Using Modelica models in real time dynamic optimization – gradient computation

Lars Imsland* Pål Kittilsen Tor Steinar Schei
Cybernetica AS
7038 Trondheim, Norway
{lars.imsland,pal.kittilsen,tor.s.schei}@cybernetica.no

Abstract

This paper reports on implementation of gradient computation for real-time dynamic optimization, where the dynamic models can be Modelica models. Analytical methods for gradient computation based on sensitivity integration is compared to finite difference-based methods. A case study reveals that analytical methods outperforms finite difference-methods as the number of inputs and/or input blocks increases.

Keywords: Nonlinear Model Predictive Control, Sequential Quadratic Programming, Gradient computation, Offshore Oil and Gas Production.

1 Introduction

Nonlinear model predictive control (NMPC) is an advanced control technology that enables the use of mechanistic multi-disciplinary process models in achieving process control objectives (economical, safety, environmental). NMPC algorithms formulate an 'open-loop' constrained dynamic optimization problem, which is re-solved and re-implemented at regular intervals to combine the advantage of the optimal control solution with the feedback achieved through updated information (measurements and estimated states and parameters). The number of applications of NMPC is increasing, especially within certain process industries [11, 4], but there is certainly potential for further growth.

Models developed primarily for dynamic process simulation and design are often not appropriate for NMPC, for example for reasons related to numerical robustness and computational speed. Nevertheless, issues such as modularity, reuse, model libraries, documentation, etc., make it advantageous to use advanced modeling languages such as Modelica also for development of prediction models for NMPC. This is discussed in [7].

NMPC optimization algorithms are often *Sequentially Quadratic Programming* (SQP) algorithms. That is, the NMPC nonlinear dynamic optimization problem is discretized and solved by applying quadratic programming sequentially to quadratic/linear approximations of the optimization problem, and upon numerical implementation the solution generally converges to a local optimum. SQP algorithms for NMPC are often classified based on how discretization is done, but nevertheless have in common that they need (at least) gradient information of the discretized dynamic optimization problem.

This paper is concerned with gradient computation for a class of NMPC optimization algorithms often referred to as sequential (or single-shooting) NMPC optimization, using Modelica models developed in Dymola as prediction models. Of particular concern is the exploitation of symbolic/analytical Jacobians of the Modelica model. Furthermore, we discuss briefly the implementation of gradient computation in commercial NMPC software such as Cybernetica Cenit [4]. Finally, we use a semi-realistic NMPC problem from offshore oil and gas production as a case to illustrate issues such as accuracy and efficiency in gradient computation.

*Also affiliated with the Norwegian University of Science and Technology, Department of Engineering Cybernetics.

DOI: 10.3384/ecp09430067

2 Nonlinear Model Predictive Control

2.1 Models

A model of the physical plant we want to control by NMPC is implemented in Modelica. The underlying mathematical representation could be either as (hybrid) ODE or DAEs, but here we assume, mainly for simplicity, that it is formulated as a (piecewise) continuous ODE:

$$\dot{x} = f(x,u,p),\ \ y = h(x,u,p),\ \ z = g(x,u,p), \quad (1)$$

where x are states, u are manipulated inputs, p are parameters (which might be candidates for online estimation), y are measured outputs and z are controlled (not necessarily measured) outputs.

Model-based control typically apply some estimation scheme, for instance a sigma-point extended Kalman-filter approach or moving horizon estimation [12]. The task of the estimation is to

- consolidate measurements to obtain a best estimate of the process state,
- estimate unknown and changing parameters (adaptation), and
- achieve zero steady state error in the desired controlled variables (integral control).

We will not be concerned about estimation in this paper, and assume therefore (unrealistically) that we know all parameters and measure the state.

Note that typically, we will in addition to manipulated inputs also have other (measured) inputs that in essence make the system time-variant. However, we will employ a discrete-time NMPC formulation and are therefore only interested in integration over one sample interval where these inputs typically are assumed constant.

For the same reason, we are only interested in the solution of (1) in the sense that it is used to calculate states and outputs at the next sampling instant. That is, we are interested in the discrete-time system

$$x_{k+1} = x_k + \int_{t_k}^{t_{k+1}} f(x(\tau), u_k, p_k)\mathrm{d}\tau, \quad (2a)$$

$$y_k = h(x_k, u_{k-1}, p_{k-1}), \quad (2b)$$

$$z_k = g(x_k, u_{k-1}, p_{k-1}). \quad (2c)$$

The integration involved is in general solved by ODE solver routines.

For NMPC, we will use the above system for prediction. In prediction for NMPC, we are not concerned with the measured outputs, and disregard these for now. With abuse of notation, we will write the time-varying discrete-time NMPC predictor system as

$$x_{k+1} = f_k(x_k, u_k), \quad z_k = g_k(x_k, u_{k-1}). \quad (3)$$

2.2 NMPC optimization problem

We formulate here a simplified discrete-time NMPC optimization problem using the model (3). We assume the desired operating point $(x,u) = (0,0)$ is an equilibrium ($f_k(0,0) = 0$, $g_k(0,0) = 0$), and we minimize at each sample (using present measured/estimated state x_0 as initial state for predictions) the objective function

$$J(x_0, u_0, u_1, \ldots, u_{N-1}) = \frac{1}{2}\sum_{i=0}^{N-1} z_{i+1}^{\mathsf{T}} Q z_{i+1} + u_i^{\mathsf{T}} R u_i$$

over future manipulated inputs u_i, where z_i are computed (predicted) from (3), and Q and R are weighting matrices. Importantly, the future behavior is optimized subject to constraints:

$$z_{\min} \le z_k = g_k(x_k, u_{k-1}) \le z_{\max}, \quad k = 1, \ldots, N$$
$$u_{\min} \le u_k \le u_{\max}, \quad k = 0, \ldots N-1.$$

The first input u_0 is then implemented to the plant.

It is important to note that the problem formulation used here is simplistic, For the sake of brevity and with little loss of generality, it does not contain features usually contained in NMPC software packages, such as:

- Features related to non-regulation problems (for instance control of batch processes).
- Input blocking (for efficiency).
- Incidence points (for efficiency and feasibility).
- Control horizon longer than input horizon.
- End-point terminal weight/region (in regulation, for efficiency/stability).
- Input moves instead of inputs as optimization variables.

Further details about such issues can be found in MPC textbooks, for instance [9].

2.3 Sequential NMPC optimization

In most cases, the (discretized) dynamic optimization problem is solved using numerical algorithms based on sequential quadratic programming (SQP). A SQP method is an iterative

method which at each iteration makes a quadratic approximation to the objective function and a linear approximation to the constraints, and solves a QP to find the search direction. Then a line-search is performed along this search direction to find the next iterate, and the process is repeated until convergence (or time has run out). General purpose SQP solvers may be applied to NMPC optimization, but it is in general advantageous to use tailor-made SQP algorithms for NMPC applications.

The main approaches found in the literature are usually categorized by how the dynamic optimization problem is discretized/parametrized. The most common method is perhaps the *sequential* approach [3], which at each iteration simulates the model using the current value of the optimization variables $(u_0, u_1, \ldots, u_{N-1})$ to obtain the gradient of the objective function (and possibly the Hessian), thus effectively removing the model equality constraints and the states $x_1, x_2, \ldots, x_N$ as optimization variables. Other methods are the *simultaneous* approach [1], and the *multiple shooting* approach [2]. In this paper, a sequential approach is taken.

3 Gradient computation in sequential NMPC optimization

3.1 The sensitivity (step/impulse response) matrix

As explained above, sequential SQP approaches to NMPC optimization sequentially simulates and optimizes. The simulation part should calculate the objective function- and constraint gradients with respect to the optimization variables u_i. This is typically done via the *step response matrix*, or as in the simplified exposition here, the impulse response matrix. To avoid confusion, we will mostly refer to this in the following as the (NMPC) sensitivity matrix.

Rewrite the objective function by stacking future inputs and outputs as

$$J(x_0, \mathbf{u}) = \frac{1}{2}\left(\mathbf{z}^\mathsf{T}\mathbf{Q}\mathbf{z} + \mathbf{u}^\mathsf{T}\mathbf{R}\mathbf{u}\right),$$

where $\mathbf{u} = (u_0, u_1, \ldots, u_{N-1})$, $\mathbf{z} = (z_1, \ldots, z_N)$, $\mathbf{Q} = \text{blkdiag}\{Q\}$ and $\mathbf{R} = \text{blkdiag}\{R\}$. The gradient of the objective function is

$$\frac{\partial J}{\partial \mathbf{u}} = \mathbf{z}^\mathsf{T}\mathbf{Q}\frac{\partial \mathbf{z}}{\partial \mathbf{u}} + \mathbf{u}^\mathsf{T}\mathbf{R}.$$

Similarly, the linearization of the output constraints are also given by the matrix $\Phi = \frac{\partial \mathbf{z}}{\partial \mathbf{u}}$, which we call the sensitivity matrix (which in this case is the truncated impulse response matrix).

That is, once we have calculated the sensitivity matrix Φ, we can easily evaluate the objective function- and constraints gradients in sequential NMPC optimization. From this, one can argue that gradient computation in sequential NMPC is mostly about efficient computation of the sensitivity matrix.

The rest of this section treats calculation of the sensitivity matrix by finite differences, and by forward ODE sensitivity analysis. We remark that one could also use adjoint sensitivity methods for calculating the desired NMPC gradients [8]. However, for NMPC problems with a significant number of constraints, this is likely to be less efficient than forward methods.

3.2 NMPC sensitivity matrix by finite differences

Finding the sensitivity matrix by finite differences is achieved by in turn perturbing each element of all input vectors u_i and simulate to find the response in the z_js. The perturbation is typically either one-sided (forward finite differences) or two-sided (central finite differences), the latter taking about twice the time but being somewhat more accurate [10].

3.3 NMPC sensitivity matrix by sensitivity integration

Assuming we have a time-varying linearization of (3) along the trajectories:

$$x_{k+1} = A_k x_k + B_k u_k, \tag{4a}$$
$$z_k = C_k x_k + D_k u_{k-1}, \tag{4b}$$

we can calculate the sensitivity matrix which in this simple case is as shown in eq. (5) on the bottom of the following page. (The sensitivity matrix shown there is the impulse response matrix, the step response matrix is the cumulative sum of the columns of the impulse response matrix, from right to left.) To find the linearization (4), it is usually most practical to calculate sensitivities of the solution of (1) with respect to initial values $x(0) = x_k$ and inputs u_k (assumed constant over each sample interval). Stacking these sensitivities

in matrices S and W, they are given by the following matrix ODEs [5]:

$$S := \frac{\partial x}{\partial u_k}: \quad \dot{S} = \frac{\partial f}{\partial x} S + \frac{\partial f}{\partial u}, \qquad S(0) = 0, \quad (6a)$$

$$W := \frac{\partial x}{\partial x_k}: \quad \dot{W} = \frac{\partial f}{\partial x} W, \qquad W(0) = I. \quad (6b)$$

We will only be interested in the sensitivities at the end of each sample interval, which are the system matrices in (4a):

$$B_k := \frac{\partial x_{k+1}}{\partial u_k} = S_{k+1}, \;\; A_k := \frac{\partial x_{k+1}}{\partial x_k} = W_{k+1}. \quad (7)$$

Defining also $C_k := \frac{\partial z_k}{\partial x_k}$, $D_k := \frac{\partial z_k}{\partial u_{k-1}}$, we get the linearized (LTV) system above. The required Jacobian matrices

$$\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial u}, \quad \frac{\partial g}{\partial x}, \quad \frac{\partial g}{\partial u}$$

can be found from finite differences, symbolically (for instance from a Dymola model), or by automatic differentiation methods. The two latter should be preferred.

The sensitivity ODEs (6) are solved together with (2a) by ODE solvers. Although the size of S and W might be large, the fact that the systems (6) have a block-diagonal Jacobian with the individual blocks being the Jacobian of (2a) can and should be exploited in ODE solvers, leading to efficient computation of the ODE sensitivities [6]. (For systems with a very large number of states and relatively few inputs/outputs, it might be more efficient to directly integrate the elements in the sensitivity matrix, and thus avoiding calculation of state sensitivities.)

It is important to note that the above is described for zero order hold and no input blocking. For efficient implementation, it is essential to exploit input blocking in the sensitivity computations.

4 Implementation

A software package for model-based estimation and control (NMPC) will typically include an *offline* part for model fitting (parameter optimization) to data, data-based testing of estimation and simulation-based testing of NMPC (including estimation), and an *online* part for a complete NMPC real-time solution (including estimation). The workflow in taking a parametrized model (implemented in Modelica/Dymola, or 'by hand' in lower level languages such as C) to online application is attempted illustrated in Figure 1. A Modelica tool (such as Dymola) will need a method for exporting the models so they can be used efficiently in the offline tool and the online system. Dymola has the option of C-code export, which is platform independent and gives models that are efficiently evaluated and easily integrated with ODE/DAE-solvers and optimizers, typically implemented in C.

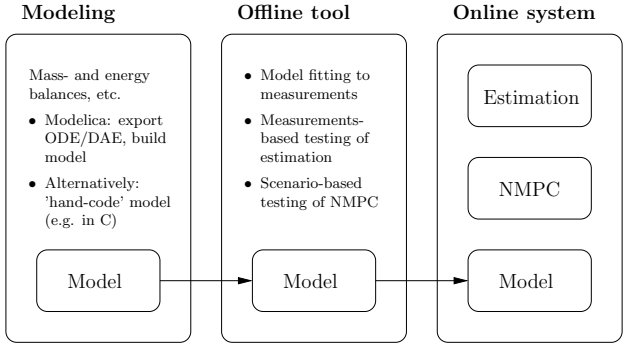

Figure 1: Overview over workflow in model usage. Data storage, data acquisition/exchange, and GUI not shown.

NMPC software that should use analytical methods for gradient computation, need the discrete-time model to provide A_k, B_k, C_k and D_k matrices (4) (typically found via sensitivity integration using the exported model, as discussed earlier). Figure 2 indicates the data flow in a discretized model component that is based on a

$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{pmatrix} = \begin{pmatrix} C_1B_0 + D_1 & 0 & 0 & 0 & \cdots \\ C_2A_1B_0 & C_2B_1 + D_2 & 0 & 0 & \cdots \\ C_3A_2A_1B_0 & C_3A_2B_1 & C_3B_2 + D_3 & 0 & \cdots \\ C_4A_3A_2A_1B_0 & C_4A_3A_2B_1 & C_4A_3B_2 & C_4B_3 + D_4 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{pmatrix} \quad (5)$$

continuous-time ODE Modelica/Dymola model, using an ODE solver which implements sensitivity integration.

Several specialized solvers for calculation of ODE sensitivities exist. For instance, CVODES [6] implements variable-order, variable-step multistep ODE solvers for stiff and non-stiff systems, with sensitivity analysis capabilities. For efficiency of sensitivity integration, it is a significant advantage if we have symbolic ODE Jacobians available, for instance from a Dymola model.

As a side-remark, for models that do not provide symbolic Jacobians, we have the option of using automatic differentiation packages (CppAD, ADOLC, or others). However, this typically requires C++ compilation.

5 Case

5.1 Case description

In the North Sea (and on other continental shelves), petroleum is produced by drilling wells into the ocean bed. From the wells, typically a stream of oil, gas and water arrives at a surface production facility (platform or ship) which main task is to separate the products. Oil and gas are exported, either through pipelines or by ship. Water is cleaned and deposited to sea or pumped back to the reservoir.

A schematic picture (in the form of a Dymola screendump) of such an offshore oil and gas processing plant is given in Figure 3. Oil, gas and water enter the plant from several sources. In reality the sources are reservoirs connected to the production facility through wells and pipelines. The separators are large tanks which split the phases oil, water and gas. The produced oil is leaving in the lower right corner of the figure, while the gas enters a compression train (not included in the model) from the first and second separator (two leftmost tanks). Water is taken off from each separator and sent to a water treatment process.

Generally, this type of process is a fairly complex system in terms of numbers of components. However, many of the components are of the same type (mainly separators, compressors, valves, controllers, in addition to minor components such as sources, sinks, splitters, sensors, etc.), which simplifies overall modeling and make it efficient to reuse model components. Furthermore, construction of this process model benefited significantly from using models and concepts introduced by the new Modelica_Fluid library. Models for valves, sources, sinks, and sensors were used directly, whereas other models and functions in the new library inspired the development of our own models. For real-time efficiency reasons, we take care to ensure that we end up with an ODE-type model. The new *stream* class is an improvement to ease the construction of models satisfying this criteria.

In addition to the unit modes, medium models are necessary in order to calculate physical properties like density and heat capacity, in addition to phase transitions between oil and gas. The model should have real time capabilities, favoring simple/explicit relations. For phase equilibrium calculations, correlations of k-values (as function of temperature, pressure and molecular weight) were used together with a simplified representation of the many chemical species found in the real process. Gas density was described by a second-order virial equation, where the model coefficients were fitted to an SRK-equation for the relevant gas composition evaluated for the temperature and pressure range of current interest.

The model we use as NMPC prediction model (cf. Figure 3) has 27 states. We consider two NMPC problem formulations, one being 2×2 (2 MVs and 2 CVs), the other 4×4.

5.2 Issues in preparing a Dymola model for use with a NMPC system

Before a Dymola simulation model can be used in a NMPC system, it must be prepared. It is worthwhile to mention some of the issues involved:

- In addition to making sure that the model does not contain nonlinear systems of equations that must be solved to evaluate it (i.e, the model is an ODE as explained above), for Dymola to export symbolic Jacobians we must of course ensure that the model is differentiable. Especially if Modelica functions are used extensively, this might in some cases involve some effort.
- Dymola provides ODE-style Jacobians (A, B, C and D matrices). Unfortunately, it seems not possible to specify a subset of inputs that we want to evaluate Jacobians with respect to. This is especially critical for the B-matrix. For instance, we have in the case study a total of 9 possible MVs/DVs, all of

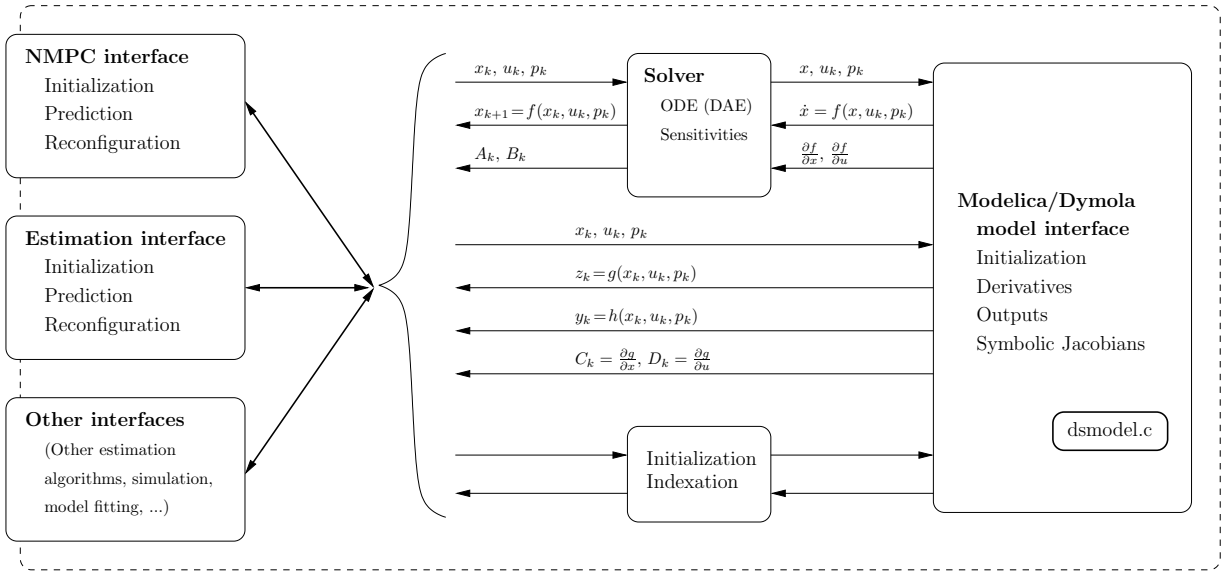

Figure 2: Overview over model component data flow.

which are modeled as Dymola inputs. We have chosen to control 2 or 4 of these. This means that we evaluate a B-matrix of dimension 27×9, instead of 27×2 or 27×4. This incurs considerable unnecessary complexity. If we in addition have a considerable number of parameters to be estimated also modeled as Dymola inputs, this makes the situation even worse.

- It seems the most natural way to implement communication between a NMPC system and the model, is to use 'top level' Modelica inputs and outputs. This is usually rather straightforward to implement for NMPC inputs and outputs (MVs, DVs and CVs) and measurements, but not very flexible: The Dymola C-code model interface could have been more sophisticated when it comes to identification and indexing of inputs, outputs and states. Furthermore, the use of top-level inputs and outputs can become rather awkward when it comes to model parameters that should be estimated.

6 Gradient computations applied to case

This section aims to illustrate the advantages and differences between finite differences and sensitivity-based sensitivity computations. Strictly speaking, the results only apply to this specific case, but we believe there is some generality in the trends reported. Issues that will be discussed, are computational complexity (timing), accuracy, and implementational aspects. The results are of course influenced by many factors not investigated (i.e., kept constant) here, as for example number of states, stiffness, exact definition of input blocks, etc.

We use a Matlab interface to the NMPC system, and choose to compare computational complexity by measuring execution time in Matlab. Although this has some drawbacks, it should give a fairly accurate picture of the relative performance. To increase the reliability, for each recording of execution time we run 5 consecutive identical NMPC scenarios (with significant excitation), and record the smallest execution time. This execution time includes the Kalman filter and NMPC optimization, but as the gradient computation is the most computationally expensive part, and the other parts are independent of choice of method for gradient computation, the difference in execution times should give a fairly good estimate of the difference in complexity of gradient computation.

6.1 Correctness and accuracy

It is clear that using finite difference (from now on FD) and analytic sensitivity methods based on sensitivity integration (AS) should give the same gradients "in the limit" (of perturbation-size and integration tolerances). Nevertheless, it is of interest to test this, also to get a feel for how large errors (or differences) relaxed integration tolerances

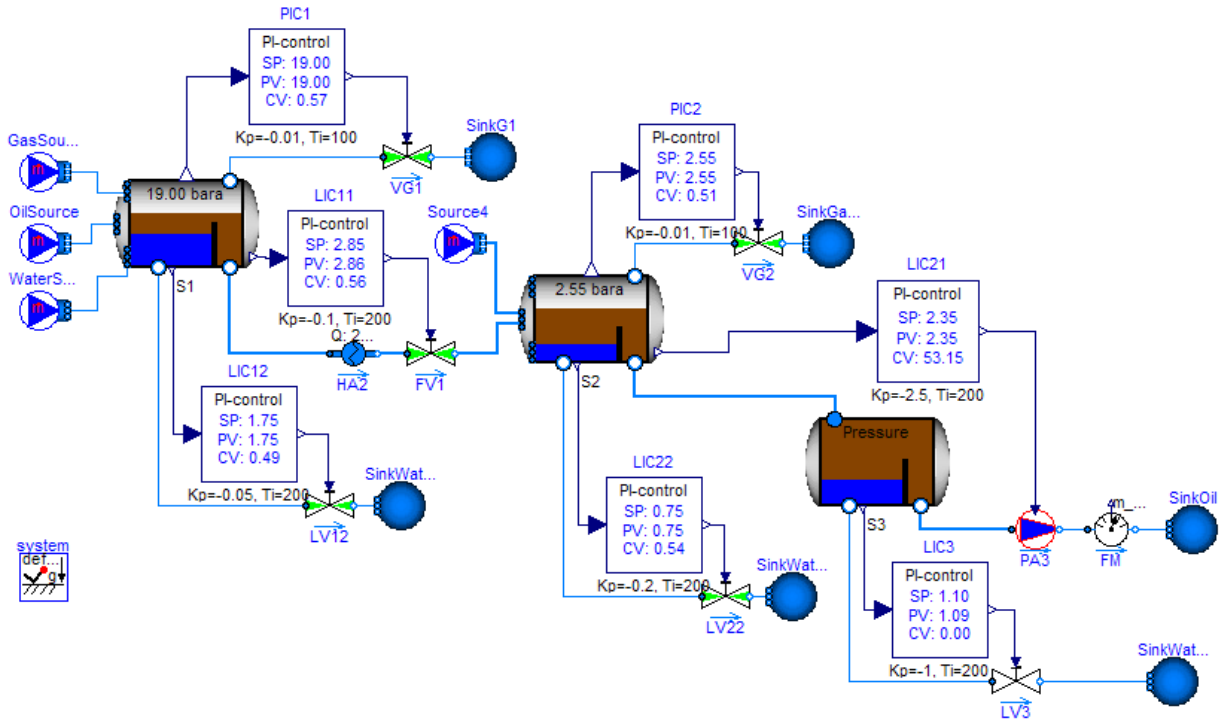

Figure 3: Overview of an offshore oil and gas processing plant, as implemented in Dymola.

and realistic perturbations will lead to.

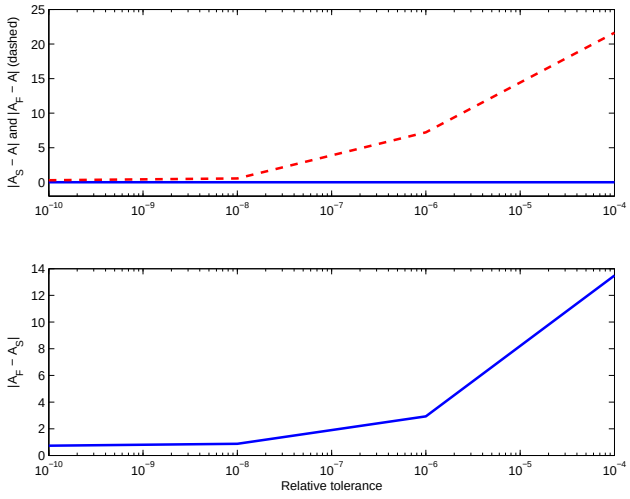

Figure 4: Top: Difference between 'real' (as found by AS methods with tolerances of 1e-12) step response matrix A and step response matrices for higher tolerances. Bottom: Step response matrices using AS and AF (almost) converge as tolerances decreases.

From Figure 4, we see that for small integration error tolerances, the gradients found are fairly correct in both methods, but the error in the FD sensitivity matrix increases much faster as the error tolerances are increased. An important note regarding the implementation of the AS-method is that we have chosen to have error control on both states *and* sensitivities, not merely states. In our experience, this can be essential to ensure accurately enough sensitivities when using AS.

We do not discuss here the choice of perturbation size in FD methods, as this does not differ from the general discussion in e.g. [10]. Suffice it to say that the general trends in Figure 4 are fairly independent of choice of perturbation size.

It is notable that even though both methods only give correct gradients in the limit, FD methods gives a direct approximation to the gradient of the objective function that is actually being optimized (including integration errors). This can in theory be an advantage in the line-search step of SQP algorithms.

6.2 Computational complexity

In this section we will compare the computational cost of different gradient computations as the number of NMPC degrees of freedom increases. We increase the degrees of freedom both in number of input blocks as well as number of inputs. The cases we will compare, are gradient computation using finite differences (FD) with or without using symbolic Jacobians in the ODE solver, and analytical gradient computation using sensi-

tivity integration (AS), also with and without using symbolic Jacobians.

We use the same ODE tolerances in all cases, and for sensitivity integration the sensitivities are included in the error control. The relative time usage for different number of inputs and input blocks are shown in Figure 5. In Figure 6, the experiment is repeated for $N_u = 2$, but without using symbolic Jacobians in the ODE solver.

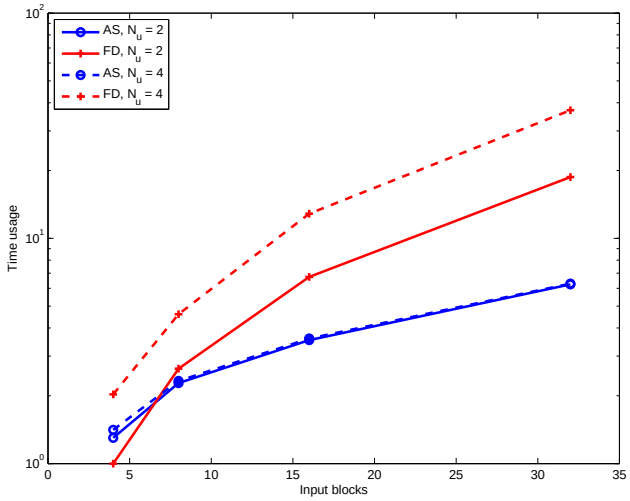

Figure 5: Relative time usage for different number of inputs and input blocks.

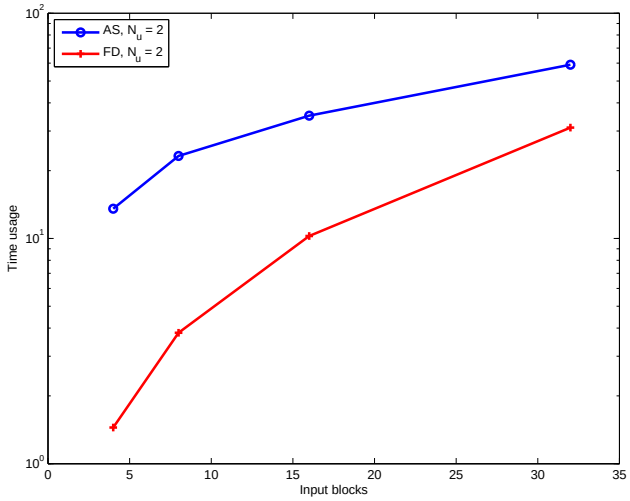

Figure 6: Relative time usage for different number of inputs and input blocks, without exporting symbolic Jacobians from Dymola.

The following observations are made:

- FD grows approximately quadratically in input blocks (as additional input blocks incurs both additional perturbations and ODE solver resetting), while AS grows approximately linearly in input blocks (incurs only additional ODE solver resetting). Note the logarithmic scale in Figure 5. To the extent that this is general, this means that AS will always outperform FD when many input blocks are used.
- Increasing number of input blocks (leads to more frequent ODE solver resetting) is more expensive than increasing number of inputs (leads to larger "sensitivity state") when using AS. We attribute this both to the efficiency of CVODES in exploiting the structure in the sensitivity equations, but also to the next issue:
- Increasing number of inputs does not significantly increase complexity of AS. This may be surprising, but can be explained in this case by the fact that all ODE Jacobians are calculated irrespectively of how many of the inputs are actually active, as discussed in Section 5.2. If Dymola allowed calculation of only those Jacobians that are needed, this could considerably speed up execution time.
- AS suffers significantly more than FD from not having symbolical Jacobians available.

Finally, we mention that in our experience, implementing Modelica-functions in C can significantly speed up FD (not done in this case), see also [7]. If this can be combined with export of symbolic Jacobians, it can also speed up AS, but to a much less extent. In other words, implementing Modelica-functions in C is less important when using AS.

7 Concluding remarks

To construct the sensitivity matrix using analytical methods becomes significantly faster than finite difference-based methods as the number of inputs and/or input blocks increases. Therefore, this technology is expected to be important in NMPC systems for 'larger' (in a NMPC context) models. Furthermore, we expect that many of these 'larger' models will be implemented in high-level languages (Modelica) rather than in lower-level languages (C), due to issues like reuse and modularity, but also due to availability of symbolic Jacobians.

Moreover, we conclude that to use analytical methods, we should have ODE Jacobians available. Dymola provides these for us, but it seems

we do not have the opportunity to evaluate 'partial' Jacobians, which would be a significant advantage.

We hope we have provided an incentive for developers of other Modelica-tools to generate symbolic Jacobians. Using automatic differentiation may also be possible in cases with C-code export-type functionality, but this could have some other drawbacks.

Acknowledgments

The authors thank StatoilHydro for providing information and input to the case study. Trond Tollefsen is acknowledged for his contributions to the development of the model library CyberneticaLib and particularly for the implementation of the case study model.

References

[1] L. T. Biegler, A. M. Cervantes, and A. Wächter. Advances in simultaneous strategies for dynamic process optimization. *Chem. Eng. Sci.*, 57:575–593, 2002.

[2] H. G. Bock, M. Diehl, D. B. Leineweber, and J. P. Schlöder. A direct multiple shooting method for real-time optimization of nonlinear DAE processes. In F. Allgöwer and A. Zheng, editors, *Nonlinear Predictive Control*, volume 26 of *Progress in Systems Theory*, pages 246–267. Birkhäuser, Basel, 2000.

[3] N. M. C. de Oliveira and L. T. Biegler. An extension of newton-type algorithms for nonlinear process control. *Automatica*, 31:281–286, 1995.

[4] B. A. Foss and T. S. Schei. Putting nonlinear model predictive control into use. In *Assessment and Future Directions Nonlinear Model Predictive Control*, LNCIS 358, pages 407–417. Springer Verlag, 2007.

[5] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I – Nonstiff problems*. Springer-Verlag, 2nd edition, 1993.

[6] A. C. Hindmarsh and R. Serban. *User Documentation for CVODES v2.5.0*. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2006.

[7] L. Imsland, P. Kittilsen, and T. S. Schei. Model-based optimizing control and estimation using modelica models. In *Proc. of Modelica'2008*, Bielefeld, Germany, 2008.

[8] J. B. Jørgensen. Adjoint sensitivity results for predictive control, state- and parameter-estimation with nonlinear models. In *Proceedings of the European Control Conference, Kos, Greece*, 2007.

[9] J. M. Maciejowski. *Predictive Control with Constraints*. Prentice-Hall, 2001.

[10] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 2006.

[11] S. J. Qin and T. A. Badgwell. A survey of industrial model predictive control technology. *Control Engineering Practice*, 11:733–764, 2003.

[12] T. S. Schei. On-line estimation for process control and optimization applications. *Journal of Process Control*, 18:821–828, 2008.

