muAO-MPC Documentation

Release 1.0.0-beta

P. Zometa, M. Kögel, R. Findeisen

Institute for Automation Engineering
Chair for Systems Theory and Automatic Control

Oct 04, 2021

CONTENTS

Warning: *muAO-MPC* is on beta. Some functionality is still quite fragile. See the list of *Known issues* for details.

Warning: This document is still a work in progress. If you find errors, or have any comments, please create an issue in the github repository

CHAPTER

ONE

INTRODUCTION

1.1 What is muAO-MPC?

μAO-MPC is a code and data generation tool for model predictive control. *μAO-MPC* is free software released under the terms of the 3-clause BSD license.

μAO-MPC mainly consists of the `muaompc` Python package.

1.1.1 The muaompc package

This package automatically generates self-contained C-code specific for a model predictive control (MPC) problem. The generated code consist of C-code for the problem, and C-code for data corresponding to that problem. The problem, as well as the data, are specified by the user in two separate text files using a high-level description language. The generated C-code is a ready-to-use fast MPC implementation.

The generated C code is fully compatible with the ISO C89/C90 standard, and is platform independent. The code can be directly used in embedded applications, using popular platforms like Arduino, and Raspberry Pi, or any other application on which C/C++ code is accepted, like many current generation programable logic controllers (PLC). Additionally, MATLAB/Simulink interfaces to the generated code are provided.

At the moment, we consider the following types of systems:

- linear discrete-time sytems (the `ldt` module)

1.1.2 The ldt module

This module creates a model predictive control (MPC) controller for a linear discrete-time (*ldt*) system with input and (optionally) state constraints, and a convex cost function. The MPC problem is reformulated as a condensed convex mathematical program, which can be solved using off-the-shelf optimization algorithms, or the ones included in muaompc.

1.2 Installation

1.2.1 Dependencies

The following packages are required:

- Python interpreter. This code has been fully tested with Python versions 3.8.5.
- NumPy, tested with versions 1.21.2. It basically manages the linear algebra operations, and some extra features are used.
- SciPy, tested with versions 1.7.1. Some features not included in numpy are used.
- PyParsing, tested with versions 2.4.7. It is used for parsing the problem.

- A C89/C90 compiler. To compile the generated code, a C/C++ compiler that supports C89/C90 or later standards is required.

Optional packages are:

- Cython to compile the Python interface to the generated C-code. Tested with version 0.29.24.

1.2.2 Building and installing

The easiest way to install `muaompc` and all its dependencies is via pip:

```
pip install muaompc
```

Alternatively, you can install `muaompc` directly from source code.

Install from source in Linux and Mac OS X

Linux and OS X users typically have all required (and most optional) packages already installed. To install `muaompc`, switch to the directory where you unpacked `muaompc` (you should find a file called `setup.py` in that directory) and in a terminal type:

```
python setup.py install --user --force
```

The `--user` option indicates that no administrator privileges are required. The `--force` option will overwritte old files from previous installations (if any). Alternatively, for a global installation, type:

```
sudo python setup.py install --force
```

And that is all about installing the package. The rest of this document will show you how to use it.

Install from source in Windows Systems

For Windows users, we recommend installing the Anaconda platform, as it contains all of the necessary python packages.

For a full installation of `muaompc` do the following:

- Install Anaconda.
- Open an `Ananconda Prompt`, switch to the directory where you unpacked `muaompc` (the one containing the file `setup.py`), and type:

  ```
  python setup.py install --force
  ```

1.3 About the developers

μAO-MPC has been developed at the Laboratory for System Theory and Automatic Control, Institute for Automation Engineering, Otto-von-Guericke University Magdeburg, Germany. The main authors are Pablo Zometa, Markus Kögel and Rolf Findeisen. Additional contributions were made by Sankar Datta, Sebastian Hörl, and Yurii Pavlovskyi.

If you have some comment, suggestions, bug reports, etc. please create an issue in the github repository

1.4 Changelog

- Version 1.0.0-beta
  - Add Python/Cython interface to generated code
  - Improve parsing error messages
  - Add advanced tutorial and more examples
- Version 1.0.0-alpha
  - Initial alpha release. (2016.07.18)
  - Fix bug in data generation for problems with more than one parameter (2016.08.05)

1.4.1 Known issues

As an beta release, there are several things that are stil fragile, namely:

- Version 1.0.0-beta
  - The parsing capabilities are very limited. Only the most common cases are at the moment supported (see the examples)
  - The SIMULINK interface is still missing.

CHAPTER

TWO

TUTORIAL

In this chapter we present a very simple step-by-step tutorial, which highlights the main features of `muaompc`. We start with an overview of the code generation process followed by a tutorial using a simple example.

2.1 Code generation at a glance

The MPC problem description is written in a file called the *problem* file. The data file for a specific problem is given in a different text file called the *data* file.

After writing these files, the next step is to actually auto-generate the C code. This is done in two easy steps:

1. create an mpc object from the problem file.
2. generate code for the data using the newly created mpc object and the data file.

The first step will automatically generate the C-code for the problem. The second step will generate code for the data for two cases: static memory allocation and dynamic memory allocation. In the first case, the data consists on several C files that statically allocate memory and need to be compiled. In the second case, a single data file in json format is written. This data contained in the json file that can be dynamically loaded. The first case (C-code) is useful for deployment in embedded systems, whereas the second case (json file) allows more flexibility during simulation (no need to compile the data).

2.2 A basic MPC problem

The code generation described in this section basically consist of the following steps:

1. write the problem file with the MPC problem description,
2. write a data file corresponding to the problem,
3. create a `muaompc` object out of that problem, and
4. create data from that object based on the data file.

The simplest problem that can be setup with the `ldt` module is an input constrained problem. The code for this example can be found inside the *tutorial* directory `muaompc_root/examples/ldt/tutorial`, where `muaompc_root` is the path to the root directory where `muaompc` sources were extracted.

2.2.1 The MPC setup description

Consider the following setup. The plant to be controlled is described by the prediction model $x^+ = Ax + Bu$, where $x \in \mathbb{R}^n$, and $u \in \mathbb{R}^m$ are the current state and input vector, respectively. The state at the next sampling time is denoted by x^+. The discrete-time system and input matrices are denoted as A and B, respectively.

The inputs are constrained to be in a box set $\mathcal{C}_u = \{u \mid u\_lb \leq u \leq u\_ub\}$.

The MPC setup is thus as follows:

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \quad & \sum_{i=0}^{N-1} (\|x_i\|_Q^2 + \|u_i\|_R^2) + \|x_N\|_P^2 \\ \text{subject to} \quad & x_{i+1} = Ax_i + Bu_i, \quad i = 0, \cdots, N-1 \\ & u\_lb \leq u_i \leq u\_ub, \quad i = 0, \cdots, N-1 \\ & x_0 = \bar{x} \end{aligned}$$

where the integer $N \geq 2$ is the prediction horizon. The symmetric matrices Q, R, and P are the state, input, and final state weighting matrices, respectively. The vector $\bar{x}$ represents the system state at the current sampling time, which is given online as a parameter to the optimization problem.

The optimization variable $\mathbf{u} \in \mathbb{R}^{Nm}$ is defined as the sequence $\mathbf{u} = \{u_0 \; u_1 \; \dots \; u_{N-1}\}$. Similarly, we define the (auxiliary) state sequence $\mathbf{x} = \{x_0 \; x_1 \; \dots \; x_N\}$, with $\mathbf{x} \in \mathbb{R}^{(N+1)n}$.

2.2.2 The MPC problem file

The MPC setup can be rather intuitively described in the problem file. In your favorite text editor write the following:

```
variable u[0:N-1](m);
auxs x[0:N](n);
parameters x_bar(n);
minimize sum(quad(x[i],Q)+quad(u[i], R), i=0:N-1)+quad(x[N],P);
subject to x[i+1] = A*x[i]+B*u[i], i=0:N-1;
u_lb <= u[i] <= u_ub, i=0:N-1;
x[0]=x_bar;
```

Save the file as `myprb.prb`. The resemblance of this file to the mathematical description should be apparent. Let us make a few remarks about the notation.

The problem description is based on *sequences*, which are defined using the format `v[a:b](m)`, where `v` is the name of the sequence, `m` is the length of each vector in the sequence, and `a` and `b` denote the index of the first and last element of the sequence. To refer to the element `i` of this sequence, we use the notation `v[i]`. In the case that a sequence consist of a single element, only the vector length need to be specified, i.e. `v(m)`. To refer to this vector no indexing is required in later parts of the problem specification, i.e. it suffices to write `v`.

The `variable` keyword identifies the optimization variable. The `parameters` keyword identify sequences that are to be specified online. The keyword `auxs` is used to specify the dimensions of the sequence `x`.

The keyword `minimize` identifies the text following it as the cost function to be minimized. Several special keywords are accepted, for example the `sum(h[i], i=a:b)` denotes the summation of the real valued functions h_i for i=a,…, b. The keyword `quad(v,M)` denotes the quadratic form $\|v\|_M^2$.

As seen in problem file, the equality and inequality constraints follow after the keyword `subject to`. The constraints optionally accept an index variable and its range. For example, the prediction model is written as `x[i+1] = A*x[i] + B*u[i], i=0:N-1`.

With the problem file already finished, we can now write the data file.

2.2.3 The MPC data file

The data file must contain the numerical values for the matrices, vectors, and scalars found in the problem file. Values for the sequences defined by the keywords `variable`, `parameters` and `aux` are not required. Thus, in our example, values for `u`, `x_bar`, and `x` do *not* need to be specified. The following elements need to be specified: the matrices `Q`, `R`, `P`, `A`, `B`, the vectors `u_lb`, `u_ub`, and the scalars `N`, `m`, `n`.

In this tutorial, we provide the data in a `.dat` file. This is great for simple matrices that can be fitted into one line.

Note: The data file can also be a python module. See *A more complex MPC problem* for an example.

In the `.dat` format, the matrices are specified using MATLAB syntax. For example, an identity matrix $I \in \mathbb{R}^{2\times 2}$ could be written in the following ways:

```
I = [1 0; 0 1]
I = [1, 0; 0, 1]
```

Without going into further details, let us write data file. In your favourite text editor write:

```
# weighting matrices
Q = [1, 0; 0, 1]
R = [1]
P = [1, 0; 0, 1]
# system matrices
A = [1.,   0.01; 0.,   0.9]
B = [1.e-04; 0.02]
# input constraints
u_lb = [-100]
u_ub = [100]
# dimensions
N = 5
n = 2
m = 1
```

Note: At the moment, each matrix or column vector must be described in a single line.

Save this file as `mydat.dat`. The matrices A and B represent the discrete time model of a DC-motor. The state vector is given by $x = [x_1\ x_2]^T \in \mathbb{R}^n$, where x_1 and x_2 are the rotor position and angular speed, respectively. The input is constrained to be between -100% and 100%.

For this example, we chose the weighting matrices to be identity matrices of appropriate size, i.e. $P = Q = I \in \mathbb{R}^{n\times n}$, and $R = 1$. Clearly, the value of dimension of the state and input vector are $n = 2$ and $m = 1$. The horizon length is specified as steps through the parameter $N = 5$.

2.2.4 Generating the C-code

Now that we have written the `myprb.prb` problem file, we proceed to create an `mpc` object. In the directory containing `myprb.prb`, launch your Python interpreter and in it type:

```
from muaompc import ldt

mpc = ldt.setup_mpc_problem('myprb.prb')
```

This will generate code specific for the problem described by `myprb.prb`. By itself, the code we just generated is very not useful. It only contains and abstract description of an MPC problem without any data. The next step is to generate code for data that can be used with the problem code for myprb.prb we just generated. To generate code that represents the data in mydat.dat, continue typing in your Python interpreter:

```
ldt.generate_mpc_data(mpc, 'mydat.dat')
```

And that's it! If everything went allright, you should now see inside current directory a new folder called mpc_myprb. As an alternative to typing the above code, you can execute the file main.py found in the *tutorial* directory, which contains exactly that code. The *tutorial* directory already contains the files myprb.prb and mydat.dat. In the next section, you will learn how to use the generated C code.

Tip: If the code generation was not succesful, try passing the verbose=True input parameter to the function setup_mpc_problem. It will print extra information about the code generation procedure. For example:

```
mpc = ldt.setup_mpc_problem('myprb', verbose=True)
```

Tip: By default, the generated code uses double precision float (64-bit) for all computations. You can specify a different numeric representation via the input parameter numeric of the function setup_mpc_problem. For example, to use single precision (32-bit) floating point numbers type:

```
mpc = ldt.setup_mpc_problem('myprb.prb', numeric='float32')
```

Structure of the generated code

In general terms, the generated code is structured as follows:

```
+ <prefix>_<prb_name>
  + src
    - C code + interfaces
  - <prefix>setup.py
  + data
    + <dat_name>
    + <dat_name_1>
    + ...
```

The folder where all the generated code is placed has a name in the form <prefix>_<prb_name>, where <prefix> is a keyword argument passed to setup_mpc_problem, and <prb_name> is the name of the problem file used to generate the code. The default prefix value is 'mpc'. For example, to change the default prefix to something like xyz, call:

```
mpc = ldt.setup_mpc_problem('myprb.prb', prefix='xyz')
```

For instance, in this tutorial the problem file is called myprb.prb, and no prefix is specified (i.e. <prefix>=mpc), then <prb_name>=myprb, and the directory for the generated code is mpc_myprb.

Inside the src folder, the code for solving a problem is generated: the C-code, and the Cython and MATLAB interfaces. All C-file names start with <prefix>, which creates a sort of *name space*. This allows you to have several generated code coexist in a single application, as long as each <prefix> is unique. Similarly, the Cython and Matlab interfaces use the prefix as part of the interface name.

The <prefix>setup.py file is used to compile the Cython interface (see next section).

Finally, inside the <prefix>_<prb_name> folder, you will find the data folder. In it, you will find the <dat_name> folder, which contains the generated data files for the <dat_name>.dat file. For each <dat_name>.dat MPC data file for which the call ldt.generate_mpc_data(mpc, '<dat_name>.dat') is made, a folder <dat_name> will be generated inside the data subfolder. This allows to generate different data sets (e.g. a <dat_name_1>.dat with different weighting matrices) for the same problem. This can be useful for controller tuning.

For instance, in this tutorial, inside the mpc_myprb folder, you will find the data folder, which in turn contains the mydat folder. mydat stores the generated data files for the data file mydat.dat that correspond to the MPC problem myprb.prb.

2.2.5 Using the generated code

In the folder mpc_myprb you will find all the automatically generated code for the current example. To use the generated code in a control loop, the following steps are to be followed:

1. setup a MPC controller
2. configure the optimization algorithm
3. set the parameters for the MPC controller
4. solve the MPC problem
5. apply the control input
6. repeat from step 3

We now proceed to exemplify the use of the generated code from steps 1 to 5. We start our tutorial using the Python interface, as it is simpler to explain. Later we show how it is done in pure C, and using the MATLAB interface.

Using the generated code in Python

The Python interface makes it possible to directly make use of the generated code and data (i.e. the MPC controller) from within Python.

Once the code has been generated, the next step is to compile the Python interface. Technically, we use Cython to define a C-extension to Python.

In a console/terminal change to *tutorial* directory muaompc_root/examples/ldt/tutorial. Change then to the generated code folder mpc_myprb. To install the Python extension, execute the mpcsetup.py installation script:

```
python mpcsetup.py install --force
```

If everything went ok, you should see no errors, and the last three lines should be (tested in Ubuntu 20.04):

```
Installed <>.egg
Processing dependencies for mpc==1.0
Finished processing dependencies for mpc==1.0
```

where <> is a general place holder.

Now you can use the interface which is encapsulated in a package called mpc which represents the MPC controller. The Python package's name is the same as the <prefix> used during code generation.

While in the folder mpc_myprb, fire up your Python interpreter, and type:

```
from mpc import mpcctl
```

The next step is to declare an instance of the class mpcctl.Ctl, which we usually call ctl (*controller*). The input parameter for the constructor of the class is the name of a json file contaning the generated data. In this example, the data is saved in the folder mpc_myprb/data/mydat. In our example, the generated json data file is called mpcmydat.json. Continue typing in the console:

```
ctl = mpcctl.Ctl('data/mydat/mpcmydat.json')
```

The next step is to configure the optimization algorithm. In this case, we have an input constrained problem. The only parameter to configure is the number of iterations of the algorithm (see section *Tuning* for details). For this simple case, let's set it to 10 iterations:

```
ctl.conf.in_iter = 10;
```

Let us assume that the current state is $\bar{x} = [0.1 \;\; -0.5]^T$. The controller object has a field for the parameters defined in the problem file. The parameter `x_bar` can be set as follows:

```
ctl.parameters.x_bar[:] = [0.1, -0.5]
```

We can finally solve our MPC problem for this state by calling:

```
ctl.solve_problem();
```

The solution is stored in an array `ctl.u_opt`, whose first `m` elements are commonly applied to the controlled plant. Print the optimal input vector `ctl.u_opt`, and if everything went okay, you should see the following:

```
print(ctl.u_opt)
array([0.03056814, 0.02406793, 0.0178332 , 0.01179073, 0.00586953])
```

Using the generated code in MATLAB

The MATLAB interface makes it possible to directly make of the generated code and data (i.e. the MPC controller) from within MATLAB.

Once the code has been generated, the next step is to compile the MATLAB interface.

Start MATLAB, and switch to the folder `mpc_myprb/src/matlab`. In the MATLAB console type `mpcmake`, which will execute the `mpcmake.m` script. The last step is to add the `matlab` directory to the PATH environment in MATLAB. For example, assuming the MATLAB current directory is the tutorial directory `muaompc_root/examples/ldt/tutorial`, in the MATLAB console type:

```
cd mpc_myprb/src/matlab
mpcmake
cd ..
addpath matlab
```

Now you can use the interface which is encapsulated in a class called `mpcctl` which represents the MPC controller. Simply declare an instance of that class, which we usually call `ctl` (*controller*). The input parameter for the constructor of the class is the name of a json file contaning the generated data. `muaompc` by default saves the data in the folder `mpc_myprb/data/mydat`. In our example, the generated json data file is called `mpcmydat.json`. Continue typing in the console:

```
ctl = mpcctl('mpc_myprb/data/mydat/mpcmydat.json');
```

The next step is to configure the optimization algorithm. In this case, we have an input constrained problem. The only parameter to configure is the number of iterations of the algorithm (see section *Tuning* for details). For this simple case, let's set it to 10 iterations:

```
ctl.conf.in_iter = 10;
```

Let us assume that the current state is $\bar{x} = [0.1 \;\; -0.5]^T$. The controller object has a field for the parameters defined in the problem file. The parameter `x_bar` can be set as follows:

```
ctl.parameters.x_bar = [0.1; -0.5];
```

We can finally solve our MPC problem for this state by calling:

```
ctl.solve_problem();
```

The solution is stored in an array `ctl.u_opt`, whose first m elements are commonly applied to the controlled plant. The complete MATLAB example can be found in the tutorial folder under `main.m`.

Using the generated code in C

The folder mpc_myprb/data/mydat already contains a template for a main file, called mpcmydatmain.c. Switch to the the folder mydat and open mpcmydatmain.c in your favourite editor. This template file shows how to solve an MPC problem using dynamic or static memory allocation. This file might look at bit daunting at first, but it just a template you can modify to fit your needs.

In the current directory you will find two main files with just the basics, that are based on the template file. The file mpcmydatmain_dynmem.c exemplifies how the dynamic memory allocation is done in C code. The file mpcmydatmain_staticmem.c exemplifies the how the static memory allocation version of the data can be used. Both files follow the same structure as the MATLAB tutorial above.

Let us take the file mpcmydatmain_staticmem.c as example.

The first thing to include is the header file of the library called mpcctl.h.

We need to have access to some of the algorithm's variables, for example the MPC system input, the parameters, and the algorithm settings. This is done through the fields of the struct mpc_ctl structure, which we denote the *controller* structure. We first create an instance of this controller structure, and we set the controller by passing a pointer to the structure to the function mpcmydat_ctl_setup_ctl, which is found in mpcmydatctldata.h. For example, after including the corresponding headers, in the body of the main function we type:

```
struct mpc_ctl ctlst;  /* Structure for static memory allocation */
struct mpc_ctl *ctl;  /* pointer to the an allocated structure */

ctl = &ctlst;
mpcmydat_ctl_setup_ctl(ctl);
```

Once we controller is setup, we can continue in a similar fashion to the MATLAB case, that is first we setup the parameters, then we configure the algorithm, solver the problem:

```
ctl->parameters->x_bar[0] = 0.1;
ctl->parameters->x_bar[1] = -0.5;
ctl->solver->conf->in_iter = 10;
mpc_ctl_solve_problem(ctl);
```

Finally, the computed control input is found in the array ctl->u_opt.

Note: At the moment the user needs to know the length of the different arrays in the controller structure. This information can be infered by the user from the problem and data files. The length of the different arrays will be available in the controller structure in future releases.

To run an compile this code do the following. Copy the file mpcmydatmain_staticmem.c into the folder mpc_myprb/data/mydat and remove the main template file mpcmydatmain.c found in mpc_myprb/data/mydat (otherwise you will end up with two main functions in two different files, and the compilation will fail). In that folder you will also find example Makefiles, called mpcmydatMakefile.*, which compiles the generated code. The Makefile mpcmydatMakefile.mk compiles the code using the GNU Compiler Collection (*gcc*). Adapt the Makefile to your compiler if necessary.

For example, to generate, compile and run the code in Linux you would type in a console:

```
cd muaompc_root/examples/ldt/tutorial  # the tutorial folder
python main.py  # generates code and data
cp mpcmydatmain_staticmem.c mpc_myprb/data/mydat  # the tutorial main file
cd mpc_myprb/data/mydat
rm mpcmydatmain.c  # remove template main file
make -f mpcmydatMakefile.mk  # compile
./main  # run the controller
```

If everything went okay, you will see the output:

```
ctl->u_opt[0] = 3.056814e-02
```

This concludes our tutorial!

2.3 A more complex MPC problem

In this section we consider a more elaborated example. However, the procedure to follow is just as discussed in *Code generation at a glance*: describe the problem, generate C-code from it, and finally use the generated code.

We now consider a problem that presents many of the features available in `muaompc`. The code for this example can be found inside the *tutorial advanced* directory `muaompc_root/examples/ldt/tutorial_advanced`, where `muaompc_root` is the path to the root directory of `muaompc`.

2.3.1 The MPC setup description

The system considered is the Cessna Citation 500 aircraft presented in ([M02], p.64). A continuous-time linear model is given by $\dot{x} = A_c x + B_c u, y = Cx$, where

$$A_c = \begin{bmatrix} -1.2822 & 0 & 0.98 & 0 \\ 0 & 0 & 1 & 0 \\ -5.4293 & 0 & -1.8366 & 0 \\ -128.2 & 128.2 & 0 & 0 \end{bmatrix}, \quad B_c = \begin{bmatrix} -0.3 \\ 0 \\ -17 \\ 0 \end{bmatrix}, C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -128.2 & 128.2 & 0 & 0 \end{bmatrix},$$

and the state vector is given by $x = [x_1 \; x_2 \; x_3 \; x_4]^T$, where:

- x_1 is the angle of attack (rad),
- x_2 is the pitch angle (rad),
- x_3 is the pitch angle rate (rad/s), and
- x_4 is the altitude (m).

The only input u_1 is the elevator angle (rad). The outputs are $y_1 = x_2$, $y_2 = x_4$, and $y_3 = -128.2x_1 + 128.2x_2$ is the altitude rate (m/s)

The system is subject to the following constraints:

- input constraints $-0.262 \leq u_1 \leq 0.262$,
- slew rate constraint in the input $-0.524 \leq \dot{u}_1 \leq 0.524$
- state constraints $-0.349 \leq x_2 \leq 0.349$,
- output constraints $-30.0 \leq y3 \leq 30.0$.

To consider the slew rate constraint in the input, we introduce an additional state x_5. The sampling interval is $dt = 0.5$ s, and the horizon length is $N = 10$ steps.

The controller parameters

The book [M02] proposes to use identity matrices of appropriate size for the weighting matrices Q and R. We instead select them diagonal with values that give a similar controller performance and much lower condition number of the Hessian of the MPC quadratic program, a desirable property for any numerical algorithm.

2.3.2 The MPC problem file

The MPC setup can be rather intuitively described in the problem file. In your favorite text editor write the following:

```
variable u[0:N-1](m);
auxs x[0:N](n);
parameters x_bar(n);
minimize sum(quad(x[i],Q)+quad(u[i], R), i=0:N-1)+quad(x[N],P);
subject to x[i+1] = A*x[i]+B*u[i], i=0:N-1;
u_lb <= u[i] <= u_ub, i=0:N-1;
e_lb <= Kx*x[i] + Ku*u[i] <= e_ub, i=0:N-1;
f_lb <= Kf*x[N] <= f_ub;
x[0]=x_bar;
```

With the problem file already finished, we can now write the data file.

2.3.3 The MPC data file

Observe that the system in the `muaompc` problem description is in discrete-time, i.e. `x[i+1] = A*x[i]+B*u[i], i=0:N-1;`. The continuous-time matrices A_c and B_c of the initial formulation need therefore to be discretized. Besides the `.dat` file format presented in the basic tutorial, a python or matlab file can be used as data input. In this tutorial, we use a Python `.py` module as our data file. As it is a regular Python file, we can, among other things, import `scipy` to helps discretized the continuous-time system matrices using a zero-order hold. All matrices must be defined as 2-dimensional numpy arrays (e.g. `R = np.array([[472.]])`). For this example, the data file looks like:

```
import numpy as np
from scipy.signal import cont2discrete as c2d
# weighting matrices
Q = np.diag([1014.7, 3.2407, 5674.8, 0.3695, 471.75])
R = np.array([[472.]])
P = Q
# system matrices (continuos time)
Ac = np.array([[-1.2822, 0, 0.98, 0], [0, 0, 1, 0], [-5.4293, 0, -1.8366, 0], [-128.2,
↪ 128.2, 0, 0]])
Bc = np.array([[0.3], [0], [-17], [0]])
Cc = np.array([[0, 1, 0, 0], [0, 0, 0, 1], [-128.2, 128.2, 0, 0]])
Dc = np.zeros((3,1))
# discretization
dt = 0.5
(A, B, C, D, dt) = c2d((Ac, Bc, Cc, Dc), dt)
# The system is extended with a 5th state, to account for slew rate constraints
# Extend A from a 4x4 matrix, to a 5x5 matrix
# Add first a row of zeros:
A = np.concatenate((A, np.zeros((1,4))))
# then a column of zeros
A = np.concatenate((A, np.zeros((5,1))), axis=1)
# Extend B from a 4x1 column vector, to a 5x1 column vector
# Add an new row at the bottom, with the element = 1:
B = np.concatenate((B, np.ones((1,1))))
# input constraints
u_lb = np.array([[-0.262]])
u_ub = np.array([[0.262]])
# state constraints
e_lb = np.array([[-0.349, -30, -0.25]]).T
e_ub = -1*e_lb
```

(continues on next page)

(continued from previous page)

```
Kx = np.array([[0, 1, 0, 0, 0], [-128.2, 128.2, 0, 0, 0], [0., 0., 0., 0., -1.]])
Ku = np.array([[0, 0, 1]]).T
# terminal state constraints
f_lb = e_lb
f_ub = e_ub
Kf = Kx
# dimensions
N = 10  # horizon length
n = 5  # number of states
m = 1  # number of inputs
```

2.3.4 Generating the C-code

Similarly to the *Code generation at a glance*, we proceed to create an mpc object. In the directory containing myprb.prb, launch your Python interpreter and in it type:

```
from muaompc import ldt

mpc = ldt.setup_mpc_problem('myprb.prb')
```

This will generate code specific for the problem described by myprb.prb. The next step is to generate code for data that can be used with the problem code for myprb.prb we just generated. To generate code that represents the data in mydat.py, continue typing in your Python interpreter:

```
ldt.generate_mpc_data(mpc, 'mydat.py', safe_mode=False)
```

Note that to use a Python module like mydat.py as input, we must set safe_mode=False. By default, safe_mode=True, and only the simple .dat format is accepted.

If everything went allright, you should now see inside current directory a new folder called mpc_myprb. As an alternative to typing the above code, you can execute the file main.py found in the *tutorial_advanced* directory, which contains exactly that code. The *tutorial advanced* directory already contains the files myprb.prb and mydat.py. In the next section, you will learn how to use the generated C code.

2.3.5 Using the generated C-code

The next step is to make use of the generated code. For further details on the generated code see *Code generation at a glance*.

Algorithm configuration

The next step is to configure the algorithm. In this case, we have a system with input and state constraints. The only parameters to configure are the number of iterations of the algorithm. The state constrained algorithm is an augmented Lagrangian method, which means it requires a double iteration loop (an *internal* and an *external* loop). From simulation we determine that 24 *internal* iterations, and 2 *external* iterations provide an acceptable approximation of the MPC problem using the warm-start strategy:

```
ctl.conf.in_iter = 24; /* number of internal iterations */
ctl.conf.ex_iter = 2; /* number of external iterations */
ctl.conf.warm_start = 1; /* automatically warm-start algorithm */
```

Using the generated code in Python

Just as in the *tutorial* example, we can use the Python interface to test our algorithm. Let's try doing the same using the Python interface. Go to the to the *tutorial_advanced* directory, change to the generated code folder `mpc_myprb`, and install the Python extension:

```
python mpcsetup.py install --force
```

Finally launch your Python interpreter, and in it type:

```
from mpc import mpcctl
ctl = mpcctl.Ctl('data/mydat/mpcmydat.json')
# controller solver configuration
ctl.conf.in_iter = 24
ctl.conf.ex_iter = 2
ctl.conf.warm_start = 1
# set current state
ctl.parameters.x_bar[:] = [0., 0., 0., -400., 0.]
# get solution
ctl.solve_problem()
```

The optimal input should be:

```
print(ctl.u_opt)
[-0.262      -0.13715448  0.00099704  0.03492525  0.05084397  0.04797471
0.04002588  0.03289448  0.02473329  0.01478007]
```

This concludes the advanced tutorial.

2.4 Where to go next

In the folder `muaompc_root/examples/ldt/` you will find further examples.

CHAPTER

THREE

TUNING

3.1 Basics of tuning

There are only two tuning parameters for the default optimization algorithm used by *μAO-MPC*: the number of *internal* and *external* iterations. We find that in many cases the tuning procedure is easy and intuitive. For problems without state constraints, only the number of internal iterations is of importance. These parameters are specified online.

At the moment, the selection of these parameters is made entirely by the user. In many embedded systems, the number of iterations may be limited by the processor computational power. More generally, the user may need to compare the MPC controller performance given by the solution of an exact solver (like CVXOPT) against that given by the solution of *μAO-MPC* for a given number of iterations. For example, the comparison could be made using the stage cost at each point of a given trajectory (see [ZKF13]). In the end, the precise number of iterations strongly depends on the application.

3.2 The penalty parameter

An optional third tuning value is the *penalty parameter* μ, which is selected off-line (i.e. specified in the data file). *μAO-MPC* will by default automatically compute a *good* value for μ if none is specified (recommended). Roughly speaking, a large penalty parameter implies that a low number of external iterations are required to reach good performance, especially when state constraints are active. However, more internal iterations are necessary, because the condition number of the internal problem increases. The opposite is also true, a small μ makes the internal problem easier to solve, especially if no state constraints are active. When the state constraint are active, however, the required number of external iterations is higher.

By now it should be clear that the selection of an appropriate value of μ (not too low, not too high) is crucial.

Although in general not recommended, *μAO-MPC* allows experienced users to explicitely set a value for μ in the data file. The selection of the penalty parameter μ is easily done via the function `find_penalty_parameters` in the `ldt` module. For example, using the the problem and data files from the tutorial:

```
from muaompc import ldt
res = ldt.find_penalty_parameters('myprb.prb', 'mydat.dat')
```

`res` is a dictionary that contains the keys `params` and `condnums`. In this case, each of them is a list consisting of a single element. For `params`, it is the value of μ used by default, and for `condnums` is the condition number for the algorithm corresponding to that parameter. Thus, to get default value of μ type:

```
mu = res['params'][0]
```

The default value of the penalty parameter is one that is not too high but not too low. By using the parameter `factors`, several values of multiples of the default μ can be tried at once:

```
res = ldt.find_penalty_parameters('myprb.prb', 'mydat.dat', factors=[1, 4])
```

Now, res will contain the two penalty parameters with their corresponding condition number. For example, by typing:

```
mu = res['params'][0]
mu4 = res['params'][1]
cn4 = res['condnums'][1]
```

we get in mu the default value for μ (i.e. $\mu * 1$), and mu4 has the value $\mu * 4$, corresponding to the second factor in the list given as input via factors. This may help to check that the parameter mu4 does not make the value of cn4 too high (i.e. the internal problem is ill-conditioned).

Once you have found a new value of mu that better suit your needs, you need to include that information in your data file. For example, if the new value of the penalty parameter is 123, modify the mydat.dat data file by adding the line:

```
mu = 123
```

Save the data file, and generate the data again as explained in the Section *Code generation at a glance* .

CHAPTER

FOUR

REFERENCES

BIBLIOGRAPHY

[ZKF13] Zometa, P., Kögel, M. and Findeisen, R., "muAO-MPC: A Free Code Generation Tool for Embedded Real-Time Linear Model Predictive Control," Proc. American Control Conference, Washington D.C., USA, 2013.

[KZF12] Kögel, M., Zometa, P. and Findeisen, R., "On tailored model predictive control for low cost embedded systems with memory and computational power constraints," technical report, 2012.

[KF11] Kögel, M. and Findeisen, R., "Fast predictive control of linear, time-invariant systems using an algorithm based on the fast gradient method and augmented Lagrange multipliers," in Proc. 2011 IEEE Multi-conference on Systems and Control, pages 780-785, Denver, USA, 2011.

[M02] Maciejowski, J. M., "Predictive Control with Constraints," Pearson Education, 2002.

[WAD11] Waschl, H. and Alberer, D. and del Re, L., "Numerically Efficient Self Tuning Strategies for MPC of Integral Gas Engines," in Proc. IFAC World Congress 2011, pp. 2482-2487.

[RM09] Rawlings, J., Mayne, D., "Model Predictive Control: Theory and Design," Nob Hill Pub., 2009.

[GP11] Grüne, L., Pannek, J., "Nonlinear Model Predictive Control," Springer-Verlag, 2011.

INDEX

M