StateMachine Class

This document will explain the class' general operation and how to use it. If you want more details, the code is available and commented in order to facilitate your comprehension.

First, the general operation: The class initializes to turn to the left, current state as initial and the Fuzzy controller to its default configuration (check FuzzyController's folder for more information). If the robot finds something in front of it, it will move forward until the points contained into the models is more than 1m away, this way it knows that it has to change lanes. After that, it will turn clockwise until its angle is around -40º and then goes backward. When the intercepts from the second left closest model and closest model sums to around 0, it means the robot is in the middle of the two lines, so it begins to turn anti-clockwise until its angle is around -18º, then merges to the lines using fuzzy controller, going the opposite direction. When going backwards, the robot repeats the same logic, but changes the direction of linear motion, this is made because the turning is different depending on the direction it was moving before.

For how to use it, it is really simple. Firstly, declare a StateMachine object passing "Number of Lines" to inform how many models will be passed later, "Number of times to turn" to inform how many times the robot will turn, "Max Velocity" to set robot's max velocity, and "Body Size" to inform how long is the robot. Then, to calculate the output and make the state transition, just use the method "makeTransition" passing a std::vector<Model> as the parameter. The machine is coded this way because every finite state machine needs a clock to work properly, and this way this machine has a "custom clock", depending on the frequency "makeTransition" is called.

Public Methods

- StateMachine( )

```
StateMachine::StateMachine( const int NLines,
                            const int NTimesTurn,
                            const double Bsize
                          )
```

Default constructor that assigns "Number of Lines", "Number of times to turn" and "Body size"

Parameters

NLines Number of models passed in the vector (must be pair)
NTimesTurn Number of times to turn
BSize Robot's length

- makeTransition( )

```
std::pair<double, double> StateMachine::makeTransition
                          (
                            const std::pair<Model, Model> & models,
                            const double dist
                          )
```

Method to calculate the output depending on the state the machine is in, and make a state transition. The output being a pair where the first value is the left wheel's target velocity and the second value the right wheel's.

Parameters

models is the vector of models found
dist is the distance between 2 adjacent lines

Private Methods

◆ initialStateRoutine( )

Transition StateMachine::initialStateRoutine(std::vector<Model> & models)

Initial state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ forwardStateRoutine( )

Transition StateMachine::forwardStateRoutine(std::vector<Model> & models)

Forward state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ backwardStateRoutine( )

Transition StateMachine::backwardStateRoutine(std::vector<Model> & models)

Backward state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ linearStopStateRoutine( )

Transition StateMachine::linearStopStateRoutine(std::vector<Model> & models)

Linear stop state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ angularStopStateRoutine( )

Transition StateMachine::angularStopStateRoutine(std::vector<Model> & models)

Angular stop state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ leftTurnBeginStateRoutine( )

Transition StateMachine::leftTurnBeginStateRoutine(std::vector<Model> & models)

Left turn begin state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ leftTurnMidStateRoutine( )

```
Transition StateMachine::leftTurnMidStateRoutine
(
    std::vector<Model> & models,
    const double dist
)
```

Left turn mid state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

dist is the distance between 2 adjacent lines

◆ leftTurnMergeStateRoutine( )

Transition StateMachine::leftTurnMergeStateRoutine(std::vector<Model> & models)

Left turn merge state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ rightTurnBeginStateRoutine( )

Transition StateMachine::rightTurnBeginStateRoutine(std::vector<Model> & models)

Right turn begin state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ rightTurnMidStateRoutine( )

```
Transition StateMachine::rightTurnMidStateRoutine
(
    std::vector<Model> & models,
    const double dist
)
```

Right turn mid state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found
dist is the distance between 2 adjacent lines

◆ rightTurnMergeStateRoutine( )

```
Transition StateMachine::rightTurnMergeStateRoutine(std::vector<Model> & models)
```

Right turn merge state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ endStateRoutine( )

```
Transition StateMachine::endStateRoutine(std::vector<Model> & models)
```

End state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

◆ impossibleStateRoutine( )

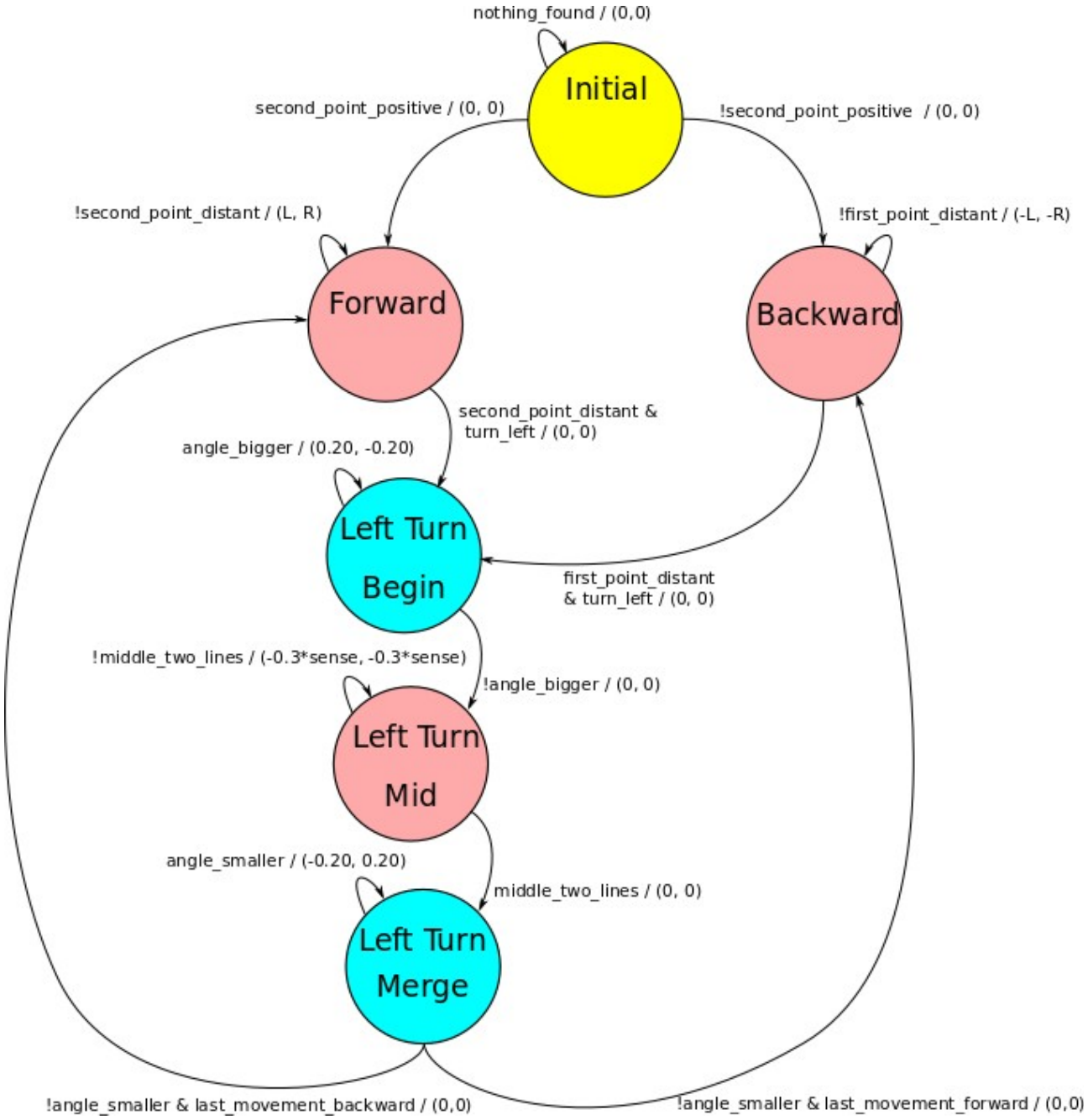
```
Transition StateMachine::impossibleStateRoutine(std::vector<Model> & models)
```

Impossible state logic, returns the Transition containing next state and output

Parameters

models is the vector of models found

Robot State Machine

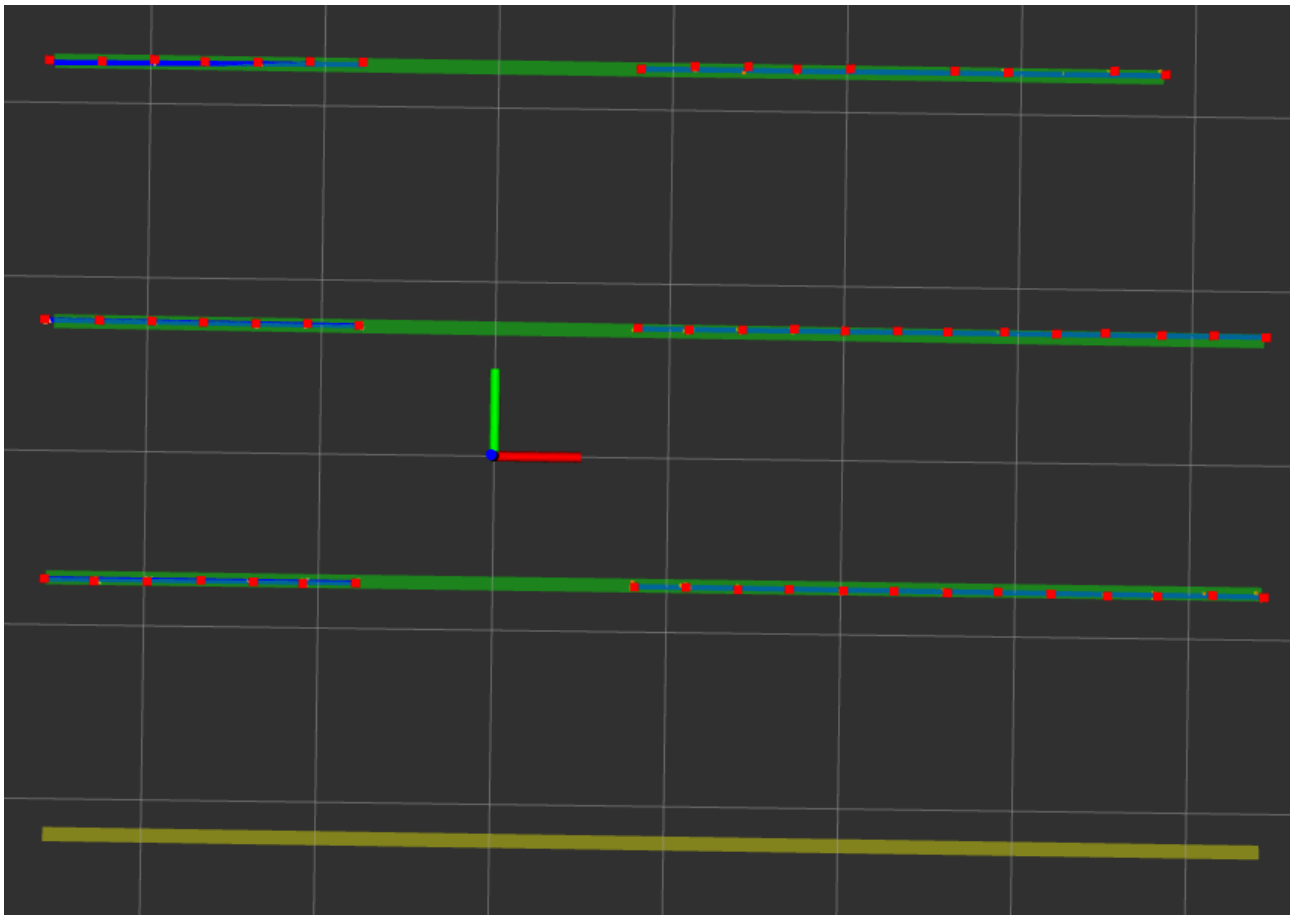


Notation: **States** are written in Bold, *variables* are written in italic

This chart shows how the Finite State Machine (FSM) works in a simplified manner. The first thing that you have to keep in mind, is that the red painted states always have an intermediary state called **Linear Stop** and the blue painted states have **Angular Stop**, to guarantee that the robot will not have overlapping velocities and movement. The *nothing_found* transition for each state was omitted to simplify the drawing, so keep in mind that all the states have a transition that goes to the Initial state if *nothing_found* is triggered. As a final simplification, only the left transition is depicted, but the same logic applies to the right turn. The notation used for this chart is the Mealy state machine notation “condition / output” where output is (Left_wheel_command, Right_wheel_command).

➢ *Background*

This state machine controls the robot's movement, receiving as input the models selected. The algorithm that finds the models and the one that selects and synchronize are out of the scope of this document. The models can be visualized using rviz like shown below. Green models represent models that were found, while yellow ones were calculated using previous information.



➢ *Initial State*

The initial state is the simplest one. Since the robot does not know if it must move forward or backward nor where or how to move, it stays still waiting for lines to be found, thus *nothing_found* is set. If the robot finds one line it decides if it has to go backwards if the last point of the model is negative, *models_found_back* is set and the transition is made to **Backward**, else as a default sense of motion *models_found_front* is set and the transition is made to **Forward**.

➢ *Forward State*

This state uses a fuzzy controller to keep the robot in the middle of the 2 lines, and set *L* as the left wheel command and *R* as the right one, continuing to do so until *last_point_distant* is set. To later use, this state sets *last_movement_forward and sense* to 1. If one of the model's last point is negative and farther than 90 cm the robot got to the end of the line and needs to begin turning. For

the maneuver, it sets *last_point_distant* and transitions to **Left Turn Begin** state, beginning to rotate on its own axis.

➢ *Backward State*

In a similar way, this state uses a fuzzy controller to keep the robot in the middle of the 2 lines until *first_point_distant* is set. To later use, this state sets *last_movement_backward and sense* to -1. If one of the model's first point is positive and farther than 90 cm the robot got to the end of the line and needs to begin turning. It then sets *first_point_distant* and transitions to **Left Turn Begin** state, beginning to rotate on its own axis.

➢ *Left Turn Begin State*

This state makes the robot turn clockwise on its own axis while the angle it makes with the x axis is bigger than -π/4.5 or about -40º. When this limit it achieved, *angle_bigger* is reset to 0 and the transition is made to state **Left Turn Mid**.

➢ *Left Turn Mid State*

The next step is to decide whether to move backwards or forwards. This information comes in the *sense* variable, and thus the robot begins to move in that particular direction, for example let's say it should go forward. Since the robot receives 4 models, it moves in a linear fashion until the second closest left model's intercept and the closest one sum to about 0. This means that both models are equidistant, and the robot is at the middle of the 2 rows. When this happens, *middle_two_lines* is toggled and a transition is made to **Left Turn Merge**.

➢ *Left Turn Merge State*

Finally, the robot starts rotating anti-clockwise on its own axis while the angle it makes with the x axis is smaller than -π/10.0 or about -18. When this limit it achieved, *angle_smaller* is reset to 0 and the transition is made to state **Forward** if the last movement was backward, or **Backward** otherwise.