become important when one continually reduces the step size to minimize the truncation error, since in many applications a small step size implies that many computations will be required, and increased numerical operations leads to greater round off error.

For example, let's say we want to solve the following initial value problem (IVP),

$$\frac{dy}{dx} = e^x \quad \text{with} \quad y(0) = 1$$

Of course, this simple IVP has an exact solution $y(x) = e^x$, but here, we are interested in a numerical solution to demonstrate the tradeoff between round off error and truncation error as described above.

A simple numerical scheme can be developed by evaluating both sides of the differential equation at discrete point x_i , or

$$\left. \frac{dy}{dx} \right|_{x_i} = e^x \Big|_{x_i}$$

Now, using the 1st order forward approximation for the first derivative from eqn. (10), we have

$$\frac{y_{i+1} - y_i}{h} = e^{x_i} \quad \text{or} \quad y_{i+1} = y_i + e^{x_i} h$$

This recursive expression represents a discrete approximation to the original continuous ODE -- you may recognize this from your Differential Equations course as the simple Euler formula, $y_{i+1} = y_i + f(x_i, y_i)h$, for solving IVPs of the form, $y' = f(x, y)$, using numerical methods.

Now, let's evaluate the discrete recursive equation over the range $x = (0, 2)$ for several different step sizes, starting at $h = 0.1$ and decreasing by a factor of 10 each time until $h = 10^{-10}$. Note that, to reach a value of $x_f = 2$, the number of steps required is $N = 20$ for $h = 0.1$. However, for $h = 10^{-10}$, we will need to do 20 billion steps (i.e. 20×10^9 steps)! Thus, although the truncation error is expected to be negligible when h is very small, the very large number of arithmetic operations required to integrate the IVP to the desired x_f may start to show some appreciable round off error -- even with 64-bit arithmetic.

The **tradeoff.m** Matlab program listed in Table 4 implements the above ideas and compares the relative error in the numerical result for $y(x)$ at $x = x_f = 2$ relative to the exact value. A short table of numerical results from **tradeoff.m** is presented below and a plot of the relative error vs. step size on log-log axes is shown in Fig. 1. Note that this case was run in Oct. 2017 on my Surface Pro 4 tablet and, as you can see, the last step took nearly 7 minutes to run (however, just two years ago my Intel I7 desktop took over 30 minutes to run this last step -- computer processing speed just keeps getting better and better!!!):

```
>> tradeoff
```

h	N	xf	yf	rel error	time(s)
1.00e-01	20	2.0000000000e+00	7.0749266202e+00	4.25e-02	0.00e+00
1.00e-02	200	2.0000000000e+00	7.3571640605e+00	4.32e-03	0.00e+00
1.00e-03	2000	2.0000000000e+00	7.3858621033e+00	4.32e-04	0.00e+00
1.00e-04	20000	2.0000000000e+00	7.3887366514e+00	4.32e-05	0.00e+00
1.00e-05	200000	2.0000000000e+00	7.3890241537e+00	4.32e-06	0.00e+00
1.00e-06	2000000	1.9999999999e+00	7.3890529042e+00	4.32e-07	4.70e-02
1.00e-07	20000000	2.0000000003e+00	7.3890557798e+00	4.32e-08	3.91e-01
1.00e-08	200000000	1.9999999962e+00	7.3890560623e+00	4.96e-09	3.92e+00
1.00e-09	2000000000	2.0000000753e+00	7.3890562860e+00	2.53e-08	3.90e+01
1.00e-10	20000000000	2.0000001522e+00	7.3890567090e+00	8.26e-08	4.13e+02

The exact result should be $y_f = \exp(2) = 7.3890560989e+00$

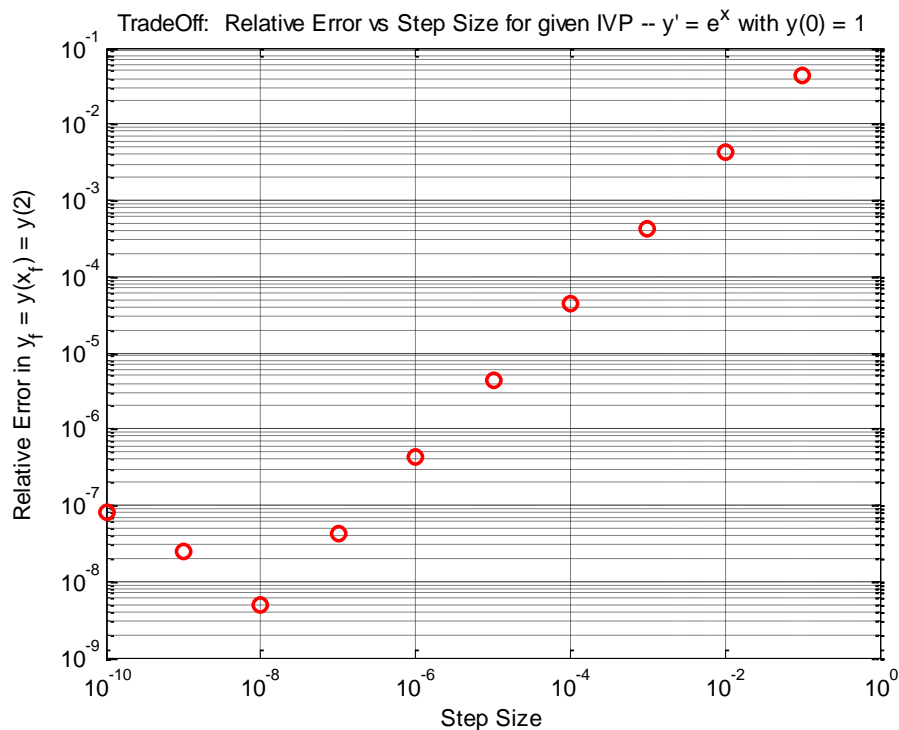


Fig. 1 Relative error vs. step size for solution of a simple IVP using the Euler method.

Table 4 Listing of the tradeoff.m program.

```

%
% TRADEOFF.M Show trade-off in truncation and round off error when solving
% a simple IVP: y' = exp(x) with y(0) = 1
% (used to generate plot of error vs step size for Fig. 1 in Lesson 4 Lecture Notes)
%
% Written by J. R. White, UMass-Lowell (last update: Oct. 2017)
%

clear all, close all

%
% set different step sizes (decreasing by a factor of 10)
% this is what I really want (note h = 1e-10 only works for 64-bit machines)
h = [1e-1 1e-2 1e-3 1e-4 1e-5 1e-6 1e-7 1e-8 1e-9 1e-10]';
% use the next line for debugging (this runs in a minute or so) -- comment out
% for real case
% h = [1e-1 1e-2 1e-3 1e-4 1e-5 1e-6 1e-7 1e-8 ]';
N = round(2./h); % number of steps

%
% solve IVP for each h
xf = zeros(size(h)); yf = zeros(size(h));
rerr = zeros(size(h)); time = zeros(size(h));
for n = 1:length(h)
    t = clock; % start timer
    x = 0; y = 1; % initial conditions
    for i = 1:N(n)
        y = y + exp(x)*h(n); % increment y
        x = x + h(n); % increment x
    end
    time(n) = etime(clock,t); % elapsed time (s)
    xf(n) = x; yf(n) = y; % final conditions
    rerr(n) = abs((yf(n)-exp(2))/exp(2)); % absolute relative error at xf = 2
end

%
% create short table of results
fprintf(' h N xf yf rel
error time(s)\n')
for n = 1:size(h)
    fprintf(' %10.2e %13i %18.10e %18.10e %10.2e %10.2e \n', ...
        h(n),N(n),xf(n),yf(n),rerr(n),time(n));
end
fprintf('\n The exact result should be yf = exp(2) = %20.10e \n',exp(2));

%
% make plot of results
loglog(h,rerr,'ro','LineWidth',2),grid
title('TradeOff: Relative Error vs Step Size for given IVP -- y'' = e^x with y(0) = 1')
xlabel('Step Size'), ylabel('Relative Error in y_f = y(x_f) = y(2)')

%
% end of problem

```

There are several interesting points to note in these data. First, for large h , the truncation error dominates the calculation, and the total relative error is directly proportional to h (as expected for the 1st order Euler method), where we see a factor of 10 decrease in the relative error in y_f each time we decrease h by a factor of 10. This behavior is also shown in Fig. 1, where the right portion of the graph of relative error vs. step size has a slope of unity (on a log-log plot, a 1 decade change in h leads to a 1 decade change in relative error -- thus, a slope of +1). However, the decreasing error does not continue indefinitely. When $h = 10^{-8}$, we start to see a slight deviation from the unity slope and clearly, when h is less than 10^{-9} , the slope is now nearly -1 -- that is, a decrease in h leads to an increase in the relative error!!!

The abrupt change when h becomes very small (about 10^{-8} for this case) is due to the fact that the truncation error is now totally negligible and the round off error is the dominant component of

the total error. And, as we know, decreasing the step size further only increases the number of computations, which, in turn, increases the round off error. Thus, there is clearly a point of diminishing returns, where decreasing the step size further actually increases the inherent error in the computations -- and this is something you should always keep in mind!!!

From this simple example, it should be clear that, in most practical calculations with well-written algorithms and a 64-bit word size (double precision arithmetic), truncation error will probably be the dominant concern. Ideally, for the best calculation possible, it would be optimum to operate in the region just to the right of the minimum in Fig. 1 -- a step size of about $10^{-6} - 10^{-7}$ for this problem. However, in practical engineering analysis, it is rare that this level of accuracy is needed. Instead, maximum relative errors on the order of $10^{-4} - 10^{-5}$ are usually more than sufficient for most problems, which relaxes the constraint on h considerably ($h = 10^{-3} - 10^{-4}$ would be a good choice for this problem). Also, in many real engineering analysis problems, the computational time is the dominant issue, so usually one relaxes the accuracy requirement somewhat to obtain increased computational efficiency. Thus, in most practical situations, the typical operating region is near the top right portion of the relative error vs. step size graph, where truncation error is definitely the dominant error component. However, although we are more often concerned about the tradeoff associated with truncation error versus computational efficiency, please do not forget about round off error -- because it can really cause headaches when it becomes a dominate issue...

Note: I should note that, because of round off error, the **tradeoff.m** program gave me a number of problems when I first wrote this routine for a 32-bit machine. For example, even though the statement $N = 2./h$ should give an integer for the values of h chosen, because of round off error, N was not exactly an integer for all cases. This became a problem in the **for ... end** loop, when trying to reach the precise value of $x_f = 2$. The **round** command resolved this issue. Also notice how round off error affects the value of x_f for large N (for N greater than 2 million). Even the simple expression, $x_{i+1} = x_i + h$, gives noticeable error when repeated enough times! You should also notice the use of the **clock** and **etime** commands in program **tradeoff.m**. These functions can be used when you are interested in obtaining timing information for a given algorithm or when comparing the efficiency of two or more methods. Finally, note that the times given in the above output from the **tradeoff.m** program are associated with my 1-year old Surface Pro 4 laptop as of Oct. 2017 -- your results, if you run the program, may be different...

Well, this completes our brief overview of the inherent errors associated with all numerical computations on the computer. You should now have a reasonable understanding of round off error and truncation error, and on the relationship of these two error components. As noted above, truncation error is usually the dominant error component in most engineering calculations, so this is something you should always consider in routine analysis -- and we will discuss this again, as needed, in subsequent lessons. Round off error, on the other hand, is usually quite small when using 64-bit arithmetic and well-written algorithms. It can be a real problem, on occasion, so you should also keep a watchful eye for anomalous results. My motto is that "Everything I do is wrong, until proven to be correct!" -- so I always take a critical look at any computer results before I believe them. I strongly encourage you to also be very critical with any computer generated results, and to use your engineering judgment and intuition to help

you interpret the data and to identify any subtle problems with your analysis. With a critical eye, you will be able to spot most anomalous numerical behavior that occurs, as well as other logic and coding bugs within your programs.

To expand on some of the concepts introduced in this lesson, I have also included three additional worked-out examples that provide some further insight into working with infinite series on the computer and on the Finite Difference method for solving differential equations. These examples are available as separate pdf files, as follows:

infinite_series.pdf -- On Evaluating Infinite Series – An Example

planewall_1.pdf -- Evaluating and Plotting Space-Time Temperature Distributions

fd_intro.pdf -- Introduction to Finite Difference Methods

After you have finished your reading assignment for this lesson and after reviewing the above examples, you should be ready to do HW #4 (see **hw4xxx.pdf**). This homework involves several problems that require some discussion as well as the development of a few Matlab programs that use several of the concepts and techniques discussed in this lesson.

As done for the previous assignments, I prefer that you collect the Matlab m-files, the resultant plots and/or tabular results, any hand calculations, and a brief description of the results of each problem in a separate solution packet for each HW problem. Note, however, that for this assignment, Problems 1 and 2 do not require any Matlab programming (wow, this is a change!), so only the discussions and manipulations requested are required for these two problems.

Well, with a brief introduction into the subject of numerical error behind us, we are now ready to jump into a sequence of traditional subjects from the field of applied numerical methods. Our first subject will be on **Root Finding and Polynomial Operations**, which will open the door for solving a wide range of engineering analysis problems. Since solving real problems is our ultimate goal in this course, we are finally ready to have some fun! So, let's do it...