WallyTutor

A Chemical Plant MPC

Walter Dal'Maz Silva

2026-07-09

Table of contents

Foreword

In this short article we will introduce the basics of model predictive controls (MPC) by implementing a continuous stirred tank reactor (CSTR) conversion plant model. This will simulate the output concentration of a chemical species whose set-point changes over time, *i.e.* to follow customer orders which require different product concentrations. This is an interesting problem because reactor dilution takes time and a simple proportional-integral controller (PI) would not be enough to track the concentration with minimal out-of-specification periods.

The problem will be implemented with help of CasADi package and its Python interface so that we can take advantage of its features such as automatic differentiation and interface to robust numerical solvers. The development will be broken down into three parts. First we will implement the model from scratch and test its functioning. Then it will be refactored into a more modular code, before being organized as library code to be reused. The goal is to provide an intuitive introduction to the topic which can serve as a basis for further learning.

Working with CasADi requires a minimal introduction to its data types. The package provides a symbolic class `SX` to enable *algorithmic differentiation* throught the construction of computational graphs. It can be used to build expressions which can be later evaluated through the use of `Function` objects. Numerical data returned from solvers is found in dense matrix format `DM`. Other functions imported from `casadi` are related to problem solution and vector concatenation `vertcat` and are discussed in their context of usage.

```
from casadi import SX, MX, DM, Function
from casadi import nlpsol, integrator
from casadi import vertcat, linspace, inf
import numpy as np
import matplotlib.pyplot as plt
```

Preliminary implementation

The problem we will model here consists of an imaginary chemical plant where reagent A is reacted with carrier fluid B in a single continuous stirred tank reactor (CSTR) to produce C, whose concentration is the target. The transformation happens at constant volume, represented by n_t moles. The chemical process is illustrated as the irreversible reaction happening at a known rate k_1:

$$A + B \rightarrow C + B$$

In this plant, we are capable of controling the molar flow rates of both A and B, $\dot{n}_A$ and $\dot{n}_B$, which affect the production rate of C leaving the reactor at a molar fraction x_C. For smooth functioning of the system, the total flow rate is required to remain constant at $\dot{n}_t$ so that $\dot{n}_B = \dot{n}_t - \dot{n}_A$, *i.e.* a compensation valve is the action we control. Thus, we have a single degree of freedom in the actions we can take to pilot this simple system.

Under these assumptions, the effecting the balance for a species i is given below. The source term $\dot{n}_{gen,B}$ is zero as B is a carrier fluid, and for the other species it is computed as a first order homogeneous kinetic rate law (thus independent on the carrier concentration).

$$\begin{aligned} n_t \dot{x}_i &= \dot{n}_i - \dot{n}_t x_i + \dot{n}_{gen,i} \\ \dot{n}_{gen,A} &= -\dot{n}_{gen,C} = -k_1 x_A \\ \dot{n}_{gen,B} &= 0 \end{aligned}$$

Before entering the details of the symbolic model implementation, we start by defining the parameters that are held constant in our problem. That includes the reaction rate k_1, the size of reactor n_t and the total flow rate of $\dot{n}_t$.

```
# Reaction rate constant [mol/s]:
k_1 = 10.0

# Reactor amount of matter [moles]:
n_t = 500.0

# Reactor total flow rate [mol/s]:
ndot_t = 3.0
```

Symbolic model definition

The model formulated in the previous section establishes differential equations for the molar fractions of the different species taking part in our fictional system. Although `SX` symbolics allows for declaring them in vector form, here we have chosen to declare them separately to keep the model code close to the mathematical formulation. The advantage of using `SX` symbolics is that is allows for automatic differentiation of the expressions, which is helpful for the computation of jacobian and hessian matrices required by the numerical solvers.

```
x_A = SX.sym("x_A")
x_B = SX.sym("x_B")
x_C = SX.sym("x_C")
```

For the inlet flow rates, we declare single symbolic variable for $\dot{n}_A$ and evaluate the value of $\dot{n}_B$ as a constraint built-in the model. Notice here that we can use numerical `ndot_t` and symbolic `ndot_A` in the same expression, illustrating the interoperability of CasADi's symbolics and plain Python numbers.

```
ndot_A = SX.sym("ndot_A")
ndot_B = ndot_t - ndot_A
ndot_C = 0.0
```

Source terms are provided below following the mathematical formulation:

```
rate = k_1 * x_A

ndot_gen_A = -rate
ndot_gen_B = 0.0
ndot_gen_C = +rate
```

These source terms complete the set of elements required to construct the differential equations describing the time-evolution of the system. The outlet flow rate is given by the product of total flow rate and molar fractions: $n_t x_i$, as per the definition of an ideal CSTR.

```
xdot_A = (ndot_A - ndot_t * x_A + ndot_gen_A) / n_t
xdot_B = (ndot_B - ndot_t * x_B + ndot_gen_B) / n_t
xdot_C = (ndot_C - ndot_t * x_C + ndot_gen_C) / n_t
```

When using CasADi, one often refers to the unknowns of the problem as `x`, and parameters as `p`. Please notice that parameters can be symbolic (as `ndot_A`) and are provided only at solution time. Here we concatenate the array of unknowns using `vertcat`. There is no need to convert `p` into an array, we just give an alias `p` to `ndot_A` so that we can use typical CasADi formalism.

```
x = vertcat(x_A, x_B, x_C)
p = ndot_A
```

> **i** More about parameters
>
> Numerical parameters (as `ndot_t`) are not modifiable as they take part in the construction of the computational graph, so one must decide whether to keep them as Python constants or use a symbolic representation before model definition. This is important because leaving everything modifiable at runtime can create cumbersome interfaces that require too many parameters to be used, so careful design is recommended.

Problem right-hand side

The assembly of the integrator consists of defining functions to evaluate the right-hand side (RHS) of the ODE system, and step over a given time window. By integrator, here we mean the full routine that will evaluate the solution output over the time-interval of the problem, not just the time-stepping algorithm, as often referred to in ODE's literature.

Using `Function` we wrap the previous expressions into something that can be evaluated in terms of `x` and `p`. Each function is provided a name, a list of inputs and a list of outputs. These could also be named, but we skip that here for simplicity. Also, we do not make use of any other optional arguments and you can check their usage in the documentation.

```
F_xdot_A = Function("F_xdot_A", [x, p], [xdot_A])
F_xdot_B = Function("F_xdot_B", [x, p], [xdot_B])
F_xdot_C = Function("F_xdot_C", [x, p], [xdot_C])
```

i Inspecting a function

It is interesting to check the string representation of a `Function`. Here we see that it receives a first input `i0` which is an array of 3 elements (the shape of `x`) and a second number `i1` (representing `p`) for the returning a single value `o0` (the derivatives). As stated before, you can name these I/O using the more complete interface of `Function`.

```
F_xdot_A
```

```
Function(F_xdot_A:(i0[3],i1)->(o0) SXFunction)
```

We can check the proper functioning of these functions as *normal* Python functions, which can be evaluated numerically. Return values are of type `DM`, CasADi's way of representing dense matrices (here an order zero matrix, a single number).

```
F_xdot_A([0.5, 0.5, 0], 10),\
F_xdot_B([0.5, 0.5, 0], 10),\
F_xdot_C([0.5, 0.5, 0], 10)
```

```
(DM(0.007), DM(-0.017), DM(0.01))
```

Prediction horizon

The first step for converting our simple ODE problem into an MPC is the definition of its *prediction horizon* N_p and associated output step τ. Their values depend on the time-scales of the process at hand the hability to change its inputs (control) paramters. This makes MPC an intrinsically multidisciplinary subject, requiring the process and controls specialist to work together.

For a CSTR, the characteristic residence time given by $\tau_c = n_t/\dot{n}_t$. This value must be at least one order of magnitude greated than the integration step τ, and reasonably smaller than the time horizon $N_p\tau$. Given the defined reactor size and total flow rate, the characteristic time of the system τ_c is of the order of $n_t/\dot{n}_t = 500\,\text{mol}/3\,\text{mol}\cdot\text{s}^{-1} \approx 170\,\text{s}$. To ensure that the time horizon is long enough so that the optimization routine foresees the dynamics of the system to apply corrections in time, and the time steps are small compared to the characteristic time, we chose to take into account the next 2000 s of the dynamics by integrating 200 steps of 10 s.

```
# Prediction horizon:
Np = 200

# Time-step of outputs:
tau = 10.0
```

> ! Important
>
> Please, notice that the horizon size N_p alone is meaningless without the definition of the time-step τ between consective corrective actions. This again must take into consideration the delays system response and the valve itself in this specific case. Also notice that the time-step τ may be the same used for integration of the problem if its stiffness allows for. For stiff problems we generally use smaller *inner time-steps* to reach output interval τ, while keeping the control action constant in between. This is often the case in combustion processes or complex gas pressure controls.

Cost function composition

Next comes the definition of the cost function, which is generally composed of a local quadratic term penalizing deviations of controlled variables from their target values (set-points), another penalizing large control actions, and a terminal penalty.

- The scale of the problem is usually selected to be one of deviation of main controlled variable so the mutiplier of quadratic term Q is set to unity (or identity matrix in more complex multidimensional formulations).
- For the command change penality, scaling R has to be chosen so that is remains in a good order of magnitude compared to target variable cost and still perform its function.
- The last parameter S is the terminal penalization, generally set to a high value so that we enforce the last point in prediction horizon to match the set-point.

```
# Set-point penality scale:
Q = 1.0

# Command change penality scale:
R = 0.1

# Terminal penalization:
S = 100.0
```

Additional constraints

There might be additional constraints to be added to the problem. These may arise from quality requirements or technical constraints of the plant itself. For instance, let's assume our compensation valve does not allow the fraction of A in the total flow to be above 90% of total feed rate. This constraint will be imposed to each action of the system over the process window.

```
ndot_A_max = 0.9 * ndot_t
```

Assembly of the integrator

First we declare the cost function (which is zero initially) and an array for the constraints. For each solution variable we also need a lower and upper bound. To close the system, we store the command v used during each output interval. These variables will be fed by the integrator in what follows.

```
# Cost function:
J = 0.0

# Array of constraints:
g = []

# Solution lower/upper boundaries:
lbx = []
ubx = []

# Control variable over prediction horizon:
v_ndot_A = []
```

> Caution
>
> In the above, the lower and upper boundaries, `lbx` and `ubx` refer to the *decision variables* of our optimization problem, here the controller command for the flow rate of A, not the states of the differential problem. It can be set from zero up to the maximum amount it can be set, here 90% of total flow rate. Note that constant numeric values could have been provided here, but for generality we will use a list here and feed it along with the integration so that you can see how one could implement a possibly *varying* control boundaries.

Another requirement is to have the production planning with the target concentration (set-point) over the prediction horizon. It is symbolically declared in `xs_C` and contains one extra point over `Np` representing the current state of the reactor (initial condition).

```
xs_C = SX.sym("xs_C", Np+1)
```

The initial state of the system is set to the symbols we already know (later we will replace them by numeric *measurements*). For the control input, it is initially set to 50% (`ndot_A_ini`) of the total capacity and that will impact the first command change. Remember that we are handling the variables individually, but in a real-world problem you would probably use a vector with all variables as we edd in `x` before.

```
xt_A, xt_B, xt_C = x_A, x_B, x_C

# Idle control input (initial state)
ndot_A_ini = 0.5 * ndot_t
```

Finally we integrate the problem over time. The loop is composed of a few characteristic steps:

1. Creation of a command variable `v_ndot_a_ts` for the current step.
2. Bounding the values of control variable throughs `lbx` and `ubx`.
3. Stacking of current system state in `xn` and current commands in `pn`.
4. Actual time integration of the system dynamics.
5. Increment of cost function with current deviations.
6. Add constraints (not done here) to the states and controls.
7. To complete the cost function, the terminal cost is added with scale `S`.

Below you will recognize a simple Euler time-stepping.

```
for ts in range(Np):
    v_ndot_A_ts = SX.sym(f"v_qdot_A_{ts}")
    v_ndot_A.append(v_ndot_A_ts)

    lbx.append(0.0)
    ubx.append(ndot_A_max)

    xn = vertcat(xt_A, xt_B, xt_C)
    pn = v_ndot_A_ts

    xt_A = xt_A + tau * F_xdot_A(xn, pn)
    xt_B = xt_B + tau * F_xdot_B(xn, pn)
    xt_C = xt_C + tau * F_xdot_C(xn, pn)

    v_prev = ndot_A_ini if ts == 0 else v_ndot_A[ts-1]

    scale_error  = xt_C - xs_C[ts]
    scale_change = v_prev - v_ndot_A_ts

    cost_error   = Q * pow(scale_error, 2)
    cost_change  = R * pow(scale_change, 2)

    J += cost_error + cost_change

J += S * pow(xt_C - xs_C[-1], 2)
```

i About the initial command

Different approaches could be used for the initial command. Here we have decided to illustrate the system with an hypothetical known idle state initial flow rate of A, and use `R` to avoid abrupt changes in the flow command. This way we make use of that value here to compute the initial step accordingly. Another approach would to use a constraint to enforce the idle state in the first step, for example.

The assembly of the system to be solved made below. Here we have chosen to optimize the problem with Ipopt (which can be accessed though interface `nlpsol`) as a nonlinear problem. In some cases you might

wish to use a quadratic solver, but it imposes some limitations in problem formulation. You should do that when solving as a NLP is too slow for your problem. In CasADi's representation `f` denotes the cost, `x` the free variables (the commands here), `g` the constraints list, and `p` the parameters, values that were left in symbolic form and we need to provide numerically. Here `p` is the array of set-points and the system initial state. Interface `nlpsol` allows for creating a solver, which may be used many times later.

```
nlp = {
    "f": J,
    "x": vertcat(*v_ndot_A),
    "g": vertcat(*g),
    "p": vertcat(xs_C, x)
}

opts = {"ipopt": {"print_level": 3}}
solver = nlpsol("solver", "ipopt", nlp, opts)
```

Below we can inspect the interface of the solver with the sizes of the arrays we must provide.

```
solver
```

```
Function(solver:
(x0[200],p[204],lbx[200],ubx[200],lbg[0],ubg[0],lam_x0[200],lam_g0[0])-
>(x[200],f,g[0],lam_x[200],lam_g[0],lam_p[204]) IpoptInterface)
```

Solving the problem

Let's now compose the parameters array. For the set-point `xs_C_num` consider that you need the solution to be delivered with a concentration of 20% in the first 120 steps (3/5 of prediction horizon) of C and for the next lot the concetration must be increased to 50%. Below we translate this requirement into an array used for proble setup.

```
n_step = 3 * Np // 5

xs_C_num = np.zeros(Np+1)
xs_C_num[:n_step] = 0.2
xs_C_num[n_step:] = 0.5
```

Let's assume that at the initial time the system is composed only of B, the second element in our composition array. With that we can merge the above defined set-points and initial composition arrays in the format expected by the solver (as declared above) to provide `p`.

```
x0_num = [0.0, 1.0, 0.0]
p = vertcat(xs_C_num, x0_num)
```

Call of the solver is mostly trivial. If this is the first call to the solver, normally provide an initial guess composed of a fixed value that makes sense to your system (here an array of ones was chosen). When

using the solver in an actual control loop, generally the results of last call are provided, what is interesting for speed-up. Since we have a single constraint in `g`, it suffices to provide `lbg=ubg=0.0` to enforce the initial flow rate to the prescribed value. Call can be a bit verbose and you might want to log it in a production environment.

```
guess = np.ones(Np)
solution = solver(x0=guess, p=p, lbx=lbx, ubx=ubx, lbg=0.0, ubg=0.0)
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit https://github.com/coin-or/Ipopt
******************************************************************************

Total number of variables............................:      200
                     variables with only lower bounds:        0
                variables with lower and upper bounds:      200
                     variables with only upper bounds:        0
Total number of equality constraints.................:        0
Total number of inequality constraints...............:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0


Number of Iterations....: 14

                                   (scaled)                 (unscaled)
Objective...............:   3.9015653462196176e-01    3.9015653462196176e-01
Dual infeasibility......:   2.1747432450719620e-16    2.1747432450719620e-16
Constraint violation....:   0.0000000000000000e+00    0.0000000000000000e+00
Variable bound violation:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity.........:   4.1114517841581608e-09    4.1114517841581608e-09
Overall NLP error.......:   4.1114517841581608e-09    4.1114517841581608e-09


Number of objective function evaluations             = 15
Number of objective gradient evaluations             = 15
Number of equality constraint evaluations            = 0
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 0
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 14
Total seconds in IPOPT                               = 0.064
```

```
EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  |        0 (       0) 435.00us ( 29.00us)        15
  nlp_grad_f  |   3.00ms (187.50us) 748.00us ( 46.75us)        16
  nlp_hess_l  |  39.00ms (  2.79ms)  41.56ms (  2.97ms)        14
       total  |  65.00ms ( 65.00ms)  65.11ms ( 65.11ms)         1
```

Below we recover the solution for reuse in system simulation. Observe that the simulation loop is essentially the same as the construction of the cost function but we implement just the time-stepping routines. This is done here just for plotting purposes and estimation of system performance. Often the simulation of the problem is not required in an MPC.

```
ndot_A_opt = solution["x"].full().ravel()

xt_A = np.zeros(Np+1)
xt_B = np.zeros(Np+1)
xt_C = np.zeros(Np+1)

xt_A[0] = x0_num[0]
xt_B[0] = x0_num[1]
xt_C[0] = x0_num[2]

for ts in range(1, Np+1):
    xn = vertcat(xt_A[ts-1], xt_B[ts-1], xt_C[ts-1])
    pn = ndot_A_opt[ts-1]

    xt_A[ts] = xt_A[ts-1] + tau * F_xdot_A(xn, pn)
    xt_B[ts] = xt_B[ts-1] + tau * F_xdot_B(xn, pn)
    xt_C[ts] = xt_C[ts-1] + tau * F_xdot_C(xn, pn)
```

Now let's assume that according to the quality deparment, the product is good to be shipped if its concentration is anywhere withing 5% of the prescribed value. A boolean array `good` is created for displaying. This could be used to predict when the exit valve feed barrels of product or when to recycle/throw the output.

```
xs_C_max = np.clip(xs_C_num + 0.05, 0.0, 1.0)
xs_C_min = np.clip(xs_C_num - 0.05, 0.0, 1.0)
good = (xt_C >= xs_C_min) & (xt_C <= xs_C_max)

steps = list(range(Np+1))
quality = 100 * sum(good.astype("u8")) / len(good)

cmd = list(ndot_A_opt / ndot_t)
cmd.append(cmd[-1])
```

Visualizing the results

The next figure illustrates the expected dynamical behavior of the system. First concentration of C rises from zero to the prescribed value in about 15 steps, and because of the change in set-point it starts deviating from target at step 110 to reach the new value. It is also interesting to observe that command saturates at 90%, what indicates that production of the second target is a limiting case for this reactor under the given transition. We respected the process window during 88.6% of the time.

```
plt.close("all")
plt.style.use("default")
fig, ax = plt.subplots(figsize=(8, 4))

ax.grid(linestyle=":")
ax.plot(steps, xt_A, lw=2, label="$X_A$")
ax.plot(steps, xt_B, lw=2, label="$X_B$")
ax.plot(steps, xt_C, lw=4, label="$X_C$")

ax.step(steps, xs_C_max, "m--", lw=2, label="_none_", where="post")
ax.step(steps, xs_C_min, "m--", lw=2, label="_none_", where="post")
ax.step(steps, xs_C_num, "m-", lw=1, label="$X_C$ (target)", where="post")

# Add *negative time* with initial flow rate.
ax.step([-1, *steps], [ndot_A_ini/ ndot_t, *cmd], "r", lw=2,
        label="$Q_A$ (relative)", where="post")

ax.fill_between(steps, xs_C_min, xs_C_max, where=good, alpha=0.3)

ax.set_title(f"Expected quality level at {quality:.1f}%")
ax.set_ylabel("Mole fractions and relative flow rate of A")
ax.set_xlabel("Action step number over prediction horizon")
ax.legend(loc="upper center", fancybox=True, framealpha=1.0,
          ncol=5, fontsize="small")
ax.set(xlim=(0, Np), ylim=(0, 1))

fig.tight_layout()
```

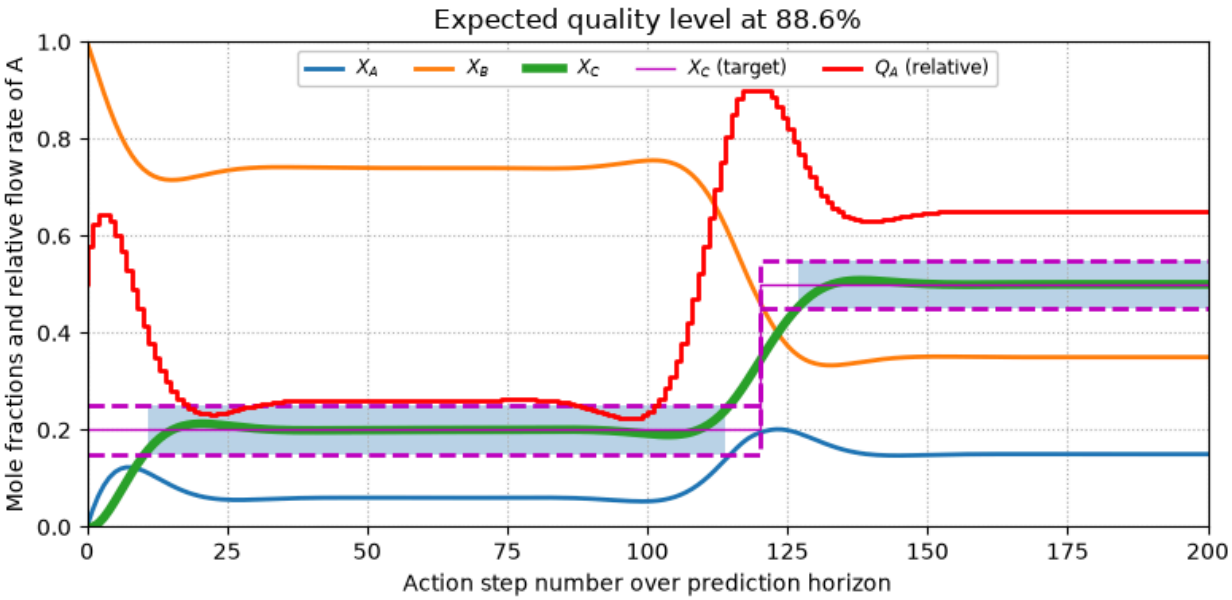

Figure 1: Reactor simulation results and controls over time.

i Representing steps in Matplotlib

Because of how `matplotlib.step` works, to display properly the commands and responses we need to add an extra copy of last command to the end of the results. Using `where="post"` is also required so that a command *starts* at the reference time-step it is supposed to. You can check the plotting behaviour with the following snippet.

```
x = [1, 2, 3, 4]
y = [0, 1, 3, 1]

z = [1, 2, 0]
z.append(z[-1])

plt.close("all")
plt.figure(figsize=(4, 3))
plt.plot(x, y, "ro-", label="Response")
plt.step(x, z, "ko-", label="Command", where="post")
plt.legend(loc=2)
plt.tight_layout()
```

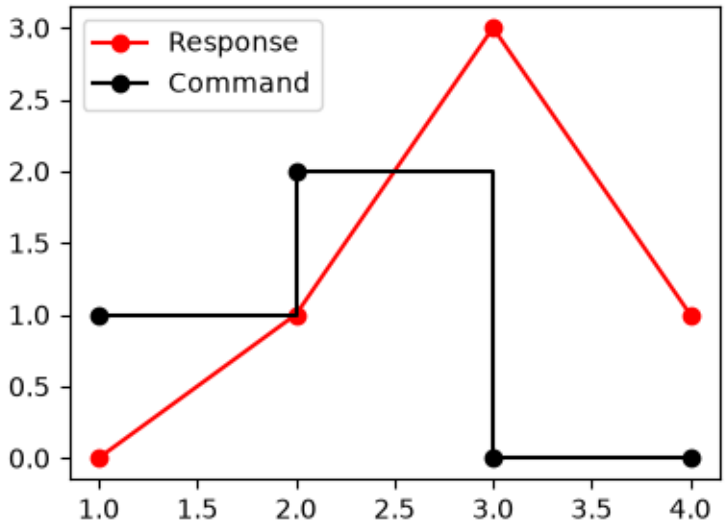

Figure 2: Illustration of how steps work in Matplotlib.

There are many ways you could propose exercises from this guided study:

- Implement a higher order time-stepping scheme.
- Provide simultaneous simulation/optimization with multiple-shooting.
- Investigate role of total flow rate or reactor size over quality.
- Increase error out of target value range (shadowed zones in figure).
- Use the solver in a simulated control loop with random noise in measurements.
- ...

Introduction to multiple shooting

In a real-world application, you would probably use a multiple-shooting approach to simultaneously simulate and optimize the problem, if that is required. That comes with an additional computational cost because the optimization problem becomes larger, as you also solve for the differential states along with controls. In the introductory part of this tutorial, it was chosen to keep focus on the main ideas an exploit a simpler approach (from the implementation standpoint).

The multiple shooting strategy is discussed in the present section using a different problem, before turning back to our MPC concept. We will reformulate a tutorial originally created by Joel Andersson in 2015 and provided with CasADi's examples pack. In this notebook we seek to solve the optimal control problem (OCP) given by

$$\min \int_0^{10} x_0^2 + x_1^2 + u^2 dt$$

subjected to

$$\begin{aligned} \dot{x}_0 &= z x_0 - x_1 + u \quad && t \in [0, 10] \\ \dot{x}_1 &= x_0 && t \in [0, 10] \\ 0 &= x_1^2 + z - 1 && t \in [0, 10] \end{aligned}$$

with boundary values

$$\begin{aligned} x_0(t=0) &= 0 \\ x_1(t=0) &= 1 \\ x_0(t=10) &= 0 \\ x_1(t=10) &= 0 \end{aligned}$$

and bounded control $u \in [-0.75; 1.00]$ over the whole interval, with z being an algebraic variable. An alternate implementation and description of this problem is provided with Dymos package.

Symbolic DAE definition

The problem is build on `SX` symbolics defined as follows:

```
x = SX.sym("x", 2)
z = SX.sym("z")
u = SX.sym("u")
```

The differential and algebraic equations can then be written as:

```
# Lagrange cost term (quadrature)
quad = x[0]**2 + x[1]**2 + u**2

# Differential equation
ode = vertcat(z * x[0] - x[1] + u, x[0])
```

```
# Algebraic equation
alg = x[1]**2 + z - 1
```

Integrator setup

Time integration parameters are then declared:

```
# Number of intervals.
nk = 50

# Final time [s].
tend = 10.0

# Time step [s].
tf = tend / nk
```

Since we have already provided `x` as a state vector it can be supplied directly to the integrator. Below we use the interface to Sundials IDAS to provide DAE time-stepping later in multiple shooting. The value of `tf` provides the integration (output) time step.

```
dae = {"x": x, "z": z, "p": u, "ode": ode, "alg": alg, "quad": quad}
I = integrator("I", "idas", dae, 0.0, tf)
```

Nonlinear problem

To differentiate the variables from DAE to those of multiple shooting NLP, we will use `w` in what follows. Below we initialize arrays for variables, bounds, constraints, and the value of cost function. These will be used to build the NLP.

```
# List of variables
w = []

# Lower bounds on w
lbw = []

# Upper bounds on w
ubw = []

# Constraints
g = []

# Cost function
f = 0.0
```

For multiple shooting we use `MX` matricial symbolics. For enforcing initial state in NPL we create a first state `Xk` and enforce its lower and upper bounds (`lbw` and `ubw`) to match the values given in problem statement.

```
Xk = MX.sym("X0", 2)

w.append(Xk)
lbw.extend([0.0, 1.0])
ubw.extend([0.0, 1.0])
```

The idea behind multiple shooting integration loop is quite simple:

1. we create a local control u_k to integrate over t_k to t_{k+1}, which is a free variable in the NLP problem, thus it must be added to `w` and its bounds to `lbw` and `ubw`,
2. then we call the DAE integrator to retrieve (symbolically here) the state and lagrangian quadradure that is produced with control u_k and current problem state x_k,
3. perform variable *lifting*, *i.e.* we store predicted state and create a new symbolic state which is now a free variable and must again be added to `w` and its bounds to `lbw` and `ubw`,
4. finally we constrain the created state to match the previous prediction in `g`, what is the *shooting* part of the integration.

```
for k in range(nk):
    # Local control
    Uk = MX.sym(f"U{k}")
    w.append(Uk)
    lbw.append(-0.75)
    ubw.append( 1.00)

    # Call integrator function
    Ik = I(x0=Xk, p=Uk)
    Xk = Ik["xf"]
    f += Ik["qf"]

    # "Lift" the variable
    X_prev = Xk

    # Create new symbolic state
    Xk = MX.sym(f"X{k+1}", 2)
    w.append(Xk)
    lbw.extend([-inf, -inf])
    ubw.extend([+inf, +inf])

    # Constrain problem
    g.append(X_prev - Xk)
```

Solving the problem

To solve the problem we allocate an NLP solver. Notice that the interface of `nlpsol` is quite similar to the one of `integrator`. Here we make use of Ipopt to perform optimization. Below we can inspect the interface of the function created by the wrapper.

```
nlp = {"x": vertcat(*w), "f": f, "g": vertcat(*g)}

# "ipopt.output_file": "ipopt.txt"
opts = {
    "ipopt.print_level": 3,
    "ipopt.linear_solver": "mumps"
}

solver = nlpsol("solver", "ipopt", nlp, opts)
solver
```

```
Function(solver:
(x0[152],p[],lbx[152],ubx[152],lbg[100],ubg[100],lam_x0[152],lam_g0[100])-
>(x[152],f,g[100],lam_x[152],lam_g[100],lam_p[]) IpoptInterface)
```

To solve the NLP we provide an initial guess, the bounds of free variables and the ones of constraints (`lbg  andubg`. Notice that you could provide constraint relaxation in some cases as allowed by specific problems.

```
sol = solver(x0=0.0, lbx=lbw, ubx=ubw, lbg=0.0, ubg=0.0)
```

```
Total number of variables............................:      150
                     variables with only lower bounds:        0
                variables with lower and upper bounds:       50
                     variables with only upper bounds:        0
Total number of equality constraints.................:      100
Total number of inequality constraints...............:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0


Number of Iterations....: 11

                                   (scaled)                 (unscaled)
Objective...............:   2.8826157589618751e+00    2.8826157589618751e+00
Dual infeasibility......:   1.9331531178611032e-10    1.9331531178611032e-10
Constraint violation....:   2.0780707155054756e-09    2.0780707155054756e-09
Variable bound violation:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity.........:   5.3030799022244858e-09    5.3030799022244858e-09
Overall NLP error.......:   5.3030799022244858e-09    5.3030799022244858e-09


Number of objective function evaluations             = 12
Number of objective gradient evaluations             = 12
```

```
Number of equality constraint evaluations            = 12
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 12
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 11
Total seconds in IPOPT                               = 1.052

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  |  30.00ms (  2.50ms)  29.39ms (  2.45ms)        12
       nlp_g  |  28.00ms (  2.33ms)  29.81ms (  2.48ms)        12
  nlp_grad_f  | 316.00ms ( 12.15ms) 311.34ms ( 11.97ms)        26
  nlp_hess_l  | 477.00ms ( 43.36ms) 478.94ms ( 43.54ms)        11
   nlp_jac_g  | 208.00ms ( 16.00ms) 205.61ms ( 15.82ms)        13
       total  |   1.07 s (  1.07 s)   1.07 s (  1.07 s)         1
```

Below you can inspect the keys in solution dictionary. We are looking for `x` here. Since the constraint violation in the report above is acceptable, we can skip the detailed inspection of `g` residuals. You can also check the value of cost `f`.

```
sol.keys()
```

```
dict_keys(['f', 'g', 'lam_g', 'lam_p', 'lam_x', 'x'])
```

Unpacking the results

Since when creating free variables vector `w` we started by initial states before entering the loop to add controls and following states, we have that starting on index 0 the variables correspond to x_0, index 1 to x_1, and index 2 to u. Since there are three variables, we recover then every 3 elements using slicing syntax.

> **i** CasADi to NumPy
>
> By calling method `full()` we convert CasADi `DM` numerical array to plain Numpy. Since this will produce one dimension per element, it is usefull to call `ravel()` and get an one-dimensional array.

```
xs = sol["x"].full().ravel()

x0 = xs[0::3]
x1 = xs[1::3]
us = xs[2::3]
```

Because we have an initial state, there are `nk+1` states and `nk` control intervals. To proper represent them graphically we allocate corresponding time arrays.

```
tx = linspace(0.0, tend, nk + 1).full()
tu = linspace(0.0, tend, nk + 0).full()
```

Visualizing the results

Finally the full solution can be visualized. Since controls are held constant over intervals, a step representation is more adapted for this variable.

```
plt.close("all")
plt.style.use("default")
fig, ax = plt.subplots(figsize=(8, 4))

ax.grid(linestyle=":")
ax.plot(tx, x0, "r", label="$x_0$")
ax.plot(tx, x1, "b", label="$x_1$")
ax.step(tu, us, "k", label="$u$", where="post")
ax.set(xlabel="Time [s]", ylabel="State [-]")
ax.set(xlim=(0, tend), ylim=(-0.75, 1.25))
ax.legend(loc="upper center", fancybox=True, framealpha=1.0,
          ncol=5, fontsize="small")

fig.tight_layout()
```

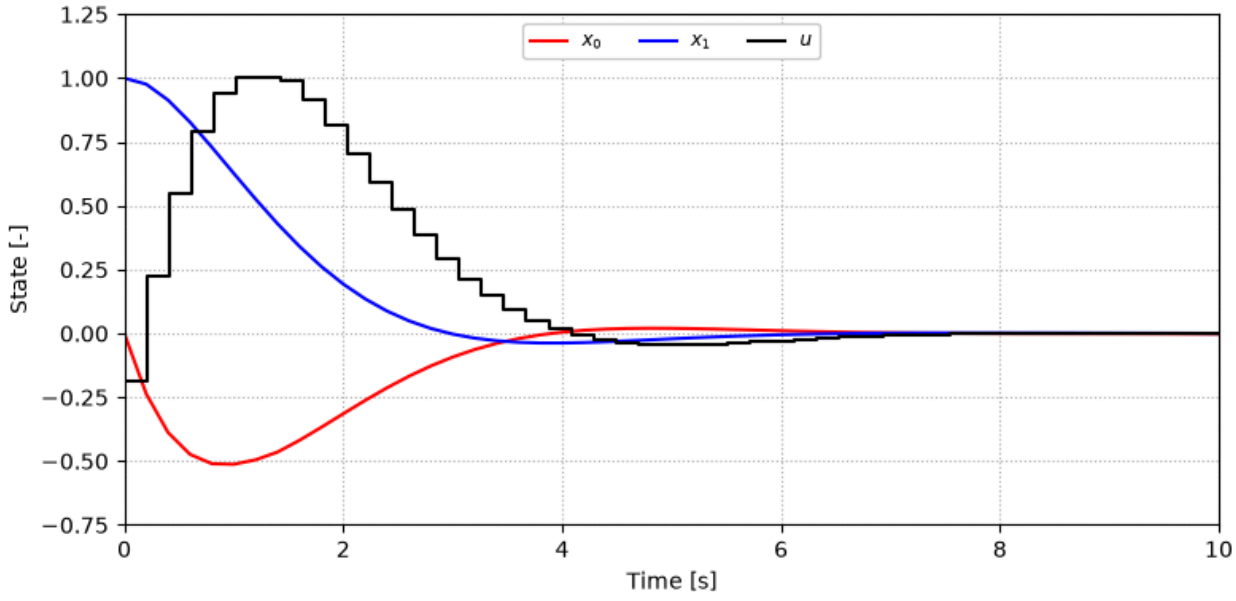

Figure 3: System dynamics and controls over time.

Refactoring the original problem

Upcoming

Wrapping for deployment

Upcoming