Exercise: Model predictive control with the race car

In numerical optimal control, we solve decision making problems involving dynamical systems. A popular approach is to use a so-called *simultaneous method* using a *collocation* integration scheme, where a polynomial representation of the solution trajectories enters as decision variables of a nonlinear program. The polynomials are defined by the values of the trajectory at certain points in time, referred to as *collocation points*.
In order for the trajectories to be feasible, we need to enforce two types of constraints:

- Firstly, we need to enforce *continuity* of the trajectory, from one polynomial segment to the next.
- Secondly, we need to enforce that the trajectory satisfies the underlying ODE. We can do this by differentiating the polynomial obtaining an expression for the state derivative at each collocation point. This derivative must be equal to the values for the state derivatives obtained by evaluating the ODE at the same time points.

In the following, we will use Opti to formulate and solve optimal control problems for the race car model from the demo.

Tasks

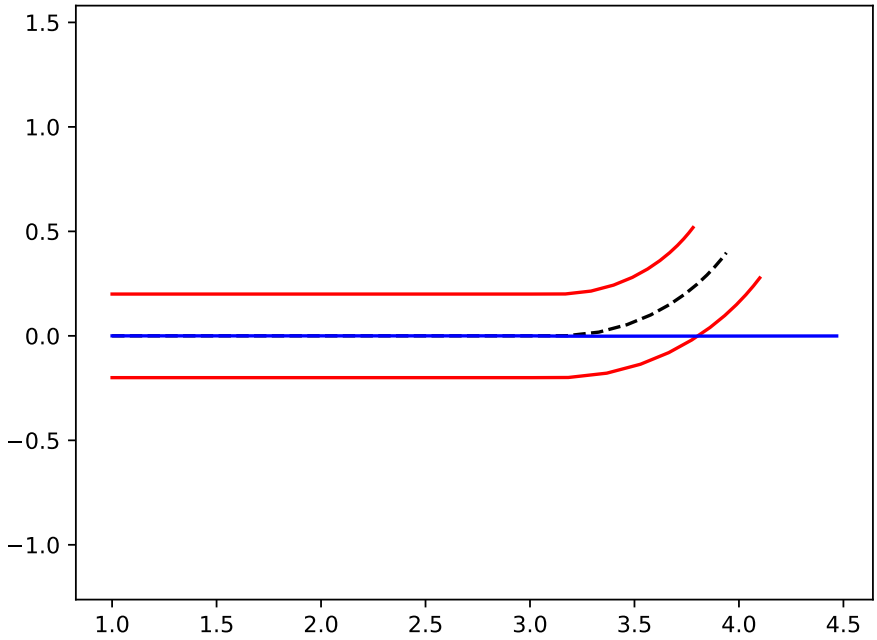
1. When solving optimal control problems, a good starting point is often to make sure that the optimizer is able to reproduce the simulation results. This can be done by formulating a nonlinear program with no degrees of freedom, as we did in Exercise 1.

   Go through the script below, line-by-line, and read the comments for each line. Then run the script, and verify that the car covers a distance of $4.066\,\mathrm{m}$

```
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()  # Names of the states

# 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)  # Initial state (will be overwitten below)

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

    # Value of the states at each collocation point
    Xc = opti.variable(nx, degree)

    # Value of the state derivatives at each collocation point
    Z  = ca.horzcat(xk,Xc)
    Pidot = (Z @ C)/dt

    # Collocation constraints
    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'}
```

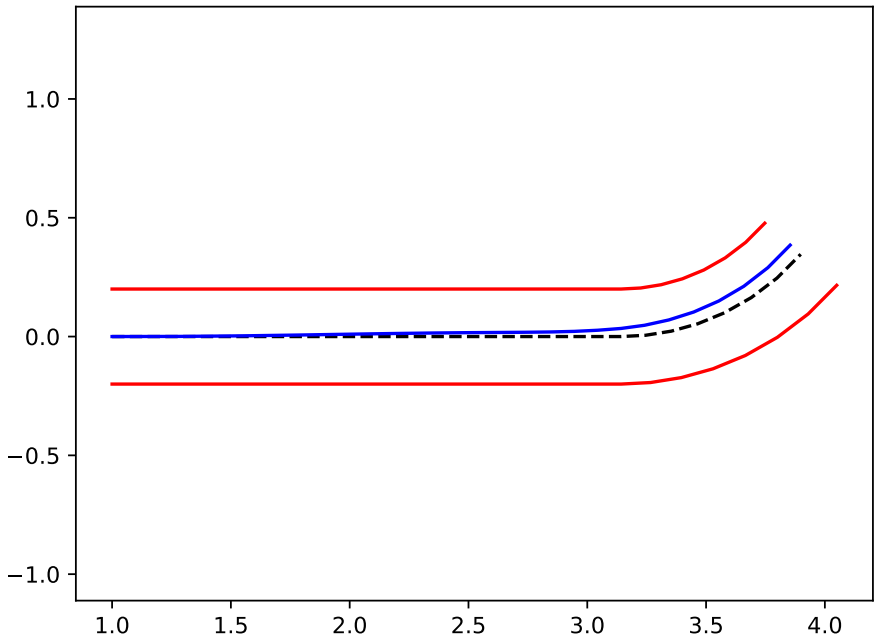
The template script also comes with plotting code that can be used to obtain:



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:

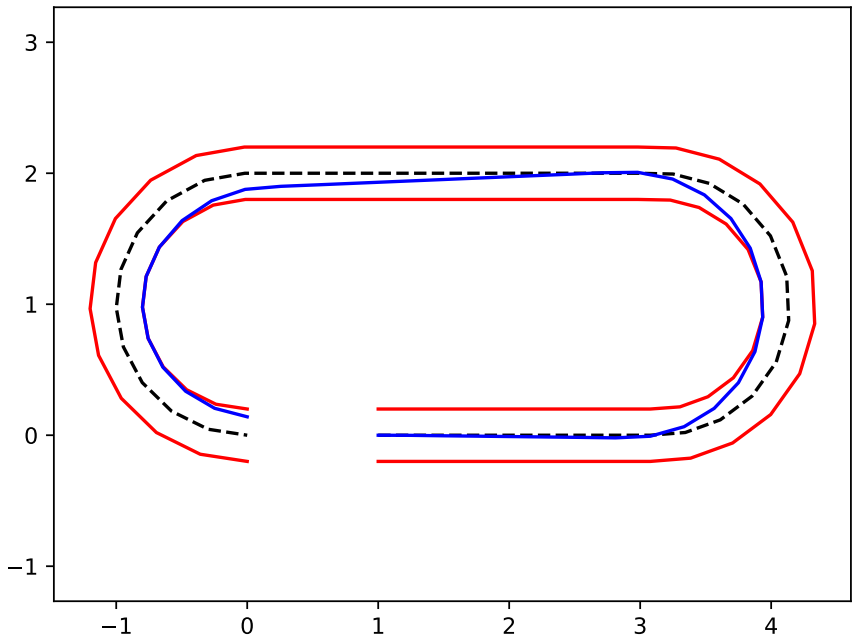


3. Optional: compute a control trajectory that minimizes lap-time. The length of the lap is 4π.

   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 $T = 0.5\,\text{s}$ 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:

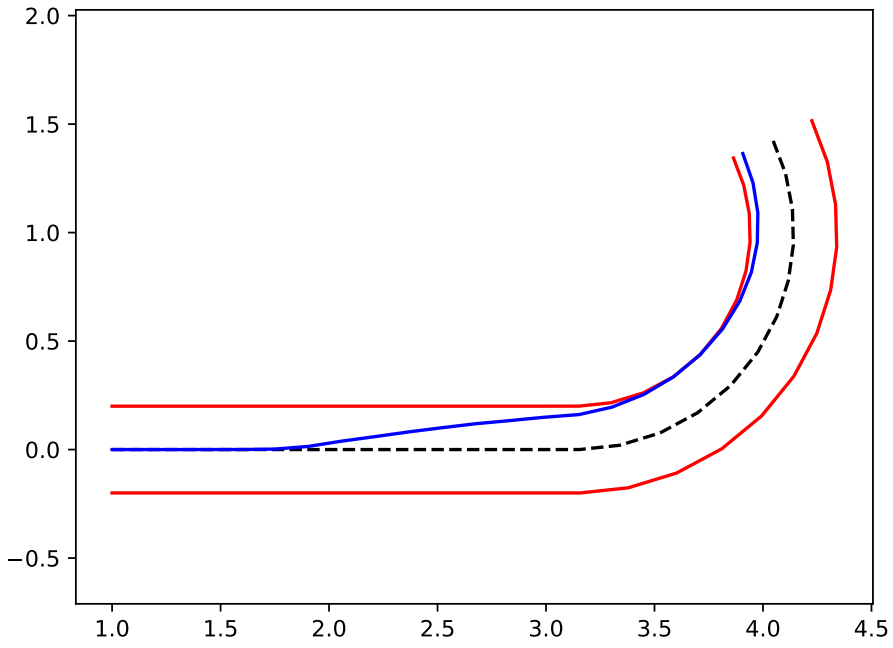
```
mpc_step = opti.to_function('mpc_step', [x0_bar, x_traj, u_traj],
    [x_traj, u_traj])
[x_opt,u_opt] = mpc_step(x0,0,0)
print("x_opt(T): ", x_opt[:,-1])
```

  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 $3.13822\,\mathrm{m}$ 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