

DOpElib: Differential Equations and Optimization Environment

Christian Goll*, Thomas Wick†, Winnifried Wollner‡

May 11, 2013

*Heidelberg University, christian.goll@iwr.uni-heidelberg.de

†Heidelberg University, thomas.wick@iwr.uni-heidelberg.de

‡University of Hamburg, winnifried.wollner@math.uni-hamburg.de

Contents

1	Foreword	4
2	Introduction	6
2.1	License information	6
2.2	Developers	6
2.3	Software requirements	6
2.3.1	g++	7
2.3.2	deal.II	7
2.3.3	ThirdPartyLibraries	7
2.4	Installation	8
2.5	FAQs	8
2.6	How this library is structured	9
2.6.1	Problem description	9
2.6.2	Numeric components	10
2.6.3	Problem specific classes	13
2.6.4	Reduced problems (Solve the PDE)	13
2.6.5	Optimization algorithms	14
2.6.6	Other Components	15
2.6.7	Data Access	15
2.6.8	Internal structures	16
2.6.9	Wrapper classes	17
2.6.10	Other	18
3	Example Handling, Creating new Examples	19
4	Examples for PDE Solution	22
4.1	Stationary PDEs	22
4.1.1	Stationary Stokes Equations	22
4.1.2	Stationary FSI with INH Material	26
4.1.3	Stationary FSI with STVK Material	28
4.1.4	Stationary Elasticity Benchmark	30
4.1.5	Stationary Plasticity Benchmark	32
4.1.6	Stationary Stokes Equations with periodic BC	33
4.1.7	Laplace Equation in 2D	34
4.1.8	Laplace Equation in 3D	35
4.1.9	Adaptive Solution of Laplace Equation in 2D	37
4.1.10	Usage of Higher Order Mappings: Approximation of π	40

Contents

4.1.11	Stationary Stokes Equations with hp-Elements	41
4.2	Nonstationary PDEs	42
4.2.1	Nonstationary Navier-Stokes Equations	42
4.2.2	Nonstationary FSI Problem	45
4.2.3	Black-Scholes Equation	47
4.2.4	Heat Equation in 1D	49
4.2.5	Heat Equation in 2D with nonlinearity	50
5	Examples with Optimization	51
5.1	Subject to a Stationary PDE	51
5.1.1	Distributed control with a linear elliptic PDE	51
5.1.2	Parameter control with a linear elliptic PDE	52
5.1.3	Parameter control with a nonlinear PDE from fluid dynamics . . .	54
5.1.4	Control in the dirichlet boundary values	55
5.1.5	Distributed Control with Different Meshes for Control and State .	56
5.1.6	Compliance Minimization of a variable Thickness MBB-Beam . . .	57
5.1.7	Distributed control with a linear elliptic PDE using IPOPT/SNOPT	58
5.1.8	Topology optimization of an MBB-Beam using SNOPT	59
5.1.9	Parameter control with a non-linear PDE from FSI dynamics . . .	60
5.2	Subject to a Non-Stationary PDE	61
5.2.1	Control of a nonlinear heat equation via the initial values	61
6	Testing examples	62
6.1	Where can I find the tests	62
6.2	How to start testing?	62
	Index	64

1 Foreword

In this report, we describe the DOpE *Differential Equations and Optimization Environment* project. Originally, the project was initiated in the year 2009 at the University of Heidelberg (Germany) in the numerical analysis group of Rolf Rannacher.

The *Differential Equations and Optimization Environment* (DOpE) project is mainly based on the *deal.II* 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 [3]. 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* 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.

1 Foreword

- 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 [5]
- 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 2 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 3. 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 4 and those for the solution of optimization problems with PDEs in Chapter 5.

These notes conclude with a section that explains how we do automated testing of the implementation in Chapter 6. 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 Introduction

2.1 License information

Copyright (C) 2012 by the DOpElib authors

This file is part of DOpElib

DOpElib is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

DOpElib is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Please refer to the file LICENSE.TXT included in this distribution for further information on this license.

2.2 Developers

The library is currently maintained by

- Christian Goll (Heidelberg University)
- Thomas Wick (The University of Texas at Austin)
- Winnifried Wollner (University of Hamburg)

Furthermore, there are more highly appreciated contributions made by

- Michael Geiger (Examples for Plasticity, and Documentation of several PDE-Examples)
- Francesco Ludovici (Augmenting the Documentation)

2.3 Software requirements

The library DOpE has been tested to work on both Linux and MAC OSX. See also the README.OSX file

2.3.1 g++

D0pElib requires a recent g++ (at least 4.4.3) due to some new C++ features implemented in C++0x and C++11. You can check the version number using the command-line argument `g++ -v`.

Under Linux-systems this typically means that you have to do nothing if you have a recent version number. Otherwise you can either install the required version of g++ using the appropriate software installation tool, or you can build the required version from source gcc.gnu.org.

Under MAC OSX, you need to install the XCode tools delivered with the operating system or available for free developer.apple.com/xcode. Unfortunately, the delivered version of g++ is too old, so you need to install the real thing. To do so, download MacPorts from macports.org. Once you have installed MacPorts you can use it to install additional linux software. First, update the MacPorts installation `sudo port selfupdate` after that you can install a new version of g++ using forinstance `sudo port install gcc45` to install version 4.5 of the compiler. Afterwards, you need to set the search path appropriate to find the macports version of g++, to check if this has been done use `g++ -v`.

2.3.2 deal.II

This library is mainly based upon deal.II hence in order to run D0pElib you need a running copy of deal.II.

The deal.II library is open source and is freely available for noncommercial project. It can be downloaded from <http://www.dealii.org/>. On this homepage, one also finds lots of further information on deal.II as well as an extensive tutorial where many features of deal.II are discussed in a well-documented example framework. In order to use DOpE, it is highly recommendable to be roughly acquainted with deal.II.

When installing deal.II you should take care to configure it to use UMFPACK, i.e., `./configure --with-umfpack`

2.3.3 ThirdPartyLibraries

In order for DOpE to be able to auto-detect some of the installed Third Party Libraries you should generate according links in the ThirdPartyLibs. See also ThirdPartyLibs/README.

SNOPT

If you would like to use the features offered in our SNOPT wrapper. You will need to obtain a license for SNOPT http://www.sbsi-sol-optimize.com/asp/sol_product_snopt.htm. Unfortunately this is at present not available for free, but you should check if there is a department license already available. For further information you should consult the file ThirdPartyLibs/SNOPT.INSTALLNOTES. In particular you need to configure deal.II with at least the following options:

`./configure --with-umfpack --with-blas --with-lapack`

IPOPT

If you would like to use the optimization routines offered by IPOPT <https://projects.coin-or.org/Ipopt> you can install this yourself and add a symlink as described in `ThirdPartyLibs/README`.

Alternatively, you can use installation script `ThirdPartyLibs/install-free-libs.sh`. Note that to use all available linear solvers you may have to obtain a corresponding license manually. This is true in particular for the HSL solvers MA27, For information on these see the information provided by the installation script.

2.4 Installation

I) Get a copy of `D0pElib` from the svn repository using

```
svn co --username=[your username] \  
https://ganymed.iwr.uni-heidelberg.de/svn/dope
```

(The backslash in the first line means that the line will be continued as one!) Note that you need a valid username and password. If you have none you can contact the maintainer by sending an EMail to dope@ganymed.iwr.uni-heidelberg.de.

II) Install `deal.II` to your home directory, i.e., it should be located in `~/deal.II`.

If you would like to have another path you will either have to manually edit the files `D0pEsrc/source/Makefile` and `Examples/Make.global_options` and replace the line `D = $(HOME)/deal.II` with the appropriate path.

Configure it properly and make the whole `deal.II` library.

III) If you want to use some of the supported third party libraries install them and follow the instructions in `ThirdPartyLibs/README`. There may be further information in some `ThirdPartyLibs/*.INSTALLNOTES` that you may want to consider.

IV) To build the `D0pElib` you have to change to the directory `D0pEsrc` where you can call

```
make all
```

to build the library. If you want to generate some documentation you may use `make pdf-doc` or `make html-doc` which will require either latex or doxygen to be installed on your computer.

V) To build the Examples change to the `Examples` directory and `make all`. VI) Finally you may wish to test if everything worked. To do so you can change to the `Examples` directory and `make tests` which will give you a list of all the examples and whether they behave as expected by the library, see also Chapter 6.

2.5 FAQs

1.) When building the library I get an error message:

- **unrecognized command line option "-std=gnu++0x"**

This means that your compiler is too old. You can check the version of your compiler using `g++ -v`. If the version is lower than 4.5 you need to get a newer compiler version.

2.) I have installed a new g++ compiler but g++ -v still finds the old one :

This means that your computer does not find the new compiler. Try `which g++` to see whether it appears in the list of available compilers (but is maybe too far in the back of the list.) Then you should modify your `$PATH` environment variable so that the new g++ compiler appears.

If `which g++` only returns one g++ compiler, then probably you need to set an appropriate symlink. Or more robust, you can configure `dealii` to use the compiler you intend by configuring `deal` with the right compiler. To do so adjust the `CC` and `CXX` environment variable appropriately before configuring `dealii`

For example on Mac OSX you will find only one g++ compiler `/usr/bin/g++` which is in fact a symlink to `/usr/bin/g++-4.2`. So that you need to install a newer compiler this may then be found ,e.g., using macports g++ 4.5 on OSX Lion, in `/opt/local/bin/g++-mp-4.5`.

2.6 How this library is structured

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

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

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

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

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.

2.6.1 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 `D0pEsrc/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.
- `optproblemcontainer.h` Is used to describe OPT problems governed by stationary PDEs.
- `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

2.6.2 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.

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.

2 Introduction

- `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.

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/integratorcontainer.h` This contains some data that should be passed to the integrator like quadrature formulas and the above cell and face data container.
- `container/refinementcontainer.h` The classes defined herein are given to the `RefineSpace` method of the `SpaceTimeHandler` and determine how we define the spatial mesh (i.e. globally or locally with a fixed fraction, fixed number or somehow 'optimized' strategy).

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:

- `tsschemes/forward_euler_problem.h`
- `tsschemes/shifted_crank_nicolson_problem.h`
- `tsschemes/backward_euler_problem.h`

2 Introduction

- `tsschemes/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!
- `tsschemes/krank_nicolson_problem.h`

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 they have a common coarse triangulation.
- `templates/integratormixeddims.h` This is used to compute integrals 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`).

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 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.

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.

2.6.3 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 $g(q, u) \leq 0$.
- `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 $g \leq 0$. 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 $a(q, u)(\phi) = 0$. 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/simplifiedirichletdata.h`.

2.6.4 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 [4]). 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 non-stationary PDE.
- `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.
- `reducedproblems/ipopt_problem.h` A wrapper file required when solving optimization problems using the `reduced_ipopt_algorithm`. This file hides the interface to IPOPT.

2.6.5 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 using the third-party library SNOPT. ((reduced) state constraints are not yet implemented.)
- `opt_algorithms/reduced_ipopt_algorithm.h` An algorithm to solve reduced optimization problems with additional control constraints. using the third-party library IPOPT. ((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`.

2.6.6 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.

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 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.

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.

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.

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.

2.6.7 Data Access

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

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.
- `include/pointconstraintsmaker.h` This class allows to set homogeneous dirichlet values at given points/components.

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.

2.6.8 Internal structures

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.

Default Classes

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

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

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`.

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`.

2.6.9 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/fevalues_wrapper.h` **Will be removed soon!**
- `wrapper/function_wrapper.h` An interface that allows to use functions that depend not only on space but also on time.

2 Introduction

- `wrapper/mapping_wrapper.h` An interface that allows to use `deal.II`-mappings as well as `deal.II`-mapping collections depending of the `DoFHandler` in use. To this end, the class has a template parameter `DOFHANDLER`.
- `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/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/solutiontransfer_wrapper.h` A wrapper for the `SolutionTransfer` class from `deal.II`.
- `wrapper/dataout_wrapper.h` A wrapper for the `DataOut` class from `deal.II`.

2.6.10 Other

- `basic/dopetypes.h` This file contains type definitions used in the library.
- `basic/sth_internals.h` Wrapper for the `MapDoFsToSupportPoints` function. The implementation of this changes with the `deal.II` version in use.
- `include/helper.h` Collection of various small helper functions.
- `reducedproblems/problemcontainer_internal.h` Houses some functions und variables common in the various `problemcontainer`.
- `tsschemes/primal_ts_base.h` This class contains the methods which all primal time stepping schemes share.
- `tsschemes/ts_base.h` This class contains the methods which all time stepping schemes share.

3 Example Handling, Creating new Examples

To implement new examples or to use existing examples from the library for own research, the user can simply copy an existing example. In this new example, own code and changes can be compiled.

Before being able to change and compile the new code, the user must follow some easy steps in order to modify the information related to the old code. In this section we explain how to modify such information using as model `PDE/StatPDE/Example1`.

1. In a first step, we copy `Example1` and renamed it, e.g., `ExampleNew`. At the same time it is important to remove the repository information that it is stored in the directory `.svn`.

After having reached the folder of the example in question in the terminal, `PDE/StatPDE` in our case, we perform these operation writing the following:

```
cp -r Example1 ExampleNew
cd ExampleNew
rm -rf .svn
cd Test
rm -rf .svn
```

Please note that removing the `.svn` sub-directories is important, as otherwise your files may be replaced or changed during your next update. Also, if you can submit information to the subversion repository you might accidentally overwrite the original example, here `Example1`.

2. We now have to change the target in the `Makefile`, otherwise our new example will refer to the executable of the old one.

Thus we open the `Makefile` in the directory `ExampleNew` to change the name of the executable. In our example you will find a line

```
target    = $(BINDIR)/DOpE-PDE-StatPDE-Example1-$(dope_dimension)d-$(deal_II_dimension)d
```

Here we need to replace the target to a new name to avoid any possible complication. Thus for example we change the line to

```
target    = $(BINDIR)/MyExampleNew
```

3 Example Handling, Creating new Examples

This will build the executable file in the standard binary directory of D0pElib, i.e., \$(D0pE)/bin, under the name MyExampleNew.

Note that you should use a different name for all your examples as otherwise you may experience a lot of strange behavior, as the executable will be overwritten by all your examples!

3. If you want to run automated tests on your program so that you can verify whether your code is running as expected after updating the library you may want to update the sub-directory **Test** as well, see also Chapter 6. Otherwise you may skip this step.

Change to the **Test** sub-directory. And then modify the test-script to contain the new name of the executable. Assuming you want to use Emacs, open the file **test.sh**

```
PDE/StatPDE/Example1/Test> emacs test.sh
```

where, in our example you find two times the lines

```
echo "Running Program ../../../../bin/D0pE-PDE-StatPDE-Example1-2d-2d test.prm"
(../../../../../../bin/D0pE-PDE-StatPDE-Example1-2d-2d test.prm 2>&1) > /dev/null
```

if you made a copy of an other example the part D0pE-PDE-StatPDE-Example1-2d-2d may differ. These lines need to be replaced with the new name of the executable, i.e., for our given example

```
echo "Running Program ../../../../bin/MyExampleNew test.prm"
(../../../../../../bin/MyExampleNew test.prm 2>&1) > /dev/null
```

4. Now, you are prepared to change any of the problem dependent data in information in the files

main.cc, **localpde.h**, **functionals.h**, **localfunctional.h**, etc

5. Once you have finished and are sure that your example is running correctly and you want to use the automated test scripts –see 3) above– You need to store new test information to account for your changes.

To do so, change to the **Test** sub-directory and run the test:

```
./test Test
```

Note that this should fail, otherwise you have not changed anything in the program, or forgot part 3) of this description.

3 Example Handling, Creating new Examples

If it failed have a look into the file `dope.log` and see whether you like the output. If you do not like it you may wish to update the file `test.prm` that takes care of the parameters for the test run.

Once your satisfied with what you see in the log-file `dope.log` you need to store that information using

```
./test Store
```

4 Examples for PDE Solution

4.1 Stationary PDEs

4.1.1 Stationary Stokes Equations

General problem description

In this example we consider the stationary incompressible Stokes equation . Here, we use the symmetric stress tensor which has a little consequence when using the do-nothing outflow condition. In strong formulation we have

$$\begin{aligned} -\frac{1}{2}\nabla \cdot (\nabla v + \nabla v^T) + \nabla p &= f \\ \nabla \cdot v &= 0 \end{aligned} \tag{4.1}$$

on the domain $\Omega = [-6, 6] \times [0, 2]$. We split $\partial\Omega = \Gamma_D \cup \Gamma_{out}$. The right hand side of the channel is Γ_{out} on which we describe the free outflow condition, on the rest of the boundary we prescribe dirichlet values (An parabolic inflow on the left hand side and zero on the upper and lower channel walls). We choose for simplicity $f = 0$.

Program structure

In all examples, the whole program is split up into several files for the sake of readability. These files are always denoted in the same way, so we only have to explain the general structure in this first example, whereas in the following examples, we will only point out differences to the current one. The content of the single files will be described in more detail below.

If we do not use one of the standard grids given in the deal.II library, we can read a grid from an input file. In our example, the domain $\Omega = [-6, 6] \times [0, 2]$ is given in the *channel.inp* file, where all nodes, cells and boundary lines are listed explicitly and the boundary is divided into disjoint parts by attributing different colors to the boundary lines.

Certain parameters occurring during the solution process, e.g. error tolerances or the maximum number of iterations in an iterative solution procedure, are fixed in a parameter file called *dope.prm*. This parameter file comprises several subsections corresponding to different solver components.

4 Examples for PDE Solution

In the *functionals.h* file we declare classes for different scalar quantities of interest (described mathematically as functionals) which we want to evaluate during the solution process.

The *localfunctional.h* file is relevant only if we want to solve an optimal control problem. In this case, it contains the cost functional, whereas the file is not needed for the forward solution of PDEs. We will get back to this later in the context of optimal control problems.

All information about the PDE problem (in the optimal control case about the constraining PDE) is included in the *localpde.h* file. In a class called `LocalPDE`, we build up the cell equation, the cell matrix and cell righthand side as well as the boundary equation, boundary matrix and boundary righthand side. Later on, the integrator collects this local information and creates the global vectors and matrices.

The most important part of each example is the *main.cc* file which contains the `int main()` function. Here we create objects of all classes described above and actually solve the respective problem.

The functionals.h file

Here, we declare all quantities of interest (functionals), e.g. point values, drag, lift, mean values of certain quantities over a subdomain etc. Each of these functionals is declared as a class of its own, but in `D0pElib` all classes are derived from a so-called `FunctionalInterface` class. In the current example we declare functionals for point values of the velocity and for the flux at the outflow boundary of the channel.

The localpde.h file

The *LocalPDE* is derived from a `PDEInterface` class. It comprises several functions which build up the cell and boundary equations, matrices and righthand sides. The weak formulation of problem (4.1) with $f = 0$ is

$$\frac{1}{2}(\nabla v, \nabla \varphi)_{\Omega} + \frac{1}{2}(\nabla v^T, \nabla \varphi)_{\Omega} - (p, \nabla \cdot \varphi)_{\Omega} + (\nabla \cdot v, \psi)_{\Omega} - (n \cdot \nabla v^T, \phi)_{\Gamma_{out}} = 0. \quad (4.2)$$

Remark 4.1.1. Note the additional term on Γ_{out} , which is a consequence of the use of the symmetric stress tensor together with the free outflow condition.

This problem is vector valued, i.e. the velocity variable v has two components and the pressure variable p is a scalar. For the implementation, we use a vector valued solution variable with three components, where the distinction between velocity and pressure is done by use of the `deal.II FEValuesExtractors` class.

4 Examples for PDE Solution

Furthermore, in `DOpElib` we always interpret the problems in the context of a Newton method. Usually, a PDE in its weak formulation is given as

$$a(u; \varphi) = f(\varphi).$$

The lefthand side is implemented in the `CellEquation` function, the righthand side is implemented in the `CellRightHandSide` function (which is unused in this example, because $f = 0$).

Remark 4.1.2. The weak formulation might contain some terms on faces or (parts of) the boundary. `DOpE` is able to handle these via `BoundaryEquation`, `BoundaryRightHandSide` etc.. To keep things simple, we neglect these terms in this introduction.

To apply Newton's method, this problem is linearized: on the lefthand side, we have the derivative of the (semilinear) form $a(\cdot; \cdot)$ with respect to the solution variable u , and the righthand side is the residual of the weak formulation:

$$a'_u(u; u^+, \varphi) = -a(u; \varphi) + f(\varphi).$$

In the `CellMatrix` function, we implement the following matrix A as representation of the derivative on the lefthand side:

$$A = (a'_u(u; \varphi_i, \varphi_j))_{j,i=1}^N$$

with the number N of the degrees of freedom. Similarly, the `CellEquation` contains the vector

$$a = a(u; \varphi_i)_{i=1}^N,$$

and the `CellRightHandSide` in the case $f \neq 0$ would contain a vector

$$\tilde{f} = (f; \varphi_i)_{i=1}^N.$$

The system of equations which is then actually solved is

$$A\tilde{u}^+ = -a + \tilde{f}.$$

Because of the linearity of equation (4.2), there is almost no difference between the two functions.

At this point, it is important to note that `DOpE` interprets any given problem as a nonlinear one which is solved by Newton's method; the special case of linear problems is included into this general framework.

The `main.cc` file

First of all, several header files have to be included that are needed during the solution process. We divide these includes into blocks corresponding to `DOpE` headers, `deal.II`

4 Examples for PDE Solution

headers, C++ headers and header files of the example itself (like the ones mentioned above).

Furthermore, we define names for certain objects via `typedef` which act as abbreviations in order to keep the code readable. In our case, these are `OP`, `IDC`, `INTEGRATOR`, `LINEARSOLVER`, `NLS`, `SSOLVER` and `STH`.

In the `int main()` function, we first create a possibility to read the parameter values from the *dope.prm* file. Then there are several standard steps for finite element codes like

- definition of a triangulation and create a grid object (which we read from the *channel.inp* file)
- creation of finite element objects for the state and the control and of quadrature formula objects

and in addition, we

- create objects of the `LocalPDE` class and of the different functional classes declared in the *functionals.h* file.

Remark 4.1.3. Up to now we have to create a pseudo time even for stationary problems. The

`MethodOfLines_StateSpaceTimeHandler` object (`DOFH`) which is needed for the initialization of `OP` requires a vector in which timepoints are specified. However, this is again merely a dummy variable, for we do not actually apply a time stepping method in the stationary case. This will also be removed in future versions of `DOpE`.

Before we initialize the `SSolver` object and actually solve the problem, we have to set the correct boundary conditions. Via the `compmask` vector, we ensure that the boundary conditions are set only for the velocity components of our solution vector. We set homogeneous Dirichlet values at the upper and lower boundaries of the channel. The inflow is described by a parabolic profile at the left boundary (the corresponding function class is declared in the *myfunctions.cc* file), whereas we do not prescribe anything at the outflow boundary (so-called do-nothing condition).

The output of the program (the two functional values) is rather unspectacular; as the problem is linear, the solution is computed within one Newton step.

4.1.2 Stationary FSI with INH Material

General problem description

In this example we consider a simple stationary FSI problem. The fluid is given as an incompressible Newtonian fluid modelled by the Stokes equation. Here, we use the symmetric stress tensor which has a little consequence when using the do-nothing outflow condition, see also section 4.1.1. The computation domain is $\Omega = [-6, 6] \times [0, 2]$ and we choose for simplicity $f = 0$.

The fluid reads:

Problem 4.1.4 (Variational fluid problem, Eulerian framework). *Find $\{v_f, p_f\} \in \{v_f^D + V\} \times L_f$, such that,*

$$\begin{aligned} (\rho_f v \cdot \nabla v_f, \phi^v)_{\Omega_f} + (\sigma_f, \nabla \phi^v)_{\Omega_f} &= \langle n_f \cdot g_s^\sigma, \phi^v \rangle_{\Gamma_i} \quad \forall \phi^v \in V_f, \\ (\operatorname{div} v_f, \phi^p)_{\Omega_f} &= 0 \quad \forall \phi^p \in L_f. \end{aligned}$$

The Cauchy stress tensor σ_f is given by

$$\sigma_f := -p_f I + \rho_f \nu_f (\nabla v_f + \nabla v_f^T), \quad (4.3)$$

with the fluid's density ρ_f and the kinematic viscosity ν_f . By n_f we denote the outer normal vector on Γ_i and by g_f^σ is a function which describes forces acting on the interface. These will be specified in the context of fluid-structure interaction models.

We define:

$$\hat{T} := \operatorname{id} + \hat{u}, \quad \hat{F} := I + \hat{\nabla} \hat{u}, \quad \hat{J} := \det(I + \hat{\nabla} \hat{u}).$$

The structure equations are given by incompressible neo-Hookean material

Problem 4.1.5 (Incompressible neo-Hookean Model (Lagrangian)).

$$\begin{aligned} (\hat{J}_s \hat{\sigma}_s \hat{F}_s^{-T}, \hat{\nabla} \hat{\phi}^v)_{\hat{\Omega}_s} &= \langle \hat{J}_s \hat{n}_s \cdot \hat{g}_s^\sigma \hat{F}_s^{-T}, \hat{\phi}^v \rangle_{\hat{\Gamma}_i} \quad \forall \hat{\phi}^v \in \hat{V}_s \\ (\hat{v}_s, \hat{\phi}^u)_{\hat{\Omega}_s} &= 0 \quad \forall \hat{\phi}^u \in \hat{V}_s, \\ (\det \hat{F}_s - 1, \hat{\phi}^p)_{\hat{\Omega}_s} &= 0 \quad \forall \hat{\phi}^p \in \hat{L}_s, \end{aligned}$$

where ρ_s is the solid's density, μ_s the Lamé coefficient, \hat{n}_s the outer normal vector at $\hat{\Gamma}_i$, \hat{g}_s^σ the force on the interface and with

$$\hat{\sigma}_s := -\hat{p}_s I + \mu_s (\hat{F}_s \hat{F}_s^T - I).$$

The resulting FSI problem is then given by:

Problem 4.1.6 (Stationary Fluid-Structure Interaction (ALE)).

$$\begin{aligned} (\hat{J} \rho_f \hat{F}^{-1} \hat{v} \cdot \hat{\nabla} \hat{v}, \hat{\phi}^v)_{\hat{\Omega}_f} + (\hat{J} \hat{\sigma}_f \hat{F}^{-T}, \hat{\nabla} \hat{\phi}^v)_{\hat{\Omega}_f} \\ + (\hat{J} \hat{\sigma}_s \hat{F}^{-T}, \hat{\nabla} \hat{\phi}^v)_{\hat{\Omega}_s} &= 0 \quad \forall \hat{\phi}^v \in \hat{V}, \\ (\hat{v}, \hat{\phi}^u)_{\hat{\Omega}_s} + (\alpha_u \hat{\nabla} \hat{u}, \hat{\nabla} \hat{\phi}^u)_{\hat{\Omega}_f} &= 0 \quad \forall \hat{\phi}^u \in \hat{V}, \\ (\widehat{\operatorname{div}}(\hat{J} \hat{F}^{-1} \hat{v}_f), \hat{\phi}^p)_{\hat{\Omega}_f} + (\hat{J} - 1, \hat{\phi}^p)_{\hat{\Omega}_s} &= 0 \quad \forall \hat{\phi}^p \in \hat{L}, \end{aligned}$$

Program description

There is not much difference to the prior example, so we will keep our remarks rather short. The *functionals.h* and *myfunctions.cc* files did not change at all, so we simply refer to the corresponding sections in the last example.

In the *localpde.h* file, all functions of the `LocalPDE` class have to be adjusted to the current FSI problem. This only makes the equations and matrices a little more complicated, and our solution vector now consists of five components (two velocity components of the fluid, the pressure component, and two additional displacement components for the structure variables). Otherwise, everything is analogous to the former example.

In the *main.cc* we only have to add two components to the `compmask` vector and prescribe boundary conditions for the structure variables. Apart from that, we define objects for the same classes as before that are even named equally and use the same solvers.

Again, the solution is reached within one Newton step, and all we see from the program output is the values of the functionals.

4.1.3 Stationary FSI with STVK Material

General problem description

This example is an extension the previous one. We solve an stationary FSI problem either with INH material (see Problem definition before) or St. Venant Kirchhoff material STVK:

Problem 4.1.7 (Compressible Saint Venant-Kirchhoff, Lagrangian framework). *Find $\{\hat{v}_f, \hat{u}_f\} \in \{\hat{v}_f^D + \hat{V}\} \times \{\hat{u}_f^D + \hat{V}\}$, such that*

$$\begin{aligned} (\hat{J}_s \hat{\sigma}_s \hat{F}_s^{-T}, \hat{\nabla} \hat{\phi}^v)_{\hat{\Omega}_s} &= \langle \hat{J}_s \hat{n}_s \cdot \hat{g}_s^\sigma \hat{F}_s^{-T}, \hat{\phi}^v \rangle_{\hat{\Gamma}_i} & \forall \hat{\phi}^v \in \hat{V}_s \\ (\hat{v}_s, \hat{\phi}^u)_{\hat{\Omega}_s} &= 0 & \forall \hat{\phi}^u \in \hat{V}_s, \end{aligned} \quad (4.4)$$

where ρ_s is the density of the structure, μ_s and λ_s the Lamé coefficients, \hat{n}_s the outer normal vector at $\hat{\Gamma}_i$, \hat{g}_s^σ some forces on the interface. The properties of the STVK material is specified by the constitutive law

$$\hat{\sigma}_s := \hat{J}^{-1} \hat{F} (\lambda_s (\text{tr} \hat{E}) I + 2\mu_s \hat{E}) \hat{F}^{-T}. \quad (4.5)$$

Often, the elasticity properties of structure materials is characterized by Poisson's ratio ν_s ($\nu_s < \frac{1}{2}$ for compressible materials) and the Young modulus E . The relationship to the Lamé coefficients μ_s and λ_s is given by:

$$\nu_s = \frac{\lambda_s}{2(\lambda_s + \mu_s)}, \quad E = \frac{\mu_s(\lambda_s + 2\mu_s)}{(\lambda_s + \mu_s)}. \quad (4.6)$$

The whole equation system is solved on the benchmark configuration domain. For details on parameters and geometry, we refer to the numerical FSI benchmark proposal from Hron and Turek [2006].

The code is established by computing the stationary FSI benchmark example FSI 1 with the following values of interest: x -displacement, y -displacement, drag, and lift.

Program description

Compared to the two former examples, there are some differences which we will briefly discuss in the following. First of all, the problem is nonlinear in contrast to the former ones. We work on a different domain (given in the *benchfst0100tw.inp* file), namely a channel with a cylinder put at half height near the inflow boundary; further *.inp* files yield the possibility to vary the domain.

Furthermore, in the *dope.prm* parameter file there are two additional subsections which are added only for the current problem. From the denotation of these subsections one can immediately see where in the code the parameters are used.

As we want to compute certain benchmark quantities, we have to regard corresponding functionals in the *functionals.h* file. The pressure at a point as well as the displacement in x - and y -directions are point values; furthermore we implement the drag and lift functionals (for which we need the additionally defined problem parameters).

4 Examples for PDE Solution

As before, we build up the cell and boundary equations and matrices in the *localpde.h* file. Apart from using the additionally defined problem parameters and modelling compressible STVK material instead of INH material (which leads to changes in the weak formulation of the equations), there are no major differences to the corresponding file in the last example.

In the *main.cc* file, we have to include additional header files from the deal.II library concerning error estimation and grid refinement. Further on, everything is pretty much the same as in the last example, but we have to use the `SetBoundaryFunctionalColors` function of the `PDEProblemContainer` class to be able to compute drag and lift in the respective functional classes in *functionals.h*.

The main innovation in contrast to the preceding examples is the refinement of the grid combined with a simple error estimator given in the deal.II `KellyErrorEstimator` class. If we look at the output of our program, everything is computed several times (once on each refinement level). Furthermore, we see that several Newton steps are needed on each refinement level; this is due to the nonlinearity of the current problem.

4.1.4 Stationary Elasticity Benchmark

General problem description

In this example we consider the following benchmark problem from elasticity theory:

$$(\sigma(u), \varepsilon(\varphi)) = (g, \varphi)_{\Gamma_N}. \quad (4.7)$$

Here $\tilde{\Omega}$ is a quadratic domain with side length 200 mm, where a circular hole with radius 10 mm around the center is cut out. Using symmetries of the domain, we restrict our actual computational domain Ω to the upper left quarter of $\tilde{\Omega}$.

In the above equation, $\varepsilon(v) := \frac{1}{2}(\nabla v + \nabla v^T)$ is the symmetric strain tensor, and

$$\sigma(v) := 2\mu\varepsilon(v)^D + \kappa \cdot \text{tr}(\varepsilon(v))I$$

denotes the symmetric stress tensor. Here τ^D is the deviatoric part of a tensor τ , in two dimensions defined as

$$\tau^D := \tau - \frac{1}{2}\text{tr}(\tau)I,$$

and the parameters μ and κ are chosen as $\mu = 80193.800283$ resp. $\kappa = 190937.589172$. The corner points of our computational domain are in anticlockwise order: $(0, 0)$, $(90, 0)$, $(100, 10)$, $(100, 100)$ and $(0, 100)$. We prescribe homogeneous Dirichlet boundary conditions in y -direction between $(0, 0)$ and $(90, 0)$ (lower boundary part), homogeneous Dirichlet boundary conditions in x -direction between $(100, 10)$ and $(100, 100)$ (right boundary part), and we interpret the righthand side of equation (1) with $g = 450$ as a boundary condition between $(0, 100)$ and $(100, 100)$ (upper boundary part).

The goal of our computations is to match the following functional reference values taken from *E. Stein (editor), Error-controlled Adaptive Finite Elements in Solid Mechanics, Wiley (2003), pp. 386 - 387*:

Functional	u_1 at $(90, 0)$	σ_{22} at $(90, 0)$	u_2 at $(100, 100)$
Reference value	0.021290	1388.732343	0.20951

Functional	u_1 at $(0, 100)$	$\int_{(100, 100)}^{(0, 100)} u_2$
Reference value	0.076758	20.40344

Program description

From the previous examples we know how to read a grid from an *.inp* file. The grid of our current example comes from the above mentioned benchmark problem. The parameter file (*dope.prm*) contains the same variables as in the first two examples.

Apart from different point values of derivatives of the solution, we want to evaluate an integral over part of the boundary. This is newly implemented in *functionals.h*.

The (linear) elasticity equation is much simpler than the FSI problems we treated before, which can be seen in *localpde.h* where the `CellEquation` and `CellMatrix` are

4 Examples for PDE Solution

implemented. In principle, everything is clear from the preceding examples.

The *main.cc* file does not contain any innovation, either. Here, we refine the grid globally instead of using an error estimator for local refinement.

The output of the program reflects again the linearity of the problem (only one Newton step is needed for solution).

4.1.5 Stationary Plasticity Benchmark

General problem description

Similar to the previous example, we consider the following benchmark problem from plasticity theory:

$$(\Pi(\sigma(u)), \varepsilon(\varphi)) = (g, \varphi)_{\Gamma_N}. \quad (4.8)$$

Here $\tilde{\Omega}$ is again the quadratic domain with a circular hole around the center cut out. Again, we restrict our actual computational domain Ω to the upper left quarter of $\tilde{\Omega}$ for reasons of symmetry.

We use the symmetric strain tensor $\varepsilon(v) := \frac{1}{2}(\nabla v + \nabla v^T)$, and the symmetric stress tensor σ is defined as

$$\sigma(v) := 2\mu\varepsilon(v)^D + \kappa \cdot \text{tr}(\varepsilon(v))I$$

where τ^D is the deviatoric part of a tensor τ , in two dimensions defined as

$$\tau^D := \tau - \frac{1}{2}\text{tr}(\tau)I.$$

The main difference with respect to the elastic case is the projection operator Π in equation (1). It is defined as follows:

$$\Pi(\tau) = \begin{cases} \tau & |\tau^D| \leq \sigma_0 \\ \sigma_0|\tau^D|^{-1}\tau^D + \frac{1}{2}\text{tr}(\tau)I & |\tau^D| > \sigma_0 \end{cases}$$

In our computations, we choose $\sigma_0 = \sqrt{\frac{2}{3}} \cdot 450$, and the above parameters μ and κ as $\mu = 80193.800283$ resp. $\kappa = 190937.589172$. The corner points of our computational domain are the same as before, and the boundary conditions are not altered, either.

The goal of our computations is to detect a subdomain in Ω where plastic behaviour occurs (compare *E. Stein (editor), Error-controlled Adaptive Finite Elements in Solid Mechanics, Wiley (2003), pp. 386 - 389*). This subdomain depends on the righthand side g in equation (1) which we write as $g = \lambda \cdot p$ with $p = 100$ and $\lambda \in [1.5; 4.5]$.

Program description

The code of the current example is nearly identical to the code of the previous one. The only difference worth mentioning is the change of the equations which leads to different implementations of the `CellEquation`, `CellMatrix` and `BoundaryEquations` functions in *localpde.h*.

Furthermore, the elasticity equations solved in the last example are linear, whereas the plasticity equations are nonlinear; this difference is evident also from the output (here, we need several Newton steps until convergence).

The functionals that appear in the output yield additional information and are not required in the above problem setting. The subdomain with plastic behaviour we want to detect can be visualized from the *.vtk* files written to the `Results/Mesh` subfolders.

4.1.6 Stationary Stokes Equations with periodic BC

General problem description

We solve the vector values laplace equation on a quadratic domain Ω with a circular hole in the middle, i.e. in strong formulation we look for $u = (u_1, u_2)$ s.t.

$$-\nabla \cdot \nabla u = f \quad \text{in } \Omega.$$

We set zero dirichlet values on the circular boundary in the middle of the domain and periodic boundary conditions on the other parts of the boundary. We choose the flux over the right hand side boundary as functional. We choose

$$f(x, y) = \left(\cos(\exp(10x)) y^2 x + \sin(y), \cos(\exp(10 * y)) x^2 y + \sin(x) \right)$$

for the right hand side.

Program description

This example show how to implement user defined DoF constraints. `DOpElib` has an interface for this, namely `UserDefinedDoFConstraints`. In our case, we derive the class `PeriodicityConstraints`, overwrite the method `MakeStateDoFConstraints` and give an instance of this class to `SpaceTimeHandler` at hand:

```
PeriodicityConstraints<DOFHANDLER, DIM> constraints_mkr;
STH DOFH(triangulation, state_fe);
DOFH.SetUserDefinedDoFConstraints(constraints_mkr);
```

This is all it takes. We refer to *myconstraintsmaker.h* for the details of the implementation of the periodicity-constraints.

4.1.7 Laplace Equation in 2D

General problem description

In this problem we solve the simple vector valued laplace equation in 2d on the unit square $\Omega = [0, 1]^2$, i.e. in strong formulation we look for $u = (u_1, u_2)$ s.t.

$$-\Delta u = f \quad \text{in } \Omega.$$

We set zero dirichlet values on $\partial\Gamma$ and choose $f = (1, 1)$. The classical example of a PDE.

4.1.8 Laplace Equation in 3D

General problem description

In this problem we solve the simple vector valued laplace equation in 3d on the unit square $\Omega = [0, 1]^3$, i.e. in strong formulation we look for $u = (u_1, u_2, u_3)$ s.t.

$$-\Delta u = f \quad \text{in } \Omega.$$

We set zero dirichlet values on $\partial\Gamma$ and choose $f = (1, 1, 1)$.

Program description

The PDE is discretized with Q3-elements on a series of locally refined grids (we use the `KellyErrorEstimator`). The algebraic equations are solved with different iterative linear solvers acting on different vector and matrix-structures (i.e. we use `dealii::BlockVector` and `dealii::Vector` plus the appropriate matrix classes).

To switch the linear solver is pretty easy since the newton solver has a template for the linear solver. Thus changing this template is all that is required.

To change the structure of the vectors and matrices involves also only the change of some template parameters. Our example programs are mostly build such that only a change of some typedefs is required, i.e. one has to interchange the lines

```
typedef SparseMatrix<double> MATRIX;
typedef SparsityPattern SPARSITYPATTERN;
typedef Vector<double> VECTOR;

with

typedef BlockSparseMatrix<double> MATRIX;
typedef BlockSparsityPattern SPARSITYPATTERN;
typedef BlockVector<double> VECTOR;
```

to switch between the block and non-block structures.

After solving the equation, we want to apply local mesh refinement. So first we extract with the help of the `SolutionExatractr`-class the vector `solution` representing the finite element solution

```
SolutionExtractor<SSolver1, VECTORBLOCK> a1(solver1);
const StateVector<VECTORBLOCK> &gu1 = a1.GetU();
solution = gu1.GetSpacialVector();
```

With this vector we estiamte the error via `KellyErrorEstimator` and get a vector holding the estimated error per cell. After choosing a refinement criterion (see *refinementcontainer.h*, we opt here for refining the top 20% of the cells), we give the `SpaceTimeHandler` an object of type `RefinementContainer` which holds all the information needed for the local mesh refinement. This is done via the `RefineSpace` method.

4 Examples for PDE Solution

```
DOFH1.RefineSpace(  
    RefineFixedNumber(estimated_error_per_cell, 0.2, 0.0));
```

This method transfers our solution onto the new mesh. The transferred solution is then taken as the starting guess of the newton method in the next solution cycle. This is especially helpful for nonlinear problems.

4.1.9 Adaptive Solution of Laplace Equation in 2D

General problem description

This example shows the use of the adaptive grid refinement and error estimation by the *DWR method* (For a description of the method, see [2].) applied to the laplace equation

$$-\Delta u = f \quad \text{in } \Omega$$

with the analytical solution

$$u = \sin\left(\frac{\pi}{x^2 + y^2}\right),$$

the corresponding right hand side $f = -\Delta u$ and appropriate Dirichlet Conditions on $\partial\Omega$, where the domain is given by

$$\Omega = [-2, 2]^2 \setminus \overline{B}_{0.5}(0).$$

We want to estimate the error in the following functional of interest

$$\begin{aligned} J &: H^1(\Omega) \longrightarrow \mathbb{R} \\ u &\longmapsto \frac{1}{|\Gamma|} \int_{\Gamma} u \, dx \end{aligned}$$

where $\Gamma = \{(x, y) \in \mathbb{R}^2 \mid x = 0, -2 < y < 0.5\}$.

For this setting, we have the error representation

$$J(e) = \sum_{K \in \mathbb{T}_h} \{(R_h, z - \psi_h)_K + (r_h, z - \psi_h)_{\partial K}\} \quad (4.9)$$

with the error $e = u - u_h$, the Triangulation \mathbb{T}_h , the dual solution z , arbitrary function $\phi_h \in V_h$ (the ansatz space) and the cell- and edge-residuals:

$$R_h|_K = f + \Delta u_h \quad (4.10)$$

resp.

$$r_h|_{\Sigma} = \begin{cases} \frac{1}{2}[\partial_n u_h], & \text{if } \Sigma \subset \partial_K \setminus \partial\Omega, \\ 0, & \text{if } \Sigma \subset \partial\Omega. \end{cases} \quad (4.11)$$

It holds $J(u) \approx 0.441956231972232$.

Program description

In this section we want to focus on what you have to do if you want to enhance your existing code to use the *DWR method*.

First, additionally to all the things one has to do when just solving the equation, we have to include the file

4 Examples for PDE Solution

`higher_order_dwrc.h`

As we approximate the so called 'weights' $z - \phi_h$ in the error representation by a patchwise higher order interpolation of z_h (the computed dual solution), we have to enforce patch-wise refinement of the grid by giving the flag

`Triangulation<2>::MeshSmoothing::patch_level_1`

to the triangulation.

To be able to solve the adjoint equation for the error estimation one needs to implement some methods regarding the equation as well as the functional of interest:

- In `pdeinterface.h`
 - `CellEquation_U`: Weak form of the adjoint equation.
 - `CellMatrix_T`: The FE matrix for the adjoint problem.
 - `FaceEquation_U`: This one is needed in this case here because we have a functional of interest that lives on faces.
- `functionalinterface.h`
 - `FaceValue_U`: This is the right hand side of the adjoint equation.

During the evaluation of (4.9), the following methods are needed

- `StrongCellResidual`: The cell residual, see (4.10).
- `StrongFaceResidual`: The terms in (4.9) that lies in the interior (i.e. the jumps).
- `StrongBoundaryResidual`: The terms in (4.9) that lies on the boundary (There are none in this case).

Note that in the above three functions we always apply the method `ResidualModifier` both to the residual as well as to the jumps on the faces. This is done to assert that we can apply both a DWR-error estimator where the residual should be multiplied with the computed weights (then this function does not do anything) as well as Residual Type error estimator for the L^2 or H^1 norm where we need to calculate element wise norms of the residual and the jumps. Then this function calculates the appropriate local terms, e.g., the square of the residual scaled with appropriate powers of the local mesh size.

After this, we tell the problem which functional we want to use for the error estimation, this is done via

`P.SetFunctionalForErrorEstimation(LFF.GetName())`

where `P` is of type `PDEProblemContainer` and `LFF` is the desired functional of interest.

The next thing we need is an object of the type

`HigherOrderDWRContainer`

4 Examples for PDE Solution

This container takes care of the computation of the weights.

To build this, we need the following:

- `DOFH_higher_order`: With some higher order Finite Elements and the already defined triangulation, we build this `SpaceTimeHandler`. This is needed because we want to use the patch-wise higher order interpolation of the weights.
- `idc_high`: A `IntegratorDataContainer` in which we put some (face)quadrature formulas for the evaluation of the error Identity.
- A string which indicates how we want to store the weight-vectors (here: `"fullmem"`).
- `pr`: The `ParameterReader` which we have already defined.
- A enum of type `EETerms` that tells the container, which error terms we want to compute (primal error indicators vs. dual error indicators, see [2]).

The last preparation step is now to initialize the `DWRDataContainer` with the problem in use:

```
solver.InitializeHigherOrderDWRC(dwrc);
```

Succeeding the solution of the state equation

```
solver.ComputeReducedFunctionals();
```

we compute the error indicators by calling

```
solver.ComputeRefinementIndicators(dwrc);
```

We can now get the error indicators (with signs!) out of `dwrc` by

```
dwrc.GetErrorIndicators();
```

With these indicators, we are now able to refine our grid adaptively (there are several mesh adaption strategies implemented, like `'RefineOptimized'`, `'RefineFixedNumber'` or `'RefineFixedFraction'`)

```
DOFH.RefineSpace(RefineOptimized(optimized, error_ind));
```

Note, that one has to take the norm of each entry in the vector of the error indicators before feeding them into the `RefineSpace` method.

4.1.10 Usage of Higher Order Mappings: Approximation of π

General problem description

This example shows the use higher order mappings in DOpElib. To this end, we solve a simple laplace equation

$$-\Delta u = -4 \quad \text{in } \Omega$$

with the analytical solution

$$u = x^2 + y^2,$$

and the Dirichlet Conditions on $u = 1$, where the domain is given by a circle with radius 1 and the center located in the origin.

We compute the L^2 -norm of the error and, additionally, we evaluate a functional which does not depend on the solution at all. We integrate the constant $\frac{1}{2}$ once over the boundary of the domain. The result is an approximation of π .

All this is standard and would not justify an additional example, however we solve the equation and the functional not one but two times. First with the standard Q1-mapping, the second time we use a higher order mapping. The exact order can be determined by the parameter file, the preset is Q2-mapping. At the end, we gather the errors and convergence rates over some refinement cycles in a nice table and notice the higher order of convergence for the higher order mapping solutions. This is due to the fact that we can approximate the circular boundary much better by a quadratic mapping.

Program description

In this section we want to focus on what you have to do if you want to enhance your existing code to use higher order mappings, which is actually pretty simple.

You have to include the file

`mapping_wrapper.h`

and create a mapping of the desired order by

```
DOPEWrapper::Mapping<dim, DOFHANDLER> mapping(order_of_mapping);
```

The last step is to give the mapping to the DoFHandler:

```
MethodOfLines_StateSpaceTimeHandler<FE, DOFHANDLER, SPARSITYPATTERN, VECTOR, 2>
DOFH(triangulation, mapping, state_fe);
```

The rest of the program is as usual.

4.1.11 Stationary Stokes Equations with hp-Elements

General problem description

In this example we consider the same setting as in subsection 4.1.1, the only difference is that we want to employ the hp-Finite-Elements. So the equation we solve is still the stationary incompressible Stokes equation. Here, we use the symmetric stress tensor which has a little consequence when using the do-nothing outflow condition. In strong formulation we have

$$\begin{aligned} -\frac{1}{2}\nabla \cdot (\nabla v + \nabla v^T) + \nabla p &= f \\ \nabla \cdot v &= 0 \end{aligned} \tag{4.12}$$

on the domain $\Omega = [-6, 6] \times [0, 2]$. We split $\partial\Omega = \Gamma_D \cup \Gamma_{out}$. The right hand side of the channel is Γ_{out} on which we describe the free outflow condition, on the rest of the boundary we prescribe dirichlet values (An parabolic inflow on the left hand side and zero on the upper and lower channel walls). We choose for simplicity $f = 0$.

Adding hp-Elements

One sees by comparing the `main.cc`-file of this problem with the one of subsection 4.1.1 that the change to hp-Elements is really easy, so we will keep the description short. In comparison to example 4.1.1 the `localpde.h` and `functionals.h` have not changed, but we have one additional file, namely `indexsetter.h`, in which the class `ActiveFEIndexSetter` is defined.

In the hp-framework we have a stack of finite elements (a `hp::FECollection`) given. We assign each cell of the triangulation an fe-index which determines which finite element we use on this cell. The `ActiveFEIndexSetter` class manages these indices, see there for more information.

The changes in `main.cc` are also minimal and are highlighted in the source code. Obviously, we use `FECollection` and `QCollection` as well as a different `DoFHandler`.

```
#define DOFHANDLER hp::DoFHandler
#define FE hp::FECollection
...
typedef hp::QCollection<DIM> QUADRATURE;
typedef hp::QCollection<DIM - 1> FACEQUADRATURE;
```

Apart from that we have only to tell the space time handler the distribution of the finite element indices:

```
ActiveFEIndexSetter<2> indexsetter(pr);
STH DOFH(triangulation, state_fe_collection, indexsetter);
```

4.2 Nonstationary PDEs

Until now, the DOpE provides various time-stepping schemes that are based on finite differences. Specifically, the user can choose between the

- Forward Euler scheme (FE), which is an explicit timestepping scheme. Here, one has to take into account that $k \leq ch^2$ where k denotes the timestep size and h the local mesh cell diameter.
- Backward Euler scheme (BE), which is an implicit timestepping scheme. It is strongly A-stable but only from first order and very dissipative. The BE-scheme is well suited for stationary numerical examples.
- Crank-Nicolson scheme (CN), which is of second order, A-stable, has very little dissipation but suffers from case to case from instabilities caused by rough initial- and/or boundary data. These properties are due to weak stability (it is not *strongly* A-stable).
- Shifted (or stabilized) Crank-Nicolson scheme (CN shifted), which is also of second order, but provides global stability.
- Fractional-step- θ scheme (FS). It has second-order accuracy and is strongly A-stable, and therefore well-suited for computing solutions with rough data.

4.2.1 Nonstationary Navier-Stokes Equations

General problem description

In this example we consider the nonstationary incompressible Navier-Stokes equation. As in the stationary PDE Example 3 4.1.3, we use the symmetric fluid stress tensor, i.e. in strong formulation we deal with

$$\begin{aligned} \partial_t v - \nabla \cdot (\nabla v + \nabla v^T) + (v \cdot \nabla)v + \nabla p &= f \\ \nabla \cdot v &= 0 \end{aligned}$$

on time interval $I = [0, T]$ and the domain $\Omega = [-6, 6] \times [0, 2]$. For simplicity, we set $f = 0$.

As introduced earlier, we formulate the time stepping scheme as *One-step- θ scheme*, which are based on finite difference schemes. In order to keep the presentation simple, we describe the scheme using the stokes equation and thus neglecting the nonlinearity. Note that in the program we use the full Navier-Stokes operator.

The time interval is given by $I = [0, T]$. Let v^n, p^n and the time step $k = t^{n+1} - t^n$ be given. Find v^{n+1}, p^{n+1} such that:

$$\begin{aligned} v^{n+1} - k\theta \left(\nabla \cdot (\nabla v^{n+1} + \nabla v^{n+1T}) + \nabla p^{n+1} \right) &= k\theta f^{n+1} + k(1 - \theta)f^n \\ &\quad + v^n + k(1 - \theta) \left(\nabla \cdot (\nabla v^n + \nabla (v^n)^T) + \nabla p^n \right) \\ \nabla \cdot v^{n+1} &= 0 \end{aligned}$$

4 Examples for PDE Solution

In the case of the BE-scheme, $\theta = 1$, and the equation is reduced to

$$\begin{aligned} v^{n+1} - k(\nabla \cdot (\nabla v^{n+1} + \nabla v^{n+1T}) + \nabla p^n) &= k\theta f^{n+1} + v^n \\ \nabla \cdot v^{n+1} &= 0 \end{aligned}$$

Note, that one should prefer a complete implicit treatment of the pressure p . Instead of using $\theta p^{n+1} + (1 - \theta)p^n$, the pressure appears only with θp^{n+1} .

After discretization in time, the space is treated, as ususally, with a Galerkin finite element scheme, here based on the Taylor-Hood element Q_2^c/Q_1^c .

The variational formulation reads:

Problem 4.2.1 (Backward Euler (BE) timestepping problem). *Let $\theta = 1$. Find $v := v^{n+1} \in V$ and $p := p^{n+1} \in L$:*

$$(v, \phi^v) + k\theta(\nabla v + \nabla v^T, \nabla \phi^v) - k(p, \nabla \cdot \phi^v) = k\theta(f^{n+1}, \phi^v) + (v^n, \phi^v) \quad (4.13)$$

$$(\nabla \cdot v, \phi^p) = 0 \quad (4.14)$$

for all suitable test functions $\phi^v, \phi^p \in V \times L$.

Derivation of the other timestepping problems is analogous.

Remark 4.2.2. Note that because of the zero right hand side we are allowed to multiply (4.14) by $k\theta$. So that we solve

$$k\theta(\nabla \cdot v, \phi^p) = 0$$

instead of $(\nabla \cdot v, \phi^p) = 0$.

Specific features for solving nonstationary problems

In the following, we explain in more detail the different member functions that are required to implement nonstationary equations.

```
void CellEquation (... , double scale, double scale_ico)
```

The two arguments are used to distinguish between explicit components and fully implicit components. For standard equations (such as the heat equation and the wave equation), there is no special treatment required needed.

However, solving the Navier-Stokes equations or multi-physics problems (like fluid-structure interaction), parts of the equations are treated with a fully implicit time-stepping scheme.

Thus, the argument

```
double scale
```

is used to indicate that the present term can be used for implicit/explicit or mixed discretization (such as time discretization with the Crank-Nicolson).

The other argument

```
double scale_ico
```

4 Examples for PDE Solution

is used to indicate that the present term only is treated in a fully implicit manner. For example, the pressure term (which is of course a Lagrange multiplier of the incompressibility term of the fluid). It is recommended to treat this term in a time discretization in a fully implicit manner.

```
void CellMatrix (... , double scale, double scale_ico)
```

The directional derivatives of the state equation are implemented in the present function. As before, the last two parameters

```
double scale, double scale_ico
```

are used to distinguish between fully implicit and other behavior.

```
void CellTimeEquation (...)
```

This function is used to implement the time derivative in weak formulation

$$(\partial_t v, \phi)_\Omega.$$

This term is time discretized via

$$k^{-1}(v^n - v^{n-1}, \phi)_\Omega.$$

Here, it suffices to implement the term

$$(v^n, \phi)_\Omega,$$

because the already known term v^{n-1} is automatically treated by the specific time stepping scheme.

In contrast to this behavior, the user has the possibility to write all terms of $\partial_t v$ explicitly. In this case, we use the

```
void CellTimeEquationExplicit (...)
```

and we write

$$(v^n - v^{n-1}, \phi)_\Omega.$$

This behavior is useful for multi-physics problems where other solution variables have to be considered around $\partial_t v$. The user should have a look in the second Example 4.2.2 for nonstationary problems for an illustration of this function.

Consequently, the directional derivatives of the cell terms are implemented in the corresponding matrix functions, i.e.,

```
void CellTimeMatrix (...), void CellTimeMatrixExplicit
```

4.2.2 Nonstationary FSI Problem

General problem description

In the present example, we solve a nonstationary fluid-structure interaction problem. The underlying equations are stated in the following:

Problem 4.2.3 (Variational fluid-structure interaction framework). *Find $\{\hat{v}, \hat{u}, \hat{p}\} \in \{\hat{v}^D + \hat{V}^0\} \times \{\hat{u}^D + \hat{V}^0\} \times \hat{\mathcal{L}}$, such that $\hat{v}(0) = \hat{v}^0$ and $\hat{u}(0) = \hat{u}^0$, for almost all time steps t , and*

$$\begin{aligned} & (\hat{J}\hat{\rho}_f\partial_t\hat{v}, \hat{\psi}^v)_{\hat{\Omega}_f} + (\hat{\rho}_f\hat{J}(\hat{F}^{-1}(\hat{v} - \partial_t\hat{u}) \cdot \hat{\nabla})\hat{v}), \hat{\psi}^v)_{\hat{\Omega}_f} \\ & + (\hat{J}\hat{\sigma}_f\hat{F}^{-T}, \hat{\nabla}\hat{\psi}^v)_{\hat{\Omega}_f} + (\hat{\rho}_s\partial_t\hat{v}, \hat{\psi}^v)_{\hat{\Omega}_s} + (\hat{J}\hat{\sigma}_s\hat{F}^{-T}, \hat{\nabla}\hat{\psi}^v)_{\hat{\Omega}_s} \\ & - \langle \hat{g}, \hat{\psi}^v \rangle_{\hat{\Gamma}_N} - (\hat{\rho}_f\hat{J}\hat{f}_f, \hat{\psi}^v)_{\hat{\Omega}_f} - (\hat{\rho}_s\hat{f}_s, \hat{\psi}^v)_{\hat{\Omega}_s} = 0 \quad \forall \hat{\psi}^v \in \hat{V}^0, \\ & (\partial_t\hat{u} - \hat{v}, \hat{\psi}^u)_{\hat{\Omega}_s} + (\hat{\sigma}_g, \hat{\nabla}\hat{\psi}^u)_{\hat{\Omega}_f} - \langle \hat{\sigma}_g\hat{n}_f, \hat{\psi}^u \rangle_{\hat{\Gamma}_i} = 0 \quad \forall \hat{\psi}^u \in \hat{V}^0, \\ & (\widehat{div}(\hat{J}\hat{F}^{-1}\hat{v}_f), \hat{\psi}^p)_{\hat{\Omega}_f} + (\hat{p}_s, \hat{\psi}^p)_{\hat{\Omega}_s} = 0 \quad \forall \hat{\psi}^p \in \hat{L}, \end{aligned}$$

with $\hat{\rho}_f$, $\hat{\rho}_s$, ν_f , μ_s , λ_s , \hat{F} , and \hat{J} . The stress tensors for the fluid and structure are implemented in $\hat{\sigma}_f$, $\hat{\sigma}_s$, and $\hat{\sigma}_g$

Code validation for fluid problems with ALE

With the ALE code implemented in Example 4.1.3 it is possible to treat fluid problems as well as FSI computations. In the case of fluid problems the deformation gradient and its determinant become:

$$F := I, \quad \det F = J = 1.$$

The code is validated by the well-known fluid- and FSI benchmark problems. Here, the results have been summarized in the table below.

Configuration	Date	Refinement	Time dis.	k	Δp	F_D	F_L
BFAC 2D-1	Mar 10, 2010	2 global	BE	1.0	0.117412		
BFAC 2D-1	Mar 10, 2010	2 global	FS	1.0	0.117412		
BFAC 2D-1	Mar 10, 2010	2 global	CN(k)	1.0	0.117412		
BFAC 2D-1	Mar 10, 2010	2 global	CN	1.0	0.117383		
BFAC 2D-2	Mar 10, 2010	2 global	CN(k)	1.0	2.31541		
BFAC 2D-2	Mar 10, 2010	3 global	CN(k)	1.0	2.32011		
BFAC 2D-2	Mar 12, 2010	3 global	CN(k)	1.0e-2	2.50288		

Code validation for FSI with ALE

The results are summarized below:

Program description

4 Examples for PDE Solution

Config.	Date	Refinement	Time dis.	k	$u_x(A)[\times 10^{-5}]$	$u_y(A)[\times 10^{-4}]$	F_D	F_L
FSI 1	Mar 12, 2010	3 global	BE	1.0	8.2003	2.2732		
FSI 1	Apr 08, 2010	2 global	BE	1.0	8.2258	2.2813		
FSI 1	Apr 08, 2010	2 global	CN(k)	0.5	8.2268	2.2813		
FSI 1	Apr 08, 2010	2 global	CN	0.5	8.2268	2.2813		

4.2.3 Black-Scholes Equation

General problem description

The problem under consideration is the so called multivariate Black-Scholes equation arising from pricing European style options in finance.

To state the general form of the equation we need some nomenclature: We consider an option on d risky assets with *maturity* $T > 0$ and *strikeprice* $K > 0$. For the sake of simplicity we assume the *interest rate* $r > 0$ and the *volatility* of the i -th asset $\sigma_i > 0$, $1 \leq i \leq d$, to be constant. Besides, we assume the matrix $\rho = (\rho_{ij})$ of the *correlation factors* ρ_{ij} with $-1 \leq \rho_{ij} \leq 1$ for $1 \leq i, j \leq d$, to be positive definite. Of course ρ is symmetric with $\rho_{ii} = 1$.

With $(t, x) \in I = (0, T] \times \mathbb{R}_+^d$ denoting the prices of the underlying assets at time t , the problem of determining the fair price u of such an option is (after a time reversal) given by the following equation:

$$\partial_t u - \frac{1}{2} \sum_{i,j=1}^d \sigma_i \sigma_j \rho_{ij} x_i x_j \partial_{x_i} \partial_{x_j} u - r \sum_{i=1}^d x_i \partial_{x_i} u + ru = 0 \quad \text{in } (0, T] \times \mathbb{R}_+^d, \quad (4.15a)$$

$$u(0) = u_0 \quad \text{in } \mathbb{R}_+^d. \quad (4.15b)$$

The initial condition $u_0 \in C^0(\mathbb{R}_+^d)$ (i.e. the *payoff*) is given depending of the type of the option. For example

$$u_0 := \begin{cases} \max(\sum_{i=1}^d \lambda_i x_i - K, 0), & u \text{ is a } Call, \\ \max(K - \sum_{i=1}^d \lambda_i x_i, 0), & u \text{ is a } Put, \end{cases} \quad (4.16)$$

for a plain vanilla European option on a basket of assets containing a share of $0 < \lambda_i \leq 1$ of the i -th asset. For the computation, we truncate the domain, i.e. we choose $\bar{x} \in \mathbb{R}_+^d$ and consider the computational domain $\Omega := (x_1, \bar{x}_1) \times \cdots \times (x_d, \bar{x}_d)$. On the new part of the boundary Γ with $\Gamma := \{x \in \partial\Omega \mid \exists 1 \leq i \leq d, x_i = \bar{x}_i\}$ we impose asymptotic values as dirichlet conditions. For a put, we take $u|_\Gamma = 0$. We emphasize that no boundary conditions will be imposed on $\partial\Omega \setminus \Gamma$.

In this particular example we examine the case of two uncorrelated stocks (with $\lambda_1 = \lambda_2 = \frac{1}{2}$) and the following parameters:

	2d-Put)
actual asset value x_0	(25,25)
strikeprice K	25
maturity date T	1
volatility σ	$(\frac{1}{2}, \frac{3}{10})$
cutoff \bar{x}	(100, 100)
interest rate r	0,05
option value $u(T, x_0)$	ca. 2,269172389

Program description

4.2.4 Heat Equation in 1D

General problem description

In this example we consider one of the prototypical instationary equations, the parabolic heat equation

$$\begin{aligned}\partial_t u(t, x) - \Delta u(t, x) &= f(t, x), \\ u(t, x)|_{\partial\Omega} &= g(t, x), \\ u(0, x) &= u_0(x)\end{aligned}$$

with unknown solution $u : \Omega \rightarrow \mathbb{R}$, where $\Omega \subset \mathbb{R}^d$. In our example, we consider the simplest case $d = 1$, where the Laplacian Δ reduces to ∂_x^2 . The computational domain is $\Omega \times I = [0, 1] \times [0, 1]$. For further simplification, we choose the righthand side as $f = 0$ as well as homogeneous Dirichlet boundary conditions ($g = 0$). The initial condition is given by $u_0(x) = \min(x, 1 - x)$.

Program description

There are few new things compared to the first two nonstationary examples. This is the first time we solve an equation in one spatial dimension. In most cases, the dimension dependence is covered by the `LOCALDOPEDIM` and `LOCALDEALDIM` variables (which are defined at the beginning of the `main.cc` file), but there might be some places in the code (especially your own code) where a concrete dimension number is given to an object. There you have to replace it manually. Do not forget to insert the correct dimension in the `Makefile`!

The most important feature of this example is the serial application of several time-stepping schemes. At the moment, the following schemes are available (see also example 4.2.1):

1. Forward Euler scheme (FE)
2. Backward Euler scheme (BE)
3. Crank-Nicolson scheme (CN)
4. shifted Crank-Nicolson scheme (sCN)
5. Fractional-Step- θ scheme (FS)

All these time-stepping methods are applied in the current example in order to check them and to compare their characteristics. To keep the computing time acceptable, we choose a one dimensional example.

One more innovation is the output format. We want to represent the output at single timepoints as a function graph on the space interval $[0, 1]$; this can be done using `GNU-PLOT`, for example, so instead of `.vtk` files as in all former examples, we now write out `.gpl` files.

4.2.5 Heat Equation in 2D with nonlinearity

General problem description

This example differs only slightly from the previous one. Again, we consider the heat equation, this time with an additional nonlinear term

$$\partial_t u(t, x, y) - \Delta u(t, x, y) + u(t, x, y)^2 = f(t, x, y),$$

but now in two space dimensions and with known solution

$$u(t, x, y) = e^{t-t^2} \sin(x) \sin(y).$$

The computational domain is $\Omega \times I = [0, \pi]^2 \times [0, 1]$. From the known solution, we can compute the appropriate data

$$\begin{aligned} f(t, x, y) &= (3 - 2t)e^{t-t^2} \sin(x) \sin(y) + e^{(t-t^2)^2} \sin^2(x) \sin^2(y), \\ u_0(x, y) &= \sin(x) \sin(y). \end{aligned}$$

Furthermore, we have to prescribe homogeneous Dirichlet boundary conditions.

Program description

The new feature of this example is the nonhomogeneous righthand side. In examples 4.1.7 and 4.1.8, we regarded stationary problems with nonhomogeneous righthand sides, but up to now, we never involved the time variable into the nonhomogeneity. To do this, `D0pElib` yields a `SetTime()` function which has to be applied in the *localpde.h* file as well as at the place where the `RightHandSideFunction` class is declared (here the *myfunctions.h* file).

5 Examples with Optimization

5.1 Subject to a Stationary PDE

5.1.1 Distributed control with a linear elliptic PDE

General problem description

This example solves the distributed minimization problem

$$\begin{aligned} \min J(q, u) &= \frac{1}{2} \|u - u^d\|^2 + \frac{\alpha}{2} \|q\|^2 d \\ \text{s.t. } (\nabla u, \nabla \phi) &= (q + f, \phi) \quad \forall \phi \in H_0^1(\Omega) \end{aligned}$$

on the domain $\Omega = [0, 1]^2$, and the data is chosen as follows:

$$\begin{aligned} f &= \left(20\pi^2 \sin(4\pi x) - \frac{1}{\alpha} \sin(\pi x) \right) \sin(2\pi y) \\ u^d &= \left(5\pi^2 \sin(\pi x) + \sin(4\pi x) \right) \sin(2\pi y) \end{aligned}$$

and $\alpha = 10^{-3}$. Hence its solution is given by:

$$\begin{aligned} \bar{q} &= \frac{1}{\alpha} \sin(\pi x) \sin(2\pi y) \\ \bar{u} &= \sin(4\pi x) \sin(2\pi y). \end{aligned}$$

Thus the exact optimal value of the cost functional can be calculated as

$$J^* = J(\bar{q}, \bar{u}) = \frac{1}{8} \left(25\pi^4 + \frac{1}{\alpha} \right).$$

In addition the following functionals are evaluated:

$$\text{MidPoint: } u(0.5; 0.5)$$

$$\text{MeanValue: } \int_{\Omega} u$$

The example shows how to estimate the error in the cost functional for stationary optimization problems.

Program description

5.1.2 Parameter control with a linear elliptic PDE

General problem description

This example solves the following pointwise minimization problem

$$\begin{aligned} \min_{(q,u) \in \mathbb{R}^3 \times H_0^1(\Omega; \mathbb{R}^2)} J(q, u) &= \frac{1}{2} \sum_{i=0}^2 |(u - \bar{u})(x_i)|^2 + \frac{\alpha}{2} \|q\|^2 \\ \text{s.t. } (\nabla u, \nabla \phi) &= (f(q), \phi) \quad \forall \phi \in H_0^1(\Omega; \mathbb{R}^2) \end{aligned}$$

on the domain $\Omega = [0, 1]^2$, with zero dirichlet boundary conditions and

- the observation points

$$x_0 = (0.5, 0.5), \quad x_1 = (0.5, 0.25), \quad x_2 = (0.25, 0.25),$$

- the regularization parameter $\alpha = 0$,
- the right hand side

$$\begin{aligned} f(q) &= q_0 \begin{pmatrix} 2\pi^2 \sin(\pi x) \sin(\pi y) \\ 0 \end{pmatrix} \\ &+ q_1 \begin{pmatrix} 5\pi^2 \sin(\pi x) \sin(2\pi y) \\ 0 \end{pmatrix} \\ &+ q_2 \begin{pmatrix} 0 \\ 8\pi^2 \sin(2\pi x) \sin(2\pi y) \end{pmatrix} \end{aligned}$$

- and the exact solution given by

$$\begin{aligned} \bar{q} &= (1; 0.5; 1) \\ \bar{u} &= \begin{pmatrix} \sin(\pi x)(\sin(\pi y) + 0.5 \sin(2\pi y)) \\ \sin(2\pi x) \sin(2\pi y) \end{pmatrix}. \end{aligned}$$

Program description

Notice that the cost functional is of mixed type (from our computational point of view), i.e. the first part is a pointfunctional whereas the regularization part requires the evaluation of a domain integral. To handle this, we need a special integrator as well as a special newtonsolver. Additionally, our `LocalFunctional` returns as his type:

```
string
GetType() const
{
    return "point domain";
}
```

5 Examples with Optimization

This brings along that we have not only to implement

- `Value`,
- `Value_U`,
- `Value_Q`,
- `Value_UU`,
- `Value_UQ`,
- `Value_QU`,
- `Value_QQ`

from `FunctionalInterface`, but also all the aforementioned methods with a preceding `Point` (`PointValue` etc.).

5.1.3 Parameter control with a nonlinear PDE from fluid dynamics

General problem description

In this example, we solve a optimization problems from fluid dynamics. The configuration is similar to the fluid optimization problem proposed by Roland Becker in 2000 “Mesh adaption for stationary flow control”.

We describe the underlying equations later...

Program description

5.1.4 Control in the dirichlet boundary values

General problem description

This example solves the minimization problem

$$\begin{aligned} \min J(q, u) &= \frac{1}{2} \|u - u^d\|^2 + \frac{\alpha}{2} \|q\|^2 \\ \text{s.t. } (\nabla u, \nabla \phi) &= (f, \phi) \quad \forall \phi \in H_0^1(\Omega; \mathbb{R}^2) \end{aligned}$$

on the domain $\Omega = [0, 1]^2$. In addition we set the dirichlet data of the state on the boundary as follows

$$\begin{aligned} u_0(0, y) &= q_0, & u_0(1, y) &= q_1, & u_0(x, 0) &= q_2, & u_0(x, 1) &= q_3, \\ u_1 &= q_4^3. \end{aligned}$$

The data is chosen as follows:

$$\begin{aligned} f &= \begin{pmatrix} 20\pi^2 \sin(\pi x) \sin(\pi y) \\ 1 \end{pmatrix} \\ u^d &= \begin{pmatrix} \sin(\pi x) \sin(\pi y) * x \\ x \end{pmatrix} \end{aligned}$$

with $\alpha = 10$.

Program description

The control in the dirichlet boundary values is incorporated via the class `LocalDirichletData` which is defined in the file `localdirichletdata.h`. The class is then given to the `OptProblemContainer` as a template argument. This is all that is needed to use the control in the dirichlet boundary values.

5.1.5 Distributed Control with Different Meshes for Control and State

General problem description

This example solves the distributed minimization problem

$$\begin{aligned} \min J(q, u) &= \frac{1}{2} \|u - u^d\|^2 + \frac{\alpha}{2} \|q\|^2 \\ \text{s.t. } (\nabla u, \nabla \phi) &= (q + f, \phi) \quad \forall \phi \in H_0^1(\Omega) \end{aligned}$$

on the domain $\Omega = [0, 1]^2$, and the data is chosen as follows:

$$\begin{aligned} f &= \left(20\pi^2 \sin(4\pi x) - \frac{1}{\alpha} \sin(\pi x) \right) \sin(2\pi y) \\ u^d &= \left(5\pi^2 \sin(\pi x) + \sin(4\pi x) \right) \sin(2\pi y) \end{aligned}$$

and $\alpha = 10^{-3}$. Hence its solution is given by:

$$\begin{aligned} \bar{q} &= \frac{1}{\alpha} \sin(\pi x) \sin(2\pi y) \\ \bar{u} &= \sin(4\pi x) \sin(2\pi y) \end{aligned}$$

In addition the following functionals are evaluated:

$$\begin{aligned} \text{MidPoint: } u_h(0.5; 0.5) & \qquad \text{L1-Value: } \int_{\Omega} |u_h| \\ \text{QError: } \int_{\Omega} |q_h - \bar{q}|^2 & \qquad \text{UError: } \int_{\Omega} |u_h - \bar{u}|^2 \end{aligned}$$

The important new feature is that we can now use two different meshes for control and state variable. This is tested first for globally refined meshes, and then for locally refined meshes with different refinements for the control and state variable.

Program description

In order to use different meshes for control and state we need to use the multimesh variants of `CellDataContainer`, `FaceDataContainer` and `Integrator` and we have to choose a space time handler capable of managing multiple meshes, so we use

`MethodOfLines_MultiMesh_SpaceTimeHandler`.

The requirement for the control and state mesh is that they have a common coarse grid, so the space time handler gets only one mesh (to ensure a common coarse grid), but this gets copied internally so that we have two separate meshes for control and state. We can then separately refine the mesh for control and state (see `RefineControlSpace` and `RefineStateSpace`).

5.1.6 Compliance Minimization of a variable Thickness MBB-Beam

General problem description

This example implements the minimum compliance problem for the thickness optimization of an MBB-Beam. Using the MMA-Method of K. Svanberg together with an augmented Lagrangian approach for the subproblems following M. Stingl.

5.1.7 Distributed control with a linear elliptic PDE using IPOPT/SNOPT

General problem description

This example solves the distributed minimization problem

$$\begin{aligned} \min J(q, u) &= \frac{1}{2} \|u - u^d\|^2 + \frac{\alpha}{2} \|q\|^2 \\ \text{s.t. } (\nabla u, \nabla \phi) &= (q + f, \phi) \quad \forall \phi \in H_0^1(\Omega) \\ \text{s.t. } -500 &\leq q \leq 500 \text{ a.e. in } \Omega \end{aligned}$$

on the domain $\Omega = [0, 1]^2$, and the data is chosen as follows:

$$\begin{aligned} f &= \left(20\pi^2 \sin(4\pi x) - \frac{1}{\alpha} \sin(\pi x) \right) \sin(2\pi y) \\ u^d &= \left(5\pi^2 \sin(\pi x) + \sin(4\pi x) \right) \sin(2\pi y) \end{aligned}$$

and $\alpha = 10^{-3}$.

In addition the following functionals are evaluated:

$$\text{MidPoint: } u(0.5; 0.5)$$

$$\text{MeanValue: } \int_{\Omega} u$$

The Problem is similar to that of `OPT/StatPDE/Example1` except for the box control constraints. The problem is solved using the optimization library IPOPT that you can obtain for free. To use it a correct link to the ipopt library needs to be created in `D0pE/ThirdPartyLibs` by the name `ipopt`, i.e., you should have the file `D0pE/ThirdPartyLibs/ipopt` pointing to the ipopt directory. If you have not done this you can compile the example but when running the example you will only get an error message like

```
Warning: During execution of 'Reduced_IpoptAlgorithm::Solve'
the following Problem occurred!
To use this algorithm you need to have IPOPT installed!
To use this set the WITH_IPOPT CompilerFlag.
```

Alternatively the commercial optimization library SNOPT can be used in this example. In order to use this library you need to install SNOPT on your computer and then generate a symlink to the snopt directory (where you have the libs and the header files) in the `D0pE/ThirdPartyLibs` directory named `snopt`, i.e., you should have the file `D0pE/ThirdPartyLibs/snopt` pointing to the snopt directory. If you have not done this you can compile the example but when running the example you will only get an error message like

```
Warning: During execution of 'Reduced_SnoptAlgorithm::Solve'
the following Problem occurred!
To use this algorithm you need to have SNOPT installed!
To use this set the WITH_SNOPT CompilerFlag.
```

5.1.8 Topology optimization of an MBB-Beam using SNOPT

General problem description

This example implements the topology optimization of an MBB-Beam given in `OPT/StatPDE/Example6` using the SIMP method.

The solution is computed using the commercial optimization library SNOPT as in `OPT/StatPDE/Example7`. This Example demonstrate how global constraints on the control variable can be included into the optimization call.

5.1.9 Parameter control with a non-linear PDE from FSI dynamics

General program description

In this example we solve an optimization problem where the equations come from fluid-structure interaction (FSI). The idea is to extend the steady FSI benchmark problem (FSI 1, proposed by Hron/Turek) to an optimization problem where the drag is minimized over the cylinder and the beam.

The example has many similarities to Opt Example 3.

Detailed description will follow later.

5.2 Subject to a Non-Stationary PDE

5.2.1 Control of a nonlinear heat equation via the initial values

General problem description

This example is a modified version of PDE/InstatPDE/Example5. Again, we consider the heat equation, this time with an additional nonlinear term

$$\partial_t u(t, x, y) - \Delta u(t, x, y) + u(t, x, y)^2 = f(t, x, y),$$

with homogeneous dirichlet-data. The computational domain is $\Omega \times I = [0, \pi]^2 \times [0, 1]$. From the known solution, we can compute the appropriate data

$$\begin{aligned} f(t, x, y) &= (3 - 2t)e^{t-t^2} \sin(x) \sin(y) + e^{(t-t^2)^2} \sin^2(x) \sin^2(y), \\ u_0(x, y) &= q(x, y). \end{aligned}$$

With the cost functional

$$\min_{q, u} J(q, u) = \frac{1}{2} \int_{\Omega} (u(1, x, y) - \sin(x) \sin(y))^2 d(x, y) + \frac{1}{2} \int_{\Omega} (q(x, y) - \sin(x) \sin(y))^2 d(x, y).$$

It has the optimal solution

$$\begin{aligned} \bar{u}(t, x, y) &= e^{t-t^2} \sin(x) \sin(y), \\ \bar{q}(t, x, y) &= \sin(x) \sin(y) \end{aligned}$$

and $J(\bar{q}, \bar{u}) = 0$.

The following new things differ from the PDE-Examples and the optimization examples with stationary PDEs:

In the file `main.cc` the `SpaceTimeHandler` now gets an additional argument. In this case `DOpEtypes::initial` which specifies that the control is entering in the initial value.

In the file `localpde.h` we now have to specify the Methods `Init_CellRhs` and `Init_CellRhs_Q`. They need to be adapted, since usually the `InitialValue` for the PDE is auto generated from a `deal.ii` function in the `ProblemContainer`. This Value is set in the `Init_CellRhs` hence we need to change this function to use the control instead. Correspondingly we need to implement the first derivative of this with respect to the control. –We don't need the second derivative since it is zero anyways.–

Note: In contrast to the „normal“ Cell-terms in the PDE we assume in the program that the `Init_CellEquation` is linear in the state and no other solution variables are present. Thus no derivatives of the `Init_CellEquation` need to be implemented.

6 Testing examples

The DOpE test suite consists of 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. In the basic example directory

6.1 Where can I find the tests

In each example directory you find a sub directory 'Test'. Herein, you find the parameter files for meshes (*.inp) and a param file (test.prm). Moreover, the executable is denoted by 'test.sh'. Please make sure, that the

```
set never_write_list
```

contains every possible output

```
Gradient;Hessian;Tangent;Residual;Update;Control;State
```

That means, no solution files are written to the output. Recall, that we are just interested in terminal output that is of course sufficient to verify the things.

Hence, the results directory should be empty

```
set results_dir = ./
```

The rest in the param file must be identically the same as in the dope.prm file in the parent directory.

6.2 How to start testing?

You start testing by typing

```
> ./test.sh Store
```

in the terminal.

After the run, you have to call

```
> ./test.sh Test
```

to compare your stored output. Of course, there should be no differences.

The useful point is now the following. After implementation of new pieces of code in the DOpE library or in the examples, you can run

6 *Testing examples*

```
> ./test.sh Test
```

Hereby, you compare your ‘new’ output with the previous stored output.

Attention: After changes you should NOT run again

```
> ./test.sh Store
```

In that case, you overwrite your previous output.

Bibliography

- [1] Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. *Differential Equations Analysis Library*, 2010.
- [2] Wolfgang Bangerth and Rolf Rannacher. *Adaptive Finite Element Methods for Differential Equations*. Birkhäuser Verlag, 2003.
- [3] R. Becker, D. Meidner, and B. Vexler. *RODOBO: A C++ library for optimization with stationary and nonstationary PDEs*, 2005.
- [4] 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.
- [5] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer, US, 2nd edition, 2000.

Index

- Cauchy stress tensor, 26
- cost functional, 23
- deviator, 30
- fluid, 26
 - Eulerian framework, 26
 - incompressible, 26
 - Newtonian, 26
- fluid-structure interaction (FSI), 26, 28
 - ALE model, 26
 - FSI benchmark, 28
 - interface, 26
- functional, 23, 28, 30
 - boundary flux, 23
 - boundary integral, 30
 - deflection, 28
 - drag, 28
 - lift, 28
 - point value, 23
- functional evaluation, 23
- grid, 22
- instationary PDE, 47
 - Black-Scholes equation, 47
- Lamé coefficients, 28
- Newton's method, 24
- parameters, 22
- Poisson's ratio, 28
- stationary PDE, 22, 26, 30, 41
 - elasticity equations, 30
 - FSI problem, 26
 - Stokes equation, 22, 41
- strain tensor, 30
- structure, 26, 28
 - compressible St.Venant-Kirchhoff (STVK) material, 28
 - incompressible neo-Hookean (INH) material, 26
 - Lagrangian framework, 26
- vector-valued problem, 23
- Young modulus, 28