> **Note:** Hi there! This isn't a purpose-built writeup; it's just the docs I wrote for this algorithm. It's better than nothing, and explains it all very thoroughly. Perhaps I'll do a better writeup in the future, but this will do for now.

# global trajectory optimization docs

There are two global opt algorithms in the MPC folder, plus the grad student's code. I haven't fully looked through the grad student's code, and it's reliably slower, so I'm only going to write about the algorithms I wrote. Also, since the first one was bad and the second iteration is better in every possible way, I'll only write about that one.

to reiterate, this covers:

### `global_opt_compiled.py` (branch: `reid-mpc`)

This is the most up-to-date version, and currently the best.

# notes

ever since I added the ability for friction and drive acceleration limits to be functions, runtime has increased significantly when using IPOPT. It's still very fast when using WORHP, but I know that's a pain in the ass to obtain and install. sorry. Just know that the performance *is* better than what you're seeing.

# API

Here's how to use it. See here for how it works.

## constructor

After importing, create a new global opt object:

```
g = GlobalOptCompiled(N)
```

where `N` is the number of points to expect in the discretized path.

You can also pass some other options. Here's the full constructor:

```
GlobalOptCompiled(N,
                  nlp_solver='ipopt', # solver library. use 'worhp' if you can
                  solver_opts=None,   # options for solver (defaults are good)
```

```
                   car_params={
                       'l_r': 1.4987,  # length from CG to front axle (m)
                       'l_f': 1.5213,  # length from CG to back axle (m)
                       'm'  : 1.0      # mass (1.0 => force=acc; don't change)
                   },
                   DF_MIN  =      - 0.5, # min steering angle
                   DF_MAX  =        0.5, # max steering angle
                   ACC_MIN =      - 3.0, # max braking acceleration (negative)
                   ACC_MAX =        2.0, # max forward acceleration
                   ACC_MAX_FN = ACC_MAX  # also limit fwd acc < ACC_MAX_FN(velocity)
                   DF_DOT_MIN =   - 0.5, # max steering derivative (left) (rad/s)
                   DF_DOT_MAX =     0.5, # max steering derivative (right) (rad/s)
                   V_MIN =          0.0, # minimum forward velocity
                   V_MAX =         25.0, # maximum forward velocity
                   FRIC_MAX =      12.0) # maximum acceleration (based on tire grip)
```

`FRIC_MAX` and `ACC_MAX` should be of type `casadi.Function` with the signature `(i0)->(o0)` (ie, one input and one output). They are functions of velocity, in m/s. `FRIC_MAX` can also be passed as a float, corresponding simply to a constant function.

## code generation

To compile a solver, use this method:

```
g.construct_solver(
                generate_c=False,
                compile_c=False,
                use_c=False,
                gcc_opt_flag='-Ofast'
):
```

If `generate_c` is true, it will generate new C code for the solver, overwriting the previous code.

If `compile_c` is true, it will compile the (new or old) C code for the solver and move the binary to the `MPC/bin/` directory.

If `use_c` is true, it will import the (new or old) binary from `MPC/bin/` and use that as the solver. Otherwise it will just use the original expression graph evaluated in CasADi's virtual machines.

`gcc_opt_flag` sets the optimization level used by GCC. The default, `-Ofast`, is maximum optimization, but this can take a long time to compile (on the order of minutes) and also sometimes fails with weird cryptic C stuff about symbols not being found. If that happens, you can take it one tick down, using `-O3`, which doesn't sacrifice much performance and fixes the issue. If you're debugging and just need faster compile times, set it to `-O2` or `-O1` or even just an empty string for the fastest possible compile time.

If you already have a compiled solver and don't want to compile or construct one from scratch, you can just call the `g.load_solver()` method. This doesn't go through the effort of setting up the whole expression graph, so it's faster.

Make sure the `N` of the compiled solver and the `N` you passed to the constructor are the same!

## solving the problem

To run the solver, use `g.solve`:

```
g.solve(left, right)
```

where `left` and `right` are the left and right sides of the track. They should be of shape `(N, 2)`, where `N` is the same as the one you passed in to the constructor. These points determine the discretization of the track - the optimizer simply linearly interpolates between corresponding pairs of points. Choosing these points given a path of cones is a good deal of effort; use the `PairPath` class in `MPC/custom_opt/pairpath.py`.

this will return a dictionary with three items:

- `z` is the full state of the car. It has shape `(N, 4)`, and its format is $z = [x \quad y \quad \psi \quad v]$.
- `u` is the control inputs. It has shape `(N, 2)`, and its format is $u = [a \quad \phi]$.
- `t` is the timestamps corresponding to each element of `z` and `u`. It will always be an increasing series, but it is not evenly spaced.

## fixing the data

data with inconsistent timesteps is really quite difficult to deal with, so we need to fix that. Use the `g.to_constant_tgrid` method:

```
res = g.solve(left, right)
states, controls = g.to_constant_tgrid(dt=0.1, **res)
```

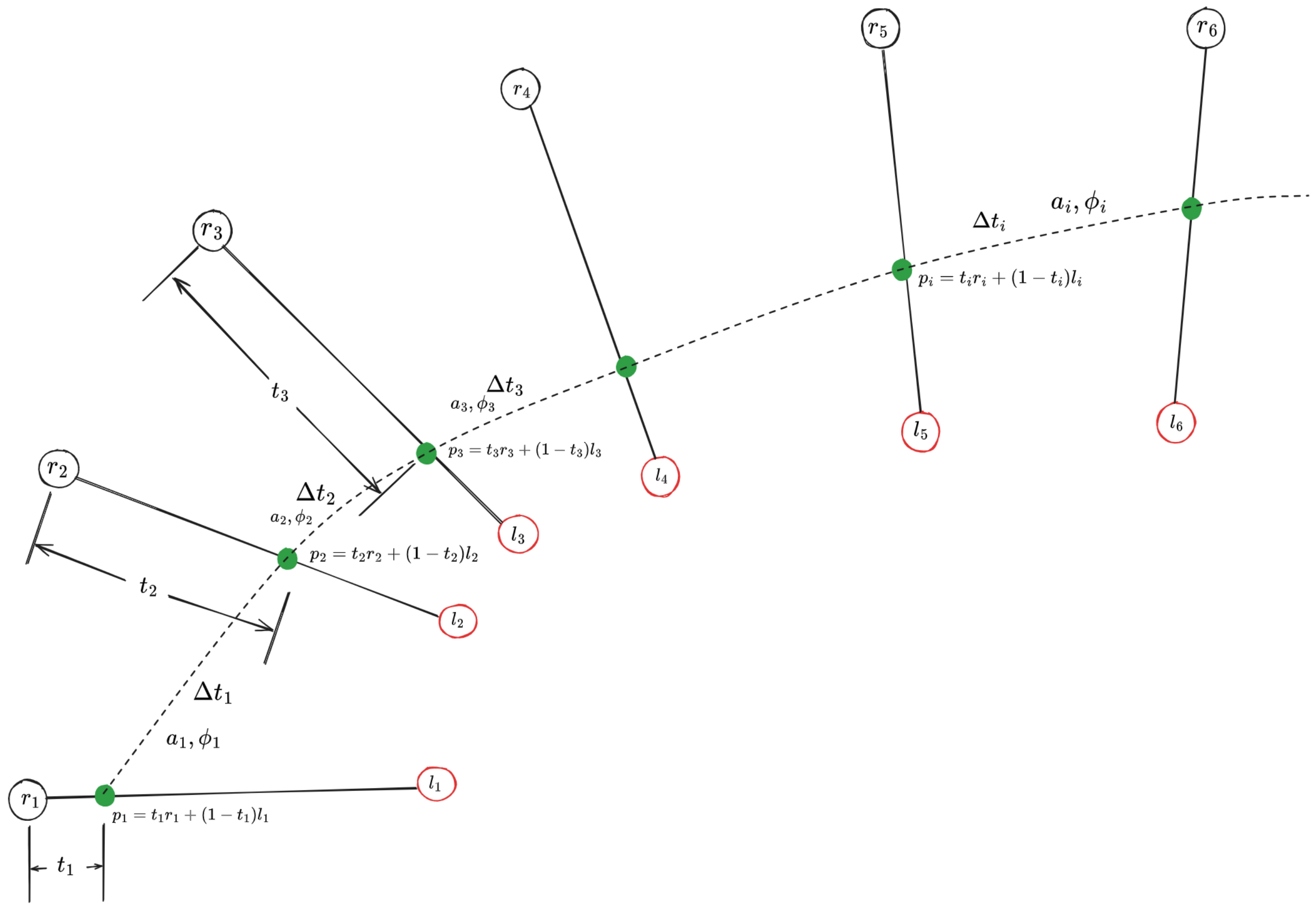
(enter whatever timestep you like for dt)

This returns two numpy arrays, here `states` and `controls`. `states` has shape `(M, 4)`, and `controls` has shape `(M, 2)`. These states and controls represent a real, feasible (to within the error of the integrator) path with even time steps of `dt` seconds.

## How it works

We're given the track in pairs of points defining the boundaries. We use this to create a parameterized path through the track in $(x, y)$ coordinates, then add the rest of the state and controls at each point. The time between points - $\Delta t$ - is also a variable.

Here's an illustration:



we're given $L_{2\times n} = [l_1 \quad l_2 \quad \dots \quad l_n]$ and $R_{2\times n} = [r_1 \quad r_2 \quad \dots \quad r_n]$. We set up the optimization problem:

$$\min_x f(x) \quad \text{subject to} \begin{cases} g_i(x) = 0 & i \in \{1, \dots, p\} \\ \text{lb}_i \le h_i(x) \le \text{ub}_i & i \in \{1, \dots, q\} \end{cases}$$

Now let's define all those variables and functions!

Let our $x$ to be all of our variables together (please excuse this abuse of notation):

$$x = \begin{bmatrix} | & | & | & | & | & | \\ t & \psi & v & a & u & \Delta t \\ | & | & | & | & | & | \end{bmatrix}.\text{flatten}()$$

and our cost function to simply be the lap time:

$$f(x) = \sum_{i=1}^{n} \Delta t$$

Now we have to make our constraint functions. First, we'll introduce the vehicle dynamics. Let $F(z(t_0), u(t_0), \Delta t) = z(t_0 + \Delta t)$ represent a function which integrates the dynamics. Then the constraints are:

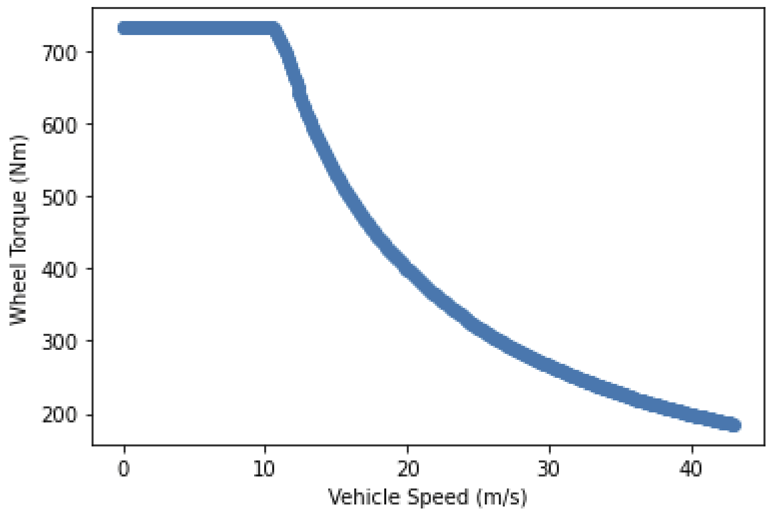
$$g_i(x) = F(z_i, u_i, \Delta t_i) - z_{i+1}$$

where $z = [x \quad y \quad \psi \quad v]$ and $u = [a \quad \phi]$.

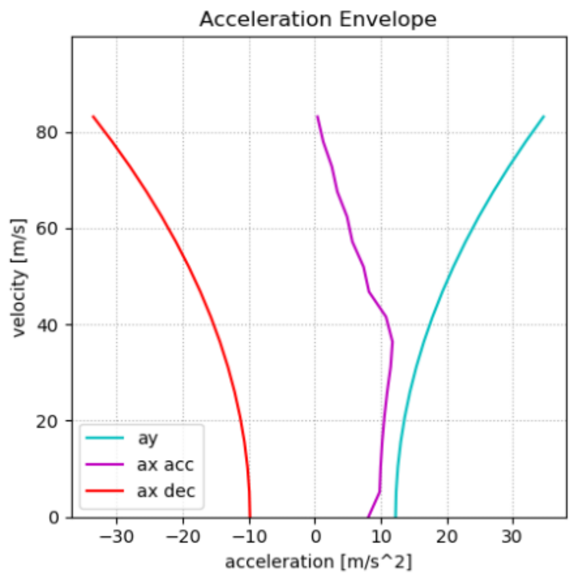
This function $F$ is implemented using the midpoint method in `discrete_custom_integrator` from `MPC/sim/sim.py`.

Now we also want to add constraints based on the car's limits:

- The car can only accelerate so fast, and that acceleration depends on the current motor speed.
  - $a_i < f(v)$ for some function $f$ determined empirically using data from dynamics:



- The car can only brake so hard.
- The steering angle is mechanically limited
- the steering wheel can only be turned so fast
- the tire friction is limited, so centripetal and driving acceleration must not exceed friction limits. These friction limits depend on velocity, since aero increases downforce.



this is an example plot from the OpenLapSim documentation. We need to gather true data from dynamics.

These are all implemented fairly simply, using direct constraints on the input variables. The last one is a bit silly:
centripetal acceleration is given by: $\frac{v^2}{r} = \frac{v^2}{\frac{v}{\dot\phi}} = \frac{v^2\dot\phi}{v} = v\dot\phi$. $v$ is an optimization variable that we use directly, but $\dot\phi$ is nontrivial to compute. It is a function of steering angle and velocity. We can take it directly from the dynamics:

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

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

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

which is where the expression for centripetal acceleration comes from:

```
ac = (self.v**2 / self.car_params['l_r']) *
sin(arctan(self.car_params['l_r']/(self.car_params['l_f'] + self.car_params['l_r'])
* tan(self.u[:, 1])))
```

then, we can combine centripetal and forward acceleration with the Pythagorean theorem, since we know they're orthogonal. This net acceleration can then be constrained to be less than some function of velocity.

## implementation

it’s all implemented using the base `casadi.nlpsol` class. This is used rather than the Opti stack because it’s more fully featured, we get more control, and most importantly, it has far better support for code generation, which will significantly reduce init and solve times.