

Model Predictive Control: Mini-Project

In this project, you will develop an MPC cruise controller for a car on a highway.

The project is worth 40% of your final grade and is due on Friday, January 10th, 2025.

Report and handing-in instructions

- Group sign up and report hand-in is via Moodle.
- You can do the project in groups of one, two or three.
- Include everyone's name and SCIPER on the title page of your project report.
- When you have completed the project, hand in one report (pdf) per group and your Matlab code (zip).
- Report:
 - Your report should contain headings according to the **Deliverables** listed below in the project description.
 - **You will be graded on the Deliverables**, and not on the Todos.
 - The report should be written in **English**.
 - Explain what you're doing and why for each deliverable, but don't be excessive. The entire report **must not exceed** more than **20** pages.
- Code:
 - Include a directory for each deliverable containing all the m-files to run the deliverable.
 - Create a file in each directory `Deliverable_xxx.m` which can be run to produce all the the figures for the deliverable.
 - Compress all the code / directories into a single zip file for submission.

Before you start

- Make sure you have installed YALMIP, MPT3, Gurobi and CasADi according to the course exercise setup instructions on Moodle.
- Download and unpack the file `car_project.zip` from Moodle.
- Run `car = Car(1/10);` If this executes correctly, then your setup should be ready to go.

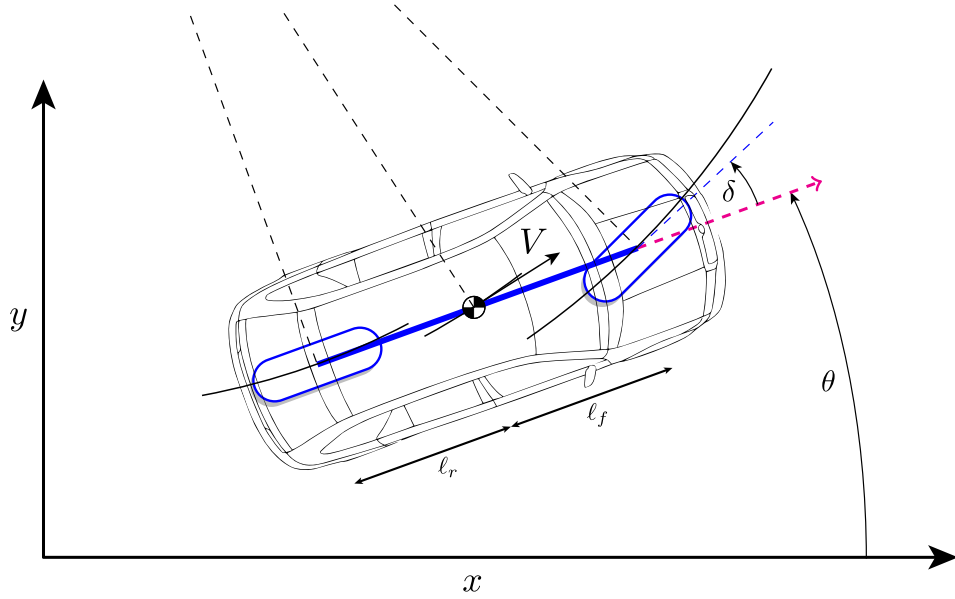


Figure 1: Car Model

Part 1 | System Dynamics

Building a model of the system dynamics from physical principles is a crucial step in the development of an MPC controller and is a significant part of the task in practice. However, as this process is out of the scope of this course, you are not required to model the car by yourself. Instead, we provide you with a nonlinear model.

System Definition We consider a four-state kinematic model of a car moving in a 2D plane. The state vector is defined as

$$\mathbf{x} = [x \ y \ \theta \ V]^T, \quad [\mathbf{x}] = [\text{m} \ \text{m} \ \text{rad} \ \text{m/s}]^T$$

where (x, y) represents the position of the car's center of mass in the world frame, θ is the heading angle of the car with respect to the x -axis, and V is the velocity of the vehicle.

The input vector of the model is

$$\mathbf{u} = [\delta \ u_T]^T, \quad [\mathbf{u}] = [\text{rad} \ -]^T$$

where δ is the steering angle of the front wheels, and u_T is the normalized throttle to the motor limited to $-1 \leq u_T \leq 1$.

Vehicle Kinematics and Dynamics We capture the motion of the car with what is called a bicycle model. The equations derived below are with reference to the constants shown in Figure 1.

The kinematic slip angle β is the angle between a wheel's actual direction of travel and the direction in which it is pointed. We define the slip angle β at the center of mass and compute it based on the steering geometry:

$$\beta = \arctan\left(\frac{\ell_r \tan(\delta)}{\ell_r + \ell_f}\right) \quad (1)$$

where ℓ_r and ℓ_f are the distances from the center of mass to the rear and front axles, respectively.

The longitudinal dynamics are influenced by the motor force, aerodynamic drag, and roll resistance:

$$\begin{aligned} F_{\text{motor}} &= \frac{u_T P_{\text{max}}}{V} \\ F_{\text{drag}} &= \frac{1}{2} \rho C_d A_f V^2 \\ F_{\text{roll}} &= C_r m g \end{aligned}$$

where m is the vehicle mass, P_{max} is the maximum motor power, ρ is the air density, C_d is the drag coefficient, A_f is the frontal area of the vehicle, C_r is the roll coefficient, and g is the gravity constant. Note that this model is only valid for $V > 0$, which we assume to hold on the highway.

In particular, F_{motor} models the motor dynamics of an electric car due to the instant torque at low velocities. Similarly, a negative input can be seen as recuperation-braking, i.e., the motor acts as a generator and thus slows down the car.

The complete state equations are:

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{V} \end{bmatrix} = \begin{bmatrix} V \cos(\theta + \beta) \\ V \sin(\theta + \beta) \\ \frac{V}{\ell_r} \sin(\beta) \\ \frac{F_{\text{motor}} - F_{\text{drag}} - F_{\text{roll}}}{m} \end{bmatrix} \quad (2)$$

These equations have been implemented in the function `f` in the Matlab class `Car.m`, which is in the directory `src`.

Todo 1.1 | Study the function `f` in the Matlab `Car` class to confirm that it implements the dynamics of the system as described above.

To evaluate the functions, you can call them independently:

```
Ts = 1/10;
car = Car(Ts);
u = [delta, u.T]'; % (Assign appropriately)
x = [x, y, theta, V]'; % (Assign appropriately)
x_dot = car.f(x, u)
```

Todo 1.2 | Simulate the car with various step inputs to confirm that the dynamics responds as expected.

To simulate the nonlinear model for two seconds starting from \mathbf{x}_0 with constant input, you can use:

```
car = Car(Ts);  
Tf = 2.0; % Simulation end time  
  
x0 = [0, 0, deg2rad(-2), 20/3.6]'; % (x, y, theta, V) Initial state  
u = [deg2rad(1), 0.7]'; % (delta, u_T) Constant input  
  
params = {}; % Setup simulation parameter struct  
params.Tf = Tf;  
params.myCar.model = car;  
params.myCar.x0 = x0;  
params.myCar.u = u;  
result = simulate(params); % Simulate nonlinear model  
visualization(car, result);
```

You can inspect the result of the simulation in the `result` struct. You will notice it has the following structure:

```
result.T % Time at every simulation step  
result.myCar.X % State trajectory  
result.myCar.U % Input trajectory
```

A few things to try to see if the car is behaving as you think it should. Find input \mathbf{u} that will cause the car to:

- Accelerate/decelerate.
- Turn left or right.

Part 2 | Linearization

In the first part of the project, we are going to control a linearized version of the car. In particular, we are linearizing in the x-direction of travel, i.e., the longitudinal motion.

Normally, one would linearize the system around a steady state. But in our case, we are interested in cruise control for a target velocity reference for which no true steady state exists (the x-position is integrated in time with a positive velocity). Hence, we are only considering the subsystem steady-state where

$$f_s(\mathbf{x}_s, \mathbf{u}_s) = \begin{bmatrix} \dot{y} \\ \dot{\theta} \\ \dot{V} \end{bmatrix} = 0, \quad (3)$$

which will result in $f(\mathbf{x}_s, \mathbf{u}_s) \neq 0$. We can then find the linearized dynamics through a Taylor series expansion up to first order

$$f(\mathbf{x}, \mathbf{u}) \approx f(\mathbf{x}_s, \mathbf{u}_s) + A(\mathbf{x} - \mathbf{x}_s) + B(\mathbf{u} - \mathbf{u}_s), \quad (4)$$

where $A = \frac{\partial f(\mathbf{x}, \mathbf{u})}{\partial \mathbf{x}}|_{(\mathbf{x}_s, \mathbf{u}_s)}$ and $B = \frac{\partial f(\mathbf{x}, \mathbf{u})}{\partial \mathbf{u}}|_{(\mathbf{x}_s, \mathbf{u}_s)}$.

Deliverable 2.1 | Derive the analytical expressions of $f(\mathbf{x}_s, \mathbf{u}_s)$, A , and B as a function of \mathbf{x}_s and \mathbf{u}_s , where $\mathbf{x}_s = (0, 0, 0, V_s)$ and $\mathbf{u}_s = (0, u_{T,s})$.

Todo 2.1 | In practice, we calculate the linearization numerically. Use the following code to generate a linearized version of the car and compare it against your analytical derivation. Note that all of the physical constants of the car are stored in the struct `car`.

```
car = Car(Ts);

Vs = 120/3.6; % 120 km/h
[xs, us] = car.steady_state(Vs); % Compute steady-state for which f_s(xs,us) = 0
sys = car.linearize(xs, us); % Linearize the nonlinear model around xs, us
```

Go through the functions `steady_state` and `linearize` to see how they work.

Note that we have named all the states in the linearized model. Type `sys` and you will see the ordering of the states and $f(\mathbf{x}_s, \mathbf{u}_s)$, \mathbf{x}_s , and \mathbf{u}_s are stored in `sys.UserData`.

Study the resulting **A**, **B**, **C** and **D** matrices until you recognize that the linearized system around the quasi-steady-state can be broken into two independent/non-interacting systems.

Deliverable 2.2 | Explain from an intuitive physical / mechanical perspective, why this separation into independent subsystems is possible.

Todo 2.2 | Compute the two independent systems above using the following command

```
[sys_lon, sys_lat] = car.decompose(sys);
```

Two models are produced:

`sys_lon` | Longitudinal dynamics, i.e., throttle u_T to position x . The system has two states: x, V .

`sys_lat` | Lateral dynamics, i.e., steering angle δ to position y . The system has two states: y, θ .

Note that these are all **continuous-time** models.

Discretization

In the following parts, you will implement discrete MPC controllers, i.e., the continuous-time models have to be discretized. **We will use a sampling period of $T_s = 1/10$ seconds.** Discretizing the continuous system (4) results in the following discrete-time dynamics:

$$\mathbf{x}^+ = f_d(\mathbf{x}_s, \mathbf{u}_s) + A_d(\mathbf{x} - \mathbf{x}_s) + B_d(\mathbf{u} - \mathbf{u}_s). \quad (5)$$

We have implemented the exact discretization for you. You can use the following code to obtain the matrices of the discretized dynamics:

```
Ts = 1/10;
[fd_xs_us, Ad, Bd, Cd, Dd] = Car.c2d_with_offset(sys, Ts);
```

Delta Dynamics

Let's consider the reduced system

$$\bar{\mathbf{x}} = \begin{bmatrix} y \\ \theta \\ V \end{bmatrix},$$

for which $\bar{\mathbf{x}}_s^+ = \bar{\mathbf{x}}_s$. Then (5) simplifies to

$$\bar{\mathbf{x}}^+ = \bar{\mathbf{x}}_s + \bar{A}_d(\bar{\mathbf{x}} - \bar{\mathbf{x}}_s) + \bar{B}_d(\mathbf{u} - \mathbf{u}_s),$$

where \bar{A}_d is the matrix of the sub-system. From this, we can then derive the delta formulation with delta state $\Delta\bar{\mathbf{x}} = \bar{\mathbf{x}} - \bar{\mathbf{x}}_s$ and delta input $\Delta\mathbf{u} = \mathbf{u} - \mathbf{u}_s$ giving us the standard linear dynamics

$$\Delta\bar{\mathbf{x}}^+ = \bar{\mathbf{x}}_s + \bar{A}_d(\bar{\mathbf{x}} - \bar{\mathbf{x}}_s) + \bar{B}_d(\mathbf{u} - \mathbf{u}_s) - \bar{\mathbf{x}}_s \quad (6)$$

$$= \bar{A}_d\Delta\bar{\mathbf{x}} + \bar{B}_d\Delta\mathbf{u} \quad (7)$$

you have seen in the lecture.

Note that throughout the project, you can use either (6) or (7). You just have to be sure to manage the steady-state offsets correctly.

Part 3 | Design MPC Controllers for Each Sub-System

For each sub-system, your goal is to design a recursively feasible, stabilizing MPC controller that can track step references.

The discretization of the continuous time models is done for you when using the provided `MpcControl.*` template files, i.e., you can find the discretization routine in the constructor of `MpcControlBase`.

Constraints

The car drives at highway speeds and should always stay inside the lanes. Also, to increase the comfort of the passenger, we limit the maximum heading of the car during a lane change. Apart from comfort, this also bounds the linearization error nicely.

$$\begin{aligned} -0.5 \text{ m} &\leq y \leq 3.5 \text{ m} \\ |\theta| &\leq 5^\circ = 0.0873 \text{ rad} \end{aligned}$$

The inputs are constrained by mechanical limitations:

$$\begin{aligned} -1 &\leq u_T \leq 1 \\ |\delta| &\leq 30^\circ = 0.5236 \text{ rad} \end{aligned}$$

Design Tracking MPC Controllers

Todo 3.1 | Design two MPC controllers for each sub-system, with the following properties:

- Recursive satisfaction of the input, heading, and lane constraints.
- Settling time no more than 10 seconds when accelerating from 80 km/h to 120 km/h or 3 seconds doing a lane change (width of track is 3 m).

To help you design the controllers, we have created two files:

- `MpcControl_lon.m`
- `MpcControl_lat.m`

which you will find in the `templates` directory. Copy these into the `Deliverable_3.1` directory and then make your changes. Your job is to fill in the functions `setup_controller` and `compute_steady_state_target` in each file.

Hint | Note that you have no cost or constraints on the x-position (although there is on the x-velocity). Hence, there is no need to optimize over the x-position (which is also not a steady state). Thus, only consider the steady-state subsystem $\bar{\mathbf{x}}$ in your controllers. You might want to take sub-matrices of A_d and B_d as described in the section above on Delta Dynamics, and extract the relevant part of the initial state $\mathbf{x}_0 = [\text{position}, \text{velocity}]$, etc

You can then get the control from solving the MPC problem via the following code:

```

Ts      = 1/10; % Sample time
car     = Car(Ts);
[xs, us] = car.steady_state(120 / 3.6);
sys     = car.linearize(xs, us);
[sys_lon, sys_lat] = car.decompose(sys);

% Design MPC controller
H_lon = .; % Horizon length in seconds
mpc_lon = MpcControl1lon(sys_lon, Ts, H_lon);

% Get control input for longitudinal subsystem
u_lon = mpc_lon.get_u(x_lon, ref_lon);

```

Before applying MPC in closed-loop, you should always check first if the optimal open-loop trajectory from a representative state is reasonable. This helps to understand whether the underlying optimal control problem is correctly formulated or, in case of unintended results, how it should be adjusted. For this you can specify debug variables in your optimization problem:

```

%%% In MpcControl1lon.m, setup_controller:
X = sdpvar(1, N);
U = sdpvar(1, N);
...
debugVars = {X, U}; % arbitrary number of variables can be passed
%%%

%%% In your test script:
[u_lon, X_lon, U_lon] = mpc_lon.get_u(x_lon, ref_lon);
% Plot and debug X_lon and U_lon accordingly
%%%

```

You can simulate your system with the following command:

```

mpc_lon = MpcControl1lon(sys_lon, Ts, H_lon);
mpc_lat = MpcControl1lat(sys_lat, Ts, H_lon);
mpc = car.merge_lin_controllers(mpc_lon, mpc_lat);

x0 = [0 0 0 80/3.6]'; % (x, y, theta, V)
ref1 = [0 80/3.6]'; % (y_ref, V_ref)
ref2 = [3 120/3.6]'; % (y_ref, V_ref)

params = {};
params.Tf = 15;
params.myCar.model = car;
params.myCar.x0 = x0;
params.myCar.u = @mpc.get_u;
params.myCar.ref = car.ref_step(ref1, ref2, 5); % delay reference step by 5s
result = simulate(params);
visualization(car, result);

```


Deliverable 3.1 |

- Explanation of design procedure that ensures recursive constraint satisfaction.
- Explanation of choice of tuning parameters. (e.g., \mathbf{Q} , \mathbf{R} , H).
- Plot the terminal invariant set for the lateral sub-system and explain how it was designed and tuned.
- Closed-loop plots for each dimension of the system starting at the origin (for x and y) with $V = 80$ km/h to a reference of $y = 3$ m and target velocity of $V = 120$ km/h. Note, that there should be no steady-state error.
- Matlab code for the two controllers, and code to produce the plots in the previous step.

Part 4 | Offset-Free Tracking

As you might already have noticed, tracking any velocity which is not the steady-state velocity we linearized around leads to a steady-state tracking error. This is due to the linearization error introduced by the motor and drag term in the system dynamics.

We assume the linearization error enters the dynamics of the system via the input u_T according to

$$\mathbf{x}^+ = f_d(\mathbf{x}_s, \mathbf{u}_s) + A_d(\mathbf{x} - \mathbf{x}_s) + B_d(\mathbf{u} - \mathbf{u}_s) + \hat{B}_d d,$$

where d is a constant, unknown disturbance and \hat{B}_d the submatrix of B_d for the input u_T . Your goal is to update your controller to reject this disturbance and track setpoint references with no offset.

Todo 4.1 | Design an offset-free tracking controller.

- Complete the relevant functions in the `LonEstimator.m` file. Note that you are only estimating a small subset of the state, i.e., you are estimating the velocity V and the disturbance d for the next time step based on the current estimate and the longitudinal subsystem state measurements.
- Update the functions `setup_controller` and `compute_steady_state_target` in `MpcControl_lon.m` to provide offset-free tracking.

Once you have updated the relevant functions, you can test it in simulation by adding `est_fcn` and `est_dist0` to `params.myCar`. Here, `est_dist0` is the initial state of the disturbance estimate, i.e., d at time $t = 0$. The controller will now act based on the state estimates from the observer. You can obtain the estimates of the z states (concatenation of the current estimation of V and d) from the corresponding rows of the `result.myCar.Z_hat` output in which the last row is the disturbance estimate d .

```
estimator = LonEstimator(sys_lon, Ts);

x0 = [0 0 0 80/3.6]'; % (x, y, theta, V)
ref1 = [0 80/3.6]'; % (y_ref, V_ref)
ref2 = [3 50/3.6]'; % (y_ref, V_ref)

params = {};
params.Tf = 15;
params.myCar.model = car;
params.myCar.x0 = x0;
params.myCar.est_fcn = @estimator.estimate;
params.myCar.est_dist0 = 0;
params.myCar.u = @mpc.get_u;
params.myCar.ref = car.ref_step(ref1, ref2, 2); % delay reference step by 2s;
result = simulate(params);
```

Deliverable 4.1 |

- Explanation of your design procedure and choice of tuning parameters.
- Plot showing the impact of the state estimator, resulting in offset-free velocity tracking. Simulate from $\mathbf{x}_0 = [0 \ 0 \ 0 \ 80/3.6]'$ with $\mathbf{ref} = [3 \ 50/3.6]'$ for 15 seconds, where the reference step is delayed by 2 seconds.
- Matlab code for your controllers, and code to produce the plots in the previous step.

Part 5 | Robust Tube MPC for Adaptive Cruise Control

In this section, you will implement a robust tube MPC controller for adaptive cruise control, i.e., you want to track your reference velocity while keeping a minimum distance to the car in front.

We will call the car that you're controlling the 'ego' car, and the one in front of it the 'lead' car. We assume that the lead car has the same dynamics as ours and that the linearized and discretized equations of motion are the same

$$\tilde{\mathbf{x}}^+ = f_d(\mathbf{x}_s, \mathbf{u}_s) + A_d(\tilde{\mathbf{x}} - \mathbf{x}_s) + B_d(\tilde{\mathbf{u}} - \mathbf{u}_s). \quad (8)$$

where the lead car state and input are denoted with a tilde.

We want to control the relative position and speed between the two cars by controlling the ego car. Define the relative longitudinal state between the two vehicles as $\mathbf{\Delta} = [\Delta_x \ \Delta_v] = \tilde{\mathbf{x}}_{long} - \mathbf{x}_{long} - \mathbf{x}_{safe}$, where $\mathbf{x}_{safe} = [x_{safe,pos} \ 0]$ is a steady-state safe point behind the lead car, which you will choose.

We see that the dynamics of $\mathbf{\Delta}$ evolve according to the equation

$$\mathbf{\Delta}^+ = A_d \mathbf{\Delta} - B_d u_T + B_d \tilde{u}_T \quad (9)$$

Todo 5.1 | Derive these relative dynamics by subtracting (8) from (5).

We don't know how the lead car will behave on the road and so we model their behaviour as an uncertain, but bounded, disturbance input. In particular, we treat the throttle of the lead car, \tilde{u}_T , as a disturbance and assume that it is bounded to lie in the set

$$\tilde{u}_T \in \mathbb{W} := [u_{T,s} - 0.5, u_{T,s} + 0.5]$$

where $u_{T,s}$ is the linearization steady-state throttle input.

Your goal is to design a tube-MPC controller for the longitudinal dimension of the system to ensure that the ego car doesn't run into the lead car, despite the lead car changing its throttle in an unknown fashion within the specified amount. The requirement is that a minimum gap of 6 m is maintained at all times:

$$\tilde{x} - x \geq 6$$

Note that because the car has a length of 4.3 m, this is equivalent to ensuring that the distance between the bumpers of the cars is at least 1.7 m.

Note that throughout this section, you only need to work with the longitudinal dynamics, and can ignore the lateral dynamics.

Todo 5.2 | Design a robust tube MPC controller following the outlined procedure below.

Hint | Calculate all components of your controller in a separate file `tube_mpc_sets.m` and save them in a `.mat` file to save time. You can save/load matrices and `Polyhedron` sets in `.mat` files using the following code:

```
save('tube_mpc_data.mat', 'var1', 'var2'); % save data
load('tube_mpc_data.mat', 'var1', 'var2'); % load data
```

- Hint** |
- The `c2d` function will convert a continuous-time LTI system to a discrete time one
 - The `ssdata` will extract the A, B, C, D matrices from an LTI model

Minimum Robust Invariant Set

Define a tracking controller K to keep the real trajectory close to the tube centers as introduced in the lectures and calculate the minimal invariant set \mathcal{E} for the error system

- Hint** |
- Notice the sign on the input matrix B_d in (9)
 - When computing your invariant sets, you will iterate through F_1, F_2, \dots as we've seen in the notes. In each iteration, the complexity of these sets will grow and so it's a good idea to call `F.minHRep` or `F.minVRep` regularly to reduce the size of these sets. Alternatively, if you plot the set in each iteration, this will be done for you.
 - Using the algorithm from the lectures leads to an infinite number of iterations and therefore you have to terminate it after a fixed number. Terminate when the size of the set you're adding is sufficiently small; `norm((A+B*K)^i) < 1e-2` is a good termination condition.

Tightened Constraints

Choose your \mathbf{x}_{safe} and specify the resulting \mathbb{X} . You want to follow the lead car fairly closely, so don't choose \mathbf{x}_{safe} to be very large.

Calculate the tightened constraints

$$\tilde{\mathbb{X}} := \mathbb{X} \ominus \mathcal{E}, \quad \tilde{\mathbb{U}} := \mathbb{U} \ominus K\mathcal{E}.$$

- Hint** | Check the control authority available to control your tube centers in $\tilde{\mathbb{U}}$. If it is not reasonable, consider changing the shape of \mathcal{E} by choosing a different tracking controller.

Terminal Components

Design a terminal controller, terminal constraint and terminal weight.

- Hint** | If your tightened state constraints do not contain the origin (`Xt.contains([0;0]) == false`), then the terminal set will be empty. Consider changing \mathbf{x}_{safe} or re-tuning your tracking controller to change the shape of \mathcal{E} .

Controller Design

Design a tube-MPC controller to ensure that the ego car remains robustly safe behind the lead car despite the lead car changing its throttle by up to ± 0.5 .

Your tuning goal is to have a good ride quality. i.e., the ego car should experience as low accelerations as possible despite sharp motions of the lead car.

Hint | The YALMIP variable `x0other` in the file `MpcControl_1lon.m` contains the initial state of the lead car.

Hint | Gurobi will be the fastest solver, but if it is returning “Numerical problems”, the you may consider switching to the solver `quadprog` in the file `MpcControl_1lon.m`.

Simulation

To simulate the lead car at a constant velocity starting 10 meters in front of the ego car, you can use the following code:

```
otherRef = 100 / 3.6;

params = {};
params.Tf = 25;
params.myCar.model = car;
params.myCar.x0 = [0 0 0 80/3.6]';
params.myCar.u = @mpc.get_u;
params.myCar.ref = ref;
params.otherCar.model = car;
params.otherCar.x0 = [15 0 0 otherRef]';
params.otherCar.u = car.u_const(otherRef);
result = simulate(params);
visualization(car, result);
```

We also provide a test case which will test the full range of uncertainty for your controller:

```
params = {};
params.Tf = 25;
params.myCar.model = car;
params.myCar.x0 = [0 0 0 115/3.6]';
params.myCar.u = @mpc.get_u;
params.myCar.ref = ref;
params.otherCar.model = car;
params.otherCar.x0 = [8 0 0 120/3.6]';
params.otherCar.u = car.u_fwd_ref();
params.otherCar.ref = car.ref_robust();
```

Deliverable 5.1 |

- Explanation of your design procedure and choice of tuning parameters.
- Plots of the minimal invariant set \mathcal{E} and terminal set \mathcal{X}_f .
- Plot showing your robust controller working when starting from `myCar.x0 = [0 0 0 100/3.6]'` with `myCar.ref = [0 120/3.6]'` and the other car starting from `otherCar.x0 = [15 0 0 100/3.6]'` with `otherCar.u = car.u_const(100/3.6)`.
- Plot showing your robust controller working when starting from `myCar.x0 = [0 0 0 115/3.6]'` with `myCar.ref = [0 120/3.6]'` and the other car starting from `otherCar.x0 = [8 0 0 120/3.6]'` with `otherCar.u = car.u_fwd_ref()` and `otherCar.ref = car.ref_robust()`.
- Matlab code for your controllers, and code to produce the plots in the previous step.

Part 6 | Nonlinear MPC

In this section, you will develop a nonlinear MPC controller for the car using CasADi. The NMPC controller operates on the nonlinear model in (2) and can therefore cover the whole state space (i.e., the car is no longer decomposed into sub-systems).

Todo 6.1 | Design Tracking Nonlinear MPC Controller

Design an NMPC controller to track the given reference (lateral position and velocity). Complete the relevant functions in the `NmpcControl.m` file which you will find in the templates directory. Your NMPC controller should be able to satisfy the following requirements:

- Satisfy all the constraints given in previous parts of the project. Note that you do not need to implement a terminal invariant set for this part.
- Settling time no more than 5 seconds when accelerating from 80km/h to 100km/h and performing a lane change (lane width is 3m).

Similar to the linear MPC case, you should always check first if the predicted open-loop trajectory from a representative state is reasonable before applying MPC in closed-loop.

```
H = ...;
mpc = NmpcControl(car, H);

x0 = [0 0 0 80/3.6]';
ref = [3 100/3.6]';
u = mpc.get_u(x0, ref); % check if the open-loop prediction is reasonable
```

Hint | To debug you can view/plot any of your optimization variables via `mpc.sol.value(mpc.X)`, if you have assigned the `x` variable in your `NmpcControl` function (see the template code for how to do this).

You can simulate and plot the result of your controller in closed-loop with the following code:

```
params = {};
params.Tf = 15;
params.myCar.model = car;
params.myCar.x0 = [0 0 0 80/3.6]';
params.myCar.u = @mpc.get_u;
ref1 = [0 80/3.6]';
ref2 = [3 100/3.6]';
params.myCar.ref = car.ref_step(ref1, ref2, 2);

result = simulate(params);
visualization(car, result);
```

Hint | You can evaluate the continuous dynamics of the car for the state `x` and `u` via the call `car.f(x,u)`. Note that if you want to pass a function `car.f` to another function `bob` in Matlab, the notation is `bob(@car.f)`

- Deliverable 6.1** |
- Explanation of your design procedure and choice of tuning parameters.
 - Describe whether or not your controller results in steady-state tracking error. Explain why.
 - Plots showing the performance of your controller in the above-mentioned settings.
 - Matlab code for your controllers, and code to produce the plots in the previous step.

Todo 6.2 | Design NMPC Controller for Overtaking

Now we consider the situation where we would like to overtake another car. The ego car will overtake another car ahead by changing lanes, passing the other car, and returning to its original lane, as illustrated by Figure 2.

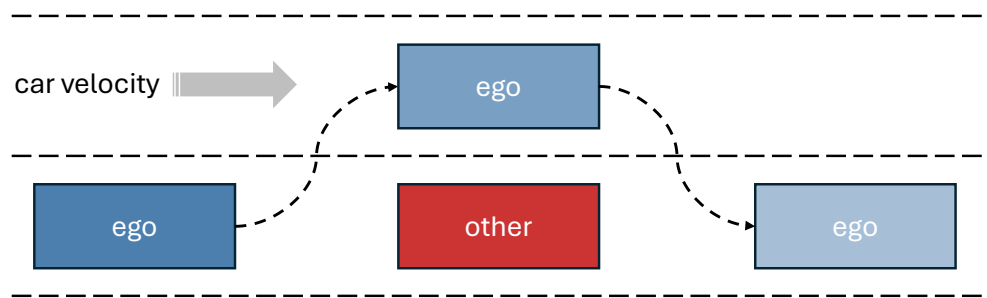


Figure 2: Illustration of the overtaking maneuver

Design a nonlinear MPC controller for the overtake maneuver. The initial velocities for both cars are 80km/h, while the other car is running ahead of the ego car in the same lane. The ego car will accelerate to 100km/h and finish the overtaking maneuver.

The only reference change that the ego car will see is that it wants to go faster than the car in front of it. You will have to implement constraints to ensure that it moves to the other lane, rather than collide into the back of the car ahead of it.

There are many methods to implement collision avoidance constraints. Here we will use a simple ellipsoidal constraint as shown in Figure 3.

Design a matrix H such that if $p = [lon \ lat]^T$ is the longitudinal and lateral position of the ego car, and p_L of the other car, then if the ellipsoidal constraint $(p - p_L)^T H (p - p_L) \geq 1$ is met, we have that the cars cannot collide. Note that the car is 4.3m long and 1.8m wide.

Hint | You will want to provide some extra margin around the cars so that they're not in danger of coming too close, but not so much that they can't pass each other on the road. A diagonal matrix H is likely a good choice.

Hint | You will likely want to forecast the location of the other car within your NMPC controller. This can be done by recalling that it is traveling at a fixed speed, which can be read out of the initial state `obj.x0other`.

Todo 6.3 | Simulate the system below, which has the ego car starting 20m behind the other car and traveling at the same speed. After one second, it accelerates to 100km/h and overtakes the other car.

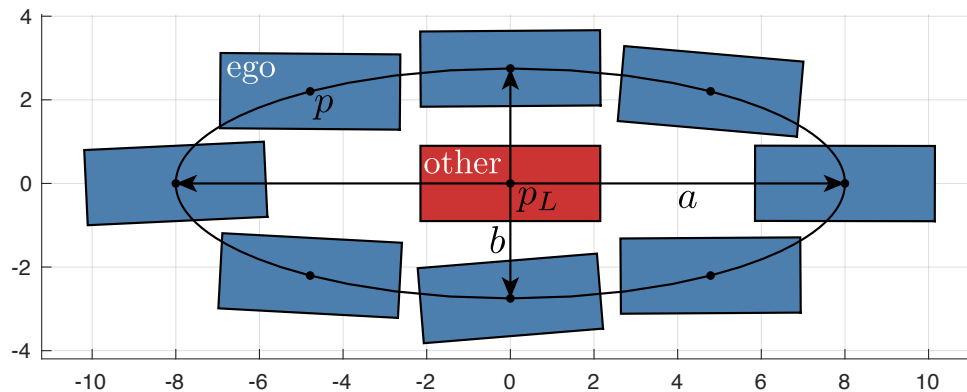


Figure 3: Ellipsoidal collision avoidance constraint

Tune your controller so that this is a car that you would want to sit in! Smooth and comfortable.

```
H = ...;
mpc = NmpcControl_overtake(car, H);

x0_ego = [0 0 0 80/3.6]';
x0_other = [20 0 0 80/3.6]';
ref1 = [0 80/3.6]';
ref2 = [0 100/3.6]';

params = {};
params.Tf = 15;
params.myCar.model = car;
params.myCar.x0 = x0_ego;
params.myCar.u = @mpc.get_u;
params.myCar.ref = car.ref_step(ref1, ref2, 1);

params.otherCar.model = car;
params.otherCar.x0 = x0_other;
params.otherCar.u = car.u_const(80/3.6);

result = simulate(params);
visualization(car, result);
```

Deliverable 6.2 |

- Explanation of your design procedure and choice of tuning parameters.
- Plots showing the performance of your controller in the above-mentioned settings.
- Matlab code for your controllers, and code to produce the plots in the previous step.