E296MA
Dr. Gabriel Gomez
GSI Spencer Frank
Fall 2017

MATLAB Tutorial 1 - Solving Linear Systems, Plotting and Visualization, and Objects

MATLAB is a user-friendly computational tool used throughout academia and industry. It has countless powerful toolboxes, and allows the user to quickly write code to expedite mathematical computation. The advantage of MATLAB over other scientific computing software, such as Python, is the ease of use. For example, almost all the necessary software packages are built into MATLAB and rarely does the user need to load particular packages. This make computation fast and easy.

In this tutorial, we will go over:

1. Importing Data

2. Visualization (2D and 3D)

3. Curve fitting

4. Solving linear systems

5. Object Oriented Programming

Download Tutorial from gitHub

1. Create a github account at github.com

2. search for spencerf/MATLABtutorial1

3. clone (or fork) the gitHub repository to your local folder

4. Open MATLABTutorial1.pdf

1 Data Import, Visualization, Curve Fitting

1.1 Data Import and Visualization

Import data from the internet by pasting the following into Matlab:

```
api = 'http://climatedataapi.worldbank.org/climateweb/rest/v1/';
url = [api 'country/cru/tas/year/USA'];
S = webread(url);
```

This will load a structured array S with information about the year and the temperature data. This data can be extracted with the following syntax

```
years = [S.year];
temps = [S.data];
temps = 9/5 * temps + 32; %Convert to Fahrenheit
yearstoplot = datetime(years,1,1); %Convert years to 'datetime'
```

Note that the brackets are necessary with this type of structured array. Without the brackets you will only access the last data entry. The last two lines convert the temperature data to Fahrenheit and convert the year data to a standard datetime format that can be directly read by MATLAB.

To plot the data

```
%Plot data
figure
plot(yearstoplot, temps,'ok');
title('USA Average Temperature 1901-2012')
xlabel('Year')
ylabel('Temperature (^{\circ}F)')
xmin = yearstoplot(1);
xmax = yearstoplot(end);
xlim([xmin xmax])
```

USA Average Temperature 1901-2012
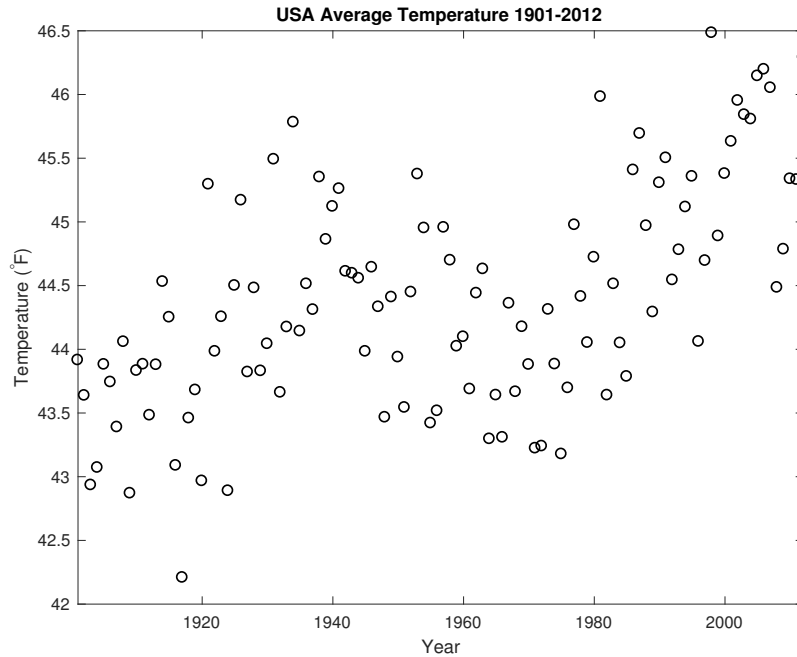
1.2 Split Data into Training and Test Set

In order to text the prediction capabilities of a model, it is important to segment out two sets of data. The training data consists of the data that we will use to 'train' our model. This is the data that we will fit a curve too. The other set of data is called the 'test' data. This dataset is used to evaluate the predictive capability of our trained model.

In our case we will split the data at the year 1990; everything before 1990 will be considered 'training' data, and everything after 1990 will be considered 'test' data. You can use the following code to segment the test and training data.

```matlab
%% Split data into training set and test set
yearfortraintest=1990;

indices_train=years<=yearfortraintest;
years_train = years(indices_train);
temps_train = temps(indices_train);
yearstoplot_train=yearstoplot(indices_train);

indices_test=years>yearfortraintest;
years_test = years(indices_test);
temps_test = temps(indices_test);
yearstoplot_test=yearstoplot(indices_test);
```

1.3   Curve Fitting on Training Data

Now, we will fit a model to our training data. We will use different degree polynomials. The simplest polynomial is a line, i.e. y=mx+b. To fit a line to our data we can use the following code:

```
%Fit 1−degree polynomial
[p1_all,~,mu1_all] = polyfit(years_train,temps_train,1);
p1temps_train = polyval(p1_all,years_train,[],mu1_all); %evaluate polynomial
hold on; xlim([yearstoplot_train(1) yearstoplot_train(end)]);
f1 = plot(yearstoplot_train, p1temps_train,'r');
xlim([yearstoplot_train(1) yearstoplot_train(end)]);
R_squared_train1=1−sum((p1temps_train−temps_train).^2)/...
    (((length(temps_train)−1) * var(temps_train)))
norm2_train1=norm(p1temps_train−temps_train,2)
```

There is also a nice interface for curve fitting in Matlab, called *cftool*. This is a user friendly interface that lets you easily fit data to a user-specified function. Try it out by typing *cftool* into the command line.

Try: Use either polyfit or cftool to fit 2nd and 3rd degree polynomial to the training data. Try the *center* and *scale* option to find a better conditioned polynomial. Plot all three polynomials on the same axis.

Try: Obtain the R-squared value of each polynomial

Try: Obtain the 2-norm value of each fitted function. The 2-norm (a.k.a. Euclidean Norm) of a vector $x_i = f_i - y_i$ is $||x||_2 = \sqrt{\sum_i (x_i)^2}$.

You should end up with a plot that looks like the following:
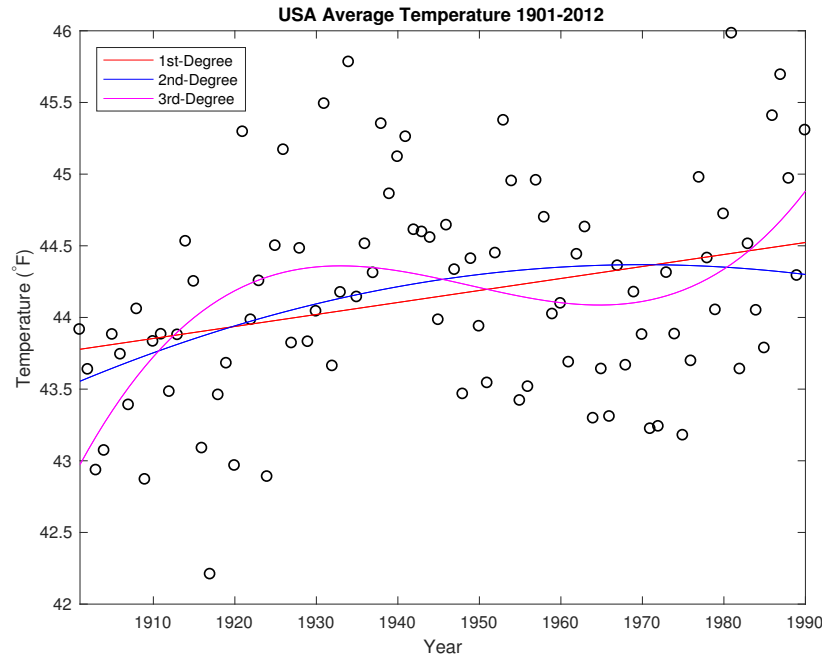
Q: Which function do you think best describes the data? Why?

1.4 Prediction

To evaluate which curve best describes the data, we need to think about not just how well the curve describes previous data, but how well the curve describes as of yet unknown data. To do this, we split our data into training and test sets, pretending that we do not know what happens after the year 1990. Now, we use the test data to evaluate how well predictions based on our curve fits describe what happens in the period from 1991-2012.

To do this, we evaluate our polynomials for years 1991-2012, and compare to the actual measured data. This can be done using the following code

```
%test 1−degree polynomial
p1temps_test = polyval(p1,years_test,[],mu1); %evaluate polynomial
hold on; xlim([yearstoplot(1) yearstoplot(end)]);
f1 = plot(yearstoplot_test, p1temps_test,'r−*');
norm2_test1=norm(p1temps_test−temps_test,2)
```

We have also calculated the 2-norm to evaluate how well our predictions describe the test data.

Try: Calculate predictions in the 1991-2012 period using the 2nd and 3rd degree polynomial to the training data. Plot all three polynomials on the same axis.

5

Try: Obtain the 2-norm value of each fitted function. The 2-norm (a.k.a. Euclidean Norm) of a vector $x_i = f_i - y_i$ is $||x||_2 = \sqrt{\sum_i (x_i)^2}$.

Now you end up with a figure that looks like the following:
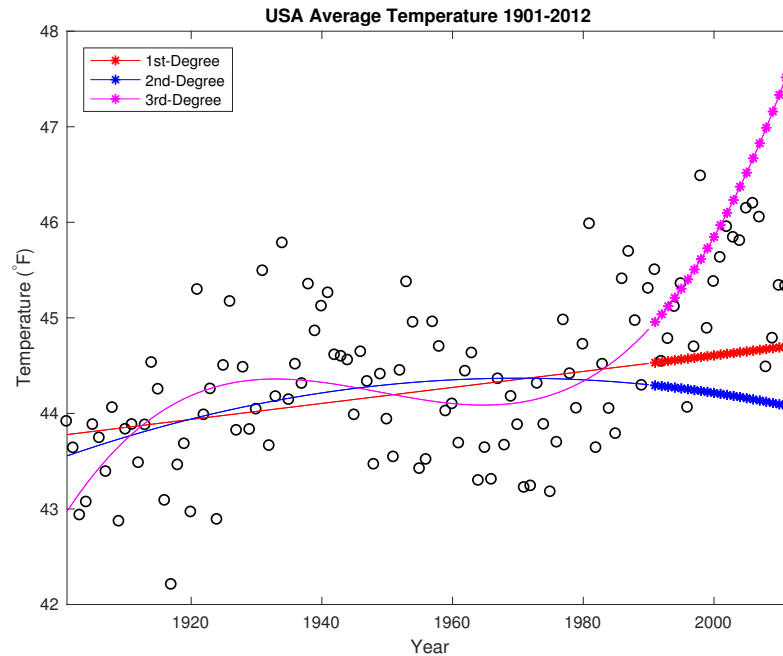
Q: Now, which function do you think best describes the data? Why?

1.5 Homework

Perform a curve fit on the entire dataset (years 1901-2012) for the 1st, 2nd, and 3rd degree polynomial. For each of the three functions, extrapolate predictions from 2013 to 2050. Plot the three predictions and comment on which function you believe gives the most realistic predictions for the period 2012-2050. Turn in with Homework set 3.

2 Gaussian Distributions

In this section we will go over statistical models. Specifically, we will go over how to use data to define a Gaussian distribution in MATLAB. The dataset we will use is measurements of height and weight of 25000 subjects in Hong Kong[1].

2.1 Data Import

This time we will import data from a .mat file that was previously created and saved. Ensure that the .mat file is in your current directory.

```
load humandata25000.mat
index=data(:,1);      %subject index
height=data(:,2);     %inches
weight=data(:,3);     %pounds
```

Once the data has been imported, we can plot and visualize it wit the following commands

```
plot(height,weight,'o'); hold on;
title('Weight vs. Height')
xlabel('Height (in)'); ylabel('Weight (kg)');
```

You should end up with a plot that looks like
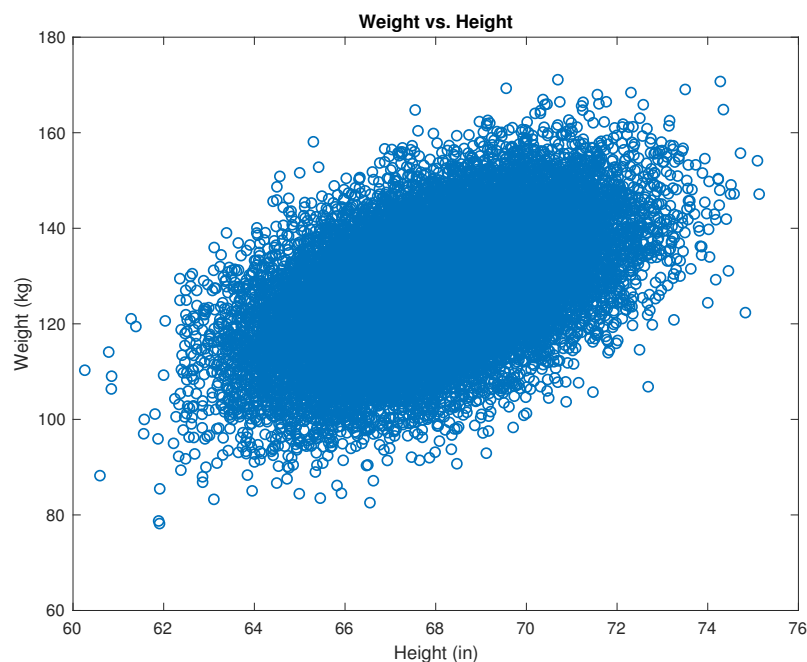
2.2 Random Sampling

We now take random samples from our large dataset of 25000 subjects to develop progressively more accurate distributions of the population. Using the code below, we can draw four histograms created with samples of 5, 10, 100, and the entire dataset of 25000. Also, log the mean of each of these samples for later use.

```
%% Random Sampling of Gaussian − Heights
samples =[5 ,10 ,100 ,25000];
xlimits_height =[min( height ),max( height )];
figure (1);
randsamples=randi ([min(index) max(index)], samples (1) ,1);
subplot (5 ,1 ,1); h25=histogram ( height (randsamples ) ,10);
title (sprintf ('Histogram %d Samples ', samples (1)))
ylabel ('Frequency ')
xlim ( xlimits_height )

randsamples=randi ([min(index) max(index)], samples (2) ,1);
subplot (5 ,1 ,2); h250=histogram ( height (randsamples ) ,10);
title (sprintf ('Histogram %d Samples ', samples (2)))
ylabel ('Frequency ')
xlim ( xlimits_height )

randsamples=randi ([min(index) max(index)], samples (3) ,1);
subplot (5 ,1 ,3); h2500=histogram ( height (randsamples ) ,25);
title (sprintf ('Histogram %d Samples ', samples (3)))
ylabel ('Frequency ')
xlim ( xlimits_height )

randsamples=randi ([min(index) max(index)], samples (4) ,1);
subplot (5 ,1 ,4); h25000=histogram ( height (randsamples ) ,50);
title (sprintf ('Histogram %d Samples ', samples (4)))
ylabel ('Frequency ')
xlim ( xlimits_height )
```

You'll end up with plots that look like the top four of the figure below.

2.3 Fitting a Gaussian to Data

We now wish to describe the height distribution with a continuous Gaussian distribution. We do this by defining a probability density function (pdf) based on all 25000 subjects. This is done with the following

```
%% fit gaussian to data
pd_height = fitdist(height, 'Normal');
prob_height=pdf(pd_height, xlimits_height(1):0.1: xlimits_height(2));
subplot(5,1,5); plot(xlimits_height(1):0.1: xlimits_height(2),prob_height)
title('Probability Distribution Function')
ylabel('Probability')
xlabel('Height [in]')
xlim(xlimits_height)
```

The fitted gaussian is plotted in the 5th frame of the figure below.
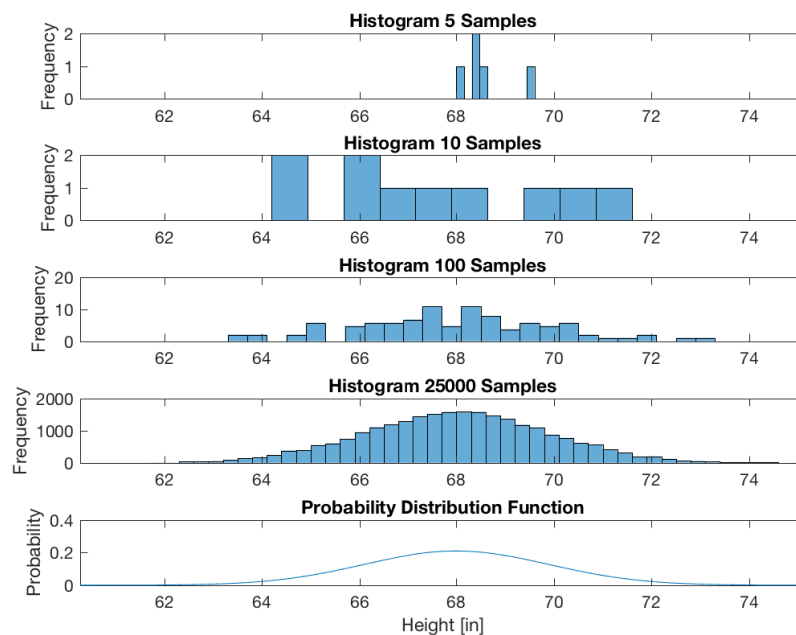
2.4   Law of Large Numbers

The law of large numbers postulates that as the number of samples of a population increases, the mean will approach the theoretical mean of the population. To demonstrate this we will take random samples of size N from our PDF. We take the mean of each of these samplings and plot the mean vs. sample size. This can be done with the following code

```
figure(2);
%calculate the mean of N random samples taken from the pd
N=1:10:10000;
for i = 1:length(N)
    samplemean(i)=mean(random(pd_height,[N(i),1]));
end
```

```
semilogx (N, samplemean ) ;
title ( 'Sample Mean vs . Number of Samples ')
xlabel ( 'Number of Samples ')
ylabel ( 'Mean ')
meanline=refline ([0 pd_height .mu]) ;
meanline . LineStyle='——';
```

Your plot will show that as the sample size increases, the mean converges to the population mean, shown by the dotted line. Note that in our case we have assumed the population mean is equivalent to the mean of the PDF.
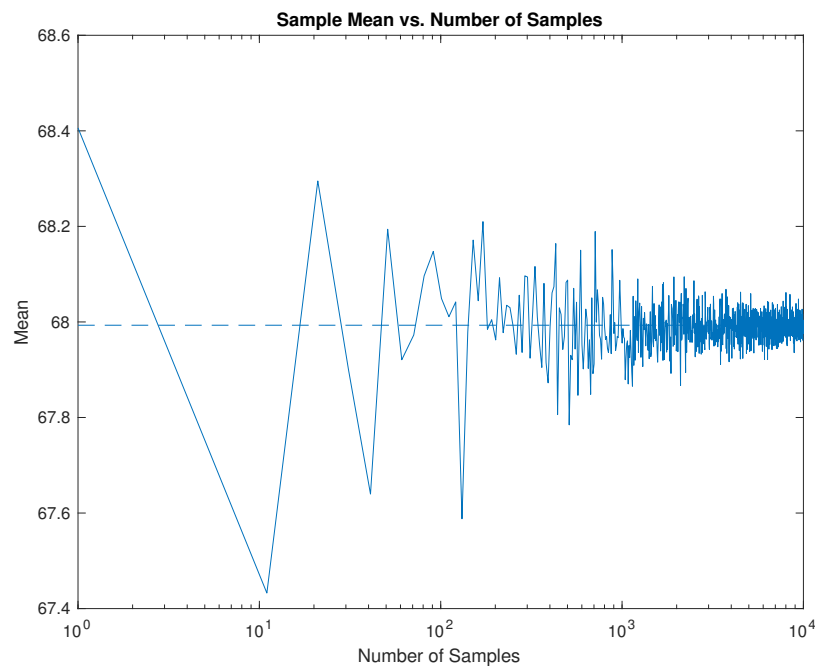
2.5   3D Plotting - Histograms

Although we have focused on the height data, the weight data is also of interest. To visualize the relationship between height and weight, we create a 3D histogram using the following code

```
%% 3D Histogram
figure ;
[a,b]=hist3 ([ height , weight ] ,[50 ,50]) ;
surf(b{1},b{2},a) ; shading interp ; axis tight
title ( '3D Histogram ')
xlabel ( 'Height (in )'); ylabel ( 'Weight (kg )'); zlabel ( 'frequency ')
```
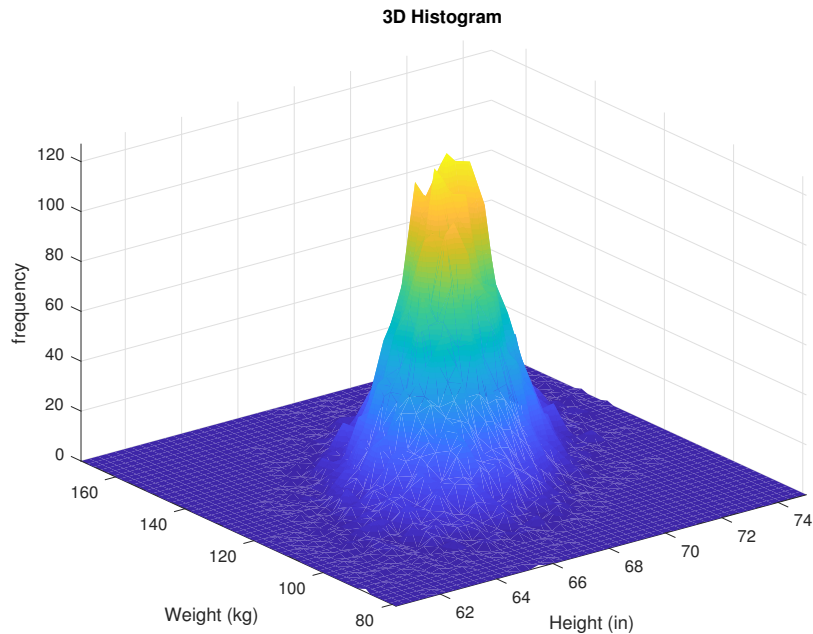
3D Histogram

If we visualize the 3D histogram from the top, it is evident that there is a strong relationship between weight and height. This 3D histogram is another representation of the data depicted in the first figure, which shows a scatter plot of height and weight.

2.6   Matrix Inversion and Linear Regression

If we wish to find the linear relationship between weight and height, we can carry out a linear regression. This can be done as in the first part of this tutorial using **polyfit**, however to demonstrate matrix inversion, we will find the best fit line using matrices.

First, in general a line (or plane in 3D, or hyperplane in multiple dimensions) can be described by the matrix system

$$Ax = y.$$

In this case, vector x can be thought of as the 'coefficients' of the line. We can solve for these coefficients by inverting matrix A we can solve for x

$$x = A^{-1}y.$$

Generally, a matrix must be square to find its inverse. However, in many cases A will not be square. When A is not square we can find what is called its 'pseudoinverse' A^+. This can

be found by multiplying $Ax = y$ by A^T. Because $A^T A$ is a square matrix, it is invertable

$$Ax = y$$
$$A^T Ax = A^T y$$
$$x = (A^T A)^{-1} A^T y$$
$$x = A^+ y.$$

Finding the pseudoinverse is akin to finding the best fit solution to an over or underdefined system (no unique solutions vs. many unique solutions).

The rigorous, the mathematical theory behind the pseudoinverse is beyond the scope of this class. The thing worth remembering is that finding the inverse of a non-square matrix is analogous to finding the best fit solution to the system, and is equivalent to linear-regression.

Now we will use MATLAB to find the pseudoinverse of A. Note that the 'backslash \' operator, which we will use will automatically decide if the matrix is square, under, or overdefined, and will calculate either the regular inverse or pseudoinverse accordingly.

To make our matrix system specific to our problem we define A as a N by 2 matrix that contains the known height data and a column of 1s, called [1]. The column of 1s allows us to add a constant coefficient to our vector x. This is equivalent to adding a y-intercept to the equation of a line: $y(:) = A(:,1) \cdot x(1) + 1 \cdot x(2)$, because $(A(:,2) = [1])$. Last we define y as an N by 1 vector that contains the weight data.

$$A = [height_{data}, [1]]$$
$$x = [x(1); x(2)]$$
$$y = [weight_{data}]$$

To solve for our coefficient vector x we invert A and multiply by y. This can be done using the following code, which calculates the pseudoinverse using the 'backslash \' operator

```
%A*x=y  or  height*x=weight.  To  solve  for  x,  we  invert
A=[height , ones( size ( height ))];  %N  by  2  matrix
y=[weight ];  %N  by  1  vector
x=A\y;  %this  is  the  matrix  inversion  step.  In  this  case  we  are  calculating  th
y_pred=polyval(x, height );
plot ( height , y_pred )
legend ( 'data ' , 'Linear  Fit ')
```

The resulting line, looks like
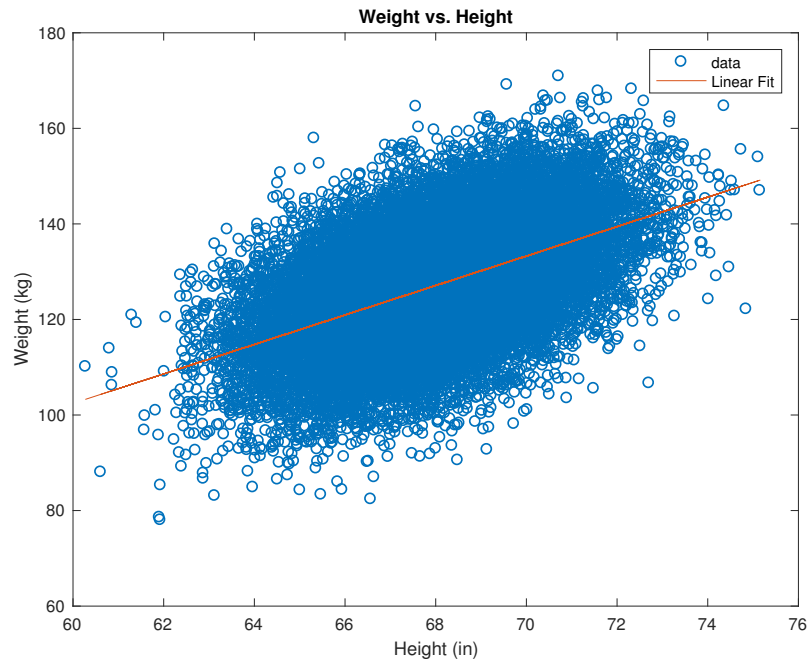
Weight vs. Height

2.7 Homework

Using the weight data (instead of height) create the four histograms, fit a gaussian distribution, and plot the sample mean vs. sample size (law of large numbers). Your figures should be similar to the height histogram and mean height vs. sample size figures above.

3 Object Oriented Programming

Consider an ant colony of N ants. Each of these ants has 6 legs, weighs less than a gram, and needs to eat. With a little bit of thought, you can draw up a blueprint of an ant that specifies all of the general properties of an ant, this general blueprint is called a *class*.

Now, imagine we have 100 ants; each of these ants have the same blueprint, however, each ant is an individual. Each individual ant has specific properties such as how hungry it is, how far it can see, and how social it wants to be.

Each individual ant can be modeled as an *object* (a.k.a. *instance*) of the Ant *class*. To model an entire colony, we simply create N objects. The concept of *object* oriented programming (OOP) allows us to model this type of problem.

The task of this tutorial is to model the behavior of an ant colony on N ants. You are provided a predefined *class* called antDef. Using this class *class* we show how to create *objects* that define the individual ants. After creating the *objects*, will will run a movement simulation based on the individual properties of ants, such as its hunger and its social desire, desire (i.e. how close does it want to be to other ants).

After demonstrating how to use OOP to run an ant Colony simulation, you will be asked to add functionality to the antDef *class* to obtain an desired Colony behavior.

3.1 The antDef *Class*

Open the class definition entitled antDef. In this file you will find three important groups in blue: classdef, properties, and methods. classdef is simply the name of the *class*, properties are the characteristics of a general ant, and methods are the operations that can be carried out on an ant *object*.

```
    properties
        loc              % ant location
        xlim             % x−boundaries
        ylim             % y−boundaries
        foodloc          % location of food
        speed            % walking speed of ant
        vision           % distance ant can see other ants that he/she wants to b
        foodDesire       % desire for food (weighting term);
        friendDesire     % desire for friends (weighting term);
        foundFood        %specified whether an ant has found food, and which loca
        maxAntsEating    %maximum number of ants that can eat at a single food s
        randMovement     % magnitude of small random movements;
    end
```

The methods refer to the actions that the ant can take. The methods are not listed in this document due to length. Refer to the antDef.m file for details. The first method is getMove, which determines the move that the ant will take, based on how close the ant is to other ants, and how close the ant is to food.

3.2   Defining Individual Ant *Objects*

To define an individual ant *object*, we use the following line:

```
ant=antDef
```

which produces the following output:

```
ant =

  antDef with properties:

                 loc: []
                xlim: []
                ylim: []
             foodloc: []
               speed: []
              vision: []
          foodDesire: []
        friendDesire: []
           foundFood: []
       maxAntsEating: []
        randMovement: []
```

The empty brackets show that the properties of the individual ant have not yet been defined. These can be define using the following syntax

```
ant.loc=[0,0]
ant.speed=0.1
```

Now if we type ant in the command line we get

```
ant =

  antDef with properties:

                 loc: [0  0]
                xlim: []
                ylim: []
```

15

```
       foodloc:  []
         speed:  0.1000
        vision:  []
    foodDesire:  []
  friendDesire:  []
     foundFood:  []
maxAntsEating:  []
 randMovement:  []
```

Now open AntSwarm.m

We can quickly define an ant colony with the following code:

```matlab
%define domain that Ants can exist in
xlim=[-1, 1]; %[m]
ylim=[-1 ,1]; %[m]

% Create Ant Objects(aka instances) and Initialize Locations
N_ants=40; %define number of ants in colony (i.e. number of objects)

f1=figure(1);
for i=1:N_ants
    antColony(i)=antDef; %Create ant object (N_ants is the number of individu
    antColony(i).loc=(rand(1,2)-0.5).*[xlim(2)-xlim(1),ylim(2)-ylim(1)]; %ran
    plot(antColony(i).loc(1),antColony(i).loc(2),'.k'); hold on
end
```

The for loop allows us to quickly create N_ants *objects*. In this case we are creating 40 *objects*. We have also initialized the location of each ant randomly, on the plane defined by xlim and ylim.

To define the other geometric properties, we can use the deal command. This command acts as a for loop, and quickly cycles through all *objects* in an *object* array to define the properties of each object. Think of 'dealing' cards to each object. We have also defined four locations of food, and have given this information to each ant.

```matlab
% Specify to ant the boundaries of the domain
[antColony.xlim]=deal(xlim);
[antColony.ylim]=deal(ylim);

%Specify food locations
food_x=[-0.75  0.75  0  0];
food_y=[0  0  0.75  -0.75];
[antColony.foodloc]=deal([food_x(:),food_y(:)]);% the 'deal' command allows y
```

3.3 Simulating Ant Movements

To simulate the ant movements, we first need to define the rest of the properties. Without defining these properties, the getMove function within antDef will not have adequate information to run. Below, we have assigned the same properties to each ant, however, we could easily assign different properties to individual ants. This can be achieved by specifying *objects* in the *object* array. For example if we wish to assign the first five ants a different max speed than the others, we write [antColony(1:5).maxSpeed]=deal(0.1).

```
%% Run movement simulation
dt=1; %[s]
t_final=100; %[s]
tvec=0:dt:t_final; %[s] vector of discrete times

%Properties
[antColony.speed]=deal(0.1);          %[m/s] max speed an ant can walk (how fast
[antColony.vision]=deal(0.2);          %[m] the distance an ant can see (how f
[antColony.foodDesire]=deal(0);        %desire for food (how hungry is the ant
[antColony.friendDesire]=deal(0.5);     %desire for friends (how much does an
[antColony.foundFood]=deal(0);          %initialize foundFood to 0
[antColony.maxAntsEating]=deal(10);        %max number of ants that can eat
[antColony.randMovement]=deal(0.1);    %random movements (how wobbly is an ant
```

dt is the timestep of each movement. t_final specifies the terminal time of of the simulation. tvec is the vector that specifies the discrete times of the simulation. The tunable parameters specify the speed of the ant, the vision of the ant, foodDesire the ants desire for food, friendDesire the ants desire for friends, foundFood specifies which pile of food the ant is at (1,2,3, or 4), maxAntsEating specifies the maximum number of ants that can eat at a given pile of food, and randomMovement. The random movements are not important, and essentially prevent the ant from getting stuck.

```
for k=1:length(tvec)
    cla% clear axis
    t=tvec(k); %print time

    [antColony.foodDesire]=deal([antColony(1).foodDesire]+0.01); %Increase fo

    for i=1:N_ants
        antColony(i).getMove(antColony(i),antColony,dt)
        plot(antColony(i).loc(1),antColony(i).loc(2),'.k','MarkerSize',10); h
    end

    %plot
    fplot=plot(food_x,food_y,'xr'); %plot food
    axis([xlim ylim])
```

```
    %make movie
    frame(k)=getframe;

end

%Count occurances
tabulate([antColony.foundFood])
```

To run the simulation, press the 'play' button in MATLAB. The for loop will simulate the movement of the ant colony for the times specified in tvec. The nested loop cycles through each individual ant and updates each ant's location based on the output from getMove. The output of the simulation should look like the Figure 3.3. Each ant is denoted by a black dot, and the food location is denoted by a red 'x'. The tabulate function counts how many ants are at each food location. It will produce this output that looks like this, except the ants will not be evenly distributed
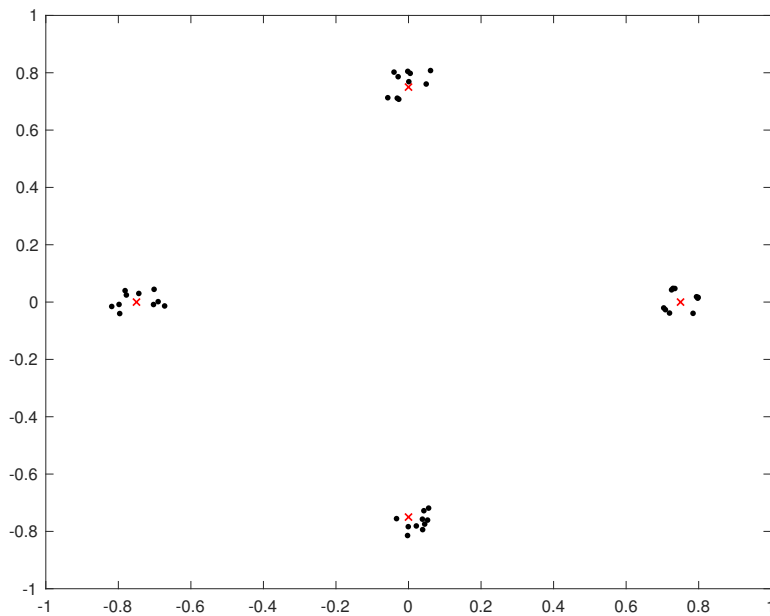
Value	Count	Percent
1	10	25.00%
2	10	25.00%
3	10	25.00%
4	10	25.00%

Note that we have added one bit of functionality in the for loop:

18

```
\lstinline|[antColony.foodDesire]=deal([antColony(1).foodDesire]+0.01);|.
```

This line increases the food desire of the entire ant colony as time increases.

3.4 Homework

Take a colony of N_ants = 40. Leave all ant properties set as their default. You'll notice in running the movement simulation that the ants are not even distributed between the four food locations. However, this is not optimial, as only 10 ants can eat at one time at each location. This property was previously set antColony.maxAntsEating]=deal(10).

Your task is to add functionality to antDef with the objective of distributing the 40 ants evenly between each food source, so that 10 ants will be eating at each location. The figures above show what to expect when you do this right.

To do this you only need to focus on one part of the code. You will not need to edit any other part of the code if you do this right.

```
elseif magnitudeNearestFood<=antCurrent.vision %if ant is close to food, coun
    %COMMENT OUT THE THREE LINES BELOW FOR HOMEWORK
    antCurrent.foundFood=foodsourcenumber;
    antCurrent.friendDesire=0;
    nNearestFood=vectorNearestFood/(magnitudeNearestFood);

    % ADD NEW CODE HERE
```

When writing the new code, think about the following three tasks

Task 1: Quantify the number of ants eating at a particular location. This can be done by defining an 'eating' ant as one that is within the vision radius (i.e. less than 0.2 [m] from the food location). The 'eating' Ants should be assigned a label that corresponds to the food location (1,2,3, or 4).

Task 2: If the number of ants 'eating' is less than maxAntsEating, then do something like continue toward the food, and perhaps change foodDesire to 1, and friendDesire to 0. No one cares about friends, once they smell food right?

Task 3: If the number of ants 'eating' is equal to maxAntsEating, then do something else, like move away from the food and head in another direction.

These three tasks can be achieved with the following code structure

```
                numAntsEating=???;
                if numAntsEating < antCurrent.maxAntsEating
                    nNearestFood=???
                    antCurrent.foundFood=???;
                    antCurrent.foodDesire=???;
                    antCurrent.friendDesire=???;
                elseif numAntsEating >= antCurrent.maxAntsEating
                    nNearestFood=???
                    antCurrent.foodloc(foodsourcenumber,:)=???;
                end
```

The last line redefines the antCurrent.foodloc so that the ant no longer goes toward the food that has many ants eating it.

Turn in your code, a video of your ant colony, and the final count.

References

[1] S. Leung, J. Lau, L. Tse, and S. Oppenheimer, "Weight-for-age and weight-for-height references for Hong Kong children from birth to 18 years," en, *Journal of Paediatrics and Child Health*, vol. 32, no. 2, pp. 103–109, Apr. 1996, ISSN: 1034-4810, 1440-1754. DOI: 10.1111/j.1440-1754.1996.tb00904.x.