UNIVERSITATEA TITU MAIORESCU

FACULTATEA: INFORMATICĂ
DEPARTAMENT: INFORMATICĂ
Programa de studii: INFORMATICĂ
DISCIPLINA: INTELIGENȚĂ ARTIFICIALĂ

IA – *Testul de evaluare nr.* 21

Navigație COLREG – UMV

Grupa	Numele și prenumele	Semnătură student	Notă evaluare

Data: ____/____/_____

CS-I dr.ing.
Lucian Ștefăniță GRIGORE

Conf.dr.ing.
Iustin PRIESCU

Ș.L.dr.ing.
Dan-Laurențiu GRECU

Cuprins

1. INTRODUCERE

Această lucrare evidențiază problematica planificării mișcării unui robot maritim, într-un mediu nestructurat, în vederea evitării coliziunii. Algoritmul de planificare a mișcării pentru vehiculele de suprafață fără pilot (USV) abordează evitarea obstacolelor staționare și în mișcare. Algoritmul ține cont de Reglementările Internaționale pentru Prevenirea Coliziunilor pe Mare (cunoscute sub numele de COLREGS). Pentru determinarea dimensiunilor obstacolelor și a constrângerilor cinematice și dinamice ale robotului, vom aplica principiul de Spațiului de Schimbare a Vitezei (VCS), folosindu-ne de modificările vitezei și direcției UVS. Se va ține cont de regulile de manevrare COLREG: traversare, depășire, orientare. De asemenea pentru a lua în calcul și viteza obstacolelor (nestaționare) se va aplica principiul Vitezei Obstacolelor (VO), care generează un con al spațiului vitezelor.

USV-urile sunt vehicule maritime autonome, al căror sistem de comandă și control poate fi preluat de către un operator uman ori de câte ori este nevoie. Această precizare vine în contextul dat de faptul că utilizarea acestora are loc într-un mediu maritim specific unei zone portuare. Așa cum s-a precizat, mediul de lucru este nestructurat, deoarece introduce perturbări, cum ar fi: configurația suprafeței apei la interfața dintre cele două medii nestructurate apă și aer, curenții subacvatici, curenții de aer, dislocuirea volumelor de apă în apropierea coastelor, epave aflate pe fundul danelor portuare, etc. [1].

Dezvoltarea de USV-uri se înscrie în politica actuală datorată nevoii crescute de monitorizare și securizare a mediilor portuare [2]. Deși navele maritime utilizează Sistemul de Identificare Automată (AIS - Automatic Identification System), navele mici (USV) pot să nu îl folosească, întrucât acestea își pot schimba direcția și viteza de deplasare în funcție de datele pe care le primesc de la senzori. Aceste sisteme autonome trebuie să poată dezvolta căi de deplasare optimizate și acțiuni adaptive, care să le permită evitarea obstacolelor și anularea perturbațiilor.

Autoritățile maritime pentru a putea conecta navele și a planifica căile de deplasare trebuie să definească diferitele niveluri navigație, acțiuni operaționale, de toleranță și redundanță privind navigarea și monitorizarea [3].

Pentru realizarea unui planificator deliberativ de traseu în timp real, astfel încât, navele autonome să poată ocoli obstacolele se folosesc regulile de deplasare prevăzute în COLREG [4,5]. Această standardizare stă la baza Modelelor Predictive de Control (MPC), care pot calcula numeric o traiectorie optimă pe un orizont de mișcare finit pe baza previziunilor mișcării obstacolelor. Pentru ca algoritmul să permită ocolirea obstacolelor indiferent dacă navele cu care se intersectează respectă cele trei reguli COLREG (Fig. 1-1).

Astfel, lucrarea prezintă un algoritm bazat pe VO [6], care generează un obstacol în formă de con în spațiul de viteză. Viteza specifică a obstacolelor și pe ce parte a robotului se află respectivul obstacol pe timpul manevrei de evitare, sunt codificate în spațiul de viteză într-un mod natural. Abordarea VO a navigației sigure servește ca un planificator local în algoritmul de planificare a mișcării.

Există mai multe modele analitico-numerice care descriu VO pentru a evita obstacolele dar și pentru a genera un set suplimentar de constrângeri în spațiul de viteză VCS pentru situațiile descrise de COLREG.

În [7-9] se prezintă diverse modelări de identificare și control în care obiectivul este de a manevra o navă pe traseele dorite la viteze diferite. Ecuațiile descriu mișcarea hidrodinamică a corpului rigid (6 DOF - degrees-of-freedom): deplasare, balansare, ridicare, translație, rostogolire, înclinare și rotire (Surge, Sway, Heave, Roll, Pitch and Yaw).

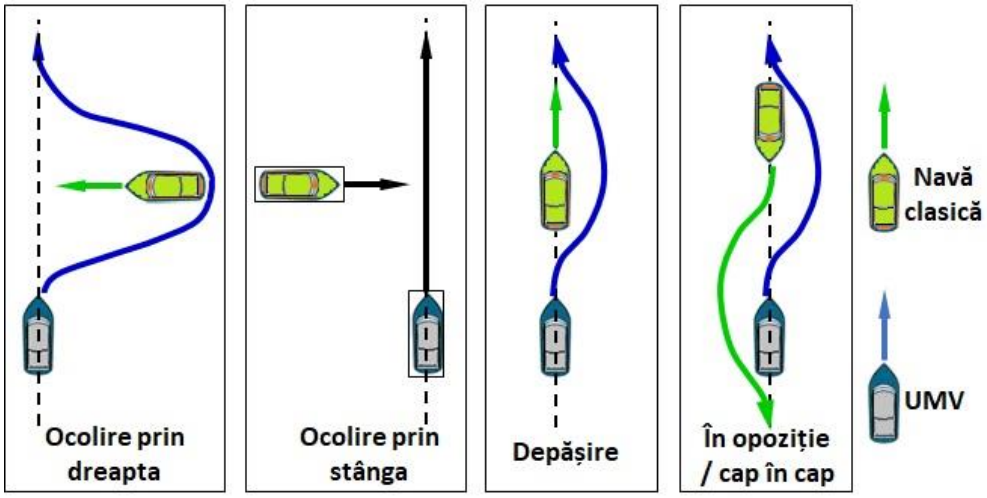

Fig. 1-1 Evitarea coliziunii ambarcațiunilor maritime conform COLREG [6]

2. ABORDAREA VO VELOCITY OBSTACLE

Prin abordarea VO înțelegem că robotul pentru a putea evita obstacolele va genera un spațiu prin care definește obiectele respective. VO generează un obstacol în formă de con, ale cărui caracteristici cinematice se desfășoară în spațiul vitezei. Această abordare urmărește ca robotul să nu intre în coliziune, atât timp cât vectorul vitezei se află în afara VO.

Evitarea unui potențial pericol se bazează pe anticiparea poziției obstacolului cu care ar putea intra în coliziune și bineînțeles cu predicția viitoarelor poziții. Pentru fiecare predicție se reiau calculele privind probabilitatea ca cele două obiecte să intre în coliziune. Astfel, calculele sunt dependente de anumite traiectorii arbitrare. VO face predicții liniare , iar verificarea coliziunii se face pentru traiectoriile arbitrare viitoare.

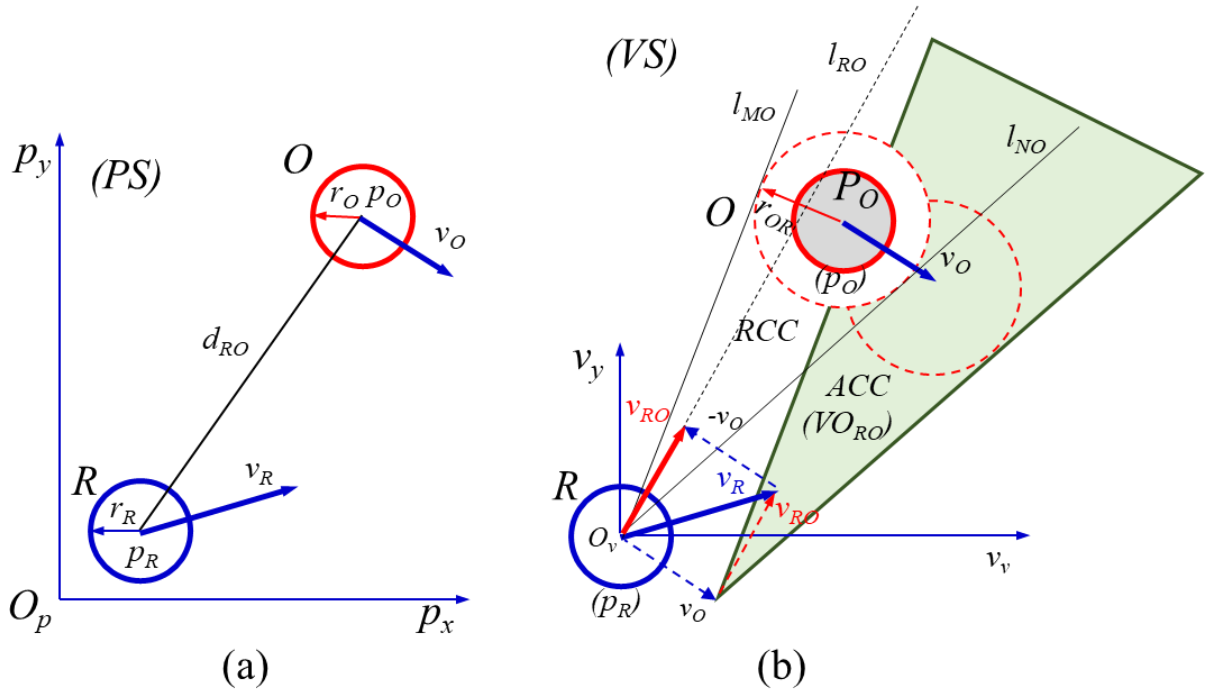

Fig. 2-1 Reprezentarea conului vitezelor

R – robot; p_R - poziția; v_R - viteza; r_R – raza lui R; O - obstacol dinamic; p_O - poziția; v_O - viteza; r_O – raza lui O; d_{RO} – distanța dintre centrele celor două obiecte R și O;

PO: $r_{OR} = r_O + r_R$; l_{MO} și l_{NO} – raza de tangență stânga și raza de tangență dreapta față de PO având ca punct de plecare poziția p_R; l_{RO} – raza ce pornește din poziția p_R și reprezintă direcția vitezei v_{RO}.

Din punct de vedere al planificării mișcării avem două probleme: planificarea traseului și planificarea vitezelor. În prima iterație se va calcula traseul optim dintre obstacolele statice PS (Position Space), PO (Position Obstacle), iar apoi viteza de-a lungul traseului ales, astfel încât, acestea să fie evitate. Dacă pe traseu apare un obstacol mobil, robotul va trebui să își modifice dinamica (Fig. 2). Metoda VO definește obstacolele în spațiul de viteză VS (Velocity Space), în care viteza este neliniară. Astfel, se va defini și un spațiu al accelerațiilor AS (Acceleration Space) [10]. VO în schimb nu va specifica unde și când se va putea realiza o coliziune, spre exemplu poate alege calea cea mai lungă .

3. CONUL DE COLIZIUNE – VITEZA OBSTACOLULUI VO

Pentru determinarea conului de coliziune se va calcula viteza relativă a robotului față de cea a obstacolului $v_{RO} = v_R - v_O$, care ajută la stabilirea spațiului necesar evitării coliziunii. Condiția de coliziune presupune ca linia imaginară care reprezintă direcția vitezei v_{RO} să nu intersecteze PO: $l_{RO} \bigcap PO \neq \varnothing$, echivalent cu un set de viteze relative, care formează Conul Relativ de Coliziune (RCC – Relative Collision Cone) $RCC = \{v_{RO} | l_{RO} \bigcap PO \neq \varnothing\}$. După cum se poate constata (Fig. 2) RCC reprezintă spațiul dintre cele două linii de tangență la obstacol l_{MO} și l_{NO}. Altfel spus, condiția de evitare a coliziunii presupune ca $v_{RO} \notin RCC$.

Dacă RCC se regăsește pe direcția obstacolului, respectiv de-a lungul lui v_O atunci vom obține un spațiu denumit Conul Absolut de Coliziune (ACC – Absolute Collision Cone) $ACC = RCC \oplus v_O$, unde $\oplus$ – este suma Minkowski [11].

Dacă se îndeplinește condiția $\{(v_R \in ACC) \equiv (v_{RO} \in RCC)\}$ atunci ACC reprezintă multitudinea de vectori de viteză v_R, care pot facilita realizarea coliziunii dintre robot și obstacol și se numește Viteza Obstacolelor VO (Velocity Obstacle).

$$\begin{cases} VO_{RO}(v_O) = \{v_{Rnew} | (v_{Rnew} - v_O) \in RCC\} \\ v_{Rnew} \in VO_{RO}(v_O) \Leftrightarrow (v_{Rnew} - v_O) \in RCC \end{cases} \quad (1)$$

unde: v_{Rnew} – este noua viteză a robotului R.

4. VITEZA INTERACTIVĂ A OBSACOLULUI IVO

Pentru ca robotul să evite coliziunea va trebui să aleagă o altă viteză, diferită de cea conținută în ACC, respectiv v_{ROnew}. Această nouă viteză se plasează în afara RCC.

Dacă obstacolul este tot un robot, atunci viteza relativă a acestuia va fi de sens invers față de cea a robotului de bază (ca sens) $v_{ROnew}\left(v_{ORnew}\right)$ și are o valoare medie a vitezei curente $v_{RO}\left(v_{OR}\right)$ și a uneia aflată în afara RCC:

$$\begin{cases} IVO_{RO}\left(v_O\right)=\left\{v_{Rnew}\left|\left(v_{Rnew}-v_O=\dfrac{v_{RO}+v_{RO\ \text{oricare}}}{2}\right),v_{RO\ \text{oricare}}\in RCC\right.\right\} \\ IVO_{RO}\left(v_O\right)=\left\{v_{Rnew}\left|\left(2\cdot v_{Rnew}-v_R-v_O\right)\in RCC\right.\right\} \end{cases} \tag{2}$$

unde: $\left[\left(2\cdot v_{Rnew}-v_R-v_O\right)\in RCC\right]\equiv\left[\left(2\cdot v_{Rnew}-v_R\right)\in RCC\right]$.

În funcție de cum se poziționează față de spațiul dintre cele două obiecte $\left(p_O-p_R\right)$ avem două situații:

1. când se află de aceeași parte a medianei viteza obstacolelor este dată de $IVO_{RO}\left(v_O\right)$;
2. când nu se află de aceeași parte se va alege $VO_{RO}\left(v_O\right)$.

Existența celor două situații de alegre a vitezelor obstacolelor a condus la definirea unei soluții hibride, care să satisfacă oricare dintre scenarii de lucru, respectiv Viteza Interactivă Hibridă a Obstacolului HIVO:

$$HIVO_{RO}\left(v_O\right)=\begin{cases} VO_{RO}\left(v_O\right)\left|\left(\theta_{ROnew}-\alpha_{RO}\right)\cdot\left(\theta_{RO}-\alpha_{RO}\right)\le 0\right. \\ IVO_{RO}\left(v_O\right)\left|\left(\theta_{ROnew}-\alpha_{RO}\right)\cdot\left(\theta_{RO}-\alpha_{RO}\right)>0\right. \end{cases} \tag{3}$$

unde: $\alpha_{RO}=\angle\left(p_O-p_R\right)$; $\theta_{RO}=\angle\left(v_{RO}\right)$; $\theta_{ROnew}=\angle\left(v_{ROnew}\right)$.

5. PLANIFICAREA MIȘCĂRII ÎN VCS

Spațiul în care se deplasează USV și potențialele obstacole este caracterizat de intersecții și muchii a căror configurație este variabilă în timp. Acest lucru face ca planificarea mișcării să fie destul de complicată. De aceea problema de selecție a accelerației într-un spațiu VCS se va baza pe cartografierea spațiului și a obstacolelor dar și prin stabilirea constrângerilor dinamice. Pentru planificare trebuie determinate: distanța de siguranță, distanța de coliziune și timpul de coliziune.

i. distanța de siguranță se calculează ținând cont de timpul perioadei de acțiune T, în care obstacolul accelerează brusc, perioadă de timp în care poate apare riscul de coliziune dintre robot și obstacol:

$$d_{safe} = r_{OR} + v_{OR} \cdot T + 0.5 \cdot a_{O\max} \cdot T^2 \tag{4}$$

unde: $a_{O\max}$ - accelerația maximă a obstacolului; $v_R \in VO_{RO}$ - distanța de siguranță.

ii. distanța de coliziune se calculează ținând cont de timpul curent de dinaintea coliziunii, deoarece VO nu specifică decât că robotul va intra în coliziune cu un obstacol, dar nu când (timp) și unde (spațiu):

$$d_c = d_{RO} - r_{OR} \tag{5}$$

$$r_{OR} \equiv r_{ORsafe} = k_{dtf} \cdot \left[f_d(\gamma_d) + f_t(\gamma_t) \right] \cdot d_{safe} \tag{6}$$

$$\begin{cases} \gamma_d = k_{df} \cdot \dfrac{d_c}{d_{safe}} \\ \gamma_t = k_{tf} \cdot \dfrac{t_c}{T} \end{cases} \tag{7}$$

unde: k_{df} și k_{tf} - sunt factori de distanță; $f_d(\gamma_d)$ și $f_t(\gamma_t)$ sunt funcții variabile în timp.

iii. Timpul până la coliziune este timpul necesar vectorului viteză să atingă limita de siguranță a robotului și a obstacolului:

$$\left| p_{\mathcal{A}} + \tau \cdot (v_{\mathcal{A}} - v_{\mathcal{B}}) \right| \in \partial(\mathcal{B} \oplus -\mathcal{A}) \tag{8}$$

unde: $\mathcal{A}$ – conturul robotului; $v_{\mathcal{A}}\left[m/s \right]$ – viteza din interiorul VO al robotului; $\mathcal{B}$ - conturul obstacolului; $v_{\mathcal{B}}\left[m/s \right]$ – viteza din interiorul VO al obstacolului; $\tau[s]$ – timpul până la coliziune; $\partial(\bullet)$ - definește conturul ce trebuie ocolit de către robot..

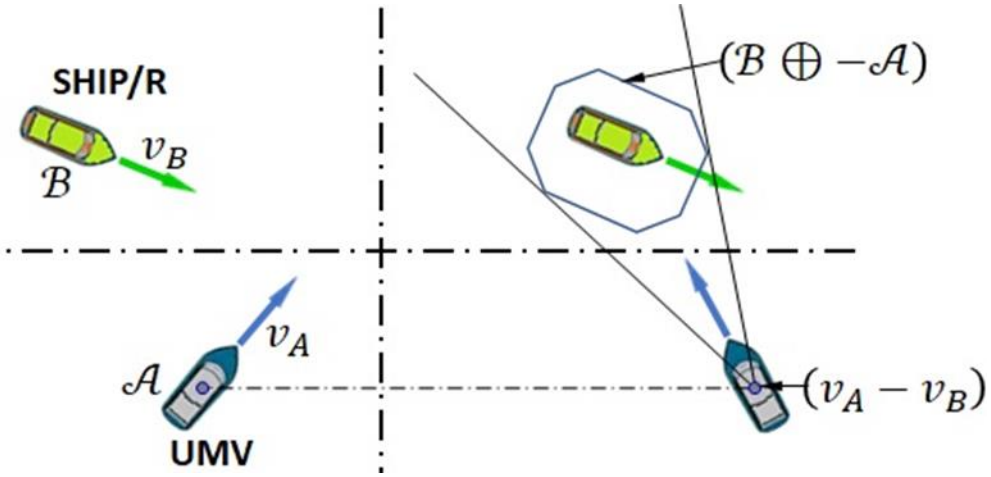

Fig. 5-1 Reprezentarea schematizată a principiului de determinare a timpului de coliziune [1].

Vectorul viteză $v_{\mathcal{A}} - v_{\mathcal{B}}$ este o reprezentare a faptului că robotul intersectează obstacolul. Se observă (Fig. 3) că spațiile de evitare a coliziunii respectă condiția anticoliziune dacă VO se încadrează într-o formă de con (conul sațiului de viteză). Explicația simplificată este aceea că atât timp cât $v_{\mathcal{A}}$ (viteza robotului) este în afara VO atunci nu se va produce nici o coliziune.

6. SOFTWARE[1]

https://github.com/uc2013171665/asv_colregs/blob/master/src/ASV/launch/asv.launch

```
<?xml version="1.0"?>
<!-- Base -->
<launch>

  <node pkg="ceiia_asv" type="Ownship_vehicle.py" name="Ownship" respawn="true" output="screen" args="3" />
  <node pkg="ceiia_asv" type="Contact_vehicle.py" name="Contact" respawn="true" args="3"/> <!-- output="screen" -->

  <!-- args == GiveWayOT = 2  HeadOn = 3  StandOnOT = 4  GiveWayX = 5 StandOnX = 6 -->

  <!--- #FIXME: <include file="$(find ASV_description)/world/_____.launch"> --->

 </include>

</launch>
```

==========

https://github.com/uc2013171665/asv_colregs/blob/master/src/ASV/src/Contact_vehicle.py

```
#!/usr/bin/env python

###---------Import modules----------###

import rospy #need to import rospy if you are writing a ROS Node
import math
import numpy as np
from ceiia_asv_msgs.msg  import AIS,debug,desired,Course
from nav_msgs.msg import Odometry
from sensor_msgs.msg import NavSatFix

from COLREGS_utils import functions, quat_euler_yaw, main_algorithm, lat_lon_convert
```

[1] https://github.com/uc2013171665/asv_colregs/tree/master

```
HeadingRangeConvert360 = functions.HeadingRangeConvert360
quaternion_to_euler_angle = quat_euler_yaw.quaternion_to_euler_angle #Convert quaternion to euler and get heading
geo_utm = lat_lon_convert.geo_utm #Function to convert lat and lon coordinates to local coordinates
Algorithm_1 = main_algorithm.Algorithm_1



class Vehicle_status(object):
    """docstring for Vehicle_status."""
    def __init__(self):
        super(Vehicle_status, self).__init__()
        ##Initialize object: Ownship or Contact##
        self.mmsi = None
        self.lat = None
        self.lon = None
        self.x = None  # [m]
        self.y = None  # [m]
        self.COG = None  # [degrees]
        self.SOG = None # [m/s]
        self.lenght = None  # [m]

        #Initialize mode and submode
        self.mode = 0
        self.submode = 0

    def ownship(self):
        self.mmsi = 123456789
        self.lat = 41.18
        self.lon = -8.70
        self.x = 0  # [m]
        self.y = 0  # [m]
        self.COG = 180  # [degrees]
        self.SOG = 0.5 # [m/s]
        self.lenght = 3  # [m]

    def contact(self):
        self.mmsi = 999999999
```

```
        self.lat = 41.18080
        self.lon = -8.703605
        self.x = -45.0  # [m]
        self.y = -105.0  # [m]
        self.COG = 20  # [degrees]
        self.SOG = 4 # [m/s]
        self.lenght = 7.5  # [m]

    def callback(self, msg): #Contact AIS data
        self.mmsi = msg.mmsi

        if self.mmsi == 123456789:
            self.lat = msg.Latitude
            self.lon = msg.Longitude
            self.SOG = msg.SOG
            self.COG = msg.COG
            self.lenght = msg.lenght
            self.mode = msg.mode

    def update_xy(self, x_cn, y_cn): #Contact AIS data

        self.x = x_cn
        self.y = y_cn

    def callback_2(self, msg): #Update Ownship pose and twist values

        v_x = msg.twist.twist.linear.x
        v_y = msg.twist.twist.linear.y

        self.SOG = math.sqrt(v_x ** 2 + v_y ** 2)

        x_e,y_e,z_e = quaternion_to_euler_angle(msg.pose.pose.orientation.x, \
        msg.pose.pose.orientation.y, msg.pose.pose.orientation.z, msg.pose.pose.orientation.w)

        teta = z_e * (-1) + 90
        self.COG = HeadingRangeConvert360(teta)

    def callback_3(self, msg): #Update Ownship pose and twist values
```

```
        self.lat = msg.latitude
        self.lon = msg.longitude

class Parameters(object):
    """docstring for Vehicle_status."""
    def __init__(self):
        super(Parameters, self).__init__()
        ##Reference Parameters:
        self.lat_0 = 41.1812101
        self.lon_0 = -8.7048601
        self.fi_headon = 20 #12 # [degrees]
        self.g_max = 100
        self.g_min = 0
        self.Safety_interval = 2
        self.Start_Manouver = 6.5

def talker(heading,velocity,mmsi): #ROS Publisher #Sets the desired velocity and yaw to the control module

    if mmsi == 123456789:
        vid_number = "1"
    else:
        vid_number = "2"

    pub = rospy.Publisher('/ceiia/internal/ctl_fbw', Course, queue_size=10)
    rate = rospy.Rate(3) # 1hz
    desired.velocity = velocity
    desired.yaw = heading #FIXME: The algorithm has the yaw value inversed with gazebo, right now the *(-1) is fixing the problem
    desired.vid = vid_number
    pub.publish(desired,False) #Course message accepts 2 inputs, one called desired which has .velocity and .yaw values.
    rate.sleep()

def Ownship_subscriber(): #ROS Subscriber #Subscribes the AIS data from other vehicles
    rospy.Subscriber("chatter", AIS, Ownship.callback)
    r = rospy.Rate(1) # 1hz
```

```
def talker_AIS(mmsi,Longitude,Latitude,SOG,COG,lenght,mode): #ROS Publisher #Publishes the Ownship AIS data
    pub = rospy.Publisher('chatter', AIS, queue_size=10)
    rate = rospy.Rate(1) # 1hz
    pub.publish(mmsi,Longitude,Latitude,SOG,COG,lenght,mode)
    rate.sleep()

def Odometry_subscriber(): #ROS Subscriber #Subscribes the odometry treated with the kalman filter
    rospy.Subscriber("p3d_odom_2", Odometry, Contact.callback_2) #topic,message,function
    r = rospy.Rate(1) # 1hz


def gps_subscriber(): #ROS Subscriber #Subscribes the odometry treated with the kalman filter
    rospy.Subscriber("gps/fix_2", NavSatFix, Contact.callback_3) #topic,message,function
    r = rospy.Rate(1) # 1hz


if __name__ == '__main__':
    rospy.init_node('Contact_AIS', anonymous=False) #Initialize node which is called COLREGS
    r = rospy.Rate(3) # 1hz

    ref = Parameters()
    Contact = Vehicle_status()
    Ownship = Vehicle_status()
    Contact.contact()
    Ownship.ownship()

    while not rospy.is_shutdown():

        node_os = Ownship_subscriber() #Subscribe AIS messages from the /chatter topic and go to the callback function
        x_os,y_os = geo_utm(ref.lat_0,ref.lon_0,Ownship.lat,Ownship.lon) #Update local coordinates of the Contact vehicle
        Ownship.update_xy(x_os,y_os) #Append to the object Contact the properties "x,y"

        r_cpa_min = Ownship.lenght * ref.Safety_interval #Inner perimeter
        r_cpa_max = Ownship.lenght * ref.Start_Manouver #Outer perimeter
        r_pwt = r_cpa_max  #FIXME: r_pwt will be equal to r_cpa_max??? Still needs to be decided
```

```
    node = Odometry_subscriber() #Subscribe odometry
    node_2 = gps_subscriber() #Subscribe lat and lon

    Contact.mode, Contact.submode,teta_d,v_d,f_function_final, yaw,mmsi = Algorithm_1(Contact,Ownship,r_cpa_max,\
                                                                    r_cpa_min,r_pwt,ref,Contact.mmsi)
    talker(yaw,v_d,mmsi)


talker_AIS(Contact.mmsi,Contact.lon,Contact.lat,Contact.SOG,Contact.COG,Contact.lenght,Contact.mode)

    r.sleep()
```

==========

https://github.com/uc2013171665/asv_colregs/blob/master/src/ASV/src/Ownship_vehicle.py

```
#!/usr/bin/env python

###---------Import modules----------###

import rospy #need to import rospy if you are writing a ROS Node
import math
import numpy as np
from ceiia_asv_msgs.msg  import AIS,debug,desired,Course
from nav_msgs.msg import Odometry
from sensor_msgs.msg import NavSatFix

from COLREGS_utils import main_algorithm, quat_euler_yaw, functions, lat_lon_convert

import sys

quaternion_to_euler_angle = quat_euler_yaw.quaternion_to_euler_angle #Convert quaternion to euler and get heading
HeadingRangeConvert360 = functions.HeadingRangeConvert360
range_func = functions.range_func
geo_utm = lat_lon_convert.geo_utm #Function to convert lat and lon coordinates to local coordinates
Algorithm_1 = main_algorithm.Algorithm_1
```

```
###----------Debug ROS message----------###

def
debugg(counter,mode_cn,submode_cn,lat_cn,lon_cn,SOG_cn,COG_cn,lenght_cn,x_cn,y_cn,mode_os,subm
ode_os,lat_os,lon_os,SOG_os,COG_os,lenght_os,x_os,y_os,r_cpa_max,r):
    pub2 = rospy.Publisher('DEBUG', debug, queue_size=10)
    rate = rospy.Rate(1) # 1hz

pub2.publish(counter,mode_cn,submode_cn,lat_cn,lon_cn,SOG_cn,COG_cn,lenght_cn,x_cn,y_cn,mode_os,
submode_os,lat_os,lon_os,SOG_os,COG_os,lenght_os,x_os,y_os,r_cpa_max,r)
    rate.sleep()

###----------ROS----------###

def talker(heading,velocity,mmsi): #ROS Publisher #Sets the desired velocity and yaw to the control module

    if mmsi == 123456789:
        vid_number = "1"
    else:
        vid_number = "2"

    pub = rospy.Publisher('/ceiia/internal/ctl_fbw', Course, queue_size=10)
    rate = rospy.Rate(3) # 1hz
    desired.velocity = velocity
    desired.yaw = heading #FIXME: The algorithm has the yaw value inversed with gazebo, right now the *(-
1) is fixing the problem
    desired.vid = vid_number
    pub.publish(desired,False) #Course message accepts 2 inputs, one called desired which has .velocity and
.yaw values.
    rate.sleep()

def talker_AIS(mmsi,Longitude,Latitude,SOG,COG,lenght,mode): #ROS Publisher #Publishes the Ownship
AIS data
    pub = rospy.Publisher('chatter', AIS, queue_size=10)
    rate = rospy.Rate(1) # 1hz
    pub.publish(mmsi,Longitude,Latitude,SOG,COG,lenght,mode)
    rate.sleep()
```

```
def Contact_subscriber(): #ROS Subscriber #Subscribes the AIS data from other vehicles
    rospy.Subscriber("chatter", AIS, Contact.callback)
    r = rospy.Rate(1) # 1hz

def Ownship_subscriber(): #ROS Subscriber #Subscribes the odometry treated with the kalman filter
    rospy.Subscriber("p3d_odom", Odometry, Ownship.callback_2) #topic,message,function
    r = rospy.Rate(1) # 1hz

def gps_subscriber(): #ROS Subscriber #Subscribes the odometry treated with the kalman filter
    rospy.Subscriber("gps/fix", NavSatFix, Ownship.callback_3) #topic,message,function
    r = rospy.Rate(1) # 1hz

###------- Class to simplify dynamic and static data ------- ###

class Vehicle_status(object):
    """docstring for Vehicle_status."""
    def __init__(self):
        super(Vehicle_status, self).__init__()
        ##Initialize object: Ownship or Contact##
        self.mmsi = None
        self.lat = None
        self.lon = None
        self.x = None  # [m]
        self.y = None  # [m]
        self.COG = None  # [degrees]
        self.SOG = None # [m/s]
        self.lenght = None  # [m]

        #Initialize mode and submode
        self.mode = 0
        self.submode = 0

    def ownship(self):
        self.mmsi = 123456789
        self.lat = 41.18
        self.lon = -8.70
        self.x = 0  # [m]
        self.y = 0  # [m]
```

```
    self.COG = 180  # [degrees]
    self.SOG = 0.5 # [m/s]
    self.lenght = 3  # [m]

def contact(self):
    self.mmsi = 999999999
    self.lat = 41.18080
    self.lon = -8.703605
    self.x = -45.0  # [m]
    self.y = -105.0  # [m]
    self.COG = 20  # [degrees]
    self.SOG = 4 # [m/s]
    self.lenght = 7.5  # [m]

def callback(self, msg): #Contact AIS data
    self.mmsi = msg.mmsi

    if self.mmsi != 123456789:
        self.lat = msg.Latitude
        self.lon = msg.Longitude
        self.SOG = msg.SOG
        self.COG = msg.COG
        self.lenght = msg.lenght
        self.mode = msg.mode

def update_xy(self, x_cn, y_cn): #Contact AIS data

    self.x = x_cn
    self.y = y_cn

def callback_2(self, msg): #Update Ownship pose and twist values

    self.x = msg.pose.pose.position.x
    self.y = msg.pose.pose.position.y
    v_x = msg.twist.twist.linear.x
    v_y = msg.twist.twist.linear.y

    self.SOG = math.sqrt(v_x ** 2 + v_y ** 2)
```

```
        x_e,y_e,z_e = quaternion_to_euler_angle(msg.pose.pose.orientation.x, \
        msg.pose.pose.orientation.y, msg.pose.pose.orientation.z, msg.pose.pose.orientation.w)

        teta = z_e * (-1) + 90
        self.COG = HeadingRangeConvert360(teta)

    def callback_3(self, msg): #Update Ownship pose and twist values
        # FIXME: Find a way to not use global variables

        self.lat = msg.latitude
        self.lon = msg.longitude

class Parameters(object):
    """docstring for Vehicle_status."""
    def __init__(self):
        super(Parameters, self).__init__()
        ##Reference Parameters:
        self.lat_0 = 41.1812101
        self.lon_0 = -8.7048601
        self.fi_headon = 20 #12 # [degrees]
        self.g_max = 100
        self.g_min = 0
        self.Safety_interval = 2
        self.Start_Manouver = 6.5

if __name__ == '__main__':
    rospy.init_node('COLREGS', anonymous=False) #Initialize node which is called COLREGS
    r = rospy.Rate(3) # 1hz

    print """
        modes   |         submodes  |
       --------- | -----    --------- | -----
         Null   |  0        None    |  0
         CPA    |  1        Port    |  1
       GiveWayOT |  2      Starboard |  2
        HeadOn   |  3        Bow     |  3
       StandOnOT |  4       Stern    |  4
```

```
    GiveWayX  |  5
    StandOnX  |  6
    """
    counter = 0 #Initialize counter to help in debbuging
    ref = Parameters()
    Contact = Vehicle_status()
    Ownship = Vehicle_status()
    Contact.contact()
    Ownship.ownship()

    #print Contact.lat,Contact.lon,Contact.SOG,Contact.COG,Contact.lenght,Contact.x,Contact.y,\
    #Ownship.lat,Ownship.lon,Ownship.SOG,Ownship.COG,Ownship.lenght,Ownship.x,Ownship.y

    while not rospy.is_shutdown():
        counter += 1
        print "          ---          "
        print("#####----Debugging--(%s)--####" % counter)
        print "          ---          "

        node = Contact_subscriber() #Subscribe AIS messages from the /chatter topic and go to the callback function
        x_cn,y_cn = geo_utm(ref.lat_0,ref.lon_0,Contact.lat,Contact.lon) #Update local coordinates of the Contact vehicle
        Contact.update_xy(x_cn,y_cn) #Append to the object Contact the properties "x,y"

        r_cpa_min = Contact.lenght * ref.Safety_interval #Inner perimeter
        r_cpa_max = Contact.lenght * ref.Start_Manouver #Outer perimeter
        r_pwt = r_cpa_max  #FIXME: r_pwt will be equal to r_cpa_max??? Still needs to be decided

        node_2 = Ownship_subscriber() # - update ownship data

        range_xy = range_func(Ownship,Contact) #Range is calculated here for the debbug message below

        Ownship.mode, Ownship.submode,teta_d,v_d,f_function_final, yaw,mmsi = Algorithm_1(Ownship,Contact,r_cpa_max,\
                    r_cpa_min,r_pwt,ref,Ownship.mmsi) #Output from Algorithm_1, getting the mode, submode and desired velocity
```

```
        talker(yaw,v_d,mmsi)

        node_gps = gps_subscriber() #Subscribe lat and lon

        if mmsi == 123456789:

talker_AIS(Ownship.mmsi,Ownship.lon,Ownship.lat,Ownship.SOG,Ownship.COG,Ownship.lenght,Ownship.mode)


debugg(counter,Contact.mode,Contact.submode,Contact.lat,Contact.lon,Contact.SOG,Contact.COG,Contact.lenght,x_cn,y_cn,\
        Ownship.mode,                                                    Ownship.submode,
Ownship.lat,Ownship.lon,Ownship.SOG,Ownship.COG,Ownship.lenght,Ownship.x,Ownship.y,r_cpa_max,range_xy)
        r.sleep()

    #plot_f_function(f_function_final) #Only Uncomment to generate a 3D plot of the f_function
    #print f_function_final[58,3] #Print for debugging


==========================================================================
https://github.com/uc2013171665/asv_colregs/blob/master/src/ASV/src/lidar.py
#!/usr/bin/env python

###---------Import modules----------###

import rospy #need to import rospy if you are writing a ROS Node
import math
import numpy as np
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from ceiia_asv_msgs.msg  import desired, Course, debug2
from COLREGS_utils import quat_euler_yaw, functions

import random
import time
import sys
```

```
ranges = [None]*60
angle_increment = 0.0532473213971

quaternion_to_euler_angle = quat_euler_yaw.quaternion_to_euler_angle #Convert quaternion to euler and get heading
angle_to_yaw = quat_euler_yaw.angle_to_yaw
HeadingRangeConvert360 = functions.HeadingRangeConvert360

###----------Debug ROS message----------###

def debugg(counter,danger,dist_to_goal,x,y,v,w,yaw):
    pub2 = rospy.Publisher('DEBUG2', debug2, queue_size=10)
    rate = rospy.Rate(2) # 1hz
    pub2.publish(counter,danger,v,w,yaw,x,y,dist_to_goal)
    rate.sleep()

def talker(heading,velocity): #ROS Publisher #Sets the desired velocity and yaw to the control module
    pub = rospy.Publisher('/ceiia/internal/ctl_fbw', Course, queue_size=10)
    rate = rospy.Rate(2) # 1hz
    desired.velocity = velocity
    desired.yaw = heading #FIXME: The algorithm has the yaw value inversed with gazebo, right now the *(-1) is fixing the problem
    pub.publish(desired,False) #Course message accepts 2 inputs, one called desired which has .velocity and .yaw values.
    rate.sleep()

def Ownship_subscriber(): #ROS Subscriber #Subscribes the odometry treated with the kalman filter
    rospy.Subscriber("p3d_odom", Odometry, Ownship.callback_2) #topic,message,function
    r = rospy.Rate(2) # 1hz

class Config():
    # simulation parameters

    def __init__(self):
        # robot parameter
        self.max_speed = 2.0  # [m/s]
        self.min_speed = 0.0  # [m/s]
```

```
        self.max_yawrate = 60.0 * math.pi / 180.0  # [rad/s]
        self.max_accel = 1.0  # [m/ss]
        self.max_dyawrate = 60.0 * math.pi / 180.0  # [rad/ss]
        self.v_reso = 0.5  # [m/s]
        self.yawrate_reso = 0.5 * math.pi / 180.0  # [rad/s]
        self.dt = 1.0  # [s]
        self.predict_time = 4.0  # [s]
        self.to_goal_cost_gain = 1.0
        self.speed_cost_gain = 1.0
        self.robot_radius = 5.0  # [m]


def motion(x, u, dt):
    # motion model

    x[0] += u[0] * math.cos(x[2]) * dt
    x[1] += u[0] * math.sin(x[2]) * dt
    x[2] += u[1] * dt
    x[3] = u[0]
    x[4] = u[1]

    return x


def calc_dynamic_window(x, config):

    # Dynamic window from robot specification
    Vs = [config.min_speed, config.max_speed,
          -config.max_yawrate, config.max_yawrate]

    # Dynamic window from motion model
    Vd = [x[3] - config.max_accel * config.dt,
          x[3] + config.max_accel * config.dt,
          x[4] - config.max_dyawrate * config.dt,
          x[4] + config.max_dyawrate * config.dt]
    #  print(Vs, Vd)

    #  [vmin,vmax, yawrate min, yawrate max]
```

```
    dw = [max(Vs[0], Vd[0]), min(Vs[1], Vd[1]),
          max(Vs[2], Vd[2]), min(Vs[3], Vd[3])]
    #  print(dw)

    return dw


def calc_trajectory(xinit, v, y, config):

    x = np.array(xinit)
    traj = np.array(x)
    time = 0
    while time <= config.predict_time:
        x = motion(x, [v, y], config.dt)
        traj = np.vstack((traj, x))
        time += config.dt

    #  print(len(traj))
    return traj


def calc_final_input(x, u, dw, config, goal, ob):

    xinit = x[:]
    min_cost = 10000.0
    min_u = u
    min_u[0] = 0.5
    best_traj = np.array([x])

    # evaluate all trajectory with sampled input in dynamic window
    for v in np.arange(dw[0], dw[1], config.v_reso):
        for y in np.arange(dw[2], dw[3], config.yawrate_reso):
            traj = calc_trajectory(xinit, v, y, config)

            # calc cost
            to_goal_cost = calc_to_goal_cost(traj, goal, config)
            speed_cost = config.speed_cost_gain * \
                (config.max_speed - traj[-1, 3])
```

```
            ob_cost = calc_obstacle_cost(traj, ob, config)
            #print("ob_cost = %s for y equal to %s:" % (ob_cost,y))

            final_cost = to_goal_cost + speed_cost + ob_cost

            # search minimum trajectory
            if min_cost >= final_cost:
                min_cost = final_cost
                min_u = [v, y]
                best_traj = traj
    #  input()
    print "best_traj:",best_traj
    print "min_u:", min_u
    dim = best_traj.shape
    if dim[0] == 1:
        min_u = [best_traj[0,3],best_traj[0,4]]
        print "min_u:", min_u
    return min_u #, best_traj


def calc_obstacle_cost(traj, ob, config):
    # calc obstacle cost inf: collistion, 0:free

    skip_n = 2
    minr = float("inf")

    for ii in range(0, len(traj[:, 1]), skip_n):
        for i in range(len(ob[:, 0])):
            ox = ob[i, 0]
            oy = ob[i, 1]
            dx = traj[ii, 0] - ox
            dy = traj[ii, 1] - oy

            r = math.sqrt(dx**2 + dy**2)
            if r <= config.robot_radius:
                return float("Inf")  # collisiton

            if minr >= r:
```

```
            minr = r

    return 1.0 / minr  # OK


def calc_to_goal_cost(traj, goal, config):
    # calc to goal cost. It is 2D norm.

    dy = goal[0] - traj[-1, 0]
    dx = goal[1] - traj[-1, 1]
    goal_dis = math.sqrt(dx**2 + dy**2)
    cost = config.to_goal_cost_gain * goal_dis

    return cost


def dwa_control(x, u, config, goal, ob):
    # Dynamic Window control
    dw = calc_dynamic_window(x, config)

    u = calc_final_input(x, u, dw, config, goal, ob) #, traj

    return u #, traj

###------- Class to simplify dynamic and static data ------- ###

class Vehicle_status(object):
    """docstring for Vehicle_status."""
    def __init__(self):
        super(Vehicle_status, self).__init__()
        ##Initialize object: Ownship or Contact##
        self.lat = 41.18
        self.lon = -8.70
        self.x = 0  # [m]
        self.y = 0  # [m]
        self.COG = 180  # [degrees]
        self.SOG = 0.5 # [m/s]
        self.lenght = 5  # [m]
```

```
        self.GiveWay = False  # [bool]
        self.StandOn = False # [bool]
        self.w = 0 # [rad/s]

    def callback_2(self, msg): #Update Ownship pose and twist values
        # FIXME: Find a way to not use global variables

        self.x = msg.pose.pose.position.x
        self.y = msg.pose.pose.position.y
        v_x = msg.twist.twist.linear.x
        v_y = msg.twist.twist.linear.y

        self.SOG = math.sqrt(v_x ** 2 + v_y ** 2)

        x_e,y_e,z_e = quaternion_to_euler_angle(msg.pose.pose.orientation.x, \
        msg.pose.pose.orientation.y, msg.pose.pose.orientation.z, msg.pose.pose.orientation.w)

        teta = z_e * (-1) + 90
        self.COG = HeadingRangeConvert360(teta)

        w_x = msg.twist.twist.angular.x
        w_y = msg.twist.twist.angular.y
        self.w = math.sqrt(w_x ** 2 + w_y ** 2)

def callback(msg): #Contact AIS data

    global ranges
    global angle_increment

    angle_increment = msg.angle_increment
    ranges = msg.ranges

def ranges_ind(ranges,number_of_points,Ownship,obstacle):
    danger = False #Initialize always as false
    for i in range(number_of_points):
        if (ranges[i] <= 30 and ranges[i] != None):
            x_o = Ownship.x + math.sin(math.radians(Ownship.COG + 90 - (3 * i))) * ranges[i]
            y_o = Ownship.y + math.cos(math.radians(Ownship.COG + 90 - (3 * i))) * ranges[i]
```

```
            danger = True
            if [int(x_o),int(y_o)] not in obstacle:
                obstacle.append([int(x_o),int(y_o)])
    return obstacle, danger

def goals(Ownship, counter):

    x_goal = Ownship.x +  math.sin(math.radians(Ownship.COG)) * 50
    y_goal = Ownship.y +  math.sin(math.radians(Ownship.COG)) * 50

    goal = ([int(x_goal), int(y_goal)])

    return goal

def lidar_data(): #ROS Subscriber #Subscribes the AIS data from other vehicles
    rospy.Subscriber("mybot/laser/scan", LaserScan, callback)
    r = rospy.Rate(2) # 1hz

if __name__ == '__main__':
    rospy.init_node('Hokuyo', anonymous=False) #Initialize node which is called Hokuyo
    r = rospy.Rate(2) # 1hz

    counter = 0
    obstacle = [[100, 100]] #FIXME: I have to give an initial obstacle to prevent error. Need fix.

    Ownship = Vehicle_status() #Initialize the Ownship Object
    goal = np.array([-50,-103]) #Goal given for debbuging
    node = lidar_data() #Subscribes the lidar data --> It was firstly called here because we need the number of
points
    number_of_points = int( (180 / (angle_increment * 360 / ( 2 * math.pi ) ) ) + 1 ) #Number of points from
the lidar measurments

    u = np.array([0.0, 0.0]) #Initialize the velocities for the DWA
    ob = np.matrix(obstacle) #Create a matrix for the obstacles

    config = Config()

    while not rospy.is_shutdown():
```

```
counter +=1

node = lidar_data() #Subscribes the lidar data
node_2 = Ownship_subscriber() # - update ownship data

#With the lidar_data and previous obstacles, update the obstacle variable
obstacle, danger = ranges_ind(ranges,number_of_points,Ownship,obstacle)
ob = np.matrix(obstacle)

# initial state [x(m), y(m), yaw(rad), v(m/s), omega(rad/s)]
x_inst = [Ownship.x, Ownship.y, - angle_to_yaw(Ownship.COG), Ownship.SOG, Ownship.w]
x = np.array(x_inst)

print """Beginning:
------------------------------
"""
print("the goal is %s and counter is %s" % (goal, counter))
print("I am in x:%s and y:%s, the yaw is %s, being the velocity %s and the rate %s" % (Ownship.x, Ownship.y, - angle_to_yaw(Ownship.COG), Ownship.SOG, Ownship.w))

dist_to_goal = math.sqrt((Ownship.x - goal[0])**2 + (Ownship.y - goal[1])**2)

if (dist_to_goal > config.robot_radius):

    start_time = time.time()
    #u = np.array([0.0, 0.0]) #Initialize the velocities for the DWA
    print "before u:", u
    u = dwa_control(x, u, config, goal, ob) #, ltraj
    print "after u:", u
    x = motion(x, u, config.dt)

    talker(-x[2],x[3])
    print("I have sent yaw:%s and velocity:%s and the theorethical x is %s and y is %s" % (-x[2], x[3], x[0], x[1]))
    print("However the real values --> yaw:%s and velocity:%s and the real x is %s and y is %s" % (- angle_to_yaw(Ownship.COG), Ownship.SOG, Ownship.x, Ownship.y))
    print ob
```

```
        dist_to_goal = math.sqrt((Ownship.x - goal[0])**2 + (Ownship.y - goal[1])**2)
        node_3 = debugg(counter,danger,dist_to_goal,x[0],x[1],x[3],x[4],-x[2])

        end_time = time.time()
        print("total time taken on loop: ", end_time - start_time)

        # check goal
        if dist_to_goal <= config.robot_radius:
            print """

            Goal!!

            """
            talker(-x[2],0)

    if danger == False:
        for i in range(len(ob[:, 0])):
            ox = ob[i, 0]
            oy = ob[i, 1]

            rang = math.sqrt( (Ownship.x - ox) ** 2 + (Ownship.y - oy) ** 2 )
            if rang < 30:
                danger = True
                break
            else:
                danger = False
        if danger == False:
                obstacle = [[100, 100]]
                ob = np.matrix(obstacle)

    r.sleep()

#rospy.spin()
```

==========

https://github.com/uc2013171665/asv_colregs/blob/master/src/ASV/src/COLREGS_utils/functions.py

```
import rospy
```

```
import math
import numpy as np

###---------- Convert Headings ----------###
def HeadingRangeConvert360(teta):
    if teta >= 0:
      new_teta = teta - math.floor(teta / 360) * 360
    else:
      new_teta = teta + (math.floor(- teta / 360) + 1) * 360
    return new_teta

def HeadingRangeConvert180(teta):
    if teta >= 0:
      new_teta = teta - math.floor((teta + 180) / 360) * 360
    else:
      new_teta = teta + (math.floor(( - teta + 180 ) / 360)) * 360
    return new_teta

def heading_deviation(teta_a,teta_b):
    teta_dev = abs(HeadingRangeConvert180(( teta_a - teta_b )))
    return teta_dev

def delta_teta(teta,teta_orig):
    delta_teta = HeadingRangeConvert180(teta - teta_orig)
    return delta_teta

###---------- Primary functions ----------##
def range_func(Ownship,Contact): #range between vehicles
    r_os_cn = math.sqrt( (Ownship.x - Contact.x) ** 2 + (Ownship.y - Contact.y) ** 2 )
    return r_os_cn

def Bearing(V_1,V_2): #Bearing from ownship to contact or in reverse
    if (V_1.x == V_2.x and V_1.y <= V_2.y):
        bng_os_cn = 0
    elif (V_1.x == V_2.x and V_1.y > V_2.y):
        bng_os_cn = 180
    elif (V_1.x < V_2.x and V_1.y <= V_2.y):
        bng_os_cn = 90 - math.atan(  abs(V_1.y-V_2.y) / abs(V_1.x - V_2.x) ) * 180 / math.pi
```

```
    elif (V_1.x < V_2.x and V_1.y > V_2.y):
        bng_os_cn = math.atan(  abs(V_1.y-V_2.y) / abs(V_1.x - V_2.x) ) * 180 / math.pi + 90
    elif (V_1.x > V_2.x and V_1.y > V_2.y):
        bng_os_cn = 270 - math.atan(  abs(V_1.y-V_2.y) / abs(V_1.x - V_2.x) ) * 180 / math.pi
    elif (V_1.x > V_2.x and V_1.y <= V_2.y):
        bng_os_cn = math.atan(  abs(V_1.y-V_2.y) / abs(V_1.x - V_2.x) ) * 180 / math.pi + 270
    return bng_os_cn

def alpha_and_beta(bng,teta): #function to obtain alpha or beta
    alpha_or_beta = HeadingRangeConvert360(bng - teta)
    return alpha_or_beta

#-------- Range Rate and Bearing Rate ---------#

def vel_in_direction(alpha_or_beta,vel):
    v = math.cos(math.radians(alpha_or_beta)) * vel
    return v

def range_rate_function(v_a,v_b):
    range_r = - ( v_a + v_b )
    return range_r

def tangent_heading(bng):
    teta_tn = HeadingRangeConvert360(bng + 90)
    return teta_tn

def speed_in_tang_head(teta,vel):
    velocity = math.cos(math.radians(teta)) * vel
    return velocity

def Bearing_rate(vel_a,vel_b,r):
    beta_r = - (vel_a + vel_b) * 360 / ( 2 * r * math.pi )
    return beta_r

###------------------ Boolean --------------------###
#------------------ Fore and Aft -------------------#

def fore_os_cn(alpha):
```

```
    if (alpha > 90 and alpha < 270):
        fore = 0
    else:
        fore = 1
    return fore

def aft_os_cn(alpha):
    if (alpha >= 90 and alpha <= 270):
        aft = 1
    else:
        aft = 0
    return aft

def fore_cn_os(beta):
    if (beta > 90 and beta < 270):
        fore = 0
    else:
        fore = 1
    return fore

def aft_cn_os(beta):
    if (beta >= 90 and beta <= 270):
        aft = 1
    else:
        aft = 0
    return aft

#---------- Port and Star ----------#

def port_os_cn(alpha):
    if (alpha >= 0 and alpha <= 180):
        port = 0
    else:
        port = 1
    return port

def star_os_cn(alpha):
    if (alpha >= 0 and alpha <= 180):
```

```
        star = 1
    else:
        star = 0
    return star

def port_cn_os(beta):
    if (beta >= 0 and beta <= 180):
        port = 0
    else:
        port = 1
    return port

def star_cn_os(beta):
    if (beta >= 0 and beta <= 180):
        star = 1
    else:
        star = 0
    return star

##------- Boolean Crossing Relationships -------##

def Crossing_relations_function(Ownship,Contact,bool,bng,r,teta_os,v_os): #teta_os changes
    if bool == 1:
        teta_g = HeadingRangeConvert360(Contact.COG + 90)
    else:
        teta_g = HeadingRangeConvert360(Contact.COG - 90)
    v_gamma = math.cos(math.radians( teta_os - teta_g )) * v_os
    r_g = r * math.cos(math.radians(teta_g - bng))
    return teta_g,v_gamma,r_g

def OwnshipCrossingContact(alpha,v_g,port,star,beta_r):
    if (alpha == 0 or alpha == 180):
        Cross_xcn = 1
    elif (v_g > 0):
        Cross_xcn = 1
    else:
        Cross_xcn = 0
    #---
```

```
    if (alpha == 0):
        Cross_xcnb = 1
    elif (v_g > 0 and port == 1 and beta_r > 0):
        Cross_xcnb = 1
    elif (v_g > 0 and star == 1 and beta_r < 0):
        Cross_xcnb = 1
    else:
        Cross_xcnb = 0
    #---
    if (alpha == 180):
        Cross_xcns = 1
    elif (v_g > 0 and port == 1 and beta_r < 0):
        Cross_xcns = 1
    elif (v_g > 0 and star == 1 and beta_r > 0):
        Cross_xcns = 1
    else:
        Cross_xcns = 0
    return Cross_xcn, Cross_xcnb, Cross_xcns

def numericalCrossing(r_gamma,v_gamma,v_cnh_os,v_cn,Cross_xcnb,Cross_xcns,r_eps):
    t_gamma_os = r_gamma / v_gamma
    r_eps_xos = t_gamma_os * v_cnh_os
    r_eps_xcn = t_gamma_os * v_cn

    if (Cross_xcnb == 1):
        r_xcnb = r_eps_xos - ( r_eps - r_eps_xcn )
    else:
        r_xcnb = -1

    if (Cross_xcns == 1):
        r_xcns = ( r_eps - r_eps_xcn ) - r_eps_xos
    else:
        r_xcns = -1
    return r_xcnb, r_xcns

##---------- Boolean Passing Relationships ----------##

def Passing_relations_function(Ownship, Contact,bool,bng,r,teta_os,v_os):  #teta_os changes
```

```
    if bool == 1: #condition changed to zero. Previously was 1
        teta_eps = HeadingRangeConvert360(Contact.COG + 180)
    else:
        teta_eps = HeadingRangeConvert360(Contact.COG)
    v_cnh = math.cos(math.radians( teta_os - teta_eps )) * v_os
    if bool == 1: #It must be 1, confirmed with Head_On situation
        v_eps = Contact.SOG + v_cnh
    else:
        v_eps = v_cnh - Contact.SOG
    r_eps = r * math.cos(math.radians(teta_eps - bng))
    return teta_eps,v_cnh,v_eps,r_eps

def OwnshipPassingContact(v_eps,aft,beta_r,fore):
    if (v_eps > 0):
        pass_cn = 1
    else:
        pass_cn = 0
    #---
    if (aft == 1 and pass_cn == 1 and beta_r > 0):
        pass_cnp = 1
    elif (fore == 1 and pass_cn == 1 and beta_r < 0):
        pass_cnp = 1
    else:
        pass_cnp = 0
    #---
    if (aft == 1 and pass_cn == 1 and beta_r < 0):
        pass_cns = 1
    elif (fore == 1 and pass_cn == 1 and beta_r > 0):
        pass_cns = 1
    else:
        pass_cns = 0
    return pass_cn, pass_cnp, pass_cns

##---------- CPA parameters ----------##

def ks(Ownship,Contact,teta_os,v_os):  #teta_os changes throught time
    k_0 = Ownship.y ** 2 - 2 * Ownship.y * Contact.y + Contact.y ** 2 + Ownship.x ** 2 - 2 * Ownship.x *
Contact.x + Contact.x ** 2
```

```
    k_1 = 2 * math.cos(math.radians(teta_os)) * v_os * Ownship.y - 2 * math.cos(math.radians(teta_os)) * v_os * Contact.y - 2 * Ownship.y * math.cos(math.radians(Contact.COG)) * Contact.SOG + 2 * math.cos(math.radians(Contact.COG)) * Contact.SOG * Contact.y + 2 * math.sin(math.radians(teta_os)) * v_os * Ownship.x - 2 * math.sin(math.radians(teta_os)) * v_os * Contact.x - 2 * Ownship.x * math.sin(math.radians(Contact.COG)) * Contact.SOG + 2 * math.sin(math.radians(Contact.COG)) * Contact.SOG * Contact.x
    k_2 = (math.cos(math.radians(teta_os))) ** 2 * v_os ** 2 - 2 * math.cos(math.radians(teta_os)) * v_os * math.cos(math.radians(Contact.COG)) * Contact.SOG + (math.cos(math.radians(Contact.COG))) ** 2 * Contact.SOG ** 2 + (math.sin(math.radians(teta_os))) ** 2 * v_os ** 2 - 2 * (math.sin(math.radians(teta_os))) * v_os * math.sin(math.radians(Contact.COG)) * Contact.SOG + (math.sin(math.radians(Contact.COG))) ** 2 * Contact.SOG ** 2
    return k_0,k_1,k_2

def CPA(k_0,k_1,k_2,range_rate):
    if (range_rate >= 0):
        t_cpa = 0
    else:
        t_cpa = - k_1 / ( 2 * k_2 )
    r_tcpa = math.sqrt(k_2 * t_cpa ** 2 + k_1 * t_cpa + k_0)
    return t_cpa,r_tcpa

##---------- Turn boolean ----------##

def turn_func(teta_init,teta):
    if (HeadingRangeConvert360(teta - teta_init) <= 180):
        turn_port = False
        turn_star = True
    else:
        turn_port = True
        turn_star = False
    value = HeadingRangeConvert360(teta - teta_init)
    print value, teta, teta_init, turn_port
    return turn_port,turn_star
```

7. BIBLIOGRAFIE

[1] *L. Grigore, I. Priescu, D.-L. Grecu*, Inteligența Artificială Aplicată în Sisteme Robotizate Fixeși Mobile, Editura AGIR, pp. 703, București, 2020; ISBN: 978-973-72-0767-9.

[2] *B. Tirthankar, L. Sarcione, and F.S. Hover*. "A Simple Reactive Obstacle Avoidance Algorithm and Its Application in Singapore Harbor." Field and Service Robotics. Ed. Andrew Howard, Karl Iagnemma, & Alonzo Kelly. Vol. 62. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010 pp. 455–465, http://dx.doi.org/10.1007/978-3-642-13408-1_41.

[3] *M. Blanke, M. Henriques, J. Bang*, A pre-analysis on autonomous ships, Technical Report, Technical University of Denmark DTU Electro, Elektrovej 326 and DTU Management Engineering, Produktionstorvet 426 DK-2800 Kongens Lyngby, 2017.

[4] *J. Larson, M. Bruch, R. Halterman, J. Rogers, R. Webster*, Advances in Autonomous Obstacle Avoidance for Unmanned Surface Vehicles, Technical Report, Space and Naval Warfare Systems Center, San Diego, 53560 Hull Street, San Diego, CA, 92152, 2007.

[5] *T.A. Johansen, A. Cristofaro, T. Perez*, Ship Collision Avoidance Using Scenario-Based Model Predictive Control, IFAC-International Federation of Automatic Control, Vol. 49, Issue 23, 2016, pp. 014–021, https://doi.org/10.1016/j.ifacol.2016.10.315.

[6] *Y. Kuwata, M.T. Wolf, D. Zarzhitsky, T.L. Huntsberger*, Safe Maritime Navigation with COLREGS Using Velocity Obstacles, IEEE Journal of Oceanic Engineering, Vol. 39, Issue: 1, January 2014, pp. 110-119, https://doi.org/10.1109/JOE.2013.2254214.

[7] *R. Skjente, Ø.N. Smogeli, T.I. Fossen*, A Nonlinear Ship Manoeuvering Model: Identification and adaptive control with experiments for a model ship, Modeling, Identification and Control, vol. 25, No. 1, pp. 3-27, 2004, https://doi.org/10.4173/mic.2004.1.1.

[8] *M. Breivik, V.E. Hovstein, T.I. Fossen*, Straight-Line Target Tracking for Unmanned Surface Vehicles, Modeling, Identification and Control, vol. 29, No. 4, pp. 131-149, 2008, https://doi.org/10.4173/mic.2008.4.2.

[9] *D.B. Lee, E. Tatlicioglu, T.C. Burg, D.M. Dawson*, Adaptive Output Tracking Control of a Surface Vessel, Proceedings of the 47th IEEE Conference on Decision and Control Cancun, Mexico, Dec. 9-11, 2008, https://doi.org/10.1109/CDC.2008.4739313.

[10] *Xunyu Zhong, Xngao Zhong, Xiafu Peng*, Velocity-Change-Space-based dynamic motion planning for mobile robots navigation, Neurocomputing, Vol. 143, 2 November 2014, pp. 153-163, http://doi.org/10.1016/j.neucom.2014.06.010.

[11] *H. Barki, F. Denis, F. Dupont*, A new algorithm for the computation of the Minkowski difference of convex polyhedra, Conference: SMI 2010, Shape Modeling International Conference, Aix en Provence, France, June 21-23 2010, pp. 206-210, http://doi.org/10.1109/SMI.2010.12.