Документация к коду робота-уборщика с двумя моторами

Импортируемые библиотеки

import time

import math

import numpy as np

import cv2

from rplidar import RPLidar

from heapq import heappush, heappop

import RPi.GPIO as GPIO

from mpu6050 import mpu6050

from RPLCD import CharLCD

from tensorflow.keras import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

import pygame

from collections import deque

import logging

from picamera import PiCamera

Описание:

Библиотеки, используемые в коде, включают модули для работы со временем, математическими операциями, полноценным управлением мотором, камерой, ЖК-дисплеем и машинным обучением.

Например, RPLidar используется для подключения к лидару, а pygame — для создания графического интерфейса.

Настройка логирования

logging.basicConfig(filename='robot.log', level=logging.INFO, format='%(asctime)s-%(levelname)s-%(message)s') logging.info("Робот запущен")

Описание:

* Установка конфигурации системы логирования, которая будет записывать все события и уведомления в файл robot.log. Формат включает временную метку, уровень логирования и само сообщение. Это позволяет отслеживать важные события в процессе работы робота, такие как ошибки или состояния.

Инициализация камеры и лидара

try:

lidar = RPLidar('/dev/ttyUSB0') # Замените на ваш порт

camera = PiCamera()

camera.resolution = (640, 480) # Обычное разрешение

camera.start\_preview()

except Exception as e:

logging.error(f"Ошибка инициализации компонентов: {e}")

Raise

Описание:

* Здесь происходит попытка инициализации основных сенсоров робота: лидар для определения расстояний до препятствий и камера для захвата изображений.
* Если инициализация не удалась, ошибка логируется, а программа прерывается, что позволяет избежать незапланированных сбоев в дальнейшем.

**Уровень заряда батареи**

battery\_level = 100

Akkumulyator\_umirayet = 20

**Описание:**

Установка начального уровня заряда батареи (100%). Также определяется минимальный порог (Akkumulyator\_umirayet), ниже которого робот начинает возвращаться на док-станцию для подзарядки.

**Настройка LCD-дисплея**

lcd = CharLCD(cols=16, rows=2, pin\_rs=15, pin\_e=14, pins\_data=[17, 18, 27, 22]) # Замените на ваши пины

**Описание:**

Инициализация LCD-дисплея с заданными параметрами: размеры (16x2 символов) и назначение управляющих пинов. Этот интерфейс будет использоваться для отображения информации о состоянии робота, уровне заряда и активном режиме.

**Инициализация MPU6050**

mpu = mpu6050()

def calibrate\_mpu():

try:

mpu.calibrate()

logging.info("MPU6050 откалиброван")

except Exception as e:

logging.error(f"Ошибка калибровки MPU6050: {e}")

calibrate\_mpu()  
  
**Описание:**

Создание объекта для работы с датчиком MPU6050. Определяется функция calibrate\_mpu(), которая запускает процесс калибровки для повышения точности измерений.

В случае ошибки калибровки она будет логироваться и указывать на проблемы с работой датчика.

**Эндкодер**

encoder\_point = 0 encoder\_pin = 23 # Замените на ваш пин GPIO.setmode(GPIO.BCM) GPIO.setup(encoder\_pin, GPIO.IN, pull\_up\_down=GPIO.PUD\_UP)

def encoder\_meter(channel): global encoder\_point encoder\_point += 1

GPIO.add\_event\_detect(encoder\_pin, GPIO.RISING, callback=encoder\_meter)

**Описание:**

* Инициализация энкодера для отслеживания движения робота. Прибор подключается к указанному GPIO пину, и функция обратного вызова encoder\_meter увеличивает счетчик encoder\_point при каждом срабатывании.
* Это позволяет отслеживать количество выполненных шагов или оборотов колес, что важно для определения пройденного расстояния.

**Обучения нейросети**

state\_size = 4 # Заменить на фактический размер

action\_size = 2 # Заменить на фактический размер

learning\_rate = 0.001

discount\_factor = 0.95

epsilon = 1.0

epsilon\_MIN = 0.01

epsilon\_DECAY = 0.995

memory\_size = 2000  
  
  
  
**Описание:**

Установка параметров для моделей машинного обучения. state\_size и action\_size определяют размеры состояния и действий соответственно.

Другие параметры, такие как learning\_rate (скорость обучения) и epsilon (для использования в методе ε-жадности), отвечают за управление процессом обучения модели.

**Создание модели нейросети**

model = Sequential([ Dense(24, input\_shape=(state\_size,), activation="relu"), Dense(action\_size, activation="linear") ]) model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss="mse") memory = deque(maxlen=memory\_size)

**Описание:**

Создание и компиляция простой нейросети с двумя слоями: входным слоем с 24 нейронами и выходным слоем с размером action\_size. Модель компилируется с использованием Adam-оптимизатора и функции потерь MSE (Mean Squared Error).

memory инициализируется как двусторонняя очередь, что позволяет хранить историю состояний, действий и результатов.

**Инициализация графического интерфейса**

pygame.init()

screen = pygame.display.set\_mode((800, 600)) # Замените на нужный размер

pygame.display.set\_caption("Робот Уборки")

clock = pygame.time.Clock()  
  
**Описание:**

Инициализация библиотеки Pygame для создания графического интерфейса. Создание окна заданного размера, установка заголовка и инициализация часов для управления частотой обновления экрана.

**Определение кнопок и цветов интерфейса**

button\_mode1 = pygame.Rect(50, 50, 200, 50)

button\_mode2 = pygame.Rect(50, 150, 200, 50)

button\_emergency = pygame.Rect(50, 250, 200, 50)

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

RED = (255, 0, 0)

GREEN = (0, 255, 0)

BLUE = (0, 0, 255)

YELLOW = (255, 255, 0)  
  
**Описание:**

Определение прямоугольных областей для кнопок управления режимами (Создание карты и Очистка) и экстренной остановки. Цвета задаются в RGB-формате и используются для визуального оформления интерфейса.

**Настройка GPIO для моторов**

# Пины для моторов   
motor\_left = 12 # Левый мотор   
motor\_right = 16 # Правый мотор   
# Настройка GPIO   
GPIO.setmode(GPIO.BCM) GPIO.setup(motor\_left, GPIO.OUT) GPIO.setup(motor\_right, GPIO.OUT)

Настройка GPIO для управления моторами робота, позволяющая их включать и выключать. Каждый мотор подключен к своему пину на Raspberry Pi, что позволяет управлять движением в разных направлениях.  
  
Определение функций движения

# Движение   
def move\_forward(): GPIO.output(motor\_left, GPIO.HIGH) # Левый мотор вперед   
 GPIO.output(motor\_right, GPIO.HIGH) # Правый мотор вперед logging.info("Движение вперед")   
def move\_backward(): GPIO.output(motor\_left, GPIO.LOW) # Левый мотор назад GPIO.output(motor\_right, GPIO.LOW) # Правый мотор назад logging.info("Движение назад") def turn\_left(): GPIO.output(motor\_left, GPIO.LOW) # Остановить левый мотор GPIO.output(motor\_right, GPIO.HIGH) # Включить правый мотор logging.info("Поворот влево") def turn\_right(): GPIO.output(motor\_left, GPIO.HIGH) # Включить левый мотор GPIO.output(motor\_right, GPIO.LOW) # Остановить правый мотор logging.info("Поворот вправо") def stop(): GPIO.output(motor\_left, GPIO.LOW) # Остановить левый мотор GPIO.output(motor\_right, GPIO.LOW) # Остановить правый мотор logging.info("Остановка")

**Описание:**

Набор функций для управления движением робота. Каждая функция активирует соответствующие моторы, чтобы déplacer в разные направления. Функция stop() останавливает моторы и логирует это событие.

**Класс для динамической карты**

class DynamicMap: def \_\_init\_\_(self): self.markup = [[0 for \_ in range(10)] for \_ in range(10)] # Образец размерности self.robot\_location\_point = (0, 0) self.zones = { "clean": [], "dock": [], "blackhole": [], "stairs": [] } self.cleaned\_zones = [] self.zone\_respawn\_time = 60  
  
**Описание:**

Класс DynamicMap реализует динамическое управление картой, которая отображает окружающую среду.

markup представляет собой двумерный массив для представления карты, где значения могут обозначать различные типы зон.

Инициализируются зоны для уборки, док-станции и других важных мест, таких как черные дыры (запрещенные зоны) и лестницы.

**Методы класса DynamicMap**

expand\_map(self, dx, dy): Расширяет карту в указанном направлении. Это будет полезно, если робот движется за границу текущей карты.

update\_map(self, x, y, value): Обновляет значение в клетке карты согласно координатам (x, y).

add\_zone(self, zone\_type, x, y): Метод, позволяющий добавлять зоны на карту (например, зоны очистки или дока).

get\_zone\_value(self, zone\_type): Возвращает значение, соответствующее определенному типу зоны, что помогает в обновлении карты.

mark\_zone\_as\_cleaned(self, x, y): Отмечает зону как очищенную и запоминает время.

respawn\_cleaned\_zones(self): Проверяет время, прошедшее с момента очистки, и восстанавливает очищенные зоны по истечении указанного времени.

**Получение данных с лидара**

def get\_lidar\_data(): scan\_data = [] try: for scan in lidar.iter\_scans(): if not scan: logging.warning("Лидар не вернул данные") break for (\_, angle, distance) in scan: scan\_data.append((angle, distance)) break except Exception as e: logging.error(f"Ошибка получения данных с лидара: {e}") return scan\_data

**Описание:**

Функция получает данные с лидара, которые используются для определения расстояний до объектов вокруг робота. Если данные отсутствуют или возникает ошибка, это регистрируется в логах.

**Обновление карты с данными лидара**

def update\_map\_with\_lidar(dynamic\_map):  
 scan\_data = get\_lidar\_data()  
 for angle, distance in scan\_data:  
 x = int(distance \* math.cos(math.radians(angle)))  
 y = int(distance \* math.sin(math.radians(angle)))  
 if 0 <= x < len(dynamic\_map.markup) and 0 <= y < len(dynamic\_map.markup[0]):   
 dynamic\_map.update\_map(x, y, 1)

**Описание:**

Обновляет карту на основе данных, полученных с лидара. Используется тригонометрия для преобразования углов и расстояний в координаты (x, y), которые затем используются для обновления карты.

**Проверка уровня заряда батареи**

def check\_battery():  
 global battery\_level  
 if battery\_level > 0:  
 battery\_level -= random.randint(0, 3) # Имитация разряда батареи  
 if battery\_level <= Akkumulyator\_umirayet:  
 logging.warning("Батарея разряжена! Возвращение на док-станцию.")  
 go\_to\_dock()  
 else:  
 logging.warning("Батарея разряжена! Робот остановлен.")  
 stop()

**Описание:**

Функция проверки батареи, обновляющая уровень заряда на основании случайного разряда для имитации реального времени работы. При низком уровне активности робот отправляется на зарядку.

**Логика возврата к док-станции**

def go\_to\_dock():  
 move\_forward() # Имитация движения робота к док-станции  
 time.sleep(2) # Задержка, чтобы имитировать движение  
 stop() # Останавливаемся  
 logging.info("Робот вернулся на док-станцию.")

**Описание:**

Функция, имитирующая движение робота к док-станции. В реальных приложениях следует реализовать более сложную навигацию. Функция делает паузу, чтобы продемонстрировать процесс возвращения.

**Обновление состояния на LCD-дисплее**

def update\_lcd():  
 lcd.clear()  
 lcd.write\_string(f"Уровень: {battery\_level}%")  
 if mode == 1:  
 lcd.write\_string("Режим: Карта")  
 else:  
 lcd.write\_string("Режим: Очистка")

**Описание:**

Функция update\_lcd очищает дисплей и отображает текущий уровень заряда, а также активный режим работы робота. Это важно для информирования пользователя в реальном времени о состоянии устройства.

**Главный цикл программы**

def main\_loop():  
 global mode  
 dynamic\_map = DynamicMap()  
  
 while True:  
 screen.fill(WHITE)  
 pygame.draw.rect(screen, GREEN, button\_mode1)  
 pygame.draw.rect(screen, BLUE, button\_mode2)  
 pygame.draw.rect(screen, RED, button\_emergency)  
  
 font = pygame.font.SysFont("Arial", 24)  
 text\_mode1 = font.render("Режим 1", True, WHITE)  
 text\_mode2 = font.render("Режим 2", True, WHITE)  
 text\_emergency = font.render("Стоп", True, WHITE)  
 screen.blit(text\_mode1, (button\_mode1.x + 10, button\_mode1.y + 10))  
 screen.blit(text\_mode2, (button\_mode2.x + 10, button\_mode2.y + 10))  
 screen.blit(text\_emergency, (button\_emergency.x + 10, button\_emergency.y + 10))  
  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 pygame.quit()  
 return  
 if event.type == pygame.MOUSEBUTTONDOWN:  
 x, y = event.pos  
 if button\_mode1.collidepoint(x, y):  
 mode = 1  
 logging.info("Переключение на режим 1: Создание карты")  
 elif button\_mode2.collidepoint(x, y):  
 mode = 2  
 logging.info("Переключение на режим 2: Очистка зон")  
 elif button\_emergency.collidepoint(x, y):  
 stop()  
 l

ogging.info("Экстренная остановка активирована")  
  
 if mode == 1:  
 update\_map\_with\_lidar(dynamic\_map)  
 elif mode == 2:  
 dynamic\_map.respawn\_cleaned\_zones()  
 check\_battery()  
  
 update\_lcd()  
 pygame.display.flip()  
 clock.tick(60) # 60 FPS

**Описание:**

В main\_loop() реализован главный цикл программы, который:

Очищает экран и заполняет его белым цветом.

Рисует кнопки для выбора режимов работы и экстренной остановки.

Обрабатывает события, такие как клики мыши для переключения режимов. В зависимости от активного режима выполняется логика обновления карты или очистки зон.

Обновляет состояние на LCD-дисплее и обновляет интерфейс Pygame.

**Запуск основного цикла и обработка исключений**

try:  
 main\_loop()  
except KeyboardInterrupt:  
 lidar.stop()  
 lidar.disconnect()  
 camera.close()  
 GPIO.cleanup()  
 pygame.quit()  
 stop()

**Описание:**

Блок try-except запускает основной «поток» программы. Если происходит прерывание (например, через нажатие Ctrl+C), все подключенные устройства корректно отключаются, освобождаются ресурсы и Pygame закрывается.