for loop

• used as a **counted** loop
• repeats an **action** a specified number of times
• an **iterator** or loop variable specifies how many times to repeat the action
• general form:
  
  ```
  for loopvar = range
      Action
  end
  ```
• the range is specified by a vector
• the action is repeated for every value of the loop variable in the specified vector
for loop examples

- **Loop that uses the iterator variable:**
  ```matlab
  >> for i = 1:3
      fprintf('i is %d\n', i)
  end
  i is 1
  i is 2
  i is 3
  ```

- **Loop that does not use the iterator variable:**
  ```matlab
  >> for i = 1:3
      disp('Howdy')
  end
  Howdy
  Howdy
  Howdy
  ```
Input in a for loop

- If it is desired to repeat the process of prompting the user and reading input a specified number of times (N), a `for` loop is used:
  ```matlab
  for i = 1:N
      % prompt and read in a value
      % do something with it!
  end
  ```

- If it is desired to store the values entered in a vector, the most efficient method is to preallocate the vector first to have N elements.
Preallocating a Vector

- Preallocating sets aside enough memory for a vector to be stored
- The alternative, extending a vector, is very inefficient because it requires finding new memory and copying values every time
- Many functions can be used to preallocate, although it is common to use `zeros`
- For example, to preallocate a vector `vec` to have N elements:
  
  ```
  vec = zeros(1,N);
  ```
for loop uses

- calculate a sum
  - initialize *running sum* variable to zero
- calculate a product
  - initialize *running product* variable to one
- input from user
  - can then *echo print* the input
- sum values in a vector
  - can also use built-in function *sum* for this
- other functions that operate on vectors: *prod*, *cumsum*, *cumprod*, *min*, *max*, *cummin*, *cummax*
For loop application: subplot

- The `subplot` function creates a matrix (or vector) in a Figure Window so that multiple plots can be viewed at once.
- If the matrix is $m \times n$, the function call `subplot(m,n,i)` refers to element $i$ (which must be an integer in the range from 1 to $m \times n$).
- The elements in the FW are numbered row-wise.
- It is sometimes possible to use a `for` loop to iterate through the elements in the Figure Window.
Subplot Example

For example, if the subplot matrix is 2 x 2, it may be possible to loop through the 4 elements to produce the 4 separate plots

\[
\begin{array}{c|c}
\text{Plot 1} & \text{Plot 2} \\
\hline
\text{Plot 3} & \text{Plot 4} \\
\end{array}
\]

for i = 1:4
    subplot(2,2,i)
    % create plot i
end
A nested for loop is one inside of (as the action of) another for loop.

General form of a nested for loop:

```plaintext
for loopvarone = rangeone  
% actionone:
  for loopvartwo = rangetwo  
    actiontwo
  end
end
```

The inner loop action is executed in its entirety for every value of the outer loop variable.
Combining for loops and if

- for loops and if statements can be combined
  - the action of a loop can include an if statement
  - the action of an if statement can include a for loop
- This is also true for nested for loops; if statements can be part of the action(s) of the outer and/or inner loops
- This is done if an action is required on an element (of a vector or matrix) only if a condition is met
while loop

- used as a conditional loop
- used to repeat an action when ahead of time it is not known how many times the action will be repeated
- general form:
  while condition
  action
  end
- the action is repeated as long as the condition is true
- an *infinite loop* can occur if the condition never becomes false (Use Ctrl-C to break out of an infinite loop)
- Note: since the condition comes before the action, it is possible that the condition will be false the first time it is evaluated and therefore the action will not be executed at all
Counting in a while loop

- it is frequently useful to count how many times the action of the loop has been repeated
- general form of a while loop that counts:

```plaintext
counter = 0;
while condition
  % action
  counter = counter + 1;
end
% use counter – do something with it!
```
while loop application: error-checking

- with most user input, there is a valid range of values
- a `while` loop can be used to keep prompting the user, reading the value, and checking it, until the user enters a value that is in the correct range
- this is called *error-checking*
- general form of a while loop that error-checks:
  
  ```
  prompt user and input value
  while value is not in correct range
    print error message
    prompt user and input value
  end
  use value
  ```
Example: Prompt for radius

```plaintext
radius = input('Enter the radius of a circle: ');
while radius <= 0
    radius = input('Invalid! Enter a positive radius: ');
end
area = pi * radius ^ 2;
fprintf('The area is %.2f\n', area)
```
While loop example

- What is desired is a script “ch5pp” that will prompt the user for a quiz grade and error-check until the user enters a valid quiz grade. The script will then echo print the grade. For this course, valid grades are in the range from 0 to 10 in steps of 0.5. Following are examples of executing the script.
- Method: create a vector of valid grades and then do 3 solutions: using any, all, and find.

```plaintext
>> ch5pp
Valid quiz grades are in the range from 0 to 10 in steps of 0.5
Enter a quiz grade: 4.5
Cool, the grade is 4.5
>> ch5pp
Valid quiz grades are in the range from 0 to 10 in steps of 0.5
Enter a quiz grade: -2
Invalid! Enter a quiz grade: .6
Invalid! Enter a quiz grade: .499
Invalid! Enter a quiz grade: 9.5
Cool, the grade is 9.5
```
Example Solution I

```matlab
fprintf('Valid quiz grades are in the range from')
fprintf('0 to 10 in steps of 0.5\n')
validvec = 0:0.5:10;
quiz = input('Enter a quiz grade: ');
while ~any(quiz==validvec)
    quiz = input('Invalid! Enter a quiz grade: ');
end
fprintf('Cool, the grade is %.1f\n', quiz)
```
fprintf('Valid quiz grades are in the range from')
fprintf('0 to 10 in steps of 0.5
')
validvec = 0:0.5:10;
quiz = input('Enter a quiz grade: ');
while all(quiz~=validvec)
    quiz = input('Invalid! Enter a quiz grade: ');
end
fprintf('Cool, the grade is %.1f
', quiz)
Example Solution III

```matlab
fprintf('Valid quiz grades are in the range from ')
fprintf('0 to 10 in steps of 0.5\n')
validvec = 0:0.5:10;
quiz = input('Enter a quiz grade: ');
while isempty(find(validvec==quiz))
    quiz = input('Invalid! Enter a quiz grade: ');
end
fprintf('Cool, the grade is %.1f\n', quiz)
```
Error-Checking for Integers

- To error-check for integers, you can input into a variable and then either round that value or use an integer type function (e.g. `int32`) to convert the number that was entered to an integer.
- If the number that was entered originally was an integer, then rounding or converting will have no effect; the values will be the same

```matlab
inputnum = input('Enter an integer: ');
um2 = int32(inputnum);
while num2 ~= inputnum
    inputnum = input('Invalid! Enter an integer: ');
    num2 = int32(inputnum);
end
```
for loops and vectors

- **for** loops can be used to accomplish the same task for every element in a vector.

- **general form of** **for** loop that iterates through a vector:
  
  ```
  for i = 1:length(vectorvariable)
      do something with vectorvariable(i)
  end
  ```

- If the purpose of the loop is to create a vector variable, it is much more efficient to **preallocate** the variable before the loop (note: the length must be known).
Nested for loops and matrices

- *nested for* loops can be used to accomplish the same task for every element in a matrix.
- One loop is over the rows, and the other is over the columns.
- General form of nested *for* loop that iterates through a matrix:
  
  ```matlab
  [r c] = size(matrixvariable)
  for row = 1:r
      for col = 1:c
          do something with matrixvariable(row,col)
      end
  end
  ```

- Note: this nested loop iterates through the matrix row-by-row; by reversing the for statements it would instead iterate column-by-column.
Use MATLAB wisely!!

- Using **for** loops with vectors and matrices is a very important programming concept, and is necessary when working with many languages.
- However... Although **for** loops are very useful in MATLAB (e.g., for the `subplot` function), they are almost **NEVER** necessary when performing an operation on every element in a vector or matrix!
- This is because MATLAB is written to work with matrices (and therefore also vectors), so functions on matrices and operations on matrices automatically iterate through all elements – no loops needed!
Vectorizing

- The term *vectorizing* is used in MATLAB for re-writing code using loops in a traditional programming language to matrix operations in MATLAB.
- For example, instead of looping through all elements in a vector `vec` to add 3 to each element, just use scalar addition:

  ```matlab
  vec = vec + 3;
  ```
Efficient Code

- In most cases, code that is faster for the programmer to write in MATLAB is also faster for MATLAB to execute.
- Keep in mind these important features:
  - Scalar and array operations
  - Logical vectors
  - Built-in functions
  - Preallocation of vectors
Preallocation Question

- Preallocation can speed up code, but in order to preallocate it is necessary to know the desired size. What if you do not know the eventual size of a vector (or matrix)? Does that mean that you have to extend it rather than preallocating?
Preallocation Answer

- If you know the maximum size that it could possibly be, you can preallocate to a size that is larger than necessary, and then delete the “unused” elements. In order to do that, you would have to count the number of elements that are actually used. For example, if you have a vector vec that has been preallocated, and a variable count that stores the number of elements that were actually used, this will trim the unnecessary elements:
  - vec = vec(1:count)
Operations on Vectors & Matrices

- Can perform numerical operations on vectors and matrices, e.g. vec + 3
- Scalar operations e.g. mat * 3
- Array operators operate term-by-term or element-by-element, so must be same size
- Addition + and subtraction -
- Array operators for any operation based on multiplication require dot in front .* / \ .^
Useful Efficient functions

- Keep in mind these useful functions:
  - `sum`, `prod`, `cumsum`, `cumprod`, `min`, `max`
  - `any`, `all`, `find`
  - `diff`
  - “is” functions including `isequal`

- `checkcode`: can check code in both scripts and functions for inefficiencies; same as information in Code Analyzer Reports
Timing Code

- The functions `tic` and `toc` are used to time code
  - Be careful; other processes running in the background will have an effect so should run multiple times and average
    ```matlab
    >> type fortictoc
    
    tic
    mysum = 0;
    for i = 1:20000000
        mysum = mysum + i;
    end
    toc
    
    >> fortictoc
    Elapsed time is 0.090699 seconds.
    >>
    ```
- There is also a Profiler that will generate detailed reports on execution times of codes
Common Pitfalls

- Forgetting to initialize a running sum or count variable to 0 or a running product to 1
- Not realizing that it is possible that the action of a `while` loop will never be executed
- Not error-checking input into a program
- Forgetting that `subplot` numbers the plots rowwise rather than columnwise.
- Not taking advantage of MATLAB; not vectorizing!
Programming Style Guidelines

• Use loops for repetition only when necessary
  • \texttt{for} statements as counted loops
  • \texttt{while} statements as conditional loops
• Do not use $i$ or $j$ for iterator variable names if the use of the built-in constants \texttt{i} and \texttt{j} is desired.
• Indent the action of loops
• Preallocate vectors and matrices whenever possible (when the size is known ahead of time).
• If the loop variable is just being used to specify how many times the action of the loop is to be executed, use the colon operator $1:n$