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 it doesn't find models, this way it knows that it has to change lanes. After that, it will turn anticlockwise until its angle is around 50-60º and then goes forward. When the right model's intercept jumps from a high value to a tiny one, it begins to turn clockwise until its angle is around -30º, 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 without parameters and then to calculate the output and make the state transition, just use the method "makeTransition" passing a std::pair<Model, Model> as the parameter, where the pair.first is the left model, and pair.second is the right model. 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( )
```

Default constructor that assigns default value to Fuzzy controller, current state and where to turn

◆ makeTransition( )

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

Method to calculate the output depending on the state the machine is in, and make a state transition. This 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 pair of found models where the first value is the left model and the second one is the right model

Private Methods

- initialStateRoutine( )

  Transition StateMachine::initialStateRoutine(const std::pair<Model, Model> & models)

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

Parameters

  models is the pair of found models where the first value is the left model and the second one is the right model

- forwardStateRoutine( )

  Transition StateMachine::forwardStateRoutine(const std::pair<Model, Model> & models)

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

Parameters

  models is the pair of found models where the first value is the left model and the second one is the right model

- backwardStateRoutine( )

  Transition StateMachine::backwardStateRoutine(const std::pair<Model, Model> & models)

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

Parameters

  models is the pair of found models where the first value is the left model and the second one is the right model

- linearStopStateRoutine( )

  Transition StateMachine::linearStopStateRoutine(const std::pair<Model, Model> & models)

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

Parameters

  models is the pair of found models where the first value is the left model and the second one is the right model

- angularStopStateRoutine( )

  Transition StateMachine::angularStopStateRoutine(const std::pair<Model, Model> & models)

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

Parameters

models is the pair of found models where the first value is the left model and the second one is the right model

◆ leftTurnBeginStateRoutine( )

Transition StateMachine::leftTurnBeginStateRoutine(const std::pair<Model, Model> & models)

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

Parameters

models is the pair of found models where the first value is the left model and the second one is the right model

◆ leftTurnMidStateRoutine( )

Transition StateMachine::leftTurnMidStateRoutine(const std::pair<Model, Model> & models)

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

Parameters

models is the pair of found models where the first value is the left model and the second one is the right model

◆ leftTurnMergeStateRoutine( )

Transition StateMachine::leftTurnMergeStateRoutine(const std::pair<Model, Model> & models)

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

Parameters

models is the pair of found models where the first value is the left model and the second one is the right model

◆ impossibleStateRoutine( )

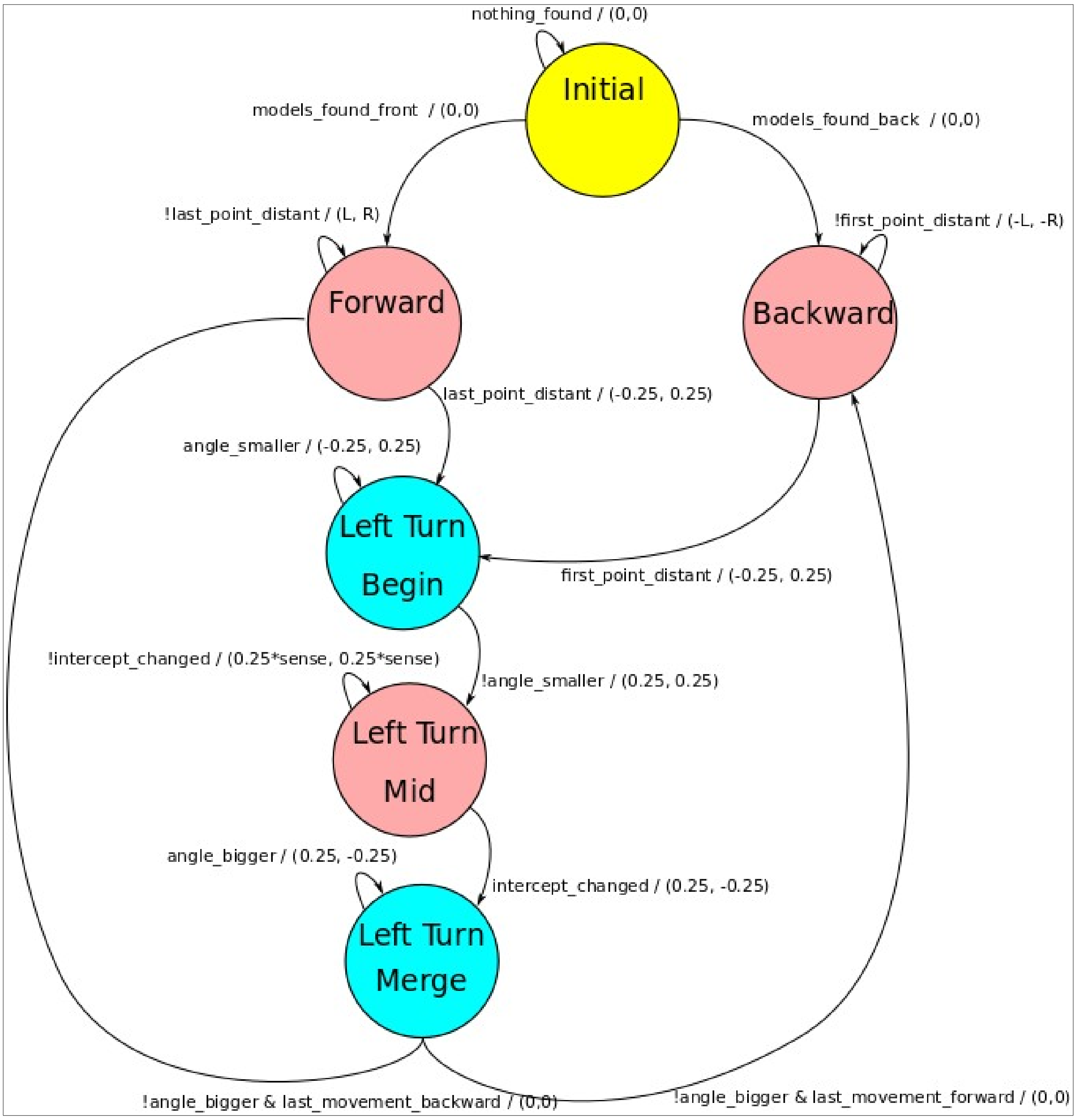
Transition StateMachine::impossibleStateRoutine(const std::pair<Model, Model> & models)

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

Parameters

models is the pair of found models where the first value is the left model and the second one is the right model

Robot State Machine

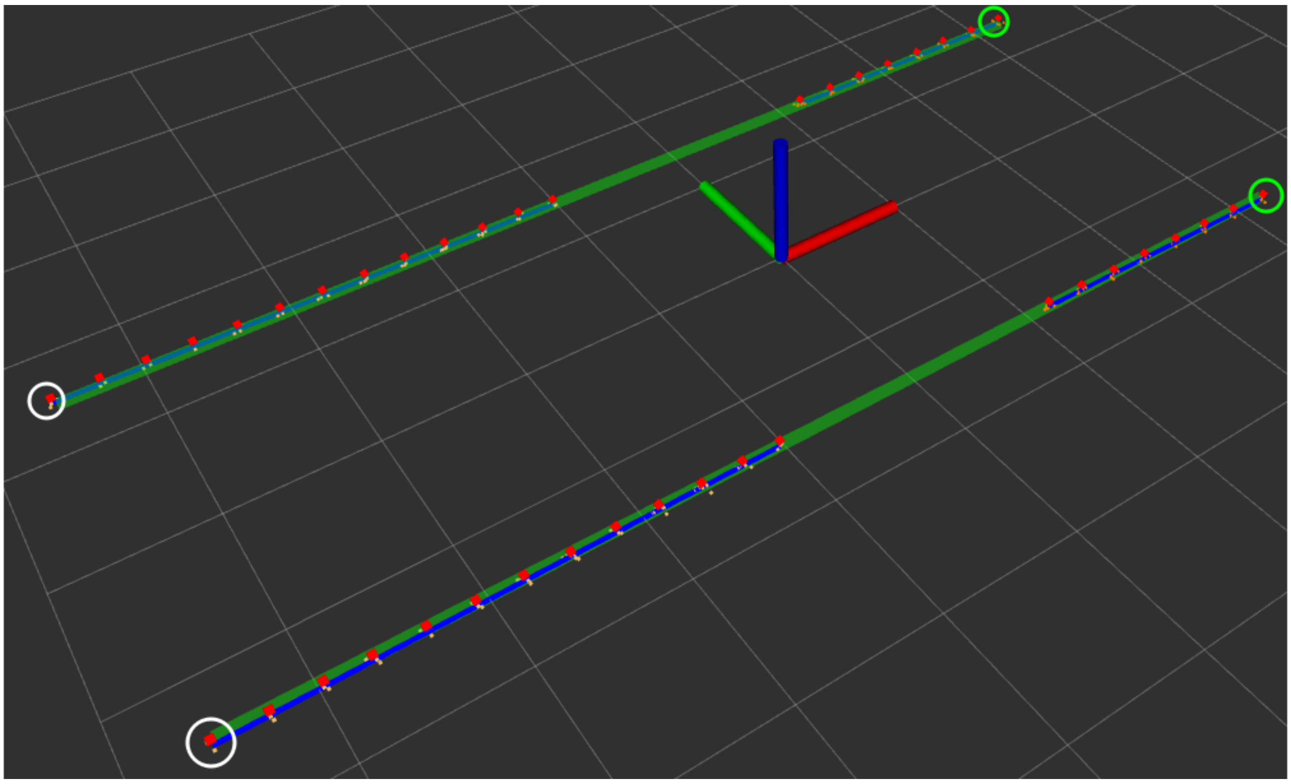

nothing_found / (0,0)
Initial
models_found_front / (0,0)
models_found_back / (0,0)
!last_point_distant / (L, R)
!first_point_distant / (-L, -R)
Forward
Backward
last_point_distant / (-0.25, 0.25)
angle_smaller / (-0.25, 0.25)
Left Turn Begin
first_point_distant / (-0.25, 0.25)
!intercept_changed / (0.25*sense, 0.25*sense)
!angle_smaller / (0.25, 0.25)
Left Turn Mid
angle_bigger / (0.25, -0.25)
intercept_changed / (0.25, -0.25)
Left Turn Merge
!angle_bigger & last_movement_backward / (0,0)
!angle_bigger & last_movement_forward / (0,0)

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. 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. Once the models are found (the green lines in the image below) we have a pair of models where the first one is the Left line that has a positive intercept and the second one is the Right line with negative intercept. Each model contains its first point, represented as the white circle, and the last point, represented as the green circle. Always keep in mind that the robot is always the Cartesian's plane origin, meaning that if it rotates anticlockwise the lines will appear to rotate clockwise.



➢ *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, or 1.5m away if it finds only 1 line 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 50 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 50 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 anticlockwise on its own axis while the angle it makes with the x axis is smaller than π/3.5 or about 50º. When this limit it achieved, *angle_smaller* 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. When moving, the robot stores the intercept of the Right model (the model the robot moves away from) and updates when the intercept gets bigger than 10% of the stored value, doing so to prevent errors in model estimation and tiny variations. When the robot identifies the right intercepts jumps from a big value to around 0.5, it means that the robot is between two rows and needs to merge. Thus, *intercept_changed* is set and the transition goes to **Left Turn Merge**.

➢ *Left Turn Merge State*

Finally, the robot starts rotating clockwise on its own axis while the angle it makes with the x axis is greater than -π/6.0 or about -30. When this limit it achieved, *angle_bigger* is reset to 0 and the transition is made to state **Forward** if the last movement was backward, or **Backward** otherwise.