Exercise 3: Model Predictive Control for a Crane

Boris Houska Hans Joachim Ferreau Moritz Diehl

1 Model Predictive Control for a Crane

In this exercise, we consider once again the crane model from the previous exercises. This time, we will use the capabilities of the ACADO Toolkit for closed-loop control of our crane model. We aim at bringing the crane back to rest after disturbances have occured. The non-trivial part of this exercise is to implement the real-time iteration loop calling controller preparation and feedback and process simulation steps in the right order. However, the following exercise sections will guide you step-by-step:

1.1 Implementation of the Controller Model

The first part of this exercise is basically a repition of previous the exercise: we setup the model equations which we want to use in the controller setup. For this aim, create a "cpp" file with suitable name in the `examples/simulation_environment` folder in `ACADOtoolkit` and start to implement the following:

```
#include <acado_toolkit.hpp>
#include <gnuplot/acado2gnuplot.hpp>

int main( ){

    USING_NAMESPACE_ACADO;

    // VARIABLES:
    // ------------------------
    DifferentialState x, v, phi, dphi;
    Control                       ax;

    // DIFFERENTIAL EQUATION:
    // ------------------------
    DifferentialEquation     f;    // The model equations

    f << dot(x) ==  v;
    f << dot(v) ==  ...   // copy crane model from the time optimal control exercise ...
```

1.2 Implementation of the Online Least-Square Minimization Problem

In the section, we learn how to setup a least-square optimization problem in ACADO, which can be used within an MPC controller. More precisely, we plan to implement an objective functional of the form

$$\Phi[x(\cdot),u(\cdot)] \quad := \quad \frac{1}{2}\int_{t_i}^{t_i+T} \|Q^{\frac{1}{2}}\left(h(x(t),u(t))-r(t)\right)\|_2^2 \, . \qquad (1)$$

Here, $r:\mathbb{R}\to\mathbb{R}^{n_h}$ is a tracking reference, $Q\in\mathbb{R}^{n_h\times n_h}$ and $h:\mathbb{R}\to\mathbb{R}^{n_h}$ the output function for which we want to achieve optimal tracking.

In our example, we use $n_h = 4$ planning to penalize the deviation of all states from r. In ACADO, r does not need to be defined when the optimal control problem for the controller is setup as the reference might in practice not be known in advance. Thus, we may intepret r as a kind of "place-holder" within the following code:

```
// DEFINE LEAST SQUARE FUNCTION:
// ----------------------------
Function h;

h << x;
h << ... // implement the other components of the function h.

Matrix Q(4,4); // LSQ coefficient matrix
Q.setIdentity();

Vector r(4); // Reference


// DEFINE AN OPTIMAL CONTROL PROBLEM:
// ----------------------------------
const double t_start = 0.0;
const double t_end   = 5.0;

OCP ocp( t_start, t_end, 25 );
ocp.minimizeLSQ( Q, h, r );

 ... // implement the remaining constraints.
```

The control input $a_x(t)$ should be between $-5\,\frac{\mathrm{m}}{\mathrm{s}^2}$ and $5\,\frac{\mathrm{m}}{\mathrm{s}^2}$ while running the closed loop system. Which other constraints do you need to implement in an MPC setup?

Note that in the real-time mode the initial value constraints will later automatically be added by ACADO, as they are only known once the state is measured (or estimated).

1.3 Implementation of the Real-Time Iterations

We start with a nominal setup, i.e. the model we use for control is identical with the one we use for simulating the controlled process. In this first setup we will also constraint ourselves to the case that no time dependent disturbance occurs. Recall from the previous section that our objective is to bring the crane to rest, thus we penalize the deviations of all states from the steady-state at the origin (i.e. we plan to use the reference $r(t) = 0$ for all times t in the horizon).

1.3.1 Setup of the Process Simulator

We start with the setup of the process which we want to simulate. For this aim, we use the trivial setup:

```
OutputFcn identity;
DynamicSystem dynamicSystem( f, identity );
Process process( dynamicSystem,INT_RK45 );
```

Recall that f is the differential equation that we have setup above. Being at this point we are going to simulate the process for a short interval in order to see whether our simulation model is up and running in an open-loop mode.

```
Vector x0(4); x0(0) = ...; x0(1) = ... ; ...
Vector uCon(1); uCon(0) = ... ;

process.init( startTime, x0, uCon );
process.step( startTime, endTime, uCon );
```

Invent some values for the initial state x_0 and the control input u which leads to a reasonable output to be checked for correctness.

```
VariablesGrid ySim;
process.getY( ySim );
ySim.print();
```

In the online mode, the process will only be initialized once, while the method `step` can be called with the current simulation interval and the current control input.

1.3.2 Setup of the NMPC controller

In this part we setup an NMPC controller with with a control horizon of 5 seconds. For controlling the crane we will run the controller at a sampling time of 0.1 seconds. At each sample we would like to implement one real-time algorithm step. The aim is to simulate the whole closed-loop system for 20 seconds.

Before, we implement the real-time loop, we initialize the controller with the following syntax:

```
double samplingTime = 0.1;
RealTimeAlgorithm alg( ocp, samplingTime );

alg.set( USE_IMMEDIATE_FEEDBACK,YES );
alg.set( GLOBALIZATION_STRATEGY,GS_FULLSTEP );

StaticReferenceTrajectory zeroReference;
Controller controller( alg, zeroReference );

controller.init( startTime,x0 );
```

Note that the class `RealTimeAlgorithm` can be used analogously to the class `OptimizationAlgorithm` which we have used in the offline setup. Initialize the simulation at the state $(0, 0, 0, 5)^T$.

The following lines of code introduce the syntax which is needed for the real-time iterations.

```
// PERFORM PREPARATION STEP:
// -------------------------
controller.preparationStep( );

// PERFORM "STATE MEASUREMENT":
// -------------------------
process.getY( ySim );
currentState = ySim.getLastVector();

// PERFORM IMMEDIATE FEEDBACK STEP:
// ------------------------------
controller.feedbackStep( currentTime, currentState );
controller.getU( uCon );

// SEND CONTROL IMMEDIATELY TO PROCESS:
// (AND SIMULATION)
// ---------------------------------
process.step( currentTime, currentTime+samplingTime, uCon );

++nSteps;
currentTime = (double)nSteps * samplingTime;
```

The exercise is now to write the closed-loop around this code and think about how the results can be exported. In order to understand the motivation for real-time iterations, measure the time which is needed for the preparation and the feedback step repsectively (avoid plotting or writing files while measuring these computation times). What would you change in the above real-time loop if you would run the controller on a real-world system?

Tip: You can have a look into the example

`example/simulation_environment/active_damping_stepped.cpp`

where a similar closed loop problem is implemented. This example also explains how to use the Gnuplot interface for plotting the controller reaction in online-mode.

1.4 Simulating Model-Plant Mismatch

In practice we never have an exact model of the process to be controlled, i.e. a model-plant mismatch is present. We want to simulate this situation by introducing a different crane model for the `Process`, while keeping the original one in the `Controller`. Define a second crane model directly below the first one by adding the following lines:

```
DifferentialEquation fSim;
L = 1.2;                        // introduce model plant mismatch
                                // by choosing a different line length
fSim << dot(x) ==  v;
fSim << dot(v) ==  ax;
fSim << dot(phi ) == dphi;
fSim << dot(dphi) == -g/L*sin(phi) -ax/L*cos(phi) - b/(m*L*L)*dphi;
```

For using this second model with a different line length, change the `Process` definition as follows:

```
OutputFcn identity;
DynamicSystem dynamicSystem( fSim,identity );

Process process( dynamicSystem,INT_RK45 );
```

Run the simulation again and compare with the nominal case. Also compare the behaviour of the standard optimization with the real-time iteration scheme. If you like, play around with different ways of introducing model-plant mismatch and find out how far you can go until the controller fails to bring the crane to rest.

1.5 Simulating Unknown Disturbances

Besides model-plant mismatch, unknown disturbances (i.e. ones that are not visible to the controller) are the main reason for using feedback control. Test how the controller behaves in the presence of disturbances by introducing them into the simulation. You can do this as follows:

(i) Introduce a disturbance variable:

```
Disturbance W;              // disturbance
```

(ii) Let this disturbance act on the second equation of the Process model:

```
fSim << dot(v) == ax + W;
```

(iii) Setup a file, say `dist.txt`, containing a time series discribing the disturbances:

```
TIME      W

 0.0     0.00
 3.0     0.00
 3.2     2.00
 3.4     2.00
 3.6     0.00
12.0     0.00
12.2    -5.00
12.4    -5.00
12.6     0.00
20.0     0.00
```

(iv) Add your disturbance to the `Process` by adding the following two lines after its definition:

```
VariablesGrid disturbance = readFromFile( "dist.txt" );
process.setProcessDisturbance( disturbance );
```

(v) Plot your disturbance by adding a subplot:

```
GnuplotWindow window;
    // ...
    window.addSubplot( disturbance, "Disturbance [m/s^2]" );
```

Run the simulation again and compare with the previous cases. Also compare the behaviour of the standard optimization with the real-time iteration scheme. If you like, change the definition of the disturbances and see how far you can go until the controller fails to bring the crane to rest.