Vehicle Steering

Wyatt Newman
February, 2015

A feedback control system requires sensory feedback and an input command (a reference input). As described in the document "Desired State Generation", a path plan can be converted into a stream of desired-state updates. In the example desired-state generator, desired states are published on the topic "desState" as messages of type nav_msgs::Odometry. These messages specify the current desired pose (including x, y and heading) and the current planned twist (speed and angular velocity). These messages are updated at the rate UPDATE_RATE (defined in the header file of the desired-state generator), currently set to 50Hz.

The desired state must be compared to the measured state, which is received on the "odom" topic, and which is also a message of type nav_msgs::Odometry, which includes both pose and twist.

The package "example_steering_algorithm" contains a node of the same name. This code is constructed to provide the necessary hooks to odom and desState. However, it does *not* contain feedback logic (this needs to be implemented).

The SteeringController object has a class constructor that performs the overhead of setting up publishers, subscribers and services, as well as initializing important states. The steering algorithm publishes twist commands to topic "cmd_vel". For convenience of rosbag post-processing, it also publishes a time-stamped twist to the topic "cmd_vel_stamped."

The job of the steering controller is to compare desired and actual states, and from this information compute an effective twist command, which is then published for execution by the robot. Feedback is performed in "odom" coordinates, which are fast and smooth. (Drift is accommodated by sending desired states in odom coordinates).

Having instantiated a steeringController object, the main program loops at a fixed frequency (UPDATE_RATE, defined in the steeringController header file) to execute the function:

steeringController.my_clever_steering_algorithm();

Callback functions on odom and desState constantly update member variables, which are conveniently available to the steering algorithm. Inside my_clever_steering_algorithm(), trajectory-following errors are computed as:

```
pos_err_xy_vec_ = des_xy_vec_ - odom_xy_vec_; // vector pointing from odom x-y to desired x-y
lateral_err = n_vec.dot(pos_err_xy_vec_); //signed scalar lateral offset error;
trip_dist_err = t_vec.dot(pos_err_xy_vec_); // progress error
heading_err = min_dang(des_state_phi_ - odom_phi_); // if positive, should rotate +omega
```

The odom state is compared to the desired state and interpreted in terms of a lateral offset error, a heading error and a trip distance error. Ideally, the controller will succeed in driving all of these errors to zero, by judicious choice of cmd_vel speed and angular velocity commands.

For analysis purposes, these error quantities are stored in a message object containing a variable-length vector:

```
    steering_errs_.data.clear();
    steering_errs_.data.push_back(lateral_err);
    steering_errs_.data.push_back(heading_err);
    steering_errs_.data.push_back(trip_dist_err);
```

This message is then published on the topic “steering_errs”:

```
    steering_errs_publisher_.publish(steering_errs_); // suitable for plotting w/ rqt_plot
```

These errors can be rosbagged, or they can be visualized in real time with rqt_plot. For the latter, the topics to be displayed should be chosen as:
/steering_errs/data[0] (the lateral offset error)
/steering_errs/data[1] (the heading error)
/steering_errs/data[2] (the trip distance error)

TO DO:
The steering code provided integrates with desired state, odom and cmd_vel. ***However***, the actual steering algorithm is not implemented. (This is left to you to complete).

Note the lines following computation of lateral and heading error:

```
     // do something clever with this information
    controller_speed = des_state_vel_; //you call that clever ?!?!?!?
    controller_omega = des_state_omega_; //ditto
```

This will cause the robot to move—but it is no smarter than open-loop dead reckoning. The above values should be computed based on a valid feedback algorithm.

Note the use of:

```
//utility fnc to compute min dang, accounting for periodicity
double SteeringController::min_dang(double dang) {
    while (dang > M_PI) dang -= 2.0 * M_PI;
    while (dang < -M_PI) dang += 2.0 * M_PI;
    return dang;
}
```

For performing feedback on heading, it is important to choose which way to spin. Since rotation is periodic, you must examine the periodic options to choose the closest approach to desired heading.

Note, also, some reasonable protection:

```
    controller_omega = MAX_OMEGA*(controller_omega/MAX_OMEGA);
```

This line saturates the omega command to +/- a reasonable maximum value.

Also note that a feedback controller can benefit from both feedback *and* feedforward. The line:

```
        controller_omega = des_state_omega_;
```

almost works. Instead of ignoring the planned omega, it should be augmented with:

```
        controller_omega += feedback_omega_correction; //to be computed
```