



Selection Statements

Chapter 4



If Statement

- The **if** statement is used to determine whether or not a statement or group of statements is to be executed
- General form:

```
if condition
    action
end
```
- the *condition* is any relational expression
- the *action* is any number of valid statements (including, possibly, just one)
- if the condition is true, the action is executed – otherwise, it is skipped entirely

Representing true/false concepts

- Note: to represent the concept of false, 0 is used. To represent the concept of true, any nonzero value can be used – so expressions like 5 or ‘x’ result in logical true
- This can lead to some common logical errors
- For example, the following expressions are always true (because the “relational expressions” on the right, 6 and ‘N’, are nonzero so they are true; therefore, it does not matter what the results of the others are):

```
number < 5 || 6
```

```
letter == 'n' || 'N'
```

If-else Statements

- The **if-else** statement chooses between two actions
- General form:

```
if condition
    action1
else
    action2
end
```
- One and only one action is executed; which one depends on the value of the condition (action₁ if it is logical true or action₂ if it is false)

Throwing an error

- MATLAB has an **error** function that can be used to display an error message in red, similar to the error messages generated by MATLAB

```
if radius <= 0  
    error('Sorry; %.2f is not a valid radius\n', radius)  
else  
    % carry on  
end
```

Nested if-else Statements

- To choose from more than two actions, *nested if-else* statements can be used (an **if** or **if-else** statement as the action of another)

- General form:

```
    if condition1
        action1
    else
        if condition2
            action2
        else
            if condition3
                action3
            % etc: there can be many of these
        else
            actionn % the nth action
        end
    end
end
end
```

The **elseif** clause

- MATLAB also has an **elseif** clause which shortens the code (and cuts down on the number of ends)

- General form:

```
if condition1
    action1
elseif condition2
    action2
elseif condition3
    action3
% etc: there can be many of these
else
    actionn % the nth action
end
```

The switch Statement

- The **switch** statement can frequently be used in place of a nested **if** statement
- General form:

```
switch switch_expression
  case caseexp1
    action1
  case caseexp2
    action2
  case caseexp3
    action3
  % etc: there can be many of these
  otherwise
    actionn
end
```

- this can be used when comparing the `switch_expression` to see if it is equal to the values on the case labels (the otherwise clause handles all other possible values)

The “is” functions

- There are many “is” functions in MATLAB that essentially ask a true/false question, and return logical 1 for true or 0 for false
- **isletter** returns 1 or 0 for every character in a string – whether it is a letter of the alphabet or not
- **isempty** returns 1 if the variable argument is empty, or 0 if not
- **iskeyword** returns 1 if the string argument is a keyword, or 0 if not
- **isa** determines whether the first argument is a specified type

Programming Style Guidelines

- Use indentation to show the structure of a script or function. In particular, the actions in an if statement should be indented.
- When the else clause isn't needed, use an if statement rather than an if-else statement



Loop Statements & Vectorizing Code

Chapter 5



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:

```
>> 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:

```
>> for i = 1:3
    disp('Howdy')
end
Howdy
Howdy
Howdy
```

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 $m*n$)
- 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

Plot 1	Plot 2
Plot 3	Plot 4

```
for i = 1:4
    subplot(2,2,i)
    % create plot i
end
```

Nested for loops

- A nested **for** loop is one inside of (as the action of) another **for** loop
- General form of a nested **for** loop:

```
for loopvarone = rangeone ← outer loop
    % actionone:
    for loopvartwo = rangetwo ← inner loop
        actiontwo
    end
end
end
```
- The inner loop action is executed in its entirety for every value of the outer loop variable

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

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

```
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 (*Practice 5.6*)

% Error checks until the user enters n positive integers

n = 4;

for i = 1:n

inputnum = input('Enter a positive integer: ');

num2 = int32(inputnum);

while num2 ~= inputnum || num2 < 0

inputnum = input('Invalid! Enter a positive integer: ');

num2 = int32(inputnum);

end

fprintf('Thanks, you entered a %d \n',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:

```
[r c] = size(matrixvariable)
for row = 1:r
    for col = 1:c
        do something with matrixvariable(row,col)
    end
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:

```
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

```
>> 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
 - for statements as counted loops
 - while statements as conditional loops
- Do not use *i* or *j* for iterator variable names if the use of the built-in constants **i** and **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

Exercises

- Write a **for** loop that will print a column of five *'s.
- Write a function *mymatmin* that finds the minimum value in each column of a matrix argument and returns a vector of the column minimums.
- Write a script *avenegnum* that will repeat the process of prompting the user for negative numbers, until the user enters a zero or positive number. Instead of echoing them, however, the script will print the average (of just the negative numbers). If no negative numbers are entered, the script will print an error message instead of the average.

Exercises

- Write a function that imitates the **cumprod** function. Use the method of preallocating the output vector. (Hint: use `help cumprod` first).
- Create a function *matrowsum* to calculate and return a vector of all of the row sums of a matrix, instead of column sums (**sum** function in Matlab returns the column sums)
- Implement vectorized versions of the previous functions and scripts that includes **tic toc** commands to test the efficiency of the code.