**Conditional Statements with Multiple Branches** (please develop programs individually)

1. **Background: Multiple Branching and the if/elseif/else structure**

Many times we need more than two alternatives in a branching program. Multiple branching can easily be added to the if/else structure by using ***elseif*** commands. The material below outlines the syntax and use of this structure. Please review it carefully

if/elseif/else structure

1. English Example: If *x is greater than zero*

then  *y equals the square root of x\_*

or else if *y is equal to zero*

then *display “x=0” and set y = 0*

or else *display “input is negative” and set y = NaN*

1. MATLAB Example

if x > 0

y = sqrt(x);

elseif x == 0

disp(‘input is zero’)

y = 0;

else

disp(‘Error: input is negative’)

y = NaN;

end

***elseif*** condition is evaluated only if the previous condition is false.

***else*** statements are only executed if all previous conditions are false. Here it is being used for “error trapping”.

y is set equal to NaN to insure it exists for later calculations.



1. MATLAB general structure & Flowchart:

if *logical statement 1*

*Statement Group 1 (executed if logical statement 1 true)*

elseif *logical statement 2*

*Statement Group 2 (executed if logical statement 1 is false & 2 is true)*

else

*Statement Group 3 (executed if logical statements 1 & 2 are both false)*

end

**Some important notes on *else* and *elseif*** (Do not skip this section, come back later and read it again)

* A structure may have many ***elseif*** statements. Their conditions will be evaluated only if the ***if*** condition and all preceding ***elseif*** conditions have all been false.
* There can be only one ***else*** statement and it must come after all ***elseif*** statements. The ***else*** statement never gets a condition
* There is a big difference between ***elseif*** and ***else if*** (with a space in the middle). In general these type of structured programming statements must be used in exactly the correct form.

1. **Introductory Programming Exercise: The Leap Day Function**

Being able to branch in a program based on a condition is one of the most important structures in programming. This example will be a chance to use the *if – elseif- else - end* structure that has been introduced. Figure 1 gives the setup portion of the Program Development Worksheet for a function to determine if it is a leap year or not. Develop and validate a command line function to go with this setup. It is critical to follow the setup exactly (e.g., command line I/O only, output is a one or a zero). Include full comments (i.e., introductory help section, variable list and logic comments). The validation must use the provided test cases (notice that the test cases check all branches of the code, this is essential).

Set Up/ Introduction Type of Program:  Script 🗹 Function

1. Problem Statement:

Develop a function that given the year will return a 1 if it is a leap year and 0 if it is not.

1. Inputs: (full name, variable to be used, units)

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Description | Units or Values | Input Source\* |
| year | the Gregorian calendar year | none | command line |

1. Output: (full name, variable to be used, units)

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Description | Units or Values | Output type\* |
| day | indicator of a leap year | 1 if a leap year  0 if not | command line |

1. Solution Steps (order of these two parts may be varied):

Perform calculation on test case(s) Identify the steps/equations

|  |  |
| --- | --- |
| **year (input)** | **day (output)** |
| 2000 | 1 |
| 1900 | 0 |
| 2008 | 1 |
| 2009 | 0 |

Steps: Flow Chart (developed earlier)

Test Cases & Correct Results 🡺

**Figure 1:** Setup for the program to determine if a given year is a leap year or not. Notice this is to be a command line only function with no interactive I/O. The output should be simply the binary result of one or zero. The flowchart for this code was developed in class.

**Help Function Peer Feedback (required):** Have a neighboring student run a help query on the function you have developed and give feedback on the help file (i.e., at the command line have them type “help” space and then your functions name). Have them read over and comment on the help that appears in the command window. Improve the help file based on this feedback and add an acknowledgement to your program comments for the student who gave you the feedback.

**To Turn In:**  • Turn in a copy of the Leap Day program and its validation including

* Code that exactly matches setup I/O requirements in Figure 1.
* full comments with the requested review of help file and acknowledgement, and
* validation with the four provided test cases in Figure 1.

1. **Background: The Switch-Case Structure**

A switch-case structure is a special conditional structure that handles the common case where it is helpful to “switch” between various “cases” in a list. For this command, relational operators are not used; it can only switch based on an exact match between the switch value and a case value. The idea of this structure can be written out in English as:

switch what you do based on the value of *\_\_x\_\_\_*

for the case where *x* equals \_\_\_\_\_\_\_ do …

for the case where *x* equals \_\_\_\_\_\_\_ do …

Figure 2 shows the syntax, flowchart, and an example of this structure in MATLAB.

**Syntax**  **Flowchart**



switch *variable or expression*

case *value*

*statements*

case *value*

*statements*

…

otherwise

*statements*

end

**Example:** New Pig Latin Function with switch:

function x = eng2PL1s(x)

% function x = eng2PL1(x)

% lots ‘o’ intro comments

% make all letters lower case

x=lower(x);

% Check for vowels, V will equal 1 if vowel

V = any(x(1)=='aeiouy')

switch V

case 1

% if x starts in a vowel add 'ay'

x = [x,'ay'];

case 0

% otherwise move consonant & add 'ay'

x = [x(2:end),x(1),'ay'];

end

**Figure 2:** The nature of the switch-case structure in MATLAB. The general syntax, flowchart and an example are shown. The switch-case structure is only testing if the value after the switch is equal to one of the case values. It cannot do greater than or less than comparisons directly.

This structure was designed to be particularly well suited to switching actions based on choices made from a menu.

1. **Program Exercise – Static Friction Function (applying the switch case structure)**

The static friction problem setup shown in Figure 3 is a menu-type problem that fits the switch-case structure logic (see section III). It is intended to calculate the static friction for a series of scenarios. Static friction is the force required to start an object moving; once the object is moving the sliding friction force is less than the static friction. This is the reason that objects are hard to start sliding and then sometimes slide too much. It is also why you want to avoid skidding to a stop in a car.

Set Up/ Introduction Type of Program:  Script ☑ Function

1. Problem Statement:

The force (F) required to start a weight (w) moving is given by: F =μw where F is the force (lbf), w is the weight in (lbf) and μ is the static coefficient of friction (lbf/lbm). This coefficient is given for several materials below:

Metal on metal μ = 0.20,

Wood on wood μ = 0.35,

Metal on wood μ = 0.40,

Rubber on concrete μ = 0.70.

Develop a program that calculates the force required to overcome static friction. User specifies the weight of the object and selects one of these four combinations of materials.

1. Inputs: (full name, variable to be used, units)

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Description | Units or Values | Input type\* |
| W | weight of object | lbf | Command line |
| M | material combination | 1 = metal on metal  2 = wood on wood  3 = metal on wood  4 = Rubber on concrete | Command line |

\* Possible types: command line, file, interactive input

1. Output: (full name, variable to be used, units)

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Description | Units or Values | Output type\* |
| F | Force required to overcome static friction | lbf | Command line |

\* Possible types: command line, file, display, figure window, speaker

1. Solution Steps (order of these two parts may be varied):

Perform calculation on test case(s) Identify the steps/equations

Example Test Case: metal on wood, 12 lbf

🡺 μ = 0.40, F = (12 lbm)(0.40 lbf/lbm) = 4.8 lbf

Steps: 1. Input material combination number and weight from the command line

2. Select correct coefficient of static friction (μ)

3. Calculate the force required to overcome static friction F = μw

4. Output the static frictional force at the command line

**Figure 3:** Shows the setup for the static friction command line function. This problem is based on a problem from Palm, *A Concise Introduction to MATLAB*, McGraw-Hill, 2008. Notice that this is again a function with I/O at the command line and that the input for the material is a number (i.e., 1, 2, 3 or 4).

**Tasks:** Using a switch-case structure (see section III):

1. Draw a flowchart by hand to figure out the logic for your program (build on the steps in Figure 3).

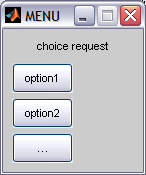
2. Develop and validate this function (reminder: all conditional branches must be verified).

**Show the result of these tasks to your instructor.**

1. **Program Exercise – Static Friction Interactive Function**

For this exercise the previous static friction function will be updated to be an interactive function. Functions that have been used so far in the course are all command line I/O plus an occasional figure window. Previously interactive scripts have been used. Sometimes the interactivity is desired without the program using the base workspace. For this case an interactive function can be used. To accomplish this we will learn a few new MATLAB commands.

* + **Warning this step is critical:**  Make a copy of your previous static friction function. Be sure and change both the file name and the name in the function definition line (so they are different from the original function but match each other). Do not change the original function; both versions of the code will be helpful later.



* + **Eliminate IO from the function definition line.** There will still be a function definition line, but it will simply be the word “function” and the name of the interactive function
  + **Input Menu:** To input the material type a small pop up menu box can be used. This function will create a small push button Graphical User Interface (GUI which is pronounced ‘gooey’).

**Form:** >> x = menu(‘choice request’, ‘option1’, ‘option2’, …)

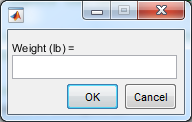
**Result:** This function results in the small pop up shown in Figure 4

**Use:** *Pressing a button will output its position in the list (1, 2, …)* to the output variable (x in the form example above).

**Figure 4:** The pop up GUI launched by the menu command x = menu(‘choice request’, ‘option1’, ‘option2’, ‘…’)

* Add this menu command to the new copy of the static friction program. Do ask for help as needed on the menu command. The buttons should be labeled with the exact material combinations (e.g., ‘option 1’ in the command above should be replaced with ‘metal on metal’)
  + **Interactive Input of Weight:** The second input to the function is the weight of the object. The *input* function used in scripts could be used here to interactively input the weight. But it is more interesting (and fun) to input them with a dialogue box. This can be done using the *Inputdlg* function described below. Do follow all the instructions carefully.

*Inputdlg* function:



If the following code is executed in MATLAB the dialog box in Figure 5 will appear.

x = inputdlg( 'Weight (lb) = ' )

The user enters a value and clicks on OK to transfer the value to the x variable in the above example.

**The following step is critical. Without it this approach will not work!** Unfortunately, the value is stored in x as text and in what is called a “cell array”. To convert this to a simple numerical variable use the code below.

**Figure 5:** The dialog box produced by *inputdlg*. User enters the value and clicks on the OK button.

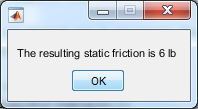
x = str2num(x{1})

The curly brackets around the index “1” will remove the x value from the cell array, and the str2num function will convert the text variable to a number.

* **Output Static Friction in a Dialogue Box**: Again, it would be possible to display the resulting friction with the *disp* function. However, in this case try out MATLAB’s *msgbox* function.

*msgbox* function

If the following code is executed in MATLAB, the message box shown in Figure 6 will display (for the variable y containing the calculated frictional force).



msgbox( [ 'The resulting static friction is ', num2str(y), ' lb' ] )

Notice the argument for *msgbox* in the parentheses. The argument is also enclosed in square brackets to combine the three text strings into one string as an input to the function. The *num2str* function is used to convert the desired data to a string variable so that it can be combined with the two strings contained in literals.

**Figure 6:** The message box produced by *msgbox*. The user clicks on the OK button to dismiss the box.

This should yield a program that is completely interactive using pop up boxes.

1. **Background Exercise: Input Arguments**

This exercise uses the ***addme*** function available on the course website. It is best done and discussed with another student.

Run the ***addme*** function 1. first with two inputs (i.e., *addme*(1,2)),   
 2. with only one input value (i.e., *addme*(2), and   
 3. with no input values (e.g., ***addme***). Look at the code.

1. What does *addme* do in each case?
2. What is the ***nargin*** variable?
3. What might the variable name (***nargin***) be short for (hint: three words)?
4. **For the Road:**  Get started this lab, **Turn in next lab**

Static Friction Combined: Create a function that combines both approaches to the problem, choosing which approach based on the number of arguments the user includes when calling the function.

**Program Requirements:** The goal is to create a function that will calculate the static friction as above but adjust to how the user supplies the inputs

* + 1. If two input arguments are provided, then only command line I/O is used, (i.e., have input and output only on the command line with no other user interaction),
    2. If no input arguments are provided, then all inputs should be interactive.
    3. If some other number of input arguments are provided, then an error message should display.
  + **Determining the arguments used:** Use nargin (investigated in the last section) in the condition of an if statement to change the response to different numbers of inputs.
  + **Two Strategies:** Below are two general ways you can accomplish this switching between the two programs. Both have their own advantages and disadvantages. Pick one and try it.
    1. **Main Program and Subroutines:**  This is probably the simplest to figure out. Write a new ***main program*** that determines the number of arguments and then (using an if or switch structure) calls the appropriate previous program. For two inputs, call the command line program (from section IV) and transfer the inputs to it and the output from that program to the output variable for the ***main program*** you are creating. For no inputs call the interactive program (from section V). Flowcharts for all three programs would be required.
    2. **Adapt one of the previous programs:**  Start with say the command line program. Add a section that detects the number of inputs: 1) uses the values given (if two are given), or 2) runs the interactive input functions if no inputs are given or 3) displays an error message if another number of inputs is given. Once the inputs are obtained the calculation part of the program can be the same for either case. This is likely the more elegant option with potentially less redundant code.

**Prepare a complete Program Development Worksheet** (setup, code and validation) for this program:

* The variable setup will include two options for input type.
* The program steps must be presented as a flowchart (created in Visio).   
  For strategy 1 this is best handled with three smaller flow charts: 1. A flowchart of the main program that calls the two subroutines 2. A flowchart for the command line subfunction and 3. A flowchart for the interactive program.   
  For strategy 2, a longer flowchart will result that may take a couple of pages.
* Code must include full comments including a useful help section, a variable list and comments on logic steps.
* Validation must include validation of all program paths (interactive case may need to be described). Alt-Print Scrn can also be used to copy and paste the dialogue boxes.

**See Conditional Summary, and Rubric Online**