MPC Path Following for a Bicycle

Josephine King, Peter Johnson

***Abstract*—This paper outlines a Model Predictive Controller for a bicycle model. In MATLAB simulations, the controlled bicycle was able to follow multiple trajectories optimally. The main limitation of the controller was that it relied on third order polynomial fitting of the provided waypoints. Thus, a badly-fit trajectory caused the bicycle to follow a wobbly or inefficient path.**

I. Introduction

FOR our final project, we modelled an autonomous bicycle and performed trajectory tracking using Model Predictive Control (MPC). Bicycle models are commonly-used approximations for cars in robotics applications, which is valuable for us to practice modeling. Model Predictive Control is often used for autonomous navigation and differs from classical control in that it uses real-time, repeated optimization to choose optimal control inputs. The main advantage of MPC is that it allows the current time step's control effort to be optimized, while also taking into account future time-steps. This is achieved by optimizing over a finite time-horizon (number of time steps to look ahead), but only implementing the current time step's effort. To choose the optimal control values, an MPC controller minimizes an objective function, which is a weighted sum of various outputs, inputs, and state parameters. MPCs optimize at each time step, thus differing from Linear-Quadratic Regulators (LQR), which optimize just once. It is also much easier to tune certain specifications of the controller using MPC, as these specifications can be included as constraints in the optimization problem.

II. Background

The Model Predictive Control Problem is formulated as follows [1]:

$$U_t^*(x(t)) := \underset{U_t}{argmin} \sum_{k=0}^{N-1} q(x_{t+k}, u_{t+k})$$

$$\begin{aligned} s.t.\quad & x_t = x(t) && \text{measurement} \\ & x_{t+k+1} = Ax_{t+k} + Bu_{t+k} && \text{system model} \\ & x_{t+k} \in \chi && \text{state constraints} \\ & u_{t+k} \in \mathcal{U} && \text{input constraints} \\ & U_t = u_t, u_{t+1}, ..., u_{t+N-1} && \text{optimization variables} \end{aligned}$$

We can see that the MPC problem is an optimization problem, where the objective function to be minimized is a cost function based on the system states and inputs. The system dynamics are included as constraints, in addition to other constraints, such as bounds on input and state variables.

At each sample time, the current state is observed and then the optimal control sequence for the entire planning window (N) is calculated : $U_t = u_t, u_{t+1}, ..., u_{t+N-1}$ However, only the first control effort in the sequence is applied.

III. Problem Statement

Our problem statement is:

"Design a Model Predictive Controller to steer a vehicle along a trajectory defined by a series of way-points as closely as possible while using the minimal amount of control efforts and minimizing the rates of change in the control efforts to make the ride more comfortable for a passenger."

We will assume that some higher-level path planner has provided a series of way-points to our vehicle controller.

IV. System Modeling

We chose to use the bicycle model as the basis for our system modeling. It is more representative of real motion than other models, such as Dubin's car, but is still simple enough to be appropriate for our purposes. Figure 1 shows a bird's eye view of the bicycle with model variables labeled.

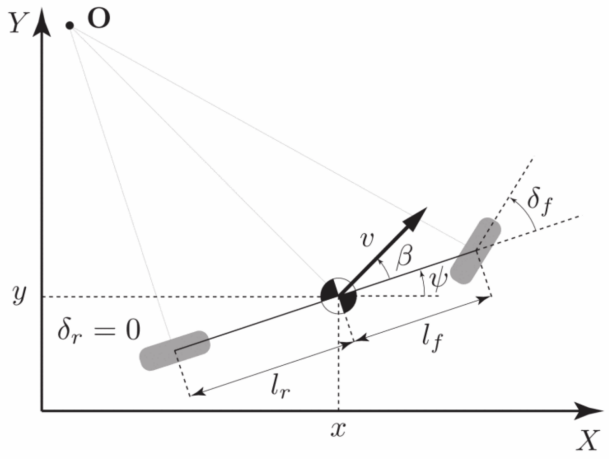

Fig. 1. Model of bicycle. The two tires are shown in grey, along with state and input values. [3]

We define x and y to be the location of the center of mass of the vehicle in the global coordinate system. The vehicle is two wheels (the front and the rear) connected by a single line. l_f and l_r are the distances from each wheel to the center of mass. We define ψ to be the heading of the vehicle (angle from the x axis). We define δ_r and δ_f to be the steering angles. The rear wheel is fixed, so $\delta_r = 0$ is fixed. δ_f is one of the control inputs, which allows us to control the steering angle of the vehicle. v is the overall linear velocity of the vehicle. This will be related to our other control input, a, which is acceleration due to pedaling or brakes. β is the angle of v relative to the vehicle's axis. β is dependent on a variety of factors including wheel traction. For simplicity, we will assume there is no

slip and $\beta = 0$. These simplifications allow us to to create a simple kinematic model of the system as opposed to a more complicated dynamic model.

Based on work by Kong et al., we derived the following discrete-time kinematic equations [3]:

$$\begin{aligned} x_{t+1} &= x_t + v_t cos(\psi_t)dt \\ y_{t+1} &= y_t + v_t sin(\psi_t)dt \\ v_{t+1} &= v_t + a_t dt \\ \psi_{t+1} &= \psi_t + \frac{v_t}{l_f}\delta_t dt \end{aligned}$$

With an appropriate time step, our kinematic equations should be a sufficient model of the real life system. We introduce two other states to keep track of how well the trajectory is being followed. These are the cross track error (cte) and the error in heading ($e\psi$). We define the cross track error as the difference between the trajectory fitted to the way points and the vehicle's current trajectory.

$$\begin{aligned} cte_{t+1} &= f(x_t) - y_t + v_t sin(e\psi_t)dt \\ e\psi_{t+1} &= \psi_t + \psi_{des}\frac{v_t}{l_f}\delta_t dt \end{aligned}$$

To devise the trajectory, we fit a third order polynomial to the next three way-points the bicycle must pass through [2]. However, if we only fit y values to x values, then the bicycle cannot follow any curve that does not pass the vertical line test. In order to resolve this problem, we instead fit x values to y values if a curve fit to the next three way-points would not pass the vertical line test. This method still has limitations; it cannot fit curves in the case that the next three way-points do not pass either the vertical or horizontal line test. However, this problem could be remedied by adding more way-points in between existing way-points, which results in a more easily-fit sequence.

For many trajectories, a third-order polynomial fit results in a smooth and natural path that resembles one followed by a human driver. Most trajectories that a bicycle would encounter, such as a winding street or bike path, could be well fit with a third-order polynomial, making this approximation appropriate. For trajectories with sharp turns and corners, such as a polygon, it seems like simply drawing a straight line between points using a piece-wise function would be more appropriate. However, sharp corners are impossible to actuate. An ideal path would probably have rounding on its corners.

Alternatively, we could have transformed way-points to the bicycle's frame and performed polynomial fitting in that frame. However, given the time constraints of the project, we chose to go with polynomial fitting in the global frame, which allows for a more easy calculation of the heading.

Using polynomial fitting, the desired y-position and heading are

$$\begin{aligned} y = f(x) &= a_3x^3 + a_2x^2 + a_1x + a_0 \\ tan(\psi_{des}) &= \frac{dy}{dx} = 3a_3x^2 + 2a_2x^2 + a_1x \end{aligned}$$

This model, with the six state variables x, y, v, ψ, cte, and $e\psi$ is applied as a constraint in our optimization problem. Our model is non-linear, so we had to use a non-linear MPC solver to solve the optimization problem.

In simulating our model, we assumed no noise in our system. We chose to omit both process noise in our state transition equations and noise in our measurements. In real life, our control efforts would not be perfectly actuated and the actual motion of the vehicle would be affected by unmodeled disturbances. It is also likely that we would not be able to observe every state through a sensor. Therefore, we would also have needed to construct an observer or state estimator to produce an optimal estimate of the entire state based on several known noisy measurements. To simplify the work that was being done and to focus on the MPC optimization aspects of our project, we designed the system assuming we had perfect actuation of our control efforts and perfect sensors for each state.

V. Optimization Problem

The optimization problem is to calculate the the optimal control inputs for the bicycle model so that it will smoothly follow a trajectory defined by a sequence of way-points. The solver chooses optimal control inputs by minimizing the cost function, which we designed to achieve the following objectives:

- Ability to follow trajectory
  - Cross track error - following as closely to the trajectory is important
  - Error in heading - As something that affects cte, this is also important
  - Error in velocity - For comfort, we generally want to be going at a constant speed. (We ignore the need to slow down in curves, as our model does not account for traction)
- Minimize necessary control effort (less actions taken is desirable)
  - Steering angle - better to just be straight, not turning if possible
  - Acceleration - better to not be accelerating (or braking)
- Minimizing the rate of change in control value
  - rate of change in steering angle - turning too quickly is not ideal
  - Rate of change in acceleration (jerk) - want a smooth ride

Based on these objectives, we defined the following cost function

$$\begin{aligned} J = \sum_{t=1}^{N} w_{cte}\|cte_t\|^2 + w_{e\psi}\|e\psi_t\|^2 + w_v\|v_t - v_{ref}\|^2 \\ + \sum_{t=1}^{N-1} w_\delta\|\delta_t\|^2 + w_a\|a_t\|^2 \\ + \sum_{t=2}^{N} w_{\dot{\delta}}\|\delta_t - \delta_{t-1}\|^2 + w_{\dot{a}}\|a_t - a_{t-1}\|^2 \end{aligned}$$

In order to follow the path, we weight cross track error the highest: $w_{cte} = 100$. All other weights (w values) were 1.

When setting the optimization problem, two of the most important tuning parameters were dt, the time-step size, and N, the number of time-steps over which to calculate the cost function. The actual amount of time we look ahead is $T = dt * N$. Both dt and N can be reasonably guessed at first and then tuned. The size of dt directly affects our kinematic model. The smaller dt is the more accurate our model will be, but there is a trade-off with computation time. Picking N can be done by determining the amount of time that the system needs to look ahead. Since the bicycle is not moving incredibly fast, looking ahead for 1s seemed appropriate. Our model worked with $dt = 0.1$ s and $N = 10$ time steps ahead for an overall look-ahead time of $T = 1$ s.

We use MATLAB's Nonlinear Model Predictive Control Toolbox to solve our optimization problem. The toolbox solves a nonlinear programming problem using the fmincon function with the Sequential Quadratic Programming (SQP) algorithm. Nonlinear MPC optimization problems often allow multiple solutions (as they could have local minima). Because of this, it is important to provide a good starting point or initial guess near the global optimum. To do so, we use the predicted state and control inputs from the previous control interval as the initial guesses for the current control interval. To use the previous state and control values as initial guesses, we return an opt output argument when calling nlmpcmove. This nlmpcmoveopt object contains the initial guesses for the state (opt.X0) and manipulated variable (opt.MV0) trajectories, and the global slack variable (opt.Slack0). In our case, we wanted our constraints to be hard, so there is no slack in the system. We constrain both control inputs to be reasonable values. The steering angle, df, is limited to 45° and the acceleration, a, is limited to 1 m/s.

Using variable names that resemble our MATLAB code, our optimization problem is:

$$
\begin{aligned}
MV_t^*(x(t)) := \underset{MV_t}{argmin} & \sum_{k=0}^{9} J(x_{t+k}, MV_{t+k}) \\
s.t. \quad & x_t = x(t) && \text{measurement} \\
& x_{t+k+1} = model(x_{t+k}, mv_{t+k}) && \text{system model} \\
& -45^\circ \leq mv_{t+k}(1) \leq 45^\circ && \delta_f \text{ constraints} \\
& -1 \leq mv_{t+k}(2) \leq 1 && a \text{ constraints} \\
& MV_t = mv_t, , ..., mv_{t+9} && \text{opt. variables}
\end{aligned}
$$

Our optimization problem is shown in full in our MATLAB code, which is attached in the Appendix.

VI. Results and Discussion

Our bicycle was able to successfully follow multiple paths, as shown in bird's eye views of the bicycle in Figures 2, 3, and 4. The bicycle is shown as a red line, the way-points are circles, and the path followed by the bicycle is a dotted line. Figures 2 and 3 show that the bicycle was able to successfully and smoothly follow the polynomial path in order to reach all of the way-points within a 0.5 m radius. Due to a limited turning angle, the bicycle's trajectory around sharp turns was more wobbly, as shown in Figure 2.

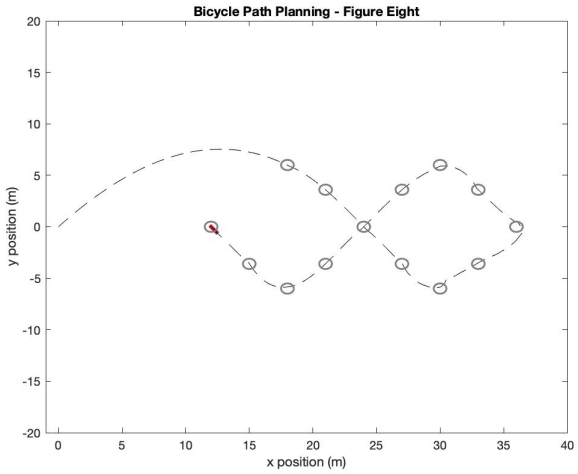

Fig. 2. Bicycle successfully following a path in the shape of a figure eight.

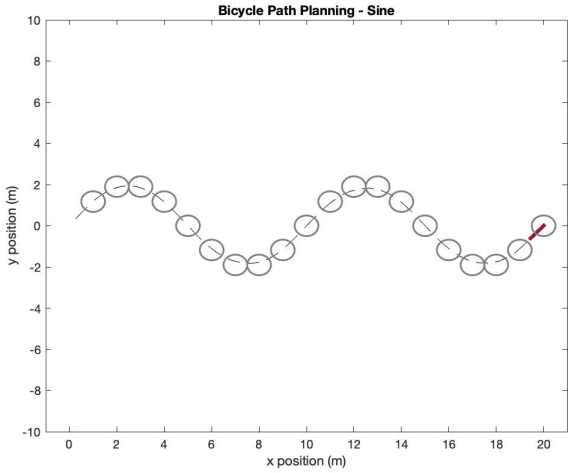

Fig. 3. Bicycle successfully following a path in the shape of a sine wave.

Because the cross track error was weighted so highly in our objective function, the bicycle followed polynomial paths very faithfully. Thus, most of the problems with the bicycle's navigation came from issues with polynomial fitting. While the bicycle could successfully follow trajectories that are easily fit with polynomials, such as a sine function, it had more trouble navigating other trajectories, as shown by the hexagonal trajectory show in Figure 4. The low cross track errors at y = 15 and y = -3 show that the bicycle was faithfully following the polynomial path; the polynomial path was just not very well-fit to the way-points (Figure 5).

In practical applications, such as steering a bicycle down a road or bike path, most trajectories could be well-approximated by a polynomial. Thus, despite the limitations of polynomial fitting, we believe that this method could function well in everyday applications.

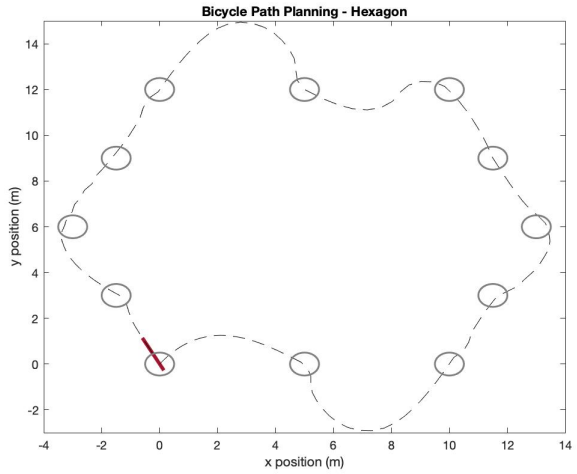

Fig. 4. Bicycle following a path in the shape of a hexagon.

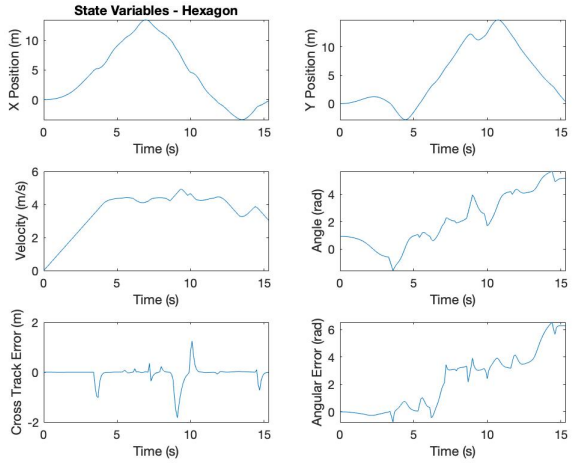

Fig. 5. State variables over time as bicycle follows hexagonal path.

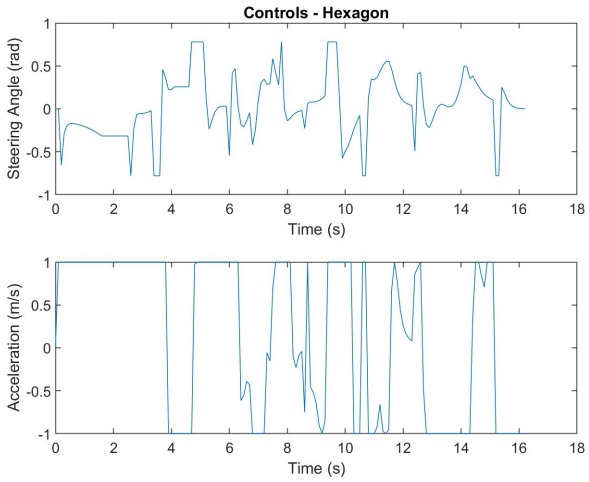

Fig. 6. Control variables over time as bicycle follows hexagonal path.

REFERENCES

[1] Francesco Borelli. "Model Predictive Control: Algorithm, Feasibility ad Stability". 2014.

[2] Jonathan Hui. "Lane keeping in autonomous driving with Model Predictive Control PID". In: *Medium* (2018).

[3] Jason Kong et al. "Kinematic and dynamic vehicle models for autonomous driving control design". In: *IEEE Intelligent Vehicles Symposium (IV)* (2015).

APPENDIX A
MATLAB CODE

A. *mpc_main_final.m*

```
%% Simulate Closed-Loop Control using Nonlinear MPC Controller

% Define Constants
global L beta poly_coef v_ref switchxy;
L = 0.8;
beta = 0;
v_ref = 4.4;
Ts = 0.1;

%% Create the nonlinear MPC object, based on starter code from MATLAB pendulum
   example
% Create a nonlinear MPC controller with 6 states, 6 outputs, and two input.
nlobj = nlmpc(6,6,2);

% Specify the sample time and horizons of the controller.
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 10;

% Specify the state function for the controller, which is in the file |mpcModel.m|.
% This discrete-time model is based on the kinematics of a bicycle model
nlobj.Model.StateFcn = "mpcModel";
nlobj.Model.IsContinuousTime = false;

% The prediction model uses an optional parameter, |Ts|, to represent the
% sample time. Specify the number of parameters.
nlobj.Model.NumberOfParameters = 1;

% Specify the output function of the model, passing the sample time parameter
% as an input argument. In our case, output is just the state
nlobj.Model.OutputFcn = @(x,u,Ts) [x(1); x(2); x(3); x(4); x(5); x(6)];

% The custom cost function replaces the default cost
% function that is typically used in MATLABS's NLMPCfeedback control.
nlobj.Optimization.CustomCostFcn = "mpcCostFunction";
nlobj.Optimization.ReplaceStandardCost = true;

% Assigns constraints to manipulated variables
nlobj.MV(1).Min = -.78; % Max steering angle is 45 degrees
nlobj.MV(1).Max =  .78;
nlobj.MV(2).Min = -1;   % Max Acceleration is 1 m/s^2
nlobj.MV(2).Max =  1;

% Specify the output reference value.
% Does not matter since we use custom cost
yref = [0 0 0 0 0 0];

%% Waypoints and initial values
% Figure Eight
%x_gdes = 3.*([2 3   4  5    6  7   8 7   6 5   4  3    2  1   0]+4);
%y_gdes = 3.*([2 1.2 0 -1.2 -2 -1.2 0 1.2 2 1.2 0 -1.2 -2 -1.2 0]);
% Hexagon
x_gdes = [5 10 11.5 13 11.5 10 5  0 -1.5 -3 -1.5 0];
y_gdes = [0 0   3   6   9   12 12 12 9    6   3  0];
```

```
% Sine
%x_gdes = (1:20);
%y_gdes = 2*sin(2*pi*x_gdes/10);
% Exponential
%x_gdes = [0:10]
%y_gdes =e^x_dges

% Obtain initial fit for trajectory
x0 = 0;
y0 = 0;
switchxy = 0;
poly_coef = polyfit([x0 x_gdes(1:3)], [y0 y_gdes(1:3)], 3);
dy_dx = poly_coef(3) + 2*poly_coef(2)*x0 + 3*poly_coef(1)*x0^2;

% Calculate initial heading
theta_des = atan2(dy_dx, 1);

% Set initial values
x = [x0; y0; 0; theta_des; 0; 0];
mv = [0; 0];

% Good practice to validate nlmpc object
validateFcns(nlobj, x, mv, [], {Ts});

%% Simulate the bicycle going to waypoints
% Create an |nlmpcmoveopt| object, and specify the sample time parameter.
nloptions = nlmpcmoveopt;
nloptions.Parameters = {Ts};

% Create variables to save state and input data
xHistory = x;
mvHistory = mv;

for i = 1:length(x_gdes)

    % calculate the index of the last waypoint to include in poly fitting
    j = min(i+2, length(x_gdes));

    % don't recalculate polynomial when there aren't enough points to
    % calculate two ahead
    if (i + 2 < length(x_gdes))
        angle = mod(x(4), 2*pi);
        % if driving forward (+x direction) and the next way points are
        % in -x direction, fit x to y
        if (((angle <= pi/2) || (angle >= 3*pi/2)) && ((x_gdes(i) <= x(1)) || (x_gdes
    (i+1) <= x_gdes(i)) || (x_gdes(i+2) <= x_gdes(i+1))))
            switchxy = 1;
            poly_coef = polyfit([x(2) y_gdes(i:j)], [x(1) x_gdes(i:j)] , 3);
        % if driving backward (-x direction) and the next way points are
        % in +x direction, fit x to y
        elseif (((angle > pi/2) && (angle < 3*pi/2)) && ((x_gdes(i) >= x(1)) || (
    x_gdes(i+1) >= x_gdes(i)) || (x_gdes(i+2) >= x_gdes(i+1))))
            switchxy = 1;
            poly_coef = polyfit([x(2) y_gdes(i:j)], [x(1) x_gdes(i:j)] , 3);
        % otherwise, trajectory passes vertical line test
        % fit y to x
        else
            poly_coef = polyfit([x(1) x_gdes(i:j)], [x(2) y_gdes(i:j)], 3);
```

```
            switchxy = 0;
        end
    % at the end, just go straight to the final point by fitting with a
    % line
    elseif (i == length(x_gdes))
        switchxy = 0;
        poly_coef = polyfit([x(1) x_gdes(i)], [x(2) y_gdes(i)] , 1);
        poly_coef = [0 0 poly_coef];
    end

    % navigate to each point within a 0.5 m radius
    while ((x(1)-x_gdes(i))^2 + (x(2)-y_gdes(i))^2 >= 0.5^2)

        if (v_ref < 0.5) % come to a complete stop after slowing down
            v_ref = 0;
        elseif (i == length(x_gdes)) % as we approach last waypoint, slowdown
            v_ref = 0.9*v_ref;
        else
            v_ref = 4.4; % typical reference velocity about 10mph
        end

        % calculate the optimal move
        [mv, nloptions] = nlmpcmove(nlobj,x,mv,yref,[],nloptions);
        % update state
        x = mpcModel(x, mv, Ts);
        % save state and control values
        xHistory = [xHistory x];
        mvHistory = [mvHistory mv];

    end
end

%% Plot the resulting state trajectories.
figure (3)
time = (0:Ts:(length(xHistory)-1)*Ts);
subplot(3,2,1)
plot(time,xHistory(1,:))
xlabel("Time (s)")
ylabel("X Position (m)");
title("State Variables - Sine");
hold on
subplot(3,2,2)
plot(time,xHistory(2,:))
xlabel("Time (s)")
ylabel("Y Position (m)");
hold on
subplot(3,2,3)
plot(time,xHistory(3,:))
xlabel("Time (s)")
ylabel("Velocity (m/s)");
hold on
subplot(3,2,4)
plot(time,xHistory(4,:))
xlabel("Time (s)")
ylabel("Angle (rad)");
hold on
subplot(3,2,5)
plot(time,xHistory(5,:))
```

```
xlabel("Time (s)")
ylabel("Cross Track Error (m)");
hold on
subplot(3,2,6)
plot(time,xHistory(6,:))
xlabel("Time (s)")
ylabel("Angular Error (rad)");

% Plot the control input values.
figure (4)
subplot(2,1,1)
plot(time,mvHistory(1,:))
xlabel("Time (s)")
ylabel("Steering Angle (rad)");
title("Controls - Sine");
hold on
subplot(2,1,2)
plot(time,mvHistory(2,:))
xlabel("Time (s)")
ylabel("Acceleration (m/s)");

%% Plot animated bicycle movement

global L;
figure(2)

% in order to save as a gif
gif('hexagon.gif');
gif('hexagon.gif','DelayTime',0.2,'LoopCount',5,'frame',gcf);

% set up components of plot
xpos = x0;
ypos = y0;
bikeHandle = plot(xpos, ypos, 'Color', [0.6350 0.0780 0.1840], 'LineWidth', 3);
h = animatedline('Color', 'k', 'LineStyle', '--');
circles = animatedline('Color', [0.5 0.5 0.5], 'LineStyle', 'none', 'LineWidth', 1.5,
    'Marker', '.', 'MarkerSize', 3);

% plot and axis settings, titles
xlim([min(0, min(x_gdes))-1,max(x_gdes)+1])
ylim([min(y_gdes)-3,max(y_gdes)+3])
xlabel("x position (m)")
ylabel("y position (m)")
title("Bicycle Path Planning - Hexagon")

% plot the waypoints
for i = 1:length(x_gdes)
    circle(circles,x_gdes(i), y_gdes(i), 0.5);
end

hold on

% plot the bike's position and path traveled
for i = 1:length(xHistory)

    % plot the bike's center position
    bike = [xHistory(1,i)-L*cos(xHistory(4,i)), xHistory(2,i)-L*sin(xHistory(4,i));
            xHistory(1,i)+L*cos(xHistory(4,i)), xHistory(2,i)+L*sin(xHistory(4,i))];
```

```
    bikeHandle.XData = bike(:, 1);
    bikeHandle.YData = bike(:, 2);

    % plot the bike's path
    addpoints(h,xHistory(1,i),xHistory(2,i));
    drawnow;
    gif;
end

% from Paulo Silva on MATLAB forum
function circle(circles,x,y,r)
    %x and y are the coordinates of the center of the circle
    %r is the radius of the circle
    %0.01 is the angle step, bigger values will draw the circle faster but
    %you might notice imperfections (not very smooth)
    ang=0:0.01:2*pi;
    xp=r*cos(ang);
    yp=r*sin(ang);
    addpoints(circles,x+xp,y+yp);
end
```

B. *mpcCostFunction.m*

```
function J = mpcCostFunction(X,U,Ts,e,data)

    global v_ref;
    [N,M] = size(X);

    % cross track error is the most highly weighted
    cte = 100*sum(X(:,5).^2);
    % error in heading
    etheta = 1*sum(X(:,6).^2);
    % error in velocity, where v_ref is the desired velocity
    v = 1*sum((X(:,3)-v_ref).^2);

    % minimize control values so as not to do unnecessary work
    df = 1*sum((U(:,1)).^2);
    a = 1*sum((U(:,2)).^2);

    % rate of change of control values (jerk)
    % minimize for smooth navigation
    dfrate = 1*sum((U(1:N-1,1)-U(2:N,1)).^2);
    arate = 1*sum((U(1:N-1,2)-U(2:N,2)).^2);

    % cost function
    J = cte + etheta + v + df + dfrate + arate;

end
```

C. *mpcModel.m*

```
function x = mpcModel(x0, u, Ts)
    global beta L poly_coef switchxy;

    %A discrete time nonlinear dynamic model of a bicycle
    x = [0;0;0;0;0;0];

    % if switchxy, then we use y as independent variable for polynomial
    % fitting
```

```
    % otherwise, we use x as the independent variable
    % this allows us to fit any curve that is either a function of x or y
    if (switchxy == 1)
        % differentiate to find heading
        dy_dx = poly_coef(3) + 2*poly_coef(2)*x0(2) + 3*poly_coef(1)*x0(2)^2;
        % calculate desired x value
        f_y0 = poly_coef(4) + poly_coef(3)*x0(2) + poly_coef(2)*x0(2).^2 + poly_coef
    (1)*x0(2).^3;
        % calculate cte
        x(5) = f_y0 - x0(1) + x0(3)*sin(x0(6))*Ts;
    else
        % differentiate to find heading
        dy_dx = poly_coef(3) + 2*poly_coef(2)*x0(1) + 3*poly_coef(1)*x0(1)^2;
        % calculate desired y value
        f_x0 = poly_coef(4) + poly_coef(3)*x0(1) + poly_coef(2)*x0(1).^2 + poly_coef
    (1)*x0(1).^3;
        % calculate cte
        x(5) = f_x0 - x0(2) + x0(3)*sin(x0(6))*Ts;
    end

    % find the desired heading
    theta_des = atan2(dy_dx, 1);
    % calculate vehicle values
    x(1) = x0(1) + x0(3).*cos(x0(4)+beta)*Ts;      % x
    x(2) = x0(2) + x0(3).*sin(x0(4)+beta)*Ts;      % y
    x(3) = x0(3) + u(2)*Ts;                        % v
    x(4) = x0(4) + x0(3)/L * u(1)*Ts;              % theta
    %angle = mod(x(4), 2*pi);
    angle = x(4);
    x(6) = angle - theta_des+(x0(3)/L)*u(1)*Ts;    % etheta

end
```

APPENDIX B
GIFS

Three animated gifs of the bicycle's movement were uploaded to the dropbox on Sakai: "figure_eight.gif", "sine.gif", and "hexagon.gif".