SECTION 5: STRUCTURED PROGRAMMING IN MATLAB

ENGR 112 – Introduction to Engineering Computing

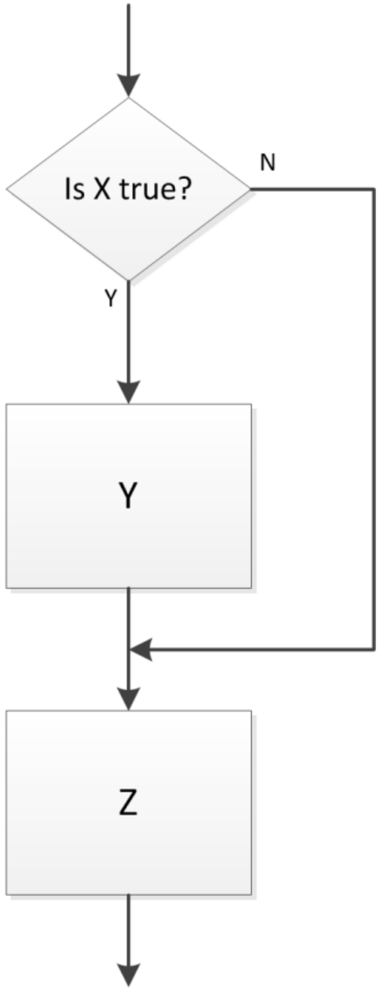
Conditional Statements

- `if` statements
- `if…else` statements
- Logical and relational operators
- `switch…case` statements

The `if` Statement

- We've already seen the ***if structure***
  - If X is true, do Y, if not, don't do Y
  - In either case, then proceed to do Z
- In MATLAB:

```
if condition
   statements
end
```



- *Statements* are executed ***if*** *condition* is ***true***
- *Condition* is a ***logical expression***
  - Either true (evaluates to 1) or false (evaluates to 0)
  - Makes use of ***logical and relational operators***
- May use a ***single line*** for a single statement:

```
if condition, statement, end
```

Logical and Relational Operators

Operator	Relationship or Logical Operation	Example
==	Equal to	`x == b`
~=	Not equal to	`k ~= 0`
<	Less than	`t < 12`
>	Greater than	`a > -5`
<=	Less than or equal to	`7 <= f`
>=	Greater than or equal to	`(4+r/6) >= 2`
~	NOT– negates the logical value of an expression	`~(b < 4*g)`
& or &&	AND – ***both*** expressions must evaluate to true for result to be true	`(t > 0)&&(c == 5)`
| or ||	OR – ***either*** expression must evaluate to true for result to be true	`(p > 1)||(m > 3)`

Short-Circuit Logical Operators

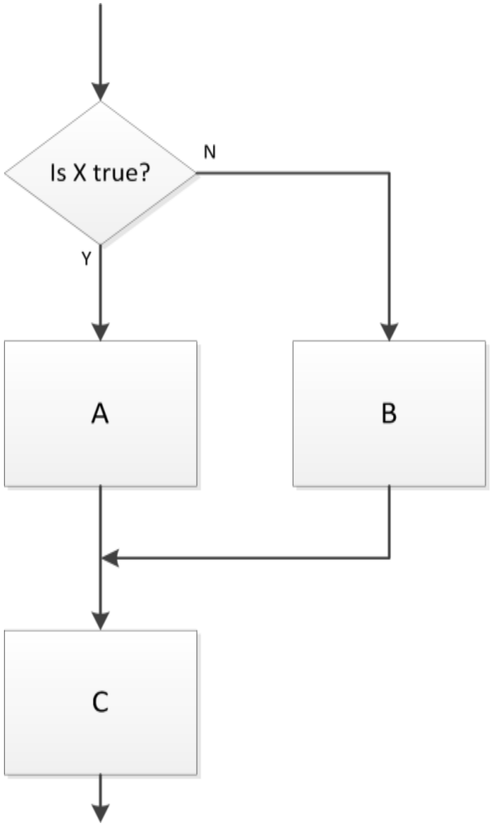
- Note that there are two *AND* and two *OR* operators available in MATLAB
  - **AND**: & or &&
  - **OR**: | or ||
- Can *always* use the single operators: & and |
- The double operators are ***short-circuit operators***
  - Only evaluate the second expression if necessary – faster
  - Can only be used with ***scalar*** expressions

The `if…else` Structure

- The ***if … else structure***
  - Perform one process if a condition is true
  - Perform another if it is false

- In MATLAB:

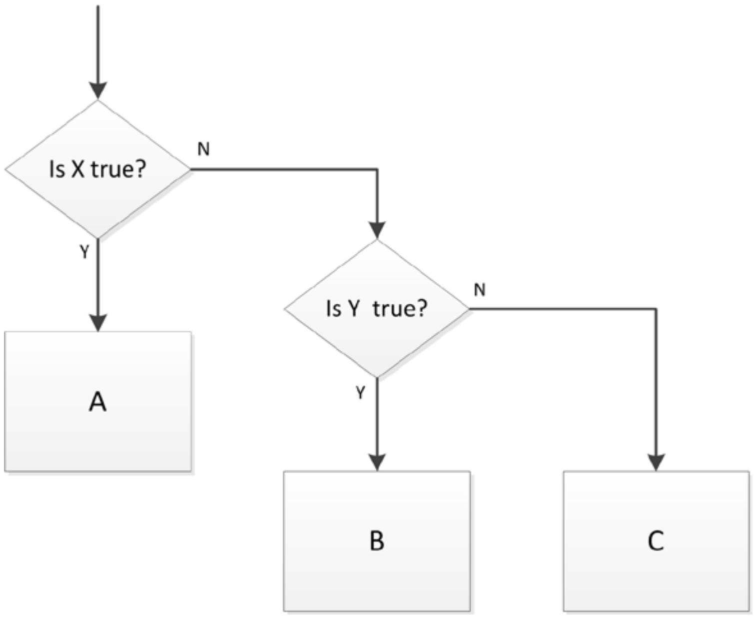
```
if condition
  statements1
else
  statements2
end
```



The if…elseif…else Structure

- The ***if … elseif … else structure***
  - If a condition evaluates as false, check another condition
  - May have an arbitrary number of ***elseif*** statements
- In MATLAB:

```
if condition_1
  statements_1
elseif condition_2
  statements_2
else
  statements_3
end
```



The `if…else`, `if…elseif…else` Structures

- Some examples:

```
if (t>=0)&&(p>8)
    x = p^2*t;
    y = 3*q + p;
else
    x = 0;
    y = q + p^2;
end
```

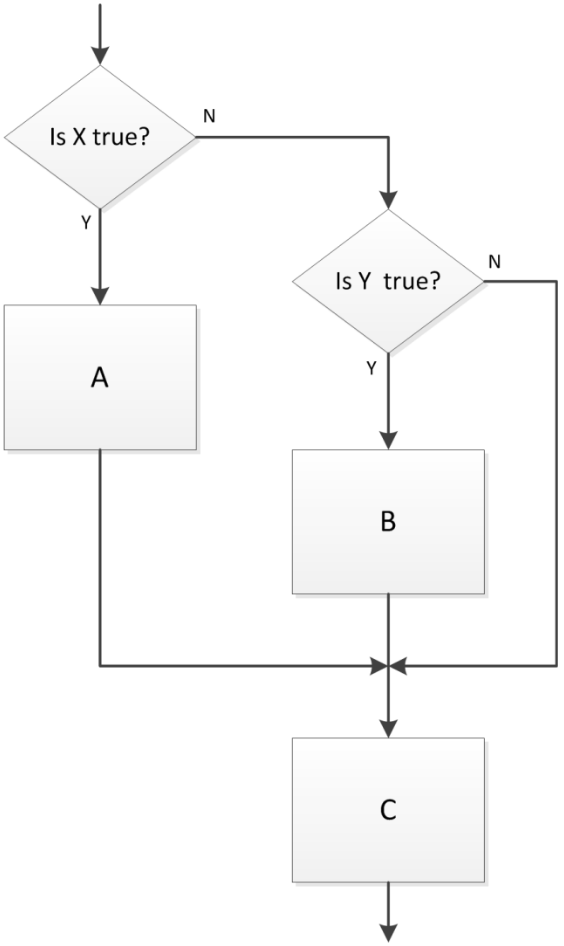
```
if x == 0
    f = 2*pi;
elseif x <= -1
    f = pi/4;
elseif y~=436 || x>18
    f = 0;
else
    f = 2*pi/3;
end
```

- Note that && and | | are used here, because expressions involve ***scalars***
  - The single logical operators, & and |, would work just as well

The `if…elseif` Structure

- We can have an `if` statement without an `else`
- Similarly, an `if…elseif` structure need not have an `else`
- In MATLAB:

```
if condition1
  statements1
elseif condition2
  statements2
end
```



The `switch` Structure

- The ***switch structure*** evaluates a single test expression
  - Branching determined by the value of the test expression

```
switch testexpression
   case value1
      statements1
   case value2
      statements2
   otherwise
      statements3
end
```

- An alternative to an `if`…`elseif`…`else` structure

The `switch` Structure

- An example – set the value of variable B to different values depending on the value of variable A:

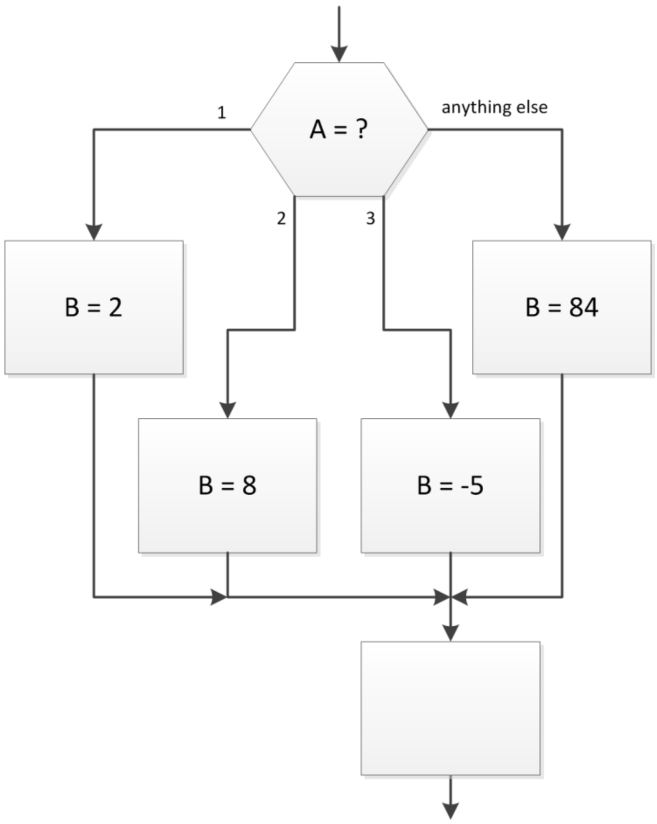
```
switch A
    case 1
        B = 2;
    case 2
        B = 8;
    case 3
        B = -5;
    otherwise
        B = 84;
end
```

- `otherwise` serves the same purpose as `else`
  - If the test expression does not equal any of the specified cases, execute the commands in the `otherwise` block

The `switch` Structure

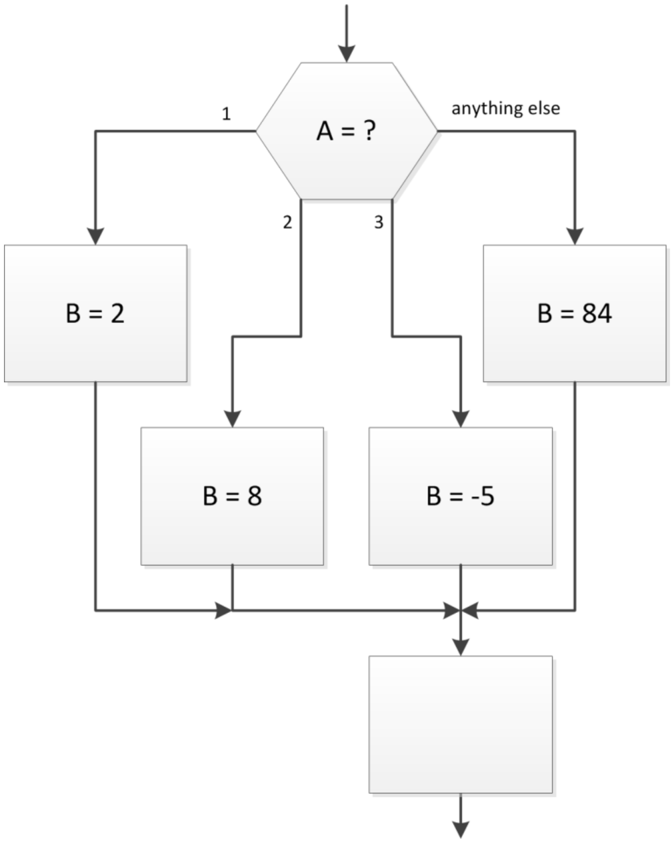
- In flowchart form, there is no direct translation for the switch structure
  - We'd represent it using an ***if…elseif…else*** structure
  - But, if there were, it might look something like this:

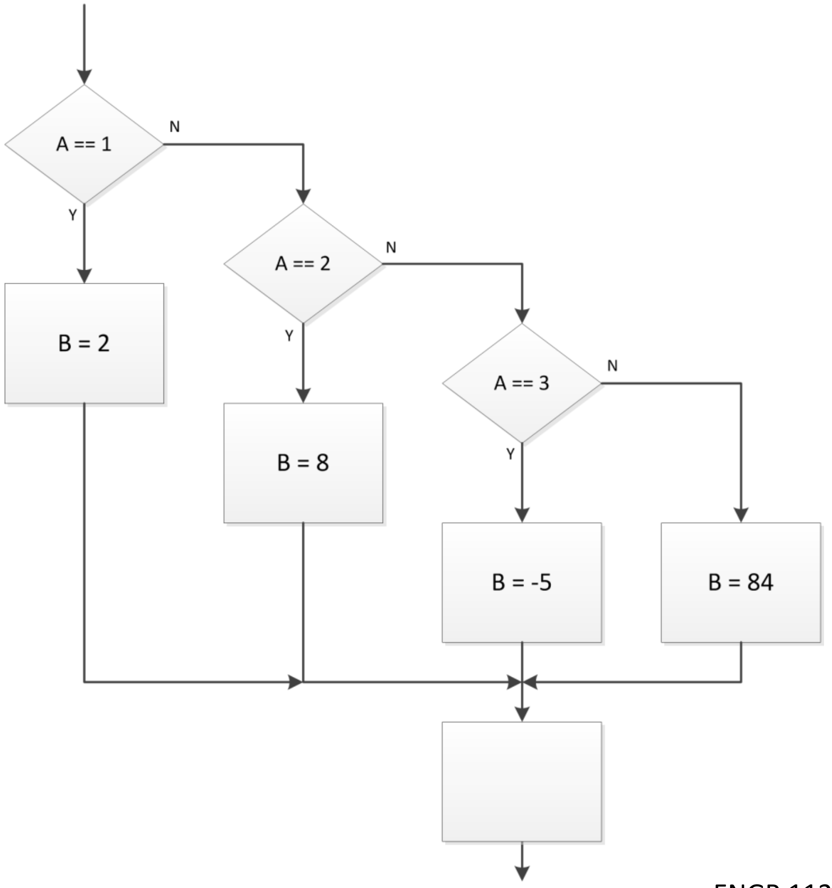
```
switch A
   case 1
      B = 2;
   case 2
      B = 8;
   case 3
      B = -5;
   otherwise
      B = 84;
end
```



The `switch` Structure

- An alternative to an ***if…elseif…else*** structure
  - Result is the same
  - Code may be more readable

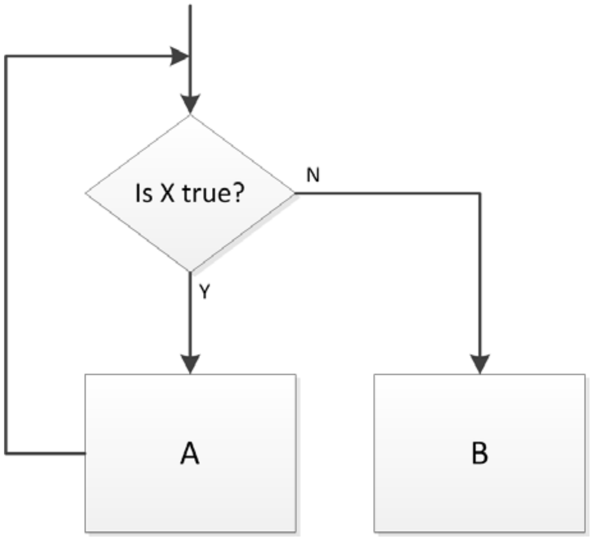




`while` Loops

The `while` loop

- The ***while loop***
  - *While* X is true, do A
  - Once X becomes false, proceed to B



- In MATLAB:

```
while condition
  statements
end
```

- *`Statements`* are executed as long as *`condition`* remains true
- *`Condition`* is a ***logical expression***

while Loop – Example 1

- Consider the following while loop example
  - Repeatedly increment x by 7 as long as x is less than or equal to 30
  - Value of x is displayed on each iteration, due to lack of output-suppressing semicolon

```
% increment a number by 7 until it exceeds 30
x = 12;
while x<=30
    x = x + 7
end
```

- x values displayed: 19, 26, 33
- x gets incremented beyond 30
  - All loop code is executed as long as the condition was true at the ***start of the loop***

The break Statement

- Let’s say we don’t want x to increment beyond 30
  - Add a conditional break statement to the loop

```
    % exit loop before exceeding 30
    x = 12;
while x<=30
        if (x+7)>30, break, end
        x = x + 7
end
```

- `break` statement causes loop exit before executing all code
- Now, if `(x+7)>30`, the program will break out of the loop and continue with the next line of code
- x values displayed: 19, 26
- For nested loops, a break statement breaks out of the current loop level only

while Loop – Example 1

- The previous example could be simplified by modifying the while condition, and not using a break at all

```
%% or, change the while condition so that x will not
% increment beyond 30
x = 12;
while (x+7)<=30
    x = x + 7
end
```

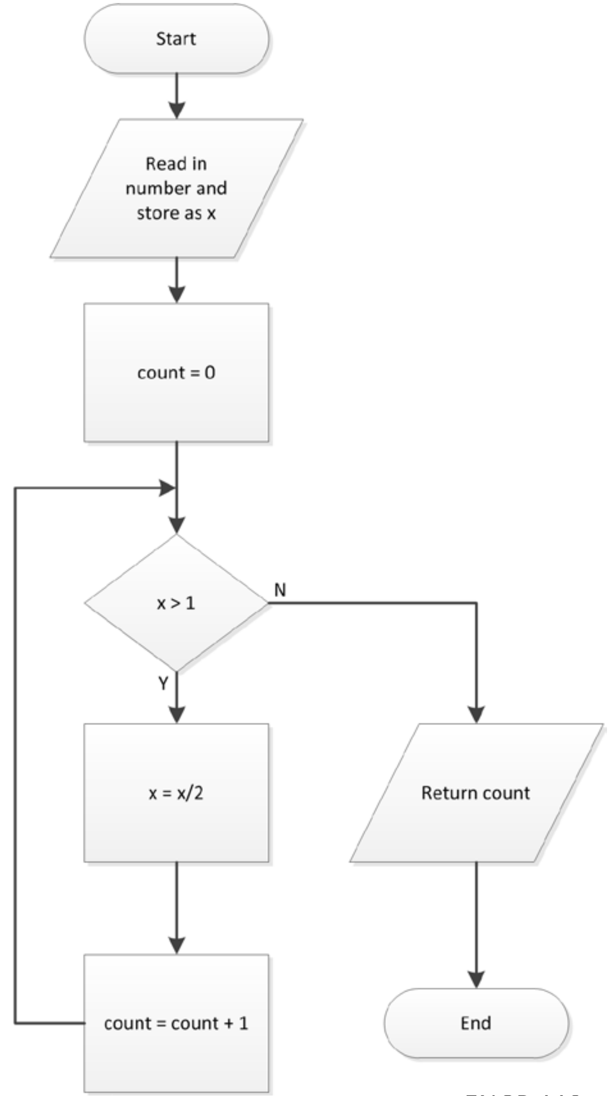
- Now the result is the same as with the break statement
  - x values displayed: 19, 26
- This is not always the case
  - The break statement can be very useful
  - May want to break based on a condition other than the loop condition
- break works with both while and for loops

while Loop – Example 2

- Next, let's revisit the while loop examples from Section 4
- Use `input.m` to prompt for input
- Use `display.m` to return the result

```
%% while loop example 2

x = input('Enter a number: ');

count = 0;

while x > 1
    x = x/2;
    count = count + 1;
end

display(count);
```

```
Enter a number: 130

count =

     8
```

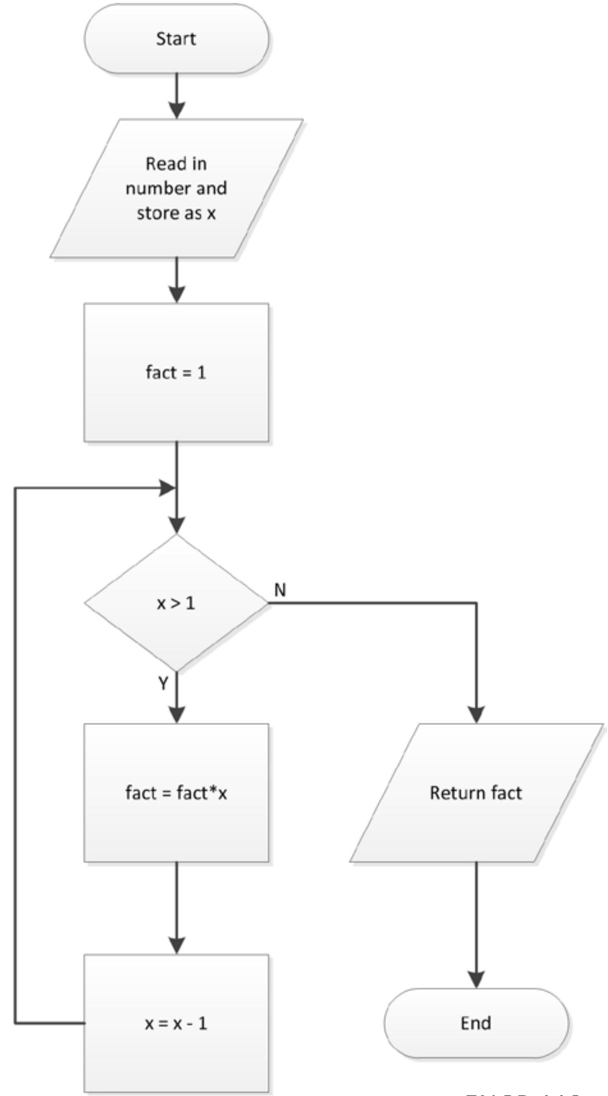


`while` Loop – Example 3

- Here, we use a `while` loop to calculate the factorial value of a specified number

```
%% while loop example 3

x = input('Enter an integer: ');

fact = 1;

while x > 1
    fact = fact*x;
    x = x - 1;
end

display(fact);
```

```
Enter an integer: 12

fact =

   479001600
```

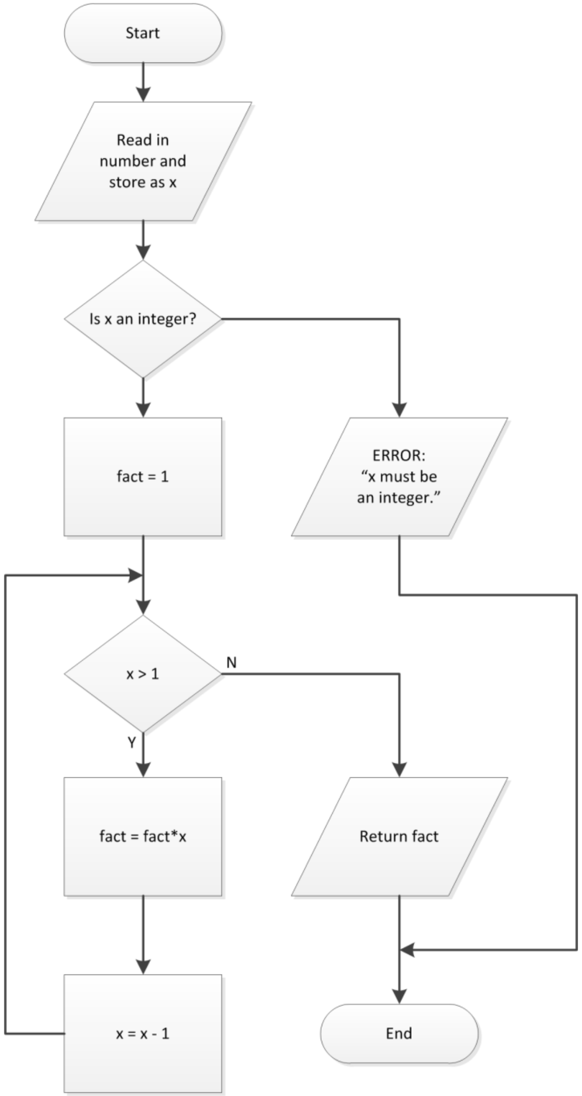


`while` Loop – Example 3

- Add error checking to ensure that x is an integer
- One way to check if x is an integer:

```
31      %% while loop example 3.1
32
33 -    x = input('Enter an integer: ');
34
35      % check if x is an integer
36 -    if x ~= int64(x)
37 -        error('ERROR: x must be an integer.')
38 -    end
39
40 -    fact = 1;
41
42 -    while x > 1
43 -        fact = fact*x;
44 -        x = x - 1;
45 -    end
46
47 -    display(fact);
```

```
Enter an integer: 11.5
Error using whileLoopEx (line 37)
ERROR: x must be an integer.
```

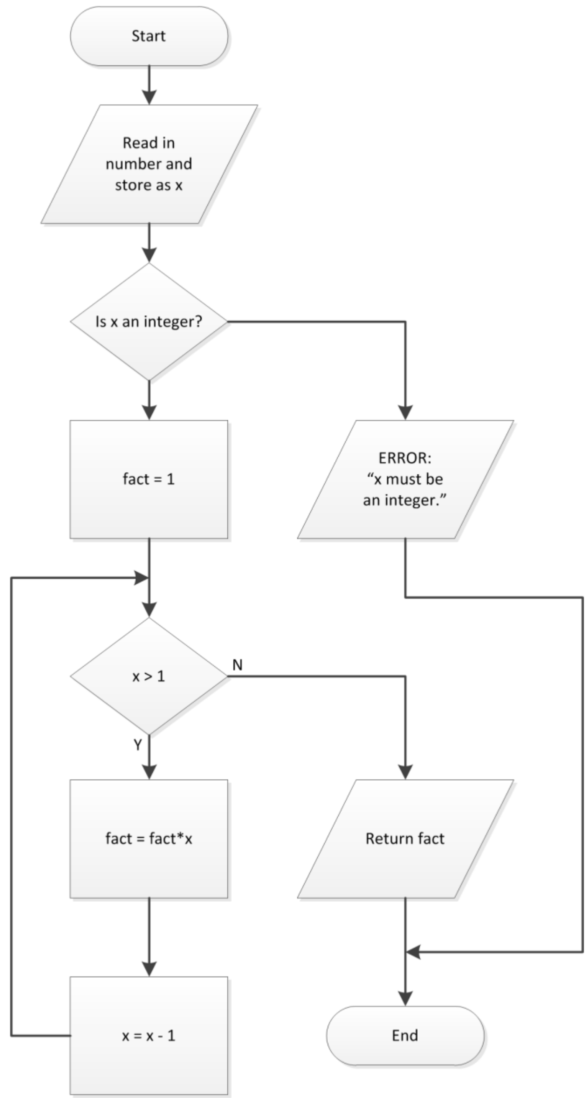


`while` Loop – Example 3

- Another possible method for checking if x is an integer:

```
49      %% while loop example 3.2
50
51 -    x = input('Enter an integer: ');
52
53      % check if x is an integer
54 -    if (x - floor(x)) ~= 0
55 -        error('ERROR: x must be an integer.')
56 -    end
57
58 -    fact = 1;
59
60 -    while x > 1
61 -        fact = fact*x;
62 -        x = x - 1;
63 -    end
64
65 -    display(fact);
66
```

```
Enter an integer: 20.3
Error using whileLoopEx (line 55)
ERROR: x must be an integer.
```



Infinite Loops

- A loop that never terminates is an ***infinite loop***
- Often, this unintentional
  - Coding error
- Other times infinite loops are intentional
  - E.g., microcontroller in a control system
- A while loop will never terminate if the while condition is always true
  - By definition, 1 is always true:

```
while (1)
  statements repeat infinitely
end
```

while (1)

- The `while (1)` syntax can be used in conjunction with a `break` statement, e.g.:

- Useful for multiple break conditions
- Control over break point
- Could also modify the while condition

```
while(1)
    iter = iter + 1;     % increment iteration index


    if xl == xu      % func(xr) == 0, exactly (unlikely)
        epsa = 0;
    else
        % update the root estimate
        xr = xu - func(xu)*(xu - xl)/(func(xu) - func(xl));
        % approximate the error
        epsa = abs((xr-xrold)/xr)*100;
    end

    % check if stopping criterion is satisfied or if maximum number
    % iterations has been reached
    if (epsa<=reltol)
        break
    elseif (iter >= maxiter)
        fprintf('\nMaximum # of iterations reached - exiting.\n\n')
        break
    end
end
```

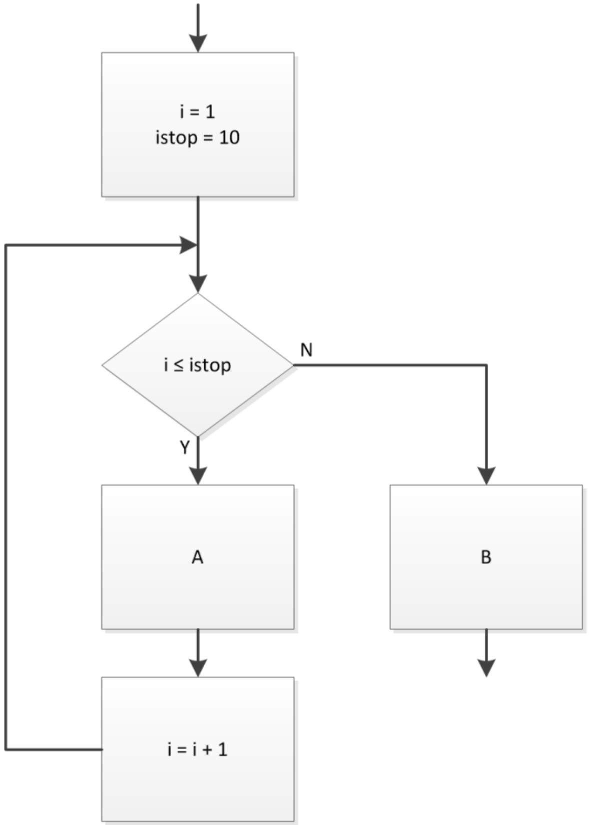
`for` Loops

The `for` Loop

- The ***for loop***
  - Loop instructions execute a specified number of times
- In MATLAB:

```
for index = start:step:stop
    statements
end
```

- Note the syntax – looks like a ***vector*** definition
  - `Statements` are executed once for each element in the vector
- However, `index` is actually a scalar
  - Increments through the vector of values

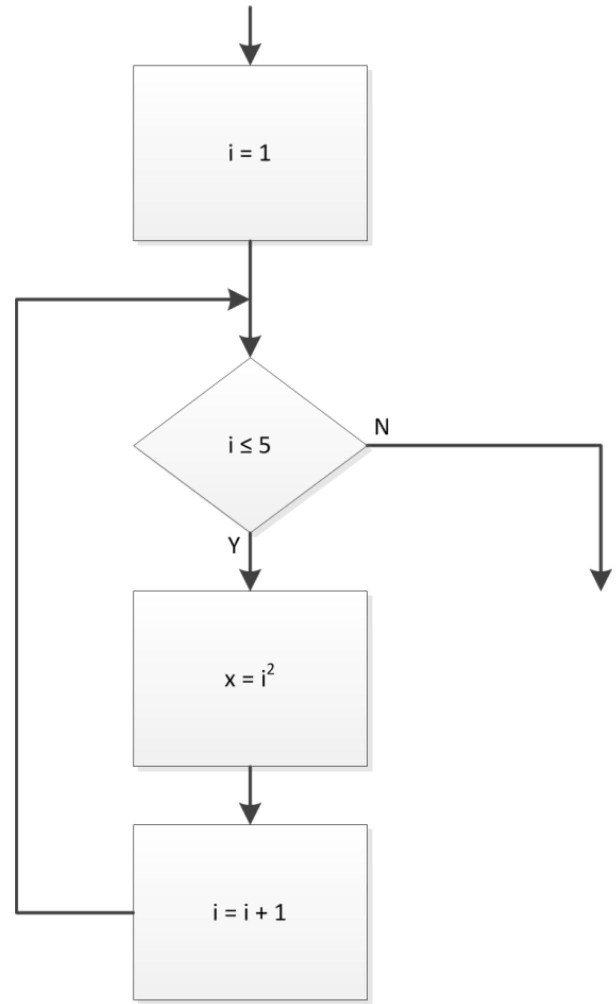


for Loop – Example 1

- Next, we'll revisit the for loop examples from Section 4
- Loop iterates 5 times
  - Value of scalar variable, x, reassigned on each iteration

```
    %% for loop example 1

    for i = 1:5
        x = i^2
    end
```

```
x =
     1
x =
     4
x =
     9
x =
    16
x =
    25
```

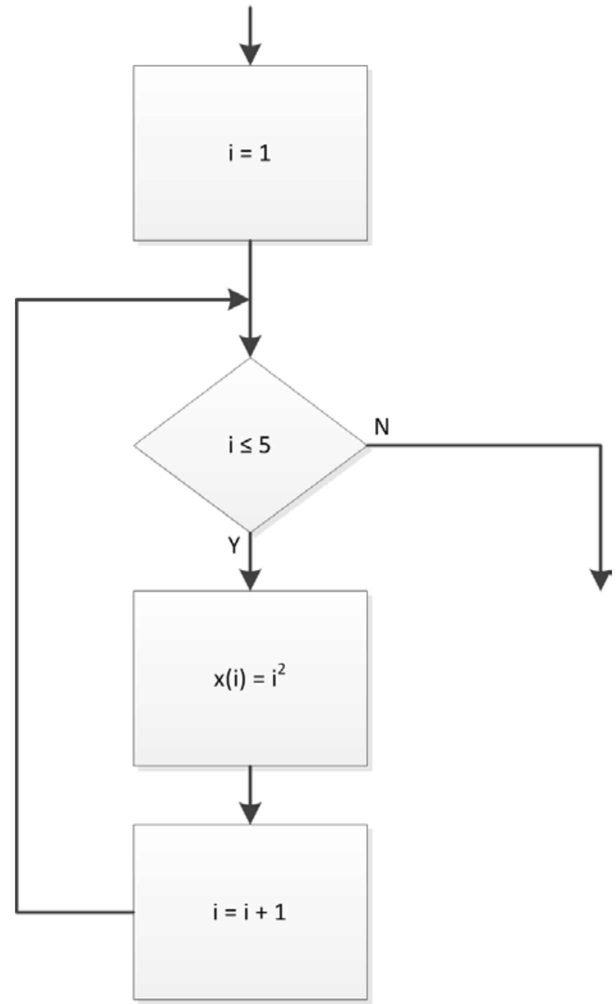


for Loop – Example 2

- Here, x is defined as a vector
- Loop still iterates 5 times
  - Successive values appended to the end of x
  - x grows with each iteration

```
%% for loop example 2

for i = 1:5
    x(i) = i^2
end
```

```
x =
     1
x =
     1     4
x =
     1     4     9
x =
     1     4     9    16
x =
     1     4     9    16    25
```



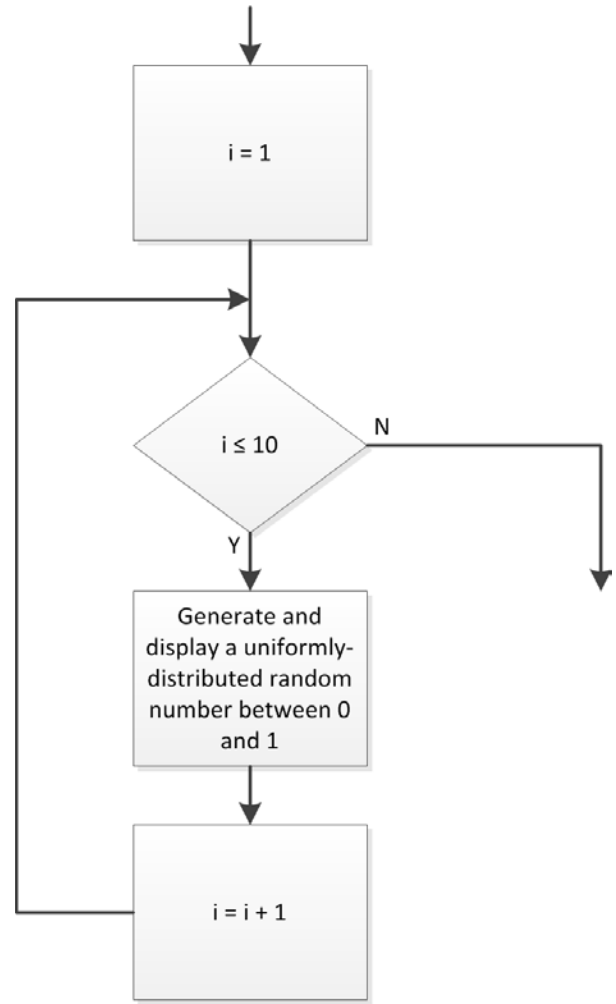
`for` Loop – Example 3

- In this case the loop counter is not used at all within the loop
- Random number generated on each of 10 iterations

```
%% for loop example 3

for i = 1:10
    x = rand;
    display(x)
end
```

```
x =
    0.5383
x =
    0.9961
x =
    0.0782
x =
    0.4427
x =
    0.1067
```

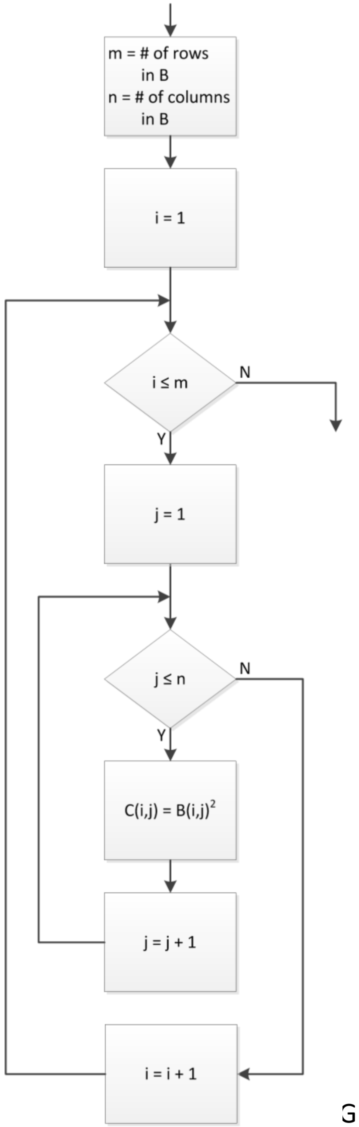
```
x =
    0.9619
x =
    0.0046
x =
    0.7749
x =
    0.8173
x =
    0.8687
```



Nested Loops

Nested Loop – Example 1

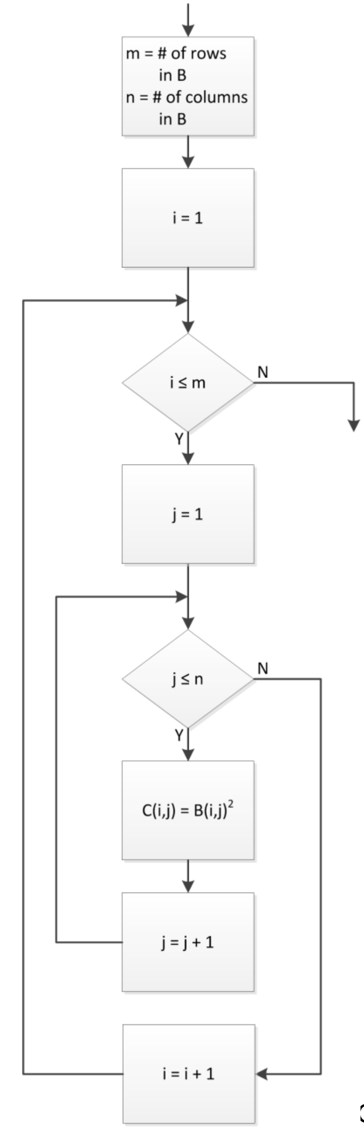
- Recall the nested for loop example from Section 4
- Generate a matrix C whose entries are the squares of the elements in B
  - ***Nested for loop***
  - Outer loop steps through rows
    - Counter is row index
  - Inner loop steps through columns
    - Counter is column index



Nested Loop – Example 1


```
    % define a matrix, B
    B = [1 2 3 4;
         9 7 5 3;
         2 4 6 8;
         8 7 6 5;
         0 1 3 9];

    m = size(B,1);      % # of rows in B
    n = size(B,2);      % # of columns in B

    for i = 1:m                 % loop through rows
        for j = 1:n             % loop through columns
            C(i,j) = B(i,j)^2;
        end
    end

    display(C);
```

```
C =

     1     4     9    16
    81    49    25     9
     4    16    36    64
    64    49    36    25
     0     1     9    81
```



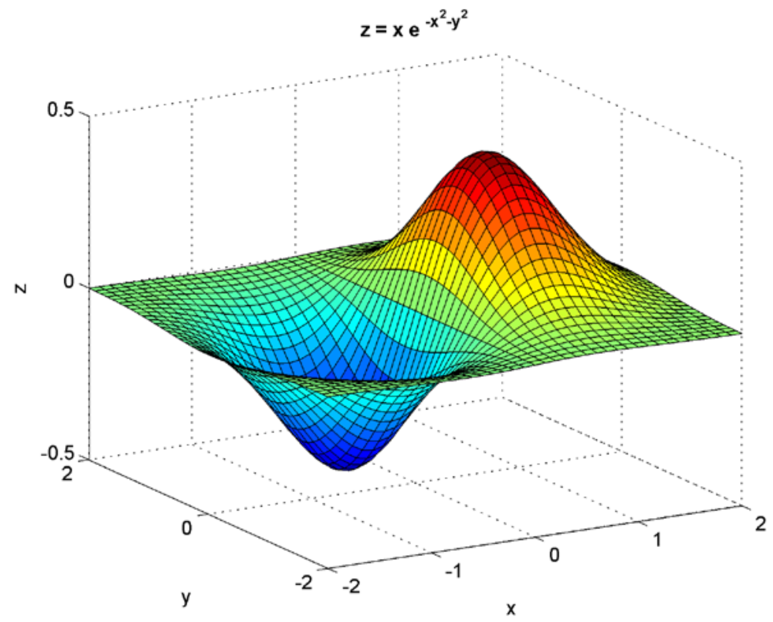
Nested for Loop – Example 2

- Evaluate a function of two variables:

$$z = x \cdot e^{-x^2 - y^2}$$

over a range of $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$

- A surface in three-dimensional space
- In Section 7, we'll learn how to generate such a plot



Nested for Loop – Example 2

$$z = x \cdot e^{-x^2 - y^2}$$

- Evaluate the function over a range of x and y

- First, define x and y vectors
- Use a nested for loop to step through all points in this range of the x-y plane

```
% nestedForEx2.m

clear all; clc

% generate x and y vectors
x = -2:0.1:2;
y = -2:0.1:2;

% loop through all x-y points and calculate z
for i = 1:length(x)
    for j = 1:length(y)
        z(i,j) = x(i)*exp(-x(i)^2 - y(j)^2);
    end
end

```

The MATLAB Debugger

Debugging

- You've probably already realized that it's not uncommon for your code to have errors
  - Computer code errors referred to as ***bugs***
- Three main categories of errors
  - ***Syntax errors*** prevent your code from running and generate a MATLAB error message
  - ***Runtime errors*** – not syntactically incorrect, but generate an error upon execution – e.g., indexing beyond matrix dimensions
  - ***Algorithmic errors*** don't prevent your code from executing, but do produce an unintended result
- Syntax and runtime errors are usually more easily fixed than algorithmic errors
- ***Debugging*** – the process of identifying and fixing errors is an important skill to develop
  - MATLAB has a built-in ***debugger*** to facilitate this process

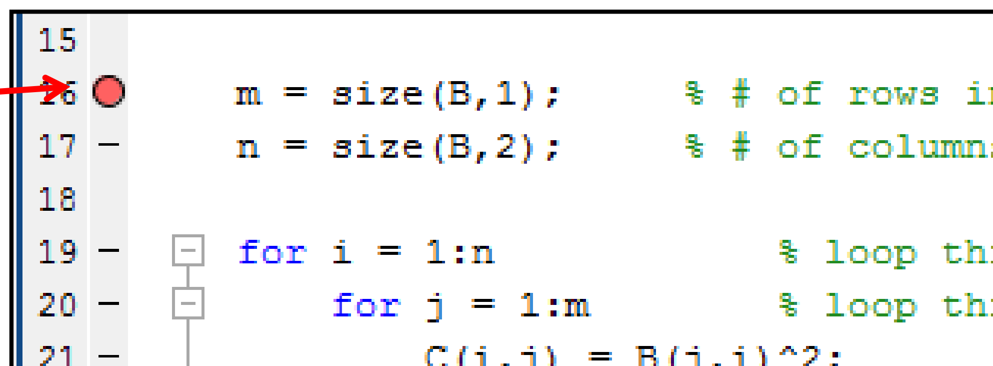
Debugging

- Identifying and fixing errors is difficult because:
  - Programs run seemingly instantaneously
  - Incorrect output results, but can't see the intermediate steps that produced that output
- ***Basic debugging principles***:
  - ***Slow code execution down*** – allow for stepping through line-by-line
  - ***Provide visibility into the code execution*** – allow for monitoring of intermediate steps and variable values

MATLAB Debugger – Breakpoints

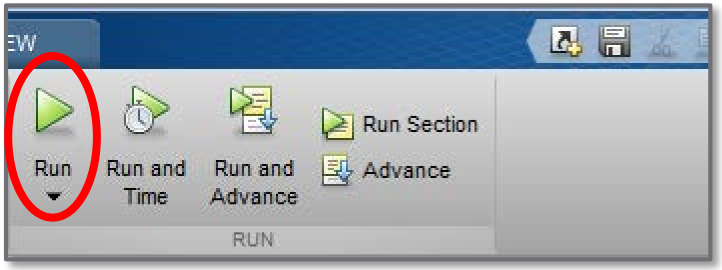
- ***Breakpoint*** – specification of a line of code at which MATLAB should pause execution
- Set by clicking on the dash to the left of a line of code in an m-file
  - MATLAB will execute the m-file *up to* this line, then pause

- Clicking here sets a breakpoint
- Indicated by red circle

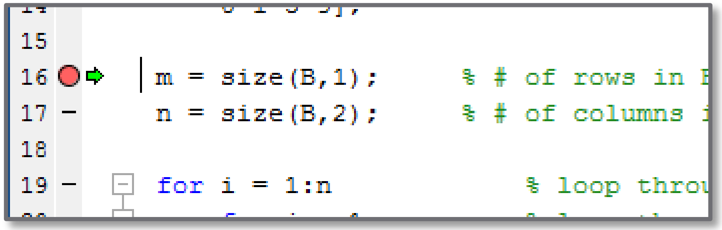


MATLAB Debugger – Breakpoints

- Click Run to begin execution

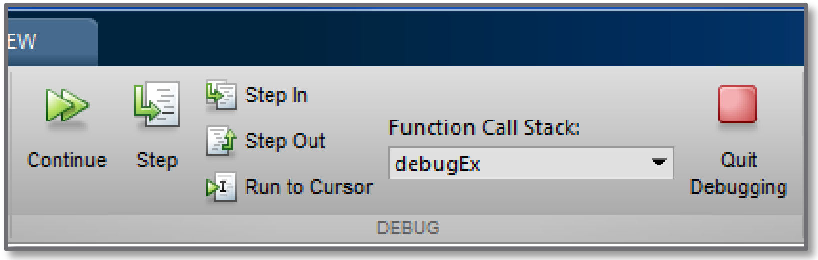


- Execution halts at the breakpoint
  - Before executing that line



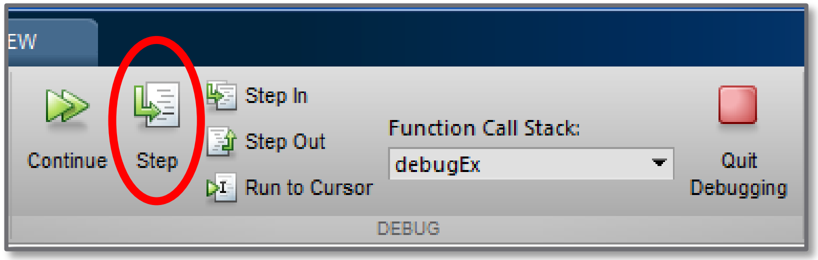
- Command window prompt changes to K>>
  - Can now interactively enter commands

- Toolbar buttons change from RUN to DEBUG

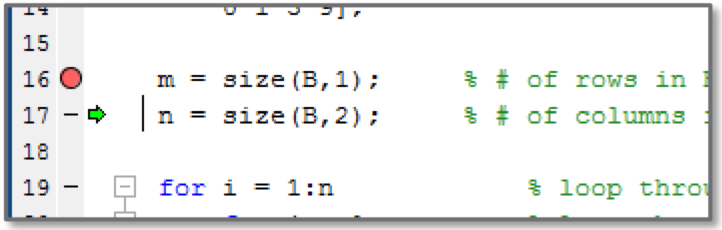


MATLAB Debugger – Breakpoints

- Click Step to execute the current line of code



- Green arrow indicator advances to the next line



- Variable, m, defined on previous line (line 16) is now available in the workspace
    - Can be displayed in the command window

Debugger – Example

- Recall a previous example of an algorithm to square every element in a matrix
- Let’s say we run our m-file and get the following result:

```
% define a matrix, B
B = [1 2 3 4;
     9 7 5 3;
     2 4 6 8;
     8 7 6 5;
     0 1 3 9];

 m = size(B,1);      % # of rows in B
 n = size(B,2);      % # of columns in B

 for i = 1:n             % loop through rows
     for j = 1:m         % loop through columns
         C(i,j) = B(j,i)^2;
     end
 end

 display(B);
 display(C);
```

Command Window

```
B =

     1     2     3     4
     9     7     5     3
     2     4     6     8
     8     7     6     5
     0     1     3     9


C =

     1    81     4    64     0
     4    49    16    49     1
     9    25    36    36     9
    16     9    64    25    81

fx >>
```

- Resulting matrix is ***transposed***
  - Use the ***debugger*** to figure out why

Debugger – Example

- Set a ***breakpoint*** in the innermost `for` loop
- Click ***Run***, code executes through the first iteration of the inner for loop
- Workspace shows `i=1` and `j=1`
- Display `B(i,j)` and `C(i,j)` in the command window
  - Both are as expected

```
m = size(B,1);         % # of rows in B
n = size(B,2);         % # of columns in B

for i = 1:n                    % loop through rows
    for j = 1:m                % loop through colum
        C(i,j) = B(j,i)^2;
    end
end
```

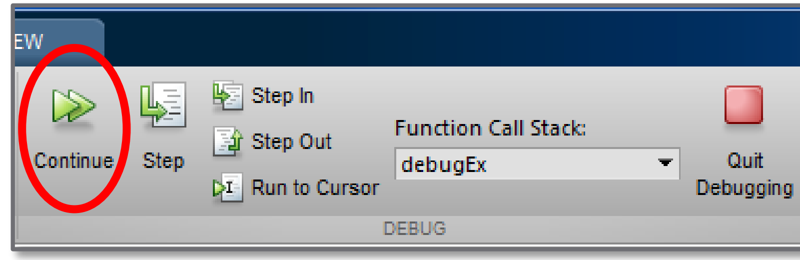
Workspace

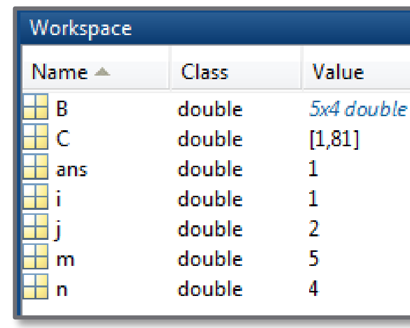
Name	Class	Value
B	double	5x4 double
C	double	1
i	double	1
j	double	1
m	double	5
n	double	4

```
K>> B(i,j)

ans =

     1

K>> C(i,j)

ans =

     1
```

Debugger – Example

- Click ***Continue***, code executes until it hits the breakpoint again
  - One more iteration of inner for loop
- Now, `i=1` and `j=2`
  - First row, second column
- `B(i,j)` = 2, as expected
- But, `C(i,j)` = 81
  - Should be 4





Debugger – Example

- We see that `C(1,2)` is being set to `B(2,1)^2`
- This leads us to an error on line 21 of the code

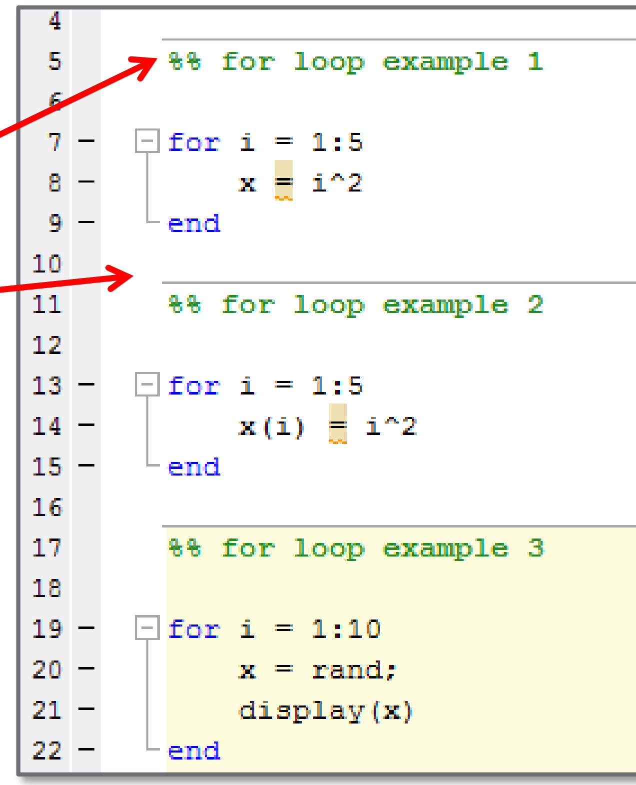
```
 9      % define a matrix, B
10 -    B = [1 2 3 4;
11           9 7 5 3;
12           2 4 6 8;
13           8 7 6 5;
14           0 1 3 9];
15
16 -     m = size(B,1);      % # of rows in B
17 -     n = size(B,2);      % # of columns in B
18
19 -     for i = 1:n                % loop through rows
20 -         for j = 1:m            % loop through columns
21 -             C(i,j) = B(j,i)^2;
22 -         end
23 -     end
24
25 -     display(B);
26 -     display(C);
27
```

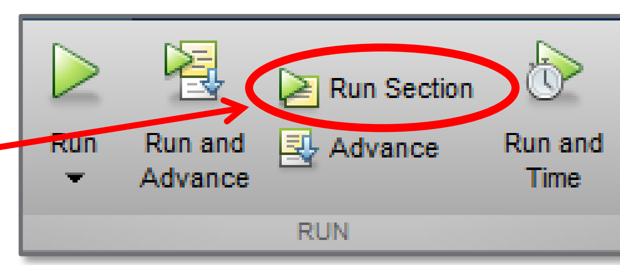
```
Command Window

B =

     1     2     3     4
     9     7     5     3
     2     4     6     8
     8     7     6     5
     0     1     3     9


C =

     1    81     4    64     0
     4    49    16    49     1
     9    25    36    36     9
    16     9    64    25    81

fx >>
```

Miscellany

Sections

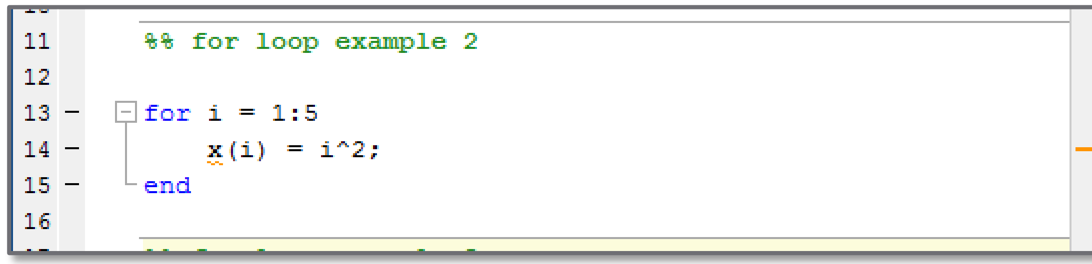
- Define ***sections*** within an m-file
  - Execute isolated blocks of code
  - Starts with a double comment
  - Ends at the start of the next section
  - Useful for debugging, particularly if running the entire m-file is time-consuming
- To run a section:
  - Place cursor in section and type Ctrl+Enter
  - Click the ***Run Section*** button



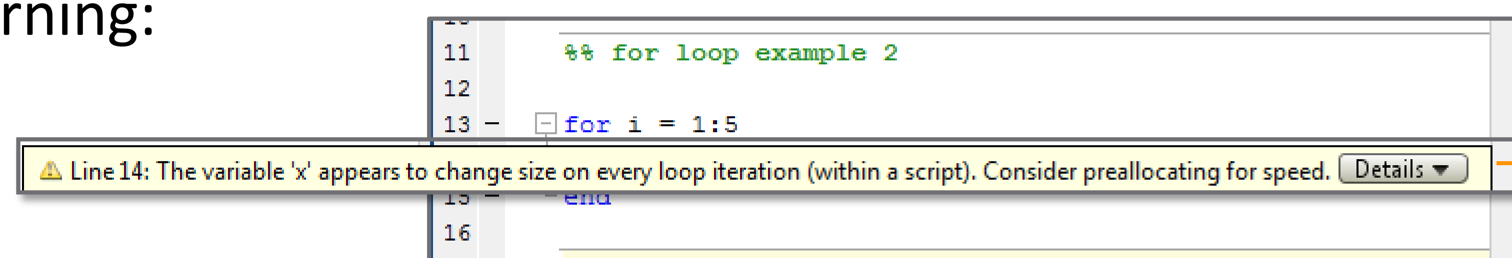


Preallocation

- Note the red line to the right of line 14 and the red squiggle under x in the following for loop:



- Mouse over the line or the squiggle to see the following warning:



- The size of x grows with each iteration of the loop
  - Inefficient - slow

Preallocation

- When you assign a variable, MATLAB must store it in memory
  - Amount of memory allocated for storage depends on the size of the array
  - If the variable grows it must be copied to a new, larger block of available memory – slow
- If the ultimate size of a variable is known ahead of time, we can ***preallocate*** memory for it
  - Assign a full-sized array of all zeros
  - Overwrite elements on each iteration
  - Array size remains constant

Preallocation – Example

- A nested for loop stepping through an $N \times N$ matrix
  - Here N = `100`
- Time the loop with and without preallocation
  - Use `tic` … `toc`
- Preallocation speeds up the loop up significantly
  - But …

```
N = 100;          % dimension of matrix

A = magic(N);     % create an NxN matrix

%% loop through rows and columns,
% creating a matrix of squares
tic                         % start timer

for i = 1:size(A,1)
    for j = 1:size(A,2)
        B(i,j) = A(i,j)^2;
    end
end

toc                         % stop the timer

%% do the same thing, but now preallocate

C = zeros(N);          % an NxN matrix of zeros

tic                    % start timer

for i = 1:size(A,1)
    for j = 1:size(A,2)
        C(i,j) = A(i,j)^2;
    end
end

toc                    % stop the timer
```

Command Window

```
Elapsed time is 0.008711 seconds.
Elapsed time is 0.005607 seconds.
>>
```

Preallocation – Example

- An accurate comparison must account for the cost of preallocation
  - Start the timer before preallocating
- Still significantly faster, even accounting for preallocation
  - Note that times vary from run to run
  - But …

```
N = 100;          % dimension of matrix

A = magic(N);     % create an NxN matrix

%% loop through rows and columns,
% creating a matrix of squares
tic                        % start timer

for i = 1:size(A,1)
    for j = 1:size(A,2)
        B(i,j) = A(i,j)^2;
    end
end

toc                        % stop the timer

%% do the same thing, but now preallocate

tic                    % start timer

C = zeros(N);          % an NxN matrix of zeros

for i = 1:size(A,1)
    for j = 1:size(A,2)
        C(i,j) = A(i,j)^2;
    end
end

toc                    % stop the timer
```

Command Window

```
Elapsed time is 0.008699 seconds.
Elapsed time is 0.005763 seconds.
>>
```

Preallocation – Example

- 6 msec vs. 9 msec? So what?
  - Difference is imperceptible
- Now, increase N to 5e3
  - 25e6 elements in A!
  - A significant, and very noticeable, difference
  - ***Preallocation*** is **always** a good practice

```
N = 5e3;            % dimension of matrix

A = magic(N);       % create an NxN matrix

%% loop through rows and columns,
% creating a matrix of squares
tic                         % start timer

for i = 1:size(A,1)
    for j = 1:size(A,2)
        B(i,j) = A(i,j)^2;
    end
end

toc                         % stop the timer

%% do the same thing, but now preallocate

tic                     % start timer

C = zeros(N);           % an NxN matrix of zeros

for i = 1:size(A,1)
    for j = 1:size(A,2)
        C(i,j) = A(i,j)^2;
    end
end

toc                     % stop the timer
```

Command Window

```
Elapsed time is 76.802444 seconds.
Elapsed time is 14.043226 seconds.
>>
```