Solvers chaining in the IDOS server for dynamic optimization

Radoslaw Pytlak¹, Jacek Blaszczyk², Andrzej Karbowski³, Kamil Krawczyk² and Tomasz Tarnawski¹

Abstract—This paper presents a newly deployed server, IDOS (Interactive Dynamic Optimization Server), devoted to solving optimal control problems. Development and deployment of the Interactive Dynamic Optimization Server is a result of a project funded by NCBiR (National Center for Research and Development in Poland). The aim of the project was to develop a prototype, online-accessible environment for solving dynamic optimization problems. Within the project we also constructed a modeling language (Dynamic Optimization Modeling Language, DOML) for defining optimal control problems. In result, a user can describe his problem in a programming-language-independent way. Then, once defined, the problem can be attempted by different solvers. In particular one can get a crude approximation to a solution of his problem by applying one solver and then continue solving the problem by another solver which gives a more accurate solution but may require a better initial guess than the first solver. The paper describes some constructs of DOML language which enable such chaining of solvers. A simple optimal control problem is used to illustrate the functionality of the IDOS server and the completeness of our language DOML.

I. INTRODUCTION

The idea of online optimization servers is not new and perhaps best incarnated in the form of the well known NEOS server, operational for about 16 years now ([8]). However, to our best knowledge, there are no servers that could be used to solve optimal control problems. It seems that such a server could serve well scientific community. First of all, solving an optimal control problem is much more difficult than solving a static optimization problem. One difficulty follows from the fact that equality constraints are in the form of system (differential) equations which may require numerical procedures for the their evaluation with varying stepsizes. Furthermore, optimality conditions for optimal control problems are much more complex especially when these problems have state constraints. Therefore, constructing a universal solver for dynamic optimization problems is a difficult task. Moreover, dynamic optimization problems are not so widespread as nonlinear programming problems and the consequence of that is that dynamic optimization solvers are not so widely accessible as optimization solvers for nonlinear programming problems are. As a result solving optimal control problems typically requires the user to code his problem directly in a programming language and tie it with particular numerical libraries. There are a few exceptions to this rule in which the user defines optimal control problems in a more programming-language-independent manner, most notably Optimica (discussed later on as the basis for DOML) and BOCOP ([11]). The aim of the IDOS project was to build a computing environment in which the user would have access to a variety of dynamic optimization solvers through a dedicated programming language. Since a typical dynamic optimization solver depends on many numerical libraries for performing linear algebra calculations, integration of system and adjoint equations, automatic differentiation or optimization iterations the administrator of the server would also take care of updating these libraries to guarantee latest (and thus with fewer bugs) versions of solvers accessible through the server.

But the most convincing argument for building a server for dynamic optimization problems is the observation that there is no universal method for solving numerically optimal control problems. One can use methods based on the maximum principle (strong, or weak) but they require (if state and terminal constraints are present in a problem formulation) a deep insight into necessary optimality conditions of the problem. But that insight can only be realized if one has at least an approximate solution to the problem which would indicate, for example, time intervals at which state constraints are active, or time intervals with singular arcs. Such approximate solutions could be provided by direct methods based on an a priori discretization of system’s equations which in turn can fail in finding accurate solutions. Therefore, very often advocated strategy is to start with a direct method, or method referring to adjoint equations in order to approximate solution and then switch to a shooting method to improve the approximation. The fact that this strategy is rarely used in practice (since these methods require substantially different data input formats which are very time consuming for use) persuaded us to construct the IDOS server. As a result, having one data input format for all optimal control solvers, and having one online available server with numerous different solvers deployed should ultimately enable to apply the strategy which we call solvers chaining.

The solvers chaining can be realized in many ways. For example, the user of the IDOS server can

• use one solver to get an approximation solution to his problem and then use the resulting control as an initial guess in another solver which essentially requires the same data input as the first solver—in this case we need the facility in the DOML language for providing time
varying initial control guesses;
• apply two consecutive solvers which use essentially different data inputs—in that case the data required by the second solver must be generated by the first solver;
• solve the 'simplified' version of his problem and then continue solving the problem by the second solver which treats the solution of first solver as its 'warm' starting point—in that case the second solver must be equipped with the 'warm' start strategies.

The paper is devoted to the second solvers chaining case. It is described by referring to the situation when the first solver based on a direct method provides an input data to the second solver which is based on a shooting method. A shooting method, when applied to an optimal control problem, requires a variety of data input such as:

• initial guesses for state trajectories;
• initial guesses for adjoint trajectories;
• time intervals of active constraints;
• time intervals of singular arcs;
• touching points of constraints;
• jumps in Lagrange’s multipliers associated with state constraints
• control law resulting from the Hamiltonian minimization.

The list shows the complexity of the task which must be undertaken to achieve that kind of solvers chaining. On one hand the first solver should be able to provide all listed information at its solution, on the other hand the DOML language must handle the information for the second solver.

The paper is organized as follows. In Section II we briefly describe the IDOS environment which is mainly composed of several dynamic optimization solvers and the application for serving submission calls (the server is available at http://idos.mchtr.pw.edu.pl). In the description of the server we concentrate on presenting types of the problems which can be handled by the server equipped with the solvers which can be applied to these types of problems. We do not outline how server handles jobs sent by its user—it will be presented elsewhere.

In Section III some detailed description of the DOML language is given with the special attention to these features of the language which enable solving optimal control problems with a shooting method. Also in this case the full description of the language will be discussed elsewhere.

Eventually, in Section IV an example of solving an optimal control problem with applying chaining of solvers is analysed in detail. In particular scripts of the DOML language related to the problem are shown together with the numerical results.

II. TYPES OF OPTIMAL CONTROL PROBLEMS AND SOLVERS TO COPE WITH THEM

The IDOS server is aimed at solving optimal control problems described by differential equations. At the moment it can handle control problems with ordinary equations and differential–algebraic equations but the incorporation into the IDOS solvers for problems with partial differential equations is well advanced.

If we concentrate only on the problems described by ordinary differential, or differential–algebraic equations then a general optimal control problem which can be solved by a solver from the IDOS server can be stated as follows:

\[
\min_u \phi(x(t_k))
\]

subject to the constraints:

\[
F(\dot{x}(t), x(t), u(t), t) = 0 \text{ a.e. } t \in T = [0, t_k] \tag{2}
\]

\[
q(x(t), u(t), t) \leq 0, t \in T \tag{3}
\]

\[
h_i^1(x(t_k)) = 0, i \in E \tag{4}
\]

\[
h_j^2(x(t_k)) \leq 0, j \in I \tag{5}
\]

\[
u(t) \in \Omega \text{ a.e. } t \in T. \tag{6}
\]

We assume that \(u(t) \in \mathbb{R}^m, x(t) \in \mathbb{R}^n, I, E \) are finite sets of indices and \(\Omega\) is a convex bounded subset of \(\mathbb{R}^m\) (defined by box constraints). We also assume that initial conditions for (2) are properly defined and fixed. A more general problem with parameters as decision variables, with an unspecified time horizon can also be considered but for the simplicity of presentation we analyze the problem stated above.

A. Optimal control problems described by ODEs

As far as optimal control problems described by ODEs are concerned essentially three different approaches to solving them are applied.

In the first approach system equations are discretized \textit{a priori} (with respect to time) and system equations are substituted by nonlinear algebraic equations. As a result a large scale nonlinear programming problem (NLP) is solved—in the problem values of controls and state variables are decision variables. To solve these structured NLP problems we use numerical procedures which are implemented within the OLADO package ([2]). In the OLADO package three different interior point QP solvers are available: procedure which implements well-known in the literature for linear programming Mehrotra’s primal-dual predictor-corrector method [14]; procedure based on the modified Mehrotra’s algorithm by R. Franke from Technical University of Ilmenau [5] and the procedure which applies the Gondzio’s multiple centrality correctors variant of the primal-dual interior-point method for convex QP [6].

The IDOS server is also equipped with sparse nonlinear optimization HQP solver and its OMUSES interfaces to ODEs integrators ([51]). In particular, HQP solver is linked with: procedures employing Euler, or simple Runge–Kutta fourth order rules (RK4); GRK4 procedure of Rosenbrock’s type ([10]); DOPRI5 procedure based on explicit Runge–Kutta scheme due to Dormand and Prince ([9]); IMP procedure which uses the implicit midpoint rule ([10]); SDIRK procedure based on singly diagonally implicit Runge–Kutta formula ([10]) and with ODETS procedure which uses Taylor’s expansion of ODEs derived from ADOL-C driver routine forodec.

In the second approach optimal control problem to be solved is treated as a problem with continuous time. It means
that during numerical treatment of the problem system equations are integrated by integration procedure with variable stepsizes. Gradients of the functionals defining the problem are evaluated with the help of the adjoint equations which are consistent with the system equations. Essentially there are two procedures in the IDOS server which follow this approach. In the first one system equations are integrated by procedures from the cvodes part of the SUNDIALS package ([20]), also adjoint equations are integrated by procedures from the cvodes. Iteration of the optimization procedure is performed by the IPOPT method ([21]). In the second procedure, implicit Runge–Kutta method, is employed for system equations integration. In the case of this procedure optimization is performed by the SQP method which applies an active set strategy ([18]).

The third approach to optimal control problems with ODEs refers to multiple shooting methods. At the moment within the IDOS server an indirect shooting method based on Oberle’s code is available ([16]).

B. Optimal control problems described by DAEs

There are two procedures for solving optimal control problems with DAEs. The first one is aimed at problems whose differential–algebraic equations have index one. This is a procedure based on the implicit Runge–Kutta integration procedure described in ([18]).

The second procedure is based on the algorithm introduced in ([19]). In the second procedure we use Radau IIA method as implemented in [10]. Reduced gradients are evaluated on the basis of adjoint equations defined for discretized (by the integration procedure) system equations. The procedure uses the algorithm for consistent initialization of system equations and for that purpose the method based the Pantelides’ algorithm ([17]) and the kinsol procedure from the SUNDIALS package was implemented.

C. Optimal control problems with integer valued controls

For the moment, at the IDOS server, optimal control problems with integer valued controls can only have ordinary differential equations. For these problems we took BONMIN package ([3]). It has been integrated with the OLADO library and with the procedures based on the cvodes procedures from SUNDIALS package.

Table I summarizes to some extent the solvers structure of the IDOS server—in fact it is much more complex since procedures such as cvodes, or BONMIN contain essentially different methods which can be accessed by setting their parameters. Furthermore, the server IDOS employs ADOL-C package for performing automatic differentiation and OpenBLAS package for linear algebra operations.

<table>
<thead>
<tr>
<th>System equations</th>
<th>Method type</th>
<th>Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODEs</td>
<td>a priori discretization</td>
<td>OLADO; SQP (Powell’s type) + Euler; IPOPT + Euler; HQP + CMU; HQP + Euler; HQP + CMU; HQP + DOPRI5; HQP + GKR5; HQP + IMP; HQP + GKR5; HQP + SDIRK; HQP + RK4</td>
</tr>
<tr>
<td>ODEs</td>
<td>adjoint equations</td>
<td>RKCON + Radau IIA</td>
</tr>
<tr>
<td>ODEs</td>
<td>shooting method</td>
<td>IPOPT + SUNDIALS(cvodes); ENDOG + RK4</td>
</tr>
<tr>
<td>ODEs (int. controls)</td>
<td>a priori discretization</td>
<td>BONMIN + Euler</td>
</tr>
<tr>
<td>ODEs (int. controls)</td>
<td>adjoint equations</td>
<td>BONMIN + SUNDIALS(cvodes)</td>
</tr>
<tr>
<td>DAEs (index = 1)</td>
<td>adjoint equations</td>
<td>RKCON + Radau IIA</td>
</tr>
<tr>
<td>DAEs (index ≤ 3)</td>
<td>adjoint equations</td>
<td>RKCON + RADA5 + SUNDIALS(kinsol) + MAXIMA</td>
</tr>
</tbody>
</table>

PDEs

Table I

Listing of IDOS main solvers

III. THE DOML LANGUAGE

DOML is meant as a universal and programming-language-independent format of communication with IDOS for specifying optimal control problems. Consequently, submitting programming language native code, say C++, Fortran, Java or other in any form is (deliberately) not supported as it would thwart the sought-after DOML’s universality of problem definition.

When developing the proposal for DOML a number of existing standard modeling languages were considered to be used as a starting point. Among existing solutions it was Modelica (see e.g. [15]) and its optimal control extension – Optimica ([1]), developed at JModelica.org ([13]) – that was chosen as the most convenient as a base for defining the language and constructing its compiler. DOML inherits the constructs proposed in Optimica. The below description deals only with the elements introduced in DOML itself, that are connected with applying the strategy of (as colloquially referred to) "chaining of solvers". The DOML compiler implementation was based on the freely available, open-source implementation of the abovementioned Optimica compiler at JModelica.org.

The presented Dynamic Optimization Modeling Language proposed a number of modifications and new language constructs going beyond Modelica/Optimica. The main directions of DOML development revolved around the ideas of:

- Structuring the compiler to be used with a large and growing number of numerical environments so that adding new solvers to the environment would not interfere with the compiler’s internal structure;
- "chaining" of the deployed solvers, i.e. using a number of solvers in a sequence so that a succeeding solver would (use as an initial guess and) improve on the solution arrived at by its predecessor.
- using DOML to specify PDE problems so that they could be passed to the solvers installed within the IDOS environment.

The first two are in fact interconnected (a sensible selection of deployed solvers must be in place to give the user any ability of "chaining" them) and they are the topic of the following sections (PDE related constructs are beyond of the scope of the current paper). A more complete description of all the constructs designed into Dynamic Optimization Modeling Language will be the topic of another paper,
A. Adaptation of Optimica compiler

The Optimica compiler – in the most general terms – reads in an input .mo file into memory, where it creates own representation of all elements of the problem’s definition. Based on that it generates output file(s) written in a regular programming language (C/C++ in this case) that are then passed to a standard compiler (e.g. gcc) and linker to produce an executable (that in the end will find the solution, if built and run successfully). Therefore the DOML compiler, formally, is actually more of a translator – from Modelica to C/C++. Its building blocks (and hence: compilation phases) are quite typical:

- front-end – composed of scanner (lexical analyzer) and parser (syntax analyzer) that jointly consume the input file and produce an AST (Abstract Syntax Tree) representing the structure of the input.
- semantic analyzer – which analyses and transforms the AST for the purpose of e.g. resolving references, type-checking or flattening the model (an element specific to Modelica, see chapter 6 in [15]).
- code generator (or back-end) – responsible for actually producing the output, here in the form of C/C++ source files.

The code generator is template-driven: in order to produce an output it reads an external template file – resembling closely a regular C++ file where only certain spots are specially marked (with so called tags). These tags mark locations in the file where problem-dependent snippets of code are to be pasted – e.g. the $n\_real\_x$ tag is to be replaced with an integer defining the number of state variables in the model. The code generating method reads in the template file(s) and reproduces it to the output. Every time it comes across a tag, however, it replaces it in the output with an appropriately elaborated fragment of code – depending on the tag it may be a single number literal or several lines of C code. A wisely devised dispatching mechanism uses the tag name to choose and call the proper code (or: snippet) generating method (i.e. a different method is called for the $n\_real\_x$ tag, different for $n\_real\_u$, and so on).

The original Optimica compiler assumed that the generated C/C++ code is to be compiled and run in a precisely predefined environment; the set of templates was fixed and designed to work with the JMI (JModelica Model Interface) API. It is not the case for DOML. Here the goal was to have an extendable environment where new solvers (or other numerical packages) could be added virtually on the fly. A key modification to the code generator mechanism was then to replace the static set of snippet generators, with a dynamic mechanism where new code generating classes can be added and connecting to the compiler on the fly.

The three key aspects of this new functionality are:

- searching, loading and instantiating the appropriate, concrete code generating class. It utilizes Java reflection mechanism to look for proper external, compiled Java classes.
- defining the set of external template files within the (dynamically loaded) code generating class and communicating it properly to the core of the code generator. This way different code generators may be based on completely different sets of templates.
- managing the overall process of generating the code, through reading all provided template files and dispatching calls to individual tag-generating methods. The external code generator classes need only to implement the logic of generating code snippets for its specific tags (i.e.: tags present in templates associated specifically with that generator).

In result, it becomes possible for a user to provide a self-contained package (as a .jar file) holding the code generator (compiled Java code in .class files) and all template files. The mechanism outlined above loads at runtime the code generating class and then uses the appropriate templates present in that .jar file. The choice as to which code generation class to instantiate is based on additional information provided by the user in the input file through a so called annotation (a standard element of Modelica language intended for conveying auxiliary and/or vendor specific information, as specified in chapter 18, in [15]). An annotation in the form

\[
\text{annotation(solver="olado", steps="50")};
\]

instructs the DOML compiler to produce outputs dedicated for a specific solver package (may also carry solver-specific options, as in the example above). It is, in turn, required that all external code generator classes are marked with a corresponding Java annotation. Hence, in this example, DOML compiler will instantiate the class:

```java
@Solver("olado")
public class CGen4Olado extends BaseDOMLCGen {
...
```

if such found. Otherwise, when no class with the required annotation is visible, the compiler uses its built-in default code generator in attempt to do its best.

B. Language’s features for chaining of solvers

In the most basic terms, the idea of chaining solvers calls for executing one solver with a good starting point obtained as an earlier (and approximate) solution of the problem, usually from a different solver. To make that possible in DOML, it needed a language construct for specifying initial guesses for control and state variables – defining their behavior (i.e. their different values at different time instances) throughout the whole horizon. The syntax in Modelica/Optimica is not sufficient in that respect; it defines the start attribute, which sets variables’ values only at time = 0 and the initialGuess attribute that may have at most parameter variability – its value cannot change in time. In addition, Optimica seems to miss a convenient mechanism for defining and passing in arbitrary input signals.
The first extension was to allow for continuous variability in the expression defining the value of the initialGuess, so that one could e.g. specify initialGuess=mySignal for any arbitrary signal changing in time. The mechanism for defining such input signals utilizes spline interpolation – chosen for its simplicity and effectiveness. It is enough for the user to specify an input of paired values: \([x(t_i), t_i]\) for a number of time instances \(t_i\) (which is a typical output from optimal control solvers) and the Spline provides the signal (continuous and smooth if desired, as controlled by the order parameter: piece-wise constant \((order = 1)\) or continuous \((order > 1)\) signal, based on polynomial interpolation). The listing below illustrates a typical use of Spline object do define an initial guess. The data type Spline was made available as an element of DOML standard library DOML.Inputs.

Listing 1. Specifying initial guess signal for a state variable

Next to the (near optimal) trajectories found for variables and controls, other (by)products of (pre-)solving an optimal control problem may be available and beneficial when working with a high-precision solver. Among such, are the values of adjoint variables (associated with equations) and Lagrange multipliers (connected with constraints) arrived at near the solution. For the sake of simplicity, for both cases DOML syntax provides the same name: lagrange. Assigning the lagrange values to equations (constraints) is done through labeling formulas, as so

\[
\text{eq_x1: } \text{der}(x1) + p1\times x1 - p2\times x2\times u = 0;
\]

with simultaneously declaring the "label variable" as

\[
\text{Formula eq_x1};
\]

Listing 2. Definition of the predefined class Formula.

The objects of the new predefined type Formula are meant to be used to specify attributes describing the particular equation or constraint – most notably the above mentioned adjoint variable/Lagrange multiplier, but for prospective use other attributes were defined as well, see Listing 2.

An example utilizing the proposed mechanism of chaining of solvers is presented in the following section. At this point it is assumed that the (human) user is responsible for interpreting output from one solver and providing the new input file with transcibing the first solver’s findings (and possibly some other additional info). In principle, it could be possible to automate the process, i.e. to have each solver produce its output in the form of "extended input”. In that case, the output would be generated in the same DOML format as a duplicate of the input file but with a number of additional pieces of information written in.

IV. NUMERICAL EXAMPLE

As an example of the chaining mechanism let us consider the problem stated by Bryson and Ho ([4]). We show how to solve the problem by applying first a solver based on an a’priori discretization of system equations and then by using a shooting method which should give an accurate solution. A shooting method is a local method which needs a good guess in order to converge to a solution of an optimal control problem. A method based on a crude discretization of system equations should give a satisfactory guess for a shooting method.

The Bryson–Ho problem is as follows.

\[
\min_{u} z(1)
\]

subject to the constraints in the form of state equations

\[
\dot{x}(t) = v(t),
\]

\[
\dot{v}(t) = u(t),
\]

\[
\dot{z}(t) = \frac{1}{2} u^2(t),
\]

subject to the terminal constraints

\[
x(1) = 0
\]

\[
v(1) = -1
\]

and the state constraint

\[
x(t) \leq 0.1 \forall t \in [0, 1].
\]

We also assume that \(x(0) = z(0) = 0\) and \(v(0) = 1\).

The problem has one state constraint \(L(x) = x - 0.1\) of order \(p = 2\) (see [12] for the nomenclature and the analysis used in this example). One can show that at the solution there is a boundary arc defined on the interval \([t_1 \ t_2]\) \((t_1 = 0.3, t_2 = 0.7)\). Consequently there are two control laws. The first one is a boundary control obtained from the relation

\[
L^2(x(t)) = \frac{d^2}{dt^2}L(x(t)) = u(t) = 0
\]

on \([t_1 \ t_2]\). The other control law is obtained from the optimization of the Hamiltonian (on the intervals: \([0 \ t_1]\) and \([t_2 \ 1]\)) which reads as follows:

\[
H(x, v, z, u, p_x, p_v, p_z) = p_x v + p_v u + \frac{1}{2} p_z u^2,
\]

where \(p_x, p_v\) and \(p_z\) are adjoint variables corresponding to the state equations. Therefore, the control law is given by the formula: \(u(t) = -p_v(t)/p_z(t)\)

The DOML file for the problem (and for the solver based on the direct method) is given in Listing 3.

From the numerical solution we assumed the entry and exit points for the active constraint as: \(t_1 = 0.23\) and \(t_2 = 0.75\). We also estimated values of the adjoint variable corresponding to the equation (8) (at the points used in the multiple shooting method) as \((24.68, 24.68, 0.1, 0.1, -20.15, -20.15)\). Since the part of the Hamiltonian corresponding to the state constraints used in the shooting method is based on \(L^p(x)\) instead of \(L\) (as
in the case of the necessary conditions based on the standard Maximum Principle) we modified these values to values (24.68, 24.68, −20.15, −20.15, −20.15, −20.15) by taking into account the fact that the considered adjoint variable assumes the constant value on the subinterval [t1,1] (this can be derived by using analysis of [12]). The entire DOML script for the BNDSCO solver is given in Listing (4). (Notice that due to the use of ADOL-C package the user does not have to build the Hamiltonian for his problem.) The script provides the data for the shooting method: 1) guesses for all state trajectories; 2) guesses for all adjoint trajectories; 3) guesses for the switching points; 4) the control laws. The BNDSCO solver estimated accurately points t1 = 0.3 and t2 = 0.7 and as the result also corrected the solution provided by the first solver.

REFERENCES


