Advice of the Day

- Find a Mentor
  - Why do you admire them?
  - How can you be like them?

- Anti-Mentors
  - Why don’t you admire them?
  - How can you avoid their faults?
Categorizing Code

- Sections of computer code can be classified into three categories
  - Sequences – lines of code are executed one after another (i.e. what we’ve been doing)
  - Selection Structures – executes some piece of code if some known condition is true, otherwise executes some sort of alternative code (can be many branches). [We talked about these last week]
  - Repetition Structures (loops) – causes a group of statements to be executed multiple times (either a fixed number, or until some stated condition is met).
Graphical View of Classification

Sequence

Straight through

Selection

As many branches as needed; may re-converge to sequence

Repetition

Fixed or Variable Number of loops
To Loop, or Not to Loop?

- In MATLAB, loops should not be your first choice
  - Low performance (bad clock times)
  - Many good alternatives:
    - Array operations
    - `find` command
    - Code vectorization (operations on an entire vector at a time instead of 1 element at a time)
- Sometimes, however, loops are unavoidable
- 3 basic types of loops:
  - `for` loop
  - `while` loop
  - `do while` loop – unavailable in MATLAB (but can often “simulate” its behavior with a `while` loop)
**for loop**

- Basic syntax:

  ```
  for some_index_variable = some_matrix
    some commands to be executed
  end
  ```

- Loop is executed once for each element in the index matrix, e.g.

  **Input:**
  ```
  for k = [1 3 5 7]
    k
  end
  ```

  **Output:**
  ```
  k = 1
  k = 3
  k = 5
  k = 7
  ```

  Note: At end of the loop, the index variable keeps its most recent value.

A common convention is to use an index `i` in many languages (continuing, if needed, through the alphabet if multiple indices are needed due to nesting).

Many people avoid `i` and `j` in MATLAB though because they by default represent $\sqrt{-1}$.

In MATLAB, a common index choice is `k`. 
More for loop examples

- **Input:**
  ```matlab
  for k = 1:5
      my_var = factorial(k)
  end
  ```

- **Output:**
  ```matlab
  my_var = 1
  my_var = 2
  my_var = 6
  my_var = 24
  my_var = 120
  ```

- **Input:**
  ```matlab
  for k = 1:5
      my_var2(k) = factorial(k)
  end
  ```

- **Output:**
  ```matlab
  my_var2 = 1
  my_var2 = 1 2
  my_var2 = 1 2 6
  my_var2 = 1 2 6 24
  my_var2 = 1 2 6 24 120
  ```
Nesting for loops

- **Input:**

  ```matlab
  for i = 1:2
      for j = 11:12
          fprintf('i: %i, j: %i\n', i, j)
      end
  end
  ```

- **Output:**

  - i: 1, j: 11
  - i: 1, j: 12
  - i: 2, j: 11
  - i: 2, j: 12

**Note:** Inner counter resets every time the loop is re-initialized
Nesting for loops Cont’d

- **Input:**

```matlab
for i = 1:2
    for j = 3:4
        for k = 5:6
            fprintf('i: %i, j: %i, k: %i\n', i,j,k)
        end % end k loop
    end % end j loop
end % end i loop
```

- **Output:**

```
i: 1, j: 3, k: 5
i: 1, j: 3, k: 6
i: 1, j: 4, k: 5
i: 1, j: 4, k: 6
i: 2, j: 3, k: 5
i: 2, j: 3, k: 6
i: 2, j: 4, k: 5
i: 2, j: 4, k: 6
```

**Note:** Inner counter resets every time the loop is re-initialized.
Indexing **for** loops with matrices

- Instead of indexing a for loop with a matrix, you have the option of using a matrix as the index (operates on the column – remember: memory is column dominant)
- These aren’t used too often, but can be useful, particularly if you want to vectorize your code

**Input:**

```plaintext
for k = [1 2 3; 4 5 6; 7 8 9]
    my_var3 = factorial(k)'
end
```

**Output:**

```
my_var3 = 1 24 5040
my_var3 = 2 120 40320
my_var3 = 6 720 362880
```
“Think-Pair-Share”

- What will result from the following code fragments? Think about it, then find a partner/small group to talk about it with. We’ll then discuss as a group.

```matlab
for i = 1:3:10
    i = i + 1
end

errors = [4.2 1.7 2.3 9.4 0.2]
best_errors = [];
for j = 1:numel(errors)
    if errors(j) < 2
        best_errors = [best_errors errors(j)]
    end
end
fprintf('Value of j is %i
', j)
```

What alternatives to this approach exist?
**while loop**

- **Basic syntax:**

```plaintext
while condition
    some commands are executed
end
```

- Loop executes if condition evaluates to make the statement “while true.” If “while false,” loop won’t execute. Potential: the loop might never execute!

**Input:**

```plaintext
k = 10
while k >= 6
    k = k - 2
end
```

**Output:**

```plaintext
k = 10
k = 8
k = 6
k = 4
```
Special **while** loop examples

**Input:**

```plaintext
k = 0
while k ~= 0
    k = k + 2;
end
```

**Output:**

```plaintext
k = 0
```

**Special Case:**

Loop never executes

**Input:**

```plaintext
k = 5
while k <= 5
    k = k - 3
end
```

**Output:**

```plaintext
k = 5
k = 2
k = -1
Continues infinitely...
```

**Special Case:**

Loop never terminates ("infinite loop" – can terminate it with ctrl + c)
Can use a `break` statement to cause the termination of the smallest enclosing `while` or `for` loop

Beware: It is often very bad form to do this without a really good reason. In general, it is much “better” to write “better” loop termination conditions

You WILL see this (and continue on next slide) though, particularly in numerical analysis

**Code:**

```plaintext
i = 0;
while (i >= 0)
    i = i + 1
    j = input('Enter number > 2: ')
    if j <= 2
        disp('You didn''t listen!')
        disp('Program terminating')
        break
    end
end
```

**Output:**

```
i = 1
Enter a number greater than 2: 4
j = 4
i = 2
Enter a number greater than 2: 1
j = 1
You didn't listen!
Program terminating
```
continue

- Can use a `continue` statement to skip the rest of the loop, advancing to the next loop pass

- `break` and `continue` may look useful – but in general, they should not be “go-to” techniques

- One useful use for these is “error trapping” (trying to get valid input from the user)

**Code:**

```plaintext
i = 0;
while (i >= 0)
    i = i + 1
    j = input('Enter number > 2: ')
    if j <= 2
        disp('You didn''t listen!')
        disp('Program terminating?')
        continue
    end
end
```

**Output:**

```
i = 1
Enter number > 2: 3
j = 3
i = 2
Enter number > 2: -2
j = -2
You didn't listen!
Program terminating?
i = 3
```

Continues ad nauseum b/c infinite loop
**clock/etime**

- We’ve seen the `clock` command before (gives you the current time)
- Another command: `etime` performs comparison between a start time and an end time

**Sample Use:**

```matlab
t0 = clock;
t1 = clock;
time_elapsed = etime(t1,t0) \% in seconds
```

**Beware:**
- Does not account for leap seconds, daylight savings time adjustments, or differences in time zones
- These functions are based on the system time, and can be adjusted periodically by the OS (so might not be 100% accurate for timing).
cputime

- Returns CPU time (in seconds) since you started MATLAB; can use differences to do timing

- Sample Use:

  t0 = cputime;
  peaks;
  t1 = cputime - t0

- Interesting, but not recommended for timing operations. We have one more technique designed especially for this!
**tic/toc**

- Can use `tic` and `toc` as stopwatches; once again, the time difference is in seconds

- **Sample Use #1:**
  ```matlab
tic
  peaks;
  toc
  ```

- **Sample Use #2:**
  ```matlab
t0 = tic;
  peaks;
  t1 = toc(t0)
  ```

- If you take the approach of Sample Use #1, note that: 1) can use multiple `toc` commands for the same `tic`, and 2) calling `tic` wipes out any previous `tic` commands
Analyze Code & Run and Time

- **Analyze Code:** "This report displays potential errors and problems, as well as opportunities to improve your MATLAB programs"
- **Run and Time:** Suggests performance improvements
Performance

- You now know various control structures:
  - vectorized operations (array ops)
  - if structures
  - switches
  - for loops
  - while loops

- You now also know about timing code
  - clock/etime
  - cputime
  - tic/toc

- We can now have meaningful discussions about code performance
Performance “Guiding Principles”

- Only print using print statements
  - It takes too much time to dump all of your calculation results to the screen - MATLAB has to stop calculations to print, so only print what you need

- If you know how big a result vector will be from a loop, consider pre-allocating a vector of that same size – potentially with the `zeros` or `ones` commands
  - Some languages actually *require* pre-allocation; MATLAB isn’t one of them, but MATLAB works more efficiently if it doesn’t have to move things around in memory

- Consider feedback from the MATLAB GUI – both the profiler and the run and time features
  - MATLAB is very smart, and can often offer helpful suggestions

- Because of quirks with OS’s, it’s often beneficial to run code multiple times to get average run times
Pre-allocating Variables Example (1 of 4)

- Code:

```matlab
    tic
    x = 0;
    y = [];
    max_iterations = 1000; % 1 thousand
    for i = 1:max_iterations
        x = x + 1
        y = [y x]
    end
    toc
```

- Output: (ignoring x and y values from each iteration)

```
Elapsed time is 4.345111 seconds.
```

Didn’t Pre-Allocate Space

Didn’t Suppress Loop Output
Pre-allocating Variables Example (2 of 4)

- Code:

```matlab
tic
x = 0;
y = [];
max_iterations = 1000; % 1 thousand
for i = 1:max_iterations
    x = x + 1;
y = [y x];
end
toc
```

- Output: (ignoring x and y values from each iteration)

Elapsed time is 0.021548 seconds.
Pre-allocating Variables Example (3 of 4)

- **Code:**

```matlab
tic
x = 0;
max_iterations = 1000; % 1 thousand
y = zeros(1,max_iterations); % pre-allocated space
for i = 1:max_iterations % pre-allocated space
    x = x + 1
    y(i) = x
end
toc
```

- **Output:** (ignoring \(x\) and \(y\) values from each iteration)

  Elapsed time is 9.262178 seconds.
Pre-allocating Variables Example (4 of 4)

- Code:

```matlab
tic
x = 0;
max_iterations = 1000; % 1 thousand
y = zeros(1,max_iterations); % pre-allocated space
for i = 1:max_iterations
    x = x + 1;
    y(i) = x;
end
toc
```

- Output:

```
Elapsed time is 0.010999 seconds.
```
Questions?

- Quiz 5
  - Available on ANGEL between 9 a.m. today and 9 p.m. on Tuesday

- Wednesday’s Class
  - Interactive Lecture in Computer Labs

- Lab 5
  - Posted Thursday at noon; due Saturday at noon