Laboratory Session 2

Basics on optimization-based control and implementations

Ionela Prodan, Minh Tuan Dinh

2024

Contents

Throughout this session we will introduce some basic definitions in optimization-based control and we will concentrate on MPC (Model Predictive Control) implementations using CasADi for solving some basic control theory problems as stabilization, set-point tracking feedback and trajectory tracking taking into account state and input constraints.

1 Optimization-based control

Optimization-based control in general terms refers to the control design using an optimization criterion and the respective resolution techniques in order to obtain the parameters of the control law, the optimality being generally equivalent to a certain desired property as for example, stability, reactivity or robustness.

A widely used optimization-based control technique in this class is Model Predictive Control (MPC)[1] also called, *receding horizon control* [2].

1.1 Model Predictive Control - Background

The idea behind MPC is to exploit in a receding manner the simplicity of the open-loop optimization-based control [5, 1, 6]. The control action $u(k)$ for a given state $x(k)$ is obtained from the control sequence $\mathbf{u} \triangleq \{u(k), u(k+1), \dots, u(k+N_p-1)\}$ as the result of the optimization problem [4]:

$$\arg\min_{\mathbf{u}} \quad V_f(x(k+N_p)) + \sum_{s=0}^{N_p-1} V_n(x(k+s), u(k+s)), \tag{1}$$

$$\text{subject to:} \quad \begin{cases} x(k+s+1) = f(x(k+s), u(k+s)), \quad s = 0, \dots, N_p - 1, \\ h(x(k+s), u(k+s)) \leq 0, \quad s = 0, \dots, N_p - 1, \\ h_f(x(k+N_p)) \leq 0, \end{cases} \tag{2}$$

over a finite horizon N_p. The cost function is comprised of two basic ingredients; namely a terminal cost function $V_f(\cdot) : \mathbb{R}^n \to \mathbb{R}$ and a cost per stage function $V_n(\cdot) : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$. Typically, in MPC, the objective (or cost) function (1) penalizes deviations of the states and inputs from their reference values, while the constraints are treated *explicitly* [3]. By solving (1), each optimization generates an open-loop optimal control trajectory taking into account the system dynamics described by $f(\cdot, \cdot)$, generally with the additional property that f(0,0)=0, the constraints on the states and control inputs $h(\cdot)$ and terminal constraint $h_f(\cdot)$. Then, applying only the first part of the trajectory to the system, based on measurement and recomputing, results in closed-loop control (see, Fig. 1 which illustrates very well the receding horizon strategy).

The restrictions (2) of the optimization problem (1) can be written in a more explicit form, stated in terms of hard constraints on the internal state variables and input control action (whenever these are separable):

$$\begin{cases} x(k+s+1) = f(x(k+s), u(k+s)), \quad s = 0, \dots, N_p - 1, \\ x(k+s) \in \mathcal{X}, \quad s = 0, \dots, N_p - 1, \\ u(k+s) \in \mathcal{U}, \quad s = 0, \dots, N_p - 1, \end{cases} \tag{3}$$

[1] The terminology "Model Predictive" comes from the use of the model to predict the system behavior over the planning horizon at each update.

[2] The terminology "receding horizon" comes from the fact that the planning horizon, which is typically fixed, moves ahead in time with each update.

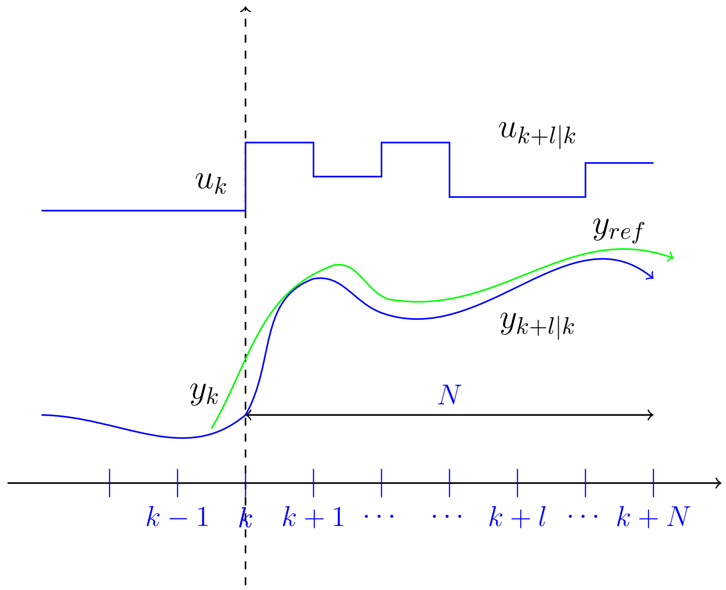

Figure 1: Receding horizon control philosophy.

where, usually, $\mathcal{X}$ is a convex, closed subset of $\mathbb{R}^n$ and $\mathcal{U}$ is a convex, compact subset of $\mathbb{R}^m$, each set containing the equilibrium point in their strict interior (generally $f(0,0) = 0$ and $0 \in \mathcal{X}$, $0 \in \mathcal{U}$). A terminal constraint could also be imposed for stability reasons mayne2009model:

$$x(k + N_p) \in \mathcal{X}_f \subset \mathcal{X}. \tag{4}$$

MPC represents one of the few methods in control that can handle generic state and control constraints. More precisely, it has the ability to include generic models (i.e., nonlinear and linear) and constraints in the optimization-based control problem (1)–(2). In addition, to this main advantage, it is worth mentioning its' capacity to redefine the cost function and the constraints to account for the changes in the system and/or the environment. In our opinion, the major inconvenient of receding horizon strategy is represented by the computational demand, i.e., it is prohibitive the requirement that an optimization algorithm must run and terminate at every update of the controller block (generally synchronous with the sampling clock).

1.2 Matlab implementations

All computations in MPC are based on a discrete time representation of the system dynamics. On state space form the model is:

$$x(k+1) = Ax(k) + Bu(k), \tag{5}$$
$$y(k) = Cx(k), \tag{6}$$

where 1 time step corresponds to the sample time T_e. There is never a direct term D from input to output in MPC models. This follows from the fact that, at a given sample k, the output is already given and can hence not be affected by the present input $u(k)$ which is to be calculated by the optimizer.

The general MPC formulation is provided in (1) where the cost function can be quadratic

$$V_n(x(k+s), u(k+s)) = x^T(k+s)Qx(k+s) + u^T(k+s)Ru(k+s), \tag{7}$$

linear

$$V_n(x(k+s), u(k+s)) = f^T x(k+s) + g^T u(k+s), \tag{8}$$

or nonlinear[3].

For implementing MPC problems in Matlab there are several options:

- There are a set of Matlab functions, referred to as MPCtools. MPCtools implements an MPC controller for use with Matlab/Simulink. MPCtools requires no installation or compilation, but it is convenient to add the directory containing the MPCtools functions to the Matlab path. The tools require Control System Toolbox and, if the Simulink extension is to be used, also Simulink.
- The quadratic programming (QP) solver *"quadprog"* may be used to solve the MPC optimization problem or the linear programming (LP) solver *"linprog"*. This feature requires Matlab Optimization Toolbox.
- MPC Toolbox 3.0 can also be used.
- Yalmip toolbox [2] permits to write optimization problems (cost and constraints) in a compact manner by hiding the complexities of constraint concatenation (thus saving time and errors).
- as indicated in the first Laboratory session, we will mostly use CasADi modeling language since it allows Octave, Matlab and Python implementation.

1.3 Stabilize a double integrator dynamics using MPC: Matlab with CasADi implementation

Let us consider the dynamical system (5) with:

```
% The double integrator dynamics
%Initialization data
clear
clc
close all

```

[3] During this course we will discuss examples with quadratic and linear cost.

```
% State-space model: x(k+1) = Ax(k) + Bu(k); y(k) = Cx(k) + Du(k)
mu=0.5;
A = [1  0 mu  0;
      0  1  0 mu;
      0  0  1  0;
      0  0  0  1];
B = [0  0;
      0  0;
      mu  0;
      0  mu];
C = [1  0  0  0;
      0  1  0  0];
D = [0  0;
      0  0];

% System dimension
[dx, du] = size(B);
dy = size(C,1);

% Initial condition
x0 = [0.8;0.2;0;0];
u0 = zeros(du, 1);

% Constraints
umin = -1 * 0.25;
umax = +1 * 0.25;
delta_u_min = -1 * 0.1;
delta_u_max = +1 * 0.1;
ymin = -10;
ymax = +10;
% xmin = [ymin; -0.2];
% xmax = [ymax; 0.5];

%Define control parameters
% weighting matrices
Q = 1*eye(dx);     % cost for the state x
R = 1;             % cost for the input u
Qy = eye(dy);      % cost for the output y
P = 10*Q;          % cost for the terminal cost
% number of predictions and simulations
Npred = 5;
Nsim = 100;


% Optimization problem using CasADi
%addpath(...CasADi)
import casadi.*

solver = casadi.Opti(); %using Opti class
% Define variables:
x = solver.variable(dx, Npred+1);
u = solver.variable(du, Npred);
xinit = solver.parameter(dx,1);
uinit = solver.parameter(du,1);
```

```
% Initialize constraints
solver.subject_to(x(:,1) == xinit)
for k = 1 : Npred
    solver.subject_to(x(:,k+1) == A*x(:,k) + B*u(:,k)) % dynamics
    solver.subject_to(umin <= u(:,k) <= umax) % input magnitude constraints
    solver.subject_to(ymin <= C*x(:,k)+D*u(:,k) <= ymax) % state magnitude
    constraints
%     solver.subject_to(xmin(2) <= [0 1]*x(:,k) <= xmax(2)) % additional
    state constraints
    if k == 1
        solver.subject_to(delta_u_min <= u(:,k) - uinit <= delta_u_max);
    else
        solver.subject_to(delta_u_min <= u(:,k) - u(:,k-1) <= delta_u_max);
    end
end

% Initialize objective
objective = 0;
for k =1 : Npred
    objective = objective + x(:,k)'*Q*x(:,k) + u(:,k)'*R*u(:,k); %
    quadratic cost function
end
objective = objective+x(:,Npred+1)'*P*x(:,Npred+1);
solver.minimize(objective)

% Define the solver
options.ipopt.print_level = 0;
options.ipopt.sb = 'yes';
options.print_time = 0;
solver.solver('ipopt', options)

% simulation loop
usim = zeros(du, Nsim);
ysim = zeros(dy, Nsim);
xsim = zeros(dx, Nsim+1);
xsim(:, 1) = x0;
usim_init = u0;

timer = tic;
for i = 1:Nsim
    solver.set_value(xinit, xsim(:, i))
    solver.set_value(uinit, usim_init)
    sol = solver.solve();
    usol = sol.value(u);
    usim_init = usol(:,1);
    usim(:, i) = usol(:,1);
    xsim(:, i+1) = A*xsim(:, i) + B*usim(:, i); % update the dynamics
    ysim(:, i) = C*xsim(:, i) + D*usim(:, i); % update the dynamics
end
time_CasAdi = toc(timer)
```

```
figure
stem(ysim(1,:));
hold on
stem(ysim(2,:));
title('output y');
legend 'y1' 'y2'

figure
scatter(xsim(1, :), xsim(2, :));
title('state space');
xlabel 'x1'
ylabel 'x2'

figure
stem(usim(1,:));
hold on
stem(usim(2,:));
title('input u');
legend 'u1' 'u2'

% how to compute the tracking error
error = sqrt(xsim(1, :).^2 + xsim(2, :).^2 + xsim(3, :).^2 + xsim(4, :).^2)
    ;
Avg_error = mean(error)
Avg_u1 = mean(abs(usim(1,:)))
Avg_u2 = mean(abs(usim(2,:)))
```

2 Exercises

The next exercises are intended to end up with the construction of “your own” MPC examples for state feedback, output feedback, set-point tracking, trajectory tracking for discrete time invariant systems and/or continuous systems affected by bounded disturbances.

To be able to answer the next questions it will be very useful to provide a table which shows:

- initial conditions
- N_p, Q, R, P (the MPC tuning)
- Average output Error, Average Input Error
- Computing time

Exercise 2.1. To reduce the computational time for the optimization one may try to reduce the prediction horizon $Npred$ (the horizon is also often considered the most important tuning parameter in MPC). Try to reduce the prediction horizon and repeat the simulation. What happens? Try to explain the observed behavior.

Exercise 2.2. Change the prediction horizon and increase/decrease the terminal weight. Run a simulation. Comparing with the result above, what is the effect of increasing/decreasing the terminal weight?

From a stability point of view, what would you recommend;

- a short prediction horizon with large penalty on the terminal state, or
- a long horizon with no particular penalty on the terminal state?

Exercise 2.3. Add constraints on the state variation. Is the controller able to satisfy all constraints for all time?

Exercise 2.4. Consider the reference tracking problem by providing an arbitrary output reference y_{ref}. The cost function is similar with the one provided in the exercise above. We may choose to penalize in the cost function the input variations. Why?

Exercise 2.5. Consider also some of the examples provided in Lab session - 1, for example a continuous time invariant system, discretize it with a sampling time T_e, implement the MPC problem and than change the sampling time. What happens? Try to explain the observed behavior.

3 Annex

3.1 Stabilize a double integrator dynamics using MPC: Python with CasADi implementation

```

import numpy as np
import casadi as cas
import time
import matplotlib.pyplot as plt

The double integrator dynamics System data


#State-space model1: x+ = Ax + Bu; y = Cx + Du
h = 0.5
A = np.block([[np.eye(2), h*np.eye(2)],
               [np.zeros((2,2)), np.eye(2)]])
B = np.vstack([np.zeros((2,2)),h*np.eye(2)])
C = np.hstack([np.eye(2), np.zeros((2,2))])
D = np.zeros((2,2))
In [37]:
# Model dimension
dx, du = np.shape(B)
dy = np.shape(C)[0]

#Initial conditions
```

```
x0 = np.array([0.8, 0.2, 0, 0])
u0 = np.zeros((du,1))

#Constraints
umin = -1 * 0.25;
umax = +1 * 0.25;
delta_u_min = -1 * 0.1;
delta_u_max = +1 * 0.1;
ymin = -10;
ymax = +10;
#xmin = np.array([ymin, -0.2]);
#xmax = np.array([ymax, 0.5]);
Define control parameters


# weighting matrices
Q = np.eye(dx);     # cost for the state x
R = 1;              # cost for the input u
Qy = np.eye(dy);    # cost for the output y
P = 10*Q;           # cost for the terminal cost
# number of predictions and simulations
Npred = 5;
Nsim = 100;
Optimization problem using CasADi


solver = cas.Opti() #create an Opti object
# Define variables:
x = solver.variable(dx, Npred+1);
u = solver.variable(du, Npred);
xinit = solver.parameter(dx,1);
uinit = solver.parameter(du,1);

# Initialize constraints
solver.subject_to(x[:,0] == xinit)
for k in range(0, Npred):
    solver.subject_to(x[:,k+1] == cas.mtimes(A,x[:,k]) + cas.mtimes(B,u[:,k])) # dynamics
    solver.subject_to(umin <= u[:,k]) # input magnitude constraints
    solver.subject_to(u[:,k] <= umax)
    solver.subject_to(ymin <= cas.mtimes(C,x[:,k]) + cas.mtimes(D,u[:,k])) # state magnitude constraints
    solver.subject_to(cas.mtimes(C,x[:,k]) + cas.mtimes(D,u[:,k]) <= ymax)
    # solver.subject_to(xmin[1] <= [0 1]*x[:,k] <= xmax[1]) # additional state constraints
    if k == 0:
        solver.subject_to(delta_u_min <= u[:,k] - uinit)
        solver.subject_to(u[:,k] - uinit <= delta_u_max)
    else:
        solver.subject_to(delta_u_min <= u[:,k] - u[:,k-1])
        solver.subject_to(u[:,k] - u[:,k-1] <= delta_u_max)

# Initialize objective
objective = 0;
```

```
for k in range(0, Npred):
    objective = objective + cas.mtimes(cas.mtimes(cas.transpose(x[:,k]),Q), x[:,k]) + \
                            cas.mtimes(cas.mtimes(cas.transpose(u[:,k]),R), u[:,k]) # quadratic cost function
objective = objective + cas.mtimes(cas.mtimes(cas.transpose(x[:,Npred]),P), x[:,Npred])
solver.minimize(objective)

# Define the solver
options = {'ipopt': {'print_level': 0, 'sb': 'yes'}, 'print_time': 0}
solver.solver('ipopt', options)

# simulation loop
usim = np.zeros((du, Nsim))
ysim = np.zeros((dy, Nsim))
xsim = np.zeros((dx, Nsim+1))
xsim[:, 0] = x0
usim_init = u0
In [ ]:
timer_start = time.time();
for i in range(Nsim):
    solver.set_value(xinit, xsim[:,i])
    solver.set_value(uinit, usim_init)
    sol = solver.solve();
    usol = sol.value(u);
    usim_init = usol[:,0];
    usim[:,i] = usol[:,0];
    xsim[:, i+1] = A@xsim[:,i] + B@usim[:,i]; # update the dynamics
    ysim[:,i] = C@xsim[:,i] + D@usim[:,i]; # update the dynamics
time_end = time.time()
time_elapsed = time_end - timer_start
print(f'Total time: {time_elapsed} (s)')
Plot the results


plt.figure()
plt.stem(ysim[0,:], label='y1')
plt.stem(ysim[1,:], label='y2', linefmt='r-', markerfmt='ro')
plt.title('Output y')

# Show the plot
plt.grid()
plt.legend()
plt.show()

I
plt.figure()
plt.scatter(xsim[0,:], xsim[1,:], edgecolors='red', facecolors='none')
plt.title('State space')

# Show the plot
plt.grid()
plt.xlabel('x1')
```

```
plt.ylabel('x2')
plt.show()


plt.figure()
plt.stem(usim[0,:], label='u1')
plt.stem(usim[1,:], label='u2', linefmt='r-', markerfmt='ro')
plt.title('Input u')

# Show the plot
plt.legend()
plt.grid()
plt.show()

# Anayze the results
error = np.sqrt(np.sum(xsim**2,axis=0))
Avg_error = np.mean(error)
print(f'Average error: {Avg_error}')

Avg_u = np.mean(np.abs(usim),axis=1)
print(f'Average input: {Avg_u}')
```

3.2 Stabilize a double integrator dynamics using MPC: Matlab with Yalmip implementation

Let us consider the dynamical system (5) with:

```
% system dynamics
% A = [1 1; 0 1];
% B = [1; 0.3];
% C = [1 0];
% D = 0;

% Here you should use the double integrator dynamics
h=0.5 ;
A = [eye(2) eye(2)*h; zeros(2) eye(2)];
B = h*[zeros(2); eye(2)];
C = [eye(2) zeros(2)];
D = [zeros(2)];
```

under state, input and variations input constraints:

```
% constraints
umin=-1*0.25;
umax=1*0.25;
delta_u_min=-0.1;
delta_u_max=0.1;

ymin=-10;
ymax=10;
```

Goal: Control the system output so that it's stabilizing while satisfying constraints.

Solution: solve an MPC problem with a quadratic cost as in (7) taking into account the model of the system and constraints.

First, we formulate the problem in Matlab.

```
% save the system dimension
[dx,du]=size(B);
dy=size(C,1);

% weighting matrices
Q=eye(dx);  % cost for the state x
Qy=eye(dy); % cost for the output y
P=10*Q;     % cost for the terminal cost
R=1;        % cost for the input u

Npred=5;  % length of the prediction horizon
Nsim=100; % length of the simulation horizon
```

Next, we define the optimization variables and write the optimization problem in the Yalmip syntax.

```
% define the optimization variables, constraints and cost
u = sdpvar(repmat(du,1,Npred),ones(1,Npred));
x = sdpvar(repmat(dx,1,Npred+1),ones(1,Npred+1));
utmp=sdpvar(du,1);
u_init=sdpvar(du,1); % due to input variation constraints we need to give
    an inital value for the input

% initialize the constraints and objective
constraints=[];
objective=0;


% write the constraints and the objective over the prediction horizon
for k=1:Npred
    if ( k==1 )
        utmp=u_init;
    else
        utmp=u{k-1};
    end
   constraints=[constraints,...
       x{k+1}==A*x{k}+B*u{k},...            % dynamics
       umin<=u{k}<=umax,...                 % input magnitude constraints
       ymin<=C*x{k}+D*u{k}<=ymax,...        % state magnitude constraints
       delta_u_min<=u{k}-utmp<=delta_u_max]; % input magnitude variation
    constraints
   objective=objective+x{k}'*Q*x{k}+u{k}'*R*u{k};  % quadratic cost
    function

end
 objective=objective+x{Npred+1}'*P*x{Npred+1};     % add the terminal cost

 %options=sdpsettings('verbose',1,'solver','+cplex'); %force here cplex as
    a solver, not strictly necessary; what is important is the + sign which
```

```
    tells yalmip that the problem is not nonlinear (it's getting confused
    by the parameters)
 options=[];

% compute the controller
%-1st and 2nd arguments are the constraints and the objective;
%-3rd are the options or sdpsettings, and most importantly:
%-4th represents the variables we consider to be the input (the variables
    that changes in the problem,i.e.,parameters) and
%-5th represents the variables we consider to be the output (the decision
    variables)

 parameters={x{1}, u_init};
 output={u{1}};
 controller = optimizer(constraints,objective,options,parameters,output);
   %define here the parameters and inputs
```

Next, we solve the optimization problem defined above over a simulation horizon.

```
% simulation loop
usim=zeros(du,Nsim);
ysim=zeros(dy,Nsim);
xsim=zeros(dx,Nsim+1);
xsim(:,1)=rand(dx,1);
usim_init=zeros(du,1);


for i=1:Nsim
   u=controller{{xsim(:,i),usim_init}};
   usim_init=u;
   xsim(:,i+1)=A*xsim(:,i)+B*u;            % update the dynamics
   usim(:,i)=u;
   ysim(:,i)=C*xsim(:,i)+D*usim(:,i);      % update the dynamics
end
```

Finally, illustrate the result.

```
figure
stem(ysim);
title('ysim');
figure
scatter(xsim(1,:),xsim(2,:));
title('xsim');
figure
stem(usim);
title('usim');
```

References

[1] C.R. Cutler et al. *Method for removal of PID dynamics from MPC models*. US Patent 7,263,473. 2007.

[2] Johan Löfberg. “YALMIP: A toolbox for modeling and optimization in MATLAB”. In: *Computer Aided Control Systems Design, 2004 IEEE International Symposium on*. IEEE. 2004, pp. 284–289.

[3] David Q. Mayne et al. “Constrained model predictive control: Stability and optimality”. In: *Automatica* 36 (2000), pp. 789–814.

[4] R.M. Murray. “Optimization-Based Control”. In: *Technical Report, California Institute of Technology, CA* (2009).

[5] A.I. Propoi. “Use of linear programming methods for synthesizing sampled-data automatic systems”. In: *Automation and Remote Control* 24.7 (1963), pp. 837–844.

[6] J. Richalet and D. O’Donovan. *Predictive Functional Control: Principles and Industrial Applications*. Springer, 2009.