Matthew Lacaire
UID: 205288439
CEE/MAE M20
October 17, 2020

Homework 2

1. Homework Assignment Calculations

a. Introduction

In this problem, the program is tasked with receiving an assignment's assignment date, getting the days taken to finish, and then calculating the finish date. In the second part, the actual due date is taken as an input, and the program has to evaluate whether the student has passed in work late and needs to evaluate accordingly.
Of course, along the way, inputs are checked for validity.

b. Model and Methods

I begin this program by asking the user for inputs in the format MM/DD/YY. Then, I also ask how long the assignment took to finish in days. Evaluating these for validity, I ensure that input values are not negative or too large:

```
dateMonth = input('Insert the assigned HW date, starting with months; e.g.
XX/XX/XXXX\n'); %Inputs regarding assigned date.
dateDay = input('\n'); %Input for nth day of the month
dateYear = input('\n');%Input for year
if ((dateMonth > 12 || dateMonth < 1) || (dateDay > 30 || dateDay < 1) || dateYear <=
0) %Input processing. If month isnt 1-12, or day isn't 1-30 or year is before 0, error
is produced.
    error('Invalid Input.');
end
assignmentTime = input('How long did the assignment take to finish (days)?\n');
%Asking for length to finish assignment.
if  (assignmentTime <= 0) %Input processing. Length of time must be positive.
    error('Invalid Input.');
end
```

I then apply two important algorithms that form the core of my process. These are described in depth below. After all of this, I use fprintf to print results:

```
fprintf('Your homework was submitted on %02.0f/%02.0f/%02.0f\n', endMonth, endDay,
endYear);

fprintf('Great work! Your score is 100!\n')
or, if late:

fprintf('You were %.0f days late. Your score is %.0f\n',result, studentScore)
or too late for credit:

fprintf('Too late! No credit is given.\n')
```

c. Calculations and Results

In my process, I used two important algorithms. The first is an algorithm that turned the length of time to finish the work (in days) into individual days, months, and years in a logical format. The entire algorithm is below:

```
timeDays = rem(assignmentTime, 30); %this following algorithm breaks the earler input
of assignmentTime into format (X days, X months, X years)
timeMonths = ((assignmentTime - timeDays) / 30); %Does so for months.
timeYears = 0; %By default, timeYears is set to zero.
if (timeMonths > 11) %If the time of the assignment is more than 11 months (12+), we
begin calculating with years.
    if (timeMonths == 12) %Sets timeYears to 0, timeMonths is zero. (12 mo = one year.
We need to subtract months then!)
        timeYears = 1;
        timeMonths = 0;
    elseif (timeMonths > 12) %If duration is MORE than 12 months, we need to convert
months into years, and store months as a remainder.
        timeYears = (timeMonths - rem(timeMonths, 12)) / 12; %Calculates 'timeYears'
and finds assignment length in years w/ mod.
        timeMonths = rem(timeMonths, 12); %Redefines 'timeMonths' as a remainder.
    end
else
    timeYears = 0; %Could be omitted. If the length of time spent on the assignment is
less than 12 months, 0 years are involved.
end
```

I make extensive use of mod (remainder in MATLAB) to turn the number of days into a format in days/months/years. For example, this algorithm turns 376 days into “1 year, 16 days” as interpreted by the program. This is important because it generates values necessary for the next algorithm.

The second algorithm actually gets the reformatted values of time spent (now in years, days, months instead of just days) and systematically adds them to the original start date. The algorithm is pasted below:

```
%% Algorithm (Part A)
timeDays = rem(assignmentTime, 30); %this following algorithm breaks the earler input
of assignmentTime into format (X days, X months, X years)
timeMonths = ((assignmentTime - timeDays) / 30); %Does so for months.
timeYears = 0; %By default, timeYears is set to zero.
if (timeMonths > 11) %If the time of the assignment is more than 11 months (12+), we
begin calculating with years.
    if (timeMonths == 12) %Sets timeYears to 0, timeMonths is zero. (12 mo = one year.
We need to subtract months then!)
        timeYears = 1;
        timeMonths = 0;
    elseif (timeMonths > 12) %If duration is MORE than 12 months, we need to convert
months into years, and store months as a remainder.
        timeYears = (timeMonths - rem(timeMonths, 12)) / 12; %Calculates 'timeYears'
and finds assignment length in years w/ mod.
        timeMonths = rem(timeMonths, 12); %Redefines 'timeMonths' as a remainder.
```

```
    end
else
    timeYears = 0; %Could be omitted. If the length of time spent on the assignment is
less than 12 months, 0 years are involved.
end

if (dateDay + timeDays <= 30) %I begin calculating the day on which the HW was
submitted. Simplest case: dateDay+timeDays <= 30
    if (dateMonth + timeMonths <= 12) %Simplest case: months sum to les than 12.
        endMonth = dateMonth + timeMonths; %Dates added.
        endDay = dateDay + timeDays; %Dates added.
        endYear = dateYear + timeYears; %Same.
    else %% "Else" being that dateMonth+timeMonths sum to more than 12
        endMonth = dateMonth + timeMonths - 12; %Months are added, and 12 is
subtracted.
        endDay = dateDay + timeDays; %Simplest case as before w/days. Just add them.
        endYear = dateYear + timeYears + 1; %End year is start date year + time spent
years + 1, because the months "start over" for summing to more than 12.
    end
else %Case where days sum to more than 30.
    if (dateMonth + timeMonths < 12) %Simplest case for months. Just add them.
        endMonth = dateMonth + timeMonths + 1; %Add an extra 1 because the days "roll
over" and create a new month.
        endDay = dateDay + timeDays - 30; %Same concept as line before. Subtract 30
and add +1 to months.
        endYear = dateYear + timeYears; %Sum years as normal.
    elseif(dateMonth + timeMonths >= 12) %% "Else" being that
dateMonth+assignmentTimeMonths sum to more than 12 (or equal)
        endMonth = dateMonth + timeMonths - 12 + 1; %adding start month, and time in
months. Subtracting 12 because months "roll over", and adding 1 because days of the
month do too.
        endDay = dateDay + timeDays - 30; %Days roll over, adding one to month.
        endYear = dateYear + timeYears + 1; %Months roll over, adding one to year.
    end
end
```

This is actually a remarkably complex algorithm. The reason why is because adding the number of months/days taken to finish to the original start date can sometimes "roll over" to the next month or year, and the date must then be adjusted. For this reason, I have doubly-nested "if" statements that deal with individual cases. As an illustrative example, consider:

```
if (dateDay + timeDays <= 30) %I begin calculating the day on which the HW was
submitted. Simplest case: dateDay+timeDays <= 30
    if (dateMonth + timeMonths <= 12) %Simplest case: months sum to les than 12.
        endMonth = dateMonth + timeMonths; %Dates added.
        endDay = dateDay + timeDays; %Dates added.
        endYear = dateYear + timeYears; %Same.
```

This considers the situation in which the dateDay (original start day date) and the timeDays (time to finish days) are less than or equal to 30. Then, another if loop is inserted, dealing with the specific situation when the two month values are less than twelve. If the months sum to more than 12, as mentioned above, we need to deal with the "rolling over" :

```
else %% "Else" being that dateMonth+timeMonths sum to more than 12
        endMonth = dateMonth + timeMonths - 12; %Months are added, and 12 is
subtracted.
        endDay = dateDay + timeDays; %Simplest case as before w/days. Just add them.
```

```
        endYear = dateYear + timeYears + 1; %End year is start date year + time spent
years + 1, because the months "start over" for summing to more than 12.
```

Here, the endMonth and assignmentTime sum to more than 12. Thus, we subtract 12 from the output month (endMonth) and add an extra year to the endYear value. This algorithm is essentially composed of conditionals that function analogously.

Then, results are simply printed:

```
fprintf('Your homework was submitted on %02.0f/%02.0f/%02.0f\n',
endMonth, endDay, endYear); %Outputting results.
```

I use the fprintf to print the submission date. I added the part "%02" to pad the output with zeros instead of spaces. Dates simply look better this way.

In another large algorithm, I deal with inputs of the second date (due date). All text is below:

```
%% Part B : Input
duedateMonth = input('Insert the assigned HW due date, starting with months; e.g.
XX/XX/XXXX\n'); %Input in XX/XX/XXXX.
duedateDay = input('\n');
duedateYear = input('\n');
if ((duedateMonth > 12 || duedateMonth < 1) || (duedateDay > 30 || duedateDay < 1) ||
duedateYear <= 0)
    error('Invalid Input.\n'); %Same as before. Vetting for invalid inputs. See part
(A).
end
if (duedateYear < dateYear) %Additional vetting of inputs. Checking to make sure due
date isn't before date assigned.
    error('Invalid Input. This is before the current date!\n'); %Does check for years.
elseif (duedateYear == dateYear && duedateMonth < dateMonth) %Does check for same
year, different month case.
    error('Invalid Input. This is before the current date!\n');
elseif (duedateYear == dateYear && duedateMonth == dateMonth && duedateDay < dateDay)
%Does check for same year, same month.
    error('Invalid Input. This is before the current date!\n');
end
```

This entire algorithm's purpose is to check if the output is logical and comes before the start date (it makes no sense to be due before assigned). I break up the process into two pieces:

1. Prove inputs aren't strange/irrational. Ex. month value can't be 13 or negative. This is the top part of the code.
2. Check to see if the due date comes after the start date. I do this as follows:
    a. Compare years
    b. If years are same, compare months
    c. If years, months, are the same, compare days.

Then, in an extremely straightforward algorithm, I calculate the "lateness" of the student as follows:

```
numberDaysSubmitted = 360 * endYear + 30 * endMonth + endDay; %Calculates the days
since year 0. This is an extremely extremely straightforward way of comparing two
dates.
numberDaysDue= 360 * duedateYear + 30* duedateMonth +duedateDay;
```

```
result = numberDaysSubmitted - numberDaysDue; %"result" is the difference b/t
submission date and due date. It is the "lateness".
```

I simply find the number of days since the hypothetical (01,00,0000) date. Then, I can directly compute the difference in days between the submission date and the due date. This is rudimentary, but also extremely efficient. The "result" value is utilized in the below process:

```
%% Output
if (result <= 0) %Case "result" <=0, work done on time.
        fprintf('Great work! Your score is 100!\n')
elseif(result > 0 && result < 5) %Work is less than five days late and more than 0
days late.
        studentScore = 100 - result * 10; %Calculate 10% reduction per day.
        if (result == 1)
        fprintf('You were %.0f day late. Your score is %.0f\n',result, studentScore)
%Written just for aesthetics. 1 "days" would not look as nice.
        else
        fprintf('You were %.0f days late. Your score is %.0f\n',result, studentScore)
%Prints lateness and result grade.
        end
else
        fprintf('Too late! No credit is given.\n') %Case HW is five or more days late.
end
```

Here, I actually deal with the "result". Because it's defined as the number of days between the submission date and the due date, a positive result implies lateness. This is dealt with in the if structure above. In all honesty, I tried to use a switch structure here for brevity, but MATLAB doesn't like printing things from switch structures for some odd reason.

The student score after "result" days late is:

```
studentScore = 100 - result * 10
```

Also, if the result is exactly 1, I add an extra nested if loop so that the program doesn't say "1 days late", but rather "1 day late".

d. Discussion

My logic in approaching this problem is quite complicated, but I have tested it for many values, and it seems to work well. I also have included enough flexibility that the student can have assignments that last several years! Rather than just including months or days, I elected to make the program valid for any dates for which the year is more than 0.

Also, this program is an example of why it's important to check for validity of inputs. Since my algorithms depend on equalities and basic mathematical processes, numbers too large, of the wrong sign, or otherwise confusing could completely destroy any accuracy or comprehensibility of results.

I'm particularly proud of the second algorithm that compares the number of days between the two dates by simply finding their total. It's very precise, and is probably an example of how I could trim up other aspects of my code in a perfect algorithm.

2. Neighbor Identification

a. Introduction

This is a script that is tasked with creating a "pseudo-2D array" (grid) that takes three arguments: #columns, #rows, and P (coordinate of choice in the grid). Then, it determines the "neighbors" of the point, including those not immediately in contact, but also those that are on a diagonal.

b. Model and Methods

In the broadest, most sweeping explanation, I take the three inputs as mentioned above, and check for errors in input:

area = numRows * numCols; %"area" acts as not only area, but also as the highest-numbered cell (bottom right cell)

```
if ((numRows < 3 || numCols < 3) || (P < 1 || P > area))  %Logical OR to make sure grid isn't 3x3 or smaller or P is less than one or bigger than area (highest cell value)
    error('Invalid Input.'); %Produces error message.
end
```

The above code makes sure that the grid generated is bigger than 3x3 (if 3x3 or smaller, it wouldn't make sense to have neighbors- they would fill the entire grid). It also makes sure that P is within the grid. For illustration, a valid 4x4 grid has cells 1-16, so P cannot be less than one or greater than 16 (also conveniently equal to the area).

Then, I split my thinking into three parts and evaluate with one principal algorithm. I deal with corner cases (where there are exactly three neighbors), border cases not in corners (where there are always five neighbors), and center cases (where there are always eight neighbors, and, visually, the principal P cell is enclosed in a complete square of neighbors. See below algorithm for detailed explanation.

c. Calculations and Results

The entire algorithm is pasted below. See my comments below the code:

```
%% Evaluation
%Cases: Eight neighbors (inside), three (corner), five(side)
if (P == 1 || P == area || P == numRows || P == area - numRows + 1) %% Use of logical OR to determine four possible corner node situations.
    fprintf('Corner Node.\n')
    if (P == 1) %Top-right corner. If P = 1, we're in the top right corner and have three neighbors as declared below.
        n1 = P + 1;
        n2 = P + numRows; %This logic is found throughout the program. adding "numRows" references the cell immediately right of a given cell.
        n3 = P + numRows + 1;
        fprintf('Neighbors: %.0f, %.0f, %.0f', n1, n2, n3); %Displays answers.
    elseif (P == area) %Case of bottom-right. Also has three neighbor possibilities
        n1 = P - 1;
```

```
        n2 = P - numRows;
        n3 = P - numRows - 1; %Cell to left and up of P.
        fprintf('Neighbors: %.0f, %.0f, %.0f', n1, n2, n3); %Displays.
    elseif (P == numRows) %Case of bottom-left. Three neighbors.
        n1 = P - 1;
        n2 = P + numRows;
        n3 = P + numRows - 1;
        fprintf('Neighbors: %.0f, %.0f, %.0f', n1, n2, n3);
    else %case when P == area - numRows + 1 (upper right corner)
        n1 = P + 1;
        n2 = P - numRows;
        n3 = P - numRows + 1;
        fprintf('Neighbors: %.0f, %.0f, %.0f', n1, n2, n3);
    end
elseif (rem(P, numRows) == 1 || rem(P, numRows) == 0 || P < numRows || P > area -
numRows + 1) %% Border cells (five neighbors)
    if (rem(P, numRows) == 1) %We can figure out that all values of P in the top row
have remainder 1 when divided by numRows.
        n1 = P - numRows; %Five neighbors using similar logic as seen before.
        n2 = P + numRows + 1;
        n3 = P - numRows + 1;
        n4 = P + numRows;
        n5 = P + 1;
        fprintf('Neighbors: %.0f, %.0f, %.0f, %.0f, %.0f', n1, n2, n3, n4, n5);
%Neighbors printed.
    elseif(rem(P, numRows) == 0) %All cells on bottom row have a remainder of 0 when
divided by number of rows. This section deals w / P on bottom row (non-corner)
        n1 = P - numRows;
        n2 = P + numRows;
        n3 = P - 1;
        n4 = P - numRows - 1;
        n5 = P + numRows - 1;
        fprintf('Neighbors: %.0f, %.0f, %.0f, %.0f, %.0f', n1, n2, n3, n4, n5);
    elseif(P < numRows) %This deals with the leftmost-column P values (on the edge).
        n1 = P + 1;
        n2 = P - 1;
        n3 = P + numRows;
        n4 = P + numRows - 1;
        n5 = P + numRows + 1;
        fprintf('Neighbors: %.0f, %.0f, %.0f, %.0f, %.0f', n1, n2, n3, n4, n5);
    else % right side border (non-corner) process. This is the (P > area - numRows +
1) case.
        n1 = P - 1;
        n2 = P + 1;
        n3 = P - numRows;
        n4 = P - numRows + 1;
        n5 = P - numRows - 1;
        fprintf('Neighbors: %.0f, %.0f, %.0f, %.0f, %.0f', n1, n2, n3, n4, n5);
    end
else %Best for last. The interior cells. Prints eight neighbors.
    n1 = P - 1;
    n2 = P + 1;
    n3 = P + numRows;
    n4 = P + numRows - 1;
    n5 = P + numRows + 1;
```

```
    n6 = P - numRows;
    n7 = P - numRows + 1;
    n8 = P - numRows - 1;
    fprintf('Neighbors: %.0f, %.0f, %.0f, %.0f, %.0f, %.0f, %.0f, %.0f', n1, n2, n3,
n4, n5, n6, n7, n8); %Prints.
end
```

In the "breaking up" process I do, I start with corner nodes, because there are four possibilities only: top left corner (P = 1), top right corner (P = area - #rows + 1), bottom left corner (P = #rows), and bottom right corner (P = area = #rows * #columns). These are the first "if" possibilities to be checked, and each possibility is checked through the logical OR (||). Then, in nested if loops, I check to see which of the four possibilities P satisfies (if the statement is true), and then define the three neighbors. To illustrate:

```
elseif (P == area) %Case of bottom-right. Also has three neighbor possibilities
        n1 = P - 1;
        n2 = P - numRows;
        n3 = P - numRows - 1; %Cell to left and up of P.
```

This block states that if P = area (last cell; bottom right), the neighbors are above (P-1), to the left (P - numRows), and left and above (P - numRows - 1). Similar logic follows for other corners. Note that no matter the position of the corner, below the first if loop, I use fprintf to print 'Corner Node' as can be seen above.

The edge case is a little different, and involves some remainders and division. The following is a snippet that showcases the thinking:

```
elseif (rem(P, numRows) == 1 || rem(P, numRows) == 0 || P < numRows || P > area -
numRows + 1)
```

Basically, through observing the HW provided grid, we can conclude that the top border of any NxM, when divided by the number of rows, always has remainder 1. This takes care of the top row.

The bottom border case comes next, and I find that the remainder of P and numRows is always zero when dealing with the bottom row. This again can be visually seen on the provided HW graph.

The left border is dealt with in the P < numRows argument. Because the grid is indexed up-down and left-right, this, by definition, is the left border (excluding the bottom left corner, but that doesn't matter as it's already been dealt with above). Similarly, the right border is P > area - numRows + 1. This is the portion of the graph involving the entire right border except for the top-right border, but that's already been accounted for.

If P satisfies any of these conditions, its neighbors are checked with similar logic involving (n1, n2, n3) in the statement shown above for the corner. Now, however, there are five neighbors.

In the final case, I simply use the unqualified "else" to skip some hassle. I realized early on that calculating a formula for interior points would be tricky, so I simply left it as an "else" statement. The entire snipped can be seen below. Note there are eight neighbors now:

```
else %Best for last. The interior cells. Prints eight neighbors.
    n1 = P - 1;
    n2 = P + 1;
    n3 = P + numRows;
```

```
    n4 = P + numRows - 1;
    n5 = P + numRows + 1;
    n6 = P - numRows;
    n7 = P - numRows + 1;
    n8 = P - numRows - 1;
    fprintf('Neighbors: %.0f, %.0f, %.0f, %.0f, %.0f, %.0f, %.0f, %.0f', n1, n2, n3,
n4, n5, n6, n7, n8); %Prints.
```

Then, as seen, the results are printed. This exists for every possibility, and the fprintf line changes depending on the number of neighbors. See full code for example.

d. Discussion

This code is fairly robust, and, in my impression, works for any size grid larger than 3x3. It's an example of how something complex like grid neighbor calculations can be broken down into three distinct possibilities and thus evaluated. This isn't even necessarily coding! It's just the knowledge to classify and observe what possibilities can arise on the grid, and to use these classifications to translate what you see more easily into computer-comprehensible language. This approach made the process very straightforward.

Otherwise, in regards to precision errors, possibly sources of error, and/or other concerns, I believe there's nothing more to say.

3. Type of Number

a. Introduction

In problem #3, I'm tasked with creating a program that takes a six-digit number and determines if it's increasing (ex 123456), decreasing (765432), symmetrical (112211), or none of the prior (238483)

b. Model and Methods

As before, I take input:

```
number = input('Input a six-digit number\n'); %Takes input as a six-digit number.
```

Then, as an overview, I "process" the number to obtain individual digits. Then, using these numbers, I compare digits and draw conclusions. Notably, the process of checking validity of inputs happens AFTER performing an algorithm. I explain below.

c. Calculations and Results

The first algorithm is "Digit Processing". See below:

```
%% Digit Processing
digit6 = rem(number, 10); %digit6 is the first-from right number calculated using
modulus.
digit5 = rem(((number - digit6)/10),10); %I subtract the previous digit, divide by
ten, and then find the new last digit.
digit4 = rem(((number - digit6 - 10 * digit5)/100),10); % I repeat the same process,
getting the original number and stripping it of its last two digits. I now divide by
100 and use rem to get the next digit.
digit3 = rem(((number - digit6 - 10 * digit5 - 100* digit4)/1000),10); % I repeat.
```

```
digit2 = rem(((number - digit6 - 10 * digit5 - 100* digit4 - 1000 *
digit3)/10000),10); %I repeat.
digit1 = ((number - digit6 - 10 * digit5 - 100* digit4 - 1000 * digit3 - 10000*
digit2)/100000); % Similar process, but remainder function isn't necessary. This
yields the farthest left number.
if (digit1 == 0 || digit1 >= 10) % If digit1 (hundred-thousands place) is zero, the
string isn't six digits. If it's more than ten, the string is more than 6 digits.
    error('Number of invalid length');
end
```

Basically, this program gets the number and expresses it as six digits. Digit1 refers to the largest digit (leftmost), and other digits are assigned in this fashion rightward. Basically, the algorithm gets the unprocessed number, and performs a remainder function, getting the last digit. This is digit6. Next, I get the original number, subtract the rightmost digit, and then divide by ten. Then, I take the remainder of this number with respect to 10. See the following pseudocode example:

Number = 123456
Digit6 = rem(123456, 10), digit6 now equals 6
>Subtract digit6 from number
123450
Divide by ten:
12345
Take remainder again, which is now digit5

This process repeats over and over again, until all digits are found. Notably, because the number decreases by a factor of ten each time, we need to divide by larger factors. For digit5, for example, we divide by ten. For digit4, we need to divide by 100.

For digit1, it's not necessary to use mod. It's the final digit, and it has to be a single-digit integer (or, the user put in a number of incorrect length). That is dealt with in the error processing below:

```
if (digit1 == 0 || digit1 >= 10) % If digit1 (hundred-thousands place) is zero, the
string isn't six digits. If it's more than ten, the string is more than 6 digits.
    error('Number of invalid length');
end
```

Basically, if digit1 is zero, the number is too small (not six digits). If digit1 is more than or equal to ten, the number is seven or more digits. Then, I tell the user the inputs are incorrect in this case.

In the "conditional logic" section, I actually determine the nature of the number:

%% Conditional Logic

```
if (digit1 == digit6 && digit2 == digit5 && digit3 == digit4) % Symmetrical case. I
compare digits n places from left with those n places from right. If all are equal, we
print "symmetrical".
    fprintf('%.0f is symmetrical\n',number); %Prints.
elseif (digit1 < digit2 && digit2 < digit3 && digit3 < digit4 && digit4 < digit5...
            && digit5 < digit6) %Case in which digits from left to right (digit1  to
digit6) are increasing using inequalities and && (and) statements.
    fprintf('%.0f is increasing\n',number); %Prints the number.
elseif (digit1 > digit2 && digit2 > digit3 && digit3 > digit4 && digit4 > digit5...
            && digit5 > digit6) %Does opposite of previous case. Checks to see if
digits are all decreasing left to right.
    fprintf('%.0f is decreasing\n',number); %Prints.
```

```
else %Collect-all basket for any strange cases that don't fit any above descriptions.
    fprintf('The number is not decreasing, increasing, or symmetrical\n'); %Prints.
end
```

As you can see, I make extensive use of && (logical AND) to compare each digit. Then, depending on the result, the correct corresponding definition is printed.

d. Discussion

The program, totaling at 37 lines, is fairly straightforward. I believe my algorithm for finding individual digits is effective and concise. However, the only issue I have with this program is what exactly a “six digit number” is. For example, in my error-checking portion, if the farthest left digit is 0, the error is thrown. What if I input the “six-digit number” 011011? Well, my program would interpret this as an error. It depends entirely on one’s definition of “six-digit number”, and my program makes the decisive choice to treat numbers like this as invalid.