J. Numer. Math., Vol. 0, No. 0, pp. 1–13 (2012)
DOI 10.1515/ JNUM.2012.000
 Prepared using jnm.sty *[Version: 20.02.2007 v1.3]*

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

Christian Goll*, Thomas Wick*, Winnifried Wollner*

Received

Abstract — In this article, we describe the software library package *Deal Optimization Environment* (DOpE). Its main feature is to give a unified interface to high level algorithms such as time stepping methods, nonlinear solvers and optimization routines. The performance and features of the software package are demonstrated by computing four numerical tests.

Keywords:

1. Introduction

The *Deal Optimization Environment* (DOpE) project is based on the *deal.II* [1] finite element library which has been developed initially by W. Bangerth, R. Hartmann, and G. Kanschat [1]. 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 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, and Boris Vexler [2]. From which some of the ideas to modularize the algorithms have arisen.

The aim of DOpE is to provide a software toolkit to solve forward PDE problems as well as optimal control problems constrained by PDE. The solution of a broad variety of PDE is possible in deal.II 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

*Institut für Angewandte Mathematik, Universität Heidelberg, Im Neuenheimer Feld 293/294, D-69120 Heidelberg, Germany.

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.

At the present stage the following 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 [4]
- 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.
- Mesh adaptation.
- Different spatial triangulations for control and state variables.

The rest of this document is structured as follows: We start with an introduction in Chapter **??** where you will learn what is needed to run `DOpElib`. Further you will learn what problems we can solve and how all the different classes work together for this purpose. This should help you figure out what the different classes do if you are in need of writing your own algorithm.

Then assuming that you can work to your satisfaction with the algorithms already implemented we will show you how to create your own running example in Chapter **??**. This will be followed by a detailed description of all examples already shipped with the library. You can find the examples for the solution of PDEs in Chapter **??** and those for the solution of optimization problems with PDEs in Chapter **??**.

These notes conclude with a section that explains how we do automated testing of the implementation in Chapter **??**. This chapter will be of interest only if you are trying to implement some new features to the library so that you can check that the new code did not break anything.

2. Detailed Description of the Features

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)

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

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. **TODO should be renamed to optproblemcontainer.h including the class name!**
- `instatoptproblemcontainer.h` Is used to describe OPT problems governed by nonstationary PDEs. The only difference to the stationary case is that we need to specify a time-stepping method.

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 dof-handler 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 celldata-container.

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. 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.

- templates/cglinearsolver.h This is a wrapper for the cg solver implemented in deal.II. The solver will build and store the stiffness matrix for the PDE.

- templates/gmreslinearsolver.h This is a wrapper for the gmres solver implemented in deal.II. The solver will build and store the stiffness matrix for the PDE.

- templates/directlinearsolver.h This is a wrapper for the direct solver implemented in deal.II using UMFPACK. The solver will build and store the stiffness matrix for the PDE.

- templates/voidlinearsolver.h This is a wrapper 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. Problem specific classes

The following classes are used to describe the problem and will usually require some implementation.

- `basic/constraints.h` This is used by the spacetimehandlers to compute the number of constraints from the control and state vectors. It must not be reimplemented by the user, but needs to be properly initialized if OPT is used with box control constraints or $g(q,u) \leqslant 0$.

- `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.

- `interfaces/constraintinterface.h` This gives an interface for both the control box constraints as well as the general constraint $g \leqslant 0$. This needs to be specified if constraints are to be used. If they are not needed a default class `problemdata/noconstraints.h` can be used. 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.

- `interfaces/pdeinterface.h` This defines an interface for the partial differential equation $a(q,u)(\varphi) = 0$. This needs to be written by the user. 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.

- `interfaces/dirichletdatainterface.h` This gives an interface to the Dirichlet data for a problem. If the Dirichlet data are simply a function (and do not depend on the control q) one can use the default class `problemdata/simpledirichletdata.h`.

2.4. 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 [3]). 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.

2.5. 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`.

2.6. Other Components

Beyond these clearly structured groups before there are some classes remaining that do not fit the above but are important for the user to know.

2.6.1. Vectors.

- `include/statevector.h` This stores all dofs in space and time for the state variable u. It is possible to select whether all this should be kept in memory or or unused parts can be written to the hard disk.

- `include/controlvector.h` This stores all dofs in space and time for the control variable q. At present no time dependence is implemented.

- `include/constraintvector.h` This stores all dofs in space and time for the non PDE constraints (and corresponding multipliers). At present no time dependence is implemented.

2.6.2. Parameter handling.

- `include/parameterreader.h` This file is used to define a parameter reader that is used to read run time parameters from a given file.

2.6.3. Exception handling.

- `include/dopeexception.h` Defines some Exceptions that are thrown by the program should it encounter any unexpected errors.

- `include/dopeexceptionhandler.h` This class is used to write information contained in the exceptions to the output in a uniform manner.

2.6.4. Output handling.

- `include/outputhandler.h` This file defines an outputhandler object which can be used to decide whether some information should be written to screen or file. In addition it can format output according to some run time parameters given by a parameter file.

2.7. Data Access

- `include/solutionextractor.h` This class is used to gain access to the finite element solutions stored in the reduced problems.

2.7.1. Constraints and system matrix.

- `include/userdefineddofconstraints.h` This class sets the constraints on the DOFs of the state and/or control FE solution. DOpE itself builds the hanging-node-constraints, but the user can reimplement this class and thus include other constraints as well (for example periodic BC). Note, that the hanging-node-constraints come first (in case of conflicting constraints.)

- `include/sparsitymaker.h` This class sets the sparsity pattern for the state FE solution. The standard implementation is just a wrapper for `dealii::DoFTools::make_sparsity_pattern`, but the user can reimplement this class to allow for more sophisticated sparsity patterns.

2.7.2. HP components.

- `interfaces/active_fe_index_setter_interface.h` In the case of hp finite elements, one has to specify for each cell which finite element to use. This is done via this interface.

2.8. Internal structures

TODO: ab hier habe ich noch nichts gemacht (TW)

2.8.1. Interface Classes.

- `interfaces/transposeddirichletdatainterface.h` This provides an interface to the functionality required by *transposed Dirichlet data*. Usually when one applies Dirichlet data g to a function one has to calculate a continuation Bg which is defined on the whole domain. In optimization problems when the Dirichlet data depends on the control one has to evaluate the dual operator B^* in order to obtain a representation for the reduced gradient of the objective J. This is done using the *transposed Dirichlet data*.

- `interfaces/reducedprobleminterface.h` In order to allow all algorithms to be written independent of the given (OPT) problem (and not requiring the problem as template argument) there is a common base class which defines the required interfaces.

- `interfaces/pdeprobleminterface.h` The same as above but for (PDE) problems.

2.8.2. Default Classes.

- `problemdata/noconstraints.h` A class that can be used for optimization problems having only a PDE constraint but no further constraints.

- `problemdata/simpledirichletdata.h` A class that can be used to implement Dirichlet data that are given as a fixed function (independent of the control).

2.8.3. Auto-generated Problem Descriptions.

- `problemdata/stateproblem.h` This is the problem description for the (forward/primal) PDE constraint. **Similar descriptors will be build for the other problems (adjoint, tangent, ...) when time allows.**

- `problemdata/initialproblem.h` This is the problem descriptor to compute the finite element representation of the initial values. This is generated by the different time-stepping schemes based upon the defined representation by the pde, which is set to the component wise L^2 projection by default.

- `problemdata/primaldirichletdata.h` This class contains the Dirichlet data for the forward/primal PDE.

- `problemdata/tangentdirichletdata.h` This class contains the Dirichlet data for the tangent PDE, i.e., the first derivative of the Dirichlet data.

- `problemdata/transposedgradientdirichletdata.h` This contains the transposed Dirichlet data needed to calculate the gradient of the reduced objective functional, for detail see `interfaces/transposeddirichletdatainterface.h`.

- `problemdata/transposedhessiandirichletdata.h` This contains the transposed Dirichlet data needed to calculate the hessian of the reduced objective functional, for detail see `interfaces/transposeddirichletdatainterface.h`.

2.8.4. Management of Time Dependent Problems.

- `include/timedofhandler.h` DoFHandler responsible for the management of the timedofs (this is a part of the `SpaceTimeDoFHandler`-classes). Basically a wrapper for a $1d$ `deal.II`-DoFHandler.

- `include/timeiterator.h` This class works as an iterator on the `TimeDoFHandler`.

2.9. Wrapper classes

- `wrapper/dofhandler_wrapper.h` A wrapper class for the `deal.II` DoFHandlers. This class is needed to provide support for the `dim` $= 0$ case and to have a uniform interface to DoFHandler and HPDoFHandler.

- `wrapper/preconditioner_wrapper.h` Contains wrappers for several of the preconditioners in `deal.II`. This is required since unfortunately the preconditioners in `deal.II` have different interfaces for their initialization.

- `wrapper/function_wrapper.h` An interface that allows to use functions that depend not only on space but also on time.

- `wrapper/snopt_wrapper.h` An interface to the SNOPT fortran library. This is an additional wrapper to the one provided by SNOPT to allow automatic construction of the functions required by SNOPT using our library.

- `wrapper/fevalues_wrapper.h` **Will be removed soon!**

2.10. Other

- `basic/dopetypes.h` This file contains type definitions used in the library.

TODO! The following files need some appropriate description here This should be done by someone who knows what they are used for. Please move the description to the right place above. tsschemes.

2.11. Interfaces to other software packages

deal.II and SNOPT

3. Numerical Examples

3.1. Nonstationary Fluid-Structure Interaction

The results are already obtained

3.2. Compliance minimization

Results already obtained.

3.3. Goal-oriented mesh refinement for Navier-Stokes

Has to be computed.

3.4. Dirichlet Velocity, Neumann Pressure, and Periodic Boundary Conditions

I do not know.

Acknowledgment

References

1. Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. *Differential Equations Analysis Library*, 2010.

2. R. Becker, D. Meidner, and B. Vexler. *RODOBO: A C++ library for optimization with stationary and nonstationary PDEs*, 2005.

3. Roland Becker, Dominik Meidner, and Boris Vexler. Efficient numerical solution of parabolic optimization problems by finite element methods. *Optim. Methods Softw.*, 22(5):813–833, 2007.

4. J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer, US, 2nd edition, 2000.