MPC CODE: User Guide

Marco Vaccari†, Gabriele Pannocchia

University of Pisa, Department of Civil and Industrial Engineering, Largo Lucio Lazzarino 2, 56126 Pisa (Italy).
September 7, 2023

†Email: marco.vaccari@unipi.it.

Contents

Chapter 1

Introduction

1.1 Installation

This document is intended to give a guideline to the user who has to prepare his own example to test on MPC_code or to understand how the given ones work.
The program is written for Python[1] and CasADi 3.4 (both are supported on Windows \ Linux \ Mac). For detailed installation instruction on CasADi go to the link:
https://web.casadi.org/get/
Scypy and Numpy packages are also required. In the case where these package are not given in your Python installation package, you can download it form the following link:
https://www.scipy.org/scipylib/download.html
In order to make the code compatible to both Python 2.7 and 3.6, the package `future` has to be installed (for more details see: http://python-future.org/index.html). This package can be installed via `pip install`, or searching for `future` among non installed packages in Anaconda Navigator (more information can be found at https://anaconda.org/anaconda/anaconda-navigator).
The MPC_code is distributed as packed file MPC_code.zip that contains the following items:

- `MPC_code.py` : main file that has to be run
- `Targ_calc.py` : it contains the steady-state target optimization module
- `Control_calc.py` : it contains the dynamic optimization module
- `Estimator.py` : it contains all the possible state estimators
- `Utilities.py` : it contains many functions used in all the other modules
- `Default_Values.py` : it contains the default values of many options the user can specify in the example file. Detailed descriptions will follow
- `SS_JAC_ID.py` : it contains a tool for an automatic system linearization
- `Ex_LMPC_WB.py`, `Ex_LMPC_CSTR.py` : examples of linear MPC. They represent the case where both the model and the process maps are linear
- `Ex_LMPC_nlplant.py` : example of linear controller MPC and non-linear process equations. The linear controller model has been implemented with linearization matrices and corresponding parameters;
- `Ex_LMPCxp_plant.py` : example of linear controller MPC and non-linear process equations where model and process have a different number of states;

[1] Both Python 2.7 and 3.6 are now supported

- `Ex_NMPC.py` : example of non-linear MPC. It represents the case where both the model and the process maps are non-linear;
- `Ex_NMPC_dis.py` : example of non-linear MPC. It represents the case where both the model and the process maps are non-linear where the state maps are in discrete-time for both the process and the model;
- `Ex_ENMPC.py` : example of non-linear economic MPC. It represents the case where the model, the process maps and also the optimization modules objective functions are non-linear.

1.2 Testing default examples

First of all, unpack the zip file and put everything into a folder. In order to run a simulation for the default examples given, write the name of the file to test (i.e. `Ex_LMPC.py`) in the following line in **MPC_code.py**:

To clarify better, a line identified by the comment "`# Insert here your file name`" indicates the exact place at the top of the file. Note also that the extension "`*.py`" is not required.
Once set the file name for the wanted file, just run the Python script `MPC_code.py` to start the simulation.

When, instead, you want to create a file (new, or starting from one of the given examples), i.e. "**Ex_1.py**"; remember to copy all the `import` lines present at the top of each given examples. As said above, after preparing the test file, write its name in the `MPC_code.py` with the following line:

```
ex_name = __import__('your_relative_path/Ex_1').
```

The '`your_relative_path`' string identifies the relativepath where the `Ex_1.py` can be found in your disc respect to `MPC_code.py`. If the file `Ex_1.py` is in the same folder of `MPC_code.py` there is no need to add any path before its name.

Chapter 2

Basic Usage and samples

To start using the MPC CODE, few basic features are introduced together with some examples contained in the zip file. Three examples are given here to introduce the user to the code:

- firstly a Non Linear example is given and explained step by step;
- then a Linear MPC approximation of the previous one is given;
- finally an Economic MPC example is explored.

It is underlined that this section provides a mix of samples for the basic user that does want to practice with a fairly standard MPC example. Anyway the reader is invited to consult Chapter 3 for the complete range of possibilities that this code offers.
It is important to note that *the user cannot change the name of the variable written in* `typographical style`, e.g. the code is case dependent respect to these variables.

2.1 Non Linear MPC

This example can be followed in the `Ex_NMPC.py` file given in the zip file.
Consider a non-linear system described by a ODE system by

$$\dot{x_0} = \frac{F_0(c_0 - x_0)}{\pi r^2 x_2} - k_{T_0} \exp\left(-\frac{E}{R}\left(\frac{1}{x_1} - \frac{1}{T_0}\right)\right) x_0$$

$$\dot{x_1} = \frac{F_0(T_0 - x_1)}{\pi r^2 x_2} - \frac{\Delta H}{\rho C_p} k_{T_0} \exp\left(-\frac{E}{R}\left(\frac{1}{x_1} - \frac{1}{T_0}\right)\right) x_0 + 2\frac{U_0}{r\rho C_p}(T_c - x_1)$$

$$\dot{x_2} = \frac{F_0 - u_1)}{\pi r^2}$$

First of all let us introduce the simulation parameters used: a discretization time step of 0.2 is chosen to build the discrete-time system, and a simulation length of 200 time steps is used. Moreover a prediction horizon of 50 time steps is selected.

```
Nsim = 201 # Simulation length

N = 50     # Horizon

h = 0.2 # Time step
```

The problem dimensions are the following: 3 states, 2 inputs, and 2 ouputs.

```
xp = SX.sym("xp", 3) # process state vector       #
x = SX.sym("x", 3)   # model state vector         #
u = SX.sym("u", 2)   # control vector             #
y = SX.sym("y", 2)   # measured output vector     #
```

Then the continuous-time non linear process state map is defined in this way:

```
def User_fxp_Cont(x,t,u,pxp,pxmp):
    """
    SUMMARY:
    It constructs the function fx_p for the non-linear case

    SYNTAX:
    assignment = User_fxp_Cont(x,t,u,pxp,pxmp)

    ARGUMENTS:
    + x          - State variable
    + t          - Current time
    + u          - Input variable
    + pxp        - Process Parameter
    + pxmp       - Measureable Process Parameter

    OUTPUTS:
    + fx_p       - Non-linear plant function
    """
    F0 = if_else(t <= 5, 0.1, if_else(t<= 15, 0.15, if_else(t<= 25, 0.08, 0.1)))
    T0 = 350  # K
    c0 = 1.0  # kmol/m^3
    r = 0.219 # m
    k0 = 7.2e10 # min^-1
    EoR = 8750 # K
    U0 = 915.6*60/1000  # kJ/min*m^2*K
    rho = 1000.0 # kg/m^3
    Cp2 = 0.239 # kJ/kg
    DH = -5.0e4 # kJ/kmol
    Ar = math.pi*(r**2)
    kT0 = k0*exp(-EoR/T0)

    fx_p = vertcat\
    (\
    F0*(c0 - x[0])/(Ar *x[2]) - kT0*exp(-EoR*(1.0/x[1]-1.0/T0))*x[0], \
    F0*(T0 - x[1])/(Ar *x[2]) -DH/(rho*Cp2)*kT0*exp(-EoR*(1.0/x[1]-1.0/T0))*x[0] + \
    2*U0/(r*rho*Cp2)*(u[0] - x[1]), \
    (F0 - u[1])/Ar\
    )

    return fx_p

Mx = 10 # Number of elements in each time step
```

For cleaner notation, the function preamble environment denoted by " " " will be omitted later on.
The parameter `Mx` represent the number of elements used in the automatic integration inside the code.
Next the output map is the one to be defined. In this case only the first and the last state are measured, in the code we define:

```
def User_fyp(x,u,t,pyp,pymp):

    fy_p = vertcat\
    (\
    x[0],\
    x[2] \
    )

    return fy_p
```

In addition let us consider some measurement noise on the process output in this form:

$$y_p = h_p(x,u,t,py_p,pym_p) + v$$

where $h_p(x,u,t,py_p,pym_p)$ is the process output map of the process while $v = R_{wn}^{\frac{1}{2}} v_{rand}$ in which $v_{rand} \in$

$\mathbb{R}^{n_y}$ is random vector normally distributed with mean “0" and variance “1". In order to define the noise, the covariance matrix, e.g. $R_{wn} = 10^{-7} \times I_{ny}$, has to be specified. In the code will be:

```
R_wn = 1e-7*np.array([[1.0, 0.0], [0.0, 1.0]]) # Output white noise covariance matrix
```

The model function is written analogously to the process one in the following way:

```
def User_fxm_Cont(x,u,d,t,px):
    F0 = d[1]
    T0 = 350  # K
    c0 = 1.0  # kmol/m^3
    r = 0.219 # m
    k0 = 7.2e10 # min^-1
    EoR = 8750 # K
    U0 = 915.6*60/1000  # kJ/min*m^2*K
    rho = 1000.0 # kg/m^3
    Cp2 = 0.239 # kJ/kg
    DH = -5.0e4 # kJ/kmol
    Ar = math.pi*(r**2)
    kT0 = k0*exp(-EoR/T0)

    x_model = vertcat\
    (\
    F0*(c0 - x[0])/(Ar *x[2]) - kT0*exp(-EoR*(1.0/x[1]-1.0/T0))*x[0], \
    F0*(T0 - x[1])/(Ar *x[2]) -DH/(rho*Cp2)*kT0*exp(-EoR*(1.0/x[1]-1.0/T0))*x[0] + \
    2*U0/(r*rho*Cp2)*(u[0] - x[1]), \
    (F0 - u[1])/Ar\
    )
    return x_model
```

Note that in this case a non linear disturbance model is used: the disturbance size is chosen to fulfill the offsett-free requirements

```
d = SX.sym("d", 2)  # disturbance                    #
```

and the `offree` tag has to be changed to the proper value

Moreover the parameter `F0` in `User_fxm_Cont(x,u,d,t,px)` has been imposed to be equal to the second component of the disturbance. The output map is defined in analogy with the process one:

```
def User_fym(x,d,t,py):
    fy_model = vertcat\
                (\
                x[0],\
                x[2]\
                )
    return fy_model
```

The setpoints are chosen to be constant for the entire simulation:

```
def defSP(t):
    xsp = np.array([0.0, 0.0, 0.0]) # State setpoints
    ysp = np.array([0.874317, 0.6528]) # Output setpoint
    usp = np.array([300.157, 0.1]) # Control setpoints
    return [ysp, usp, xsp]
```

Being the considered system a non-linear one, as state estimator an Extended Kalman filter represented by the following matrices is chosen:

$$Q_{kf} = \begin{pmatrix} 10^{-5} I_{nx} & \\ & I_{nd} \end{pmatrix} \qquad R_{kf} = 10^{-4} I_{ny} \qquad P_0 = I_{nx+nd}$$

where $I_{n\xi}$ is the identity matrix of dimension of the variable ξ. In the code the estimator is defined as:

```
ekf = True # Set True if you want the Extended Kalman filter
Qx_kf = 1.0e-5*np.eye(x.size1())
Qd_kf = np.eye(d.size1())
```

```
Q_kf = scla.block_diag(Qx_kf, Qd_kf)
R_kf = 1.0e-4*np.eye(y.size1())
P0 = 1*np.ones((x.size1()+d.size1(),x.size1()+d.size1()))
```

In order for the simulation and the two optimization problems to run the initial state and the bound constraint for the considered variables need to be defined.

```
# Initial conditions
x0_p = np.array([0.874317, 325, 0.6528]) # plant
x0_m = np.array([0.874317, 325, 0.6528]) # model
u0 = np.array([300.157, 0.1])
dhat0 = np.array([0, 0.1])

## Input bounds
umin = np.array([295, 0.00])
umax = np.array([305, 0.25])

## State bounds
xmin = np.array([0.0, 315, 0.50])
xmax = np.array([1.0, 375, 0.75])

# Output bounds
ymin = np.array([0.0, 0.5])
ymax = np.array([1.0, 1.0])

## Disturbance bounds
dmin = -100*np.ones((d.size1(),1))
dmax = 100*np.ones((d.size1(),1))
```

The two objective functions for the optimization problems are chosen to be in a quadratic form, that is for the steady-state optimizer

$$Fss_{obj} = \frac{1}{2}(y - y_{sp})^T Q_{ss}(y - y_{sp})$$

while for the dynamic optimizer

$$F_{obj} = \sum_{i=0}^{i=N-1} \left[\frac{1}{2}(x(i) - x_s)^T Q(x(i) - x_s) + + \frac{1}{2}(u(i) - u_s)^T R(u(i) - u_s) \right]$$

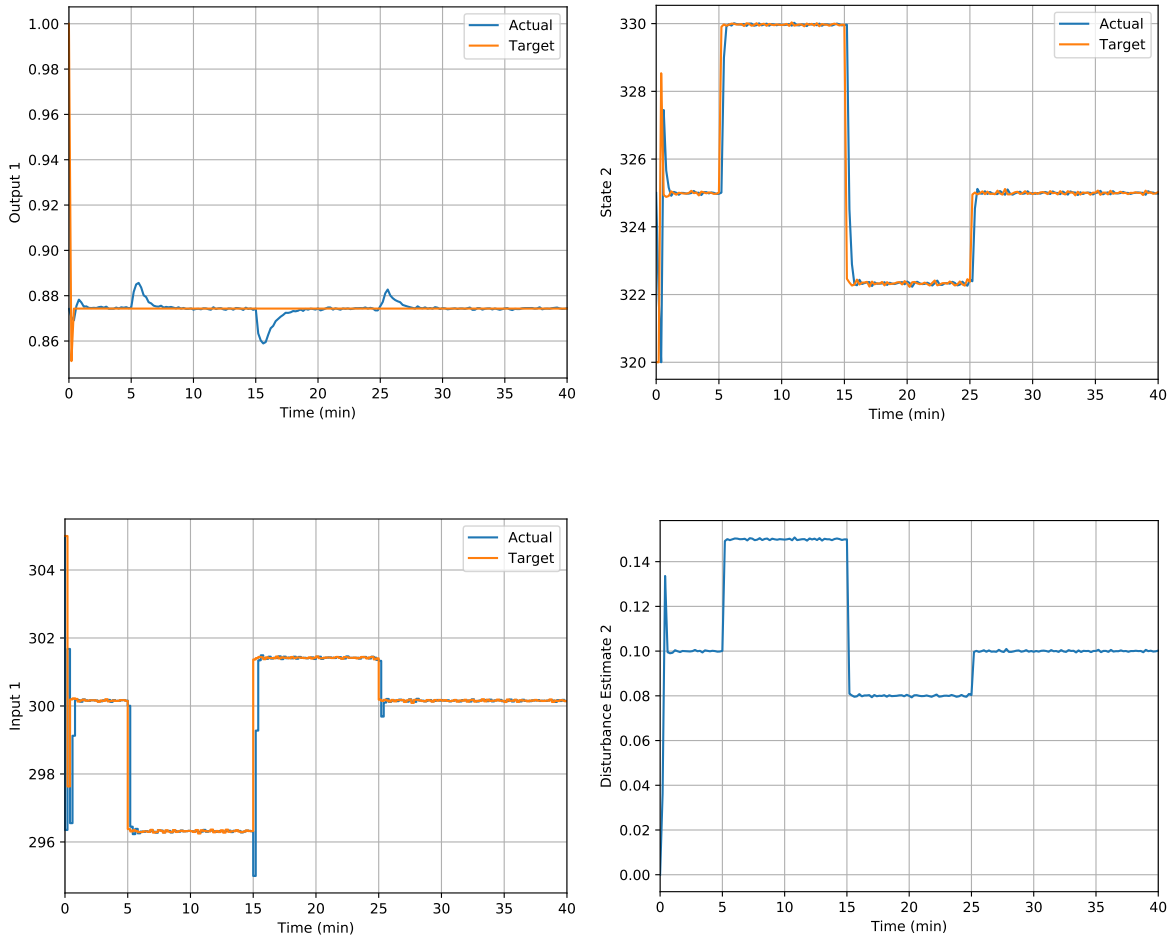
In the code the chosen matrices are defined as follows:

```
## Steady-state optimization
Qss = np.array([[10.0, 0.0], [0.0, 1.0]]) #Output matrix
Rss = np.array([[0.0, 0.0], [0.0, 0.0]]) # Control matrix

## Dynamic optimization
Q = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])
R = np.array([[0.1, 0.0], [0.0, 0.1]])
```

As result of the simulation the following quantities can be analyzed:

- `X_HAT`: it represents the matrix of all the estimated states;
- `U`: it represents the matrix of all the inputs;
- `Xp`: it represents the matrix of all the process states;
- `Yp`: it represents the matrix of all the process output;
- `XS`: it represents the matrix of all the target states;
- `US`: it represents the matrix of all the target inputs;
- `YS`: it represents the matrix of all the target outputs;
- `D_HAT`: it represents the matrix of all the estimated model disturbances;

These quantities can be scanned in the current Python console but they are also displayed in plots like the ones below. Usually, the model states (Actual) and its steady-state (Target) are plotted together, as well as the process output and its target values, and the inputs quantities.
The code prints a plot for each variable considered, e.g. in this case there will be 3 state plots, 2 output plots, 2 input plots and 2 disturbance plots.



2.2 Linear MPC

This example can be followed in the `Ex_LMPC_nlplant.py` file given in the zip file.

The same non-linear system described above in §2.1 is presented, but this time the parameter `F0` is assumed to be fixed: in particular inside the function `User_fxp_Cont(x,t,u,pxp,pxmp)` the `F0` value is substituted with

```
F0 = 0.1   # m^3/min
```

The ODE system is modeled with a discrete-time linear one as follows

$$x^+ - x_{lin} = A(x - x_{lin}) + B(u - u_{lin})$$
$$y = C(x - x_{lin})$$

Let us consider the same simulation parameters used in the previous example and the same dimensions of the problem. The output map is now defined. As before, only the first and the last state are measured so we define the matrix:

```
Cp = np.array([[1.0, 0.0, 0.0], [0.0, 0.0, 1.0]])
```

Then the linear model to be used in the MPC algorithm is defined as:

```
A = np.array([[0.51448, -0.00917517, -0.117995],[53.6817, 2.15004, -3.77725], [0.0,
    0.0, 1]])
B = np.array([[-0.0017669, 0.0864569], [0.639423, 1.60696],  [0.0, -1.32737]])
C = np.array([[1.0, 0.0, 0.0], [0.0, 0.0, 1.0]])

# Linearization parameters
xlin = np.array([0.5, 350, 0.659])
ulin = np.array([300, 0.1])
```

Now an input disturbance model is chosen so the linear system can be rewritten as:

$$x^+ - x_{lin} = A(x - x_{lin}) + B(u - u_{lin}) + Bd$$
$$y = C(x - x_{lin})$$

In the code the disturbance dimension is chosen as before and the matrices `Bd` and `Cd` are implemented as follows:

```
offree = "lin" # set "lin" or "nl" to have a disturbance model linear or non linear. "
    no" means no disturbance model will be implmented
Bd = np.array([[-0.0017669, 0.0864569], [0.639423, 1.60696],  [0.0, -1.32737]])
Cd = np.zeros((y.size1(),d.size1()))
```

Note that in this case the `offree` tag value has been changed to "`lin`".
Setpoints are then defined in a way that the first output will change at time 20 min (note that the total time of simulation is $200 \times 0.2 = 40$ min), while the second one has remain at same value.

```
def defSP(t):
    xsp = np.array([0.0, 0.0, 0.0]) # State setpoints
    usp = np.array([299.963, 0.1]) # Control setpoints

    if t < 20:
        ysp = np.array([0.5, 0.659]) # Output setpoint
    elif t <= 40:
        ysp = np.array([0.51, 0.659]) # Output setpoint

    return [ysp, usp, xsp]
```

As state estimator a Kalman filter with the following matrices is chosen:

$$Q_{kf} = \begin{pmatrix} 10^{-5} I_{nx} & \\ & I_{nd} \end{pmatrix} \qquad R_{kf} = 10^{-4} I_{ny} \qquad P_0 = 10^{-4} Q_{kf}$$

where $I_{n\xi}$ is the identity matrix of dimension of the variable ξ. In the code the estimator is defined in the following way:

```
kal = True # Set True if you want the Kalman filter
nx = x.size1()
ny = y.size1()
nd = d.size1()
Qx_kf = 1.0e-5*np.eye(nx)
Qd_kf = np.eye(nd)
Q_kf = scla.block_diag(Qx_kf, Qd_kf)
R_kf = 1.0e-4*np.eye(ny)
P0 = 1e-3*Q_kf
```

In order to define the simulation and the two optimization problems the initial state and the bound constraints need to be defined.

```
# Initial conditions
x0_p = np.array([0.5, 350, 0.659]) # plant
x0_m = np.array([0.5, 350, 0.659]) # model
u0 = np.array([300, 0.1])

## Input bounds
umin = np.array([295, 0.00])
umax = np.array([305, 0.25])

## State bounds
xmin = np.array([0.0, 320, 0.45])
xmax = np.array([1.0, 375, 0.75])
```

The two objective functions for the optimization problems are the next to be defined. In particular we choose to formulate two QP problems, that is for the steady-state optimizer

$$Fss_{obj} = \frac{1}{2}(y - y_{sp})^T Q_{ss}(y - y_{sp})$$

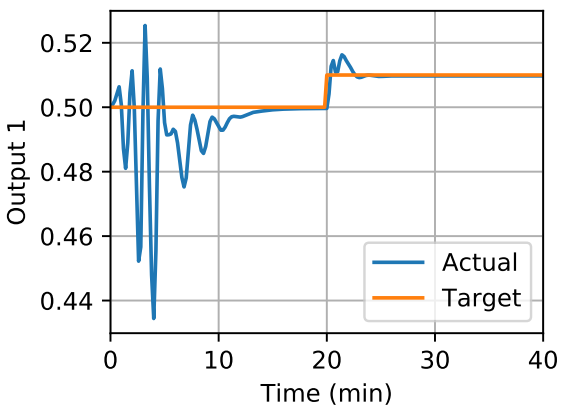
while for the dynamic optimizer

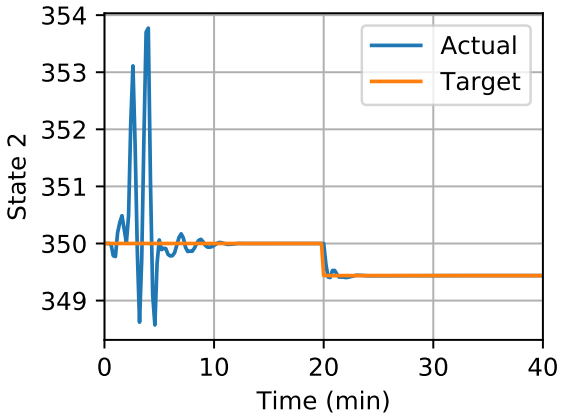
$$F_{obj} = \sum_{i=0}^{i=N-1} \left[\frac{1}{2}(x(i) - x_s)^T Q(x(i) - x_s) + \frac{1}{2}\Delta u(i)^T S \Delta u(i) \right] + \frac{1}{2}(x(N) - x_s)^T P(x(N) - x_s)$$

(where $\Delta u(i) = u(i) - u(i-1)$). In the code the chosen matrices are defined in this way:

```
## Steady-state optimization
Qss = np.array([[10.0,  0.0],  [0.0, 0.01]]) #Output matrix
Rss = np.array([[0.0, 0.0], [0.0, 0.0]]) # Control matrix

## Dynamic optimization
Q = np.array([[10.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])
S = np.array([[0.1, 0.0], [0.0, 0.1]]) # DeltaU matrix
```

Note that in this case (being the considered system linear and the objective function quadratic) the terminal weight for the dynamic problem will be automatically calculated in the code in a quadratic form with a weighting matrix result of the algebraic Riccati equation (see §3.4.4 for details). As in the previous case, the quantities can be scanned in the current Python console but they are also displayed in plots like the ones below. The code will print a plot for each variable considered, e.g. in this case there will be 3 state plots, 2 output plots, 2 input plots and 2 disturbance plots.

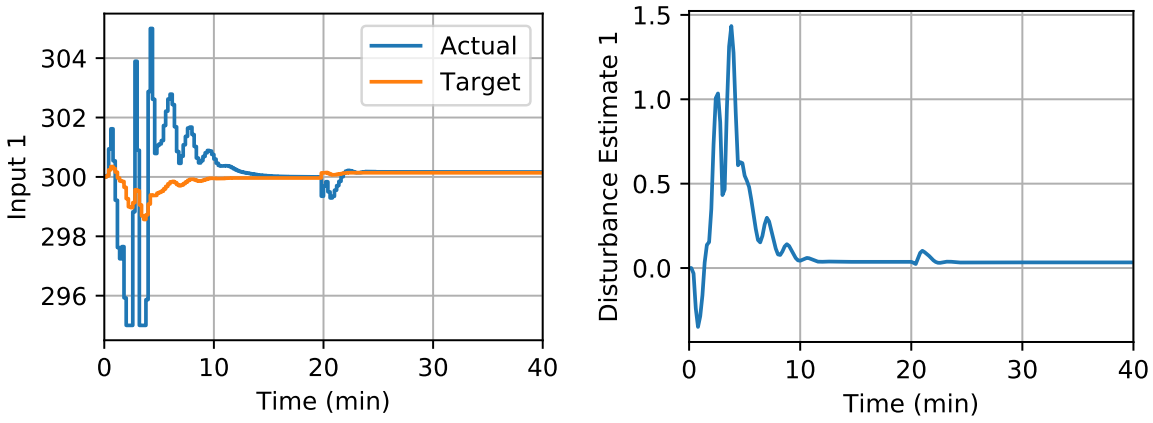




2.3 Economic MPC

This example can be followed in the `Ex_ENMPC.py` file given in the zip file.
A non-linear system described by a ODE system is considered:

$$\dot{x_0} = \frac{u(c_{A0} - x_0)}{V} - k_1 x_0$$



$$\dot{x_1} = \frac{-ux_1}{V} + k_1 x_0 - k_2 x_1$$

Suppose that there is no plant/model mismatch, i.e. `User_fxp_Cont(xp,t,u,pxp,pxmp)` and `User_fxm_Cont(x,u,d,t,px)` are the same. We then assume that all the states are measured, i.e. $y = x$, that is in the code we simply write:

```
StateFeedback = True # Set to True if you have all the states measured
```

without specifying any output map. In addition let us consider some process noise on the state map, i.e.:

$$x_p^+ = F_p(x_p, t, u, px_p, pxm_p) + G_{wn} w$$

where $F_p(x_p, t, u, px_p, pxm_p)$ is the integrated process state map while $w = Q_{wn}^{\frac{1}{2}} w_{rand}$ in which $w_{rand} \in \mathbb{R}^{n_x}$ is a random vector normally distributed with mean "0" and variance "1". In order to define the noise, the covariance and the weighting matrices need to be specified, e.g. $Q_{wn} = 10^{-1} \times I_{nx}, G_{wn} = 10^{-2} \times I_{nx}$. In the code will be:

```
G_wn = 1e-2*np.array([[1.0, 0.0], [0.0, 1.0]]) # State white noise matrix
Q_wn = 1e-1*np.array([[1.0, 0.0], [0.0, 1.0]]) # State white noise covariance matrix
```

Let us then introduce the simulation parameters used: a discretization time step of 2.0 is chosen to build our discrete-time system, and a simulation length of 21 time steps is used. Moreover a prediction horizon of 25 time steps is selected.

```
Nsim = 21 # Simulation length

N = 25    # Horizon

h = 2.0   # Time step
```

The problem dimensions are: 2 states, 1 inputs, 2 ouputs and 2 disturbances.

```
xp = SX.sym("xp", 2) # process state vector        #
x = SX.sym("x", 2)   # model state vector          #
u = SX.sym("u", 1)   # control vector              #
y = SX.sym("y", 2)   # measured output vector      #
d = SX.sym("d", 2)   # disturbance                 #
```

Now an output disturbance model is chosen so the system can be rewritten as:

$$x^+ = F(x, u, t, px)$$
$$y = x + d$$

where $F(x, u, t, px)$ is the integrated model state map. As seen above, in the code only the matrices `Bd` and `Cd` need to be implemented:

```
Bd = np.zeros((d.size1(),d.size1()))
Cd = np.eye(d.size1())
```

As this is an economic MPC example, setpoints are not useful, so there is no need to define them.
This time, being the system non-linear, a Moving Horizon Estimation (MHE) method with a smoothed updating is chosen. Let us define the MHE horizon to be $N_{MHE} = 10$ time steps, and the cost function to be quadratic, i.e.:

$$F_{obj,MHE} = \sum_{i=0}^{i=N_{MHE}} \left[\frac{1}{2} w(i)^T w(i) + \frac{1}{2} v(i)^T v(i) \right] + \frac{1}{2} (x(0) - \bar{x}_0)^T (x(0) - \bar{x}_0).$$

A linear additive noise to the state is considered so that the augmented model map will be:

$$\chi^+ = \tilde{F}(\chi, u, t, px) + w$$

where $\chi = \begin{bmatrix} x \\ d \end{bmatrix}$ and $\tilde{F} = \begin{bmatrix} F \\ d \end{bmatrix}$. In the code we simply define the following lines:

```
    nx = x.size1()
    ny = y.size1()
    nd = d.size1()
    w = SX.sym("w", nx+nd)  # state noise
    P0 = np.eye(nx+nd)
    x_bar = np.row_stack((np.atleast_2d(x0_m).T,np.zeros((nd,1))))

    # Defining the state map
    def User_fx_mhe_Cont(x,u,d,t,px,w):

        x_model = vertcat(u[0]*(cA0 - x[0])/V - k1*x[0], \
                          -u[0]*x[1]/V + k1*x[0] - k2*x[1])

        return x_model

    # Defining the MHE cost function
    def User_fobj_mhe(w,v,t):

        Q = np.eye(nx+nd)
        R = np.eye(ny)
        fobj_mhe = 0.5*(xQx(w,inv(Q))+xQx(v,inv(R)))

        return fobj_mhe
```

Note that in order to define the MHE problem, the definition of the additional variable `w` is required. The problem augmenting is automatically done into the code and so the noise adding.

The initial state and the bound constraint for the considered variables are as follows (note that a mismatch on the initial point is introduced):

```
x0_p = vertcat(0.9, 0.1) # plant
x0_m = vertcat(1.2, 0.5) # model
u0 = vertcat(0.)

## Input bounds
umin = [0.00]
umax = [2.0]

## State bounds
xmin = vertcat(0.00, 0.00)
xmax = vertcat(1.00, 1.00)
```

Then the two objective functions for the optimization problems are defined. The process economics can be expressed by the running cost:

$$\ell(u, y_1) = \beta_A u c_{A0} - \beta_B u y_1$$

where β_A, β_B are the prices for the x_0 and x_1, respectively.
While the target objective function can be formulated without any problem as $\ell(\cdot)$, for the dynamic optimizer there is a clarification to me made. The use of the cost function integrated over the sampling time is necessary to achieve an asymptotically stable closed-loop equilibrium. As a matter of fact, if the point-wise evaluation of $\ell(\cdot)$ was used as stage cost in the dynamic problem, the closed-loop system would not be stable.
To recap, the two objective functions for the optimizations problems are:

$$Fss_{obj} = \ell(u, y_1)$$

and

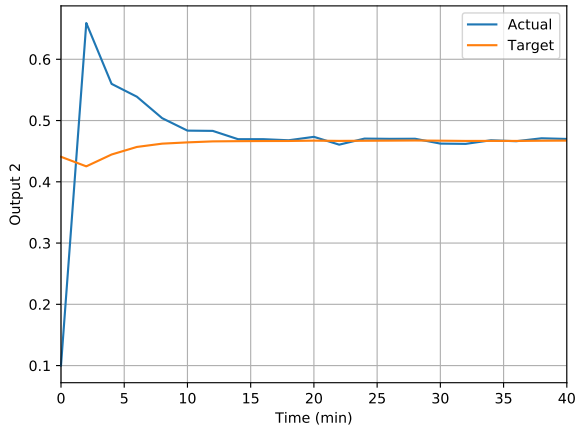
$$F_{obj} = \sum_{i=0}^{i=N-1} \left[\int_{i \cdot h}^{(i+1) \cdot h} \ell(u(t), y_1(t)) dt \right] + \frac{1}{2} x(N)^T P x(N)$$

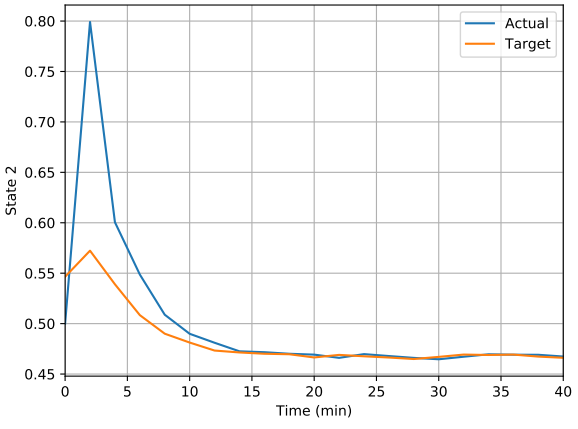
where $P = 2 \cdot 10^3 \times I_{nx}$. In the code we can simply define the following lines:

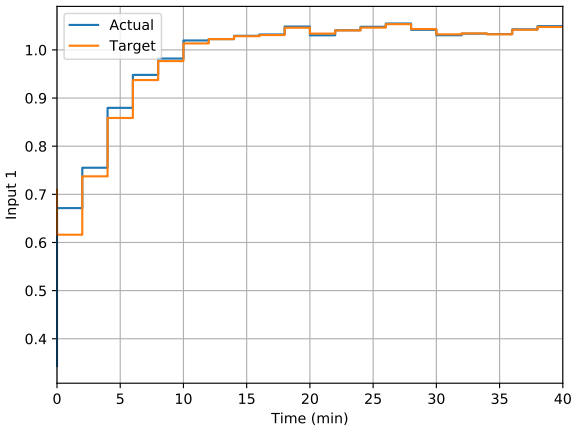
```
cA0 = 1.0  # kmol/m^3
alfa = 1. # reactant price
beta = 4. # product price

## Steady-state optimization
def User_fssobj(x,u,y,xsp,usp,ysp):
    obj = u[0]*(alfa*cA0 - beta*y[1])
    return obj

## Dynamic optimization
def User_fobj_Cont(x,u,y,xs,us,ys):
    obj = u[0]*(alfa*cA0 - beta*y[1])
    return obj

# Terminal weight
def User_vfin(x,xs):
    diffx = x-xs
    vfin = mtimes(diffx.T,mtimes(2000,diffx))
    return vfin
```

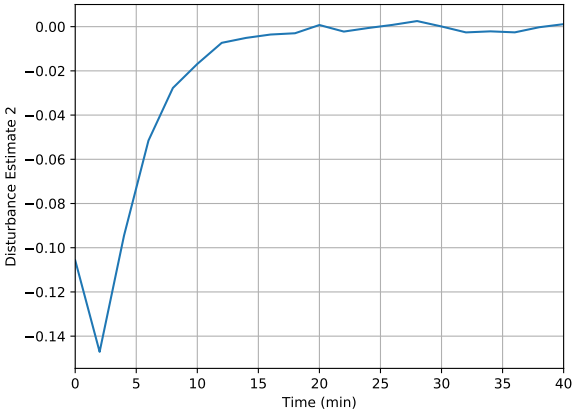
Note how to represent the integrated form of $\ell(\cdot)$, a continuous-time version of the objective function is defined. Note also that in this case (being the considered system non linear and the objective function non quadratic) the terminal weight for the dynamic problem will not be automatically calculated, hence a custom Python function "`User_vfin(x,xs)`" needs to be specified.

The code will print a plot for each variable considered, in this case 2 state plots, 2 output plots, 1 input plot and 2 disturbance plots. Examples are given below.






Actual
Target
Input 1
Time (min)
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
5
10
15
20
25
30
35
40


Disturbance Estimate 2
Time (min)
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
−0.14
0
5
10
15
20
25
30
35
40

Chapter 3

Advanced Usage

Details for an advanced usage of the code are given in this chapter. Every aspect of the code useful for the compilation of a new example is explained here. The user can explore better what are all the possible options to build a custom example. This section is developed as follows: firstly, the definition of the plant dynamics, which produces measurements, and the corresponding model dynamics are introduced; then, the main blocks that constitute the MPC algorithm are detailed (state estimation, steady-state and dynamic optimizer); in the end, further options and plotting information are given.

3.1 Simulation Fundamentals

3.1.1 Simulation discretization parameters

These are the three simulation parameters that are always to be defined in any example file.

- `Nsim` : the simulation length;
- `N` : the prediction horizon length;
- `h` : the discretization time step. Note that if you are already in discrete-time just insert the value “1.0".

All the previous time lengths are expressed in units consistent with the example studied.

3.1.2 Symbolic variables

Being the MPC CODE built with a CasADi interface, you have to define some basic symbolic variables. These variables are used for function evaluations over the simulation horizon: set here the dimension of each single variable as your specific example requires.

- `xp` : the state process vector ($\in \mathbb{R}^{n_{xp}}$);
- `x` : the state model vector ($\in \mathbb{R}^{n_x}$);
- `u` : the control action vector ($\in \mathbb{R}^{n_u}$);
- `y` : the measured output vector ($\in \mathbb{R}^{n_y}$);
- `d` : the disturbance model vector ($\in \mathbb{R}^{n_d}$);
  If no disturbance model is selected you have to enter the value “0"
- `px` : the parameter for the state model map ($\in \mathbb{R}^{n_{px}}$);
- `py` : the parameter for the output model map ($\in \mathbb{R}^{n_{py}}$);
  `px` and `py` are to be defined only when `LinPar` is set to `False`

- `pxp` : the parameter for the state process map ($\in \mathbb{R}^{n_{pxp}}$);
- `pyp` : the parameter for the output process map ($\in \mathbb{R}^{n_{pyp}}$);
- `w` : the state update model noise.
  This variable is used only when `mhe` is selected as estimator (see §3.3 for further details)

3.2 Process and Model definition

3.2.1 Process definition

All the possible dynamics that can be chosen for the process equations are now explained.

1. **Linear.** If the process equations are linear (and time-invariant), then the following matrices need to be defined:

   - `Ap` : the state transition matrix ($\in \mathbb{R}^{n_{x_p} \times n_{x_p}}$);
   - `Bp` : the input-state matrix ($\in \mathbb{R}^{n_{x_p} \times n_u}$);
   - `Cp` : the state-output matrix ($\in \mathbb{R}^{n_y \times n_{x_p}}$); ;

   Note that the feedthrough matrix `Dp` $\in \mathbb{R}^{n_y \times n_u}$ is not considered, i.e. the output `y` is not function of the input `u`.

2. **Non Linear.** If the process equations are general (non-linear and/or time-variant), then the functions explained below have to be defined. Note that the syntax in these functions has to be consistent with the CasADi language:

   - State map.
     (a) `Continuous-time`: Define the function "`User_fxp_Cont(x,t,u,pxp,pxmp)`". The input "`t`" gives the possibility to introduce time-variant parameters and/or dynamics. The output describes the state dynamic to be integrated[1], i.e. $\dot{x} = User\_fxp\_Cont(x,t,u,px_p,pxm_p)$; for this purpose, the user has to define also the parameter `Mx` that is necessary for the explicit Runge-Kutta integration: it represents the number of elements in each integration step;
     (b) `Discrete-time`: Define the function "`User_fxp_Dis(x,t,u,pxp,pxmp)`". The input "`t`" gives the possibility to introduce time-variant parameters and/or dynamics. The output describes the discrete state dynamic, i.e. $x = User\_fxp\_Dis(x,t,u,px_p,pxm_p)$.
   - Output map. Define the function "`User_fyp(x,u,t,pyp,pymp)`". The input "`t`" gives the possibility to introduce time-variant parameters and/or dynamics. The output describes the map, i.e. $y = User\_fyp(x,u,t,py_p,pym_p)$.

White noise

Another option that can be introduced is represented by process and measurement noise. The white noise is modeled as described below.

$$x^+ = f_p(x_p,t,u,px_p,pxm_p) + G_{wn}w, \qquad y = h_p(x_p,u,t,py_p,pym_p) + v$$

where

$$w = Q_{wn}^{\frac{1}{2}} w_{rand} \quad \text{and} \quad v = R_{wn}^{\frac{1}{2}} v_{rand}$$

The matrices to be defined are then as follows:

[1] At the moment only the Multiple Shooting integration scheme is implemented

- `G_wn` : the state white noise matrix ($\in \mathbb{R}^{n_{xp} \times n_{xp}}$);
- `Q_wn` : the state white noise covariance matrix ($\in \mathbb{R}^{n_{xp} \times n_{xp}}$);
- `R_wn` : the output white noise covariance matrix ($\in \mathbb{R}^{n_y \times n_y}$).

Note that $w_{rand} \in \mathbb{R}^{n_x}$ and $v_{rand} \in \mathbb{R}^{n_y}$ are random vector normally distributed with zero mean and unitary variance. In addition, note that all the outputs, being measured, are affected by noise, that is, the output white noise matrix is just an identity matrix $\in \mathbb{R}^{n_y \times n_y}$.

Process parameters

Process parameters can describe various modeling needs, e.g. process disturbances deriving from different sources. In the code two classes of parameters can be defined within the example file: “measurable" and “non-measurable".

Process parameters are added to the process equations in continuous-time or discrete-time form (linear or not) and can be included into the state dynamics and/or to the output one. This is possible by defining the following functions:

- *Non-Measurable Parameters*
  - `def_pxp(t)`: for the state dynamics through the Python function “$px_p = def\_px_p(t)$"
  - `def_pyp(t)`: for the output dynamics through the Python function “$py_p = def\_py_p(t)$"
- *Measurable Parameters*
  - `def_pxmp(t)`: for the state dynamics through the Python function “$pxm_p = def\_pxm_p(t)$"
  - `def_pymp(t)`: for the output dynamics through the Python function “$pym_p = def\_pym_p(t)$"

The input t gives the opportunity to formulate a time-variant parameter.
To set the parameters as a generic (time-invariant) input of the model maps, the tag `LinPar` must be set as `False`, and the parameters have to be initialized as symbolic variables by specifying their dimensions.

Summary

The user can choose any combination of the previous options keeping in mind that just one state dynamics and one output must be selected. For instance, the following are some possible combinations:

- $x = A_p x + B_p u, \qquad y = C_p x$
- $x = f_p(x,t,u,px_p,pxm_p) + pxm_p,$
  $y = h_p(x,u,t,py_p,pym_p)$
- $x = f_p(x,t,u,px_p,pxm_p),$
  $y = C_p x + pym_p$
- $x = A_p x + B_p u + G_{wn} w,$
  $y = h_p(x,u,t,py_p,pym_p) + v$
- $x = f_p(x,t,u,px_p,pxm_p),$
  $y = C_p x + v + py_p$
- $x = A_p x + B_p u + px_p + pxm_p,$
  $y = h_p(x,u,t,py_p,pym_p)$

3.2.2 Model definition

The following options for the model equations are now explained.

1. **Linear.** If the model equations are linear (and time-invariant), the following matrices have to be defined:
   - `A` : the state matrix ($\in \mathbb{R}^{n_x \times n_x}$);
   - `B` : the input-state matrix ($\in \mathbb{R}^{n_x \times n_u}$);

- `C` : the state-output matrix ($\in \mathbb{R}^{n_y \times n_x}$);

The model matrices can also represent linearization matrices of a non-linear model map. In this case the user has to define the linearization point:

- `xlin` : the state linearization vector ($\in \mathbb{R}^{n_x}$);
- `ulin` : the control linearization vector ($\in \mathbb{R}^{n_u}$);
- `ylin` : the output linearization vector ($\in \mathbb{R}^{n_y}$);

2. **Non Linear.** If the model equations are general (non-linear and/or time-variant), then the following functions need to be defined. It has to be noted that the syntax in these functions has to be consistent with the CasADi language:
   - State map.
     (a) `Continuous-time`: Define the function "`User_fxm_Cont(x,u,d,t,px)`". The output describes the state dynamic to be integrated[2], i.e. $\dot{x} = User\_fxm\_Cont(x,u,d,t,px)$; for this purpose, the user has to define also the parameter `Mx` that is necessary for the explicit Euler integration: it represents the number of elements in each integration step;
     (b) `Discrete-time`: Define the function "`User_fxm_Dis(x,u,d,t,px)`". The output describes the discrete state dynamic,i.e. $x = User\_fxm\_Dis(x,u,d,t,px)$;
   - Output map. Define the function "`User_fym(x,u,d,t,py)`". The output describes the integrated output dynamic, i.e. $y = User\_fym(x,u,d,t,py)$.

As for the process dynamics the variable `t` can be used for time-varying parameters and/or dynamics, while variable `d` is used to implement directly a generic disturbance model, when selected (see §3.2.3).

Measurable parameters

When process parameters are "measurable" they can be considered into the model dynamics as done for the process dynamics. Hence, as well as the process ones, model measurable parameters are added to the model equations in continuous-time or discrete-time form (linear or not) and can be added to the state dynamics and/or to the output ones. This is possible by defining the following functions:

- `def_px(t)`: for the state dynamics through the Python function "$px = def\_px(t)$"
- `def_py(t)`: for the output dynamics through the Python function "$py = def\_py(t)$"

The input `t` gives the opportunity to formulate time-variant parameters. Note that, unlike for the process dynamic, in this case the time-varying parameter is passed to the dynamic optimization problem. This means that, according to its function definition, the parameter can change along the prediction horizon.

If the tag `LinPar` is set to `True` and the function `def_px(t)` or `def_py(t)` is defined in the example file, these parameters are linearly added to the discrete-time dynamics they perturb. , i.e. $px \in \mathbb{R}^{nx}$ and $py \in \mathbb{R}^{ny}$. The default value of `LinPar` is set to `True`.

When model parameters are considered, they need to be defined both for the model and the process dynamics. If measurable process parameters dynamics are supposed perfectly known, e.g. $px = pxm_p$, the user can define only the function `def_px(t)` and/or `def_py(t)` and the corresponding process measurable parameters are automatically defined, i.e. `def_pxmp(t)` = `def_px(t)` and/or `def_pymp(t)` = `def_py(t)`.

Summary

The user can choose any combination of the previous model dynamics keeping in mind that just one state transition map and one output map must be selected. For instance, the following are some possible choices:

[2] At the moment only the Multiple Shooting integration scheme is implemented

- $x = Ax + Bu, \qquad y = Cx$
- $x = f(x,u,d,t,px) + px, \qquad y = Cx$
- $x = f(x,u,d,t,px)$,
  $y = h(x,u,d,t,py) + py$
- $x = Ax + Bu, \qquad y = h(x,u,d,t,py)$

3.2.3 Disturbance model for Offset-free control

To select a disturbance model, you have to set the tag "**offree**" as displayed below:

- `offree = "no"`: no disturbance model is implemented (default option);
- `offree = "lin"`: linear disturbance model is adopted. In this case, the augmented model is as follow:
  $$x = F(x,u,t,px) + B_d d, \qquad y = H(x,u,t,py) + C_d d$$
  where $F(x,u,t,px)$ is any integrated state transition map and $H(x,u,t,py)$ any output map linear and/or non-linear . The matrices to be defined are:
  - `Bd` : the state disturbance matrix ($\in \mathbb{R}^{n_x \times n_d}$);
  - `Cd` : the output disturbance matrix ($\in \mathbb{R}^{n_y \times n_d}$).
- `offree = "nl"`: non-linear disturbance model is adopted. Note that in this case the user has to insert the disturbance dynamics `d` into the non-linear model dynamics described by `User_fxm_Cont(x,u,d,t)`, `User_fxm_Dis(x,u,d,t,px)`, or `User_fym(x,u,d,t,py)`.

Warning: if no disturbance model is selected, then `d` dimension must be "0"; vice-versa, if the disturbance `d` dimension is "0", then the user must set `offree = "no"`. If this condition is not fulfilled, an error arises.

3.2.4 Initial condition

The user can set different initial conditions for process and model dynamics in order to give more generality to the code.

- `x0_p` : the state initial condition for the process ($\in \mathbb{R}^{n_{x_p}}$);
- `x0_m` : the state initial condition for the model ($\in \mathbb{R}^{n_x}$);
- `u0` : the control action initial condition ($\in \mathbb{R}^{n_u}$);
- `dhat0` : the disturbance model initial condition ($\in \mathbb{R}^{n_d}$) (**optional**; default value is $\mathbf{0}^{n_d}$).

3.3 State estimation

The state estimators implemented in the code are now illustrated. To set an estimator, the corresponding tag has to be set to "`True`".

- **Kalman Filter** [tag: "`kal`"] : for the Kalman filter the matrices to be defined are three:
  - `Q_kf` : the process noise covariance matrix ($\in \mathbb{R}^{n_x+n_d \times n_x+n_d}$);
  - `R_kf` : the measurements noise covariance matrix ($\in \mathbb{R}^{n_y \times n_y}$);
  - `P0` : the covariance of the state error at the initial point ($\in \mathbb{R}^{(n_x+n_d)\times(n_x+n_d)}$).
- **Steady-state Kalman Filter** [tag: "`kalss`"] : for the steady-state Kalman filter the matrices and vectors to be defined are:
  - `Q_kf` : the process noise covariance matrix ($\in \mathbb{R}^{(n_x+n_d)\times(n_x+n_d)}$);
  - `R_kf` : the measurements noise covariance matrix ($\in \mathbb{R}^{n_y \times n_y}$);

  - Steady-state point: this has to be defined only if "A" and/or "C" have not been previously defined for the model (see §3.2.2):
    * x_ss : the steady-state state vector at which the estimator gain has to be calculated ($\in \mathbb{R}^{n_x}$);
    * u_ss : the steady-state control action vector at which the estimator gain has to be calculated ($\in \mathbb{R}^{n_u}$);
    * px_ss : the steady-state state parameter vector at which the estimator gain has to be calculated ($\in \mathbb{R}^{n_{px}}$);
    * py_ss : the steady-state output parameter vector at which the estimator gain has to be calculated ($\in \mathbb{R}^{n_{py}}$).

- **Extended Kalman Filter** [tag: "ekf"] : for the extended Kalman filter the matrices to be defined are the same three of the Kalman Filter:
  - Q_kf : the process noise covariance matrix ($\in \mathbb{R}^{(n_x+n_d)\times(n_x+n_d)}$);
  - R_kf : the measurements noise covariance matrix ($\in \mathbb{R}^{n_y \times n_y}$);
  - P0 : the covariance of the state error at the initial point ($\in \mathbb{R}^{(n_x+n_d)\times(n_x+n_d)}$).

- **Luenberger observer** [tag: "lue"] : for the Luenberger observer just one matrix has to be specified as:
  - K : the estimator gain matrix ($\in \mathbb{R}^{(n_x+n_d)\times n_y}$).
    As an example, one could define two sub-matrices, one for state and one for disturbance update, as $K = [K_x,\ K_d]^T$, where $K_x \in \mathbb{R}^{n_x \times n_y}$ and $K_d \in \mathbb{R}^{n_d \times n_y}$. These sub-matrices are connected to the typical industrial estimator design for stable systems (e.g., when matrix A is strictly Schur), that is, adopting a gain K in combination with the so-called *Output disturbance model* ($B_d = 0$, $C_d = I$, $K_x = 0$, $K_d = I$) or *Input disturbance model* ($B_d = B$, $C_d = 0$).

- **Moving Horizon Estimation** [tag: "mhe"] : for the Moving Horizon Estimation technique, there are several parameters/functions to be defined:
  - P0 : the covariance of the state error at the initial point ($\in \mathbb{R}^{(n_x+n_d)\times(n_x+n_d)}$);
  - x_bar : the *a priori* state estimate the initial point ($\in \mathbb{R}^{(n_x+n_d)}$);
  - N_mhe : the MHE horizon length ($\in \mathbb{R}$);
  - mhe_up : a flag to decide which prior weight updating technique has to be used in the MHE problem. It can be set with two tags:
    * "filter" : the filtering update will be applied
    * "smooth" : the smoothing update will be applied
  - **Model Function**: in order to build a general implementation of noise w, a specific function including the state evolution map has to be defined. Note that the syntax in this function has to be consistent with the CasADi language:
    * Continuous-time: Define the function "User_fx_mhe_Cont(x,u,d,t,px,w)". The output describes the state dynamic to be integrated[3],i.e. $\dot{x} = User\_fx\_mhe\_Cont(x,u,d,t,px,w)$; At this purpose, the user has to define also the parameter Mx that is necessary for the explicit Euler integration: it represents the number of elements in each integration step;
    * Discrete-time: Define the function "User_fx_mhe_Dis(x,u,d,t,px,w)". The output describes the discrete state dynamic,i.e. $x = User\_fx\_mhe\_Dis(x,u,d,t,px,w)$;

    Note that the measurement noise v is always considered linearly added to the output map, whatever the latter is defined, i.e. $y = h(x,d,t,py) + v$. One more remark has to be done about using

[3] At the moment only the Multiple Shooting integration scheme is implemented

MHE with linear and non-linear offset-free model. In this case the noise `w` is added to the augmented model map as follows:

$$\begin{cases} x^+ = F(x,u,t,px) + B_d d \\ \qquad d^+ = d \end{cases} \quad \text{if } \texttt{offree = "lin"}$$
$$\begin{cases} x^+ = F(x,u,d,t,px) \\ \qquad d^+ = d \end{cases} \quad \text{if } \texttt{offree = "nl"}$$
$$\rightarrow \quad \chi^+ = \tilde{F}(\chi,u,t,px) + w$$

where $F(\cdot)$ is any integrated state transition map, $\chi = \begin{bmatrix} x \\ d \end{bmatrix}$ and $\tilde{F} = \begin{bmatrix} F \\ d \end{bmatrix}$.

- **Objective Function**: As for the other optimization problems, there are three possibilities to define the objective function for the MHE problem:
  1. **Linear.** If the objective function is linear, define the following vectors:
     - `r_w` : the vector weight on the state ($\in \mathbb{R}^{n_w}$);
     - `r_v` : the vector weight on the control ($\in \mathbb{R}^{n_y}$).

     Defined the appropriate vectors, the linear objective function is:
     $$F_{obj,mhe} = r_w^T w + r_v^T v$$
  2. **Quadratic.** If the objective function is quadratic, define the following matrices:
     - `Q_mhe` : the matrix weight on the state noise ($\in \mathbb{R}^{n_w \times n_w}$);
     - `R_mhe` : the matrix weight on the measurements noise ($\in \mathbb{R}^{n_y \times n_y}$);

     Defined the appropriate matrices, the quadratic objective function is:
     $$F_{obj,mhe} = \frac{1}{2} w^T Q_{mhe} w + \frac{1}{2} v^T R_{mhe} v$$
  3. **Non Linear.** If a non linear objective function is desired, define the following Python function:"`User_fobj_mhe(w,v,t)`". The output describes the discrete-time cost function; "`w, v, t`" represent the process noise, the measurement noise and the time index respectively. Note that the syntax in this function has to be consistent with the CasADi language.

- **Bound Constraints**: Is it possible to add bound constraints on the two optimization variables, "`w, v`" that can be written as:
  - `wmin` : the lower bound for the process noise vector ($\in \mathbb{R}^{n_w}$);
  - `wmax` : the upper bound for the process noise vector ($\in \mathbb{R}^{n_w}$);
  - `vmin` : the lower bound for the measurement noise vector ($\in \mathbb{R}^{n_y}$);
  - `vmax` : the upper bound for the measurement noise vector ($\in \mathbb{R}^{n_y}$);

  When these quantities are not defined they are automatically set to infinite (positive or negative) values.

3.4 Steady-state and dynamic optimizers

3.4.1 Setpoints

Setpoints can be defined within the function "`defSP(t)`" that takes as input the current time index. The function input `t` gives the possibility to study a time-varying setpoint problem. The output of this function has to be a list composed in this order [`ysp`, `usp`, `xsp`], where:

- `ysp` : the output setpoint vector ($\in \mathbb{R}^{n_y}$);
- `usp` : the control action setpoint vector ($\in \mathbb{R}^{n_u}$);
- `xsp` : the state setpoint vector ($\in \mathbb{R}^{n_x}$).

Note that even though `xsp` is not used into the chosen target problem, it has to be defined anyway.

3.4.2 Bounds constraints

As the MPC structure has two, possibly (and more realistically) constrained, optimization problems, bound constrains need to be specified when they are not infinite. To define *hard* bounds for the specific variables, the following syntax has to be respected:

- `umin` : the lower bound for the input vector ($\in \mathbb{R}^{n_u}$);
- `umax` : the upper bound for the input vector ($\in \mathbb{R}^{n_u}$);
- `xmin` : the lower bound for the state vector ($\in \mathbb{R}^{n_x}$);
- `xmax` : the upper bound for the state vector ($\in \mathbb{R}^{n_x}$);
- `ymin` : the lower bound for the output vector ($\in \mathbb{R}^{n_y}$);
- `ymax` : the upper bound for the output vector ($\in \mathbb{R}^{n_y}$);
- `dmin` : the lower bound for the disturbance vector ($\in \mathbb{R}^{n_d}$);
- `dmax` : the upper bound for the disturbance vector ($\in \mathbb{R}^{n_d}$);
- `Dumin` : the lower bound for the input rate-of-change vector ($\in \mathbb{R}^{n_u}$);
- `Dumax` : the upper bound for the input rate-of-change vector ($\in \mathbb{R}^{n_u}$);

The definition of *soft* constraints, slack variables and corresponding weights for bound violations is not yet fully implemented in the code but there is a workaround to solve this limitation; as a matter of fact, the user can define different bounds (for `x`, `u`, and `y`) for the target and the dynamic optimization problems. Specifically, the syntax to be used is as follows:

- Steady-state optimization:
  - `umin_ss` and `umax_ss` $\in \mathbb{R}^{n_u}$;
  - `xmin_ss` and `xmax_ss` $\in \mathbb{R}^{n_x}$;
  - `ymin_ss` and `ymax_ss` $\in \mathbb{R}^{n_y}$;
- Dynamic optimization:
  - `umin_dyn` and `umax_dyn` $\in \mathbb{R}^{n_u}$;
  - `xmin_dyn` and `xmax_dyn` $\in \mathbb{R}^{n_x}$;
  - `ymin_dyn` and `ymax_dyn` $\in \mathbb{R}^{n_y}$;

The notation with the subscripts "`_ss`" or "`_dyn`" is always more important than the one without, that is, if for example the following is written in the user file for the `y` bounds:

$$\texttt{ymin} = [0,0,0];\ \texttt{ymax} = [1,1,1];\ \texttt{ymax\_dyn} = [1.2,1,1.5]$$

than the constraint to be fulfilled in the steady-state problem is $y_{\min} \leq y \leq y_{\max}$ while in the dynamic problem is: $y_{\min} \leq y \leq y_{\max,dyn}$.

Not specified bounds are automatically set to positive or negative infinite values respectively.

3.4.3 Steady-state optimization: objective function

1. **Linear.** If the objective function is linear, the following vectors need to be defined:

   - `rss_y` : the output cost vector ($\in \mathbb{R}^{n_y}$);
   - `rss_u` : the control cost vector ($\in \mathbb{R}^{n_u}$);
   - `rss_Du` : the control difference cost vector ($\in \mathbb{R}^{n_u}$).

   `rss_u` or `rss_Du` must be used alternatively. Defined the appropriate vectors, the objective function will be linear and in particular in one of the two following forms:

   $$Fss_{obj} = r_{ss,y}^T y + r_u^T u$$

   or

   $$Fss_{obj} = r_{ss,y}^T y + r_{ss,Du}^T |(u - u_s(k-1))|$$

   where $u_s(k-1)$ is the steady-state input value calculated at the previous iteration.

2. **Quadratic.** If the objective function is quadratic, the following matrices need to be defined:

   - `Qss` : the output cost matrix ($\in \mathbb{R}^{n_y \times n_y}$);
   - `Rss` : the control action cost matrix ($\in \mathbb{R}^{n_u \times n_u}$);
   - `Sss` : the control difference cost matrix ($\in \mathbb{R}^{n_u \times n_u}$).

   `Rss` or `Sss` must be used alternatively. Defined the appropriate matrices, the objective function will be quadratic and in particular in one of the two following forms:

   $$Fss_{obj} = \frac{1}{2}(y - y_{sp})^T Q_{ss}(y - y_{sp}) + \frac{1}{2}(u - u_{sp})^T R_{ss}(u - u_{sp})$$

   or

   $$Fss_{obj} = \frac{1}{2}(y - y_{sp})^T Q_{ss}(y - y_{sp}) + \frac{1}{2}\Delta u^T S_{ss}\Delta u$$

   where $\Delta u = u - u_s(k-1)$

3. **Non Linear.** The Python function "`User_fssobj(x,u,y,xsp,usp,ysp)`" has to be defined when a non linear objective cost is desired. The output describes the discrete-time cost function; "`xsp`, `usp`, `ysp`" represent the setpoint values calculated by the function `defSP(t)` previously defined. Note that the syntax in this function has to be consistent with the CasADi language

3.4.4 Dynamic optimization: objective function

1. **Linear.** If the objective function is linear, the following vectors need to be defined:

   - `r_x` : the state cost vector ($\in \mathbb{R}^{n_x}$);
   - `r_u` : the control action cost vector ($\in \mathbb{R}^{n_u}$);
   - `r_Du` : the control difference cost vector ($\in \mathbb{R}^{n_u}$).

   `r_u` or `r_Du` must be used alternatively. Defined the appropriate vectors, the objective function will be linear and in particular in one of the two following forms:

   $$F_{obj} = \sum_{i=0}^{i=N-1} r_x^T |x(i) - x_s| + r_u^T |u(i) - u_s|$$

   or

   $$F_{obj} = \sum_{i=0}^{i=N-1} r_x^T |x(i) - x_s| + r_{Du}^T |(u(i) - u(i-1))|$$

2. **Quadratic.** If the objective function is quadratic, the following matrices need to be defined:

   - `Q` : the state cost matrix ($\in \mathbb{R}^{n_x \times n_x}$);
   - `R` : the control action cost matrix ($\in \mathbb{R}^{n_u \times n_u}$);
   - `S` : the control difference matrix ($\in \mathbb{R}^{n_u \times n_u}$).

   `R` or `S` must be used alternatively. Defined the appropriate matrices, the objective function will be quadratic and in particular in one of the two following forms:

$$F_{obj} = \sum_{i=0}^{i=N-1} \frac{1}{2}(x(i)-x_s)^T Q(x(i)-x_s) + \frac{1}{2}(u(i)-u_s)^T R(u(i)-u_s)$$

   or

$$F_{obj} = \sum_{i=0}^{i=N-1} \frac{1}{2}(x(i)-x_s)^T Q(x(i)-x_s) + \frac{1}{2}\Delta u(i)^T S \Delta u(i)$$

   where $\Delta u(i) = u(i) - u(i-1)$.

3. **Non Linear.** If a non-linear cost function is desired, a Python function has to be defined. Note that the syntax in these functions has to be consistent with the CasADi language. The non linear function can be chosen to be:

   - `Continuous-time`: Define: "`User_fobj_Cont(x,u,y,xs,us,ys)`". The output describes the cost function to be integrated[4];
   - `Discrete-time`: Define: "`User_fobj_Dis(x,u,y,xs,us,ys)`". The output describes the discrete-time cost function;

   "`xs, us, ys`" represent the target values that come as result from the steady-state optimization.

Terminal weight

To define the terminal weight for the dynamic objective cost function the following rule is applied. If and only if the state dynamic map is linear and the dynamic objective function is quadratic, then the terminal weight is defined as $v_{fin} = \frac{1}{2}(x(N)-x_s)^T P(x(N)-x_s)$ where $P \in \mathbb{R}^{n_x \times n_x}$ and calculated through the Algebraic Riccati Equation. If a custom terminal weight is desired, the Python function "`User_vfin(x,xs)`" must be defined. In any other case, the terminal weight is set to "0.0".

3.4.5 Time-varying generic constraints

To add generic Time-varying constraints, the steady-state and the dynamic optimization problems are equipped with additional equality and/or inequality constraints vectors defined through time-varying functions in the example file.

External constraints can derive from different sources and can be referred to any system parameter, thus it is necessary to distinguish the constraints referring to steady-state optimization and those referring to dynamic optimization defining the following functions.

- *Steady-state optimization*

  - `User_g_ineq_SS(x,u,d,t,px)`: for the inequality constraint vector $g_{ss} \leq 0$; e.g. $g_{ss} = x - x_{max}$ or $g_{ss} = x_{min} - x$ to define an upper bound or a lower bound constraint that varies along the MPC iterations.
  - `User_h_eq_SS(x,u,d,t,px)`: for the equality constraint vector $h_{ss} = 0$.

[4] `cvodes` from CasADi is used

- *Dynamic optimization*

  - `User_g_ineq(x,u,d,t,px)`: for the inequality constraint vector $g \leq 0$; e.g. $g_{ss} = x - x_{max}$ or $g_{ss} = x_{min} - x$ to define an upper bound or a lower bound constraint that varies along the MPC iterations.
  - `User_h_eq(x,u,d,t,px)`: for the equality constraint vector $h = 0$.

Warning: the vectors g and h must be initialized in the example file with an SX variable.

The input `t` gives the opportunity to formulate a time-variant constraint, the inputs `x, u, d`, and `px` allow to obtain a constraint function of the properties of the system, and of external parameters to the optimization problem.

3.4.6 Solver options

There is the possibility to pass some options to the solver `IPOPT` implemented in the code.

- `Sol_itmax` = tot: specifies the maximum tot number of iterations made by the solver, i.e. `Sol_itmax` = 40 means 40 iterations are the maximum. Default value is set to 100.
- `Sol_Hess_constss`: means that the Hessian matrix of the steady-state optimization problem is constant. To set the option on, write `Sol_Hess_constss = True`. If and only if both the model maps (state and output) are linear and the steady-state problem is quadratic, the option is automatically set to `True`. The default value is `False`.
- `Sol_Hess_constdyn`: means that the Hessian matrix of the dynamic optimization problem is constant. To set the option on, write `Sol_Hess_constdyn = True`. If and only if both the model maps are linear and the dynamic problem is quadratic, the option is automatically set to `True`. The default value is `False`.

3.4.7 Collocation methods

An optimal solving structure of the dynamic optimization problem that can be used in the code, alternatively to the multiple shooting method adopted by default, is included in the family of Collocation Methods. The collocation method implemented in the code is the Gauss-Legendre 4 exploiting the state representative framework with two additional internal states. The user/reader interested to the theory of the implemented method can refer to Chapter 8.5 of *Rawlings, J.B., Mayne, D.Q. and Diehl, M., 2017. Model predictive control: theory, computation, and design (Vol. 2). Madison, WI: Nob Hill Publishing.* Clearly, this method is available only when there is a continuous-time formulation of the problem.

For the Collocation methods technique, the tag `Collocation` must be set as `True` and the additional optimization variables $s1$ and $s2$ are initialized as a vector of symbolic variables internally, i.e.:

- `s_Coll = SX.sym("s_Coll",2*x.size1())`

To define the objective function the user must define the function with the syntax consistent with the CasADi language, i.e.:

- `User_fobj_Coll(x,u,y,xs,us,ys,s_Coll)`

Note that the input *s_Coll* allows one to choose an objective function in which the internal states are weighted.

3.5 Further options

In this section some additional useful options are described. To activate them, you have to set them as `True`. These options are:

- `StateFeedback`: This option automatically sets the output dynamic map as: $H_p(x,t,u) = x_p$ for the process and $H(x,u) = x$ for the model. In this case, the estimator will be set automatically to a Luenberger observer with a gain equal to the identity matrix.
- `Fp_nominal`: This option automatically sets the process dynamic maps equal to the ones defined for the model. This means: $F_p(x,u) = F(x,u)$ and $H_p(x,u) = H(x,u)$.
- `QForm_ss`: This option sets that the optimization variables entering the steady-state objective function are reduced by their setpoints. It means that $y - y_{sp}$ and $u - u_{sp}$ are the new optimization variables considered;
- `QForm`: This option sets that the optimization variables entering the dynamic objective function are reduced by the steady states. It means that $x(i) - x_s$ and $u(i) - u_s$ are the new optimization variables considered.
- `DUForm`: This option activates the $\Delta u(i) = u(i) - u(i-1)$ variable instead of $u(i)$ in the dynamic optimization problem even when the objective function is non-linear.
- `DUssForm`: This option activates the $\Delta u = u - u_s(k-1)$ variable instead of u in the steady-state optimization problem even when the objective function is non-linear.
- `DUFormEcon`: This option add the optimization variable $\Delta u(i) = u(i) - u(i-1)$ to $u(i)$. In particular, this option can be used when the general objective function is selected (e.g., in economic MPC) in which Δu takes the place of u_s; the function inputs in this case should be hence read as follows: "$F_{obj}(x,u,y,x_s,\Delta u,y_s)$"
- `TermCons`: This option activates the terminal constraint on the dynamic optimization module. The constraint will be: $x(N) = x_s$.
- `estimating`: This option activates the simulation in open loop, i.e. $u = 0$. When `estimating = 'True'` there is no need to define any of the following: setpoints (§3.4.1), Steady-state (Target) optimization module (§3.4.3), Dynamic optimization module (§3.4.4).
- `ssjacid`: This option activates the steady-state point search for the model and then linearize it evaluating the linearization matrices at the steady-state point found. This tag has to be used only when you have a non-linear model dynamics, i.e., functions $\dot{x} = User\_fxm\_Cont(x,u,d,t)$ or $x = User\_fxm\_Dis(x,u,d,t)$ or/and $y = User\_fym(x,d,t)$.
  When `ssjacid = 'True'`, despite the original model functions definition, the system considered in the MPC problem are the one represented in §3.2.2 with linearization parameters.

The default value of all these options is `False`.

3.6 Plotting

The following listed variables are all plotted against the vector `tsim` that represents the simulation time. These variables are all also available in matrix form; e.g., if there are three states: $X = [x_1, x_2, x_3]$ where x_i is the column vector that represents the behavior of the i-th state over `tsim`. Here a description of all printed information:

- `X_HAT`: it represents the matrix of all the estimated states;
- `Y_HAT`: it represents the matrix of all the model outputs;

- `U`: it represents the matrix of all the inputs;
- `Xp`: it represents the matrix of all the process states;
- `Yp`: it represents the matrix of all the process output;
- `XS`: it represents the matrix of all the target (steady values) states;
- `US`: it represents the matrix of all the target inputs;
- `YS`: it represents the matrix of all the target outputs;
- `D_HAT`: it represents the matrix of all the estimated model disturbances;

When `estimating = 'True'` the only variables displayed in plots will be `X_HAT`, `Xp` and `Y_HAT`, `Yp`.

Furthermore, all the plotted figures are automatically saved in the `MPC_code.py` folder. To set a specific pattern or folder where to save the generated figures, just define a string variable named `pathfigure` within your example file. This custom path has to be defined respective to the `MPC_code.py` folder, otherwise it can also be an absolute path. For example, if the chosen folder is named “Images" and it is in the same location of `MPC_code.py`, `pathfigure = './Images/'` or `pathfigure = 'C:/.../Images/'`.