DOpElib: A Goal Oriented Software Library for Computing PDEs and Optimization Problems

CHRISTIAN GOLL, University of Heidelberg
THOMAS WICK, University of Heidelberg
WINNIFRIED WOLLNER, University of Hamburg

In this article, we describe the software library *Deal.II Optimization Environment* (DOpElib). The main feature of DOpElib is that it provides a unified interface to high level algorithms such as time stepping methods, nonlinear solvers and optimization routines. This structure ensures that first of all the user is required to write those sections of code that are specific to the considered problem. Second, the exchange of parts of the used routines is possible with the need for just a few lines of code to change. The article illustrates performance and features of the software package by four numerical tests.

1. INTRODUCTION

The *Deal Optimization Environment* (DOpElib) provides a software toolkit to solve forward PDE problems as well as optimal control problems constrained by a PDE. Its main feature is to give a unified interface to high level algorithms such as time stepping methods, nonlinear solvers and optimization routines. We aim that the user should only need to write those parts of the code that are problem dependent while all invariant parts of the algorithms should be reusable without any need for further coding. In particular, the user should be able to switch between various different algorithms without the need to rewrite the problem dependent code, though he or she will have to replace the algorithm object with an other one.

The solution of a broad variety of PDE is possible in other software libraries (like in deal.II [**?**], dune [**?**] , or commercial solvers like Ansys [**?**]) as well, but DOpE concentrates on a unified approach for both linear and nonlinear problems by interpreting every PDE problem as nonlinear and applying a Newton method to solve it. While deal.II leaves much of the work and many decisions to the user, DOpE intends to be user-optimized by delivering prefabricated tools which require from the user only adjustments connected to his specific problem. The solution of optimal control problems with PDE constraints is an innovation in the DOpE framework. The focus is on the numerical solution of both stationary and nonstationary problems which come from different application fields, like elasticity and plasticity, fluid dynamics, and fluid-structure interactions.

The DOpElib project is based on the *deal.II* [**?**] finite element library which has been developed initially by W. Bangerth, R. Hartmann, and G. Kanschat [**?**]. The authors acknowledge their past experience as well as discussions with the authors of the libraries Gascoigne/RoDoBo project, which was initiated by Roland Becker, Dominik Meidner,

Author's adresses: C. Goll and T. Wick, Institut für Angewandte Mathematik, Universität Heidelberg; W. Wollner, Fachbereich Mathematik, Universität Hamburg.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

and Boris Vexler [**?**]. From which some of the ideas to modularize the algorithms have arisen.

At the present stage the following key features are supported by the library

— Solution of stationary and nonstationary PDEs in 1d, 2d, and 3d.
— Various time stepping schemes (based on finite differences), such as forward Euler, backward Euler, Crank-Nicolson, shifted Crank-Nicolson, and Fractional-Step-Θ scheme.
— All finite elements of from deal.II including hp-support.
— Several examples showing the solution of several PDEs including Poisson, Navier-Stokes, Plasticity and fluid-structure interaction problems.
— Self written line search and trust region newton algorithms for the solution of optimization problems with PDEs [**?**]
— Interface to SNOPT for the solution of optimization problems with PDEs and additional other constraints.
— Several examples showing how to solve various kinds of optimization problems involving stationary PDE constraints.
— Goal-oriented mesh adaptation with the dual-weighted residual method.
— Different spatial triangulations for control and state variables.

The article is organized as follows. In Section 2

2. DETAILED DESCRIPTION OF THE FEATURES

TODO: AUS DER SICHT DES USERS ANHAND EINES KONKRETEN BEISPIELS BESCHREIBEN TODO: EINFACHES BEISPIEL NEHMEN TODO: ERWAEHNEN, DASS WIR ALLES IMMER NICHT-LINEAR BETRACHTEN TODO: AUSSEHEN EINES TYPISCHEN PDE UND OPT LOOP UND DEREN GEMEINSAMKEITEN ERKLAEREN. VIELLEICHT MIR PSEUDOCODE This library is designed to allow easy implementation and numerical solutions of problems involving partial differential equations (PDEs). The easiest case is that of a PDE in weak form to find some u

$$a(u)(\varphi) = 0 \quad \forall \varphi \in V,$$

with some appropriate space V. More complex cases involve optimization problems given in the form (OPT)

$$\begin{aligned} \min\ & J(q,u) \\ & \text{s.t. } a(q,u)(\varphi) = 0 \quad \forall \varphi \in V, \\ & a \leq q \leq b, \\ & g(q,u) \leq 0, \end{aligned}$$

where u is a FE-function and q can either be a FE-function or some fixed number of parameters, a and b are constraint bounds for the control q, and $g(\cdot)$ is some state constraint.

2.1. Problem description

In order to allow our algorithms the automatic assembly of all required data we need to have some container which contains the complete problem description in a common data format. For this we have the following classes in `DOpEsrc/container`

— `pdeproblemcontainer.h` Is used to describe stationary PDE problems.
— `instatpdeproblemcontainer.h` **TODO is still missing but will be used for nonstationary problems.** This will be implemented once we have nonstationary optimization problems running to avoid error duplication in the coding process.

— `optproblem.h` Is used to describe OPT problems governed by stationary PDEs.
— `instatoptproblemcontainer.h` Is used to describe OPT problems governed by non-stationary PDEs. The only difference to the stationary case is that we need to specify a time-stepping method.
— `interfaces/functionalinterface.h` This gives an interface for the functional $J(\cdot)$ and any other functional you may want to evaluate. In general this can be used as a base class to write your own functionals in examples. We note that we only need to write the integrands on elements or faces the loop over elements will be taken care of in the integrator. Specifically, derivatives are written therein, too.

In order to fill these containers there are two things to be done, first we need to actually write some data, for instance, the semilinear form $a(\cdot)(\cdot)$, a target functional $J(\cdot)$, etc., which describe the problem. Then we have to select some numerical algorithm components like finite elements, linear solvers ... . The latter ones should be written such that when exchanging these components none of the problem descriptions should require changes. Note that it still may be necessary to write some additional descriptions, e.g., if you solve the PDE with a fix point iteration you don't need derivatives but if you want to use Newton's method, derivatives are needed.

We will start by discussing the problem description components implemented so far

2.2. Numeric components

These are the components from which a user needs to select some in order to actually solve the given problem. They will not require any rewriting, but sometimes it is advisable to write other than the default parameter into the param file for the solution.

2.2.1. Space-time handler. First we need to select a method how to handle all dofs in space and time.

— `basic/spacetimehandler_base.h` This class is used to define an interface to the dimension independent functionality of all space time dof handlers. **TODO: Beispiele geben**
— `basic/statespacetimehandler.h` Another intermediate interface class which adds the dimension dependent functionality if only the variable u is considered, i.e., a PDE problem.
— `basic/spacetimehandler.h` Same as above but with both q and u, i.e., for OPT problems.
— `basic/mol_statespacetimehandler.h` Implementation of a method of line space time dof handler for PDE problems. It has only one spatial dofhandler that is used for all time intervals.
— `basic/mol_spacetimehandler.h` Same as above for OPT problems. A separate spatial dof handler for each of the variables q and u is maintained but only one triangulation.
— `basic/mol_multimesh_spacetimehandler.h` Same as above, but now in addition the triangulations for q and u can be refined separately from one common initial coarse triangulation. Note that this will in addition require the use of the multimesh version for integrator and face- as well as celldatacontainer.

Note that we use these for stationary problems as well, but then you don't have to specify any time information.

2.2.2. Container classes. Second you will need to specify some container classes to be used to pass data between objects. At present you don't have much choice, but you may wish to reimplement some of these if you need data that is not currently included in the containers.

— `container/celldatacontainer.h` This object is used to pass data given on the current element (cell) of the mesh to the functions in PDE, functional, ...
— `container/facedatacontainer.h` This object is used to pass data given on the current face of the mesh to the functions in PDE, functional, ...
— `container/multimesh_celldatacontainer.h` This is the same as the celldatacontainer, but it is capable to handle data defined on an alternative triangulation.
— `container/multimesh_facedatacontainer.h` This is the same as the facedatacontainer, but it is capable to handle data defined on an alternative triangulation.
— `container/integratordatacontainer.h` This contains some data that should be passed to the integrator like quadrature formulas and the above cell and face data container.

2.2.3. Time stepping schemes. **TODO: KURZ HALTEN** Third, at least for nonstationary PDEs we need to select a time stepping scheme the file names of which are mostly self explanatory:

— `include/forward_euler_problem.h`
— `include/shifted_crank_nicolson_problem.h`
— `include/backward_euler_problem.h`
— `include/fractional_step_theta_problem.h` Note that the use of this scheme requires a special Newton solver, which is, however, already implemented for the convenience of the user!
— `include/crank_nicolson_problem.h`

2.2.4. Integrator routines. Finally, we need to select a way how to integrate and solve linear and nonlinear equations

— `templates/integrator.h` This class computes integrals over a given triangulation (including its faces).
— `templates/integrator_multimesh.h` The same as above but it is possible that some of the FE functions are defined on an other triangulation as long as the have a common coarse triangulation.
— `templates/integratormixeddims.h` This is used to compute integral which are given in another (larger) dimension than the current variable. (This is exclusively used if the control variable is given by some parameters. Which means `dopedim == 0`).

2.2.5. Nonlinear solvers

— `templates/newtonsolver.h` This solves some nonlinear equation using a line-search Newton method.
— `templates/newtonsolvermixeddims.h` The same but in the case when there is another variable in a (larger) dimension is involved. See `integratormixeddims.h`.
— `templates/instat_step_newtonsolver.h` This is a Newton method as above to invert the next time-step. It differs from the plain vanilla version in that it computes certain data from the previous time step only once and not in every Newton iteration.
— `templates/fractional_step_theta_step_newtonsolver.h` This is the Newton solver for the time step in a fractional-step-theta scheme. It combines the computation of all three sub steps.

2.2.6. Linear solvers. To solve the linear equations within Newton's method, we use standard solvers from iterative type such as the method of the conjugate gradients (CG method), GMRES, or direct solvers such as `UMFPACK`. In addition, we DOpE provides a wrapper class `templates/voidlinearsolver.h` for certain cases when we know that the matrix to be inverted is the identity. It simply copies the rhs to the lhs. This is only needed for compatibility reasons some other components.

2.3. Reduced problems (Solve the PDE)

At times it is nice to remove the PDE constraint in (OPT). This is handled by so called reduced problems (for algorithmic aspects we refer the reader to [**?**]). This means that the reduced problem implicitly solves the PDE whenever required and eliminates the variable u from the problem.

— `reducedproblems/statpdeproblem.h` This is used to remove the variable u in a stationary PDE problem. This means that call the method `StatPDEProblem::ComputeReducedFunctionals` will evaluate the functionals defined in the problem description, i.e., in `PDEProblemContainer`, in the solution of the given PDE.
— `reducedproblems/statreducedproblem.h` This eliminates u from the OPT problem with a stationary PDE.
— `reducedproblems/instatreducedproblem.h` The same as above but for a nonstationary PDE. **FIXME there is something wrong in this file see FIXME comment in the source.**
— `reducedproblems/voidreducedproblem.h` A wrapper file that eliminates u if it is not present anyways. This is used so that we can use the same routines to solve problems that have no PDE constraint.

2.4. Optimization algorithms

Now, in order to solve optimization algorithms we need to define some algorithms. At present we offer a selection of algorithms that solve the reduced optimization problem where the PDE constraint has been eliminated as explained in the previous section.

— `opt_algorithms/reducedalgorithm.h` An interface for all optimization problems in the reduced formulation. It offers some test functionality to assert that the derivatives of the problem are computed correctly.
— `opt_algorithms/reducednewtonalgorithm.h` A line-search Newton algorithm using a cg method to invert the reduced hessian. Implementation ignores any additional constraints.
— `opt_algorithms/reducedtrustregionnewton.h` A trust region Newton algorithm using a cg method to invert the reduced hessian. Implementation ignores any additional constraints.
— `opt_algorithms/reduced_snopt_algorithm.h` An algorithm to solve reduced optimization problems with additional control constraints. ((reduced) state constraints are not yet implemented.)
— `opt_algorithms/reducednewtonalgorithmwithinverse.h` Line-search Newton algorithm that assumes there exists a method in the reduced problem that can invert the reduced hessian. (This usually makes sense only if there is no PDE constraint.)
— `opt_algorithms/generalized_mma_algorithm.h` An implementation of the MMA-Algorithm for structural optimization using an augmented Lagrangian formulation for the subproblems. The subproblem is implemented using the special purpose file `include/augmentedlagrangianproblem.h`.

2.5. Interfaces to other software packages

Since DOpE uses as basis the software library deal.II, all its interfaces to other packages can be accessed to. For instance, such libararies are trilinos for with an algebraic multigrid method, MPI for a parallel solution of the problem.

In addition DOpE itself has an interface to SNOPT.

3. DOCUMENTATION, CODE DEVELOPMENT, WEBPAGE

At the present step, the DOpElib project comes with an detailed documentation of all features and examples in pdf format and an detailed programming code documentation.

TODO: LICENSE? HOW TO ACCESS TO DOPE? TODO: NAME OF THE WEBSITE

In addition, the DOpElib test suite provides some regression tests. They are run to compare the output to previous outputs. This is useful (necessary) after changing programming code anywhere in the library. If a test succeeds, everything is fine in the library. If not, you should not check in your code into DOpE. Please make sure what is going wrong and WHY! Every command is computed via a Makefile.

4. APPLICATIONS

We present some numerical tests that have been already computed with DOpElib. These tests demonstrate the performance of DOpElib by covering different computational aspects.

— The first numerical test shows various time-stepping schemes and the numerical solution of nonlinear coupled partial differential equations, formulated in a monolithic solution algorithm. Specifically, the equations are solved by the method of lines, i.e, first spatial discretization and then temporal discretization with a Galerkin finite element scheme. The nonlinaer problem is solved by Newton's method. The Jacobian is derived by analytical expressions of the derivatives. In DOpElib this example is implemented in `Example/PDE/InstatPDE/Example2`
— The second example presents an optimization problem and its numerical solution.
— The main purpose of the third example is the application of refinement techniques for local mesh refinement for two and three dimensional problems. We consider the numerical solution of fluid flows governed by the Navier-Stokes equations. In contrast to the previous tests, we use finite elements on locally adapted meshes with goal-oriented mesh refinement with the help of the dual-weighted residual method. Using this method of local mesh refinement, the error estimator is derived with respect to some goal functional and provides optimal meshes to the accurate measurement of the considered functional value. Consequently, this is an effiecient method to reduce the computational cost. Most other software libraries does not provide goal-oriented mesh refinement.
— Here, we show a time-dependent optimization problem or FSI optimization.

4.1. Nonstationary Fluid-Structure Interaction

In this example, the three proposed mesh motion models are applied to an unsteady fluid-structure interaction problem. We consider the numerical benchmark test FSI 2, which was proposed in [**?**]. The configuration is sketched in Figure **??**. New results can be found in [**?**; **?**; **?**]. The Fractional-Step-θ scheme was used for time discretization with different time step sizes k.

4.2. Compliance minimization

Results already obtained.

4.3. Goal-oriented mesh refinement for Navier-Stokes

Has to be computed in two and three dimensions.

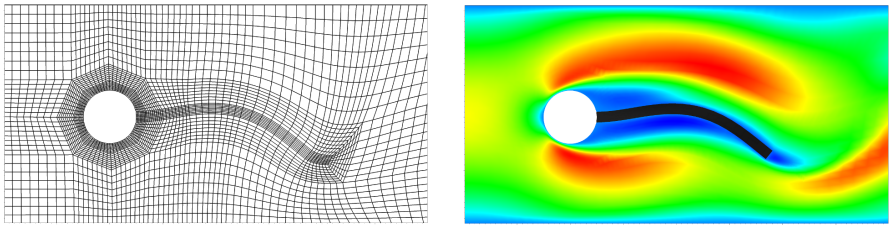

Fig. 1: FSI 2 test case: mesh (left) and velocity profile in vertical direction (right) at time $t = 16.14s$.

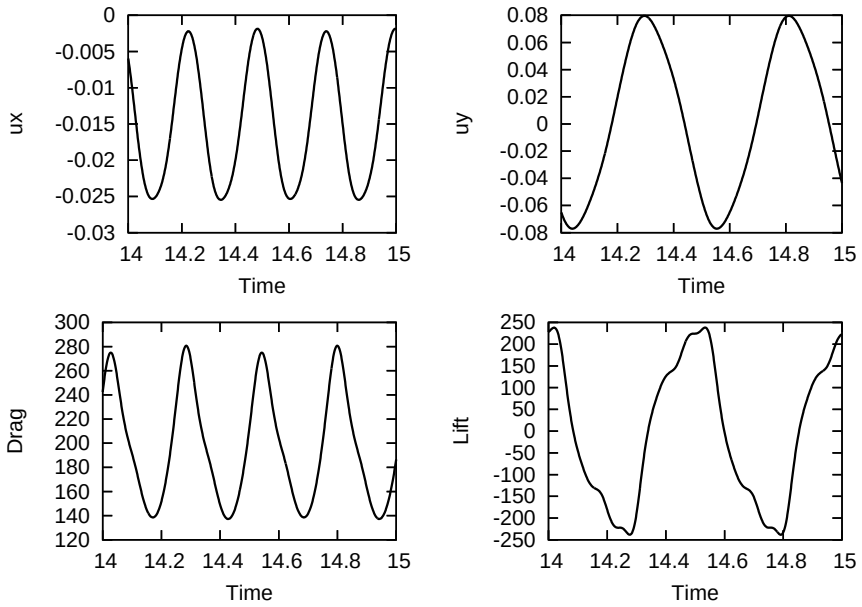

Fig. 2: FSI 2: the deflections of the beam, $u_x(A)$ and $u_y(A)$ (in cm), and the drag and the lift evaluation (in $kg/m\,s^2$) are displayed versus time (in s).

4.4. Time-dependent Opimization or FSI optimization

5. CONCLUSIONS AND OUTLOOK

In this article, we described the features and applications of the DOpElib project, which was initiated at the Heidelberg University in 2010. Specifically, the applications cover a broad spectrum of numerical exmamples motivated by different disciplines. Currently, we are developing concepts for the efficient numerical solution of time-dependent optimization problems governed by PDEs. **TODO: WAS WOLLEN WIR NOCH IN DEN NÃĎCHSTEN 1-2 JAHREN ERREICHEN?**

Acknowledgment

The second author thanks Rolf Rannacher from the Institute for Applied mathematics at the Heidelberg University for financial suppport.