A Framework for Nonlinear Model Predictive Control in JModelica.org

Magdalena Axelsson[1] Fredrik Magnusson[2] Toivo Henningsson[1]

[1]Modelon AB, Lund, Sweden, {magdalena.axelsson, toivo.henningsson}@modelon.com
[2]Department of Automatic Control, Lund University, Sweden fredrik.magnusson@control.lth.se

Abstract

Nonlinear Model Predictive Control (NMPC) is a control strategy based on repeatedly solving an optimal control problem. In this paper we present a new MPC framework for the JModelica.org platform, developed specifically for use in NMPC schemes. The new framework utilizes the fact that the optimal control problem to be solved does not change between solutions, thus decreasing the computation time needed to solve it. The new framework is compared to the old optimization framework in JModelica.org in regards to computation time and solution obtained through a benchmark on a combined cycle power plant. The results show that the new framework obtains the same solution as the old framework, but in less than half the time.
Keywords: Nonlinear Model Predictive Control, JModelica.org, Optimization, IPOPT

1 Introduction

Model Predictive Control (MPC) is an optimization-based control strategy based on the repeated on-line solution of an open-loop optimal control problem at discrete time points. Feedback is incorporated by measuring the state at each of these discrete timepoints and using the measured state as the initial state in the optimal control problem. From each optimization the first input in the optimal control sequence computed is applied to the system. Two of the main advantages of MPC compared to other control methods are that

- it easily extends to multivariable systems with multiple inputs and outputs.
- it intrinsically handles constraints on all system variables.

In general, one distinguishes between linear and nonlinear model predictive control (LMPC/NMPC). In the case of linear MPC, where the system model and any constraints imposed upon the system are linear and the cost is quadratic, the optimal control problem can be cast as a quadratic program. Quadratic programs can be solved efficiently on-line. In case of nonlinear MPC, the optimal control problem is instead cast as a NonLinear Program (NLP), which is more computationally demanding to solve. The long computation time of the optimization, along with the risk that sometimes an optimal solution is not found at all, are two of the main limiting factors for successful application of NMPC in industry. (Allgöwer et al., 2004).

JModelica.org is an open-source platform for simulation, optimization and analysis of complex dynamic systems described by Modelica models (Åkesson et al., 2010). In recent research its use has been proposed for the solution of the optimal control problem for NMPC applications in several different fields, including (Cavey et al., 2014) where a JModelica.org/NMPC scheme was successfully implemented to control the heating system in a building, (Berntorp and Magnusson, 2015) where the use of JModelica.org was proposed to solve the NMPC optimal control problem in a hierarchical predictive control scheme for the lane keeping of a vechicle and (Larsson et al., 2013) where a case study of the start up of a combined cycle power plant using a JModelica.org/NMPC scheme was made. Features and performance for NMPC application using JModelica.org has been evaluated in (Hartlep and Henningsson, 2015).

The optimization algorithm in JModelica.org is currently embedded into an open-loop framework, which is well suited for solving dynamic optimization problems once. This paper describes a new optimization framework, the *MPC framework*, developed specifically for the repeated solution of the optimal control problem in NMPC schemes (Axelsson, 2015). The new MPC framework is built around the same optimization algorithm as the open-loop framework, but for efficiency it exploits the fact that the optimal control problem to solve has the same structure in each consecutive optimization. The main goals of the new MPC framework has been to decrease the average computational time for one optimization as much as possible while streamlining the setup of NMPC schemes, making JModelica.org faster and easier to use for NMPC applications.

The rest of the paper is outlined as follows. Section 2 gives general background on Nonlinear Model Predictive Control and optimization using JModelica.org. Section 3 presents the new MPC framework implemented in this paper. Section 4 compares the MPC framework to the existing open-loop framework in terms of performance on an NMPC setup of a combined cycle power plant. The section also evaluates the effects of warm starting the NLP solver. Finally, Section 5 summarizes the paper and further work discussed.

2 Background

2.1 MPC

This subsection presents the type of optimal control problems that are considered in this paper, and a basic MPC control algorithm. A common problem in MPC and how it can be solved is also discussed.

2.1.1 Optimal Control Problem

An optimal control problem includes a model of the system that is to be controlled, an objective function and, if desired, constraints on variables in the system.

The objective function, also commonly referred to as the cost function, expresses what is to be minimized in the optimization. For MPC problems the objective function is typically formulated to penalize deviations of some variables from their set points. The variables that have set points are called the *controlled variables* and may be any of the different types of variables in the system. For ease of notation we introduce w as the controlled variables

$$w = (x_{\text{controlled}}, y_{\text{controlled}}, u_{\text{controlled}}) \tag{1}$$

where $x_{\text{controlled}} \subset x, y_{\text{controlled}} \subset y$ and $u_{\text{controlled}} \subset u$.

In practice all processes are subject to constraints. These include physical constraints, such as actuators that have a limited working range and slew rate as well as constructive, safety and environmental constraints imposed on the process to make sure it is operating in a safe and desired manner. Examples of constraints included in the second category are maximum and/or minimum levels in tanks, temperatures, pressures, flow rates etc.

The purpose of an optimal control problem is to find the input that minimizes the objective function, while upholding the constraints imposed upon the system. A typical optimal control problem for MPC applications, expressed in continuous time, can have the form

minimize

$$f(w) = \int_{t_0}^{t_f} (w_{\text{ref}} - w(t))^T Q (w_{\text{ref}} - w(t)) dt \tag{2a}$$

with respect to

$$x(t) \in \mathbb{R}^{n_x}, \qquad y(t) \in \mathbb{R}^{n_y}, \qquad u(t) \in \mathbb{R}^{n_u},$$

subject to

$$F(\dot{x}(t), x(t), y(t), u(t)) = 0, \tag{2b}$$

$$x(t_0) = x_0, \tag{2c}$$

$$x_L \leq x(t) \leq x_U, \tag{2d}$$

$$y_L \leq y(t) \leq y_U, \tag{2e}$$

$$u_L \leq u(t) \leq u_U, \tag{2f}$$

$$g(\dot{x}(t), x(t), y(t), u(t)) \leq 0, \tag{2g}$$

$$G(\dot{x}(t_f), x(t_f), y(t_f), u(t_f)) \leq 0, \tag{2h}$$

$$\forall t \in [t_0, t_f]$$

where (2a) is the objective function where w are the controlled variables, w_{ref} are their set points and Q is a weighting matrix. The Modelica model, expressed by a set of Differential Algebraic Equations (DAE) describing the system dynamics, is included in (2b) where $x(t)$ are the differentiated variables, $y(t)$ are the algebraic variables and $u(t)$ are the control variables. We rely on Modelica compilers to perform index reduction and thus only consider DAE systems of index at most 1, so the differentiated variables correspond to the system state. The initial conditions (2c) define the initial state $x(t_0)$ of the system. Here x_0 are the initial condition parameters. Additionally, (2d)-(2h) are the constraints imposed upon the system, where(2d)-(2f) are variable bounds with lower limit $\{x, y, u\}_L$ and upper limit $\{x, y, u\}_U$, (2g) is path constraints and (2h) is terminal constraints. The optimal control problem is considered over a prediction horizon of $H_p = t_f - t_0$ seconds.

2.1.2 Control Algorithm

The general idea with MPC is that the optimal control problem (2) is solved on-line at each sample point t_k. The solution to (2) will determine the input that is to be applied to the system until the next sample point t_{k+1}. The time between two sample points is called the sample period.

The control algorithm states that at each sample point t_k, the following steps should be carried out:

1. Obtain an estimate of the initial state $x_{\text{est}}(t_k)$.
2. Set the initial condition parameters $x_0 = x_{\text{est}}(t_k)$, the start time $t_0 = t_k$ and the final time $t_f = t_0 + H_p$.
3. Solve (2).
4. Apply u_1 to the system, where u_1 is the first value in the resulting control sequence. Hold the input constant through the entire sample period.

DOI 10.3384/ecp15118301

To be able to solve (2) we need to know the initial state of the system. Typically however, all the states are not measurable. It is therefore assumed that a state estimator is used to estimate the initial state in step 1. A Moving Horizon Estimator is an example of an optimization-based state estimator for nonlinear systems. An MHE framework is currently being developed for JModelica.org (Larsson, 2015).

2.1.3 Constraint softening

A common problem in MPC is that (2) is infeasible for the estimated initial conditions. This can happen if the process is running close to a limit and a particularly large disturbance occurs, or if the model is not good enough and the process behaves differently than predicted. Infeasibility caused by constraint violations can be prevented by softening the constraints. This means that rather than to regard constraints as hard limits which may never be crossed, we soften them by allowing them to be crossed, but only if necessary. One way of softening a constraint is by adding a new variable, a so-called *slack variable*, to the problem. This slack variable is heavily penalized in the cost function and is defined in such a way that it needs to be non-zero if the constraint is violated. With a large enough constraint penalty this gives the solver an incentive to keep the slack variable at a small value, meaning that the original constraint is upheld (Maciejowski, 2002). The MPC framework supports automatic softening of variable bounds using this method. More details on the automatic softening will be presented in Section 3.3.1.

2.1.4 Other NMPC tools

Another framewoork that may be used for NMPC applications based on Modelica models is the one described in (Franke et al., 2003). The framework described in that article uses multiple shooting to discretize the problem and HQP (Franke et al.), to solve the resulting NLP. One drawback with this tool, compared to the MPC framework described here, is that it implements a quasi-Newton type algorithm, meaning only first order derivatives are utilized.

ACADO toolkit is another tool suitable for NMPC applications on embedded hardware (Houska et al., 2011). However, ACADO toolkit does not have a Modelica interface and models are instead written in C++.

2.2 Optimization in JModelica.org

This subsection presents how optimization problems are solved in JModelica.org. It briefly explains the theory of the discretization and solution process, as well as how the open-loop optimization framework in JModelica.org works.

2.2.1 Discretization

The optimization algorithm in JModelica.org can solve different types of dynamic optimization problems, including the optimal control problem for MPC applications but also parameter estimation and parameter optimization problems. The optimization problems to be solved are expressed using *Optimica* (Åkesson, 2008); a Modelica extension including language constructs to e.g. formulate the objective function and constraints.

The optimization problem needs to be discretized in order for numerical solvers to solve it. To discretize the problem we let a finite number of discrete time points on the prediction horizon represent the trajectories of all variables in the optimization problem.

The optimization algorithm in JModelica.org uses direct collocation to transcribe the infinite-dimensional optimization problem into a finite-dimensional NLP (Magnusson and Åkesson, 2015). The collocation methods supported in JModelica.org are Radau and Gauss collocation. They both start with dividing the prediction horizon into n_e *collocation elements*. In each element, n_c number of *collocation points* are placed. The total number of collocation points thus becomes $n_e \cdot n_c$, and it is in these points that we consider the optimization problem. This means that each time-dependent variable in the original optimization problem, yields a set of $n_e \cdot n_c$ optimization variables in the NLP, one at each collocation point. The collocation points approximate the system trajectories by polynomials through interpolation. This in turn means that each constraint or equation in the original optimization problem, which include a time-dependent variable, is transcribed into a set of $n_e \cdot n_c$ constraints or equations in the NLP, one at each collocation point. The structure of the resulting NLP is dependent on the structure of the original optimization problem and the collocation options chosen.

2.2.2 Solving the NLP

JModelica.org uses the third party NLP solver IPOPT (Interior Point OPTimizer) to solve the resulting NLP (Wächter and Biegler, 2006). IPOPT implements a primal-dual interior point method to find a solution to the NLP, which after transcription has the general form

minimize

$$f(z) \tag{3a}$$

with respect to

$$z \in \mathbb{R}^{n_z},$$

subject to

$$z_L \leq z \leq z_U \tag{3b}$$
$$g_e(z) = 0, \tag{3c}$$
$$g_i(z) \leq 0, \tag{3d}$$

where z are the optimization variables and (3b) their bounds. All constraints have been categorized depending on whether they are equality constraints g_e (3c) or inequality constraints g_i (3d). An optimal solution to the NLP requires the Karush-Kuhn-Tucker (KKT) conditions to be satisfied (Boyd and Vandenberghe, 2004). The KKT conditions can be derived from the Lagrangian function, which is defined as

$$L(z,\lambda,\nu) = f(z) + \lambda \cdot g_e(z) + \nu \cdot g_i(z), \tag{4}$$

where $\lambda \in \mathbb{R}^{n_{ge}}$ and $\nu \in \mathbb{R}^{n_{gi}}$ are the Lagrange multipliers. The Lagrange multipliers are also treated as iteration variables in the solution process. To separate them from the optimization variables z, the Lagrange multipliers are often called the *dual* variables while z are called the *primal* variables.

As an interior point method IPOPT considers the auxiliary barrier problem formulation

$$\min_z \; J_\mu(z) = f(z) - \mu \sum_{i=0}^{n_z} ln(z_i) \tag{5a}$$

$$\text{s.t. } g(z) = 0 \tag{5b}$$

where μ is the barrier parameter (Wächter, 2009). This transformation from (3) to (5) is handled internally in IPOPT and for ease of notation it has here been assumed that the variables z only have lower bounds of zero. Given a value of the barrier parameter $\mu > 0$, which tends to zero during the solution procedure, the barrier objective function J will go towards infinity if any variable z approaches its bound of zero. The initial value of the barrier parameter μ_{init} determines how far away from the constraints that the intermediate solution will be pushed. For an initial guess very close to the optimal solution, a small value of μ_{init} might decrease the iterations needed to get to the optimal solution, while for a less accurate initial guess a larger value of μ_{init} typically gives faster convergence.

Given a good enough initial guess of the optimization variables the solver will converge to a local optimal solution of the NLP. An initial guess closer to the optimum will also in most cases reduce the number of iterations needed to get there. For MPC applications, it is typically a good idea to use the solution to the last optimization as the initial guess for the next.

Since the dual variables are iteration variables as well, they also need an initial guess. IPOPT has a method to compute an initial guess for the dual variables automatically. However, in the same way as for the primal variables, it might be a good idea to use the previous result of the dual variables as initial guess instead. Providing an initial guess of both primal and dual variables is called warm starting the solver and will be evaluated in section 4.3.

JModelica.org is interfaced with IPOPT through CasADi (Computer algebra system with Automatic Differentiation)(Andersson, 2013). CasADi is an open-source, symbolic framework for automatic differentiation. It is used in JModelica.org for two main reasons; to give all optimization variables and expressions a symbolic representation using CasADi Interface (Lennernäs, 2013) and to calculate function derivatives. Scripts for JModelica.org are written in Python.

2.2.3 Optimization framework

Solving an optimization problem using the open-loop framework in JModelica.org is done in three steps:

1. Pre-processing: In the pre-processing step, the optimization problem is transcribed into an NLP by means of direct collocation as described in the previous section. All optimization variables in the resulting NLP are given a symbolic representation using CasADi and a solver object is created and initialized.

2. Solution: The solution step is handled completely by the third-party NLP solver IPOPT and includes the iterative steps that the solver takes to find a solution to the NLP.

3. Post-processing: The NLP solver returns the result for all optimization variables in one long vector. The post-processing step includes processing the result so that it is presented to the user in a convenient way, which includes creating a result object and writing the result to file.

The total computation time to solve an optimization problem is thus the time for each of these steps combined.

3 MPC framework

This section presents the new MPC framework implemented in this paper. It includes a comparison to the open-loop framework as well as a presentation of how it is used and a few of the features included in it.

3.1 Compared to open-loop framework

The MPC framework was created to make the total computation time for solving the optimal control problem shorter, while making JModelica.org easier to use for MPC applications. The reason the computation time is shorter using the MPC framework compared to directly using the open-loop framework is that the MPC framework utilizes the fact that the structure of the discretized optimal control problem is the same in each consecutive optimization. This allows performing the discretization only once, and reusing the resulting NLP for all optimizations. Solving the optimal control problem using the MPC framework is done in these steps:

DOI 10.3384/ecp15118301

0. Initialization: In the initialization step, the optimal control problem is transcribed into an NLP as in the pre-processing step of the open-loop framework.

1. Pre-processing: The initial condition parameters as well as the start and final time of the optimization horizon are updated. A new initial guess for the optimization variables is also set.

2. Solution: The solution step includes the same things as this step in the open-loop framework, with the difference that warm start of the solver can be enabled.

3. Post-processing: All u_1 values are extracted from the result and returned to the user.

Step 0 is only done once, off-line, when an MPC object is created, while steps 1-3 are executed in each optimization. Since the time-consuming discretization has been moved to initialization the pre-processing time in the MPC framework is significantly decreased. The post-processing time is also decreased due to the MPC framework not creating a result object after each optimization but rather only returning the computed u_1 values instead.

The open-loop framework hardcodes the values of Modelica parameters, including initial conditions. To enable the update of the initial conditions in Step 1 for the NLP constructed in Step 0, the initial conditions are instead introduced as symbolic NLP parameters. Defining the initial conditions as parameters x_0 thus makes it possible to update their values between optimizations.

3.2 Example

The MPC framework includes features that simplify the use of JModelica.org for MPC purposes. After the setup, the MPC object requires very little interaction from the user as most things are handled internally. The only information that has to be supplied to the MPC object is the next initial state. The Python code excerpt below gives an example on how the MPC framework is used. Here it is assumed that the optimization problem `opt_problem`, the optimization options `options`, the sample period `sample_period` and the prediction horizon `horizon` have already been defined. A detailed description of how to do this is found in (Axelsson, 2015). The Optimica code for the benchmark system used in this article will be presented in Section 4.2.

The first line of this example shows how to utilize the support for automatically softening variable bounds, in this case for the variable `plant.sigma`. In this example, artificial measurement data is created by simulating an FMU of the system from the initial state and one sample period forward in time, with the optimal input obtained from the optimization.

```
# Define variable bounds to be softened
cvc = {'plant.sigma': 1e5}

# Create the MPC object
MPC_object = MPC(opt_problem, options,
  sample_period, horizon, constr_viol_costs=cvc
      )

# Set initial state
x_k = {}
for name in op.get_state_names():
    x_k["_start_"+name] = opt_problem.get("
        _start_"+name)

for k in range(nbr_opt):
# Update the state and optimize nbr_opt times
    MPC_object.update_state(x_k)
    u_k = MPC_object.sample()

    # Simulate for one sample period with the
    # optimal input u_k
    sim_model.reset()
    sim_model.set(x_k.keys(), x_k.values())
    sim_res = sim_model.simulate(
        start_time = k*sample_period,
        final_time = (k+1)*sample_period,
        input=u_k, options=sim_opts)

    # Extract state values at end of sim_res
    x_k = MPC_object.extract_states(sim_res)
    # Add measurement noise to states

# Get result and extract variable profiles
opt_res = MPC_object.get_complete_results()
opt_plant_sigma = opt_res['plant.sigma']
```

3.3 Features

3.3.1 Softening Variable Bounds

The need for constraints to be softened was discussed in Section 2.1.3. The MPC framework has a method that automatically softens variable bounds. The softening is done before the discretization, and will thus be described in continuous time. For each variable bound that is to be softened, a slack variable is added to the problem formulation. That means that if a variable z has both an upper limit z_U and a lower limit z_L, the same slack variable, z_{slack}, will be used when softening both bounds. The softening is done in four steps:

1. A new input, the slack variable z_{slack}, is added to the optimization problem. The slack variable is bounded to be larger than 0 and the nominal value is set to 0.0001 times the nominal value of the base variable. That is,

$$z_{\text{slack}} \geq 0 \tag{6}$$

$$z_{\text{slack, nominal}} = 0.0001 \cdot z_{\text{nominal}} \tag{7}$$

2. The slack variable times a constraint violation penalty P_z is added to the cost function. That is, the cost function $f(w)$ is changed to:

$$f(w) + P_z \cdot \int_{t_0}^{t_f} z_{\text{slack}}(t)dt \tag{8}$$

Once discretized, this formulation will be equivalent with adding the 1-norm of the slack variable times the constraint violation penalty.

3. The old variable bounds are transformed into path constraints on the form

$$z \leq z_U + z_{\text{slack}} \tag{9}$$

$$z \geq z_L - z_{\text{slack}} \tag{10}$$

These four steps are done for all variables that have bounds to be softened. If a variable has only either an upper or a lower bound, step 3 is modified accordingly. Ideally, if the initial condition has not violated the constraint, the slack variable should be zero or very close to zero at all times. However, choosing the nominal value of the slack variable has to be done with care to avoid numerical issues. This is why we have made the nominal value of the slack variable proportional to the nominal value of the base variable. The factor of 0.0001 included in the calculation of the slack nominal value was decided through testing.

3.3.2 Unsuccessful Optimization

Since finding a solution to the NLP is not guaranteed, it is important to have a fallback method in case of unsuccessful optimization. Using the MPC framework, if the solver terminates without finding a solution to the given problem the input returned will be the second input u_2 in the input sequence of the *previous* optimization (which was successful). If the next optimization after that is unsuccessful as well, the third input u_3 in the input sequence in the last successful optimization is returned, and so on. This way of returning optimal inputs from the last successful optimization continues until the solver finds a feasible solution again, or until there are no more values in the last successful optimization to return.

This is the default fallback method in case of unsuccessful optimizations the MPC class uses. However, it is straightforward to detect if an optimization was successful or not, so it is possible for the user to create a custom fallback method instead.

3.3.3 Next initial guess

Having a good initial guess for the optimization variables is important to decrease the risk of not finding a solution and to speed up the solution process. Defining a new initial guess of the optimization variables prior to each optimization is handled internally in the MPC framework. There are three different methods of computing the initial guess in the MPC framework:

1. *Extracting it from a result object*. This method uses the same methods that are used by the open-loop framework to extract an initial guess of the optimization variables from the trajectories of a result object. This method is quite time-consuming and requires that a result object, from which to extract the initial guess, is available.

2. *Shifting the result vector.* The NLP solver returns the solution of an optimization in one vector containing the value of each variable at each of the collocation points. The result vector is on the same form as the vector corresponding to the initial guess, but offset by one sample period in time. Looking at the result vector, this method discards all the values included in the first sample period and shifts the rest of the values to cover the voids. This means that all the values corresponding to the second sample period in the result vector will be shifted to the values corresponding to the first sample period in the initial guess vector. The values of the last sample period in the initial guess vector are all set to the value of the last collocation point from the result vector. On a uniform mesh, this method yields the same initial guess as method 1 and is less time-consuming.

3. *Using the result vector without shifting it*. This method sets the new initial guess to the result from the previous optimization directly, without shifting it. Using this method will yield the least accurate initial guess, since all values will be offset by one sample period in time, but it is the least time consuming method of the three.

The default method of computing the next initial guess in the MPC framework is method 2, mainly because it is faster than method 1 and yields a better initial guess than method 3.

3.4 Limitations

Because the MPC framework reuses the NLP for each optimization it is not as flexible as the open-loop framework. There are currently some collocation options that are not compatible or will not work as desired with the MPC framework and there are also some restrictions regarding the formulation of the optimal control problem. These are described in more detail in (Axelsson, 2015).

4 Results

4.1 Test setup

In this section we will evaluate the performance of the MPC framework through two different tests. The first test is to evaluate how the performance of the NLP solver is affected by the warm start options chosen. The second test is a benchmark where the aim is to compare the results of the MPC framework to the open-loop framework. For both tests we provide some or all of the following statistics:

DOI 10.3384/ecp15118301

- **Opt_{fail}**. The sample number of the optimizations which were unsuccessful. For IPOPT it is assumed that the return statuses 'Solve_Succeeded' and 'Solved_To_Acceptable_Level' denote a successful optimization. All other return statuses are regarded as unsuccessful.
- **Iterations**. The average number of iterations in IPOPT for one sample.
- **T_{pre}**. The average pre-processing time for one sample.
- **T_{sol}**. The average solution time in IPOPT for one sample.
- **T_{post}**. The average post-processing time for one sample.
- **T_{tot}**. The average total computation time for one sample.

All tests are run using the MA27 solver for IPOPT (HSL, 2013).

4.2 Test problem

The system we are going to evaluate the performance of the MPC framework on is a Combined-Cycle Power Plant (CCPP) (Casella et al., 2011), during start-up. The aim of the MPC controller is to take the system from an off state to full capacity. The plant is considered to be at full capacity once the evaporator pressure p has reached 8.35 MPa and the plant load *load* has reached 100%. During the start-up there is an upper bound on the thermal stress σ in the steam turbine, which may not exceed 260 MPa. The MPC framework will soften this bound automatically as discussed in section 3.3.1. We are going to extend the model with an integrator at the input by connecting the plant load with a new state variable u and thus having $\dot{u}$ as the input in the optimization problem. This yields a plant load which is piecewise linear, rather than piecewise constant. This also allows for setting variable bounds on $\dot{u}$. Variable bounds on u and $\dot{u}$ are

$$0 \leq u \leq 1,$$
$$0 \leq \dot{u} \leq 0.1/60.$$

The Optimica code for this system is presented below.

```
optimization Startup(objectiveIntegrand=((
    plant.p-8.35e6)/1e6)^2 + 0.5*(u-1)^2,
    startTime=0,finalTime=4000)

parameter Real sigma_max = 2.6e8;
CombinedCycle.Optimization.Plants.CC0D_WarmStartUp
     plant(sigma(max=sigma_max));
Modelica.Blocks.Interfaces.RealInput du(min=0,
    max=0.1/60);
RealConnector u(start=0.15,fixed=true,min=0,max
    =1);

equation
der(u) = du;
connect(u, plant.load);

end Startup;
```

On the first line, the keyword `objectiveIntegrand` is used to define the Lagrange part of the cost function, while `startTime` and `finalTime` denote the beginning and end of the prediction horizon. A model of the plant is instansiated, `plant`, and an upper variable bound is added to `sigma` using the keyword `max`. The following two lines show how to add variable bounds to the inputs, `u` and `du`, and how to connect them to the model. Additional constraints are not present in this example, but could be added under a new section started with the keyword `constraint`.

To emulate noise a normally distributed disturbance, with the mean 0 and the standard deviation 0.001 times the current state value, will be added at each sample point to all the states except for the extra state u.
With the addition of u as a state, and the extra input σ_{slack} which the MPC framework will add to the problem when softening the bound on σ, the resulting optimization problem has 10 states, 123 algebraic variables and 2 inputs. With the MPC and collocation options chosen, presented in Table 1, the resulting NLP has 4564 optimization variables after the discretization.

Table 1. The MPC and collocation options used for all tests on the CCPP system.

MPC options	value
Sample period	100 [s]
Prediction horizon	1000 [s]
Collocation options	**value**
n_e	10
n_c	3

4.3 Warm start test

The warm start test aims to evaluate whether we can improve the robustness and speed of the solver by providing an initial guess of the dual variables to the solver. The options we consider in IPOPT are 'warm_start_init_point', which indicates whether an initial guess of the dual vairables will be provided by the user or should be estimated by IPOPT, and 'mu_init', which is the initial value of the barrier parameter. For the cases where an initial guess of the dual variables will be provided, the guess will be the result from the previous optimization. Note that since there is no implemented support for shifting the dual variables yet, they will be

implicitly offset by one sample period in time. The result of this test is presented in Table 2.

Table 2. Summary of the results for the warm start test. The two leftmost columns define which options were used, while the other three are the results obtained. Warm start on means that an initial guess of the dual variables was provided to the solver while warm start off means that the solver estimated it's own initial guess for them.

Warm start	μ_{init}	Opt_{fail} $[k]$	Iterations $[nbr]$	T_{sol} $[s]$
Off	1e-1	-	32	0.282
Off	1e-2	-	34	0.289
Off	1e-3	5, 24	32	0.282
Off	1e-4	-	34	0.282
On	1e-1	-	31	0.266
On	1e-2	-	31	0.261
On	1e-3	-	32	0.262
On	1e-4	-	30	0.259

From the data in Table 2 it can be seen that providing an initial guess of the dual variables decreases both the number of iterations needed to find a solution and the solution time slightly. The robustness also seems to be improved since optimal solutions were found for all samples in the case where warm start was on, while two unsuccessful optimizations were noted when warm start was off. The best average solution time was in the case where warm start was on and $\mu_{\text{init}} = 10^{-4}$.

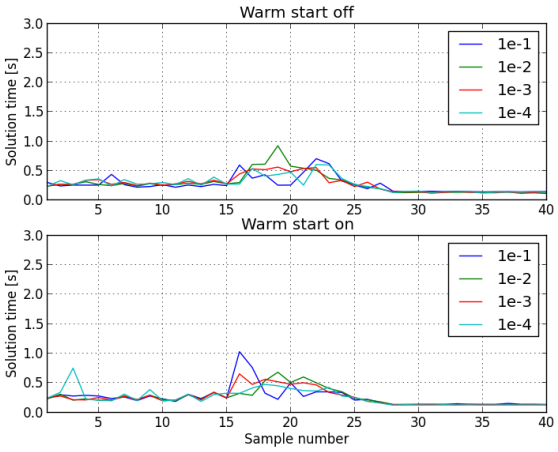

Figure 1. The solution time for each of the samples in the warm start test. The upper plot is for warm start being turned off and the μ_{init} values as specified by the legend and the lower plot is for warm start being turned on.

In Figure 1 the solution time for each of the samples is plotted for all the options tested. Since 8 different option combinations were tested, the results have been split into two separate plots, one where warm start is turned off and one where warm start is turned on. The barrier parameters impact on the solution time is especially noticable in the region between sample number 15 and 25, where the largest deviations are present. The overall conclusion from this test is that turning the warm start on i.e. providing an initial guess of the dual variables to the solver, has a positive effect on the solution time and robustness. This even though the dual variables provided are offset by one sample period in time. The gain of warm starting the solver might improve even more if a shift method for the dual variable was to be implemented.

4.4 Benchmark

In this section the results of using the MPC framework will be compared to the results of using the open-loop framework for an MPC setup. We will look specifically at the result trajectories as well as the different average times for one sample (pre-processing, solution, post-processing and total).

To get equivalent problem formulations the variable bound on σ is softened manually for the case where the open-loop framework is used. The manual softening is done in exactly the same way as the MPC framework does it. The collocation and MPC options chosen are the same in both cases and the resulting NLP:s shall thus be identical in both cases. For the case running with the MPC framework warm start of the solver is activated and the barrier parameter is set to $\mu_{\text{init}} = 10^{-4}$, since those were the options that gave the best results in the warm start test. The results are summarized in Table 3.

Table 3. Results from the benchmark of the CCPP system.

	Opt_{fail} [k]	T_{pre}, [s]	T_{sol}, [s]	T_{post}, [s]	T_{tot}, [s]
MPC fw.	-	0.053	0.267	0.012	0.332
OL fw.	12	0.901	0.295	0.044	1.241

From the data in Table 3 it can be concluded that using the MPC framework compared to using the open-loop framework has decreased the total average computation time by 70%. This is also clearly illustrated in the total computational time per sample plot in Figure 2. Looking closer at the average times we can conclude that the majority of the time saved is in the pre-processing step, which was what we had expected since the time consuming discretization has been moved outside the MPC loop. The post-processing time is also decreased due to the MPC framework not creating a result object after each optimization.

Figure 3 shows the CCPP system simulated with the optimal inputs obtained, where we can conclude that the results obtained in both cases are almost identical.

DOI 10.3384/ecp15118301

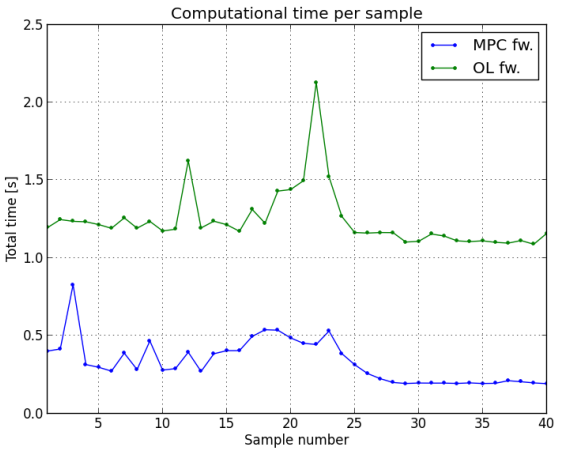

Figure 2. Total computation time for each sample in the benchmark, using the MPC framework and the open-loop framework respectively.

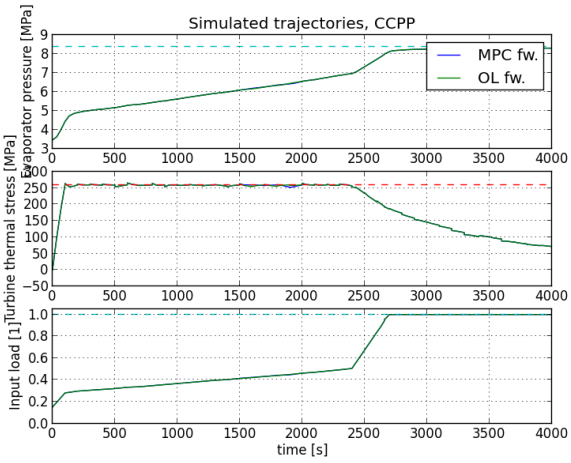

Figure 3. The CCPP system simulated with the optimal inputs obtained in both cases. The dashed cyan lines are the set points while the dashed red line is the variable bound.

5 Conclusions

This paper describes the implementation of a new MPC framework in JModelica.org, which significantly decreases the total average computation time of solving an NMPC optimal control problem. The main reason the computation time has been shortened is due to the MPC framework reusing the NLP for all optimizations, rather than creating a new NLP each time the optimal control problem is solved. For the benchmark presented in this paper, using the MPC framework compared to using the open-loop framework, the total average computation time went from 1.24 s to 0.33 s, a relative decrease of 70%. The benchmark also shows that the same results were obtained with both frameworks.

In addition to being faster than the open-loop framework, the MPC framework is also easier to use since a lot of things are handled internally. This includes the initial guess being set and the prediction horizon being shifted automatically as well as the built in fall back method in case of unsuccessful optimization and a method that automatically softens variable bounds.

Further work includes adding support for nominal trajectories and external data, two of the collocation options that do not work correctly when reusing the NLP. Nominal trajectories are used for scaling the optimization variables and external data could be used to define set point trajectories, rather than constant set points, for the controlled variables. A shift method for the dual variables could also be implemented to, hopefully, decrease the solution time in the solver further. The automatic softening of variable bounds method could be extended to support automatic softening of constraints as well as different softening schemes.

Acknowledgments

Fredrik Magnusson acknowledges support from the Swedish Research Council through the LCCC Linneaus Center and is also a member of the eLLIIT Excellence Center at Lund University.

References

"HSL. A collection of Fortran codes for large scale scientific computation.", 2013. URL `http://www.hsl.rl.ac.uk/`.

Johan Åkesson. Optimica—an extension of modelica supporting dynamic optimization. In *6th International Modelica Conference 2008*, 2008.

Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problems. *Computers and Chemical Engineering*, 34(11):1737–1749, November 2010.

Frank Allgöwer, Rolf Findeisen, and Zoltan K Nagy. Nonlinear model predictive control: From theory to application. *J. Chin. Inst. Chem. Engrs*, 35(3):299–315, 2004.

Joel Andersson. *A General-Purpose Software Framework for Dynamic Optimization*. PhD thesis, Arenberg Doctoral School, KU Leuven, October 2013.

Magdalena Axelsson. Nonlinear Model Predictive Control in JModelica.org. Master's thesis, Department of Automatic Control, Lund University, Sweden, August 2015.

Karl Berntorp and Fredrik Magnusson. Hierarchical predictive control for ground-vehicle manuevering. In *2015 American Control Conference*, 2015.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004. ISBN 0521833787.

Francesco Casella, Filippo Donida, and Johan Åkesson. Object-oriented modeling and optimal control: A case study in power plant start-up. In *18th IFAC World Congress*, 2011.

Mats Vande Cavey, Roel De Coninck, and Lieve Helsen. Setting up a framework for model predictive control with moving horizon state estimation using jmodelica. In *10th International Modelica Conference 2014*, 2014.

R. Franke, E. Arnold, and H. Linke. HQP: a solver for nonlinearly constrained large-scale optimization. URL `http://hqp.sourceforge.net/`.

Rüdiger Franke, Manfred Rode, and Klaus Krüger. On-line optimization of drum boiler startup. 2003.

Christian Hartlep and Toivo Henningsson. NMPC Application using JModelica.org: Features and Performance. In *11th International Modelica Conference 2015*, 2015.

B. Houska, H.J. Ferreau, and M. Diehl. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

Per-Ola Larsson, Francesco Casella, Fredrik Magnusson, Joel Andersson, Moritz Diehl, and Johan Å kesson. A framework for nonlinear model-predictive control using object-oriented modeling with a case study in power plant start-up. In *Computer Aided Control System Design (CACSD), 2013 IEEE Conference on*, 2013.

Tor Larsson. Moving Horizon Estimation in JModelica.org. Master's Thesis ISRN LUTFD2/TFRT--5982--SE, Department of Automatic Control, Lund University, Sweden, 2015.

Björn Lennernäs. A CasADi based toolchain for JModelica.org. Master's thesis, Department of Automatic Control, Lund University, Sweden, June 2013.

J.M. Maciejowski. *Predictive Control with Constraints*. Prentice-Hall, 2002.

Fredrik Magnusson and Johan Åkesson. Dynamic optimization in jmodelica.org. *Processes*, 3(2):471–496, 2015.

Andreas Wächter. Short tutorial: getting started with ipopt in 90 minutes. *Combinatorial Scientific Computing (U. Naumann, O. Schenk, HD Simon, eds.)*, 34(56):118, 2009.

Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57, 2006.

 DOI 10.3384/ecp15118301