MPC CONTROLLER

Abdalla Mohamed

August 25, 2024

1 Principles of MPC

It is a control method that relies on solving an optimization problem at each time step. It predicts the future behavior of the system using a model and chooses control inputs that optimize a specific cost function subject to constraints. To begin with , the car model mathematical formulation shall be done and discretized , followed by formulating an adequate cost / objective function which defines our optimization problem .

2 The Kinematic Model

The discrete-time kinematic equations of motion for the car model are:

$$
\begin{aligned}
x_{k+1} &= x_k + v_k \cdot \cos(\psi_k) \cdot \Delta t \\
y_{k+1} &= y_k + v_k \cdot \sin(\psi_k) \cdot \Delta t \\
\psi_{k+1} &= \psi_k + \frac{v_k}{L} \cdot \tan(\delta_k) \cdot \Delta t \\
v_{k+1} &= v_k + a_k \cdot \Delta t
\end{aligned}
$$

- x_k and y_k represent the vehicle's position at time step k.
- ψ_k is the vehicle's orientation.
- v_k is the vehicle's velocity.
- δ_k is the steering angle which is one of our input, which determines the rate of change of the vehicle's heading.
- a_k is the acceleration which is our input .
- L is the wheelbase of the vehicle, i.e., the distance between the front and rear axles.
- Δt is the time step.

3 Assumptions of the Kinematic Bicycle Model

- **Kinematic Simplification**: The kinematic bicycle model captures the essential behavior of a car by reducing the complex dynamics of the vehicle to a two-wheel system. The car is treated as having a single front and rear wheel, which simplifies the equations of motion without sacrificing significant accuracy, particularly at low and moderate speeds.
- **Rear Axle Reference for Easier Calculation**:
  - **Rear-Axle Reference**: The rear axle is used as the reference point, simplifying calculations. The rear wheel is typically assumed to follow the trajectory path, while the front wheel determines the heading angle based on the steering angle.

– **Non-Slip Condition**: Referring to the rear axle facilitates assuming a non-slip condition for the wheels, meaning that the lateral velocity at the rear axle is often zero, which further simplifies the kinematic equations.

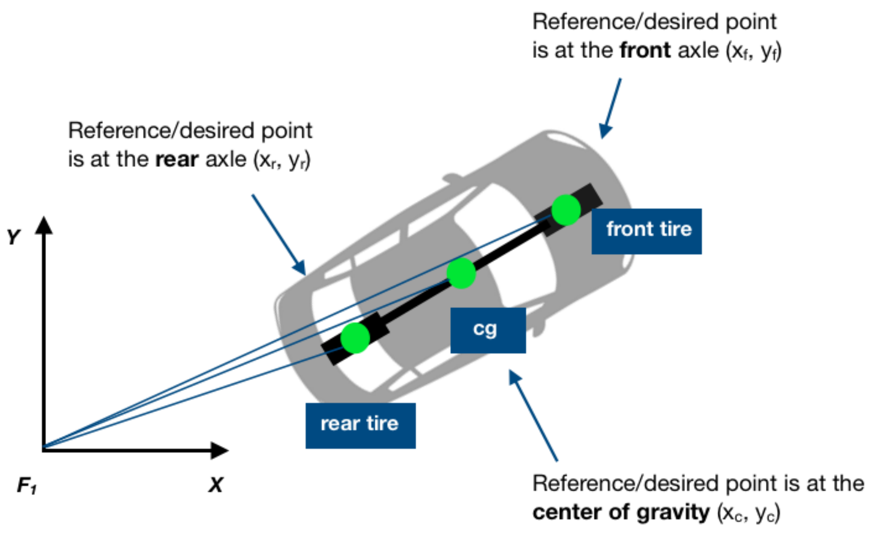
Reference/desired point is at the **front** axle (x_f, y_f)
Reference/desired point is at the **rear** axle (x_r, y_r)
Y
front tire
cg
rear tire
F1
X
Reference/desired point is at the **center of gravity** (x_c, y_c)

Figure 1: Kinematic Bicyle Model

4 Singleton Class Approach for the Kinematic Bicycle Model

4.1 Class Overview

The `KinematicBicycleModel` class is designed to model the kinematic behavior of a bicycle using the singleton pattern. This ensures that only one instance of the class exists across the application, providing a consistent interface for managing and simulating the bicycle's state. The class encapsulates the essential functionality to simulate the bicycle's kinematic dynamics, including state management and state transition calculations.

4.2 Class Description

```
class KinematicBicycleModel {
public:
    // Singleton instance accessor
    static KinematicBicycleModel& getInstance();
    // Setters for state and parameters
    void setState(double x, double y, double theta, double v);
    void setParameters(double wheelbase, double delta, double dt);

    // Getters for state and parameters
    std::vector<double> getState() const;
    double getWheelbase() const;
    double getDelta() const;
    double getDt() const;

    // Generate the next state based on current state and control inputs
    void generateNextState();

private:
    // Private constructor for singleton pattern
    KinematicBicycleModel();

    // Internal state vector: [x, y, theta, v]
    std::vector<double> state;
    double wheelbase;
    double delta;
    double dt;

    // Disallow copying and assignment
    KinematicBicycleModel(const KinematicBicycleModel&) = delete;
    KinematicBicycleModel& operator=(const KinematicBicycleModel&) = delete;
};
```

- **Singleton Instance Accessor**: `static KinematicBicycleModel& getInstance()` - Provides access to the single instance of the class, ensuring a unique and centralized management of the bicycle model.
- **Generate the Next State**:
  - `void generateNextState()` - Computes the next state of the bicycle model based on the current state , discretization time and control inputs. This method updates the internal state using the kinematic equations of motion, taking into account the bicycle's wheelbase, steering angle, and time step.

5 Optimizing the Cost Function and Using IPOPT Nonlinear Solver

5.1 Cost Function

In Model Predictive Control (MPC), the cost function quantifies the penalty associated with a particular solution. The objective is to minimize this cost function while adhering to system constraints.

6 Cost Function Equation

The cost function J used in Model Predictive Control (MPC) can be formulated as follows:

$$J = \sum_{k=0}^{N-1} \left[(x_k - x_{ref}(k))^T Q (x_k - x_{ref}(k)) + (u_k - u_{ref}(k))^T W (u_k - u_{ref}(k)) \right] + (x_N - x_{goal})^T P (x_N - x_{goal}) \tag{1}$$

6.1 Components

- x_k: State vector at time step k.
- u_k: Control vector at time step k.
- $x_{ref}(k)$: Desired state at time step k.
- $u_{ref}(k)$: Desired control input at time step k.
- x_N: State vector at the end of the prediction horizon N.
- x_{goal}: Desired final state.
- Q: State weighting matrix that penalizes deviations from the reference state.
- W: Control weighting matrix that penalizes deviations from the reference control input.
- P: Terminal cost matrix that penalizes deviations from the desired final state.

7 Concept of Cost Function Components

7.1 Tracking Error Cost

- **Objective**: To minimize the deviation of the vehicle's state from the desired trajectory and speed.

- **Components**:
  - **Cross-Tracking Error**: Penalizes the lateral position error relative to the desired path. This term is scaled by a factor (e.g., 3000) to emphasize its importance. A lower cross-tracking error indicates that the vehicle is closely following the desired trajectory.
  - **Heading Error**: Penalizes deviations in the vehicle's heading angle from the desired orientation. By squaring this error and scaling it (e.g., 500), the control system encourages the vehicle to maintain the correct heading.
  - **Velocity Error**: Penalizes the difference between the vehicle's actual velocity and the reference velocity. The goal is to ensure that the vehicle maintains a speed close to the desired reference speed.

7.2 Actuator Effort Cost

- **Objective**: To minimize the control effort required by the vehicle's actuators (steering and acceleration).
- **Components**:
  - **Steering Angle**: Penalizes large steering angles to avoid excessive steering effort. By squaring the steering angle and including it in the cost function, the system discourages abrupt or large steering maneuvers.
  - **Acceleration**: Penalizes large acceleration values. The goal is to reduce aggressive acceleration or braking, which helps in smoother vehicle operation and better energy efficiency.

7.3 Sequential Actuation Smoothness Cost

- **Objective**: To ensure smooth transitions in control inputs over time, which leads to more stable and less erratic vehicle behavior.
- **Components**:
  - **Steering Angle Change**: Penalizes abrupt changes in steering angles between consecutive time steps. By squaring the difference between steering angles at adjacent steps and scaling it, the system encourages smoother steering adjustments.
  - **Acceleration Change**: Penalizes sudden changes in acceleration between consecutive time steps. It promotes gradual adjustments to acceleration, leading to a smoother driving experience.

State variables x_k represent the system state at time k, while control variables u_k denote the control inputs.

7.4 Using IPOPT Nonlinear Solver

IPOPT solver is then configured by passing the cost function and our constraints,state variables,inputs,upper and lower bounds at all timesteps along with the MPC paramters. Then it outputs a result vector containing the optimized inputs to our next timestep.

7.5 Post-Optimization Process in Model Predictive Control

After implementing the objective function and setting up the nonlinear solver, such as IPOPT, the following steps are typically performed:

- **Optimization Execution:** The nonlinear solver, such as IPOPT, is executed to solve the optimization problem defined by the cost function and constraints. The solver iteratively adjusts the control inputs to minimize the cost function while satisfying all constraints.
- **Obtaining Results:** Once the solver converges, it provides the optimal control inputs and corresponding state trajectories for the entire prediction horizon N. These results are crucial for the next steps in the control process.
- **Extracting the First Control Input:** From the optimized sequence of control inputs, only the first control input (i.e., u_0) is implemented. This input is applied to the system for the current time step.
- **Updating the State:** The system state is updated based on the first control input and the system dynamics. This new state will serve as the starting point for the next optimization cycle.
- **Shifting the Horizon:** The optimization horizon is shifted forward by one time step. This involves discarding the first control input from the previous solution and extending the prediction horizon with a new time step.
- **Re-Optimization:** A new optimization problem is formulated using the updated state and the extended horizon. The solver is then run again to compute a new sequence of optimal control inputs.

7.5.1 Iterative Main Method

- **Feedback Loop:** This process is repeated at each control step in a feedback loop. The updated control input is applied, the state is recalculated, and the optimization problem is re-solved to determine the next optimal control inputs.

- **Adaptation to Changes:** The iterative nature of MPC allows the controller to adapt to changes in the system dynamics and disturbances, continually optimizing the control actions based on the latest state information.

This process ensures that the control inputs are optimized in real-time, leading to improved performance and stability of the system.

8 Using MPC Libraries Second Approach

In the second approach, a library of Model Predictive Control (MPC) was integrated into the system to handle various maneuvers. The approach involved formulating a cost function, setting up upper and lower bounds, Weights and solving the optimization problem for four distinct maneuvers. Each maneuver was designed to assess different aspects of vehicle control, with specific focus on lateral and longitudinal dynamics.

8.1 Maneuver A: `MPC_Solver_Lane_Changer_Constant_Speed`

`MPC_Solver_Lane_Changer_Constant_Speed` is designed to evaluate the lateral control of the vehicle during lane changes at a constant speed. The primary objectives of this maneuver are to:

- Assess the vehicle's ability to execute lane changes while maintaining a steady velocity.
- Evaluate the effectiveness of lateral control inputs in managing the lane change process.

The cost function for this maneuver emphasizes minimizing lateral deviations and ensuring smooth lane transitions while the speed remains constant. Upper and lower bounds are set to ensure feasible lane change maneuvers. Also , Weights on the cross track error was increased as well as the velocity error .

8.2 Maneuver B: `MPC_Solver_Lane_Change_Varying_Speed`

`MPC_Solver_Lane_Change_Varying_Speed` extends the evaluation to scenarios where the vehicle's speed varies during lane changes. This maneuver focuses on:

- Assessing both lateral and longitudinal control as the vehicle changes lanes while experiencing speed variations.
- Evaluating the ability of the control system to manage changing speeds and lane changes simultaneously.

The cost function includes terms for minimizing lateral and longitudinal deviations, with constraints on both speed and lane position. The bounds are adjusted to accommodate varying speeds during lane changes. Weights are balanced and empahsiszed on both velocity and lateral distance (cross track) error.

8.3 Maneuver C: `MPC_Solver_Trajectory_Follower`

`MPC_Solver_Trajectory_Follower` involves a polynomial generator method to create a desired trajectory for the vehicle. Specifically:

- A polynomial generator produces desired coordinates (x, y, r) based on a cubic polynomial x^3 trajectory.
- The maneuver assesses the vehicle's capability to follow a predefined polynomial trajectory.

The cost function for this maneuver focuses on minimizing deviations from the generated trajectory. Bounds are set to ensure that the vehicle adheres to the polynomial path accurately. The coordinates x,y desired were subtracted from the current coordinates and squared then added tot he cost function and the their weights were high relatively.

8.4 Maneuver D: `MPC_Solver_Trajectory_and_Speed_Control`

`MPC_Solver_Trajectory_and_Speed_Control` combines trajectory following with speed control to maintain a constant speed while adhering to a desired trajectory. This maneuver aims to:

- Maintain a constant speed as the vehicle follows a trajectory.
- Integrate both trajectory following and speed control into a unified control strategy.

The cost function incorporates terms for both trajectory following and maintaining speed, with appropriate bounds , Weights to ensure consistency and adherence to the desired path and speed.

9 Resutls

Each of these maneuvers uses a tailored cost function and set of constraints to address specific aspects of vehicle control, ensuring comprehensive evaluation of the MPC library's capabilities. The integration of the MPC library and the implementation of various maneuvers is shown in the figures below :

10 MPC Maneuvers and Figures

The following maneuvers were evaluated using the MPC library, each with specific objectives and associated figures to illustrate the performance:

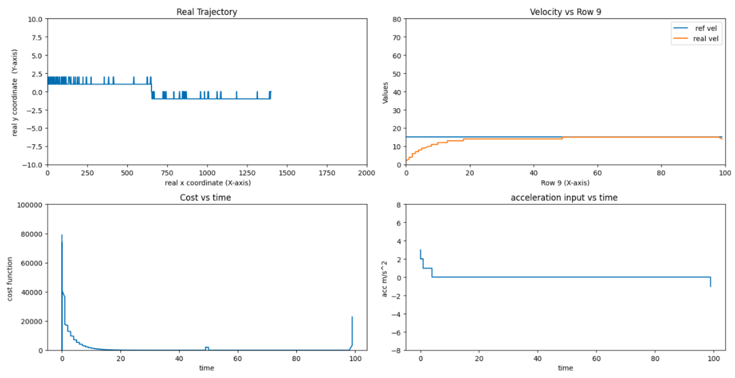

Figure 2: MANEUVER A

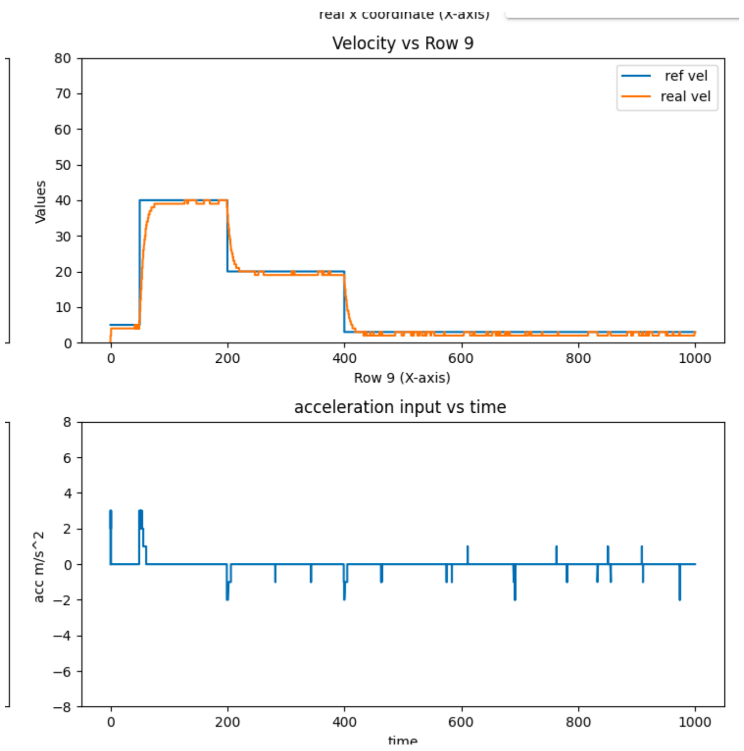

Figure 3: MANEUVER B

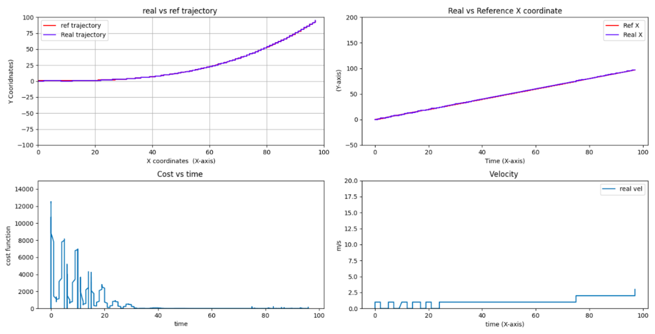

Figure 4: MANEUVER C

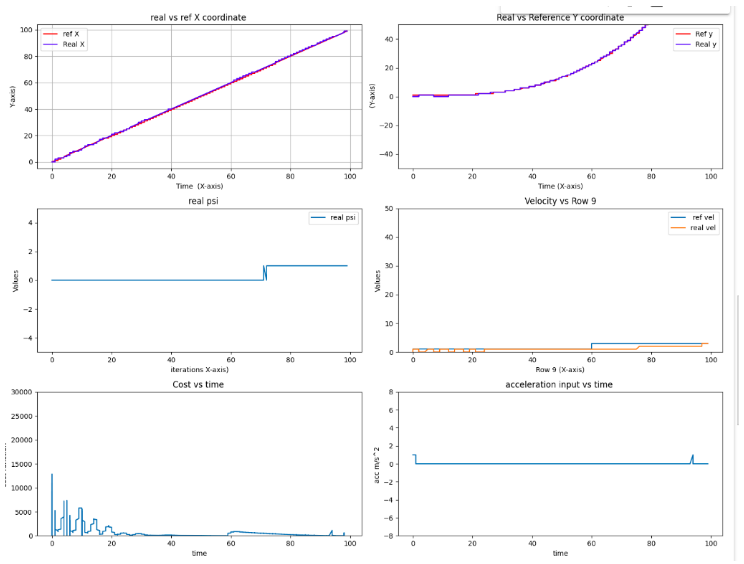

Figure 5: MANEUVER D