Pathfinder solver using the CiberRato simulation environment

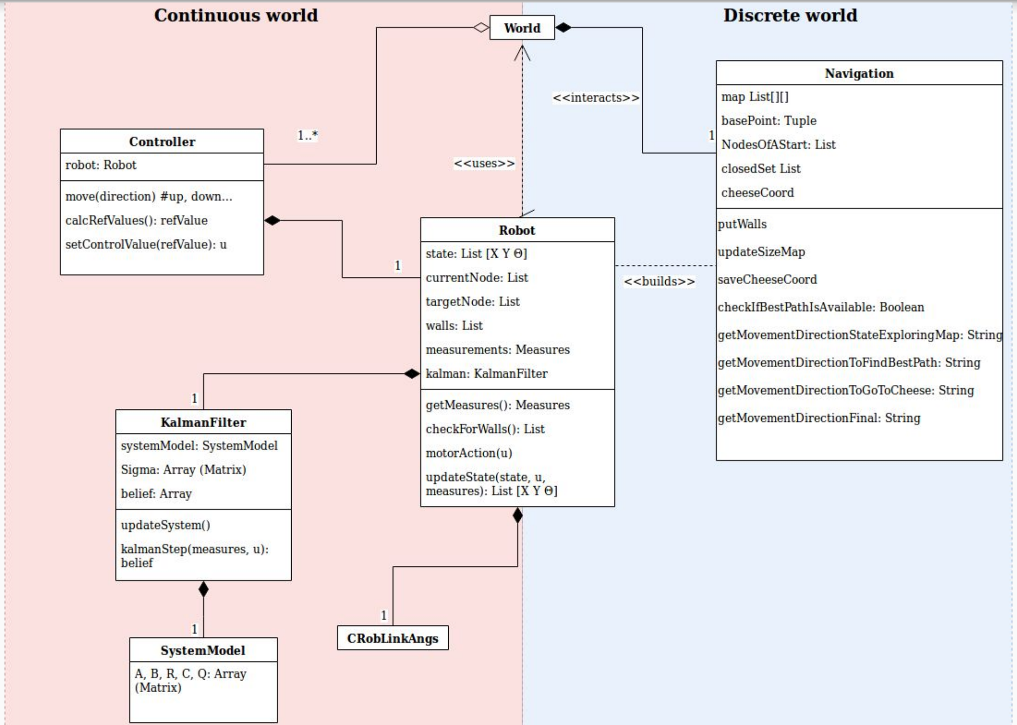
RMI - Robótica Móvel Inteligente

Assignment 2

Pedro Santos, 76533
Ruben Kins, 92078

Overview - Code Structure

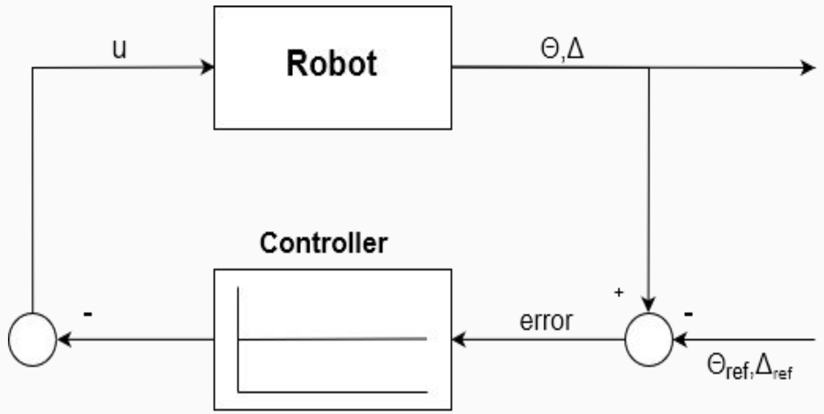
- Continuous and discrete world
- Continuous:
  - Movement between cells
  - Motor control
  - Sensor Fusion
- Discrete:
  - Map building
  - Movement cell to cell
  - Pathfinding



Continuous World

Movement between cells

- Controller function: move(direction)
- Two control loops: Angular error (θ) and distance error (Δ)
- P-controllers: Robust, simple, robot integrator itself
- Control input: Motor power
    - $u_{right} = -P_{\Delta} * \Delta - P_{\Theta} * (\Theta - \Theta_{ref})$
    - $u_{left} = -P_{\Delta} * \Delta + P_{\Theta} * (\Theta - \Theta_{ref})$

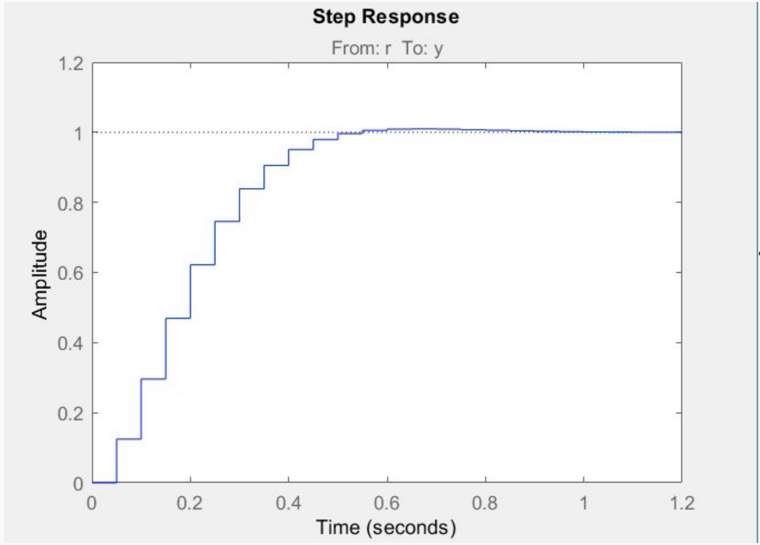


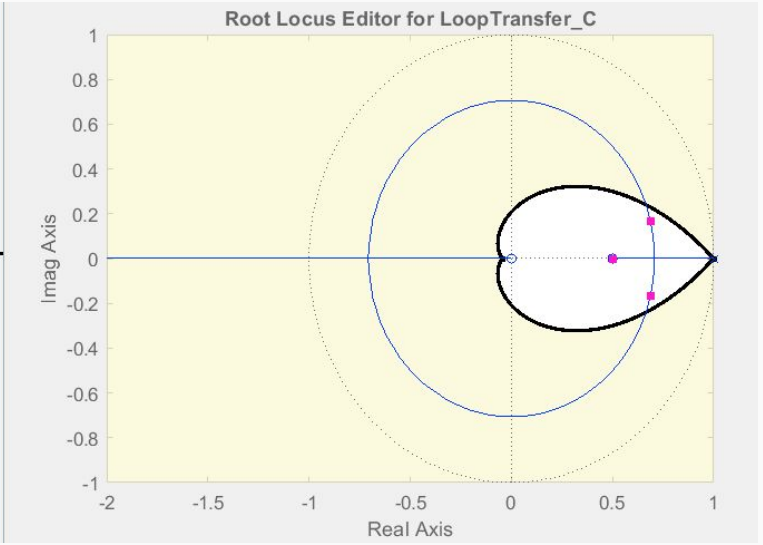
Other considerations:

- No forward movement when Θ
- Motor power is limited
- Disable controller if Δ below tolerance

Controller tuning - system dynamics

Results of simulation and tuning using Matlab:

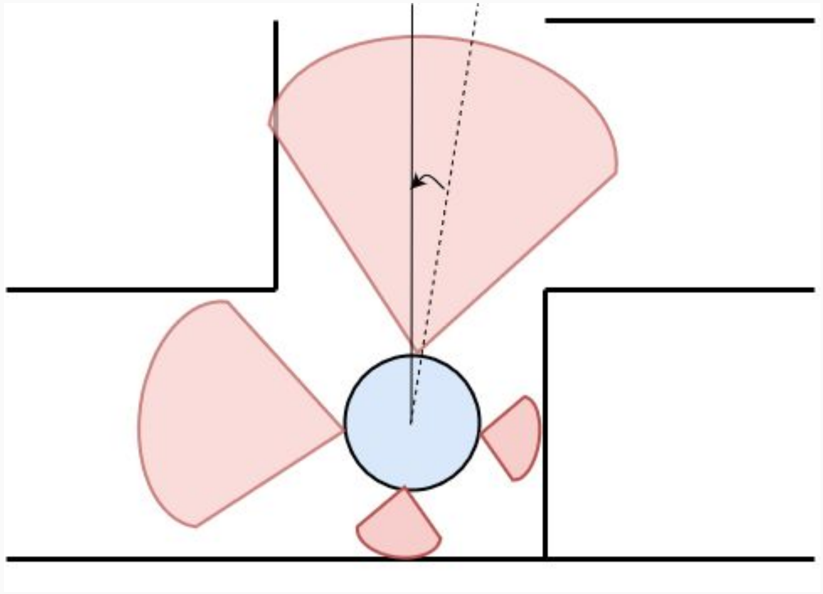




Sensors - IR Sensors

Transformations of Sensor data:

1. Distance is inverse of measurement
2. Map robot orientation to global orientation
3. Wall detected when distance is below dynamic threshold
   - Only measure when robot is aligned
4. Measurement: Distance to wall
   - Map to global coordinates using cell information



Sensor Fusion - Kalman Filter

1. Make prediction based on system dynamics, last states and control actions (Added Noise!):

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

Technically a nonlinear system: Update System dynamics after each time step

Sensor Fusion - Kalman Filter

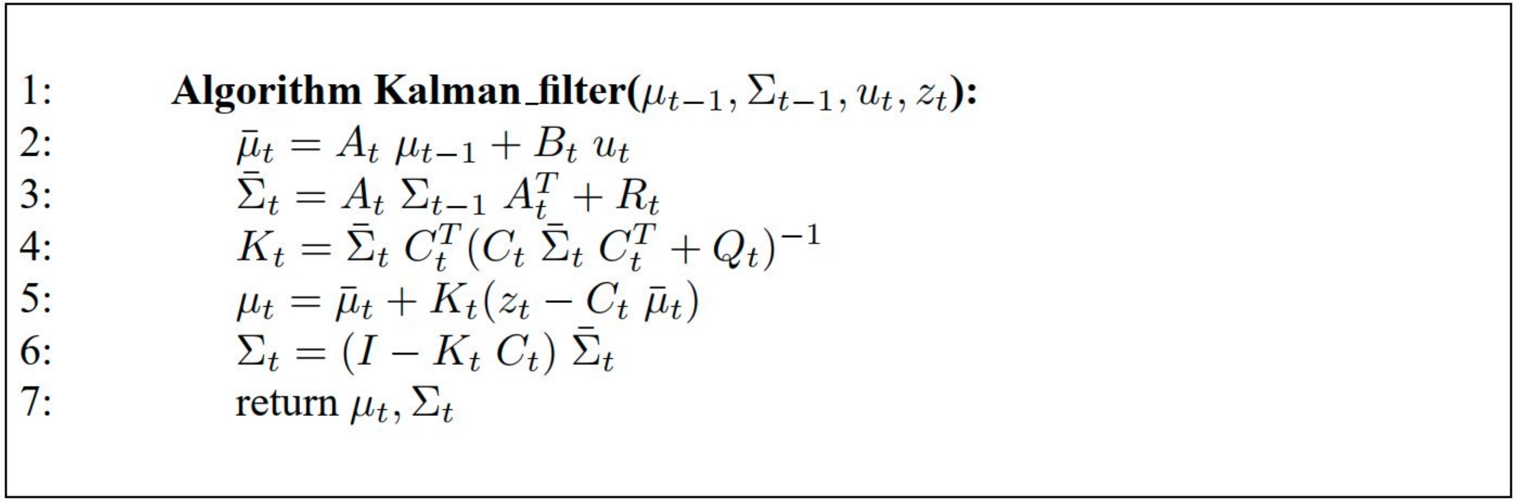
2. Get information about states using sensor input (With Noise):

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

Matrix size changes based on available measurements

Sensor Fusion - Kalman Filter

3. Fuse prediction and measurement together using Kalman Gain:



```
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 $\mu_t, \Sigma_t$
```

Source: 'Probabilistic Robotics', Thrun, Burgard et al.

Discrete World

Challenge/Goal

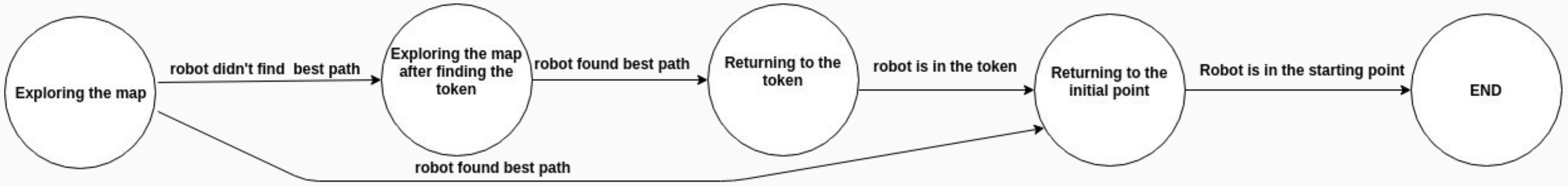
The challenge is to develop an agent to explore an unknown world where there is a token that must be found and then return to the starting point using the best path possible.

After finding the token the robot can explore more the map.

States

The agent has 4 main states:

- Exploring the map
- Exploring the map after finding the token
- Returning to the token
- Returning to the initial point


Exploring the map
robot didn't find best path
Exploring the map after finding the token
robot found best path
Returning to the token
robot is in the token
Returning to the initial point
Robot is in the starting point
END
robot found best path

The Map

- Maximum size of map is 7 cells tall and 14 cells wide
    - 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 and the walls

Functions to build Map

- Function to receive the node position and it will return the location of that node in the bi dimensional list.
- Function that by knowing the current position of the agent in the world 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.
- Function to save the walls surrounding each cell

State - Exploring the Map

- Update the information of the walls
- Update the size of the map
- Check if it found the token
- Get the decision of the movement to perform

```
putWalls()
updateSizeMap()
If( found the token):
        if( check_if_found_the_best_path() )
                state=Returning_To_Base
        Else:
                state=Exploring_Map_After_Finding_Token
Else:
        controller.move(get_MovementDirection(robot.currentNode))
```

get_MovementDirectionToExploreMap()

- Calculate the unknown neighbours of the current Node
- Agent will choose the neighbour to go to using the following priority: north,east,south and west.
- 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

check_if_found_the_best_path()

Three verifications after applying the A* algorithm for the path between the token position and the initial position.

- 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.
- Checks if all the nodes in the closed Set of A* have neighbors that are known.
- Checks whether the nodes with unknown neighbours in the closed set the one with lower fCost and hCost is the initial point.

State - Exploring the map after finding the token

- Update the information of the walls
- Update the size of the map
- Check if it found the best possible path
- Get the decision of the movement to perform

```
putWalls()
updateSizeMap()
if( check_if_found_the_best_path() )
        state=Returning_To_Token
Else:
        controller.move(get_MovementDirectionToFindBestPath(robot.currentNode))
```

get_MovementDirectionToFindBestPath()

- Performs the algorithm A* to find the best available path between the token and the initial position and save the nodes from the closedSet.
- Finds the node from the closedSet that has unknown neighbours and has the lowest fCost and if there are multiple nodes with the minimum fCost it will select the one with a lower hCost.
- 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.
- It will calculate the Manhattan distance for each unknown neighbor to the initial point and it will go to the one with lower distance

State - Returning to the token

- The agent will go to the token
- The function getMovementDirectionToGoToCheese() will use the A* to calculate the best path to go from the current position to the token and returns the direction of the movement that must be done to reach the target position

```
If(robot.currentNode == navigation.cheeseCoord):
    State = Returning_To_Initial_Point
Else:
    controller.move(getMovementDirectionToGoToCheese(robot.currentNode)
)
```

State - Returning to the initial point

- The agent will go to the token
- The function getMovementDirectionToGoToCheese() will use the A* to calculate the best path to go from the current position to the initial point and returns the direction of the movement that must be done to reach the target position

```
If(robot.currentNode != [0,0]):
        getMovementDirectionFinal()
Else:
        finish()
```

Results

- 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 exploration of the map is effective and the algorithm designed to find the best path ensures that the best path is efficiently found.