TDDE05: Lab 1: Control and State Machines

Cyrille Berger

January 23, 2017

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 git@gitlab.ida.liu.se:tdde05/air_labs.git master
```

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.

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:

- `/home/TDDE05/software/catkin_ws/src/lrs_march/march`

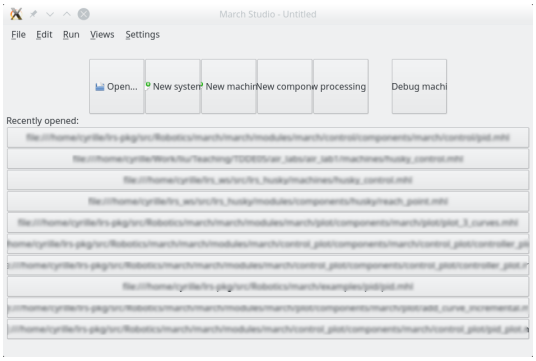

Figure 1: *March Studio*'s welcome screen

- `/home/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.

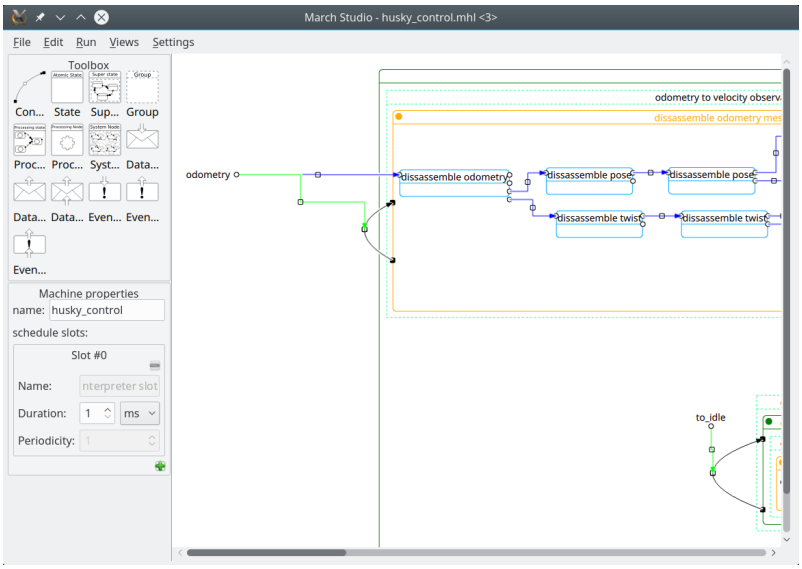

Figure 2: Husky control machine opened in *March Studio*

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 Linear/angular 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

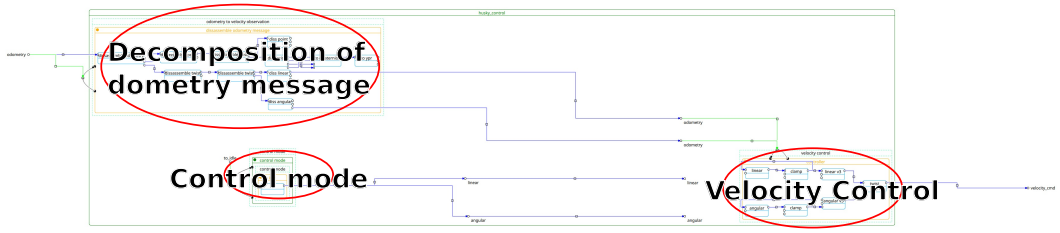

Figure 3: Overview of the control machine

velocity of the robot, the simple controller of the first lab is not good enough. The gold standard in control to solve this problem is the PID (proportional–integral–derivative) controller, see figure 4.

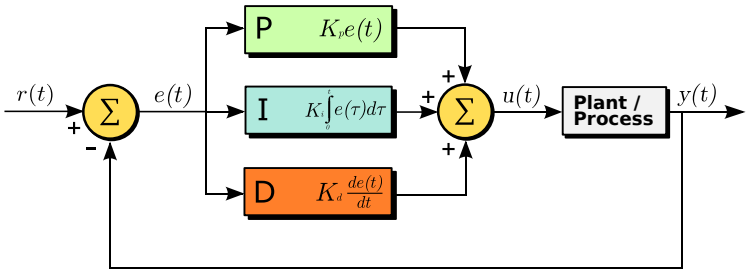

Figure 4: PID controller (source: wikipedia)

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

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

In a discrete time system:

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

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 acceleration. In case of the Husky, the input and output are velocity, the input correspond to the velocity that we want the robot to have while the output correspond to the velocity of the wheel. $e(t)$ therefore correspond to the error between the target velocity and the odometer velocity. While $u(t)$ is the update to the wheel velocity that is then given to the robot, so that:

$$v_{wheel}(t) = v_{wheel}(t-1) + u(t) \tag{3}$$

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 output of the PID controller is clamped, the clamping is already done for you, as well as equation 3. 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.

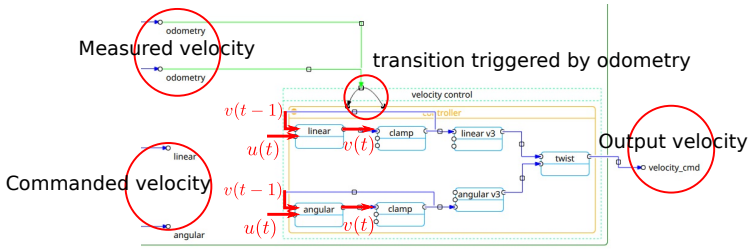

Figure 5: Velocity control

3.3 In practice

For this part of the lab you will need to use two PID loops, one for controlling the linear velocity and one for controlling the angular velocity. 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)
- `current_time (std_msgs/float64)` expressed in seconds
- `initial_previous_time (std_msgs/float64)` which you can ignore

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 

to activate the *processing node creation tool*.
- Then click within the *controller* state to create a new node. You will need three of them, so click three 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.

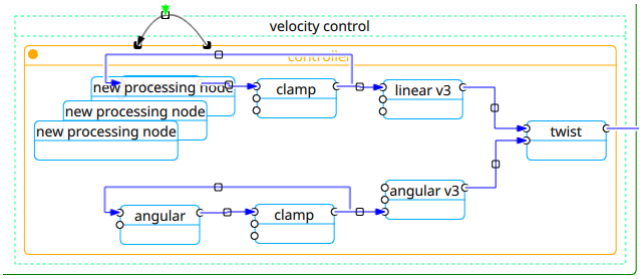

Figure 6: Velocity control with three new nodes.

Then you need to assign the type for each of those new nodes, two of them should be `march_control/pid` and one `march_ros/current_time`. 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 will need to connect the control and observation to the desired velocity and the odometry, using the connection tool 

. 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.

It should then look like on figure 7.

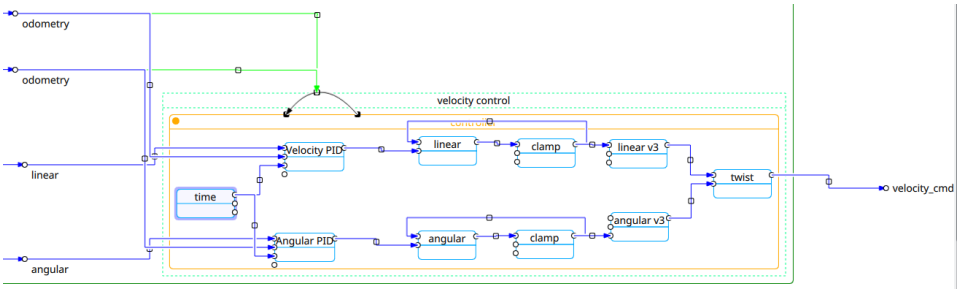

Figure 7: Velocity control with PID nodes.

One last thing you need to do is to adjust the limit of the clamping of the velocity. The `march_math/clamp` operation take as input:

- `arg0 (number)` the value to clamp
- `arg1 (number)` the minimum value
- `arg2 (number)` the maximum value

It is possible to set a default value for an input connector (without creating a specific node), to do that, you can click on a connector of clamp, and then in the property set the *initial message*. A reasonable clamping for the velocity is -1 to 1 (both for linear and angular).

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 been created for you and can be seen of figure 8, it contains a single idle mode.

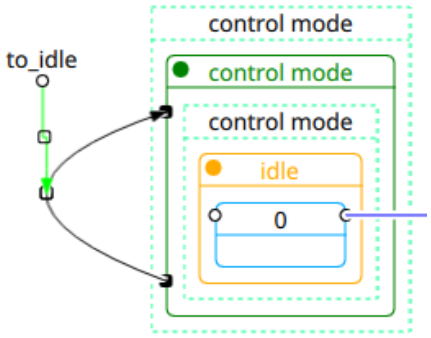

Figure 8: Control modes super state.

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 reseting 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 input connector* using the *external input data creation tool* 

. You can set the name to

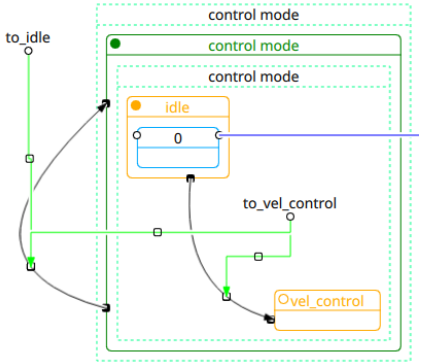

Figure 9: Control modes with *idle* mode and *vel_control* mode.

`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.

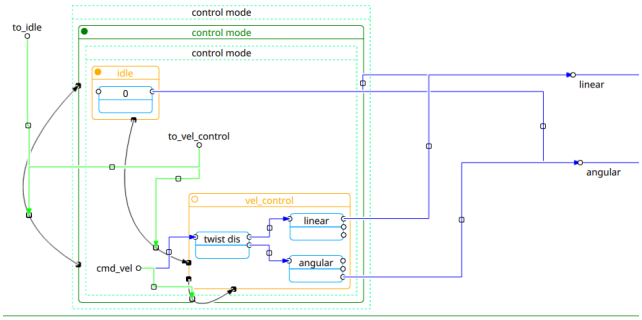

Figure 10: The *vel_control* state is finished and ready to run.

3.5 Run the controller

First you will need to build it:

```
load_ros
cd ~/TDDE05/catkin_ws
catkin build
```

Then in `packages.xml` add:

```
  <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:

```
rosrun nodelet nodelet standalone air_lab1/husky_control_node
        __ns:=/husky0
```

You can also add it to a launch file:

```
<node pkg="nodelet" type="nodelet"
      name="HuskyControlNodelet"
      args="standalone air_lab1/husky_control_node"  >
```

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:

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

And then to switch to *velocity control*:

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

Then you can use rqt to steer your robot around like in the first lab (set the topic to `/husky0/cmd_vel`).

3.6 Troubleshooting

If the robot does not move:

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

3.7 Calibrate the PID loops

Then you will need to calibrate the PID loops. This is a try/error operation, you should use the `evaluate_controller.py` script developed during lab0 to evaluate the quality.

4 Position controller

In the remainder of the lab, we are going to implement three control mode related to move the robot toward a position, or a set of positions. From the book *Introduction to Autonomous Mobile Robots*, you should read section 3.6.

Since the position controller is going to be used by three different mode, it should be added in its own group (at the same level as *odometry*, *control mode* and *velocity control* groups). You can create a new group with the *group creation tool* 

.

The controller itself is already implemented, it is available in the `air\_lab1_march` module. First you need to load the module, to do that, in the *file* menu, select *load state description libraries...* and then load the module from `${HOME}/TDDE05/catkin_ws/src/air_labs/air_lab1/modules/`.

The controller should only output velocity when we are in position control mode (so not when idle, nor when in velocity mode), so in your group, you should have two states, one for idling and one that is set to `air\_lab1_march/reach_point`.

The `air\_lab1_march/reach_point` takes three inputs:

- `cmd_position (ros_geometry_msgs/pose_stamped)` the destination of the robot (come from the control mode)
- `cmd_max_vel (ros_geometry_msgs/twist)` the maximum velocity that the robot should drive when reaching the point (should be set from outside the state machine with an external connector)
- `current_position (ros_geometry_msgs/pose)` the current position from the odometry

It has two outputs:

- `linear (std_msgs/float64)` linear velocity
- `angular (std_msgs/float64)` angular velocity

Those should get connected to your PID controllers.
It has three parameters:

- 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

$$k_\rho > 0; k_\beta < 0; k_\alpha - k_\rho > 0 \tag{4}$$

The idle state should output a velocity of 0. The idle state is the inital state, to set the initial state you need to click in the circle in the top left corner of the state.

Transition you will need to create a transition from *idle* state to *reach_point*, this transition should be triggered every time a new position is set. You will need a recursive transition on *reach_point* state that is triggered every time a new `current_position` is given by the odometry.

You will also need to create a transition from the *reach_point* state to *idle* state, which should be triggered once the robot reach its destination. This transition should be triggered on the event *finished*, but only if the robot has reached its destination, which means we need to prevent the transition to be triggered until the condition is reached. This can be done using a *guard*.

Guard You can define a guard using the expression field of the transition option. The expression should be a valid C++ expression, in your case it should check that the current position of the robot is close to the target position. By default, guards on a transition only have acccess to the output of the source state. However, you can connect any data source to a transition, you have already done it to trigger events, but you can also do it to give data to the transition:

- With the connection tool, connect the odometry pose to the transition

- By default it creates an event connection, click on the event connection, and in the property select *data*
- You also need to set a name, this name will be used in the guard expression as a variable name.

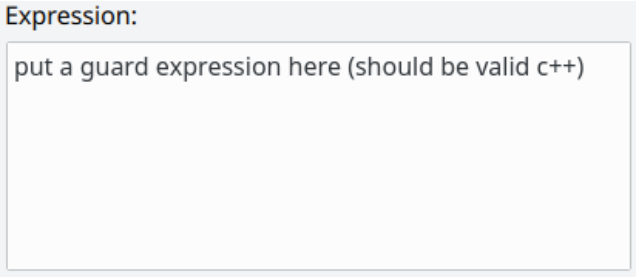

Figure 11: The expression box to set a guard on a transition.

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 an event input connector named `to_position_control`
- connect everything similarly to the *vel_control* state

The output of `march_std_lib/pass_through` will be used as input to the position controller that you implemented in the previous section.

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`.

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`

In the control mode 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. 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.

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

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

7 Position queuing

The final step is to create a mode in which you enqueue position received on the topic `enqueue_position`. To switch to that mode you should use an event called `to_position_queue_control`. You can use `march_std_lib/list_append` to add a new element to the queue. You can use Rviz to set the destination position.

8 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.