Human Following Robot || Linear MPC

Pseudocode || Daksh Raval

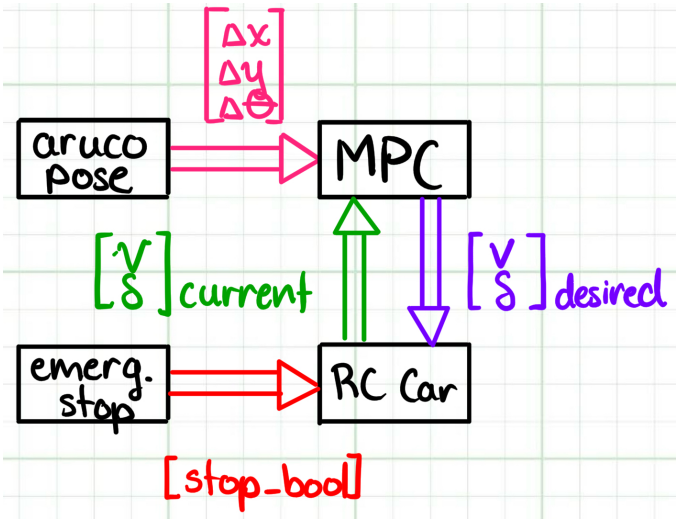
February 19, 2026

Contents

1 ArUco Pose

1.1 ROS Node Connections

Nutshell 1. *One to estimate ArUco pose, other for MPC*



Nutshell 2. *Detect markers, plug into PnP solver, get pose. Mostly out-of-the-box, difficulty lies in formatting chain of data transferring*

```
import cv # openCV

MARKER_SIZE = 0.10 # meters
```

```
CAMERA_INTRINSCIS = [focal, principal, distortion] # set manually

subscribe(rgb_image)

def main():
    aruco_corners = cv.detectArUco(rgb_image)

    # spits out pose from camera to ArUco in camera/OpenCV frame
    trans_vec, rot_vec = cv.ArUcoPose(aruco_corners, CAMERA_INTRINSICS)

    # camera frame: [Z, Y, X] = [Forward, Down, Right]
    # RC car frame: [X, Y, Z] = [Forward, Left, Up]
    convert_frames(trans_vec, rot_vec)
    != not sure how to convert rotation vector

    # invert so ArUco reference
    pose = - [converted_trans_vec, converted_rot_vec]
    != not sure if i should do this cause camera to ArUco is ArUco - camera

    publish(aruco_pose)
```

2 MPC

Nutshell 3. *Linearize and discretize at operating point, plug into MPC solver, get trajectory.*

2.1 Overarching Pseudocode

Nutshell 4. *Functional programming. Callbacks = 'don't bother until you got mail for me'.*

- `make_mpc_template()` *: compile boilerplate 'optimization-problem'-object for CasADi solver*
- `parse_prev_control_callback()` *: unpack actual throttle/steer motor encoder values from Float32MultiArray*
- `trajectory_planner_callback()` *: when new human position comes, compute new trajectory*
- `get_linearized_matrices()` *: linearize matrices at operating point*
- `solve_mpc()` *: plug into CasADi's IPOPT solver, get trajectory*
- `pack_publish_control()` *: turns numpy array into Twist and publishes to car*
- `differential_wheel_speed()` *: computes rear wheel speeds using Ackerman kinematic model*

```
def main():
    solver_template = make_mpc_template()

    subscribe('/current_vel_steer', parse_prev_control_callback)
    subscribe('/aruco_pose_topic', trajectory_planner_callback)

def make_mpc_template():
    mpc_template = state_op_param, A_d_param, B_d_param, ..., x_ref_param
    return mpc template

def parse_prev_control_callback():
    return control_op

def get_linearized_matrices(state_op, control_op, dt, L):
    linearized_matrices = A_d, B_d, d_d
    return linearized_matrices

def trajectory_planner_callback(aruco_pose):

    trajectory = solve_mpc(mpc_template, linearized_matrices, state_op,
    ↪  control_op)
    next_action = trajectory[0]
    pack_publish_control(next_action)
    return next_action

def pack_publish_control(next_action):
    left_wheel_speed, right_wheel_speed = differential_wheel_speed()
    return next_action_Twist_vector

def differential_wheel_speed(velocity, steer):
    return left_wheel_speed, right_wheel_speed
```

2.2 Overarching Pseudocode

```
subscribe(aruco_pose) # [x, y, \theta] from aruco tracking/pose estimation
subscribe(speed_steer) # [v, \delta] from car's motor encoders

def parse_state_callback(aruco_pose):



def get_linearized_matrices(state: list, control: list):


horizon = 10 # intervals
```

```
optim = casadi.Optim()

X = optim.variable(3, horizon + 1) # 3 states, 3x11 matrix for storage
U = optim.variable(2, N)

A_d = optim.parameter(3, 3) # matrix size, custom casadi storage object
B_d = optim.parameter(3, 2)
X_current = opti.parameter(3) # only current state vector

opti.subject_to(X[0] == X_current) # start at current state

for k in range(horizon):
    x_next = X[k+1]
    x_current = X[k]
    u_current = U[k]

    # constrains to state-space model
    optim.subject_to(x_next == A_d @ x_current + B_d @ u_current # matrix-vector
    ↪  multiplication

    # control constraints
    optim.subject_to(-1.0, U[0], 1.0) # velocity
    optim.subject_to(-0.45, U[1], 0.45) # steer


### Cost Function setup


# weights
Q = [10, 10, 1] # tracking error penalizer
R = [1, 1] # control effort penalizer

cost = 0
for k in range(horizon):
    state_error = X[k] # pose, minus some offset, is simply the state error
    control_effort = U[k]
    cost = xˆT @ Q @ x + uˆT @ R @ u

## Actual Trajectory Computation
casadi.minimize(cost)

# main loop
while true:
    x, y, theta = aruco_pose
    v, delta = speed, steer

    dt = time_since_last_loop()
```

```
A_lin = linearize_at_operating_point(v, theta)
B_lin = linearize_at_operating_point(v, theta, delta)
optim.A_d = I + A_lin * dt # I is identity matrix
optim.B_d = B_lin * dt

try:
    trajectory = optim.solve()

    u_next = trajectory[0]
```