# Annex A: Index of the MATLAB files

1. **Single Obstacle Algorithms**
   1. Projection Method
      1. Projection (The main algorithm)
      2. Projection\_LP (The algorithm producing the longer path)
      3. plot\_circle
   2. Tangent Method
      1. Tangent (The main algorithm)
      2. plot\_circle
2. **Multiple Obstacles Algorithms**
   1. Parallel Method
      1. Parallel (The main algorithm)
      2. plot\_circle
   2. Segment Method
      1. check\_intersection
      2. plot\_circle
      3. plot\_obstacles
      4. Segment (The main algorithm)
      5. Segment\_NoPrint (The main algorithm printing only the shortest and optimized paths)
      6. Segment\_Order (This algorithm is avoiding the obstacles using the order they are given in the obstacle array)
      7. Segment\_Random (The algorithm is avoiding the obstacles using a random order)
      8. vessel\_find\_path
      9. vessel\_fun
   3. Segment Method Virtual
      1. check\_intersection
      2. find\_route
      3. plot\_circle
      4. plot\_obstacles
      5. Segment\_Virtual (The main algorithm with the addition of the virtual waypoint functionality)
      6. vessel\_find\_path
      7. vessel\_fun

# Annex B: MATLAB code for the Projection Algorithm

Following the MATLAB code for the collision avoidance algorithm based on the Projection Method is presented. The code has been written using the R2017b version of MATLAB and its main objective is to determine a collision free path around a fixed obstacle.

|  |
| --- |
| **Projection.m** |
| %% - Autonomous USV Collision Avoidance Algorithm - %%  % This code uses an algorithm based on the Projection Method %  % to find a collision free path around an obstacle %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    % Clean the workspace and close the open figures  clear  clc  close all    %% Parameters - Setting up the problem    % Start point S (XS, YS)  XS = 2;  YS = 1;    % Target point T (XT, YT)  XT = 20;  YT = 18;    % Obstacle representation: circle with centre at (XO, YO) and radius RO  XO = 10;  YO = 8;  RO = 4;    % Safety radius RB  % Was set equal to the radius of the vessel region (RV), for the simulations  RB = 0.571;    %% Core calculations    % Position vectors, to be used for plotting  X\_pos = XS;  Y\_pos = YS;    % Find the straight-line equation (Y = a\*X+b) connecting the start and  % target points  a = (YS - YT)/(XS - XT);  b = (XS\*YT - XT\*YS)/(XS - XT);    % Find the determinant radius RD  RD = RO + RB;    % Plot the centre of the circle  plot(XO, YO, '.b')  hold on  axis equal; box on;  xlabel('X (m)'); ylabel('Y (m)');    % Plot the circle (X-XO)^2+(Y-YO)^2=RD^2 with XO, YO and RD red --- line  plot\_circle(XO, YO, RD, 'r');    % Plot the circle (X-XO)^2+(Y-YO)^2=RO^2 with XO, YO and RO blue --- line  plot\_circle(XO, YO, RO, 'b');    % Add description (text) to data points  txt1 = ' Start point';  text(XS,YS,txt1,'VerticalAlignment','top')    txt2 = ' Target point';  text(XT,YT,txt2,'VerticalAlignment','top')    % Check if the vessel is already inside obstacle region  if (sqrt((XS - XO)^2 + (YS - YO)^2) < RD) || (sqrt((XT - XO)^2 + (YT - YO)^2) < RD)  error('Start/Target point(s) inside obstacle region')  end    % Calculate the length of the straight line from S to T  L = sqrt((XT - XS)^2 + (YT - YS)^2);  disp(['Straight-line length: ' num2str(L)])    % Find the intersection point(s) between the line and the circle (equation: (X-XO)^2+(Y-YO)^2 = RD^2)  % Need to solve: (a^2 + 1)\*X^2 + 2\*(a\*b - a\*YO - XO)\*X + (YO^2 - RD^2 + XO^2 - 2\*b\*YO + b^2) = 0  % Substitutions in the quadratic equation  A = (a^2 + 1);  B = 2\*(a\*b - a\*YO - XO);  C = (YO^2 - RD^2 + XO^2 - 2\*b\*YO + b^2);    % Determinant calculation  D = B^2 - 4\*A\*C;    %% Finding the relative position between the straight line and the obstacle    % Check if the straight line intersects with the obstacle  if ((XS == XT) && ((XS <= XO - RD) || (XS >= XO + RD)))  % Route parallel to Y-axis and no intersection point  % Continue moving on the straight line  NoIntersection = true;  disp('No Intersection')    elseif ((YS == YT) && ((YS <= YO - RD) || (YS >= YO + RD)))  % Route parallel to X-axis and no intersection point  % Continue moving on the straight line  NoIntersection = true;  disp('No Intersection')    elseif (D <= 0) % Check determinant value  % 0 or 1 solutions, i.e. none or one intersection point  % Continue moving on the straight line  NoIntersection = true;  disp('No Intersection')    else % Two solutions (intersection points), with coordinates (X1, Y1) & (X2, Y2)  if (XS == XT) % Route parallel to Y-axis  X1 = XS;  X2 = XS;  Y1 = YO - sqrt (RD^2 - (XS-XO)^2);  Y2 = YO + sqrt (RD^2 - (XS-XO)^2);    elseif (YS == YT) % Route parallel to X-axis  X1 = XO - sqrt (RD^2 - (YS-YO)^2);  X2 = XO + sqrt (RD^2 - (YS-YO)^2);  Y1 = YS;  Y2 = YS;    else % Route with random orientation  X1 = (-B + sqrt(B^2 - 4\*A\*C))/(2\*A);  X2 = (-B - sqrt(B^2 - 4\*A\*C))/(2\*A);  Y1 = a\*X1 + b;  Y2 = a\*X2 + b;  end    % Plot the intersection points  plot(X1, Y1, 'xk')  plot(X2, Y2, 'xk')    % Check if the intersection points belong to the line segment from S to T  if (XT > XS)  if (XS < X1 && X2 < XT)  NoIntersection = false;  disp('Intersection')  else  NoIntersection = true;  disp('No Intersection')  end    elseif (XT == XS)  if (YT > YS)  if (YS < Y1 && Y2 < YT)  NoIntersection = false;  disp('Intersection')  else  NoIntersection = true;  disp('No Intersection')  end  else % YT < YS  if (YT < Y1 && Y2 < YS)  NoIntersection = false;  disp('Intersection')  else  NoIntersection = true;  disp('No Intersection')  end  end    else % XT < XS  if (XT < X1 && X2 < XS)  NoIntersection = false;  disp('Intersection')  else  NoIntersection = true;  disp('No Intersection')  end  end  end    %% Way of updating position vector    if (NoIntersection)  % Continue moving on the straight line  % Update position vector  X\_pos = [X\_pos XT];  Y\_pos = [Y\_pos YT];    else % Determine the direction to turn  if (XS == XT) % Extra criterion because in this case YCRIT=YO and cannot determine the direction to turn  if (XS >= XO)  if (YS > YO) % First quadrant  CCW = true;  disp('First quadrant')  else % YS < YO Fourth quadrant  CCW = false;  disp('Fourth quadrant')  end  else % XS < XO  if (YS > YO) % Second quadrant  CCW = false;  disp('Second quadrant')  else % YS < YO Third quadrant  CCW = true;  disp('Third quadrant')  end  end  XC = XS;  YC = YO;    else % XS not equal with XT  if (YS == YT)  YCRIT = YS;  XC = XO;  YC = YS;  else % Calculate the YCRIT  YCRIT = a\*XO + b;  % Find the line equation (Y = a2\*X + b2) which is lateral to the  % initial one and is crossing from the centre of the obstacle  a2 = -1/a;  b2 = (a\*YO + XO)/a;  % Find the cross point of the two lines (XC, YC)  % Solve the system (Y = a\*X + b) and (Y = a2\*X + b2)  XC = (b2 - b)/(a - a2);  YC = a\*XC + b;  end    % If YCRIT>=YO then turn CCW angle f1 and CW f2  if (YCRIT >= YO)  CCW = true;  else % If YCRIT<YO then turn CW angle f1 and CCW f2  CCW = false;  end  end    % Calculate the distance from the centre of the obstacle to the cross point  LR = sqrt((XC - XO)^2 + (YC - YO)^2);    % Calculate the distance from the cross point to the circumference  LM = RD - LR;    % Calculate the distances from start point to cross point  tmp0 = sqrt((X1 - XS)^2 + (Y1 - YS)^2);  tmp1 = sqrt((X2 - XS)^2 + (Y2 - YS)^2);    % SET the smaller D1 and the bigger D2  if (tmp0 < tmp1)  D1 = tmp0;  D2 = tmp1;  else  D1 = tmp1;  D2 = tmp0;  end    % Calculate the distance from (X2, Y2) to the target point  D3 = L - D2;    % Calculate the distance between the cross points  LP = sqrt((X1 - X2)^2 + (Y1 - Y2)^2);    % Calculate the length of hypotenuse in the start triangle  L1 = sqrt(LM^2 + D1^2);    % Calculate the length of hypotenuse in the target triangle  L2 = sqrt(LM^2 + D3^2);    % Calculate the first and second turn angle  % If YCRIT>=YO then turn CCW angle f1 and CW f2  if (CCW == true)  f1 = atan(LM/D1);  f2 = -atan(LM/D3);  else % If YCRIT<YO then turn CW angle f1 and CCW f2  f1 = -atan(LM/D1);  f2 = atan(LM/D3);  end    % Update position vector when there is intersection  if (XT > XS) % Forward motion  % Update position vector - first part  X\_pos = [X\_pos, (XS + L1\*cos(atan(a) + f1))];  Y\_pos = [Y\_pos, (YS + L1\*sin(atan(a) + f1))];    % Update position vector - second part  X\_pos = [X\_pos, (XS + L1\*cos(atan(a) + f1) + LP\*cos(atan(a)))];  Y\_pos = [Y\_pos, (YS + L1\*sin(atan(a) + f1) + LP\*sin(atan(a)))];    % Update position vector - third part  X\_pos = [X\_pos, (XS + L1\*cos(atan(a) + f1) + LP\*cos(atan(a)) + L2\*cos(atan(a) + f2))];  Y\_pos = [Y\_pos, (YS + L1\*sin(atan(a) + f1) + LP\*sin(atan(a)) + L2\*sin(atan(a) + f2))];    elseif (XS == XT) % Parallel to Y-axis motion  % Update position vector - first part  X\_pos = [X\_pos, (XS + sign(YS-YO)\*L1\*cos(pi/2 - f1))];  Y\_pos = [Y\_pos, (YS - sign(YS-YO)\*L1\*sin(pi/2 - f1))];    % Update position vector - second part  X\_pos = [X\_pos, (XS + sign(YS-YO)\*L1\*cos(pi/2 - f1) - sign(YS - YO)\*LP\*cos(pi/2))];  Y\_pos = [Y\_pos, (YS - sign(YS-YO)\*L1\*sin(pi/2 - f1) - sign(YS - YO)\*LP\*sin(pi/2))];    % Update position vector - third part  X\_pos = [X\_pos, (XS + sign(YS-YO)\*L1\*cos(pi/2 - f1) - sign(YS - YO)\*LP\*cos(pi/2) + sign(YS-YO)\*L2\*cos(pi/2 - f2))];  Y\_pos = [Y\_pos, (YS - sign(YS-YO)\*L1\*sin(pi/2 - f1) - sign(YS - YO)\*LP\*sin(pi/2) - sign(YS-YO)\*L2\*sin(pi/2 - f2))];    else % Backward motion  % Update position vector - first part  X\_pos = [X\_pos, (XS - L1\*cos(atan(a) - f1))];  Y\_pos = [Y\_pos, (YS - L1\*sin(atan(a) - f1))];    % Update position vector - second part  X\_pos = [X\_pos, (XS - L1\*cos(atan(a) - f1) - LP\*cos(atan(a)))];  Y\_pos = [Y\_pos, (YS - L1\*sin(atan(a) - f1) - LP\*sin(atan(a)))];    % Update position vector - third part  X\_pos = [X\_pos, (XS - L1\*cos(atan(a) - f1) - LP\*cos(atan(a)) - L2\*cos(atan(a) - f2))];  Y\_pos = [Y\_pos, (YS - L1\*sin(atan(a) - f1) - LP\*sin(atan(a)) - L2\*sin(atan(a) - f2))];  end    % Calculate the total distance  L\_travel = L1 + LP + L2;  disp(['Trajectory length: ' num2str(L\_travel)])    % Calculate the extra distance  L\_extra = L\_travel - L;  disp(['Extra distance travelled due to obstacle: ' num2str(L\_extra)])    end    % Plot trajectory with magenta dash-dot line  plot(X\_pos, Y\_pos, '-.om', 'LineWidth', 1.5)    % Plot line Y=a\*X+b with black dotted line  x = [XS, XT];  y = [YS, YT];  plot (x, y, ':xk') |

The following function has been used in all cited codes to plot the circles representing the obstacle and the obstacle region.

|  |
| --- |
| **plot\_circle.m** |
| %% - Function to plot circles - %%  % This code introduces a function to plot a circle %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    function h = plot\_circle(x,y,r,c)    th = 0:pi/50:2\*pi;  xunit = r \* cos(th) + x;  yunit = r \* sin(th) + y;  h = plot(xunit, yunit, c);    end |

# Annex C: MATLAB code for the Tangent Algorithm

Following the MATLAB code for the collision avoidance algorithm based on the Tangent Method is presented. The code has been written using the R2017b version of MATLAB and its main objective is to determine a collision free path around a fixed obstacle.

|  |
| --- |
| **Tangent.m** |
| %% - Autonomous USV Obstacle Avoidance Algorithm - %%  % This code uses an algorithm based on the Tangent Method %  % to find a collision free path around an obstacle %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    % Clean the workspace and close the open figures  clear  clc  close all    %% Parameters - Setting up the problem    % Start point S (XS, YS)  XS = -1;  YS = 7;    % Target point T (XT, YT)  XT = 12;  YT = 15;    % Obstacle representation: circle with centre at (XO, YO) and radius RO  XO = 5;  YO = 10;  RO = 3;    % Safety radius RB  % Was set equal to the radius of the vessel region (RV), for the simulations  RB = 0.571;    %% Core calculations    % Position vectors, to be used for plotting  X\_pos = XS;  Y\_pos = YS;    % Find the straight-line equation (Y = a\*X+b) connecting the initial and  % target points  a = (YS - YT)/(XS - XT);  b = (XS\*YT - XT\*YS)/(XS - XT);    % Find the determinant radius RD  RD = RO + RB;    % Plot the centre of the circle  plot(XO, YO, '.b')  hold on  axis equal; box on;  xlabel('X (m)'); ylabel('Y (m)');    % Plot the circle (X-XO)^2+(Y-YO)^2=RD^2 with XO, YO and RD red --- line  plot\_circle(XO, YO, RD, 'r');    % Plot the circle (X-XO)^2+(Y-YO)^2=RO^2 with XO, YO and RO blue --- line  plot\_circle(XO, YO, RO, 'b');    % Add description (text) to data points  txt1 = ' Start point';  text(XS,YS,txt1,'VerticalAlignment','top')    txt2 = ' Target point';  text(XT,YT,txt2,'VerticalAlignment','top')    % Check if the vessel is already inside obstacle region  if (sqrt((XS - XO)^2 + (YS - YO)^2) < RD) || (sqrt((XT - XO)^2 + (YT - YO)^2) < RD)  error('Start/Target point(s) inside obstacle region')  end    % Calculate the length of the straight line from S to T  L = sqrt((XT - XS)^2 + (YT - YS)^2);  disp(['Straight-line length: ' num2str(L)])    % Find the intersection point(s) between the line and the circle (equation: (X-XO)^2+(Y-YO)^2 = RD^2)  % Need to solve: (a^2 + 1)\*X^2 + 2\*(a\*b - a\*YO - XO)\*X + (YO^2 - RD^2 + XO^2 - 2\*b\*YO + b^2) = 0  % Substitutions in the quadratic equation  A = (a^2 + 1);  B = 2\*(a\*b - a\*YO - XO);  C = (YO^2 - RD^2 + XO^2 - 2\*b\*YO + b^2);    % Determinant calculation  D = B^2 - 4\*A\*C;    %% Finding the relative position between the straight line and the obstacle    % Check if the straight line intersects with the obstacle  if ((XS == XT) && ((XS <= XO - RD) || (XS >= XO + RD)))  % Route parallel to Y-axis and no intersection point  % Continue moving on the straight line  NoIntersection = true;  disp('No Intersection')    elseif ((YS == YT) && ((YS <= YO - RD) || (YS >= YO + RD)))  % Route parallel to X-axis and no intersection point  % Continue moving on the straight line  NoIntersection = true;  disp('No Intersection')    elseif (D <= 0) % Check determinant value  % 0 or 1 solutions, i.e. none or one intersection point  % Continue moving on the straight line  NoIntersection = true;  disp('No Intersection')    else % Two solutions (intersection points), with coordinates (X1, Y1) & (X2, Y2)  if (XS == XT) % Route parallel to Y-axis  X1 = XS;  X2 = XS;  Y1 = YO - sqrt (RD^2 - (XS-XO)^2);  Y2 = YO + sqrt (RD^2 - (XS-XO)^2);    elseif (YS == YT) % Route parallel to X-axis  X1 = XO - sqrt (RD^2 - (YS-YO)^2);  X2 = XO + sqrt (RD^2 - (YS-YO)^2);  Y1 = YS;  Y2 = YS;    else % Route with random orientation  X1 = (-B + sqrt(B^2 - 4\*A\*C))/(2\*A);  X2 = (-B - sqrt(B^2 - 4\*A\*C))/(2\*A);  Y1 = a\*X1 + b;  Y2 = a\*X2 + b;  end    % Plot the intersection points  plot(X1, Y1, 'xk')  plot(X2, Y2, 'xk')    % Check if the intersection points belong to the line segment from S to T  if (XT > XS)  if (XS < X1 && X2 < XT)  NoIntersection = false;  disp('Intersection')  else  NoIntersection = true;  disp('No Intersection')  end    elseif (XT == XS)  if (YT > YS)  if (YS < Y1 && Y2 < YT)  NoIntersection = false;  disp('Intersection')  else  NoIntersection = true;  disp('No Intersection')  end  else % YT < YS  if (YT < Y1 && Y2 < YS)  NoIntersection = false;  disp('Intersection')  else  NoIntersection = true;  disp('No Intersection')  end  end    else % XT < XS  if (XT < X1 && X2 < XS)  NoIntersection = false;  disp('Intersection')  else  NoIntersection = true;  disp('No Intersection')  end  end  end    %% Way of updating position vector    if (NoIntersection)  % Continue moving on the straight line  % Update position vector  X\_pos = [X\_pos XT];  Y\_pos = [Y\_pos YT];    else % Determine the direction to turn  % Finding the tangent lines and tangential points  % Solve Y = a1\*X + b1 and (X - XO)^2 + (Y - YO)^2 = RD^2 and require the  % determinant equal to zero in order to have one contact point  % syms X XO YO RD a1 b1  % eqn = (X - XO)^2 + (a1\*X + b1 - YO)^2 == RD^2;  % solx = solve(eqn, X)  % X1 = (XO + YO\*a1 - a1\*b1 + (RD^2\*a1^2 + RD^2 - XO^2\*a1^2 + 2\*XO\*YO\*a1 - 2\*XO\*a1\*b1 - YO^2 + 2\*YO\*b1 - b1^2)^(1/2))/(a1^2 + 1)  % X2 = (XO + YO\*a1 - a1\*b1 - (RD^2\*a1^2 + RD^2 - XO^2\*a1^2 + 2\*XO\*YO\*a1 - 2\*XO\*a1\*b1 - YO^2 + 2\*YO\*b1 - b1^2)^(1/2))/(a1^2 + 1)    % b1 = (YS - a1\*XS)  % syms X XO YO RD a1 YS XS  % eqn = RD^2\*a1^2 + RD^2 - XO^2\*a1^2 + 2\*XO\*YO\*a1 - 2\*XO\*a1\*(YS - a1\*XS) - YO^2 + 2\*YO\*(YS - a1\*XS) - (YS - a1\*XS)^2 == 0;  % solx = solve(eqn, a1)    % These are the straight-lines starting from S and been tangential to the obstacle  a11 = (XO\*YS - XO\*YO + XS\*YO - XS\*YS + RD\*(- RD^2 + XO^2 - 2\*XO\*XS + XS^2 + YO^2 - 2\*YO\*YS + YS^2)^(1/2))/(RD^2 - XO^2 + 2\*XO\*XS - XS^2);  a12 = -(XO\*YO - XO\*YS - XS\*YO + XS\*YS + RD\*(- RD^2 + XO^2 - 2\*XO\*XS + XS^2 + YO^2 - 2\*YO\*YS + YS^2)^(1/2))/(RD^2 - XO^2 + 2\*XO\*XS - XS^2);  b11 = (YS - a11\*XS);  b12 = (YS - a12\*XS);    % Equation (Y = a11v\*X + b11v) for the line vertical to the first  % tangential line (Y = a11\*X + b11)  a11v = -1/a11;  b11v = YO - a11v\*XO;    % Finding the cross point C11 (XC11, YC11) of two lines (Y = a11v\*X + b11v and Y = a11\*X + b11) = the point where the tangent line  % meet the circle  XC11 = (b11v - b11)/(a11 - a11v);  YC11 = a11\*XC11 + b11;    % Equation (Y = a12v\*X + b12v) for the line vertical to the first  % tangential line (Y = a12\*X + b12)  a12v = -1/a12;  b12v = YO - a12v\*XO;    % Finding the cross point C12 (XC12, YC12) of two lines (Y = a12v\*X + b12v and Y = a12\*X + b12) = the point where the tangent line  % meet the circle  XC12 = (b12v - b12)/(a12 - a12v);  YC12 = a12\*XC12 + b12;    % Plot tangential points C11 and C12  plot(XC11, YC11, 'ob')  plot(XC12, YC12, 'ok')    % This are the straight-lines starting from T and been tangential to the circle  a21 = (XO\*YT - XO\*YO + XT\*YO - XT\*YT + RD\*(- RD^2 + XO^2 - 2\*XO\*XT + XT^2 + YO^2 - 2\*YO\*YT + YT^2)^(1/2))/(RD^2 - XO^2 + 2\*XO\*XT - XT^2);  a22 = -(XO\*YO - XO\*YT - XT\*YO + XT\*YT + RD\*(- RD^2 + XO^2 - 2\*XO\*XT + XT^2 + YO^2 - 2\*YO\*YT + YT^2)^(1/2))/(RD^2 - XO^2 + 2\*XO\*XT - XT^2);  b21 = (YT - a21\*XT);  b22 = (YT - a22\*XT);    % Equation (Y = a21v\*X + b21v) for the line vertical to the first  % tangential line (Y = a21\*X + b21)  a21v = -1/a21;  b21v = YO - a21v\*XO;    % Finding the cross point C21 (XC21, YC21) of two lines (Y = a21v\*X + b21v and Y = a21\*X + b21) = the point where the tangent line  % meet the circle  XC21 = (b21v - b21)/(a21 - a21v);  YC21 = a21\*XC21 + b21;    % Equation (Y = a22v\*X + b22v) for the line vertical to the first  % tangential line (Y = a22\*X + b22)  a22v = -1/a22;  b22v = YO - a22v\*XO;    % Finding the cross point C22 (XC22, YC22) of two lines (Y = a22v\*X + b22v and Y = a22\*X + b22) = the point where the tangent line  % meet the circle  XC22 = (b22v - b22)/(a22 - a22v);  YC22 = a22\*XC22 + b22;    % Plot tangential points C21 and C22  plot(XC21, YC21, 'ok')  plot(XC22, YC22, 'ob')    if (XS == XT) % Extra criterion because in this case YCRIT=YO and cannot determine the direction to turn  if (XS >= XO)  if (YS > YO) % First quadrant  CCW = true;  disp('First quadrant')  else % YS < YO Fourth quadrant  CCW = false;  disp('Fourth quadrant')  end  else % XS < XO  if (YS > YO) % Second quadrant  CCW = false;  disp('Second quadrant')  else % YS < YO Third quadrant  CCW = true;  disp('Third quadrant')  end  end  XC = XS;  YC = YO;    else % XS not equal with XT  if (YS == YT)  YCRIT = YS;  XC = XO;  YC = YS;  else % Calculate the YCRIT  YCRIT = a\*XO + b;  % Find the line equation (Y = av\*X + bv) which is lateral to the  % initial one (S to T) and is crossing from the centre of the obstacle  av = -1/a;  bv = (a\*YO + XO)/a;  % Find the cross point of the two lines (XC, YC)  % Solve the system (Y = a\*X + b) and (Y = av\*X + bv)  XC = (bv - b)/(a - av);  YC = a\*XC + b;  end    % If YCRIT>=YO then turn CCW angle f1 and CW f2  if (YCRIT >= YO)  CCW = true;  else % If YCRIT<YO then turn CW angle f1 and CCW f2  CCW = false;  end  end    % %Plot the cross point  % plot(XC,YC,'+m')    % Calculate the distance from the centre of the obstacle (XO, YO) to the  % cross point (XC, YC)  LR = sqrt((XC - XO)^2 + (YC - YO)^2);    % Calculate the distance from the cross point to the circumference  LM = RD - LR;    if (((CCW == true) && (XT<XS)) || ((CCW == false) && (XT>=XS))) % turn CW and use C11 & C22    % Find the line equation (Y = as\*X + bs) between S to C11  as = a11;  bs = b11;    % Find the line equation (Y = at\*X + bt) between T to C22  at = a22;  bt = b22;    if (XT == XS)  if (XS < XO)  % Find the cross point (XC1, YC1) between (X = XS - LM) and (Y = as\*X + bs)  XC1 = XS - LM;  YC1 = as\*XC1 + bs;  % Find the cross point (XC2, YC2) between (X = XS - LM) and (Y = at\*X + bt)  XC2 = XS - LM;  YC2 = at\*XC2 + bt;    else % XS >= XO  % Find the cross point (XC1, YC1) between (X = XS + LM) and (Y = as\*X + bs)  XC1 = XS + LM;  YC1 = as\*XC1 + bs;  % Find the cross point (XC2, YC2) between (X = XS + LM) and (Y = at\*X + bt)  XC2 = XS + LM;  YC2 = at\*XC2 + bt;  end  else  % Find the line equation (Y = a\*X + bp) which is parallel to the (Y = a\*X+b) and is tangential to the circle  if (XT > XS)  bp = b - LM/cos(atan(a));  else  bp = b + LM/cos(atan(a));  end    % Another way to find bp is by solving the system (Y = a\*X + bp) and (Y = av\*X+bv) for (X,Y) and demand X,Y to be valid for the equation (X-XO)^2+(Y-YO)^2=RD^2  % syms bv bp a av XO YO RD  % eqn = ((bv - bp)/(a - av) - XO)^2 + (av\*(bv - bp)/(a - av) + bv - YO)^2 == RD^2;  % solx = solve(eqn, bp)  % bp1 = (bv - XO\*a + XO\*av + a\*(RD^2\*av^2 + RD^2 - XO^2\*av^2 + 2\*XO\*YO\*av - 2\*XO\*av\*bv - YO^2 + 2\*YO\*bv - bv^2)^(1/2) - av\*(RD^2\*av^2 + RD^2 - XO^2\*av^2 + 2\*XO\*YO\*av - 2\*XO\*av\*bv - YO^2 + 2\*YO\*bv - bv^2)^(1/2) + YO\*av^2 - YO\*a\*av + a\*av\*bv)/(av^2 + 1)  % bp2 = (bv - XO\*a + XO\*av - a\*(RD^2\*av^2 + RD^2 - XO^2\*av^2 + 2\*XO\*YO\*av - 2\*XO\*av\*bv - YO^2 + 2\*YO\*bv - bv^2)^(1/2) + av\*(RD^2\*av^2 + RD^2 - XO^2\*av^2 + 2\*XO\*YO\*av - 2\*XO\*av\*bv - YO^2 + 2\*YO\*bv - bv^2)^(1/2) + YO\*av^2 - YO\*a\*av + a\*av\*bv)/(av^2 + 1)    % Find the cross point (XC1, YC1) between (Y = a\*X + bp) and (Y = as\*X + bs)  XC1 = (bp - bs)/(as - a);  YC1 = as\*XC1 + bs;    % Find the cross point (XC2, YC2) between (Y = a\*X + bp) and (Y = at\*X + bt)  XC2 = (bp - bt)/(at - a);  YC2 = at\*XC2 + bt;  end  else % (((YCRIT>YO) && (XT>XS)) || ((YCRIT<YO) && (XT<XS))) turn CCW and use C12 & C21  % Find the line equation (Y = as\*X + bs) between S to C12  as = a12;  bs = b12;    % Find the line equation (Y = at\*X + bt) between T to C21  at = a21;  bt = b21;    if (XT == XS)  if (XS < XO)  % Find the cross point (XC1, YC1) between (X = XS - LM) and (Y = as\*X + bs)  XC1 = XS - LM;  YC1 = as\*XC1 + bs;  % Find the cross point (XC2, YC2) between (X = XS - LM) and (Y = at\*X + bt)  XC2 = XS - LM;  YC2 = at\*XC2 + bt;    else % XS >= XO  % Find the cross point (XC1, YC1) between (X = XS + LM) and (Y = as\*X + bs)  XC1 = XS + LM;  YC1 = as\*XC1 + bs;    % Find the cross point (XC2, YC2) between (X = XS + LM) and (Y = at\*X + bt)  XC2 = XS + LM;  YC2 = at\*XC2 + bt;  end  else  if (XT > XS)  bp = b + LM/cos(atan(a));  else  bp = b - LM/cos(atan(a));  end    % Find the cross point (XC1, YC1) between (Y = a\*X + bp) and (Y = as\*X + bs)  XC1 = (bp - bs)/(as - a);  YC1 = as\*XC1 + bs;  % Find the cross point (XC2, YC2) between (Y = a\*X + bp) and (Y = at\*X + bt)  XC2 = (bp - bt)/(at - a);  YC2 = at\*XC2 + bt;  end  end    % Plot some important points for debugging  plot(XC1, YC1, 'xk')  plot(XC2, YC2, 'xk')    % Calculate the distance between the cross points  LP = sqrt((XC1 - XC2)^2 + (YC1 - YC2)^2);    % Calculate the distance between the S and the C1\_ points  L1 = sqrt((XS - XC1)^2 + (YS - YC1)^2);    % Calculate the distance between the T and the C2\_ points  L2 = sqrt((XT - XC2)^2 + (YT - YC2)^2);    % Calculate the first and second turn angle  % If YCRIT>=YO then turn CCW angle f1 and CW f2  if (CCW == true)  f1 = asin(LM/L1);  f2 = -asin(LM/L2);  else % If YCRIT<YO then turn CW angle f1 and CCW f2  f1 = -asin(LM/L1);  f2 = asin(LM/L2);  end    %% Way of updating position vector    % Update position vector when there is intersection  if (XT > XS) % Forward motion  % Update position vector - first part  X\_pos = [X\_pos, (XS + L1\*cos(atan(a) + f1))];  Y\_pos = [Y\_pos, (YS + L1\*sin(atan(a) + f1))];    % Update position vector - second part  X\_pos = [X\_pos, (XS + L1\*cos(atan(a) + f1) + LP\*cos(atan(a)))];  Y\_pos = [Y\_pos, (YS + L1\*sin(atan(a) + f1) + LP\*sin(atan(a)))];    % Update position vector - third part  X\_pos = [X\_pos, (XS + L1\*cos(atan(a) + f1) + LP\*cos(atan(a)) + L2\*cos(atan(a) + f2))];  Y\_pos = [Y\_pos, (YS + L1\*sin(atan(a) + f1) + LP\*sin(atan(a)) + L2\*sin(atan(a) + f2))];    elseif (XS == XT) % Parallel to Y-axis motion  % Update position vector - first part  X\_pos = [X\_pos, (XS + sign(YS-YO)\*L1\*cos(pi/2 - f1))];  Y\_pos = [Y\_pos, (YS - sign(YS-YO)\*L1\*sin(pi/2 - f1))];    % Update position vector - second part  X\_pos = [X\_pos, (XS + sign(YS-YO)\*L1\*cos(pi/2 - f1) - sign(YS - YO)\*LP\*cos(pi/2))];  Y\_pos = [Y\_pos, (YS - sign(YS-YO)\*L1\*sin(pi/2 - f1) - sign(YS - YO)\*LP\*sin(pi/2))];    % Update position vector - third part  X\_pos = [X\_pos, (XS + sign(YS-YO)\*L1\*cos(pi/2 - f1) - sign(YS - YO)\*LP\*cos(pi/2) + sign(YS-YO)\*L2\*cos(pi/2 - f2))];  Y\_pos = [Y\_pos, (YS - sign(YS-YO)\*L1\*sin(pi/2 - f1) - sign(YS - YO)\*LP\*sin(pi/2) - sign(YS-YO)\*L2\*sin(pi/2 - f2))];    else % Backward motion  % Update position vector - first part  X\_pos = [X\_pos, (XS - L1\*cos(atan(a) - f1))];  Y\_pos = [Y\_pos, (YS - L1\*sin(atan(a) - f1))];    % Update position vector - second part  X\_pos = [X\_pos, (XS - L1\*cos(atan(a) - f1) - LP\*cos(atan(a)))];  Y\_pos = [Y\_pos, (YS - L1\*sin(atan(a) - f1) - LP\*sin(atan(a)))];    % Update position vector - third part  X\_pos = [X\_pos, (XS - L1\*cos(atan(a) - f1) - LP\*cos(atan(a)) - L2\*cos(atan(a) - f2))];  Y\_pos = [Y\_pos, (YS - L1\*sin(atan(a) - f1) - LP\*sin(atan(a)) - L2\*sin(atan(a) - f2))];  end    % Calculate the total distance  L\_travel = L1 + LP + L2;  disp(['Trajectory length: ' num2str(L\_travel)])    % Calculate the extra distance  L\_extra = L\_travel - L;  disp(['Extra distance travelled due to obstacle: ' num2str(L\_extra)])    end    % Plot trajectory with magenta dash-dot line  plot(X\_pos, Y\_pos, '-.om', 'LineWidth', 1.5)    % Plot line Y=a\*X+b with black dotted line  x = [XS, XT];  y = [YS, YT];  plot (x, y, ':xk') |

# Annex D: MATLAB code for the Parallel Algorithm

Following the MATLAB code for the path planning algorithm based on the Parallel Method is presented. The code has been written using the R2017b version of MATLAB and its main objective is to determine a safe path, in a multiple obstacles domain, guiding the vessel to its destination.

|  |
| --- |
| **Parallel.m** |
| %% - Autonomous USV Path Planning Algorithm - %%  % This code uses an algorithm based on the projection collision avoidance %  % method to find a path from the start point to the target point %  % keeping the tangential to the obstacles segments parallel %  % to the straight line connecting S and T %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    % Clean the workspace and close the open figures  clear  clc  close all    %% Parameters - Setting up the problem    % Start point S (XS, YS)  XS = 0;  YS = 10;    % Target point T (XT, YT)  XT = 53;  YT = 43;    % Obstacle representation: circle with centre at (XO, YO) and radius RO  XO = [5, 13, 23, 42];  YO = [12, 20, 35, 36];  RO = [2, 4, 6, 8];    % Safety radius RB  % Was set equal to the radius of the vessel region (RV), for the simulations  RB = 0.571;    % Number of obstacles N  N = length(XO);    %% Plotting basic features    % Add description to data points S and T  txt1 = ' Start point';  text(XS,YS,txt1,'VerticalAlignment','bottom')  hold on; axis equal; box on; xlabel('X (m)'); ylabel('Y (m)');    txt2 = ' Target point';  text(XT,YT,txt2,'VerticalAlignment','bottom')    % % Update position vector  X\_pos = [XS];  Y\_pos = [YS];    % Plot straight line from S to T with black dotted line  plot([XS XT], [YS YT], ':xk')    % Calculate the length of the straight line from S to T  Ls = sqrt((XT - XS)^2 + (YT - YS)^2);      for i=1:N  % Calculate the determinant radius  RD(i) = RO(i) + RB;  % Plot centre of obstacle circle  plot(XO(i), YO(i), '.b')  % Plot the circle with RO radius  plot\_circle(XO(i), YO(i), RO(i), 'b');  % Plot the circle with RD radius  plot\_circle(XO(i), YO(i), RD(i), 'r');  end    %% Core calculations    % Find the straight-line equation (Y = a\*X+b) connecting the start and target point  a = (YS - YT)/(XS - XT);  b = (XS\*YT - XT\*YS)/(XS - XT);    % Setting the order of the obstacles  for i=1:N  % The coordinates of the first point in the initial system  XFI(i)=XO(i) - RD(i)\*cos(a);  YFI(i)=YO(i) - RD(i)\*sin(a);  % The coordinates of the first point in the rotated system  XFR(i)=XFI(i)\*cos(a) + YFI(i)\*sin(a);  YFR(i)=YFI(i)\*cos(a) - XFI(i)\*sin(a);  end    % Creating a matrix to know the order of the obstacles, moving parallel to  % the S to T straight-line  [B,j]=sort(XFR);    for i=1:N  % Check if boat is already inside obstacle region  if (sqrt((XS - XO(j(i)))^2 + (YS - YO(j(i)))^2) < RD(j(i))) || (sqrt((XT - XO(j(i)))^2 + (YT - YO(j(i)))^2) < RD(j(i)))  error('Start/Target point(s) inside obstacle region')  end    % Check if path is unobstructed by obstacle by checking if both S and T points  % are in the same quadrant with reference to the centre of the obstacle circle  path\_unobstructed = (sign(XS - XO(j(i))) == sign(XT - XO(j(i)))) && (sign(YS - YO(j(i))) == sign(YT - YO(j(i))));  end    %% Basic calculations for the obstacles    for i=1:N % N is the number of the obstacle    % I need to know the straight line equation that the vessel is following  % Initially was Y=a\*X+b connecting the S and T points.  % After every run the straight line equation update itself.  % The new b is calculated in the end of the loop  % No need to calculate a new 'a' because all the lines are parallel to the  % initial one.    % Find the cross points between the straight line and the (i) circle  A(i) = (a^2 + 1);  B(i) = 2\*(a\*b - a\*YO(j(i)) - XO(j(i)));  C(i) = (YO(j(i))^2 - RD(j(i))^2 + XO(j(i))^2 - 2\*b\*YO(j(i)) + b^2);    % Determinant calculation  D(j(i)) = B(i)^2 - 4\*A(i)\*C(i);    % If D(j(i))<=0 then solve straight line equation with the next (i+1)circle  if (D(j(i))<=0)  disp('Obstacle skipped')  continue  end    % If D(j(i))>0 then two solutions  % The cross points (X1,Y1) & (X2,Y2)  X1(i) = (-B(i) + sqrt(B(i)^2 - 4\*A(i)\*C(i)))/(2\*A(i));  X2(i) = (-B(i) - sqrt(B(i)^2 - 4\*A(i)\*C(i)))/(2\*A(i));  Y1(i) = a\*X1(i) + b;  Y2(i) = a\*X2(i) + b;    % % Plot the cross points  plot(X1(i), Y1(i), 'xk')  plot(X2(i), Y2(i), 'xk')    % Find the line equation (Y = av\*X + bv) which is vertical to the initial one and is crossing from the centre of the (i) circle  av(i) = -1/a;  bv(i) = (a\*YO(j(i)) + XO(j(i)))/a;    % Find the cross point (XC, YC) of the two lines. Solve the system (Y = a\*X + b) and (Y = av\*X + bv)  XC(i) = (bv(i) - b)/(a - av(i));  YC(i) = a\*XC(i) + b;    % % Plot some important points  % plot(XC(i), YC(i), 'or')    % Calculate the distance between the cross points  LP(i) = sqrt((X1(i) - X2(i))^2 + (Y1(i) - Y2(i))^2);    % Calculate the distance from the centre of the (i) circle to the cross point (XC, YC)  LR(i) = sqrt((XC(i) - XO(j(i)))^2 + (YC(i) - YO(j(i)))^2);    % Calculate the distance from the cross point to the circumference  LM(i) = RD(j(i)) - LR(i);  LN(i) = RD(j(i)) + LR(i);    % Calculate the distances from start point S to cross point  tmp0 = sqrt((X1(i) - XS)^2 + (Y1(i) - YS)^2);  tmp1 = sqrt((X2(i) - XS)^2 + (Y2(i) - YS)^2);    % SET the smaller D1 and the bigger D2  if (tmp0 < tmp1)  D1 = tmp0;  D2 = tmp1;  else  D1 = tmp1;  D2 = tmp0;  end    % Calculate the YCRIT for circle  YCRIT(i) = a\*XO(j(i)) + b;    % If YCRIT>=YO then turn CCW angle f  if (YCRIT(i) >= YO(j(i)))  f(i) = atan(LM(i)/D1);    % Calculate the distance between start point and the first manoeuvre point  L(i) = sqrt(LM(i)^2 + D1^2);  % Calculate the end points of the first manoeuvre  Xend1(i) = XS + L(i)\*cos(atan(a) + f(i));  Yend1(i) = YS + L(i)\*sin(atan(a) + f(i));  Xend2(i) = XS + L(i)\*cos(atan(a) + f(i)) + LP(i)\*cos(atan(a));  Yend2(i) = YS + L(i)\*sin(atan(a) + f(i)) + LP(i)\*sin(atan(a));    % Plot each manoeuvre end point  plot(Xend2(i),Yend2(i),'+r')    % Check the maximum allowable turn angle  at2(i) = -(XO(j(i))\*YO(j(i)) - XO(j(i))\*Yend2(i) - Xend2(i)\*YO(j(i)) + Xend2(i)\*Yend2(i) + RD(j(i))\*(- RD(j(i))^2 + XO(j(i))^2 - 2\*XO(j(i))\*Xend2(i) + Xend2(i)^2 + YO(j(i))^2 - 2\*YO(j(i))\*Yend2(i) + Yend2(i)^2)^(1/2))/(RD(j(i))^2 - XO(j(i))^2 + 2\*XO(j(i))\*Xend2(i) - Xend2(i)^2);  limit(i) = atan(at2(i)) - atan(a);    % Must compare f(i from 2 to 5) with limit (i from 1 to 4)  if (i > 1)  if (abs(limit(i-1)) > abs(f(i)))  % disp('allow');  else % limit  % disp('do not allow');  % Calculate the distance between start point and the first manoeuvre point  L(i) = sqrt(LN(i)^2 + D1^2);  % Then turn CW angle f  f(i) = -atan(LN(i)/D1);  end  end    else % If YCRIT<YO then turn CW angle f  f(i) = -atan(LM(i)/D1);  % Calculate the distance between start point and the first manoeuvre point  L(i) = sqrt(LM(i)^2 + D1^2);  % Calculate the end points of the first manoeuvre  Xend1(i) = XS + L(i)\*cos(atan(a) + f(i));  Yend1(i) = YS + L(i)\*sin(atan(a) + f(i));  Xend2(i) = XS + L(i)\*cos(atan(a) + f(i)) + LP(i)\*cos(atan(a));  Yend2(i) = YS + L(i)\*sin(atan(a) + f(i)) + LP(i)\*sin(atan(a));    % Plot each manoeuvre end point  plot(Xend2(i),Yend2(i),'+r')    % Check the maximum allowable turn angle  at1(i) = (XO(j(i))\*Yend2(i) - XO(j(i))\*YO(j(i)) + Xend2(i)\*YO(j(i)) - Xend2(i)\*Yend2(i) + RD(j(i))\*(- RD(j(i))^2 + XO(j(i))^2 - 2\*XO(j(i))\*Xend2(i) + Xend2(i)^2 + YO(j(i))^2 - 2\*YO(j(i))\*Yend2(i) + Yend2(i)^2)^(1/2))/(RD(j(i))^2 - XO(j(i))^2 + 2\*XO(j(i))\*Xend2(i) - Xend2(i)^2);  limit(i) = pi + atan(at1(i)) - atan(a);    if (i > 1)  if (abs(limit(i-1)) > abs(f(i)))  % disp('allow');  else % limit  % disp('do not allow');  % Calculate the distance between start point and the first manoeuvre point  L(i) = sqrt(LN(i)^2 + D1^2);  % Then turn CCW angle f  f(i) = atan(LN(i)/D1);  end  end    end    %% Updates    % Update position vector - side section  X\_pos = [X\_pos, (XS + L(i)\*cos(atan(a) + f(i)))];  Y\_pos = [Y\_pos, (YS + L(i)\*sin(atan(a) + f(i)))];    % Update position vector - parallel section  X\_pos = [X\_pos, (XS + L(i)\*cos(atan(a) + f(i)) + LP(i)\*cos(atan(a)))];  Y\_pos = [Y\_pos, (YS + L(i)\*sin(atan(a) + f(i)) + LP(i)\*sin(atan(a)))];    % Upadate XS YS and b for the next repetition  XS = XS + L(i)\*cos(atan(a) + f(i)) + LP(i)\*cos(atan(a));  YS = YS + L(i)\*sin(atan(a) + f(i)) + LP(i)\*sin(atan(a));  b = b + sign(YCRIT(i) - YO(j(i)))\*LM(i)/cos(atan(a));    % Maybe I need this b  % b = b - sign(YCRIT(i)+Y(j(i)))\*LM(i)/cos(a);    % Plot the end points of each loop  % By bringing the tangential to each circle from these points the maximum allowable turn angle will be determined  plot(XS,YS,'+k')    end    % For the last segment of the trajectory update position vector  X\_pos = [X\_pos, XT];  Y\_pos = [Y\_pos, YT];    % Plot final part of the path with magenta dash-dot line  plot(X\_pos, Y\_pos, '-.om', 'LineWidth', 1.5)    % Display straight-line length  disp(['Straight-line length: ' num2str(Ls)])    % Total number of line segments  S = size(X\_pos,2)-1;    % Display the number of line segments  disp(['Number of line-segments: ' num2str(S)])    % Find the length of each segment  for z=1:S  L(z) = sqrt((X\_pos(z + 1) - X\_pos(z))^2 + (Y\_pos(z + 1) - Y\_pos(z))^2);  end    % Find the total length  TL = sum(L (1:S));  disp(['Trajectory length: ' num2str(TL)]) |

# Annex E: MATLAB code for the Segment Algorithm

Following the MATLAB code for the path planning algorithm based on the Segment Method is presented. The code has been written using the R2017b version of MATLAB and its main objective is to determine a safe path, in a multiple obstacles domain, guiding the vessel to its destination.

|  |
| --- |
| **Segment.m** |
| %% - Autonomous USV Path Planning Algorithm - %%  % This code uses an algorithm based on the projection collision avoidance %  % method to find a path from the start point to the target point %  % Every obstacle is avoided using all the permutations of obstacle array %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    % Clean the workspace and close the open figures  clear  clc  close all    %% Parameters - Setting up the problem    % Start point S (XS, YS)  XS = 5;  YS = 2;    % Target point T (XT, YT)  XT = 36;  YT = 30;    % Obstacle representation: circle with centre at (XO, YO) and radius RO  XO = [10, 19, 29];  YO = [9, 17, 24];  RO = [4, 6, 3];    % Safety radius RB  % Was set equal to the radius of the vessel region (RV), for the simulations  RB = 0.571;    % Number of obstacles N  N = length(XO);    % Calculate the length of the straight line from S to T  Lstr = sqrt((XT - XS)^2 + (YT - YS)^2);    allCombos = perms(1:N);  disp(allCombos);    % Check if boat is already inside obstacle region  err = false;  for io = 1:N  % Start point inside obstacle region  check1 = (sqrt((XS - XO(io))^2 + (YS - YO(io))^2) < (RO(io) + RB));    % Target point inside obstacle region  check2 = (sqrt((XT - XO(io))^2 + (YT - YO(io))^2) < (RO(io) + RB));    if (check1 || check2)  fprintf("No solution. Start/Target point(s) inside obstacle region.\n");  err = true;  break  end  end    if err  return  end    % Exit this loop if a solution is found or if we have tested all combos  % Extra stop criterion should be added  i\_attempt = 1;  solution\_found = false;  previousTrajectoryLength = 0;    while (i\_attempt <= length(allCombos))    disp(i\_attempt)  figure(i\_attempt);    sort\_idx = allCombos(i\_attempt, :);  disp(sort\_idx);    XO = XO(sort\_idx);  YO = YO(sort\_idx);  RO = RO(sort\_idx);    clf    %% Plotting basic features  [RD] = plot\_obstacles(XS, YS, XT, YT, XO, YO, RO, RB, N);    %% Core calculations    % Initial path from S to T  Px = [XS, XT];  Py = [YS, YT];    % Loop while there are line segments to be resolved  % Initialize loop  k = 1; % Number of line segments  K = 1; % Number of lines    while k <= K  for n = 1:N % Loop for number of obstacles  beginAgain = false;  % Check if manoeuvre is needed  % Inputs are, path segment start P(k) & end P(k+1), obstacle and boat settings  % Outputs are the extra points due to the manoeuvre or empty if manoeuvre not needed  [Xa, Ya, Xb, Yb, err] = vessel\_find\_path(Px(k), Py(k), Px(k+1), Py(k+1), XO(n), YO(n), RO(n), RB, XO, YO, RO);    if (err)  warning('Route point(s) inside obstacle region.')  break  end    % Case that manoeuvre is needed  if ~isempty(Xa)    % Add extra points in path due to manoeuvre  Px = [Px(1:k), Xa, Xb, Px(k+1:end)];  Py = [Py(1:k), Ya, Yb, Py(k+1:end)];    beginAgain = true;  % Add number of new segments in the total counter  K = K + 2;  disp('K');  disp(K);  break  end  end    % Move to next segment  if (beginAgain)  k = 1;  else  k = k + 1;  end    % Check if destination was reached or path too complex  if (k > K)  break  end  end    %% Check if any point in path is within an obstacle area  current\_solution\_found = true;  err = false;    % Loop for all points (except starting, ending where there is nothing to do)  for k = (2:length(Px) - 1)    % Loop for all obstacles  for m = (1:length(XO))    % Check if within radius  if (sqrt((Px(k) - XO(m))^2 + (Py(k) - YO(m))^2) < RD(m))    warning('Solution invalid! Trying again.')  current\_solution\_found = false;    % Increase counter to make sure we have not exhausted all permutations  i\_attempt = i\_attempt + 1;    % Indicate an error to exit second loop  err = true;  drawnow  break  end  end    % If this is an error, exit this loop as well  if err  break  end  end    if current\_solution\_found  %if at least one solution found, solution found  solution\_found = true;  fprintf("Solution found. %d line segments.\n", K);    % Find the length of each segment  for is=1:K  Ls(is) = sqrt((Px(is + 1) - Px(is))^2 + (Py(is + 1) - Py(is))^2);  end    % Find the total length  TL = sum(Ls (1:K));    if (previousTrajectoryLength == 0 || (TL < previousTrajectoryLength))  prev\_i\_attempt = i\_attempt;  prev\_sort\_idx = sort\_idx;  previousTrajectoryLength = TL;  K\_min = K;  Px\_min = Px;  Py\_min = Py;  end    disp(['Trajectory length: ' num2str(TL)])  plot(Px, Py, '-.oy', 'LineWidth', 1.5)  % Increase counter to check the next combination  % and continue to "while loop"  i\_attempt = i\_attempt + 1;  continue  end    end    if solution\_found  disp('Finally');  disp(prev\_i\_attempt);  K\_simple = K\_min;  figure(length(allCombos) + 1);    %% Plotting basic features  [RD] = plot\_obstacles(XS, YS, XT, YT, XO, YO, RO, RB, N);  plot(Px\_min, Py\_min, '-.og', 'LineWidth', 1.5)    figure(length(allCombos) + 2);  %% Plotting basic features  [RD] = plot\_obstacles(XS, YS, XT, YT, XO, YO, RO, RB, N);    % check if any point is not needed  atLeastOneSimplification = true;  while atLeastOneSimplification  disp('Simplification feasible.');  atLeastOneSimplification = false;  for i = 1:length(Px\_min) -2  disp('i');  disp(i)    for obstacle\_no = 1:N  [noIntersection, ferr] = check\_intersection(Px\_min(i), Py\_min(i), Px\_min(i+2), Py\_min(i+2), XO(obstacle\_no), YO(obstacle\_no), RO(obstacle\_no), RB);    % if there is an intersection, exit for loop  if ~noIntersection  break;  end  end    if noIntersection  disp('No Intersection');  atLeastOneSimplification = true;  Px\_min(i+1) = [];  Py\_min(i+1) = [];  K\_simple = K\_simple -1;  break;  end  end  end    % Find the length of each segment  disp('K')  disp(K\_simple)  for is=1:K\_simple  Ls(is) = sqrt((Px\_min(is + 1) - Px\_min(is))^2 + (Py\_min(is + 1) - Py\_min(is))^2);  end    % Find the total length  TL = sum(Ls (1:K\_simple));    disp(['Straight-line Length: ' num2str(Lstr)])  fprintf('\n')  disp(['Minimum Trajectory No. of segments: ' num2str(K\_min)])  disp(['Minimum Trajectory Length: ' num2str(previousTrajectoryLength)])  fprintf('\n')    disp(['Simplified Trajectory No. of segments: ' num2str(K\_simple)])  disp(['Simplified Trajectory Length: ' num2str(TL)])    % Plot the simplified trajectory  plot(Px\_min, Py\_min, '-.om', 'LineWidth', 1.5)    else  fprintf("Error, algorithm did not converge.\n");  end    drawnow |

|  |
| --- |
| **plot\_obstacles.m** |
| %% - Function to plot obstacles - %%  % This code introduces a function to plot the obstacle array %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    function [RD] = plot\_obstacles(XS, YS, XT, YT, XO, YO, RO, RB, N)  %% Plotting basic features    % Add description to data points S and T  txt1 = ' Start point';  text(XS,YS,txt1,'VerticalAlignment','bottom')  hold on; box on; xlabel('X (m)'); ylabel('Y (m)');    txt2 = ' Target point';  text(XT,YT,txt2,'VerticalAlignment','bottom')    % Plot straight line from S to T with black dotted line  plot([XS XT], [YS YT], ':xk')    % plot(XS, YS, ':xk')  % plot(XT, YT, ':xk')    % Plot all obstacles  RD = zeros(1, N);  for i = 1:N  % Find the determinant radius RD for the obstacle region  RD(i) = RO(i) + RB;    % Plot centre of obstacle circle  plot(XO(i), YO(i), '.b')    % Plot circle (X-XO)^2+(Y-YO)^2=RD^2 with XO, YO and RD red --- line  plot\_circle(XO(i), YO(i), RD(i), 'r');    % Plot circle (X-XO)^2+(Y-YO)^2=RO^2 with XO, YO and RO blue --- line  plot\_circle(XO(i), YO(i), RO(i), 'b');    if (i == 1)  axis equal  end  end  end |

|  |
| --- |
| **vessel\_fun.m** |
| %% - Function to determine the manoeuvre points - %%  % %  % input: the start S (XS, YS) and target T (XT, YT) points of each segment, %  % the radius of the obstacle located between S and T, %  % the safety radius RB %  % output: the manoeuvre points (Xa, Ya) and (Xb, Yb) created in order %  % to avoid the obstacle %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    function [Xa, Ya, Xb, Yb, err] = vessel\_fun(XS, YS, XT, YT, XO, YO, RO, RB, direction)    % Create the manoeuvre points arrays  Xa = [];  Ya = [];  Xb = [];  Yb = [];    err = false;    % Find the straight-line equation (Y = a\*X+b) connecting the start and  % target points  a = (YS - YT)/(XS - XT);  b = (XS\*YT - XT\*YS)/(XS - XT);    % Find the determinant radius RD  RD = RO + RB;    % Check if path inside obstacle region  if (sqrt((XS - XO)^2 + (YS - YO)^2) < RD) || (sqrt((XT - XO)^2 + (YT - YO)^2) < RD)  err = true;  return  end    % Find the interception point(s) between the line and the circle (equation: (X-XO)^2+(Y-YO)^2 = RD^2)  % Need to solve: (a^2 + 1)\*X^2 + 2\*(a\*b - a\*YO - XO)\*X + (YO^2 - RD^2 + XO^2 - 2\*b\*YO + b^2) = 0  % Substitutions in two quadratic equation coefficients  A = (a^2 + 1);  B = 2\*(a\*b - a\*YO - XO);  C = (YO^2 - RD^2 + XO^2 - 2\*b\*YO + b^2);    % Determinant calculation  D = B^2 - 4\*A\*C;    %% Finding the relative position between the straight line and the obstacle  check\_1 = ((XS == XT) && ((XS <= XO - RD) || (XS >= XO + RD))); % Route parallel to Y-axis and no intersection points  check\_2 = ((YS == YT) && ((YS <= YO - RD) || (YS >= YO + RD))); % Route parallel to X-axis and no intersection points  check\_3 = (D <= 0); % 0 or 1 solutions, i.e. none or one intersection point    % Two solutions (intersection points), with coordinates (X1, Y1) & (X2, Y2)  if ~(check\_1 || check\_2 || check\_3)    if (XS == XT) % Route parallel to Y-axis  X1 = XS;  X2 = XS;  Y1 = YO - sqrt (RD^2 - (XS-XO)^2);  Y2 = YO + sqrt (RD^2 - (XS-XO)^2);    elseif (YS == YT) % Route parallel to X-axis  X1 = XO - sqrt (RD^2 - (YS-YO)^2);  X2 = XO + sqrt (RD^2 - (YS-YO)^2);  Y1 = YS;  Y2 = YS;    else % Route with random orientation  X1 = (-B + sqrt(B^2 - 4\*A\*C))/(2\*A);  X2 = (-B - sqrt(B^2 - 4\*A\*C))/(2\*A);  Y1 = a\*X1 + b;  Y2 = a\*X2 + b;  end    % Check if the intersection points belong to the line segment from S to T  NoIntersection = true;  if (XT > XS)  if (XS < X1 && X2 < XT)  NoIntersection = false;  end  elseif (XT == XS)  if (YT > YS)  if (YS < Y1 && Y2 < YT)  NoIntersection = false;  end  else % YT < YS  if (YT < Y1 && Y2 < YS)  NoIntersection = false;  end  end  else % XT < XS  if (XT < X1 && X2 < XS)  NoIntersection = false;  end  end    %% Way of updating the position vector  if ~(NoIntersection)  % Determine the direction to turn    % Plot the intersection points  plot(X1, Y1, 'xk')  plot(X2, Y2, 'xk')    if (XS == XT) % Extra criterion because in this case YCRIT=YO and cannot determine the direction to turn  if (XS >= XO)  if (YS > YO) % First quadrant  CCW = true;  else % YS < YO Fourth quadrant  CCW = false;  end  else % XS < XO  if (YS > YO) % Second quadrant  CCW = false;  else % YS < YO Third quadrant  CCW = true;  end  end  XC = XS;  YC = YO;    else % XS not equal with XT  if (YS == YT)  YCRIT = YS;  XC = XO;  YC = YS;  else % Calculate the YCRIT  YCRIT = a\*XO + b;  % Find the line equation (Y = a2\*X + b2) which is lateral to the  % initial one and is crossing from the centre of the obstacle  a2 = -1/a;  b2 = (a\*YO + XO)/a;  % Find the cross point of the two lines (XC, YC)  % Solve the system (Y = a\*X + b) and (Y = a2\*X + b2)  XC = (b2 - b)/(a - a2);  YC = a\*XC + b;  end    % If YCRIT>=YO then turn CCW angle f1 and CW f2  if (YCRIT >= YO)  CCW = true;  else % If YCRIT<YO then turn CW angle f1 and CCW f2  CCW = false;  end  end    % Calculate the distance from the centre of the obstacle to the cross point  LR = sqrt((XC - XO)^2 + (YC - YO)^2);    % Calculate the distance from the cross point to the circumference  if direction  LM = RD - LR;  else  LM = RD + LR;  end    % Calculate the distances from start point to cross point  tmp0 = sqrt((X1 - XS)^2 + (Y1 - YS)^2);  tmp1 = sqrt((X2 - XS)^2 + (Y2 - YS)^2);    % SET the smaller D1 and the bigger D2  if (tmp0 < tmp1)  D1 = tmp0;  else  D1 = tmp1;  end    % Calculate the distance between the cross points  LP = sqrt((X1 - X2)^2 + (Y1 - Y2)^2);    % Calculate the length of hypotenuse in the start triangle  L1 = sqrt(LM^2 + D1^2);    % Calculate the turn angle  % If YCRIT>=YO then turn CW/CCW angle f1  if (CCW == true)  if direction  f = atan(LM/D1);  f1 = f + 2\*pi/3600; % Extra angle added to compensate for numerical inaccuracy  else  f = -atan(LM/D1);  f1 = f - 2\*pi/3600; % Extra angle added to compensate for numerical inaccuracy  end  else % If YCRIT<YO then turn CW angle f1  if direction  f = -atan(LM/D1);  f1 = f - 2\*pi/3600; % Extra angle added to compensate for numerical inaccuracy  else  f = atan(LM/D1);  f1 = f + 2\*pi/3600; % Extra angle added to compensate for numerical inaccuracy  end  end  % The extra angle is equal to 0.1 degrees and is going to play an  % insignificant role to the trajectory length while will solve the  % rounding decimals problem    %% Finding the manoeuvre points  if (XT > XS) % Forward motion  % First manoeuvre point  Xa = XS + L1\*cos(atan(a) + f1);  Ya = YS + L1\*sin(atan(a) + f1);    % Second manoeuvre point  Xb = Xa + LP\*cos(atan(a));  Yb = Ya + LP\*sin(atan(a));    elseif (XS == XT) % Parallel to Y-axis motion  % First manoeuvre point  Xa = XS + sign(YS-YO)\*L1\*cos(pi/2 - f1);  Ya = YS - sign(YS-YO)\*L1\*sin(pi/2 - f1);    % Second manoeuvre point  Xb = Xa - sign(YS - YO)\*LP\*cos(pi/2);  Yb = Ya - sign(YS - YO)\*LP\*sin(pi/2);    else % Backward motion  % First manoeuvre point  Xa = XS - L1\*cos(atan(a) - f1);  Ya = YS - L1\*sin(atan(a) - f1);    % Second manoeuvre point  Xb = Xa - LP\*cos(atan(a));  Yb = Ya - LP\*sin(atan(a));  end  end  end  end |

|  |
| --- |
| **vessel\_find\_path.m** |
| %% - Function to determine the direction to avoid an obstacle - %%  % This code introduces a function to execute the vessel\_fun code %  % with two different directions %  % initially finding the short path and if it is not possible to %  % execute the vessel\_fun code again finding the long path %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    function [Xa, Ya, Xb, Yb, err] = vessel\_find\_path(XS, YS, XT, YT, XO, YO, RO, RB, XO\_ARR, YO\_ARR, RO\_ARR)  %% Core calculations    % Try the fast route around the obstacle  [Xa, Ya, Xb, Yb, err] = vessel\_fun(XS, YS, XT, YT, XO, YO, RO, RB, true);    if ~isempty(Xa)  for j = 1:length(XO\_ARR)    % Start point within obstacle  check1 = (sqrt((Xa - XO\_ARR(j))^2 + (Ya - YO\_ARR(j))^2) < (RO\_ARR(j) + RB));    % End point within obstacle  check2 = (sqrt((Xb - XO\_ARR(j))^2 + (Yb - YO\_ARR(j))^2) < (RO\_ARR(j) + RB));    % Take the slow route around the obstacle  if (check1 || check2)  [Xa, Ya, Xb, Yb, err] = vessel\_fun(XS, YS, XT, YT, XO, YO, RO, RB, false);  end  end  end    end |

|  |
| --- |
| **check\_intersection.m** |
| %% - Function for path simplification - %%  % This code introduces a function to check if the line segment %  % connecting the i and i+2 manoeuvre points intersects %  % with any of the given obstacles %  % %  % If no intersection exists the i+1 point is skipped %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    function [NoIntersection, err] = check\_intersection(XS, YS, XT, YT, XO, YO, RO, RB)    err = false;  NoIntersection = true;    % Find the straight-line equation (Y = a\*X+b) connecting the start and  % target points  a = (YS - YT)/(XS - XT);  b = (XS\*YT - XT\*YS)/(XS - XT);    % Find the determinant radius RD  RD = RO + RB;    % Check if path inside obstacle region  if (sqrt((XS - XO)^2 + (YS - YO)^2) < RD) || (sqrt((XT - XO)^2 + (YT - YO)^2) < RD)  err = true;  NoIntersection = false;  return  end    % Find the interception point(s) between the line and the circle (equation: (X-XO)^2+(Y-YO)^2 = RD^2)  % Need to solve: (a^2 + 1)\*X^2 + 2\*(a\*b - a\*YO - XO)\*X + (YO^2 - RD^2 + XO^2 - 2\*b\*YO + b^2) = 0  % Substitutions in two quadratic equation coefficients  A = (a^2 + 1);  B = 2\*(a\*b - a\*YO - XO);  C = (YO^2 - RD^2 + XO^2 - 2\*b\*YO + b^2);    % Determinant calculation  D = B^2 - 4\*A\*C;    %% Finding the relative position between the straight line and the obstacle  check\_1 = ((XS == XT) && ((XS <= XO - RD) || (XS >= XO + RD))); % Route parallel to Y-axis and no intersection points  check\_2 = ((YS == YT) && ((YS <= YO - RD) || (YS >= YO + RD))); % Route parallel to X-axis and no intersection points  check\_3 = (D < 0); % 0 or 1 solution, i.e. none or one intersection point    % Two solutions (intersection points), with coordinates (X1, Y1) & (X2, Y2)  if ~(check\_1 || check\_2 || check\_3)    if (XS == XT) % Route parallel to Y-axis  X1 = XS;  X2 = XS;  Y1 = YO - sqrt (RD^2 - (XS-XO)^2);  Y2 = YO + sqrt (RD^2 - (XS-XO)^2);    elseif (YS == YT) % Route parallel to X-axis  X1 = XO - sqrt (RD^2 - (YS-YO)^2);  X2 = XO + sqrt (RD^2 - (YS-YO)^2);  Y1 = YS;  Y2 = YS;    else % Route with random orientation  X1 = (-B + sqrt(B^2 - 4\*A\*C))/(2\*A);  X2 = (-B - sqrt(B^2 - 4\*A\*C))/(2\*A);  Y1 = a\*X1 + b;  Y2 = a\*X2 + b;  end    % Check if the intersection points belong to the line segment from S to T  NoIntersection = true;  if (XT > XS)  if (XS < X1 && X2 < XT)  NoIntersection = false;  end  elseif (XT == XS)  if (YT > YS)  if (YS < Y1 && Y2 < YT)  NoIntersection = false;  end  else % YT < YS  if (YT < Y1 && Y2 < YS)  NoIntersection = false;  end  end  else % XT < XS  if (XT < X1 && X2 < XS)  NoIntersection = false;  end  end  end  end |

# Annex F: MATLAB code for the Segment Algorithm using Virtual Waypoints

Following the MATLAB code for the path planning algorithm based on the Segment Method enhanced with the virtual waypoint functionality is presented. The code has been written using the R2017b version of MATLAB and its main objective is to determine a safe path, in a multiple obstacles domain, guiding the vessel to its destination.

It is an enhanced version of the Segment Algorithm introducing virtual waypoints to overcome the limitations inherent in the collision avoidance algorithm.

|  |
| --- |
| **Segment\_Virtual.m** |
| %% - Autonomous USV Path Planning Algorithm with virtual waypoint - %%  % This code uses an algorithm based on the projection collision avoidance %  % method to find a path from the start point to the target point %  % Adopts virtual waypoints %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    % Clean the workspace and close the open figures  clear  clc  close all    %% Parameters - Setting up the problem    % Start point S (XS, YS)  XS = -5;  YS = 6;    % Target point T (XT, YT)  XT = 60;  YT = 26;    % Obstacle representation: circle with centre at (XO, YO) and radius RO  XO = [10, 25, 25, 40];  YO = [20, 5, 35, 20];  RO = [9.429, 9.429, 9.429, 9.429];    % Safety radius RB  % Was set equal to the radius of the vessel region (RV), for the simulations  RB = 0.571;    % Number of obstacles N  N = length(XO);  min\_x\_array = zeros(1,N);  max\_x\_array = zeros(1,N);  min\_y\_array = zeros(1,N);  max\_y\_array = zeros(1,N);  solution\_found = false;    [solution\_found, err] = find\_route(XS, YS, XT, YT, XO, YO, RO, RB, N);    if (solution\_found)  return;  else  i=1;    % get the minimum x and y without radius  % get the maximum x and y with radius  for n = 1:length(XO)  min\_x\_array(n) = XO(n) - RO(n);  max\_x\_array(n) = XO(n) + RO(n);  min\_y\_array(n) = YO(n) - RO(n);  max\_y\_array(n) = YO(n) + RO(n);  end    min\_x = min(min\_x\_array);  max\_x = max(max\_x\_array);  min\_y = min(min\_y\_array);  max\_y = max(max\_y\_array);    % find possible intermediate points  x\_intermediate = [min\_x, max\_x, XS, XS, min\_x, max\_x];  y\_intermediate = [YS, YS, min\_y, max\_y, min\_y, max\_y];    while ~solution\_found && i <= length(x\_intermediate)  disp('Another try:');  disp('Intermediate point');  disp(x\_intermediate(i));  disp(y\_intermediate(i));  Px\_min = [];  Py\_min = [];    % and try to make two routes  [solution\_found, err, Px\_min\_1, Py\_min\_1] = find\_route(XS, YS, x\_intermediate(i), y\_intermediate(i), XO, YO, RO, RB, N);    if (solution\_found)  [solution\_found, err, Px\_min\_2, Py\_min\_2] = find\_route(x\_intermediate(i), y\_intermediate(i), XT, YT, XO, YO, RO, RB, N);    if (solution\_found)  Px\_min = horzcat(Px\_min\_1, Px\_min\_2);  Py\_min = horzcat(Py\_min\_1, Py\_min\_2);  end  end    i=i+1;  end    if (solution\_found)  fprintf('\n')  disp(['Number of virtual waypoints examined: ' num2str(i - 1)])  fprintf('Solution found');  figure(1000000);  [RD] = plot\_obstacles(XS, YS, XT, YT, XO, YO, RO, RB, N);  plot(Px\_min, Py\_min, '-.om', 'LineWidth', 1.5)  end  end |

|  |
| --- |
| **find\_route.m** |
| %% - Function to produce a path - %%  % This code is the main path planning method but is given %  % in a function form to be possible to be used in %  % the virtual waypoint code %  % %  % Written by Dimitrios Stergianelis on August 2018 %  % %  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%    function [solution\_found, err, Px\_min, Py\_min] = find\_route(XS, YS, XT, YT, XO, YO, RO, RB, N)  test = randi(1000);  figure(test);  solution\_found = false;  Px\_min = [XS, XT];  Py\_min = [YS, YT];    % Calculate the length of the straight line from S to T  Lstr = sqrt((XT - XS)^2 + (YT - YS)^2);    allCombos = perms(1:N);  % disp(allCombos);    % Check if boat is already inside obstacle region  err = false;  for io = 1:N  % Start point inside obstacle region  check1 = (sqrt((XS - XO(io))^2 + (YS - YO(io))^2) < (RO(io) + RB));    % Target point inside obstacle region  check2 = (sqrt((XT - XO(io))^2 + (YT - YO(io))^2) < (RO(io) + RB));    if (check1 || check2)  fprintf("No solution. Start/Target point(s) inside obstacle region.\n");  err = true;  break  end  end    if err  return  end    % Exit this loop if a solution is found or if we have test all combos  % Extra stop criterion should be added  i\_attempt = 1;    previousTrajectoryLength = 0;    while (i\_attempt <= length(allCombos))    % disp(i\_attempt)  % figure(i\_attempt);    sort\_idx = allCombos(i\_attempt, :);  % disp(sort\_idx);    XO = XO(sort\_idx);  YO = YO(sort\_idx);  RO = RO(sort\_idx);    clf    %% Plotting basic features  [RD] = plot\_obstacles(XS, YS, XT, YT, XO, YO, RO, RB, N);    %% Core calculations    % Initial path from S to T  Px = [XS, XT];  Py = [YS, YT];    % Loop while there are line segments to be resolved  % Initialize loop  k = 1; % Number of line segments  K = 1; % Number of lines    while k <= K  for n = 1:N % Loop for number of obstacles  beginAgain = false;  % Check if manoeuvre is needed  % Inputs are, path segment start P(k) & end P(k+1), obstacle and boat settings  % Outputs are the extra points due to the manoeuvre or empty if manoeuvre not needed  [Xa, Ya, Xb, Yb, err] = vessel\_find\_path(Px(k), Py(k), Px(k+1), Py(k+1), XO(n), YO(n), RO(n), RB, XO, YO, RO);    if (err)  % warning('Route point(s) inside obstacle region.')  break  end    % Case that manoeuver is needed  if ~isempty(Xa)    % Add extra points in path due to manoeuvre  Px = [Px(1:k), Xa, Xb, Px(k+1:end)];  Py = [Py(1:k), Ya, Yb, Py(k+1:end)];    beginAgain = true;  % Add number of new segments in the total counter  K = K + 2;  % disp('K');  % disp(K);  break  end  end    % Move to next segment  if (beginAgain)  k = 1;  else  k = k + 1;  end    % Check if destination was reached or path too complex  if ((k > K) || (K > 10\*N))  break  end  end    %% Check if any point in path is within an obstacle area  current\_solution\_found = true;  err = false;    % Loop for all points (except starting, ending where there is nothing to do)  for k = (2:length(Px) - 1)    % Loop for all obstacles  for m = (1:length(XO))    % Check if within radius  if (sqrt((Px(k) - XO(m))^2 + (Py(k) - YO(m))^2) < RD(m))    % warning('Solution invalid! Trying again.');  current\_solution\_found = false;    % Increase counter to make sure we have not exhausted all permutations  i\_attempt = i\_attempt + 1;    % Indicate an error to exit second loop  err = true;  drawnow  break  end  end    % If this is an error, exit this loop as well  if err  break  end  end    if current\_solution\_found  %if at least one solution found, solution found  solution\_found = true;  % fprintf("Solution found. %d line segments.\n", K);    % Find the length of each segment  for is=1:K  Ls(is) = sqrt((Px(is + 1) - Px(is))^2 + (Py(is + 1) - Py(is))^2);  end    % Find the total length  TL = sum(Ls (1:K));    if (previousTrajectoryLength == 0 || (TL < previousTrajectoryLength))  prev\_i\_attempt = i\_attempt;  prev\_sort\_idx = sort\_idx;  previousTrajectoryLength = TL;  K\_min = K;  Px\_min = Px;  Py\_min = Py;  end    % disp(['Trajectory length: ' num2str(TL)]);  % plot(Px, Py, '-.oy', 'LineWidth', 1.5)  % Increase counter to check the next combination and continue to "while loop"  i\_attempt = i\_attempt + 1;  continue  end    end    if solution\_found  disp('Finally');  K\_simple = K\_min;    % figure(length(allCombos) + 1);  % test = randi(1000);  % figure(test);  [RD] = plot\_obstacles(XS, YS, XT, YT, XO, YO, RO, RB, N);  plot(Px\_min, Py\_min, '-.ob', 'LineWidth', 1.5)    % figure(length(allCombos) + 2);  test = randi(1000);  figure(test);  [RD] = plot\_obstacles(XS, YS, XT, YT, XO, YO, RO, RB, N);    % check if any point is not needed  atLeastOneSimplification = true;  while atLeastOneSimplification  % disp('Simplification feasible.');  atLeastOneSimplification = false;  for i = 1:length(Px\_min) -2  for obstacle\_no = 1:N  [noIntersection, ferr] = check\_intersection(Px\_min(i), Py\_min(i), Px\_min(i+2), Py\_min(i+2), XO(obstacle\_no), YO(obstacle\_no), RO(obstacle\_no), RB);    % if there is an intersection, exit for loop  if ~noIntersection  break;  end  end    if noIntersection  atLeastOneSimplification = true;  Px\_min(i+1) = [];  Py\_min(i+1) = [];  K\_simple = K\_simple -1;  break;  end  end  end    for is=1:K\_simple  Ls(is) = sqrt((Px\_min(is + 1) - Px\_min(is))^2 + (Py\_min(is + 1) - Py\_min(is))^2);  end    % Find the total length  TL = sum(Ls (1:K\_simple));    disp(['Straight-line Length: ' num2str(Lstr)])  fprintf('\n')  disp(['Minimum Trajectory No. of segments: ' num2str(K\_min)])  disp(['Minimum Trajectory Length: ' num2str(previousTrajectoryLength)])  fprintf('\n')    disp(['Simplified Trajectory No. of segments: ' num2str(K\_simple)])  disp(['Simplified Trajectory Length: ' num2str(TL)])    % Plot the simplified trajectory  plot(Px\_min, Py\_min, '-.om', 'LineWidth', 1.5)    else  fprintf("Error, algorithm did not converge.\n");  end    drawnow  end |