Lab Assignment #10

Due 4/8/2016 at 4pm on bCourses

This assignment will be partially graded by an autograder that will check the values of the required outputs, so it is important that your function names are exactly what they are supposed to be (function names ARE case-sensitive). Instructions for submitting your assignment are included at the end of this document.

1 Writing your own trigonometric function

There is no direct way to calculate trigonometric functions generically, even on a computer. In order to find the solution to a given trigonometric function, the computer uses a high accuracy estimate. For this problem, you will code your own trigonometric function using a Taylor series and an old school table look-up. Computers actually use the CORDIC method to implement trigonometric functions, which is pretty cool. You can look into this more here, but you will not have to code it for this assignment. You can watch this video for more information on how Taylor series are derived.

1.1 Taylor series for sine

The Taylor series expansion about zero for sine (with input in radians) is:

$$\sin(x) \approx x - x^3/3! + x^5/5! - x^7/7! + x^9/9! - ... + (-1)^{(n-1)} \frac{x^{(2n-1)}}{(2n-1)!}$$

Or, in summation notation:

$$\sin(x) \approx \sum_{n=1}^{N} (-1)^{(n-1)} \frac{x^{(2n-1)}}{(2n-1)!}$$

For this problem, we will consider the N^{th} Taylor expansion of sine to be the sum of the first N terms of this series. The approximation above becomes exact as N approaches infinity.

Write a function with header:

```
function [sinSeries] = mySin(X,N)
```

which computes sine at every radian value given in the 2D array `X` (class `double`) using the N^{th} Taylor expansion for sine. The input `N` is a scalar double which is an integer greater than 0. The output `sinSeries` should be a double with the same size as `X`.

Test Cases:

```
>> a = mySin(2.4,3)

a =
```

```
    0.7596

>> b = mySin( [0, 1; 2, 3], 8 )

b =
         0    0.8415
    0.9093    0.1411
```

1.2 Order of convergence for the Taylor series

For a finite number of terms, the error of the Taylor series estimate for sine (expanded about the origin) increases the further you get from the origin. This error is often approximated based on the next term of the Taylor series, since that term is the largest that is being neglected. Thus, the 2^{nd} Taylor approximation for sine would have error that is of the order of $x^5/5!$ (the constant coefficient is often dropped and the expression written in shorthand as $O(x^5)$). For this problem, you will write a function that finds the error between your approximation for sine and the Matlab `sin` function. You will also compute the value of the next term in the Taylor series expansion for sine, so that we can compare the two to see if they actually do produce similar results.

Write a function with header:

```
function [x, err, next_term] = CompareTaylorConvergence(N)
```

which finds the error in the Nth Taylor approximation for sine, along with the absolute value of the (N+1)th term of the Taylor expansion. Your function should evaluate each of these for x data ranging from $[-\frac{N\pi}{2}, \frac{N\pi}{2}]$ in increments of $\frac{\pi}{8}$. As an output, your function should return a $1 \times (8N+1)$ row vector for each variable, where `x` is the x values where the errors were evaluated, `err` is the actual (absolute value) error between your Taylor approximation and the Matlab `sin` function, and `next_term` is the absolute value of the next term in the Taylor series.

Test Case (produces Figure 1):

```
[x, err, next_term] = CompareTaylorConvergence(2);
figure;
plot(x,[err;next_term],'Linewidth',2);
axis([-pi/2*N pi/2*N 0 2]);
title('Order of Convergence Plot','FontSize',16);
xlabel('x');
ylabel('Errors');
legend('Actual Error','Next Term','Location','Best');
```

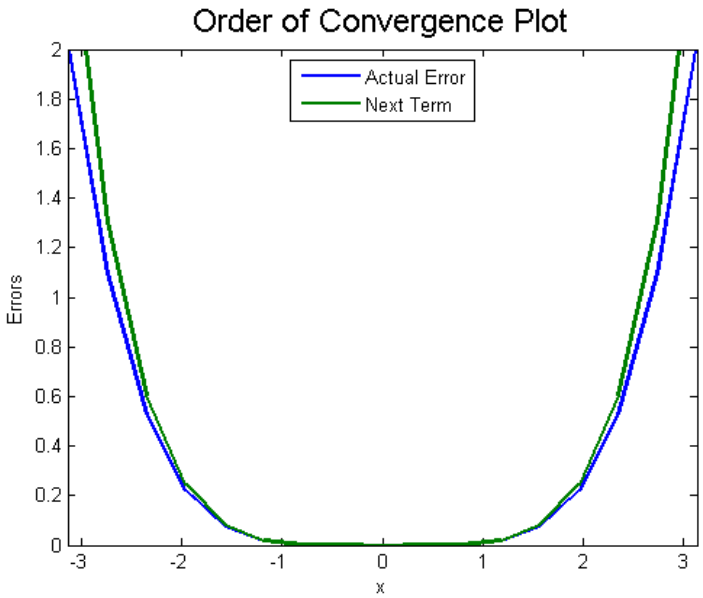

Figure 1: Test case plot for 1.2

1.3 Using the periodicity of sine

From the last problem, you can see that, far enough away from the origin, a finite Taylor expansion always diverges from the actual solution for sine. One way to improve our accuracy from inputs far from the origin is to use the periodicity of sine. i.e.

$$\sin(x \pm 2\pi) = \sin(x)$$

Write a function with header:

```
function [sinPeriodic] = mySinPeriodic(X,N)
```

which converts every radian value in the 2D matrix X to the range [-π, π] using the periodicity of sine and then calculates sine at each point using the Nth Taylor expansion for sine. The output `sinPeriodic` should be the same size as the input `X`.
Hint: You may wish to use the Matlab function `mod()` for this problem; you will have to be a bit clever about it.

Test Case:

```
>> c = mySinPeriodic([6; -2],3)

c =

   -0.2794
   -0.9333
```

1.4 Accurate range of the Taylor series

Now, we will examine how well our new and improved approximation for sine works by finding the range of values for which its accuracy is acceptable.

Write a function with header:

```
function [range] = TaylorAccRange(N, tol)
```

That finds the range of angles for which the error in the N^th Taylor expansion is less than the given tolerance, `tol`. The range should be returned as a 1×2 array containing the minimum and maximum angle in radians for which every angle in between can be found to within the given tolerance. You should consider the periodicity of sine in the function, such that angles outside of the range [-π, π] are taken as their corresponding angle within that range. (i.e. What does this imply if your tolerance range is greater than or equal to [-π, π] without considering periodicity?). If your Taylor approximation is within the tolerance for all angles, your function should return the vector $[-Inf, Inf]$.

Hint: You may wish to rewrite this problem as a root solving problem; you are free to use Matlab predefined functions (or your own functions from a previous problem set!) to solve for the root.

Test Case:

```
>> range1 = TaylorAccRange(1, 0.001)

range1 =

   -0.1818    0.1818
```

* Note that this test case gives the range in radians that the small angle approximation ($sin(x) \approx x$) is accurate to the third decimal place. This approximation is commonly used in many engineering problems to simplify the solution.

```
>> range2 = TaylorAccRange(5, 0.1)

range2 =

  -Inf   Inf
```

1.5 Table lookup

Before the widespread use of handheld calculators, it was common practice to look up the values of trigonometric functions from tables (many older math textbooks have such tables). Even today, engineers use such lookup tables for more complex functions. Engineers will typically use linear interpolation to find values between entries in such tables. In this problem, we will apply this same methodology to make a new version of the trigonometric function

we have been developing.

Write a function with the header:

```
function [sinTable] = sinLookup(X, table)
```

which finds sine at each angle (given in radians) in the 2D matrix `X` using linear interpolation of a table of sine values, given in the input argument `table`. This table can be loaded from the file `SineLookup.csv` (see test case below), which includes two columns of data. The first column contains the angle in degrees and the second column contains the value of sine at that angle (calculated out to 5 decimal places). This data table only goes from -180 to 180 degrees, so you will have to take the periodicity of sine into account in order to calculate values outside of that range. The output `sinTable` should be the same size as the input `X`.

Note: Your function should **not** make use of any predefined Matlab interpolation functions (e.g. `interp1`).

Test Case:

```
>> table = csvread('SineLookup.csv');
>> d = sinLookup([-5.4 2.3 8.4], table)

d =

    0.7727    0.7457    0.8546
```

2 Interpolation: Biomechanics

In the field of biomechanics, data interpolation is often used to define anatomical joint angles as a function of the walking cycle (clinically, this is known as the gait cycle). For example, engineers designing a prosthetic leg may be interested in analyzing the hip angle of an able-bodied individual in order to design a prosthetic leg that mimics natural leg motions. For clarity, illustrations defining hip angle (Figure 2) as well as the gait cycle (Figure 3) are provided. As shown in Figure 3, the gait cycle goes from 0 to 100% for each foot and is periodic in nature. For a given leg, an entire period of the gait cycle begins with the heel striking the floor and ends when the same heel strikes the floor again. Additionally, when the thigh and torso are completely vertical (i.e. standing perfectly vertical), the hip angle is zero degrees. Assuming a perfectly vertical torso, moving the thigh forward signifies hip flexion (positive angle) whereas moving the thigh backward signifies hip extension (negative angle).

To plot the hip angle as a function of the gait cycle, some researchers make use of state-of-the-art gait laboratories equipped with infrared cameras to get very accurate estimates of the hip angle at all instances of the gait cycle. However, most researchers do not have access to these gait labs (they cost hundreds of thousands of dollars) and instead resort to less costly methods of calculating hip angles. For instance, a graduate student researcher

may simply take pictures of a human subject at specific instances of the gait cycle and use a protractor on each image to estimate the hip angle at each specific moment. Once the researcher has obtained a few data points, they may then use software such as MATLAB to interpolate between these points. That way, the hip angle is now known across the entire gait cycle and not just at a few discrete points.

In this problem, you are asked to write a function that performs the interpolation mentioned in the previous paragraph. You are provided with a data file `GaitLabData.mat` which contains four double arrays: `crude_gait_cycle`, `crude_hip_angles`, `ideal_gait_cycle`, and `gait_lab_hip_angles`. The arrays `crude_gait_cycle` and `crude_hip_angles` correspond to the gait cycle instances and hip angles calculated using a protractor and a traditional camera. The array `ideal_gait_cycle` corresponds to the values of the gait cycle in which it is desired to have interpolated hip angle values. Thus, the number of elements in `ideal_gait_cycle` is much larger than the number of elements in `crude_gait_cycle`. The array `gait_lab_hip_angles` contains data from an actual gait laboratory, which we can use to evaluate the effectiveness of different interpolation methods (the more closely the interpolation matches the gait lab data, the better it is).

Write a function with the following header:

```
function [interpolated_angles] = gait_data_interp(crude_gait_cycle,...
    crude_angles, ideal_gait_cycle, interp_method)
```

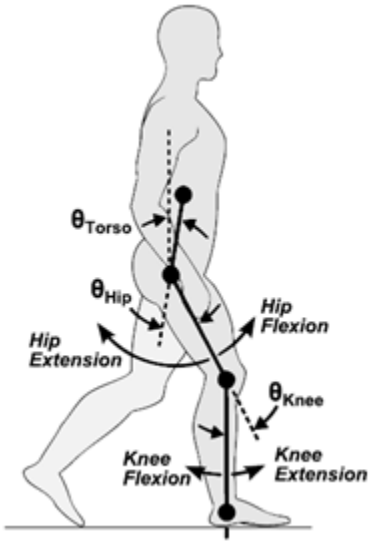

Figure 2: Definition of Hip Angle

Your function will take inputs of the crude gait cycle points, the crude hip angles, the ideal gait cycle points, and finally a character array specifying the interpolation method. Your function should be able to perform linear, cubic, or spline interpolations, which will correspond to values of `interp_method`: `linear`,`cubic`, and `spline`. Using the MATLAB

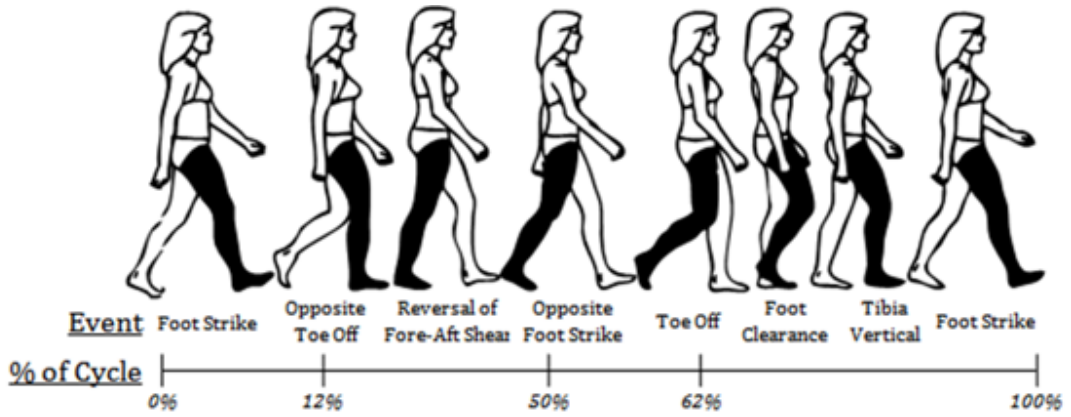

Figure 3: Definition of Gait Cycle

function `interp1`, the output of interpolated hip angles should be produced. If an interpolation method is called that is not linear, cubic, or spline, the function should output an empty array `[ ]` and display "Please input either linear, cubic, or spline." Note that the `spline` option for the MATLAB `interp1` function performs a traditional cubic spline, as discussed in lecture and the textbook, while the `cubic` option performs a more sophisticated method that never locally overshoots its target. You do not need to worry about the details of this method for this class, but you are free to look into it more if you are interested. Once you have created your function, the following test case should result in a figure that is identical to Figure 4:

Test Case:

```
load('GaitLabData.mat')
angle_linear = gait_data_interp(crude_gait_cycle, crude_hip_angles,...
    ideal_gait_cycle, 'linear');
angle_cubic = gait_data_interp(crude_gait_cycle, crude_hip_angles,...
    ideal_gait_cycle, 'cubic');
angle_spline = gait_data_interp(crude_gait_cycle, crude_hip_angles,...
    ideal_gait_cycle, 'spline');

figure
%Plot crude data points
plot(crude_gait_cycle, crude_hip_angles, 'k.', 'MarkerSize', 30);
hold on;
grid on;
%Plot linear interpolation
plot(ideal_gait_cycle, angle_linear, 'g', 'LineWidth', 2);
%Plot cubic interpolation
plot(ideal_gait_cycle, angle_cubic, 'r', 'LineWidth', 2);
%Plot spline interpolation
```

```
plot(ideal_gait_cycle, angle_spline, 'b', 'LineWidth', 2);
%Plot the actual gait lab data
plot(ideal_gait_cycle, gait_lab_hip_angles, 'k', 'LineWidth', 2);

xlabel('Percent Gait Cycle', 'FontSize', 16);
ylabel('Hip Angle (Degrees)', 'FontSize', 16);
title('Interpolation Methods for Hip Angle Analysis', 'FontSize', 20);
legend('Crude Data', 'Linear Interpolation', 'Cubic Interpolation',...
    'Spline Interpolation', 'Gait Lab Data', 'Location', 'SouthEast');
```

You should be able to see the ability of each interpolation method and how it compares to the extremely accurate data that can be obtained from a gait laboratory. Can you make sense of the data based on the definition of the gait cycle and the definition of hip flexion and hip extension?

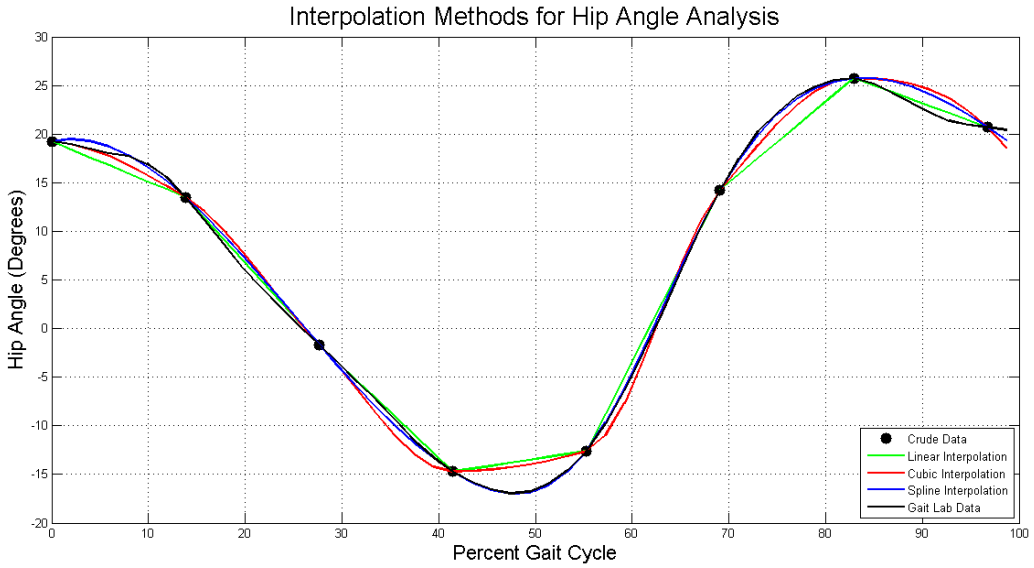

Figure 4: Problem 2 Interpolation Results

3 Interpolation: Soil Layers

Civil engineers often collect and analyze soil samples to better understand the properties of the soil and groundwater at a site before beginning a construction project. This data is often collected by drilling vertical boreholes. This allows engineers to bring soil samples back to the lab for testing of various properties and also to distinguish layers of different soil types as well as the depth to the water table. One of the key challenges remaining is that soil and groundwater properties can only be collected at a limited number of points on the site, while the properties across the entire site are extremely relevant to the design process. For this problem, you will implement a cubic spline that will interpolate the soil layering geometry

across a cross-section of the site. In other words, you will be interpolating the vertical (y) coordinate of the top and bottom of each layer at horizontal (x) coordinates where there is no data (the top of a given layer is also the bottom of the layer above it).

Your function should take a matrix of data that matches the format in the file `Borings.csv`. Before you attempt to write this function, open up the file and look at how the data is formatted. Note that the water table and soil layers are measured in terms of depth from the surface. This means that you will have to subtract these values from the surface elevation to convert them to elevations within a consistent reference frame. Once you've done this, you should perform 1D cubic spline interpolation to get new values for the surface, water table, and soil layer elevations. Lastly, you should convert these new values for the water table and soil layers back into depths and format them in a matrix similar to the input.

You should also notice that the depth to the clay layer is missing in the data for two of the boreholes. This means that the clay layer is absent in the locations of those boreholes. If a layer is present in one borehole and absent in the next, your function should use extrapolation to map that layer until it ceases to exist. When two layer interfaces intersect (i.e. the thickness of a layer becomes zero), this indicates that the layer no longer exists (see Figure 5). In other words, your function should find the horizontal end point(s) of a layer which does not span the entire cross-section. Past this point, your function should return `NaN` for that layer. Your function should be able to handle missing data in any layer, not just the clay layer in the given example. However, you may assume that the data will only be missing at the beginning and/or end of the data series (to program a function that would accomplish this generically, you would have to parse out the data, which is beyond the scope of this problem). You may also assume that any soil layer with missing data is sandwiched between two soil layers with no missing data.

3.1 Interpolating the data

Your function should have this header:

```
function [site_data] = mySoilSpline(borehole_data, site_locations)
```

where `site_locations` is a $1 \times N$ matrix of horizontal (x) locations along the cross section (in meters) where you will interpolate the borehole data, `borehole_data` is an $M \times B$ matrix of soil data at the boreholes, and `site_data` is an $M \times N$ matrix of the soil data at the interpolated locations, where M is the number of rows of data within `Borings.csv`, N is the number of site locations, and B is the number of boreholes drilled (i.e. columns in `Borings.csv`). Each row of `site_data` should refer to the same type of information as the corresponding row of the data in `Borings.csv`.

Hint: You are free to use Matlab predefined functions to do the cubic spline interpolation.

Test Case:

```
>> borehole_data = csvread('Borings.csv',1,2);
>> interpolated_layers = mySoilSpline(borehole_data, ...
```

```
    [1350, 1450, 1500, 1700])

interpolated_layers =

   1.0e+03 *

    1.3500    1.4500    1.5000    1.7000
    0.3546    0.3556    0.3545    0.3587
    0.0126    0.0120    0.0100    0.0143
    0.0048    0.0044    0.0033    0.0058
       NaN       NaN    0.0186    0.0170
    0.0237    0.0220    0.0210    0.0393
    0.0492    0.0449    0.0422    0.0625
```

Note that in reality, it is very difficult to accurately predict the soil layering geometry in between data points (boreholes). Without any further information, straight lines are usually drawn, along with "question marks" indicating high levels of uncertainty. In cases where additional accuracy is necessary, the judgment of a geologist or geotechnical engineer with experience mapping the local geology is utilized. Sometimes more sophisticated geostatistical interpolation techniques are used. For this problem, we use cubic spline interpolation due to its simplicity and to arrive at smooth pictures like the one shown in Figure 5.

3.2 Visualizing the interpolated data

A function has been provided on bCourses that uses a working soil spline function to generate a soil profile. These profiles are commonly made to visualize the soil layer geometry along an entire cross section. To run successfully, both your spline function and the `Borings.csv` data file must be in the same folder. It may be helpful to look at (or modify) this code, but it is only there to help you develop your function and will not be used for grading.

```
>> boring_data = csvread('Borings.csv',1,2);
>> SoilViewer(boring_data)
```

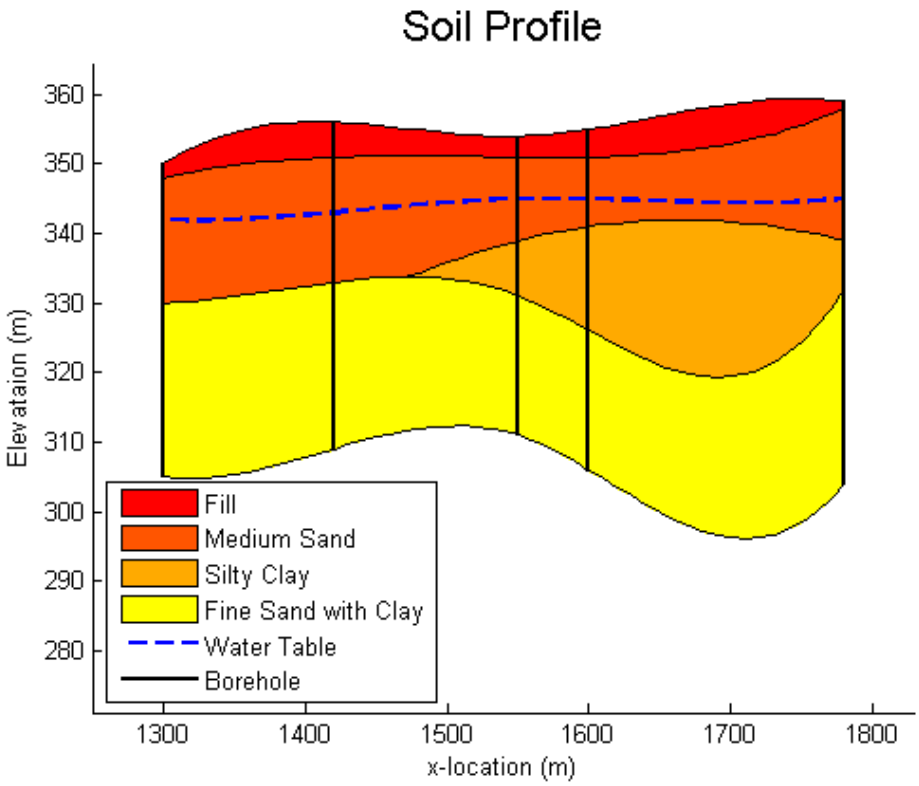

Figure 5: Example output from the soil viewer

Submission Instructions

Your submission should include the following following function files in a single zip file named Lab10.zip

- `mySin.m`
- `CompareTaylorConvergence.m`
- `mySinPeriodic.m`
- `TaylorAccRange.m`
- `sinLookup.m`
- `gait_data_interp.m`
- `mySoilSpline.m`

Note: you do not need to include any figure files for this lab.