Lecture 3 - Functions and Control Structures

Range variables vs. vectors:
Continuing our discussion of using range vectors vs. using vectors. Recall our general rule will be to use range variables to create and reference vectors, and vectors will be used to store our information for manipulation (equations etc.) and graphing.

Mail Box Analogy: Think of the process of storing data like putting stuff into a series of mailboxes. Not only do we need to keep track of what we put into the mailboxes, but we also need to keep track of where the mailboxes are. Range variables will be used to point at mailbox locations. Vectors will be used to represent the contents of the mailboxes. If I want only the stuff stored in the 5th mailbox, I will point to that mailbox to retrieve its contents. This is done using an index as in \( x_5 \).

IMPORTANT: An index to an array must be an integer. The index represents a discrete location in memory. There is no 3.4th mailbox on the block, and there is no 3.4th spot in a vector. The contents of that discrete spot can be a non-integer, a word, etc., but its location (the index, the subscript) must be an integer.

Range Variables:
Indexing an array (pointing to a spot) can be done one spot at a time (with a constant or scalar variable), or multiple spots at one time (with a range variable). A range variable can be used to point to multiple locations at one time.

- Use an index to reference specific locations within an array
- e.g. \( i := 4 \) Grade\(_i\) OR JUST Grade\(_4\) refers to 89.3 in the vector
- The index can be a number, a scalar variable, or a range variable. In any case the index must be an integer, as it is pointing to one or more discretely numbered locations (mailboxes)

Example: Indexing specific spots within a list of grandes:

\[
\text{Grade := [56 72 65 98 84 91]};
\]

I can use a range variable to refer to only the first 3 spots, or some interior spots:

\[
\begin{align*}
\text{i := 1..3} & \quad \text{Grade}_{i} = & \quad \text{i := 3..5} & \quad \text{Grade}_{i} = & \quad \text{i := 1,3..5} & \quad \text{Grade}_{i} = \\
56 & \quad 72 & \quad 65 & \quad 65 & \quad 98 & \quad 84 & \quad 56 & \quad 65 & \quad 84
\end{align*}
\]
Example: Using a range variable to calculate the mean value of the same list of grades:

\[
\sum_{i} \text{Grade}_i \quad \text{The summation comes from the keystroke $\$}\n\]

\[
i := 1..6 \quad \text{average} := \frac{\sum_{i} \text{Grade}_i}{\text{length(Grade)}} \quad \text{average} = 77.667
\]

Example: Using a range variable to calculate the mean value of portions of the list of grades:

\[
\sum_{i} \text{Grade}_i \quad \text{average} := \frac{i}{3} \quad \text{average} = 64.333 \quad \sum_{i} \text{Grade}_i \quad \text{average} := \frac{i}{3} \quad \text{average} = 68.333
\]

Example: Take the summation of a complete vector to find an average (different kinda summation)

\[
\text{average} := \frac{\sum \text{Grade}}{\text{length(Grade)}} \quad \text{average} = 77.667 \quad \text{This summation (Ctrl 4) sums the entire vector without need of a range variable}
\]

Example: Multiplying two different vectors together by default gives the scalar dot-product as a result. If we wish to multiply element by element, use the range variable idea. Here we only want the first 3 values of Inertia. So we use a range variable with values 1, 2 and 3.

\[
b := \begin{pmatrix} 3 \\ 9 \\ 2 \\ 4 \end{pmatrix} \quad h := \begin{pmatrix} 5 \\ 1 \\ 8 \\ 2 \end{pmatrix} \quad \frac{b \cdot h^3}{12} = 120 \quad i := 1..3 \quad I_i := \frac{b_i \cdot (h_i)^3}{12} \quad I = \begin{pmatrix} 31.25 \\ 0.75 \\ 85.333 \end{pmatrix} \quad \text{Dot Product}
\]
Functions

We had a brief look at creating functions in the previous lecture. These functions will be a necessary part of learning and using the three basic control structures (decisions, counted loops, conditional loops). The examples in the following sections all use control structures written within functions.

Control Structures

Three ways to control the flow of a program by controlling which commands are executed and how many times.

1) Decision (if-statements)
2) Counted loop (for-loops)
3) Conditional loop (while-loops)

If Statements   -   Decision (no loop)

```
do this if (condition true) single line to execute
   do this other thing otherwise optional addition to structure
```

```
if "condition is true"
    "statement 1"
    "statement 2"
```

For loop   -   counted loop (repeat statements a pre-determined number of times)

```
for variable start, next, end
   "statement 1"
   "statement 2"
```

While loop   -   conditional loop (repeat statements as long as condition remains true)

```
while "condition is true"
   "statement 1"
   "statement 2"
```
More details and examples: IF-Statements

Venn diagram of branching

Example #1: assigning letter grades

grade >= 90
'A'

80 > grade >= 70
'C'

70 > grade >= 60
'D'

90 > grade >= 80
'B'

grades < 60
'F'
Write Mathcad code to express the above grade classifications
This structure uses 5 separate if - statements with upper and lower limits
These if - statements - in - series do not interface with each other

assignment equal sign within a function

| grade(x) := | out ← "you get an A" if (x ≥ 90) | out ← "you get a B" if (x ≥ 80) \(\land\) (x < 90) | out ← "you get a C" if (x ≥ 70) \(\land\) (x < 80) | out ← "you get a D" if (x ≥ 60) \(\land\) (x < 70) | out ← "you fail" if (x < 60) |
| note that these lines that use the function appear beneath the first line of the function |
| grade(50) = "you fail" | grade(61) = "you get a D" | grade(75) = "you get a C" | grade(88) = "you get a B" | grade(93) = "you get an A" |

• each decision structure is evaluated separately
• if one condition is found true, all other conditions are still checked

example #2: grades : find the logic error in the structure below

| grade(x) := | out ← "you get an A" if x ≥ 90 | out ← "you get a B" if x ≥ 80 | out ← "you get a C" if x ≥ 70 | out ← "you get a D" if x ≥ 60 | out ← "you fail" if x < 60 |
| grade(50) = "you fail" | grade(61) = "you get a D" | grade(75) = "you get a D" | grade(88) = "you get a D" | grade(93) = "you get a D" |

• several conditions overlap
• decision structure will execute **sequentially** just like any other command
• First condition found true is used, all others are still evaluated
• As long as grade is \(\geq\)60, letter grade will always be assigned ‘D’
• venn diagram for previous example

CONCLUSION: Using a string of if - statements in series must be done cautiously
Example #3: grades: a working version of example #2

Now we will NEST the if - statements instead of using them in series. In this case, the next if - statement will only be evaluated if the previous one is true. In this way, we won’t overlap assignments

• each condition is evaluated based on the result of the previous condition
• So if x < 60, none of the IFs are seen, and the ‘otherwise’ action is triggered

Example #4: grades: another working version of example #2

Finally, we can just use the series if - statement approach if we stack the decisions in the reverse order of example #2

quick quiz: when will the following statement be true?

logical symbol for ‘or’

if \((x \geq 70) \lor (x \leq 80)\)

the choice of ‘and’ / ‘or’ completely changes the meaning of the condition, test your logic before using them
IF - Function

We’ve just seen some examples of the if - statement. Another version of ‘if’ is the IF - Function

\[
\text{if} (\text{cond, x, y}) \text{ Returns } x \text{ if logical condition } \text{cond} \text{ is true (non-zero), or } y \text{ otherwise.}
\]

It's really a short-cut version of the if statement we just saw above. You can find out a bit about this through the help -> index -> IF function -> Quick Sheet Example

Here we’ll see that we can use either the If - function or the If-statement to do the same stuff. Following the example, start by defining a range variable from -2 to 2, and creating a function to work with.

\[
x := -2, -1.9.. 2
\]
\[
f(x) := x^2 - 1
\]

I’ll show you how to plot in class

Let’s use the two different kinds of decision - if to alter the basic function. We’ll make g(x) and g2(x) to do the same thing. Note that this example differs from the others that follow in that we are making decisions based on f(x) instead of x alone (based on the y - axis value rather than the x-axis value).

\[
g(x) \text{ is } f(x) \text{ when } f(x) > 0 \text{ and } 0 \text{ otherwise}:
\]
\[
g(x) := \text{if}(f(x) > 0, f(x), 0)
\]
\[
g2(x) := \begin{align*}
\text{out} & \leftarrow f(x) \text{ if } f(x) > 0 \\
\text{out} & \leftarrow 0 \text{ otherwise}
\end{align*}
\]

equivalent
Another example. We’ll make \( h(x) \) and \( h_2(x) \) to do the same thing

\[
h(x) \text{ is } f(x) \text{ when } x \geq 1 \text{ and } -f(x) \text{ otherwise},
\]

\[
h(x) := \text{if}(x \geq 1, f(x), -f(x))
\]

\[
h_2(x) := \text{out} \leftarrow f(x) \text{ if } x \geq 1
\]
\[
\text{otherwise}
\]

Another example. We’ll make \( k(x) \) and \( k_2(x) \) to do the same thing

\[
k(x) \text{ is } f(x) \text{ when } -1 < x < 1 \text{ and } -f(x) \text{ otherwise},
\]

\[
k(x) := \text{if}(x > -1) \land (x < 1), f(x), -f(x)]
\]

\[
k_2(x) := \text{out} \leftarrow f(x) \text{ if } (x > -1) \land (x < 1)
\]
\[
\text{otherwise}
\]

Another example. We’ll make \( l(x) \) and \( l_2(x) \) to do the same thing

\[
l(x) \text{ is } f(x) \text{ when } 1 < x \text{ or } x < -1, \text{ and } -f(x) \text{ otherwise},
\]

\[
l(x) := \text{if}(x < -1) \lor (x > 1), f(x), -f(x)]
\]

\[
l_2(x) := \text{out} \leftarrow f(x) \text{ if } (x < -1) \lor (x > 1)
\]
\[
\text{otherwise}
\]
Another example. We’ll make \( m(x) \) and \( m_2(x) \) to do the same thing.

\[
m(x) \text{ is } f(x) \text{ when } x < -1, \text{ is } -f(x) \text{ when } x > 1, \text{ and 0 otherwise:}
\]

\[
m(x) := \text{if}(x < -1, f(x), \text{if}(x > 1, -f(x), 0))
\]

\[
m_2(x) := \begin{cases} 
\text{out} \leftarrow f(x) & \text{if } x < -1 \\
\text{out} \leftarrow -f(x) & \text{if } (x > 1) \\
\text{out} \leftarrow 0 & \text{otherwise}
\end{cases}
\]

\[
\begin{align*}
\text{for loop Example #1:} & \quad \text{Use a for-loop to create a function that generates vectors of numbers.} \\
\text{Create_Vec} &: \left( \begin{array}{c}
\text{start} \\
\text{stop} \\
\text{inc}
\end{array} \right) \\
\text{Create_Vec}(0, 2, .5) &= \begin{pmatrix}
0 \\
0.5 \\
1 \\
1.5 \\
2
\end{pmatrix} \\
\text{Create_Vec}(-2, 6, 2) &= \begin{pmatrix}
-2 \\
0 \\
2 \\
4 \\
6
\end{pmatrix}
\end{align*}
\]

- The indented line is inside the for-loop. The index ‘i’ increases by one each time through until ‘i’ takes a value greater than ‘num’, at which point the loop stops.
**For loop Example #2:** Create a function that sums up the values in a vector that is passed in

\[
\text{sum}_\text{vec}(\text{vec}) := \begin{cases} 
\text{sum} \leftarrow 0 \\
\text{for } i \in 1..\text{length}(\text{vec}) \\
\text{sum} \leftarrow \text{sum} + \text{vec}_i \\
\text{sum} 
\end{cases}
\]

Length is a built in function that operates on vectors.

\[
\begin{pmatrix} 
98 \\
84 \\
71 \\
88 \\
56 
\end{pmatrix}
\]

Grades := summation := \text{sum}_\text{vec}(\text{Grades})

\text{summation} = 397

We are keeping a running sum by adding sum to itself plus the next grade on the list each time through the list.

**For loop Example #3:** Create a function that finds the mean value of all numbers in a list that are greater than 60. In this case, we’ll have to make a decision about each number before we add it to a running sum to calculate the average value. We also have to keep track of how many values go into the sum (how many are greater than 60).

\[
\text{Calc}_\text{Ave}(\text{vec}) := \begin{cases} 
\text{sum} \leftarrow 0 \\
\text{total} \leftarrow 0 \\
\text{for } i \in 1..\text{length}(\text{vec}) \\
\text{if } \text{vec}_i > 60 \\
\text{sum} \leftarrow \text{sum} + \text{vec}_i \\
\text{total} \leftarrow \text{total} + 1 \\
\text{average} \leftarrow \frac{\text{sum}}{\text{total}} 
\end{cases}
\]

\[
\begin{pmatrix} 
98 \\
84 \\
71 \\
88 \\
56 
\end{pmatrix}
\]

Grades := \text{Calc}_\text{Ave}(\text{Grades}) = \left( \frac{85.25}{4} \right)
For loop Example #4:
Now we’ll take the previous example up a notch. Write a program that processes a vector of grades to count how many students pass and how many fail, and calculate the average overall score.

Solution process:
Input: individual student grades
output: number of failing students
        number of passing students
        average of all grades

Pseudocode:
enter the vector of student grades
start a loop that executes once for each student
decide if grade is passing or failing
  if passing, add one to the passing variable
  if failing, add one to the failing column
add grade to running total regardless of pass or fail so we can calculate average
go back to top of the loop to get next grade
average grade is running total grade divided by number of grades entered
output the results (# pass, # fail, average)

Grades := \[
\begin{pmatrix}
75 \\
84 \\
23 \\
96 \\
43 \\
\end{pmatrix}
\]

\[\text{class}(x) :=
\begin{align*}
\text{sum} & \leftarrow 0 \\
\text{pass} & \leftarrow 0 \\
\text{fail} & \leftarrow 0 \\
\text{for } i \in 1..\text{length}(x) \\
\text{pass} & \leftarrow \text{pass} + 1 \text{ if } x_i \geq 60 \\
\text{fail} & \leftarrow \text{fail} + 1 \text{ otherwise} \\
\text{sum} & \leftarrow \text{sum} + x_i \\
\text{ave} & \leftarrow \frac{\text{sum}}{\text{length}(x)} \\
\text{out} & \leftarrow \begin{pmatrix} \text{fail} \\ \text{pass} \\ \text{ave} \end{pmatrix} \\
\end{align*}\]

\[\begin{align*}
\text{failures} & := \text{class}(\text{Grades}) \\
\text{passes} & := \text{result}_1 \\
\text{average} & := \text{result}_3 \\
\end{align*}\]

Why are we setting these three to zero?

A function can only send out one ‘thing’, so we make it contain all three things we need

failures = 2    passes = 3    average = 64.2
For loop Example #5: Consider the cantilevered beam illustrated to the right with a variable valued point load $P$ at the end. An equation is provided which describes the beam deflection at any point $x$ along the beam. The program below uses a for loop to: a) create a vector of $x$-locations along the length of the beam, b) create a vector with the corresponding deflections, c) adds one to a scalar being used to index the vectors being created. NP is the number of points at which we want to calculate the deflection between 0 and the total length $L$

\[ defl = \frac{-P}{6EI} (3Lx^2 - x^3) \]

For loop Example #5

```
L := 20  
I := 600  
E := 29000  
P := 20  
NP := 10  

BeamDefl(L, NP, E, I, P) :=

\[ \text{inc} \leftarrow \frac{L}{(NP - 1)} \]
\[ i \leftarrow 1 \]
for $x \in 0, \text{inc} \ldots L$
\[ \text{xaxis}_i \leftarrow x \]
\[ \text{defl}_i \leftarrow \frac{-P}{6EI} (3Lx^2 - x^3) \]
\[ i \leftarrow i + 1 \]

\[ \text{results} := \text{BeamDefl}(20, 10, E, I, P) \]
```

We've set up the output from the function to contain the two vectors xaxis and defl. They are placed as individual elements in a 2x1 vector. Note that when we display the contents of 'results', it tells us that each of the two elements are a 10x1 vector. Thus 'results' is called a 'data structure'. The difference is that each element in a data structure can be a vector or matrix, not just a scalar. These elements are indexed just like a vector (with a subscript). In order to display and use the contents of the data structure, we need to assign a name to each of the elements in the data structure. This is done below above the graph.

\[ \text{xaxis} := \text{results}_1 \]
\[ \text{defl} := \text{results}_2 \]

Assigning names to elements in the data structure
while - loop  conditional loop (decision and loop)

while (condition)
    statements inside conditional loop

This loop continues to repeat as long as the condition(s) are true
statements inside loop must be able to change variable(s) used in the condition

While loop Example #1:
We have a long vector of numbers, and there is only one value less than zero in that vector. We want to find where in that list the negative value is.

Find_Negative(vec) :=
stop ← 1
location ← 1
while stop ≠ 1
    if vec_location < 0
        out_location ← location
        stop ← 0
    location ← location + 1
out_location

Something New: notice the condition in the while statement above. We are asking if stop is equal to 1 each time through the loop. If that condition is true, the loop is executed again. We are not assigning 1 to the variable stop, we are comparing the existing variable stop to a value. That equal sign is in bold, which means it's a comparison. We get that one by using Ctrl =

‘location’ is what we are using to index the vector. It has to be integer values, and is set up to be 1 first time through the loop, then 2, etc. so that in the if statement we are sequentially comparing the numbers in the vector vs. 0 one at a time. When we find the negative value, the value of ‘location’ is pointing to the place where the negative value resides in the vector, so we save it into ‘out_location’, which is listed at the bottom of the function as the variable that is output from the function. We also re-assign ‘stop’ to 0, which will cause the while loop to stop executing (it only continues if stop = 1). Thus our algorithm will keep looking until it finds a negative value, save the location of that negative value, then exit the loop, and we’re done.
While loop Example #2:
We’ll change the previous example of calculating the deflection of a cantilevered beam so that it is now a design problem. Let’s say that following parameters are fixed (non-negotiable values): I, NP, P, E
And the length of the beam L is to be designed such that the maximum deflection is close to 0.06 inches
Obviously, the longer the beam, the greater the tip deflection. We will iteratively increase the length of the beam until the deflection is greater than 0.06 inches.

What we will do is calculate the tip (maximum) deflection starting with the length of 20 inches. If the tip deflection is less than 0.06 inches, we will increase the length of the beam by a factor of 5 % of its current value. This will be done as many times as necessary until the tip deflection exceeds 0.06 inches.

Pseudocode:
receive the input (length of beam, number of points along the beam).
Calculate the maximum deflection at the tip of the beam
  if the deflection magnitude is less than 0.06 inches, increase length by 5 %
  recalculate the tip deflection and repeat using a while loop
Now that length has been chosen so that tip deflection exceeds 0.06 in, calculate the increment
Start a loop to calculate the x-axis values and deflection at these locations
place the output into a data structure
\[ kips := 1000\text{lbf} \quad ksi := \frac{kips}{\text{in}^2} \]

\[ E := 29000\text{ksi} \quad P := -20\text{kips} \quad I := 50\text{in}^4 \quad L := 20\text{in} \quad NP := 10 \]

\[
\text{BeamDefl} (L, NP, P, E, I) :=
\frac{-P \cdot L^3}{3 \cdot E \cdot I}
\]

\[ \text{maxdefl} \quad \text{while} \quad |\text{maxdefl}| < 0.06\text{in} \]

This control structure adjusts \( L \) until it suits my given constraint of \(< 0.06\text{in} \)

\[
\text{maxdefl} \quad \text{L} \quad \text{i} \quad \text{inc} \quad \text{for} \quad x \quad \text{defl} \quad \text{i} \quad \text{inc} \quad \text{for} \quad x \quad \text{defl}
\]

These two outputs are copied and pasted into the variable names given in the yellow ( )

To plot vectors that have units on a scale with the final units you want, divide the vector name by the unit you want to the right, both the \( x \)-axis and the \( y \)-axis are displayed in inches.
Another Example: finding maximum value in a vector (list of numbers)

Given a vector of any size with random numbers inside, write a program that will identify two things:

1) The maximum value in the vector
2) The location (index) of that maximum value in the vector

This is a task that can be accomplished using built-in Mathcad functions. Here we will do it the hard way to learn the logic and learn to use simple control structures.

Solution Procedure

It helps to create a simple example to work with.
Let’s work with a vector that contains only six integers.

>> Grade = [56 72 65 98 84 91];

Goal: Have Mathcad identify 98 as the maximum value, and find its index location to be 4
we will need the following tools:

a) decisions
b) a loop
c) vector indexing
d) some scalar variables to keep track of decisions

Pseudo-code:

1) input the student grades
2) find out how many grades there are
3) look at each grade and compare it with the other grades
   a) pick the first two, save the bigger of the two as the current max save its location
   b) pick #3 spot and compare it with the current max, if #3 is bigger, save it as the new current max, and save the new location
   c) pick #4 and compare it to current max, repeat b)
   d) repeat for each remaining grade, updating current max if new max is found

Note that c) and d) are repeating the instructions in b) as we iterate the location being compared (loop)
Note that b) makes a decision (if current bigger than previous, then...)
Now let’s refine the Pseudocode to reflect these observations

1) Enter grades and determine total number
2) Assume first grade is largest (assign max := grade1, maxptr := 1)
3) Loop from i = 2 to the final grade:
   compare gradei with max
   if gradei > max,
   new value for max is gradei
   new value for maxptr is i
   if gradei not > max
   move to next value to compare

Mathcad Code:
Now that we have a good pseudocode, we translate it into Mathcad

\[
\begin{align*}
\text{findmax}(\text{in}) & := \\
& \text{maximum} \leftarrow \text{in}_1 \\
& \text{maxptr} \leftarrow 1 \\
& \text{for } i \leftarrow 2 \ldots \text{length}(\text{in}) \\
& \quad \text{if } \text{in}_i > \text{maximum} \\
& \quad \quad \text{maximum} \leftarrow \text{in}_i \\
& \quad \quad \text{maxptr} \leftarrow i \\
& \left(\text{maximum} \atop \text{maxptr}\right) \\
& = \left(\begin{array}{c} 56 \\ 72 \\ 65 \\ 98 \\ 84 \\ 61 \end{array}\right) \\
& \left(\begin{array}{c} \text{biggest} \\ \text{location}\end{array}\right) := \text{findmax}(\text{grades}) \\
& \text{biggest} = 98 \quad \text{location} = 4
\end{align*}
\]

Isn’t saving the location of the maximum, and the max itself a little redundant? We only need to save the location (index) of the number. Why not just save the location, and retrieve the value when needed? We can by using the idea of a pointer.

**Pointer**
- A variable whose purpose is to keep track of where a value is within an array, not the value itself
- Must be an INTEGER since it is used as an index to a vector

Let’s re-work the maximum algorithm, but only save the location of the current maximum, not the max itself...using a pointer
Finding a maximum value in a vector using the pointer concept

\[
\text{findmax}(\text{in}) := \text{maxptr} \leftarrow 1 \\
\text{for } i \in 2..\text{length}(\text{in}) \\
\quad \text{maxptr} \leftarrow i \text{ if } \text{in}_i > \text{in}_{\text{maxptr}} \\
\text{maxptr}
\]

I find the pointer that points to the location of the biggest value

\[
\text{pointer_to_max} := \text{findmax}(\text{grades})
\]

I use the pointer as an index to the array

\[
\text{biggest} := \text{grades}_{\text{pointer_to_max}}
\]

This is a little more efficient (fewer steps) and reduces the number of variables needed (maximum is not used this time)

For the sake of showing off Mathcad’s nice features, let’s use some more advanced built in functions to solve this problem.

‘max’ is a built in function that finds the maximum value of a vector

\[
\text{max}(\text{grades}) = 98
\]

But note that this function does not return its location, just the value. How is this a drawback??