Brendan Higgins
09/15/16
ME570
HW #1

Question 1.1: The “triangleDraw” function is created to generate a triangle by connecting 3 lines. The 3 lines are created by calling the "points2line" function (which is technically created as part of Question 1.2), and the input variable 'vertices' is a 2x6 matrix, where each column represents the x and y coordinates of the 3 vertices of the triangle.

```
    function [] = triangleDraw(vertices)
% Create Xi vectors that take coordinates from 'vertices'
    X1 = [vertices(1,1) vertices(1,2)];
    X2 = [vertices(1,1) vertices(1,3)];
    X3 = [vertices(1,2) vertices(1,3)];
    Y1 = [vertices(2,1) vertices(2,2)];
    Y2 = [vertices(2,1) vertices(2,3)];
    Y3 = [vertices(2,2) vertices(2,3)];
% Line 1: pt1 --> pt2
    pt1_1 = [X1(1) Y1(1)];
    pt2_1 = [X1(2) Y1(2)];
    line1 = points2line(pt1_1,pt2_1); % call points2line function
% Line 2: pt1 --> pt3
    pt1_2 = [X2(1) Y2(1)];
    pt2_2 = [X2(2) Y2(2)];
    line2 = points2line(pt1_2,pt2_2); % call points2line function
% Line 3: pt2 --> pt3
    pt1_3 = [X3(1) Y3(1)];
    pt2_3 = [X3(2) Y3(2)];
    line3 = points2line(pt1_3,pt2_3); % call points2line function
% Plot triangle
    plot(X1,line1,'--b',X2,line2,'--b',X3,line3,'--b','LineWidth',2)
    end
```

Question 1.2: Three functions are created to satisfy this question: “points2line”, “lineCheckSide”, and “triangleCheckCollisions”.

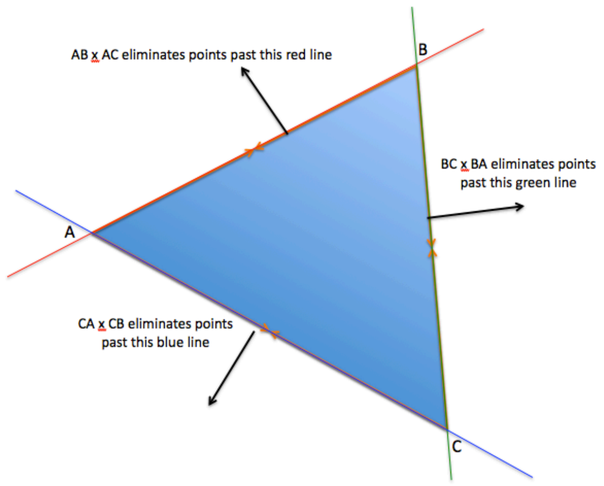
“points2line” is a function that takes the input arguments as the x and y coordinates of 2 points and calculates the slope and intercept of the line formed by connecting the 2 points.

```
    function [line] = points2line(Pt1,Pt2)
    X = [Pt1(1) Pt2(1)];
    Y = [Pt1(2) Pt2(2)];
    m = (Y(2) - Y(1))/(X(2) - X(1)); % calculate slope
    b = Y(2) - m*X(2); % calculate intercept
    line = m.*X + b; % equation of a line
    end
```

“lineCheckSide” is a function that determines whether a point called 'p' is on the correct side of a line. It is performed by creating 3 vectors. Imagine 'a', 'b', and 'c' are 3 corners of a triangle. Then vector 'ab' is a->b, 'ac' is a->c, and 'ap' is a->p. By taking the cross product of 'ab' with 'ap' and of 'ab' with 'ac', one can determine whether point 'p' is on the correct side of the line or not, as long as both cross products are the same sign.

```
    function [isPtInside] = lineCheckSide(p,a,b,c)

% Calculate components of vectors 'ab', 'ac', and 'ap'
    abx = b(1) - a(1);
    aby = b(2) - a(2);
    apx = p(1) - a(1);
    apy = p(2) - a(2);
    acx = c(1) - a(1);
    acy = c(2) - a(2);

% Calculate cross product of vectors 'ab' and 'ap' (ab x ap)
    cp1 = abx*apy - aby*apx;

% Calculate cross product of vectors 'ab' and 'ac' (ab x ac)
    cp2 = abx*acy - aby*acx;

% Assign logical values to define where point is
    if cp1*cp2 > 0
        isPtInside = 1; % p is on correct side of line
    elseif cp1*cp2 < 0
        isPtInside = 0; % p is not on correct side of line
    end
    end
```

Illustration of how this function “lineCheckSide” uses the cross product to eliminate points:


AB x AC eliminates points past this red line
B
BC x BA eliminates points past this green line
A
CA x CB eliminates points past this blue line
C

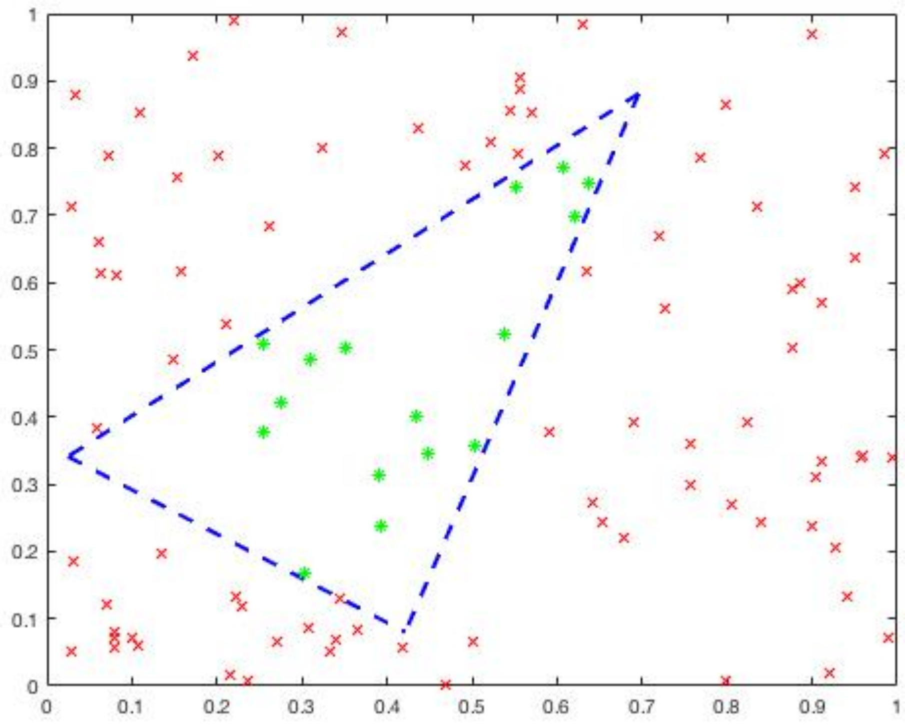
“triangleCheckCollisions” is a function that determines whether the point is inside or outside the triangle by calling the "lineCheckSide" function. The input variable 'points' is a 2x8 matrix, where each column represents the coordinates of the point of interest, triangle pt 'a', triangle pt 'b', and triangle pt 'c', respectively.

```
    function [flagPoints] = triangleCheckCollisions(points)
% Coordinates of point of interest
        p = points(:,1);

% 3 vertices of triangle
        a = points(:,2);
        b = points(:,3);
        c = points(:,4);

% Check if the point is on the correct side of lines using cross
products
% (this calls "lineCheckSide" function)
        if lineCheckSide(p,a,b,c) == 1 && lineCheckSide(p,b,c,a) == 1
&& lineCheckSide(p,c,a,b) == 1
            flagPoints = 1; % point 'p' is inside triangle
        else
            flagPoints = 0; % point 'p' is outside triangle
        end
    end
```

Question 1.3: For this question, the “triangleTest” function is created to generate a random triangle by calling the "triangleDraw" function and then generate 100 random points and determine whether they are inside or outside the triangle by using the "triangleCheckCollisions" function.

```
function [] = triangleTest()

% Clean Up
close all % close all previously generated figures
clc % clear command window

% Call triangleDraw function and provide input
vertices = rand(2,3); % generate random 3 points to create triangle
triangleDraw(vertices) % draw triangle based on random 3 points

% plot 100 random points over triangle figure
hold on
for i = 1:100
point = rand(1,2); % generates a random point & for loop iterates 100
times

% Call triangleCheckCollisions and provide input
points = [point' vertices];
flagPoints = triangleCheckCollisions(points);
if flagPoints == 1
    plot(point(1),point(2),'*g') % plot point as green * if inside
triangle
elseif flagPoints == 0
    plot(point(1),point(2),'xr') % plot point as red x if outside
triangle
end
end

end
```

Here is an example plot from the “triangleCheckCollisions” function:



Question 2.1: For this question a function “priorityPrepare” is created that returns a vector pq of structs with fields key and cost.

```
function [pq] = priorityPrepare()
% This function returns a vector pq of structs with fields key and cost
pq = repmat(struct('key',[],'cost',[]),0,1);

end
```

Question 2.2: For this question, a function called “priorityPush” is created that adds a structure with arguments key and cost to their corresponding fields in vector pq.

```
function [pq] = priorityPush(pq,key,cost)
% This function adds a structure with arguments key and cost to their
corresponding fields in vector pq.
for i = 1:length(key)
    pq(i).key = key(i);
    pq(i).cost = cost(i);
end
end
```

Question 2.3: For this question, a function is created that accepts a vector pq that has structs with fields key and cost. It then finds the minimum cost and outputs the minimum cost and its corresponding "key". Finally, it removes that "key" and cost from the vector pq and outputs pq.

```
function [PQ,key,cost] = priorityMinPop(pq)
```

```
% Assign cost to a vector to calculate the minimum
for i = 1:length(pq)
    v(i) = pq(i).cost;
end

% Loop through the cost vector just created and for each value check if
it
% is the minimum. Also match the minimum cost with its associated key.
for k = 1:length(pq)
    if v(k) == min(v)
        key = pq(k).key;
        cost = pq(k).cost;
        index = k;
    end
end

% Remove minimum cost and associated key element from fields in pq.

% If 1st element is min, assign rest of elements in order to new pq
vector
if index == 1
    for j = 2:length(pq)
    PQ(j-1).key = pq(j).key;
    PQ(j-1).cost = pq(j).cost;
    end

% If last element is min, assign all prior elements to new pq vector
elseif index == length(pq)
    for j = 1:length(pq)-1
    PQ(j).key = pq(j).key;
    PQ(j).cost = pq(j).cost;
    end

% If min cost is somewhere in between 1st and last elements, skip the
% element where it is the min
else
    for j = 1:index-1
        PQ(j).key = pq(j).key;
        PQ(j).cost = pq(j).cost;
    end
    for h = index+1:length(pq)
        PQ(h-1).key = pq(h).key;
        PQ(h-1).cost = pq(h).cost;
    end
end

end
```

Question 2.4: For this question a function is created that takes a vector pq (that has structs with fields key and cost) and checks each cost to determine if it is above a particular threshold. The function removes all costs and associated keys above the input threshold and outputs the new pq vector with the values above removed.

```
function [PQ] = priorityRemoveAbove(pq,threshold)
c = 0;
for i = 1:length(pq) % go through each cost element one-by-one
```

```
    if pq(i).cost <= threshold % check if it's less than the threshold
% if the cost of the ith element is below or equal to the threshold
then
% add it to the new pq vector to be outputted. also add associated key
    PQ(c+1).cost = pq(i).cost;
    PQ(c+1).key = pq(i).key;

% This is a counter used to increment the new pq vector
    c = c + 1;
    end
end
```

Question 2.5: This function takes a randomly generated vector of length 20 and pushes it through a priority, then sorts it and extracts the sorted vector in order to compare with the built-in Matlab "sort" function.

```
function [check] = priorityTest(v)

% First create pq by calling "priorityPrepare"
[pq] = priorityPrepare();

% Generate a vector of 20 random integers between 1 and 10
a = 1;
b = 10;
v = round(a + (b-a)*rand(20,1));

% Create key vector
key =
['a';'b';'c';'d';'e';'f';'g';'h';'i';'j';'k';'l';'m';'n';'o';'p';'q';'r
';'s';'t'];

% Push randomly generated 20x1 vector into "priorityPush" to assign
random
% values to "cost" field
[pq] = priorityPush(pq,key,v);

% Call "priorityMinPop" to determine the min of vector "v". Need to
iterate
% through in order to get the min of each new population and reorder
the
% vector from min to max
for j = 1:19
[pq,key,cost(j)] = priorityMinPop(pq);
% The cost variable is built from min to max by looping through the
% priorityMinPop function multiple times. After doing the 1st 19
values,
% add the max of vector "v" to the end and this vector matches sort(v),
% which is Matlab's built-in function that sorts a vector from min to
max
end

% Compare the 2: built-in matlab "sort" function vs priorityMinPop
function
Matlab_func = sort(v);
priority_func = [cost(1,:)';max(v)];
```

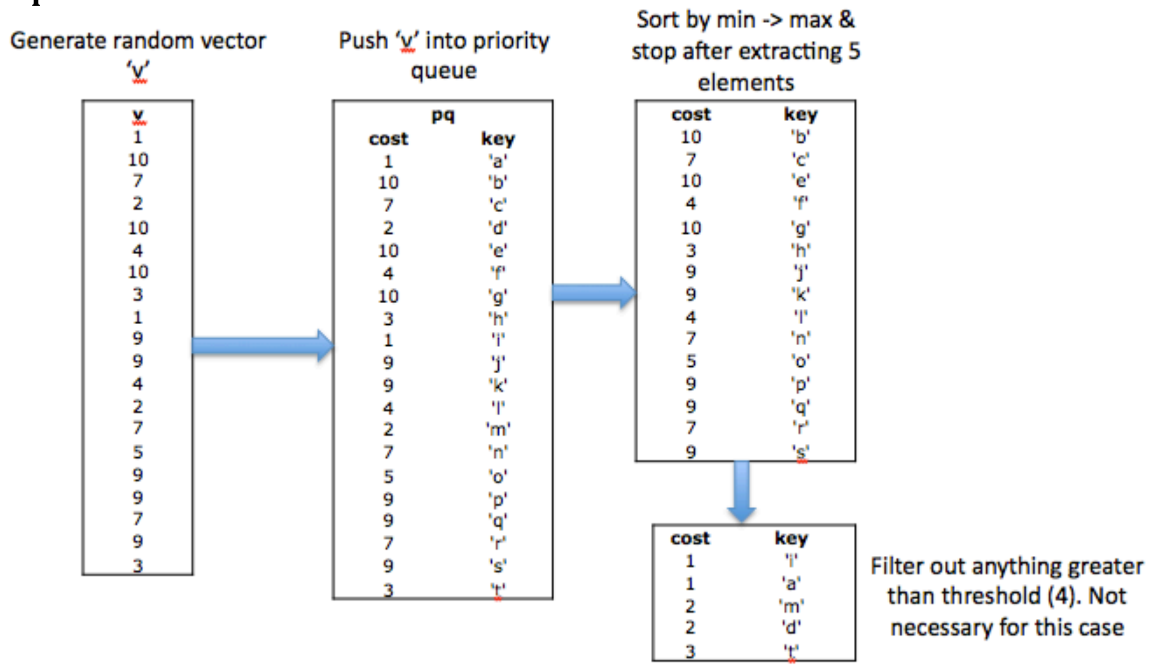
```
% Check to see that the vectors match -> all columns = 0
for i = 1:20
check(i) = Matlab_func(i) - priority_func(i);
end
end
```

Question 2.6: This function takes a randomly generated vector of length 20 and pushes it through a priority queue, then sorts it and extracts the sorted vector. After extracting the first 5 elements of the sorted vector, it calls "priorityRemoveAbove" in order to filter out any elements in the remaining vector that are above a specified threshold.

```
function [pq,PQ,v] = priorityTestRemove()

% Clean up
clear all
clc

% First create pq by calling "priorityPrepare"
[pq] = priorityPrepare();

% Generate a vector of 20 random integers between 1 and 10
a = 1;
b = 10;
v = round(a + (b-a)*rand(20,1));

% Create key vector
key =
['a';'b';'c';'d';'e';'f';'g';'h';'i';'j';'k';'l';'m';'n';'o';'p';'q';'r
';'s';'t'];

% Push randomly generated 20x1 vector into "priorityPush" to assign
random
% values to "cost" field
[pq] = priorityPush(pq,key,v);

% Call "priorityMinPop" to determine the min of vector "v". Need to
iterate
% through in order to get the min of each new population and reorder
the
% vector from min to max

for j = 1:5 % only want the 1st 5 elements
[pq,key(j),cost(j)] = priorityMinPop(pq);
% The cost variable is built from min to max by looping through the
% priorityMinPop function multiple times.
end

% Now that the vector has been sorted and the 1st 5 elements are
extracted,
% call "priorityRemoveAbove" function to filter out anything above 4:
threshold = 4;
for i = 1:5
p(i).key = key(i);
p(i).cost = cost(i);
```

```
end
[PQ] = priorityRemoveAbove(p,threshold);

end
```

Flow chart to illustrate a particular example of the queue working as expected:


Generate random vector 'v'
Push 'v' into priority queue
Sort by min -> max & stop after extracting 5 elements
Filter out anything greater than threshold (4). Not necessary for this case

v
1
10
7
2
10
4
10
3
1
9
9
4
2
7
5
9
9
7
9
3

pq	
cost	key
1	'a'
10	'b'
7	'c'
2	'd'
10	'e'
4	'f'
10	'g'
3	'h'
1	'i'
9	'j'
9	'k'
4	'l'
2	'm'
7	'n'
5	'o'
9	'p'
9	'q'
7	'r'
9	's'
3	't'

cost	key
10	'b'
7	'c'
10	'e'
4	'f'
10	'g'
3	'h'
9	'j'
9	'k'
4	'l'
7	'n'
5	'o'
9	'p'
9	'q'
7	'r'
9	's'

cost	key
1	'i'
1	'a'
2	'm'
2	'd'
3	't'

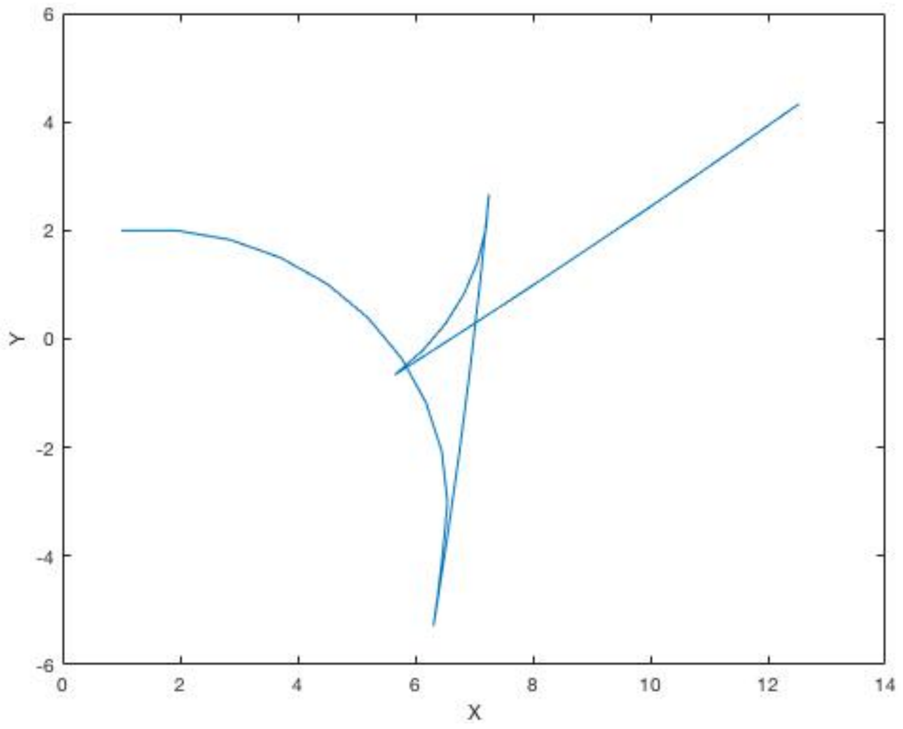
Question 3.1: For this question, a unicycle model, also known as the kinematic cart, is created. A function "unicycleModel" is created, which takes the equation used to describe the model and calculates the vector at the next position based on the current position. The position is defined by the x and y coordinates as well as θ, which is the angle with respect to the x axis. The linear forward speed and the angular velocity are also used to help determine the next position.

```
function [xVectorNext] = unicycleModel(xVectorCurrent,v,w)
% For each vector component (x,y,theta), calculate the next position
based on the current position, forward linear speed, and angular
velocity

% The following equation is provided in the problem statement:
    xVectorNext = [xVectorCurrent(1) + v*cos(xVectorCurrent(3)); ...
        xVectorCurrent(2) + v*sin(xVectorCurrent(3)); ...
        xVectorCurrent(3) + w];
end
```

Question 3.2: For this question, "unicycleTest" is created to generate random linear speeds and angular velocities for 5 randomly generated time intervals. The function "unicycleModel" is called and the first 2 components (x,y) of vector **X** are plotted over the time intervals.

```
function [x] = unicycleTest()

% Clean up
close all
clc

x = [1;2;0]; % VERY initial conditions
c = 1; % initiate counter

a = 1; % min time interval
b = 10; % max time interval

t=1; % for indexing purposes
for i = 1:5
% Create 5 random time intervals with integer lengths between 1 and 10
t(i+1) = round(a + (b-a)*rand);

% Define fwd speed & angular velocity inputs for unicycle model
function
v = randn;
w = 0.1*randn;

% Call unicycle model function
for k = 1:t(i+1)
    x(:,k+sum(t(1:i))) = unicycleModel(x(:,k-1+sum(t(1:i))),v,w);
end
c = c+1; % increment counter to match current sequence with next
sequence
end

% Create Plot
plot(x(1,:),x(2,:))
xlabel('X')
ylabel('Y')

end
```

Here is an example of the plot:


Y
X
6
4
2
0
-2
-4
-6
0
2
4
6
8
10
12
14