Efficient Implementation of Collocation Methods for Optimization using OpenModelica and ADOL-C

Vitalij Ruge Willi Braun Bernhard Bachmann
{vitalij.ruge, willi.braun, bernhard.bachmann}@fh-bielefeld.de
Bielefeld University of Applied Sciences, Department of Mathematics and Engineering,
Bielefeld, Germany

Andrea Walther Kshitij Kulshreshtha
andrea.walther@uni-paderborn.de kshitij@math.upb.de
Universität Paderborn, Institut für Mathematik,
Paderborn, Germany

Abstract

Efficient calculation of the solutions of nonlinear optimal control problems (NOCPs) is becoming more and more important for today's control engineers. The systems to be controlled are typically described using differential-algebraic equations (DAEs), which can be conveniently formulated in Modelica. In addition, the corresponding optimization problem can be expressed using Optimica.
Solution algorithms based on collocation methods are highly suitable for discretizing the underlying dynamic model formulation. Thereafter, the corresponding discretized optimization problem can be solved, e.g. by the interior-point optimizer Ipopt. The performance of the optimizer heavily depends on the availability of derivative information for the underlying optimization problem. Typically, the gradient of the objective function, the Jacobian of the DAEs as well as the Hessian matrix of the corresponding Lagrangian formulation need to be determined. If only some or none of these derivatives are provided, usually numerical approximations are used by the optimizer internally.
OpenModelica supports the Optimica language and is capable of automatically generating the discretized optimization problem using collocation methods as well as the whole symbolic machinery available. In addition, all necessary derivative information is determined using the automatic differentiation capabilities of ADOL-C, which has now been integrated into the OpenModelica environment.
Keywords: Modelica; optimization; automatic differentiation; collocation; OpenModelica; ADOL-C

1 Introduction

The aim of this paper is to describe an efficient new solution process implemented in OpenModelica [11] for nonlinear optimal control problems. This effort continues the development of the collocation approach already discussed in [3], which has been successfully tested using the algorithmic differentiation tool CasADi [18]. Several enhancements, e.g. special treatment of the first collocation interval, integration of the automatic differentiation tool ADOL-C, as well as efficient and stable calculation of all derivative information, have been realized in OpenModelica and are demonstrated within this paper.
Efficient calculation of first order derivatives is possible with OpenModelica based on symbolic differentiation and has been successfully demonstrated using real world problems in [5]. This calculation of the derivatives benefits on the one hand from the simplification of expressions and on the other hand from the code, which is efficiently generated by OpenModelica. For optimization purposes the second order derivatives are important as well, since most of the optimization algorithms rely on them, e.g. Ipopt [19], which is used in this work. Second order derivatives are currently not symbolically available in OpenModelica, but could be provided numerically based on the already mentioned first order derivatives.
Another possibility is using the automatic differentiation tool ADOL-C, which is capable of working directly with the generated code of OpenModelica and has already been used successfully with Ipopt. Moreover, ADOL-C comes with a lot of additional features, e.g. efficient calculation of derivatives of different or-

ders. Last but not least, the implementation leads to a very fast solution with little memory costs for the underlying NOCP. The modeling and problem description is done in Modelica [8] extended with the optimization objective functions specified in Optimica [1]. This paper is organized as follows. In section 2, the mathematical representation of the nonlinear optimal control problem is discussed. The main idea of discretizing the NOCP based on orthogonal collocation principles is described in section 3. The efficient realization of the derivative calculation using ADOL-C is demonstrated in section 4. Section 5 presents the implementation details with respect to scaling, initialization and derivative calculation. Finally, the performance of the newly developed tool chain is discussed in section 6. The paper concludes the work with final remarks and suggestions for future work.

2 Nonlinear Optimal Control Problem (NOCP)

In many applications the NOCP is described by the following mathematical representation [10]:

$$\min_{u(t)} \quad \mathrm{J}(x(t),u(t),t) = E(x(t_f),u(t_f),t_f) + \int_{t_0}^{t_f} L(x(t),u(t),t)\, dt \quad (2.1)$$

s.t.

$$x(t_0) = x_0 \quad (2.2)$$

$$\dot{x}(t) = f(x(t),u(t),t) \quad (2.3)$$

$$\hat{g}(x(t),u(t),t) \leq 0 \quad (2.4)$$

$$r(x(t_f)) = 0 \quad (2.5)$$

where $x(t) = \left[x^{(1)}(t),\ldots,x^{(n_x)}(t)\right]^\top$ and $u(t) = \left[u^{(1)}(t),\ldots,u^{(n_u)}(t)\right]^\top$ are the state vector and control variable vector for $t \in [t_0,t_f]$, respectively. The constraints (2.2), (2.3), (2.4) and (2.5) represent the initial conditions, the nonlinear dynamic model description based on differential algebraic equations (DAEs), the path constraints $\hat{g}(x(t),u(t),t) \in \mathbb{R}^{n_{\hat{g}}}$ and the terminal constraints [3]. With respect to the implementation in Ipopt and the Modelica language, it is appropriate to split the box constraints from $\hat{g}(x(t),u(t),t) \leq 0$, i.e.

$$\begin{array}{ccccc} x_{\min} & \leq & x(t) & \leq & x_{\max} \\ u_{\min} & \leq & u(t) & \leq & u_{\max} \end{array}$$

and to introduce so-called slack variables for the rest

$$g(x(t),u(t),t) + s(t) = 0$$

with $s(t) \geq 0 \in \mathbb{R}^{n_g}$. Therefore, it is possible to use the attributes `min` and `max` already available in Modelica for the description[9].

Modelica model description

The mathematical representation of a Modelica model is typically given by DAEs

$$F(x(t),u(t),y(t),t) = 0.$$

However, most Modelica tools, especially OpenModelica, are capable of converting this formulation (by means of the so-called BLT transformation [2]) into a semi-explicit ODE form as formulated also in (2.3)

$$\begin{aligned} \dot{x}(t) =&\ f(x(t),u(t),t), \\ y(t) =&\ h(x(t),u(t),t). \end{aligned}$$

In general, there is no closed expression for the functions f and g, but rather, iterative techniques, e.g., Newton's method, are employed to solve the so-called occurrent linear or nonlinear algebraic loops [2].
At this point it is possible to choose between two strategies for the discretization of (2.3). On the one hand, it is feasible to transform these algebraic loops into residual form and to add them subsequently to the discretized NOCP formulation. This approach has the advantage that the costs for solving the algebraic loop in each evaluation of the function f are saved. However, the solver space as well as the workload of the optimizer increases. On the other hand, the algebraic loop can be solved during each optimizer step by a linear or nonlinear solver, depending on the type of problem. This procedure might have the drawback, that the solution of these algebraic loops might be also quite time consuming, especially in the case of a highly nonlinear equation system.
This paper focuses on the second strategy. The semi-explicit ODE form is generated using OpenModelica and solved for each optimizer step. In addition, a more general NOCP formulation can be converted by means of the BLT transformation to a semi-explicit NOCP as stated in (2.1), (2.2), (2.3), (2.4) and (2.5).

3 Collocation

Previous work [3] has shown that solving the NOCP with a collocation approach is efficient. Therefore, the choice of collocation nodes is important, since that influences the stability and order of the integration method [7, 16]. The RADAU IIA method [4] is

DOI 10.3384/ECP140961017

one possible method to choose the collocation nodes. RADAU IIA is an implicit Runge-Kutta method and is typically described in the form [16]:

$$\begin{bmatrix} x_{i,1}-x_i \\ \vdots \\ x_{i,m}-x_i \end{bmatrix} = \Delta t_i \cdot (A \otimes I) \cdot \begin{bmatrix} f(x_{i,1},u_{i,1},t_{i,1}) \\ \vdots \\ f(x_{i,m},u_{i,m},t_{i,m}) \end{bmatrix} \quad (3.1)$$

with $x_{i,j} = x(t_{i,j})$, $x_i := x(t_i)$, $u_{i,j} = u(t_{i,j})$, $t_{i,j} := t_i + \tau_j \cdot \Delta t_i$ and $t_i := t_0 + \sum_{l=1}^{i} \cdot \Delta t_l$ where Δt_i, $i = 0,\dots,n$ is the length of a subinterval and $\tau_j \in [0,1]$, $j = 1,\dots,m$ are the collocation nodes. A and I are the Butcher and identity matrix, respectively. If $\det(A) \neq 0$, equation (3.1) can be transformed [7] to the following form

$$F_i(.) := \left(A^{-1} \otimes I\right) \cdot \begin{bmatrix} x_{i,1}-x_i \\ \vdots \\ x_{i,m}-x_i \end{bmatrix} - \Delta t_i \cdot \begin{bmatrix} f(x_{i,1},u_{i,1},t_{i,1}) \\ \vdots \\ f(x_{i,m},u_{i,m},t_{i,m}) \end{bmatrix}$$

This form leads to a sparse structure for the Jacobian matrix, since in equation (3.1) the sparse structure is destroyed by multiplication of the dense matrix A.
The RADAU IIA can be interpreted as a Lagrange interpolation of the state $x^{(l)}(t)$ [3, 4]

$$x^{(l)}(t_i + \tau \cdot \Delta t_i) \approx \sum_{j=1}^{m} p_j(\tau) \cdot x_{i,j}^{(l)} \text{ for } \tau \in [0,1].$$

The corresponding Lagrangian polynomials are p_j. In order to handle constraints for `der(u)` the control variable $u^{(l)}(t)$ needs to be interpreted as a Lagrange interpolation as well:

$$u^{(l)}(t_i + \tau \cdot \Delta t_i) \approx \sum_{j=1}^{m} p_j(\tau) \cdot u_{i,j}^{(l)}.$$

Therefore, it is possible to calculate `der(u)` as

$$\frac{du^{(l)}(t)}{dt} = \frac{\partial u^{(l)}(t_i + \tau \cdot \Delta t_i)}{\partial \tau} \cdot \Delta t_i \approx \Delta t_i \cdot \sum_{j=1}^{m} \frac{\partial p_j}{\partial \tau}(\tau) \cdot u_{i,j}^{(l)}.$$

Moreover, the constraints based on RADAU IIA result in an unbounded expression $f(x_0,u_0,t_0)$, since the node $\tau_j = 0$ is not part of the RADAU IIA integration scheme, especially the expression $u(t_0)$ is unbounded. The principle is visualized in figure 1. This issue can be addressed by using the LOBATTO IIIA method, which includes the nodes $\tau_1 = 0$ and $\tau_m = 1$. Thus, the LOBATTO IIIA method yields an influence of $u(t_0)$ on the NOCP. Therefore, the principles of the collocation discretization will be applied not only to states, but also to the control variables. This approach is referred to as total collocation [3, 18].

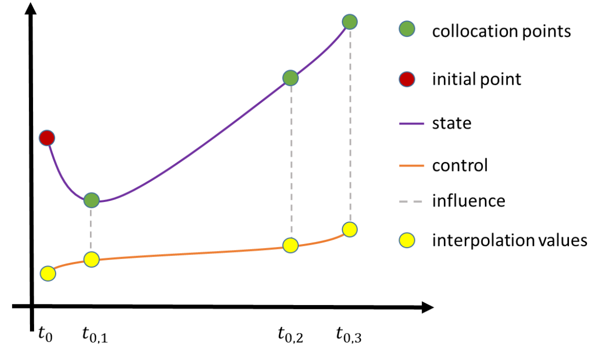

Figure 1: RADAU IIA schema

3.1 Lobatto IIIA

In comparison to RADAU IIA the LOBATTO IIIA method has a singular matrix A [16]. The number of nonzero elements of the Jacobian matrix can be reduced by multiplying the matrix A with B^{-1} so that

$$\left[0 \mid B^{-1}\right] \cdot A = \left[0 \mid B^{-1}\right] \cdot \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,m} \end{bmatrix}$$
$$= \left[0 \mid B^{-1}\right] \cdot \begin{bmatrix} 0 & 0 \\ A_1 & B \end{bmatrix} = \left[\hat{A}_1 \mid I\right].$$

Furthermore, the equation (3.1) can be transform for the special case LOBATTO IIIA as follows

$$\hat{F}_0(.) := \left(B^{-1} \otimes I\right) \cdot \begin{bmatrix} x_{0,2}-x_0 \\ \vdots \\ x_{0,m}-x_0 \end{bmatrix} -$$
$$(A_1 \otimes I) \cdot \Delta t_0 \cdot \begin{bmatrix} f(x_{0,1},u_{0,1},t_{0,1}) \\ \vdots \\ f(x_{0,1},u_{0,1},t_{0,1}) \end{bmatrix} + \quad (3.2)$$
$$\Delta t_0 \cdot \begin{bmatrix} f(x_{0,2},u_{0,2},t_{0,2}) \\ \vdots \\ f(x_{0,m},u_{0,m},t_{0,m}) \end{bmatrix}$$

Note that x_0 is bounded and solved with equation (2.2), so x_0 is known for the optimization process. Thus, there is no need to differentiate equation (3.2) with respect to x_0, this results only in nonzero elements for u_0.
Besides, for RADAU IIA and LOBATTO IIIA applies

$$x_0 = x_{\,0,1} \quad \text{and} \quad x_{\,i+1} = x_{\,i,m},$$

which is solved symbolically without the optimization loop.

3.2 Discretized Lagrange term

Now, it is possible to use the property of the collocation method that

$$x(t_{i,j}) \approx x_{i,j}$$

for the approximation of the Lagrange term (2.1) based on quadrature formulas. Obviously, it makes sense to apply methods of Lobatto and Radau quadrature.

$$\Phi(\mathtt{x},\ \mathfrak{u},\ \mathfrak{t}) := \Delta t_0 \cdot \sum_{j=1}^{m} \hat{w}_j \cdot L_{0,j} + \sum_{i=1}^{n-1} \Delta t_i \cdot \sum_{j=1}^{m} w_j \cdot L_{i,j} \approx \int_{t_0}^{t_f} L(x(t),u(t),t)\ dt \tag{3.3}$$

where $L_{i,j} := L(x_{i,j}, u_{i,j}, t_{i,j})$, $\hat{w}$ represents the Lobatto weights, w the Radau weights, and the abbreviations $\mathtt{x} := [x_{0,1},\ldots,x_{n,m}]$, $\mathfrak{u} := [u_{0,1},\ldots,u_{n,m}]$, and $\mathfrak{t} := [t_{0,1},\ldots,t_{n,m}]$.

3.3 Discretized NOCP

Finally, the NOCP can be discretized:

$$\min\ \mathtt{J}(\mathtt{x},\mathfrak{u},\mathfrak{s},\mathfrak{t}) = E(x_{m,n-1},u_{m,n-1},t_{m,n-1}) + \Phi(\mathtt{x},\mathfrak{u},\mathfrak{t})$$

s.t.

$$\begin{array}{ccccc} c(\mathtt{x},\mathfrak{u},\mathfrak{s},\mathfrak{t}) & \overset{!}{=} & 0 & & \\ u_{\max} & \leq & \mathfrak{u} & \leq & u_{\min} \\ x_{\max} & \leq & \mathtt{x} & \leq & x_{\min} \\ 0 & \leq & \mathfrak{s} & & \end{array}$$

where

$$c(\mathtt{x},\mathfrak{u},\mathfrak{s},\mathfrak{t}) := \begin{bmatrix} \hat{F}_0(.) \\ g(x_{0,1},u_{0,1},t_{0,1}) + s_{0,1} \\ \vdots \\ g(x_{0,m},u_{0,m},t_{0,m}) + s_{0,m} \\ F_1(.) \\ g(x_{1,1},u_{1,1},t_{1,1}) + s_{1,1} \\ \vdots \\ g(x_{1,m},u_{1,m},t_{1,m}) + s_{1,1} \\ F_n(.) \\ g(x_{n,1},u_{n,1},t_{n,1}) + s_{n,1} \\ \vdots \\ g(x_{n,m},u_{n,m},t_{n,m}) + s_{n,m} \\ r(x_{n,m},u_{n,m},t_{n,m}) \end{bmatrix}$$

and $x_{0,1} = x_0$ with $\mathfrak{s} = [s_{0,1},\ldots,s_{m,n}]$, $s_{i,j} = s(t_{i,j})$.

3.4 Nonlinear optimization

Now, the original NOCP is transformed to a nonlinear optimization problem, where the optimizer needs to find the optimal discretized control vector $\mathfrak{u}$ and to adapt $\mathtt{x},\mathfrak{s}$ so that the constraints are fulfilled. For this operation the optimizer requires the first order derivatives from $E(.)$, $\Phi(.)$ and $c(.)$ as well as the second order derivatives from the Lagrangian function

$$L(z,\lambda,\mathfrak{t}) = E(.) + \Phi(.) + \lambda^\top \cdot c(.) \tag{3.4}$$

to find the solution. The sorting of $c(.)$ and $z = [\mathtt{x},\mathfrak{u},\mathfrak{s}]$ is substantial for a good Jacobian- and Hessian-structure. The block

$$G_i(.) := \begin{bmatrix} F_i(.) \\ g(x_{i,1},u_{i,1},t_{i,1}) + s_{i,1} \\ \vdots \\ g(x_{i,m},u_{i,m},t_{i,m}) + s_{i,m} \end{bmatrix}$$

can be sorted more efficiently, if this is investigated in more detail. Furthermore, it applies

$$\frac{\partial x_{i,j}^{(l)}}{\partial x_{a,b}^{(c)}} = \frac{\partial x_{i,j}^{(l)}}{\partial u_{a,b}^{(d)}} = \frac{\partial u_{i,j}^{(e)}}{\partial u_{a,b}^{(d)}} = \frac{\partial u_{i,j}^{(e)}}{\partial x_{a,b}^{(c)}} = 0$$

for $i,a = 0,\ldots,n,\ \ j,b = 1\ldots,m,\ \ l,c = 1\ldots n_x,$ $e,d = 1,\ldots,n_u$ and $l \neq c,\ e \neq d$ as well as

$$\frac{\partial x_{i,j}^{(l)}}{\partial x_{i,j}^{(l)}} = 1 = \frac{\partial u_{i,j}^{(e)}}{\partial u_{i,j}^{(e)}}$$

Therefore, the Jacobian and Hessian matrices become very sparse. Furthermore, it should be taken advantage of the cyclic structure in $c(.)$, which results from the same structure in $G_1(.),\ldots,G_n(.)$.

4 Derivatives

There are at least three different ways to compute the derivative information required by a calculus-based optimization approach: Finite Differences, Symbolic Differentiation and Algorithmic Differentiation. The first technique, i.e., finite differences (FD), is based on the Taylor expansion and yields to relatively imprecise derivative information. Furthermore, the resulting computational cost is high in comparison to the two other approaches. For example, the gradient of a scalar-valued function with N input variables is approximated using FD with $N+1$ function evaluations. For these reasons, FD approximations of derivatives

DOI 10.3384/ECP140961017

will not be considered in this section. Alternatively, one may analytically derive expressions to evaluate the exact derivative based on the obtained formula. This can be done by hand, which results in an error-prone process, or automatically as provided, e.g., by Maple. Natural a Modelica compiler like OpenModelica has also capabilities to differentiate symbolically a Modelica model (see [5], [2]).

This purely symbolic method usually yields a very efficient way to compute first-order derivatives for closed-form expressions. However, the computation of higher-order derivatives or the handling of iterative solution procedures still form major challenges for this approach. The usage of algorithmic differentiation (AD), also called automatic differentiation, offers a third alternative to compute gradients, Jacobians and/or Hessians required for optimization. Based on the exploitation of the chain rule, AD provides derivative information of arbitrary order within working accuracy for a function $F : \mathbb{R}^N \rightarrow \mathbb{R}^M$ evaluated in a code segment on a computer [14]. The complexity estimates for the two basic approaches of AD, namely the forward mode and the reverse mode, are based on the operation count O_F, i.e., the number of floating point operations required to evaluate $y = F(z)$. Using the forward mode, one computes the required derivatives together with the function evaluation in one sweep. This approach yields one Jacobian-vector product $\nabla F v, v \in \mathbb{R}^N$ for no more than 2.5 times O_F. One vector-Jacobian product, or equivalently $\nabla F^\top w, w \in \mathbb{R}^M$, is obtained using the reverse mode in its basic form also for no more than four times O_F. If $M = 1$, i.e. ∇F corresponds to the gradient of a scalar-valued function, this complexity bound for the reverse mode is completely independent of the number n of input variables. Therefore, it is also known as the *cheap gradient result*. More details about AD can be found in the books [15] and [17] as well as on the web-page www.autodiff.org.

4.1 Efficient Jacobian evaluation

For the examples considered here, the Jacobian of the equality constraints $c : \mathbb{R}^N \rightarrow \mathbb{R}^N$ is an almost square matrix of dimension N, where $N = (n+1) \cdot m \cdot (n_x + n_g)$. Hence, as a first approach to evaluate the full Jacobian, one may compute the Jacobian-vector products

$$\frac{\partial c}{\partial z_i}(z) = \nabla c(z) e_i, \quad i = 1 \ldots, N,$$

where e_i denotes the ith unit vector, yielding the N rows of the Jacobian using either the symbolic approach or the forward mode of AD N times. For the forward mode of AD, the following theoretical bound of the computational cost to evaluate the full Jacobian can be shown [15]

$$\mathrm{OPS}(\nabla c(z)) \leq 2.5 N O_F,$$

where $\mathrm{OPS}(f)$ denotes the number of floating point operations required to evaluate f. To reduce this operation count, one may use the so-called vector forward mode, where not only one derivative information is propagated with the function evaluation but a bundle of p directional derivatives. Hence, for a so-called seed matrix $\Sigma \in \mathbb{R}^{N\times p}$ this variant of AD yields the Jacobian-matrix product $\nabla c(z)\Sigma \in \mathbb{R}^{N\times p}$ at a computational cost that can be bounded above by

$$\mathrm{OPS}(\nabla c(z)\Sigma) \leq (1 + 1.5p) O_F,$$

see [15]. Using $\Sigma = I_N$ as the identity matrix in $\mathbb{R}^{N\times N}$, one obtains the full Jacobian with the vector forward mode of AD at a computational cost bounded above by $(1 + 1.5N)O_F$ instead of $2.5NO_F$ when using the standard forward mode of AD. This makes a significant difference if N is large or the function evaluation is costly.

4.2 Exploiting the structure of the Jacobian

The derivative computation described so far completely ignores any structure within the Jacobian matrix. However, for the target applications of this research project, the Jacobian of the equality constraints has a block structure as shown in Fig. 2 for the Van der Pol oscillator. When computing the full Jacobian using

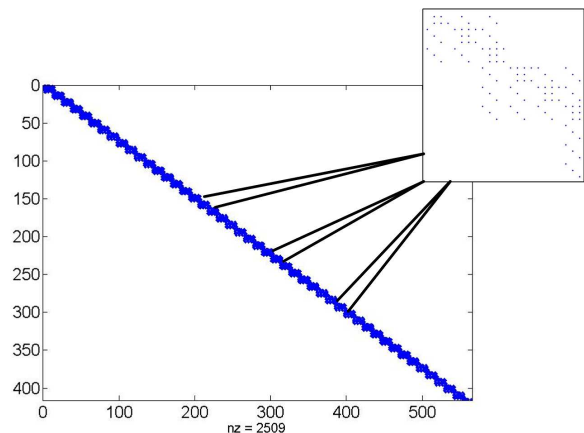

Figure 2: Van der Pol oscillator: Jacobian structure with 50 subintervals

the approach as explained in the last subsection, a lot of zero entries are computed, despite the fact that one knows that these entries are zero. To exploit the inherent structure of a sparse Jacobian, so-called compression techniques were developed. In the general case all compression techniques rely on the following four-step procedure: First determine the *sparsity structure* of the Jacobian $\nabla c(z)$. Second, obtain a suitable seed matrix Σ that defines a column partition of the Jacobian using, for example, a specialized *coloring* on the adjacency graph of the Jacobian. Then compute the *compressed* Jacobian matrix $B = \nabla c(z)\Sigma$. Finally, recover the numerical values of the entries of $\nabla c(z)$ from B. The sparsity structure may be known from the application, as can be seen from the block structure in the example of the Van der Pol oscillator, or determined by a suitable variant of AD as explained in [15]. In our applications the structure is known apriori so the first two steps may be skipped and a seed matrix is available directly from the block structure. Appropriate coloring methods with the corresponding recovery strategies for the general case are discussed for example in [12]. The compressed Jacobian B is evaluated using the vector forward mode of AD. For our applications, one may consider also the sparsity structure within the blocks of the Jacobian to reduce the computational cost even further. This will be the subject of future work.

4.3 Exploiting the structure of the Hessian

Using a combination of the forward mode and the reverse mode of AD, one can compute Hessian-vector products for a function $F(z)$ for a computational cost not larger than ten times O_F [15]. To exploit this facility to the full extend, for the target applications of this research project once more the sparsity of the Hessian can be taken into account. This is due to the fact that these derivative matrices have also a block structure as illustrated again for the Van der Pol oscillator in Fig. 3. The four step procedure explained above has to be adapted appropriately for the computation of second-order information in the general case. The sparsity structure of the Hessian may be known from the application, as is the case for the application discussed here, or it may be determined by a suitable variant of AD as described for example in [20]. For the general case corresponding coloring approaches together with suitable recovery strategies are presented in [13]. In our applications the seed matrix is obtained directly from the block structure known apriori.

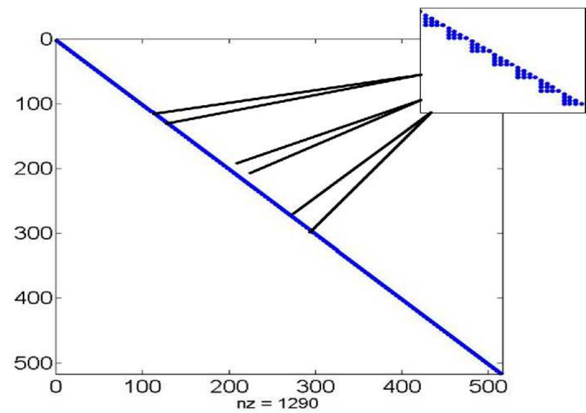

Figure 3: Van der Pol oscillator: Hessian structure with 50 subintervals

5 Implementation Details

The rough principle of the implementation is visualized in figure 4. At the first step the optimizer requires a sufficiently good starting point. In order to keep the constraint error for equation (2.3) small the method in OpenModelica creates a starting solution based on a simulation run. The initial guess of the control variable is set constant with the value of the `start` attribute. Obviously, a good setting of the initial values of the control variables can accelerate the NOCP solution process. Furthermore, this implementation supports the attribute `nominal` in Modelica for scaling variables and constraints. The effects will be presented in section 6.1.

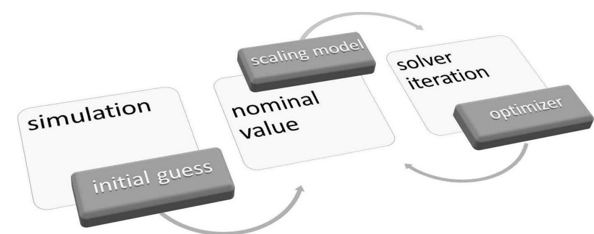

Figure 4: Implementation details

Coupling of OpenModelica and ADOL-C

The AD tool ADOL-C [21] uses the technique of operator overloading provided by the C++ standard to implement a wide variety of AD-based techniques. Within the research project described in this paper, the C code generation of OpenModelica was adapted such that ADOL-C can be used to evaluate the blocks in the Jacobian of the equality constraints and the blocks of the Hessian of the Lagrangian for a class of generic test problems. That is, the block structure was ex-

DOI 10.3384/ECP140961017

ploited manually. To compute the required derivative information the standard drivers jacobian(...) and hessian(...) of ADOL-C were used. For the applications considered here, the jacobian(...) routine uses the vector forward mode as described above. The current coupling of OpenModelica and ADOL-C allows a flexible choice between the symbolic derivative computation already implemented in OpenModelica and the AD-based derivative computation provided by ADOL-C. The structure of the coupling is illustrated in Fig. 5.

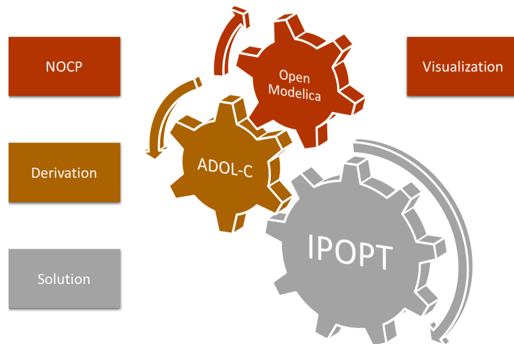

Figure 5: Structure of the coupling

6 Modelica Application and Performance Measurements

6.1 Model formulation

Currently, the user can influence the solution convergence by using native Modelica attributes like `nominal` and `start`. It should be emphasized that it is not the natural way to use `start`, since this attribute is usually reserved for initial values at start time for the simulation. Nevertheless, the initial trajectory of all problem variables is provided by a simulation, where all control variables are kept constant equal to the value of the `start` attribute. The described formulation of scaling and start trajectory will be shown on a simple model, based on the Batch Reactor found in [4]. The model was modified so that the states have the `nominal` values 10^{10} and 10^{-10}. The mathematical formulation is given by

$$\min_{u(t)} x_2(t_f)$$

s.t.

$$\begin{aligned} x_2(t) &= 10^{10} \cdot y_2(t) \\ x_1(t) &= 10^{-10} \cdot y_1(t) \\ 10^{-10} \cdot \dot{y}_1(t) &= -\left(u(t) + \frac{u(t)^2}{2}\right) \cdot x_1(t) \\ 10^{10} \cdot \dot{y}_2(t) &= u(t) \cdot x_1(t) \end{aligned}$$

$$u(t) \in [0,\ 5],\ y_1(0) = 1,\ y_2(0) = 0,\ t_f = 1$$

which can easily be formulated in a Modelica/Optimica representation:

```
optimization BatchReactor(
                    objective = cost(finalTime),
                    finalTime = 1)
  Real cost = -x2;
/*DAE Modelica*/
  /*states */
  Real y1(start=1e10,
          fixed=true, nominal=1e10);
  Real y2(start=0,
          fixed=true, nominal=1e-10);
  /* tuner */
  input Real u(min=0, max=5, start=1.0);
protected
  Real x1;
  Real x2;
equation
  x1 = 1e-10*y1;
  x2 = 1e10*y2;
  1e-10*der(y1) = -(u+u^2/2)*x1;
  1e10*der(y2) = u*x1;
end BatchReactor;
```

When setting the correct values for the `nominal` attribute the solution is calculated as expected. Setting the `nominal` attribute to 1 yields the wrong result, nevertheless the optimizer finishes without any error detection.

6.2 Combined Cycle Power Plant

A more industry-relevant benchmark is a model of a combined cycle power plant model, see figure 6. The model contains equation-based implementations of the thermodynamic functions for water and steam, which in turn are used in the components corresponding to pipes and the boiler. The model also contains components for the economizer, the super heater, as well as the gas and steam turbines. The model has one input, 10 states, and 131 equations. For additional details on the model, see [6].

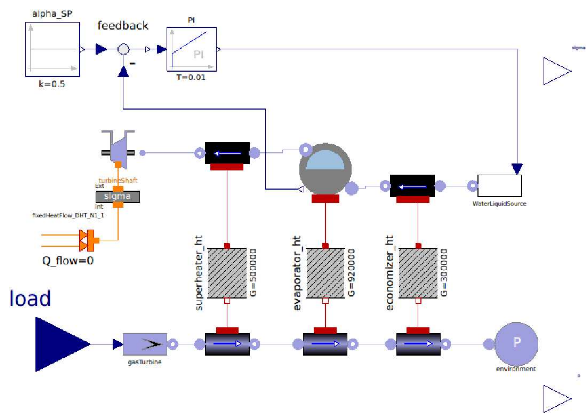

Figure 6: CombinedCycle display with OMEdit

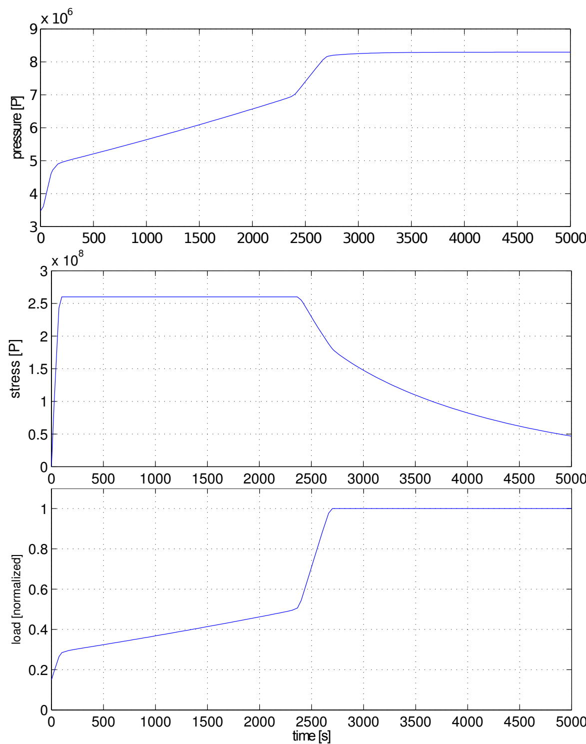

Figure 7: Optimal start-up trajectories. The upper curve shows the pressure in the evaporator, the middle curve shows the thermal stress in the steam turbine shaft and the lower curve shows the control input represented by the load.

The optimization problem is set up to use 50 collocation points that results in 1651 variables for the NOCP and was solved on a PC with a 3.2GHz Intel(R) Core(TM) i7. The algorithm requires an initial trajectory of all problem variables, which is provided by a simulation where the rate of change of the gas turbine load is set to a constant value. The optimization results are shown in figure 7 and 8 and correspond with the results that are discussed in detail in [6]. Here, the trajectories are smoother, and the performance has been improved substantially.

options		iteration	time [s]
Jacobian	Hessian		
ADOL-C	ADOL-C	39	2.29472
ADOL-C	BFGS	48	0.86425
OMC	BFGS	48	0.88558

Table 1: Time measurements of the solving process.

In table 1 the time measurements of the solving process are summarized for different options of derivatives calculation. One can see that the solution with ADOL-C needs less iterations, which is a strong indication that the solution is more accurate and more

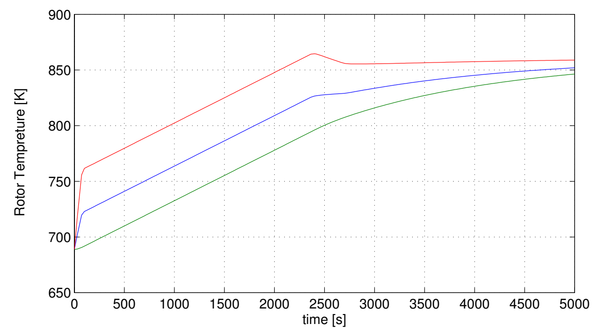

Figure 8: Optimal start-up trajectories. The upper curve shows the live steam temperature, the middle and low curves show the turbine rotor surface and mean temperatures.

stable. This is even more important for stiff models. However, the calculation of the Hessian with ADOL-C need currently a factor of three more computational time. Alternative approaches for a further improvement of the runtime needed for the Hessian calculation are the subject of current research.

7 Conclusions

This paper presents a newly developed tool chain for solving nonlinear optimization control problems. The underlying dynamic model formulation is done in Modelica and Optimica. The demonstrated solution method is based on orthogonal collocation methods, whereby the first interval is specially treated in order to consider control variables and their derivatives also at the initial time point. The derivative information is derived using the automatic differentiation tool ADOL-C, which efficiently calculates the corresponding Jacobian and Hessian matrices for the discretized optimization problem. Special treatments of the matrices with the focus on yielding optimal sparsity patterns with respect to block and cyclic structure are performed. The resulting optimization process proves to be stable and efficient.

8 Acknowledgments

This work has been partially supported by the German Ministry BMBF (BMBF Förderkennzeichen: 01IS12022I) in the ITEA2 MODRIO project. The authors also would like to acknowledge the kind support from Francesco Casella, who provided the power plant model used for benchmarks. The Open Source Modelica Consortium supports the OpenModelica work.

References

[1] J. Åkesson. *Languages and Tools for Optimization of Large-Scale Systems*. PhD thesis, Regler, nov 2007.

[2] J. Åkesson, W. Braun, P. Lindholm, and B. Bachmann. Generation of sparse jacobians for the function mock-up interface 2.0. In *Proceedings of the 9th International Modelica Conference*, 2012.

[3] B. Bachmann, L. Ochel, V. Ruge, M. Gebremedhin, P. Fritzson, V. Nezhadali, L. Eriksson, and M. Sivertsson. Parallel multiple-shooting and collocation optimization with openmodelica. In *Proceedings of the 9th International Modelica Conference*, 2012.

[4] L.T. Biegler. *Nonlinear programming. Concepts, algorithms, and applications in chemical processes*. SIAM, 2010.

[5] W. Braun, S. Gallardo-Yances, B. Bachmann, and K. Link. Fast simulation of fluid models with colored jacobians. In *Proceedings of the 9th International Modelica Conference*, 2012.

[6] F. Casella, F. Donida, and J. Åkesson. Object-oriented modeling and optimal control: A case study in power plant start-up. In *Proceedings of the 8th IFAC World Congress*, Milano, Italy, 2011.

[7] P. Deuflhard and F.A. Bornemann. *Numerische Mathematik 2: Gewöhnliche Differentialgleichungen*. De Gruyter Lehrbuch Series. Walter De Gruyter Incorporated, 2008.

[8] H. Elmqvist, S. E. Mattsson, and M. Otter. Modelica - a language for physical system modeling, visualization and interaction. *1999 IEEE Symposium on Computer-Aided Control System Design*, 1999.

[9] R. Franke. Formulation of dynamic optimization problems using modelica and their efficient solution. In *Proceedings of the 2th International Modelica Conference*, pages 315 –323, 2002.

[10] T.L. Friesz. *Dynamic optimization and differential games*. Springer, 2010.

[11] P. Fritzson, P. Aronsson, H. Lundvall, K. Nyström, A. Pop, L. Saldamli, and D. Broman. Openmodelica - a free open-source environment for system modeling, simulation, and teaching. In *IEEE International Symposium on Computer-Aided Control Systems Design,2006*, pages 1588 –1595, oct. 2006.

[12] A.H. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.

[13] A.H. Gebremedhin, A. Tarafdar, F. Manne, and A. Pothen. New acyclic and star coloring algorithms with application to computing Hessians. *SIAM J. Sci. Comput.*, 29:1042–1072, 2007.

[14] A. Griewank, K. Kulshreshtha, and A. Walther. On the numerical stability of algorithmic differentiation. *Computing*, 94(2-4):125–149, 2012.

[15] A. Griewank and A. Walther. *Principles and Techniques of Algorithmic Differentiation, Second Edition*. SIAM, 2008.

[16] E. Hairer and G. Wanner. *Solving ordinary differential equations. II: Stiff and differential-algebraic problems*. Springer, 2010.

[17] U. Naumann. *The art of differentiating computer programs. An introduction to algorithmic differentiation*. SIAM, 2012.

[18] A. Shitahun, V. Ruge, M. Gebremedhin, B. Bachmann, L. Eriksson, J. Andersson, M. Diehl, and P. Fritzson. Model-based dynamic optimization with openmodelica and casadi. In *Proceedings of the 7th IFAC Symposium on Advances in Automotive Control*, pages 446–451, 2013.

[19] A. Wächter and L. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, 2006.

[20] A. Walther. Computing sparse Hessians with automatic differentiation. *ACM Trans. Math. Softw.*, 34(1), 2008. Paper 3.

[21] A. Walther and A. Griewank. Getting started with ADOL-C. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, pages 181–202. Chapman-Hall, 2012. see also http://www.coin-or.org/projects/ADOL-C.xml.