

- Let $x = a$ and $x = b$ be the brackets for a root-finding problem for a scalar function $f(x)$. If $\delta = b - a$ is the size of the *original* bracket, the size of the interval containing the root after *four* iterations of the bisection method is $\delta/16$.
- Show calculations to justify your answer to the preceding problem.

Solution: For each iteration, the width of the current interval is reduced by a factor of 2. The table to the right shows how the interval containing the root is reduced from its original value of δ .

The $f(x)$ function does not need to be known in order to compute the size of the interval at each iteration. Starting with an initial interval of $\delta = b - a$, the size of the interval bracketing the root is reduced to $\delta/2^n$ after n iterations.

Iteration	Interval
0	δ
1	$\delta/2$
2	$\delta/4$
3	$\delta/8$
4	$\delta/16$

- What value is stored in **s** after the code snippet listed on the right is executed?

$x > 5$ is *false*

$x < 7$ is *true* so $s = s + 2$ is executed.

Therefore, the value stored in **s** is replaced with $1 + 2 = 3$.

```
x = 3;
s = 1;
if x > 5
    s = s + 3;
elseif x < 7
    s = s + 2;
else
    s = s + 1;
end
```

- Given that the `testfun.m` m-file is in the MATLAB path, what value(s) are printed when the code in the block labeled “Command Window” is executed?

Solution:

In the command window:

```
x = 3;
y = 1;
```

Inside `testfun`:

```
a = 3    (input value)
b = 1    (input value)
x = 3 + 1 = 4
y = 1 - 3 = -2
```

Back in the command window:

```
a = 4    (returned value)
b = -2   (returned value)
```

So, after `testfun` is executed the following values are defined in the command window: $a = 4$, $b = -2$, $x = 3$, $y = 1$, and the `fprintf` statement produces the following output

```
4 -2 3 1
```

The values of **x** and **y** are unchanged in the command window memory space. The **x** and **y** variables inside `testfun` are in a separate memory space.

`testfun.m`:

```
function [x,y] = testfun(a,b)
x = a+b;
y = b-a;
```

Command Window:

```
>> x = 3; y = 1;
>> [a,b] = testfun(x,y);
>> fprintf('%d %d %d %d\n',a,b,x,y);
```

5. Which of the following sets of MATLAB code will draw a horizontal line? Circle all that apply.

Solution:

- (c) `x=[1 0]; y=[0 0]; plot(x,y,'-');`
 (d) `x=[0 1]; y=5*[1 1]; plot(x,y,'-');`

Commands that do not:

- (a) `x=1; y=0; plot(x,y,'-');`
 No line is drawn because only one point $(x, y) = (1, 0)$ is plotted.
 (b) `x=[1 1]; y=[0 0]; plot(x,y,'-');` No line is drawn because the two (x, y) points are identical.
 (e) `x=zeros(1,5); y=2+5x; plot(x,y,'-');`
 The statements have an error: $y = 2+5x$ should be $y = 2+5*x$. However, even if that error is fixed the statements do not create a straight line because all of the x values are the same.

6. Consider the following code snippet from a function that computes the series approximation to the $\sin(x)$ function.

```
term = x; s = term; tol = 5e-6;
i = 1; n=1;
while abs(term/s)>tol && n<maxterms
    i = i + 2; n = n + 1;
    term = -term*(x^2)/(i*(i-1));
    s = s + term;
end
```

The preceding code works, i.e., it does not cause an error and it results in a plausible approximation to $\sin(x)$.

Following a recommendation by Hamming¹, the convergence criteria could be rewritten as

```
while abs(term/(max([abs(x),abs(s)]))>tol && n<maxterms
```

List one advantage and one disadvantage of using the modified criterion when $x = \pi$.

Advantage:

The modified convergence criterion reduces unnecessary computations when the correct, converged value of s is close to zero. Specifically, the modified criterion prevents extra iterations with \mathbf{term} and \mathbf{s} are both small.

At $x = \pi$, $\sin(x) = 0$, so $\mathbf{s} \rightarrow 0$ as \mathbf{i} (or \mathbf{n}) increase.

The original convergence test, with \mathbf{s} alone in the denominator, is not meaningful when $\mathbf{s} \rightarrow 0$ because $\mathbf{abs}(\mathbf{term}/\mathbf{s})$ does a poor job of estimating the true size of \mathbf{term} . Replacing the denominator of the test with $\max(|x|, |s|)$ avoids the problem of $\mathbf{s} \rightarrow 0$ as long as \mathbf{x} is not also zero.

Disadvantage:

There are at least two disadvantages. The use of $\max([\mathbf{abs}(x), \mathbf{abs}(s)])$ in the denominator does not help as $x \rightarrow 0$, where $\sin(0)$ is also 0.

$|\sin(x)| \leq 1$ for all x . Therefore, $|x|$ is not a good scale for the sine function for $x > 1$. With the modified convergence criterion, the relative error when the tolerance is met increases

¹R. W. Hamming, *Numerical Methods for Scientists and Engineers*, 2nd ed., 1973, p. 23, McGraw Hill, New York.

with x . For example, the convergence tolerance is smaller at $x = 2\pi$ than it is at 4π , 6π , 8π , etcetera, while the true value of $\sin(x)$ is the same for all values of $x = n\pi$ where n is a positive or negative even integer.

For this problem an *absolute* tolerance such as

```
while abs(term)>tol && n<maxterms
```

should be used since the quantity being computed ($\sin(x)$) is bounded in a narrow range, i.e., $|\sin(x)| \leq 1$.

7. Consider the `nTermSine2.m` and `sinn.m` functions listed on the sheet labeled **Code Listings**.

Running `sinn(2*pi)` produces the following text output.

```
>> sinn(2*pi)

Approximate value of sin(x) for x =    6.283e+00

   n      sum      abs error      rel error
   2  -3.506e+01  -3.506e+01  1.431e+17
   4  -3.016e+01  -3.016e+01  1.231e+17
   8  -9.325e-02  -9.325e-02  3.807e+14
  16  -2.438e-11  -2.438e-11  9.953e+04
  32   4.373e-16   6.822e-16  -2.785e+00
  64   4.373e-16   6.822e-16  -2.785e+00
 128   4.373e-16   6.822e-16  -2.785e+00
```

Normally we expect the error to decrease as the number of terms in the series increases. In this case, the error appears to stall.

- (a) if n is allowed to increase, will the error eventually be reduced?

No.

- (b) What is the cause of the error in the series approximation for $n \geq 32$?

The errors remaining for $n \geq 32$ are due to **roundoff and cancellation** in accumulating the first few terms in the sum. Because $x = 2\pi > 1$, the numerator of the first terms in the series are larger than the denominator, which causes these terms to be larger than one. Notice that $\text{sum} \approx 35$ for $n = 2$. Therefore, many terms are needed before series converges to zero. Cancellation errors that occur when the large leading terms (the first few terms at small i) cannot be undone by adding more terms.

- (c) Is the large relative error an indication of a significant problem with the algorithm? Provide some justification for your answer.

No. The large relative error occurs when the absolute error, ea , is divided by the (approximate) value of $\sin(2\pi)$ that is returned by MATLAB's built-in `sin` function.

8. The equation $xe^x = 1$ has a root on the interval $0.4 \leq x \leq 0.7$.

- (a) Rearrange the given equation into a form suitable for use with an automated root-finding procedure. In other words, what is a useful form of the $f(x)$ function?

$$f(x) = xe^x - 1$$

- (b) Write the m-file that can be used *in conjunction with* the `bisect` function from the NMM toolbox to find x . The help text for `bisect` is included on the Universal Cheat Sheet.

Make sure that your m-file is complete and ready to use with `bisect`. *Do not* put the bisection code into your m-file!

```
function f = fun(x)
f = x.*exp(x) - 1;
```

- (c) What is the file name for the code you wrote in part (b)?

`fun.m`

- (d) Write the command line statement(s) (or equivalently, the m-file code) that calls `bisect` to find the root.

```
r = bisect('fun', [0.4, 0.7])
```

- (e) Manually perform two iterations of the bisection algorithm. *Show your calculations* and then put your intermediate results into the following table. x_{new} is the new guess at the root at the end of each iteration. Do not leave empty cells in the table.

a	b	$f(a)$	$f(b)$	x_{new}	$f(x_{\text{new}})$
0.4	0.7	-0.4033	0.4096	0.55	-0.04671
0.55	0.7	-0.04671	0.4096	0.625	0.1677
0.55	0.625	-0.04671	0.1677	0.5875	0.05720

9. The author of `sineTable` intended to print a table of $\sin(x)$ values. The contents of `sineTable.m` is listed on the left. Running `sineTable` with $n = 5$ and $n = 10$ produces the output displayed in the box on the right.

```
sineTable.m:
function sineTable(n)
% sineTable Print (x,sin(x)) values for 0 <= x <= 1

x = 0.0;   xmax = 1;   dx = xmax/n;

fprintf('  x      sin(x)\n');
while x<1
    fprintf('%6.3f   %8.6f\n',x,sin(x))
    x = x + dx;
end
```

```
Command Window:
>> sineTable(5)
  x      sin(x)
0.000  0.000000
0.200  0.198669
0.400  0.389418
0.600  0.564642
0.800  0.717356

>> sineTable(10)
  x      sin(x)
0.000  0.000000
0.100  0.099833
0.200  0.198669
0.300  0.295520
0.400  0.389418
0.500  0.479426
0.600  0.564642
0.700  0.644218
0.800  0.717356
0.900  0.783327
1.000  0.841471
```

- (a) Explain why `sineTable(5)` causes the loop to stop at $x = 0.8$, whereas `sineTable(10)` causes the loop to stop at $x = 1.0$

Solution: $n = 5 \Rightarrow dx = 0.2$ and $n = 10 \Rightarrow dx = 0.1$, but both values of dx are not exact due to roundoff. The decimal values 0.2 and 0.1 cannot be represented by a finite sum of powers of $1/2$.

x is computed by repeated addition of dx . Due to roundoff in the calculation of dx , the value of x after n additions may be less than, greater than, or equal to 1.0.

When $n = 5$ ($dx = 0.2$), the roundoff accumulates so that after the fifth pass through the loop, the value produced by $x+dx$ is a very small amount *greater* than 1.0. The roundoff accumulates differently for $n = 10$ ($dx = 0.1$) so that after ten passes through the loop x is a very small amount *less* than 1.0.

- (b) Modify the code so that the last row of the table has $x = 1.0$ for any n . Make sure the last row is not repeated. A maximum of 7 points out of 10 can be earned if your solution involves eliminating the loop for the calculation of the $\sin(x)$ values.

Solution: It is tempting to replace the `while` statement with `while x<=1`, and that change happens to work for $n = 5$ and $n = 10$. However, that change does not work for $n = 11$. Try it. However if after n steps the value stored in x is $1 + \varepsilon_m$ (or greater), then the test `while x<=1` will also fail.

A reliable solution is to introduce a counter to the loop to determine when to stop. Integer arithmetic does not suffer from roundoff. Here is a solution that implements integer arithmetic.

```

function sineTable(n)
% sineTable Print (x,sin(x)) values for 0 <= x <= 1

x = 0.0;  xmax = 1;  dx = xmax/n;
k = 0;

fprintf('  x      sin(x)\n');
while k<=n
  fprintf('%4d %6.3f %8.6f\n',k,x,sin(x))
  x = x + dx;
  k = k + 1;
end

```

An alternative solution is to replace the while loop with a for loop.

```

function sineTable(n)
% sineTable Print (x,sin(x)) values for 0 <= x <= 1

x = 0.0;  xmax = 1;  dx = xmax/n;

fprintf('  x      sin(x)\n');
for k=0:n
  fprintf('%4d %6.3f %8.6f\n',k,x,sin(x))
  x = x + dx;
end

```

Yet another solution is to precompute the values of x as

```
x = linspace(0,1,n+1);
```

and use a for loop to select each $x(i)$. The last argument of `linspace` is $n+1$ not n to reproduce the same values of x as in the exam question. for loop.

```

function sineTable(n)
% sineTable Print (x,sin(x)) values for 0 <= x <= 1

x = linspace(0,1,n+1);

fprintf('  x      sin(x)\n');
for i=1:length(x)
  fprintf('%6.3f %8.6f\n',x(i),sin(x(i)))
end

```

(c) Explain why your solution works for any n .

Solution: Calculations involving integers do not have roundoff. Both solutions shown above use integer arithmetic to determine the number of times the loop is executed. Floating point calculations are used only in the calculation of dx , x , and $\sin(x)$.