ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automatic Control Laboratory

Model Predictive Contouring Control

Semester thesis

Florian Perrodin

20. June 2011

Abstract

Following a track automatically as fast as possible is an open challenge for real cars. The framework derived in this report aims to adress this problem using model predictive contouring control (MPCC). Instead of the classical scheme where a 4D trajectory (path, velocity and steering angle) is supposed to be given, MPCC requires only to give a 2D trajectory: the path that has to be tracked. A virtual state is added to the real state which represents the position of the car on the track. This virtual state enables to control the deviation to the path that has to be tracked. Further developments include offline path planning and online path planning. The framework developped here is to a certain extent independant of the model used.

Introduction

To control automatically a car is a current challenge that could change our lives: how much time do people lose at driving? Additionally, according to World Health Organization (WHO) car accident is the 10th cause of mortality in the world, and the first one for the 15-29 years old. Numerous projects are currently under heavy development and big actors are massively investing in it. For example, Google presented at Technology Entertainment and Design conference 2011 (TED 2011) a car automatically controlled shown below (Fig. 1).

The three main challenges in this topic are

- to get the right data from sensors;
- to use this data in an optimal way to get estimates of the car position and the road (and possibly obstacles);
- and to drive carefully the car using these informations.

For the first challenge, a lot of effort has been done by Google to reduce the number of sensors: in this respect, comparing this car to the one used by MIT in the DARPA challenge is startling although they do not have the same goal.

In this report, we will put our effort on the third challenge: we assume to have a good estimate (although with some uncertainties) of the position of the car, and that we know the track.

Classical approaches involving PID controllers have been used [2] in order to track the trajectory: the velocity is controlled with respect to a precomputed velocity profile and the steering angle is set using some empirical assumptions. Although decent performances have been demonstrated, they require an important tuning effort. Other methods have been developed using model predictive control [5], trying to maximize the speed but this has some drawbacks as it will be shown in Part 1. Model contouring control enables to compute and control easily the error as well as to set up additionnal constraints to ensure to stay on a given track.

In chapter 1, we will describe the technique proposed by [3]. We will describe the underlying assumptions of the algorithm and derive the equations for our case study.

In chapter 2, some simple simulations are done to assess algorithm performance and assumptions made in part 1. We introduce two simple models for the car and apply it to the algorithm.

In chapter 3, we extend what has been done in part 1. From path tracking, we adapt the algorithm to path planning. Off-line and on-line path planning are demonstrated.

Figure 1: Google self-driven car at TED 2011.

Figure 2: MIT car for the DARPA challenge. A lot of rotating lasers are used to get estimation of the terrain. Although the goal is not the same as for Google's car, a serious effort has been done by Google's engineers to reduce the number of sensors.

Contents

Abstract iii

Introduction v

Notations viii

List of Figures xi

1 Model predictive contouring control 1
- 1.1 Mathematical set-up . . . 1
  - 1.1.1 Geometrical errors . . . 2
  - 1.1.2 Cost definition . . . 2
    - 1.1.2.1 Linearization of the geometrical errors . . . 4
    - 1.1.2.2 Spline parametrization . . . 5
- 1.2 Putting cost in standard form . . . 5
  - 1.2.1 Constraints . . . 7
  - 1.2.2 Model specific constraints . . . 8
- 1.3 General algorithm . . . 9

2 Simulations 11
- 2.1 On a simple model . . . 11
  - 2.1.1 Equation of the model . . . 11
  - 2.1.2 Simulation results . . . 12
    - 2.1.2.1 Effect of R . . . 12
    - 2.1.2.2 Computing time . . . 13
- 2.2 On the model developed by Spengler and Gammeter . . . 14
  - 2.2.1 Equation of the model . . . 14
  - 2.2.2 Initialization of the trajectory . . . 15
  - 2.2.3 Choosing the right horizon . . . 15

3 Extensions of model predictive contouring control 17
- 3.1 How to stay on a given track? . . . 17
  - 3.1.1 Derivation of constraints . . . 17
  - 3.1.2 Model noise . . . 18
  - 3.1.3 Correspondence between center of the track and planned trajectory 19
- 3.2 On-line path planning . . . 19
- 3.3 Off-line path planning . . . 20

Conclusion 20

Bibliography **23**

A Calculus of the derivative of ϕ **27**

B Matrix exponential formula for discretization **29**

Notations

General conventions

$\boldsymbol{v}$	Bold notation is used for vectors.
F	Capital letters are used for matrices.
$\boldsymbol{F}$	Capital letters in bold are used for vectors whose size is dependent of the horizon.
$\mathcal{F}$	Curly letters are used to represent whose size is dependent of the horizon. It could also represents ensemble.
δx	Small delta is used to represent the difference between the value at the current iteration and the previous one.
Δx	Big delta represents a time difference derivative.
∇	Gradient.
$\partial_s x$	Partial derivative of x with respect to s.
$\mathcal{M}_n(\mathbb{K})$	Vector space of square matrices of size n on the field $\mathbb{K}$.
$\mathcal{S}^+$	Ensemble of positive symmetric matrices.
$\mathcal{S}^{++}$	Ensemble of strictly positive symmetric matrices.

Names

$\boldsymbol{\xi}$	Original state of the system of dimension $n_{\boldsymbol{\xi}}$.
$\mathcal{X}$	Ensemble of possible states.
$\boldsymbol{u}$	Original input of dimension $n_{\boldsymbol{u}}$.
$\mathcal{U}$	Ensemble of possible inputs.
N	Horizon of prediction.
θ	Virtual state which represents the curvilinear abscissa of the desired position of the car. In practice, it is the parameter of the followed curve.
v	Virtual speed.
$\boldsymbol{x}$	Augmented state (original and virtual state) of dimension $n_{\boldsymbol{x}}$.
$\boldsymbol{X}$	States for the planned trajectory. Dimension is $n_{\boldsymbol{X}} = N \cdot n_{\boldsymbol{x}}$.

Q	Weight matrix for the penalty of lagging and contouring errors.
R	Weight matrix for the penalty on the input.
$\hat{\epsilon}^c$, $\hat{\epsilon}^l$	Contouring and lagging error.
$\hat{\epsilon}^{a,c}$, $\hat{\epsilon}^{a,l}$	Approximate contouring and lagging error.
$\boldsymbol{t}(\theta)$, $\boldsymbol{n}(\theta)$	Tangent and normal vector of the curve at curvilinear abscissa θ.

List of Figures

1 Google self-driven car at TED 2011. vi
2 MIT car for the DARPA challenge vi

1.1 Problems with classical MPC for controlling a car 2
1.2 Geometrical error definitions 3
1.3 Comparison between normalized and unnormalized splines 6

2.1 The effect of R matrix 12
2.2 Computing time for one iteration of the algorithm 13
2.3 Calculation of the initial trajectory. 15
2.4 Mean distance of the trajectory to the desired path 16

3.1 Constraints are set using the virtual state 18
3.2 The effect of removing the $\mathcal{L}$ first constraints 19
3.3 The correspondence between the car state and the constraint to stay on the track 20
3.4 Optimal trajectory found using the simple model. 21

B.1 Matrix exponential implementation in MATLAB is $\mathcal{O}(n^3)$. 30

Chapter 1

Model predictive contouring control

Model predictive contouring control is an algorithm which aims to make a dynamic system (e.g a car) follow a given trajectory while minimizing the time to cover the track. Like model predictive control (MPC), we predict the state of the car and the inputs through an optimization problem. Additionally, the algorithm enables to control the deviation of the parameters and how likely it will promote speed versus trajectory deviation. We have to keep in mind that maximizing the speed of the car and minimizing the deviation to the track are two incompatible objectives, and that a trade-off has to be made, which is a point on the famous Pareto front.

In previous works about car control using MPC, the speed evaluated in the optimization routine is the speed of the car. This evaluation is unfair since the speed of the car can be higher, while the time to cover the track can be greater. This is illustrated Fig. 1.1.

The fundamental idea of model predictive contouring control (MPCC) is to add a virtual state which represent a virtual car on the trajectory we want to follow. The speed which is optimized is the speed of this virtual car. This avoids situation presented Fig. 1.1.

In this part, the equations of MPCC are derived for a general dynamical system.

1.1 Mathematical set-up

The car is assumed to be represented by a possibly non-linear system

$$\dot{\boldsymbol{\xi}} = f\left(\boldsymbol{\xi}, \boldsymbol{u}\right) \tag{1.1}$$

whose linearization in $(\boldsymbol{\xi}, \boldsymbol{u}) = \left(\delta\boldsymbol{\xi}^0, \delta\boldsymbol{u}^0\right)$ and time discretization is given by:

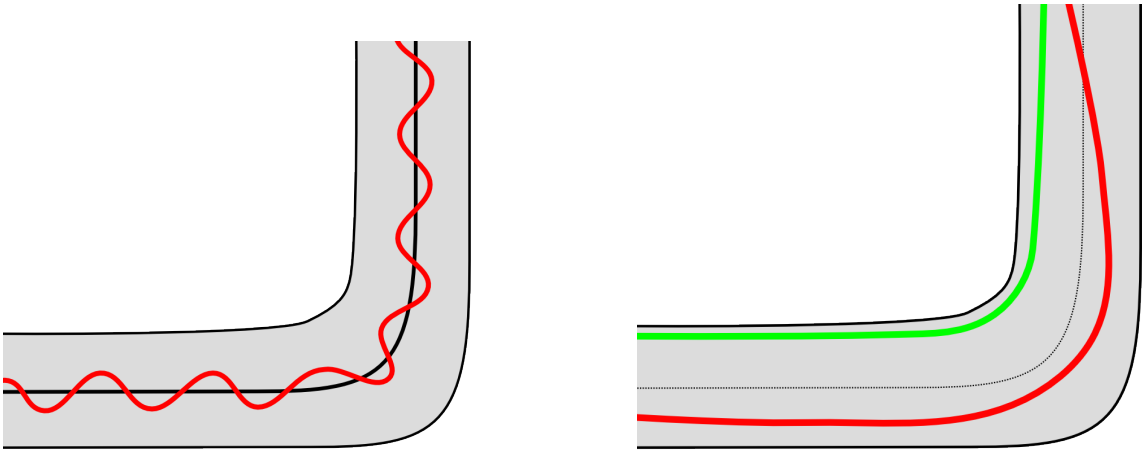
$$\begin{aligned} \delta\dot{\boldsymbol{\xi}} &= \left.\frac{\partial f}{\partial \boldsymbol{\xi}}\right|_{\boldsymbol{\xi}=\boldsymbol{\xi}^0, \boldsymbol{u}=\boldsymbol{u}^0} \delta\boldsymbol{\xi} + \left.\frac{\partial f}{\partial \boldsymbol{u}}\right|_{\boldsymbol{\xi}=\boldsymbol{\xi}^0, \boldsymbol{u}=\boldsymbol{u}^0} \delta\boldsymbol{u} \\ &= A_c\left(\boldsymbol{\xi}^0, \boldsymbol{u}^0\right)\delta\boldsymbol{\xi} + B_c\left(\boldsymbol{\xi}^0, \boldsymbol{u}^0\right)\delta\boldsymbol{u} \end{aligned} \tag{1.2}$$

where

$$\delta\boldsymbol{\xi} = \boldsymbol{\xi} - \boldsymbol{\xi}^0 \text{ and } \delta\boldsymbol{u} = \boldsymbol{u} - \boldsymbol{u}^0$$

Part 2 shows two examples of such systems and how they can be derived.

The method proposed by [3] is to add a virtual state θ which represent the curvilinear abscissa of the desired position of the car on the desired trajectory. Then, this virtual state θ is used to compute the error between desired position and the actual one. This error ϵ_k is the distance between the virtual and the real car and it depends on the state $(\boldsymbol{\xi}_k, \theta_k)$ at time k.



(a) Considering the speed criterion, one can see that if the red curve can be followed faster than the black curve, the red one will be chosen, although the time to travel would not necessarily be minimized by this path.

(b) Is the red or the green curve better, if $v_{\text{red}} = 4\text{m/s}$ and $v_{\text{green}} = 3.5\text{m/s}$? The time to travel for the green curve will be smaller although the car go faster on the red curve. A classical MPC controller, like described in the text will choose the red curve although the objective that one want to optimize is the time to travel. MPCC solve this issue.

Figure 1.1: Problems with classical MPC for controlling a car. When following a given trajectory — the path tracking problem Fig. 1.1a, or when planning a trajectory, Fig. 1.1b, the desired trajectory is not guaranteed to be preferred by the algorithm.

1.1.1 Geometrical errors

The curve to follow is assumed to be parametrized by θ which is close to the the arc-length s, that is $\theta \simeq s$. $\frac{ds}{d\theta} \simeq 1$ is also required which is generally the case if $\theta \simeq s$. This is important because the virtual speed v used is the speed the car would have if it was able to perfectly follow the desired path and if $\frac{ds}{d\theta} = 1$.

The contouring error decomposition is defined Fig. 1.2: the lag error $\hat{\epsilon}^l$ is the projection onto the tangent of the path at the virtual current state of the error. The orthogonal part is called contouring error and noted $\hat{\epsilon}^c$.

The normalized tangent vector $\boldsymbol{t}(\theta_k) = \begin{pmatrix} \cos\phi(\theta_k) \\ \sin\phi(\theta_k) \end{pmatrix}$ of the curve in θ_k and $\boldsymbol{n}(\theta_k) = \begin{pmatrix} \sin\phi(\theta_k) \\ -\cos\phi(\theta_k) \end{pmatrix}$ the normal vector are used to derive the errors. From 1.2, we have[1]

$$\hat{\epsilon}^c(\boldsymbol{\xi}_k, \theta_k) = \boldsymbol{n}(\theta_k) \cdot \begin{pmatrix} x_k - x_d(\theta_k) \\ y_k - y_d(\theta_k) \end{pmatrix} = \sin\phi(\theta_k) \cdot (x_k - x_d(\theta_k)) - \cos\phi(\theta_k) \cdot (y_k - y_d(\theta_k)) \quad (1.3)$$

$$\hat{\epsilon}^l(\boldsymbol{\xi}_k, \theta_k) = -\boldsymbol{t}(\theta_k) \cdot \begin{pmatrix} x_k - x_d(\theta_k) \\ y_k - y_d(\theta_k) \end{pmatrix} = -\cos\phi(\theta_k) \cdot (x_k - x_d(\theta_k)) - \sin\phi(\theta_k) \cdot (y_k - y_d(\theta_k)) \quad (1.4)$$

where $(x_d(\theta), y_d(\theta))$ is the point on the path at abscissa θ.

1.1.2 Cost definition

One wants to minimize these errors represented at each step $k+i$ by $\begin{bmatrix} \hat{\epsilon}^c_{k+i} & \hat{\epsilon}^l_{k+i} \end{bmatrix} Q \begin{bmatrix} \hat{\epsilon}^c_{k+i} \\ \hat{\epsilon}^l_{k+i} \end{bmatrix}$ for some symmetric positive matrix $Q \in \mathcal{S}^+$ usually diagonal which represents the relative

[1] The sign of $\hat{\epsilon}^c$ and $\hat{\epsilon}^l$ has been kept coherent with [3]

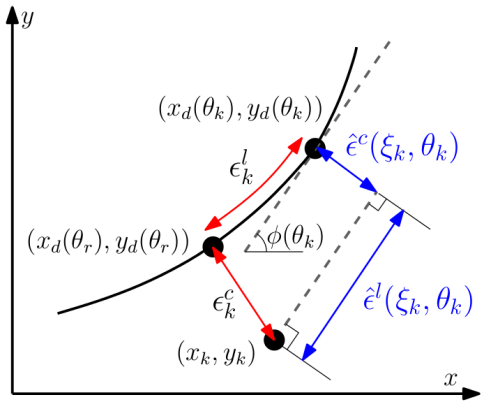


Figure 1.2: Geometrical error (from [3]). The point were the car should be is $(x_d(\theta_k), y_d(\theta_k))$ but the real position is (x_k, y_k). Since we do not know what is the nearest point $(x_d(\theta_r), y_d(\theta_r))$ of the current position onto the desired path, we approximate it by $(x_d(\theta_k), y_d(\theta_k))$ to get estimate of contouring $(\hat{\epsilon}^c)$ and lagging $(\hat{\epsilon}^l)$ errors. This is valid if $\epsilon_k^l \approx 0$.

penalty of the two errors. Since one also wants to minimize the time to do the trajectory, a linear term is added at each step $k+i$, $-q_\theta \theta_{k+i}$ for some $q_\theta \in \mathbb{R}^{+*}$ which will penalize low speeds.

Last but not least, one has to add a cost on the input $\boldsymbol{u}$ for the stability of the solution. Penalizing the time derivative $\Delta \boldsymbol{u}_k = \boldsymbol{u}_{k+1} - \boldsymbol{u}_k$ has been chosen because it was more adapted for the model used. Adding cost on $\boldsymbol{u}$ directly is however possible.

The dynamic of the virtual state θ is simulated using a virtual input v, such that

$$\theta_{k+1} = \theta_k + v_k$$

Finally, the following cost is obtained:

$$J_k = \sum_{i=1}^{N} \begin{bmatrix} \hat{\epsilon}^c_{k+i,k} & \hat{\epsilon}^l_{k+i,k} \end{bmatrix} Q \begin{bmatrix} \hat{\epsilon}^c_{k+i,k} \\ \hat{\epsilon}^l_{k+i,k} \end{bmatrix} - q_\theta \theta_{k+i} + \sum_{i=1}^{N-1} \begin{bmatrix} \Delta \boldsymbol{u}_{k+i}^\top & \Delta v_{k+i} \end{bmatrix} R \begin{bmatrix} \Delta \boldsymbol{u}_{k+i} \\ \Delta v_{k+i} \end{bmatrix} \tag{1.5}$$

where

- $\hat{\epsilon}^c_{k+i,k} = \hat{\epsilon}^c\left(\boldsymbol{\xi}_{k+i,k}, \theta_{k+i,k}\right)$ (respectively $\hat{\epsilon}^l_{k+i,k} = \hat{\epsilon}^l\left(\boldsymbol{\xi}_{k+i,k}, \theta_{k+i,k}\right)$) is the contouring error (resp. the lag error) predicted for time $k+i$ at time k. See section 1.1.1 for a definition.
- Typically, one has $R = \begin{pmatrix} R_{\Delta \boldsymbol{u}} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} \in \mathcal{S}^{++}$.

In the following, the second index k is omitted in $\boldsymbol{\xi}_{k+i,k}$ for clarity, assuming that the current time is k.

Because one wants to follow a path, the state is assumed to be made up of the position:$\boldsymbol{\xi} = \begin{pmatrix} x \\ y \\ \vdots \end{pmatrix}$.

To solve this problem using standard solver (e.g QP solvers), we need to derive (1.5) and to find an appropriate state in order to solve a quadratic problem. All the rest of this part is dedicated to put the cost (1.5) into a such form.

1.1.2.1 Linearization of the geometrical errors

Geometrical errors are not linear functions of the state. Since one wants to solve a QP, linearizing the errors $\hat{\epsilon}^c$ and $\hat{\epsilon}^l$ around the current planned trajectory at time k, $\Xi_k^0 = \left\{\boldsymbol{\xi}_{k+i}^0, \theta_{k+i}^0 \dots \boldsymbol{\xi}_{k+N}^0, \theta_{k+N}^0\right\}$ are required. This linearization is called a Linear Time Varying approximation (LTV) by [3].

An approximation $\hat{\epsilon}^{a,c}$ of $\hat{\epsilon}^c$ at the point $(\boldsymbol{\xi}_0, \theta_0)$ is given using the Taylor formula:

$$\forall \left(\delta\boldsymbol{\xi}, \delta\theta\right) \in \mathbb{R}^{n_{\boldsymbol{x}}},\ \hat{\epsilon}^{a,c} = \hat{\epsilon}^c\left(\boldsymbol{\xi}_0, \theta_0\right) + \nabla\hat{\epsilon}^c\left(\boldsymbol{\xi}_0, \theta_0\right) \left[\begin{array}{c} \delta\boldsymbol{\xi} \\ \delta\theta \end{array}\right]$$

The same approximation $\hat{\epsilon}^{a,l}$ for the lag error $\hat{\epsilon}^l$ yields to the condensed form:

$$\forall \left(\delta\boldsymbol{\xi}, \delta\theta\right) \in \mathbb{R}^{n_{\boldsymbol{x}}},\ \left[\begin{array}{c} \hat{\epsilon}^{a,c} \\ \hat{\epsilon}^{a,l} \end{array}\right] = \boldsymbol{\epsilon}^0 + \boldsymbol{\epsilon}^1 \left[\begin{array}{c} \delta\boldsymbol{\xi} \\ \delta\theta \end{array}\right] \tag{1.6}$$

with

$$\boldsymbol{\epsilon}^0 = \left[\begin{array}{c} \hat{\epsilon}^c\left(\boldsymbol{\xi}_0, \theta_0\right) \\ \hat{\epsilon}^l\left(\boldsymbol{\xi}_0, \theta_0\right) \end{array}\right] \in \mathbb{R}^2 \text{ and } \boldsymbol{\epsilon}_k^1 = \left[\begin{array}{c} \nabla\hat{\epsilon}^c\left(\boldsymbol{\xi}_0, \theta_0\right) \\ \nabla\hat{\epsilon}^l\left(\boldsymbol{\xi}_0, \theta_0\right) \end{array}\right] \in \mathbb{R}^{2\times\left(n_{\boldsymbol{\xi}}+1\right)}$$

denote respectively the zeroth order and the first order of the approximation.

One needs to compute explicitly $\boldsymbol{\epsilon}^0$ and $\boldsymbol{\epsilon}^1$ with respect to the parameter of the curve that is followed (angle $\phi\left(\theta\right)$ and first derivatives).

In this part, the physicist notation $\frac{\partial f}{\partial x}\left(x_0, y_0, z_0\right) = \partial_x f\left(x_0, y_0, z_0\right)$ is adopted for clarity. The point of linearization is assumed to be $\left(\boldsymbol{\xi}_0, \theta_0\right) = \left(x_0, y_0, \cdots, \theta_0\right)$. The curve that has to be followed is assumed to be $\mathcal{C}^2$-class, which is the case for cubic splines that will be used.

Zero orders are straightforward, just replace $\left(\boldsymbol{\xi}_k, \theta_k\right)$ by $\left(\boldsymbol{\xi}_0, \theta_0\right)$ in (1.3) and (1.4).

First order are given by:

$$\nabla\hat{\epsilon}^c\left(\boldsymbol{\xi}_0, \theta_0\right)^\top = \left(\begin{array}{c} \sin\phi\left(\theta_0\right) \\ -\cos\phi\left(\theta_0\right) \\ \mathbf{0} \\ \partial_\theta\nabla\hat{\epsilon}^c\left(\boldsymbol{\xi}_0, \theta_0\right) \end{array}\right)$$

where

$$\begin{array}{rcl} \partial_\theta\nabla\hat{\epsilon}^c\left(\boldsymbol{\xi}_0, \theta_0\right) & = & \partial_\theta\phi\left(\theta_0\right) \cdot \left[\cos\phi\left(\theta_0\right) \cdot \left(x_0 - x_d\left(\theta_0\right)\right) + \sin\phi\left(\theta_0\right) \cdot \left(y_0 - y_d\left(\theta_0\right)\right)\right] \\ & - & \partial_\theta x_d\left(\theta_0\right) \cdot \sin\phi\left(\theta_0\right) + \partial_\theta y_d\left(\theta_0\right) \cdot \cos\phi\left(\theta_0\right) \end{array} \tag{1.7}$$

And

$$\nabla\hat{\epsilon}^l\left(\boldsymbol{\xi}_0, \theta_0\right)^\top = \left(\begin{array}{c} -\cos\phi\left(\theta_0\right) \\ -\sin\phi\left(\theta_0\right) \\ \mathbf{0} \\ \partial_\theta\nabla\hat{\epsilon}^l\left(\boldsymbol{\xi}_0, \theta_0\right) \end{array}\right)$$

where

$$\begin{array}{rcl} \partial_\theta\nabla\hat{\epsilon}^l\left(\boldsymbol{\xi}_0, \theta_0\right) & = & \partial_\theta\phi\left(\theta_0\right) \cdot \left[\sin\phi\left(\theta_0\right) \cdot \left(x_0 - x_d\left(\theta_0\right)\right) - \cos\phi\left(\theta_0\right) \cdot \left(y_0 - y_d\left(\theta_0\right)\right)\right] \\ & + & \partial_\theta x_d\left(\theta_0\right) \cdot \cos\phi\left(\theta_0\right) + \partial_\theta y_d\left(\theta_0\right) \cdot \sin\phi\left(\theta_0\right) \end{array} \tag{1.8}$$

One has to find $\partial_\theta\phi\left(\theta_0\right)$ and $\partial_\theta x_d\left(\theta_0\right)$. Since $s \approx \theta$, as discussed in section (1.1.1), it follows:

$$\partial_\theta x_d\left(\theta_0\right) = \partial_s x_d\left(\theta_0\right) \cdot \cancel{\partial_\theta s}^{\,1} \approx \partial_s x_d\left(\theta_0\right)$$

which is given by the parametrization of the curve. Similarly, one has

$$\partial_\theta \phi(\theta_0) = \partial_s \phi(\theta_0) \cdot \partial_\theta s \approx \partial_s \phi(\theta_0)$$

but calculating $\partial_s \phi(\theta_0)$ knowing $\partial_s x_d(\theta_0)$ and $\partial_s y_d(\theta_0)$ is needed. The tangent half-angle formula[2] for $\phi(\theta)$ is used:

$$\phi(\theta) = \phi(\partial_s x, \partial_s y) = 2\text{atan}\frac{\partial_s y}{\sqrt{(\partial_s x)^2 + (\partial_s y)^2} + \partial_s x} \tag{1.9}$$

Therefore, after some calculus detailed in Appendix A:

$$\partial_s \phi = \frac{\partial_s x \partial_{ss} y - \partial_{ss} x \partial_s y}{\partial_s x^2 + \partial_s y^2} \tag{1.10}$$

The use of tangent half-angle formula is motivated by the validity domain: (1.9) is well defined for all regular points of the trajectory (that is every points that verifies $\partial_s x \neq 0$ or $\partial_s y \neq 0$). Additionally, the formula has a well defined derivative over the same domain. However, the built-in function (in MATLAB) `atan2` is actually used in the calculation of ϕ rather than the expression given in (1.9) because it is inappropriate for floating point calculation[3].

One finally gets

$$\boldsymbol{\epsilon}^1_{\Xi,k,i} = \begin{bmatrix} \sin\phi(\theta_0) & -\cos\phi(\theta_0) & (0) & \partial_\theta \nabla \hat{\epsilon}^c(\boldsymbol{\xi}_0, \theta_0) \\ -\cos\phi(\theta_0) & -\sin\phi(\theta_0) & (0) & \partial_\theta \nabla \hat{\epsilon}^l(\boldsymbol{\xi}_0, \theta_0) \end{bmatrix}$$

with $\partial_\theta \nabla \hat{\epsilon}^c(\boldsymbol{\xi}_0, \theta_0)$ and $\partial_\theta \nabla \hat{\epsilon}^l(\boldsymbol{\xi}_0, \theta_0)$ given in (1.7) and (1.8).

1.1.2.2 Spline parametrization

The reference trajectory of the car has to be chosen sufficiently smooth — at least $\mathcal{C}^2$ — to permit the previous calculation. Having points of the desired trajectory, the simplest approach is to calculate the cubic spline interpolation of these points.

Cubic splines meet the requirement of being $\mathcal{C}^2$. As mentioned in the previous subsection, we also need that $\frac{ds}{d\theta} \simeq 1$ where θ is the parameter of the spline, and s is the arc-length. This condition has to be checked, because this assumption does not hold in the general case.

The correction that has been made is to normalize each part the spline: each part of the spline, between two points of passage, has a certain length which has to be equal to the arc-length. Figure 1.3 shows a comparison between unnormalized and normalized spline.

The effect of $\frac{ds}{d\theta} \neq 1$ will be briefly discussed in section 2.1.2. A better way to ensure $\frac{ds}{d\theta} \simeq 1$ is to add some degrees of freedom in the computation of the spline, for example by using quartic or quintic splines. This is developed in [1].

1.2 Putting cost in standard form

The goal of this section is to transform the problem to a cost

$$\delta \boldsymbol{X}_k^\top \mathcal{Q}_k \delta \boldsymbol{X}_k + \mathcal{F}_k^\top \delta \boldsymbol{X}_k \tag{1.11}$$

[2] in contrary to what is it written in [3]

[3] For example, it is undefined for $x < 0$ and $y = 0$, and may overflow near these regions

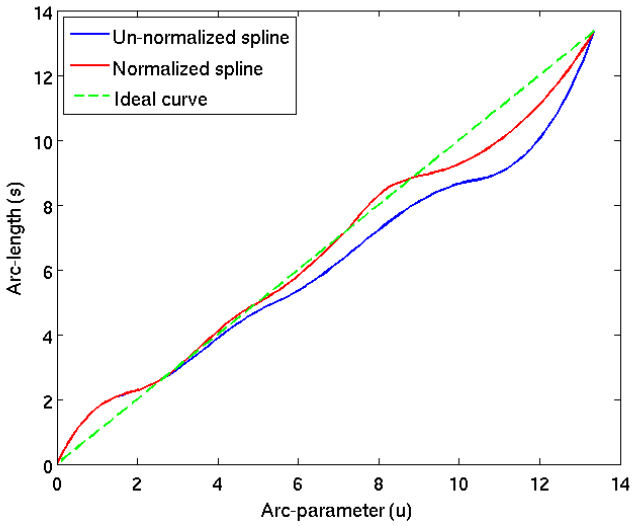


Figure 1.3: Comparison between normalized and unnormalized splines. Ideal curve is given by $s = \theta$.

for certain $\delta \boldsymbol{X}_k$, $\mathcal{Q}_k$ and $\mathcal{F}_k$, so that it can be solved by a standard QP solver.

The cost under the LTV approximation is:

$$J_k^a = \sum_{i=1}^{N} \begin{bmatrix} \hat{\epsilon}_{k+i}^{a,c} & \hat{\epsilon}_{k+i}^{a,l} \end{bmatrix} Q \begin{bmatrix} \hat{\epsilon}_{k+i}^{a,c} \\ \hat{\epsilon}_{k+i}^{a,l} \end{bmatrix} - q_\theta \theta_{k+i} + \sum_{i=1}^{N-1} \begin{bmatrix} \Delta \boldsymbol{u}_{k+i} & \Delta v_{k+i} \end{bmatrix} R \begin{bmatrix} \Delta \boldsymbol{u}_{k+i} \\ \Delta v_{k+i} \end{bmatrix}$$

The variables to optimize are $\{\delta \boldsymbol{\xi}_{k+1} \dots \delta \boldsymbol{\xi}_{k+N}\}$ and $\{\delta\theta_{k+1}, \dots \delta\theta_{k+N}\}$. Equation (1.6) is used to expand the i-th quadratic term in the previous equation:

$$\begin{aligned} \begin{bmatrix} \hat{\epsilon}_{k+i}^{a,c} & \hat{\epsilon}_{k+i}^{a,l} \end{bmatrix} Q \begin{bmatrix} \hat{\epsilon}_{k+i}^{a,c} \\ \hat{\epsilon}_{k+i}^{a,l} \end{bmatrix} &= \cancel{\boldsymbol{\epsilon}_{k+i}^{0,\top} Q \boldsymbol{\epsilon}_{k+i}^{0}} \rightarrow \text{const} + \begin{bmatrix} \delta \boldsymbol{\xi}_{k+i} \\ \delta\theta_{k+i} \end{bmatrix}^\top \underbrace{\boldsymbol{\epsilon}_{k+i}^{1,\top} Q \boldsymbol{\epsilon}_{k+i}^{1}}_{\tilde{Q}_{k+i}} \begin{bmatrix} \delta \boldsymbol{\xi}_{k+i} \\ \delta\theta_{k+i} \end{bmatrix} \\ &+ 2\boldsymbol{\epsilon}_{k+i}^{0,\top} Q \boldsymbol{\epsilon}_{k+i}^{1} \begin{bmatrix} \delta \boldsymbol{\xi}_{k+i} \\ \delta\theta_{k+i} \end{bmatrix} \end{aligned} \tag{1.12}$$

Adding the term $-q_\theta \theta_{k+i}$ — which is equivalent to $-q_\theta \delta\theta_{k+i}$, we get

$$\tilde{f}_{k+i} = 2\boldsymbol{\epsilon}_{k+i}^{0,\top} Q \boldsymbol{\epsilon}_{k+i}^{1} - q_\theta \cdot \begin{bmatrix} \mathbf{0}_{n_\xi} & 1 \end{bmatrix}$$

The state is defined by:

$$\delta \boldsymbol{X}_k = \begin{pmatrix} \delta \boldsymbol{\xi}_{k+1} \\ \delta\theta_{k+1} \\ \vdots \\ \delta \boldsymbol{\xi}_{k+N} \\ \delta\theta_{k+N} \\ \delta \boldsymbol{u}_k \\ \delta v_k \\ \vdots \\ \delta \boldsymbol{u}_{k+N-1} \\ \delta v_{k+N-1} \end{pmatrix} \in \mathbb{R}^{N \cdot (n_\xi + 1) + N \cdot (n_{\boldsymbol{u}} + 1)} \equiv \mathbb{R}^{n_{\boldsymbol{X}}} \tag{1.13}$$

Since one wants to penalize the (discrete) temporal derivative of the input $\boldsymbol{u}$

$$\Delta \boldsymbol{U}_k = \begin{pmatrix} \boldsymbol{u}_{k+1} - \boldsymbol{u}_k \\ v_{k+1} - v_k \\ \vdots \\ \boldsymbol{u}_{k+N-1} - \boldsymbol{u}_{k+N-2} \\ v_{k+N-1} - v_{k+N-2} \end{pmatrix} \in \mathbb{R}^{(N-1)\cdot n_{\boldsymbol{U}}},\ n_{\boldsymbol{U}} = n_{\boldsymbol{u}} + 1$$

and not $\boldsymbol{U}_k = (\boldsymbol{u}_k, v_k, \ldots, \boldsymbol{u}_{k+N-1}, v_{k+N-1})$ — remember that the solver can only solve (1.11) and that $\delta \boldsymbol{U}_k$ is part of $\delta \boldsymbol{X}$, one has to find a matrix $\mathcal{R}$ such that

$$\boldsymbol{U}_k \mathcal{R} \boldsymbol{U}_k = \Delta \boldsymbol{U}_k \begin{pmatrix} R & & \boldsymbol{0} \\ & \ddots & \\ \boldsymbol{0} & & R \end{pmatrix} \Delta \boldsymbol{U}_k$$

where $\mathcal{R} \in \mathbb{R}^{(N-1)n_{\boldsymbol{U}} \times (N-1)n_{\boldsymbol{U}}}$.

Clearly, $\mathcal{R}$ has to be symmetric and block-tridiagonal. One can see that the sum of all elements in $\mathcal{R}$ has to be 0 because in $\Delta \boldsymbol{U}$, there are the same number of positive and negative numbers. Therefore, one can think of (recall that $R \in \mathcal{S}^{++}$)

$$\mathcal{R} = \begin{pmatrix} R & -R & & & \\ -R & 2R & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & 2R & -R \\ & & & -R & R \end{pmatrix} \in \mathcal{M}_{N \cdot n_{\boldsymbol{U}}}(\mathbb{R})$$

$\boldsymbol{U}_k = (\boldsymbol{u}_k, v_k, \ldots, \boldsymbol{u}_{k+N-1}, v_{k+N-1})$ is not in the state but $\delta \boldsymbol{U}_k = (\delta \boldsymbol{u}_k, \delta v_k, \ldots, \delta \boldsymbol{u}_{k+N-1}, \delta v_{k+N-1})$. Therefore, a calculus like (1.12) gives:

$$\boldsymbol{U}_k \mathcal{R} \boldsymbol{U}_k = \left(\boldsymbol{U}_k^0 + \delta \boldsymbol{U}_k\right)^\top \mathcal{R} \left(\boldsymbol{U}_k^0 + \delta \boldsymbol{U}_k\right) = \delta \boldsymbol{U}_k^\top \mathcal{R} \delta \boldsymbol{U}_k + 2 \boldsymbol{U}_k^{0,\top} \mathcal{R} \delta \boldsymbol{U}_k + \text{const}$$

One has

$$J_k^a = \delta \boldsymbol{X}_k^\top \mathcal{Q}_k \delta \boldsymbol{X}_k + \mathcal{F}_k^\top \delta \boldsymbol{X}_k \tag{1.14}$$

where

$$\mathcal{Q}_k = \begin{pmatrix} \tilde{Q}_{k+1} & & & \boldsymbol{0} \\ & \ddots & & \\ & & \tilde{Q}_{k+N} & \\ \boldsymbol{0} & & & \mathcal{R} \end{pmatrix} \in \mathcal{M}_{n_{\boldsymbol{X}}}(\mathbb{R}) \tag{1.15}$$

and

$$\mathcal{F}_k^\top = \begin{pmatrix} \tilde{f}_{k+1} & \cdots & \tilde{f}_{k+N} & 2\boldsymbol{U}_k^{0,\top} \mathcal{R} \end{pmatrix} \in \mathbb{R}^{n_{\boldsymbol{X}}} \tag{1.16}$$

with $\tilde{Q}_{k+i} = \boldsymbol{\epsilon}_{k+i}^{1,\top} Q \boldsymbol{\epsilon}_{k+i}^1$ and $\tilde{f}_{k+i} = 2\boldsymbol{\epsilon}_{k+i}^{0,\top} Q \boldsymbol{\epsilon}_{k+i}^1 - q_\theta \cdot \begin{bmatrix} \boldsymbol{0}_{n_\xi} & 1 \end{bmatrix}$.

1.2.1 Constraints

The constraints have also to be put in a standard form to be solved by a QP solver, which amounts to finding $\mathcal{A}_{\text{eq}}$ and $\boldsymbol{b}_{\text{eq}}$ such that $\mathcal{A}_{\text{eq}} \delta \boldsymbol{X}_k = \boldsymbol{b}_{\text{eq}}$.

The first constraints are given by the linearized system (1.2). Linearizing around the trajectory $\Xi_k^0 = \left\{\boldsymbol{\xi}_{k+i}^0, \theta_{k+i}^0 \dots \boldsymbol{\xi}_{k+N}^0, \theta_{k+N}^0\right\}$ and the corresponding inputs $\boldsymbol{U}_k^0 = \left\{\boldsymbol{u}_k^0, \dots, \boldsymbol{u}_k^{N-1}\right\}$, we get:

$$\forall i \in 1 \dots N,\ \delta\boldsymbol{\xi}_{k+i} = A_{k+i-1}^0 \delta\boldsymbol{\xi}_{k+i-1} + B_{k+i-1}^0 \delta\boldsymbol{u}_{k+i-1}$$

where

$$\forall k,\ A_k^0 = \partial_{\boldsymbol{\xi}} f\left(\boldsymbol{\xi}_k^0, \boldsymbol{u}_k^0\right) \text{ and } B_k^0 = \partial_{\boldsymbol{u}} f\left(\boldsymbol{\xi}_k^0, \boldsymbol{u}_k^0\right)$$

Additionally, one has

$$\forall i \in 1 \dots N,\ \delta\theta_{k+i} = \delta\theta_{k+i-1} + \delta v_{k+i-1}$$

Let

$$\forall i \in 0 \dots N-1,\ \mathcal{A}_{k+i}^0 = \begin{pmatrix} A_{k+i}^0 & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} \in \mathcal{M}_{n_{\boldsymbol{x}}}(\mathbb{R}) \text{ and } \mathcal{B}_{k+i}^0 = \begin{pmatrix} B_{k+i}^0 & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} \in \mathcal{M}_{n_{\boldsymbol{x}}}(\mathbb{R})$$

In standard form, one gets

$$\mathcal{A}_{\text{eq}} \delta\boldsymbol{X}_k = \boldsymbol{b}_{\text{eq}} \tag{1.17}$$

where

$$\mathcal{A}_{\text{eq}} = \begin{pmatrix} -I_{n_{\boldsymbol{x}}} & & & & \mathcal{B}_k^0 & & \\ \mathcal{A}_{k+1}^0 & \ddots & & & & \ddots & \\ & \ddots & \ddots & & & & \ddots \\ & & \mathcal{A}_{k+N-1}^0 & -I_{n_{\boldsymbol{x}}} & (0) & & \mathcal{B}_{k+N-1}^0 \end{pmatrix} \in \mathbb{R}^{N \cdot n_{\boldsymbol{x}} \times n_{\boldsymbol{X}}} \tag{1.18}$$

and ($\delta\boldsymbol{x}_k = 0$ is the current state, which is known)

$$\boldsymbol{b}_{\text{eq}} = -\begin{pmatrix} \mathcal{A}_k^0 \delta\boldsymbol{x}_k \\ \mathbf{0}_{n_{\boldsymbol{x}}} \\ \vdots \\ \mathbf{0}_{n_{\boldsymbol{x}}} \end{pmatrix} = \mathbf{0}_{n_{\boldsymbol{X}}} \in \mathbb{R}^{n_{N \cdot n_{\boldsymbol{x}}}} \tag{1.19}$$

Then, the inequalities

$$\begin{gathered} \boldsymbol{u}_{k+i-1} \in \mathcal{U},\ v_{k+i-1} \in [0, v_{\max}] \\ \boldsymbol{\xi}_{k+i} \in \mathcal{X},\ \theta_{k+i} \in [\theta^s, 0] \end{gathered}$$

are handled in a standard way.

The problem is therefore written

$$\min_{\delta\boldsymbol{X}} \quad \delta\boldsymbol{X}^\top \mathcal{Q}_k \delta\boldsymbol{X} + \mathcal{F}_k^\top \delta\boldsymbol{X} \tag{1.20}$$

$$\text{s.t} \quad \begin{cases} \mathcal{A}_{\text{eq}} \delta\boldsymbol{X} = \boldsymbol{b}_{\text{eq}} \\ \boldsymbol{u}_{k+i-1} \in \mathcal{U},\ v_{k+i-1} \in [0, v_{\max}] \\ \boldsymbol{\xi}_{k+i} \in \mathcal{X},\ \theta_{k+i} \in [\theta^s, 0] \end{cases}$$

where corresponding matrices are found in (1.15)–(1.19).

1.2.2 Model specific constraints

For some models, like the one studied in section 2.2, some additional constraints are needed, for example that the speed is positive. For a given index i and time index k it could be therefore required that

$$x_{i,k} > \alpha$$

where α is a constant. Since the state is $\delta\boldsymbol{x}_k$, one has $\delta x_{i,k} > \alpha - x_{i,k}$.

1.3 General algorithm

1. Find a initial feasible trajectory Ξ_0 using following steps
   (a) Initialize $\boldsymbol{U}_0 = \mathbf{0}$ and compute corresponding $\boldsymbol{X}$ by integrating the non-linear ODE (1.1).
   (b) while not converging
      i. Compute $\forall i \in 1 \dots N, \boldsymbol{\epsilon}_i^0$ and $\boldsymbol{\epsilon}_i^1$ using $\boldsymbol{U}_0$, which enables to compute $\mathcal{Q}_0$ and $\mathcal{F}_0$.
      ii. Solve (1.20) to find $\delta\boldsymbol{U}$. Set $\boldsymbol{U}_0 := \boldsymbol{U}_0 + \delta\boldsymbol{U}$, compute $\boldsymbol{X}$ from the non-linear system and from new $\boldsymbol{U}_0$ and loop.
2. At each time-step $k \geq 1$
   (a) Shift command vector $\delta\boldsymbol{U}_{k-1}$, append a feasible input (e.g. $\mathbf{0}$), and replace the current state by its current value to get $\delta\boldsymbol{U}_k$:

$$\delta\boldsymbol{U}_k \leftarrow \begin{bmatrix} \delta\boldsymbol{U}_{k-1}^{(2)} \\ \vdots \\ \delta\boldsymbol{U}_{k-1}^{(N-1)} \\ \mathbf{0} \end{bmatrix}$$

   (b) Compute the new trajectory Ξ_k by integrating the non-linear ODE (1.1).
   (c) Compute $\mathcal{Q}_k$ and $\mathcal{F}_k$.
   (d) Solve (1.20) to find the new $\delta\boldsymbol{U}_k$.

Chapter 2

Simulations

In this part, simulation results for two different models are presented. The first one is the simplest model one can imagine for a car. Its behavior can be simply understood and is therefore an important tool to verify the algorithm properties and behaviors. The second one is a more complicated model, derived in a previous semester thesis, which is based on a bicycle model. Some assumptions are made to establish a more realistic behavior for the speed. In both models, the car is assumed not to slip.

2.1 On a simple model

The model studied in this section is really simple but has the advantage that one can easily predict its behavior. The speed v can be set instantly, and one can directly set the time derivative of the speed angle. With this model, any track (provided there is no constraint) can be perfectly followed at any speed. This is therefore an important model to assess the algorithm validity.

2.1.1 Equation of the model

The following model is used:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\varphi} \end{pmatrix} = f(\boldsymbol{\xi}, \boldsymbol{u}) = \begin{pmatrix} v\cos\varphi \\ v\sin\varphi \\ \delta \end{pmatrix} \tag{2.1}$$

The jacobians are given by

$$A_c = \left.\frac{\partial f}{\partial \boldsymbol{\xi}}\right|_{\boldsymbol{\xi}=(x_0,y_0,\varphi_0),\boldsymbol{u}=(v_0,\delta_0)} = \begin{pmatrix} 0 & 0 & -v_0\sin\varphi_0 \\ 0 & 0 & v_0\cos\varphi_0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$B_c = \left.\frac{\partial f}{\partial \boldsymbol{u}}\right|_{\boldsymbol{\xi}=(x_0,y_0,\varphi_0),\boldsymbol{u}=(v_0,\delta_0)} = \begin{pmatrix} \cos\varphi_0 & 0 \\ \sin\varphi_0 & 0 \\ 0 & 1 \end{pmatrix}$$

Therefore the discrete corresponding matrices are

$$A_d = \exp(A_c \cdot t_s) = I + A_c \cdot t_s$$

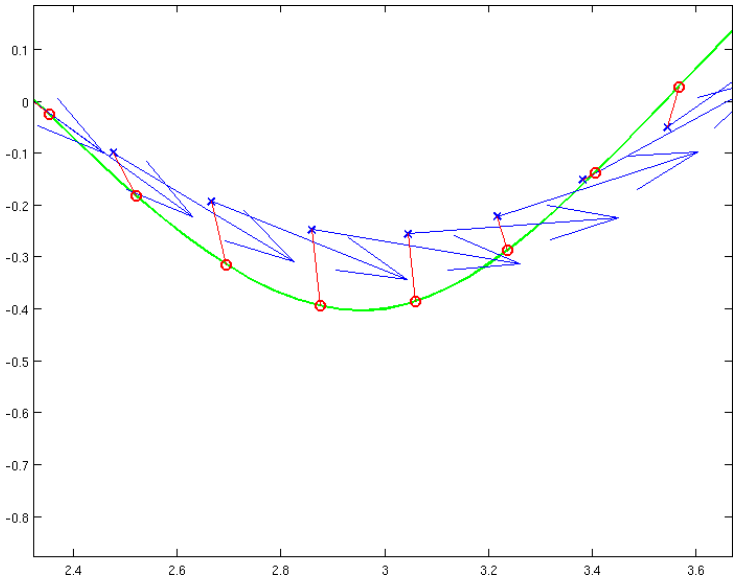


Figure 2.1: The reference path is shown in green, the position of the car (with its heading direction) in blue, and red dots represents the virtual state on the path. As R increases, the optimizer has to find a trade-off between following the path and minimizing input changes.

where t_s is the sampling time and since $A_c^2 = 0$; and

$$\begin{aligned} B_d &= \int_0^{ts} \exp\left(A\left(t_s - \tau\right)\right) d\tau \cdot B_c \\ &= \int_0^{ts} \exp\left(A \cdot \tau\right) d\tau \cdot B_c \\ &= \int_0^{ts} \left(I + A_c \cdot \tau\right) d\tau \cdot B_c \\ &= \left[t_s \cdot I + \frac{t_s^2}{2} \cdot A_c\right] B_c \end{aligned}$$

Given (2.1), one can see that the next step is colinear to $\boldsymbol{T}\left(\varphi\right) = \begin{pmatrix} \cos\varphi \\ \sin\varphi \end{pmatrix}$, and that the speed can be arbitrarily set (within the eventual constraints). Therefore, one can deduce that without penalty on $\boldsymbol{U}$, the optimal trajectory has a cost of zero in terms of lagging error and contouring error.

2.1.2 Simulation results

2.1.2.1 Effect of R

The R matrix enables to penalize $\Delta\boldsymbol{U}$ and plays a double role: on the one hand, it decreases the energy usage; on the other hand, it promotes a stable solution by avoiding very fast changes of the input, which also improves the convergence of the algorithm. Therefore R has to be chosen sufficiently big to avoid bad solutions and improve convergence of the algorithm.

With R increasing, the optimizer has to do a trade-off between following the path and minizing the change of inputs, for example the change of steering angle. This leads to artifacts as shown in Fig. 2.1.

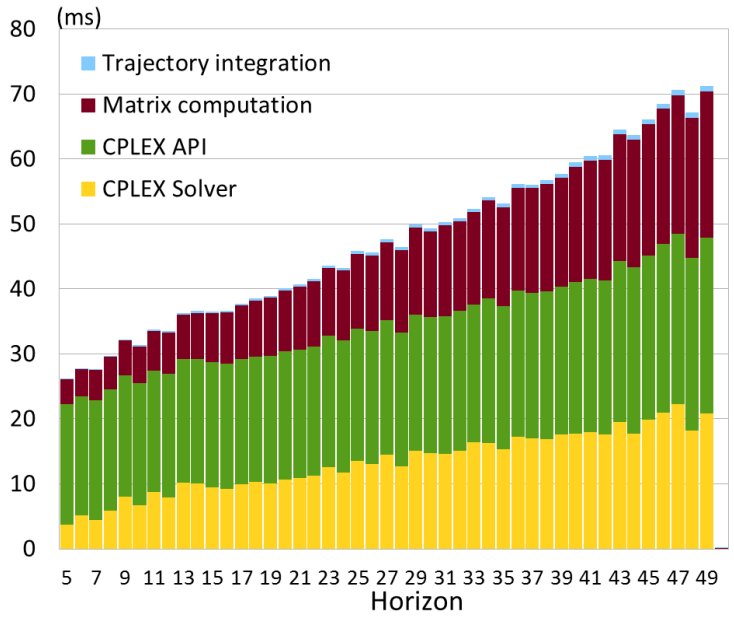


Figure 2.2: Computing time for one iteration of the algorithm for the Spengel & Gammeter model. The computing time for each step, increases approximately linearly with N. "Matrix computation" corresponds to the time to compute the matrices $\mathcal{Q}_k$, $\mathcal{F}_k$, $\mathcal{A}_{\text{eq}}$ and $\boldsymbol{b}_{\text{eq}}$; "CPLEX Solver" is the time claimed by CPLEX to solve the problem; "CPLEX API" is the time taken by the MATLAB API of CPLEX, that is the timespan timed in MATLAB of the CPLEX call minus the time to claimed by CPLEX to solve the problem; "Trajectory integration" corresponds to the time to integrate the non-linear differential equation 2.1. The most time consuming task is the factoring that MATLAB APIs of CPLEX are doing on the data.

2.1.2.2 Computing time

Since the goal is to have real-time control, computing time has to be small enough. Real-time control in the case of ACL experiments means that one iteration runs in less than 50ms. Therefore, the horizon has to be chosen so as to fit within this bound.

The first naive MATLAB implementation used 1.5 s per iteration for $N = 40$ which is by far too slow. Using a profiler, time consuming functions have been rewriten, dynamic allocation has been removed, anonymous functions have been replaced by tables and the code has been simplified. This leads to 8X performance improvement. The computing time of one iteration for the optimized version is shown Fig. 2.2. One can see that all computing times increases almost linearly with N. This is especially noticeable for the time to solve the quadratic problem: as the matrix $\mathcal{Q}_k$ is sparse, the number of terms increase quadratically although the number of non-zero terms increase linearly. Therefore, one can expect quadratic complexity (or even more) with respect to N for the solver computing time. But the linear increase is the signature that CPLEX takes in account the sparsity of the input matrix.

A further improvement has been made by rewriting the integration of the trajectory in C through MATLAB MEX wrappers. As one can see Fig. 2.2, the integration of the trajectory takes the major part of the computing time. This is not due to the computational expense of solving such a non-linear system, but because of the MATLAB API for solving the trajectory: each time the integrator has to evaluate the model, a call to the anonymous function representing the model is done leading to a huge waste of time. Rewriting the code in C using GNU Scientific library (GSL), allows 200X performance improvement on this part of the algorithm, allowing to compute one step for $N = 30$ in

49ms.

The next step to improve performance is to call the CPLEX solver directly in C in order to avoid the time consumming MATLAB APIs as shown in Fig. 2.2. An another choice would be to use a handcrafted code to solve the QP problem.

2.2 On the model developed by Spengler and Gammeter

Unlike the previous model, this one adds some dynamics on the speed. The input is now the duty cycle controlling the throttle and the steering angle. The geometrical underlying model is not anymore a point, but a bicycle. For more details, see [4].

Although more realistic, this model causes some problems of convergence for the algorithm.

2.2.1 Equation of the model

The model is given, for $v > 0$ by:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\varphi} \\ \dot{v} \end{pmatrix} = f(\boldsymbol{\xi}, \boldsymbol{u}) = \begin{pmatrix} v\cos(\varphi + C_1\delta) \\ v\sin(\varphi + C_1\delta) \\ C_2\delta v \\ C_{m1}D - C_{m2}Dv - C_{r0} - C_{r2}v^2 - (v\delta)^2 C_2C_1 \end{pmatrix}$$

The jacobians are given by

$$A_c = \begin{pmatrix} 0 & 0 & -v\sin(\varphi + C_1\delta) & \cos(\varphi + C_1\delta) \\ 0 & 0 & v\cos(\varphi + C_1\delta) & \sin(\varphi + C_1\delta) \\ 0 & 0 & 0 & C_2\delta \\ 0 & 0 & 0 & -C_{m2}D - 2(C_{r2} + C_2C_1\delta^2)v \end{pmatrix}$$

$$B_c = \begin{pmatrix} 0 & -C_1 v\sin(\varphi + C_1\delta) \\ 0 & C_1 v\cos(\varphi + C_1\delta) \\ 0 & C_2 v \\ C_{m1} - C_{m2}v & -2C_2C_1v^2\delta \end{pmatrix}$$

The discrete matrices are more complicated to derive than for the simple model. $A_d = \exp(A_c \cdot t_s)$ can not be further simplified. For B_d, we derive a general form:

$$\begin{aligned} B_d &= \int_0^{ts} \exp(A_c \cdot \tau)\, d\tau \cdot B_c \\ &= A_c^D [\exp(A_c \cdot t_s) - I] \cdot B_c \end{aligned}$$

where A_c^D is the Drazin inverse, which verifies in particular $A_c^D A_c^{k+1} = A_c^k$. Computing explicitly this Drazin inverse, in case of A_c being not invertible (which is the case here), is not straightforward. Another option could be to decompose A_c into a sum of matrices which commute, and for which there exist a simple formula for matrix exponential (typically nilpotent matrices, using Dunford decomposition).

A simpler and more generic approach is to use the formula (proven in appendix B)

$$\exp\begin{pmatrix} A_c \cdot t_s & B_c \cdot t_s \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} A_d & B_d \\ 0 & 0 \end{pmatrix}$$

Although this formula is general, it is much faster to use a direct formula if available since expm complexity is $\mathcal{O}(n^3)$ where n is the size of the matrix, as shown in appendix B. That is why it has not been used in the previous problem.

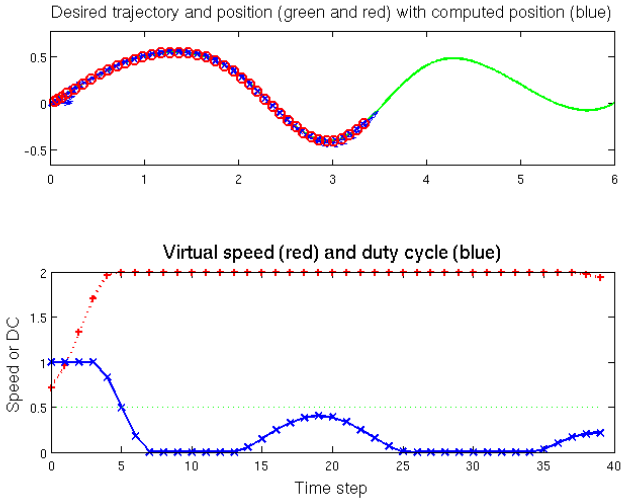


Figure 2.3: Calculation of the initial trajectory.

2.2.2 Initialization of the trajectory

The initialization of the trajectory did not work when applying the algorithm of section 1.3. The strong non-linearities of the model decrease the domain of validity of the linearized model: if the input update is too big, as in the case of an initialization, the linearized and the original model are too different, and algorithm diverges. A filter is therefore added when updating $\boldsymbol{U}$ at each iteration:

$$\boldsymbol{U} \leftarrow \begin{cases} \boldsymbol{U} + \frac{1}{10}\delta\boldsymbol{U} & \text{if } \delta\boldsymbol{U} > \text{criterion} \\ \boldsymbol{U} + \delta\boldsymbol{U} & \text{otherwise} \end{cases}$$

A function smoother than the piecewise gain could also be implemented. Figure 2.3 shows the result of initialization. The virtual speed has first to be limited to ensure convergence. After a few iterations, this limit can be progressively increased.

2.2.3 Choosing the right horizon

N is the main parameter that has to be chosen. It influences a lot of the properties of the algorithm like robustness against noise, computing time and optimality of the trajectory.

The robustness has been studied by adding random Gaussian noise to the (real) current state in the following set up. For a given horizon and a given noise standard deviation, the trajectory is initialized. After convergence, 20 steps are computed; at each step, the current state is set to be the prediction calculated in the previous step, and some random Gaussian noise is added to take into account the imperfections of the model. After that, one step is computed without adding noise, and the median distance of the last $\left\lceil \frac{N}{2} \right\rceil$ predicted[1] steps to the desired trajectory are computed. This is done many times (16) to average the prediction, and for a given range of N and standard deviations. Results are shown in Fig. 2.4.

The sensitivity to the noise depends of the horizon:

- for a very large horizon ($N \simeq 40$, depends on the curvature), the system is very sensitive: initial errors (especially angular one) become very big at the end of the

[1] in case of absence of noise, predicted steps correspond to future real steps

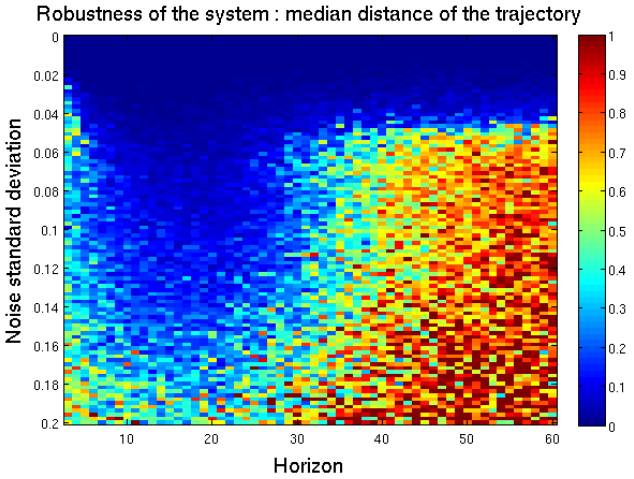


Figure 2.4: Mean distance of the trajectory to the desired path. One can see that if N increases, the system becomes more sensitive to noise.

predicted trajectory, which leads the system very far from the optimal solution, and therefore it may not converge. This is purely an effect of the linearization of the system.

- for a small horizon ($N \simeq 2$), the system is not sensitive to noise but this leads to short term decisions which can be very sub-optimal in case for example of a sharp turn. However, the instability measured in Fig. 2.4 for small N is mainly due to the way of measuring deviation: when the horizon is small, the car cannot physically reach the trajectory to be tracked within N steps.

Another parameter one have to take into account is the time to solve one iteration: if one increases N, the time to compute one iteration increases almost linearly; and one has a time limit for computation which is the sampling time. And if N is sufficiently small, one can imagine doing more than one iteration at each time-step to improve convergence.

Finally, N has to be chosen not too big but also not too small. A good methodology to choose N is to determine the level of noise one expects in the real system, and using Fig. 2.4 or simulations, to determine the appropriate N.

Additionally, if the limit on the virtual speed v is set to high, the system further loses robustness as mentionned in [3].

Chapter 3

Extensions of model predictive contouring control

In this part, constraints to ensure that the car stay on a given track are derived. This allows the car to deviate from the planned trajectory if for example an obstacle appears unexpectedly: on-line path planning becomes possible. Additionally, relaxing some components of the cost is possible — like the contouring cost, to enable more global path planning: if the horizon is sufficiently large, the car will be able to optimize the path while enforcing constraints (stay on the track).

3.1 How to stay on a given track?

Until now, nothing guarantees that we are staying near the desired trajectory. In this section, we derive some conditions to enforce this requirement.

3.1.1 Derivation of constraints

The main idea is to add linear constraints (half-plane constraints) which depend not on the car position but on the virtual state. For each virtual state, the best line which fits the left or the right boundary of the track (2 constraints per position) is calculated. Since the path is defined by its center and its width, a correspondence (off-line computation) is made between the spline of the path and the spline for the center of the track. The result is shown Fig. 3.1. Using the virtual state to set the constraints enables the correspondence to be done off-line. Otherwise, the calculus would have to be done on-line. The constraint is then applied to the real car position.

Let $(X_\theta, Y_\theta, \phi_\theta)$ be the center of the track with direction ϕ_θ at virtual abscissa θ, and d the width of the track. d is supposed not to vary with θ, although the derivation stays the same if d varies with θ (see section 3.2). Since linear constraints are required, one models the track locally to be a lane made of two linear inequalities.

For $\boldsymbol{t} = \begin{pmatrix} \cos\phi_\theta & \sin\phi_\theta \end{pmatrix}^\top$, the orthogonal vector is given by $\boldsymbol{n} = \begin{pmatrix} \sin\phi_\theta & -\cos\phi_\theta \end{pmatrix}^\top$. Therefore, we get the two inequalities

$$\begin{aligned} -\left(X - \left(X_\theta + \frac{d}{2}\sin\phi_\theta\right)\right)\sin\phi_\theta + \left(Y - \left(Y_\theta - \frac{d}{2}\cos\phi_\theta\right)\right)\cos\phi_\theta &\geq 0 \\ -\left(X - \left(X_\theta - \frac{d}{2}\sin\phi_\theta\right)\right)\sin\phi_\theta + \left(Y - \left(Y_\theta + \frac{d}{2}\cos\phi_\theta\right)\right)\cos\phi_\theta &\leq 0 \end{aligned}$$

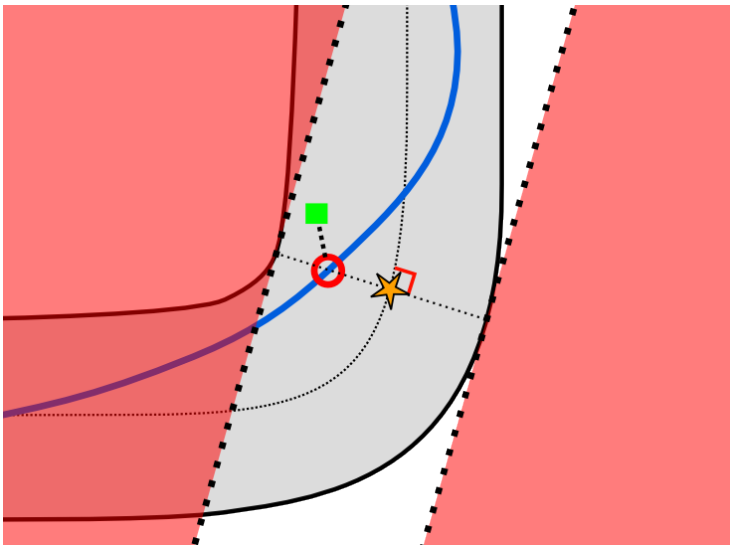


Figure 3.1: Constraints are set using the virtual state (red circle). Because the path has been defined using its center and a width, a correspondence between the trajectory and the track's center has to be determined (orange star). Then two constraints are defined for each virtual state.

using $X = X_0 + \delta X$, we get

$$\begin{aligned} -\delta X \sin\phi_\theta + \delta Y \cos\phi_\theta &\geq (X_0 - X_\theta)\sin\phi_\theta - (Y_0 - Y_\theta)\cos\phi_\theta - \frac{d}{2} \\ -\delta X \sin\phi_\theta + \delta Y \cos\phi_\theta &\leq (X_0 - X_\theta)\sin\phi_\theta - (Y_0 - Y_\theta)\cos\phi_\theta + \frac{d}{2} \end{aligned}$$

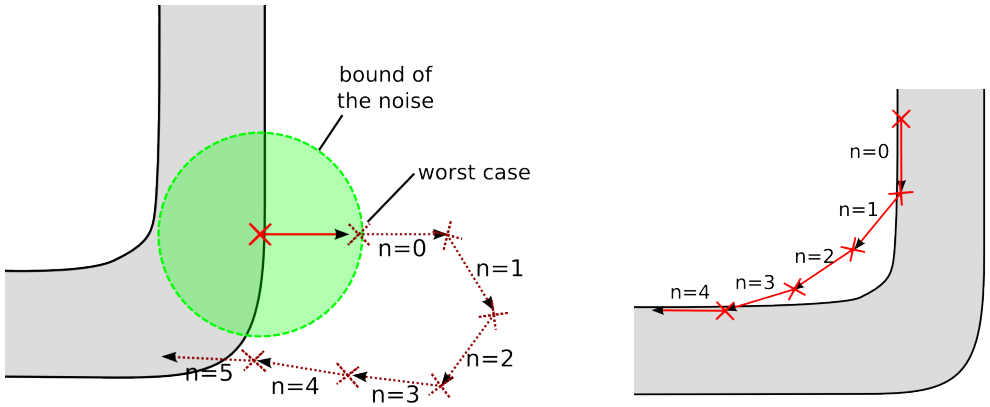
Therefore we can rewrite conditions as matrix of the form $A_{\text{ineq}}\delta\boldsymbol{X}_k \leq \boldsymbol{b}_{\text{ineq}}$:

$$\begin{pmatrix} \sin\phi_{\theta_1} & -\cos\phi_{\theta_1} & \mathbf{0}_{n_x-2} & & & (0) \\ -\sin\phi_{\theta_1} & \cos\phi_{\theta_1} & \mathbf{0}_{n_x-2} & & & \\ & & & -\sin\phi_{\theta_2} & \cos\phi_{\theta_2} & \\ & (0) & & \sin\phi_{\theta_2} & -\cos\phi_{\theta_2} & \\ & & & & & \ddots \end{pmatrix} \delta\boldsymbol{X}_k \leq$$

$$\begin{pmatrix} -\left(X_{0_{\theta_1}} - X_{\theta_1}\right)\sin\phi_{\theta_1} + \left(Y_{0_{\theta_1}} - Y_{\theta_1}\right)\cos\phi_{\theta_1} + \frac{d}{2} \\ \left(X_{0_{\theta_1}} - X_{\theta_1}\right)\sin\phi_{\theta_1} - \left(Y_{0_{\theta_1}} - Y_{\theta_1}\right)\cos\phi_{\theta_1} + \frac{d}{2} \\ -\left(X_{0_{\theta_2}} - X_{\theta_2}\right)\sin\phi_{\theta_2} + \left(Y_{0_{\theta_2}} - Y_{\theta_2}\right)\cos\phi_{\theta_2} + \frac{d}{2} \\ \left(X_{0_{\theta_2}} - X_{\theta_2}\right)\sin\phi_{\theta_2} - \left(Y_{0_{\theta_2}} - Y_{\theta_2}\right)\cos\phi_{\theta_2} + \frac{d}{2} \\ \vdots \end{pmatrix}$$

3.1.2 Model noise

If there is some noise (e.g. model noise), an important problem arises: these constraints could possibly be infeasible, especially if the noise is unbounded (e.g Gaussian noise). In the case of bounded noise, one can derive a duration $\mathcal{L}$ (in time or time-steps) after which it is always possible to be within the bounds of the trajectory and remove the $\mathcal{L}$ initial constraints. Note that these bounds depend on the model and on the shape of the road.



(a) Estimation of $\mathcal{L}$ in the case of bounded spatial noise. $\mathcal{L}$ is found by searching the case where noise imposes to violate the biggest number of initial constraints. Here $\mathcal{L} = 5$.

(b) If we remove $\mathcal{L}$ first constraints, incorrect moves can occurs.

Figure 3.2: Removing the $\mathcal{L}$ first constraints could ensure feasibility but can lead to incorrect results.

Figure 3.2a shows an example for the simple model (section 2.1). Additionally, removing the first constraints could lead to incorrect results as shown Fig. 3.2b.

Another strategy is to relax initial constraints only if infeasibility is detected in a first try. However, this requires more computing power since we have to solve twice the quadratic problem.

3.1.3 Correspondence between center of the track and planned trajectory

The correspondence can be made off-line. Given the two splines representing the planned trajectory $\mathcal{T}$ and the center of the track $\mathcal{C}$, we want to change the indices θ of the center of the track such that for each point $\mathcal{T}_\theta$ of the planned trajectory of curvilinear abscissa θ, the point $\mathcal{C}_\theta$ is the orthogonal projection of $\mathcal{T}_\theta$ on $\mathcal{C}$.

As illustrated Fig. 3.3, this is done approximately by computing for each "break" points (i.e. initial points that the spline interpolates) of the center line $\mathcal{C}$ the point on $\mathcal{T}$ which is the nearest. First, only break points are searched, then the correspondence is refined by computing a set of interpolating points between the 2 neighboring break points of the center of the track. Note that this is not perfect (no orthogonality condition is enforced), but seems to be sufficient.

3.2 On-line path planning

Since one ensures at each step that the car stays on the road, on-line path planning becomes possible. The principle is the following: given a trajectory to track, one can follow it using the same algorithm, with the constraints defined in the previous section. One can now lower the weight of the contouring error without be afraid of going out of the track, since the algorithm ensures that the car stays on it.

Since the constraints are calculated in real time, one can change them when, for example, an obstacle appears. If the current planned trajectory violates the new constraints due to the new obstacle, the algorithm will try to find a solution.

The only restriction is to remember that constraints are defined with respect to the virtual state and are half-planes. Therefore, no choice problem (e.g. a car in the middle

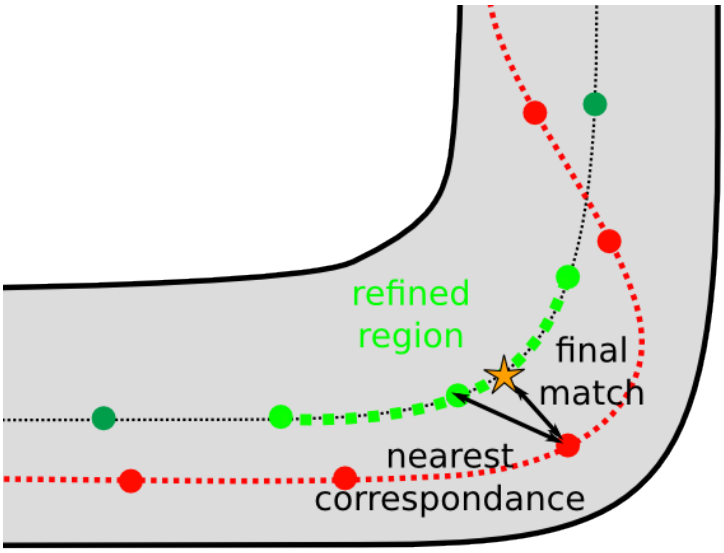


Figure 3.3: The correspondence between the car state and the constraint to stay on the track is made by using the virtual state. The center of the track is used to simplify the description of the track. For that, a correspondence computation between the curvilinear abscissa of the planned trajectory and the one of the center of the track has to be performed.

of the track, with one path to the right and one path to the left) can be handled. The simplest solution to this problem is to artificially forbid one side of the track depending of the position of the obstacle. A better solution would be to use a switch (hybrid MPC).

3.3 Off-line path planning

Since the constraints of staying on the road has been added, one can generate an optimal trajectory for the car. The center of the track is used as the reference (the track is symmetric): it ensures that maximizing the virtual speed is equivalent to minimizing the total time. Then, the horizon is increased step-by-step and the problem solved, until one reaches the end of the track. This step-by-step approach has to be performed to facilitate the convergence.

Since each step requires $\mathcal{O}(N)$ operations, the overall complexity is $\mathcal{O}(N^2)$. This can be expensive especially if the time-step is small and track is long. Some reasonable assumptions can be made to reduce the overall complexity: one can assume that the trajectory of the car is only dependent on the two next bends. Using this Markovian assumption, one can only make the horizon growing until the max distance between to bends (e.g K steps), and then shift the trajectory while keeping horizon constant. The complexity drops from $\mathcal{O}(N^2)$ to $\mathcal{O}(K \cdot N)$. If the trajectory is closed, one has to complete the trajectory several times (i.e. planning several laps) since the initial conditions are not the same.

Figure 3.4 shows the results using $\mathcal{O}(N^2)$ algorithm.

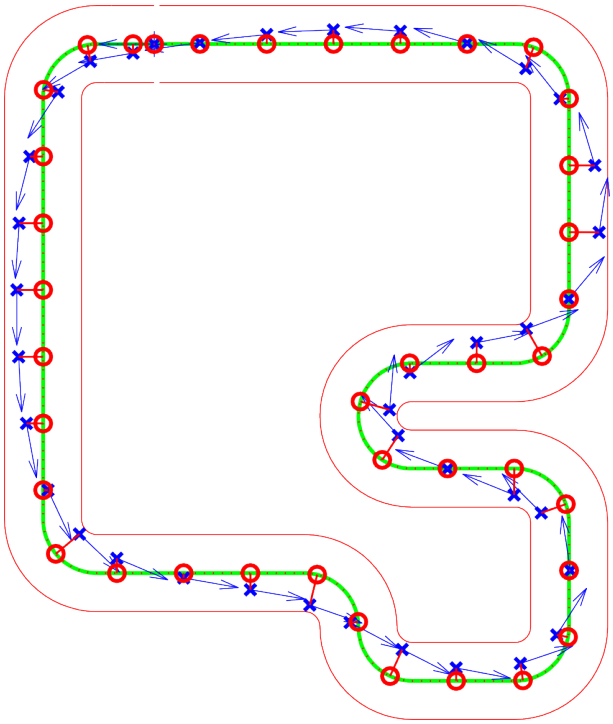


Figure 3.4: Optimal trajectory found using the simple model.

Conclusion

In this work, a model predictive contouring control applied to follow a planned track given a model of the car is demonstrated. Model predictive contouring control cannot only used to follow a given track, but also to plan a trajectory, for example for obstacle avoidance and to get an optimal path for the whole trajectory if the track is known.

Simulation were presented for two models, a simple one which enables to assess algorithm validity; the other one, more realistic allows the next step: testing the code on the real mini-car.

The code has been specifically tuned to be real-time, by rewriting part of the MATLAB code to improve its efficiency and porting some function into C code, thanks to MEX wrappers. It allows to solve the problem with horizon $N = 30$ in less than 50ms. Further improvements can be done by using C interface of CPLEX (or another more specialized solver).

In order to improve handling of a moving obstacle, a hybrid MPC problem has to be set up, to enable the choice between overtaking the obstacle right or left.

Another improvement could be done by using a more accurate model, taking in account slip and more precise car dynamics and geometry. The developped framework stays the same; only the dynamic model has to be adapted. This characteristic can be used for doing path tracking with other system than a car. One can imagine 3D path tracking for a flying drone. In this case, the geometrical errors calculations has to be derived for the 3D case.

Bibliography

[1] K. Erkorkmaz and Y. Altintas. Quintic Spline Interpolation With Minimal Feed Fluctuation. *Journal of Manufacturing Science and Engineering*, 127(2):339, 2005.

[2] F. Ferrara. Position Control of 1:43 Scale Race Cars. Semester Thesis, 2011.

[3] D. Lam, C. Manzie, and M. Good. Model predictive contouring control. *49th IEEE Conference on Decision and Control (CDC)*, pages 6137–6142, Dec. 2010.

[4] P. Spengler and C. Gammeter. Modeling of 1:43 scale race cars. ETHZ Semester Thesis, 2010.

[5] L. Wunderli. MPC based Trajectory Tracking for 1:43 scale Race Cars. ETHZ Semester Thesis, 2011.

Appendix A

Calculus of the derivative of ϕ

Using the half-angle formula $\tan\frac{1}{2}\phi = \frac{\sin\phi}{1+\cos\phi}$, one has a formula valid for all ϕ. With $\sin\phi = \frac{\partial_s y}{\rho}$ and $\cos\phi = \frac{\partial_s x}{\rho}$ where $\rho = \sqrt{(\partial_s x)^2 + (\partial_s y)^2}$, one gets the following formula:

$$\phi(\theta) = \phi(\partial_s x, \partial_s y) = 2\text{atan}\frac{\partial_s y}{\sqrt{(\partial_s x)^2 + (\partial_s y)^2} + \partial_s x} \tag{A.1}$$

The derivation of (A.1) with respect to s gives

$$\partial_s \phi = 2 \cdot \frac{\partial^2_{ss} y \cdot (\rho + \partial_s x) - \partial_s y \left(\partial^2_{ss} x + \frac{\partial_s x \partial^2_{ss} x + \partial_s y \partial^2_{ss} y}{\rho}\right)}{(\rho + \partial_s x)^2 + (\partial_s y)^2}$$

When expanding and regrouping terms in $\partial^2_{ss} x$ and $\partial^2_{ss} y$ in the numerator, one gets:

$$\partial_s \phi = 2 \cdot \frac{\partial^2_{ss} y \cdot (\rho^2 + \rho\partial_s x - \partial_s y^2) - \partial^2_{ss} x \cdot (\rho\partial_s y + \partial_s x \partial_s y)}{\rho(\rho^2 + \partial_s x^2 + 2 \cdot \rho\partial_s x + \partial_s y^2)}$$

which simplifies in

$$\partial_s \phi = \frac{\partial^2_{ss} y \cdot (\partial_s x^2 + \rho\partial_s x) - \partial^2_{ss} x \cdot \partial_s y(\rho + \partial_s x)}{\rho^2(\rho + \partial_s x)}$$

Finally,

$$\partial_s \phi = \frac{\partial_s x \partial_{ss} y - \partial_{ss} x \partial_s y}{\partial_s x^2 + \partial_s y^2}$$

Appendix B

Matrix exponential formula for discretization

One needs to calculate the following matrices

$$\begin{aligned} A_d &= \exp\left(A_c \cdot t_s\right) \\ B_d &= A_c^D \left[\exp\left(A_c \cdot t_s\right) - I\right] \cdot B_c \end{aligned}$$

For A_d, one can use MATLAB function expm to compute it. In the following, a formula to compute B_d is derived.

Using the definition of the matrix exponential, one gets

$$B_d = \sum_{n=1}^{\infty} A^{n-1} \frac{t_s^n}{n!}$$

If one calculates

$$M = \exp \begin{pmatrix} A_c \cdot t_s & B_c \cdot t_s \\ 0 & 0 \end{pmatrix} = \exp N$$

one has

$$N^n = \begin{pmatrix} A_c^n \cdot t_s^n & A_c^{n-1} B_c \\ 0 & 0 \end{pmatrix}$$

therefore

$$\begin{aligned} M &= \begin{pmatrix} \exp\left(A_c \cdot t_s\right) & A^D \left[\exp\left(A_c \cdot t_s\right) - I\right] B_c \\ 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} A_d & B_d \\ 0 & 0 \end{pmatrix} \end{aligned}$$

Therefore M can be calculated, and then A_d and B_d can be extracted.

As mentioned in section 2.2, a short experiment on MATLAB could be done to show that matrix exponential is $\mathcal{O}\left(n^3\right)$ as shown Fig. B.1.

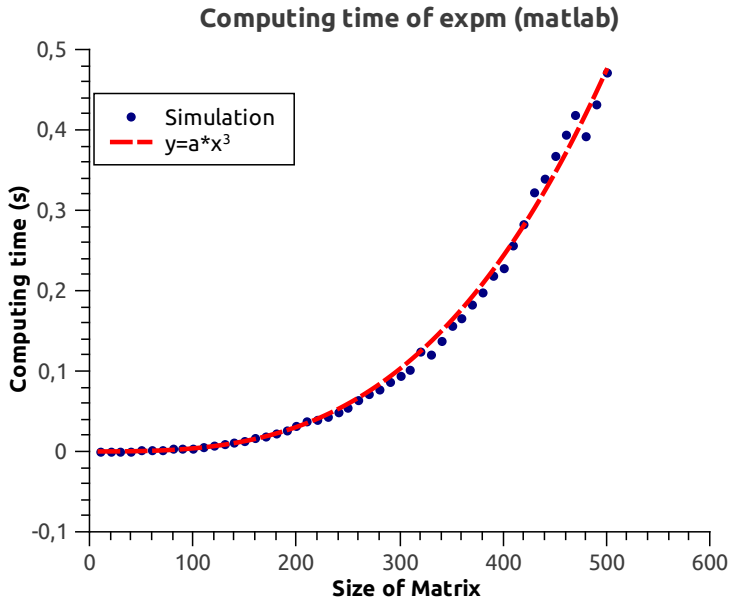


Figure B.1: Matrix exponential implementation in MATLAB is $\mathcal{O}\left(n^3\right)$.