This page outlines how MPC works.

Main Idea

MPC is a control algorithm. Given a system state x, a target state $\hat{x}$, and a control input u, it tries to drive $x - \hat{x}$ to zero.

The way it does this is by using a model of the system to predict how the system will respond to any given control inputs, then running a big optimization with the controls as inputs. You can think of it like MPC thinking some constant time into the future, planning its controls to optimally control the system. After each optimization, we execute the first step of that plan, re-measure the system, and re-optimize.

This optimization is very expensive, and it's very important that we do this very quickly. There are two main things allowing real-time MPC to be truly feasible:

1. computers are fast and people have spent a LOT of time and money optimizing silly fortran routines to solve matrix equations
2. subsequent optimal control problems (OCPs) are very similar to each other, so the optimizer doesn't have to move the points very far each time.

Despite the immense computational load, MPC is desirable for several reasons:

- it can take into account very nonlinear dynamics
- it can deal with constraints, like "don't go faster than 10 mph"

To recap, in order to implement MPC, we need four things:

- the system model
- a cost function which incentivizes following our path
- constraints ensuring the control is feasible (ex. within the steering limits) and good (ex. not falling off the track)
- a fast nonlinear optimization algorithm

Implementation Overview

There are a couple of ways to implement MPC, but I'll just focus on the way we do it. We use a technique called Direct Multiple Shooting.

Dense versus Sparse problems

First, let's take a segway into optimization. A *dense* optimization problem is one whose jacobian has relatively few structural zeros, as opposed to a *sparse* optimization problem, whose jacobian has many structural zeros (imagine a block diagonal matrix).

Now, what does that mean?

Essentially, any given optimization problem has a certain amount of complexity - a certain amount of information. In order to solve the optimization problem, we must sort through a large portion of that information. There are multiple ways to do this:

1. relatively few optimization variables, each encoding more information (more nonlinearity)
2. relatively many optimization variables, each encoding less information (less nonlinearity)

If you've ever done nonlinear optimization, you know how hard it can be sometimes to solve the system of equations with the lagrange multipliers. Sparse systems would have many more variables, but would be easier to break down and have simpler parts. Dense systems would have fewer variables, but be a huge pain to solve. So which one would you rather do?

It depends on:

- available memory
- nonlinearity of your problem
- your optimization algorithm
- and more

generally speaking, however, if you have the memory, it's better to go sparser (huge grain of salt!)

Thus, we are going to set up our problem to be very sparse. We will have MANY optimization variables, and we will connect them all with constraints.

okok actually how is MPC going to work???

MPC looks ahead `N` steps into the future, each of size `dt`. We take our initial state and our planned controls and put them together according to our system model to figure out how the system will evolve. This gives us `N` positions the system is in. Our cost function then compares this to our desired trajectory to tell us how "off" we are.

Let's make our optimization variables. These are the numbers MPC is going to optimize. Remember, an optimization problem takes the following form:

$$\min_x f(x) \quad \text{subject to} \quad \text{lb}_g \leq g(x) \leq \text{ub}_g$$

we need to figure out all the numbers in the vector x.
At each time t_i, we'll need the state of the car, z_i, and the controls to apply, u_i. Thus, we can formulate our optimization variables as follows:

$$x = \begin{bmatrix} z_0 & u_0 \\ z_1 & u_1 \\ \vdots & \vdots \\ z_n & u_n \end{bmatrix}.\text{flatten}() = \begin{bmatrix} | & | \\ Z & U \\ | & | \end{bmatrix}.\text{flatten}()$$

(please excuse the abuse of notation for the $.\text{flatten}()$ - it's just to make x a vector.)

Now, we need constraints. not all combinations of these variables are acceptable! We need to make sure that for each time t_i, if we apply control u_i starting from state z_i, after `dt` time we'll end up at state z_{i+1}. This is

where the "model" part of model predictive control comes in. We have a model of the system, so we can just ask that model, "what happens if I start at this initial state and apply these controls?" In other words, we need to solve an initial value problem. We'll learn more about how this happens in System Model, so for now, let's just assume we have the follow function to evolve the system:

$$\text{let } F(\text{state}(t_0), \text{controls}(t_0)) = \text{state}(t_0 + \Delta t)$$

With this function, we can impose our *dynamics constraints*:

$$F(z_i, u_i) - z_{i+1} == 0 \text{ for all } i \in \{0, 1, \dots, n\}$$

We need some other constraints as well to ensure the path is actually feasible:

- steering angle must be within specified bounds (you can't turn the steering wheel 50 turns to the right)
- acceleration (driving and braking) must be in some bounds (can't acclerate or brake infinitely fast)
- steering angle can't turn infinitely fast
- forward velocity must be less than some constant
  Note that these constraints are fairly simple - they don't accurately reflect the real limits of the car. This is necessary because increasing the complexity of the cost function increases the time needed for the optimization, and MPC must run very very fast. Additionally, the path we're trying to follow has been optimized by the global trajectory optimizer, which we trust to accurately limit the trajectory to something acheivable.

cost function

Now that we know the output path will be valid, we can construct our cost function.

$$\text{let } \widehat{Z}_{4\times n} = \begin{bmatrix} | & | & \dots & | \\ \hat{z}_1 & \hat{z}_2 & \dots & \hat{z}_n \\ | & | & \dots & | \end{bmatrix} \text{ be our target trajectory}$$

$$\text{let } Z_{4\times n} = \begin{bmatrix} | & | & \dots & | \\ z_1 & z_2 & \dots & z_n \\ | & | & \dots & | \end{bmatrix} \text{ be our planned trajectory}$$

$$\text{let } E_{4\times n} = \begin{bmatrix} | & | & \dots & | \\ e_1 & e_2 & \dots & e_n \\ | & | & \dots & | \end{bmatrix} = \begin{bmatrix} | & | & \dots & | \\ (\hat{z}_1 - z_1) & (\hat{z}_2 - z_2) & \dots & (\hat{z}_n - z_n) \\ | & | & \dots & | \end{bmatrix}$$

$$\text{then let cost be defined as:}$$

$$f(Z, U) = \sum_{i=1}^{n} \left[e_i^T Q e_i + u_i^T R u_i \right] + e_n^T F e_n$$

This cost function is fairly standard. It penalizes:

- distance away from the target path at each point
- usage of the controls at each point (this helps stop jerky driving and makes it conserve battery)
  There's also an extra penalty which only applies to the final timestep of the plan. This just makes it prioritize getting to the path quickly.

If you haven't seen $x^\intercal A x$ before, it's called a quadratic form. It allows us to express weighted sums of squares in a very idiomatic way. Here's an example:

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1^2 + 6x_1x_2 + 2x_2^2$$

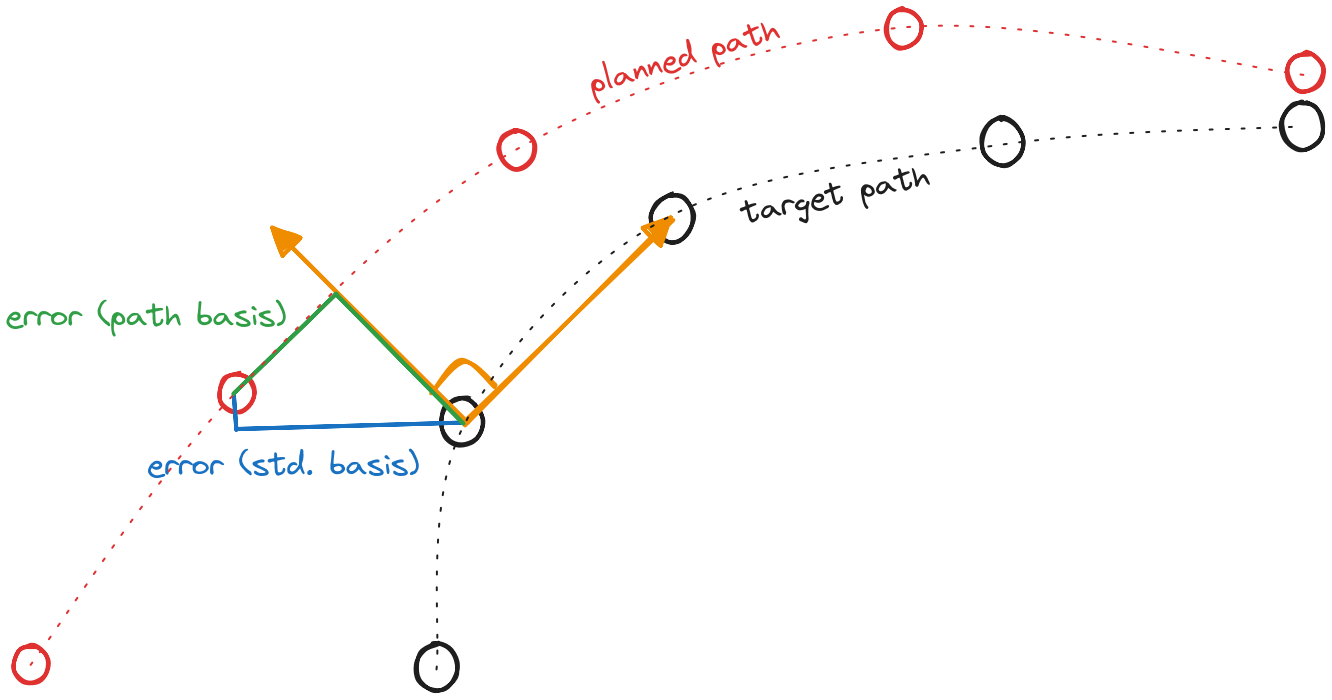
it's just a quadratic! Most of the time, A will be diagonal, since we rarely ever need the mixed terms like $6x_1x_2$. However, our current code illustrates one interesting exception to that rule.
As it turns out, I lied about the cost function. The $e_i^T Q e_i$ term, which we'll call C_p (cost_path) is slightly different. In reality, it looks like this:

$$\begin{aligned} &\text{let } y_i = Me_i \\ &C_p = y_i^T Q y_i \end{aligned}$$

We perform a cheeky little *change of variables*, introducing this new matrix M. If we plug everything in, we can see that this is like a diagonalization:

$$\begin{aligned} C_p &= y_i^T Q y_i \\ &= (Me_i)^T Q (Me_i) \\ &= e_i^T (M^T Q M) e_i \end{aligned}$$

The purpose of this M is to rotate the coordinate system so that when we construct Q as diagonal, the weights are for "error along the path" and "error orthogonal to the path", instead of just x and y error. Since M is a rotation, it is an orthogonal matrix, and thus $M^T = M^{-1}$.
Here's a diagram which may or may not help:



We want to find the error under the "path" basis for each point, not the standard basis. This allows us to differentiate between the car lagging behind the path and being off the path entirely.

System Model

Let's come back to that dynamics function F. How can we construct it?
The car is modeled using the bicycle model:

$$\text{given state } z = \begin{bmatrix} x \\ y \\ \psi \\ v \end{bmatrix} \text{ and control vector } u = \begin{bmatrix} a \\ \phi \end{bmatrix}$$

$$\text{then dynamics are given by:}$$

$$\dot{z} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos(\psi + \beta) \\ v\sin(\psi + \beta) \\ \frac{v}{l_r}\sin\beta \\ a \end{bmatrix} \qquad \text{where } \beta = \tan^{-1}\left(\frac{l_r \tan\phi}{l_f + l_r}\right)$$

Where:

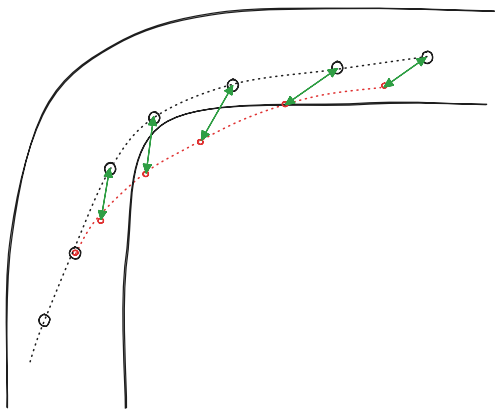
- x and y are car's location in cartesian coordinates (with the origin pinned somewhere on the ground)
- ψ is the car's heading
- v is the car's forward velocity
- a is the car's forward acceleration
- ϕ is the car's steering angle (the angle of the front tires with respect to the car's centerline)
- l_r is the distance from the rear wheels to the center of gravity
- l_f is the distance from the front wheels to the center of gravity
  This is a differential equation which governs the car's motion.
  With this differential equation, we can apply a variety of numerical integration techniques to solve the initial value problem (IVP). The current code defaults to using a single step of Euler's method per timestep, but can also use runge-kutta. Euler's method is notoriously inaccurate, but it is very fast, and very easy to take derivatives through. Since the car's movement is relatively linear on the timescale we're looking at, euler's method isn't too bad.

Formally, we define F as follows:

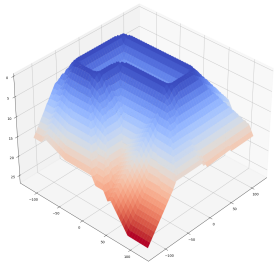
$$F(z, u) = z + \dot{z}\Delta t$$

Track Constraints

The car shouldn't drive outside of the track. Since the target trajectory does not go outside the track, this usually isn't a problem. However, there are certain situations when it might, particularly when the car is significantly far from the target trajectory (ie, at the start when it needs to accelerate and catch up).
Here's an example:

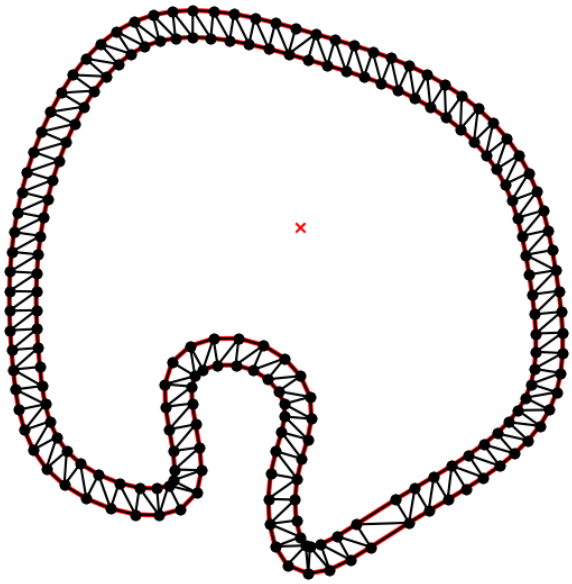


Here, the car starts too slow to keep up with the target trajectory (black), so MPC tries to cut the corner to catch up.
This is bad. We would lose points! We can't be having that.
To have a constraint that says "stay inside the track", we need a function which is greater than zero outside the track, less than zero inside the track, and ideally has smooth, simple derivatives everywhere. We want a function like this:



The L shaped top is where the track is. Notice that the big-picture shape of the function contains the correct information about which direction to travel to get back to the track.

The way we compute this is somewhat silly, but it works. The main difficulty is that the track is non-convex, and traditional point-in-polygon methods don't work because they aren't differentiable.
First, we triangulate the track using a fast library made by a CS professor here at Berkeley. Here's the result:



Notice that red `x` in the center. In order to get a proper triangulation, we need to tell the triangulation library, "hey! this void inside here is a hole! don't triangulate it!". Currently the code just uses the origin. We need a better way to do this in case the origin doesn't fall inside the track when we actually run the code on the car.

Now that it's triangulated, we just have to check:

```
for each point in path:
        for each triangle in mesh:
```

```
                if point in triangle:
                        log this point OK
```

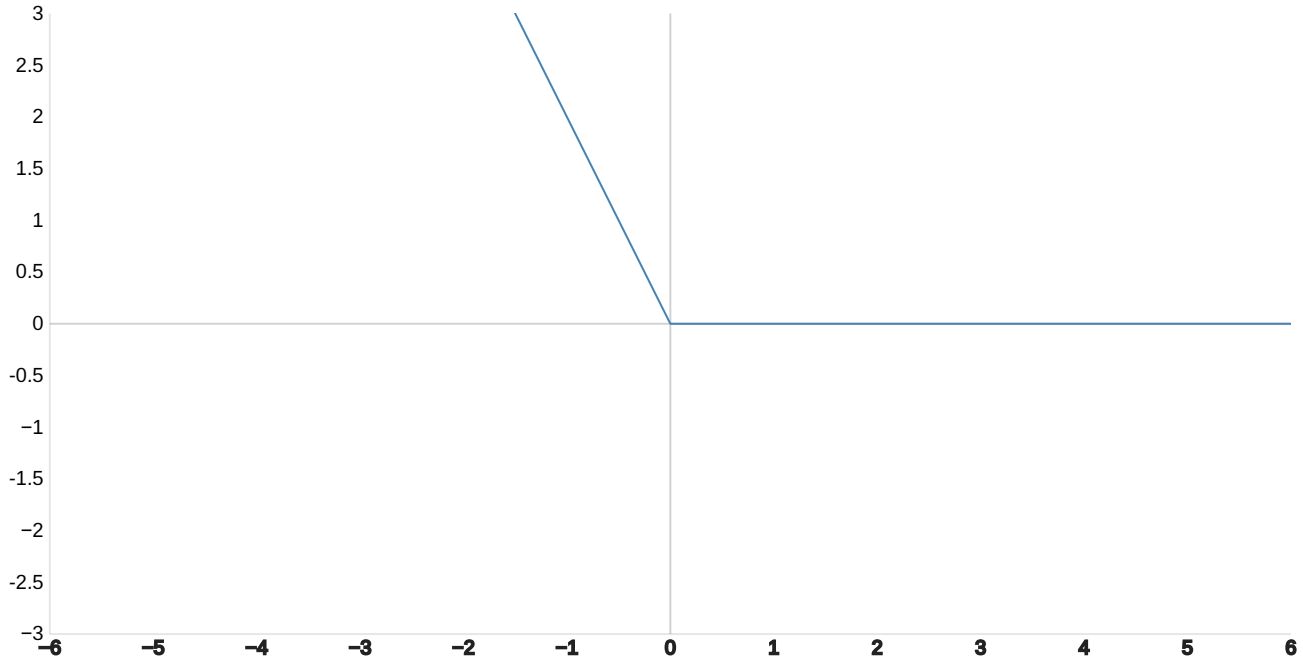
and we need to, at the end, check if all the points are OK. But first, how do we differentiably check if a point is inside a triangle?

$$\text{let } x \in \mathbb{R}^2 \text{ be our point}$$
$$\text{let } t_1, t_2, t_3 \in \mathbb{R}^2 \text{ be our triangle}$$

$$\text{if the system:}$$
$$\begin{bmatrix} t_1 & t_2 & t_3 \\ 1 & 1 & 1 \end{bmatrix} c = \begin{bmatrix} x \\ 1 \end{bmatrix}$$
$$\text{has a solution } c \text{ with all positive entries, } x \in \text{hull}\{t_1, t_2, t_3\}$$

I won't explain how this works here. If you want to learn more, use search terms "affine span", "affine hull", and "barycentric coordinates".
Since we're using triangles, we have 3 points, making our matrix invertible. Thus we can compute all of the 3×3 matrices - one for each triangle - and invert them ahead of time. All we need to do at runtime is compute the matrix multiplication, then check if there are any vectors c for which $\min(c) > 0$. But how do we do that check?

$$[x \cdot (\text{sign}\,(x) - 1)]$$



The above function is very nice. We can pass our vector c through it, making all positive values zero. Then, if we compute $\text{sum}(c)$, it will equal zero if and only if all entries were positive initially.
You might complain - smartly - that this function is not differentiable. However, we can ever so slightly round out that corner without impacting performance. Yes, maybe we'll be staying two centimeters more inside the track, but it's fine. It's close enough.

Let's put all the steps together. For each x on the path, we need to check k triangles, so we'll get k different c vectors. Let's put them all together:

$$C = [c_1 \quad \dots \quad c_k]$$

to compute our constraint, we do this:

$$\text{sum2}[C \times (\text{sign}(C) - 1)]$$

where $\text{sum2}()$ sums each column and $\times$ represents elementwise multiplication. Now we have a length-k row vector which, if x was inside the track, have precisely one element which is zero. We *could* just take the product of all the elements in this row vector, but that's painful and involves big numbers. Instead, casadi provides a nice function called `mmin`, which finds the minimum value of a matrix *differentiably*. Read more here: https://en.wikipedia.org/wiki/Smooth_maximum.
This value will be zero if we're inside the track and greater than zero - *and proportional to the closest distance to the track* - if we're outside the track. Yay!

Soft Constraints

There's one more problem to solve, though. Implementing track constraints as true constraints is very dangerous, since if the car *does* happen to go off the track, there might be a case where it's impossible to get all of the points in the planned path to satisfy the constraints. In this scenario, the solver will give an error, saying "no feasible solution found" or "interior point not found" or similar. It's like if I gave you this optimization problem:

$$\min_x \ x^2 + 2x + 5 \quad \text{subject to} \quad \begin{cases} x < 10 \\ x > 20 \end{cases}$$

Obviously, this is bad. Our solver should never return an error! We need *soft constraints*, which act like constraints but can be violated if neccessary. We implement these using slack variables.

$$\begin{gathered} \text{let } T(x) \text{ be our track constraint function} \\ \text{then our constraint is:} \\ T(x) + s < \epsilon \quad \text{for some small constant } \epsilon \end{gathered}$$

Here, s is a *slack variable*, and additional optimization variable that the solver gets to choose. You might think we just eliminated the entire point of the track constraint, since no matter what value $T(x)$ takes on, the solver could always choose some value for s that satisfies the constraint. If so, you would be right! In order to make the solver go back to obeying the constraint, we're going to heavily penalize it for using s. We can add a term to our cost function like this:

$$\text{cost} = \text{mpcCost}(Z, U) + 10,000 s^2$$

If we choose the coefficient of s^2 to be suitably high, any violation of the constraint will far outweigh the cost of the path, so the constraint will still be satisfied!

Nonlinear Optimization

Now that we have our problem all set up with variables, a cost function, and constraints, we need to solve it. There are many methods for doing this; if you've done AI work before, you might recognize some of these:

- Interior Point Methods (IPMs)
- Sequential Quadratic Programming (SQP)

- Alternating Direction Method of Multipliers (ADMM)
- Gradient Descent Methods
- Trust-Region Methods
  The first two are generally the most common for this kind of problem. Interior Point methods, also known as Primal-Dual Interior Point Methods are very popular. We are very lucky because people have written very very fast C and Fortran code implementing these algorithms for us to use. If you would like to learn more about how this works, I recommend this video on interior point methods. SQP methods are simpler; they just approximate a function as a quadratic, approximate the constraints as linear, solve, and repeat.

The two main libraries we use for nonlinear optimization are IPOPT (Interior Point OPTimizer) and WORHP (We Optimize Really Huge Problems) also known as eNLP (European NonLinear Programming solver). WORHP uses the SQP method.

Both of these, along with several other solvers, are implemented as CasADi plugins, so we can use the `nlpsol` and `Opti` stack interfaces to them.

There are several other important libraries we use; if you need help installing everything, please contact me (reid) and I'll do my best. A lot of the installation is a huge pain to do.