A simple introduction to PANOC

Willem Melis

January 18, 2018

Abstract

This is a simple introduction to the nmpc codegen library. It provides a description of the basic functionality of the library.

1 Introduction

The main goal of the nmpc-codegen library is to generate non-linear model predictive controllers. The user must provide the nonlinear model of the system and defined the stage and terminal cost. The system equations can either be continuous or discrete. If the system equations are continuous, the user must select a discretization scheme.

The system equations are defined in python Casadi syntax.The Casadi functions are very similar to the Matlab functions. A short summary of the available functions, will be provided at the end of this document.

The nonlinear non-convex optimization problem is solved using the PANOC algorithm. This algorithm is guaranteed to converge to a critical point, but not necessarily an global optimum. There are still some parameters that need manual tuning, more on this in the section on the nmpc-controller optional features.

The nmpc-codegen library will generate the controller in C89 code. As the library is aimed at embedded developers. However it is possible to test the controller directly from python.

1.1 Installation

The nmpc-codegen library requires the casadi library to be installed, the rest of the dependencies are fulfilled if the dependencies of casadi are fulfilled.

Clone the nmpc-codegen repository from github, and add the src_python folder to the Python path.

1.2 Quick Start

The nmpc-codegen library is separated into five parts, Cfunctions containing the constraints, controller containing the obstacles and the controller code, models containing the two different kinds of models. And finally tools containing the two optional features the simulator and the bootstrapper. The five different parts are discussed in the section below, each in its own chapter.

The Python code used in this document can be found in the repo under the name toturial_nmpc_codegen.py .

The final library has the interface display in Listing 1. An initialization and cleanup function and a solve function.

```
int nmpc_init();
int nmpc_cleanup();
int npmc_solve(const real_t* current_state,real_t* optimal_inputs);
```

Listing 1: interface library

2 Model

2.1 Setup a model

The model contains the relationship between the current state and the next state of the system. The user can provide a discrete model of the system that calculates the next state using the current state. The model also contains the constraints on the input.

These constraints need to have a special mathematical property of having the proximal operator analytically defined. The available constraints are displayed in table 2.1, it is possible to add your own constraints more on this later.

Math function	Python function
I[box(x)]	IndicatorBoxFunction(array_with_lower_bounds,array_with_upper_bounds)

The discrete system equations should be in the form $f(x, u)$ where X is the current states and u are the current inputs. This is demonstrated in listing 2.

```
import nmpccodegen.models as models
import nmpccodegen.Cfunctions as cfunctions

step_size = 0.05
horizon = 300
integrator = "RK" # select a Runga-Kutta integrator
```

```
constraint_input = cfunctions.IndicatorBoxFunction([-1, -1], [1, 1])
model = models.Model(system_equations,
        constraint_input, step_size, number_of_states, \
        number_of_inputs, coordinates_indices)
```

Listing 2: Discrete model

In case the system is defined by continuous function, the model can be defined by the class Model_continious as demonstrated in the listing 3. The construct has an additional parameter to identified integration scheme.

```
import nmpccodegen.models as models

step_size = 0.05
horizon = 300
integrator = "RK" # select a Runga-Kutta  integrator

constraint_input = indbox.IndicatorBoxFunction([-1, -1], [1, 1])
model = models.Model_continious(system_equations,
        constraint_input, step_size, number_of_states, \
        number_of_inputs, coordinates_indices, integrator)
```

Listing 3: Continue model

2.2 Defining your own constraints

2.2.1 Basics

It is possible to define your own constraint functions. Listing 4 illustrates how Cfunctions should actually look. Listing 5 illustrates how the Python side should look. Every constraint must be a proximal function object. A proximal function of object also contains a function that is the proximal result.

```
void casadi_interface_proxg(const real_t* input,real_t* output){
/* ... add your function in here */
}
real_t casadi_interface_g(const real_t* input){
/* ... add your function in here */
}
```

Listing 4: c code implementation proximal functions

```
class Cfunction:
def __init__(self):
raise NotImplementedError

# save the implementation in c to "location"
```

function	purpose
open()	open source file
close()	close source file
start_for(iterator_name,length,indent)	Start a for loop
close_for(indent)	Close the for loop
write_line(line,indent)	Write one line of code
write_define(name,value,indent)	Define a preprocessor constant
write_comment_line(self,line,indent)	Write one line of comment
set_output(output_index,value,indent)	Set the value of output array

Table 1: Table of the available functions in the Source_file_generator Loss

```
def generate_c_code(self,location):
raise NotImplementedError
```

Listing 5: c function interface

```
class ProximalFunction(Cfunction):
def __init__(self,prox):
self._prox=prox

@property
def prox(self):
return self._prox
```

Listing 6: c proximal function interface

2.2.2 Source file operations library

Obviously the user is free to use its own libraries to generate the C code. However a small set of functions in the form of class is available to the user, which can make things easier.

The class Source_file_generator is available under the Cfunctions sub package. Table contains all the functions available to the user. The constructor of the Source_file_generator class specifies if it is either a function or proximal of a function. It is important to make the distinction between the two, because the prototype of the function is different.

In the following two subsections the indicator box function will be implemented. The mathematical definition of the indicator box function can be found in the appendix. A constraint with an indicator box function allows a certain area of inputs. The proximal function of the me get a box function, simply gives the closest valid input.

2.2.3 Example: Index box function

First an object is constructed with a certain location and a function type. In this case the function type equal to "g" which means that it's an normal function. The proximal function of this function will be of the type "proxg". Before we can write to the source file we must open a stream, at the end this stream must also be closed.

What follows next is the actual implementation of the function. Followed by the closing of the file stream.

```
source_file = Source_file_generator(location,"g")
source_file.open()
source_file.start_for("i","MPC_HORIZON",indent=1)

for dimension in range(0,self._dimension):
        source_file.write_line("if(input["+str(dimension)+"]<"\
                +str(self._lower_limits[dimension])\
                +" || input["+str(dimension)+"]>"\
                +str(self._upper_limits[dimension])+ "){"\
                ,indent=1)
        source_file.write_line("return LARGE;", indent=2)
        source_file.write_line("}",indent=1)

source_file.write_line("input+="+str(self._dimension)+ ";",2)
source_file.close_for(indent=1)

source_file.write_line("return 0;", indent=1)
source_file.close()
```

Listing 7: example generate g function

2.2.4 Example: Proximal index box function

In the previous subsection the indicator box function was implemented. In this subsection the proximal of this function will be implemented. This is why the type of the function is now "proxg" and not "g".

```
source_file = Source_file_generator(location,"proxg")
source_file.open()
source_file.start_for("i","MPC_HORIZON",indent=1)

for dimension in range(0,self._dimension):
        source_file.write_line("if(input["\
                +str(dimension)\
                +"]<"+str(self._lower_limits[dimension])\
                +"){",indent=2)
```

```
        source_file.set_output(dimension,str(self._lower_limits[dimension])\
                ,3)
        source_file.write_line("}else  if(input["  \
                + str(dimension)  \
                + "]>"  \
                + str(self._upper_limits[dimension])  \
                + "){",\
                indent=2)
        source_file.set_output(dimension,  \
                str(self._upper_limits[dimension]),  3)
        source_file.write_line("}else{",  2)
        source_file.set_output(dimension,  "input["  \
                + str(dimension)  \
                + "]",  3)
        source_file.write_line("}",  2)

source_file.write_line("input+="+str(self._dimension)+  ";",2)
source_file.write_line("output+="+str(self._dimension)+  ";",2)
source_file.close_for(indent=1)
source_file.close()
```

Listing 8: example generate proxg function

3 Controller

First the absolute minimum amount of arguments to create a controller will be discussed. In the following section the optional features will be discussed.

3.1 Absolute minimum controller

In order to create an mpc-controller the stage cost must be defined. The different available stage costs are displayed in table 3.1. The left side of the table contains the mathematical function and the right side contains the corresponding python function.

Listing 9 contains a simple example of a minimal controller. In this case the stage costs needs a reference state. The arrays are always numpy arrays. Finally the controller location (more on this in the chapter on bootstrapper) the model and the stage cost are passed on to the control.

The actual code will only be generated when the generate_code() function is called. This allows the user, to define additional options.

Math function	Python function
$\int[x^T Qx + u^T Ru]$	Stage_cost_QR(model, Q, R)
$\int[(x - x_{ref})^T Q(x - x_{ref}) + u^T Ru]$	Stage_cost_QR(model, Q, R, reference_state)

```
import nmpccodegen.controller as controller

Q = np.diag([1., 100., 1.])
R = np.eye(model.number_of_inputs, model.number_of_inputs) * 1.

reference_state = np.array([2, 0.5, 0])
stage_cost = controller.Stage_cost_QR_reference(model, Q, R, reference_state)

trailer_controller = npc.Nmpc_panoc(trailer_controller_location,\
        model, stage_cost)
```

Listing 9: simple controller

3.2 Optional features

The optional features are displayed in table 2, and demonstrated in listing 10.

```
import nmpccodegen.controller as controller

reference_state = np.array([2, 0.5, 0])
stage_cost = controller.Stage_cost_QR_reference(model, \
        Q, R, reference_state)

trailer_controller = npc.Nmpc_panoc(trailer_controller_location,\
        model, stage_cost)
trailer_controller.horizon = horizon
trailer_controller.integrator_casadi = True
trailer_controller.panoc_max_steps = 100
trailer_controller.add_obstacle(obstacle, obstacle_weight)
```

Listing 10: Optional features

attribute	default value	possible values
data_type	”double precision”	[double precision, single precision]
number_of_steps	10	integer
lbgfs_buffer_size	10	integer
panoc_max_steps	10	integer
shooting_mode	”single shot”	[single shot, multiple shot]
integrator_casadi	False	Boolean=[True,False]

Table 2: Features of the controller

4 Simulator

4.1 Using the Simulator class

After the controller is successfully generated, it might be useful to run some simulations in order to get an idea of how well the controller works. That way the user can compare different controllers with each other.

Listing 11 illustrates a simple example, where the controller is simulated using the simulator. It is necessary to called the initialize function before the simulation is executed. And the cleanup function after the simulation has been executed.

The results of the simulations, are safe in a simulation object. A simulation object contains the optimal input, and the time to get this input. The time is expressed by six parameters, hours, minutes, seconds, milliseconds, microseconds and nanoseconds. The accuracy of the convergence time, depends on the accuracy of the internal time of the operating system. If an accuracy lower than a millisecond is required, then the users should take in account the internal scheduler of the opening system.

```
import nmpccodegen.tools as tools

# setup a simulator to test
sim = tools.Simulator(trailer_controller)

# init the controller
sim.simulator_init()

initial_state = np.array([0.01, 0., 0.])
state = initial_state
state_history = np.zeros((number_of_states, number_of_steps))

for i in range(1, number_of_steps):
        result_simulation= sim.simulate_nmpc(state)
        print("Step ["+str(i)+"/"+str(number_of_steps)+ \
                "]: The optimal input is: [" \
                + str(result_simulation.optimal_input[0]) + "," + \
                str(result_simulation.optimal_input[0]) + "]" \
                + " time=" + result_simulation.time_string)

        state = np.asarray(model.get_next_state(state, \
                result_simulation.optimal_input))
        state_history[:, i] = np.reshape(state[:], number_of_states)

# cleanup the controller
```

```
sim.simulator_cleanup()
```

Listing 11: Simulator example

4.2 Calling the interface directly

The Ctypes library is a simple way to call C code from within Python code.If you used the bootstrapper with the property Python_interface_enabled set as true that you must still call Cmake using the command "Cmake .". A library with the extension ".so" or "dll" will be compiled. It isn't this library that we must call the function "simulation_init()", "simulation_cleanup()" and "simulate_nmpc_panoc.

```
class Panoc_time(ctypes.Structure):
        _fields_ = [("hours", ctypes.c_int),\
                ("minutes", ctypes.c_int),\
                ("seconds", ctypes.c_int),\
                ("milli_seconds", ctypes.c_int),\
                ("micro_seconds", ctypes.c_int),\
                ("nano_seconds", ctypes.c_int)]
```

Listing 12: ctype structure return by simulation

```
# Unix-like machines:
nmpc_python_interface = ctypes.CDLL(lib_location)
# Windows machines
nmpc_python_interface = ctypes.windll.LoadLibrary(lib_location)

array_state = ctypes.c_double * length_state

array_optimal_input = ctypes.c_double * model.number_of_inputs
optimal_input = array_optimal_input()

nmpc_python_interface.simulation_init()

nmpc_python_interface.simulate_nmpc_panoc.restype =
        ctypes.POINTER(Panoc_time)
convergence_time = nmpc_python_interface.simulate_nmpc_panoc(\
        array_state(*current_state),\
        optimal_input\
        )

nmpc_python_interface.simulation_cleanup()
```

Listing 13: calling c code directly example

5 Bootstrapper

Every controller exists out of static and a dynamic code, the static code can be generated with the bootstrapper. The dynamic code is generated by the nmpc_controller class. The bootstrap has two parameters, the output location of the controller and the name of the controller.

The optional parameter python_interface_enabled can either be enable or disabled the python interface on the controller. This optional parameter is by default true, and should only be set on false if the user will not use the simulator.

The controller name is used as folder name of the controller. The bootstraper will copy the static code from the library into the new folder. If the folder already exists, it will leave it in place. If the static code is already present, it will print a warning to the screen and replace the files. If the folder structure is already in place, it will leave the folders in place.

```
import nmpccodegen.tools as tools


output_locationcontroller = "./test_controller_builds"
trailer_controller_location = output_locationcontroller + "/" + \
        controller_name + "/"

tools.Bootstrapper.bootstrap(output_locationcontroller,\
        controller_name, python_interface_enabled=True)
```

Listing 14: Optional features

A Function Definitions

A.1 box function

1 dimension:

$$box_{1D}(u) = \begin{matrix} 1 & u \in [-U_b : U_b] \\ 0 & otherwise \end{matrix} \tag{1}$$

N dimensions:

$$box(u) = min\left[\sum_{k=1}^{N} box_{1D}(u)\right] \tag{2}$$

A.2 Indicator Box function

$$I[box] = \begin{matrix} 0 & u \in [-U_b : U_b] \\ \text{inf} & otherwise \end{matrix} \tag{3}$$

$$prox[I[box]] = \begin{matrix} u & u \in [-U_b : U_b] \\ -U_b & u \in [-\text{inf} : -U_b] \\ -U_b & u \in [U_b : \text{inf}] \end{matrix} \tag{4}$$