David Curry
ID: 304755606
CEE/MAE M20
April 10, 2018

# Homework 2

## 1 Lunar Phase Calculator

### 1.1 Introduction

The purpose of this script is to determine the lunar phase of any date. This can be done by knowing the difference between the inputted date and a known new moon date and knowing the number of days in a lunar revolution. The script also needs to take into account leap years by using the Julian Day number. After inputting the month date and year, the script prints out the percent illumination and whether the moon is waxing or waning.

### 1.2 Models and Methods

The script first uses the `input` function to get string values for the month, date, and year. It also checks that the input is of the correct length; month is MMM, day is DD, and year is YYYY. If they are not of correct length, the `error` function is used. An example is shown below.

```
year = input('Please enter the year as YYYY: ','s');
if (length(year)~= 4)
     error('Error: Incorrect format for year input')
     return
end
```

Next, the script uses the function `str2num` to convert the day and year to doubles. It then checks that these doubles are positive integers by using the `mod` function.

```
if (mod(day,1)~= 0 || day < 0)
     error('Error: Day must be a positive integer');
     return
end
```

Changing the month to a double requires a long `if else` statement. I also defined the maximum number of days in a month for each month. The script also produces an error if the input is not one of the all capitalized twelve months. One part of this statement is shown below.

```
if (month == 'JAN')
     mon = 1;
     max_days = 31;
elseif (month == 'FEB')
     mon = 2;
     max_days = 28;
```

Next, the script checks if the year is a leap year, and if so, changes the maximum number of days in February. It also produces an error if the value inputted is larger than the maximum number of days in a month. This is shown below.

```
leapyear = (mod(yr,4)==0 && mod(yr,100)~= 0)|| mod(yr,400)==0 ;
if (leapyear == 1 && mon == 2)
     max_days = 29;
end
if (day > max_days)
     error('Error: Too large of a value for the day');
end
```

Finally the script finds the percent illumination by converting the date to the Julian Day and using a `sine` function. Several intermediate calculations are done to make this easier. This is shown below.

```
y = yr - a + 4800;
m = mon + 12*a - 3 ;
J = day + floor((153*m + 2)/5) + 365*y + floor(y/4) -
floor(y/100) + floor(y/400) - 32045 ;
DelJ = J - 2415021 ;
T = 29.530588853 ;
L = (sin((pi/T)*mod(DelJ,T)))^2;
percent = L*100;
```

The script next determines if the moon is waxing or waning. Then it uses the `fprintf` function to print out the date, percent illumination and either waxing or waning. One part of this code is shown below.

```
fprintf('%s %s %s\n',month, days, year);
```

1.3 Calculations and Results

After inputting the day, month, and year values for the date JAN 15, 1900. The script prints out the following.

```
Please enter the month as MMM: JAN
Please enter the day as DD: 15
Please enter the year as YYYY: 1900
JAN 15 1900
Illumination = 99.3 percent
Waxing
```

1.4 Discussion

Using our model of the lunar phase calculator we can predict when the next full moon will be. The best way to do this is to input today's date APR 17 2018 and check the percent and whether it is waxing or waning. This is shown below.

```
Please enter the month as MMM: APR
Please enter the day as DD: 18
Please enter the year as YYYY: 2018
APR 18 2018
Illumination = 8.3 percent
waxing
```

Based on this output we can see that the moon is almost new but the illuminated part is getting bigger. Therefore the full moon should be around 20 days away, so I guessed APR 29 2018.

```
Please enter the month as MMM: APR
Please enter the day as DD: 29
Please enter the year as YYYY: 2018
APR 29 2018
Illumination = 98.8 percent
Waxing
```

This output shows that the moon is almost completely full and still getting brighter, so I guessed APR 30 2018.

```
Please enter the month as MMM: APR
Please enter the day as DD: 30
Please enter the year as YYYY: 2018
APR 30 2018
Illumination = 100.0 percent
waxing
```

As shown above, the next full moon should be on the April 30, 2018. The actual next full moon date is the 29th, and we are off by one because of some assumptions. Our model calculates the percent illumination right when that day starts or midnight of the night before. So our calculation for April 29th was actually for the previous night. The full moon does start on the 29th but at around 9 pm, and is full at the start of April 30th. We could improve this model by adding in the time of day into the equation because the moon does not jump from 98.8% to 100%

immediately. It moves gradually throughout the day, so our model should move gradually and not in steps.

# 2 Neighbor Identification

## 2.1 Introduction

The purpose of this problem is to determine all the neighbors of any given cell in a rectangular array. There are many cases we have to consider since corner cells only have 3 neighbors, edge cells have 5 neighbors, and all the interior cells have 8 neighbors. After the user inputs values for *M, N,* and *P* the code runs thru the cases and prints out all the neighbors of the given cell *P*.

## 2.2 Models and Methods

The script first uses the `input` function to input the values for *M, N,* and *P*. The code also checks that *M* and *N* are larger than 2 and that *P* is less than the total number of cells *M*N* with the `error` function. One part of this is shown below.

```
M = input('Please enter value for M:');
if( M < 2)
      error('Error: M must be greater than 2');
end
```

Similar lines of code are used for *N* and *P*. Next the script checks that the values are actually integers with the `mod` function. This is shown below.

```
if (mod(M,1)~= 0 || mod(N,1)~= 0 || mod(P,1)~= 0)
      error('Error: inputs must be integers');
end
```

The script then defines the eight neighbors of the cell *P*, based on *M* and *N*. These are for an arbitrary value of *P*, as the next part of the script determines which neighbors are needed. Four of the neighbors are shown below as an example.

```
N1 = P - M - 1;
N2 = P - M;
N3 = P - M + 1;
N4 = P - 1;
```

Finally, the script uses nested `if else` statements to print out the neighbors of *P*. There are 5 major cases to define, and each case may have its own smaller cases. The first `if` statement is for the case where *P* is on the left wall, the next `elseif` is for the right wall case, the next `elseif` is for the top wall, the 4th case is for the bottom wall, and finally the last `else` is for the interior cells. The left wall case is shown below, and the other cases are variations of this.

```
if (P <= M)  % for the left wall
     if (P == 1)
          fprintf('Cell ID: %.0f\n', P);
          fprintf('Neighbors: %.0f %.0f %.0f\n', N5, N7, N8);
     elseif (P == M)
          fprintf('Cell ID: %.0f\n', P);
          fprintf('Neighbors: %.0f %.0f %.0f\n', N4, N6, N7);
     else
          fprintf('Cell ID: %.0f\n', P);
          fprintf('Neighbors: %.0f %.0f %.0f %.0f %.0f\n',N4, N5,
N6, N7, N8);
     end
```

The top wall, bottom wall, and interior cells do not require a nested `if else` statement so they are even simpler.

2.3 Calculations and Results

For the array where *M = 6, N= 4, P = 4* the following is printed.

```
Please enter value for M:4
Please enter a value for N:6
Please enter a value for P:4
Cell ID: 4
Neighbors: 3 7 8
```

This value is a corner cell so only 3 neighbors are printed. Next I inputted the values *M = 6, N= 4, P = 18*. This is an interior cell so it should have 8 cells, shown below.

```
Please enter value for M:4
Please enter a value for N:6
Please enter a value for P:18
Cell ID: 18
Neighbors: 13 14 15 17 19 21 22 23
```

2.4 Discussion

This linear indexing scheme could work for a 3-D matrix, it would just be more complicated. One way to do this would be to start in the top left corner of the first plane (L=1) and follow the same path as the 2-D case. Then once the path reached the bottom right corner of that plane it would jump to the top left corner of the next plane and continue. This linear indexing convention would result in 4 different cases: corner, edge, face, and interior. This is one more case than the 2-D case, which did not have a face condition. Corner cells would have 7 neighbors, edge cells would have 11 neighbors, face cells would have 17 neighbors, and interior cells would have 26 neighbors.