

# Model Predictive Control

Philippe Weingertner for Polytech Nice Sophia course

August 16, 2018

## Contents

<b>1</b>	<b>Motion Control</b>	<b>2</b>
<b>2</b>	<b>Non linear optimization under constraints</b>	<b>2</b>
2.1	Definition . . . . .	2
2.2	Example . . . . .	3
2.3	Solving with ipopt . . . . .	3
<b>3</b>	<b>Basic Math utilities</b>	<b>6</b>
3.1	Translate into vehicle's coordinate system . . . . .	6
3.2	Fit a polynomial to a set of waypoints . . . . .	6
3.3	Compute the tangential angle of a polynomial curve . . . . .	7
<b>4</b>	<b>Vehicle Models</b>	<b>7</b>
4.1	Dynamic vs Kinematic Models . . . . .	7
4.2	Kinematic Model . . . . .	7
4.2.1	State . . . . .	7
4.2.2	Deriving the kinematic model . . . . .	7
4.2.3	Errors . . . . .	10
4.2.4	Actuator Constraints . . . . .	11
4.2.5	Constraints summary . . . . .	12
4.3	Dynamic Models . . . . .	12
<b>5</b>	<b>Model Predictive Control</b>	<b>12</b>
5.1	Minimization of a cost function . . . . .	13
5.2	Timsestep length and Elapsed duration . . . . .	14
5.3	Latency handling . . . . .	14
5.4	MPC Solver algorithm . . . . .	15
5.5	MPC Solver code . . . . .	17

# 1 Motion Control

Motion Control deals with the last stage of an autonomous driving pipeline: the control module. The input to the control module will be provided by the output of the path planning module via a set of waypoints to follow as close as possible. The control module will have to provide the actuators commands (in our case steering angle and throttling; acceleration or deceleration) so that the automated driving comply with a set of rules:

- follow the planned waypoints as close as possible
- drives smoothly
- try to adjust the speed: as fast as a configurable reference when possible and driving more slowly during curves

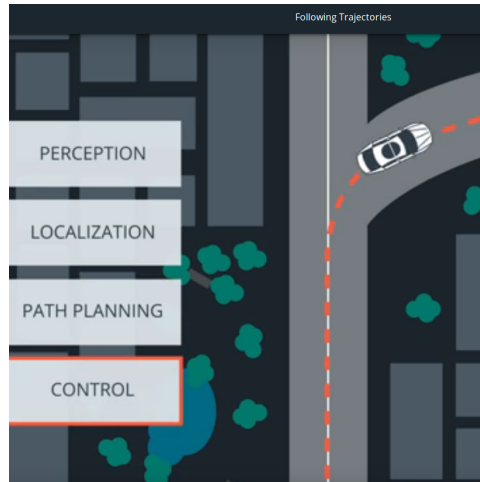


Figure 1: Autonomous Driving pipeline

## 2 Non linear optimization under constraints

### 2.1 Definition

In its most generic form we are dealing with the following problem:

$$\begin{aligned} & \underset{x}{\text{minimize}} && f_0(x) \\ & \text{subject to} && \text{lower}_i \leq f_i(x) \leq \text{upper}_i, \ i = 1, \dots, m. \end{aligned}$$

Note that by setting  $lower_i = upper_i$  we can define constraints as equalities as well.

## 2.2 Example

$$\begin{aligned} &\text{minimize} && x_1 * x_4 * (x_1 + x_2 + x_3) + x_3 \\ &\text{subject to} && x_1 * x_2 * x_3 * x_4 \geq 25 \\ & && x_1^2 + x_2^2 + x_3^2 + x_4^2 = 40 \\ & && 1 \leq x_1, x_2, x_3, x_4 \leq 5 \end{aligned}$$

## 2.3 Solving with ipopt

ipopt and cppad are used to solve non-linear minimization problems. ipopt requires the computation of first order (Jacobians) and 2nd order derivatives (Hessians). These derivatives will be computed automatically thanks to cppad: providing automatic differentiation services.

The previous example is solved with ipopt and CppAD here: [https://www.coin-or.org/CppAD/Doc/ipopt\\_solve\\_get\\_started.cpp.htm](https://www.coin-or.org/CppAD/Doc/ipopt_solve_get_started.cpp.htm)

---

Listing 1: Simple example with ipopt

---

```

1
2
3 # include <cppad/ipopt/solve.hpp>
4
5 namespace {
6     using CppAD::AD;
7
8     class FG_eval {
9     public:
10         typedef CPPAD_TESTVECTOR( AD<double> ) ADvector;
11         void operator()(ADvector& fg, const ADvector& x)
12         {
13             assert( fg.size() == 3 );
14             assert( x.size() == 4 );
15
16             // Fortran style indexing
17             AD<double> x1 = x[0];
18             AD<double> x2 = x[1];
19             AD<double> x3 = x[2];
20             AD<double> x4 = x[3];
21             // f(x)
22             fg[0] = x1 * x4 * (x1 + x2 + x3) + x3;
```

```

22             // g-1 (x)
23             fg[1] = x1 * x2 * x3 * x4;
24             // g-2 (x)
25             fg[2] = x1 * x1 + x2 * x2 + x3 * x3 + x4 * x4;
26             //
27             return;
28         }
29     };
30 }
31
32 bool get_started(void)
33 {
34     bool ok = true;
35     size_t i;
36     typedef CPPAD_TESTVECTOR( double ) Dvector;
37     // number of independent variables (domain dimension for f and g)
38     size_t nx = 4;
39     // number of constraints (range dimension for g)
40     size_t ng = 2;
41     // initial value of the independent variables
42     Dvector xi(nx);
43     xi[0] = 1.0;
44     xi[1] = 5.0;
45     xi[2] = 5.0;
46     xi[3] = 1.0;
47     // lower and upper limits for x
48     Dvector xl(nx), xu(nx);
49     for(i = 0; i < nx; i++)
50     {
51         xl[i] = 1.0;
52         xu[i] = 5.0;
53     }
54     // lower and upper limits for g
55     Dvector gl(ng), gu(ng);
56     gl[0] = 25.0;    gu[0] = 1.0e19;
57     gl[1] = 40.0;    gu[1] = 40.0;
58
59     // object that computes objective and constraints
60     FG_eval fg_eval;
61
62     // options

```

```

62     std::string options;
63     // turn off any printing
64     options += "Integer print_level    0\n";
65     options += "String  sb              yes\n";
66     // maximum number of iterations
67     options += "Integer max_iter       10\n";
68     // approximate accuracy in first order necessary conditions;
69     // see Mathematical Programming, Volume 106, Number 1,
70     // Pages 25–57, Equation (6)
71     options += "Numeric tol            1e-6\n";
72     // derivative testing
73     options += "String  derivative_test      second-order\n";
74     // maximum amount of random perturbation; e.g.,
75     // when evaluation finite diff
76     options += "Numeric point_perturbation_radius  0.\n";
77
78     // place to return solution
79     CppAD::ipopt::solve_result<Dvector> solution;
80
81     // solve the problem
82     CppAD::ipopt::solve<Dvector, FG_eval>(
83         options, xi, xl, xu, gl, gu, fg_eval, solution
84     );
85     //
86     // Check some of the solution values
87     //
88     ok &= solution.status == CppAD::ipopt::solve_result<Dvector>::success
89     //
90     double check_x[] = { 1.000000, 4.743000, 3.82115, 1.379408 };
91     double check_zl[] = { 1.087871, 0.,          0.,
0.         };
92     double check_zu[] = { 0.,          0.,          0.,
0.         };
93     double rel_tol    = 1e-6; // relative tolerance
94     double abs_tol    = 1e-6; // absolute tolerance
95     for(i = 0; i < nx; i++)
96     {
97         ok &= CppAD::NearEqual(
98             check_x[i], solution.x[i], rel_tol, abs_tol
99         );
100         ok &= CppAD::NearEqual(

```

```

100             check_zl[i], solution.zl[i], rel_tol, abs_tol
101         );
102         ok &= CppAD::NearEqual(
103             check_zu[i], solution.zu[i], rel_tol, abs_tol
104         );
105     }
106
107     return ok;
108 }

```

---

### 3 Basic Math utilities

#### 3.1 Translate into vehicle's coordinate system

To make the computations more simple we will work in vehicle's coordinate system at time  $t$ : so we have to operate  $Rot(-\psi_t) \circ Translation(\begin{bmatrix} x_t \\ y_t \end{bmatrix})$  on current world's coordinates to translate them into vehicle's coordinates

As a reminder, for a rotation matrix of angle  $\theta$ , we have  $Rot(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$ .

In our case we are dealing with  $Rot(-\psi_t) = \begin{bmatrix} \cos(-\psi_t) & -\sin(-\psi_t) \\ \sin(-\psi_t) & \cos(-\psi_t) \end{bmatrix}$

So we have:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(-\psi_t) & -\sin(-\psi_t) \\ \sin(-\psi_t) & \cos(-\psi_t) \end{bmatrix} \begin{bmatrix} x - x_t \\ y - y_t \end{bmatrix}$$

#### 3.2 Fit a polynomial to a set of waypoints

Let's assume the reference trajectory provided by the Path Planning module is a set of waypoints approximated or fitted by a (3rd order typically) polynomial  $f(x_t) = a_0 + a_1 x_t + a_2 x_t^2 + a_3 x_t^3$  and our real position on the y-axis is  $y_t$

TODO: pseudo inverse, QR decomposition, sample C++ code

### 3.3 Compute the tangential angle of a polynomial curve

[https://en.wikipedia.org/wiki/Tangential\\_angle](https://en.wikipedia.org/wiki/Tangential_angle)

## 4 Vehicle Models

### 4.1 Dynamic vs Kinematic Models

### 4.2 Kinematic Model

#### 4.2.1 State

The state variables are the following:

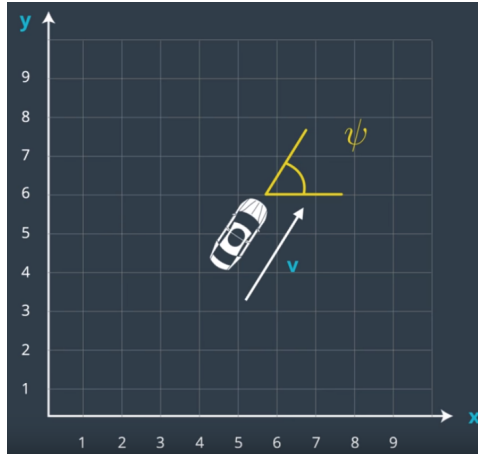


Figure 2: Model state

- $x$ : x position
- $y$ : y position
- $\psi$ : angle between speed vector and x-axis
- $v$ : speed vector

#### 4.2.2 Deriving the kinematic model

Our state vector is

$$S_t = [x_t, y_t, \psi_t, v_t]$$

We derive an approximation model, kinematic, relating  $S_{t+1}$  and  $S_t$ . The smaller the  $dt$  the more accurate the model.

The model used is the kinematic bicycle model:

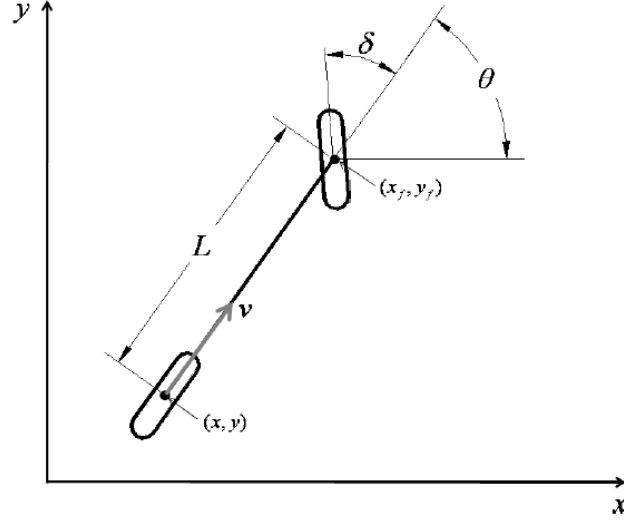


Figure 3: Kinematic bicycle model

**Linear movement approximation:** assuming during  $dt$  that  $v_t$  and  $\psi_t$  are constant:

$$x_{t+1} = x_t + v_t * \cos(\psi_t) * dt$$

$$y_{t+1} = y_t + v_t * \sin(\psi_t) * dt$$

**Rotational movement approximation:** assuming during  $dt$  that  $v_t$  and steering angle  $\delta_t$  are constant:

Let first compute the radius of curvature  $R$  based on the steering angle  $\delta$  and the distance  $L$  between rear and front wheels:



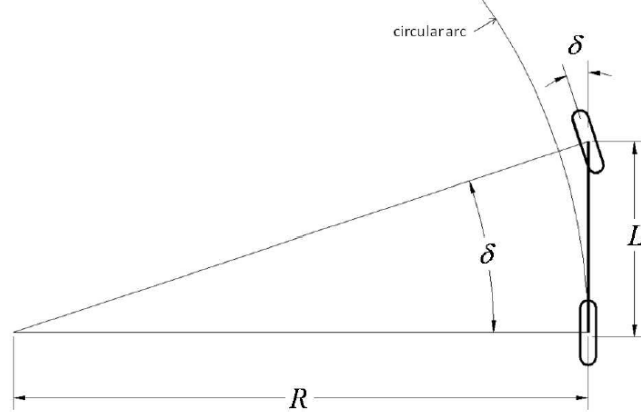


Figure 4: Radius of curvature R

We have

$$\tan(\delta) \approx L/R$$

We are moving on a circle from position  $M_t$  to position  $M_{t+1}$  at a constant speed  $v_t$

$$M_{t+1}M_t = R * (\psi_{t+1} - \psi_t) \approx v_t * dt$$

So we have:

$$\psi_{t+1} = \psi_t + (v_t/R) * dt$$

$$\psi_{t+1} = \psi_t + (v_t/L) * \tan(\delta_t) * dt$$

*Note that for small  $\delta_t$  we have  $\tan(\delta_t) \approx \delta_t$*

**Speed update:** assuming during  $dt$  that  $a_t$  is constant:

$$v_{t+1} = v_t + a_t * dt$$

So to summarize our kinematic bicycle model which is a **kinematic constraint model** is:

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

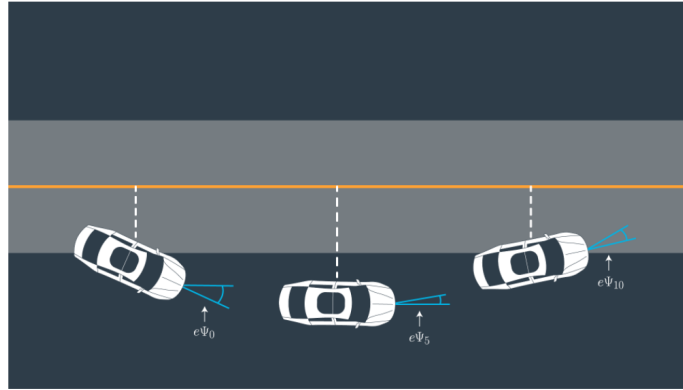
The state vector is  $S_t = [x_t, y_t, \psi_t, v_t]$ .

The actuator command  $A_t = [a_t, \delta_t]$  defines a **constraint** between  $S_{t+1}$  and  $S_t$ .

Note also that for a bicycle we have  $\delta_t \in [-\pi/2, \pi/2]$  whereas for a car we have  $\delta_t \in (-\delta_{max}, \delta_{max})$  with  $\delta_{max} < \pi/2$ .

### 4.2.3 Errors

The errors variables are the following:



The dashed white line is the cross track error.

Figure 5: Model errors

- *cte*: cross track error. It corresponds to distance of vehicle from the planned trajectory (as planned by path planning module)
- *eΨ*: psie error is the angle difference of the vehicle trajectory with the planned trajectory (as planned by path planning module)

**The new state vector is  $[x_t, y_t, \psi_t, v_t, cte_t, e\Psi_t]$ .**

Let's assume the reference trajectory provided by the Path Planning module is a set of waypoints approximated or fitted by a (3rd order typically) polynomial  $f(x_t) = a_0 + a_1x_t + a_2x_t^2 + a_3x_t^3$  and our real position on the y-axis is  $y_t$

**Compute current Cross Track Error:**

The current Cross Track Error is:

$$cte_t = y_t - f(x_t)$$

Note: we are dealing with a trajectory which is a set of  $(x, y, t)$  or  $(x_t, y_t)$  coordinates. We are not dealing with a path which a set of  $(x, y)$  coordinates.

**Compute current Orientation Error:**

The current Orientation Error is:

$$e\Psi_t = \psi_t - \text{tangential angle of curve of } f \text{ at } x_t = \psi_t - \arctan(f'(x_t))$$

**Update Cross Track Error:**

The Cross Track Error will increase during  $dt$  proportionnaly to the speed and Orientation error (along local y-axis):

$$cte_{t+1} = cte_t + v_t * \sin(e\Psi_t) * dt$$

**Update Orientation Error:**

The update rule for  $e\Psi_t$  is the same as for  $\psi_t$ :

$$e\Psi_{t+1} = e\Psi_t + (v_t/L) * \tan(\delta_t) * dt$$

So to summarize our kinematic bicycle error model which is a **kinematic error model constraint** is:

$$\begin{aligned} cte_{t+1} &= cte_t + v_t * \sin(e\Psi_t) * dt \\ e\Psi_{t+1} &= e\Psi_t + (v_t/L) * \tan(\delta_t) * dt \end{aligned}$$

#### 4.2.4 Actuator Constraints

In a real vehicle, actuators are limited by the design of the vehicle and fundamental physics. For example, a vehicle can't have a steering angle of 90 degrees. It is impossible. Thus it does not make sense for us to even consider these kinds of inputs. There is actually a vocabulary for describing this constraint. We call this model **nonholonomic** because the vehicle can't move in any arbitrary directions. It's limited by steering angle constraints. We can solve this by setting lower and upper bounds for the actuators.

We have such **actuator constraints** as:

$$\begin{array}{c} -30^\circ \leq \delta_t \leq +30^\circ \\ -1 \leq a_t \leq +1 \end{array}$$

where  $-1$  is full brake and  $+1$  is full acceleration.

#### 4.2.5 Constraints summary

We have now specified a set of constraints accounting for vehicle constraints. To summarize our kinematic model constraints are:

$$\begin{array}{l} x_{t+1} = x_t + v_t * \cos(\psi_t) * dt \\ y_{t+1} = y_t + v_t * \sin(\psi_t) * dt \\ \psi_{t+1} = \psi_t + (v_t/L) * \tan(\delta_t) * dt \\ v_{t+1} = v_t + a_t * dt \\ cte_{t+1} = cte_t + v_t * \sin(e\Psi_t) * dt \\ e\Psi_{t+1} = e\Psi_t + (v_t/L) * \tan(\delta_t) * dt \end{array}$$

And our actuator constraints are:

$$\begin{array}{c} -30^\circ \leq \delta_t \leq +30^\circ \\ -1 \leq a_t \leq +1 \end{array}$$

So far these constraints are defined for a 1-step duration of  $dt$  milliseconds. But we can rollout these constraints over  $N$ -steps. Typically we may use  $N = 10$  and  $dt = 20$  ms.

### 4.3 Dynamic Models

Forces, Slip Angle, Slip ratio and Tire Models

## 5 Model Predictive Control

MPC reframes the task of following a trajectory as an optimization problem. The solution to the optimization problem is the optimal trajectory.

MPC involves simulating different actuator inputs, predicting the resulting trajectory and minimizing a set of constraints .

Once we found the lowest cost trajectory, we implement the very first set of actuation commands. Then we throw away the rest of the trajectory we calculated. Instead of using the old trajectory we predicted, we take our new state and use that to calculate a new optimal trajectory. In that sense, we are constantly calculating inputs over a future horizon. That's why this approach is also called Receding Horizon Control. We constantly reevaluate the trajectory because our vehicle model is not perfect and the next predicted (or planned) state may (slightly...) differ with our prediction (in the sense of a consequence of a command sent).

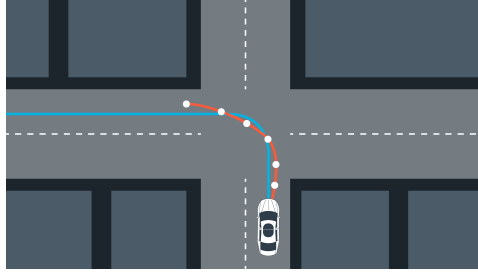


Figure 6: Minimization problem

### 5.1 Minimization of a cost function

We typically want to minimize things related to Tracking Error and comfort:

- Cross Track Error
- Orientation Error
- speed delta vs a target reference speed
- use of actuators (you prefer to use low values of  $a_t$  and  $\delta_t$  as long as you can remain on track)
- limit the rate of change of actuator commands from one timestep to another

To account for these different objectives, we define different sub cost functions and weight them according to our priorities.

Listing 2: Cost function example

---

```

1  for (size_t t = 0; t < N; t++) {
2      fg[0] += 4 * 2000 * CppAD::pow(vars[cte_start + t], 2);
3      fg[0] += 4 * 2000 * CppAD::pow(vars[epsi_start + t], 2);
4      fg[0] += CppAD::pow(vars[v_start + t] - ref_v, 2);
5  }
6
7  // Minimize the use of actuators.
8  for (size_t t = 0; t < N - 1; t++) {
9      fg[0] += 5 * CppAD::pow(vars[delta_start + t], 2);
10     fg[0] += 5 * CppAD::pow(vars[a_start + t], 2);
11 }
12
13 // smooth
14 for (size_t t = 0; t < N - 2; t++) {
15     fg[0] += 200 * CppAD::pow(vars[delta_start + t + 1] - vars[delta_start + t], 2);
16     fg[0] += 10 * CppAD::pow(vars[a_start + t + 1] - vars[a_start + t], 2);
17 }

```

---

## 5.2 Timestep length and Elapsed duration

$N=10$  and  $dt=100$  ms are used so that we are working on 1 second of data. This is a trade-off: we need enough data visibility to ensure a good prediction, but we also have to limit the amount of computation. In general, smaller  $dt$  gives better accuracy, but that will require higher  $N$  for given horizon ( $N*dt$ ). However, increasing  $N$  will result in longer computational time which increases the latency. The most common choice of values is  $N=10$  and  $dt=0.1$  but anything between  $N=20$ ,  $dt=0.05$  should work.

## 5.3 Latency handling

A contributing factor to latency is actuator dynamics. For example the time elapsed between when you command a steering angle to when that angle is actually achieved. This could easily be modeled by a simple dynamic system and incorporated into the vehicle model. One approach would be running a simulation using the vehicle model starting from the current state for the duration of the latency. The resulting state from the simulation is the new initial state for MPC.

Thus, MPC can deal with latency much more effectively, by explicitly taking it into account, than a PID controller.

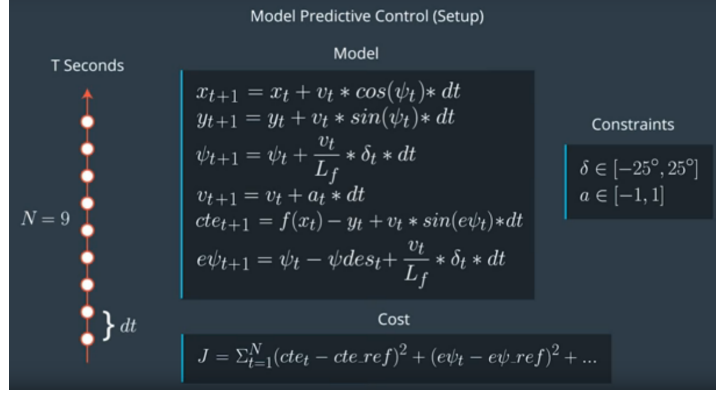


Figure 7: Solver setup with  $N*dt$  time horizon

#### 5.4 MPC Solver algorithm

To summarize **we have defined a cost function we want to minimize under a set of constraints**. The cost function accounts for what we want to produce ideally: a comfortable trajectory that follows a reference path as close as possible. Whereas the constraints account for the constraints we are dealing with: typically at this stage of the Autonomous Driving pipeline, we are dealing with vehicle model constraints. So our non-linear optimization problem is defined and can be summarized visually as:

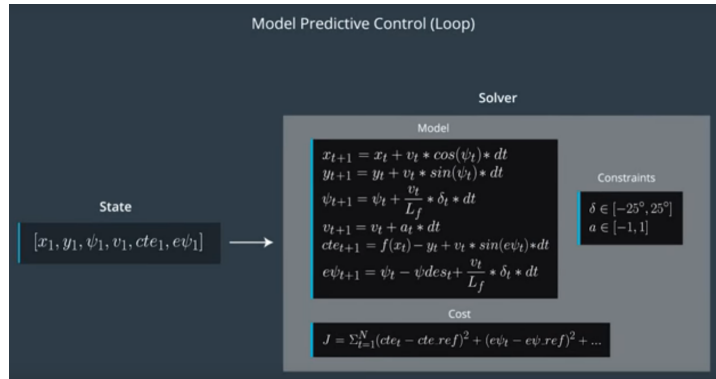


Figure 8: Solver input

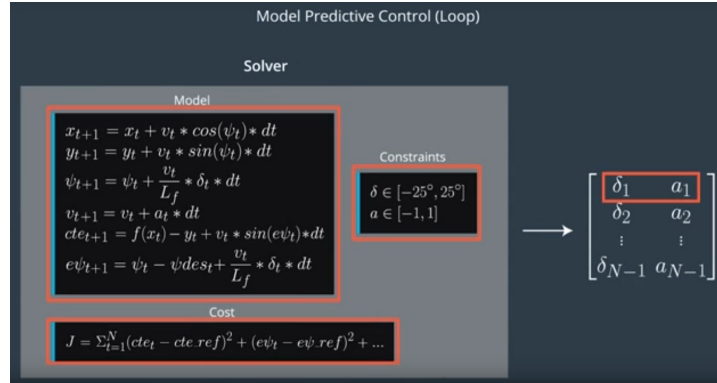


Figure 9: Solver output

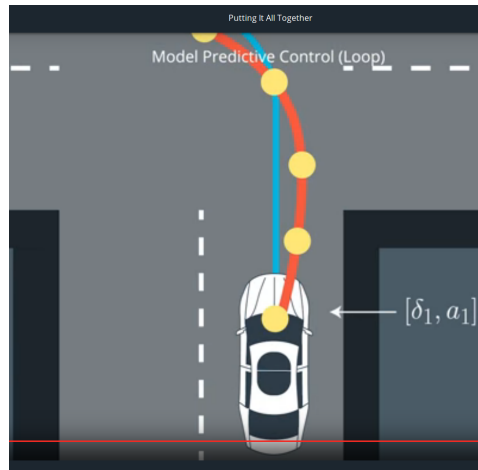


Figure 10: Solver actuator commands



The next step is to solve it in software. The model, a cost function to minimize and a set of constraints to comply with, will be described in software and a mathematical library will be used to solve this non-linear optimization problem.

## 5.5 MPC Solver code

Listing 3: MPC solver with ipopt

---

```
1 #include "MPC.h"
2 #include <cppad/cppad.hpp>
3 #include <cppad/ipopt/solve.hpp>
4 #include "Eigen-3.3/Eigen/Core"
5 #include "Eigen-3.3/Eigen/QR"
6
7 using CppAD::AD;
8
9 // TODO: Set the timestep length and duration
10 size_t N = 10;
11 double dt = 0.1;
12
13 // This value assumes the model presented in the classroom is used.
14 //
15 // It was obtained by measuring the radius formed by running the vehicle in
16 // simulator around in a circle with a constant steering angle and velocity
17 // flat terrain.
18 //
19 // Lf was tuned until the the radius formed by the simulating the model
20 // presented in the classroom matched the previous radius.
21 //
22 // This is the length from front to CoG that has a similar radius.
23 const double Lf = 2.67;
24
25 // NOTE: feel free to play around with this
26 // or do something completely different
27
28 double ref_v = 120;
29
30 // The solver takes all the state variables and actuator
31 // variables in a singular vector. Thus, we should to establish
```

```

32 // when one variable starts and another ends to make our lives easier.
33 size_t x_start = 0;
34 size_t y_start = x_start + N;
35 size_t psi_start = y_start + N;
36 size_t v_start = psi_start + N;
37 size_t cte_start = v_start + N;
38 size_t epsi_start = cte_start + N;
39 size_t delta_start = epsi_start + N;
40 size_t a_start = delta_start + N - 1;
41
42 class FG_eval {
43 public:
44     // Fitted polynomial coefficients
45     Eigen::VectorXd coeffs;
46     FG_eval(Eigen::VectorXd coeffs) { this->coeffs = coeffs; }
47
48     typedef CPPAD_TESTVECTOR(AD<double>) ADvector;
49     void operator()(ADvector& fg, const ADvector& vars) {
50         // TODO: implement MPC
51         // 'fg' a vector of the cost constraints, 'vars' is a vector of variab
52         // NOTE: You'll probably go back and forth between this function and
53         // the Solver function below.
54
55         // The cost is stored is the first element of 'fg'.
56         // Any additions to the cost should be added to 'fg[0]'.
57         fg[0] = 0;
58
59         // Reference State Cost
60         // TODO: Define the cost related the reference state and
61         // any anything you think may be beneficial.
62         for (size_t t = 0; t < N; t++) {
63             fg[0] += 4 * 2000 * CppAD::pow(vars[cte_start + t], 2);
64             fg[0] += 4 * 2000 * CppAD::pow(vars[epsi_start + t], 2);
65             fg[0] += CppAD::pow(vars[v_start + t] - ref_v, 2);
66         }
67
68         // Minimize the use of actuators.
69         for (size_t t = 0; t < N - 1; t++) {
70             fg[0] += 5 * CppAD::pow(vars[delta_start + t], 2);
71             fg[0] += 5 * CppAD::pow(vars[a_start + t], 2);

```

```

72     }
73
74     // smooth
75     for (size_t t = 0; t < N - 2; t++) {
76         fg[0] += 200 * CppAD::pow(vars[delta_start + t + 1] - vars[delta_start + t], 2);
77         fg[0] += 10 * CppAD::pow(vars[a_start + t + 1] - vars[a_start + t], 2);
78     }
79
80     //
81     // Setup Constraints
82     //
83     // NOTE: In this section you'll setup the model constraints.
84
85     // Initial constraints
86     //
87     // We add 1 to each of the starting indices due to cost being located at
88     // index 0 of 'fg'.
89     // This bumps up the position of all the other values.
90     fg[1 + x_start] = vars[x_start];
91     fg[1 + y_start] = vars[y_start];
92     fg[1 + psi_start] = vars[psi_start];
93     fg[1 + v_start] = vars[v_start];
94     fg[1 + cte_start] = vars[cte_start];
95     fg[1 + epsi_start] = vars[epsi_start];
96
97     // The rest of the constraints
98     for (size_t t = 1; t < N; t++) {
99         // at time t+1
100         AD<double> x1 = vars[x_start + t];
101         AD<double> y1 = vars[y_start + t];
102         AD<double> psi1 = vars[psi_start + t];
103         AD<double> v1 = vars[v_start + t];
104         AD<double> cte1 = vars[cte_start + t];
105         AD<double> epsi1 = vars[epsi_start + t];
106
107         // at time t
108         AD<double> x0 = vars[x_start + t - 1];
109         AD<double> y0 = vars[y_start + t - 1];
110         AD<double> psi0 = vars[psi_start + t - 1];
111         AD<double> v0 = vars[v_start + t - 1];

```

```

112     AD<double> cte0 = vars[cte_start + t - 1];
113     AD<double> epsi0 = vars[epsi_start + t - 1];
114
115     AD<double> delta0 = vars[delta_start + t - 1];
116     AD<double> a0 = vars[a_start + t - 1];
117
118     // XXX: to be updated
119     AD<double> f0 = coeffs[0] + coeffs[1] * x0 + coeffs[2] * x0 * x0 + c
120     AD<double> psides0 = CppAD::atan(coeffs[1] + 2 * coeffs[2] * x0 + 3 *
121
122     // Here's 'x' to get you started.
123     // The idea here is to constraint this value to be 0.
124     //
125     // NOTE: The use of 'AD<double>' and use of 'CppAD'!
126     // This is also CppAD can compute derivatives and pass
127     // these to the solver.
128
129     // TODO: Setup the rest of the model constraints
130     fg[1 + x_start + t] = x1 - (x0 + v0 * CppAD::cos(psi0) * dt);
131     fg[1 + y_start + t] = y1 - (y0 + v0 * CppAD::sin(psi0) * dt);
132     fg[1 + psi_start + t] = psi1 - (psi0 - v0 * delta0 / Lf * dt); // XXX
133     fg[1 + v_start + t] = v1 - (v0 + a0 * dt);
134
135     // BUG fg[1 + cte_start + t] = cte1 - (cte0 + v0 * CppAD::sin(epsi0)
136     // BUG fg[1 + epsi_start + t] = epsi1 - (epsi0 + v0 * delta0 / Lf *
137     fg[1 + cte_start + t] = cte1 - ((f0 - y0) + (v0 * CppAD::sin(epsi0)
138     fg[1 + epsi_start + t] = epsi1 - ((psi0 - psides0) - v0 * delta0 / L
139 }
140 }
141 };
142
143 //
144 // MPC class definition implementation.
145 //
146 MPC::MPC() {}
147 MPC::~MPC() {}
148
149 vector<double> MPC::Solve(Eigen::VectorXd state, Eigen::VectorXd coeffs) {
150     bool ok = true;
151     size_t i;

```

```

152     typedef CPPAD_TESTVECTOR(double) Dvector;
153
154     double x = state[0];
155     double y = state[1];
156     double psi = state[2];
157     double v = state[3];
158     double cte = state[4];
159     double epsi = state[5];
160
161     // TODO: Set the number of model variables (includes both states and inputs)
162     // For example: If the state is a 4 element vector, the actuators is a 2
163     // element vector and there are 10 timesteps. The number of variables is
164     //
165     // 4 * 10 + 2 * 9
166     size_t n_vars = N * 6 + (N - 1) * 2;
167     // TODO: Set the number of constraints
168     size_t n_constraints = N * 6;
169
170     // Initial value of the independent variables.
171     // Should be 0 except for the initial values.
172     Dvector vars(n_vars);
173     for (i = 0; i < n_vars; i++) {
174         vars[i] = 0.0;
175     }
176     // Set the initial variable values
177     vars[x_start] = x;
178     vars[y_start] = y;
179     vars[psi_start] = psi;
180     vars[v_start] = v;
181     vars[cte_start] = cte;
182     vars[epsi_start] = epsi;
183
184     // Lower and upper limits for x
185     Dvector vars_lowerbound(n_vars);
186     Dvector vars_upperbound(n_vars);
187     // TODO: Set lower and upper limits for variables.
188     // Set all non-actuators upper and lowerlimits
189     // to the max negative and positive values.
190     for (i = 0; i < delta_start; i++) {
191         vars_lowerbound[i] = -1.0e19;

```

```

192     vars_upperbound[i] = 1.0e19;
193 }
194
195 // The upper and lower limits of delta are set to -25 and 25
196 // degrees (values in radians).
197 // NOTE: Feel free to change this to something else.
198 for (i = delta_start; i < a_start; i++) {
199     vars_lowerbound[i] = -0.436332 * Lf; // *Lf ? XXX
200     vars_upperbound[i] = 0.436332 * Lf; // *Lf ?
201 }
202
203 // Acceleration/decceleration upper and lower limits.
204 // NOTE: Feel free to change this to something else.
205 for (i = a_start; i < n_vars; i++) {
206     vars_lowerbound[i] = -1.0;
207     vars_upperbound[i] = 1.0;
208 }
209
210
211 // Lower and upper limits for the constraints
212 // Should be 0 besides initial state.
213 Dvector constraints_lowerbound(n_constraints);
214 Dvector constraints_upperbound(n_constraints);
215 for (i = 0; i < n_constraints; i++) {
216     constraints_lowerbound[i] = 0;
217     constraints_upperbound[i] = 0;
218 }
219 constraints_lowerbound[x_start] = x;
220 constraints_lowerbound[y_start] = y;
221 constraints_lowerbound[psi_start] = psi;
222 constraints_lowerbound[v_start] = v;
223 constraints_lowerbound[cte_start] = cte;
224 constraints_lowerbound[epsi_start] = epsi;
225
226 constraints_upperbound[x_start] = x;
227 constraints_upperbound[y_start] = y;
228 constraints_upperbound[psi_start] = psi;
229 constraints_upperbound[v_start] = v;
230 constraints_upperbound[cte_start] = cte;
231 constraints_upperbound[epsi_start] = epsi;

```

```

232
233 // Object that computes objective and constraints
234 FG_eval fg_eval(coeffs);
235
236 //
237 // NOTE: You don't have to worry about these options
238 //
239 // options for IPOPT solver
240 std::string options;
241 // Uncomment this if you'd like more print information
242 options += "Integer print_level 0\n";
243 // NOTE: Setting sparse to true allows the solver to take advantage
244 // of sparse routines, this makes the computation MUCH FASTER. If you
245 // can uncomment 1 of these and see if it makes a difference or not but
246 // if you uncomment both the computation time should go up in orders of
247 // magnitude.
248 options += "Sparse true forward\n";
249 options += "Sparse true reverse\n";
250 // NOTE: Currently the solver has a maximum time limit of 0.5 seconds.
251 // Change this as you see fit.
252 options += "Numeric max_cpu_time 0.5\n";
253
254 // place to return solution
255 CppAD::ipopt::solve_result<Dvector> solution;
256
257 // solve the problem
258 CppAD::ipopt::solve<Dvector, FG_eval>(
259     options, vars, vars_lowerbound, vars_upperbound, constraints_lowerbou
260     constraints_upperbound, fg_eval, solution);
261
262 //
263 // Check some of the solution values
264 ok &= solution.status == CppAD::ipopt::solve_result<Dvector>::success;
265
266 // Cost
267 auto cost = solution.obj_value;
268 //std::cout << "Cost " << cost << std::endl;
269
270 // TODO: Return the first actuator values. The variables can be accessed
271 // 'solution.x[i]'.

```

```

272  //
273  // {...} is shorthand for creating a vector, so auto x1 = {1.0,2.0}
274  // creates a 2 element double vector.
275  //return {solution.x[delta_start],    solution.x[a_start]};
276
277  vector<double> result;
278
279  result.push_back(solution.x[delta_start]);
280  result.push_back(solution.x[a_start]);
281
282  for (size_t i = 0; i < N - 1; i++) {
283      result.push_back(solution.x[x_start + i + 1]);
284      result.push_back(solution.x[y_start + i + 1]);
285  }
286
287  return result;
288 }

```

---