ME145 Robotic Planning and Kinematics: Lab 5

Robotic Planning and Kinematics

Eric Perez

Spring 2024

May 20, 2024

E4.3 Programming: Sampling algorithms

- formulas for the n = k^2 sample points in the uniform Sukharev center grid:
    - $$\text{dispersion}_{\text{square}}(P_{\text{center grid}}(n, d)) = \frac{1}{2\sqrt[d]{n}}.$$
    - The sphere-dispersion of the center grid is 1/2 the length of the longest diagonal of a sub-cube. In d dimensions, the unit cube has a diagonal of length √ d. With d = 2, the unit square has a diagonal with length √ 2. The sphere-dispersion of the center grid is √ d/(2√d n)
- formulas for the n = k^2 sample points in the uniform corner grid:
    - The corner grid is defined as follows: (1) divide the [0, 1] interval into (k − 1) subintervals of equal length and therefore compute (k−1)d sub-cubes of X, (2) place one grid point at each vertex of each sub-cube.

computeGridSukharev

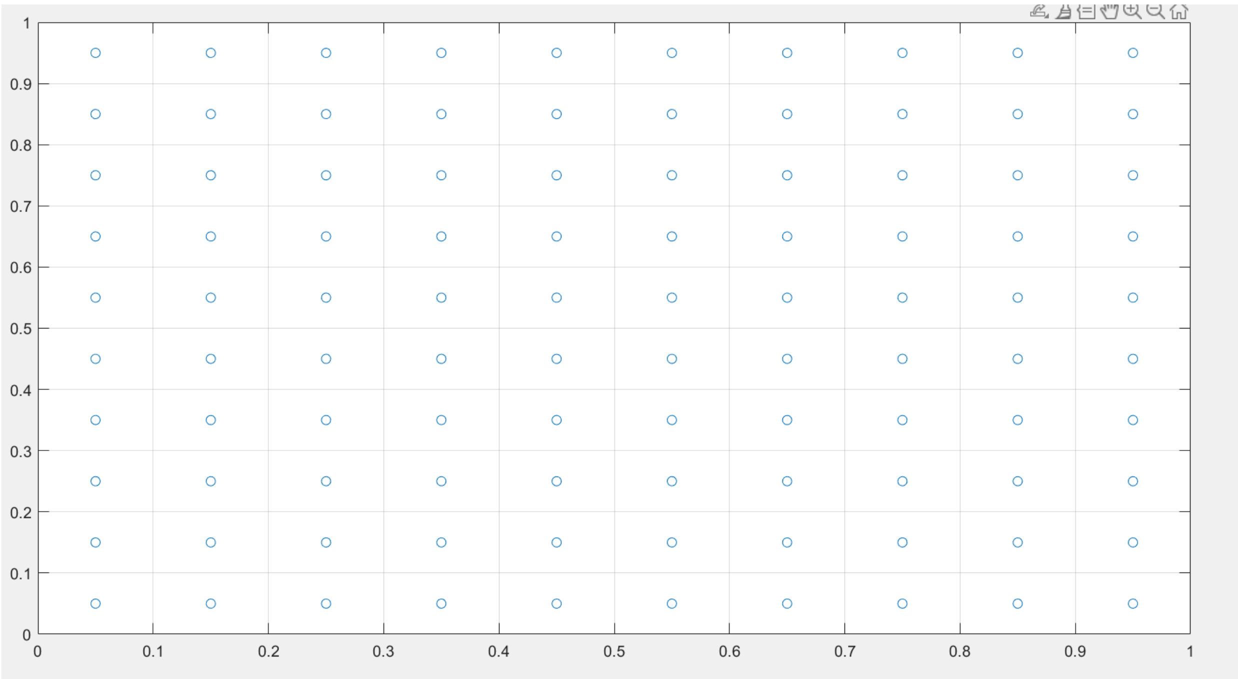
```
% Eric Perez
% ME 145 Lab 5


function [Grid] = computeGridSukharev(n)
    k = sqrt(n);
    if k ~= floor(k)
        error('n is not a perfect square.');
    end
    Grid = zeros(n, 2);
    int = 1;
    for i = 0:k-1
        for j = 0:k-1
            Grid(int, :) = [(i + 0.5) / k, (j + 0.5) / k];
            int = int + 1;
        end
    end


plot(Grid(:,1), Grid(:,2),'o')
grid on;
end

```

The function above accomplishes the center grid sampling by increasing the x-value by 0.5 and dividing that same x-value by the number of columns to get equal spacing. Multiple y-values are obtained for every new x-value. The “Grid” output was capitalized since the “gid on” command was used and “grid” could not be variable. The function also checks if the input creates a perfect square as the grid requires a perfect square to sample.

Testing function (with n = 100):

Input: [Grid] = computeGridSukharev(100)

Output:


0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

<u>*computeGridRandom*</u>

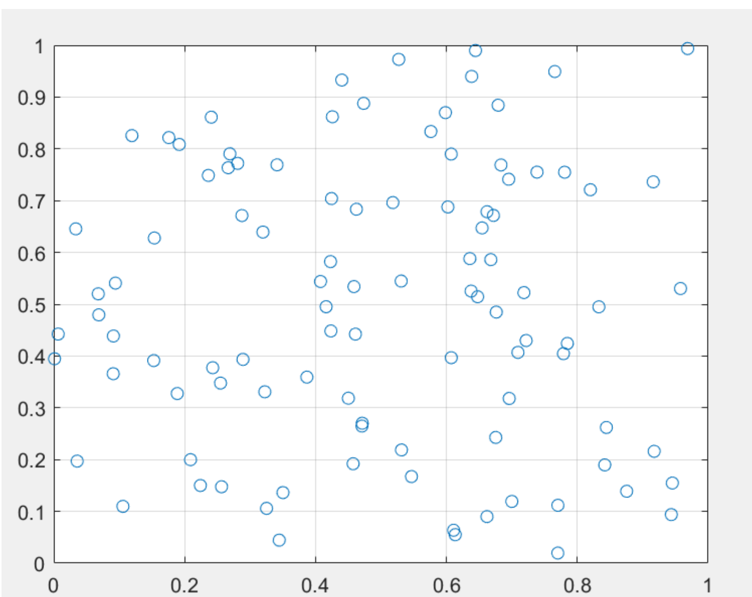
```
function Grid = computeGridRandom(n)
    Grid = rand(n, 2);
    plot(Grid(:,1), Grid(:,2),'o')
    grid on
end
```

This function was created using the built-in Matlab function rand() to generate a random, two-column sampling list. The function also does not need to check if the number of samples is a perfect square since they are randomly inputted.

Testing function (with n = 100):

Input: [Grid] = computeGridRandom(100)

Output:



computeGridHalton

```
function Grid = computeGridHalton(n, b1, b2)

     base = [b1,b2];


 for  k = 1:2
      seq= zeros(n, 1);
      TF = isprime(base(k));
  if TF == 1
    for i = 1:n
      f = 1 / base(k);
        r = 0;
        idx = i;
        while idx > 0
            r = r + f * mod(idx, base(k));
            idx = floor(idx / base(k));
            f = f / base(k);
        end
        seq(i) = r;
    end
    Grid(:,k) = seq;
  else
      error('Input b1 and b2 need to be prime values')
  end
 end
 plot(Grid(:,1), Grid(:,2),'o')
    grid on
end

```

The computeGridHalton was created by implementing the Halton sequence. Pseudo code from the textbook helped with the development of the function. Below is the Pseudo code used.

Halton sequence algorithm

Input: length of the sequence $N \in \mathbb{N}$ and prime number $p \in \mathbb{N}$
Output: an array S with the first N samples of the Halton sequence generated by p

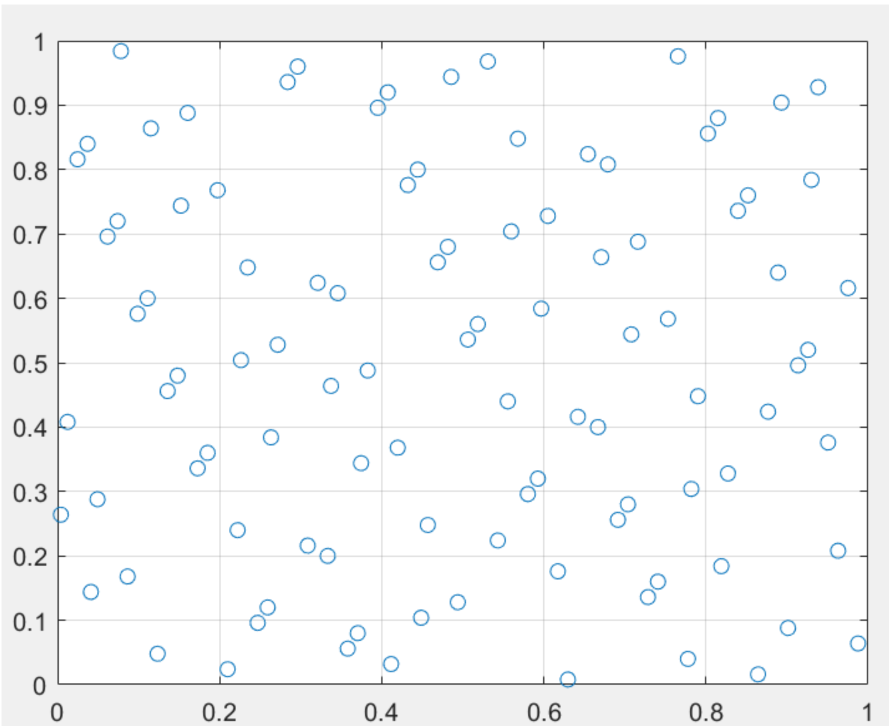
initialize: S to be an array of N zeros (i.e., $S(i) := 0$ for each i from 1 to N)
for each i from 1 to N :
    initialize: $i_{\text{tmp}} := i$, and $f := 1/p$
    while $i_{\text{tmp}} > 0$:
        compute the quotient q and the remainder r of the division i_{tmp}/p
        $S(i) := S(i) + f \cdot r$
        $i_{\text{tmp}} := q$
        $f := f/p$
return S

The two base numbers inputted must be prime numbers so the function checks for this and sends an error message if this requirement is not met.

Testing function (with n = 100, b1 =3, b2 =5):

Input: [Grid] = computeGridHalton(100,3,5)

Output:



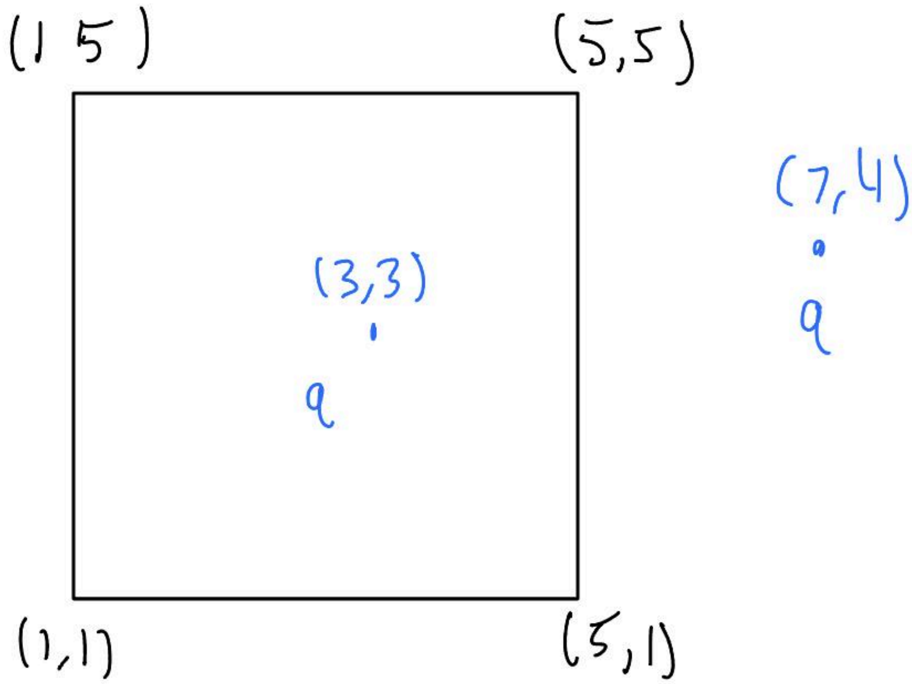
E4.4 Programming: Collision detection primitives

isPointInConvexPolygon

```
function inside = isPointInConvexPolygon(q, P)
    n = size(P, 1);
    inside = true; % Assume the point is inside the polygon

    for i = 1:n
        p1 = P(i, :);
        p2 = P(mod(i, n) + 1, :); % grab first point once n is maximum

        % Compute the interior normal vector
        normal = [- (p2(2) - p1(2)), (p2(1) - p1(1))];

        % Compute the vector from p1 to q
        pq = [q(1) - p1(1), q(2) - p1(2)];

        % dot product
        dot_product = normal(1) * pq(1) + normal(2) * pq(2);

        % If any dot product is negative, the point is outside the polygon
        if dot_product < 0
            inside = false;
            return;
        end
    end
end
```

The isPointInConvex polygon function was created using the textbook following section 4.4.2 "Basic primitive #2: is a point in a convex polygon?". The method involves computing the interior normal vector and comparing the vector to the vector from p1 to q by using the dot product. If the dot product is negative, the point is not in the convex polygon.

Testing function with the below drawing:



Ex 1) Input: [inside] = isPointInConvexPolygon([3, 3], [1 1; 5 1; 5 5; 1 5])

Output:

```
>> [inside] = isPointInConvexPolygon([3, 3], [1 1; 5 1; 5 5;

inside =

  logical

   1
```

Ex 2) Input: [inside] = isPointInConvexPolygon([7, 4], [1 1; 5 1; 5 5; 1 5])

Output:

```
>> [inside] = isPointInConvexPolygon([7, 4], [1 1; 5 1; 5 5; 1 5])

inside =

  logical

   0
```

doTwoSegmentsIntersect

```
function[intersect,intersectPoint] = doTwoSegmentsInter
intersect = false;
intersectPoint = false;
x1 = P1(1);
y1 = P1(2);

x2 = P2(1);
y2 = P2(2);

x3 = P3(1);
y3= P3(2);

x4 = P4(1);
y4 = P4(2);
P1x = [x1,x2];
P2x = [x3,x4];
P1y = [y1,y2];
P2y = [y3,y4];


plot(P1x,P1y);
hold on
plot(P2x,P2y,'-r');

numA =(x4 - x3)*(y1 - y3) - (y4 - y3)*(x1 - x3);
denA = (y4 - y3)*(x2 - x1) - (x4 - x3)*(y2 - y1);

numB = (x2 - x1)*(y3-y1) - (y2-y1)*(x3 - x1);
denB = (y2 - y1)*(x4 - x3) - (y4 - y3)*(x2-x1);

sa = numA/denA;
sb = numB/denB;
intersectPoint1 = P1 + sa*(P2 - P1);
intersectPoint2 = P3 + sb*(P4 - P3);
```

```

if (denA ~= 0)
     if (sa >= 0) && (sa <= 1) && (sb >= 0) && (sb <= 1)
     intersect = true;
     intersectPoint = intersectPoint1;
     end
end


end
```

This function was created based of the sa equation derived in the textbook and also the sb equation derived from the sa. The function returns if the segments intersect and a point of intersection if they do. The value of the numerator and denominator of the sa equation determine if the segments intersect and what point the intersection occurs. The sa equation and the meaning of the the value in the numerator and denominator are below.

$$s_a = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} =: \frac{\mathtt{num}}{\mathtt{den}}.$$

(i) if $\mathtt{num} = \mathtt{den} = 0$, then the two lines are coincident,

(ii) if $\mathtt{num} \neq 0$ and $\mathtt{den} = 0$, then the two lines are parallel and distinct, and

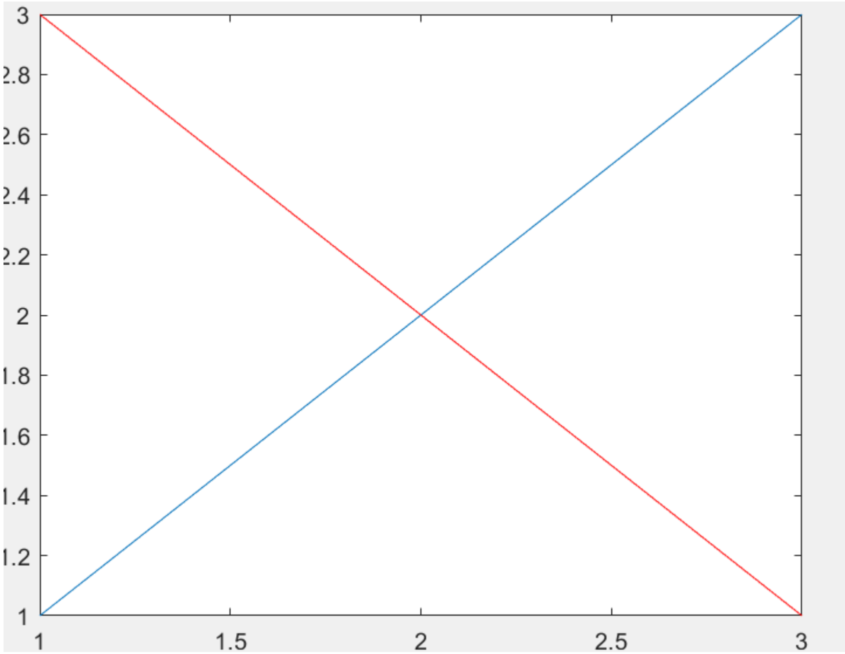
(iii) if $\mathtt{den} \neq 0$, then the two lines are not parallel and therefore intersect at a single point.

Testing function:

Input: [intersect,Point] = doTwoSegmentsIntersect([1 1],[3 3],[3 1],[1 3])

Output:

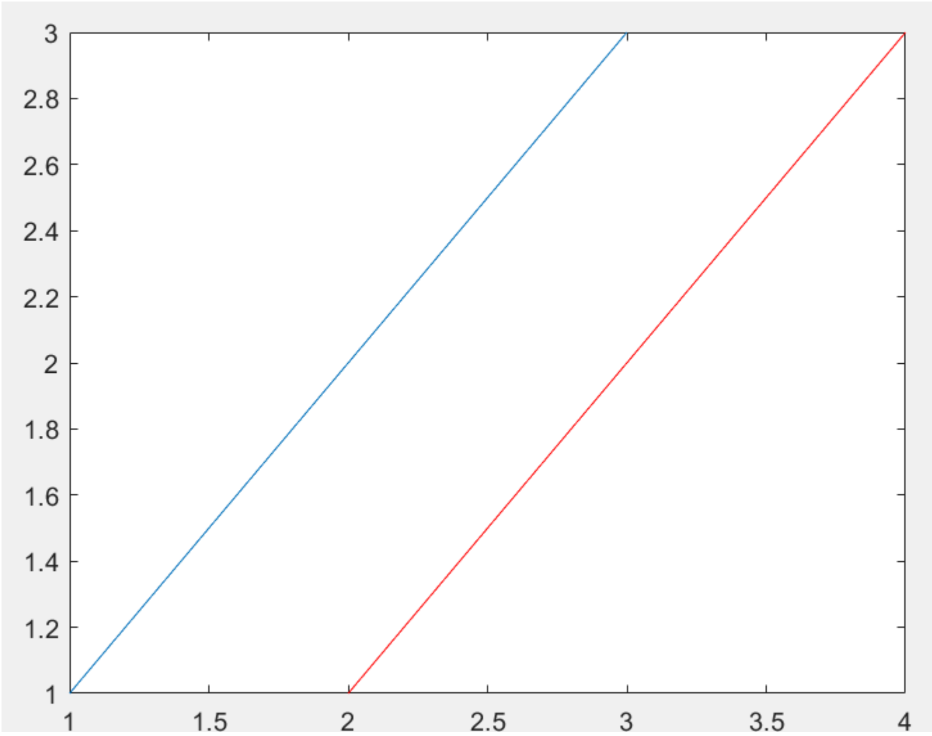
```
intersect =

  logical

   1


Point =

     2     2
```



Input: [intersect,Point] = doTwoSegmentsIntersect([1 1],[3 3],[2 1],[4 3])

Output:

```
  logical

   0


Point =

  logical

   0
```


3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
1
1.5
2
2.5
3
3.5
4

doTwoConvexPolygonsIntersect

```
function [TF] = doTwoConvexPolygonsIntersect(P1,P2)
[n1,~] = size(P1);
[n2,~] = size(P2);
TF = false;

polysize1 = size(P1);
if polysize1(2)~=2
    error ('P1 input size is incorrect')
end
polysize2 = size(P2);
if polysize2(2)~=2
    error ('P2 input size is incorrect')
end

for i = 1:n1
    v1 = P1(i,:);
    v1Next = P1(mod(i,n1)+1,:);
    for j = 1:n2
        v2 = P2(j,:);
    v2Next = P2(mod(j,n2)+1,:);
        if (v1 == v2)
        TF = true;
        end

        [intersect,~] = doTwoSegmentsIntersect(v1, v1Next,v2,v2Next);

        if intersect == true
            TF = true;
        end

```

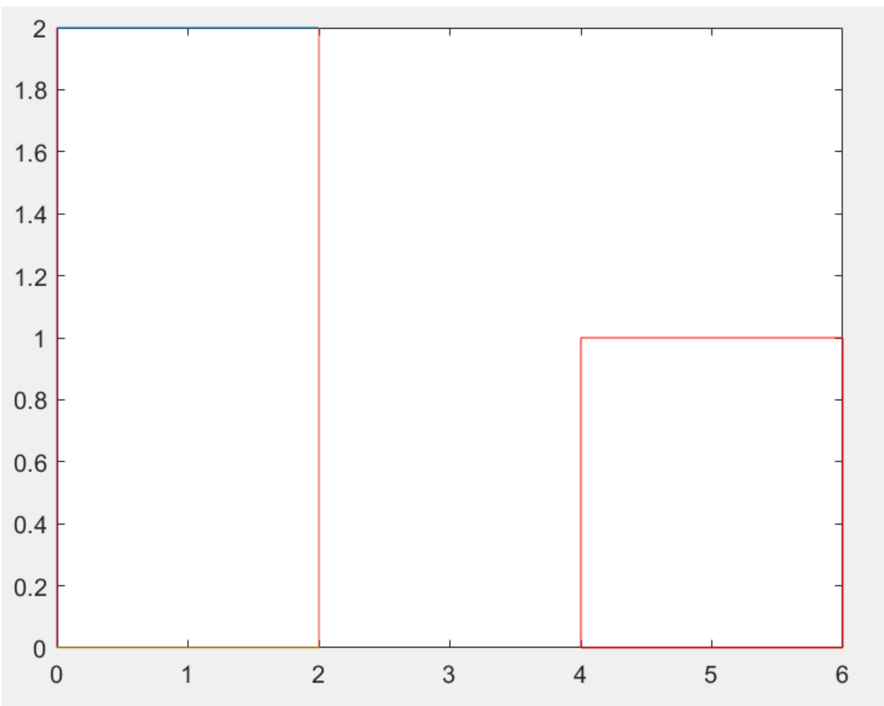
This function takes in two convex polygon inputs and determines if they intersect. This was done by comparing each segment in the polygon with each other using the previous function "doTwoSegmentsIntersect." The function also checks if each polygon is inputted in the correct format.

Testing function:

Input: TF = doTwoConvexPolygonsIntersect([0 0; 0 2; 2 2 ; 2 0],[4 0; 4 1; 6 1; 6 0])

Output:

```
TF =

  logical

   0
```



Input: TF = doTwoConvexPolygonsIntersect([0 0; 0 2; 2 2 ; 2 0],[1 0; 1 1; 3 1; 3 0])

Output:

```
TF =

  logical

   1
```