Bálint Villányi
13484079
balint.villanyi@student.uva.nl



# 1 Manual driving

The manual driving component is the simplest. It is implemented as a composite state with a single internal state. This internal state has an entry action that limits the values of the speed and rotation to their respective maximum values. The internal listen state has four transitions, all back to the same state, each changing either the speed or the rotation according to the requirements. The values are incremented or decremented by a set static value, defined by us and adjustable as needed.

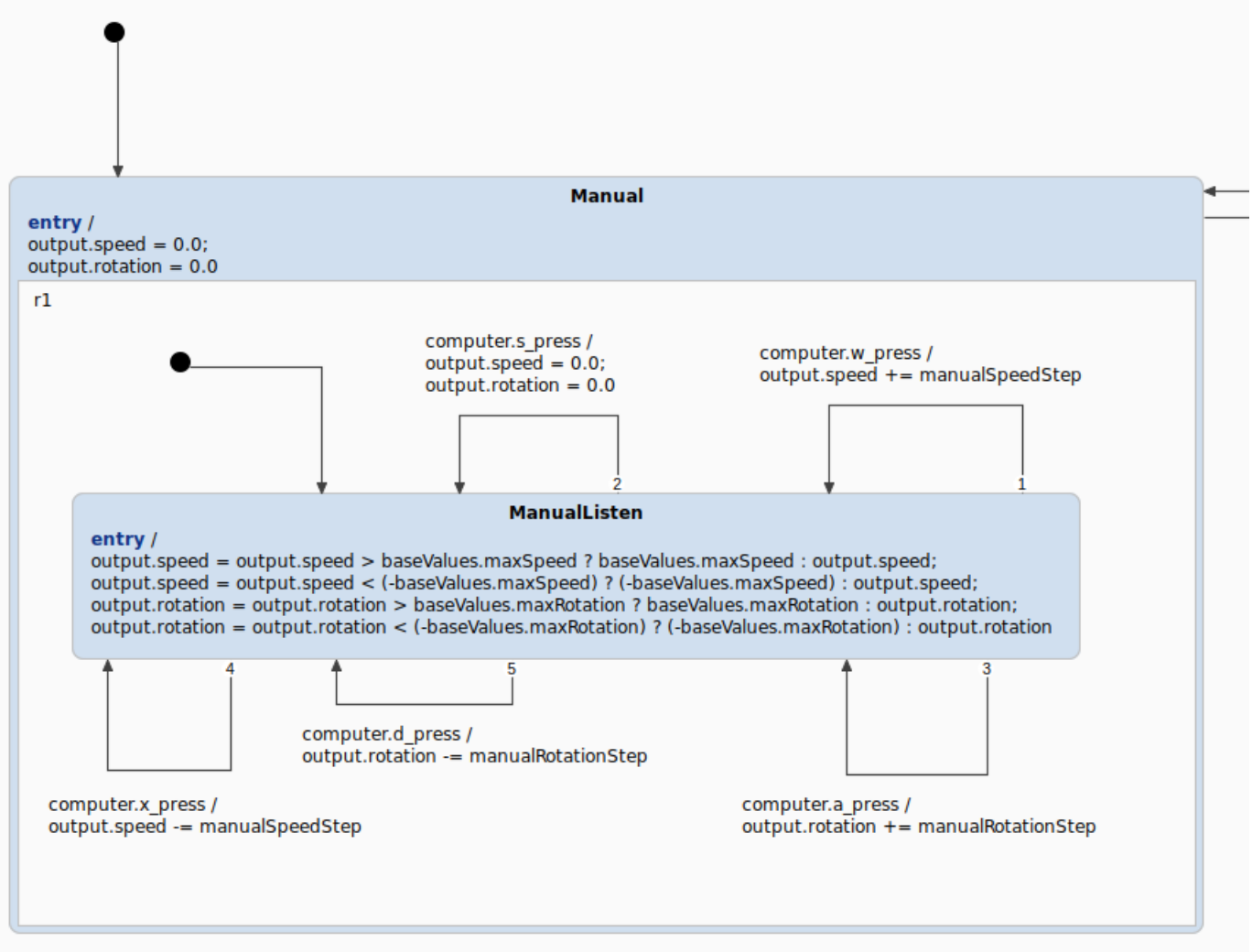


Figure 1: Manual driving component

The composite state has a single transition exiting it, a reaction to the key *m*, which transitions into the autonomous driving mode.

# 2 Autonomous driving

The autonomous driving component is based on the simple left-hand logic. At every cell the bot decides the direction it will continue in, based on its current orientation and the existence — or non existence — of walls around it. In case there is no wall to the left of the bot, then it will turn left and go forward one cell-length. The bot follows the same logic with the wall in front, to the right, or to its back, in this order of priority. This

logic ensures that all cells of the maze are visited, if we assume that the maze is a *simply connected* maze. If the maze is not simply connected, then the wall follower technique can fail.

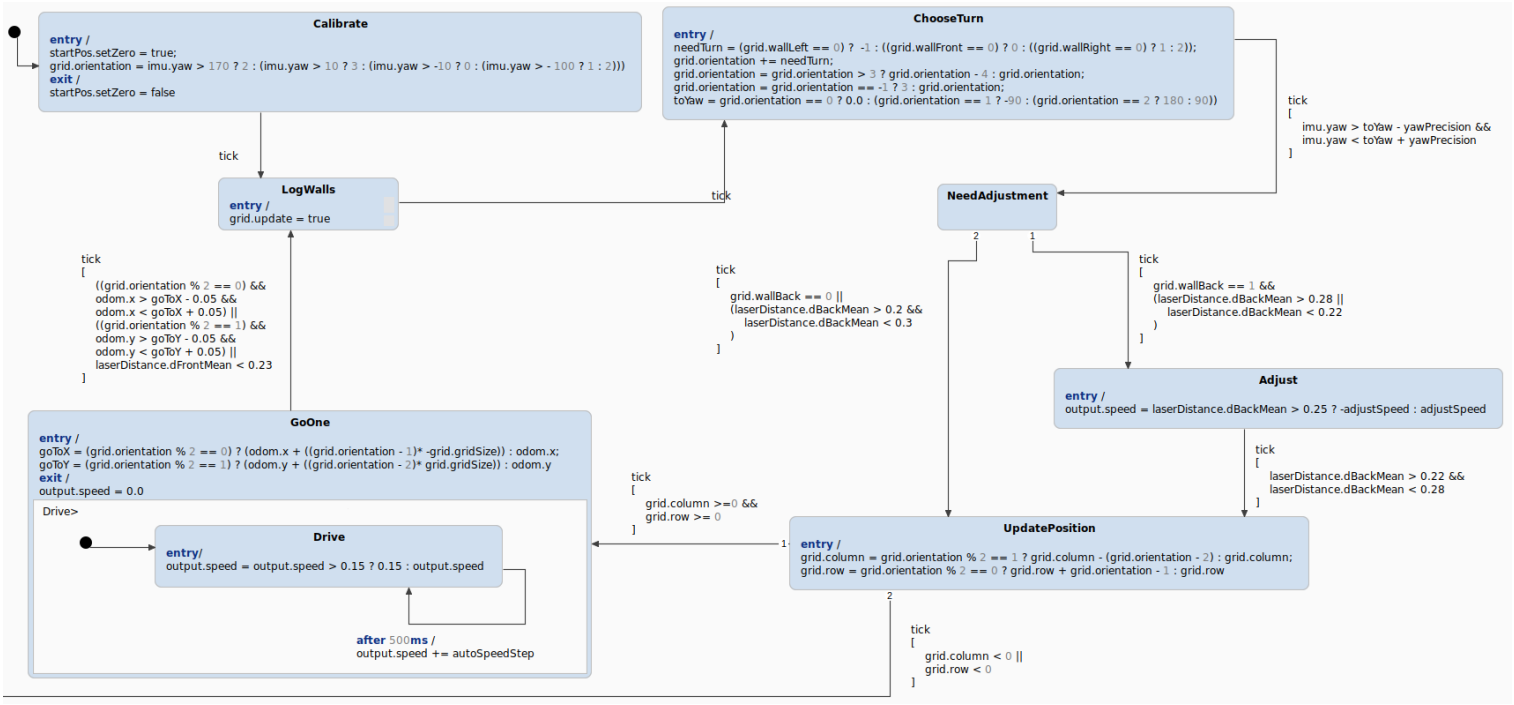


Figure 2: Autonomous driving component, logic

The driving component is made up of two larger parts, one governing the logic and forward movement, and another responsible only for the rotation of the bot. These two parts are connected in an orthogonal state. Figure 2 shows the logic part and Figure 3 shows the rotation part

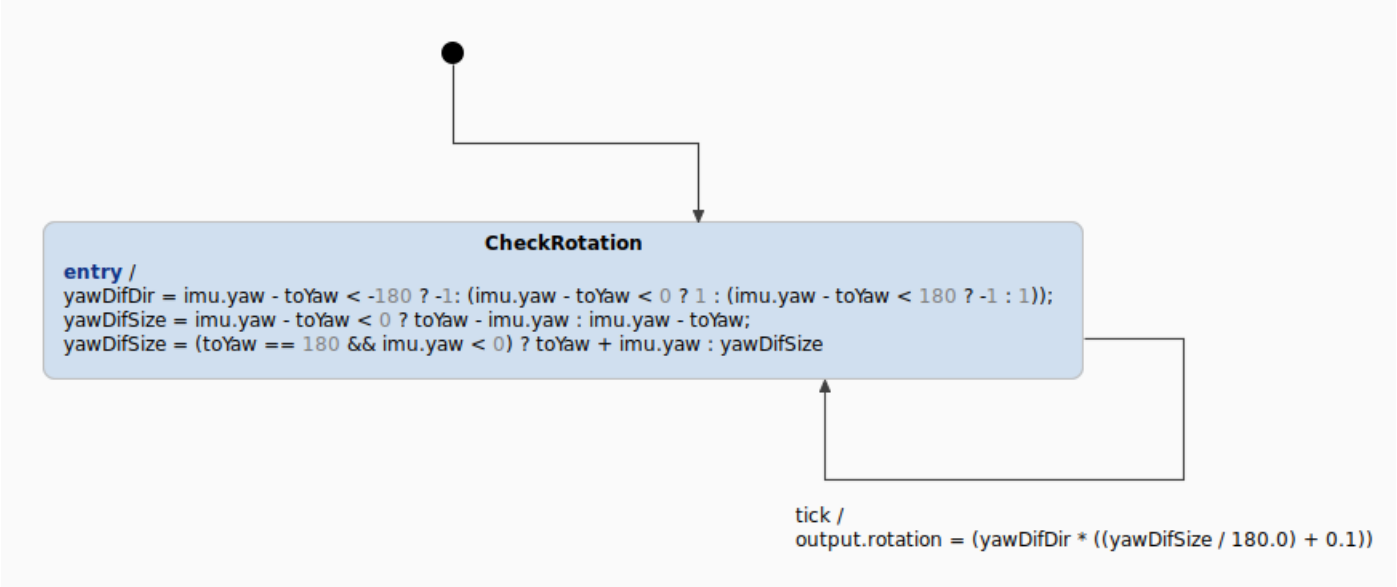


Figure 3: Autonomous driving component, rotation

The logic starts by calibrating the starting position upon entering into the autonomous driving component. A loop starts, that is only exited when the maze traversal is complete.

1. The walls of the current cell are logged
2. The next direction and the rotation necessary to achieve this direction is determined
3. Wait until the rotation is accurate enough
4. If an adjustment is needed then carry out this adjustment
5. Determine the row and column position we are travelling to
6. If we are travelling outside the maze, exit the loop
7. Travel forward one cell length, incrementing the speed in steps
8. Wait until desired spacial location is reached with enough accuracy
9. Back to step 1

The rotation part is continuously looking at the desired yaw value, held by the *toYaw* internal variable. The larger the difference to the current yaw value of the IMU, the faster the rotation speed. The rotation is always adjusted, even while moving, ensuring that the bot never really drifts more than a few degrees.

The only way to exit the loop is by completing the traversal by arriving at the exit again in such a way, that we would exit the maze in the next movement phase of the loop. The visual representation is drawn at this point and the program terminates.

# 3 Adjustments

The turning and the forward movement are based on the IMU and odometry data only. Testing showed that these are accurate enough by themselves to traverse a maze at least once. Still, there is a small adjustment algorithm built in as well.

Every time when the bot starts with a wall to its back, if the wall is too far or too close, the distance to the wall is adjusted until it is below an accuracy margin of 0.03m. There are some turns where there is no wall to the bot's back, there is no adjustment in these turns. There is also a safety feature of stopping if a wall to the front comes too close, which also acts as an adjustment. Testing showed that this amount of adjustment is enough to keep the bot centered in the cells.

As already mentioned the rotation is always adjusted towards the desired yaw value.

# 4 Additional components

Besides the states governing the movement of the bot, we have two states recording and changing helper information. Both of these are implemented as a large orthogonal state, orthogonal both with each other and with the composite state of the driving component.

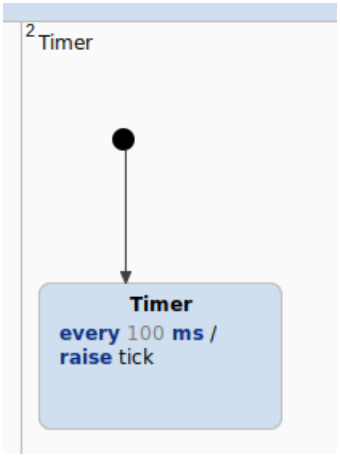


Figure 4: Timer

The system works in a synchronous manner, there is a *tick* event fired every 100 ms. Almost all transitions react to this event annotated by some guard condition, except for the transitions reacting to key presses.

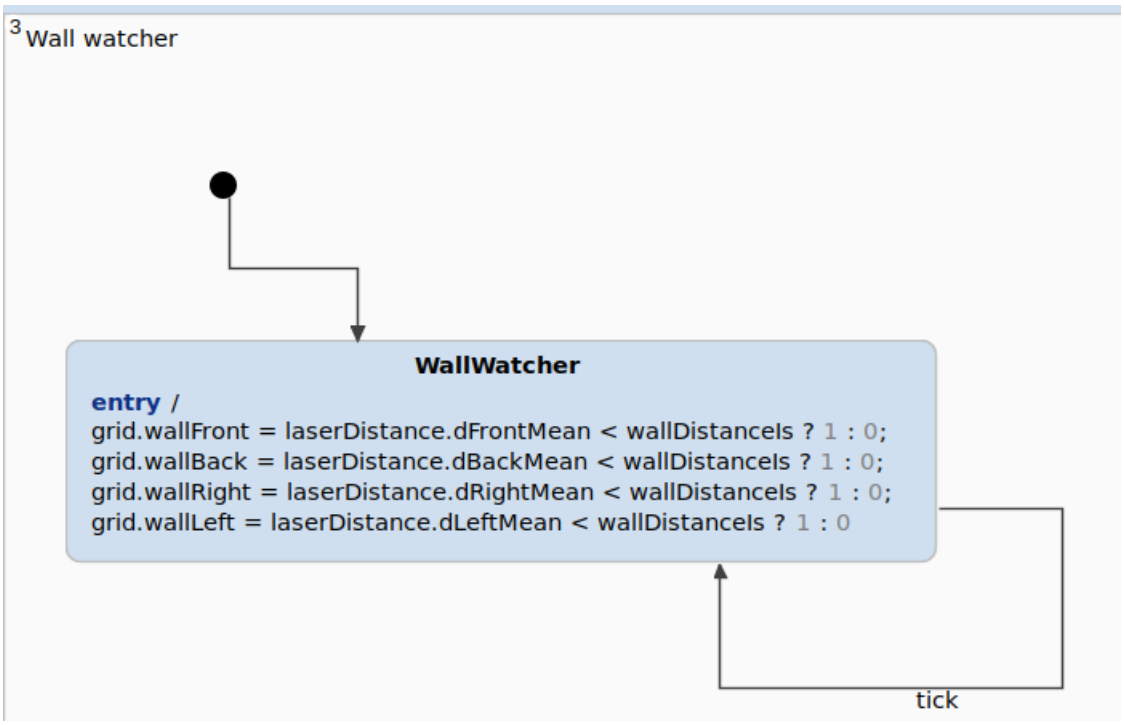


Figure 5: Wall watcher

Another orthogonal region detects the state of the walls around the bot. The *WallWatcher* state has a single transition, to itself, reacting to every tick. Its entry action then sets whether we are in close proximity to a wall through the *grid* interface. All four walls are set at the same time. This information is continuously available to all other components. Wall information is never received in my solution, so there was no need to suspend this watcher component. The wall distance limit was determined by trial and error and has the value of 0.45m.