DO-MPC

A toolbox for the easy, efficient and modular implementation of NMPC

User's Guide

Authors:
Sergio Lucia
Alexandru Tatulea-Codrean
Sebastian Engell

Last updated: May 25, 2015

Contents

Chapter 1

Introduction

DO-MPC is a toolbox for the efficient, easy and modular implementation of nonlinear model predictive control, and especially of multi-stage nonlinear model predictive control (multi-stage NMPC). Multi-stage NMPC is a recenlty proposed approach that considers explicitly the presence of uncertainty in the model of a system. The main idea of DO-MPC is to divide any NMPC implementation into four main *modules*: model, optimizer, observer and simulator. This modularization makes the design, deployment and maintenance of an NMPC implementation easy and transparent. We provide in this simplified version of DO-MPC a set of python templates for each module that the user can modify to adapt to each specific problem.

1.1 Capabilities and Limitations

There are several tools for nonlinear model predictive control which are already available, as e.g. the ACADO Toolkit [1]. The main features that differentiate DO-MPC from other tools are outlined in the following. DO-MPC provides a modularized implementation of NMPC separated in four modules: model, optimizer, observer and simulator. In this manner, if a model is updated only the model module has to be modified and this information is automatically used by the other parts of the implementation. Additionally, DO-MPC provides an automatic design of a scenario tree for multi-stage NMPC based on the possible values of the uncertain parameters or disturbances given by the user. We also provide an efficient implementation of the approach using orthogonal collocation on finite elements (or multiple shooting) which is solved via IPOPT[2] using exact first and second order derivatives which are computed using CasADi[3]. The current version of DO-MPC has been successfully tested both in Windows and in Linux.

DO-MPC could be especially useful for you if:

[1] Houska, B.; Ferreau, H. and Diehl, M. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization Optimal Control Applications and Methods, 2011, 32, 298-312

[2] Wächter, A. and Biegler, L. On the implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming Mathematical Programming, 2006, 106, 25-5

[3] Andersson, J.; Åkesson, J. and Diehl, M. CasADi – A symbolic package for automatic differentiation and optimal control Recent Advances in Algorithmic Differentiation, Springer, 2012, 297-307

- You want an easy an efficient implementation of robust NMPC
- You want an efficient but very flexible implementation of NMPC
- You need a modular implementation of NMPC for future developments and transparent comparisons of different approaches

DO-MPC does not provide a code-generation tool that can be directly deployed on an embedded platform as self-contained C-code. In order to use DO-MPC it is necessary to install third-party software such as IPOPT, SUNDIALS[4] and CasADi.

1.2 Citing DO-MPC

If you use DO-MPC for published work please cite it as:

S. Lucia, A. Tatulea-Codrean, C. Schoppmeyer, and S. Engell. Fast, Efficient and Sustainable Development of Robust NMPC Solutions. Submitted to *IEEE Transactions on Control Systems Technology*, 2015.

1.3 License

[4]Hindmarsh, A. C.; Brown, P. N.; Grant, K. E.; Lee, S. L.; Serban, R.; Shumaker, D. E. and Woodward, C. S. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers ACM Transactions on Mathematical Software,, 2005, 31, 363-396

Chapter 2

Installation Instructions

DO-MPC is a set of Python modules and scripts that is based on third party software. DO-MPC is strongly based on the use of CasADi (`http://www.casadi.org`). The easiest way to install all the third party software that is necessary for DO-MPC to work is to install one of the precompiled versions of CasADi, which are available for Windows and for Linux, from (`https://github.com/casadi/casadi/wiki/InstallationInstructions`). Depending on your operating system you need some additional software (mainly a Python distribution). For detailed description follow carefully the installation instructions for CasADi. The precompiled versions of CasADi include a version of IPOPT.

Once CasADi with IPOPT is running, the examples for DO-MPC should also run. You can then run the code. In Linux and Mac you can use:

`git clone https`//github.com/do-mpc/do-mpc.git do-mpc:
and then you can test the code running the main script using:

`cd do-mpc/code/examples/CSTR; python do-mpc.py`

In Windows you can you can download git for windows from (`https://git-scm.com/downloads`) and follow the same procedure as for Linux or download the code directly from `https://github.com/do-mpc/do-mpc/archive/master.zip`

It is highly recommended that you use a better linear solver than the one that comes with the precompiled version of IPOPT (mumps). You can do this by downloading the precompiled libraries from `http://www.hsl.rl.ac.uk/ipopt/`, which are available for Windows and for Linux. Once downloaded, add the path of the directory where the library has been store to your environment variable `PATH` on Windows or to `LD_LIBRARY_PATH` on Linux. After restarting your Python session the linear solver MA27 should be available.

The current version of DO-MPC has been successfully tested on Windows 7 and 8, and on Linux 12.04 and 14.04 using Python 2.7x and versions of CasADi 2.2 and 2.3. **Note that for earlier versions of CasADi the code will not work due to changes in the syntax.**

Chapter 3

Your Efficient NMPC in 60 Minutes

The spirit of DO-MPC is that any user with just a moderate knowledge of model predictive control and of a scripting language like Python (or Matlab), should be able to get results of an efficient implementation of robust (or standard) nonlinear model predictive control in less than 60 minutes, just by looking at the examples and modifying the code accordingly in a very intuitive way. To achieve this, we provide templates for the problem specific information (model equations, constraints, cost function, definition of the simulator, observer) so that the user does not have to get into the tedious task of coding the discretization and formulation of the NLP, getting derivatives (thanks to CasaADi) or even plotting the results.

Although everything is programmed using Python, the code runs in C/C++ and therefore an excellent computational performance is achieved, but with a fast and easy development phase. To get your efficient implementation of a robust NMPC in 60 minutes follow the steps enumerated below:

1. Write the model equations in the file `template_model.py`
2. Describe the Optimal Control Problem (OCP) in the file `template_model.py`. The OCP description contains:
   (a) Initial Condition
   (b) Cost function
   (c) Constraints
3. Describe the controller parameters in the file `template_optimizer.py`
4. Describe the simulator parameters, the plotting options and the real valuse of the uncertain parameters in the file `template_simulator.py`
5. Describe the observer in the file `template_observer.py`
6. Run DO-MPC using `python do-mpc.py`

Chapter 4

A Simple Example

We illustrate the implementation of a robust NMPC using DO-MPC using a nonlinear CSTR benchmark problem adapted from [1]. The dynamics of the CSTR are described by the following set of differential equations:

4.1 Description of the Problem

$$\dot{c}_\mathrm{A} = F(c_\mathrm{A0} - c_\mathrm{A}) - k_1 c_\mathrm{A} - k_3 c_\mathrm{A}^2, \tag{4.1a}$$

$$\dot{c}_\mathrm{B} = -F c_\mathrm{B} + k_1 c_\mathrm{A} - k_2 c_\mathrm{B}, \tag{4.1b}$$

$$\dot{T}_\mathrm{R} = F(T_\mathrm{in} - T_\mathrm{R}) + \frac{k_\mathrm{W} A}{\rho c_\mathrm{p} V_\mathrm{R}}(T_\mathrm{K} - T_\mathrm{R}) - \frac{k_1 c_\mathrm{A} \Delta H_\mathrm{AB} + k_2 c_\mathrm{B} \Delta H_\mathrm{BC} + k_3 c_\mathrm{A}^2 \Delta H_\mathrm{AD}}{\rho c_\mathrm{p}}, \tag{4.1c}$$

$$\dot{T}_\mathrm{K} = \frac{1}{m_\mathrm{K} c_\mathrm{pK}}(\dot{Q}_\mathrm{K} + k_\mathrm{W} A(T_\mathrm{R} - T_\mathrm{K})), \tag{4.1d}$$

where the reaction rates k_i follow the Arrhenius law:

$$k_i = k_{0,i} e^{\frac{-E_{A,i}}{R(T_\mathrm{R}+273.15)}}. \tag{4.2}$$

The ODEs are derived from component balances for the concentration of component A (c_A) and for the the concentration of component B (c_B). The energy balances for the temperature of the reactor T_R and for the coolant temperature (T_K) form the last two differential equations. The control inputs are the inflow ($F = \dot{V}_\mathrm{in}/V_\mathrm{R}$) normalized by the volume of the reactor and the heat removed by the coolant ($\dot{Q}_K$).

The initial conditions of the states together with the constraints on the states are described in Table 4.2. The constraints for the control inputs are shown in Table 4.3. It is considered that the activation energy $E_{A,3}$ and the reaction rate $k_{0,1}$ are uncertain

[1] Klatt, K. and Engell, S. Gain-scheduling trajectory control of a continuous stirred tank reactor Computers & Chemical Engineering, 1998, 22, 491-502

Table 4.1: Parameter values of the CSTR.

Parameter	**Value**	**Unit**
$k_{0,1}$	1.287 10^{12}	h^{-1}
$k_{0,2}$	1.287 10^{12}	h^{-1}
$k_{0,3}$	9.043 10^{9}	$\mathrm{l\,mol}^{-1}\mathrm{h}^{-1}$
$E_{A,1}/R$	9758.3	K
$E_{A,2}/R$	9758.3	K
$E_{A,3}/R$	8560.0	K
ΔH_{AB}	4.2	KJ mol^{-1}
ΔH_{BC}	-11.0	KJ mol^{-1}
ΔH_{AD}	-41.85	KJ mol^{-1}
ρ	0.9342	$\mathrm{kg\,l}^{-1}$
c_{p}	3.01	$\mathrm{kJ\,kg}^{-1}\,\mathrm{K}^{-1}$
c_{pK}	2.0	$\mathrm{kJ\,kg}^{-1}\,\mathrm{K}^{-1}$
A	0.215	m^2
V_{R}	10.01	l
m_{k}	5.0	kg
T_{in}	130.0	°C
k_{W}	4032	KJ h^{-1} m^{-2} K^{-1}

Table 4.2: Initial conditions and state constraints.

State	**Init. cond.**	**Min.**	**Max.**	**Unit**
c_{A}	0.8	0.1	2.0	$\mathrm{mol\,l}^{-1}$
c_{B}	0.5	0.1	2.0	$\mathrm{mol\,l}^{-1}$
T_{R}	134.14	50.0	180	°C
T_{J}	134.0	50.0	180.0	°C

Table 4.3: Bounds on the manipulated variables.

Control	**Min.**	**Max.**	**Unit**
F	5	100	h^{-1}
$\dot{Q}_{\mathrm{K}}$	0	-8500	$\mathrm{kJ\,h}^{-1}$

and it is assumed that they vary by ±10% with respect to its nominal value. The uncertainties will be addressed using multi-stage NMPC[2]. The stage cost minimized at each time stage for each scenario is chosen as:

$$L = 10^4(c_{\mathrm{B}} - 0.9) + 10^4(c_{\mathrm{A}} - 1.2) \tag{4.3}$$

The prediction horizon is $N_p = 20$ steps and the sampling time of the controller is $t_{\mathrm{step}} = 0.005$ h.

4.2 Implementation using DO-MPC

Within DO-MPC, any multi-stage NMPC implementation is based on the definition of four different modules: model, optimizer, observer and simulator. In the following subsections the full code necessary to solve standard and multi-stage NMPC for the nonlinear CSTR is presented.

4.2.1 Model Template

The model template (`template_model.py`) contains the definition of the model equations and the definition of the optimal control problem (OCP). The states, inputs, and uncertainties are defined as CasADi symbols as:

```
# Define the uncertainties as CasADi symbols

alpha   = SX.sym("alpha")
beta    = SX.sym("beta")
# Define the differential states as CasADi symbols

C_a    = SX.sym("C_a") # Concentration A
C_b    = SX.sym("C_b") # Concentration B
T_R    = SX.sym("T_R") # Reactor Temprature
T_K    = SX.sym("T_K") # Jacket Temprature

# Define the algebraic states as CasADi symbols

# Define the control inputs as CasADi symbols

F      = SX.sym("F") # Vdot/V_R [h^-1]
Q_dot  = SX.sym("Q_dot") #Q_dot second control input
```

The parameters are defined:

```

K0_ab = 1.287e12 # K0 [h^-1]
K0_bc = 1.287e12 # K0 [h^-1]
K0_ad = 9.043e9 # K0 [l/mol.h]
R_gas = 8.3144621e-3 # Universal Gas constant [kj.K^1.mol^1]
E_A_ab = 9758.3*1.00 #* R_gas# [kj/mol]
E_A_bc = 9758.3*1.00 #* R_gas# [kj/mol]
E_A_ad = 8560.0*1.0 #* R_gas# [kj/mol]
H_R_ab = 4.2 # [kj/mol A]
H_R_bc = -11.0 # [kj/mol B] Exothermic
H_R_ad = -41.85 # [kj/mol A] Exothermic
Rou = 0.9342 # Density [kg/l]
Cp = 3.01 # Specific Heat capacity [kj/Kg.K]
```

[2]Lucia, S.; Finkler, T. and Engell, S. Multi-stage Nonlinear Model Predictive Control Applied to a Semi-batch Polymerization Reactor under Uncertainty Journal of Process Control, 2013, 23, 1306-1319

```
Cp_k = 2.0 # Coolant heat capacity [kj/kg.k]
A_R = 0.215 # Area of reactor wall [m^2]
V_R = 10.01 #0.01 # Volume of reactor [l]
m_k = 5.0 # Coolant mass[kg]
T_in = 130.0 # Temp of inflow [Celsius]
K_w = 4032.0 # [kj/h.m^2.K]
C_A0 = (5.7+4.5)/2.0*1.0 # Concentration of A in input Upper bound 5.7 lower bound 4.5 [mol/l]
```

The algebraic and differential equations are defined as:

```
K_1 = beta * K0_ab * exp((-E_A_ab)/((T_R+273.15)))
K_2 =  K0_bc * exp((-E_A_bc)/((T_R+273.15)))
K_3 = K0_ad * exp((-alpha*E_A_ad)/((T_R+273.15)))

# Define the differential equations

dC_a = F*(C_A0 - C_a) -K_1*C_a - K_3*(C_a**2)
dC_b = -F*C_b + K_1*C_a -K_2*C_b
dT_R = ((K_1*C_a*H_R_ab + K_2*C_b*H_R_bc + K_3*(C_a**2)*H_R_ad)/(-Rou*Cp)) + F*(T_in-T_R) +(((K_w*A_R)*(T_K-T_R))/(Rou*Cp
dT_K = (Q_dot + K_w*A_R*(T_R-T_K))/(m_k*Cp_k)
```

To define the OCP first the initial condition is defined as:

```
# Initial condition for the states
C_a_0 = 0.8 # [mol/l]
C_b_0 = 0.5 # [mol/l]
T_R_0 = 134.14 #[C]
T_K_0 = 130.0 #[C]
x0 = NP.array([C_a_0, C_b_0, T_R_0, T_K_0])
```

Then the constraints on the states and on the control inputs are written

```
Bounds on the states. Use "inf" for unconstrained states
C_a_lb = 0.1;                          C_a_ub = 2.0
C_b_lb = 0.1;                          C_b_ub = 2.0
T_R_lb = 50.0;                          T_R_ub = 180
T_K_lb = 50.0;                          T_K_ub = 180
x_lb = NP.array([C_a_lb, C_b_lb, T_R_lb, T_K_lb])
x_ub = NP.array([C_a_ub, C_b_ub, T_R_ub, T_K_ub])

# Bounds on the control inputs. Use "inf" for unconstrained inputs
F_lb = 5.0;                 F_ub = +100.0;
Q_dot_lb = -8500.0;         Q_dot_ub = 0.0;
u_lb = NP.array([F_lb, Q_dot_lb])
u_ub = NP.array([F_ub, Q_dot_ub])
u0 = NP.array([30.0,-6000.0])

# Scaling factors for the states and control inputs. Important if the system is ill-conditioned
x_scaling = NP.array([1.0, 1.0, 1.0, 1.0])
u_scaling = NP.array([1.0, 1.0])
```

Other nonlinear constraints can be defined here, including terminal constraints (they are not used in this example):

```
# Other possibly nonlinear constraints in the form cons(x,u,p) <= cons_ub
# Define the expresion of the constraint (leave it empty if not necessary)
cons = vertcat([])
# Define the lower and upper bounds of the constraint (leave it empty if not necessary)
cons_ub = NP.array([])
# Define the terminal constraint (leave it empty if not necessary)
cons_terminal = vertcat([])
# Define the lower and upper bounds of the constraint (leave it empty if not necessary)
cons_terminal_lb = NP.array([])
cons_terminal_ub = NP.array([])
```

The nonlinear constraints can be formulated as soft constraints automatically if it is desired (set `soft_constraint` to 1). Finally the cost function is defined by writting the Lagrange and Mayer terms as a function of the previously defined symbolic expressions. Additionally a penalty term for the control movements can be also defined:

```
# Define the cost function
# Mayer term
mterm =  1e4*(C_b - 0.9)**2 + 1e4*(C_a - 1.2)**2
# Lagrange term
lterm =  0
# Penalty term for the control movements
rterm = NP.array([0, 0])
```

4.2.2 Optimizer Template

In the optimizer template (`template_optimizer.py`) the tuning parameters of the controller are chosen: the prediction horizon, the discretization method, the number of finite elements and degree of the orthogonal polynomials, the simulation time and the sampling time of the controller.

```
# Prediction horizon
nk = 20

# Sampling time
t_step = 0.005

# Robust horizon, set to 0 for standard NMPC
n_robust = 1

# State discretization scheme: 'multiple-shooting' or 'collocation'
state_discretization = 'collocation'

# Collocation-specific options
# Degree of interpolating polynomials: 1 to 5
deg = 2
# Collocation points: 'legendre' or 'radau'
coll = 'radau'

# Number of finite elements per control interval
ni = 2

# Simulation Time
end_time = 0.2

# NLP Solver and linear solver
nlp_solver = 'ipopt'

# It is highly recommended that you use a more efficient linear solver
# such as the hsl linear solver MA27, which can be downloaded as a precompiled
# library and can be used by IPOPT on run time

linear_solver = 'mumps'
```

Here also the possible values of the uncertainties are given. This information is used to build a scenario tree that branches up the stage `n_robust`. If `n_robust` is chosen to be 0 then standard NMPC is used, and the value chosen in the model used to predict the behavior of the system is the first element of the possible values given in the vector `uncertainty_values`

```
# Define the different possible values of the uncertain parameters in the scenario tree
alpha_values = NP.array([1.0, 1.1, 0.9])
beta_values  = NP.array([1.0, 1.1, 0.9])
uncertainty_values = NP.array([alpha_values,beta_values])
```

4.2.3 Observer Template

We consider in this case full state feedback defined in the script `template_observer`

```
# Full state feedback
 x = y
```

4.2.4 Simulator Template

Define the model that will be used for the simulation of the real system in `template_simulator.py`. Note that here a more complex model can be used than the one used for the design of the controller. Also the simulation parameters can be chosen.

```
# Choose the integrator (CVODES, IDAS)
integrator = Integrator('cvodes',f_sim)
# Choose the integrator parameters
integrator.setOption("abstol",1e-10) # tolerance
integrator.setOption("reltol",1e-10) # tolerance
integrator.setOption("steps_per_checkpoint",100)
```

Choose the states and control inputs that you want to plot at the end of the simulation. In addition, an especial feature of DO-MPC is that you can activate an *animation* that shows the prediction of the controller (including all scenarios) at each sampling time. Especially in the case of multi-stage NMPC, looking at such predictions is very important to understand the behavior of the controller

```
Choose the indices of the states to plot
plot_states = [0, 1, 2]
Choose the indices of the controls to plot
plot_control = [0, 1]
Plot animation (False or True)
plot_anim = False
```

Choose the real value of the uncertainty. They can be constant, random, or time-varying.

```
p_real =  NP.array([1.0,1.0])
```

After defining all the modules the NMPC problem can be run by executing the script `do-mpc.py` in the folder of the corresponding example.