A simple introduction to PANOC

Willem Melis

January 13, 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 must be 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 will be 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.

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 must also contain 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 1.

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

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

Listing 1: 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 2. The construct has an additional parameter to identified integration scheme.

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

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

Listing 2: Continue model

2.2 Defining your own constraints

Some users will want to define their own constraint functions. Listing 3 illustrates how to C functions should actually look. Listing 4 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 3: c code implementation proximal functions

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

# save the implementation in c to "location"
def generate_c_code(self,location):
raise NotImplementedError
```

Listing 4: c function interface

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

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

Listing 5: c proximal function interface

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 6 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
$\int [x^T Q x + u^T R u]$	Stage_cost_QR(model, Q, R)
$\int [(x - x_{ref})^T Q (x - x_{ref}) + u^T R u]$	Stage_cost_QR(model, Q, R, reference_state)

```
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 = stage_costs.Stage_cost_QR_reference(model, Q, R, reference_state)

trailer_controller = npc.Nmpc_panoc(trailer_controller_location, model, stage_co
```

Listing 6: simple controller

3.2 Optional features

The optional features are displayed in table 1, and demonstrated in listing 7.

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

Listing 7: Optional features

Table 1: Features of the controller

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]

4 Simulator

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 8 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.

```
# setup a simulator to test
sim = simulator.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 8: Simulator 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 three parameters, the location of

the library, 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.