ENME/ENAE 202 - Computing Fundamentals for Engineers
Matlab Lab 5 — Fall 2025

This studio contains 3 problems, the first 2 of which require you to write functions. Keep all of your code in a single m-file, with the functions placed at the end of the file and all other code above. Please add comment lines before each problem and function in the m-file to make it easier to identify the code organization.

1. Create a function called `mytranspose()` that will take a 2D array (matrix) as an argument, and return the transpose of that matrix, but *without using the built-in Matlab transpose operator or transpose function*. Before you begin, think about the mechanics of the transpose process: for each value of the initial matrix `A(i,j)` the given value should be assigned to element `B(j,i)` in the new transpose matrix. The most direct way to implement this process is through the use of a pair of nested loops.

Once you think your function is operational, check it by adding the following code above the function definition:

TA

```
A = mytranspose([1 2 3; 4 5 6; 7 8 9; 10 11 12]]
```

to produce:

```
A =
  1  4  7  10
  2  5  8  11
  3  6  9  12
```

2. No simple formula exists to compute the zeros of a general function $f(x)$, *i.e.*, the roots of $f(x) = 0$. Instead, numerical methods must be used to find these zeros. Of course, Matlab has a built-in function that does this for us: `roots()`. Here we want to make our own roots function that will do the same thing, called `myroots()`. We have previously considered a brute force approach to this problem: compute $f(x)$ on a large range of values for x, and look for the one which results in $f(x)$ being close to zero. However, this approach is inelegant, imprecise, and requires a large amount unnecessary computation time. A more efficient and precise approach called Newton's method involves using successive guesses to "home in" on a value of x for which $f(x) = 0$. Newton's method is implemented as follows:

a. Start with an initial guess for z, where z is a zero of the function $f(x)$.
b. Compute $f(z)$.
c. If $f(z) = 0$, then z is a zero and we are done.
d. If $f(z) \neq 0$, generate a new guess for the zero z according to the formula:

$$z_{\text{new}} = z - \frac{f(z)}{f'(z)} \tag{1}$$

e. Using the new guess for z, repeat from step (b)

In this formula, the old guess z is used to determine the new guess z_{new} based on the current value of the function $f(z)$. We then check this new z to see if $f(z) = 0$, and if not, we repeat the above calculation with the new z to generate yet another new z, continuing in this fashion as long as necessary until $f(z) = 0$.

Of course, the finite precision of the computer means that we can't really continue the process of generating better and better guesses until $f(z)$ is exactly zero, as this would require essentially infinite time. Rather, we must settle for zeros which satisfy $|f(z)| < \varepsilon$ where ε is a small tolerance limit. For this assignment, use $\varepsilon = 10^{-10}$ (`1e-10`).

Write a Matlab function called `myroots()` such that the command `z = myroots(p,zi)` uses Newton's method to find one of the zeros of a polynomial defined by `p`, using a starting guess of `zi` for the zero.

Hints: Use `polyval()` *to compute* $f(z)$*, and use* `polyder()` *together with* `polyval()` *to compute the derivative* $f'(z)$*. Since you don't know how many iterations will be required to reach the desired accuracy, use a* `while` *loop to keep generating new guesses until the tolerance is met.*

As a check, add the following code above the function definition in your m-file:

```
myroots([1 54 -4705 -24750], 0)
```

This code should yield:

```
ans =
   50
```

3. In this problem, you will need to think about how to combine nested loops, multiple conditionals, and flags. Consider the 4th order polynomial used to test the `myroots()` function in Problem 2:

```
p = [1 54 -4705 -24750]
```

This particular polynomial possesses 3 zeros, all of which are unique real numbers somewhere in the range of $(-200, 200)$. Unfortunately, our algorithm can only identify a single zero, and this zero will vary depending on the initial guess for z. The goal of this problem is to write code that will find *all* of the zeros of the polynomial.

To do this, use a loop to choose initial guesses ranging from -200 to +200 in steps of 1. For each guess, use your `myroots()` function to find a nearby root, and add it to an array called `z_list` to hold all of the identified zeros. Each time a root is found, your code will need to loop through the current values in `z_list` to see if the new root has already been found from a previous guess. If it has not, add it to `z_list`, otherwise skip it and move on to the next starting guess. In this way `z_list` will only contain 3 values (each of the 3 unique zeros) at the end of your code. When checking whether a zero is already in the array, you will need to evaluate equality within some tolerance limit, since finite numerical precision may lead to slightly different calculations depending on the starting guess. For this problem, any zeros within a tolerance of `1e-2` can be treated as identical.

Hint: use a flag variable to determine whether a zero is already in `z_list` *when looping through the array. For example, you could start the loop by assuming the zero is unique, and flip the flag if*

this is found not to be the case. After the loop, you can then test the flag to determine whether or not to add the zero as a new entry in `z_list`.

Finally, display the identified roots to the screen. All code for problem 3 should be placed above the function definitions in your m-file (note that this problem does not involve creating a new function).

Check your code by comparing the resulting zeros with the built-in function: `roots(p)`.

TA