DOpElib: Differential Equations and Optimization Environment

Thomas Wick*, Winnifried Wollner†

April 18, 2017

*École Polytechnique, thomas.wick@polytechnique.edu
†Technische Universität Darmstadt, wollner@mathematik.tu-darmstadt.de

Contents

1 Foreword

In this report, we describe the DOpElib *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 main feature of DOpElib is to give a unified interface to high level algorithms such as time-stepping methods, nonlinear solvers and optimization routines. DOpElib is designed in such a way that the user only needs to write those parts of the code that are problem dependent while all invariant parts of the algorithms are reusable without any need for further coding. In particular, the user is enabled to switch between various different algorithms without the need to rewrite the problem dependent code, though obviously he or she will have to replace the algorithm object with an other one. This replacement can be done by replacing the appropriate object at only one point in the code. In addition to the finite element code provided by deal.II –which at present is the only FE-toolkit to which we provide an interface– the presented library DOpElib is user-focused by delivering prefabricated tools which require adjustments by the user only for parts connected to his specific problem. This is in contrast to deal.II which leaves the implementation of all high-level algorithms to the user.

An innovative feature of DOpElib is to provide a software toolkit to solve forward PDE problems as well as optimal control problems constrained by PDE. DOpElib 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. 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 multiphysics problems such as 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.
- Self-written line search and trust region newton algorithms for the solution of optimization problems with PDEs [12]
- 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 and goal-oriented error estimation with the dual-weighted residual (DWR) method.
- Different spatial triangulations for control and state variables.
- Several examples showing the solution of several PDEs including Poisson, Navier-Stokes, plasticity, fluid-structure interaction problems, a coupled Biot-Lamé-Navier system, and finally from financial mathematics the Black-Scholes equations.

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 4. 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 5 and those for the solution of optimization problems with PDEs in Chapter 6.

These notes conclude with a section that explains how we do automated testing of the implementation in Chapter 7. 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.

Thanks: The `DOpElib` 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] and now maintained by [2]. Special thanks go to them!

In addition to deal.II, the authors gratefully acknowledge their past experience as well as discussions with the authors of the software toolkits `Gascoigne` and `RoDoBo` [10] and [14], from which some of the ideas to modularize the algorithms have arisen. Similar thanks go to the developers of `Ipopt`, [16].

Last, but not least, we would like to express our gratitude to all contributors listed in Section 2.4 for their respective contribution to the library.

Special thanks go to Christian Goll who has been a great help in maintaining and developing this library from 2009 until 2015.

2 Introduction

2.1 How to get DOpElib

There are two ways to obtain a copy of DOpElib:

A) You can obtain a copy of DOpElib from the developers git repository using
`git clone git://git.mathematik.tu-darmstadt.de/dopelib`
in the command line of your terminal.

B) You can download the sources as a tar-ball from the project website
`http://www.dopelib.net`.

2.2 License information

2.3 References

If you like the library and use it for your own projects, please give credits by referencing the project `DOpElib` [9] using the following bibtex entry:

```
@MISC{dope,
  key    = {DOpElib},
  title  = {The {D}ifferential {E}quation and
            {O}ptimization {E}nvironment: \textsc{DOpElib}},
  url    = {http://www.dopelib.net},
  note   = {\texttt{http://www.dopelib.net}}
}
```

2.4 Contributors & Developers

The library is currently maintained by

- Thomas Wick (Centre de Mathématiques Appliquées (CMAP) École Polytechnique)
- Winnifried Wollner (Technische Universität Darmstadt)

We would like to express our gratitude to the former maintainer

- Christian Goll (Maintainer from 2009–2015)

Furthermore, there are more highly appreciated contributions made by

- Daniel Jodlbauer (current work on parallelization)
- Bernhard Endtmayer (currently working on phase-field fracture)
- Michael Geiger (Examples for Plasticity, and Documentation of several PDE-Examples)
- Masoud Ghaderi (Augmenting the Documentation)
- Uwe Köcher (Makefile compatibility)
- Francesco Ludovici (Augmenting the Documentation)
- Matthias Maier (CMake)

2.5 Software requirements

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

2.5.1 g++

DOpElib requires a recent g++ (at least 4.8) due to some new C++ features implemented in 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 for instance sudo port install gcc48 to install version 4.8 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.

2.5.2 deal.II

This library is mainly based upon deal.II hence in order to run DOpElib 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 recommendable to be roughly acquainted with deal.II.

When installing deal.II (at least version 8.0) you should take care to configure it to use UMFPACK.

Remark 2.5.1. Our current DOpE installation has been successfully tested for deal.II version 8.5.0.

A configuration of deal.II working with DOpElib can be obtained with the following commands assuming that one is currently in the directory where the deal.II sources are located.

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/path/to/install/dir ../deal.II
make install
```

If in addition, features are desired corresponding flags need to be passed to cmake. An excellent description is provided on the deal.II webpage

```
http://www.dealii.org/8.5.0/readme.html
```

Currently, for all features of DOpElib the following settings are useful to be passed to cmake

```
cmake -DCMAKE_INSTALL_PREFIX=[path to install dir] \
    -DDEAL_II_WITH_UMFPACK=true \
    -DDEAL_II_FORCE_BUNDLED_UMFPACK=true \
    -DDEAL_II_WITH_TRILINOS=true \
    -DTRILINOS_DIR=[path to trilinos] \
    -DDEAL_II_WITH_MPI=true \
    -DDEAL_II_WITH_P4EST=true \
    -DP4EST_DIR=[path to p4est] \
    [Path to deal.II sources]
```

2.5.3 deal.II ThirdPartyLibraties

To install third party libraries used for deal.II we recommend to use the candi script:

https://github.com/koecher/candi

On this webpage, you will find also enough information how to use the candi script.

2.5.4 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 with enabled UMFPACK.

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.

The installation is straighforward and has been tested on openSUSE 12.1 machines as well as on MAC. At the end of the installtion do not forget to add ipopt to your `LD_LIBRARY_PATH`:

```
*************************************************************
                  Installation complete!
Add /home/..../dopelib-4.0/ThirdPartyLibs/ipopt/lib64
    to your LD\_LIBRARY\_PATH variable
*************************************************************
```

2.6 Installation

To work with DOpElib, your need necessarily deal.II, which installation we describe first. Afterwards, the DOpElib installation is described and finally other optional packages might be installed.

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

  If you prefer another position you can install it anywhere but need to set the `DEAL_II_DIR` environment variable, so that it will be found by cmake, i.e.,

  ```
  export DEAL_II_DIR=[path to deal.ii]
  ```

  Remark 2.6.1. **Be careful** if you work in several terminal windows and have several deal.II versions. In this case, the export-command might have to be done again in that new window. Otherwise another deal.II version might be linked resulting in linker errors or segmentation faults without useful error message. In order to check to which deal.II version DOpE is linked, type

  ```
  echo $DEAL_II_DIR
  ```

  in the current terminal window.

  Detailed installing instructions for deal.II (here the last version 8.4.0) can be found on `http://www.dealii.org/8.4.0/readme.html`. The deal.II installation instructions are descriptive enough and we omit any further comments and refer to their webpage.

- Get a copy of `DOpElib`, see Section 2.1 for details.

- Unpack in your preferred directory:

  ```
  ~/Software/> tar -xzf dopelib-4.0.tar.gz
  ```

  to get

  ```
  ~/Software/dopelib-4.0
  ```

  To build the `DOpElib` you have to change to the directory `DOpEsrc` where you can call

  ```
  ~/Software/dopelib-4.0/DOpEsrc> make c-all
  ```

to build and configure the library using cmake. You also get various installation options by just typing `make`.

- To generate documentation, please go into:

  ~/Software/dopelib-4.0/Examples

  Here, you get all options by typing:

  make

  Specifically, you can create and run all examples, running a test suite, and generating documentation:

```
==========================================================================
=              Makefile for the DOpE documentation
=
==========================================================================
=
=
= The following targets exist:
=
=    c-all      :  Make all examples using cmake
=
=    clean      :  Cleaning up all examples
=
=    tests      :  Run all test param data.
=
=    c-cat      :  Run clean, run c-all, run tests (combine these commands)=
=    doc        :  Create documentation in pdf file format via latexmk
=
=    distclean  :  Cleaning up, including documentation
=
=    warncheck  :  Checks whether all Examples compile without warnings
=
==========================================================================
```

  The options are $c-all$ and $c-cat$ utilizing the cmake build system.

  It is **important** to generate the examples via make c-all in the home folder of the examples and **not** as in deal.II by using cmake . in the specific example itself. With this you would currently destroy the delivered Makefile. See also the FAQ below.

  For html documentation, change into:

  ~/Software/dopelib-4.0/doxygen

  and then typing `make`. Doxygen documentation will require either latex or doxygen to be installed on your computer.

- If you 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 7.

- 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.

  As example, the installation of ipopt works as follows:
  In the path `/dope/ThirdPartyLibs>` type in the terminal
  `./install-free-libs.sh`

2.7 FAQs

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

- **unrecognized command line option "-std=c++11"**
  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 `4.8` 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 `deal.II` 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 `deal.II`

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. You can do so, for instance using macports. Then you can find, e.g., g++ version 4.8 on OSX Lion in `/opt/local/bin/g++-mp-4.8`.

3.) Problem using make c-all in the Examples folder :

```
wick@linux:~/Software/dopelib/Examples> make c-all
cd OPT/StatPDE/Example1; make -s c-all
make[2]: *** No targets specified and no makefile found.  Stop.
../../../../Examples/Make.global_options:2: recipe for target 'c-all' failed
make[1]: *** [c-all] Error 2
Makefile:24: recipe for target 'OPT/StatPDE/Example1' failed
make: *** [OPT/StatPDE/Example1] Error 2
```

This means that the Makefile in the specific examples folder has been modified in some illegal way. Remove this Makefile

```
rm Makefile
rm -r CMakeFiles
rm CMakeCache.txt
rm -r autobuild
```

and get the original one from DOpElib:

```
git checkout Makefile
```

Then try again make c-all (make always sure that before typing make c-all that the deal.II path is exported - see above).

3 The Structure of DOpElib

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.

3.1 Problem description

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

- `pdeproblemcontainer.h` Is used to describe stationary PDE problems.
- `instatpdeproblemcontainer.h` 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

3.2 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.

3.2.1 Space-time handler

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

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

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

3.2.2 Container classes

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

- `container/elementdatacontainer.h` This object is used to pass data given on the current element 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_elementdatacontainer.h` This is the same as the elementdatacontainer, but it is capable to handle data defined on an alternative triangulation.
- `container/multimesh_facedatacontainer.h` This is the same as the facedatacontainer, but it is capable to handle data defined on an alternative triangulation.
- `container/integratordatacontainer.h` This contains some data that should be passed to the integrator like quadrature formulas and the above element 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 'optimized' strategy).

3.2.3 Time stepping schemes

Third, at least for nonstationary PDEs we need to select a time stepping scheme the file names of which are mostly self explanatory:

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

3.2.4 Integrator routines

Finally, we need to select a way how to integrate and solve linear and nonlinear equations

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

3.2.5 Nonlinear solvers

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

3.2.6 Linear solvers

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

- templates/voidlinearsolver.h This is a wrapper for certain cases when we know that the matrix to be inverted is the identity. It simply copies the right hand side to the left hand side. This is only needed for compatibility reasons some other components.

3.3 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/simpledirichletdata.h.

3.4 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/ipopt_problem.h` A wrapper file required when solving optimization problems using the reduced_ipopt_algorithm. This file hides the interface to IPOPT.

3.5 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.)

3.6 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.

3.6.1 Vectors

- `include/statevector.h` This stores all dofs in space and time for the state variable u. It is possible to select whether all this should be kept in memory or or unused parts can be written to the hard disk.
- `include/controlvector.h` This stores all dofs in space and time for the control variable q. At present no time dependence is implemented.
- `include/constraintvector.h` This stores all dofs in space and time for the non PDE constraints (and corresponding multipliers). At present no time dependence is implemented.

Remark 3.6.1. We notice that the behavior of the statevector can be chosen as `fullmem`, `only_recent`, or `store_on_disc`. In the first state, the RAM memory of the computer is used. In the second state, only the spatial vectors at the current time step (and the preious one) are stored. This reduces memory requirements, but also prohibits access to the whole space-time trajectory after the computation. In the third state, all vectors are written on disc, to avoid the RAM. This might take some time at the beginning of a new executing program (cleary depending on the number of spatial and temporal unknowns and the capabilities of your local machine). In addition, if the program aborts abnormally in the using `store_on_disc` behavior, please make sure to remove manually the `tmp_state` folder in your `Results` folder.

3.6.2 Parameter handling

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

3.6.3 Exception handling

- `include/dopeexception.h` Defines some Exceptions that are thrown by the program should it encounter any unexpected errors.
- `include/dopeexceptionhandler.h` This class is used to write information contained in the exceptions to the output in a uniform manner.

3.6.4 Output handling

- `include/outputhandler.h` This file defines an outputhandler object which can be used to decide whether some information should be written to screen or file. In

addition it can format output according to some run time parameters given by a parameter file.

3.7 Data Access

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

3.7.1 Constraints and system matrix

- `include/userdefineddofconstraints.h` This class sets the constraints on the DOFs of the state and/or control FE solution. DOpE itself builds the hanging-node-constraints, but the user can reimplement this class and thus include other constraints as well (for example periodic BC). Note, that the hanging-node-constraints come first (in case of conflicting constraints.)
- `include/sparsitymaker.h` This class sets the sparsity pattern for the state FE solution. The standard implementation is just a wrapper for `dealii::DoFTools::make_sparsity_pattern`, but the user can reimplement this class to allow for more sophisticated sparsity patterns.
- `include/pointconstraintsmaker.h` This class allows to set homogeneous Dirichlet values at given points/components.

3.7.2 HP components

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

3.8 Internal structures

3.8.1 Interface Classes

- `interfaces/transposeddirichletdatainterface.h` This provides an interface to the functionality required by *transposed Dirichlet data*. Usually when one applies Dirichlet data g to a function one has to calculate a continuation Bg which is defined on the whole domain. In optimization problems when the Dirichlet data depends on the control one has to evaluate the dual operator B^* in order to obtain a representation for the reduced gradient of the objective J. This is done using the *transposed Dirichlet data*.
- `interfaces/reducedprobleminterface.h` In order to allow all algorithms to be written independent of the given (OPT) problem (and not requiring the problem as template argument) there is a common base class which defines the required interfaces.

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

3.8.2 Default Classes

- problemdata/noconstraints.h A class that can be used for optimization problems having only a PDE constraint but no further constraints.
- problemdata/simpledirichletdata.h A class that can be used to implement Dirichlet data that are given as a fixed function (independent of the control).

3.8.3 Auto-generated Problem Descriptions

- problemdata/stateproblem.h This is the problem description for the (forward/primal) PDE constraint. **Similar descriptors will be build for the other problems (adjoint, tangent, ...) when time allows.**
- problemdata/initialproblem.h This is the problem descriptor to compute the finite element representation of the initial values. This is generated by the different time-stepping schemes based upon the defined representation by the PDE, which is set to the component wise L^2 projection by default.
- problemdata/primaldirichletdata.h This class contains the Dirichlet data for the forward/primal PDE.
- problemdata/tangentdirichletdata.h This class contains the Dirichlet data for the tangent PDE, i.e., the first derivative of the Dirichlet data.
- problemdata/transposedgradientdirichletdata.h This contains the transposed Dirichlet data needed to calculate the gradient of the reduced objective functional, for detail see interfaces/transposeddirichletdatainterface.h.
- problemdata/transposedhessiandirichletdata.h This contains the transposed Dirichlet data needed to calculate the hessian of the reduced objective functional, for detail see interfaces/transposeddirichletdatainterface.h.

3.8.4 Management of Time Dependent Problems

- include/timedofhandler.h DoFHandler responsible for the management of the timedofs (this is a part of the SpaceTimeDoFHandler-classes). Basically a wrapper for a $1d$ deal.II-DoFHandler.
- include/timeiterator.h This class works as an iterator on the TimeDoFHandler.

3.9 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.
- 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 FORTRAN library SNOPT. 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.

3.10 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 and 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.

4 Example Handling, Creating new Examples

4.1 Getting started

Beside the fact that DOpElib is still under development, it offers already various different (linear and nonlinear) examples for a lots of different applications in two and three dimensions; we refer the reader to the next two Chapters 5 and 6.

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. Here is some advice to get started:

- If you are a first time user of DOpElib with some numerics background, you might be familiar with the Poisson (or more general Laplace) equation. DOpElib has it too. Check-out Example 5.1.4, to see how DOpElib implements this well-known equation in two dimensions or 5.1.6 for its three-dimensional version.
- Before you implement a new example, please check which existing example might be similar to your goals and get familiar to it. Then, proceed as described in Section 4.4.

4.2 How to run existing examples

4.2.1 The global way

The easiest way is to first build all examples. Go into

```
dopelib/Examples/
```

Herein, type

```
make c-all
```

By typing only 'make' you will see all options (we also refer to Section 2.6). This procedure will take some minutes. Afterwards, go into the examples folder of your choice. For instance:

```
dopelib/Examples/PDE/InstatPDE/Example1/autobuild
```

Herein you create the executable via

```
make
```

and then go back via ../ and

```
./DOpE-PDE-InstatPDE-Example1
```

Or both commands together:

```
autobuild> make && cd .. && ./DOpE-PDE-InstatPDE-Example1
```

You will find the results on the screen in the terminal as well as some graphical output in the 'Results' folder.

4.2.2 Building, making and running in the local folder

In case we have run first globally as previsously described, in each examples folder we find an autobuild subfolder. If we now want to work locally (which is the usual way) then we have to type 'make' in the autobuild folder:

```
autobuild> make
```

In case we need to modify the Makefile, we need first to do:

```
autobuild> cmake ..
```

For instance one reason to modify the Makefile could be to change to debug mode as described in Section 4.3. Finally, go one folder back and run the object file.

In case you work and test out new things (modifiying the main.cc and localpde.h files etc.), one command for instance in Example 9 is:

```
Example9> cd autobuild/ && make && cd .. && ./DOpE-PDE-StatPDE-Example9
```

4.3 Changing from Release mode to Debug mode

In a specific example go into the autobuild folder and change therein:

```
cmake CMAKE_BUILD_TYPE=Release ..
```

to some other behavior:

```
Debug Release RelWithDebInfo MinSizeRel
```

Then type 'make' to build the executable file and go back into the parent folder and execute the object file.

4.4 Creating new examples

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. **New: in git from March 2017** In a first step, we copy `Example1` and renamed it, e.g., `MyWonderfulFirstExample`. 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 MyWonderfulFirstExample
cd MyWonderfulFirstExample
```

2. **Old: in the svn up to version 8.3** In a first step, we copy `Example1` and renamed it, e.g., `MyWonderfulFirstExample`. 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 MyWonderfulFirstExample
   cd MyWonderfulFirstExample
   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`.

3. You will find a file `CMakeLists.txt` in the folder `MyWonderfulFirstExample`. Open this file, in it you can find the line

   ```
   SET(TARGET "DOpE-PDE-StatPDE-Example1")
   ```

   the string `DOpE-PDE-StatPDE-Example1` will be the name of the executable of your program. Change it to something suitable, e.g.,

   ```
   SET(TARGET "MyWonderfulFirstExample")
   ```

   Moreover, this file contains the lines

   ```
   SET(dope_dimension 2)
   SET(deal_dimension 2)
   ```

   which define the dimension of the domain for the control-variable (`dope_dimension`) and the PDE solution (`deal_dimension`). If for your example one of these differs from 2 adjust the number accordingly. This file will not need any further modifications.

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
   ```

   If above you have changed the dimensions, make sure to adjust all files accordingly!

5. The cmake system can - in principle be used "in-source" to create the executable. However, this may break some of the automated tests (later), so you are encouraged to proceed differently. To do this end proceed as follows in the directory `MyWonderfulFirstExample`:

   ```
   MyWonderfulFirstExample$ mkdir build
   MyWonderfulFirstExample$ cd build
   MyWonderfulFirstExample/build$ cmake -DCMAKE_BUILD_TYPE=Release ../
   ```

   which will configure the build (if you want to debug it is useful to replace the string `Release` with `Debug`). Once this is done, you can compile and run the code:

   ```
   MyWonderfulFirstExample/build$ make
   MyWonderfulFirstExample/build$ cd ..
   MyWonderfulFirstExample$ ./MyWonderfulFirstExample
   ```

   (Assuming that no errors occurred during the make call)

6. The `Makefile` in the directory is present only to preserve backward compatibility. If you wish to use the automated build/test routines in DOpE lib, you need to make sure that it its configured correctly. If you just want to use the cmake capabilities you can safely ignore this passage.

   Furthermore, if you have followed the instructions above, then no changes will be needed.

   If you have moved the directory to some other place, i.e., it is not in the same folder as Example1, then open the `Makefile` in the directory `MyWonderfulFirstExample` you will find the line

   ```
   DOpE = ../../../../
   ```

   which points to the root directory of your DOpE installation. Adjust the path to match accordingly.

7. If you want to run automated tests on you 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 7. 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 the line

```
PROGRAM=../DOpE-PDE-StatPDE-Example1-2d-2d
```

if you made a copy of an other example the part `DOpE-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

```
PROGRAM=../MyWonderfulFirstExample
```

8. 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.sh Test
   ```

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

   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.sh Store
   ```

5 Examples for PDE Solution

5.1 Stationary PDEs

5.1.1 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

$$-\frac{1}{2}\nabla \cdot (\nabla v + \nabla v^T) + \nabla p = f \tag{5.1}$$
$$\nabla \cdot v = 0$$

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 $f = 0$.

As code verification, we evaluate two different types of functionals. First a point functional measuring the x-velocity and a flux functional

$$\int_{\Gamma_{out}} v \cdot n \, ds,$$

on the outflow boundary. Both a described in the `functionals.h` file as described below.

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, elements 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.

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 element equation, the element matrix and element right hand side as well as the boundary equation, boundary matrix and boundary right hand 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 `DOpElib` all classes are derived from a so-called `FunctionalInterface` class.
As already mentioned previsously, 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 element and boundary equations, matrices and right hand sides. The weak formulation of problem (5.1) with $f = 0$ is

$$\frac{1}{2}(\nabla v, \nabla \varphi)_\Omega + \frac{1}{2}(\nabla v^T, \nabla \varphi)_\Omega - (p, \nabla \cdot \varphi)_\Omega + (\nabla \cdot v, \psi)_\Omega - (n \cdot \nabla v^T, \phi)_{\Gamma_{out}} = 0. \quad (5.2)$$

Remark 5.1.1. 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.
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 left hand side is implemented in the `ElementEquation` function, the right hand side is implemented in the `ElementRightHandSide` function (which is unused in this example, because $f = 0$).

Remark 5.1.2. 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 left hand side, we have the derivative of the (semilinear) form $a(\cdot;\cdot)$ with respect to the solution variable u, and the right hand side is the residual of the weak formulation:

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

In the `ElementMatrix` function, we implement the following matrix A as representation of the derivative on the left hand side:

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

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

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

and the `ElementRightHandSide` 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 (5.2), 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 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 5.1.3. 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.

5.1.2 Laplace equation 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

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

for the right hand side. As code verification, the mass flux on one boundary part is evaluated.

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.

5.1.3 Stationary Stokes Equations with hp-Elements

General problem description

In this example we consider the same setting as in subsection 5.1.1, 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

$$-\frac{1}{2}\nabla \cdot (\nabla v + \nabla v^T) + \nabla p = f \tag{5.3}$$
$$\nabla \cdot v = 0$$

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 5.1.1 that the change to hp-Elements is really easy, so we will keep the description short. In comparison to example 5.1.1 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 element of the triangulation an fe-index which determines which finite element we use on this element. 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);
```

5.1.4 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.

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

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

Remark 5.1.4 (Why this example?). Originally, `DOpElib` was designed for coupled and nonlinear problems with possible PDE-based optimization extensions. Later, we decided to add the most simplest PDE (the Laplace/Poisson equation) in order to demonstrate how DOpE treats this well-known example. In addition, the first-time user might start here to get a feeling for `DOpElib` and its capabilities.

5.1.5 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 [3].) 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

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

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

For this setting, we have the error representation

$$J(e) = \sum_{K \in \mathbb{T}_h} \left\{(R_h, z - \psi_h)_K + (r_h, z - \psi_h)_{\partial_K}\right\} \tag{5.4}$$

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 element- and edge-residuals:

$$R_h\big|_K = f + \Delta u_h \tag{5.5}$$

resp.

$$r_h\big|_\Sigma = \begin{cases} \frac{1}{2}[\partial_n u_h], & \text{if } \Sigma \subset \partial_K \setminus \partial\Omega, \\ 0, & \text{if } \Sigma \subset \partial\Omega. \end{cases} \tag{5.6}$$

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

`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 es the functional of interest:

- In `pdeinterface.h`
  - `ElementEquation_U`: Weak form of the adjoint equation.
  - `ElementMatrix_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 (5.4), the following methods are needed

- `StrongElementResidual`: The element residual, see (5.5).
- `StrongFaceResidual`: The terms in (5.4) that lies in the interior (i.e. the jumps).
- `StrongBoundaryResidual`:The terms in (5.4) 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`

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 [3]).

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.

5.1.6 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 estimate the error via `KellyErrorEstimator` and get a vector holding the estimated error per element. After choosing a refinement criterion (see *refinementcontainer.h*, we opt here for refining the top 20% of the elements), 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.

```
DOFH1.RefineSpace(
       RefineFixedNumber(estimated_error_per_element, 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.

5.1.7 Stationary Elasticity Benchmark

General problem description

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

$$(\sigma(u), \varepsilon(\varphi)) = (g, \varphi)_{\Gamma_N}. \tag{5.7}$$

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

$$\sigma(v) := 2\mu\varepsilon(v)^D + \rho\, tr(\varepsilon(v))I = 2\mu\varepsilon(v) + \lambda tr(\varepsilon(v))I,$$

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

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

and the parameters μ and ρ are chosen as $\mu = 80193.800283$ resp. $\rho = (\mu + \lambda) = 190937.589172$. We notice that μ and λ denote the usual Lamé parameters.
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 right hand 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.
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*.
In principle, everything is clear from the preceding examples. 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).

5.1.8 Stationary Plasticity Benchmark

General problem description

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

$$(\Pi(\sigma(u)), \varepsilon(\varphi)) = (g, \varphi)_{\Gamma_N}. \tag{5.8}$$

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

$$\sigma(v) := 2\mu\varepsilon(v)^D + \rho\, tr(\varepsilon(v))I = 2\mu\varepsilon(v) + \lambda tr(\varepsilon(v))I,$$

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

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

Furthermore, the (standard) Lamé parameters are denoted by μ and λ and which are more conveniently (here and in the code) expressed through $\rho = \mu + \lambda$ and $\kappa = 2\mu + \lambda$. The main difference with respect to the elastic case is the projection operator Π in equation (1). It is defined as follows:

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

In our computations, we choose $\sigma_0 = \sqrt{\frac{2}{3}} \cdot 450$, and the above parameters μ, λ and ρ as $\mu = 80193.800283, \lambda = 110743.788889$, and $\rho = 190937.589172$, respectively. 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 behavior occurs (compare *E. Stein (editor), Error-controlled Adaptive Finite Elements in Solid Mechanics, Wiley (2003), pp. 386 - 389*). This subdomain depends on the right hand 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 `ElementEquation, ElementMatrix` 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 behavior we want to detect can be visualized from the *.vtk* files written to the `Results/Mesh` subfolders.

5.1.9 Stationary FSI with Stokes and INH Material

General problem description

In this example we consider a simple stationary FSI problem. The fluid is given as an incompressible Newtonian fluid modeled 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 5.1.1. The flow is driven by non-homogeneous Dirichlet condition on the left bounddary.

The computation domain is $\Omega = [-6, 6] \times [0, 2]$ and we choose for simplicity $f = 0$. We add in the subdomain $\Omega_s = [0, 2] \times [0, 1]$ a solid obstacle. In this solid we prescribe an incompressible neo-Hookean material law.

The fluid reads:

Problem 5.1.5 (Variational fluid problem, Eulerian framework)**.** *Find* $\{v_f, p_f\} \in \{v_f^D + V\} \times L_f$*, such that,*

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

The Cauchy stress tensor σ_f *is given by*

$$\sigma_f := -p_f I + \rho_f \nu_f (\nabla v_f + \nabla v_f^T), \tag{5.9}$$

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:

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

The structure equations are given by incompressible neo-Hookean material

Problem 5.1.6 (Incompressible neo-Hookean Model (Lagrangian))**.**

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

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

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

Remark 5.1.7. At our developer meeting on Apr 13, 2017, we also added a simplified STVK material for testing purposes. This STVK material is implemented in a different header file (see also further comments below).

The resulting FSI problem is then given by:

Problem 5.1.8 (Stationary Fluid-Structure Interaction (ALE))**.**

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

Remark 5.1.9. In the problems above and the code, we implement the term

$$(\hat{v}_s, \hat{\phi}^u),$$

although this is not physically necessary. It is first for computational convenience in order to extend the fluid velocity variable to the whole domain. This could be resolved by using the FE Nothing element. Second, using $\hat{v}_s$ here makes it easier to understand the nonstationary FSI problem.

Program description

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.

We also demonstrate another very convenient feature. Since we have two different material laws (INH and simplified STVK), we do not implement them together in one *localpde.h* file by using if-conditions etc. But we implement them seperately in two different *.h files, namely

```
localpde.h
localpde_stvk_material.h
```

In the `main.cc` function we can now simply comment or uncomment the respective file we want to work with. This allows us to keep a clean file for a running example and experiment in other files (possibly more than 2) and by just changing two lines in the main.cc in order to change the equations.

5.1.10 Stationary FSI with Navier-Stokes and 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 5.1.10 (Compressible Saint Venant-Kirchhoff, Lagrangian framework)**.** *Find* $\{\hat{u}_s\} \in \{\hat{u}_s^D + \hat{V}\}$*, such that*

$$(\hat{J}_s \hat{\sigma}_s(\hat{u}_s)\hat{F}_s^{-T}, \hat{\nabla}\hat{\phi}^v)_{\hat{\Omega}_s} = \langle \hat{J}_s \hat{n}_s \cdot \hat{g}_s^\sigma \hat{F}_s^{-T}, \hat{\phi}^v \rangle_{\hat{\Gamma}_i} \qquad \forall \hat{\phi}^v \in \hat{V}_s \tag{5.10}$$

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* $\hat{\Gamma}_i$*,* $\hat{g}_s^\sigma$ *some forces on the interface. The properties of the STVK material is specified by the constitutive law*

$$\hat{\sigma}_s(\hat{u}_s) := \hat{J}^{-1}\hat{F}(\lambda_s(tr\hat{E})I + 2\mu_s\hat{E})\hat{F}^{-T}. \tag{5.11}$$

Remark 5.1.11. In the code, we also implement

$$(\hat{v}_s, \hat{\phi}),$$

although this is not physically necessary. It is first for computational convenience in order to extend the fluid velocity variable to the whole domain. This could be resolved by using the FE Nothing element. Second, using $\hat{v}_s$ here makes it easier to understand the nonstationary FSI problem. The same holds for the (artificial) pressure variable in the STVK case.

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:

$$\nu_s = \frac{\lambda_s}{2(\lambda_s + \mu_s)}, \qquad E = \frac{\mu_s(\lambda_s + 2\mu_s)}{(\lambda_s + \mu_s)}. \tag{5.12}$$

On fluid side, we extend the problem from Stokes flow to stationary Navier-Stokes flow considering the convection term

$$v \cdot \nabla v$$

which reads in transformed form [17]

$$(\hat{J}\rho_f \hat{F}^{-1}\hat{v} \cdot \hat{\nabla}\hat{v}, \hat{\phi}^v)_{\hat{\Omega}_f}.$$

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 previous Example 5.1.10, 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).
As before, we build up the element and boundary equations and matrices in the *localpde.h* file. Apart from using the additionally defined problem parameters and modeling 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.

Finally this example demonstrates how to use the direct solvers provided by trilinos. This is done by the line

```
typedef TrilinosDirectLinearSolverWithMatrix
         <SPARSITYPATTERN, MATRIX, VECTOR> LINEARSOLVER;
```

interfacing to the trilinos library. The selection of the precise direct solver can be done using the parameter file.

5.1.11 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.

5.1.12 Use of Raviart-Thomas element; Special linear solvers

General problem description

This example shows the use of the Raviart-Thomas (RT) element in `DOpElib`. The example is taken from `dealii` step-20 and shows how this step can be implemented in `DOpElib`. The vector valued laplace equation is solved in the mixed formulation.

Most things are identical to all the other programs. However there is a subtle difference in `localpde.h`: Although the element used has 3 components (two for the RT-Element and 1 for the pressure) the block_component vector mapping blocks to components has only 2 entries. 1 for each finite element used!

The second difference is that we have to initialize the mapping explicitly. This is due to the fact that the default

```
DOpEWrapper::Mapping<DIM,DOFHANDLER> mapping(1,false);
```

is not working with the RT-element and leads to elements on which the divergence is NaN.

An additional feature of this example is that for the solution of the PDE in mixed form we are using the Schur complement solver provided in `dealii` step-20. Hence this example shows how simple it is to use self-made linear solvers within the `DOpElib` framework.

5.1.13 Discontinuous Galerkin

General problem description

Within this example, we demonstrate how to use the dG (discontinuous Galerkin) method for the solution of a transport equation. The example corresponds to the dealii step-12. Here we want to solve the transport equation

$$\begin{aligned} \nabla \cdot (\beta u) &= 0 && \text{in } \Omega, \\ u &= g && \text{on } \Gamma_- := \{x \in \partial\Omega \,|\, \beta(x) \cdot n(x) < 0\}. \end{aligned}$$

Where n is the outward unit normal, $\Omega = (0,1)^2$ and

$$\beta(x) = \frac{1}{|x|} \begin{pmatrix} -x_2 \\ x_1 \end{pmatrix}.$$

For the numerical solution, as in dealii step 12, we use the upwind discontinuous Galerkin . Hence we solve the problem of finding u_h such that

$$-\sum_{T \in \mathcal{T}_h} (u_h \beta \cdot \nabla v_h)_T + \sum_{F \in \mathcal{F}_h} (u_h^-, [\beta \cdot n v_h])_F + (u_h, \beta \cdot n v_h)_{\Gamma_+} = -(g, \beta \cdot n v_h)_{\Gamma_-}$$

where $\Gamma_+ = \{x \in \partial\Omega \,|\, \beta(x) \cdot n(x) > 0\}$, $\mathcal{T}_h$ and $\mathcal{F}_h$ denote the elements and interior faces of the mesh, respectively. The jump is defined as

$$[\beta \cdot n v_h] = (v^+ - v^-)\beta \cdot n^+$$

where the superscript $+$ or $-$ denotes the dependence on the upstream $+$ or downstream $-$ element.

Implementational Details

Within this program, we need to make use of the additional `Face*` and `Interface*` methods as given in the `PDEInterface` class. The Face* methods define all integrals on $\mathcal{F}$ in which the element interacts with it self. The Interface* methods are used for the coupling between the two neighboring elements over the given face.

The program requires the following changes in contrast to the prior examples:

main.cc We utilize Block Preconditioners for the solution of the resulting system. To this end we included the line

```
typedef DOpEWrapper::PreconditionBlockSSOR_Wrapper<MATRIX,4>
        PRECONDITIONERSSOR;
```

In contrast to all other preconditioners, we need to specify the block size. This number needs to correspond to the number of unknowns per elements; here 4 since we use Q1-elements. Note that this works for dG elements only.

The next important change is that we now not only use a discontinuous element, but we will need to assemble terms on faces between elements that couple the unknowns in the different elements together. For this, the matrix needs to have the corresponding non-zero entries specified in the sparsity pattern. To do this, the space time handler has an argument `bool flux_pattern` in the constructor that needs to be set to true, i.e., we instantiate as follows:

```
STH DOFH(triangulation, state_fe, true);
```

Finally, the integration will utilize a special function to be declared in the `LocalPDE`, hence all objects must use the `LocalPDE` and not the `PDEInterface`. To make sure that this is the case, the `PDEProblemContainer` needs to be initialized with the following arguments

```
typedef PDEProblemContainer<LocalPDE<CDC, FDC, DOFHANDLER, VECTOR, DIM>,
    SimpleDirichletData<VECTOR, DIM>, SPARSITYPATTERN, VECTOR, DIM> OP;
```

localpde.h In order to integrate the PDE above, we have to deal with one term that has not been considered before

$$\sum_{F\in\mathcal{F}_h} (u_h^-, [\beta\cdot n v_h])_F$$

Since internally all terms by sums over elements we split this term into contributions on the element edges ∂K. On an element K a face F, with outward normal n connects to another element K', depending on the sign of $\beta\cdot n$ we have two cases. $\beta\cdot n>0$ in which case $u_h^- = u_h$ or $\beta\cdot n<0$ in which case $u_h^- = u_h^* = u_h|_{K'}$, i.e. the value from the neighbor. The jump always contains the values v_h and v_h^*.

Let now $\beta\cdot n>0$. Then we assemble the contributions coming only from this element in the `FaceEquation` (and `FaceMatrix`), i.e.,

$$(u_h, \beta\cdot n v_h)_F$$

The other part of the jump, namely

$$-(u_h, \beta\cdot n v_h^*)_F$$

is not assembled here, since the test functions do not live on the selected element K. These contributions will be assembled once the element K' is selected (and hence on the same face $\beta\cdot n<0$. Once this is the case, i.e. $\beta\cdot n<0$, we assemble the other part of the jump, which is now

$$(u_h^*, \beta\cdot n v_h)_F.$$

This is done in the `InterfaceEquation` (and `InterfaceMatrix`) since we couple unknowns for the neighboring element K^* (the values of u_h^*) with those on K (the values of v_h). Note that in contrast to the view on the element with $\beta\cdot n>0$ the term has

apparently switched signs. This is no typo, but due to the fact, that the outward normal has changed direction.

The precise assembly is analogous to the usual integrals, hence we don't provide more details. The only thing different in the `InterfaceEquation` and `InterfaceMatrix` we need to access the values on the element on the other side of the face. To this end, all `Get*` functions used, such as `GetFEFaceValuesState`, have a counterpart `GetNbr*`, i.e., `GetNbrFEFaceValuesState`, to access the corresponding values on the neighboring element.

Naturally the two functions

```
bool HasFaces() const;
bool HasInterfaces() const;
```

need to return `true`.

A last and important change is that we now need to implement the method

```
template<typename ELEMENTITERATOR>
 bool AtInterface(ELEMENTITERATOR& element, unsigned int face) const
 {
   if (element[0]->neighbor_index(face) != -1)
     return true;
   return false;
 }
```

that returns true whenever we are on an interior face and false otherwise.

5.2 Nonstationary PDEs

Until now, `DOpElib` 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 as 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.

5.2.1 Nonstationary Navier-Stokes Equations

General problem description

In this example we consider the nonstationary incompressible Navier-Stokes equation. As in the stationary PDE Example 5.1.1, we use the symmetric fluid stress tensor, i.e. in strong formulation we deal with

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

on time interval $I = [0,T]$ (with $T = 80$) and the fluid benchmark domain (Schaefer/-Turek 1996). Here, we set $f = 0$ and the flow is driven by a Dirichlet inflow condition.

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:

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

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

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

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 usually, with a Galerkin finite element scheme, here based on the Taylor-Hood element Q_2^c/Q_1^c.

The variational formulation reads:

Problem 5.2.1 (Backward Euler (BE) timestepping problem)**.** *Let $\theta = 1$. Find $v := v^{n+1} \in V$ and $p := p^{n+1} \in L$:*

$$(v, \phi^v) + k\theta(\nabla v + \nabla v^T, \nabla \phi^v) - k(p, \nabla \cdot \phi^v) = k\theta(f^{n+1}, \phi^v) + (v^n, \phi^v) \tag{5.13}$$
$$(\nabla \cdot v, \phi^p) = 0 \tag{5.14}$$

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

Derivation of the other timestepping problems is analogous.

Remark 5.2.2. Note that because of the zero right hand side we are allowed to multiply (5.14) 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 ElementEquation (..., 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
```

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 ElementMatrix (..., 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 are other behavior.

```
void ElementTimeEquation (...)
```

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 ElementTimeEquationExplicit (...)
```

and we write

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

This behavior is useful for multi-physics problems where other solutions variables have to be considered around $\partial_t v$. The user should have a look in the second Example 5.2.2 for nonstationary problems for an illustration of this function.

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

```
void ElementTimeMatrix (...), void ElementTimeMatrixExplicit
```

5.2.2 Nonstationary FSI Problem

General problem description

In the present example, we solve a nonstationary fluid-structure interaction problem in arbitrary Lagrangian-Eulerian (ALE) coordinates. The mesh motion model is based on solving a biharmonic equation [17] rather than a linear-elastic model. The underlying equations are stated in the following:

Problem 5.2.3 (FSI with biharmonic mesh motion)**.** *Find* $\{\hat{v}, \hat{u}, \hat{w}, \hat{p}\} \in \{\hat{v}^D + \hat{\mathcal{V}}^0\} \times \{\hat{u}^D + \hat{\mathcal{V}}^0\} \times \hat{\mathcal{V}} \times \hat{\mathcal{L}}$*, such that* $\hat{v}(0) = \hat{v}^0$ *and* $\hat{u}(0) = \hat{u}^0$*, for almost all time steps* t*, and*

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

with the densities $\hat{\rho}_f$ *and* $\hat{\rho}_s$*, the viscosity* ν_f*, the Lamé parameters* μ_s*,* λ_s *and the deformation gradient* $\hat{F}$*, and its determinant* $\hat{J}$*. The stress tensors for the fluid and structure are implemented by*

$$\hat{\sigma}_f = -\hat{p}I + \hat{\rho}_f \nu_f (\hat{\nabla}\hat{v}\hat{F}^{-1} + \hat{F}^{-T}\hat{\nabla}\hat{v}^T),$$

and

$$\hat{\sigma}_s = \hat{F}(\lambda_s tr\hat{E}I + 2\mu_s \hat{E})$$

with $\hat{E} = \frac{1}{2}(\hat{F}^T\hat{F} - I)$*. Finally, we notice that this problem is driven by a Dirichlet inflow condition. It is possible to add a gravity term* $\hat{f}_f$ *or* $\hat{f}_s$*, which would enter as a right hand side force*

$$-(\hat{\rho}_f \hat{J}\hat{f}_f, \hat{\psi}^v)_{\hat{\Omega}_f} - (\hat{\rho}_s \hat{f}_s, \hat{\psi}^v)_{\hat{\Omega}_s}$$

into the problem.

The ALE approach belongs to interface-tracking methods in which the mesh is moved such that it fits in all time steps with the FSI-interface. However, this leads to a degeneration of the ALE map. Methods to circumvent such as degeneration as long as possible are re-meshing techniques or to use (as suggested here) a biharmonic mesh motion technique.

Code validation for ALE-fluid and FSI problems

With the ALE code implemented in Example 5.1.10 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:

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

The code is validated by the well-known fluid- and FSI benchmark problems [15, 11]. For the FSI test cases, the basic configuration is sketched in Fig. 5.1 at which an elastic beam is attached behind the rigid cylinder.

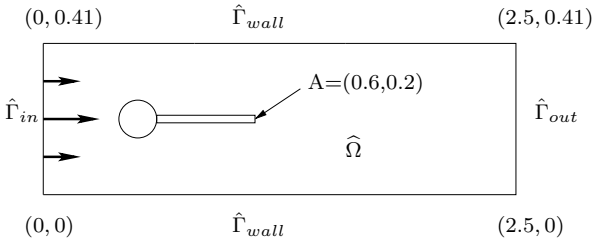

Figure 5.1: Flow around cylinder with elastic beam with circle-center $C = (0.2, 0.2)$ and radius $r = 0.05$.

The elastic beam has length $l = 0.35m$ and height $h = 0.02m$. The right lower end is positioned at $(0.6m, 0.19m)$, and the left end is attached to the circle. Control points $A(t)$ (with $A(0) = (0.6, 0.2)$) are fixed at the trailing edge of the structure, measuring x- and y-deflections of the beam. Details on parameters and evaluation functionals and other results can be found in [11, 8, 17]. The time-stepping scheme can be very easily chosen in the `main.cc` function by choosing an appropriate time-stepping scheme as explained at the beginning of this manual and detailed in the previous example.

The quantities of interest are evaluations of x- and y displacement at the point $A(0) = (0.6, 0.2)$ and the drag and lift forces acting on the cylinder and the elastic beam:

$$(F_D, F_L) = \int_{S_f} \hat{\sigma}_f \cdot \hat{n}_f \, ds + \int_{\hat{\Gamma}_i} \hat{\sigma}_s \cdot \hat{n}_s \, ds, \tag{5.15}$$

where S_f denotes the path over the cylinder in the fluid part and Γ_i the interface between the elastic beam and the fluid.

Program description

The major difference to the first nonstationary program is the introduction of the

```
void ElementTimeEquationExplicit (...)
```

to write all time derivative terms explicitly:

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

This behavior is useful since (as shown in the above equations) other solutions variables have to be considered around $\partial_t v$ such as $J := J(u)$. The same holds for the corresponding matrix part.

Further, this example shows, how to change the vector behavior, from our default option `fullmem`, where the whole vector is stored in the computers main memory. Here, we are only interested in calculating the solution once, hence two vectors, one for the

current time point and one for the previous one are sufficient. Hence we choose the option `only_recent` so that we don't have to reserve unneccessary memory. If we need to store the whole trajectory for some reason another option is available to circumvent the restrictions due to the size of the main memory, it will be described in Example 5.2.3.

5.2.3 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:

$$\partial_t u - \frac{1}{2} \sum_{i,j=1}^{d} \sigma_i \sigma_j \rho_{ij} x_i x_j \partial_{x_i} \partial_{x_j} u - r \sum_{i=1}^{d} x_i \partial_{x_i} u + ru = 0 \qquad \text{in } (0, T] \times \mathbb{R}^d_+, \tag{5.16a}$$

$$u(0) = u_0 \qquad \text{in } \mathbb{R}^d_+. \tag{5.16b}$$

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

$$u_0 := \begin{cases} \max(\sum_{i=1}^{d} \lambda_i x_i - K, 0), & u \text{ is a } \textit{Call}, \\ \max(K - \sum_{i=1}^{d} \lambda_i x_i, 0), & u \text{ is a } \textit{Put}, \end{cases} \tag{5.17}$$

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 $\overline{x} \in \mathbb{R}^d_+$ and consider the computational domain $\Omega := (x_1, \overline{x}_1) \times \cdots \times (x_d, \overline{x}_d)$. On the new part of the boundary Γ with $\Gamma := \{x \in \partial\Omega | \exists_{1 \leq i \leq d} x_i = \overline{x}_i\}$) 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 $\overline{x}$	$(100, 100)$
interest rate r	0,05
option value $u(T, x_0)$	ca. 2,269172389

Program description

Note that the initial conditions are only H^1-regular. Because of this, the Crank-Nicolson scheme, which is not strongly A-stable, is not suited to solve this problem. As we want to use a second order accurate time stepping scheme, we use the shifted Crank-Nicolson method. The rest of the program is as before.

In addition, this program shows how to change the vector behavior, from our default option `fullmem`, where the whole vector is stored in the computers main memory. Here the storage behavior is set to `store_on_disc`, where only those unknowns are loaded into main memory that are needed in the current time step, while all other unknowns are stored on the hard drive.

This behavior is particularly useful, if many time steps are taken; so that the whole set of unknowns can no longer be stored in main memory, but access to all parts of the solution is required after the solution process and thus the option `only_recent` used in Example 5.2.2 is not sufficient. Note that all vectors will allocate the required memory when the vector is reinitialized to a new size. Hence, for large vectors this may need some time.

To avoid multiple programs accessing the same files on the hard drive, a `lock` file is initialized. Under normal conditions, this will be deleted once the program terminates. However, should the program exit exceptionally, the lock file will still exist. Calling the program in this case will produce an exception, with the following text:

```
Warning: During execution of ‘StateVector<VECTOR>::StateVector‘
         the following Problem occurred!
The directory Results/tmp_state/ is probably already in use.
```

To resolve the issue, you have to delete the named directory with the temporary storage files manually.

5.2.4 Heat Equation in 1D

General problem description

In this example we consider one of the prototypical nonstationary equations, the parabolic heat equation, i.e. for $x \in \Omega \subset \mathbb{R}^d, t \in I = [0,T], T \in \mathbb{R}^+$, we search for the unknown solution $u : I \times \Omega \to \mathbb{R}$

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

In our example, we consider the simplest case $d = 1$, where the Laplacian Δ reduces to ∂_x^2. The computational domain is $I \times \Omega = [0,1] \times [0,1]$. For further simplification, we choose the right hand 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 other 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 5.2.1):

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 *GNUPLOT*, for example, so instead of *.vtk* files as in all former examples, we now write out *.gpl* files.

5.2.5 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 $I \times \Omega = [0,1] \times [0,\pi]^2$. 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 right hand side. In examples 5.1.4 and 5.1.6, we regarded stationary problems with nonhomogeneous right hand sides, but up to now, we never involved the time variable into the nonhomogeneity. To do this, `DOpElib` 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.

5.2.6 Biot-Lamé-Navier problem

General problem description

The modeling part of this example is based on the coupled Biot-Lamé-Navier system. The Biot system itself is a standard model in subsurface modeling [5, 6, 7]. Here, a reservoir (the pay-zone) is modeled as a poroelastic medium with the help of Biot's equations. A surrounding medium (the non-pay zone) is modeled as a static elastic solid. In fact, the Biot system is a multi-scale problem which is identified on the micro-scale as a fluid-structure interaction problem (details on the interface law are found in Mikelić and Wheeler (2011). This system is specifically suited for applications in subsurface modeling for the poroelastic part, the so-called pay-zone. On the other hand, surrounding rock (the non-pay zone) is modeled with the help of linear elasticity (Ciarlet 1984). Therefore, the final configuration belongs to a multiphysics problem in non-overlapping domains. The nonstationary coupled system for the state is formulated within a variational monolithically-coupled framework, which is known to be more robust than partitioned solutions algorithms. Its discretization is carried out with help of the Rothe method in which we first discretize in time and then in space. The configuration is based on the augmented Mandel problem which shows the important Mandel-Cryer effect: First increasing pressure and then decreasing pressure in time while applying some traction force on the top boundary.

We begin by describing the setting for a pure poroelastic setting, the so-called pay-zone. Let Ω_B the domain of interest and $\partial\Omega_B$ its boundary with the partition:

$$\partial\Omega_B = \Gamma_u \cup \Gamma_t = \Gamma_p \cup \Gamma_f,$$

where Γ_u denotes the displacement boundary (Dirichlet), Γ_t the total stress or traction boundary (Neumann), Γ_p the pore pressure boundary (Dirichlet), and Γ_f the fluid flux boundary (Neumann). Concretely, we have for a material with the displacement variable u and its Cauchy stress tensor σ:

$$\begin{aligned} u &= \bar{u} \quad \text{on } \Gamma_u, \\ \sigma n &= \bar{t} \quad \text{on } \Gamma_t, \end{aligned}$$

for given $\bar{u}$ and $\bar{t}$, and the normal vector n. For the pressure with the permeability tensor K and fluid's viscosity η_f, we have the conditions:

$$\begin{aligned} p &= \bar{p} \quad \text{on } \Gamma_p, \\ -\frac{K}{\eta_f}\Big(\nabla p - \rho_f g\Big) \cdot n &= \bar{q} \quad \text{on } \Gamma_f, \end{aligned}$$

for given $\bar{p}$ and $\bar{q}$; and the density ρ_f and the gravity g. For the initial conditions at time $\tau = 0$, we prescribe

$$\begin{aligned} p(\tau = 0) &= p_0 \quad \text{in } \Omega_B, \\ \sigma(\tau = 0) &= \sigma_0 \quad \text{in } \Omega_B, \end{aligned}$$

In this case of this extension (if Ω_B is totally embedded in Ω_S) the boundary conditions on $\partial\Omega_B$ reduce to interface conditions $\partial\Omega_B := \Gamma_i = \Omega_B \cap \Omega_S$. Let $I := [0, T]$ denote the time interval.

Problem 5.2.4 (The Biot system)**.** *Find the pressure p_B and displacement u_B such that*

$$\begin{aligned} &\partial_t(c_B p_B + \alpha_B div u_B) \\ &-\frac{1}{\eta_f} div\, K(\nabla p_B - \rho_f g) = q \quad \text{in } \Omega_B \times I, \\ &-div\big(\sigma_B(u)\big) + \alpha_B \nabla p_B = f_B \quad \text{in } \Omega_B \times I, \end{aligned}$$

with

$$\sigma_B(u_B) := \mu_B(\nabla u_B + \nabla u_B^T) + \lambda_B div\, u_B I,$$

and the coefficients $c_B \geq 0$, the Biot-Willis constant $\alpha_B \in [0, 1]$, (in fact, this constant relates to the amount of coupling between the flow part and the elastic part) and the permeability tensor K, fluid's viscosity and its density η_f and ρ_f, gravity g and a volume source term q (i.e., usually, wells for oil production and fluid injection). In the second equation, the Lamé coefficients are denoted by $\lambda_B > 0$ and $\mu_B > 0$ and f_B is a volume force.

The velocity v_B in the porous medium is obtained with the help of Darcy's law (Darcy 1856) and the Darcy equations which are obtained through homogenization of Stokes's equations It holds:

$$v_\mathrm{B} = -\frac{1}{\eta_f} K(\nabla p_\mathrm{B} - \rho_f g).$$

Usually the non-pay zone is described in terms of linear elasticity:

Problem 5.2.5. *Find a displacement u_S such that*

$$-div\big(\sigma_S(u_S)\big) = f_S \quad \text{in } \Omega_S \times I,$$

with

$$\sigma_S(u_S) := \mu_S(\nabla u_S + \nabla u_S^T) + \lambda_S div\, u_S I,$$

with the Lamé coefficients μ_S and λ_S and a volume force f_S. On the boundary $\partial\Omega_S := \Gamma_D \cup \Gamma_N$, the conditions

$$u_S = \bar{u}_S \quad \text{on } \Gamma_D, \quad \sigma_S(u_S) n_S = \bar{t}_S \quad \text{on } \Gamma_N,$$

are prescribed with given $\bar{u}_S$ and $\bar{t}_S$.

It finally remains to describe the interface conditions on Γ_i between the two subsystems:

$$\begin{aligned} u_\mathrm{B} &= u_S, \\ \sigma_B(u_B) n_B - \sigma_S(u_S) n_S &= \alpha p_\mathrm{B} n_B, \\ -\frac{1}{\eta_f} K(\nabla p_\mathrm{B} - \rho_f g) \cdot n_S &= 0. \end{aligned} \tag{5.18}$$

Program description

In this example, the crucial aspect (from mathematical point of view as well as from the implementation) are the interface conditions (5.18). Here, it is important to notice that the second condition in (5.18), requires careful implementation on the interface, which is carried out with the help of deal.II's FE Nothing element. Second, please do not forget to activate the flag

```
HasFaces() const
{
  return true;
}
```

The problem is driven by traction forces (Neumann conditions), which are imposed via the

```
void
BoundaryEquation(...)
{
  ...
}
```

As functionals, we evaluate the pressure in two different points of the domain. The observation should be that the pressure first starts increasing reaching a maximum and then starts decaying. This is the so-called Mandel-Creyer effect.

5.2.7 Isothermal Euler equations

General problem description

This example solves the isothermal Euler equations

$$\partial_t(\rho v) + \partial_x(p(\rho) + \rho v^2) = -\frac{\lambda}{2D} v|v| - gh'\rho$$
$$\partial_t \rho + \partial_x(\rho v) = 0$$

on the set $t \in (0,1)$, $x \in (0,2)$. The relation

$$p(\rho) = \frac{RT\,\rho}{1 - \alpha RT\,\rho}$$

depends on the user provided data for gas-constant R and temperature T as well as a parameter α ($\alpha = 0$ for ideal gases). The other parameters in the system to be provided by the user are the friction parameter λ, the diameter D of the pipe, the gravity g and the slope h' of the pipe.

The discretization is done by a dG-method in space for the variables $w = \rho v$ and ρ.

6 Examples with Optimization

6.1 Subject to a Stationary PDE

6.1.1 Distributed control with a linear elliptic PDE

General problem description

This example solves a distributed minimization problem and shows how to estimate the error in the cost functional for stationary optimization problems. The problem reads:

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

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

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

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

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

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

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

In addition the following functionals are evaluated:

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

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

Background information and program description

In the following, we describe all extensions to the previous problems relevant to solving PDE-based optimization with `DOpElib`. So far, we had only to implement the `ElementEquation` and the corresponding matrix `ElementMatrix`. Now, based on the idea of the reduced cost functional, we have to compute certain additional equations representing the adjoint, tangent, and adjoint hessian equations `ElementEquation_U, ElementEquation_UT, ElementEquation_UTT` for the state equation and in the same terms arising from the functional itself. Let us shed some light into all equations by giving some background information and overview first.

In abstract form, we are given the following optimization problem:

$$J(q,u) \to \min, \quad a(q,u)(\psi) = 0 \quad \forall \psi \in V$$

Lagrangian:

$$\mathcal{L}(q,u,z) := J(q,u) - a(q,u)(z)$$

Optimality system (KKT system):

$$\begin{aligned} a'_u(q,u)(\phi,z) &= J'_u(q,u)(\phi) \quad \forall \phi \in V \\ a'_q(q,u)(\chi,z) &= J'_q(q,u)(\chi) \quad \forall \chi \in Q \\ a(q,u)(\psi) &= 0 \qquad \forall \psi \in V \end{aligned}$$

or equivalently, in terms of the Lagrangian

$$\begin{aligned} \mathcal{L}'_u(q,u,z)(\phi) &= 0 \quad \forall \phi \in V \quad \text{(Adjoint Equation)} \\ \mathcal{L}'_q(q,u,z)(\chi) &= 0 \quad \forall \chi \in V \quad \text{(Gradient Equation)} \\ \mathcal{L}'_z(q,u,z)(\psi) &= 0 \quad \forall \psi \in V \quad \text{(State Equation)} \end{aligned}$$

The continuous problem is discretized by a standard Galerkin method using finite dimensional subspaces $Q_h \times V_h \subset Q \times V$:

$$J(q,u) \to \min, \quad a(q,u)(\psi) = 0 \quad \forall \psi \in V$$

Discrete saddle-point problems

$$\begin{aligned} a'_u(q_h,u_h)(\phi_h,z_h) &= J'_u(q_h,u_h)(\phi_h) \quad \forall \phi_h \in V_h \\ a'_q(q_h,u_h)(\chi_h,z_h) &= J'_q(q_h,u_h)(\chi_h) \quad \forall \chi_h \in Q_h \\ a(q_h,u_h)(\psi_h) &= 0 \qquad \forall \psi_h \in V_h \end{aligned}$$

Solution process

In this section, we briefly discuss the solution process for the optimization problem. For further details, we refer to the standard literature. The unconstrained optimal control

problem is reformulated as follows. We introduce the solution operator $S : Q \to V$ of the state equation. Then:

$$j(q) := J(q, S(q)) \to \min, \quad a(q, S(q))(\Psi) = 0 \quad \forall \Psi \in V.$$

The local existence and sufficient regularity of S is assumed. The necessary optimality conditions of first and second order are

$$j'(q)(\delta q) = 0, \quad j''(q)(\delta q, \delta q) \geq 0 \quad \forall \delta q \in Q.$$

The derivatives of the reduced functional can be computed using the Lagrangian

$$\mathcal{L}(q, u, z) = J(q, u) - a(q, u)(z)$$

as already introduced. Let $q \in Q$, and the corresponding state $u = S(q) \in V$ be given. To calculate the derivative of the reduced cost functional j, we introduce the dual variable $z \in V$ solving the

Dual equation

$$\mathcal{L}'_u(q, u, z)(\psi) = 0 \quad \forall \psi \in V.$$

Then,

$$j'(q)(\delta q) = \mathcal{L}'_q(q, u, z)(\delta q) \quad \text{for } \delta q \in Q.$$

To calculate the second derivatives, we need to solve additional equations.
Let $\delta q \in Q$ be a given direction. Then we search $\delta u \in V$ solving the

Tangent equation

$$\mathcal{L}''_{qz}(q, u, z)(\delta q, \phi) + \mathcal{L}''_{uz}(q, u, z)(\delta u, \phi) = 0 \quad \forall \phi \in V.$$

Further, we have an auxilliary

Dual for Hessian equation to find $\delta z \in V$ solving

$$\mathcal{L}''_{qu}(q, u, z)(\delta q, \phi) + \mathcal{L}''_{uu}(q, u, z)(\delta u, \phi) + \mathcal{L}''_{zu}(q, u, z)(\delta z, \phi) = 0 \quad \forall \phi \in V.$$

Then, for $\delta r \in Q$, we can express the second derivatives of j by

$$\begin{aligned} j''(q)(\delta q, \delta r) = {} & \mathcal{L}''_{qq}(q, u, z)(\delta q, \delta r) \\ & + \mathcal{L}''_{uq}(q, u, z)(\delta u, \delta r) \\ & + \mathcal{L}''_{zq}(q, u, z)(\delta z, \delta r). \end{aligned}$$

With these terms, we can calculate the Newton direction δq, at a given iterate q^n, as solution to the problem

$$j''(q^n)(\delta q, \chi) = -\, j'(q^n)(\chi) \quad \forall \chi \in Q.$$

Moreover, we would like to work in the Hilbert space Q. However, the derivative $j'(q) \in H^*$ only. Hence, we need to calculate the Riesz representation for the gradient $\nabla j(q) \in H$ using the definition:

$$(\nabla j(q), \delta q)_Q = j'(q)(\delta q) \quad \forall\, \delta q \in Q.$$

In the given example, $Q = L^2(\Omega)$ and hence the scalar product will be the standard L^2-inner product. Simillarly, we can define the Hessian operator $H(q) \in \mathcal{L}(Q, Q)$ by defining

$$(H(q)\tau q, \delta q)_Q = j''(q)(\tau q, \delta q) \quad \forall\, \delta q, \tau q \in Q.$$

Implementation in `DOpElib`

From the previous details, and the definition of the Lagrangian and its derivatives it is clear, that the user has to provide the respective derivatives. Since the Lagrangian consists of the PDE and the cost functional it is sufficient to provide the respective derivatives, while `DOpElib` will assemble them as required. Test functions for vector valued terms will be denoted by an index i while matrix valued terms are indexed in i and j. Test functions in the control space Q are denoted as ϕ_i^q while those in the state space V are denoted as ϕ_i.

To solve the linear equations the following matrices are needed

`ElementMatrix`	$\Leftrightarrow$	$a_{i,j} = a'_u(q,u)(\phi_j, \phi_i)$,
`ControlElementMatrix`	$\Leftrightarrow$	$a_{i,j} = (\phi_j^q, \phi_i^q)_Q$.

The first one is required for all primal and dual PDE solves, while the second one is needed to calculate the Riesz representation of the derivatives of j. If desired, the matrix for the adjoint PDEs can be provided separately as `ElementMatrix_T`, but otherwise this will be calculated automatically from the primal matrix.

Additional terms are needed to calculate the corresponding right hand sides. These are for the PDE the following:

`ElementEquation` (state)	$\Leftrightarrow$	$a(q,u)(\phi_i)$,
`ElementRightHandSide` (state)	$\Leftrightarrow$	$f(\phi_i)$,
`ControlElementEquation` (gradient or hessian)	$\Leftrightarrow$	$(\nabla j(q), \phi_i^q)_Q$,

as well as

`ElementEquation_U` (adjoint)	$\Leftrightarrow$	$a'_u(q,u)(\phi_i, z)$,
`ElementEquation_Q` (gradient)	$\Leftrightarrow$	$a'_q(q,u)(\phi_i^q, z)$,

the terms

`ElementEquation_UU` (adjoint hessian)	$\Leftrightarrow$	$a''_{uu}(q,u)(\delta u, \phi_i, z)$,
`ElementEquation_UQ` (hessian)	$\Leftrightarrow$	$a''_{uq}(q,u)(\delta u, \phi_i^q, z)$,
`ElementEquation_QU` (adjoint hessian)	$\Leftrightarrow$	$a''_{qu}(q,u)(\delta q, \phi_i, z)$,
`ElementEquation_QQ` (hessian)	$\Leftrightarrow$	$a''_{qq}(q,u)(\delta q, \phi_i^q, z)$,
`ElementEquation_UT` (tangent)	$\Leftrightarrow$	$a'_u(q,u)(\delta u, \phi_i)$,
`ElementEquation_QT` (tangent)	$\Leftrightarrow$	$a'_q(q,u)(\delta q, \phi_i)$,

and finally

`ElementEquation_UTT` (adjoint hessian)	$\Leftrightarrow$	$a'_u(q,u)(\phi_i, \delta z)$,
`ElementEquation_QTT` (hessian)	$\Leftrightarrow$	$a'_q(q,u)(\phi_i^q, \delta z)$.

As for PDE problems, it is up to the user to decide if the `ElementRightHandSide` is used, or if the terms are included in the `ElementEquation`.

For the cost functional, we have to provide

`ElementValue` (all)	$\Leftrightarrow$	$J(q,u)$,
`ElementValue_U` (all)	$\Leftrightarrow$	$J'_u(q,u)(\phi_i)$,
`ElementValue_Q` (all)	$\Leftrightarrow$	$J'_q(q,u)(\phi_i)$,
`ElementValue_UU` (all)	$\Leftrightarrow$	$J''_{uu}(q,u)(\delta u, \phi_i)$,
`ElementValue_UQ` (all)	$\Leftrightarrow$	$J''_{uq}(q,u)(\delta u, \phi_i^q)$,
`ElementValue_QU` (all)	$\Leftrightarrow$	$J''_{qu}(q,u)(\delta q, \phi_i)$,
`ElementValue_QQ` (all)	$\Leftrightarrow$	$J''_{qq}(q,u)(\delta q, \phi_i^q)$.

Clearly, if the PDE or cost functional contains other terms, such as boundary or face integrals corresponding derivatives must be provided as well.

Back to the specific equations in this example

We have

$$
\begin{aligned}
a(q,u)(\phi) &= (\nabla u, \nabla \phi) - (q + f, \phi),\\
a'_u(q,u)(\phi, z) &= (\nabla \phi, \nabla z),\\
a'_u(q,u)(\delta u, \phi) &= (\nabla \delta u, \nabla \phi),\\
a'_u(q,u)(\phi, \delta z) &= (\nabla \phi, \nabla \delta z),\\
a'_q(q,u)(\delta q, \phi) &= -(z, \psi^q),\\
a'_q(q,u)(\delta q, \phi) &= (\delta q, \phi),\\
a'_q(q,u)(\delta q, \delta z) &= -(\delta z, \psi^q).
\end{aligned}
$$

For the cost functional, we have the following terms:

$$\begin{aligned} J(q,u) &= \frac{1}{2}\|u-u^d\|^2 + \frac{\alpha}{2}\|q\|^2, \\ J'_u(q,u)(\phi) &= (u-u^d,\phi), \\ J'_q(q,u)(\phi) &= \alpha(q,\psi^q), \\ J''_{uu}(q,u)(\psi,\phi) &= (\delta u,\phi). \end{aligned}$$

All other terms, specifically mixed terms with QU etc. are zero in this example.

`main.cc`

Finally, the main file of the optimization examples does not look very much different than for pure PDE computations - which is one of the crucial aims of our library. Here, instead of using a `pdeproblemcontainer`, we use now an `optproblemcontainer` which can assemble all additionally needed informations, such as adjoint and tangent PDEs. Furthermore, we define `ReducedNewtonAlgorithm` and `ReducedTrustregion_NewtonAlgorithm` to solve the optimization problem with a linesearch and a trust-region Newton algorithm. Of course one would be sufficient, but we wanted to show how to change optimization solvers easily using `DOpElib`.
Next, in the body of the main file, we introduce a second FE function for the control variable. Then, we define a `COSTFUNCTIONAL`. Finally, the problem is either solved by calling `Alg.Solve(q)` and/or the user might check if the derivatives are implemented correctly by calling `Alg.CheckGrads` or `Alg.CheckHessian`. The latter two functionalities are highly recommended to check your implementation before wondering about your results.
Finally, this example uses a DWR-error estimator to estimate the error made in the cost functional. In contrast to the error estimation for PDEs here, we have to include the error in the control by using the `HigherOrderDWRContainerControl`.

6.1.2 Parameter control with a linear elliptic PDE

General problem description

This example solves the following pointwise minimization problem

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

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

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

- and the exact solution given by

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

Program description

In contrast to the first example, the control is now a discrete quantity (for the three observation points) where we use the `FENothing` element to assign the three controls. Here, the number of components equals the number of controls. In addition, 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";
}
```

Indicating, we have both some point values in the functional, as well as a domain contribution, i.e., we calculate $\|q\| = \int_\Omega |q|\,dx$, even though this in not necessary, since $\alpha = 0$ and the euclidean norm of $q \in \mathbb{R}^3$ could be evaluated more easily using an AlgebraicValue in the functional.

This brings along that we have not only to implement the methods

`ElementValue`,	`ElementValue_U`,
`ElementValue_Q`,	`ElementValue_UU`,
`ElementValue_UQ`,	`ElementValue_QU`,
`ElementValue_QQ`,	

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

6.1.3 Parameter control with a nonlinear PDE from fluid dynamcis

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 "Mesh adaption for stationary flow control" (2000).

The configuration comes from the original fluid benchmark problem and has been modified to reduce drag around the cylinder. To gain the solvability of the optimization problem we add a quadratic regularization term to the cost functional.

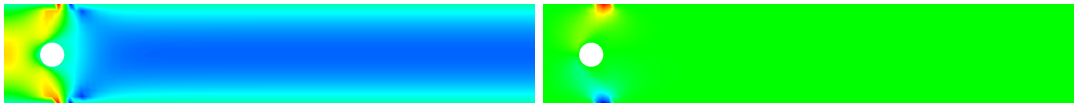

Figure 6.1: Configuration of the cylinder-drag minimization problem. By sucking out the fluid (right the y-velocity), the force on the cylinder is reduced. At left, the x-velocity field is shown. Behind the cylinder, almost no fluid goes from left to right, which is shown in blue color.

The computational domain is denoted by Ω with the boundary $\partial\Omega = \Gamma_in \cup \Gamma_out \cup \Gamma_Q \cup \Gamma_w$, where Γ_in and Γ_out denote the inflow and outflow boundaries, respectively. On Γ_in, we prescribe a fixed parabolic inflow profile. The part(s) Γ_w denote the top and bottom boundaries. Finally, Γ_Q, represent Neumann control boundaries. Here, we prescribe

$$\rho\nu\partial_n v - pn = qn \quad \text{on } \Gamma_Q$$

where n denotes the outer unit normal to Γ_Q.

The state equations are given by: Find v and p such that

$$\begin{aligned} -\nabla\cdot\sigma(v,p) + v\cdot\nabla v &= 0, \\ \nabla\cdot v &= 0 \end{aligned}$$

with $\sigma(v,p) = -pI + \rho\nu(\nabla v + \nabla v^T)$.

Remark 6.1.1. Since, we use the symmetric stress tensor, we need to subtract the non-symmetric part on the outflow boundary, related to the do-nothing condition. $\diamond$

The control q enters via the weak formulation. It reads,

$$a(q,v,p)(\phi) = (\sigma(v,p),\phi) + (v\cdot\nabla v,\phi) + \langle q,\phi\cdot n\rangle + (\nabla\cdot v,\chi) = 0,$$

The target functional is considered as

$$k(v,p) = \int_{\Gamma_O} n\cdot\sigma(v,p)\cdot d\,ds,$$

where Γ_O denotes the cylinder boundary, and d is a vector in the direction of the mean flow. For theoretical and numerical reasons, this functional needs to be regularized,

including the control variable q, such that

$$K(q, v, p) = k(v, p) + \frac{\alpha}{2}||q - q_0||^2,$$

where α is the Tikhonov parameter and q_0 some reference control.

The rest of the program is similar to the previous optimization problems where we formulate the state equation in a weak form $a(v, p)(\phi)$ such that the final problem reads

$$K(q, v, p) \to \min \quad \text{s.t.} \quad a(q, v, p)(\phi) = 0.$$

Program description

The implementation of this example does not introduce any new `DOpElib`-specific features but shows that more complicated equations such as the Navier-Stokes system can be used as forward problem for the optimization process. This example builds on the previous Example 6.1.2.

6.1.4 Control in the dirichlet boundary values

General problem description

This example solves the minimization problem

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

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

$$u_0(0,y) = q_0, \quad u_0(1,y) = q_1, \quad u_0(x,0) = q_2, \quad u_0(x,1) = q_3,$$
$$u_1 = q_4^3.$$

The data is chosen as follows:

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

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. In the main file and the localpde program, we work still with the `FENothing` element to assign the five controls.

6.1.5 Distributed Control with Different Meshes for Control and State

General problem description

This example solves the distributed minimization problem

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

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

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

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

$$\overline{q} = \frac{1}{\alpha}\sin(\pi x)\sin(2\pi y)$$
$$\overline{u} = \sin(4\pi x)\sin(2\pi y)$$

In addition, the following functionals are evaluated:

MidPoint: $u_h(0.125; 0.75)$ L1-Value: $\int_\Omega |u_h|$

QError: $\int_\Omega |q_h - \overline{q}|^2$ UError: $\int_\Omega |u_h - \overline{u}|^2$

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 `ElementDataContainer`, `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`).

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

General problem description

This example solves the distributed minimization problem

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

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

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

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

In addition the following functionals are evaluated:

$$\text{MidPoint: } u(0.125; 0.75)$$

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

Program description

The Problem is similar to that of `OPT/StatPDE/Example1` except for the box control constraints. The implementation of these constraints is taken care of in the main file (where we add a constrained description `lcc`) and in the `localconstraints.h` file. This files serves to implement the actual constraints.

In this example, we introduce the handling of *local* constraints whereas the mixture of local and *global* constraints will be discussed in the Example 6.1.7. First, we implement the upper and lower control bounds in `localconstraints.h`, i.e,

$$q_{\min} \leq q \leq q_{\max}.$$

with $q_{\min} = -500$ and $q_{\max} = 500$. The constraints are 'local', by which we mean the constraints are imposed on the nodal values of the control vector. Thus, in the constraint description `localconstraints.h`, these vectors are manipulated directly without additional integration. We note that the constraints need to be written such, that a feasible control generates non positive entried, i.e., we calculate the vector

$$\mathcal{C}(q) = \begin{pmatrix} q_{\min} - q \\ q - q_{\max} \end{pmatrix}.$$

Second, the `lcc` vector is used to describe the amount of unknowns that need to be reserved to store the constraints, and, eventually, corresponding Lagrange multipliers.

Further information can be found in `basic/constraints.h`. In our example, we have only one block in the control (The control is stored in a `deal.II::BlockVector`). Hence, the `lcc` vector has a length of one. The only entry `lcc[0]` is a vector of size two (in the case of more than one control block, each block would be given a size of 2). Each of the two entries has a specific meaning. The first entry `lcc[0][0] = 1` tells you how many local entries in the present block are locally constrained, here it is one local entry. (*Note that this entry will typically be one. it is not one would be if we have constraints of the type* $q_i + q_{i+1} \leq 1$ *for each even* i*, or similar combinations of multiple entries in the control vector.*)

The second entry `lcc[0][1] = 2`, determines the number of constraints on this local entry. Here, we impose a lower and an upper bound, i.e., we give 2 constraints. This information tells the SpaceTimeHandler, that the vector $\mathcal{C}(q)$ needs exactely twice the amount of unknowns as the vector q. In general, the space needed for $\mathcal{C}(q)$ is given as

$$\frac{\texttt{lcc[0][1]}}{\texttt{lcc[0][0]}}$$

times the unknowns for the control.

External optimization solver

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 `DOpE/ThirdPartyLibs` by the name `ipopt`, i.e., you should have the file `DOpE/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.
```

If you receive this message and have the ipopt installation complete, you might have overseen to add ipopt to your `LD_LIBRARY_PATH`:

```
************************************************************
                Installation complete!
Add /home/..../dopelib-2.0/ThirdPartyLibs/ipopt/lib64
    to your $LD_LIBRARY_PATH variable
************************************************************
```

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 `DOpE/ThirdPartyLibs` directory named `snopt`, i.e., you should have the file `DOpE/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.
```

6.1.7 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.

The implementation is done using the following three additional files:

- `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 `augmentedlagrangianproblem.h`.
- `augmentedlagrangianproblem.h` The problem container for the augmented Lagrangian problem.
- `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. This is used to fit the augmentedlagrangian problem into our framework.

Program description

In addition to the previous Example 6.1.6, we consider now in addition one global constraint. To calculate the correct storrage needed we use the second argument of `constraints(lcc, 1)`, which is now one.

We use `localconstraints.h` and `localconstraintaccessor.h` to impose all constraints. First, we have again one control block with a lower and an upper bound,

$$\rho_{min} \leq q \leq \rho_{max}$$

with $\rho_{min} = 10^{-4}$ and $\rho_{max} = 1$ (ρ denotes the density of the material). These are implemented in `localconstraintaccessor.h`. The global constraint is the maximum volume of the material, which should remain constant with the value $V_{\max} = 0.5$, i.e.,

$$\int_\Omega q - V_{\max}\, dx \leq 0.$$

Its implementation is provided in `localconstraints.h` where the global constraint is handled as a functional, which again is normalized to be nonpositive if the control is feasible.

6.1.8 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/Example7` using the SIMP method.

The solution is computed using the commercial optimization library SNOPT, similar to `OPT/StatPDE/Example6` where IPOPT is used. This Example demonstrate how global constraints on the control variable can be included into the optimization call.

Program description

In this example, we have now some local (point) constraints, local constraints and a global constraint. In constrast to the previous example, these constraints are all taken care of in the `localconstraints.h` file. Specifically, the point constraints are handled by the `DOpE::PointConstraints<...>` function, which is also initializad in the main file.

6.1.9 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 setting is similar to Opt Example 3. In fact, the only novel things are to attach an elastic beam at the cylinder and to extend the equation to fluid-structure interaction (instead of pure fluid as in Example 3).

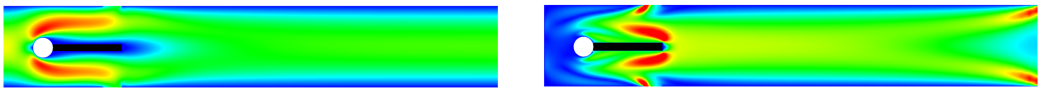

Figure 6.2: Configuration of the FSI cylinder-beam-drag minimization problem. By sucking out the fluid, the force on the cylinder is reduced. At left, the x-velocity field is shown. In right figure, we the corresponding adjoint solution is shown.

The state equation system reads:

Problem 6.1.2 (Stationary Fluid-Structure Interaction with STVK material)**.** *Let q denote the control variable. Find $U := \{\hat{v}, \hat{p}, \hat{u}\}$ such that*

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

The target functional is considered as

$$k(U) = \int_{\hat{\Gamma}_O\cup\hat{\Gamma}_i} \hat{n}\cdot\hat{J}\hat{\sigma}(v,p)\hat{F}^{-T}\cdot\hat{d}\,ds,$$

where Γ_O denotes the cylinder boundary and Γ_i the interface between fluid and solid, and $\hat{d}$ is a vector in the direction of the mean flow. Moreover, $\hat{J}$ and $\hat{F}$ denote the deformation gradient and its determinant as well known in fluid-structure interaction. For theoretical and numerical reasons, this functional needs to be regularized, including the control variable q, such that

$$K(q,U) = k(U) + \frac{\alpha}{2}||q-q_0||^2,$$

where α is the Tikhonov parameter and q_0 some reference control.

The rest of the program is similar to the previous optimization problems where we formulate the state equation in a weak form $a(q,U)(\phi)$ such that the final problem reads

$$K(q,U)\to\min \quad \text{s.t.} \quad a(q,U)(\phi) = 0.$$

Program description

This example is similar to Example 6.1.3 (both based on Example 6.1.2), except that we have again much more complicated (nonlinear) equations. A modification of this example and several numerical tests are presented in [13].

6.1.10 Parameter control with a nonlinear PDE from fluid dynamics - with cost functional not given as an integral but as function of an integral

General problem description

This example is similar to Example 6.1.3. The notable difference in the setting is that now, the boundary control areas, where the fluid can be sucked out of the domain, are in front of the circular inclusion. Hence the drag

$$k(v,p) = \int_{\Gamma_O} n \cdot \sigma(v,p) \cdot d \,\mathrm{d}s,$$

where Γ_O denotes the cylinder boundary, and d is a vector in the direction of the mean in-flow can become negative, i.e., minimizing the value of $k(v,p)$ is no longer viable.

Instead, we consider the objective

$$K(q,v,p) = \frac{1}{2}|k(v,p)|^2 + \frac{\alpha}{2}||q - q_0||^2$$

is to be minimized.

In contrast to all previous examples, this means that the functional can no longer be calculated by one integration, but instead the values of the integration (for the drag) need to be post-processed.

To do so, the calculation of the functional and its derivatives is reordered in to two steps. First the value of $k(v,p)$ is calculated (and stored) then in a second sweep. The value of K is calculated.

To this end the following modifications are needed in the code:

`localpde.h` There is a new method `unsigned int NeedPrecomputations() const` returning the value 1 as we need one calculation of k before we can assemble the value of the cost-functional.

This pre-iteration has the Type `cost_functional_pre` and a corresponding number (here 0 as only one pre-iteration is performed.

In the cost functional, for the pre-iteration we set the type to `boundary` since k is a boundary functional. For the evaluation of K itself the type is `boundary algebraic` as we calculate the boundary integral $\|q\|^2$ and the algebraic calculation $|k|^2$.

For higher derivatives, we notice that for a differentiable functions $g, f\colon \mathbb{R} \to \mathbb{R}$ it holds for the directional derivative in direction δu

$$\begin{aligned}\left((g\left(\int f(u(x))\,\mathrm{d}x\right)\right)' \delta u &= g'(\int f(u(x))\,\mathrm{d}x) \int f'(u(x))\delta u(x)\mathrm{d}x \\ &= \int g'(\int f(u(x))\,\mathrm{d}x) f'(u(x))\delta u(x)\mathrm{d}x\end{aligned}$$

Consequently, the first derivative can be calculated with only a single integration – as a boundary integral in the present example where the factor g' can be calculated using the drag value in the last iterate

$$g'(k(v,p)) \quad \text{and} \quad k'(v,p)\delta u(x) = k(\delta v, \delta p)$$

since the drag is linear in (v, p).

For the second derivative, we can calculate in the directions δu and τu by the following observation

$$\begin{aligned}
&\left(g\left(\int f(u(x))\,\mathrm{d}x\right)\right)''(\delta u, \tau u) \\
&\qquad = g''(\int f(u(x))\,\mathrm{d}x)\int f'(u(x))\delta u(x)\mathrm{d}x\int f'(u(x))\tau u(x)\mathrm{d}x \\
&\qquad\quad + g'(\int f(u(x))\,\mathrm{d}x)\int f''(u(x))\delta u(x)\tau u(x)\mathrm{d}x \\
&\qquad = \int\left(g''(\int f(u(x))\,\mathrm{d}x)\int f'(u(x))\delta u(x)\mathrm{d}x\right)f'(u(x))\tau u(x)\mathrm{d}x \\
&\qquad\quad + \int g'(\int f(u(x))\,\mathrm{d}x)f''(u(x))\delta u(x)\tau u(x)\mathrm{d}x.
\end{aligned}$$

Consequently, the second derivative can be calculated as one boundary integral, if the values of

$$\int f(u(x))\,\mathrm{d}x \quad \text{and} \quad \int f'(u(x))\delta u(x)\mathrm{d}x$$

in the tangent direction δu are available by pre-computations.

To do so in the present example, the following modifications are needed in the cost functional:

- There is a method `AlgebraicValue` which calculates $0.5x^2$ for a given value x - here the pre-computed value of the drag.
- In the method `BoundaryValue`, we have to distinguish several cases. based upon the problem type evaluated. The current value can be accessed by `GetProblemType()` and if this matches

  cost_functional_pre then we must calculate the drag-value, i.e., $k(u)$.

  cost_functional_pre_tangent then we must calculate the derivative of the drag in direction δu (the provided tangent-direction), i.e., $k(\delta u)$ since k is linear in its argument.

  cost_functional then we must calculate the rest of the functional. Here only the integral over the control costs is needed.

- In the methods `BoundaryValue_U` and `BoundaryValue_UU` we calculate the entire derivative w.r.t u (or second derivative respectively) as one integral using the formulas for the first and second derivative, respectively. The needed values, i.e.,

  $$\int f(u)\mathrm{d}x = \int_{\Gamma_O} n\cdot\sigma(v,p)\,d\mathrm{d}s$$

  and

  $$\int f'(u(x))\delta u(x)\mathrm{d}x = \int_{\Gamma_O} n\cdot\sigma(\delta v,\delta p)\,d\mathrm{d}s$$

are available as `ParamValues` with the respective labels `cost_functional_pre` and `cost_functional_pre_tangent`.

Notice, that both `ParamValues` are a vector of the size given by `unsigned int NeedPrecomputations() const`. I.e. if multiple functional parts need a pre-integration these can be calculated in an arbitrary number of pre-integration runs. In the case of multiple pre-integrations the method `*Value` needs not only to consider the value of `GetProblemType()` but also the respective `GetProblemNum()` running between 0 and `NeedPrecomputations()`-1. The order in the ParamValues Vector corresponds to the chosen order of integration.

6.2 Subject to a Nonstationary PDE

6.2.1 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 and most important a time derivative leading to a nonstationary optimization problem. The governing equation is

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

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

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

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

Program description

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

First, we need to introduce a reasonable dual time-stepping scheme in the main file since we have to compute the adjoint equation backward in time. Next, 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_ElementRhs` and `Init_ElementRhs_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_ElementRhs` 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" Element-terms in the PDE we assume in the program that the `Init_ElementEquation` is linear in the state and no other solution variables are present. Thus no derivatives of the `Init_ElementEquation` need to be implemented.

6.2.2 Control of the heat equation via a space dependent right hand side

General problem description

This example is a modified version of `OPT/InstatPDE/Example1`. Again, we consider the heat equation, this time without an additional nonlinear term. The governing equation is

$$\partial_t u(t,x,y) - \Delta u(t,x,y) = f(t) \cdot q(x,y),$$

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

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

With the cost functional

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

It has the optimal solution

$$\begin{aligned} \overline{u}(t,x,y) &= \sin(x)\sin(y), \\ \overline{q}(x,y) &= \sin(x)\sin(y) \end{aligned}$$

together with the optimal adjoint state

$$\overline{z}(t,x,y) = \frac{e^2}{1-e^2} e^{2(t-1)} \sin(x)\sin(y).$$

and the corresponding cost functional value

$$J(\overline{q},\overline{u}) = \frac{1}{2}\Big(\frac{e^4}{(e^2-1)^2} + 1\Big)\frac{\pi^2}{4} \approx 2.88382.$$

Program description

In this example, we demonstrate how to implement a control that acts distributed in space and time, but has no temporal dependence. For this, the control vector type is set to be ControlType::stationary (the default for stationary equations). In contrast to the case of control in the initial values, the control vector can be accessed at a all times.

Since the control has no time dependence `DOpElib` assumes that the control part of the cost functional, here

$$\frac{1}{2}\int_\Omega q^2\, dx$$

is evaluated at initial time (i.e., $t = 0$ in this example).

6.2.3 Control of the heat equation via a time dependent right hand side

General problem description

This example is a modified version of `OPT/InstatPDE/Example2`. The governing equation is

$$\partial_t u(t,x,y) - \Delta u(t,x,y) = q(t) \cdot f(x,y),$$

with homogeneous Dirichlet-data. Hence, we now allow for a time dependence of the control.

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

$$\begin{aligned} f(x,y) &= \sin(x)\sin(y), \\ u_0(x,y) &= 0, \\ u^d(t,x,y) &= \sin(x)\sin(y)\big(\tfrac{1}{4}(3-2t-3e^{-2t}) + \tfrac{4}{\pi^2} + (1-t)\tfrac{8}{\pi^2}\big). \end{aligned}$$

With the cost functional

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

It has the optimal solution

$$\begin{aligned} \overline{u}(t,x,y) &= \tfrac{1}{4}(3-2t-3e^{-2t})\sin(x)\sin(y), \\ \overline{q}(t) &= 1-t \end{aligned}$$

together with the optimal adjoint state

$$\overline{z}(t,x,y) = \sin(x)\sin(y)\frac{4(t-1)}{\pi^2}.$$

Additionally, we evaluate the following functionals

$$\begin{aligned} \overline{u}_h(1, 0.5\pi, 0.5\pi) &= \frac{1-3e^{-2}}{4} \approx 0.148499, \\ \|\overline{u}_h - \overline{u}\|^2_{\Omega\times I}, \\ \|\overline{q}_h - \overline{q}\|^2_I. \end{aligned}$$

Program description

In this example, we demonstrate how to implement a control that acts distributed in time. For this, the control vector type is set to be `ControlType::nonstationary`. Obviously one can implement space and time dependence by considering a ControlVector in dimension $\neq 0$.

6.2.4 Minimizing the spatial mean-value - cost functional involves functions of integrals

General problem description

This example is a modified version of `OPT/InstatPDE/Example1`. Again, we consider the heat equation, this time with an additional nonlinear term and most important a time derivative leading to a nonstationary optimization problem. The governing equation is

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

with homogeneous Dirichlet-data. The computational domain is $\Omega \times I = [0,\pi]^2 \times [0,1]$ with the data

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

In contrast to `OPT/InstatPDE/Example1`, we modify the cost-functional such that a (spatial) mean-value of zero is desired for the state. The corresponding functional is

$$\min_{q,u} J(q,u) = \int_0^1 \frac{1}{2}\left|\int_\Omega u(t,x,y)\,\mathrm{d}(x,y)\right|^2 \mathrm{d}t + \frac{1}{2}\int_\Omega (q(x,y)-\sin(x)\sin(y))^2\,\mathrm{d}(x,y).$$

Program description

The following things differ from the previous examples:

We are now considering a cost-functional, that can not be evaluated as one single integral. Instead, we need to evaluate the integral $\int_\Omega u(t,x,y)\,d(x,y)$ in space first in each time step and then integrate over its absolute value in time. We have seen a similar structure in `OPT/StatPDE/Example10` 6.1.10. Here, the integral in question is of the form

$$\int_I g(\int f(u(t,x))\mathrm{d}x)\mathrm{d}t$$

Consequently, we need to set `unsigned int NeedPrecomputations() const` to the value 1 as we need to calculate – in each time point – the values $\tilde{f}(t) = \int f(u(t,x))\mathrm{d}x$ prior to the evaluation of the cost functional. Then the first derivative of the functional in the direction δu is given as

$$\begin{aligned} \left(\int_I (g\left(\tilde{f}(t)\right)\mathrm{d}t\right)'\delta u &= \int_I g'(\tilde{f}(t))\int f'(u(t,x))\delta u(t,x)\mathrm{d}x\;\mathrm{d}t \\ &= \int_I \int g'(\tilde{f}(t))f'(u(t,x))\delta u(t,x)\mathrm{d}x\;\mathrm{d}t \end{aligned}$$

And for the second derivative, we also calculate the tangential derivatives

$$\tilde{f}_u(t) = \int f'(u(t,x))\delta u(t,x)\mathrm{d}x = \int_\Omega \delta u(t,x,y)\,\mathrm{d}(x,y).$$

With this the second derivative in direction δu and τu is given as

$$\begin{aligned}
&\left(\int_I g\left(\tilde{f}(t)\right)\mathrm{d}t\right)''(\delta u, \tau u) \\
&\quad= \int_I g''(\tilde{f}(t))\tilde{f}_u(t)\int f'(u(t,x))\tau u(t,x)\mathrm{d}x\ \mathrm{d}t \\
&\qquad+ \int_I\int g'(\tilde{f}(t))f''(u(t,x))\delta u(t,x)\tau u(t,x)\mathrm{d}x\ \mathrm{d}t \\
&\quad= \int_I\int \left(g''(\tilde{f}(t))\tilde{f}_u(t)f'(u(t,x)) + g'(\tilde{f}(t))f''(u(t,x))\delta u(t,x)\right)\tau u(t,x)\mathrm{d}x\ \mathrm{d}t.
\end{aligned}$$

As in `OPT/StatPDE/Example10` the higher derivatives can be calculated by one space-time integral if the needed values $\tilde{f}$ and $\tilde{f}_u$ are available at the different time-points. These values can be accessed as ParamValues with the respective labels `cost_functional_pre` and `cost_functional_pre_tangent`.

One further thing differs from the previous examples. This is that the cost functional now has one part being integrated over space-time while the other is acting in space (at initial-time) only. The two types of functionals can at present not be mixed. Hence both terms are evaluated as a distributed integral over space-time by adding an additional weight to the control-cost integral. This weight needs to be specially modified for the derivatives w.r.t. the control which are evaluated at the initial time only! This modification is not exact and consequently the derivatives are not exactly calculated. The accuracy is increased as the temporal meshes are refined. Hence, if more accuracy in the residual is desired the temporal mesh needs to be further refined.

7 Regression Tests

The `DOpElib` 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 `DOpElib`. Please make sure what is going wrong and WHY!
- Every command is computed via a Makefile in the basic example directory.

Remark 7.0.1. Please keep in mind that there are two optimization examples that need external packages to be able to run successfully. These tests will fail if these libraries are not installed!

7.1 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.

7.2 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

```
> ./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, Timo Heister, and Guido Kanschat. *Differential Equations Analysis Library*, 2013.

[3] Wolfgang Bangerth and Rolf Rannacher. *Adaptive Finite Element Methods for Differential Equations*. Birkhäuser Verlag, 2003.

[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] M.A. Biot. Consolidation settlement under a rectangular load distribution. *J. Appl. Phys.*, 12(5):426–430, 1941.

[6] M.A. Biot. General theory of three-dimensional consolidation. *J. Appl. Phys.*, 12(2):155–164, 1941.

[7] M.A. Biot. Theory of elasticity and consolidation for a porous anisotropic solid. *J. Appl. Phys.*, 25:182–185, 1955.

[8] H.-J. Bungartz and M. Schäfer. *Fluid-Structure Interaction: Modelling, Simulation, Optimization*, volume 53 of *Lecture Notes in Computational Science and Engineering*. Springer, 2006.

[9] The Differential Equation and Optimization Environment: DOpElib. `http://www.dopelib.net`.

[10] The finite element toolkit Gascoigne. `http://www.gascoigne.uni-hd.de`.

[11] Jaroslav Hron and Stefan Turek. *Proposal for numerical benchmarking of fluid-structure interaction between an elastic object and laminar incompressible flow*, volume 53, pages 146 – 170. Springer-Verlag, 2006.

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

[13] T. Richter and T. Wick. Optimal control and parameter estimation for stationary fluid-structure interaction. *SIAM J. Sci. Comput.*, 35(5):B1085–B1104, 2013.

[14] RoDoBo: A C++ library for optimization with stationary and nonstationary PDEs. `http://www.rodobo.uni-hd.de`.

[15] M. Schäfer and S. Turek. *Flow Simulation with High-Performance Computer II*, volume 52 of *Notes on Numerical Fluid Mechanics*, chapter Benchmark Computations of laminar flow around a cylinder. Vieweg, Braunschweig Wiesbaden, 1996.

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

[17] Thomas Wick. Fluid-structure interactions using different mesh motion techniques. *Comput. Struct.*, 89(13-14):1456–1467, 2011.

Index