

TDDE05: Lab 1: Control and State Machines

Cyrille Berger

January 26, 2021

The goal of this lab is to implement the position control subsystem for your robot. The robot can have the following control mode:

- Idle: the robot should keep a null velocity
- Velocity: the robot should keep a specific velocity (linear and angular)
- Position: the robot should reach and stop at a specific location
- Waypoints: the robot is given a set of waypoints (a path) to follow
- Positions queue: the robot keep a queue of positions and when it reaches a position it starts moving toward the next position

The simulated robot takes as input velocity commands, on the topic `/husky0/velocity_cmd`. This should be the output of your control subsystem.

1 Get the code for lab1

Get the skeleton code for lab1 from gitlab:

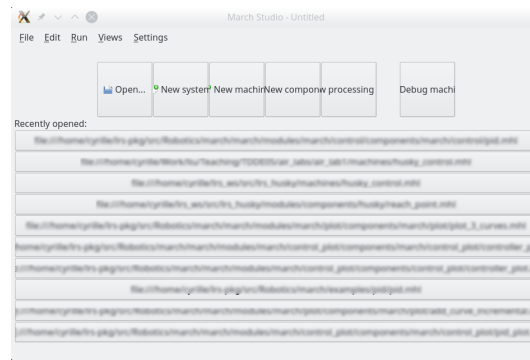
```
1 cd ~/TDDE05/catkin_ws/src/air_labs/  
2 git pull --allow-unrelated-histories \  
3 https://gitlab.ida.liu.se/tdde05/air_labs.git master
```

WARNING: If you copy paste the `git pull` command and it paste on a single line, you need to remove the `\` from the line.

WARNING: if you try to build at this point, you are very likely to get an error, you need to implement the PID controller before you can compile.

You also won't be able to build the second lab, run this to only build lab1:

```
1 cd ~/TDDE05/catkin_ws/  
2 catkin build air_lab1
```



2 March

March is the *Modeled Architecture*. It is a state chart framework which allow to define state machines that integrate with ROS. State machine are designed using *March Studio*, which can be launched with the following command:

```
1 march_studio
```

It should show a window like in figure 1. This window contains buttons for the different type of project supported by *March*. You are mostly concerned with the machine, component and processing component.

To get the ROS modules, you need to add the search path to the studio, go in settings, in paths tab, add:

- /courses/TDDE05/software/catkin_ws/src/lrs_march/march
- /courses/TDDE05/software/catkin_ws/build/lrs_march/march

In this lab, however, you are already provided a base state machine: `air_labs/air_lab1/machines/husky_control.mhl`. If you open that file you will see a window similar to figure 2. On the left of the window, you will see the toolbox used to create connection, states, nodes, connectors... On the right the actual machine.

On figure 3, you can see an overview of the state machine you were given. In the top left corner is a group that decompose a `nav_msgs/Odometry` messages so that we can use its components in the state machine. In the bottom left corner is the group that control which control mode is currently used (idle, velocity, waypoints...). On the right is the velocity control group.

3 Left/Right wheel velocity PID

3.1 PID

The first step of the lab is to add a PID controller to control the velocity of your robot. To simulate the effect of the robot sliding on the ground, the simulator add noise to the velocity of the robot, the simple controller of the first lab is not good enough. The gold

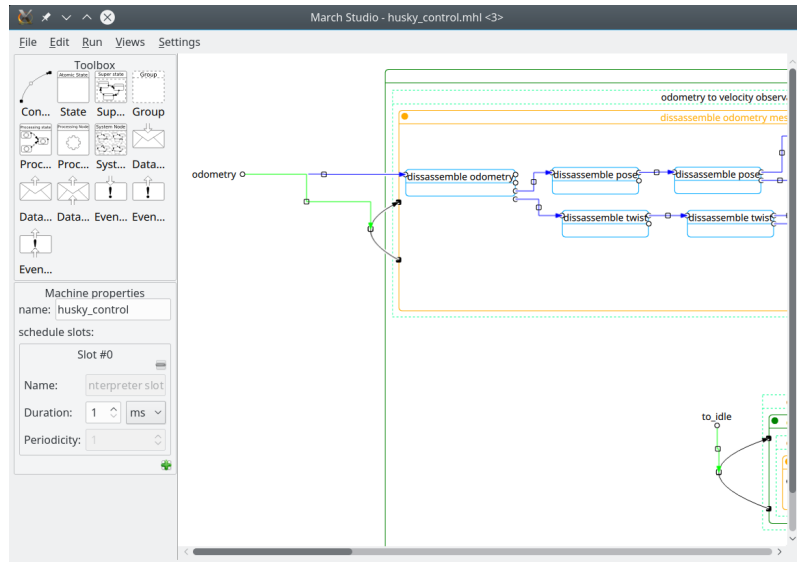


Figure 2: Husky control machine opened in *March Studio*

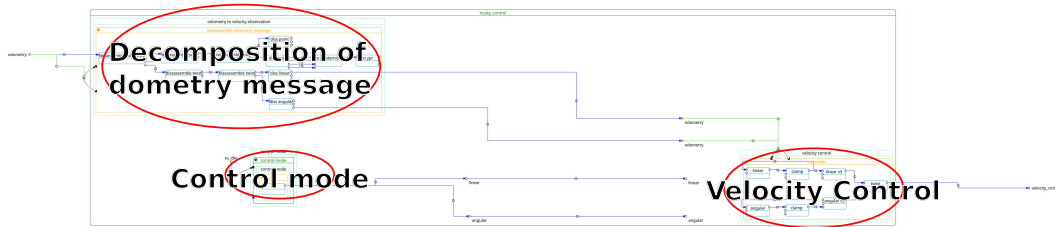


Figure 3: Overview of the control machine

standard in control to solve this problem is the PID (proportional–integral–derivative) controller, see figure 4.

$$u(t) = u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \quad (1)$$

where K_p , K_i and K_d are non negative constant corresponding to the proportional, integral and derivative components of the controller.

- the *proportional* term correspond to the *current* error
- the *integral* term takes into account *past* errors
- the *derivative* term correspond to the prediction of *future* errors

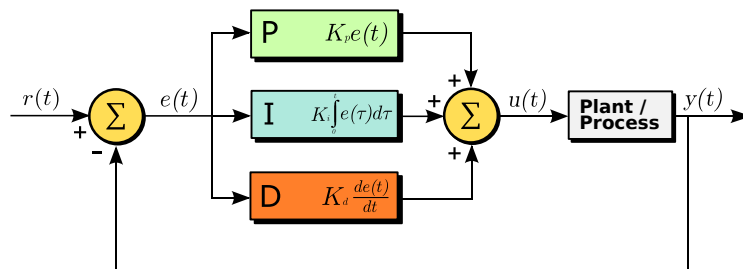


Figure 4: PID controller (source: wikipedia)

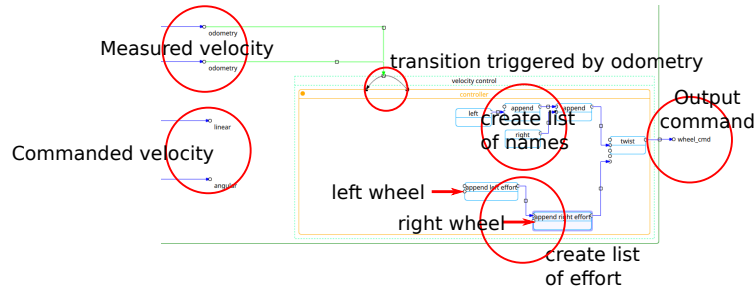


Figure 5: Velocity control

In a discrete time system:

$$e(t) = e_t \int_0^t e(\tau) d\tau = \sum_{\tau=0}^t e_t \cdot \delta_\tau \frac{de(t)}{dt} = \frac{e_t - e_{t-1}}{\delta_t} \quad (2)$$

Where δ_t is the time interval between t and $t - 1$.

3.2 PID for Husky velocity

Commonly a velocity controller takes a velocity as input and output an effort or power level. In case of the Husky, the input correspond to the velocity that we want the robot to have while the output correspond to the energy level applied to the left and right wheels. $e(t)$ therefore correspond to the error between the target wheel velocity and the odometer wheel velocity. While $u(t)$ is the update to the energy that is then given to the robot, so that:

In the state machine that you were provided with, you will now need to add the PID controller to the velocity control, see figure 5. You are provided with the input to the PID controller (the odometry and the desired velocity), you can see the connector on the left in figure 5.

The velocity control is a group with a single state, that state should transition to itself every time a new odometry message is received, as it is shown through the connection of the odometry to the transition. The state, named *controller* is actually implemented as a *processing state*, which means it contains a flow chart that is executed every time the state is activated.

3.3 In practice

For this part of the lab you will need to use two PID loops, one for controlling the power given to the left wheel and the other one for the right wheel. The PID controller is already implemented as part of the standard distribution of *March*, it can be found in the *march_control* library. It has a single output, which is the update $u(t)$ in figure 4. It takes four inputs:

- `control` (`std_msgs/float64`) a number which correspond to the target velocity ($r(t)$ in figure 4)
- `observation` (`std_msgs/float64`) a number which correspond to the odometry value ($y(t)$ in figure 4)

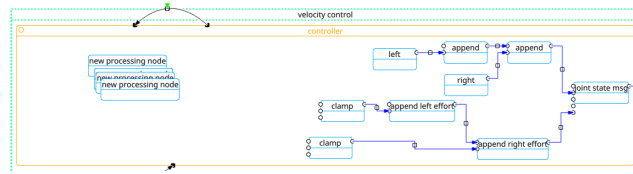


Figure 6: Velocity control with three new nodes.

- `current_time` (`std_msgs/float64`) expressed in seconds
- `initial_previous_time` (`std_msgs/float64`) which correspond to the time when starting to run your state machine

To add a PID node to your controller, you need to create a new *processing node*, using the *processing node creation tool* from the toolbox:



- Click on icon `Proc...` to activate the *processing node creation tool*.
- Then click within the *controller* state to create a new node. You will need four of them, so click four times.
- Press *Escape* to deselect the tool

If you created too many nodes, you can select them and press *delete* or in the menu *Edit, Delete* to remove the superfluous nodes. It should look like on figure 6.

Then you need to assign the type for each of those new nodes, two of them should be `march_control/pid`, one `march_ros/current_time` and one `air_lab1_march/linear_angular_to`. To assign the type of a node, you should click on the node and then in the property panel on the left, you can choose the library and the type for each node. Once you have selected the type, you will also be able to set the parameters for each node, for a `pid` you can choose K_p , K_i and K_d . By default, they are set to 0.0, if you set $K_p = 1.0$, the PID controller will behave the same way as the lab0 controller. You need to set i_{min} to -1.0 and i_{max} to 1.0 , those are the values use to bound the integration term, as it can get saturated, in particular, if commanding a too high velocity.

You will need to connect the control and observation to the desired velocity and the



odometry, using the connection tool `Con...`. You will then need to connect the output of the `march_ros/current_time` node to the `current_time` connector of the two PID nodes. And finally connect the output of *initial time* to the `initial_previous_time` connector of the two PID nodes.

It should then look like on figure 7.

3.4 Velocity mode

At this point, you now have a velocity controller, but the only input it ever get is a velocity of 0. For handling the mode, we will use multiple states within a super state. Each state correspond to one of our control mode (idle, velocity...). The super state has already

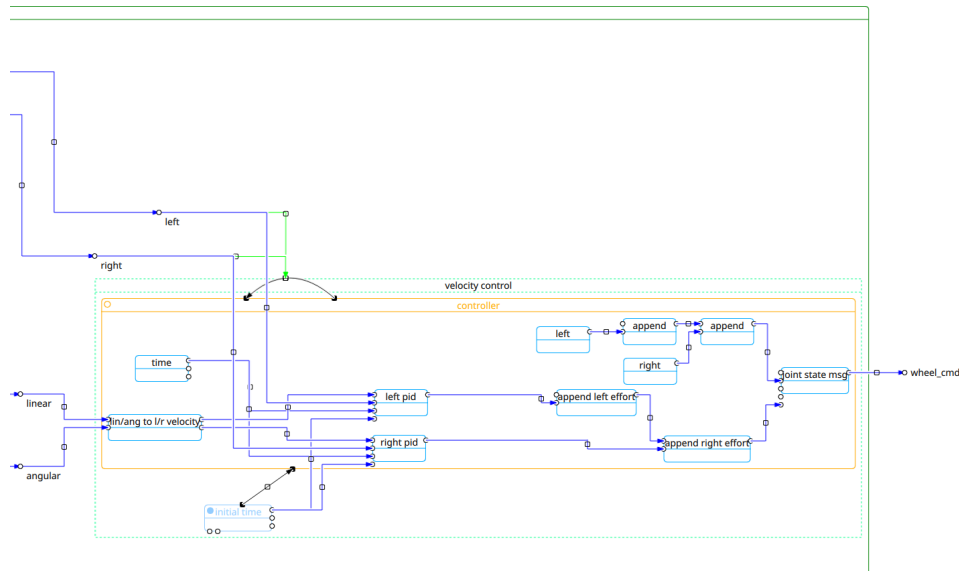




Figure 7: Velocity control with PID nodes.

been created for you and can be seen of figure 8, it contains a idle mode and a super state for position control modes, that you can ignore for now.

Before creating a state, we should think about what will be done. In this case, we will set the velocity of our controller using the `geometry_msgs/Twist`, which we will need to decompose into linear velocity along x and angular velocity along z. This is several operations that needs to be executed once, it is therefore required to use a *processing state* rather than a simple *state*.

You can create a *processing state* using the *processing state creation tool* . Once you have done, that you can use the *connection creation tool* to create a transition from *idle* state to *velocity control* state. This transition should be triggered when an external event `to_vel_control` is triggered, we therefore need to create an external event

connector with the *external input event creation tool* . Rename the newly created event by clicking on *untitled* and write `to_vel_control` instead.

Then you need to connect this newly created event so that when it is triggered, it first reset the super state and then switch from *idle* state to *vel_control* state. Using the connection tool, you can click on the event connector to create a new connection and then click on the middle connector of a transition connection.

You should now have something that looks like on figure 9.

There is a potential problem, events and transitions are executed in creation order, so if you connected `to_vel_control` to the transition from *idle* to *vel_control* state, then this will be considered before the reset of the super state, which means your system will end up in the *idle* state. To solve that problem, you can set the priority of the event, by clicking on the middle of an event connection, a higher priority means that the event is executed first, so set the priority to the event resetting the super state to a higher value than the one connecting *idle* to *vel_control*.

The next step is to set the commanded velocity. You will need to create a *data*

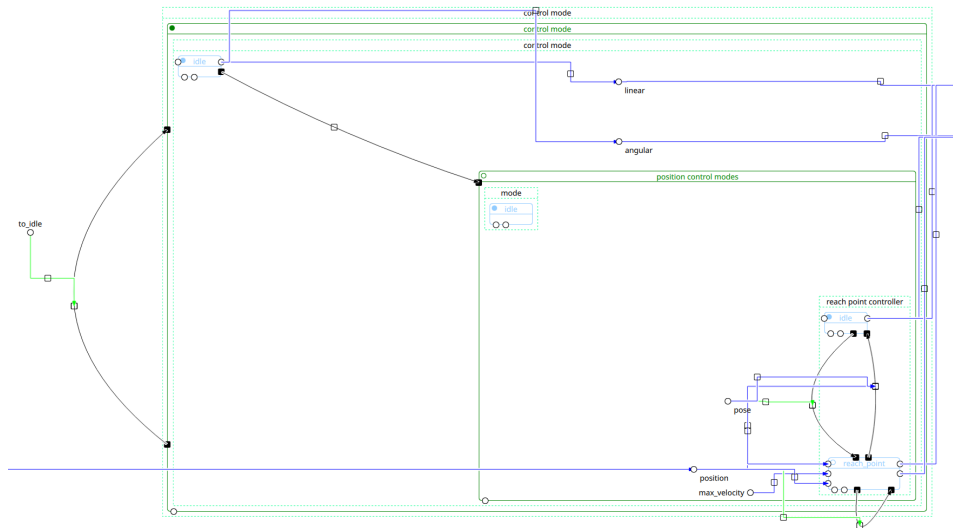


Figure 8: Control modes super state.

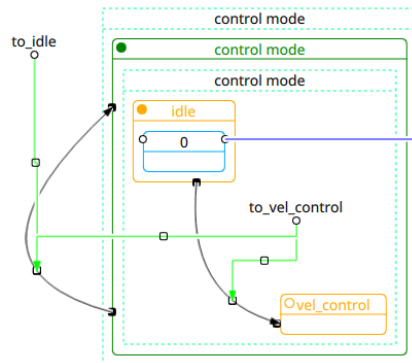


Figure 9: Control modes with *idle* mode and *vel_control* mode.



input connector using the *external input data creation tool* `Data...`. You can set the name to `cmd_vel`. You will need to set the type to `ros_geometry_msgs/twist` (this will be automatically mapped to a `geometry_msgs/Twist` message in ROS).

Then in your *processing state* you should create processing nodes and use `ros_geometry_msgs_tools` library to disassemble the message to be able to extract the *linear* and *angular* components. Those components can then be connected to the PID controllers. Also, you will need to update the velocity every time a new message is received on the `cmd_vel` topic, therefore you should create a transition on the *vel_control* state that reset it, every time a message is received.

3.5 Run the controller

First you will need to build it:

```
1 start-tdde05
2 cd ~/TDDE05/catkin_ws
```

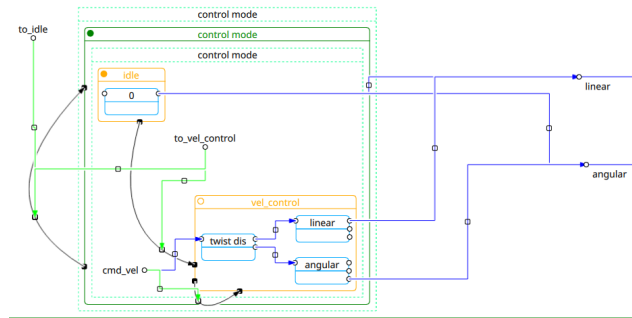


Figure 10: The *vel_control* state is finished and ready to run.

3 catkin build

Then in `packages.xml` add:

```
1 <run_depend>nodelet</run_depend>
```

Then this will actually build the controller as a ROS nodelet, to launch it you can use the following command:

```
1 rosrunc nodelet nodelet standalone air_lab1/husky_control_node \
2   __ns:=/husky0
```

In case of error, try to run `start-tdde05` in the terminal.

When the state machine is converted into a ROS node, it will create topic for events and data automatically. To switch between different control modes, you can send a message on the corresponding topic, for instance, to switch to idle:

```
1 rostopic pub /husky0/to_idle std_msgs/Empty {}
```

And then to switch to *velocity control*:

```
1 rostopic pub /husky0/to_vel_control std_msgs/Empty {}
```

3.6 Integration with rqt

You should add `/husky0/to_idle` and `/husky0/to_vel_control` in your `rqt` interface.

In `rqt`, you can add a robot steering panel, like shown on Figure 11. To do that in the Plugins menu, Robot Tools select Robot Steering. In the Robot Steering panel, set the topic to `/husky0/cmd_vel`. You should now be able to control your robot.

3.7 Troubleshooting

If the robot does not move:

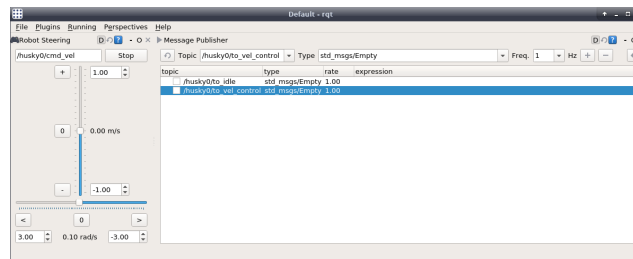


Figure 11: rqt with a *Message Publisher* and *Robot Steering* panel.

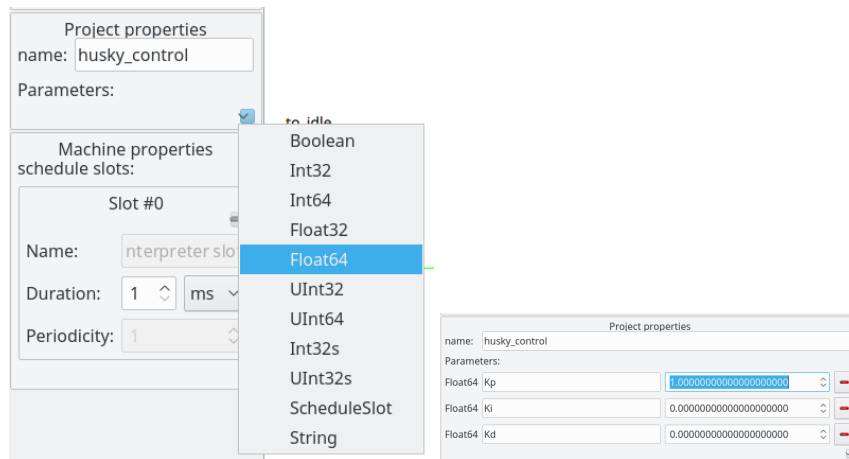


Figure 12: On the left, parameters panel and on the right parameters panel with all parameters

- Check your transitions
- Check the parameters of PID
- To help you debug, use the `std_msgs/print` operation to print values on the standard console

3.8 Calibrate the PID loops

Then you will need to calibrate the PID loops. We will use the `evaluate_controller.py` and `autotune.py` scripts provided in `air_lab1/src`.

To help with the process, you can defined parameters that are read from the command line. To create a parameter, make sure nothing is selected (press ESC). You should then see the parameters panel like on figure 12, using the plus icon you can create three parameters Kp, Kd and Ki (create them as float64).

Then click on the `left pid` node and you should see a panel similar to figure 13, you can assign the global parameters you have created. Do the same for the right pid as well. There is no reason to use different parameters for left and right.

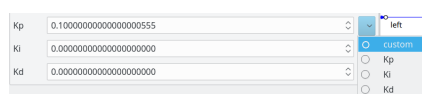


Figure 13: Parameters set on the activity

```

1 roslaunch nodelet standalone air_lab1/husky_control_node \
2   __ns:=/husky0 _Kp:=1.0 _Ki:=0.1 _Kd:=0.01

```

3.8.1 Plotting the error

We can plot the error using `evaluate_controller.py` and `rqt`. You should inspect the source code of `evaluate_controller.py`, if you try to run it, it will show an error:

```

1 roslaunch air_lab1 evaluate_controller.py __ns:=/husky0

```

It is missing the message `air_lab1/ControllerEvaluationStat`. We will need to create it:

```

1 cd ~/TDDE05/catkin_ws/src/air_labs/air_lab1/
2 mkdir msg
3 cd msg
4 touch ControllerEvaluationStat.msg

```

Then in your favorite editor, in `ControllerEvaluationStat.msg`:

```

1 int32 samples          # Number of samples used for computing
2                        # the error
3 float64 last_error     # Last error
4 float64 average_error  # Average error

```

Then in `CMakeLists.txt`, find the section with `add_message_files` and add:

```

1 add_message_files(
2   FILES
3   ControllerEvaluationStat.msg
4 )
5 generate_messages(DEPENDENCIES std_msgs )

```

Look for the line with `find_package(catkin` and after `COMPONENTS` add `message_generation` and `std_msgs`, which are two packages needed for compiling the message.

This should show an empty application with a menu:

Then to generate the code for the message, you need to build the module:

```

1 catkin build air_lab1
2 start-tdde05

```

The command `start-tdde05` ensure that the definition of your new messages is properly loaded in the environment.

Now you should be able to run the controller evaluator:

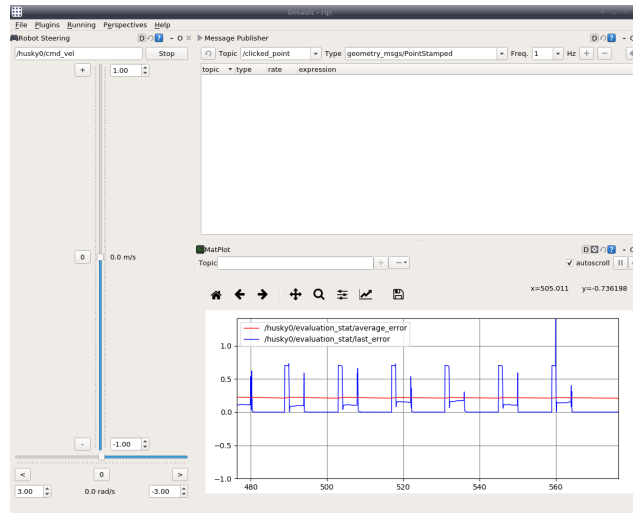


Figure 14: *rqt* plotting the error.

```
1 rosrun air_lab1 evaluate_controller.py __ns:=/husky0
```

3.8.2 Plotting the error

In *rqt*, add a Plot panel (in *ROS noetic*, the Plot panel is a bit buggy, and moving it around may crashes *rqt*). In the Plot panel, in Topic write `/husky0/evaluation_stat/last_error` and click +, this will subscribe to `/husky0/evaluation_stat` and plot the `last_error` field. You should also plot the `average_error` field. If you click on the second to last icon (some kind of plot figure with an arrow), you can set the value for the x axis to *left* = 0 and *right* = 100 and for the y axis to *bottom* = -1.0 and *top* = 1.0. This should show the last 100s of error. The resulting *rqt* window is shown on Figure 14.

Try to see the effect of different values of `_Kp`, `_Ki` and `_Kd`. You can alternate between a velocity of 0 and 0.5 in *rqt*.

3.8.3 Autotuning

You can use the `autotune.py` to tune the PID parameters of your controller:

```
1 rosrun air_lab1 autotune.py __ns:=/husky0
```

Make sure that no other instance of your state machine is running and that `evaluate_controller.py` is running.

`autotune.py` use a brute force approach to callibration, it tries different values and check which values have the lowest error on a curve segment followed by controlling a null velocity.

Do note that the results given by `autotune.py` are unstable, and the values you get back from it can vary from run to run (you might even get null values for K_i and K_d). An interesting project can be to test better method for autotuning.

```

Terminal - screen
File Edit View Terminal Tabs Help
WARNING: Failed to set real time priority: Operation not permitted
Send linear velocity
Send null linear velocity
Finish up
Error for [0.000000000000002, 0.000000000000000, 0.000000000000000] was 21.9983665059
Start testing: [0.000000000000002, 0.000000000000000, 0.000000000000000]
type is air_lab1/husky_control_node
WARNING: Failed to set real time priority: Operation not permitted
Send linear velocity
Send null linear velocity
Finish up
Error for [0.000000000000002, 0.000000000000000, 0.000000000000000] was 21.4999107692
New lowest error!
Start testing: [0.000000000000002, 0.000000000000000, 0.000000000000004]
type is air_lab1/husky_control_node
WARNING: Failed to set real time priority: Operation not permitted
Send linear velocity
Send null linear velocity
Finish up
Error for [0.000000000000002, 0.000000000000000, 0.000000000000000] was 23.073054692
Best values are [Kp, Ki, Kd] = [0.000000000000002, 0.000000000000000, 0.000000000000000] with error 21.499
9107692
cyb@290:hw.5.2:~$
00 autotune

```

Figure 15: *autotune* output.

3.9 Launch files

Since typing all those command lines all the time is not so convenient, we can use launch files to define how to launch our ros program.

```

1 cd ~/TDDE05/catkin_ws/src/air_labs/air_lab0/
2 mkdir launch
3 cd launch
4 touch start_everything.launch

```

Then in your favorite text editor:

```

1 <launch>
2   <!-- Launch a node of type nodelet from the package
3       air_lab0 give it the name "HuskyControlNodelet" -->
4   <node pkg="nodelet" type="nodelet" name="HuskyControlNodelet"
5       args="standalone air_lab1/husky_control_node" >
6       <param name="Kp" value="1.0" />
7   </node>
8 </launch>

```

This tell ros that we want to launch a node of type `nodelet` (type) from the package `nodelet` (pkg) with arguments `standalone air_lab1/husky_control_node`. And we can also the parameters, we have shown an example for `Kp`, you should add `Ki` and `Kd`.

Then we can run it using `roslaunch` *roslaunch*:

```

1 roslaunch air_lab0 start_everything.launch __ns:=/husky0

```

It is possible to define the namespace in the launch file, but this would bind the launch file to use with *husky0* for sake of generality, it is better to pass the namespace on the command line.

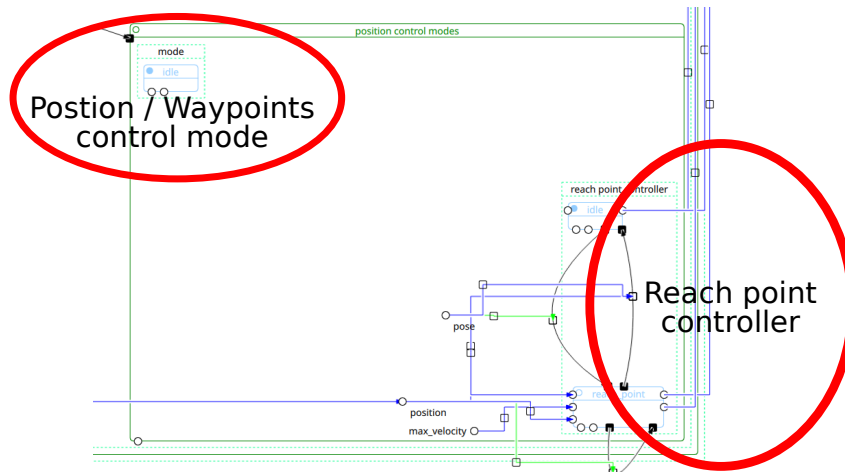


Figure 16: Position controllers

4 Position controllers

In the remainder of the lab, we are going to implement two control modes related to moving the robot toward a position. From the book *Introduction to Autonomous Mobile Robots*, you should read section 3.6, we call it *reach point* for this lab.

Since the *reach point* is going to be used by the two different modes, and it should only be active when in those control modes, then we will have a super state in the control mode with two parallel states, one for selecting the control mode and one for running the *reach point* controller.

The *reach point* controller contains two states, one *idle* that output a null velocity and the *reach point* state that will adjust the velocity of the robot until it reach the destination. The *reach point* controller takes the following inputs:

- *pose* which is the destination point (which you will connect to your control modes)
- *position* which is the current position of the robot (the position comes from the odometry message)
- *max_velocity* which is the maximum velocity the robot should drive to reach its destination (this velocity is set on a topic)


The *reach point* controller has three parameters that you need to set:

- k_ρ (Krho) which determine how fast the robot should move toward the goal
- k_α (Kalpha) which determine how much the robot should aim toward the goal
- k_β (Kbeta) which determine how much the robot should face the target orientation

$$k_\rho > 0; k_\beta < 0; k_\alpha - k_\rho > 0 \quad (3)$$

5 Position control

For this mode, you will just need to represent it with a regular state:

- create a data input connector named `cmd_position` of type `ros_geometry_msgs/pose_stamped`
- create a state  set it to `march_std_lib/pass_through` (create the state in the *mode* group of *position control modes*)
- create an event input connector named `to_position_control`
- You need to connect that event to:
 - the reset transition of *control mode*
 - the transition from *idle* to *position control modes*
 - the transition from *idle* to your new state

The output of `march_std_lib/pass_through` will be used as input to the *reach point* controller's *pose*.

5.1 Rviz

You can use rviz to easily select the destination point, launch rviz:

```
1 rviz
```

In the panels menu, make sure `tool properties` is checked. In the `tool properties` panel, set `2D nav goal` to `/husky0/cmd_position`.

5.2 Troubleshooting

- Check your transtions
- Check the parameters of reach point
- Check that you are setting a maximum velocity

6 Waypoints control

Next step is to create a *trajectory control mode*:

- create a data input connector named `waypoints` of type `ros_nav_msgs/Path`
- create an event input connector named `to_waypoints`

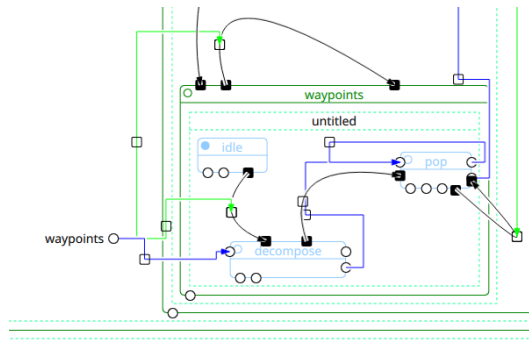


Figure 17: Suggestion of waypoint control mode

In the *position control modes* group, you will need to create a new super state in which you will receive the waypoints from the connector and dequeue a waypoint everytime the previous point has been reached. You can use an event from the position controller to detect when the current waypoint has been reached (for instance when its idle mode is reached). You can use `march_std_lib/list_pop` to remove the first element from a list and output it. `march_std_lib/list_pop` input a list, and output the first element and that list without the first element. A suggestion of how it could look is shown on figure 17.

To test your mode, you can use the following command:

```
1 rostopic pub /husky0/waypoints nav_msgs/Path
2   "{ poses: [ { pose: { position: { x: 0, y: -2 } } },
3     { pose: { position: { x: 0, y: 2 } } } ] }"
```

7 Launch file

You should create a launch file for your robot, in which you start the state machine. You will add other nodes to that launch file in the next labs.

7.1 Demonstration

- Show your launch file
- Show in `rviz` how your robot respond to the different command mode: `velocity control`, `position control`, `waypoint control` and `idle`.