

Exercise: Model predictive control with the race car

In the following exercise, we will play around with variations of optimal control.

Tasks

1. We will start by casting a simulation task as a nonlinear optimization problem solved in Opti. For this purpose, we use a collocation scheme. With some hand-waving, this boils down to:

- Subdivide the integration horizon T into N integration intervals referred to by index k .
- Introduce state variables at the beginning of each interval: $X_k \in \mathbb{R}^{n_x}$.
- Introduce helper variables: $X_k^c \in \mathbb{R}^{n_x \times \text{degree}}$.
- On each interval k , consider a polynomial that exactly interpolates through X_k and X_k^c .
- Enforce that the polynomial's slope should match the ODE, point-wise.
- Enforce continuity of the polynomials at the interval boundaries.
- All of the above polynomial activity can be represented by constant linear maps.

Run the Opti example below, and verify that the car covers a distance of 4.066 m

```
x0 = dae.start(dae.x()) # Initial state
f = dae.create('f', ['x', 'u'], ['ode']) # System dynamics

T = 2 # Integration horizon [s]
N = 20 # Number of integration intervals
dt = T/N # Length of one interval

nx = dae.nx() # Number of states

xvar = dae.x()

# Numeric coefficient matrices for collocation
degree = 3
method = 'radau'
tau = ca.collocation_points(degree, method)
[C, D, B] = ca.collocation_coeff(tau)

opti = ca.Opti() # Opti context

xk = ca.MX(x0)

x_traj = [xk] # Place to store the state solution trajectory
for k in range(N): # Loop over integration intervals
```

```

# Decision variables for helper states at each collocation point
Xc = opti.variable(nx, degree)

# Slope of polynomial at collocation points
Z = ca.horzcat(xk, Xc)
Pidot = (Z @ C)/dt

# Collocation constraints (slope matching with dynamics)
opti.subject_to(Pidot==f(x=Xc)["ode"])

# Continuity constraints
xk_next = opti.variable(nx)
opti.subject_to(Z @ D==xk_next)

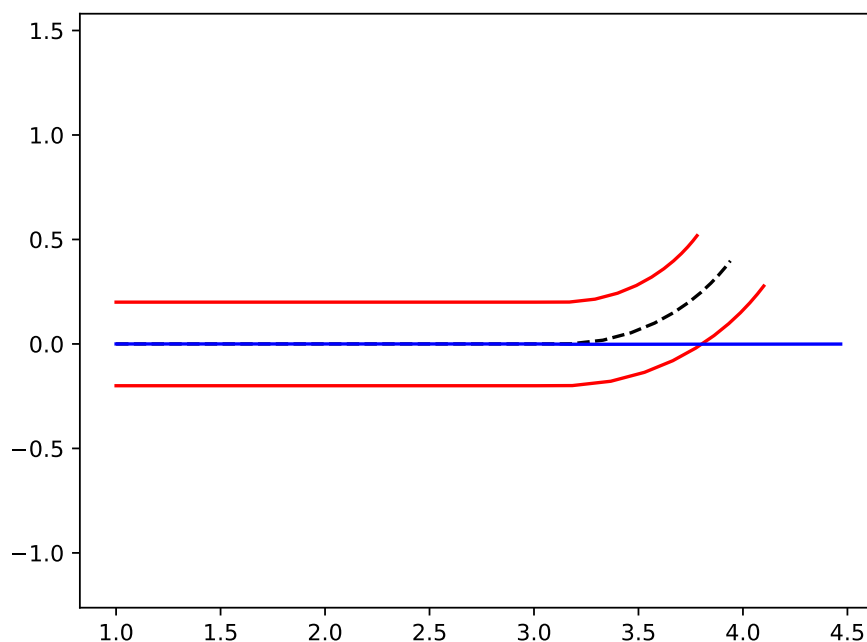
# Initial guesses
opti.set_initial(Xc, ca.repmat(x0,1,degree))
opti.set_initial(xk_next, x0)

xk = xk_next
x_traj.append(xk)
x_traj = ca.hcat(x_traj)

opti.minimize(0)
options = {'ipopt.hessian_approximation': 'limited-memory'}

```

The template script also comes with plotting code that can be used to obtain:

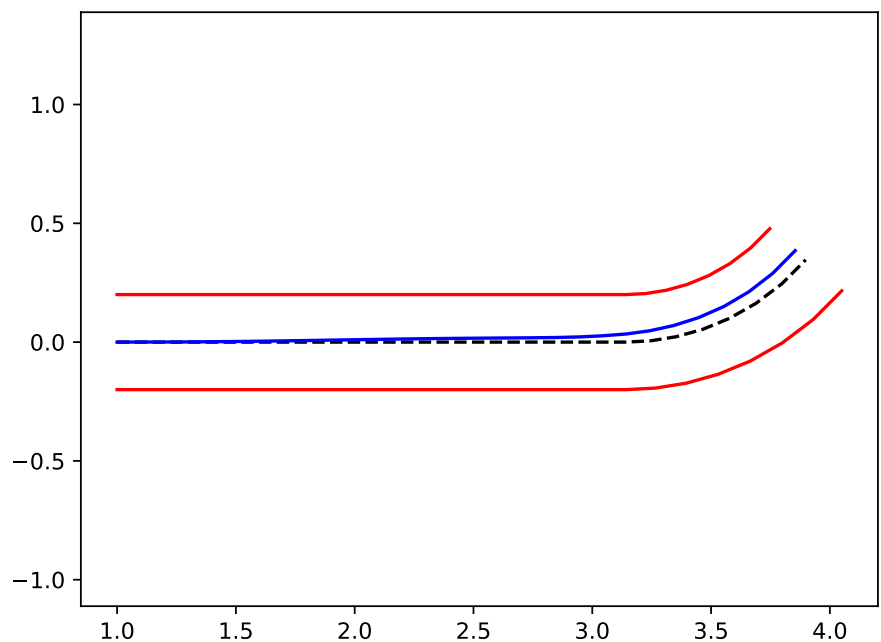


2. Start from the solution of the previous task, but extend the problem's decision variables and constraints.

Find a formulation and a control trajectory such that the car reaches $s(T) = 4$ at the end of

the horizon, whilst staying within track boundaries: $-0.2 \leq n(t) \leq 0.2$.

The result may look something like this:

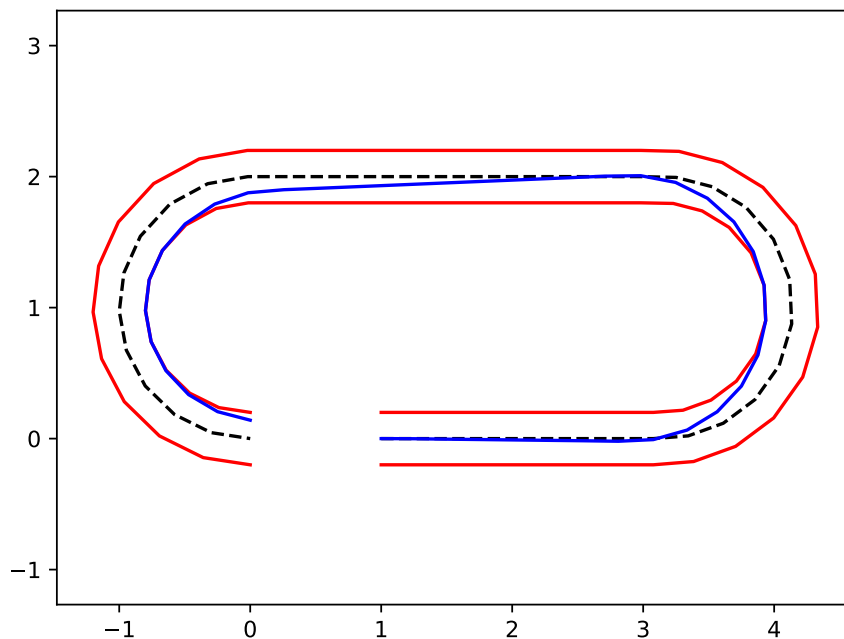


3. Optional: compute a control trajectory that minimizes lap-time. The length of the lap is 4π .

Meaningful bounds of the problem's quantities are given in the table below.

Quantity	Lower bound	Upper bound
n	-0.2	0.2
D	-1	1
\dot{D}	-20	20
δ	-0.8	0.8
$\dot{\delta}$	-4	4
a_{long}	-10	6
a_{lat}	-10	10

With $N = 40$, the optimal solution should look like the following picture:



4. We will work towards model predictive control (MPC) now. Start from the solution script of Task 3.

- Pick a prediction horizon of $T = 0.5$ s and $N = 10$.
- Introduce a parameter \bar{x}_0 , i.e. a changable quantity that is not optimized over:

```
x0_bar = opti.parameter(nx)
opti.set_value(x0_bar, x0)
```

- Introduce an extra decision variable for x_0 :

```
xk = opti.variable(nx)
opti.subject_to(xk==x0_bar)
```

- Instead of a final constraint on $s(T)$, use $-s(T)$ in the objective.
- Relax the solver tolerance with the `ipopt.tol` option set to $1e-5$.
- Obtain a pure CasADi function out of the Opti problem:

```
mpc_step = opti.to_function('mpc_step', [x0_bar, x_traj, u_traj],
    [x_traj, u_traj])
[x_opt, u_opt] = mpc_step(x0, 0, 0)
print("x_opt(T): ", x_opt[:, -1])
```

Here, `mpc_step` maps from the current state \bar{x}_0 and initial guesses for states & controls to the optimal states & controls. Verify that the car ends up at 3.13822 m at the end of the horizon.

MPC is simply the computation and application of `mpc_step` in a loop: Now, MPC is about repeatedly solving this problem in a receding horizon fashion:

```

simulator = ca.integrator('simulator', 'cvodes', dae.create(), 0, dt)

y_traj = [H(x0)]
for k in range(25):
    # Compute optimal trajectories
    [x_opt,u_opt] = mpc_step(x0, x_opt, u_opt)

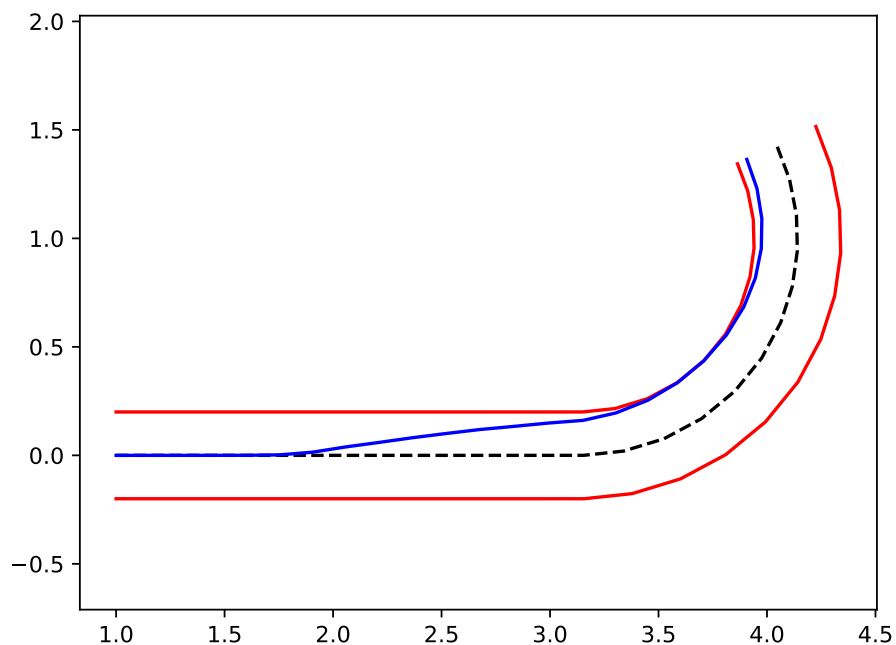
    # What part of the trajectory to apply?
    u_apply = u_opt[:,0]

    # Plant model
    x0 = simulator(x0=x0,u=u_apply)["xf"]

    y_traj.append(H(x0))
y_opt = ca.hcat(y_traj)

```

The resulting closed-loop trajectory should look something like this:



5. Optional: Gauss-Newton and fatrop solver