Real Time Model Predictive Control in JModelica.org

Sebastian Ekström

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
ISRN LUTFD2/TFRT--5986--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2015

Abstract

In this thesis a framework for real-time model predictive control has been developed for JModelica.org, which is an open-source platform for simulation and analysis of dynamical systems. Model predictive control (MPC) is an advanced optimization-based control method that uses a model of the process being controlled to optimize control. The framework was tested on three different processes, real and simulated, and its performance was compared with that of an linear-quadratic regulator (LQR), which is a simpler type of controller that uses multiplication with a pre-calculated matrix to calculate the control signal from the state vector. The MPC controller was found to perform as well as or better than the LQR controller in all cases, with the main improvements being seen in the MPC controller's ability to handle process constraints or when far from the LQR controller's linearization point; however, the LQR controller was much faster in calculating the control signal. This also served as a first test of using JModelica.org to perform MPC on real processes, and although it performed well on the two it was tested on, further work will be needed if the MPC framework should be able to handle processes that are much faster or more complex.

Acknowledgements

Big thanks go out to Toivo Henningsson and Anton Cervin for all their help and guidance throughout this project. Additionally, I would like to thank Magdalena Axelsson for helping me getting the MPC class to work (not to mention developing it in the first place), and Anders Blomdell for all his work in getting JModelica to get along with the computers at the Department of Control.

Contents

1

Introduction

1.1 Aim of the thesis

Model predictive control is an advanced control method that uses a model of the process that should be controlled to optimize control. It has several advantages over less complex control methods: most notably, it can easily handle constraints in the process. However, it is also computationally demanding, and therefore it is important to implement it as efficiently as possible so as to allow it to work on as many processes as possible.

In this thesis, a framework for running real-time model predictive control has been developed using JModelica.org, which is an open-source platform for simulation and optimization of dynamic systems. JModelica.org already has an existing class for MPC; however, it has been developed for usage with simulated processes and does not have support for real-time control. The framework was built around the existing MPC class, handling communication with the process and making sure that the timing of the control is correct. Additionally, the framework should be easily extendable to work with any type of process, both real and simulated.

To test the real-time MPC framework, it was used to control three different processes: two real and one simulated. A linear-quadratic regulator was also used to control the same processes, so that the results from the two controllers could be compared for correctness and performance.

Lastly, the work in this thesis also served as a first test of using JModelica.org to perform MPC on real processes. An important goal was to evaluate how well it performs in this context, as well as what needs to be improved.

1.2 Outline

Chapter 2 describes the basics of model predictive control and the linear-quadratic regulator, as well as briefly describing JModelica.org. Chapter 3 describes the implementation of the real-time MPC framework, showing the different methods of the base class as well as the class for running real-time MPC on simulated processes,

and provides an example of how to use it. Chapter 4 contains results obtained by applying the MPC controller to three different processes, comparisons with LQR control, and notes on performance. Finally, Chapter 5 provides some concluding remarks and describes additional improvements that could be made to the framework.

2

Background

2.1 Model predictive control

Model predictive control (MPC) is a collection of methods for optimal control, characterized by using a model to predict the behaviour of the process being controlled and optimize control over a moving time horizon. Its usage was first described in a 1978 paper where "Model Predictive Heuristic Control" was used to control several industrial processes using impulse response models [Richalet et al., 1978], and it has been used for industrial applications ever since. In particular, it has been very well recieved in the chemical process industry, due to its ability to easily handle constraints in the processes it is used for while still keeping the operating point close to the constraints to maximize effectiveness. [García et al., 1989] Due to being rather computationally demanding compared to other control methods, it has traditionally mostly been used for controlling linear processes with low required sample rate. However, with computer performance improving with time, along with more efficient types of MPC controllers such as advanced-step model predictive control [Zavala and Biegler, 2009] being developed, applications of MPC to more complex processes have started to see use.

All methods of model predictive control use a model of the process to be controlled in order to optimize control over a set time horizon from the current time, given an objective function to minimize. In its most general formulation, given a continuous system of differential algebraic equations

$$F(\dot{x}(t), x(t), u(t)) = 0 \tag{2.1}$$

$$x(0) = x_0 \tag{2.2}$$

where x is the state vector, u is the control signal, h is the time step, F is a arbitrary function, and x_0 is the initial state of the system, in addition to an objective function

$$J = \int_t^{t+T} G(x(t), u(t))\, dt \tag{2.3}$$

where T is a given time horizon for the optimization and G is another arbitrary function (often the square of the deviation of the state vector from a reference vector),

and a set of constraints

$$H_k(x(t), u(t)) \geq 0, \quad k \in [1, n] \tag{2.4}$$

the controller for each time step performs the following three tasks in order:

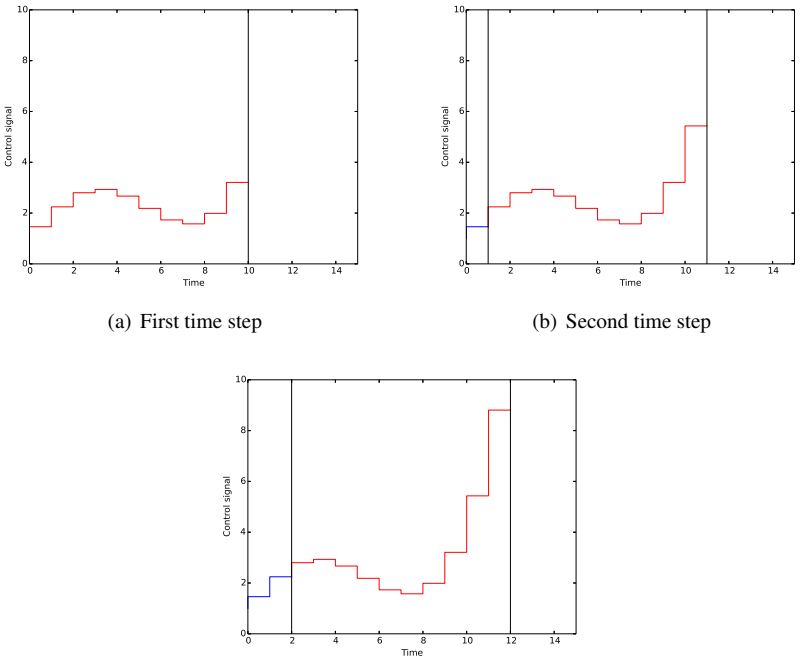
1. Calculate the optimal control sequence up to the optimization horizon, given the current state values.
2. Extract the first step of the optimal control sequence and apply it to the process.
3. Move the optimization horizon one step forward.

Figure 2.1 shows an example of how this works: 2.1 (a) shows the optimal control sequence calculated in the first time step, then (b) and (c) shows how in each time step the control signal for the first time step of the optimal control sequence from the last time step is applied, the optimization horizon is moved forward one step, and a new optimal control sequence is calculated over the new horizon. This allows the controller to optimize the control for the current time while still taking the future into account. The moving horizon gives MPC the alternate name receding horizon predictive control.

All methods of model predictive control follow this same basic three-step strategy. The main difference between them lies in what kind of model is used to represent the process; one of the most popular in the industry is the impulse response model, while the research community often uses the state space and transfer function formulations. [Camacho and Bordons, 2007, ch. 1.1] Other differences between methods include the type of objective function and optimization solver used.

Model predictive control has a number of advantages over other methods of control. The main advantage is that solving the control problem as a series of optimization problems means that it is easy to take constraints on the process into account, since they can just be passed on as constraints to the optimization solver. Additionally, optimizing control over a time interval rather than just the current time step means that an MPC controller can take future events, such as changes in the reference values or in system parameters, into account before they happen, which stands in contrast to more simple control methods like PID only using past events when calculating their control signals. MPC also has no trouble handling non-minimum phase, unstable, or multivariable systems, and is further made attractive by being a rather intuitive method of control, not requiring much knowledge of control theory to understand the basic ideas behind it.

MPC is not without its drawbacks, however. Having to solve an optimization problem each time step in order to calculate the control signal means that it is a computationally intensive method, which limits what processes it can be used on: if the time taken between acquiring the state values from the process and sending the control signal to the process (i.e. the time used by the optimization solver) is



(a) First time step

(b) Second time step

(c) Third time step

Figure 2.1 An example of model predictive control. The blue line in each figure is the optimal control sequence calculated in that time step, the red line is the control signal applied to the process, and the vertical black lines denote the beginning and end of the current optimization horizon.

too long, then the values of the state vector may have changed significantly from the ones used to calculate the control signal, resulting in worse control. Another drawback is that it requires a good model of the process to exist; the less accurate the model, the worse the control will inevitably be. This is not much of a problem for simple processses where the process equations can be derived directly, but can prove troublesome for more complex processes. And while MPC has been very successful in practice, analyzing its stability and robustness has proven very difficult in many cases, and the theoretical results that have been obtained only apply to a small subset of MPC problems. [Camacho and Bordons, 2007, ch. 1.3]

2.2 Linear-quadratic regulator

The linear-quadratic regulator (LQR) is a control method that is based on using a quadratic cost function to optimize control for a linear system. For a discrete time system

$$x(k+1) = \Phi x(k) + \Gamma u(k) \tag{2.5}$$

where x is the state vector and u is the control signal, given a quadratic cost function

$$J = \sum_{k=0}^{\infty} \left(x(k)^T Q_1 x(k) + u(k)^T Q_2 u(k) + 2x(k)^T Q_{12} u(k) \right) \tag{2.6}$$

it can be proven [Åström and Wittenmark, 1997] that the cost is minimized by using the control signal $u_k = -Lx_k$, where

$$L = (Q_2 + \Gamma^T S \Gamma)^{-1} (\Gamma^T S \Phi + Q_{12}^T) \tag{2.7}$$

and S is a symmetric positive semidefinite matrix that can be obtained by solving the discrete-time algebraic Ricatti equation

$$S = \Phi^T S \Phi + Q_1 - (\Phi S \Gamma + Q_{12})(\Gamma^T S \Gamma + Q_2)^{-1} (\Gamma^T S \Phi + Q_{12}^T) \tag{2.8}$$

The controller is designed for a linear process, but it can still be used on some non-linear processes as long as they are sufficiently close to linear around the working point. In this case, a linearization is needed to acquire the Φ and Γ matrices.

Due to calculating the control signal only requiring a single matrix multiplication (since the L matrix can be pre-calculated), LQR is a very fast control method. However, it can not be made to take constraints into account; the closest thing to doing so would be clamping the control signal to a certain bound, but even then the controller itself will not be able to take this into account. This can cause problems when controlling real processes, since they usually have a given range for their inputs; however, it can be worked around by changing the Q matrices to increase the penalty on the control signal. There is no simple way for an LQR controller to handle state constraints, though, a problem which has been explored in [Balandat et al., 2012].

The reason that LQR in particular was chosen for providing a comparison is because of its similarities to MPC. It, too, uses a model of the process to optimize control; although while MPC does so for any process using any horizon length, LQR only works on a linear (or linearized) process with an infinite prediction horizon.

2.3 JModelica.org

JModelica.org is an open-source platform for simulation, optimization and analysis of dynamical systems. A recent master's thesis project [Axelsson, 2015] has developed a class for working with nonlinear model predictive control in JModelica.org;

however, this class is built to work with simulating processes through FMUs (Functional Mock-Up Units), and is not directly applicable to real processes. The class uses the non-linear solver IPOPT for solving the optimization problems. [Wächter and Biegler, 2006]

JModelica.org uses Modelica to describe process models and Optimica to describe optimization problems. Modelica is a modeling language designed to describe the behaviour of dynamic systems through their governing equations, and Optimica is an extension to Modelica for using Modelica models to specify optimization problems for dynamic systems. [Åkesson et al., 2010] For usage examples of Modelica and Optimica, see Chapter 3.6.1.

2.3.1 Code generation

JModelica.org uses the `CasADi` symbolic framework for handling optimization. The optimization solvers use `CasADi Function` objects to describe the mathematical functions needed for the optimization problem: the function to be optimized as well as some of its derivatives. These `Function` objects have a method that allows them to generate C code for the functions they represent, which can then be compile and used to replace them. [Andersson et al., 2015] This will be used in this thesis to investigate whether or not using pre-compiled code in this way has any noticable effect on performance.

3

Implementation

The real-time MPC framework implemented in this thesis consists of a base class `RealTimeMPCBase` that needs to be extended by the user. This class is described in Chapter 3.2. For the sake of convenience, a version of this class extended to work with simulated processes was also created: this `MPCSimBase` class is described in Chapter 3.3. Finally, two classes similar to the first two ones, but for LQR control rather than MPC, were created in order to compare the two methods of control: `RealTimeLQRBase` and `LQRSimBase`. A brief overview of these can be found in Chapter 3.5.

Initially, the only requirements for the framework were being easily usable to to run MPC on multiple different real and simulated processes, as well as being able to use pre-compiled code in order to see how much it would improve performance. During testing, one of the real processes was found to have stationary errors, which prompted the addition of a function to enable integral action for the solver in order to attempt to get rid of the error. Additionally, in order to be able to change the control point during an experiment so that more complex tests could be run, a data structure containing model parameters and the times they should be applied had to be created. Since the rest of the JModelica.org interface is coded in Python, that is also the language used for the framework.

3.1 Theory

3.1.1 Integral action

Integral action is a method for counteracting stationary errors. By calculating an estimate of what the state vector ought to become in the next time step for the given control signal and then comparing the estimate with the actual value, an estimate of the error in the process input can be obtained. After running it through a high pass filter, this error estimate can then be subtracted from the calculated control signal before it is sent to the process to counteract the error.

The way the integral action is implemented in the real-time MPC framework is that each time step, after getting the state values from the process, they are

compared with values predicted from the state and control signal values from the last time step, and the difference is used to estimate the input error using a linear approximation of the process dynamics provided by the matrix M provided to `enable_integral_action`.

The process being controlled can be seen as discrete, since it is only sampled and the control signal only changes at set time points. Thus, a linear approximation of it can be given as

$$x_{k+1} = Ax_k + Bu_k \tag{3.1}$$

where x and u are the state and input vectors respectively. Adding a stationary input error u_e and using an estimate of the error $\hat{u}_{e,k}$ to attempt to cancel it out then gives us the actual next state vector

$$x_{k+1} = Ax_k + B(u_k + u_e - \hat{u}_{e,k}) \tag{3.2}$$

as well as the predicted one

$$\hat{x}_{k+1} = Ax_k + B(u_k) \tag{3.3}$$

The state error is then the difference between the two state vectors:

$$e_{k+1} = x_{k+1} - \hat{x}_{k+1} = B(u_e - \hat{u}_{e,k}) \tag{3.4}$$

This can be rewritten to solve for the error:

$$B^T e_{k+1} = B^T B(\hat{u}_e - \hat{u}_{e,k}) \tag{3.5}$$

$$\Rightarrow \quad (B^T B)^{-1} B^T e_{k+1} = \hat{u}_e - \hat{u}_{e,k} \tag{3.6}$$

$$\Rightarrow \quad \hat{u}_e = (B^T B)^{-1} B^T e_{k+1} + \hat{u}_{e,k} = M e_{k+1} + \hat{u}_{e,k} \tag{3.7}$$

where $M = (B^T B)^{-1} B^T$ is the matrix that should be given as an argument to `enable_integral_action`. Note that this is only possible if $(B^T B)^{-1}$ is invertible, which requires the process has more states than inputs. Finally this calculated input error is used to update the input error estimate using a first order high pass filter:

$$\hat{u}_{e,k+1} = \mu \hat{u}_{e,k} + (1 - \mu)\hat{u}_e \tag{3.8}$$

3.2 Base real-time MPC class

In this section the various methods of the `RealTimeMPCBase` class are described. This base class contains an MPC solver and the code needed for the control loop, as well as some additional functions for enabling use of pre-generated code and integral action, and for plotting and saving result data from a successful run. The only functions needed to be provided by the user are the ones that are specific to

the process that should be controlled: that is sending a control signal to the process, getting the values of the states from the process, and (if necessary) estimating the states of the process that cannot be directly observed.

Once a process-specific class has been created by extending the base class, it is used by first creating an instance of the extended class through the constructor, enabling code generation and/or integral action if desired, and then calling the `run` method. Once the experiment has been run, the results can be plotted, saved to disk, or extracted from the solver for further processing. See Chapter 3.6 for a full usage example.

3.2.1 Initialization

In order to use the base class, the following arguments need to be passed to the constructor. The first nine are mandatory, and the last four have default values and are as such optional. The class is abstract and cannot be instantiated directly, but classes that extend it can still make use of its constructor via `super`.

`file_path`	File path of the `.mop` file that contains the optimization problem to be solved.
`opt_name`	Name of the Optimica optimization class to be solved, in the file specified by `file_path`.
`dt`	Time between samples.
`t_hor`	Horizon time to use for the MPC solver. Must be a multiple of `dt`.
`t_final`	Time to run the MPC control experiment.
`start_values`	A dictionary containing starting values for the states.
`par_values`	A dictionary containing values for any model parameters that should be different from their values in the .mop file.
`output_names`	A list of the names of the output states of the process.
`input_names`	A list of the names of the inputs to the process.
`par_changes`	A `ParameterChanges` object (see Chapter 3.4) determining the reference points to follow. Default value: an empty `ParameterChanges` object.
`mpc_options`	A dictionary containing the options to pass on to the MPC solver. See the documentation of the `MPC` class for more details. Default value: an empty dictionary.
`constr_viol_costs`	A dictionary determining the cost the MPC solver should use for violating constraints. See the documentation of the `MPC` class for more details. Default value: an empty dictionary.
`noise`	A floating point number determining the standard deviation of the noise to add to the input signals, using a Gaussian distribution. Default value: 0.

The constructor for the class is not particularly complex. All it does is set up the MPC solver to be used, initialize the data structures used for saving results and statistics, and save the arguments needed for running the control loop later.

3.2.2 Methods

`enable_codegen`

```
enable_codegen(self, name=None):
"""
Enables use of generated C code for the MPC solver.

Generates and compiles code for the NLP, gradient of f, Jacobian of g,
and Hessian of the Lagrangian Function objects, and then replaces
the solver object in the solver's collocator with a new one that makes
use of the compiled functions as ExternalFunction objects.

Parameters::

    name --
        A string that if it is not None, loads existing files
        nlp_[name].so, grad_f_[name].so, jac_g_[name].so and
        hess_lag_[name].so as ExternalFunction objects to be used
        by the solver rather than generating new code.
        Default: None
"""
```

Calling this method takes the internal `NlpSolver` object from the MPC solver, extracts all of the relevant attributes (bounds, initial values, options) from it, takes the four `Function` objects described in the docstring from the solver and calls the `generateCode` methods on them, compiles the generated code into `.so` files (using the `-O3` flag to make the code as efficient as possible) and loads them as `ExternalFunction` objects, and then creates a new solver object with the resulting `ExternalFunction` objects and the other attributes. If `name` is given, the function searches for the four .so files with the suffix given by the parameter; if all four files exist, those are used instead and new code is not generated.

enable_integral_action

```
enable_integral_action(self, mu, M, error_names=None, u_e=None):
"""
Enables integral action for the inputs.

If integral action is enabled, in each step the input error is
calculated and used to update the estimation of the error. By
default, the input error is calculated as the matrix M times the
difference between the current state vector as predicted by the
solver in the last time step and as measured from the process
in the current time step; however, this can be changed by
overriding the estimate_input_error method.

The formula used to update the estimate is
[new estimate] = mu*[old estimate] + (1-mu)*[current estimate]

Parameters::

    mu --
        Controls the convergence rate of the error estimate.
        See above.

    M --
        Used for calculating the input error from the state error.
        See above.

    error_names --
        A list containing the names of the model variables for
        the input errors. If set to None, it is assumed be the
        same as the list of input variables with the prefix '_e'
        appended to each one.
        Default: None

    u_e --
        A list containing a set of values to be applied as
        stationary errors to the input signals. Used for
        simulating a stationary error where there otherwise wouldn't
        be one. If set to None, no stationary error is applied.
        Default: None.
"""
```

Calling this method enables integral action for the inputs. See Chapter 3.1.1 for a detailed description on how this works.

`run`

```
run(self, save=False):
"""
Run the real time MPC controller defined by the object.

Parameters::

    save --
        Determines whether or not to save data after running. If set
        to False, the same data can still be saved manually by using
        the save_results function.
        Default: False

    Returns::

        The results and statistics from the run.
"""
```

This is the function that runs the main control loop for the time specified when creating the object. Each time step it goes through these steps in order:

1. Update solver parameter values if necessary (see Chapter 3.4)
2. Update the MPC solver with the current state values
3. Have the MPC solver calculate the next control signal
4. Apply noise and simulated stationary error to the control signal (if enabled)
5. Send the control signal to the process
6. Wait for the next sample time
7. Get measurements from the process
8. Calculate state values from measurements (if not all states are observable)
9. Update the estimate of the stationary error (if integral action is enabled)
10. Save the results from the last time step

The reason that the current state values are retrieved at the end of the control loop rather than the beginning is that for simulated processes, the current values are gotten as the result of the simulation of the last time step, which cannot be done if no time steps have passed yet. This means that the initial state needs to be provided in the constructor, which is not an issue when starting in a stationary point, but may prove problematic otherwise. However, this can easily be worked around when extending the class to work with a real process, by making the `run` method start by reading the current state from the process.

get_results

```
get_results(self):
"""
Return the results from the run.

Returns::

    A dictionary where the keys are variable names and the values are
    lists of values.
"""
```

This method returns a dictionary containing the results from running the control loop.

print_stats

```
print_stats(self):
"""
Print statistics from the run.

The times printed are the sums of the corresponding
statistics from the NLP solver over all time steps.
"""
```

The statistics printed by this method are gotten from the `getStats` function of the MPC solver's internal `NlpSolver` object, summed over all of the time steps.

save_results

```
save_results(self, filename=None):
"""
Pickle and save data from the run to a file name either passed
as an argument or entered by the user. If an empty string is
entered as the file name, no data is saved.

Parameters::

    filename --
        The file name to save as. If it is not provided, the
        user will be prompted to input it.
        Default: None
"""
```

This method uses Python's builtin `pickle` module to save data from running the control loop. The data saved by the function is a dictionary containing the following:

Key	Value
`results`	The results from the run.
`stats`	The statistics from the run. Same as the ones printed by the `print_stats` method.
`ptime`	Total processor time taken for the run, as provided by Python's `time.clock` function.
`rtime`	Total real time taken for the run, as provided by Python's `time.time` function.
`noise`	The standard deviation of the noise added to the control signal. Same as the constructor parameter of the same name.
`late_times`	A list of how late the sampling was in each time step.
`´wait_times`	A list of long the solver had to wait for the next sample each time step.
`solve_times`	A list of how long it took between sampling and sending the control signal each time step.

plot_results

```
plot_results(self, outputs=None, inputs=None, var_labels={},
                  title="", cols=1):
"""
Plot the results of the run.

Parameters::

    outputs --
        A list containing the names of the output variables to be
        plotted. If set to None, plots all outputs.
        Default: None

    inputs --
        A list containing the names of the input variables to be
        plotted. If set to None, plots all inputs.
        Default: None

    var_labels --
        A dictionary containing variable names as keys and the
        labels to use for their plots as values. If a variable
        name to be plotted isn't present in the dictionary, its
        name is used as the label.
        Default: {}

    title --
        The title of the plot.
        Default: ""

    cols --
        The number of columns to divide the subplots into.
        Default: 1
"""
```

This method uses the `matplotlib` library to plot the results from the run. Each variable defined in the `outputs` and `inputs` arguments is plotted in its own subplot, which are divided into `cols` columns in the final plot.

3.2.3 User-provided methods

The methods in this section are all ones that need to be overridden by the user when extending the class.

`send_control_signal`

```
send_control_signal(self, u_k):
"""
Dummy method for sending a control signal to the process.

In order to use this class, this method needs to be overridden.
The override method should take take the control signal, convert
it to a format understandable by the process if necessary, and
pass it on to the process that shouldbe controlled.

Parameters::

    u_k --
        The control signal. It consists of a pair where the
        first element is a list of input names, and the second
        is a function that takes the time and returns a Numpy
        array of values corresponding to those inputs.
"""
```

As the name suggest, this function should take the control signal calculated by the controller and pass it on to the process. Note the format of the control signal: while it can be passed on to an FMU directly, it will most likely need to be processed in some way before it can used in a real process. The reason for the format is that it is the one used by both the MPC solver and the FMU.

`get_measurements`

```
get_measurements(self):
"""
Dummy method for getting measurements from the process.

In order to use this class, this method needs to be overridden.
The override method should get measurements from the process and
return them in the form of a dictionary, where the keys are
names of output variables prefixed by '_start_' and the values
are their values.
"""
```

This function should communicate with the process being controlled to extract measurements of the observable states, and then package them in a dictionary to be used by the solver.

`estimate_states`

```
estimate_states(self, x_k, x_k_last):
"""
Estimate the full output state dictionary.

This method takes the measurements gotten from the process in this
time step and the full state dictionary from the last step, and uses
them to calculate an estimate of the full state vector for this
time step. In the base class, all it does is return the measurement
vector from the process, but if your process has non-observable
states, you should override this method and use it to calculate
those states.

Parameters::

    x_k --
        The measurement dictionary gotten from the process with
        get_measurements(). The keys are variable names prefixed
        with '_start_', and the values are the values of those
        variables.
    x_k_last --
        The full state dictionary from the last time step. The
        format is the same as for x_k.

    Returns::

        An estimate of the full output state dictionary. The format is
        the same as for the inputs.
"""
```

This function should estimate the values of any non-observable states in the process. Note that as opposed to `send_control_signal` and `get_states` this is not an abstract function; if the process is completely observable, this function does not need to be overridden.

3.3 MPC class for simulated processes

For convenience, a version of the real-time MPC base class set up specifically for usage with simulated processes has also been developed. This `MPCSimBase` class has the `send_control_signal` and `get_states` methods already defined, so the only method that needs to be provided by the user is `estimate_states`, and then only if it is required that any of the states in the simulated process should be treated as non-observable. As opposed to the `RealTimeMPCBase` class, this class does not run in real time, since doing so for a simulated process is unnecessary.

The constructor for this class looks slightly different than the one for the base class; it has additional arguments for the process model (`model_name`), simulation options (`sim_options`), and which states should be considered observable (`obs_var_names`). It takes the following arguments:

`file_path`	File path of the .mop file that contains the optimization problem to be solved.
`opt_name`	Name of the Optimica optimization class to be solved, in the file specified by `file_path`.
`model_name`	Name of the Modelica model class to use for simulating the process, in the file specified by `file_path`.
`dt`	Time between samples.
`t_hor`	Horizon time to use for the MPC solver. Must be a multiple of `dt`.
`t_final`	Time to run the MPC control experiment.
`start_values`	A dictionary containing starting values for the states.
`par_values`	A dictionary containing values for any model parameters that should be different from their values in the .mop file.
`output_names`	A list of the names of the output states of the process.
`input_names`	A list of the names of the inputs to the process.
`obs_var_names`	A list of the names of the states that should be treated as observable. If set to None, all states are treated as observable. Default value: None.
`par_changes`	A `ParameterChanges` object (see Chapter 3.4) determining the reference points to follow. Default value: an empty `ParameterChanges` object.
`mpc_options`	A dictionary containing the options to pass on to the MPC solver. See the documentation of the `MPC` class for more details. Default value: an empty dictionary.
`sim_options`	A dictionary containing the simulation options to pass to the FMU. See the documentation for the FMU class for more information. Default value: an empty dictionary.
`constr_viol_costs`	A dictionary determining the cost the MPC solver should use for violating constraints. See the documentation of the `MPC` class for more details. Default value: an empty dictionary.
`noise`	A floating point number determining the standard deviation of the noise to add to the input signals, using a Gaussian distribution. Default value: 0.

3.4 Handling changing reference points

In order to handle changing reference points at certain points during a run, a class `ParameterChanges` has been created. This class is little more than a wrapper around a dictionary, where the keys are times and the values are dictionaries containing the parameter values to use from that point onwards. The class has the following methods:

Constructor

The constructor takes up to two arguments:

`par_change_dict`	A dictionary containing times as keys and parameter name/value dictionaries as items. Default value: empty dictionary.
`tol`	The tolerance for time discrepancies when fetching a change. Default value: 10^{-10}.

The first argument allows for building the entire object in the constructor, rather than using the `add_change` method. The second one exists due to floating point errors sometimes causing the time sent into the `get_change` method to not be exactly equal to the expected one; it determines how big the difference between the two values can be while still treating them as the same.

`add_change`

```
add_change(time, par_dict):
"""
Adds a parameter change.

    Parameters::

        time --
            The time when the parameter change should be applied.

        par_dict --
            A dictionary containing the names of parameters to be
            changed as keys, and their new values as values.
"""
```

This function allows the user to build up the data structure piece by piece, rather than doing it all in the constructor.

`get_change`

```
get_new_pars(self, time):
"""
Fetches the new parameter values for a given time.

Parameters::

    time --
        The time for which to fetch the parameter changes.

Returns::

    A dictionary containing the parameter changes for the given
    time if it was found (within the tolerance set in the
    constructor), and None otherwise.
"""
```

This function gets the parameter changes to be applied at a certain time. As stated earlier, it allows for a difference (given in the constructor) between the time sent into the method and the time it finds.

3.5 Real time and simulation LQR classes

Similar to the `RealTimeMPCBase` and `MPCSimBase` classes, two classes `RealTimeLQRBase` and `LQRSimBase` have also been created for controlling real and simulated processes with LQR. These are used very similarly to their MPC counterparts, with a couple of important differences:

- Due to the differences in how the two control methods work, the constructors of the LQR classes do not take arguments for optimization file name, class name or horizon, but does take the matrix L to be used in the control law. Additionaly, they also take an argument for a dictionary containing the state values for the initial linearization point.
- The reference trajectory is handled a bit differently for the LQR classes. The values of the ParameterChanges object, rather than names and values for model parameters, should contain names and values of states corresponding to the linearization point to control around.

The states and control signal of the LQR controller are the deviations of the normal states and control signal from their linearization point values. The solver handles this by subtracting the linearization point state values from the state values

acquired from the process prior to calculating the control signal, and then adding the linearization point control signal values to the calculated control signal:

$$\Delta x_k = x_k - x_{ref} \tag{3.9}$$
$$\Delta u_k = -L\Delta x_k \tag{3.10}$$
$$u_k = u_{ref} + \Delta u_k \tag{3.11}$$

where x_k is the state vector, u_k is the control signal vector, x_{ref} and u_{ref} are their respective values in the linearization point, and L is the control law matrix (see Chapter 2.2).

3.6 Usage example

In this section an example will be given of extending the `MPCSimBase` class for usage on a specific simulated process. The process to be used for this example is the ball and beam process, as described in Chapter 4.1.1.

3.6.1 The .mop file

The first thing that needs to be done is setting up the file containing the model and optimization to be used. For this process, the file can be seen below:

```
package Ball_Beam

model Ball_Beam "ball and beam"
    parameter Real k_phi = 4.4;
    parameter Real k_v = -10.0;
    Input Real u "Input voltage";
    Real phi "Beam angle";
    Real v "Ball velocity";
    Real z "Ball position";
equation
    der(phi) = k_phi*u;
    der(v) = k_v*phi;
    der(z) = v;
end Bar;

model Ball_Beam_MPC_Model
    extends Ball_Beam;
    parameter Real Q_11 = 0.0;
    parameter Real Q_22 = 0.1;
    parameter Real Q_33 = 1.0;
    parameter Real rho = 1.0;
    parameter Real z_ref = 0.0;
end Ball_Beam_MPC_Model;

optimization Ball_Beam_MPC(
    objectiveIntegrand = Q_11*phi^2+Q_22*v^2+Q_33*(z-z_ref)^2+rho*u^2,
    startTime = 0.0, finalTime = 60.0)
    extends Ball_Beam_MPC_Model(
        phi(min = -5, max = 5),
        z(min = -10, max = 10),
        u(min = -5, max = 5));
end Ball_Beam_MPC;

end Ball_Beam;
```

Here `Ball_Beam` is the basic model of the process dynamics, `Ball_Beam_MPC_Model` is the extended model to be used for simulating the process, and `Ball_Beam_MPC` is the optimization to use in the solver. The main Modelica model `Ball_Beam` consists of two main parts: one that defines the model variables (model parameters, inputs, and states), and one that describes the system equations. This model is then extended into `Ball_Beam_MPC_model`, adding a few extra parameters to be used in defining the cost function. Notice in particular the parameter `z_ref`; it is what will be used to set up the reference trajectory. The Optimica optimization `Ball_Beam_MPC` then finally describes the optimization problem, as well as any constraints to be used. Other than the basic range constraints on the states and input

that can be seen in the example, more complex constraints can be added using the `constraints` keyword.

3.6.2 The solver class

Next, the `RealTimeMPCSim` class is extended to work with the specific process. Since in the real process the velocity of the ball is not observable, it will not be treated as such here either, and it will be estimated using a simple first order high pass filter.

```
class MPCSimBallBeam(MPCSimBase):
    def __init__(self, dt, t_hor, t_final, noise=0.,
                 par_changes=ParameterChanges(), mpc_options={},
                 sim_options={}):
        super(MPCSimBallBeam, self).__init__(
              'ball_beam.mop', 'Ball_Beam.Ball_Beam_MPC',
              'Ball_Beam.Ball_Beam_MPC_Model', dt, t_hor, t_final,
              {'_start_phi': 0, '_start_v': 0, '_start_z': 0}, {},
              ['phi', 'v', 'z'], ['u'], ['phi', 'z'], par_changes,
              mpc_options, sim_options, {}, noise)

    def estimate_states(self, x_k, x_k_last):
        _lambda = 0.3

        phi_k = x_k['_start_phi']
        z_k = x_k['_start_z']
        z_k_last = x_k_last['_start_z']
        v_k_last = x_k_last['_start_v']
        v_k = _lambda*v_k_last + (_lambda)*(z_k - z_k_last)/self.dt
        x_k = {'_start_phi': phi_k, '_start_v': v_k, '_start_z': z_k}
        return x_k

    def plot_results(self, title='Ball and beam'):
        super(MPCSimBallBeam, self).plot_results(
              outputs=['phi', 'z'], var_labels=var_labels_bar,
              title=title)
```

3.6.3 Running the experiment

Finally, the `ParameterChanges` object for the desired reference trajectory is set up, and the object of our class is created and run.

```
>>> par_changes = ParameterChanges()
>>> par_changes.add_change(15, {'z_ref': 5})
>>> par_changes.add_change(30, {'z_ref': -5})
>>> par_changes.add_change(45, {'z_ref': 0})
>>> mpc = RealTimeMPCSimBallBeam(0.05, 1, 60, 0.1, par_changes)
Creating JVM
Created JVM, JNI version 1.6

Default blocking factors have been applied to all inputs.
The prediction horizon is 1.0
>>> _ = mpc.run()
[solver output cut out]
>>> mpc.plot_results()
```

The resulting graph can be seen in Figure 3.1.

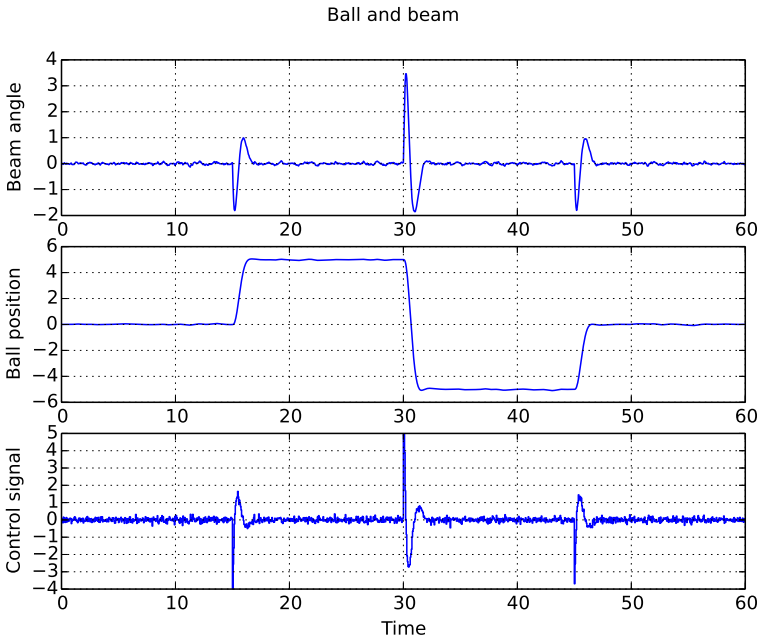


Figure 3.1 The beam angle, ball position and control voltage for a simulated run of the ball and beam process.

4

Results

To test the real-time MPC framework, it was applied to three different processes (two real and one simulated) with different properties. For each of them, an LQR controller was also used for comparison.

4.1 Ball and beam

4.1.1 Process description

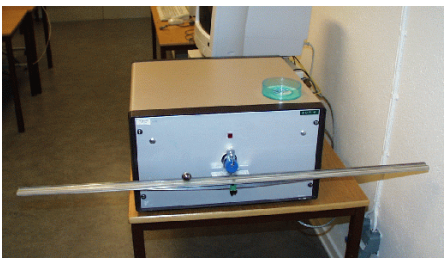


Figure 4.1 The ball and beam process.

The first process consists of a ball balanced on a beam that can rotate around an axis through its middle where the angular velocity of the beam can be controlled by a motor, as described in [Hägglund, 2011]; see Figure 4.1. The input for this process is a reference angular velocity $\dot{\phi}_{ref}$, with an internal controller making sure that this reference value becomes the actual angular velocity, and the process states are the ball position z, the ball velocity v and the beam angle ϕ; while z and ϕ can be observed, v cannot, which means that it needs to be estimated. This is done by using a simple difference quotient to estimate the derivative of z, and then taking a weighted mean of the new estimate and the one from the previous step. The process provides measurements of the states as voltages between -10 and 10 V, which is also the unit that will be used in the graphs for it.

For small angles of the beam, the process is almost completely linear; it is this linear version that is used both as the model for the MPC solver and for calculating the LQR control matrix. There is one exception that is not included in the model, though: if the vertical acceleration of the ball becomes too big, the ball gets flung off of the beam. The MPC controller can avoid this through constraints, though more care needs to be taken when using LQR: it means that the LQR controller can not be used to control the ball too far from the center of the beam. Additionally, in the actual process setup the axis that the beam pivots around is located slightly below the beam itself, something that the model does not take into account.

The cost function used for both the MPC and LQR controllers is a weighted sum of the squares of the control signal and the deviation of the ball position from its reference value. Since the MPC and LQR controllers use the same linear model and the same cost function, both controllers should give the same result.

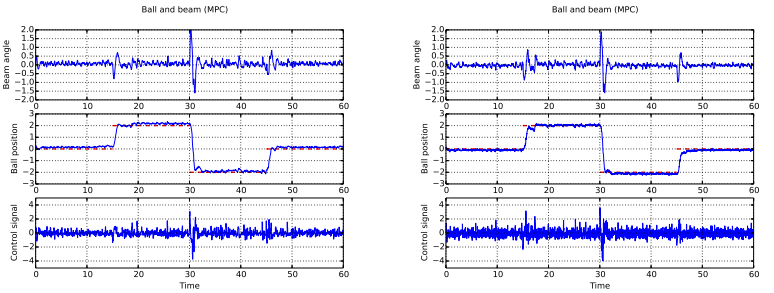
4.1.2 Results

Figure 4.2 shows experimental results of letting the MPC controller first having the ball stay at the middle of the beam, then moving it to one side, then to the other, and finally back to the middle for a few different values for the sample time and horizon, and Figure 4.3 shows the same for the LQR controller.

Comparing Figure 4.2 (a) with Figure 4.3 (a) and (b) with (b), the results look very similar. The amplitude of the control signal is a bit higher for the LQR controller than for MPC, but the resulting trajectories show barely any differences. Furthermore, comparing the (a) figures to the (b) ones shows that decreasing the sample time by half does not affect the control of the process very much either; however, halving the sample time once more resulted in the MPC controller taking too long to calculate its control signal and dropping the ball off the edge within a matter of seconds.

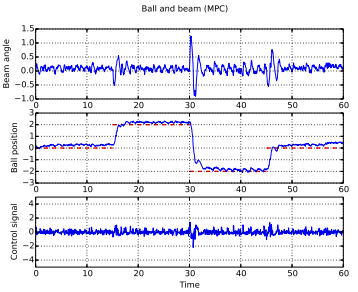
Additionally, shortening the optimization horizon for the MPC controller from one second to half a second results in slightly worse control, as comparing Figure 4.2 (a) and (c) shows: the oscillations of the ball position are larger, and the stationary error seems to be slightly larger. Halving the horizon once again then results in the process becoming unstable, as seen in (d).

As mentioned earlier, giving the ball too large of a vertical acceleration will result in it getting thrown off the beam, which is why all of the previous examples have shown the ball being controlled close to the center of the beam. This can be circumvented in the MPC controller by the use of additional constraints, showing one of its advantages to the LQR controller. The vertical acceleration of the ball is approximately proportional to the product of the beam angular acceleration $\ddot{\phi}$ and the position of the ball z for small angles of the beam, and the angular acceleration is proportional to the derivative of the control signal u, so limiting the change in the control signal Δu from one time step to the next allows for controlling the ball further from the middle of the beam without the ball getting thrown off.

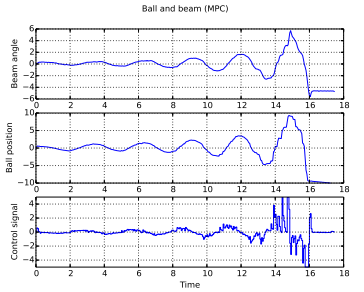


(a) Sample time 0.05 s, horizon 1 s

(b) Sample time 0.025 s, horizon 1 s



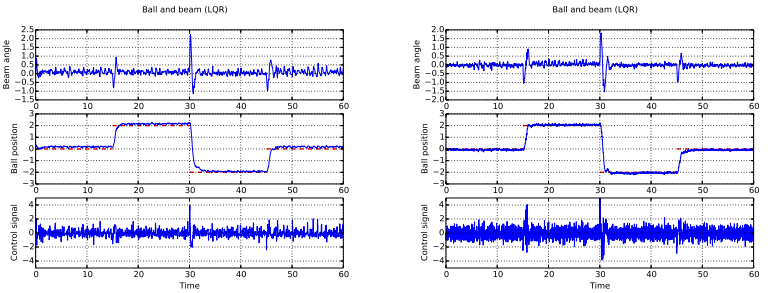
(c) Sample time 0.05 s, horizon 0.5 s



(d) Sample time 0.05 s, horizon 0.25 s

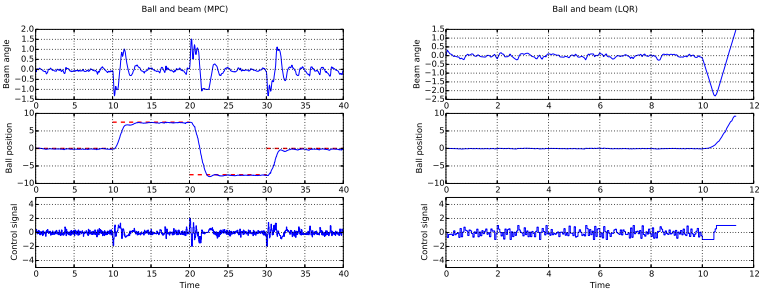
Figure 4.2 Results of applying the MPC controller to the ball and beam process for different sample times and horizon lengths. The red dashed line shows the reference value.

Figure 4.4 (a) shows this strategy, along with an additional constraint on the beam angle to slow down the movement of the ball (since the controller was too slow to react otherwise, resulting in the ball falling off the edge), being used successfully to place the ball three-quarters of the way from the center of the beam. An unsuccessful attempt to get the LQR controller to follow the same trajectory can be seen in Figure 4.4 (b). While clamping the LQR control signal to a certain voltage prevents the ball from getting flung off of the beam, it also makes the LQR controller unable to stop the ball in time for it to not fall off the ledge, since it still attempts to control the process as if it had an infinite range for its control signal.



(a) Sample time 0.05 s (b) Sample time 0.025 s

Figure 4.3 Shows the results of applying the LQR controller to the ball and beam process for different sample times.



(a) MPC (b) LQR

Figure 4.4 Shows the result of applying the MPC and LQR controller to the ball and beam process with a different reference trajectory. The MPC controller has the additional constraints $|\phi| < 1$ and $|\Delta u| < 0.9$, while the LQR controller has the control signal clamped so that $|u| < 1$.

4.2 Quadruple tank

4.2.1 Process description

The second process consists of four water tanks and two pumps, as described in [Cervin, 2013], set up as in Figure 4.5. Water can flow from the upper tanks into the lower ones and out of the lower ones, and the pumps are connected crosswise to the tanks so that one pump is connected to the upper left and lower right tanks and the other to the upper right and lower left ones. The states that are to be controlled are the water levels of the lower tanks. The process has two modes of operation: a minimum phase mode where 70% of the flow from a pump goes into the lower tank and 30% goes into the upper one, and a non-minimum phase mode where the

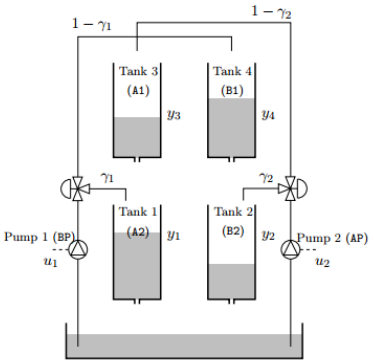


Figure 4.5 A diagram of the quadruple tank process.

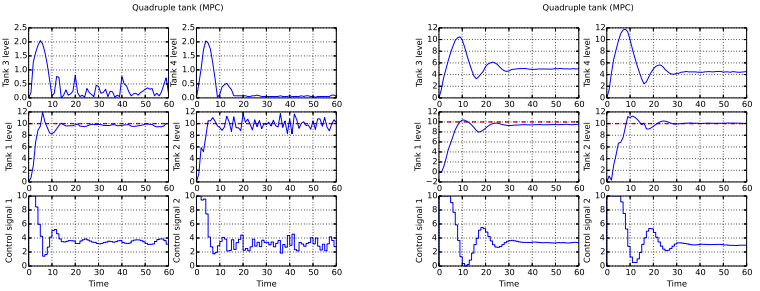
percentages are reversed. The non-minimum phase mode has slower dynamics than the minimum phase mode, and requires a longer prediction horizon, in particular when changing the difference between the water levels in the lower tanks.

Since the water flow from a tank is proportional to the square root of the water level of the tank, this process is nonlinear. Since the LQR controller is designed to work with a linear system, this means that the MPC controller ought to perform better, since the LQR controller will be operating on a linearized approximation of the actual process dynamics. The cost function for the MPC controller is the sum of the squares of the differences between the actual and reference values of the water levels in the lower tanks and the derivatives of the control signals. The `MPC` class has built-in functionality to apply a quadratic cost to the derivative of the control signal, but since this cannot be done easily in LQR, the cost function for the LQR controller uses the squares of the differences between the control signal values and what those values would be at the reference point instead.

4.2.2 Results

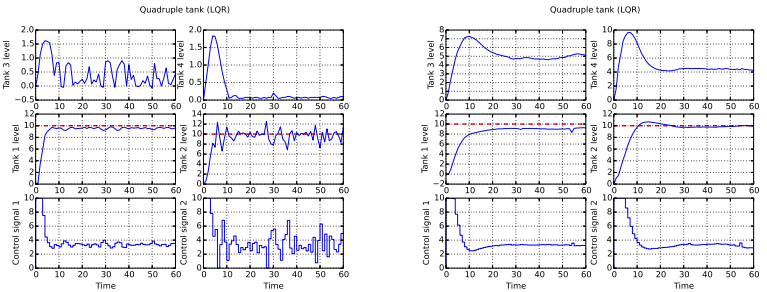
Figure 4.6 shows using the MPC controller to get the system from a starting point where both tanks are empty to an operating point where both tanks are half full for both modes, and Figure 4.7 shows doing the same using the LQR controller instead. The behaviour for both controllers are very similar, with the main differences being that the stationary error for the tank 1 is noticably larger for both modes when using the LQR controller than when using the MPC one.

Even though the MPC controller seems to perform better at eliminating stationary errors, there is still a noticable error in the level of the tank 3 in the non-minimum phase mode, as seen in Figure 4.6 (b). Figure 4.8 shows the effect of enabling integral action for the same problem. Comparing the two figures shows that enabling integral action reduces the stationary error significantly, as expected.



(a) Minimum phase, horizon: 30 s (b) Non-minimum phase, horizon: 60 s

Figure 4.6 Results of applying the MPC controller to the quadruple tank process. The sample time used is one second.



(a) Minimum phase (b) Non-minimum phase

Figure 4.7 Results of applying the LQR controller to the quadruple tank process. The sample time used is one second.

As stated earlier, the non-minimum phase mode of operation requires a longer optimization horizon than the minimum-phase mode. Figure 4.9 (a) shows the effects of halving the horizon length from 60 to 30 seconds. As can be seen, this significantly decreases the quality of control. In order to get around this, an additional cost term can be introduced into the optimization: the predicted cost of running the process with LQR control from the end of the optimization horizon to infinity, which can be written as $(x - x_{ref})^T S(x - x_{ref})$ where x is the state vector, x_{ref} is the reference state vector, and S is the matrix that solves the discrete-time algebraic Ricatti equation (see Chapter 2.2). Adding this final cost to the optimization gives the result in 4.9 (b), which is a significant improvement over (a).

Finally, Figure 4.10 shows simulations using a working point close to the maximum capacity of the lower tanks in non-minimum phase mode using both MPC and LQR controller. The MPC controller respects the constraints of not overflowing the

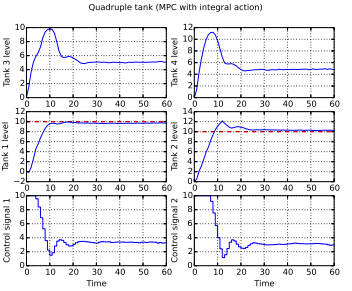
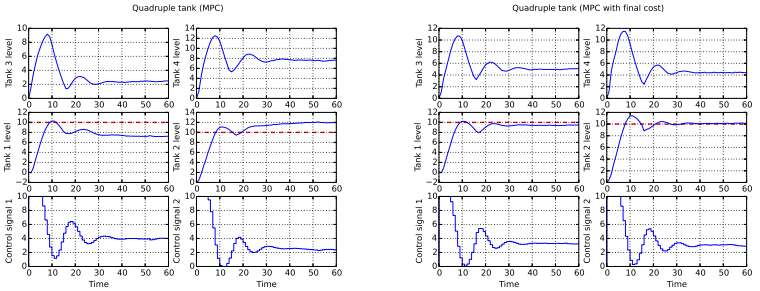


Figure 4.8 Results of applying the MPC controller to the non-minimum-phase quadruple tank process with integral action enabled. Sample time: 1 s, horizon: 60 s.



(a) Without final cost

(b) With final cost

Figure 4.9 Results of applying the MPC controller to the non-minimum-phase quadruple tank process with and without final cost. Sample time: 1 s, horizon: 30 s.

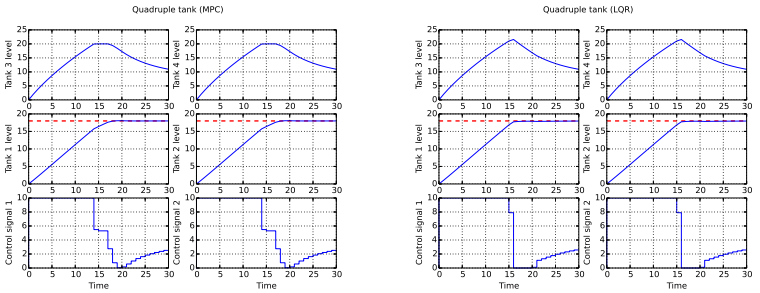
upper tanks (which have a mazimum capacity of 20), while the LQR controller is unable to do so.

4.3 Crane

4.3.1 Process description

The last process is a three-dimensional crane, with a trolley that can move freely in the horizontal plane and a load hanging from it by a variable-length rope, as shown in Figure 4.11 and as described in [Lee, 1998]. Due to insufficient documentation it was not possible to get the real process working in time, so all results in this section are from a simulated version of it.

The process is non-linear, as its governing equations involve trigonometric functions. However, much like a simple non-elastic pendulum, it can be approximated



(a) MPC (b) LQR

Figure 4.10 A simulation of trying to almost fill the lower tank in non-minimum phase mode with both MPC and LQR controllers. The maximum capacitiy of both the upper and lower tanks are 20.

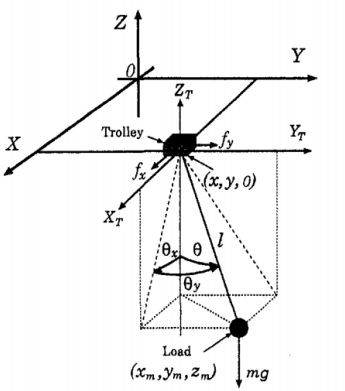


Figure 4.11 A diagram of the crane process.

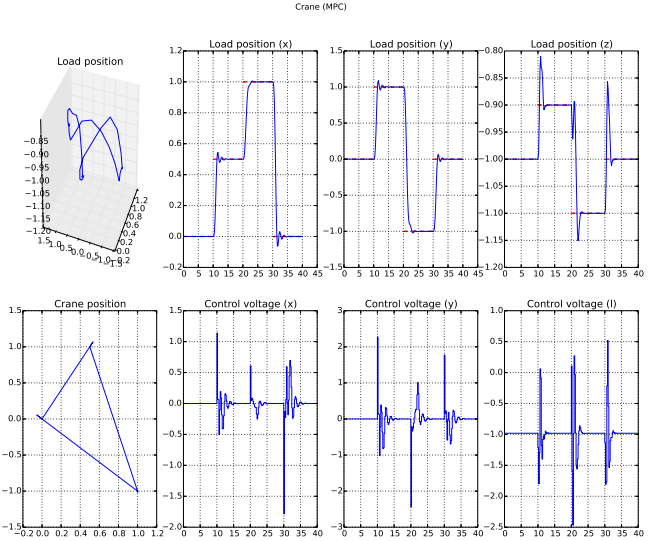
with a linear system when the angle of the load is small, although this requires the length l of the rope holding the load to be constant. This means that the LQR controller should be able to be used to decent effect as long as the change in l is small. The cost function for the process is, similarly to the one for the ball and beam, a weighted sum of the squares of the control signal and the distance from the current position of the load to its desired position.

4.3.2 Results

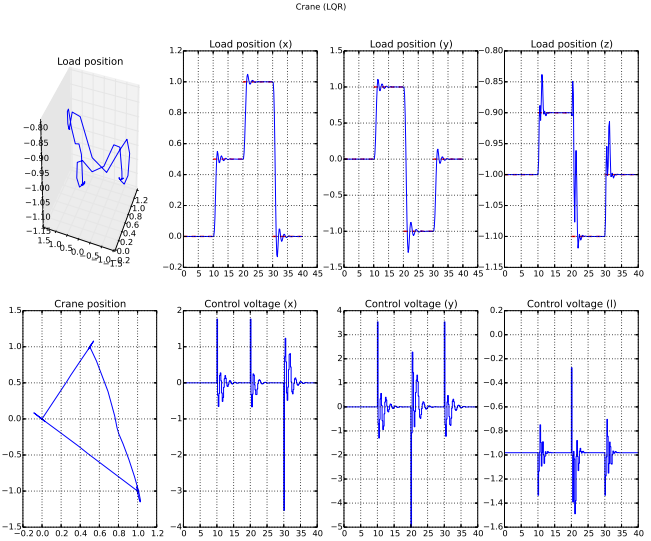
Figure 4.12 shows using the MPC and LQR controllers to move the load of the crane between the points $(0.0, 0.0, -1.0)$, $(0.5, 1.0, -0.9)$, and $(1.0, -1.0, -1.1)$. As can be seen in the figure, there is not much of a difference between the per-

formance of the two controllers. However, in Figure 4.13, where the reference points are more spread out along the z axis, the MPC controller is vastly superior to the LQR controller. In this figure, the reference points are instead $(0.0, 0.0, -1.0)$, $(0.5, 1.0, -0.5)$, and $(1.0, -1.0, -1.5)$. Since the linearization of the crane process depends on the length of the rope holding the load, changing this length significantly makes the LQR controller not work very well; in order for the LQR controller to work in this case, a whole new controller with a different gain matrix would need to be calculated for each time the reference point is changed.

Something that can be done with the MPC controller with the help of constraints but that cannot be done with the LQR controller is making the load avoid obstacles. Figure 4.14 shows moving the load from the point $(0, 0, -1)$ to the points $(1, 0, -1)$, while avoiding the elliptic cylinder $25(x_l - 0.5)^2 + 16(z_l + 1.0)^2 \geq 1$. The reason a cylinder was choosen as the obstacle is that the optimization solver does not handle discontinuities very well, such as the edges of a cuboid. Due to the constraint only applying at the sample points, the load does pass through the cylinder briefly; if this method was to be used in control of a real crane, the margin of error ought to be larger than it otherwise would need to be to take this into account..

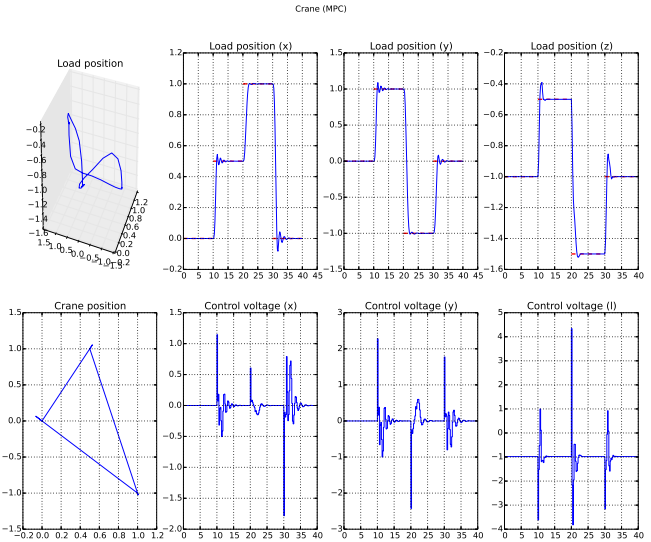


(a) MPC

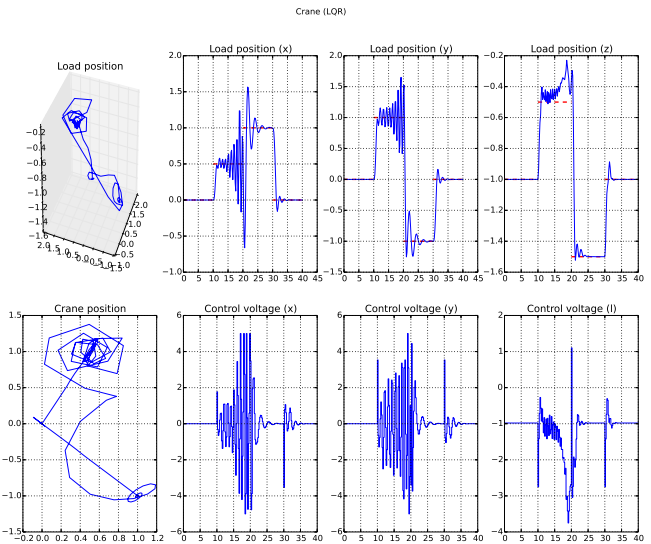


(b) LQR

Figure 4.12 Results of applying the MPC and LQR controllers to the crane process, with reference points that don't require the load to move very much vertically.



(a) MPC (sample time: 0.2 s, horizon: 2 s)



(b) LQR

Figure 4.13 Results of applying the MPC and LQR controllers to the crane process, with reference points that require more vertical movement of the load.

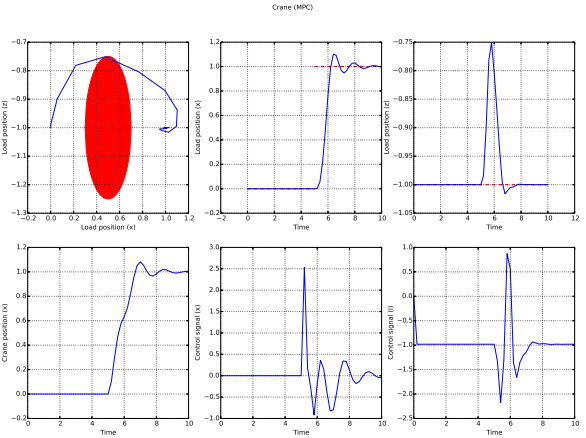
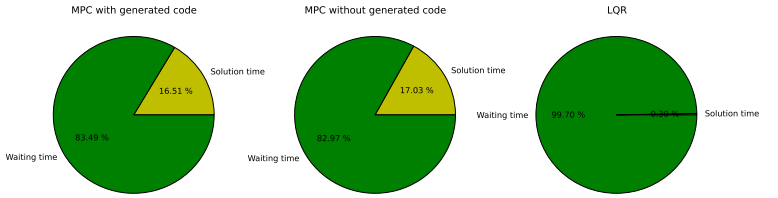


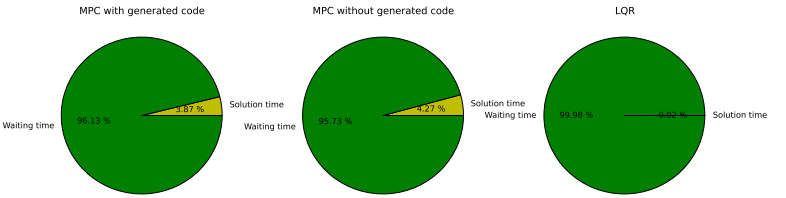
Figure 4.14 Results of applying the MPC controller to the crane process, with the additional constraint that the load should never be inside the red ellipsoid.

4.4 Performance

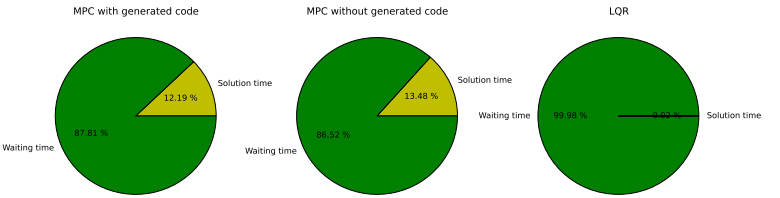
Figure 4.15 compares the time used by the solver and the total time for each process for the LQR controller and the MPC controller with and without using generated code. It can be easily confirmed that LQR is a much faster control method than MPC, and it can also be seen that using pre-generated code for the MPC solver does not result in any noticable speedup for any of the computations.



(a) Ball and beam



(b) Quadruple tank



(c) Crane

Figure 4.15 Total percentage of time taken for calculating the control signal for the three processes, using MPC with and without generated code and LQR.

Table 4.1 shows the average time taken to calculate the control signal each sample for the MPC solver. The ball and beam is by far the fastest, followed by the crane and the quadruple tank. This fits with the number of variables used by the

optimization solver for these runs, which is 323 for the ball and beam, 1324 for the crane, and 1193 for the quadruple tank.

	Total time	Solver time	Solver time/sample
Ball and beam (sample time 0.05 s, horizon 1 s)			
MPC with generated code	60.32 s	9.96 s	8.3 ms
MPC without generated code	60.32 s	10.27 s	8.6 ms
LQR	60.47 s	0.18 s	0.15 ms
Quadruple tank (sample time 1 s, horizon 30 s)			
MPC with generated code	60.35 s	2.34 s	39 ms
MPC without generated code	60.39 s	2.58 s	43 ms
LQR	60.31 s	0.012 s	0.20 ms
Crane (sample time 0.2 s, horizon 4 s)			
MPC with generated code	40 s	4.87 s	24 ms
MPC without generated code	40 s	5.39 s	27 ms
LQR	40 s	0.0082 s	0.041 ms

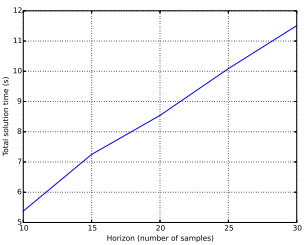
Table 4.1 Times measured from controlling the three processes.

Table 4.2 shows data obtained from running ipython's profiler on a simulation of the crane process. The main points of interest here are `sample` (the function of the `MPC` class that calculates the next control signal), and `time.sleep` (a Python standard library function that just makes the program execution wait). Studying the time spent in these three function shows that the whole simulation took 10.298 s, 5.513 s of which was spent waiting, which leaves 4.766 s. Out of this time, 3.285 s or 68.9% was spent in the solver, with most of the rest (1.349 s or 28.3%) being spent simulating the process. This only leaves 0.151 s (1.5%) spent on framework overhead. Overall, this means that it would not be possible to save much time just by optimizing the code of the framework; in order to improve performance, other methods would need to be sought.

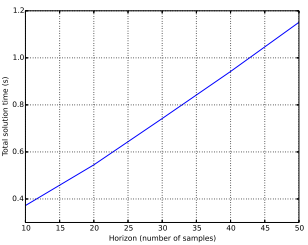
Figure 4.16 shows that the total time taken by the MPC solver increases linearly with the number of samples in the optimization horizon. This of course means that decreasing the sample time while keeping the horizon constant will increase the computation time needed; however, it also means reducing the time available for computation each sample, meaning that trying to improve control by just increasing the sample rate can easily backfire by rendering the solver unable to calculate the control signal fast enough.

cumtime	filename:lineno(function)
10.298	<string>:1(<module>)
10.298	testassets.py:797(run)
10.298	testassets.py:245(run)
5.532	testassets.py:383(wait_and_get_states)
5.513	{time.sleep}
3.285	mpc.py:802(sample)
2.853	__init__.py:138(foo)
2.843	casadi_collocation.py:536(solve_nlp)
2.808	casadi_core.py:18964(evaluate)
2.807	{_casadi_core.Function_evaluate}
1.349	testassets.py:674(send_control_signal)
1.345	{method 'simulate' of 'pyfmi.fmi.FMUModelME1' objects}
0.767	fmi_algorithm_drivers.py:663(solve)
0.766	{method 'simulate' of 'assimulo.ode.ODE' objects}
0.387	casadi_collocation.py:4187(get_result)

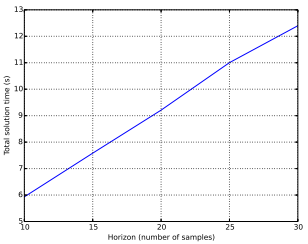
Table 4.2 Profiling data from running a ten-second simulation of the crane process. The first column is cumulative processor time in seconds (all time spent in a function and all functions called from it), and the second is the function name and where the function is located.



(a) Ball and beam (sample time: 0.05 s)



(b) Quadruple tank (sample time: 1 s)



(c) Crane (sample time: 0.2 s)

Figure 4.16 Time taken by the MPC solver as a function of the horizon length in number of samples.

5

Conclusion

In this thesis, a framework for real-time model predictive control has been implemented, and with that the fact has been demonstrated that it is possible to use the optimization toolchain of JModelica.org for MPC on real processes. The framework has been applied to controlling two real processes (one fast and linear, one slow and non-linear) as well as a simulated one (fast and non-linear). Comparisons with an LQR controller have shown that while LQR is much faster, MPC has features which LQR lacks, such as the ability to adhere to constraints. Additionally, the LQR controller has trouble controlling non-linear processes when the reference point moves far from the linearization point, a problem that does not exist for the MPC controller since it does not need a linear problem to work.

Profiling of the real-time MPC framework shows that it does not have any significant inefficiencies. The vast majority of the time between getting the state values from the process and sending the next control signal is spent inside the MPC solver, with not much time spent on framework overhead. While it would probably be possible to optimize it further, doing so would most likely not result in any major improvements. Instead, speedups would have to be sought by improving either the solver or the problem formulation.

5.1 Further work

The main thing that still needs to be done is testing the real-time MPC framework on a real process that is at least as fast as and more complex than the ball and beam, to study if it is able to control the process or if it needs further improvements in terms of time taken by the solver. The crane process was supposed to fill this slot, but as it was not possible to get working in time tests were only done on the simulated version. While the framework was able to control the simulation, the documentation that does exist for the real process suggests that it might require a sample frequency of 100 Hz, which is twenty times faster than what was used for the simulation and beyond what the current version of the framework can do. Since the speed of the current framework does not seem to be able to be improved

much by simple optimization, this would require looking into ways of speeding up the MPC or optimization solvers used by it. One way that this might be possible to do would be by using advanced-step MPC, where a parametrized version of the optimal control problem is solved while waiting for the next sample, decreasing the computation time needed for calculating the control signal after getting the new state vector. [Zavala and Biegler, 2009]

Additionally, a feature that might improve performance a bit would be the ability to set a new initial trajectory when switching reference points. The MPC solver uses the optimization results from the last time step as an initial guess, which means that when the reference point changes, the initial guess can be very far from the actual solution, which causes the solver to take longer or even in some cases fail to converge. The former is not necessarily much of a problem if the current state is a stationary point, but can be troublesome otherwise. Another possible way to handle this would be having the solver be aware of the reference trajectory for the entire horizon rather than just the current reference point, so that it can take reference changes into account before they happen; however, this is not something that the MPC solver currently supports.

Finally, if the framework is to be used for more serious control of real processes, some sort of error handling needs to be implemented. As it is, if the solver runs into an error during control the process needs to be shut down manually. The bare minimum to be done would be to just have the controller send a control signal of zero in case of errors, but even better would be switching to another controller that could gracefully bring the process to a desired shutdown state.

Bibliography

Åkesson, J., K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit (2010). "Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problems". *Computers and Chemical Engineering* **34**:11, pp. 1737–1749.

Andersson, J., J. Gillis, and M. Diehl (2015). *User Documentation for CasADi v2.2.0+436.d16f241*.

Åström, K. J. and B. Wittenmark (1997). *Computer-Controlled Systems*. Prentice Hall. Chap. 11.

Axelsson, M. (2015). *Nonlinear Model Predictive Control in JModelica.org*. MA thesis. Department of Automatic Control, Lund University.

Balandat, M., W. Zhang, and A. Abate (2012). "On infinite horizon switched lqr problems with state and control constraints". *Systems & Control Letters* **61**.

Camacho, E. F. and C. Bordons (2007). *Model Predictive Control*. Springer-Verlag London.

Cervin, A. (2013). *Multivariable Control - Laboratory Exercise 2 - The Quadruple Tank*. Department of Automatic Control, Lund University.

García, C. E., D. M. Prett, and M. Morari (1989). "Model predictive control: theory and practice – a survey". *Automatica* **25**:3, p. 335.

Hägglund, T. (2011). *Reglerteknik AK - Föreläsningar*. Chap. 14.

Lee, H.-H. (1998). "Modeling and control of a three-dimensional overhead crane". *Journal of Dynamic Systems, Measurement and Control*, p. 471.

Richalet, J., A. Rault, J. L. Testud, and J. Papon (1978). "Model predictive heuristic control: application to industrial processes". *Automatica* **14**, p. 412.

Wächter, A. and L. T. Biegler (2006). "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". *Mathematical Programming* **106**:1, p. 25.

Zavala, V. M. and L. T. Biegler (2009). "The advanced-step nmpc controller: optimality, stability and robustness". *Automatica* **45**, p. 86.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER´S THESIS

Date of issue
July 2015

Document Number
ISRN LUTFD2/TFRT--5986--SE

Author(s)
Sebastian Ekström

Supervisor
Toivo Henningsson, Modelon
Anton Cervin, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle
Real Time Model Predictive Control in JModelica.org

Abstract

In this thesis a framework for real-time model predictive control has been developed for JModelica.org, which is an open-source platform for simulation and analysis of dynamical systems. Model predictive control (MPC) is an advanced optimizationbased control method that uses a model of the process being controlled to optimize control. The framework was tested on three different processes, real and simulated, and its performance was compared with that of an linear-quadratic regulator (LQR), which is a simpler type of controller that uses multiplication with a pre-calculated matrix to calculate the control signal from the state vector. The MPC controller was found to perform as well as or better than the LQR controller in all cases, with the main improvements being seen in the MPC controller's ability to handle process constraints or when far from the LQR controller's linearization point; however, the LQR controller was much faster in calculating the control signal. This also served as a first test of using JModelica.org to perform MPC on real processes, and although it performed well on the two it was tested on, further work will be needed if the MPC framework should be able to handle processes that are much faster or more complex.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-51

Recipient's notes

Security classification