ME145 Robotic Planning and Kinematic: Lab 5

Robotic Planning and kinematics

Elijah Perez

Winter 2025, March 1

E4.3 written solutions:

- formula for $n=k^2$ sampling points in the sukharev center grid is,

  $$\text{dispersion}_{\text{square}}(P_{cg}(n,d)) = \frac{1}{2\sqrt[d]{n}}$$

  for the sphear center grid it is ½ the Length of the Longest diagnal of the sub cube. In dimens d, the unite cube will have a diag Length of $\sqrt{d}$. for ex, with $d=2$ The unit squar diag will be $\sqrt{2}$. The sphears dispersment of the center grid is $\sqrt{d/2\sqrt{d}\,n}$.

- formulas for the uniform grid where $n=k^2$. The corner grid is defined as 1), divid the [0,1] interval into (k-1) sub intervals of equal Lengths and there for comput $(k-1)^d$ sub cubs of X, 2) place one grid point at each Vertex of each sub-cube.

Part 1: Compute Grid Sukharev

Algo:

```
function Grid = computeGridSukharev(n)

k = sqrt(n);
% check for perfect squar

if k ~= floor(k)
error("n is not a perfect square!")
end

m=0.5;
p=0;

for i = 1:k
n=0;
    for j =1:k
    p = p+1;
    Gridx(p) = n+0.5;
    Gridy(p) = m;
    n = n+1;
    end
m = m+1;
end

Grid = [Gridx./k; Gridy./k];

figure
plot(Grid(1,:),Grid(2,:),".",LineWidth=2)
xlim([0 1])
ylim([0 1])
grid on
end
```

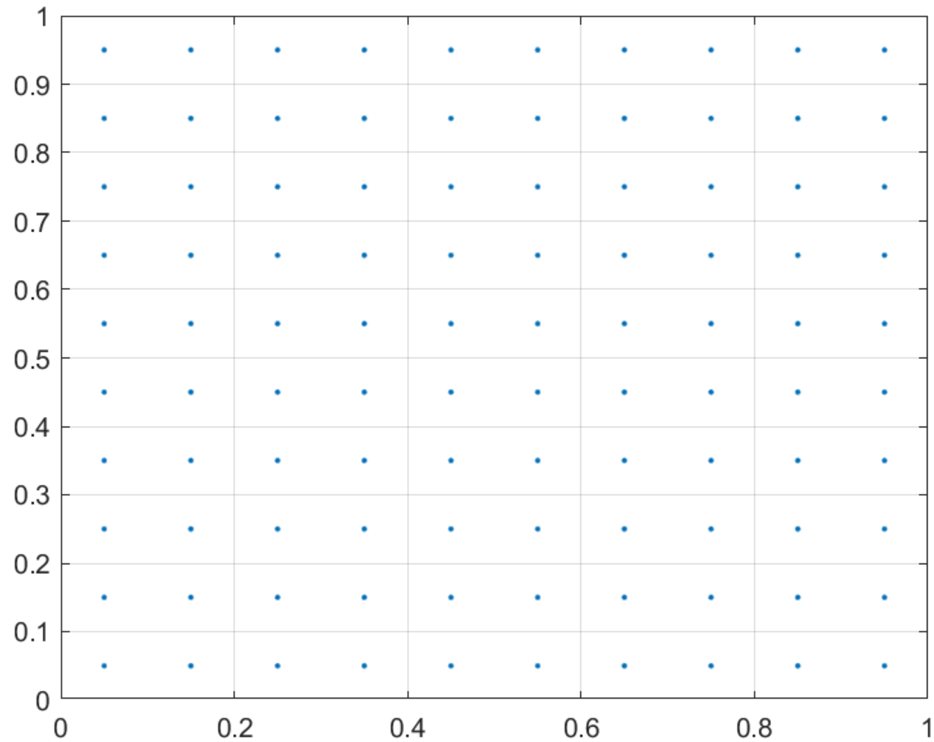
The function performs center grid sampling by incrementing the x-value by 0.5 and normalizing it using the number of columns to ensure equal spacing. For each new x-value, multiple corresponding y-values are generated, maintaining a structured grid. The output "Grid" is capitalized because the grid on command prevents "grid" from being used as a variable name. Additionally, the function verifies that the input forms a perfect square, as the sampling process requires a square grid.

Examples:

Input

```
% Varify test
n = 100;
Grid = computeGridSukharev(n); %Functions plots for me

```

Output:



Grid 2x100 double

Error test:

Input:

```
% error test
n = 56;
Grid = computeGridSukharev(n);

```

Output:

```
Error using computeGridSukharev
(line 7)
n is not a perfect square!
```

Part 2: Compute random Grid
algo :

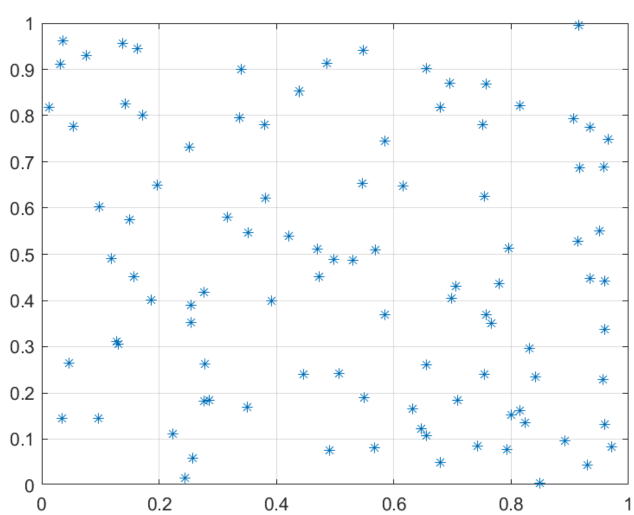
```
function Grid_random = computeGridRandom(n)

if length(n)~=1
error("n must be a single integer, n = # of points")
end

Grid_random = rand(n,2);
figure
plot(Grid_random(:,1),Grid_random(:,2),"*")
xlim([0 1])
ylim([0 1])
grid on
end
```

This function uses Matlab's built-in rand function to generate a n by 2 matrix of random numbers [0 1]. The first row of the generated random number is the X-axis and the second row is used for the Y-axis.

Verify
Input:

```
%% Part 2 ComputeGridRandom

%Varify test
n = 100;
Grid = computeGridRandom(n);

```

Output:



Error Test

Input:

```
%error test
n = 1:100;
Grid = computeGridRandom(n);
```

Output:

N has to always be a single number.

```
Error using computeGridRandom (line 4)
n must be a single integer, n = # of points
```

Part 3: CompueGridHalton

Algo

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

Grid = zeros(n,2);
B = [b1 b2];
for i = 1:2
   if isprime(B(i))
        for j = 1:n
            num = 1/B(i);
            m = 0;
            delta = j;
            while delta>0
                m = m+num*mod(delta,B(i));
                delta = floor(delta/B(i));
                num = num/B(i);
            end
        Grid(j,i) = m;
        end
    else
        error("Please make sure b1 and b2 are both prime numbers")
    end
end
figure
plot(Grid(:,1),Grid(:,2),"*")
grid on
xlim([0 1])
ylim([0 1])

end
```

This algorithm was heavily based on the pseudo-code in the notes. 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

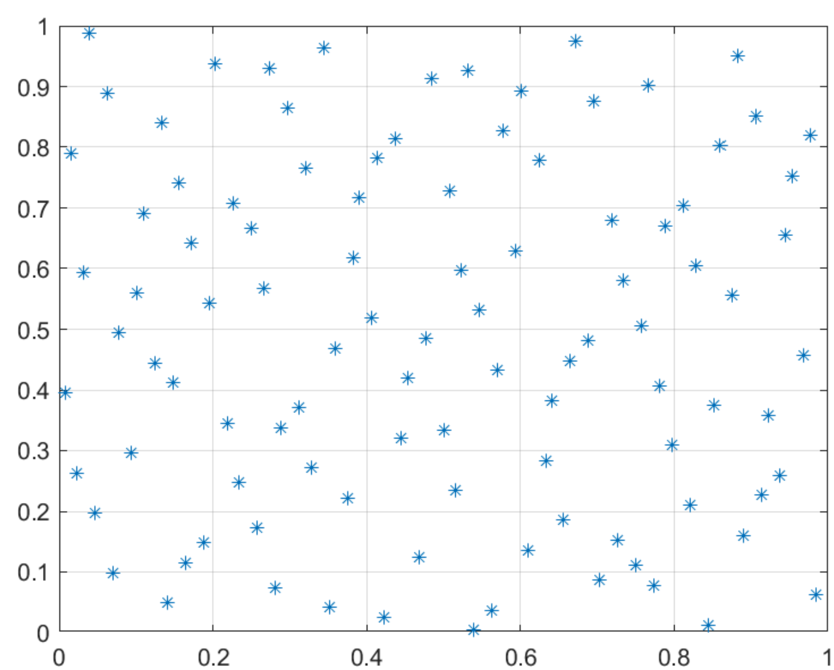
The n input can be anything, but the two base numbers must be prime numbers to generate non-patterned results.

Verify

Input

```
%% Part 3 CompueGridHalton

% varify test
n = 100;
b1 = 2;
b2 = 3;
Grid = computeGridHalton(n,b1,b2);

```

Output:



This results in a better distribution of random saplings in the workspace over the complete random sampling method used for part two.

Error test:

Input

```
% error test
n = 100;
b1 = 4;
b2 = 8;
Grid = computeGridHalton(n,b1,b2);
```

Output

```
Error using computeGridHalton (line 19)
Please make sure b1 and b2 are both prime numbers
```

Part 4: Is Point In Convex Polygon

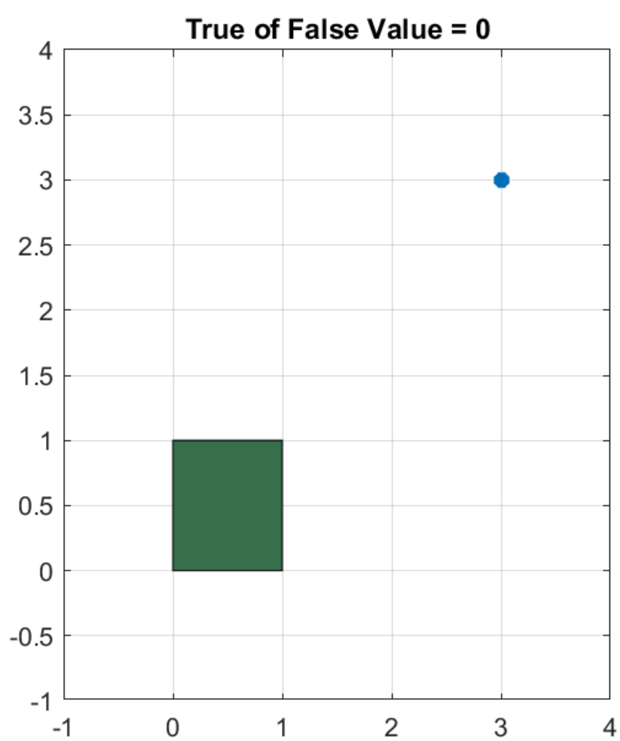
Algo

```
function TF = isPointinConvexPolygon(q,P)

if length(P(1,1,:)) ~=1
error("One Polygon at a time please!")
end

if length(q) ~= 2
    error("Point q must be two dimentional (x,y)")
end

TF = 1;
Info = [1:length(P) 1];
for i = 1:length(P)

    info1 = Info(i);
    info2 = Info(i+1);
    InnerVector = [-(P(info2,2)-P(info1,2)) P(info2,1)-P(info1,1)];
    % PCurrent = [P(info1,1) P(info1,2)];
    % Pnext    = [P(info2,1) P(info2,2)];
    V = [q(1)-InnerVector(1) q(2)-InnerVector(2)];
    DP = InnerVector(1)*V(1) + InnerVector(2)*V(2);

    if DP>0
        TF = 0;
        break
    end
end

end
```

The (isPointInConvexPolygon) function was implemented based on Section 4.4.2 of the textbook, titled *"Basic Primitive #2: Is a Point in a Convex Polygon?"* This method determines whether a point lies inside a convex polygon by computing the interior normal vector of each edge and comparing it to the vector from a given point p1 to the query point q using the dot product. If the dot product is negative for any edge, the point q is outside the polygon.

Varfy:
Input

```
% Varify 1 not in Polygon
P = [1 1; 0 1; 0 0; 1 0];
q = [3 3];
TF = isPointinConvexPolygon(q,P);

tiledlayout(1,2)
nexttile
fill(P(:,1),P(:,2),[rand(1) rand(1) rand(1)])
hold on
plot(q(1),q(2),'*',LineWidth=3)
xlim([-1 4])
ylim([-1 4])
grid on
title("True of False Value = " + TF)
```

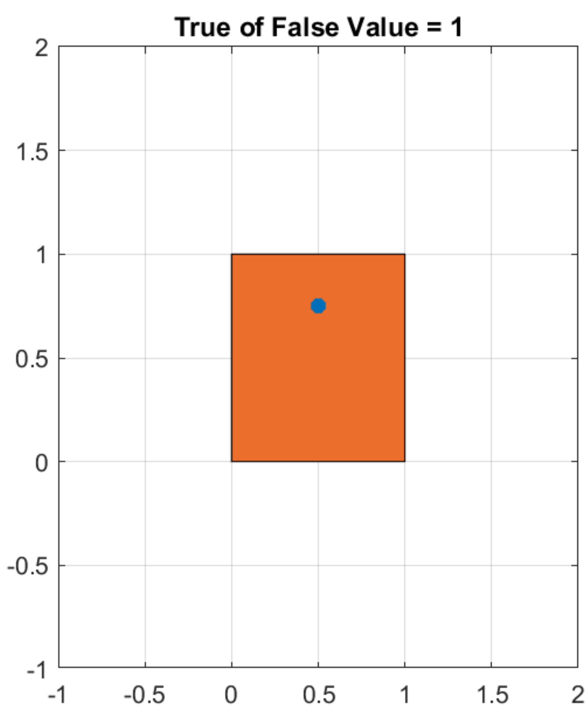


The point is not in the polygon so the output is a zero for false.

Input

```
% Varify 2 in Polygon
P = [1 1; 0 1; 0 0; 1 0];
q = [0.5 0.75];
TF = isPointinConvexPolygon(q,P);

nexttile
fill(P(:,1),P(:,2),[rand(1) rand(1) rand(1)])
hold on
plot(q(1),q(2),'*',LineWidth=3)
xlim([-1 2])
ylim([-1 2])
grid on
title("True of False Value = " + TF)
clc
clear
```

Output



The point lies in the polygon and now the output is a 1 for true.

Error check:

Input

```
% erorr = too many polygons
P1 = [1 1; 0 1; 0 0; 1 0];
P2 = [2 2; 0 1; 2 1; 1 0];
P(:,:,1) = P1;
P(:,:,2) = P2;
q = [0.5 0.75];
TF = isPointinConvexPolygon(q,P);
```

Output

```
Error using isPointinConvexPolygon (line 4)
One Polygon at a time please!
```

Input

```
% error = not 2d point
P = [1 1; 0 1; 0 0; 1 0];
q = [3 3 3];
TF = isPointinConvexPolygon(q,P);
```

Output

```
Error using isPointinConvexPolygon (line 8)
Point q must be two dimentional (x,y)
```

Part 5: Do two segments intersect

Algo

```
function TF = doTwoSegmentsIntersect(p1,p2,p3,p4)

if length(p1)~=2
    error("Must be two dimension points")
end

TF = 0;

Segment1 = [p1;p2];
Segment2 = [p3;p4];

x1 = p1(1);
x2 = p2(1);
x3 = p3(1);
x4 = p4(1);

y1 = p1(2);
y2 = p2(2);
y3 = p3(2);
y4 = p4(2);

num1 = (x4-x3)*(y1-y3)-(y4-y3)*(x1-x3);
den1 = (y4-y3)*(x2-x1)-(x4-x3)*(y2-y1);

num2 = (x2-x1)*(y3-y1)-(y2-y1)*(x3-x1);
den2 = (y2-y1)*(x4-x3)-(y4-y3)*(x2-x1);


if den1 ~= 0
   S1 = num1/den1;
    if den2~=0
       S2 = num2/den2;
    end


    if (S1>=0) && (S1<=1) && (S2>=0) && (S2<=1)
        TF = 1;
    end

else

end
```

This function was developed using the S1 equation derived from the textbook, along with the S2 equation, which is based on S1. The function determines whether two segments intersect and, if they do, returns the point of intersection. The numerator and denominator of the S1 equation play a key role in identifying whether an intersection occurs and determining its exact location. Below, the S1 equation is provided, along with an explanation of the significance of its numerator and denominator values.

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

From the notes:

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

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

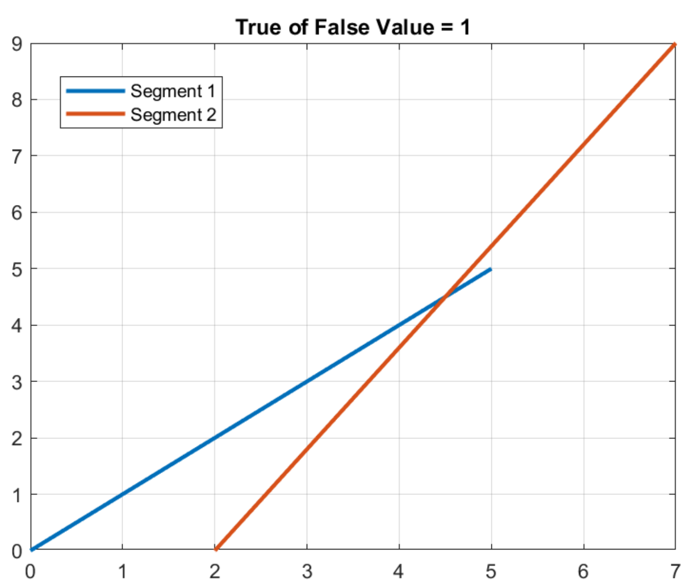
(iii) if den $\neq 0$, then the two lines are not parallel and therefore intersect at a single point.

Verify:

Input

```
%% Part 5

% Varify check
p1 = [0 0];
p2 = [5 5];
p3 = [2 0];
p4 = [7 9];

TF = doTwoSegmentsIntersect(p1,p2,p3,p4)
```

Output

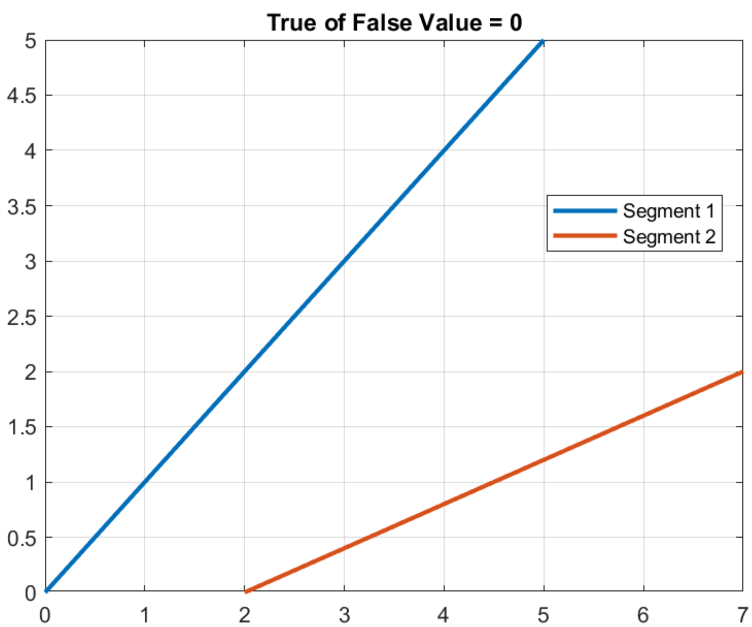


The output is one since they do intersect, 1== true.

Input

```
%Varify check
p1 = [0 0];
p2 = [5 5];
p3 = [2 0];
p4 = [7 2];

TF = doTwoSegmentsIntersect(p1,p2,p3,p4)

```

Output



The output is a zero this time since the segments do not interact.

Error check:

Input

```
% error check
p1 = [0 0 1];
p2 = [5 5 2];
p3 = [2 0 1];
p4 = [7 2 4];

TF = doTwoSegmentsIntersect(p1,p2,p3,p4)
```

Output

```
Error using doTwoSegmentsIntersect (line 4)
Must be two dimension points
```

Part 6: Do Two Convex Polgons Intersect

Algo

```
function TF = doTwoConvexPolygonsIntersect(P1,P2)

TF = 0;

LengthsP1 = length(P1);
LengthsP2 = length(P2);

Index1 = [1:LengthsP1 1];
Index2 = [1:LengthsP2 1];

for j = 1:LengthsP1
point1_current = P1(Index1(j),:);
point1_next    = P1(Index2(j+1),:);

    for i = 1:LengthsP2
        point2_current = P2(Index2(i),:);
        point2_next    = P2(Index2(i+1),:);
        Intersection_Check = doTwoSegmentsIntersect(point1_current,point1_next,point2_current,point2_next);

        if Intersection_Check == 1
            TF = 1;
            break
        end
    end
    if TF == 1
        break
    end
end

end
```

This function pretty much just runs the "doTwoSegentsIntersect" for each segment making up the polygons for every segment option. So the functions loops run a tottle of N*M times where N = the number of segments in Polygon 1 and M = a number of segments in Polygon 2.

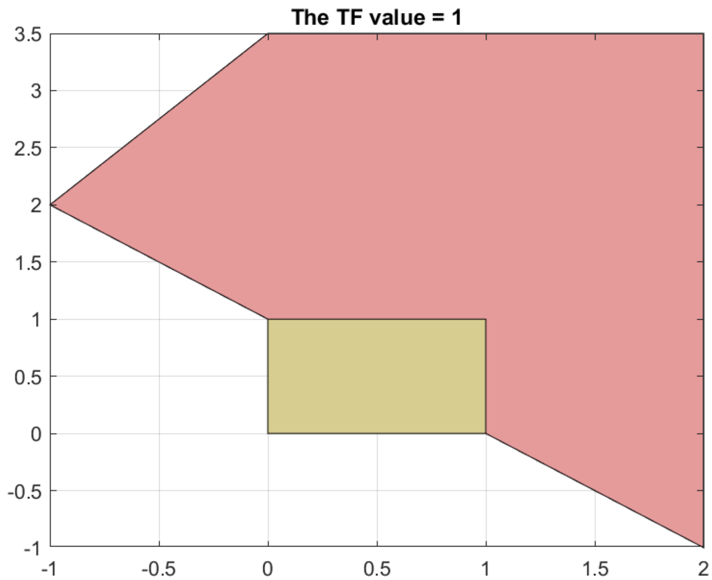
Verify:

Input

```
%% Part 6

P2 = [1 1; 0 1; 0 0; 1 0];
P1 = [2 0.5+3; 0 0.5+3; -1 -1+3; 2 -1];

P(:,:,1) = P1;
P(:,:,2) = P2;
for j = 1:length(P(1,1,:))
    X = P(:,1,j);
    Y = P(:,2,j);
    fill(X, Y, [rand(1) rand(1) rand(1)]);
    hold on
    grid on
end


TF = doTwoConvexPolygonsIntersect(P1,P2);
```

The for loop just graphs the polygons for us.

Output

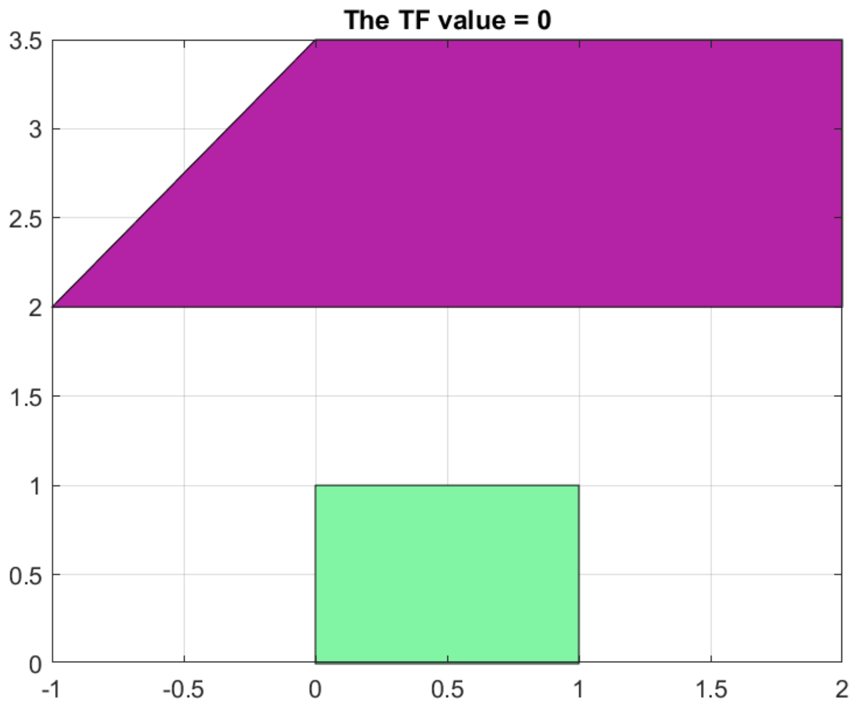


Since the TF value is 1, this indicates the the polygons do touch each other.

Input

```
P2 = [1 1; 0 1; 0 0; 1 0];
P1 = [2 0.5+3; 0 0.5+3; -1 -1+3; 2 2];

P(:,:,1) = P1;
P(:,:,2) = P2;

for j = 1:length(P(1,1,:))
    X = P(:,1,j);
    Y = P(:,2,j);
    fill(X, Y, [rand(1) rand(1) rand(1)]);
    hold on
    grid on
end

TF = doTwoConvexPolygonsIntersect(P1,P2);

title("The TF value = "+ TF)
```

Output



The TF value is zero indicating that the polygons do not touch each other.

Error check:

Input

```
% Errorr check

P2 = [1 1; 0 1; 0 0; 1 0];
P1 = [1 1; 1 0];

TF = doTwoConvexPolygonsIntersect(P1,P2);

title("The TF value = "+ TF)
```

Output

```
doTwoConvexPolygonsIntersect
(line 4)
P1 must be a polygon
```