0HM280 Assignment 2

In this assignment you will learn to implement behavior-based navigation as described by Elena Torta in her thesis (chapter 3). The challenge here is to go from theory to something that is actually working. To do so you will work with a ‘nao simulator’ written in Python and you will need to modify the part that controls the behavior-based navigation.

# Nao simulator

The nao simulator simulates a 2D world with a robot, a target and a number of obstacles. The goal of the “game” is to navigate to the target without bumping into the obstacles. When the target is reached, the game ends.

A screenshot of a cell phone

Description automatically generated

Figure Screenshot of the nao simulator with the robot, target and obstacles shown.

## Manual mode

The game starts in manual mode. Using the arrow keys you can speed up the robot (arrow UP) or slow down the robot (arrow DOWN). In additions, you can increase the turnrate (arrow LEFT) or decrease the turnrate (arrow RIGHT). Pressing SPACE will stop the robot.

If the robot hits an obstacle, you will hear a sound and the robot will stop moving. If you reach the target you win the game and the program ends.

Menu:

ESC exit game

UP speed up robot

DOWN slow down robot

LEFT increase turnrate

RIGHT decrease turnrate

SPACE stop robot

a toggle autonomous navigation

c toggle collision detection

d draw sonar detections (in autonomous mode)

Figure Overview of commands for the nao simulator.

## Autonomous mode

When pressing “a” the robot switches to autonomous mode and will use its sonar sensor to update the turnrate and the velocity of the robot. This will not do much until you change the code for the behavior-based navigation (explained later). By pressing “c” collision detection is turned off or on. When off, the robot will move through obstacles. By pressing “d” you will see the scanlines of the robot’s sonar sensors including the collision points. This only works in autonomous mode. Note that the sonar only reports back the smallest detected distance. It does not know the direction of this shortest distance. It only knows the direction of the sensors are pointing (30 or -30 degree) and the angle within which it can detect something (60 degrees).

## The program

The program consists of several files. To start the game you need to run the program “*mainloop.py*”.

**definitions.py** – this file contains all the globals. It is always a good idea to define constants in one place only, and then import these definitions wherever you need them using from definitions import \*.

**mainloop.py** – this file handles the game logic. It initializes everything, creates the world, and it contains the main event loop that handles keypresses. It also updates the robot position and draws the screen.

**sprite\_routines.py** – defines the robot, target and obstacle classes. The robot class contains the simulated sonar sensor including the function to update the robot’s state and the functions to draw the robot. There are also several helper functions. You should not touch this file unless you know what you are doing.

**intersection.py** – contains linear algebra routines to find the intersections of a scanline and a polygon.

**my\_navigation\_code.py** – This code links the scan\_worldfunction that is called from *mainloop.py* to the behavior-based navigation algorithms present in *behavior\_based\_navigation.py*. You will probably need to make some modifications here.

**behavior\_based\_navigation.py** – This file needs to be created. A template *behavior\_based\_navigation\_empty.py* is provided, which you can copy and modify. The basic structure is provided, but the forces are all zero.

## The assignment

This assignment has 4 challenges with increasing difficulty level. The first two challenges should be easy enough for everyone. The third challenge will be difficult but doable. The fourth and last challenge is meant for those students who like to put their teeth in a hard problem. The last challenge gives bonus points.

**Preparation**

1. First install pygame if you haven’t done so already:

pip install pygame

1. Download “nao simulator.zip” from canvas and unpack it your documents folder (or wherever you keep your Python projects).
2. Make your own copy of **behavior\_based\_navigation.py** by copying behavior\_based\_navigation\_empty.py and renaming it.
3. Programming tips:
   1. Always make useful comments in your code
   2. Use easy to read and understand variable names
   3. Do not hard-code constants, but use variable names as much as possible.
   4. Use print statements to check the variables while the program is running.

**Challenges**

1. Run *mainloop.py* and try to manually reach the target to get a feeling of the problem the robot needs to solve. Record one videoclip by capturing the screen (Use windows + G to access the game bar menu).
2. Create the Braitenberg vehicles “love”, “coward”, “aggressive” and “explore”
   1. Use collision detection off or modify mainloop.py/create\_world(), so that the allobstacles contains no obstacles  
      allobstacles = sp.pygame.sprite.Group(())
   2. Modify the compute\_turnrate() and compute\_velocity() functions in the *behavior\_based\_navigation.py* file. Note that the original Braitenberg vehicles have two sensors and two wheels. Here you need to fake the input by using the distance and the direction of the target, to compute the speed and turnrate. Note that if you change the arguments of these functions, you also need to change them in *my\_navigation\_code.py*
   3. Click anywhere in the game screen to move the target around, and test the behavior.
   4. Save copies of the *behavior\_based\_navigation.py* files you create and label them *behavior\_based\_navigation\_love.py*, etc. You need to hand in these files.
3. Implement the behavior-based navigation form Torta’s thesis chapter 3
   1. Revert to the default compute\_turnrate() and compute\_velocity() functions from the template (behavior\_based\_navigation\_empty.py), and revert to the original create\_world() function in mainloop.py
   2. First make a working attractive force for the target by modifying the FTar() function in *behavior\_based\_navigation.py*
   3. Make a repelling force for obstacles by modifying the FObs() function in *behavior\_based\_navigation.py* . **HINT: draw the functions as function of distance or angle and determine when one force is bigger than another.**
   4. Add a aligning force by modifying the FOrienting() function in *behavior\_based\_navigation.py*
   5. Run the program in autonomous mode and try to solve the maze. Save the working *behavior\_based\_navigation.py* file and add “\_challenge3” to the name.
4. **Bonus challenge**. Implement a Kalman filter for the sonar data. Make a new file called kalman.py and access it by importing it in my\_navigation\_code.py.
   1. The Kalman filter should receive sonar\_left and sonar\_right as inputs. Output are the filtered values.
   2. First implement the measurement update. Use the numpy library to do all the matrix calculations.
   3. Implement the movement update by assuming that the sonar distance will change by cos(30\*degree) times the distance moved by the robot. You will need to add the movement command to the inputs of the Kalman filter.

**Output**

* Challenge 1: A video clip
* Challenge 2: 4 *behaviour\_based\_navigation.py* files and a brief description of how the code is working. Include *my\_navigation\_code.py* if you made any modifications.
* Challenge 3: 1 *behaviour\_based\_navigation.py* file and a brief description of how the coder is working. Include *my\_navigation\_code.py* if you made any modifications.
* Challenge 4: *kalman.py* and a brief description of how the coder is working. Include any other files that were modified.
* Upload everything to canvas.