ДОДАТОК А

Приклади

```
import sys
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

class Vehicle():
    def __init__(self):
         self.a_0 = 400
        self.a_1 = 0.1
        self.a_2 = -0.0002
        self.GR = 0.35
        self.r_e = 0.3
        self.J_e = 10
        self.m = 2000
        self.g = 9.81
        self.c_a = 1.36
        self.c_r1 = 0.01
        self.c = 10000
        self.F_max = 10000
        self.x = 0
        self.v = 5
        self.a = 0
        self.w_e = 100
        self.w_e_dot = 0
        self.sample_time = 0.01

    def reset(self):
        self.x = 0
        self.v = 5
        self.a = 0
        self.w_e = 100
        self.w_e_dot = 0

class Vehicle(Vehicle):

  def step(self, throttle, alpha):
     T_e =throttle*(self.a_0+self.a_1*self.w_e+self.a_2*self.w_e**2)
     F_areo = self.c_a*self.v**2
     R_x = self.c_r1*self.v
     F_g = self.m*self.g*np.sin(alpha)
     F_load = F_areo + R_x + F_g
     self.w_e_dot = (T_e - self.GR*self.r_e*F_load) / self.J_e
     w_w = self.GR * self.w_e
     s = (w_w*self.r_e - self.v) / self.v
     if abs(s) < 1: F_x = self.c * s
     else: F_x = self.F_max
     self.a = (F_x - F_load) / self.m
     self.w_e += self.w_e_dot * self.sample_time
     self.v += self.a * self.sample_time
     self.x+=(self.v*self.sample_time)-(0.5*self.a*self.sample_time**2)

time_end = 20
t_data = np.arange(0,time_end,sample_time)
x_data = np.zeros_like(t_data)
v_data = np.zeros_like(t_data)
w_e_data = np.zeros_like(t_data)
model.reset()

def angle(i, alpha, x):
    if x < 60: alpha[i] = np.arctan(3/60)
    elif x < 150: alpha[i] = np.arctan(9/90)
    else: alpha[i] = 0

throttle = np.zeros_like(t_data)
alpha = np.zeros_like(t_data
for i in range(t_data.shape[0]):
    if t_data[i] < 5:
        throttle[i] = 0.2 + ((0.5 - 0.2)/5)*t_data[i]
        angle(i, alpha, model.x)
    elif t_data[i] < 15:
        throttle[i] = 0.5
```

```
        angle(i, alpha, model.x)
    else:
        throttle[i] = ((0 - 0.5)/(20 - 15))*(t_data[i] - 20)
        angle(i, alpha, model.x)
    model.step(throttle[i], alpha[i])
    x_data[i] = model.x
    v_data[i] = model.v
    w_e_data[i] = model.w_e
```

Приклад 3.1 - Реалізація поздовжньої моделі транспортного засобу

```
import pickle
with open('data/data.pickle', 'rb') as f: data = pickle.load(f)
t = data['t']
x_init  = data['x_init']
y_init  = data['y_init']
th_init = data['th_init']
v  = data['v']
om = data['om']
b = data['b']
r = data['r']
l = data['l']
d = data['d']
v_var = 0.01
om_var = 0.01
r_var = 0.01
b_var = 10
Q_km = np.diag([v_var, om_var])
cov_y = np.diag([r_var, b_var])
x_est = np.zeros([len(v), 3])
P_est = np.zeros([len(v), 3, 3])
x_est[0] = np.array([x_init, y_init, th_init])
P_est[0] = np.diag([1, 1, 0.1])

def wraptopi(x):
    if x > np.pi: x = x - (np.floor(x / (2 * np.pi)) + 1) * 2 * np.pi
    elif x < -np.pi: x = x + (np.floor(x/(-2 * np.pi)) + 1) * 2 * np.pi
    return np.array(x)

def measurement_update(lk, rk, bk, P_check, x_check):
  x_k = x_check[0]
  y_k = x_check[1]
  theta_k = wraptopi(x_check[2])
  x_l = lk[0]
  y_l = lk[1]
  d_x = x_l - x_k - d*np.cos(theta_k)
  d_y = y_l - y_k - d*np.sin(theta_k)
  r = np.sqrt(d_x**2 + d_y**2)
  phi = np.arctan2(d_y, d_x) - theta_k
  H_k = np.zeros((2,3))
  H_k[0,0] = -d_x/r
  H_k[0,1] = -d_y/r
  H_k[0,2] = d*(d_x*np.sin(theta_k) - d_y*np.cos(theta_k))/r
  H_k[1,0] = d_y/r**2
  H_k[1,1] = -d_x/r**2
  H_k[1,2] = -1-d*(d_y*np.sin(theta_k) + d_x*np.cos(theta_k))/r**2
  M_k = np.identity(2)
  y_out = np.vstack([r, wraptopi(phi)])
  y_mes = np.vstack([rk, wraptopi(bk)])
  K_k =P_check.dot(H_k.T).dot(np.linalg.inv(H_k.dot(P_check).dot(H_k.T)+M_k.dot(cov_y).dot(M_k.T)))
  x_check = x_check + K_k.dot(y_mes - y_out)
  x_check[2] = wraptopi(x_check[2])
  P_check = (np.identity(3) - K_k.dot(H_k)).dot(P_check)
  return x_check, P_check

P_check = P_est[0]
x_check = x_est[0, :].reshape(3,1)
for k in range(1, len(t)):
```

```
    delta_t = t[k] - t[k -1]
    theta = wraptopi(x_check[2])
    F = np.array([[np.cos(theta), 0], [np.sin(theta), 0], [0, 1]], dtype='float')
    inp = np.array([[v[k-1]], [om[k-1]]])
    x_check = x_check + F.dot(inp).dot(delta_t)
    x_check[2] = wraptopi(x_check[2])
    F_km = np.zeros([3, 3])
    F_km = np.array([[1, 0, -np.sin(theta)*delta_t*v[k-1]], [0, 1, np.cos(theta)*delta_t*v[k-1]],
                     [0, 0, 1]], dtype='float')
    L_km = np.zeros([3, 2])
    L_km = np.array([[np.cos(theta)*delta_t, 0], [np.sin(theta)*delta_t, 0], [0,1]], dtype='float')
    P_check = F_km.dot(P_check.dot(F_km.T)) + L_km.dot(Q_km.dot(L_km.T))
    for i in range(len(r[k])):
      x_check, P_check = measurement_update(l[i], r[k, i], b[k, i], P_check, x_check)
    x_est[k, 0] = x_check[0]
    x_est[k, 1] = x_check[1]
    x_est[k, 2] = x_check[2]
    P_est[k, :, :] = P_check
```

Приклад 3.2 - Реализація моделі руху та моделі оцінки

```
import torch
import torch.nn as nn
import torchvision
from collections import namedtuple
Config = namedtuple('Config', ['iSz', 'oSz', 'gSz'])
default_config = Config(iSz=160, oSz=56, gSz=112)

class Reshape(nn.Module):
    def __init__(self, oSz):
        super(Reshape, self).__init__()
        self.oSz = oSz

    def forward(self, x): return x.permute(0, 2, 3, 1).view(x.shape[0], -1, self.oSz, self.oSz)

class SymmetricPad2d(nn.Module):
    def __init__(self, padding):
        super(SymmetricPad2d, self).__init__()
        self.padding = padding
        try: self.pad_l, self.pad_b, self.pad_r, self.pad_t = padding
        except: self.pad_l,self.pad_b,self.pad_r, self.pad_t = [padding,]*4

    def forward(self, input):
        h = input.shape[2] + self.pad_t + self.pad_b
        w = input.shape[3] + self.pad_l + self.pad_r
        output = torch.zeros(input.shape[0], input.shape[1], h, w).to(input.device)
        c_input = input
        if self.pad_t < 0: c_input = c_input.narrow(2, -self.pad_t, c_input.shape[2] + self.pad_t)
        if self.pad_b < 0: c_input = c_input.narrow(2,0,c_input.shape[2] + self.pad_b)
        if self.pad_l < 0: c_input = c_input.narrow(3, -self.pad_l, c_input.shape[3] + self.pad_l)
        if self.pad_r < 0: c_input = c_input.narrow(3,0,c_input.shape[3] + self.pad_r)
        c_output = output
        if self.pad_t>0:c_output=c_output.narrow(2, self.pad_t, c_output.shape[2] - self.pad_t)
        if self.pad_b > 0: c_output = c_output.narrow(2, 0, c_output.shape[2] - self.pad_b)
        if self.pad_l > 0:c_output = c_output.narrow(3, self.pad_l, c_output.shape[3] - self.pad_l)
        if self.pad_r > 0: c_output = c_output.narrow(3, 0, c_output.shape[3] - self.pad_r)
        c_output.copy_(c_input)
        for i in range(self.pad_t):
            output.narrow(2, self.pad_t-i-1, 1).copy_(output.narrow(2, self.pad_t+i, 1))
        for i in range(self.pad_b):
            output.narrow(2, output.shape[2] - self.pad_b + i, 1).copy_(
                output.narrow(2, output.shape[2] - self.pad_b - i-1, 1))
        for i in range(self.pad_l):
            output.narrow(3, self.pad_l-i-1, 1).copy_(output.narrow(3, self.pad_l+i, 1))
        for i in range(self.pad_r):
            output.narrow(3, output.shape[3] - self.pad_r + i, 1).copy_(
                output.narrow(3, output.shape[3] - self.pad_r - i-1, 1))
```

```
        return output

def updatePadding(net, nn_padding):
    typename = torch.typename(net)
    if typename.find('Sequential') >= 0 or typename.find('Bottleneck') >= 0:
        modules_keys = list(net._modules.keys())
        for i in reversed(range(len(modules_keys))):
            subnet = net._modules[modules_keys[i]]
            out = updatePadding(subnet, nn_padding)
            if out != -1:
                p = out
                in_c, out_c, k, s, _, d, g, b = subnet.in_channels, subnet.out_channels, \
                    subnet.kernel_size[0],subnet.stride[0],subnet.padding[0], subnet.dilation[0], \
                    subnet.groups, subnet.bias,
                conv_temple = nn.Conv2d(in_c,out_c,k,stride=s,padding=0,dilation=d,groups=g,bias=b)
                conv_temple.weight = subnet.weight
                conv_temple.bias = subnet.bias
                if p > 1:net._modules[modules_keys[i]]=nn.Sequential(SymmetricPad2d(p),conv_temple)
                else:net._modules[modules_keys[i]] = nn.Sequential(nn_padding(p), conv_temple)
    else:
        if typename.find('torch.nn.modules.conv.Conv2d') >= 0:
            k_sz, p_sz = net.kernel_size[0], net.padding[0]
            if ((k_sz == 3) or (k_sz == 7)) and p_sz != 0: return p_sz
    return -1

class DeepMask(nn.Module):
    def __init__(self, config=default_config, context=True):
        super(DeepMask, self).__init__()
        self.config = config
        self.context = context
        self.strides = 16
        self.fSz = -(-self.config.iSz // self.strides)
        self.trunk = self.creatTrunk()
        updatePadding(self.trunk, nn.ReflectionPad2d)
        self.crop_trick = nn.ZeroPad2d(-16//self.strides)
        self.maskBranch = self.createMaskBranch()
        self.scoreBranch = self.createScoreBranch()
        npt = sum(p.numel() for p in self.trunk.parameters()) / 1e+06
        npm = sum(p.numel() for p in self.maskBranch.parameters()) / 1e+06
        nps = sum(p.numel() for p in self.scoreBranch.parameters()) / 1e+06

    def forward(self, x):
        feat = self.trunk(x)
        if self.context: feat = self.crop_trick(feat)
        mask = self.maskBranch(feat)
        score = self.scoreBranch(feat)
        return mask, score

    def creatTrunk(self):
        resnet50 = torchvision.models.resnet50(pretrained=True)
        trunk1 = nn.Sequential(*list(resnet50.children())[:-3])
        trunk2 = nn.Sequential(nn.Conv2d(1024, 128, 1), nn.ReLU(inplace=True),
            nn.Conv2d(128, 512, self.fSz))
        return nn.Sequential(trunk1, trunk2)

    def createMaskBranch(self):
        maskBranch = nn.Sequential(nn.Conv2d(512, self.config.oSz**2, 1),Reshape(self.config.oSz),)
        if self.config.gSz > self.config.oSz:
           upSample = nn.UpsamplingBilinear2d(size=[self.config.gSz, self.config.gSz])
           maskBranch = nn.Sequential(maskBranch, upSample)
        return maskBranch

    def createScoreBranch(self):
        scoreBranch = nn.Sequential(nn.Dropout(0.5),  nn.Conv2d(512, 1024, 1),  nn.Threshold(0, 1e-6),
nn.Dropout(0.5), nn.Conv2d(1024, 1, 1),)
        return scoreBranch

if __name__ == '__main__':
    a = SymmetricPad2d(3)
    x = torch.tensor([[[[1, 2, 3], [4, 5, 6], [7, 8, 9]]]])
    out = a(x)
```

```
    print(out)
    import torch
    Config = namedtuple('Config', ['iSz', 'oSz', 'gSz'])
    config = Config(iSz=160, oSz=56, gSz=112)
    model = DeepMask(config).cuda()
    x = torch.rand(32, 3, config.iSz+32, config.iSz+32).cuda()
    pred_mask, pred_cls = model(x)
    model.context = False  # really important!!
    input_size = config.iSz + model.strides * 16 + (model.context * 32)
    x = torch.rand(8, 3, input_size, input_size).cuda()
    pred_mask, pred_cls = model(x)

import os
import torch.nn as nn
from collections import namedtuple
from models.DeepMask import DeepMask
from utils.load_helper import load_pretrain
Config = namedtuple('Config', ['iSz', 'oSz', 'gSz', 'km', 'ks'])
default_config = Config(iSz=160, oSz=56, gSz=160, km=32, ks=32)

class RefineModule(nn.Module):
    def __init__(self, l1, l2, l3):
        super(RefineModule, self).__init__()
        self.layer1 = l1
        self.layer2 = l2
        self.layer3 = l3

    def forward(self, x):
        x1 = self.layer1(x[0])
        x2 = self.layer2(x[1])
        y = x1 + x2
        y = self.layer3(y)
        return y

class SharpMask(nn.Module):
    def __init__(self, config=default_config, context=True):
        super(SharpMask, self).__init__()
        self.context = context  # with context
        self.km, self.ks = config.km, config.ks
        self.skpos = [6, 5, 4, 2]
        deepmask = DeepMask(config)
        deeomask_resume = os.path.join('exps', 'deepmask', 'train', 'model_best.pth.tar')
        deepmask = load_pretrain(deepmask, deeomask_resume)
        self.trunk = deepmask.trunk
        self.crop_trick = deepmask.crop_trick
        self.scoreBranch = deepmask.scoreBranch
        self.maskBranchDM = deepmask.maskBranch
        self.fSz = deepmask.fSz
        self.refs = self.createTopDownRefinement()
        nph = sum(p.numel() for h in self.neths for p in h.parameters()) / 1e+06
        npv = sum(p.numel() for h in self.netvs for p in h.parameters()) / 1e+06

    def forward(self, x):
        inps = list()
        for i, l in enumerate(self.trunk.children()):
            for j, ll in enumerate(l.children()):
                x = ll(x)
                if i == 0 and j == (len(l)-1) and self.context: x = self.crop_trick(x)
                if i == 0 and j in self.skpos: inps.append(x)
        currentOutput = self.refs[0](x)
        for k in range(len(self.refs)-2):
            x_f = inps[-(k+1)]
            currentOutput = self.refs[k+1]((x_f, currentOutput))
        currentOutput = self.refs[-1](currentOutput)
        return currentOutput, self.scoreBranch(x)

    def train(self, mode=True):
        self.training = mode
        if mode:
            for module in self.children():module.train(False)
            for module in self.refs.children():module.train(mode)
```

```
        else:
            for module in self.children():module.train(mode)
        return self

    def createHorizontal(self):
        neths = nn.ModuleList()
        nhu1, nhu2, crop = 0, 0, 0
        for i in range(len(self.skpos)):
            h = []
            nInps = self.ks // 2 ** i
            if i == 0: nhu1, nhu2, crop = 1024, 64, 0 if self.context else 0
            elif i == 1: nhu1, nhu2, crop = 512, 64, -2 if self.context else 0
            elif i == 2: nhu1, nhu2, crop = 256, 64, -4 if self.context else 0
            elif i == 3: nhu1, nhu2, crop = 64, 64, -8 if self.context else 0
            if crop != 0: h.append(nn.ZeroPad2d(crop))
            h.append(nn.ReflectionPad2d(1))
            h.append(nn.Conv2d(nhu1, nhu2, 3))
            h.append(nn.ReLU(inplace=True))
            h.append(nn.ReflectionPad2d(1))
            h.append(nn.Conv2d(nhu2, nInps, 3))
            h.append(nn.ReLU(inplace=True))
            h.append(nn.ReflectionPad2d(1))
            h.append(nn.Conv2d(nInps, nInps // 2, 3))
            neths.append(nn.Sequential(*h))
        return neths

    def createVertical(self):
        netvs = nn.ModuleList()
        netvs.append(nn.ConvTranspose2d(512, self.km, self.fSz))
        for i in range(len(self.skpos)):
            netv = []
            nInps = self.km // 2 ** i
            netv.append(nn.ReflectionPad2d(1))
            netv.append(nn.Conv2d(nInps, nInps, 3))
            netv.append(nn.ReLU(inplace=True))
            netv.append(nn.ReflectionPad2d(1))
            netv.append(nn.Conv2d(nInps, nInps // 2, 3))
            netvs.append(nn.Sequential(*netv))
        return netvs

    def refinement(self, neth, netv):
        return RefineModule(neth, netv, nn.Sequential(nn.ReLU(inplace=True),
                            nn.UpsamplingNearest2d(scale_factor=2)))

    def createTopDownRefinement(self):
        self.neths = self.createHorizontal()
        self.netvs = self.createVertical()
        refs = nn.ModuleList()
        refs.append(self.netvs[0])
        for i in range(len(self.skpos)):
            refs.append(self.refinement(self.neths[i], self.netvs[i+1]))
        refs.append(nn.Sequential(nn.ReflectionPad2d(1),nn.Conv2d(self.km//2**(len(refs)-1), 1,3)))
        return refs

if __name__ == '__main__':
    import torch
    Config = namedtuple('Config', ['iSz', 'oSz', 'gSz', 'km', 'ks'])
    config = Config(iSz=160, oSz=56, gSz=160, km=32, ks=32)
    model = SharpMask(config).cuda()
    x = torch.rand(32, 3, config.iSz+32, config.iSz+32).cuda()
    pred_mask = model(x, True)
```

Приклад 3.3 - Реалізація нейроних мереж

```
import cv2
from matplotlib import pyplot as plt
from matplotlib import patches
```

```
%matplotlib inline
%load_ext autoreload
%autoreload 2
%precision %.2f
import files_management
img_left = files_management.read_left_image()
img_right = files_management.read_right_image()
p_left, p_right = files_management.get_projection_matrices()
np.set_printoptions(suppress=True)

def compute_left_disparity_map(img_left, img_right):
    img_left = cv2.cvtColor(img_left, cv2.COLOR_BGR2GRAY)
    img_right = cv2.cvtColor(img_right, cv2.COLOR_BGR2GRAY)
    stereoProcessor = cv2.StereoSGBM_create(minDisparity=0, numDisparities=16*6, blockSize=11, P1=8
*3*7**2, P2=32*3*7**2, uniquenessRatio=0)
    disp_left = stereoProcessor.compute(img_left, img_right).astype(np.float32)/16
    return disp_left

disp_left = compute_left_disparity_map(img_left, img_right)

def decompose_projection_matrix(p):
    cameraMatrix, rotMatrix, transVect, rotMatrixX, rotMatrixY, rotMatrixZ, eulerAngles = cv2.decom
poseProjectionMatrix(p)
    k = cameraMatrix
    r = rotMatrix
    t = transVect / transVect[3]
    return k, r, t
k_left, r_left, t_left = decompose_projection_matrix(p_left)
k_right, r_right, t_right = decompose_projection_matrix(p_right)

def calc_depth_map(disp_left, k_left, t_left, t_right):
    focal_length = k_left[0,0]
    baseline = t_left[1] - t_right[1]
    disp_left[disp_left==0] = 0.9
    disp_left[disp_left==-1] = 0.9
    depth_map = np.ones(disp_left.shape, np.single)
    depth_map[:] = (focal_length*baseline) / disp_left[:]
    return depth_map
depth_map_left = calc_depth_map(disp_left, k_left, t_left, t_right)
obstacle_image = files_management.get_obstacle_image()

def locate_obstacle_in_image(image, obstacle_image):

    cross_corr_map = cv2.matchTemplate(image, obstacle_image, method=cv2.TM_CCORR_NORMED)
    minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(cross_corr_map)
    obstacle_location = maxLoc
    return cross_corr_map, obstacle_location

cross_corr_map, obstacle_location = locate_obstacle_in_image(img_left, obstacle_image)

def calculate_nearest_point(depth_map, obstacle_location, obstacle_img):
 obstacle_min_x_pos = obstacle_location[1]
 obstacle_min_y_pos = obstacle_location[0]
 obstacle_width = obstacle_img.shape[0]
 obstacle_height = obstacle_img.shape[1]
 obstacle_max_x_pos = obstacle_location[1] + obstacle_width
 obstacle_max_y_pos = obstacle_location[0] + obstacle_height
 obstacle_depth=depth_map[obstacle_min_x_pos:obstacle_max_x_pos,obstacle_min_y_pos:obstacle_max_y_p
os]
 closest_point_depth = obstacle_depth.min()
 obstacle_bbox = patches.Rectangle((obstacle_min_y_pos, obstacle_min_x_pos), obstacle_height, obsta
cle_width, linewidth=1, edgecolor='r', facecolor='none')
 return closest_point_depth, obstacle_bbox

closest_point_depth, obstacle_bbox = calculate_nearest_point(depth_map_left, obstacle_location, obs
tacle_image)
```

Приклад 3.4 - Реалізація моделі розрахунку глибини

```
from m2bk import *
np.random.seed(1)
np.set_printoptions(threshold=np.nan)
dataset_handler = DatasetHandler()
v, u = depth.shape
depth_val = depth[v-1, u-1]

def extract_features(image):
    surf = cv2.xfeatures2d.SURF_create(500)
    kp, des = surf.detectAndCompute(image,None)
    return kp, des

def extract_features_dataset(images, extract_features_function):
    kp_list = []
    des_list = []
    for img in images:
        kp, des = extract_features(img)
        kp_list.append(kp)
        des_list.append(des)
    return kp_list, des_list

def match_features(des1, des2):
    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params,search_params)
    match = flann.knnMatch(des1,des2,k=2)
    return match

def filter_matches_distance(match, dist_threshold):
    filtered_match = []
    for i,(m,n) in enumerate(match):
        if m.distance < dist_threshold*n.distance:
            filtered_match.append(m)
    return filtered_match

def match_features_dataset(des_list, match_features):
    matches = []
    for i in range(len(des_list)-1):
        descriptor1 = des_list[i]
        descriptor2 = des_list[i+1]
        match = match_features(descriptor1, descriptor2)
        matches.append(match)
    return matches

def filter_matches_dataset(filter_matches_distance, matches, dist_threshold):
    filtered_matches = []
    for m in matches:
        new_match = filter_matches_distance(m, dist_threshold)
        filtered_matches.append(new_match)
    return filtered_matches

dist_threshold = 0.6
filtered_matches = filter_matches_dataset(filter_matches_distance, matches, dist_threshold)
if len(filtered_matches) > 0:
    is_main_filtered_m = True
    if is_main_filtered_m: matches = filtered_matches
    i = 0

def estimate_motion(match, kp1, kp2, k):
    rmat = np.eye(3)
    tvec = np.zeros((3, 1))
    image1_points = []
    image2_points = []
    for m in match:
        train_idx = m.trainIdx
        query_idx = m.queryIdx
        p1x, p1y = kp1[query_idx].pt
        image1_points.append([p1x, p1y])
```

```
        p2x, p2y = kp2[train_idx].pt
        image2_points.append([p2x, p2y])
    E, mask = cv2.findEssentialMat(np.array(image1_points), np.array(image2_points), k)
    retval, rmat, tvec, mask =cv2.recoverPose(E,np.array(image1_points),np.array(image2_points), k)
    return rmat, tvec, image1_points, image2_points

def estimate_trajectory(estimate_motion, matches, kp_list, k, depth_maps=[]):

    trajectory = [np.array([0, 0, 0])]
    R = np.diag([1,1,1])
    T = np.zeros([3, 1])
    RT = np.hstack([R, T])
    RT = np.vstack([RT, np.zeros([1, 4])])
    RT[-1, -1] = 1
    for i in range(len(matches)):
        match = matches[i]
        kp1 = kp_list[i]
        kp2 = kp_list[i+1]
        depth = depth_maps[i]
        rmat, tvec, image1_points, image2_points = estimate_motion(match, kp1, kp2, k, depth)
        rt_mtx = np.hstack([rmat, tvec])
        rt_mtx = np.vstack([rt_mtx, np.zeros([1, 4])])
        rt_mtx[-1, -1] = 1
        rt_mtx_inv = np.linalg.inv(rt_mtx)
        RT = np.dot(RT, rt_mtx_inv)
        new_trajectory = RT[:3, 3]
        trajectory.append(new_trajectory)
    trajectory = np.array(trajectory).T
    return trajectory
```

Приклад 3.5 - Реалізація моделі розрахунку траєкторії

```
from m6bk import *
np.set_printoptions(precision=2, threshold=np.nan)
image = dataset_handler.image
k = dataset_handler.k
depth = dataset_handler.depth
segmentation = dataset_handler.segmentation
object_detection_output = dataset_handler.object_detection
colored_segmentation = dataset_handler.vis_segmentation(segmentation)

def xy_from_depth(depth, k):
    H, W = np.shape(depth)
    f = k[0, 0]
    c_u = k[0, 2]
    c_v = k[1, 2]
    x = np.zeros((H, W))
    y = np.zeros((H, W))
    for i in range(H):
        for j in range(W):
            x[i, j] = ((j+1 - c_u)*depth[i, j]) / f
            y[i, j] = ((i+1 - c_v)*depth[i, j]) / f
    return x, y

z = dataset_handler.depth
x, y = xy_from_depth(z, k)
road_mask = np.zeros(segmentation.shape)
road_mask[segmentation == 7] = 1
plt.imshow(road_mask)
x_ground = x[road_mask == 1]
y_ground = y[road_mask == 1]
z_ground = dataset_handler.depth[road_mask == 1]
xyz_ground = np.stack((x_ground, y_ground, z_ground))

def ransac_plane_fit(xyz_data):
    num_itr = 100
    min_num_inliers = xyz_data.shape[1] / 2
    distance_threshold = 0.01
```

```
    largest_number_of_inliers = 0
    largest_inlier_set_indexes = 0

    for i in range(num_itr):
        indexes = np.random.choice(xyz_data.shape[1], 3, replace=False)
        pts = xyz_data[:, indexes]
        p = compute_plane(pts)
        distance = dist_to_plane(p, xyz_data[0, :].T, xyz_data[1, :].T, xyz_data[2, :].T)
        number_of_inliers = len(distance[distance > distance_threshold])
        if number_of_inliers > largest_number_of_inliers:
            largest_number_of_inliers = number_of_inliers
            largest_inlier_set_indexes = np.where(distance < distance_threshold)[0]
        if (number_of_inliers > min_num_inliers): break
    output_plane = compute_plane(xyz_data[:, largest_inlier_set_indexes])
    return output_plane
p_final = ransac_plane_fit(xyz_ground)
dist = np.abs(dist_to_plane(p_final, x, y, z))
ground_mask = np.zeros(dist.shape)
ground_mask[dist < 0.1] = 1
ground_mask[dist > 0.1] = 0

def estimate_lane_lines(segmentation_output):
  lane_boundary_mask = np.zeros(segmentation_output.shape).astype(np.uint8)
  lane_boundary_mask[segmentation_output==6] = 255
  lane_boundary_mask[segmentation_output==8] = 255
  edges = cv2.Canny(lane_boundary_mask, 100, 150)
  lines=cv2.HoughLinesP(edges,rho=10,theta=np.pi/180,threshold=200,minLineLength=150,maxLineGap=50)
  lines = lines.reshape((-1, 4))
  return lines

lane_lines = estimate_lane_lines(segmentation)

def merge_lane_lines(lines):
    slope_similarity_threshold = 0.1
    intercept_similarity_threshold = 40
    min_slope_threshold = 0.3
    clusters = []
    current_inds = []
    itr = 0
    slopes, intercepts = get_slope_intecept(lines)
    slopes_horizontal = np.abs(slopes) > min_slope_threshold
    for slope, intercept in zip(slopes, intercepts):
     in_clusters=np.array([itr in current for current in current_inds])
     if not in_clusters.any():
        slope_cluster = np.logical_and(slopes < (slope+slope_similarity_threshold), slopes > (slope
-slope_similarity_threshold))
        intercept_cluster = np.logical_and(intercepts < (intercept+intercept_similarity_threshold),
 intercepts > (intercept-intercept_similarity_threshold))
        inds = np.argwhere(slope_cluster & intercept_cluster & slopes_horizontal).T
        if inds.size:
            current_inds.append(inds.flatten())
            clusters.append(lines[inds])
     itr += 1
    merged_lines = [np.mean(cluster, axis=1) for cluster in clusters]
    merged_lines = np.array(merged_lines).reshape((-1, 4))
    return merged_lines

merged_lane_lines = merge_lane_lines(lane_lines)
max_y = dataset_handler.image.shape[0]
min_y = np.min(np.argwhere(road_mask == 1)[:, 0])
extrapolated_lanes = extrapolate_lines(merged_lane_lines, max_y, min_y)
final_lanes = find_closest_lines(extrapolated_lanes, dataset_handler.lane_midpoint)
detections = dataset_handler.object_detection

def filter_detections_by_segmentation(detections, segmentation_output):
    ratio_threshold = 0.3
    filtered_detections = []
    for detection in detections:
        class_name, x_min, y_min, x_max, y_max, score = detection
        x_min = int(float(x_min))
        y_min = int(float(y_min))
```

```
        x_max = int(float(x_max))
        y_max = int(float(y_max))
        box_area = (x_max-x_min) * (y_max-y_min)
        if class_name == 'Car':
            class_index = 10
        elif class_name == 'Pedestrian':
            class_index = 4
        correct_pixels=len(np.where(segmentation_output[y_min:y_max,x_min:x_max]== class_index)[0])
        ratio = correct_pixels / box_area
        if ratio > ratio_threshold: filtered_detections.append(detection)
    return filtered_detections

filtered_detections = filter_detections_by_segmentation(detections, segmentation)

def find_min_distance_to_detection(detections, x, y, z):
    min_distances = []
    for detection in detections:
        class_name, x_min, y_min, x_max, y_max, score = detection
        x_min = int(float(x_min))
        y_min = int(float(y_min))
        x_max = int(float(x_max))
        y_max = int(float(y_max))
        box_x = x[y_min:y_max, x_min:x_max]
        box_y = y[y_min:y_max, x_min:x_max]
        box_z = z[y_min:y_max, x_min:x_max]
        box_distances = np.sqrt(box_x**2 + box_y**2 + box_z**2)
        min_distances.append(np.min(box_distances))
    return min_distances
```

Приклад 3.6 - Реалізація моделі сприйняття довкілля

```
import math
import matplotlib.animation as anim
from IPython.display import HTML

def inverse_scanner(num_rows, num_cols, x, y, theta, meas_phi, meas_r, rmax, alpha, beta):
    m = np.zeros((M, N))
    for i in range(num_rows):
        for j in range(num_cols):
            r = math.sqrt((i - x)**2 + (j - y)**2)
            phi = (math.atan2(j - y, i - x) - theta + math.pi) % (2 * math.pi) - math.pi
            k = np.argmin(np.abs(np.subtract(phi, meas_phi)))
            if (r > min(rmax, meas_r[k]+alpha/2.0))or(abs(phi-meas_phi[k])>beta/2.0): m[i, j] = 0.5
            elif (meas_r[k]<rmax) and (abs(r-meas_r[k]) < alpha / 2.0): m[i, j] = 0.7
            elif r < meas_r[k]: m[i, j] = 0.3
    return m

def get_ranges(true_map, X, meas_phi, rmax):
    (M, N) = np.shape(true_map)
    x = X[0]
    y = X[1]
    theta = X[2]
    meas_r = rmax * np.ones(meas_phi.shape)
    for i in range(len(meas_phi)):
        for r in range(1, rmax+1):
            xi = int(round(x + r * math.cos(theta + meas_phi[i])))
            yi = int(round(y + r * math.sin(theta + meas_phi[i])))
            if (xi <= 0 or xi >= M-1 or yi <= 0 or yi >= N-1):
                meas_r[i] = r
                break
            elif true_map[int(round(xi)), int(round(yi))] == 1:
                meas_r[i] = r
                break
    return meas_r

T_MAX = 150
time_steps = np.arange(T_MAX)
x_0 = [30, 30, 0]
```

```
u = np.array([[3, 0, -3, 0], [0, 3, 0, -3]])
u_i = 1
w = np.multiply(0.3, np.ones(len(time_steps)))
M = 50
N = 60
true_map = np.zeros((M, N))
true_map[0:10, 0:10] = 1
true_map[30:35, 40:45] = 1
true_map[3:6,40:60] = 1;
true_map[20:30,25:29] = 1;
true_map[40:50,5:25] = 1;
m = np.multiply(0.5, np.ones((M, N)))
L0 = np.log(np.divide(m, np.subtract(1, m)))
L = L0
meas_phi = np.arange(-0.4, 0.4, 0.05)
rmax = 30
alpha = 1
beta = 0.05
x = np.zeros((3, len(time_steps)))
x[:, 0] = x_0
map_fig = plt.figure()
map_ax = map_fig.add_subplot(111)
map_ax.set_xlim(0, N)
map_ax.set_ylim(0, M)
invmod_fig = plt.figure()
invmod_ax = invmod_fig.add_subplot(111)
invmod_ax.set_xlim(0, N)
invmod_ax.set_ylim(0, M)
belief_fig = plt.figure()
belief_ax = belief_fig.add_subplot(111)
belief_ax.set_xlim(0, N)
belief_ax.set_ylim(0, M)
meas_rs = []
meas_r = get_ranges(true_map, x[:, 0], meas_phi, rmax)
meas_rs.append(meas_r)
invmods = []
invmod = inverse_scanner(M, N, x[0, 0], x[1, 0], x[2, 0], meas_phi, meas_r, rmax, alpha, beta)
invmods.append(invmod)
ms = []
ms.append(m)
for t in range(1, len(time_steps)):
    move = np.add(x[0:2, t-1], u[:, u_i])
    if (move[0]>=M-1)or(move[1]>=N-
1)or(move[0] <= 0) or (move[1] <= 0) or true_map[int(round(move[0])), int(round(move[1]))] == 1:
        x[:, t] = x[:, t-1]
        u_i = (u_i + 1) % 4
    else: x[0:2, t] = move
    x[2, t] = (x[2, t-1] + w[t]) % (2 * math.pi)
    meas_r = get_ranges(true_map, x[:, t], meas_phi, rmax)
    meas_rs.append(meas_r)
    invmod = inverse_scanner(M, N, x[0, t], x[1, t], x[2, t], meas_phi, meas_r, rmax, alpha, beta)
    invmods.append(invmod)
    L = np.log(np.divide(invmod, np.subtract(1, invmod))) + L - L0
    m = (np.exp(L)) / (1 + np.exp(L))
    ms.append(m)

def map_update(i):
    map_ax.clear()
    map_ax.set_xlim(0, N)
    map_ax.set_ylim(0, M)
    map_ax.imshow(np.subtract(1, true_map), cmap='gray', origin='lower', vmin=0.0, vmax=1.0)
    x_plot = x[1, :i+1]
    y_plot = x[0, :i+1]
    map_ax.plot(x_plot, y_plot, "bx-")

def invmod_update(i):
    invmod_ax.clear()
    invmod_ax.set_xlim(0, N)
    invmod_ax.set_ylim(0, M)
    invmod_ax.imshow(invmods[i], cmap='gray', origin='lower', vmin=0.0, vmax=1.0)
    for j in range(len(meas_rs[i])):
```

```
        invmod_ax.plot(x[1, i] + meas_rs[i][j] * math.sin(meas_phi[j] + x[2, i]), x[0, i] + meas_rs
[i][j] * math.cos(meas_phi[j] + x[2, i]), "ko")
    invmod_ax.plot(x[1, i], x[0, i], 'bx')

def belief_update(i):
    belief_ax.clear()
    belief_ax.set_xlim(0, N)
    belief_ax.set_ylim(0, M)
    belief_ax.imshow(ms[i],cmap='gray',origin='lower',vmin=0,vmax=1.0)
    belief_ax.plot(x[1, max(0, i-10):i], x[0, max(0, i-10):i], 'bx-')
```

Приклад 3.7 - Реалізація моделі побудови карти навколишнього середовища

```
import osmnx as ox
import networkx as nx
import queue
import priority_dict

def dijkstras_search(origin_key, goal_key, graph):
    open_queue = priority_dict.priority_dict({})
    closed_dict = {}
    predecessors = {}
    open_queue[origin_key] = 0.0
    goal_found = False
    while (open_queue):
        u, ucost = open_queue.pop_smallest()
        if u == goal_key:
            goal_found = True
            break
        for edge_dict in graph.out_edges([u], data=True):
            v = edge_dict[1]
            if v in closed_dict: continue
            uvcost = edge_dict[2]['length']
            if v not in open_queue:
                open_queue[v] = ucost + uvcost
                predecessors[v] = u
            else:
                vcost = open_queue[v]
                if ucost + uvcost < vcost:
                    open_queue[v] = ucost + uvcost
                    predecessors[v] = u
        closed_dict[u] = 1
    if not goal_found: raise ValueError("Goal not found in search.")
    return get_path(origin_key, goal_key, predecessors)

def get_path(origin_key, goal_key, predecessors):
    key = goal_key
    path = [goal_key]
    while (key != origin_key):
        key = predecessors[key]
        path.insert(0, key)
    return path

def distance_heuristic(state_key, goal_key, node_data):
    n1 = node_data[state_key]
    n2 = node_data[goal_key]
    long1 = n1['x']*math.pi/180.0
    lat1 = n1['y']*math.pi/180.0
    long2 = n2['x']*math.pi/180.0
    lat2 = n2['y']*math.pi/180.0
    r = 6371000
    x1 = r*math.cos(lat1)*math.cos(long1)
    y1 = r*math.cos(lat1)*math.sin(long1)
    z1 = r*math.sin(lat1)
    x2 = r*math.cos(lat2)*math.cos(long2)
    y2 = r*math.cos(lat2)*math.sin(long2)
    z2 = r*math.sin(lat2)
    d = ((x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2)**0.5
    return d
```

```
def a_star_search(origin_key, goal_key, graph):
    open_queue = priority_dict.priority_dict({})
    closed_dict = {}
    predecessors = {}
    costs = {}
    node_data = graph.nodes(True)
    costs[origin_key] = 0.0
    open_queue[origin_key] = distance_heuristic(origin_key, goal_key, node_data)
    goal_found = False
    while (open_queue):
        u, uheuristic = open_queue.pop_smallest()
        ucost = costs[u]
        if u == goal_key:
            goal_found = True
            break
        for edge_dict in graph.out_edges([u], data=True):
            v = edge_dict[1]
            if v in closed_dict: continue
            uvcost = edge_dict[2]['length']
            if v not in open_queue:
                costs[v] = ucost + uvcost
                open_queue[v] = ucost + uvcost + distance_heuristic(v, goal_key, node_data)
                predecessors[v] = u
            else:
                vcost = costs[v]
                if ucost + uvcost < vcost:
                    costs[v] = ucost + uvcost
                    open_queue[v] = ucost + uvcost + distance_heuristic(v, goal_key, node_data)
                    predecessors[v] = u
        closed_dict[u] = 1
    if not goal_found: raise ValueError("Goal not found in search.")
    return get_path(origin_key, goal_key, predecessors)
```

Приклад 3.8 - Реалізація моделі побудови маршруту

```
class CUtils(object):
    def __init__(self):
        pass

    def create_var(self, var_name, value):
        if not var_name in self.__dict__:
            self.__dict__[var_name] = value
import numpy as np
import scipy.spatial
import matplotlib.pyplot as plt
from math import sin, cos, pi, sqrt

class CollisionChecker:
    def __init__(self, circle_offsets, circle_radii, weight):
        self._circle_offsets = np.array(circle_offsets)
        self._circle_radii   = circle_radii
        self._weight         = weight

    def collision_check(self, paths, obstacles):
        collision_check_array = np.zeros(len(paths), dtype=bool)
        for i in range(len(paths)):
            collision_free = True
            path           = paths[i]
            for j in range(len(path[0])):
                circle_locations = np.zeros((len(self._circle_offsets), 2))
                circle_locations[:, 0] = path[0][j] + self._circle_offsets * cos(path[2][j])
                circle_locations[:, 1] = path[1][j] + self._circle_offsets * sin(path[2][j])
                for k in range(len(obstacles)):
                    collision_dists = scipy.spatial.distance.cdist(obstacles[k], circle_locations)
                    collision_dists = np.subtract(collision_dists, self._circle_radii)
                    collision_free = collision_free and not np.any(collision_dists < 0)
                    if not collision_free:break
```

```
                if not collision_free:break
            collision_check_array[i] = collision_free
        return collision_check_array

    def select_best_path_index(self, paths, collision_check_array, goal_state):
        best_index = None
        best_score = float('Inf')
        for i in range(len(paths)):
            if collision_check_array[i]:
                score = sqrt((paths[i][1][-1] - goal_state[1])**2 + (paths[i][0][-1] -
 goal_state[0])**2)
                for j in range(len(paths)):
                    if j == i: continue
                    else:
                        if not collision_check_array[j]:
                            score += self._weight * sqrt((paths[i][1][-1] - paths[j][1][-
1])**2 + (paths[i][0][-1] - paths[j][0][-1])**2)
            else:score = float('Inf')
            if score < best_score:
                best_score = score
                best_index = i
        return best_index
import numpy as np
import math
FOLLOW_LANE = 0
DECELERATE_TO_STOP = 1
STAY_STOPPED = 2
STOP_THRESHOLD = 0.02
STOP_COUNTS = 10

class BehaviouralPlanner:
    def __init__(self, lookahead, stopsign_fences, lead_vehicle_lookahead):
        self._lookahead                     = lookahead
        self._stopsign_fences               = stopsign_fences
        self._follow_lead_vehicle_lookahead = lead_vehicle_lookahead
        self._state                         = FOLLOW_LANE
        self._follow_lead_vehicle           = False
        self._goal_state                    = [0.0, 0.0, 0.0]
        self._goal_index                    = 0
        self._stop_count                    = 0

    def set_lookahead(self, lookahead):
        self._lookahead = lookahead

    def transition_state(self, waypoints, ego_state, closed_loop_speed):
        if self._state == FOLLOW_LANE:
          closest_len, closest_index = get_closest_index(waypoints, ego_state)
          goal_index = self.get_goal_index(waypoints, ego_state, closest_len, closest_index)
          new_index, stop_sign_found =self.check_for_stop_signs(waypoints,closest_index,goal_index)
          self._goal_index = new_index if stop_sign_found else goal_index
          self._goal_state = waypoints[self._goal_index]
          if stop_sign_found:
              self._goal_state[2] = 0
              self._state = DECELERATE_TO_STOP
        elif self._state == DECELERATE_TO_STOP:
            self._state = STAY_STOPPED if closed_loop_speed<STOP_THRESHOLD else DECELERATE_TO_STOP
        elif self._state == STAY_STOPPED:
            if self._stop_count == STOP_COUNTS:
                closest_len, closest_index = get_closest_index(waypoints, ego_state)
                goal_index = self.get_goal_index(waypoints, ego_state, closest_len, closest_index)
                _, stop_sign_found = self.check_for_stop_signs(waypoints,closest_index, goal_index)
                self._goal_index = goal_index
                self._goal_state = waypoints[goal_index]
                if not stop_sign_found:
                    self._stop_count = 0
                    self._state = FOLLOW_LANE
            else: self._stop_count += 1
        else:raise ValueError('Invalid state value.')

    def get_goal_index(self, waypoints, ego_state, closest_len, closest_index):
        arc_length = closest_len
```

```
        wp_index = closest_index
        if arc_length > self._lookahead:return wp_index
        if wp_index == len(waypoints) - 1:return wp_index
        num_waypoints = len(waypoints)
        for i in range(wp_index + 1, num_waypoints):
            arc_length += math.sqrt((waypoints[i][0] - waypoints[i-1][0])**2 + (waypoints[i][1] -
 waypoints[i-1][1])**2)
            if arc_length > self._lookahead: break
        return i

    def check_for_stop_signs(self, waypoints, closest_index, goal_index):
        for i in range(closest_index, goal_index):
            intersect_flag = False
            for stopsign_fence in self._stopsign_fences:
                wp_1   = np.array(waypoints[i][0:2])
                wp_2   = np.array(waypoints[i+1][0:2])
                s_1    = np.array(stopsign_fence[0:2])
                s_2    = np.array(stopsign_fence[2:4])
                v1     = np.subtract(wp_2, wp_1)
                v2     = np.subtract(s_1, wp_2)
                sign_1 = np.sign(np.cross(v1, v2))
                v2     = np.subtract(s_2, wp_2)
                sign_2 = np.sign(np.cross(v1, v2))
                v1     = np.subtract(s_2, s_1)
                v2     = np.subtract(wp_1, s_2)
                sign_3 = np.sign(np.cross(v1, v2))
                v2     = np.subtract(wp_2, s_2)
                sign_4 = np.sign(np.cross(v1, v2))
                if (sign_1 != sign_2) and (sign_3 != sign_4): intersect_flag = True
                if (sign_1 == 0) and pointOnSegment(wp_1, s_1, wp_2): intersect_flag = True
                if (sign_2 == 0) and pointOnSegment(wp_1, s_2, wp_2): intersect_flag = True
                if (sign_3 == 0) and pointOnSegment(s_1, wp_1, s_2): intersect_flag = True
                if (sign_3 == 0) and pointOnSegment(s_1, wp_2, s_2): intersect_flag = True
                if intersect_flag:
                    goal_index = i
                    return goal_index, True
        return goal_index, False

    def check_for_lead_vehicle(self, ego_state, lead_car_position):
      if not self._follow_lead_vehicle:
          lead_car_delta_vector=[lead_car_position[0]-ego_state[0],lead_car_position[1]-
 ego_state[1]]
          lead_car_distance = np.linalg.norm(lead_car_delta_vector)
          if lead_car_distance > self._follow_lead_vehicle_lookahead: return
          lead_car_delta_vector = np.divide(lead_car_delta_vector, lead_car_distance)
          ego_heading_vector = [math.cos(ego_state[2]), math.sin(ego_state[2])]
          if np.dot(lead_car_delta_vector, ego_heading_vector) < (1 / math.sqrt(2)):return
          self._follow_lead_vehicle = True
      else:
          lead_car_delta_vector = [lead_car_position[0] - ego_state[0],
                                   lead_car_position[1] - ego_state[1]]
          lead_car_distance = np.linalg.norm(lead_car_delta_vector)
          if lead_car_distance < self._follow_lead_vehicle_lookahead + 15:return
          lead_car_delta_vector = np.divide(lead_car_delta_vector, lead_car_distance)
          ego_heading_vector = [math.cos(ego_state[2]), math.sin(ego_state[2])]
          if np.dot(lead_car_delta_vector, ego_heading_vector) > (1 / math.sqrt(2)): return
          self._follow_lead_vehicle = False

def get_closest_index(waypoints, ego_state):
    closest_len = float('Inf')
    closest_index = 0
    for i, wp in enumerate(waypoints):
        d = math.sqrt((ego_state[0] - wp[0])**2 + (ego_state[1] - wp[1])**2)
        if d < closest_len:
            closest_len = d
            closest_index = i
    return closest_len, closest_index

def pointOnSegment(p1, p2, p3):
    if (p2[0] <= max(p1[0], p3[0]) and (p2[0] >= min(p1[0], p3[0])) and \
       (p2[1] <= max(p1[1], p3[1])) and (p2[1] >= min(p1[1], p3[1]))):
```

```
        return True
    else:
        return False
import cutils
import numpy as np

class Controller2D(object):
    def __init__(self, waypoints):
        self.vars                   = cutils.CUtils()
        self._lookahead_distance    = 2.0
        self._current_x             = 0
        self._current_y             = 0
        self._current_yaw           = 0
        self._current_speed         = 0
        self._desired_speed         = 0
        self._current_frame         = 0
        self._current_timestamp     = 0
        self._start_control_loop    = False
        self._set_throttle          = 0
        self._set_brake             = 0
        self._set_steer             = 0
        self._waypoints             = waypoints
        self._conv_rad_to_steer     = 180.0 / 70.0 / np.pi
        self._pi                    = np.pi
        self._2pi                   = 2.0 * np.pi

    def update_values(self, x, y, yaw, speed, timestamp, frame):
        self._current_x         = x
        self._current_y         = y
        self._current_yaw       = yaw
        self._current_speed     = speed
        self._current_timestamp = timestamp
        self._current_frame     = frame
        if self._current_frame:
            self._start_control_loop = True

    def get_lookahead_index(self, lookahead_distance):
        min_idx       = 0
        min_dist      = float("inf")
        for i in range(len(self._waypoints)):
            dist = np.linalg.norm(np.array([
                    self._waypoints[i][0] - self._current_x,
                    self._waypoints[i][1] - self._current_y]))
            if dist < min_dist:
                min_dist = dist
                min_idx = i

        total_dist = min_dist
        lookahead_idx = min_idx
        for i in range(min_idx + 1, len(self._waypoints)):
            if total_dist >= lookahead_distance:
                break
            total_dist += np.linalg.norm(np.array([
                    self._waypoints[i][0] - self._waypoints[i-1][0],
                    self._waypoints[i][1] - self._waypoints[i-1][1]]))
            lookahead_idx = i
        return lookahead_idx

    def update_desired_speed(self):
        min_idx       = 0
        min_dist      = float("inf")
        desired_speed = 0
        for i in range(len(self._waypoints)):
            dist = np.linalg.norm(np.array([
                    self._waypoints[i][0] - self._current_x,
                    self._waypoints[i][1] - self._current_y]))
            if dist < min_dist:
                min_dist = dist
                min_idx = i
        self._desired_speed = self._waypoints[min_idx][2]
```

```
    def update_waypoints(self, new_waypoints): self._waypoints = new_waypoints

    def get_commands(self):return self._set_throttle, self._set_steer, self._set_brake

    def set_throttle(self, input_throttle):
        throttle           = np.fmax(np.fmin(input_throttle, 1.0), 0.0)
        self._set_throttle = throttle

    def set_steer(self, input_steer_in_rad):
        input_steer = self._conv_rad_to_steer * input_steer_in_rad
        steer           = np.fmax(np.fmin(input_steer, 1.0), -1.0)
        self._set_steer = steer

    def set_brake(self, input_brake):
        brake           = np.fmax(np.fmin(input_brake, 1.0), 0.0)
        self._set_brake = brake

    def update_controls(self):
        x               = self._current_x
        y               = self._current_y
        yaw             = self._current_yaw
        v               = self._current_speed
        self.update_desired_speed()
        v_desired       = self._desired_speed
        t               = self._current_timestamp
        waypoints       = self._waypoints
        throttle_output = 0
        steer_output    = 0
        brake_output    = 0
        self.vars.create_var('kp', 0.50)
        self.vars.create_var('ki', 0.30)
        self.vars.create_var('integrator_min', 0.0)
        self.vars.create_var('integrator_max', 10.0)
        self.vars.create_var('kd', 0.13)
        self.vars.create_var('kp_heading', 8.00)
        self.vars.create_var('k_speed_crosstrack', 0.00)
        self.vars.create_var('cross_track_deadband', 0.01)
        self.vars.create_var('x_prev', 0.0)
        self.vars.create_var('y_prev', 0.0)
        self.vars.create_var('yaw_prev', 0.0)
        self.vars.create_var('v_prev', 0.0)
        self.vars.create_var('t_prev', 0.0)
        self.vars.create_var('v_error', 0.0)
        self.vars.create_var('v_error_prev', 0.0)
        self.vars.create_var('v_error_integral', 0.0)
        if self._start_control_loop:
            self.vars.v_error = v_desired - v
            self.vars.v_error_integral += self.vars.v_error * (t - self.vars.t_prev)
            v_error_rate_of_change=(self.vars.v_error-self.vars.v_error_prev)/(t- self.vars.t_prev)
            self.vars.v_error_integral =np.fmax(np.fmin(self.vars.v_error_integral,
                                        self.vars.integrator_max), self.vars.integrator_min)
            throttle_output = self.vars.kp * self.vars.v_error +self.vars.ki * self.vars.v_error_in
tegral +self.vars.kd * v_error_rate_of_change
            crosstrack_error = float("inf")
            crosstrack_vector = np.array([float("inf"), float("inf")])
            ce_idx = self.get_lookahead_index(self._lookahead_distance)
            crosstrack_vector = np.array([waypoints[ce_idx][0] - x -
 self._lookahead_distance*np.cos(yaw), waypoints[ce_idx][1] - y -
 self._lookahead_distance*np.sin(yaw)])
            crosstrack_error = np.linalg.norm(crosstrack_vector)
            if crosstrack_error < self.vars.cross_track_deadband: crosstrack_error = 0.0
            crosstrack_heading = np.arctan2(crosstrack_vector[1], crosstrack_vector[0])
            crosstrack_heading_error = crosstrack_heading - yaw
            crosstrack_heading_error = (crosstrack_heading_error + self._pi)%self._2pi - self._pi
            crosstrack_sign = np.sign(crosstrack_heading_error)
            if ce_idx < len(waypoints)-1:
               vect_wp0_to_wp1 = np.array([waypoints[ce_idx+1][0] - waypoints[ce_idx][0],
waypoints[ce_idx+1][1] - waypoints[ce_idx][1]])
               trajectory_heading = np.arctan2(vect_wp0_to_wp1[1],vect_wp0_to_wp1[0])
            else:
```

```
            vect_wp0_to_wp1 = np.array([waypoints[0][0] - waypoints[-1][0], waypoints[0][1] -
 waypoints[-1][1]])
            trajectory_heading=np.arctan2(vect_wp0_to_wp1[1],vect_wp0_to_wp1[0])
            heading_error = trajectory_heading - yaw
            heading_error = (heading_error + self._pi) % self._2pi - self._pi
            steer_output = heading_error + np.arctan(self.vars.kp_heading *crosstrack_sign * crosst
rack_error /(v + self.vars.k_speed_crosstrack))
            self.set_throttle(throttle_output)
            self.set_steer(steer_output)
            self.set_brake(brake_output)
        self.vars.x_prev       = x
        self.vars.y_prev       = y
        self.vars.yaw_prev     = yaw
        self.vars.v_prev       = v
        self.vars.v_error_prev = self.vars.v_error
        self.vars.t_prev       = t

import numpy as np
import copy
import path_optimizer
import collision_checker
import velocity_planner
from math import sin, cos, pi, sqrt

class LocalPlanner:
    def __init__(self, num_paths, path_offset, circle_offsets, circle_radii,
path_select_weight, time_gap, a_max, slow_speed, stop_line_buffer):
        self._num_paths = num_paths
        self._path_offset = path_offset
        self._path_optimizer = path_optimizer.PathOptimizer()
        self._collision_checker =collision_checker.CollisionChecker(circle_offsets, circle_radii,
                                                path_select_weight)
        self._velocity_planner =velocity_planner.VelocityPlanner(time_gap, a_max, slow_speed,
stop_line_buffer)
    def get_goal_state_set(self, goal_index, goal_state, waypoints, ego_state):
        if goal_index == len(waypoints) - 1:
            delta_x = waypoints[goal_index][0] - waypoints[goal_index-1][0]
            delta_y = waypoints[goal_index][1] - waypoints[goal_index-1][1]
        else:
            delta_x = waypoints[goal_index+1][0] - waypoints[goal_index][0]
            delta_y = waypoints[goal_index+1][1] - waypoints[goal_index][1]
        heading = np.arctan2(delta_y, delta_x)
        goal_state_local = copy.copy(goal_state)
        goal_state_local[0] -= ego_state[0]
        goal_state_local[1] -= ego_state[1]
        x = goal_state_local[0]
        y = goal_state_local[1]
        theta = -ego_state[2]
        goal_x = cos(theta) * x - sin(theta) * y
        goal_y = sin(theta) * x + cos(theta) * y
        goal_t = heading + theta
        goal_v = goal_state[2]
        if goal_t > pi: goal_t -= 2*pi
        elif goal_t < -pi: goal_t += 2*pi
        goal_state_set = []
        for i in range(self._num_paths):
            offset = (i - self._num_paths // 2) * self._path_offset
            x_offset = cos(goal_t + pi/2) * offset
            y_offset = sin(goal_t + pi/2) * offset
            goal_state_set.append([goal_x+x_offset,goal_y+y_offset,goal_t,goal_v])
        return goal_state_set

    def plan_paths(self, goal_state_set):
        paths         = []
        path_validity = []
        for goal_state in goal_state_set:
            path = self._path_optimizer.optimize_spiral(goal_state[0],goal_state[1], goal_state[2])
            if np.linalg.norm([path[0][-1] - goal_state[0], path[1][-1] - goal_state[1], path[2][-
1] - goal_state[2]]) > 0.1: path_validity.append(False)
            else:
                paths.append(path)
```

```
                path_validity.append(True)
        return paths, path_validity

def transform_paths(paths, ego_state):
  transformed_paths = []
  for path in paths:
    x_transformed = []
    y_transformed = []
    t_transformed = []
    for i in range(len(path[0])):
      x_transformed.append(ego_state[0]+path[0][i]*cos(ego_state[2])-path[1][i]*sin(ego_state[2]))
      y_transformed.append(ego_state[1]+ path[0][i]*sin(ego_state[2])+path[1][i]*cos(ego_state[2]))
      t_transformed.append(path[2][i] + ego_state[2])
    transformed_paths.append([x_transformed, y_transformed, t_transformed])
  return transformed_paths

from __future__ import print_function
from __future__ import division
import sys
import os
import argparse
import logging
import time
import math
import numpy as np
import csv
import matplotlib.pyplot as plt
import controller2d
import configparser
import local_planner
import behavioural_planner
sys.path.append(os.path.abspath(sys.path[0] + '/..'))
import live_plotter as lv
from carla            import sensor
from carla.client     import make_carla_client, VehicleControl
from carla.settings   import CarlaSettings
from carla.tcp        import TCPConnectionError
from carla.controller import utils
ITER_FOR_SIM_TIMESTEP  = 10
WAIT_TIME_BEFORE_START = 1.00
TOTAL_RUN_TIME         = 100.00
TOTAL_FRAME_BUFFER     = 300
NUM_PEDESTRIANS        = 0
NUM_VEHICLES           = 2
SEED_PEDESTRIANS       = 0
SEED_VEHICLES          = 0
CLIENT_WAIT_TIME       = 3
WEATHERID = {"DEFAULT":0, "CLEARNOON":1, "CLOUDYNOON":2, "WETNOON":3,
"WETCLOUDYNOON":4, "MIDRAINYNOON":5, "HARDRAINNOON":6,
"SOFTRAINNOON":7, "CLEARSUNSET": 8, "CLOUDYSUNSET":9, "WETSUNSET":10,
"WETCLOUDYSUNSET":11, "MIDRAINSUNSET":12, "HARDRAINSUNSET":13,"SOFTRAINSUNSET":14,}
SIMWEATHER = WEATHERID["CLEARNOON"]
PLAYER_START_INDEX = 1
FIGSIZE_X_INCHES   = 8
FIGSIZE_Y_INCHES   = 8
PLOT_LEFT          = 0.1
PLOT_BOT           = 0.1
PLOT_WIDTH         = 0.8
PLOT_HEIGHT        = 0.8
DIST_THRESHOLD_TO_LAST_WAYPOINT = 2.0
NUM_PATHS = 7
BP_LOOKAHEAD_BASE      = 8.0
BP_LOOKAHEAD_TIME      = 2.0
PATH_OFFSET            = 1.5
CIRCLE_OFFSETS         = [-1.0, 1.0, 3.0]
CIRCLE_RADII           = [1.5, 1.5, 1.5]
TIME_GAP               = 1.0
PATH_SELECT_WEIGHT     = 10
A_MAX                  = 1.5
SLOW_SPEED             = 2.0
STOP_LINE_BUFFER       = 3.5
```

```
LEAD_VEHICLE_LOOKAHEAD = 20.0
LP_FREQUENCY_DIVISOR   = 2
C4_STOP_SIGN_FILE        = 'stop_sign_params.txt'
C4_STOP_SIGN_FENCELENGTH = 5
C4_PARKED_CAR_FILE       = 'parked_vehicle_params.txt'
INTERP_MAX_POINTS_PLOT    = 10
INTERP_DISTANCE_RES       = 0.01
CONTROLLER_OUTPUT_FOLDER = os.path.dirname(os.path.realpath(__file__)) +'/controller_output/'

def make_carla_settings(args):
    settings = CarlaSettings()
    get_non_player_agents_info = False
    if (NUM_PEDESTRIANS > 0 or NUM_VEHICLES > 0): get_non_player_agents_info = True
    settings.set(SynchronousMode=True, SendNonPlayerAgentsInfo=get_non_player_agents_info,
        NumberOfVehicles=NUM_VEHICLES, NumberOfPedestrians=NUM_PEDESTRIANS,
        SeedVehicles=SEED_VEHICLES, SeedPedestrians=SEED_PEDESTRIANS, WeatherId=SIMWEATHER,
        QualityLevel=args.quality_level)
    return settings

class Timer(object):
    def __init__(self, period):
        self.step = 0
        self._lap_step = 0
        self._lap_time = time.time()
        self._period_for_lap = period

    def tick(self): self.step += 1

    def has_exceeded_lap_period(self):
        if self.elapsed_seconds_since_lap() >= self._period_for_lap: return True
        else: return False

    def lap(self):
        self._lap_step = self.step
        self._lap_time = time.time()

    def ticks_per_second(self):
        return float(self.step - self._lap_step) / self.elapsed_seconds_since_lap()

    def elapsed_seconds_since_lap(self):return time.time() - self._lap_time

def get_current_pose(measurement):
    x   = measurement.player_measurements.transform.location.x
    y   = measurement.player_measurements.transform.location.y
    yaw = math.radians(measurement.player_measurements.transform.rotation.yaw)
    return (x, y, yaw)

def get_start_pos(scene):
    x = scene.player_start_spots[0].location.x
    y = scene.player_start_spots[0].location.y
    yaw = math.radians(scene.player_start_spots[0].rotation.yaw)
    return (x, y, yaw)

def get_player_collided_flag(measurement, prev_collision_vehicles, prev_collision_pedestrians,
                             prev_collision_other):
    player_meas = measurement.player_measurements
    current_collision_vehicles = player_meas.collision_vehicles
    current_collision_pedestrians = player_meas.collision_pedestrians
    current_collision_other = player_meas.collision_other
    collided_vehicles = current_collision_vehicles > prev_collision_vehicles
    collided_pedestrians = current_collision_pedestrians>prev_collision_pedestrians
    collided_other = current_collision_other > prev_collision_other
    return (collided_vehicles or collided_pedestrians or collided_other,  current_collision_vehicles,
            current_collision_pedestrians, current_collision_other)

def send_control_command(client, throttle, steer, brake, hand_brake=False, reverse=False):
    control = VehicleControl()
    steer = np.fmax(np.fmin(steer, 1.0), -1.0)
    throttle = np.fmax(np.fmin(throttle, 1.0), 0)
    brake = np.fmax(np.fmin(brake, 1.0), 0)
    control.steer = steer
```

```
    control.throttle = throttle
    control.brake = brake
    control.hand_brake = hand_brake
    control.reverse = reverse
    client.send_control(control)

def create_controller_output_dir(output_folder):
    if not os.path.exists(output_folder): os.makedirs(output_folder)

def store_trajectory_plot(graph, fname):
    create_controller_output_dir(CONTROLLER_OUTPUT_FOLDER)
    file_name = os.path.join(CONTROLLER_OUTPUT_FOLDER, fname)
    graph.savefig(file_name)

def write_trajectory_file(x_list, y_list, v_list, t_list, collided_list):
    create_controller_output_dir(CONTROLLER_OUTPUT_FOLDER)
    file_name = os.path.join(CONTROLLER_OUTPUT_FOLDER, 'trajectory.txt')
    with open(file_name, 'w') as trajectory_file:
        for i in range(len(x_list)):
            trajectory_file.write('%3.3f, %3.3f, %2.3f, %6.3f %r\n' %(x_list[i], y_list[i], v_list[
i], t_list[i], collided_list[i]))

def write_collisioncount_file(collided_list):
    create_controller_output_dir(CONTROLLER_OUTPUT_FOLDER)
    file_name = os.path.join(CONTROLLER_OUTPUT_FOLDER, 'collision_count.txt')
    with open(file_name, 'w') as collision_file:
        collision_file.write(str(sum(collided_list)))

def exec_waypoint_nav_demo(args):
    with make_carla_client(args.host, args.port) as client:
        print('Carla client connected.')
        settings = make_carla_settings(args)
        scene = client.load_settings(settings)
        player_start = PLAYER_START_INDEX
        client.start_episode(player_start)
        time.sleep(CLIENT_WAIT_TIME);
        print('Starting new episode at %r...' % scene.map_name)
        client.start_episode(player_start)
        config = configparser.ConfigParser()
        config.read(os.path.join(os.path.dirname(os.path.realpath(__file__)), 'options.cfg'))
        demo_opt = config['Demo Parameters']
        enable_live_plot = demo_opt.get('live_plotting', 'true').capitalize()
        enable_live_plot = enable_live_plot == 'True'
        live_plot_period = float(demo_opt.get('live_plotting_period', 0))
        live_plot_timer = Timer(live_plot_period)
        stopsign_data = None
        stopsign_fences = []
        with open(C4_STOP_SIGN_FILE, 'r') as stopsign_file:
            next(stopsign_file)
            stopsign_reader = csv.reader(stopsign_file, delimiter=',',quoting=csv.QUOTE_NONNUMERIC)
            stopsign_data = list(stopsign_reader)
            for i in range(len(stopsign_data)):stopsign_data[i][3]=stopsign_data[i][3]*np.pi/ 180.0
        for i in range(len(stopsign_data)):
            x = stopsign_data[i][0]
            y = stopsign_data[i][1]
            z = stopsign_data[i][2]
            yaw = stopsign_data[i][3] + np.pi / 2.0
            spos = np.array([[0, 0], [0, C4_STOP_SIGN_FENCELENGTH]])
            rotyaw = np.array([[np.cos(yaw), np.sin(yaw)],[-np.sin(yaw), np.cos(yaw)]])
            spos_shift = np.array([[x, x],[y, y]])
            spos = np.add(np.matmul(rotyaw, spos), spos_shift)
            stopsign_fences.append([spos[0,0], spos[1,0], spos[0,1], spos[1,1]])
        parkedcar_data = None
        parkedcar_box_pts = []
        with open(C4_PARKED_CAR_FILE, 'r') as parkedcar_file:
            next(parkedcar_file)
            parkedcar_reader = csv.reader(parkedcar_file, delimiter=',',
                                          quoting=csv.QUOTE_NONNUMERIC)
            parkedcar_data = list(parkedcar_reader)
            for i in range(len(parkedcar_data)):
                parkedcar_data[i][3] = parkedcar_data[i][3] * np.pi / 180.0
```

```
        for i in range(len(parkedcar_data)):
            x = parkedcar_data[i][0]
            y = parkedcar_data[i][1]
            z = parkedcar_data[i][2]
            yaw = parkedcar_data[i][3]
            xrad = parkedcar_data[i][4]
            yrad = parkedcar_data[i][5]
            zrad = parkedcar_data[i][6]
            cpos = np.array([[-xrad, -xrad, -xrad, 0,    xrad, xrad, xrad,  0    ],
                    [-yrad, 0,     yrad,  yrad, yrad, 0,    -yrad, -yrad]])
            rotyaw = np.array([[np.cos(yaw), np.sin(yaw)], [-np.sin(yaw), np.cos(yaw)]])
            cpos_shift = np.array([[x, x, x, x, x, x, x, x],[y, y, y, y, y, y, y, y]])
            cpos = np.add(np.matmul(rotyaw, cpos), cpos_shift)
            for j in range(cpos.shape[1]): parkedcar_box_pts.append([cpos[0,j], cpos[1,j]])
        waypoints_file = WAYPOINTS_FILENAME
        waypoints_np   = None
        with open(waypoints_file) as waypoints_file_handle:
            waypoints = list(csv.reader(waypoints_file_handle, delimiter=',',
                                        quoting=csv.QUOTE_NONNUMERIC))
            waypoints_np = np.array(waypoints)
        controller = controller2d.Controller2D(waypoints)
        num_iterations = ITER_FOR_SIM_TIMESTEP
        if (ITER_FOR_SIM_TIMESTEP < 1): num_iterations = 1
        measurement_data, sensor_data = client.read_data()
        sim_start_stamp = measurement_data.game_timestamp / 1000.0
        send_control_command(client, throttle=0.0, steer=0, brake=1.0)
        sim_duration = 0
        for i in range(num_iterations):
            measurement_data, sensor_data = client.read_data()
            send_control_command(client, throttle=0.0, steer=0, brake=1.0)
            if i==num_iterations-1:sim_duration = measurement_data.game_timestamp
 / 1000.0 - sim_start_stamp
        SIMULATION_TIME_STEP = sim_duration / float(num_iterations)
        TOTAL_EPISODE_FRAMES = int((TOTAL_RUN_TIME + WAIT_TIME_BEFORE_START) /\
                               SIMULATION_TIME_STEP) + TOTAL_FRAME_BUFFER
        measurement_data, sensor_data = client.read_data()
        start_timestamp = measurement_data.game_timestamp / 1000.0
        start_x, start_y, start_yaw = get_current_pose(measurement_data)
        send_control_command(client, throttle=0.0, steer=0, brake=1.0)
        x_history     = [start_x]
        y_history     = [start_y]
        yaw_history   = [start_yaw]
        time_history  = [0]
        speed_history = [0]
        collided_flag_history = [False]
        lp_traj = lv.LivePlotter(tk_title="Trajectory Trace")
        lp_1d = lv.LivePlotter(tk_title="Controls Feedback")
        trajectory_fig = lp_traj.plot_new_dynamic_2d_figure(
                title='Vehicle Trajectory',figsize=(FIGSIZE_X_INCHES, FIGSIZE_Y_INCHES),
edgecolor="black",rect=[PLOT_LEFT, PLOT_BOT, PLOT_WIDTH, PLOT_HEIGHT])
        trajectory_fig.set_invert_x_axis()
        trajectory_fig.set_axis_equal()
        trajectory_fig.add_graph("waypoints", window_size=waypoints_np.shape[0],
x0=waypoints_np[:,0], y0=waypoints_np[:,1], linestyle="-", marker="", color='g')
        trajectory_fig.add_graph("trajectory", window_size=TOTAL_EPISODE_FRAMES,
x0=[start_x]*TOTAL_EPISODE_FRAMES, y0=[start_y]*TOTAL_EPISODE_FRAMES,color=[1,0.5,0])
        trajectory_fig.add_graph("start_pos",window_size=1,x0=[start_x],y0=[start_y],
marker=11, color=[1, 0.5, 0], markertext="Start", marker_text_offset=1)
        trajectory_fig.add_graph("end_pos", window_size=1, x0=[waypoints_np[-1, 0]],
y0=[waypoints_np[-1,1]],marker="D",color='r',markertext="End",marker_text_offset=1)
        trajectory_fig.add_graph("car", window_size=1,marker="s", color='b', markertext="Car",marke
r_text_offset=1)
        trajectory_fig.add_graph("leadcar", window_size=1,marker="s", color='g', markertext="Lead C
ar",marker_text_offset=1)
        trajectory_fig.add_graph("stopsign", window_size=1,x0=[stopsign_fences[0][0]], y0=[stopsign
_fences[0][1]],marker="H", color="r",markertext="Stop Sign", marker_text_offset=1)
        trajectory_fig.add_graph("stopsign_fence", window_size=1,x0=[stopsign_fences[0][0], stopsig
n_fences[0][2]], y0=[stopsign_fences[0][1], stopsign_fences[0][3]], color="r")
        parkedcar_box_pts_np = np.array(parkedcar_box_pts)
        trajectory_fig.add_graph("parkedcar_pts", window_size=parkedcar_box_pts_np.shape[0],
x0=parkedcar_box_pts_np[:,0], y0=parkedcar_box_pts_np[:,1], linestyle="", marker="+", color='b')
```

```
        trajectory_fig.add_graph("selected_path",window_size=INTERP_MAX_POINTS_PLOT,x0=[start_x]*IN
TERP_MAX_POINTS_PLOT,y0=[start_y]*INTERP_MAX_POINTS_PLOT, color=[1, 0.5, 0.0], linewidth=3)
        for i in range(NUM_PATHS):
            trajectory_fig.add_graph("local_path " + str(i), window_size=200,
                                     x0=None, y0=None, color=[0.0, 0.0, 1.0])
        forward_speed_fig=lp_1d.plot_new_dynamic_figure(title="Forward Speed(m/s)")
        forward_speed_fig.add_graph("forward_speed",label="forward_speed",window_size=TOTAL_EPISODE
_FRAMES)
        forward_speed_fig.add_graph("reference_signal",label="reference_Signal",
                                    window_size=TOTAL_EPISODE_FRAMES)
        throttle_fig = lp_1d.plot_new_dynamic_figure(title="Throttle")
        throttle_fig.add_graph("throttle",label="throttle",window_size=TOTAL_EPISODE_FRAMES)
        brake_fig = lp_1d.plot_new_dynamic_figure(title="Brake")
        brake_fig.add_graph("brake",label="brake",window_size=TOTAL_EPISODE_FRAMES)
        steer_fig = lp_1d.plot_new_dynamic_figure(title="Steer")
        steer_fig.add_graph("steer",label="steer",window_size=TOTAL_EPISODE_FRAMES)
        if not enable_live_plot:
            lp_traj._root.withdraw()
            lp_1d._root.withdraw()
        wp_goal_index   = 0
        local_waypoints = None
        path_validity   = np.zeros((NUM_PATHS, 1), dtype=bool)
        lp = local_planner.LocalPlanner(NUM_PATHS, PATH_OFFSET, CIRCLE_OFFSETS, CIRCLE_RADII,
                                PATH_SELECT_WEIGHT, TIME_GAP, A_MAX, SLOW_SPEED, STOP_LINE_BUFFER)
        bp = behavioural_planner.BehaviouralPlanner(BP_LOOKAHEAD_BASE, stopsign_fences,
                                                    LEAD_VEHICLE_LOOKAHEAD)
        reached_the_end = False
        skip_first_frame = True
        current_timestamp = start_timestamp
        prev_collision_vehicles    = 0
        prev_collision_pedestrians = 0
        prev_collision_other       = 0

        for frame in range(TOTAL_EPISODE_FRAMES):
            measurement_data, sensor_data = client.read_data()
            prev_timestamp = current_timestamp
            current_x, current_y, current_yaw = get_current_pose(measurement_data)
            current_speed = measurement_data.player_measurements.forward_speed
            current_timestamp = float(measurement_data.game_timestamp) / 1000.0
            if current_timestamp <= WAIT_TIME_BEFORE_START:
                send_control_command(client, throttle=0.0, steer=0, brake=1.0)
                continue
            else: current_timestamp = current_timestamp - WAIT_TIME_BEFORE_START
            x_history.append(current_x)
            y_history.append(current_y)
            yaw_history.append(current_yaw)
            speed_history.append(current_speed)
            time_history.append(current_timestamp)
            collided_flag, prev_collision_vehicles, prev_collision_pedestrians,\
            prev_collision_other = get_player_collided_flag(measurement_data,prev_collision_vehicle
s, prev_collision_pedestrians, prev_collision_other)
            collided_flag_history.append(collided_flag)
            lead_car_pos    = []
            lead_car_length = []
            lead_car_speed  = []
            for agent in measurement_data.non_player_agents:
                agent_id = agent.id
                if agent.HasField('vehicle'):
                    lead_car_pos.append([agent.vehicle.transform.location.x,
                             agent.vehicle.transform.location.y])
                    lead_car_length.append(agent.vehicle.bounding_box.extent.x)
                    lead_car_speed.append(agent.vehicle.forward_speed)
            if frame % LP_FREQUENCY_DIVISOR == 0:
                open_loop_speed = lp._velocity_planner.get_open_loop_speed(current_timestamp -
 prev_timestamp)
                ego_state = [current_x, current_y, current_yaw, open_loop_speed]
                bp.set_lookahead(BP_LOOKAHEAD_BASE + BP_LOOKAHEAD_TIME * open_loop_speed)
                bp.transition_state(waypoints, ego_state, current_speed)
                bp.check_for_lead_vehicle(ego_state, lead_car_pos[1])
                goal_state_set=lp.get_goal_state_set(bp._goal_index,bp._goal_state,waypoints,ego_st
ate)
```

```
                paths, path_validity = lp.plan_paths(goal_state_set)
                paths = local_planner.transform_paths(paths, ego_state)
                collision_check_array=lp._collision_checker.collision_check(paths,[parkedcar_box_pt
s])
                best_index = lp._collision_checker.select_best_path_index(paths, collision_check_ar
ray, bp._goal_state)
                if best_index == None:best_path = lp._prev_best_path
                else:
                    best_path = paths[best_index]
                    lp._prev_best_path = best_path
                desired_speed = bp._goal_state[2]
                lead_car_state = [lead_car_pos[1][0], lead_car_pos[1][1], lead_car_speed[1]]
                decelerate_to_stop = bp._state == behavioural_planner.DECELERATE_TO_STOP
                local_waypoints = lp._velocity_planner.compute_velocity_profile(best_path, desired_
speed, ego_state, current_speed, decelerate_to_stop, lead_car_state, bp._follow_lead_vehicle)
                if local_waypoints != None:
                    wp_distance = []
                    local_waypoints_np = np.array(local_waypoints)
                    for i in range(1, local_waypoints_np.shape[0]):
                        wp_distance.append(np.sqrt((local_waypoints_np[i,0]- local_waypoints_np[i-
1,0])**2+(local_waypoints_np[i, 1]-local_waypoints_np[i-1,1])**2))
                    wp_distance.append(0)
                    wp_interp      = []
                    for i in range(local_waypoints_np.shape[0] - 1):
                      wp_interp.append(list(local_waypoints_np[i]))
                      num_pts_to_interp =int(np.floor(wp_distance[i]/float(INTERP_DISTANCE_RES))-1)
                      wp_vector = local_waypoints_np[i+1] - local_waypoints_np[i]
                      wp_uvector = wp_vector / np.linalg.norm(wp_vector[0:2])
                      for j in range(num_pts_to_interp):
                        next_wp_vector=INTERP_DISTANCE_RES*float(j+1)*wp_uvector
                        wp_interp.append(list(local_waypoints_np[i]+next_wp_vector))
                    wp_interp.append(list(local_waypoints_np[-1]))
                    controller.update_waypoints(wp_interp)
                    pass
            if local_waypoints != None and local_waypoints != []:
                controller.update_values(current_x, current_y, current_yaw,
                                         current_speed, current_timestamp, frame)
                controller.update_controls()
                cmd_throttle, cmd_steer, cmd_brake = controller.get_commands()
            else:
                cmd_throttle = 0.0
                cmd_steer = 0.0
                cmd_brake = 0.0
            if skip_first_frame and frame == 0:pass
            elif local_waypoints == None:pass
            else:
              trajectory_fig.roll("trajectory", current_x, current_y)
              trajectory_fig.roll("car", current_x, current_y)
              if lead_car_pos:trajectory_fig.roll("leadcar",lead_car_pos[1][0],lead_car_pos[1][1])
              forward_speed_fig.roll("forward_speed", current_timestamp,
                                     current_speed)
              forward_speed_fig.roll("reference_signal",current_timestamp,
                                     controller._desired_speed)
              throttle_fig.roll("throttle", current_timestamp, cmd_throttle)
              brake_fig.roll("brake", current_timestamp, cmd_brake)
              steer_fig.roll("steer", current_timestamp, cmd_steer)
              if frame % LP_FREQUENCY_DIVISOR == 0:
                path_counter = 0
                for i in range(NUM_PATHS):
                 if path_validity[i]:
                      if not collision_check_array[path_counter]:colour = 'r'
                      elif i == best_index: colour = 'k'
                      else: colour = 'b'
                      trajectory_fig.update("local_path " + str(i), paths[path_counter][0], paths[p
ath_counter][1], colour)
                      path_counter += 1
                 else:trajectory_fig.update("local_path "+str(i),[ego_state[0]],[ego_state[1]],'r')
              wp_interp_np = np.array(wp_interp)
              path_indices = np.floor(np.linspace(0, wp_interp_np.shape[0]-1,
INTERP_MAX_POINTS_PLOT))
              trajectory_fig.update("selected_path",wp_interp_np[path_indices.astype(int), 0],
```

```
                    wp_interp_np[path_indices.astype(int), 1], new_colour=[1, 0.5, 0.0])
            if enable_live_plot and live_plot_timer.has_exceeded_lap_period():
                lp_traj.refresh()
                lp_1d.refresh()
                live_plot_timer.lap()
            send_control_command(client, throttle=cmd_throttle, steer=cmd_steer, brake=cmd_brake)
            dist_to_last_waypoint = np.linalg.norm(np.array([waypoints[-1][0] - current_x,
                waypoints[-1][1] - current_y]))
            if  dist_to_last_waypoint <DIST_THRESHOLD_TO_LAST_WAYPOINT: reached_the_end = True
            if reached_the_end: break
        if reached_the_end: print("Reached the end of path.")
        else: print("Exceeded assessment time. Writing to controller_output...")
        send_control_command(client, throttle=0.0,steer=0.0, brake=1.0)
        store_trajectory_plot(trajectory_fig.fig, 'trajectory.png')
        store_trajectory_plot(forward_speed_fig.fig,'forward_speed.png')
        store_trajectory_plot(throttle_fig.fig, 'throttle_output.png')
        store_trajectory_plot(brake_fig.fig, 'brake_output.png')
        store_trajectory_plot(steer_fig.fig, 'steer_output.png')
        write_trajectory_file(x_history,y_history,speed_history,time_history,collided_flag_history)
        write_collisioncount_file(collided_flag_history)

def main():
    argparser = argparse.ArgumentParser(description=__doc__)
    argparser.add_argument('-v', '--verbose', action='store_true', dest='debug',
        help='print debug information')
    argparser.add_argument('--host', metavar='H', default='localhost',
        help='IP of the host server (default: localhost)')
    argparser.add_argument('-p', '--port', metavar='P', default=2000, type=int,
        help='TCP port to listen to (default: 2000)')
    argparser.add_argument('-a', '--autopilot', action='store_true',
        help='enable autopilot')
    argparser.add_argument('-q', '--quality-level', choices=['Low', 'Epic'],
        type=lambda s: s.title(),default='Low', help='graphics quality level.')
    argparser.add_argument('-c', '--carla-settings', metavar='PATH', dest='settings_filepath',
default=None, help='Path to a "CarlaSettings.ini" file')
    args = argparser.parse_args()
    log_level = logging.DEBUG if args.debug else logging.INFO
    logging.basicConfig(format='%(levelname)s: %(message)s', level=log_level)
    logging.info('listening to server %s:%s', args.host, args.port)
    args.out_filename_format = '_out/episode_{:0>4d}/{:s}/{:0>6d}'
    while True:
        try:
            exec_waypoint_nav_demo(args)
            return
        except TCPConnectionError as error:
            logging.error(error)
            time.sleep(1)
if __name__ == '__main__':
    try: main()
    except KeyboardInterrupt: print('\nCancelled by user. Bye!')

from math import sin, cos, pi, sqrt

class VelocityPlanner:
    def __init__(self, time_gap, a_max, slow_speed, stop_line_buffer):
        self._time_gap         = time_gap
        self._a_max            = a_max
        self._slow_speed       = slow_speed
        self._stop_line_buffer = stop_line_buffer
        self._prev_trajectory  = [[0.0, 0.0, 0.0]]
    def get_open_loop_speed(self, timestep):
        if len(self._prev_trajectory) == 1: return self._prev_trajectory[0][2]
        if timestep < 1e-4: return self._prev_trajectory[0][2]
        for i in range(len(self._prev_trajectory)-1):
            distance_step = np.linalg.norm(np.subtract(self._prev_trajectory[i+1][0:2],
self._prev_trajectory[i][0:2]))
            velocity = self._prev_trajectory[i][2]
            time_delta = distance_step / velocity
            if time_delta > timestep:
                v1 = self._prev_trajectory[i][2]
                v2 = self._prev_trajectory[i+1][2]
```

```
                v_delta = v2 - v1
                interpolation_ratio = timestep / time_delta
                return v1 + interpolation_ratio * v_delta
            else: timestep -= time_delta
        return self._prev_trajectory[-1][2]

    def compute_velocity_profile(self, path, desired_speed, ego_state,
closed_loop_speed, decelerate_to_stop, lead_car_state, follow_lead_vehicle):
        profile = []
        start_speed = ego_state[3]
        if decelerate_to_stop: profile = self.decelerate_profile(path, start_speed)
        elif follow_lead_vehicle:profile = self.follow_profile(path, start_speed, desired_speed,
lead_car_state)
        else: profile = self.nominal_profile(path, start_speed, desired_speed)
        if len(profile) > 1:
            interpolated_state = [(profile[1][0] - profile[0][0]) * 0.1 + profile[0][0],
(profile[1][1]-profile[0][1])*0.1+profile[0][1],(profile[1][2]-profile[0][2])*0.1 + profile[0][2]]
            del profile[0]
            profile.insert(0, interpolated_state)
        self._prev_trajectory = profile
        return profile

    def decelerate_profile(self, path, start_speed):
        profile          = []
        slow_speed       = self._slow_speed
        stop_line_buffer = self._stop_line_buffer
        decel_distance = calc_distance(start_speed, slow_speed, -self._a_max)
        brake_distance = calc_distance(slow_speed, 0, -self._a_max)
        path_length = 0.0
        for i in range(len(path[0])-1):
            path_length += np.linalg.norm([path[0][i+1] - path[0][i], path[1][i+1] - path[1][i]])
        stop_index = len(path[0]) - 1
        temp_dist = 0.0
        while (stop_index > 0) and (temp_dist < stop_line_buffer):
            temp_dist += np.linalg.norm([path[0][stop_index] - path[0][stop_index-1],
path[1][stop_index] - path[1][stop_index-1]])
            stop_index -= 1
        if brake_distance + decel_distance + stop_line_buffer > path_length:
            speeds = []
            vf = 0.0
            for i in reversed(range(stop_index, len(path[0]))):speeds.insert(0,0.0)
            for i in reversed(range(stop_index)):
                dist = np.linalg.norm([path[0][i+1] - path[0][i], path[1][i+1] - path[1][i]])
                vi = calc_final_speed(vf, -self._a_max, dist)
                if vi > start_speed: vi = start_speed
                speeds.insert(0, vi)
                vf = vi
            for i in range(len(speeds)): profile.append([path[0][i], path[1][i], speeds[i]])
        else:
            brake_index = stop_index
            temp_dist = 0.0
            while (brake_index > 0) and (temp_dist < brake_distance):
                temp_dist += np.linalg.norm([path[0][brake_index] - path[0][brake_index-1],
path[1][brake_index] - path[1][brake_index-1]])
                brake_index -= 1
            decel_index = 0
            temp_dist = 0.0
            while (decel_index < brake_index) and (temp_dist < decel_distance):
                temp_dist += np.linalg.norm([path[0][decel_index+1] - path[0][decel_index],
path[1][decel_index+1] - path[1][decel_index]])
                decel_index += 1
            vi = start_speed
            for i in range(decel_index):
                dist = np.linalg.norm([path[0][i+1] - path[0][i], path[1][i+1] - path[1][i]])
                vf = calc_final_speed(vi, -self._a_max, dist)
                if vf < slow_speed: vf = slow_speed
                profile.append([path[0][i], path[1][i], vi])
                vi = vf
            for i in range(decel_index, brake_index): profile.append([path[0][i], path[1][i], vi])
            for i in range(brake_index, stop_index):
                dist = np.linalg.norm([path[0][i+1] - path[0][i], path[1][i+1] - path[1][i]])
```

```
                vf = calc_final_speed(vi, -self._a_max, dist)
                profile.append([path[0][i], path[1][i], vi])
                vi = vf
            for i in range(stop_index, len(path[0])): profile.append([path[0][i], path[1][i], 0.0])
        return profile

    def follow_profile(self, path, start_speed, desired_speed, lead_car_state):
        profile = []
        min_index = len(path[0]) - 1
        min_dist = float('Inf')
        for i in range(len(path)):
            dist = np.linalg.norm([path[0][i] - lead_car_state[0], path[1][i] - lead_car_state[1]])
            if dist < min_dist:
                min_dist = dist
                min_index = i
        desired_speed = min(lead_car_state[2], desired_speed)
        ramp_end_index = min_index
        distance = min_dist
        distance_gap = desired_speed * self._time_gap
        while (ramp_end_index > 0) and (distance > distance_gap):
            distance += np.linalg.norm([path[0][ramp_end_index] - path[0][ramp_end_index-1],
path[1][ramp_end_index] - path[1][ramp_end_index-1]])
            ramp_end_index -= 1
        if desired_speed<start_speed:decel_distance=calc_distance(start_speed,desired_speed,-
self._a_max)
        else:decel_distance = calc_distance(start_speed, desired_speed,self._a_max)
        vi = start_speed
        for i in range(ramp_end_index + 1):
            dist=np.linalg.norm([path[0][i+1]-path[0][i],path[1][i+1]- path[1][i]])
            if desired_speed < start_speed: vf = calc_final_speed(vi, -self._a_max, dist)
            else: vf = calc_final_speed(vi, self._a_max, dist)
            profile.append([path[0][i], path[1][i], vi])
            vi = vf
        for i in range(ramp_end_index + 1, len(path[0])):
            profile.append([path[0][i], path[1][i], desired_speed])
        return profile

    def nominal_profile(self, path, start_speed, desired_speed):
        profile = []
        if desired_speed < start_speed:accel_distance = calc_distance(start_speed, desired_speed, -
self._a_max)
        else:accel_distance = calc_distance(start_speed, desired_speed,self._a_max)
        ramp_end_index = 0
        distance = 0.0
        while (ramp_end_index < len(path[0])-1) and (distance < accel_distance):
            distance += np.linalg.norm([path[0][ramp_end_index+1] - path[0][ramp_end_index],
path[1][ramp_end_index+1] - path[1][ramp_end_index]])
            ramp_end_index += 1
        vi = start_speed
        for i in range(ramp_end_index):
          dist = np.linalg.norm([path[0][i+1]-path[0][i],path[1][i+1]-path[1][i]])
          if desired_speed < start_speed:
                vf = calc_final_speed(vi, -self._a_max, dist)
                if vf < desired_speed: vf = desired_speed
          else:
                vf = calc_final_speed(vi, self._a_max, dist)
                if vf > desired_speed: vf = desired_speed
          profile.append([path[0][i], path[1][i], vi])
          vi = vf
        for i in range(ramp_end_index+1, len(path[0])):
            profile.append([path[0][i], path[1][i], desired_speed])
        return profile

def calc_distance(v_i, v_f, a): return (v_f**2 - v_i**2) / (2 * a)

def calc_final_speed(v_i, a, d): return sqrt(v_i**2 + 2 * a * d) if (v_i**2 + 2 * a * d) > 0 else 0
```

Приклад 3.9 - Реалізація допоміжних функцій

ДОДАТОК Б

Відомість

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	126 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Шаповалова К.Є.			Дослідження алгоритмів та розробка системи автоматичного керування автомобілем	Шифр групи	Код напр./спец.
Перевірив		Шергін В.Л.				СШІм-19-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					