CONTENTS

1 Introduction 2
  1.1 What is muAO-MPC? 2
  1.2 Installation 3
  1.3 About the developers 4
  1.4 Changelog 4

2 Tutorial 5
  2.1 A short Python tutorial 5
  2.2 A basic MPC problem 6
  2.3 A more complex MPC problem 10
  2.4 Where to go next? 13

3 MPC Controllers 14
  3.1 Creating MPC controllers 14
  3.2 Using MPC controllers 17

4 Extra features 21
  4.1 A simulation class in Python 21
  4.2 Tuning 22
  4.3 Automatic selection of stabilizing matrices 25
  4.4 MATLAB/Simulink interface 25

5 References 28

Bibliography 29

Index 30
Warning: This document is still a work in progress. If you find errors, or have any comments, please contact Pablo Zometa at pablo.zometa@ovgu.de.
1.1 What is muAO-MPC?

µAO-MPC stands for microcontroller applications online model predictive control. It mainly consists of the muao MPC Python package. This package automatically generates self-contained C-code specific for a model predictive control (MPC) problem. The problem is described in a Python module created by the user. The generated C-code is a ready-to-use fast MPC implementation. For an extended description see [ZKF13].

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

- linear time-invariant discrete-time systems (the ltidt module)

Additionally, MATLAB/Simulink interfaces to the generated code are provided. µAO-MPC is free software released under the terms of the three-clause BSD License.

1.1.1 The ltidt module

This module creates a model predictive control (MPC) controller for a linear time-invariant (lti) discrete-time (dt) system with input and (optionally) state constraints, and a quadratic cost function. The MPC problem is reformulated as a condensed convex quadratic program, which is solved using an augmented Lagrangian method together with Nesterov’s gradient method, as described in [KZF12], [KF11].

The MPC problem description is written in a file we call the system module (see The system module for details). After writing this file, the next step is to actually auto-generate the C code. This is done in two easy steps:

1. create an muao MPC object from to the system module, and
2. write the C-code based on that object.

The MPC setup

The plant to be controlled is described by \( x^+ = A_d x + B_d u \), where \( x \in \mathbb{C}_x \subseteq \mathbb{R}^n \), and \( u \in \mathbb{C}_u \subseteq \mathbb{R}^m \) are the current state and input vector, respectively. \( x^+ \) is the state at the next sampling time. The discrete-time system and input matrices are denoted as \( A_d \) and \( B_d \), respectively.
The MPC setup is as follows:

\[
\begin{align*}
\minimize_{\mathbf{u}} & \quad \frac{1}{2} \sum_{j=0}^{N-1} ((x_j - x_{\text{ref}j})^T Q (x_j - x_{\text{ref}j}) + (u_j - u_{\text{ref}j})^T R (u_j - u_{\text{ref}j}) + \\
& \quad \frac{1}{2} (x_N - x_{\text{ref}N})^T P (x_N - x_{\text{ref}N}) \\
\text{subject to} & \quad x_{j+1} = A d x_j + B d u_j, \quad j = 0, \ldots , N - 1 \\
& \quad u_{\text{lb}} \leq u_j \leq u_{\text{ub}}, \quad j = 0, \ldots , N - 1 \\
& \quad e_{\text{lb}} \leq K x_j + K u_j \leq e_{\text{ub}}, \quad j = 0, \ldots , N - 1 \\
& \quad f_{\text{lb}} \leq F x_N \leq f_{\text{ub}} \\
& \quad x_0 = x
\end{align*}
\]

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 inputs are constrained by a box set, defined by the lower and upper bound \( u_{\text{lb}} \) and \( u_{\text{ub}} \), respectively. Furthermore, the state and input vectors are also constrained (mixed constraints), using \( K_x \) and \( K_u \), and delimited by the lower and upper bounds \( e_{\text{lb}} \) and \( e_{\text{ub}} \in \mathbb{R}^q \), respectively. Additionally, the terminal state needs to satisfy some constraints defined by \( f_{\text{lb}}, f_{\text{ub}} \in \mathbb{R}^r \), and the matrix \( F \).

The sequences \( x_{\text{ref}} \) and \( u_{\text{ref}} \) denote the state and input references, respectively. They are specified online. By default, the reference is the origin.

### 1.2 Installation

#### 1.2.1 Dependencies

The following packages are required:

- **Python** interpreter. This code has been fully tested with Python version 3.2.3 and Python 2.7.3.
- **NumPy**, tested with version 1.6.2. It basically manages the linear algebra operations, and some extra features are used.
- **GCC**, tested with version 4.6. It is a popular compiler that supports the C90 and C99 standards of the C programming language. In fact, any compiler that supports the C90/C99 standards should work. This excludes Visual C++ Express Edition (it does not support C99, and thus will not work). A GCC port for Windows is MinGW.

Optionally, to get a few more features, the following are required:

- **SciPy**, version 0.11 or greater is strongly recommended. It is required for several features.
- **Slycot**. It is used to compute stabilizing matrices.

#### 1.2.2 Building and installing

**muao MPC** installation is made directly from source.

**Install from source**

**muao MPC** has been tested in Linux and Mac OS X and Windows XP. The source code can be found [here](#). Switch to the directory where you unpacked it and in a terminal type:

```
python setup.py install --user
```
The --user option indicates that no administrator privileges are required. Alternatively, for a global installation, type:

\[\texttt{sudo python setup.py install}\]

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

## 1.3 About the developers

\(\mu\text{AO-MPC}\) 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 Sebastian Hörl and Yurii Pavlovskyi.

## 1.4 Changelog

- **Version 0.3.0**
  - Improved documentation (more examples and docstrings)
  - Fully check the correctness of the MPC setup
  - Improve the "auto" keyword error messages
  - Penalty parameter is now optional
  - Check the generated QP is strictly convex
  - Add the simulation class
  - Several bug fixes
  - Some code clean up

- **Version 0.2.1**
  - Slight overall improvements
  - Improved documentation (e.g. more examples)
  - Improved Windows installation (support MSVC++ compiler)

- **Version 0.2.0**
  - Initial release.
In this chapter we present a very simple step-by-step tutorial, which highlights the main features of muaompc. We consider two examples for the ltidt module. The tutorials for this module consider two types of MPC problems:

1. a small system with input constraints only, and
2. a larger system with mixed constraints.

The former helps understand the automatic code generation procedure. The latter show some of the most common features of the muaompc package.

Before we proceed with the muaompc tutorials, we give a brief tutorial of the Python skills required to write the system module.

### 2.1 A short Python tutorial

**Note:** If you are already familiar with Python, you can skip this section. All you need to know is that the matrices describing the MPC setup can be entered as a list of lists, or as 2-dimensional NumPy arrays or NumPy matrices.

Although Python knowledge is not required to use this tool, there are a few things that users not familiar with Python should know before going into next sections:

1. white spaces count,
2. Python *lists* help build matrices,
3. NumPy provides some MATLAB-like functionality, and
4. array indices start at 0.

A detailed explanation of each item follows.

#### 2.1.1 White spaces

When writing the system module in the following sections, make sure there are no white spaces (tabs, spaces, etc.) at the *beginning* of each line.

#### 2.1.2 Python lists

Python lists are simply a collection of objects separated by commas within squared brackets. Matrices and vectors are entered as a list of numeric lists. For example, a 2x2 identity matrix is entered as:

\[
I = \begin{bmatrix}
1 & 0 \\
0 & 1 \\
\end{bmatrix}
\]

whereas a 2x1 column vector is entered as:
c = [[5], [5]]

Strictly speaking, I and c are not really matrices, but they are internally converted to such by the ltidt module.

### 2.1.3 NumPy

At the top of the system module, you can write:

```python
from numpy import *
```

This makes available some functions similar to MATLAB. Of interest are `diag`, `eye`, `zeros`, `ones`. For example, a 2x2 identity matrix can also be entered as:

```python
I = eye(2)
```

or:

```python
I = diag([1, 1])
```

The `zeros` and `ones` commands have a special notation, as they require the size of the matrix as a list. For example, the c column vector from above can be written for example as:

```python
c = 5 * ones([2, 1])
```

### 2.1.4 Indexing

Finally, a few remark on indexing. We could also create the 2x2 identity matrix as follows:

```python
I = zeros([2, 2])
I[0, 0] = 1
I[1, 1] = 1
```

Note that indexing starts at 0. Slicing rules are similar to those of MATLAB.

### 2.1.5 More information

For more details see the Python tutorial, and the NumPy for MATLAB users tutorial.

### 2.2 A basic MPC problem

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

1. write the system module with the MPC problem description,
2. create a muao MPC object out of that problem, and
3. write the C-code from that object.

#### 2.2.1 The MPC problem description

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

We consider as system a DC-motor, which can be modeled as:

\[ \ddot{y} = \frac{1}{T} (K u - \dot{y}) \]

where \( T \) is the time constant in seconds, and \( K \) is amplification factor. The continuous-time state-space representation \( \dot{x} = A_c x + B_c u \) is given by the following matrices:

\[
A_c = \begin{bmatrix} 0 & 1 \\ 0 & -1/T \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 \\ K/T \end{bmatrix}
\]

with the state vector \( x = [x_1 \ x_2]^T \), where \( x_1 \) and \( x_2 \) are the rotor position and angular speed, respectively. The input is the PWM voltage, which is the percentage of the full-scale voltage applied to the motor. It is therefore constrained to be between -100% and 100%. This constraint can be written as \(-100 \leq u \leq 100\).

As the continuous-time system should be controlled by the MPC digital controller, a suitable discretization time must be chosen. A rule of thumb is to choose the discretization time as one-tenth of the system’s time constant. In this case, \( dt = T/10 \).

Note: to discretize a system, SciPy needs to be installed.

The controller parameters

The horizon length is specified as steps through the parameter \( N \). In this case we choose the value \( N = 10 \). The additional parameters for the controller are the weighting matrices. They are usually chosen via a tuning procedure. For this example, we set them to be identity matrices of appropriate size, i.e. \( Q = I \in \mathbb{R}^{2 \times 2} \), and \( R = 1 \). Additionally we set \( P \) as "auto", which will compute it as a stabilizing matrix.

Note: to use the "auto" feature, Slycot needs to be installed.

The system module

We have now the required information to write the Python system module. The only requirements are that it should contain valid Python syntax, the name of the matrices should be as described in section The system module, and the file name should end with (the extension) .py. In your favorite text editor, type the following (note that in Python tabs and spaces at the beginning of a line do matter, see A short Python tutorial):

```python
T = 0.1
K = 0.2
dt = T / 10
N = 10

# continuous time system
Ac = [[0, 1], [0, -1/T]]
Bc = [[0], [K/T]]

# input constraints
u_lb = [[-100]]
ubar = [[100]]

# weighting matrices
Q = [[1, 0], [0, 1]]
R = [[1]]
P = "auto"
```

Save the file as sys_motor.py.
2.2.2 Generating the C-code

Now that we have written the `sys_motor.py` module, we proceed to create an `mpc` object. In the directory containing `sys_motor.py`, launch your Python 3 interpreter and in it type:

```python
import muaompc
mpc = muaompc.ltidt.setup_mpc_problem("sys_motor")
mpc.generate_c_files()
```

And that’s it! If everything went alright, you should now see inside current directory a new directory called `cmpc`. As an alternative to typing the above code, you can execute the file `main_motor.py` found in the `tutorial` directory, which contains exactly that code. The `tutorial` directory already contains the `sys_motor.py` example. 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:

```python
mpc = muaompc.ltidt.setup_mpc_problem("sys_motor", 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 `generate_c_files`. For example, to use single precision (32-bit) floating point numbers type:

```python
mpc.generate_c_files(numeric="float32")
```

2.2.3 Using the generated C-code

In the `cmpc` directory you will find all the automatically generated code for the current example. Included is also an example Makefile, which compiles the generated code into a library using the GNU Compiler Collection (`gcc`). Adapt the Makefile to your compiler if necessary.

We now proceed to make use of the generated code. Let’s create a `main_motor.c` in the current directory (i.e. one level above the `cmpc` directory). The first thing to include is the header file of the library, which is found under `cmpc/include/mpc.h`. Before we continue, there are a few things to note first. The `mpc.h` header makes available to the programmer some helpful constants, for example: `MPC_STATES` is the number of states, `MPC_INPUTS` is the number of inputs (all the available constants are found in `cmpc/include/mpc_const.h`). This helps us easily define all variables with the appropriate size. Additionally, `cmpc/include/mpc_base.h` includes the type definition `real_t`, which is the type for all numeric operations of the algorithm. It is then easy to switch the numerical type of the entire algorithm (for example, from single precision floating-point to double precision). For this example, it is by default set to double precision floating point (64-bit). With this in mind, we can declare a state vector as

```c
real_t x[MPC_STATES];
```

inside our main function.

We need to have access to some of the algorithm’s variables, for example the MPC system input and the algorithm settings. This is done through the fields of the `struct mpc_ctl` structure (declared in `mpc_base.h`). An instance of this structure has been automatically defined and named `ctl`. To access it in our program, we need to declare it inside our main function as

```c
extern struct mpc_ctl ctl;
```

The next step is to configure the algorithm. In this case, we have an input constrained case. The only parameter to configure is the number of iterations of the algorithm. For this simple case, let’s set it to 10 iterations, by setting `ctl.conf->in_iter = 10`; (See section Basics of tuning for details).

Let us assume that the current state is \( x = \begin{bmatrix} 0.1 & -0.5 \end{bmatrix}^T \). We can finally solve our MPC problem for this state by calling `mpc_ctl_solve_problem(&ctl, x);`. The solution is stored in an array of size `MPC_HOR_INPUTS` (the horizon length times the number of inputs) pointed by `ctl.u_opt`. We can get access to its first element using array notation, e.g. `ctl.u_opt[0]`. The complete example code looks like:
#include <stdio.h> /* printf */
#include "cmpc/include/mpc.h" /* the auto-generated code */

/* This file is a test of the C routines of the ALM+FGM MPC algorithm. The same routines can be used in real systems. */

int main(void)
{
    real_t x[MPC_STATES]; /* current state of the system */
    extern struct mpc_ctl ctl; /* already defined */
    ctl.conf->in_iter = 10; /* number of iterations */

    /* The current state */
    x[0] = 0.1;
    x[1] = -0.5;

    /* Solve MPC problem and print the first element of input sequence */
    mpc_ctl_solve_problem(&ctl, x); /* solve the MPC problem */
    printf("u[0] = %f \n", ctl.u_opt[0]);
    printf("\n SUCCESS! \n");

    return 0;
}

In the tutorial directory you will find, among others, a main_motor.c file with the code above, together with a Makefile. Compile the code by typing make motor (you might need to modify the provided Makefile or create your own). If compilation was successful, you should see a new executable file called motor. If you run it, the word SUCCESS! should be visible at the end of the text displayed in the console.

**Warning:** Everytime you auto generate the C files, the whole cmpc directory is deleted. For precaution, DO NOT save in this directory any important file.

## 2.2.4 Testing with Python

The Python interface presents the user with almost the same functionality as the generated code. However, Python’s simpler syntax and powerful scientific libraries makes it an excellent platform for prototyping.

Let’s compare it to the pure C implementation. Just like in the C tutorial, change to the tutorial directory, launch your Python 3 interpreter, and in it type:

```python
import muaompc
import numpy as np
mpc = muaompc.ltidt.setup_mpc_problem("sys_motor")
```

The mpc object contains many methods and data structures that will help you test your controller before implementing it in C. We’ve already learned about the method generate_c_files. The Python interface to the MPC controller will be set up automatically when you access the ctl attribute of your mpc object, without the need to compile anything. Thus, we can do exactly the same as in the C-code above with a simpler Python syntax.

Continue typing in the Python interpreter the following:

```python
ctl = mpc.ctl
ctl.conf.in_iter = 10
x = np.matrix([[0.1], [-0.5]])
ctl.solve_problem(x)
print(ctl.u_opt[0])
print("SUCCESS!")
```

Compare with the C code above.

### 2.2. A basic MPC problem
2.3 A more complex MPC problem

In this section we consider a more elaborated example. However, the procedure to follow is the same: describe the problem, generate C-code from it, and finally use the generated code.

2.3.1 The MPC problem Description

We now consider a problem that presents many of the features available in the ltitd module. The code for this example can be found inside the tutorial directory muaompc_root/muaompc/_ltidt/tutorial, where muaompc_root is the path to the root directory of muaompc.

The system description

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

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

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 \( y_3 = -128.2 x_1 + 128.2 x_2 \) 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 \(-30.0 \leq y_3 \leq 30.0.\)

To consider the slew rate constraint in the input, we introduce an additional state \( x_5. \) The sampling interval is \( dt = 0.5 \text{s}, \) 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 (see Conditioning the Hessian), a desirable property for any numerical algorithm.

The system module

In this case, our system module is called sys_aircraft.py. It has been already discretized, because the slew rate constraint is easier to include in this way. It looks as follows:
from numpy import diag  # similar effect with: from numpy import *

dt = 0.5
N = 10
mu = 100
# discrete-time system
Ad = [[ 0.23996015, 0., 0.17871287, 0., 0.],
     [-0.37221757, 1., 0.27026411, 0., 0.],
     [-0.99008755, 0., 0.13885973, 0., 0.],
     [-48.93540655, 64.1, 2.39923411, 1., 0.],
     [0., 0., 0., 0., 0.]]
Bd = [[-1.2346445 ],
      [-1.43828223],
      [-4.48282454],
      [-1.79989043],
      [1.1]]
# Weighting matrices for a problem with a better condition number
Q = diag([1014.7, 3.2407, 5674.8, 0.3695, 471.75])
R = diag([471.65])
P = Q
# input constraints
eui = 0.262  # rad (15 degrees). Elevator angle.
u_lb = [[-eui]]
u_ub = [[eui]]
# mixed constraints
ex2 = 0.349  # rad/s (20 degrees). Pitch angle constraint.
ex5 = 0.524 * dt  # rad/s * dt input slew rate constraint in discrete time
ey3 = 30.
# bounds
e_lb = [[-ex2], [-ey3], [-ex5]]
e_ub = [[ex2], [ey3], [ex5]]
# constraint matrices
Kx = [[0, 1, 0, 0, 0],
     [-128.2, 128.2, 0, 0, 0],
     [0., 0., 0., 0., -1.]]
Ku = [[0],
      [0],
      [1]]
# terminal state constraints
f_lb = e_lb
f_ub = e_ub
F = Kx

Before we continue, let us make a few remarks. We use numpy to help us build the diagonal matrices Q and R, using the function diag. Finally, compare the name of the variables used in the system module against the MPC problem described in Section The ltidt module. Additionally, the optional penalty parameter mu has been selected using the procedure described in Section Basics of tuning. Finally, the weighting matrices Q and R where transformed from identity matrices to the ones shown above using the functions presented in Section Conditioning the Hessian.

2.3.2 Generating the C-code

Now that we have written the system module, we proceed to create an instance of a muaompc.ltidt class. Change to the directory containing the file sys_aircraft.py and in a Python interpreter type:

import muaompc

mpc = muaompc.ltidt.setup_mpc_problem('sys_aircraft')
mpc.generate_c_files()

If everything went okay, you should now see a new directory called cmpc. Alternatively, switch to the tutorial...
directory and execute the file called `main_aircraft.py`, which contains the same Python code as above.

### 2.3.3 Using the generated C-code

In the `cmpc` directory you will find the automatically generated code for the current example (if you previously generated code for the basic tutorial, it will be overwritten). Included is also an example Makefile, which compiles the generated code into a library using the GNU Compiler Collection (**gcc**).

The next step is to make use of the generated code. The first part of the `main_aircraft.c` file found in the `tutorial` directory is identical to the first part of the `main_motor.c` file found in *A basic MPC problem*.

#### 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 warmstart strategy:

```c
ctl.conf.in_iter = 24; /* number of internal iterations */
ctl.conf.ex_iter = 2;  /* number of external iterations */
ctl.conf.warmstart = 1; /* automatically warmstart algorithm */
```

#### Closed loop simulation

We can finally simulate our system. We start at some state $x = (0, 0, 0, -400, 0)^T$, and the controller should bring the system to the origin. In this case, we simulate for $s=40$ steps. We solve the problem for the current state by calling `mpc_ctl_solve_problem(&ctl, x);`. The solution is stored in an array of size `MPC_HOR_INPUTS` (the number of inputs times the horizon length) pointed by `ctl.u_opt`. We can get access to its first element using array notation, e.g. `ctl.u_opt[0]`. The function `mpc_predict_next_state` replaces the current state with the successor state. The complete example code looks like:

```c
#include <stdio.h> /* printf */
#include "cmpc/include/mpc.h" /* the auto-generated code */

/* This file is a test of the C routines of the ALM+FGM MPC */
/* algorithm. The same routines can be used in real systems. */
int main(void)
{
    real_t x[MPC_STATES]; /* current state of the system */
    extern struct mpc_ctl ctl; /* already defined */
    int i; /* loop iterator */
    int j; /* print state iterator */
    int s = 40; /* number of simulation steps */

    ctl.conf->in_iter = 24; /* iterations internal loop */
    ctl.conf->ex_iter = 2; /* iterations external loop */
    ctl.conf->warmstart = 1; /* warmstart each iteration */

    /* The current state */
    x[0] = 0.;
    x[1] = 0.;
    x[2] = 0.;
    x[3] = -400.;
    x[4] = 0.;

    for (i = 0; i < s; i++) {
        /* Solve and simulate MPC problem */
```

#### 2.3. A more complex MPC problem
mpc_ctl_solve_problem(&ctl, x);  /* solve the MPC problem */  
mpc_predict_next_state(&ctl, x);  /* print first element of input sequence and predicted state */  
printf("\n step: %d - ", i);
printf("u[0] = %f; ", ctl.u_opt[0]);
for (j = 0; j < MPC_STATES; j++) {
    printf("x[%d] = %f; ", j, x[j]);
}
}
printf("\n SUCCESS! \n");
return 0;
}

Running the code

In the current directory, you will find among others, the file `main_aircraft.c` which contains the code above, together with a `Makefile`. Compile the code by typing `make aircraft` (you might need to edit your `Makefile`). If compilation was successful, you should see a new executable file called `aircraft`. If you run it, the word `SUCCESS!` should be visible at the end of the text displayed in the console.

2.3.4 Testing with Python

Let’s try doing the same using the Python interface. As usual, go to the to the `tutorial` directory, launch your Python 3 interpreter, and in it type:

```python
import muaompc
import numpy as np
mpc = muaompc.ltidt.setup_mpc_problem("sys_aircraft")
ctl = mpc.ctl
s = 40
ctl.conf.in_iter = 24
ctl.conf.ex_iter = 2
ctl.conf.warmstart = True
n, m = mpc.size.states, mpc.size.inputs
x = np.zeros([n, 1])
x[3] = -400
for i in range(s):
    ctl.solve_problem(x)
    x = mpc.sim.predict_next_state(x, ctl.u_opt[:m])
    # x = ctl.sys.Ad.dot(x) + ctl.sys.Bd.dot(ctl.u_opt[:m]) # predict
    print("step":, i, ", u[0] = ", ctl.u_opt[0], ", x = ", x.T)
print("SUCCESS!")
```

Compared to the C code above, there are a few things to note. A function similar to `mpc_predict_next_state` is available in Python under the object `sim` (see `A simulation class in Python`). This is a convenience function for C and Python that computes exactly what the line marked with `# predict` does. Note that in Python and in C the structure `ctl.sys` (the system) and many other data structures are available. The MPC object (not available in C) offers in Python additional data structures. In this example we used `mpc.size`, which contains the size of all relevant vectors and matrices. Also note that `ctl.sys.Ad` and `x` are NumPy `arrays`, therefore the need to use the `dot` method.

2.4 Where to go next?

Several examples are included in the folder `muaompc_root/examples`. Detailed explanation of µAO-MPC functionality are presented in the following chapters.
This chapter deals with the implementation of MPC controllers. This basically consists of generating the code offline and finding approximate solutions to the MPC optimization problem online. It explains in detail all the functions available to the user of \( \mu AO-MPC \). Note that this chapter is not intended as an introduction or tutorial to MPC. It rather intends to serve as reference to the software functionality.

### 3.1 Creating MPC controllers

This section will explain in detail the generation of MPC controllers for a given MPC setup.

#### 3.1.1 The MPC setup

The plant to be controlled is described by

\[
\begin{align*}
    x_{j+1} &= A_d x_j + B_d u_j, \\
    u_{j} &\in C_u \subseteq \mathbb{R}^m, \quad \text{and} \quad u_{j} \in C_u \subseteq \mathbb{R}^m
\end{align*}
\]

where \( x \in C_x \subseteq \mathbb{R}^n \), and \( u \in C_u \subseteq \mathbb{R}^m \) are the current state and input vector, respectively. \( x^+ \) is the state at the next sampling time. The discrete-time system and input matrices are denoted as \( A_d \) and \( B_d \), respectively.

The MPC setup is as follows:

\[
\begin{align*}
    \text{minimize} & \quad \frac{1}{2} \sum_{j=0}^{N-1} ((x_j - x_{\text{ref}j})^T Q (x_j - x_{\text{ref}j}) + (u_j - u_{\text{ref}j})^T R (u_j - u_{\text{ref}j})) + \\
    & \quad \frac{1}{2} (x_N - x_{\text{ref}N})^T P (x_N - x_{\text{ref}N}) \\
    \text{subject to} & \quad x_{j+1} = A_d x_j + B_d u_j, \quad j = 0, \ldots, N - 1 \\
    & \quad u_{lb} \leq u_j \leq u_{ub}, \quad j = 0, \ldots, N - 1 \\
    & \quad e_{lb} \leq K_x x_j + K_u u_j \leq e_{ub}, \quad j = 0, \ldots, N - 1 \\
    & \quad f_{lb} \leq F x_N \leq f_{ub} \\
    & \quad x_0 = x
\end{align*}
\]

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 inputs are constrained by a box set, defined by the lower and upper bound \( u_{lb} \) and \( u_{ub} \), respectively. Furthermore, the state and input vectors are also constrained (mixed constraints), using \( K_x \) and \( K_u \), and delimited by the lower and upper bounds \( e_{lb} \) and \( e_{ub} \in \mathbb{R}^q \), respectively. Additionally, the terminal state needs to satisfy some constraints defined by \( f_{lb} \), \( f_{ub} \in \mathbb{R}^r \), and the matrix \( F \).

The sequences \( x_{\text{ref}} \) and \( u_{\text{ref}} \) denote the state and input references, respectively. They are specified online. By default, the reference is the origin.


### 3.1.2 The system module

The matrices describing the MPC setup are read from the system module. It basically contains the matrices that describe the MPC setup.

- The system matrices can be given either in discrete or continuous time.
  - In the first case, they must be called $A_c$ and $B_c$, respectively (in this case SciPy is required).
  - In the discrete-time case, they must be called $A_d$ and $B_d$, respectively.
  - The zero-order hold discretization time should be specified as $dt$ in both cases.

- The state and input weighting matrices $Q$ and $R$ must be called $Q$ and $R$, respectively.

- The terminal weight matrix $P$ is called $P$. It can be declared as "auto", in which case it will be computed as a stabilizing matrix (the Python package Slycot is required).

- The lower and upper input constraints bounds $u_{lb}$, $u_{ub}$ are called $u_lb$ and $u_ub$, respectively.

- The MPC horizon length $N$ represents steps (not time) and is an integer greater than one called $N$.

Additionally, for a state constrained problem, the following are required:

- Mixed constraints
  - The lower and upper mixed constraints bounds $e_{lb}, e_{ub} \in \mathbb{R}^q$ should be called $e_{lb}$ and $e_{ub}$, respectively.
  - $K_x$, must be called $Kx$.
  - $K_u$, is optional and is called $Ku$. If not specified (or None), it is set to a zero matrix of appropriate size.

If any of $e_{lb}$, $e_{ub}$, or $Kx$ is not specified (or None), then the MPC setup is considered to be without state constraints.

- Terminal state constraints are optional:
  - The terminal state bounds $f_{lb}, f_{ub} \in \mathbb{R}^q$, are called $f_{lb}$ and $f_{ub}$, respectively.
  - $F$, the terminal state constraint matrix, must be called $F$.
  - If any of $F$, $f_{lb}$, or $f_{ub}$ is not specified (or None), each of them is set to $Kx$, $e_{lb}$, and $e_{ub}$, respectively.

- The penalty parameter $\mu$ of the augmented Lagrangian method is optional. If specified, it must be called $mu$, and must be a positive real number. If not specified (or None), it is computed automatically (recommended).

All matrices are accepted as Python lists of lists, or as numpy arrays, or numpy matrices. The system module can also be written as a MATLAB mat file containing the required matrices using the names above.

For example, an input-constrained second-order continuous-time LTI system could be described by the following system.py Python module:

```python
Ac = [[0, 1], [-1, -1]]
Bc = [[0], [1]]
dt = 0.5
N = 10
Q = [[1, 0], [0, 1]]
R = [[1]]
P = Q
u_lb = [[-10]]
u_ub = [[10]]
```

### 3.1.3 Creating a Python MPC object

The following function creates the MPC object for the system module. This function belongs to the ltidt module.
ltidt.setup_mpc_problem(system_name, verbose=False)

Create an MPC object from a given system module.

Parameters

• system_name (a string or a Python module) – a string with the name of a Python module containing the system description, e.g. for a system.py, system_name="system", or system_name="system.py". It can also be the name of a MATLAB mat file. In that case, the file name must end with the extension .mat, e.g. for a system.mat the parameter is given as system_name="system.mat"

Alternatively, a Python module can be given as input. For example, if you import system, then system_name=system.

• verbose (bool) – if True, print on-screen information about the problem.

Returns an instance of the appropriate MPC class according to the given system module.

For example, assuming system1.py and system2.mat both contain exactly the same MPC setup, we can write:

```python
import muaompc
mpcx = muaompc.ltidt.setup_mpc_problem("system1") # short way
mpcy = muaompc.ltidt.setup_mpc_problem("system1.py", verbose=True)
mpcz = muaompc.ltidt.setup_mpc_problem("system2.mat") # MATLAB way
```

The three objects, mpcx, mpcy, and mpcz contain the same information.

Additionally, you can quickly create setups that differ only slightly between them:

```python
from muaompc.ltidt import setup_mpc_problem
import system

system.N = 10 # set horizon length to 10 steps
mpcx = setup_mpc_problem(system) # system is a module, not a string
system.N = 5 # same setup but with a shorter horizon
mpcy = setup_mpc_problem(system) # mpcx is not the same as mpcy
```

### 3.1.4 Generating the C code

The following function is available for all MPC objects, and generates C-code from the MPC object data.

```python
_MPCBase.generate_c_files(numeric='float64', fracbits=None, matlab=False)
```

Write, in the current path, C code for the current MPC problem.

Parameters

• numeric (string) – indicates the numeric representation of the C variables, valid strings are:
  – "float64": double precision floating point (64 bits)
  – "float32": single precision floating point (32 bits)
  – "accum": fixed-point data type of the C extension to support embedded processors (signed 32 bits). Your compiler must support the extension.
  – "fip": (EXPERIMENTAL) fixed-point (signed 32 bits).

• fracbits (integer or None) – an integer greater than 0 that indicates the number of fractional bits of fixed-point representation. (EXPERIMENTAL) Only required if numeric is “fip”.

• matlab (bool) – if True, generate code for MATLAB/Simulink interfaces.
For example, assuming that `system.py` contains a valid MPC setup, and we want to generate code appropriate for a microcontroller to be programmed via Simulink:

```python
import muaompc
mpc = muaompc.ltitd.setup_mpc_problem("system")
mpc.generate_c_files(numeric="float32", matlab=True)
# you will find the generated C code in the current directory
```

The next section will show you how to use the generated code.

## 3.2 Using MPC controllers

As seen in the tutorials, to make basic use of µAO-MPC there’s no need to know too much about MPC theory. However, to make use of more advanced features, a better understanding of MPC internals is required. This section will start with some basic theory about MPC and continue with a detailed description of the several ways µAO-MPC can help solve MPC problems.

### 3.2.1 Basics of MPC

The **MPC setup** can be equivalently expressed as a parametric quadratic program (QP), a special type of optimization problem. Under certain conditions (which µAO-MPC ensures are always met), the QP is strictly convex. It basically means that the QP can be efficiently solved applying convex optimization theory.

#### Parametric QP

The QP depends on the current system state and on the reference trajectories, if any. We express our parametric QP in the following form:

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} u^T H u + u^T g(x, x_{ref}, u_{ref}) \\
\text{subject to} & \quad \underline{u} \leq u \leq \overline{u} \\
& \quad z(x) \leq \underline{z} \leq \overline{z}(x)
\end{align*}
\]

A MPC controller is based on the repetitive solution of a QP which is parameterized by the current state \(x\), and the current state and input reference trajectories \(x_{ref}, u_{ref}\), respectively. In other words, at every sampling time we find an approximate solution to a different QP. We emphasize the fact that the MPC solution \(\tilde{u}\) is only a (rough) approximation of the exact solution \(u^*\). In some applications, even rough approximations may deliver acceptable controller performance. Exploiting this fact is of extreme importance for embedded applications, which have low computational resources.

There is another important property to note. Some optimization algorithms require an initial guess to the solution of the QP. Clearly, a good (bad) guess, i.e. close (far) to the solution, will deliver a good (bad) approximate solution (all other conditions being equal). This property is used for the controller’s advantage. There are basically two strategies on how good initial guesses can be computed. One is called cold-start strategy. This means that the initial guess is always the same for all QP problems. This strategy is mainly used when sudden changes on the state are expected (e.g. high frequency electrical applications). The other strategy is called warm-start. It means that the previous MPC approximate solution is used to compute the initial guess for the current QP. In applications where the state changes slowly with respect to the sampling frequency (e.g. most mechanical systems), two consecutive QPs arising from the MPC scheme have very similar solutions.

A last thing to note is that \(g(\cdot)\) is the only term that depends on the reference trajectories. For the special case of regulation to the origin (where both references are zero) we will simply write \(g(x)\).
3.2.2 Using the Python interface

In this section we describe the two main controller functions available to users of µAO-MPC. For easier explanation, we will first discuss the Python interface. Later we will discuss the equivalent, albeit slightly more complex, C functions. The Python interface can be used for prototyping and simulating MPC controllers. The C functions are the actual controller implementation.

Solving MPC problems

The most straightforward way to solve the MPC optimization problem is using µAO-MPC’s default QP solver.

\[
\text{ctl.solve_problem}(x)
\]

Solve the MPC problem for the given state using the default solver.

**Parameters**  
\[
x \text{(numpy array)} - \text{the current state. It must be of size states.}
\]

This method is an interface to the C code. See its documentation for further details.

This method relies on other fields of the ctl structure:

1. **conf** configuration structure.
2. **x_ref** state reference trajectory
3. **u_ref** input reference trajectory
4. **u_ini** initial guess for the optimization variable
5. **l_ini** initial guess for the Lagrange multipliers
6. **u_opt** approximate solution to the optimization problem
7. **l_opt** approximate optimal Lagrange multiplier
8. **x_trj** state trajectory under the current u_opt

**conf** contains the basic configuration parameters of the optimization algorithm. It consist of the fields:

- **warmstart**: if True, use a warmstart strategy (default: False)
- **in_iter**: number of internal iterations (default: 1)
- **ex_iter**: number of external iterations. If mixed constraints are not present, it should be set to 1. (default: 1)

**x_ref** is an array of shape (hor_states, 1), whereas **u_ref** is an array of shape (hor_inputs, 1). By default, **x_ref** and **u_ref** are zero vectors of appropriate size. In other words, the default case is MPC regulation to the origin.

**u_ini** is an array of shape (hor_inputs, 1). **l_ini** is an array of shape (hor_mxconstrs, 1). **l_ini** is only of interest for problems with mixed constraints. By default, these are also zero. These mainly need to be set by the user in the case of manually coldstarting the MPC algorithm (conf.warmstart = False). If conf.warmstart = True, they are automatically computed based on the previous solution.

**u_opt** is an array of shape (hor_inputs, 1). **l_opt** is an array of shape (hor_mxconstrs, 1). **l_opt** is only of interest for problems with mixed constraints. **x_trj** is an array of shape (hor_states+states, 1).

Usually in an MPC scheme, only the first input vector of the optimal input sequence **u_opt** is of interest, i.e. the first inputs elements of **u_opt**.

For example, assuming **mpc** is an MPC object with only input constraints, and we want all states to be regulated to 2, starting at states all 3:
mpc.ctl.conf.in_iter = 5  # configuration done only once
mpc.ctl.conf.warmstart = True  # use warmstart

mpc.ctl.x_ref = numpy.ones(mpc.ctl.x_ref.shape) * 2

# repeat the following lines for every new state x
x = numpy.ones(mpc.ctl.x_ref.shape) * 3
mpc.ctl.solve_problem(x)
u0 = mpc.ctl.u_opt[:mpc.size.inputs]  # 1st input vector in sequence

Using a different solver

Optionally, the user can use a different QP solver together with µAO-MPC. This can be used for example for prototyping MPC algorithms, or for finding exact solutions using standard QP solvers.

c11.form_qp(x)
Compute the parametric quadratic program data using x as parameter.

Parameters x (numpy array) – the current state. It must be of size states.

This method is an interface to the C code. See its documentation for further details.

This method relies on other fields of the c11 structure:
- 1.x_ref state reference trajectory
- 2.u_ref input reference trajectory
- 3.qpx the structure with the created quadratic program data

x_ref is an array of shape (hor_states, 1), whereas u_ref is an array of shape (hor_inputs, 1).
By default, x_ref and u_ref are zero vectors of appropriate size. In other words, the default case is MPC regulation to the origin.

qpx contains the computed data of the Parametric QP using the given state and references. It consist of the fields:
- HoL Hessian matrix
- gxoL gradient vector for the current state and references
- u_lb lower bound on the optimization variable
- u_ub upper bound on the optimization variable
- E matrix of state dependent constraints
- zx_lb lower bound on the state dependent constraints
- zx_ub upper bound on the state dependent constraints

Refer to the MPC description for a precise definition of these fields.

For example, assuming mpc is an MPC object with only input constraints, and we want all states to be regulated to 2, starting at states all 3:

mpc.ctl.x_ref = numpy.ones(mpc.ctl.x_ref.shape) * 2

# repeat the following lines for every new state x
x = numpy.ones(mpc.ctl.x_ref.shape) * 3
mpc.ctl.form_qp(x)
# use mpc.ctl.qpx together with the QP solver of your preference

An example on how to use form_qp together with the QP solver CVXOPT in Python can be found at examples/ltidt/solver_cvxopt. Additionally, an example on how to use form_qp together with the QP solver qpOASES in C can be found at examples/ltidt/solver_qpoases.
3.2.3 Using the C implementation

The C functions

The functions available to the user are described in the C API in the mpc.h file. The Python and MATLAB interfaces offer exactly the same functionality as the C interface, but with a simplified syntax.

For example, the C function

```c
void mpc_ctl_solve_problem(struct mpc_ctl *ctl, real_t x[])
```

will be called in C as

```c
mpc_ctl_solve_problem(&ctl, x)
```

assuming both arguments exist. In Python it is called as `mpc.ctl.solve_problem(x)`, assuming the MPC Python object was called `mpc`.

Similarly, in MATLAB, the same function is called using `ctl.solve_problem(x)`, assuming the MPC controller was called `ctl`.

In all cases, the approximate solution $u_{opt}$ is found inside the `ctl` structure. For example, in C the first element of the sequence is accessed via `ctl->u_opt[0]`, in Python and MATLAB simply via `ctl.u_opt[0]`.

The full C documentation is available as doxygen documentation.
This chapter explains some of the additional functionality that helps create improved MPC controllers.

### 4.1 A simulation class in Python

This subsection shows how a simulation of the system behavior can be easily done. Any created MPC object contains an instance of the simulation class `MPCSim`, which by default is called `sim`.

#### 4.1.1 One step simulation

The most basic method simply computes the successor state, i.e. the state at the next sampling time for a given input.

```
MPCSim.predict_next_state(xk, uk)
```

Compute the successor state $x_{k1} = Ax_k + Bu_k$

**Parameters**

- `xk` (*numpy array*) – the current system state. It must be of size `states`.
- `uk` (*numpy array*) – the input to apply to the system. It must be of size `inputs`.

**Returns**

the successor state.

**Return type**

`numpy array` of size `states`.

For example, assuming the `mpc` object exist and $x$ represents the current system state, we can do:

```
uk = mpc.ctl.u_opt[:mpc.size.inputs, 0]  # use only the first input
xk1 = mpc.sim.predict_next_state(xk, uk)
```

#### 4.1.2 Multi-step simulation

An additional method considers regulation to a fixed reference.

```
MPCSim.regulate_ref(steps, x_ini)
```

Simulate MPC regulation to a reference point for a number of steps.

**Parameters**

- `steps (int)` – number of discret steps to simulate. The period of each step is the discretization time $dt$.
- `x_ini (numpy array)` – initial state where the simulation starts. It must be of size `states`.
Starting at a point $x_{\text{ini}}$, this function will apply to the discrete time system $(\text{sys}.A_d, \text{sys}.B_d)$ the first input vector of the control sequence $\text{ctl}.u_{\text{opt}}$ at each time instant for the given number of steps. The goal of the controller is to stabilize the system at the point specified by $\text{ctl}.x_{\text{ref}}$ and $\text{ctl}.u_{\text{ref}}$. The simulation results are stored in the structure $\text{sim}.\text{data}$.

For example, assuming $\text{mpc}$ is a valid MPC object, and $\text{mpc}.\text{ctl}$ has been already configured, we can simulate the regulation to the origin starting at all states 1 (one) for $T = 10$ steps, i.e. from time $t_i = 0$, to $t_f = (T - 1) * dt$:

```python
import numpy as np
T = 10
mpc.ctl.x_ref = np.zeros(mpc.size.hor_states)
mpc.ctl.u_ref = np.zeros(mpc.size.hor_inputs)
mpc.sim.regulate_ref(T, np.ones(mpc.size.states))
mpc.sim.data.x # contains the state evolution
mpc.sim.data.u0 # contains the applied inputs
```

After each run of `regulate_ref`, the member `sim.data` will contain the simulation data. It is an instance of the following class:

```python
class muaompc._ltidt.simula._DataPlot (mpc, steps)
    This class contains the simulation data after running regulate_ref.
```

The available members are described below:

- $J = \text{None}$
  The value of the cost function at each sampling time.

- $l = \text{None}$
  The Lagrange multiplier vectors computed by the MPC controller at each sampling time.

- $t = \text{None}$
  The discrete time vector, i.e. the sampling times.

- $u = \text{None}$
  The input sequence computed by the MPC controller at each sampling time.

- $u0 = \text{None}$
  The input vector applied to the system at each sampling time.

- $x = \text{None}$
  The system state vector at each sampling time.

**Example**

An example that plots the simulation results using `matplotlib` is available under `examples/ltidt/sim_matplotlib`.

### 4.2 Tuning

**Warning:** The tuning functionality is in an *experimental* stage. There may be many changes in the near future.

#### 4.2.1 Basics of tuning

There are only two tuning parameters: 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 has to compare MPC controller performance given by the solution of an exact solver (like CVXOPT) against that given by the solution of $\mu$AO-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.

### 4.2.2 The penalty parameter

An optional third tuning value is the penalty parameter $\mu$, which is selected off-line (i.e. specified in the system module). $\mu$AO-MPC will by default automatically compute a good value for $\mu$ 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 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 $\mu$ 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 $\mu$ (not too low, not too high) is crucial. Although in general not recommended, $\mu$AO-MPC allows experienced users to explicitly set a value for $\mu$ in the system module. The selection of the penalty parameter $\mu$ is easily done via the function `find_penalty_parameter` under the `tun` object. For example, using the aircraft system from the tutorial:

```python
import muaompc
mpc = muaompc.ltidt.setup_mpc_problem('sys_aircraft')
mu = mpc.tun.find_penalty_parameter()
```

$\mu$ now contains a value of the penalty parameter that is not too high but not too low. This is just an initial value that may help as starting point for further fine tuning. A function that may come in handy while fine tuning $\mu$ is `calc_int_cn`. It calculates the condition number of the internal problem for a list of penalty parameters. Continuing with the example:

```python
mpc.tun.calc_int_cn([0.5 * mu, 2 * mu])
```

will return the condition number for each penalty parameter on the list. This may help to check that the selected $\mu$ does not make the internal problem ill-conditioned. Next section presents a procedure that may help reduce the condition number of the Hessian of the QP, and therefore, also of the internal problem, allowing the use of higher values of $\mu$ and making the solver faster.

### 4.2.3 Conditioning the Hessian

The underlying optimization algorithm used by $\mu$AO-MPC, greatly benefits from a low condition number (say, below 100). To achieve this, we provide the function `reduce_H_cn`, which basically solves a nonlinear program (NLP).

```python
MPCTuning.reduce_H_cn(xref, uref, stw, inw, cn_ub=100, stw0=None, inw0=None)
```

Find weighting matrices that decrease the Hessian's condition number.

**Parameters**

- `xref` (*numpy array*) – the trajectory the states should follow. `xref` has shape (states, points).
- `uref` (*numpy array*) – the sequence of inputs for the given state trajectory `xref.uref` has shape (inputs, points).
- `stw` (*numpy array*) – the relative weights for the states given as a vector of shape (states,). This vector correspond to the diagonal of the state weighting matrix $Q$, i.e. $Q = \text{diag}(stw)$.
• **inw** (*numpy array*) – the relative weights for the inputs given as a vector of shape (inputs,). This vector correspond to the diagonal of the input weighting matrix $R$, i.e. $R = \text{diag}(\text{inw})$.

• **cn_ub** (*scalar*) – upper bound for the condition number of the Hessian. It must be greater than 1.

• **stw0** (*None or numpy array*) – initial guess of the solution for the state weights. If None, it will be computed as an identity matrix of appropriate shape.

• **inw0** (*None or numpy array*) – initial guess of the solution for the inputs weights of shape inputs. If None, it will be computed as an identity matrix of appropriate shape.

**Returns**

a dictionary with the diagonals of the newly tuned weights. The keys are as follows:

- “stw” is the diagonal of the weighting matrix for the states.
- “inw” is the diagonal of the weighting matrix for the inputs.

Given a reference state and input trajectory $(x_{\text{ref}}, u_{\text{ref}})$, the difference of a simulated trajectory for the unconstrained system (with initial condition the first point of the given trajectory $(x_{\text{ref}}, u_{\text{ref}})$) should be minimized with respect to the diagonal weighing matrices of the QP. This problem is posed as a nonlinear program (NLP).

This NLP takes a reference trajectory as input (for the states and the input sequences), the reference weighting matrices that set the desired controller performance, and an upper bound on the condition number of the QP’s Hessian. The NLP returns weighting matrices that give a similar controller performance and make $\kappa_H$ (the condition number of the Hessian) lower than the specified upper bound. As this NLP is in general nonconvex, there might not be a unique solution, and finding a solution or not might depend on the initial guesses for the weighting matrices. For details see [WAD11].

The procedure to find suitable weighting matrices that reduce $\kappa_H$ is as follows:

1. Manually tune your MPC controller, as you would normally do. At the moment, only diagonal matrices are accepted. These matrices will be used as base for the NLP.

2. Generate a trajectory for the states and the corresponding input sequences that is representative of the MPC application.

3. Optionally, select an upper bound for $\kappa_H$.

4. Optionally, select an initial guess for the weighting matrices.

5. Repeat for different parameters if the optimization was not successful, or if you are not satisfied with the results.

Take for example the system in a *more complex MPC problem*. We consider a change in the altitude to represent the typical behaviour of the control system. A very brief example:

```python
from numpy import eye, diag
import muaompc

mpc = muaompc.ltidt.setup_mpc_problem('sys_aircraft')
# xref, uref have been already generated via simulation
# the reference weighting matrices have been manually tuned as identity
Qref = diag(eye(5))
Rref = diag(eye(1))

r = mpc.tun.reduce_H_cn(xref, uref, Qref, Rref)
Qtun = r['stw']
Rtun = r['inw']
```

**Example**

A complete example of how to use this function can be found in the tutorial/ltidt/tun_h_cn.
4.3 Automatic selection of stabilizing matrices

4.3.1 The auto keyword

The auto keyword indicates that the terminal state weighting matrix $P$ is to be selected automatically, such that the terminal cost is a control Lyapunov function for the model predictive control setup. We consider the following special cases:

- the open-loop system is stable and there are only constraints on the inputs. The solution of a matrix Lyapunov equation is returned. In this case the regulation point of the closed-loop system is globally exponentially stable.
- the pair $(A_d, B_d)$ is stabilizable and there are constraints on the inputs and states. The solution of a discrete algebraic Riccati equation is returned. The regulation point is asymptotically (exponentially) stable with region of attraction $X_N (X_S)$.

The following conditions are expected on the MPC setup:

- $Q$ is symmetric positive definite.
- $R$ is symmetric positive definite.
- The pair $(A_d, B_d)$ is stabilizable.
- If $Q$ is positive semi-definite, the pair $(A_d, Q^{1/2})$ must be detectable. This is not checked at the moment.

The following conditions are checked:

1. Is a problem with only input constraints?
2. Has $A_d$ all its eigenvalues strictly inside the unit circle?

If conditions 1. and 2. are both true, a discrete Lyapunov equation is solved. If any of the conditions 1. and 2. is false, then a discrete algebraic Riccati equation is solved.

In this context, $X_N$ is defined as the set of states for which the MPC problem is feasible. The set $X_S$ is any sublevel set of the MPC optimal value [RM09].

4.4 MATLAB/Simulink interface

The MATLAB and Simulink interfaces make it possible to use and control a precompiled solver from MATLAB/Simulink.

4.4.1 Dependencies

To make use of this functionality, the following are required:

- MATLAB (tested using version 7.8.0 (R2009a))
- a C compiler (tested using GCC version 4.4 (gcc4.4))

4.4.2 Creating the interfaces

For basic usage, the procedure consist on writing the system module, generating and compiling the code, and finally using the interface.
Create the system module

The system module can also be created in MATLAB. Simply define the required matrices (e.g. \( A_c, B_c, Q, u_{lb}, e_{lb} \)) and save them in a mat file. In this example, we will assume the sys_motor.mat system module was created, based on the MPC setup described on A basic MPC problem.

Note: to read mat files, SciPy is required.

Generate the code

At first you have to generate the code that represents the MATLAB interface. The only thing you have to do is add the option `matlab=True` to the generator call. The following Python commands will generate the MATLAB/Simulink interfaces for the sys_motor.mat example:

```python
import muaompc
mpc = muaompc.ltidt.setup_mpc_problem('sys_motor.mat')
mpc.generate_c_files(matlab=True)
```

The generated `cmpc` directory now contains two new directories called `matlab` and `simulink` besides the usual C files. It contains all the code of the MATLAB and Simulink interfaces. A detailed description follows.

4.4.3 Using the MATLAB interface

Compiling the interface

Once the code the interfaces have been created, the next step is to compile the MATLAB interface. Start MATLAB, set the generated `cmpc/matlab` directory as the working directory and call `make`, which will execute the `make.m` script. For example, assuming you are in the directory where you generated the data, inside MATLAB type:

```plaintext
>> cd cmpc/matlab
>> make
```

After this the compiled code is put inside the `@mpc_ctl` directory.

Adding the interface to MATLAB’s path

The last step is to add the `matlab` directory to the PATH environment in MATLAB.

Using the interface

Now you can use the interface which is encapsulated in a class called `mpc_ctl`, which represents the MPC controller. Simply declare an instance of that class, which we usually call `ctl (controller)`. For example:

```plaintext
ctl = mpc_ctl; % create an instance of the class
ctl.conf.in_iter = 10; % configure the algorithm
x = [0.1; -0.5]; % current state
ctl.form_qp(x) % form the QP for the current state
qpx = ctl.qpx; % qpx contains the QP in standard form
u = quadprog(qpx.HoL, qpx.gxoL, [], [], [], qpx.u_lb, qpx.u_ub);
ctl.solve_problem(x); % solve the MPC problem for current state
ctl.u_opt % display the computed input sequence
norm(u - ctl.u_opt) % ctl.u_opt is the MPC approximation of u
```

Example

The complete example can be found under the `examples/ltidt/matlab_interface` folder.

4.4. MATLAB/Simulink interface
4.4.4 Using the Simulink interface

It is possible to use generated code in Simulink to build models for simulation and real time execution on targets which are supported with MATLAB Coder, including compilation for embedded targets.

The Simulink feature that allows to do it is called S-Functions. S-functions (system-functions) provide a powerful mechanism for extending the capabilities of Simulink. They allow to execute C code as a Simulink block.

Compiling the interface

As explained above, to generate preconfigured Simulink model you need to pass `matlab=True` parameter to the generator call. The model is placed in `cmpc/simulink` folder. To run the model follow the next steps:

1. In MATLAB change the current folder to `cmpc/simulink`.
2. Open `cmpc/simulink/simulinkmpc.mdl` in Simulink.
3. Double click the S-Function Builder block to open the block properties window.
4. Click Build to build the S-Function.

Now the model is ready to be extended and used in your design.

Example

The example system `simulink_example.mdl` is located in the folder `examples/ltidt/matlab_interface`. It shows a sample use of a MPC controller. To run the example follow the next steps:

1. Set the current MATLAB folder to `examples/ltidt/matlab_interface`.
2. Type `open_simulink_example` in the MATLAB console.
3. Press the Build button in the block properties window which will appear on the screen.
4. Run the simulation.

The example includes the functionality of solving the control problem and prediction of the next state of the plant. The model can operate using closed-loop and open-loop mode. Also the model shows how to check the execution speed on the device.
REFERENCES


## Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DataPlot</td>
<td>(class in muaompc._ltidt.simula), 22</td>
</tr>
<tr>
<td>F</td>
<td>form_qp() (muaompc.cmpc.ctl method), 19</td>
</tr>
<tr>
<td>G</td>
<td>generate_c_files() (muaompc.ltidt._MPCBase method), 16</td>
</tr>
<tr>
<td>J</td>
<td>J (muaompc._ltidt.simula._DataPlot attribute), 22</td>
</tr>
<tr>
<td>L</td>
<td>l (muaompc._ltidt.simula._DataPlot attribute), 22</td>
</tr>
<tr>
<td>M</td>
<td>muaompc (module), 2, 15</td>
</tr>
<tr>
<td></td>
<td>muaompc._ltidt.simula (module), 21</td>
</tr>
<tr>
<td></td>
<td>muaompc._ltidt.stability (module), 25</td>
</tr>
<tr>
<td></td>
<td>muaompc._ltidt.tuning (module), 23</td>
</tr>
<tr>
<td></td>
<td>muaompc._ltidt.tutorial (module), 5</td>
</tr>
<tr>
<td></td>
<td>muaompc._ltidt.tutorial.sys_aircraft (module), 10</td>
</tr>
<tr>
<td></td>
<td>muaompc._ltidt.tutorial.sys_motor (module), 6</td>
</tr>
<tr>
<td></td>
<td>muaompc.cmpc (module), 18</td>
</tr>
<tr>
<td></td>
<td>muaompc.ltidt (module), 2, 16</td>
</tr>
<tr>
<td></td>
<td>muaompc.mpcsetup (module), 2, 14</td>
</tr>
<tr>
<td></td>
<td>muaompc.sysmod (module), 15</td>
</tr>
<tr>
<td>P</td>
<td>predict_next_state() (muaompc._ltidt.simula.MPCSim method), 21</td>
</tr>
<tr>
<td>R</td>
<td>reduce_H_cn() (muaompc._ltidt.tuning.MPCTuning method), 23</td>
</tr>
<tr>
<td></td>
<td>regulate_ref() (muaompc._ltidt.simula.MPCSim method), 21</td>
</tr>
<tr>
<td>S</td>
<td>setup_mpc_problem() (muaompc.ltid method), 15</td>
</tr>
<tr>
<td></td>
<td>solve_problem() (muaompc.cmpc.ctl method), 18</td>
</tr>
<tr>
<td>T</td>
<td>t (muaompc._ltidt.simula._DataPlot attribute), 22</td>
</tr>
<tr>
<td>U</td>
<td>u (muaompc._ltidt.simula._DataPlot attribute), 22</td>
</tr>
<tr>
<td></td>
<td>u0 (muaompc._ltidt.simula._DataPlot attribute), 22</td>
</tr>
<tr>
<td></td>
<td>x (muaompc._ltidt.simula._DataPlot attribute), 22</td>
</tr>
</tbody>
</table>