CYBER-PHYSICAL SYSTEMS AND ROBOTICS

Lab 2. Particle filter localization

1 Preparation

1. The `pass` keyword is used when there is a need to write a statement but as of now, nothing has been implemented yet. Nothing will happen when executing this line of code. The idea is to replace it afterward.
2. The `Idle` class allows to execute the loop in `main.py` with a fixed time step. This way one can control the update rate of the measurements as well as the commands given to the robot. This limits the data transfer to an acceptable value which is a trade-off between resource efficiency and precision of the robot control.
3. It is impossible to instantiate the `Robot` class. Indeed, it is an abstract class, implemented thanks to the `ABC` class which is part of the abc package. An abstract class must be complemented to be instantiated. The Robot class has abstract methods, like move and sense, which must be implemented by the class `RobotP3DX`, inheriting from Robot. Then, it is possible to create an instance of `RobotP3DX`.
4. In the `__init__` method of class `RobotP3DX`, there is a call to the `__init__` method of the mother class Robot, which initialize attributes defined in `Robot`. `RobotP3DX` can access them as it inherits from this class.

2 Code

2.1 Particle filter initialization

This method for robot localization relies on the generation of particles, each of which is then assigned a probability of the robot being there as a function of the measurements of theoretical sensors. Therefore, the first step in deploying the solution is to generate said particles. Each of this particles has three properties that completely describe them: x and y coordinates and θ which indicates the direction the particle is moving in. This is why we will store the particles in an array of tuples, as shown in the code below.

```
particles = np.empty((particle_count, 3), dtype=object)

map_bounds = self._map.bounds()  # retrieve the bounds of the map (rectangle
containing the map)


for particle in particles:
    is_valid = False
    while not is_valid:
        particle[0] = random.uniform(map_bounds[0], map_bounds[2])
        particle[1] = random.uniform(map_bounds[1], map_bounds[3])

        # the orientation has only 4 possible values
        is_valid = self._map.contains((particle[0], particle[1]))  # check if
particle is in map

    particle[2] = random.choice([0, np.pi / 2, np.pi, 3 * np.pi / 2])

return particles
```
Code 1: Particle initialization

As seen in Code 1, the particle generation is very straight forward. The while loop is used so that if any particle is generated in an "unreachable" area of the map, it will be discarded and substituted by a reasonable one. For this we use the contains method in the map.py script which returns False when said particle is not within map the boundaries (either internal or external ones). Figure 1 shows an example run of the generation of the first round of particles.

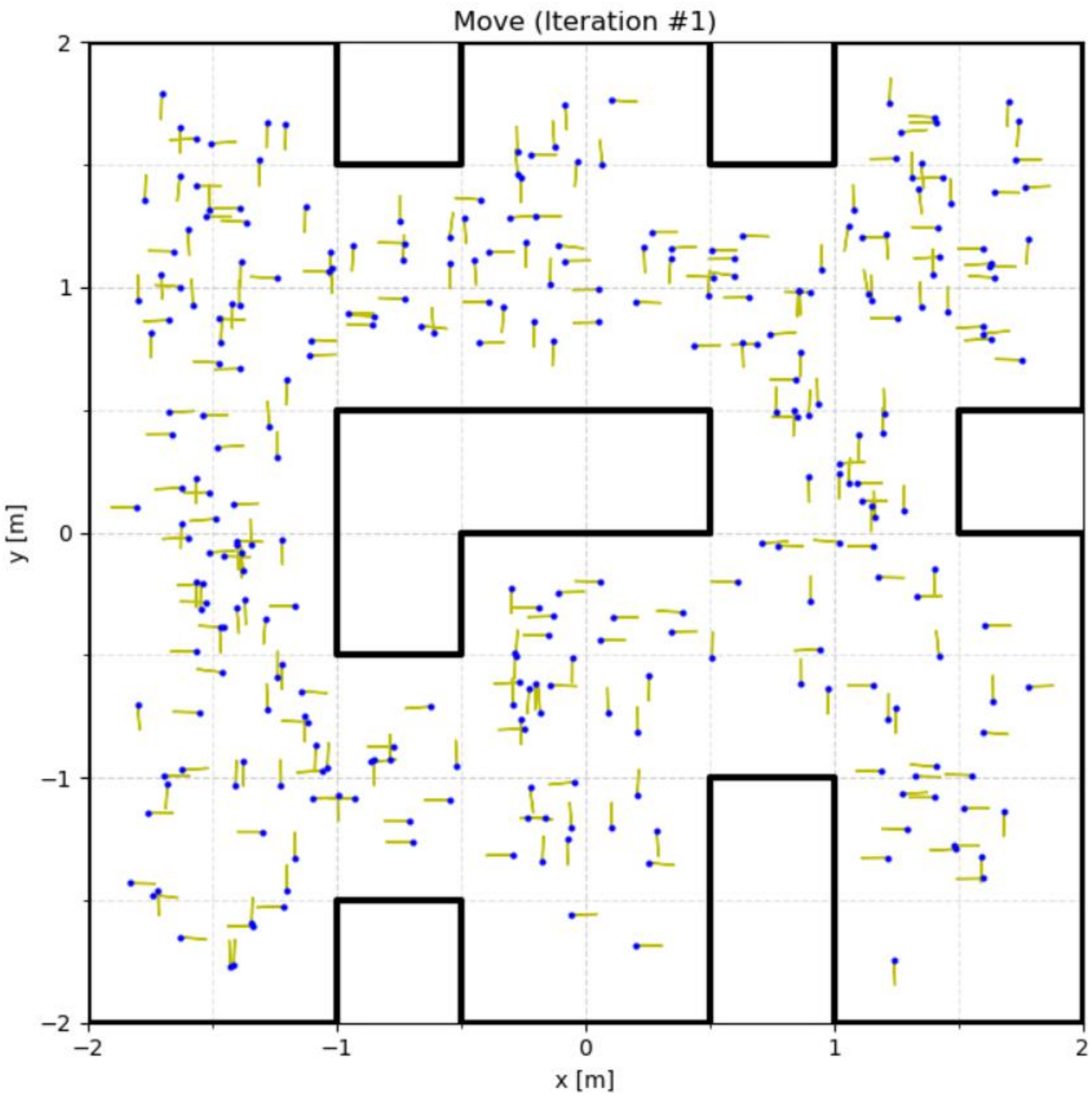


Figure 1: First iteration of particles

2.2 The particles go on a round trip

The next step for developing our localization algorithm is to make the particles move according to the robots velocities but following their direction. The velocities are not precise, since there is always a certain amount of noise in the signal, so this was taken into account in the code. We also made sure that the angle stayed in the [0, 2π) range. The exact code is shown in Code 2.

```
self._iteration += 1

for particle in self._particles:
    v_noise = v + np.random.normal(0, self._v_noise)
    w_noise = w + np.random.normal(0, self._w_noise)

    x = particle[0] + v_noise * dt * math.cos(particle[2])
    y = particle[1] + v_noise * dt * math.sin(particle[2])
    th = particle[2] + w_noise * dt

    # Check th in range [0, 2 pi)
    th = th % (2 * pi)
```

Code 2: Making the particles move (Move method)

As seen, the x and y coordinates are calculated taking into account the velocity and noise, while the direction is modified as a function of the angular velocity. The last section of the code, checks if the particle at hand has found any of the limits of the map with the check_collision method explained in the next section.

2.3 Learning to crash

The first steps were initializing the particles and making them move. However, now we want to recognize when did they crash and make the particles stop at that point. Therefore, we want to use the check_collision function from map to determine if there has been a crash, and update the position of the particle depending on the existence of an intersection between the planned movement and the map walls.

We used the following code to update the position of the particle and make sure we know when they crash.

```
intersection, _ = self._map.check_collision([(particle[0], particle[1]), (x, y)],
False)
if intersection:
    x = intersection[0]
    y = intersection[1]

particle[0] = x
particle[1] = y
particle[2] = th
```

Code 3: Stop if collision happens

So we obtain the first return value of map.check_collision which is the intersection point. Then, if we find an intersection point, the next position of that particle will be the intersection point. If not, the new position will be the one calculated in the 2.2 section.

2.4 Taking a virtual look around

Until now, we were only finding the intersections between the predicted trajectory of the robot and the map's walls. Now, we will get the measurements of the "virtual" sensors of each particles with the map. After that, we will compare those measurements with the real ones.

First, we create the "sensor" rays for each of the particles. We use the given function _sensor_rays to obtain all the segments representing the sensors with their range. After that, for each of the sensors' rays we will try to find any intersections with the map's walls and if any exists, we will save the value of that distance. In the check_collision function that belongs to the map class it's specified that if there isn't any intersection, it will return infinity.

```
def _sense(self, particle: Tuple[float, float, float]) -> List[float]:
    """Obtains the predicted measurement of every sensor given the robot's
location.

    Args:
        particle: Particle pose (x, y, theta) in [m] and [rad].

    Returns: List of predicted measurements; inf if a sensor is out of range.

    """

rays = self._sensor_rays(particle)

pred_measurements = []

for ray in rays:
    _, distance = self._map.check_collision(ray, True)
    pred_measurements.append(distance)

return pred_measurements
```

Code 4: Stop if collision happens

2.5 Everyone is important in their own way

We have to complete the function to calculate the value of the gaussian; given the mean, the standard deviation and the x value of the function.

The gaussian is defined as:

$$\frac{1}{\sigma\sqrt{2*\pi}} * e^{-\frac{1}{2}*(\frac{x-\mu}{\sigma})^2}$$

In our case, the mean is the measurements, the standard deviation is the sensor noise property of the filter and the variable is the value of the predicted measurements. We will perform this process for each pair of measurement and predicted measurement.

However, in the Gaussian function and in our case, the denominator part can be eliminated since we are normalizing the value of the weights later. Therefore, we end up only using the exponential part of the function.

```
def _gaussian(mu: float, sigma: float, x: float) -> float:
    """Computes the value of a Gaussian.

    Args:
        mu: Mean. - medida
        sigma: Standard deviation. - ruido
        x: Variable. - estimada

    Returns:
        float: Gaussian.

    """
    value = math.exp(- 0.5 * ((x - mu) / sigma) **2)

    return value
```

Code 5: Gaussian function

2.6 Do you believe your eyes?

In this method, we basically compute how close our actual measurements are to the ones the car would make if it was were the particle is at the moment. Once these are compared, a probability is assigned to the particle, depending on whether the measurements are similar or very different. This is shown in Code 5.

```
probability = 1

measurements = [2 * self._sensor_range if math.isinf(z) else z for z in
measurements]

pred_measurements = self._sense(particle)  # Predicted measurements


pred_measurements = [measure if not measure == float('inf') else 2 *
self._sensor_range for measure in
                     pred_measurements]

for measure, pred_measure in zip(measurements, pred_measurements):
    probability *= self._gaussian(measure, self._sense_noise, pred_measure)  #
Compute weight

return probability
```

Code 6: Probability calculation

It is worth noting that the conditionals and loops are written in one line in order to improve performance. We also replace the infinite values for twice the measurement range which may not be the optimal approach but it seems to work just fine.

2.7 Survival of the fittest

The last part of the algorithm is to finish the phase by resampling all the particles only at places where particles where already present. As some simulated measurements are more likely than others, we use the weights calculated earlier to pick some positions accounting for this probability.

To do this, we use the wheel method presented in class. The main difficulty is to distinguish between the particle `i` which is in the process of being resampled and the particle `index` which maybe picked. Also, special attention should be paid to the case of index = N, which doesn´t correspond to any particle (indexed from 0 to N-1). In this situation, index should be set to zero, which means a circle has been completed.

```
def resample(self, measurements: List[float]):
    """Samples a new set of set of particles using the resampling wheel
method.

    Args:
        measurements: Sensor measurements [m].
    """
    # First obtain normalized weights
    beta = 0
    new_particles = np.empty((len(self._particles), 3), dtype=object)

    N = len(self._particles)

    weights = [self._measurement_probability(measurements, particle) for
particle in self._particles]

    weights_total = sum(weights)
    weights_normalized = [w / weights_total for w in weights]  # normalize
weights

    index = int(np.random.uniform(0, N))
    weight_max = max(weights_normalized)
    #  Then iterate to pick the particle with the wheel algorithm
    for i in range(0, N):
        beta = beta + random.random() * 2.0 * weight_max
        while beta >= weights_normalized[index]:
            beta = beta - weights_normalized[index]
            index = (index + 1)
            if index == N:  # to restart the index
                index = 0
        new_particles[i] = self._particles[index] # the particle i has been
picked

    self._particles = new_particles  # the particles of the object are now
the ones that have been picked
```

Code 7: resample method

2.8 Sensitivity analyses

1. With the base parameters, the predicted position of the robot is shown in the figure below:

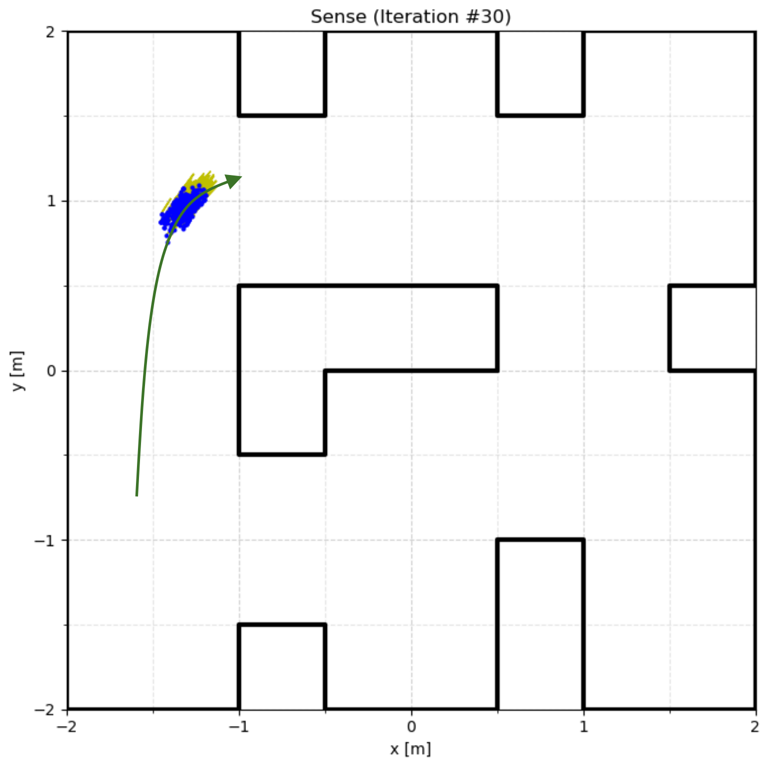
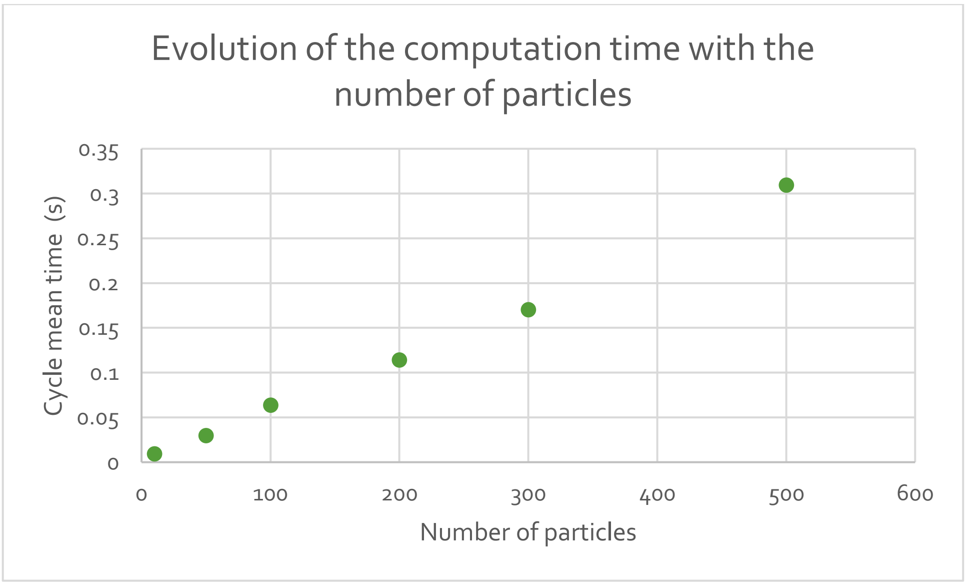


Figure 2: Predicted position of the robot with base parameters

2. In the particle_filter.py file, a test is performed after the class definition. It is possible to change the number of particles in the following line:

```
pf = ParticleFilter(m, RobotP3DX.SENSORS[:8], RobotP3DX.SENSOR_RANGE,
particle_count=500)
```

Code 8: Instantiation of the ParticleFilter class for the test



Graph 1: Evolution of computation time

Its hard to decide which is the optimal number. After several tests, 300 particles seemed to have a reasonable balance between accuracy and computing time, although it sometimes missed if not enough particles were generated initially in the real position of the car.

3. If the noise values a reduced to a value close to 0, the algorithm already finds the position in the second iteration. This is because the measurements are so close to the real ones that it gives a huge weight to the particle at hand, however if it decides to "bet" on a particle far from the real position, it will never backtrack and a localization error will follow. On the other side, if we try to increase the noise we get highly unpredictable particles whose measurements aren't very reliable, therefore we take a higher number of iterations to reach the location (if the noise is high enough we will not even get a correct result anyways). This shows the importance of correct measurements and trying to reduce as much as possible the noise when dealing with the real world.

3 Annex: Using C to calculate intersections

Due to the huge amount of times the segment_intersect function is called, the use of a C function call really helps the execution of the code get faster, achieving times of around 250ms for cycle. The library used to connect C and python is a native package of python (in the latest versions), called ctypes. Ctypes allows to connect python to almost every other programming language in an easy and simple way. First, we create a function to obtain the intersection of two segments in case there is one.

```
#include <stdbool.h>
#include <math.h>

int intersect(float p0_x, float p1_x, float p2_x, float p3_x, float p0_y, float
p1_y, float p2_y, float p3_y, float *point) {

   float s02_x, s02_y, s10_x, s10_y, s32_x, s32_y, s_numer, t_numer, denom, t;
   s10_x = p1_x - p0_x;
   s10_y = p1_y - p0_y;
   s32_x = p3_x - p2_x;
   s32_y = p3_y - p2_y;

   float d_x, d_y;

   denom = s10_x * s32_y - s32_x * s10_y;
   if (denom == 0){
      point[0] = 0.0;
      point[1] = 0.0;
      return 0; // Collinear
   }
   bool denomPositive;
   denomPositive = denom > 0;

   s02_x = p0_x - p2_x;
   s02_y = p0_y - p2_y;
   s_numer = s10_x * s02_y - s10_y * s02_x;
   if ((s_numer < 0) == denomPositive){
      point[0] = 0.0;
      point[1] = 0.0;
      return 1; // No collision
   }
   t_numer = s32_x * s02_y - s32_y * s02_x;
   if ((t_numer < 0) == denomPositive){
      point[0] = 0.0;
      point[1] = 0.0;
      return 1; // No collision
   }

   if (((s_numer > denom) == denomPositive) || ((t_numer > denom) ==
denomPositive)){
      point[0] = 0.0;
      point[1] = 0.0;
      return 1; // No collision
   }
   // Collision detected
   t = t_numer / denom;
   point[0] = (p0_x + (t * s10_x));
   point[1] = (p0_y + (t * s10_y));

   d_x = point[0] - p0_x;
   d_y = point[1] - p0_y;


   point[2] = sqrt(pow(d_x,2) + pow(d_y,2));


   return 2;
 }
```

In this function, we pass each coordinate of the four points forming the two segments and a point array of length three. The first two for the x and y coordinates of the intersection, and the third one for the distance. We first calculate the displacement in x and y of the segments, and then we perform some calculations in order to find if they are collinear, they don't collide or if they collide. The return value is there in case we want to perform some kind of action in the python code with that information.

When the C code was finished, it was compiled into a dll file in the case of Windows systems, and a so file in the case of MaxOS systems.

Then, we created a cConst.py file in our project, that will hold all the constants used for the connection with C, as well as the declaration of variables and the initialization of the library.

```
import ctypes
import platform
import os

file_extension = '.so'
if platform.system() =='cli':
    file_extension = '.dll'
elif platform.system() =='Windows':
    file_extension = '.dll'
elif platform.system() == 'Darwin':
    file_extension = '.so'
else:
    file_extension = '.so'
path = os.path.join(os.path.dirname(__file__), 'cpsr' + file_extension)

intersect_in_c = ctypes.CDLL(path)

intersect_in_c.intersect.argtypes = (ctypes.c_float, ctypes.c_float,
ctypes.c_float, ctypes.c_float, ctypes.c_float, ctypes.c_float, ctypes.c_float,
ctypes.c_float, ctypes.POINTER(ctypes.ARRAY(ctypes.c_float,3)))
```

```
point = (ctypes.c_float * 3)()


inter = intersect_in_c.intersect
```

First, we import the necessary packages and we import the library we created on the format suitable for each OS. Then, with the CDLL method of ctypes we create the intersect_in_c library in python and import the intersect function as a ctypes function. Then, we declare all the arguments the function will get (keep in mind that the point is treated as a pointer of an array so we can get the 3 values we need from the C function). Then we create the point as an empty array and save the method as "inter" in order to accelerate the execution of the code. This last step is unnecessary and it only provides a really small improve on performance.

Then, on the segment_intersect function of the Intersect class, we call the function, retrieve the results and pass them to whatever function may call it.

```
inter(segment1[0][0], segment1[1][0], segment2[0][0], segment2[1][0],
segment1[0][1], segment1[1][1], segment2[0][1], segment2[1][1], point)

if point[0] == 0.0 and point[1] == 0.0:
    return None
else:
    return point
```