ME 350 Lab Exercise 4 `if` constructs, `while` loops
Fall 2017

Practice Reading `if` Constructs

What is the output of the following command sequences? Some of the sequences may print messages that are not correct or are consistent with the numerical and logical statements. The goal is to practice seeing both correct and incorrect code. When we are debugging, we are looking for incorrect code.

1. What is printed when the following code is executed?

```
z = 17;
x = sqrt(z);
if x<5
  fprintf('z is less than 25\n')
end
```

   What is printed for the preceding code if `z = 35` was used instead?

2. What is printed when the following code is executed?

```
z = 17;
x = sqrt(z);
if x<5
  fprintf('z is less than 25\n')
else
  fprintf('z is greater than 25\n')
end
```

   What is printed for the preceding code if `z = 35` was used instead?

3. Is this code correct?

```
z = 17;
x = sqrt(z);
if x<5 && x>6
  fprintf('z is less than 25\n')
else
  fprintf('z is greater than 36\n')
end
```

4. Does the output make sense? Are the printed messages consisten with the logic of the `if`...`else` construct?

```
z = 17;
x = sqrt(z);
if x>5 && x<6
  fprintf('z is greater than 25\n')
else
  fprintf('z is less than 36\n')
end
```

5. Is the message in the `fprintf` statement consistent with the logic of the `if` construct?

```
z = 17;
x = sqrt(z);
if x>5 && x<6
  fprintf('z is between 25 and 36\n')
end
```

6. The following code might not do what you think.

```
x = linspace(0,10);
y = zeros(size(x));

for i = 1:length(x)
  if x<0
    y = 0;
  elseif x<3
    y = 1.4;
  elseif x<7
    y = 2.7;
  else
    y = 7;
  end
end

plot(x,y,'.')
```

7. This code works as expected. Can you explain the difference between this (following) code block and the preceding code block?

```
x = linspace(0,10);
y = zeros(size(x));

for i = 1:length(x)
  if x(i)<0
    y(i) = 0;
  elseif x(i)<3
    y(i) = 1.4;
  elseif x(i)<7
    y(i) = 2.7;
  else
    y(i) = 7;
  end
end

plot(x,y,'.')
```

Equality tests are susceptible to roundoff errors

Does the code in Example 1 and Example 2 do what you expect?

Example 1:

```
x = 30*pi/180;
f = 1 - sin(x)^2 - cos(x)^2;     %  Identity:  sin(x)^2 + cos(x)^2 = 1

if f==0
  disp('Formula is correct')
else
  disp('Formula is incorrect')
end
```

Example 2:

```
s = 2.6;
v = s + 0.6;
w = s + 0.2 + 0.2 + 0.2;

if v == w
  disp('Addition works')
else
  disp('Addition is broken')
end
```

In general, avoid exact tests for equality

Redesign the tests to avoid round-off trouble

Example 1 Revised:

```
x = 30*pi/180;
f = 1 - sin(x)^2 - cos(x)^2;     %  Identity:  sin(x)^2 + cos(x)^2 = 1

delta = 5.0e-9;                  %  delta is tolerance on "equality" test

if abs(f)<delta
  disp('Formula is correct')
else
  disp('Formula is incorrect')
end
```

Example 2 Revised:

```
s = 2.6;
v = s + 0.6;
w = s + 0.2 + 0.2 + 0.2;

delta = 5.0e-9;                  %  delta is tolerance on "equality" test

if abs(v-w) < delta
  disp('Addition works')
else
  disp('Addition is broken')
end
```

Practice with `if` Constructs

Finding the Maximum Value in a Vector

What does the following code do?

```
function xmax = mymax(x)

xmax = x(1);
for i=2:length(x)
  if x(i)>xmax
    xmax = x(i);
  end
end
```

What built-in function performs the same task? Write an m-file to test whether this function works by comparing it to the outcome of the built-in `max` function.

Hint:

```
x = 300*rand(...)
err = mymax(x) - max(x);
```

Is this a situation where "close enough" applies, or would an exact match be appropriate?
How would you use `mymax` to find the maximum absolute value in a vector?

Testing properties of distributions generated by `randn`

1. Download the `testStats` function from the lab web site.
2. Modify the `testStats` function so that it also prints the number of sample values that are greater than two standard deviations from the mean. Keep the code for counting and printing the number of sample values that are greater than one standard deviation from the mean.
   - Create a variable to store the counts. *Hint*: create `n2`
   - Add a test to determine whether to increase the count.
   - Add an `fprintf` statement to display `n2` and `n2/n`.
3. Modify the plot so that the decoration includes a vertical dashed lines at $\pm 2\sigma$

In completing the preceding assignment two students developed two different versions of the code. Do these codes produce different results? If they produce the same results, is there an advantage to using on approach over the other?

Version A:

```
dx = abs(x(i)-xave);

if dx > sig
  n1 = n1 + 1;
end
if dx > 2*sig
  n2 = n2 + 1;
end
```

Version B:

```
dx = abs(x(i)-xave);

if dx > sig
  n1 = n1 + 1;
  if dx > 2*sig
    n2 = n2 + 1;
  end
end
```

Automatic truncation of iterative sequences

Background

Iteration is a common component of numerical algorithms. In the most abstract form an iteration generates a sequence of scalar values $x_k,\ k = 1, 2, 3, \ldots$. The sequence converges to a limit ξ if

$$|x_k - \xi| < \delta \quad \text{for all} \quad k > N$$

where δ is a small number called the convergence tolerance. In this case we say that the sequence has converged to within the tolerance δ after N iterations. The problem with this statement of convergence is that in general the limit ξ is not known unless (for a convergent iteration) the value of k is allowed to approach infinity.

In a practical calculation it is important to be able to detect convergence as soon as the tolerance is met, not after $k \to \infty$. At convergence one is in essence declaring that x_k is "close enough" to the *unknown* value of ξ.

Fortunately the preceding condition of convergence is equivalent to

$$|x_\ell - x_k| < \delta \quad \text{for all } \ell, k > N$$

In practice the test is expressed as

$$|x_k - x_{k-1}| < \delta \quad \text{when } k > N \tag{1}$$

where k is an iteration counter. Equation (1) says that an sequence converges when successive values of the iteration differ by less than a tolerance. This is not a foolproof criterion, but it's a good start.

Many numerical methods involve iterations that we hope converge toward a limit. The index k is the *iteration counter*, and δ is a convergence tolerance.

We implementing a convergence test in a numerical method, have a choice to use either an absolute or a relative convergence criterion

$$|x_k - x_{k-1}| < \delta_a \qquad \left|\frac{x_k - x_{k-1}}{x_{k-1}}\right| < \delta_r \tag{2}$$

where δ_a and δ_r are absolute and relative convergence tolerances, respectively.

Iterative Calculation of $\sqrt{x}$

Consider the iterative formula for approximating $\sqrt{x}$.

$$r_k = \frac{1}{2}\left(r_{k-1} + \frac{x}{r_{k-1}}\right) \tag{3}$$

1. Using the `newtsqrtShell.m` m-file in Listing 1 as a starting point, create a new m-file called `newtsqrta` that implements an absolute convergence tolerance for the approximation to $\sqrt{x}$.

   Note the first `if` statement in `newtSqrtShell` is a test to make sure that the input is positive.

```
if x<0,  error('Negative input to newtsqrt not allowed');  end
```

It is good practice to check that the inputs to your function are within an acceptable range. It is better to get this error message than some strange computational result.

2. Write a `testsqrt` function that calls your `newtsqrta` function for a range of x values and plots the absolute and relative error.
3. Using the `newtsqrta.m` m-file created in the preceding exercise as a starting point, implement a relative convergence tolerance for the approximation to $\sqrt{x}$. Call this new function `newtsqrtr`.
4. Write a `testsqrtr` function that calls your `newtsqrtr` function for a range of x values and plots the absolute and relative error.

```
function r = newtsqrt(x,delta,maxit)
% newtsqrt  Use Newton's method to compute the square root of a number.
%           Convergence is determined with an absolute tolerance.
%
% Synopsis:  r = newtsqrt(x,delta,maxit)
%
% Input:     x     = number for which the square root is desired
%            delta = (optional) absolute convergence tolerance.  Default: delta = 5e-9
%                    Iterations continue until abs(r-rold) < delta, where r and rold
%                    are the current and previous estimates of the square root
%            maxit = (optional) maximum number of iterations.  Default:  maxit = 25
%
% Output:    r = square root of x to within delta/2

if x<0,  error('Negative input to newtsqrt not allowed');  end
if x==0,      r=x;  return;   end
if nargin<2,  delta = 5e-9;   end
if nargin<3,  maxit=25;       end

r = x/2;  rold = x;   %  Initialize, making sure that convergence test fails on first try
it = 0;

while % ***  YOUR CODE GOES HERE  ***

   rold = r;                   %  Save old value for convergence test
   r = 0.5*(rold + x/rold);    %  Update the guess
   it = it + 1;                %  Increment the iteration counter

end

end
```

Listing 1: The `newtsqrtShell.m` m-file has an incomplete implementation Newton's method for computing $\sqrt{x}$.