Pathfinder solver using the CiberRato simulation environment

Pedro Santos (76532) and Ruben Kins (92078)

Universidade de Aveiro, Aveiro, Portugal

Abstract. This document summarizes the development of a software project that allows a robot in a simulated environment to explore the map and return back to the starting position after finding a special token. The project was a success and the robot could master all the challenges given in the example course. The chosen programming language was Python 2.7.

Keywords: Agent · robots · Python · Kalman Filter · pathfinding

1 Project Development

This report contains the overall approach for the solutions to the challenges. In this report the software architecture is explained as well as the modules that are all designed to deal with a specific problem. The world the robots navigates in is continuous even though the map is equally divided into quadratic cells. The robot perceives its surroundings through a compass and distance sensors as well as a sensor for the token detection. The sensor signals and the motor inputs used to drive the wheels all contain noise. The challenges the robot has to complete are:

1. Movement: The robot should be able to move towards a given point without colliding with a wall
2. State estimation: The robot should have an accurate estimation of its current state, even with noisy sensor signals. This ensures that the robot acts appropriately.
3. Map building and exploration: The robot should explore its surroundings and store the results.
4. Search: After finding the token the robot must return to its starting position as quickly as possible

In the basic world model that was developed for this project the world is divided into a continuous and a discrete part. The continuous part deals with the basic functions such as movement and sensor signal processing and the discrete part manages exploration and pathfinding. The discrete part only knows about the nodes (the quadratic cells) and moves the robot from node to node. The overall design approach is shown in figure 1. In the following chapters every system part and its purpose is explained in detail.

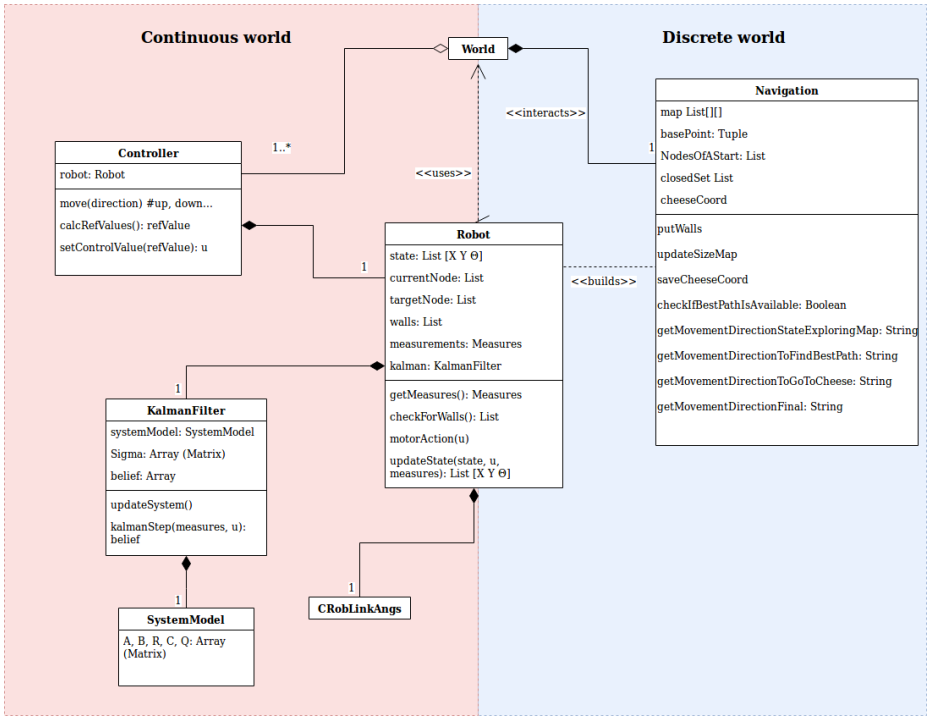

Fig. 1. Class structure of the project

2 Continuous World

2.1 Controller

The controller is the topmost part in the hierarchy of the continuous world. It accepts commands from the navigation algorithm to move one cell up, down, left or right. To do this, the controller compares the coordinates of the robot with the ones of the target node given by the move command. The results are two values called the raw distance Δ and the angle Θ_{ref} of the node.

$$x = [X \quad Y \quad \Theta]^T \tag{1}$$

$$\Delta = \sqrt{(X_{node} - X)^2 + Y_{node} - Y)^2} \tag{2}$$

$$\Theta_{ref} = atan2(Y_{node} - Y, \quad X_{node} - X) \tag{3}$$

The distance and angle difference are then fed into two separate proportional P-controllers (cf. figure 2), one for angular control and one for distance control. P controllers were chosen for their simplicity and robustness as they are time-

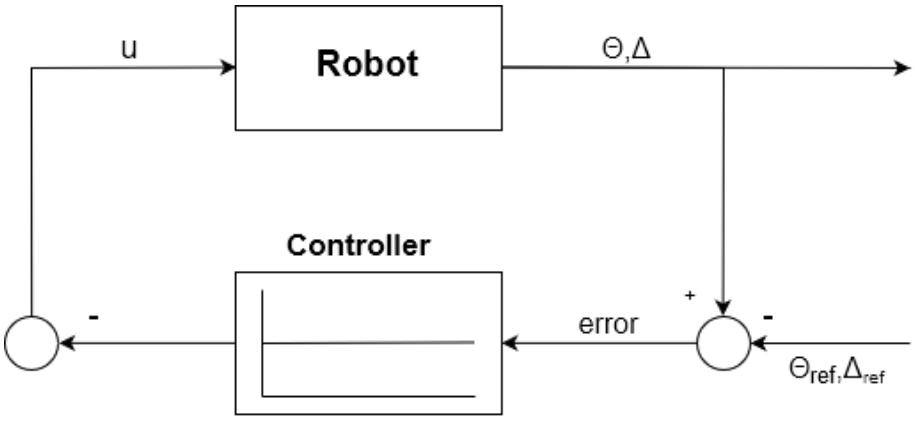

Fig. 2. Control Loops used for moving the robot

invariant. Also, no integrating element is needed for static accuracy as the robot is an integrator itself: The controlled robot position is the integral over the wheel velocities that are the control values. The outputs of the two separate control loops are added together to form the motor power input u:

$$u = [u_{right} \quad u_{left}]^T \tag{4}$$

$$u_{right} = -P_\Delta * \Delta - P_\Theta * (\Theta - \Theta_{ref}); \quad u_{left} = -P_\Delta * \Delta + P_\Theta * (\Theta - \Theta_{ref}) \tag{5}$$

Both controller parameters P_Δ and P_Θ have been carefully tuned so that the closed-loop system has a damping ratio of 0.707. This means that there is no control overshoot happening as the step response shown in figure 3 proves. Other aspects of the control loop that are required for stability are:

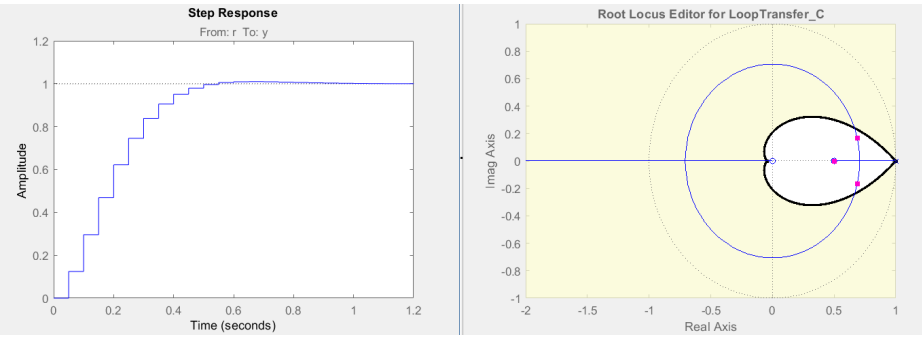

Fig. 3. Parameter tuning for minimal overshoot

1. The forward control is disabled if the angle error is too high in order to avoid interference between the two control loops.
2. The output values of the controllers are limited because the control variable u (the motor power input) is also bounded.
3. Once the robot reaches a tolerance area around the target node, the controllers get disabled because correcting those small errors takes up much time.
4. The robot can only accept new commands after reaching the new node. If the new command leads to a collision it will return to its previous state.

2.2 Robot

The Robot class is a passive element that mediates between the other components of the software and keeps track of its own states. The robot communicates with the server using the already given class 'CRobLinkAngs'. The four distance sensors of the robot are all aligned in 90 degree angles between each other for a quick perception of its environment without any movement required. The main task of the robot class is getting measurement data from the sensors and processing it in a way that other classes can use the data. The robot also keeps track of its states (X and Y position, rotation) using the attached Kalman Filter. This process will be discussed later. The last thing the robot does is interact with its environment by driving the wheel motors with the values given by the controller.

The measurement of the compass has a range from -180 to 180 degrees. The problem with that is that the value is not continuous because the value can change rapidly after a 180 degree turn. To solve this problem, the robot maps the sensor output to full rotations with multiples of 360 degrees using the memory of its current angle and assuming that a 180 degree turn in one time step is impossible.

The biggest challenge is processing the data from the infrared sensors. The data from the distance sensors is used for measuring the robots position and also to detect walls. It is important to know that the sensor output is the inverse value of the distance. If the robots orientation is not close to multiples of 90 degrees the

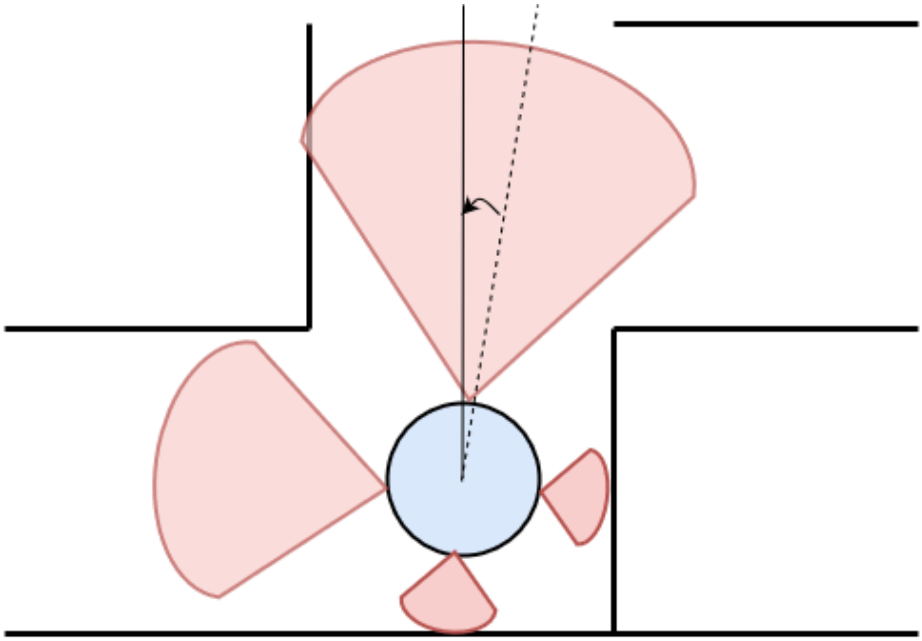

Fig. 4. Infrared sensors of the robot

sensors are disabled because the measurements are unreliable. The reason for the unreliability is the fact that it is not clear which wall the sensor is detecting. For a better data processing the sensor outputs are mapped to a global position that is invariant to the robots local orientation. To do this, the list entries containing the measurement data are simply rotated around. For example, after processing the first entry in the list is always the sensor pointing north. The sensor data is still incompatible with the robots state because the sensors read the distance to walls and not the global position. The solution is to use the coordinates of the current cell the robot is in to calculate global coordinates. For that to work the robots estimate of its position should never be off by more than a full cell.

Also important for building a map of the environment is information about the walls surrounding the current cell. A wall is detected when the sensor facing the wall detects a distance lower than a threshold value. The threshold is dynamic, meaning that it also considers the robots position within the cell as well as the orientation. The dynamic threshold makes sure that walls are only detected when the robots situation allows a stable measurement.

2.3 Kalman Filter

The general algorithm presented in this chapter is based on chapter 3 of the book 'Probabilistic Robotics' by Thrun, Burgard et al. The task of the Kalman Filter

is to estimate the state x of the robot at any time step given the information the robot has about:

1. Its state in the last time step μ_{t-1} and Σ_{t-1},
2. the actions u (in this case motor commands) the robot takes and
3. the measurements z_t from the sensors (compass and IR).

The Kalman Filter approach is chosen because it is an effective way of dealing with noise in the sensor measurements. It can also use the existing knowledge of the system dynamics for predictions of future states. Another advantage is that is does not need any information about the map unlike the particle filter.

At first, the Algorithm assumes that the next state is a result of the last state, actions and random noise:

$$x_t = A_t x_{t-1} + B_t u_t + \epsilon_t \tag{6}$$

In this case, the full system dynamics based on the information given can be expressed as:

$$\begin{bmatrix} X_t \\ Y_t \\ \Theta_t \\ o_r \\ o_l \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \frac{\cos\Theta_{t-1}}{4} & \frac{\cos\Theta_{t-1}}{4} \\ 0 & 1 & 0 & \frac{\sin\Theta_{t-1}}{4} & \frac{\sin\Theta_{t-1}}{4} \\ 0 & 0 & 1 & 0.5 & -0.5 \\ 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} X_{t-1} \\ Y_{t-1} \\ \Theta_{t-1} \\ o_{t-1} \\ o_{t-1} \end{bmatrix} + \begin{bmatrix} \frac{\cos\Theta_{t-1}}{4} & \frac{\cos\Theta_{t-1}}{4} \\ \frac{\sin\Theta_{t-1}}{4} & \frac{\sin\Theta_{t-1}}{4} \\ 0.5 & -0.5 \\ 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} u_r \\ u_l \end{bmatrix} + \epsilon_t \tag{7}$$

In this case the noise only affects the motor outputs. The robot is not a linear system because the system matrix contains values that are not constant. Upon closer inspection one can see that only values from the past time step are needed to establish the system matrix. Thus, the system matrices simply need to be updated after every prediction step. The Kalman Filter also assumes a linear measurement model that is affected by additive noise:

$$z_t = C_t x_t + \delta_t \tag{8}$$

The measurement model of the robot results to:

$$\begin{bmatrix} z_{up} \\ z_{right} \\ z_{down} \\ z_{left} \\ z_{compass} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} X_t \\ Y_t \\ \Theta_t \\ o_t \\ o_t \end{bmatrix} + \delta_t \tag{9}$$

The matrix C as shown above is only valid if all IR sensor measurements are passed through. It needs to be dynamically changed based on the number of currently valid measurements. The full Kalman Filter algorithm is shown in figure 5. The algorithm first calculates a belief based on system dynamics and then updates the belief based on the measurements. A matrix called the Kalman Gain plays an important role in weighting both sources of information against

Algorithm Kalman_filter($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$)**:**

$\bar{\mu}_t = A_t\ \mu_{t-1} + B_t\ u_t$

$\bar{\Sigma}_t = A_t\ \Sigma_{t-1}\ A_t^T + R_t$

$K_t = \bar{\Sigma}_t\ C_t^T (C_t\ \bar{\Sigma}_t\ C_t^T + Q_t)^{-1}$

$\mu_t = \bar{\mu}_t + K_t(z_t - C_t\ \bar{\mu}_t)$

$\Sigma_t = (I - K_t\ C_t)\ \bar{\Sigma}_t$

return μ_t, Σ_t

Fig. 5. Kalman Filter algorithm

each other using information about the noise both contain. The result is called the belief, it is the best estimate of the current robot state. The second result - Σ - contains the noise or uncertainty of the estimate. The noise of the robots states and measurements is often non-gaussian which means that the variance of the noise has to be estimated.

3 Discrete World

This section contains the reasoning behind the developed solution for the robot to be able to find the token and then return to the initial point going the best possible path. The developed agent has 4 main states and each one corresponds to the current navigation goal of the agent:

1. Exploring the map
2. Exploring the map after finding the token
3. Returning to the token
4. Returning to the initial point

In the figure 6 is the state diagram of the agent.

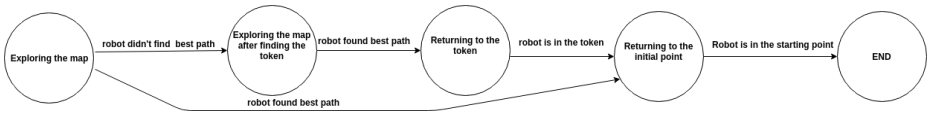

Fig. 6. State Diagram

In each cycle the agent will only take any action if it is centered in the cell. The agent will take an action according to the current state it is in (see list above). In the next subsection the behavior of the robot and what was done to overcome the challenge will be explained.

3.1 The Map

Before entering the first cycle the agent instantiates the objects needed in the continuous world and for the discrete world it instantiates an object of the class Navigation. In the instance of Navigation several attributes of the class are initialized and the map is built. The maximum size of the arena is 7 cells tall and 14 cells wide and in the beginning the position of the robot is unknown. For that reason a bi dimensional list is created that has 13 lists which have 27 elements each. Each element is an object of the class Node. Each node has the necessary attributes for saving the position, the walls and other attributes that will be used later to find the best path. The "middle" object of this bi dimensional list will be the point (0,0) that corresponds that corresponds to the robots initial position. Each list in Map corresponds to a row in X direction and each list has the nodes between the X positions -13 and 13. For example the position of the node of the bottom right corner is (13,-6), which corresponds to the entry map[12][26]: It is the 26th entry of the 13th list. The node of the the top left corner is (-13,6) to which the corresponding map element is map[0][0]. Each one of these elements corresponds respectively to the last element and the first element of the bi dimensional list.

To handle the construction of the map three main functions were created. The first one is the function that receives the currentNode position and that will return the location of that node in the bi dimensional list. This function is called getElementLocation(). So, taking the previous example, when the node (13,-6) is passed to the function as an argument it will return the tuple (12,26) that corresponds to the nodes location in the bi dimensional list. Basically, the function converts the coordinates of the world to the coordinates of the list that contains the map. The other function created is updateSizeMap(). This function has one argument that is the current position of the agent in the world and by knowing the position of the robot it will delete the nodes in the bi dimension list that are impossible to reach because the maximum size of the map is 7-cells tall and 14-cells wide. For example, if the robot goes up one cell it will delete the last list from the bi dimensional list because it can be concluded that it is not possible to reach these nodes. The last main function to update the map is putWalls(). This function is used to save the walls surrounding each cell. The function has two arguments: The current nodes position and the list containing the information about whether there is a wall or not in each direction. This function also will also update the neighboring cells of the current cell. For example, if there is a wall in the north direction, it is possible to know that the cell above the current position has a wall to the south. Doing this makes the exploration more effective.

3.2 Exploring the Map

When the agent is exploring the map it will save the currently detected surrounding walls using the function putWalls() and it will update the size of the map using updateSizeMap(). During the next step it will check if it found the

token, in this case the cheese. If it didn't find the token the agent calls the function getMovementDirectionStateExploringMap(). This function will return the direction that the agent has to go to explore the map. To calculate the value to be returned the function checks the unknown neighbours of the current Node. Then the agent will choose the neighbour to go to using the following priority: north,east,south and west. So, if the agent has three options to choose and one of them is going north, it will go north. It is good to have static rules so that the behavior of the robot is predictable. Other implementations, such as giving priority to the current orientation of the robot, were also tested. However, in the tests they didn't show obvious advantages and with this approach the map will be explored from top to bottom, with the other approach was not so predictable the exploration of the agent. Then, if there are no unknown neighbours in the current position it will perform an A* search to the closest node that still has unknown neighbors. The agent will explore the map like this until it finds the token.

When the agent finds the token it will save the position of the cheese on the map and it will turn on the visiting LEDs and check if the best path possible was found. If the best possible path was found it will activate the state 'Returning to initial point' immediately, but if the best possible path was not found it will go to the state 'Exploring the map' after finding the token. The function that decides whether the returning path is optimal is called checkIfBestPathIsAvailable(). It is a very complex function that is explained in the next section. This function is of special importance because it yields the highest potential for saving time while the robot is returning. The function performs three verifications after applying the A star algorithm for the path between the token position and the initial position. The first verification is to check whether the size of the best possible path using the information of the currently explored map is equal to the Manhattan distance between the initial point and the token. It is then possible to conclude that currently there is no best path. The other verification checks if all the nodes in the closed Set of A* have neighbors that are known. The last verification is to verify whether the nodes with unknown neighbours in the closed set the one with lower fCost and hCost is the initial point. The fCost corresponds to the sum between the hCost (the Manhattan distance between the node and the start position) and Gcost (the mahantam distance between the node and the start position). If none of the verifications are true the best path was not yet found and the agent continues exploring the map.

3.3 Exploring the map after finding the token

In this state the agent also updates the map by using the functions putWalls() and updateSizeMap(). Then it will check if it found the best possible path like in the previous state. If an optimal solution is found it will go to the state 'Returning to token'. If it doesn't find the path it will explore the map until finds the best path.

The solution consists in analyzing the nodes that are in the closed set when the the A star search is ran to find the best path from the token to the ini-

tial point. This algorithm saves the nodes that were analyzed and can not be used for the best path in a list called closedSet. However, in this case the map is not totally explored, so if these nodes have neighbours that are not known maybe the best path goes through these nodes. The function getMovementDirectionToFindBestPath() is used to calculate the direction to go when trying to find the best path. It will do the algorithm A* to find the best available path between the token and the initial position and save the nodes from the closedSet. Then, it will iterate that list and it will find the node that has the lowest fCost and if there are multiple nodes with the minimum fCost it will select the one with a lower hCost. This means that will select the node that is nearest to the initial point from those who have the lowest fCost. Then, it will perform the algorithm A* to find the path from the current location to the calculated node and then the agent will go to that node. Then, when it is in the node with a lower cost it will calculate the Manhattan distance for each unknown neighbor to the initial point and it will go to the one with lower distance. This process repeats until it finds the best path.

3.4 Returning to token

In this state the agent will go to the token using the path from the A* algorithm, to be able to perform the best possible path from the token to the initial point.

3.5 Returning to initial point

This is the last state. The robot it will go from the token to the initial position using the best path possible.

4 Results

The robot was able to master all given challenges without any major problems in the test environment. The robot is able to navigate safely from node to node due to the robust P-controller action. Noise in both motors and sensors can be reliably filtered using the designed Kalman filter. The designed map building ensures that the map is explored until the cheese is found. It also makes sure that not only some path is found, but the optimal one. All algorithms combined make for a quick and robust navigation of the robot through the labyrinth.