OOPDE

An object oriented toolbox for finite elements in Matlab

Quickstart Guide

Uwe Prüfert*

Version 2015b†

*Technische Universität Bergakademie Freiberg, Institute for Numerical Analysis and Control, Freiberg, Germany
†Printed February 26, 2016
## Contents

1 OOPDE .............................. 2
  1.1 Introduction ......................... 2
  1.2 Finite Elements in a Nutshell .......... 5
  1.3 Excursion. Definition of the boundary conditions ......... 10
  1.4 Installation .......................... 11
  1.5 Object oriented programming in MATLAB ............. 12
    1.5.1 Some background ................... 12
    1.5.2 Defining a class in MATLAB ......... 13
    1.5.3 MATLAB special issues ............. 15
    1.5.4 The use of multiple inheritance ....... 16

2 Examples ............................. 17
  2.1 Geometries .......................... 17
    2.1.1 1D “meshes” ....................... 18
    2.1.2 2D meshes .......................... 19
    2.1.3 Be save: write a classes for our geometry ........ 28
    2.1.4 3D meshes .......................... 33
  2.2 Solving PDEs ........................ 39
    2.2.1 A heat transfer problem ............ 39
    2.2.2 Burgers’ equation ................. 43
    2.2.3 A parabolic PDE with time dependent data ......... 46
    2.2.4 The use of P2 elements ............ 49
    2.2.5 Dirichlet boundary conditions ....... 51
  2.3 Systems of PDEs ...................... 53
    2.3.1 An optimal control problem without inequality constraints .... 53
    2.3.2 Reacting Flow—A system of PDEs with non linear coupling .... 56
    2.3.3 KKT system for a problem with control constraints .......... 61
    2.3.4 The Schnakenberg Model ............ 72
  2.4 Adaptive Mesh refinement ............. 78
    2.4.1 General approach ................... 78
    2.4.2 Ad-hoc adaptive solving ............ 79
    2.4.3 Writing an adaptive solver .......... 80
  2.5 Some special issues .................. 82
    2.5.1 The meaning of pde.initialized .......... 82
    2.5.2 Resetting pde.fc and pde.g ........... 82

3 License ............................. 84
  3.1 GNU GENERAL PUBLIC LICENSE (GPL) .............. 84
Chapter 1

1 OOPDE

1.1 Introduction

The idea behind OOPDE was to extend MATLAB’s old pdetoolbox. The users—intentionally undergraduate students—should be able to solve simple partial differential equations (PDE) and PDE constrained optimal control problems without a deeper knowledge of the finite element method (FEM) or advanced programming in Matlab. However, a welcome add-on should be the possibility to solve 1D problems (rather trivial) with the same syntax as the PDE toolbox as well as to solve also problems with a convection term, which is not implemented in the pdetoolbox at all.

Beginning with R2010b, MATLAB offers a “new” syntax to write object oriented code. This re-implementation of object oriented programming (OOP) in MATLAB provides now the possibility to write clean, robust and readable code in a C++ like syntax.

At this point, we start to re-write the code in the new object oriented syntax and write a system of classes, that provides an user-friendly, open, readable, powerful MATLAB code, that helps students to solve PDE problems and e.g. optimality systems of PDE constrained optimal control problems. Using the concept of abstract classes and inheritance, OOPDE can be easily extended by the users. For instance, to implement quadratic finite elements, we need only to add a few lines of code while the main methods as for instance matrix assembling is done by abstract classes and methods.

Solving PDE problems with the FEM has a typical work flow:

(i) Define the PDE problem.
   a) Define the domain.
   b) Define the boundary conditions.
   c) Define initial conditions.
   d) Define the PDE.

(ii) Choose suitable finite elements.

(iii) Solve the algebraic/time dependent system.

(iv) Post process the solution.

\[^{1}\text{MATLAB is a registered trademark of The MathWorks Inc.}\]
The idea is to define classes representing the domain, the finite element method and the PDE. These classes provides all methods to setup and solve the problem and finally to post-process the solutions etc. The implementation details should be hidden from the common user.

As an example we solve the simple problem

\[-\Delta u + u = 10\]  \hspace{1cm} (1.1)

in—only for simplicity, OOPDE provides methods to to define a wide variety of geometries—the domain \(\Omega = (0, 1)^2\) and with the Dirichlet boundary condition

\[u = 0\]

on the boundary \(\partial \Omega\). We set the dimension of the problem to two and define \(\Omega\) as the unit square. Using OOPDE, the following code solves the problem.

First we define a user class that defines the problem.

```matlab
1 classdef heatTransfer < pde
2     methods(Access = protected)
3         function dy = df(obj,~,y)
4             s = obj.fem.stiffSpring(obj.K+obj.M);
5             dy = -(obj.K+obj.M+...
6                 s*(obj.H'*obj.H))*y +...
7                 obj.F+s*(obj.H'*obj.R);
8             end
9         end
10 end
```

Users of the MATLAB pdetoolbox (as well as Femlab or early versions of COMSOL Multiphysics) may understand this code immediately, but for all others we will explain it in detail. Readers not familiar with the finite elements method should read the Section 1.2 first.

We go through the code line by line:

<table>
<thead>
<tr>
<th>line</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The class <code>heatTransfer</code> is derived from the OOPDE class <code>pde</code>. The <code>pde</code> class will give us all methods needed to solve the problem.</td>
</tr>
<tr>
<td>2</td>
<td>We declare the methods as “protected”. This is required by the parent class <code>pde</code> from OOPDE.</td>
</tr>
</tbody>
</table>
| 3–8  | We define the method `df`, OOPDE requires the formulation of the PDE in the form  
|      | \(0 = f(y, x)\)  
|      | for stationary problems or  
|      | \(\frac{d}{dt}y = f(y, t, x)\)  
method which were introduced in the original MATLAB pde-toolbox. In particular, \texttt{obj.K} is the stiffness matrix, \texttt{obj.M} is the mass matrix, \texttt{obj.F} is the source vector and \texttt{obj.Q}, \texttt{obj.H}, \texttt{obj.G} and \texttt{obj.R} are the discrete counterparts to the boundary integrals in the weak formulation of the PDE, for details see Section 1.2.

4,6, and 7 The variable \texttt{s} is the “stiff spring” coefficient to model Dirichlet-Boundary conditions.

That is all user the code we have to write for this example. The following “main program” uses exclusively methods provided by OOPDE:

```java
1 pde = heatTransfer();
2 pde.fem = lagrange12D();
3 pde.grid = grid2D();
4 pde.grid.unitSquare(0.05)
5 pde.grid.refineMesh();
6 pde.grid.makeBoundaryMatrix(pde.grid.dirichletBC('0'));
7 pde.initialize(1,[0;0],1,10);
8 pde.solve('LINEAR')
9 pde.grid.plot(pde.y)
```

Again, we comment the code line by line:

<table>
<thead>
<tr>
<th>line</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create the object \texttt{pde} from class \texttt{heatTransfer}. This corresponds to point (ib) in the agenda.</td>
</tr>
<tr>
<td>3</td>
<td>Create a \texttt{lagrange12D} object and assign it to \texttt{pde.fem}. This corresponds to point (ii) in the agenda.</td>
</tr>
<tr>
<td>5</td>
<td>Create a \texttt{grid2D} object and assign it to \texttt{pde.grid}. The class \texttt{grid2D} is also provided by OOPDE. This, together with lines 6 and 7, corresponds to point (ia) in the agenda.</td>
</tr>
<tr>
<td>6</td>
<td>Mesh the unit square with mesh size 0.05.</td>
</tr>
<tr>
<td>7</td>
<td>Refine the mesh uniformly.</td>
</tr>
<tr>
<td>9</td>
<td>Create the Dirichlet boundary conditions. The unit square has four boundaries, we can define it once and for all or separately. This corresponds to point (ic) in our agenda.</td>
</tr>
<tr>
<td>11</td>
<td>Call \texttt{pde.initialize}. Despite the fact that our definition of class \texttt{heatTransfer} does not contain a convection term, we nonetheless pass a zero convection term. This is so because the initialize method called here is that of the parent class \texttt{pde} and requires four parameters. The can be avoided by overwriting initialize in the \texttt{heatTransfer} class. The matrices used in the method df are computed here, \texttt{obj.K} etc. are properties of the class \texttt{pde}.</td>
</tr>
</tbody>
</table>
Solve the problem using the “LINEAR” option. This is point (iv) in the agenda, while point (id) is worked off by defining the class heatTransfer, cf. the listing above.

Plot the solution. This corresponds to item (iv) on our agenda.

In contrast to most MATLAB PDE software, e.g. the MATLAB pdetoolbox, in OOPDE there is no pre-defined PDE problem where the user has only to provide some coefficients. The definition of the PDE itself lies in the hand of the user. It must be by done coding a user class derived from the OOPDE class pde, cf. the listing above. This is a compromise between flexibility with respect the problem class that can be solved and a user-friendly interface.

Note that in later sections of this guide we will improve the heatTransfer class also by changing the interface of the initialize method by overwriting it.

If the user keeps in mind some basic rules, OOPDE offers him the opportunity to use this “toolbox” to solve a wide variety of PDE problems like convection-diffusion problems, coupled systems of PDEs, vector-valued PDEs, and much more.

As you can see in the listing of the driver script, the pde class method initialize computes without any additional user input the matrices needed to solve simple problems involving mass, stiffness, and convection matrices, as well as Dirichlet and Neumann boundary conditions matrices. With this at hand, the user can solve problems like Burgers’ equation out of the box.

However, we have a general rule: “The more individual the problem, the deeper one must understand the software”. In summary, OOPDE provides

(i) A uniform interface for PDE problems with 1, 2, 3 space dimensions.

(ii) Lagrange elements of order one for dimension 1–3. This can be easily extended by the user since OOPDE offers an open interface for that purpose.

(iii) A wide variety to define the functions defining PDE data.

(iv) A user-friendly interface to define boundary conditions.

In Chapter 2 we will illustrate all this features in some examples.

### 1.2 Finite Elements in a Nutshell

Let \( \Omega \subset \mathbb{R}^n \), \( n = 1, 2, 3, \) be open and let \( \Gamma \) be the boundary of \( \Omega \). We consider the scalar (initial) boundary problem for the partial differential equation (PDE)

\[
\begin{align*}
\frac{d}{dt} u & - \nabla \cdot (c \nabla u) + \vec{b} \cdot \nabla u + a u = f & \text{in } \Omega \times (0,T) \\
\vec{n} \cdot (c \nabla u) + q u &= g & \text{on } \Gamma \times (0,T) \\
u &= u_0 & \text{in } \Omega
\end{align*}
\]

(1.2)(1.3)(1.4)

where \( c \) is a positive definite matrix \( c : \Omega \to \mathbb{R}^{n,n} \). The function \( u \) depends form the spatial variable \( x \), or—for time-dependent problems, i.e. \( t \neq 0 \)—from the time \( t \) and the spatial variable \( x \), i.e. \( u = u(x) \) or \( u = u(t,x) \), respectively. Note that for \( n = 1 \) the \( \nabla \) operator means \( \frac{d}{dx} \) and all coefficients are scalar functions. \( \vec{n} \cdot (c \nabla u) \) means the unit outward normal derivative. By bold letters we denote here the data. All data can
depend on the spatial variable $x$ and—if applicable—on the time $t$. Note that by these dependencies the type of this equation can change with time. Data can be identically zero, e.g. with $c = a \equiv 0$, (1.3) is a hyperbolic equation, and for $d = 0$ we have an elliptic equation. Note that if $d = 0$ the time interval $(0, T)$ disappears.

By multiplying (1.2) with a test function $v \in H^1(\Omega) =: V$ and using Green’s formula we obtain the weak formulation of (1.2) as the integral equation

$$\int_{\Omega} d \left( \frac{d}{dt} u \right) v \, dx + \int_{\Omega} (c \cdot \nabla u) \nabla v \, dx + \int_{\Omega} \mathbf{b} \cdot (\nabla u) v \, dx + \int_{\Omega} \mathbf{a} u v \, dx - \int_{\Gamma} \mathbf{n} \cdot (c \nabla u) v \, ds = \int_{\Omega} f v \, dx \quad \forall v \in V.$$

Now we can replace the integrand within the boundary integral by the Robin boundary condition (1.3), i.e.

$$- \int_{\Gamma} \mathbf{n} \cdot (c \nabla u) v \, ds = \int_{\Gamma} (q u - g) v \, ds.$$

We can write (1.2) as

$$\int_{\Omega} d \left( \frac{d}{dt} u \right) v \, dx + \int_{\Omega} (c \nabla u) \nabla v \, dx + \int_{\Omega} \mathbf{b} \cdot (\nabla u) v \, dx + \int_{\Omega} \mathbf{a} u v \, dx + \int_{\Gamma} q u v \, ds$$

$$= \int_{\Omega} f v \, dx + \int_{\Gamma} g v \, ds \quad \forall v \in V, \quad (1.5)$$

while in the case of time-dependent problems, i.e. $d \neq 0$ we additionally have $u(0) = u_0$. Next, we replace $V$ by a finite dimensional subspace, $V_h \subset V$ and consider $u_h = \sum_{i=1}^{nx} u_i \phi_i$ and $v_h = \sum_{j=1}^{nx} v_j \phi_j$ where the $\phi_i$ are a set of basis functions of $V_h$ and $nx$ is the number of basis functions. The idea is that instead of testing with all $v \in V_h$ it is sufficient to test only against all basis functions of $V_h$. We choose now $\phi_i$ such that $\phi_i \in H^1(\Omega_\Lambda)$, where $\Omega_\Lambda$ is an approximation of $\Omega$ by—depending on the dimension of $\Omega$—triangles or tetrahedral. The simplest member of this class of functions are piecewise linear. In one and two dimensional spaces they were often called hat-functions. Actually, in three dimensional space it is hard to imagine how they look like. The number of basis functions is here equal the number of mesh points.

Equation (1.5) becomes

$$\sum_{j=1}^{nx} u_j \left( \int_{\Omega_\Lambda} d(x) \frac{d}{dt} \phi_j(x) \, dx + \int_{\Omega_\Lambda} c(x) \nabla \phi_j(x) \nabla \phi_j(x) \, dx + \int_{\Omega_\Lambda} \mathbf{b}(x) \cdot (\nabla \phi_j(x)) \phi_j(x) \, dx + \int_{\Omega_\Lambda} \mathbf{a}(x) \phi_j(x) \phi_j(x) \, dx + \int_{\Gamma_\Lambda} q(x) \phi_j(x) \phi_j(x) \, ds \right) = \int_{\Omega_\Lambda} f(x) \phi_j(x) \, dx + \int_{\Gamma_\Lambda} g(x) \phi_j(x) \, ds$$

for $i = 1, \ldots, nx$, where we assume that all coefficient functions are from $L^\infty(\Omega)$ or $L^\infty(\Gamma)$ and can be approximated by piecewise constant functions on the interior of every single

---

2To be more precise, we should write $u_h = u_h(t, x) = \sum_{i=1}^{nx} u_i(t) \phi_i(x)$ for time dependent problems, or $u_h(x) = \sum_{i=1}^{nx} u_i \phi_i(x)$ for stationary problems. Especially by the splitting of $u_h(t, x)$ into a time-dependent coefficient $u_i(t)$ and a spatial dependent basis function $\phi_i(x)$ we can transform the time-dependent PDE problem into a semi-continuous ODE problem.
element of $\Omega$. Note that coefficient function can depend from $x$ and $u(x)$, and for time-dependent problems also from $t$. After the evaluation of all integrals we finally obtain the possibly non-linear system

$$D \frac{d}{dt} u + (K + M + C + Q)u = (F + G),$$

with $u = u_j$, $i = 1, \ldots, nx$. Note that we use here the same notation $u$ for the function as well as for the coefficient vector of $u_h = \sum u_i \phi_i$. The coefficient $\tilde{b}$ is a vector of length $N$, i.e. for problems with $\Omega \subset \mathbb{R}$ it is a scalar function and for $N = 2, 3$ it is $\tilde{b} = [b(x)_1 \ldots b(x)_N]$. For time dependent problems it can also depend—as all coefficients—from time $t$.

The matrices and vectors are in detail

$$D_{ij} = \int_{\Omega} d(x)\phi_j(x)\phi_i(x) \, dx$$

$$K_{ij} = \int_{\Omega} c(x)\nabla \phi_j(x)\nabla \phi_i(x) \, dx$$

$$C_{ij} = \int_{\Omega} \tilde{b}(x) \cdot (\nabla \phi_j(x)) \phi_i(x) \, dx$$

$$M_{ij} = \int_{\Omega} a(x)\phi_j(x)\phi_i(x) \, dx$$

$$Q_{ij} = \int_{\Gamma} q(x)\phi_j(x)\phi_i(x) \, ds$$

$$F_i = \int_{\Omega} f(x)\phi_i(x) \, dx$$

$$G_i = \int_{\Gamma} g(x)\phi_i(x) \, ds$$

for $i, j = 1, \ldots, n_p$. Obviously, this holds for $\Omega \subset \mathbb{R}^n$ and $\Gamma \subset \mathbb{R}^{n-1}$ with $n = 1, 2, 3$.

Note that for time dependent data the related matrices and vectors depend on time, hence they must be updated in every time step of the time integration. Similarly, for non-linear problems at least on matrix or vector depends on the solution $u$ and must be updated within the solution process, e.g. within a Newton iteration.

Since $\Omega$ consists of a finite number of elements we can write the definitions above also as a sum of integrals over the single elements $(\Omega_l)_l$, where $l = 1, \ldots, n_e$.

Additional one can define Dirichlet boundary conditions

$$h u = r \text{ on } \Gamma_D$$

in a PDETOOLBOX like style. The associated matrix $H$ and the vector $R$ do not arise from the integrals but they define point-wise the Dirichlet boundary conditions by defining a constraint. We consider the problem

$$-\nabla (c\nabla u) + au = f$$

$$\bar{n} \cdot c \nabla u = 0$$

The bilinearform
\[ A(u, v) := \int_{\Omega} c \nabla u \nabla v + auv \, dx \]

is symmetric and defines an inner product of \( H^1(\Omega) \) and hence also a norm. The term \( F(v) := \int_{\Omega} fv \, dx \) defines a linear form. We can see solving the PDE problem as solving the minimization problem

\[ \min f(u) := A(u, u) - F(u). \]

Now we consider the constrained problem

\[ \min f(u) := \frac{1}{2} A(u, u) - F(u) \]

subject to

\[ hu = r \text{ on } \Gamma. \]

The discrete version of this problem reads

\[ \min g(u) := u^\top Ku + u^\top Mu - Fu \]

subject to

\[ Hu = R. \]

The matrix \( H \in \mathbb{R}^{n_d \times n_d} \) is defined by

\[ H_{ij} := \begin{cases} r(x_i) & \text{if } x_i \text{ is a Dirichlet boundary point} \\ 0 & \text{otherwise} \end{cases} \]

and \( i = 1, \ldots, n_d \), where \( n_d \) is the number of Dirichlet boundary points. The vector \( R \) is defined by \( R_i = r(x_i) \) for all \( i = 1, \ldots, n_d \).

By formal use of the Lagrange technique we define the Lagrange function

\[ L(u, \lambda) = g(u) - \lambda^\top (Hu - R). \]

The optimality system reads

\[
\begin{align*}
(K + M)u + H^\top \lambda &= F \\
Hu &= R,
\end{align*}
\]

The matrix \( H \) and the vector \( R \) will be also generated automatically by OOPDE.

By this, we obtain an alternative approach to Dirichlet boundary conditions.

```matlab
1 classdef heatTransferD < pde
2     methods (Access = protected)
3         function dy = df(obj,~,y)
4         N = sparse(size(obj.H,1),size(obj.H,1));
5         dy = [- (obj.K+obj.M+obj.Q) obj.H'...]
6             obj.H N]*y + ...
7             [obj.F obj.R];
8     end
9 end
10 end
11```
1.2 Finite Elements in a Nutshell

Note that this code has some issues we will discuss later.

A different approach to involve the equality constraint

\[ hu = r \]

is to add a penalty term to the functional, i.e. in the discrete case we solve

\[
\min g(u) := u^\top Ku + u^\top Mu - Fu + s(Hu - R),
\]

where \( s \) is a sufficient large number (which has to be chosen). The derivative of \( g(u) \) reads

\[
Ku + Mu - F + sH^\top Hu - H^\top R,
\]

i.e. we have to solve

\[
0 = -(K + M + sH^\top H)u - (F + H^\top R).
\]

If we compare this with the code of our introducing example `heatTransfer`, we identify all matrices and vectors used in the code of `df` method.

Since Matlab is able to handle empty matrices in a smart way, we can define a linear PDE problem always using all possible matrices and vectors, even in the case they are empty. The code

```matlab
classdef heatTransfer < pde
    methods(Access = protected)
    function dy = df(obj,~,y)
        s = obj.fem.stiffSpring(obj.K+obj.M);
        dy = -(obj.K+obj.M+obj.Q... 
            s*(obj.H'*obj.H))*y + ...
            obj.F+s*(obj.H'*obj.R)+obj.G;
    end
end
```

is prepared to solve problems

\[
-\nabla \cdot (c \nabla u) + au = f \text{ in } \Omega
\]

\[
\vec{n} \cdot \nabla u + qu = g \text{ on } \Gamma_R
\]

\[ hu = r \text{ on } \Gamma_D
\]

even in the case \( \Gamma_D \) or \( \Gamma_N \) are empty.

The relation between the coefficients names and the associated matrices may on a first view to be not straight-forward. While the coefficients \( d, f, q, g, h \) and \( r \) belong to its capitalized matrix and vector counterparts, the coefficient \( c \) belongs to the Stiffness matrix \( K \), the coefficient \( a \) belongs to the mass matrix \( M \) and the coefficient \( \vec{b} \) belongs to the convection matrix \( C \). The naming in the analytical formulation of our general PDE problem follows the literature, e.g. the PDEToolbox user guide. The naming of the matrices is common in textbooks, except for \( C \) and \( F \) which are named after “convection” and “force”.

9
1.3 Excursion. Definition of the boundary conditions

The method `makeBoundaryMatrix`

The method `makeBoundaryMatrix` from class `gridd` creates the boundary condition matrix. It stores only the information which boundary conditions is defined for the associated boundary segments. Note that the number of boundary segments is a property of the geometry respectively of the grid object, that can be changed by the user, see the example in Section ??.

The matrices Q, H and the vectors R, G will be computed by the method `assemble` from class `fem`, by default encapsulated in the `initialize` method from class `pde`.

How to “read” the boundary condition matrix

Since OOPDE uses the format of boundary condition definition of MATLAB’s PDE toolbox `pdetool`, we quote the manual of `pdetool`:

“The format of each column is according to the following list:

- Row one contains the dimension N of the system.
- Row two contains the number M of Dirichlet boundary conditions.
- Row three to $3 + N^2 - 1$ contain the lengths for the strings representing $q$. The lengths are stored in column-wise order with respect to $q$.
- Row $3 + N^2$ to $3 + N^2 + N - 1$ contain the lengths for the strings representing $g$.
- Row to $3 + N^2 + N$ to $3 + N^2 + N + MN - 1$ contain the lengths for the strings representing $h$. The lengths are stored in column-wise order with respect to $h$.
- Row $3 + N^2 + N + NM$ to $3 + N^2 + N + MN + M - 1$ contain the lengths for the strings representing $r$.

The following rows contain text expressions representing the actual boundary condition functions. The text strings have the lengths according to above. The MATLAB text expressions are stored in column-wise order with respect to $h$ and $q$. There are no separation characters between the strings.” [The95]

Since OOPDE handles systems of PDEs as combinations of coupled scalar PDE problems, in OOPDE we have $N = 1$.

Example. The boundary condition matrix

```
1 1 1 1
0 0 0 0
1 1 1 1
6 6 6 6
48 48 48 48
115 115 115 115
105 105 105 105
110 110 110 110
```
1.4 Installation

codes the Neumann type boundary condition \( \vec{n} \cdot (\mathbf{c} \nabla u) = \sin(s) \), i.e. \( q = 0 \) and \( g = \sin(s) \) where the domain has four boundary segments. We give a detailed explanation for this example:

- first line: scalar problem, \( u \) itself is not a vector. Not relevant in OOPDE, only for compatibility.
- second line: it is not a Dirichlet boundary condition, hence it must be Robin boundary condition.
- third line: the length of the entry for \( q \) is one
- fourth line: the length of \( g \) is six.
- fifth line is \( q \). We have “48”. It is the code number in ASCII code. Applying char returns

\[
\text{char(48)}
\]

0

- The next six lines representing \( g \). Applying the function char we get

\[
\text{char([115 105 110 40 115 41])}
\]

\[
\sin(s)
\]

Despite the fact that Dirichlet-boundary conditions can be approximated by Robin boundary condition, MATLAB’s PDE toolbox can also define Dirichlet boundary conditions \( \mathbf{h}u = \mathbf{r} \) directly. In this case we have the matrix \( H' \mathbf{H} \) for the left hand side and on the right hand side the vector \( H' \mathbf{R} \). This formulation is caused by the definition of Dirichlet boundary conditions as constraints of the PDE, the term \( H' \mathbf{H} \) and \( H' \mathbf{R} \) come from the derivative of this constraint. The somehow unnatural looking splittings \( H' \mathbf{H} \) and \( H' \mathbf{R} \)—this is only a fix matrix and vector that could be stored as one—can be use for rapid updating the boundary conditions in the time dependent and non-linear cases. We demonstrate this in the Chapter Examples.

However, the OOPDE user never needs and never should edit the boundary condition matrix. OOPDE will do it for the user.

1.4 Installation

OOPDE itself is a set of class definition files. Every class is coded in a single file \texttt{classname.m}. All these classes are intentionally not made to be used directly in user code. Most of them define abstract classes inheriting their properties and methods to child classes. These files are in the sub-folder \texttt{classes}.

Predefined user classes are in the sub-folder \texttt{examples}. These classes are in folders starting with an @. There are also driver functions or scripts, respectively, implementing application cases.

To install OOPDE unpack the oopde.zip or oopde.tar.bz package on a suitable place, e.g. in a folder matlab in your home folder. In MATLAB, call the “Set path” dialog.
1 OOPDE

(R<2013 File set path or R>2013 on top of MATLAB command window). Click “Add with subfolders”. Browse to oopde. Click “Save” or “Close”.

To test the installation, try to locate OOPDE files, e.g. the pde.m class definition file by calling the which command. MATLAB should return the path to the pde class definition file, e.g.

```matlab
>> which pde
/home/uwe/Arbeit/Matlab/OOPDE/@pde/pde.m
% pde constructor
>>
```

Alternatively, you can run an example, e.g.

```matlab
>>testConvectionDiffusion2D
```

OOPDE includes only one MATLAB mex file: findpoint.mexa64 or findpoint.mexw32. In case of problems with this files you can create our own by compiling findpoint.c

```matlab
>>mex findpoint.c
```

1.5 Object oriented programming in Matlab

1.5.1 Some background

In this section we give very briefly an overview on object oriented programming (OOP) in MATLAB. The basic concepts of OOP are abstraction, attributes, classes, inheritance, methods, persistence, polymorphism, properties.

**Abstraction** Each object within the system can be seen as an abstract model of a player, which accomplish tasks, reports its state and communicates with other objects within the system, but without reviling open in which form this is implemented. Such abstractions are classes.

**Attributes** Attributes describe the behavior of methods and properties, e.g. the access to a property from outside the class.

**Class** The data of an object is defined by the properties of its class definition. The behavior of the object is determined by its methods. Classes can be derived from other classes. This is called inheritance. The child class inherits the data structure and the methods of its parent class(es).

**Encapsulation** Encapsulation enforces modularity Encapsulation refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called classes, and one instance of a class is an object. Encapsulation ensures good code modularity, which keeps routines separate and less prone to conflict with each other.

**Inheritance** Inheritance passes structure form parent to child classes. Classes are created in hierarchies, and inheritance lets the structure and methods in one class pass down the hierarchy. That means less programming is required when adding functionality existence systems. If a feature is added at the bottom of a hierarchy,
1.5 Object oriented programming in MATLAB

only the processing and data associated with that unique feature must be added. Everything else above that step is inherited.

**Persistence** Objects’ variables exist as long the objects exist and do not expire after the termination of a method.

**Polymorphism** Polymorphism is the provision of a single interface to entities of different classes.

1.5.2 Defining a class in Matlab

Fortunately, the syntax of a MATLAB class definition is very simple.

To write a MATLAB class from scratch, we have to write a MATLAB file that starts with the keyword `classdef` followed by the class name. Close the definition by an `end`.

```matlab
classdef myClass
end
```

Now we can define properties (that store data) and methods (the functions of a class). To include properties in our class, we define one or more properties block(s). Here we can set the properties attributes. The same holds for the definition of methods.

```matlab
classdef myClass
    properties(List_of_Attributes)
        % Define here properties
    end
    properties(List_of_Attributes)
        % Define here properties
    end
    % Add further properties blocks
    methods(List_of_Attributes)
        % Define here methods (functions)
    end
    methods(List_of_Attributes)
        % Define here methods (functions)
    end
    % Add further methods blocks
end
```

Most used attributes used in OOPDE for properties are

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>public, private, protected</td>
</tr>
<tr>
<td>SetAccess</td>
<td>public, private, protected</td>
</tr>
<tr>
<td>GetAccess</td>
<td>public, private, protected</td>
</tr>
<tr>
<td>Constant</td>
<td>true, false</td>
</tr>
<tr>
<td>Hidden</td>
<td>true, false</td>
</tr>
</tbody>
</table>
OOPDE

and for methods

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>public, private, protected</td>
</tr>
<tr>
<td>Static</td>
<td>true, false</td>
</tr>
<tr>
<td>Hidden</td>
<td>true, false</td>
</tr>
</tbody>
</table>

Bold values are default, i.e. an empty attributes list stands for

\[
\text{Access} = \text{public}, \ \text{Constant} = \text{false}, \ \text{Hidden} = \text{false}
\]

Also \text{Access} = \text{value} stands for \text{SetAccess} = \text{value}, \text{GetAccess} = \text{value}.

Alternatively, MATLAB's editor provides a draft class. Click New > Class.

classdef untitled
\%UNTITLED Summary of this class goes here
\% Detailed explanation goes here
properties
end
methods
end
end

The programmer only has to change the name of the class and save it. The file name must be identical with the class name. In the case of a class that consists only of one class definition file, it can be stored somewhere in the MATLAB path. Only in the case that a class consist of more than one files, it is mandatory to create a folder @classname and store all files associated with this class in this folder. Of course, we must fill the dummy properties and methods blocks (and add additional blocks with adapted attributes) and write the code of all methods and properties we need in our class.

The decision to write a single or a multi-file class is more or less a question of personal preference. By smart code folding provided by MATLAB's editor also long files with thousands of lines of code may be readable by the programmer.

Class HeatTransfer revisited

In Section 1.1 we introduced the simple class heatTransfer for solving stationary and transient heat transfer equation.

```matlab
classdef heatTransfer < pde
    methods(Access = protected)
        function dy = df(obj,~,y)
            s = obj.fem.stiffSpring(obj.K+obj.M);
            dy = -(obj.K+obj.M+ ... 
                  s*(obj.H'*obj.H))*y + ...
                  obj.F+s*(obj.H'*obj.R);
        end
        end
end
```
In the first line we have an example for inheritance: `heatTransfer` is derived from the abstract class `pde`. In the second line the method block starts. Here we find the definition of the attribute `Access` as protected.

The following lines define the method `df`. The code of a method is simply a common MATLAB function. In the following we give a two-file version of the class definition for `heatTransfer`.

Now, the classdef file contains only the signature of the method `df`. Note that only the signature (interface) of `df` is given, the keyword `function` is absent.

```matlab
classdef heatTransfer < pde
    methods (Access = protected)
        dy = df(obj,~,y)
    end
end
```

In a second file, we implement `df` as a common MATLAB function. This file must be named `df.m` and must be stored in the `@heatTransfer` folder.

```matlab
function dy = df(obj,~,y)
    s = obj.fem.stiffSpring(obj.K+obj.M);
    dy = -(obj.K+obj.M+...
        s*(obj.H'*obj.H))*y + ...
    obj.F+s*(obj.H'*obj.R);
end
```

To summarize: We can write class definitions also in a C++ like style where the class definition file `classname.m` contains only declarations. All methods are implemented in separate files. However, one can mix these programming styles, but this may lead to a more complex and harder readable code. In this guide, we will write the code of every example class in one single file.

### 1.5.3 Matlab special issues

Writing complex programs is a process of trial and error. Often, some code will not work as expected or will miss-behave in some sense. The usual work flow is to rewrite some lines of code and to run the program again. In some cases, MATLAB objects seems to behave “strange”: The last changes in the code show no effect. Often newly added methods cannot found, but they appear in the methods list when calling

```
>>methods className
```

This is not a bug in MATLAB but it is causes by the fact that an instance of the class in question is already loaded by MATLAB. Usually, MATLAB will print a warning but in some cases not. In this case all classes should be removed from the memory by a call of

```
>>clear classes
```

or (stronger)

```
>>clear all
```

Note that all classes delivered with OOPDE are handle classes. In the example
1 OOPDE

```matlab
>> pde = heatTransfer()
>> clear pde
```

The `clear` command will remove only the handle from the workspace but not remove the `heatTransfer` object from the memory. Use instead

```matlab
>> pde.delete
>> pde = handle to deleted heatTransfer
>> clear pde
```

All handle classes inherit from their parent class `handle` methods.

Methods for class `handle`:

- `addlistener`  
- `eq`  
- `findprop`  
- `gt`  
- `le`  
- `ne`  
- `delete`  
- `findobj`  
- `ge`  
- `isvalid`  
- `lt`  
- `notify`

To learn more about advanced object-oriented programming with MATLAB, read the section Advanced Software Development in the MATLAB documentation.

1.5.4 The use of multiple inheritance

Building object/class hierarchies is not even simple. In OOPDE we use an rather linear approach. However, sometimes one wish to add methods to classes from case to case. For instance, the class `pde` is not specialized for elliptic or parabolic PDEs. This is not a problem and for applications, sometimes this is rather a clever idea, for instance if the user is interested in the solution of a non-linear stationary problem but to obtain a good initial choice he wants to solve an artificial transient problem.

On the other hand, post processing PDE solutions depends on the dimension of the problem and if the solution depends from time or not. This is the reason for adding the plot methods to the (spatial dimension depending) grid classes. Time dependent solutions can be shown as snapshots or animations, where plotting the single snapshot can be done by using `gridXD` methods.

Not it may a good idea to collect specialized plot and animation methods in a package, that can be added to user classes if the user needs time-dependent visualization. For that purpose, OOPDE contains some abstract declared classes. These classes contain only non-abstract methods, implementing algorithms for visualization, but no properties.

For instance, the class `plotUtilsTimeDependent1D` provides the methods `plotTimesSpace` and `animation`.

To add this methods to a user class, we use multiple inheritance. By

```matlab
classdef userClass < pde & plotUtilsTimeDependent
```

we derive `userClass` from `pde` and from `plotUtilsTimeDependent`.

Since the `plotUtilsTimeDependent` class has no properties, no name conflicts (the main disadvantage of multiple inheritance) are possible. Of cause, the methods inherited from `plotUtilsTimeDependent` can be overwritten again in `userClass`. 
Chapter 2

2 Examples

2.1 Geometries

OOPDE’s grid classes provides some methods to create, refine and show FEM grids. The format to store all informations related with the domain is an adaption of the points–edges–triangle (p-e-t) format used by MATLAB’s PDE toolbox. Every mesh is defined by four properties, p, e, t, and b. The property p is a matrix of size (dimension of the domain) \times (number of points) matrix, storing the coordinates of the grid points. The property e stores the relation between the indexes of the boundary points and boundary elements. Additional information depending on the dimension of the domain and hence the boundary can be added. For all dimensions the last row contains the number of boundary segment the boundary element is associated with. The property p contains the relation point-index–element. Also a subdomain number is stored in the last row.

OOPDE uses different methods to generate meshes:

(i) Direct definition by setting points, point–element and point–edge relations.

(ii) An interface to distmesh [PS04], an open source mesh generator. OOPDE provides an adapted version of the code.

(iii) It is also possible to use meshes in p-e-t format created by external programs as for instance Matlab’s PDE toolbox or COMSOL Multiphysics [Com].

Method (i) can be used for simple geometries as squares, L-shapes, triangles etc.

\[
p = \begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
e = \begin{bmatrix}
1 & 2 & 3 \\
2 & 3 & 1 \\
0 & 0 & 0 \\
1 & 1 & 1 \\
1 & 1 & 1
\end{bmatrix}
\]

\[
t = [1 \\
2 \\
3 \\
1]
\]

defines a triangle. Note that since refineMeshRGB method contains a bug, all geometries should contain at least two elements.
2 Examples

2.1.1 1D “meshes”

In one space dimension, grids are simply partitions of an interval, say $I = [a, b]$ by $x_1 = a, x_{n_x} = b$ and $x_{k+1} = x_k + \delta x_k$ for some $\delta x_k k = 1, ..., n_x - 1$. We will denote the $\delta x_k$ by $h$, the mesh width, a number that characterizes meshes not only in 1D: In 2D it is the inner circle of the triangle, in 3D the inner ball of e.g. a tetrahedral.

To create a grid we call the constructor grid1D, followed by a call of interval method. Its arguments are the left and right boundary of the interval or a vector of points and optionally, a value for minimal $\delta x_k$.

```plaintext
g = grid1D;
g.interval([0, pi], 0.5);
```

The object contains now a formal grid defined by the coordinates of the points $x_k$ stored in the property p, a property, that describes the element–point relation $t$ and a property, that stores the information on the edge points $e$.

```plaintext
>> g.p
ans =

0 3.1416 1.5708 0.7854 2.3562 0.3927 1.9635 1.1781 2.7489

>> g.t
ans =

1 3 4 5 6 7 8 9
6 7 8 9 4 5 3 2
1 1 1 1 1 1 1 1

>> g.e
ans =

1 2
0 0
0 0
1 2
```

Note that the vector g.p is not sorted. This rather complex structure (that is not really necessary in 1D) is chosen to have an unified structure for grid data in all dimension. Actually, the order of the points in p is determined by the used refinement method.

The order of the points in the “grid” is determined by the t property: The first “element” consists of the point one and six, the second of the points three and seven etc. The third row stores a subdomain number, in this case one. The property e has two columns. Every column is one boundary segment. In 1D, we always have two. The first row connects the first boundary segment with point one and the second with point two. These are the start and end point of our interval.

If no mesh-width is given, the mesh will consist of the points in the first argument. If this vector is of length two, only these points are included in the mesh.

It is possible to create a 1D mesh from a vector.
2.1 Geometries

In this example, the points are stored in increasing order in p (The `refineMesh` method will be not be called from `interval` in this case). An additional mesh width argument can change this, if the value of the meshwidth is smaller than the meshwidth given (implicitly) by the vector. In this case, the mesh will be refined until the actual mesh width is smaller than `hmax` given by the optional argument.

OOPDE provides methods to refine meshes. The call

```matlab
>> g.refineMesh
```
refines the mesh uniformly. The method has one optional argument, a list of elements to be refined.

```matlab
>> g.refineMesh(listOfElements)
```

This argument is useful for local mesh refinement.

2.1.2 2D meshes

Meshes for two dimensional domains use the same data structure as explained for `grid1D` objects. Since the relations between elements, edges, boundary segments, and point indexes are more complex, it is possible to construct such grids only for very simple domains, e.g. triangles or rectangles. Fortunately, `grid2D` provides methods to approximate almost all 2D domains by a `grid2D` object.

The constructor method has (as usual in OOPDE) no arguments. The call

```matlab
>> g = grid2D
```
creates an empty object. There are several methods defining “standard” domains.
In the following, we give some examples.
2.1 Geometries

**Unit square** Let $\Omega$ be the unit square.

```matlab
>> g = grid2D();
>> g.unitSquare(0.1);
```

We use the optional parameter $h_{\text{max}}$ to obtain a mesh with maximal mesh width 0.1.

**Square** Let $\Omega = (0, \pi) \times (0, 1)$. We use the standard mesh width. The arguments are $x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}$.

```matlab
>> g = grid2D;
>> g.square(0,pi,0,1);
>> g.plot;
```

![Square mesh](image)

**Circle** Let $\Omega$ be bounded by a circle with radius $R = 2$ and midpoint $(1, 1)$. We set $h_{\text{max}}$ to 0.1.

```matlab
>> g = grid2D;
>> g.circle(2,1,1,0.1)
```

![Circle mesh](image)

For a circle with center $(0, 0)$ we can use the call

```matlab
>> g.circle(2,0.1);
```
2 Examples

**Ellipse**  Let $\Omega$ be bounded by an ellipse with half-axes $A = 1.5$ and $B = 2$, with center $(4,2)$. We set $h_{\text{max}}$ to 0.2

```python
>> g.grid2D;
>> g.ellipse(1.5,2,4,2,0.2);
```

For an ellipse with center $(0,0)$ we can use the call

```python
>> g.ellipse(1.5,2,0.2);
```

**Double T**  The method doubleT meshes by default this double T shaped geometry:

```python
  g = grid2D()
g.doubleT()
```

Optional parameters are scale factors with respect to the x- and y-axis and $h_{\text{max}}$. Examples:

Standard shape with $h_{\text{max}}=0.1$

```python
  g.doubleT(0.1)
```
2.1 Geometries

Scaled by 0.5 in x-direction:
\[ g.\text{doubleT}(0.5,1) \]

Scaled by 0.5 in x-direction, by factor one in y-direction and hmax=0.1.
\[ g.\text{doubleT}(0.5,1,0.1) \]

**freeGeometry**  The method freeGeometry uses closed curves — approximated by polygons — to define domains via their boundary(s).

Let \( \Omega \) be given by \( \partial \Omega = \begin{bmatrix} \sin(s) + 0.2 \sin(2s) \\ -\cos(s) \end{bmatrix} \), where \( s \in [0,2\pi] \).
2 Examples

```matlab
s = 0:0.1:2*pi;
g = grid2D;
g.freeGeometry(@(s) [sin(s)+0.2*sin(2*s);-cos(s)]);
g.plot;
```

We obtain the following mesh:

![Mesh Diagram](image)

Obviously, near the boundary there are triangles with large angles. This influences the solution process negatively. The method

```matlab
getMeshQuality
```

measures the quality of the mesh. We define the quality of a mesh by the absolute value of the difference of the interior angles of each triangle from the value $\pi/3$. `getMeshQuality` returns for every triangle a value between zero (deformed triangle) and one (best quality). Useful is `getMeshQuality` in connection with the plot method:

```matlab
g.plot(g.getMeshQuality)
```

visualize the quality of the mesh.

![Quality Diagram](image)

We see some blue (bad) triangles at the boundary but also nearly optimal shaped triangles. The criss-cross pattern in the center of the domain is with its angles of $\pi/2$ and $\pi/4$ of good, but not optimal mesh quality.

We can try to improve the mesh quality by jiggling the mesh: The method `jiggleMesh` tries to move all interior mesh points such that the distance of each interior point to
2.1 Geometries

its neighbors becomes nearly equal. By using the option 'quality' with value 0.3, \texttt{jiggleMesh} will try to move points until every triangle has a quality value of 0.3 or better.

\begin{verbatim}
g.jiggleMesh('quality',0.3)
\end{verbatim}

The result is a mesh with an overall much better quality. Note that \texttt{jiggleMesh} stops after 15 jiggle procedures also if the requested quality is not reached. Note further that \texttt{jiggleMesh} can locally worsen the mesh quality.

A more complex example is the following.

\begin{verbatim}
k = 5; r=.3;
pde.grid.freeGeometry(...
  [r*(k-1)*cos(s)+r*cos((k-1)*s);...
   r*(k-1)*sin(s)-r*sin((k-1)*s)],...
  [[linspace(-0.3,0.25,10),0.3*ones(1,10),...
    linspace(0.3,-0.25,10),-0.3*ones(1,10)];
  [-0.3*ones(1,10),linspace(-0.3,0.25,10),...
    0.3*ones(1,10),linspace(0.3,-0.25,10)])
\end{verbatim}

It is possible to remove more than one part from the interior. We define a component part by a circle with holes by
2 Examples

\[ s = \text{linspace}(0,2\pi,141); \]
\[ s2 = \text{linspace}(0,2\pi,25); \]
\[ \text{pde.grid.freeGeometry}([[\sin(s); -\cos(s)],...\]
\[ [0.125*\sin(s2)+0.6;0.125*\cos(s2)],...,\]
\[ [0.125*\sin(s2)-0.6;0.125*\cos(s2)],...,\]
\[ [0.125*\sin(s2);0.125*\cos(s2)+0.6],...,\]
\[ [0.125*\sin(s2);0.125*\cos(s2)-0.6]); \]

The following mesh is the result.

The grid has five boundary segments: the “outer” boundary and four “inner” boundaries. A call of

\[ \text{g.identifyBoundarySegment} \]

shows the numbering:

**Mesh refinement**  Having called one of the geometry building methods, we can refine the mesh uniformly or locally.

\[ \gg g.\text{refineMesh}; \]

Note that the `refineMesh` method do not correct the approximation on the boundary. The geometry in the next example stays a polygon with 12 edges.
To obtain a better approximated circle geometry we should use `g.circle` with `hmax` option.

Sometimes we want to a-priory refine a mesh locally, e.g. if we know that there is a source concentrated in a point. In our next example, we refine a circle geometry two times around the center.

```matlab
>> g.circle(2,0.2);
>> indx = find(sqrt(sum(g.midpts.^2))<0.5);
>> g.refineMesh(indx);
>> indx = find(sqrt(sum(g.midpts.^2))<0.2);
>> g.refineMesh(indx);
>> g.plot
```

In some cases we want to rotate and move our grids. In the following example, we turn a square by \( \pi/4 \) anti-clockwise and move it one unit left in x-direction.

```matlab
>> g.square(0,pi,0,1)
>> g.turnMesh(pi/4);
```
2 Examples

```python
>> g.moveMesh(-1,0);
>> g.plot
```

2.1.3 Be save: write a classes for our geometry

Fix the shape of a domain

The main disadvantage of our approach—the shape of a domain is defined by a grid objects own method—is, that we can change the shape of the domain at run time.

```python
g = grid2D()
g.unitSquare()
g.unitCircle()
```

This behavior can be useful, but in most applications the form of the domain is fixed for all times. For that purpose we write a class that “removes” all shape defining methods. We demonstrate this by an example.

We want to discretize the following construction component:
2.1 Geometries

The following class part is derived from grid2d. In the constructor we first implement the polygon segments and then we use freeGeometry (the one from class part) to construct the “L shape with four holes” geometry. Next, we overwrite all methods that can change the shape of the domain, including freeGeometry. Since the constructor uses freeGeometry, we must ensure that it can be used only one time for the actual instance of the class object. See the code in lines 32–40. Methods like circle must be overwritten by a code that returns an “unknown method” exception. We implement a method that determines the caller method and returns an exception that returns the name of the caller method. See the “fancy” code in lines 70–78.

```matlab
classdef part < grid2D
    % A construction part
    methods (Access = public)
    function obj = part()
        N1 = 20; N2= 8; N3 = 12;
        N4 = 12; N5 = 8; N6 = 20;
        phi = linspace(0,2*pi,20);
        x1 = linspace(0,100,N1); x1(end) = []; %
        x2 = 100*ones(1,N2); x2(end) = [];
        x3 = linspace(100,40,N3); x3(end) = [];
        x4 = 40*ones(1,N4); x4(end) = [];
        x5 = linspace(40,0,N5); x5(end) = [];
        x6 = zeros(1,N6); x6(end) = [];
        y1 = zeros(1,N1); y1(end) = [];
        y2 = linspace(0,40,N2); y2(end) = [];
        y3 = 40*ones(1,N3); y3(end) = [];
        y4 = linspace(40,100,N4); y4(end) = [];
        y5 = 100*ones(1,N5); y5(end) = [];
```
y6 = linspace(100, 0, N6); y6(end) = [];

obj.freeGeometry([x1 x2 x3 x4 x5 x6;...
    y1 y2 y3 y4 y5 y6;...
    [8*sin(phi)+50;8*cos(phi)+20],...
    [8*sin(phi)+80;8*cos(phi)+20],...
    [8*sin(phi)+20;8*cos(phi)+50],...
    [8*sin(phi)+20;8*cos(phi)+80]);

% only one boundary segment
obj.e(5,:) = ones(size(obj.e(5,:)));
end

function freeGeometry(obj, varargin)
    % Check if already constructed
    if isempty(obj.p)
        freeGeometry@grid2D(obj, varargin{:});
    else
        % Do not change it...
        obj.notSuchAMethod.throwAsCaller;
    end
end

% Throw exceptions if one of these methods is called
function circle(obj)
    obj.notSuchAMethod.throwAsCaller;
end
function doubleT(obj)
    obj.notSuchAMethod.throwAsCaller;
end
function ellipse(obj)
    obj.notSuchAMethod.throwAsCaller;
end
function holeInPlane(obj)
    obj.notSuchAMethod.throwAsCaller;
end
function lshape(obj)
    obj.notSuchAMethod.throwAsCaller;
end
function square(obj)
    obj.notSuchAMethod.throwAsCaller;
end
function unitCircle(obj)
    obj.notSuchAMethod.throwAsCaller;
end
function unitSquare(obj)
    obj.notSuchAMethod.throwAsCaller;
end
% The exception is a method...
methods(Static,Access=private)

function ME = notSuchAMethod()
[Stack,~] = dbstack;
ME = MException('PART:NOSUCHMETHOD',
   ['There is no method with name ',...
   Stack(end).name(...
   strfind(Stack(end).name,'.')+1:end),...
   ' in class ',...
   Stack(end).name...
   (1:strfind(Stack(end).name,'.')-1),'.'])
end
end

Now we can work with the class but we cannot change the the shape of the geometry

>> g = part()
g =
Grid object with
955 mesh points
150 edges/lateral faces
1766 elements.
>> g.refineMesh
>> g.plot
>> g.jiggleMesh
>> g.jiggleMesh
>> g.plot Trial
>> g.unitCircle
There is no method with name unitCircle in class part.

Change the number of boundary segments

We creates an L-shaped domain, but with a special ordering of the boundary segments. The call

    g = grid2D()
g.lshape

creates a L-shaped domain with six boundary segments. If the user wants to have only one boundary condition, the call

    g.makeBoundaryMatrix(bc);

will define the given boundary condition on all six boundary segments. If the user needs two boundary conditions on two boundary segments, he can create his own L-shape class.

Note that by deriving this class from grid2D it will inherit also methods that change the shape of the domain, e.g unitSquare etc. Here we have to restrict the inheritance.
2 Examples

We assume, that we want to have two boundary segments and we want to accumulate the old boundary segments one, two, five and six into the new boundary segment one and to accumulate the old boundary segments three and four into the new boundary segment two.

We first derive LShape class from grid2D. Next, we call inside the constructor the inherited method obj.lshape. Correcting of the boundary segment number is done in the next five lines by indexing. We have now already a working class. However, to prevent the L-Shape for e.g. becoming a square by calling LShape.square, we overwrite all classes that changes the form of the geometry by empty methods. Declaring Hidden = true will hide them.

classdef LShape < grid2D
    methods(Access = public)
        function obj = LShape()
            obj.lshape;
            indx3 = obj.e(5,:) == 3;
            indx4 = obj.e(5,:) == 4;
            obj.e(5,:) = 1;
            obj.e(5,indx3|indx4) = 2;
        end
    end
    methods(Access = public, Hidden = true)
        function doubleT(obj)
            end
        function freeGeometry(obj)
            end
        function holeInPlane(obj)
            end
        function square(obj)
            end
        function unitCircle(obj)
            end
        function unitSquare(obj)
        end
    end
end

Result of identifyBoundarySegment
2.1.4 3D meshes

grid3Dpr—Prism elements

Often 3D objects are symmetric with respect to one or more axis. Such objects can be created by extruding a two dimensional domain into the third dimension.

As an example we consider the construction part from section 2.1.3. The following figure gives an impression.

To create a 3D geometry based on the extrusion of an 2D geometry, OOPDE offers prism elements. Having a 2D grid, the creation of a 3D grid is now a two liner:

```cpp
g = grid3Dpr();
g.extrude(part(), 0:1:10);
```

While the first argument must be a grid2D object, the second argument must be an interval.

We can plot the mesh by calling

```cpp
g.plot
```
2 Examples

or we can plot the outer faces of the mesh by calling

g.plotFaces
2.1 Geometries

Note that 3D meshes potentially contain a large number of elements and nodes:

\[
\text{g.nElements} \\
\text{ans} = \\
17660 \\
\text{g.nPoints} \\
\text{ans} = \\
10505
\]

There is also a refineMesh method.

\[
\text{g.refineMesh} \\
\text{refines the mesh uniformly. Note that one refinement increases the number of elements by the factor of eight.}
\]

\[
\text{g.nElements} \\
\text{ans} = \\
141280
\]

An adaptive refinement of single elements would lead to hanging nodes and is not implemented.

In general, using freeGeometry one can extrude every 2D geometry into 3D. See

where we used

\[
s = 0:0.1:2*\pi; \\
\text{base = grid2D; } \\
\text{base.freeGeometry([sin(s)+0.2*sin(2*s);-cos(s)]); } \\
g = \text{grid3Dpr(); } \\
g.extrude(base,[0:.2:10]);
\]

However, there are some shortcuts for standard geometries as e.g.
2 Examples

grid3Dpr.cylinder
gird3Dpr.unitCube
gird3Dpr.bar
gird3Dpr.rail

To look inside a 3D object, we can cut away some parts. For instance,

pde.grid.cutawayPlot([],20,[])

cuts through y=20 parallel to x-z axis.

grid3D—tetrahedral grids

Although tetrahedral grids are common in 3D discretizations, the creation of them is much harder than the creation of prism grids. However, it is possible to write also in the 3D tetrahedral case geometries from the scratch. An example of this approach is the Fichera cube.

```
g = grid3D
g.ficheraCube
```
However, since it complexity, we cannot suggest to create 3D object from scratch, see the approx. 130 lines of code of the method grid3D.fiercaCube.

A second method to create simple 3D geometry object is to use MATLAB’s delaunayTriangulation or the (obsolete) DelaunayTri class. An example using this approach is the grid3D.cylinder class.

A disadvantage of the use of delaunayTriangulation/DelaunayTri classes is, that they cannot handle non-convex domains in 3D.

A third approach to 3D geometries is the use of an adapted version of the code distmesh3D by Per-Olof Person and Gilbert Strang [PS04].

The code

```
g = grid3D()
g.unitBall(0.1)
```

creates a mesh approximating the unit ball with mesh width 0.1.
<table>
<thead>
<tr>
<th>method</th>
<th>Geometry</th>
<th>Argument(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid3D.unitCube</td>
<td>( \Omega = (0,1)^3 ).</td>
<td>[hmax]</td>
</tr>
<tr>
<td>grid3D.bar</td>
<td>( \Omega = (a,b) \times (c,d) \times (e,f) )</td>
<td>( a,b,c,d,e,f[hmax] )</td>
</tr>
<tr>
<td>grid3D.unitBall</td>
<td>( \Omega = { x \in \mathbb{R}^3</td>
<td>| x |_2 &lt; 1 } )</td>
</tr>
<tr>
<td>grid3D.ellipsoid</td>
<td>( \Omega = { x \in \mathbb{R}^3</td>
<td>\left( \frac{x_1}{A} \right)^2 + \left( \frac{x_2}{B} \right)^2 + \left( \frac{x_3}{C} \right)^2 &lt; 1 } )</td>
</tr>
<tr>
<td>grid2D.ficheraCube</td>
<td>( \Omega = (0,1)^3 \backslash [0.5,1]^3 )</td>
<td>[hmax]</td>
</tr>
<tr>
<td>grid2D.cylinder</td>
<td>( \Omega = { x \in \mathbb{R}^2</td>
<td>x_1^2 + x_2^2 &lt; R } \times (0,h) )</td>
</tr>
</tbody>
</table>
All visualization methods for grid3Dpr are also available in grid3D. The code

```cpp
grid3D g;
g.ellipsoid(1,1,2,0.1);
g.cutawayPlot([],0,[]);
```

produces the following figure:

2.2 Solving PDEs

2.2.1 A heat transfer problem

We solve the stationary heat transfer problem with constant source

\[-\nabla \cdot (0.1 \nabla u) + u = 10 \quad \text{in } \Omega \]
\[u = 0 \quad \text{on } \Gamma.\]

The domain \(\Omega\) may be the “unit” L-shaped domain, \((0,1)^2 \setminus [0.5,1]^2\), a subset of \(\mathbb{R}^2\). We compare the PDE with (1.2). The coefficient are \(c = 0.1\) and \(a = 1\) and the source \(f = 10\). We have further \(d = 0\), and \(\mathbf{b} = \mathbf{0}\) and \(\mathbf{q} = \mathbf{g} = 0\), but homogeneous Dirichlet boundary conditions, i.e. \(h = 1\) and \(r = 0\).

Although the source here is fixed and we have homogeneous Dirichlet boundary conditions, we will write a class that solves a wider range of problems, i.e. for all sources and coefficients \(c \in \mathbb{R}_+, a \in \mathbb{R}\), and for all kinds of boundary conditions, i.e. both, Dirichlet, Robin, and Neumann type boundary conditions. From Section 1.2 and especially (1.6), we know that we need the matrices \(K, M, Q\) and the vectors \(G\) and \(F\). To include the Dirichlet boundary conditions, we need additionally the matrix \(H\) and the vector \(R\). All these data will be stored in the properties of the pde object after calling the initialize method form the pde class. We do not need the matrices \(C\) and \(D\).
The idea is now to restrict the interface to parameters c, a and f. This required the redefinition of initialize, since we intentionally want to change its interface. The trick will be that one call in the overwritten initialize method the original version of initialize method inherited from pde. To save computational costs, we will pre-compute the linear system within initialize and use the property A to store the all-in-one systems matrix and b to store all contribution to the right-hand-side.

First we (re) define the user class heatTransfer.

```matlab
classdef heatTransfer < pde
    methods(Access = protected)
    function dy = df(obj,~,y)
        dy = obj.A*y + obj.b;
    end
    function J = jacobian(obj,~,~)
        J = obj.A;
    end
end
methods(Access = public)
    function initialize(obj,c,a,f)
        b = zeros(size(obj.grid.p,1),1);
        initialize@pde(obj,1,c,b,a,f);
        s = obj.fem.stiffSpring(obj.K+obj.M)
        obj.b = obj.F+s*(obj.H'*obj.R)+obj.G;
    end
end
```

We explain the code line by line

<table>
<thead>
<tr>
<th>Line</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Derive class heatTransfer from OOPDEs class pde</td>
</tr>
<tr>
<td>2</td>
<td>Declare a methods-block with Access=protected. This is a demand from pde class. A different declaration will give errors.</td>
</tr>
<tr>
<td>3</td>
<td>Define the method df. This method defines the PDE in means of vector and matrix operations. The interface must be (object,time,solution). Here, the time is not involved, so we can use the placeholder ~. We define the linear PDE problem abstract as Ay + b. The meaning of A and b will be given in initialize.</td>
</tr>
<tr>
<td>6–8</td>
<td>Overwrite the method jacobian. This is optional. If no jacobian method is defined here, the jacobian method of the parent class pde will be used. In this case, the Jacobian matrix will be computed numerically. It is always a good idea to provide the Jacobian matrix if it is known. Here, jacobian is simply the constant matrix obj.A.</td>
</tr>
<tr>
<td>10</td>
<td>Declare a public methods block</td>
</tr>
<tr>
<td>11</td>
<td>Overwrite the parents’ class initialize. To simplify the interface, we define it as initialize(obj,c,a,f) and drop the coefficient b, cf. the empty</td>
</tr>
</tbody>
</table>

\[1\] In MATLAB, one cannot change the Access attribute of an inherited class.
2.2 Solving PDEs

vector in the Listing above. (This heat transfer problem knows no convection term).

12 We do not have convection in our problem. By defining a vector \( \vec{0} \in \mathbb{R}^N \) we have for every dimension the correct dimension of \( b \).

The call `size(obj.grid.p,1)` determines the dimension of the spatial variable.

13 Assemble all matrices by calling the `initialize` method from `pde` class. This is an abstract call. Note the special syntax. The arguments are simply the coefficients \( d, c, b, a, f \) of the PDE

\[
\frac{d u}{d t} - \nabla (c \nabla u) + \vec{b} \nabla u + au = f
\]

where we set \( \vec{b} = \vec{0} \) in Line 12. Note in the call `initialize@pde` the different syntax: The argument `obj`, must be include in the parameter list like a common function argument.

14 Compute the stiff-spring coefficient \( s \) for the Dirichlet boundary conditions.

15 Define the property `obj.A`, the linear system matrix. We add simply stiff matrix `obj.K`, mass matrix `obj.M`, and the boundary condition matrices `obj.Q` and \( s*obj.H'*obj.H \).

16 Define the vector part of the linear system. The source term `obj.F`, the boundary condition vectors `obj.G`, and the stiff spring coefficient \( s \) appear here.

Note that this class works for every spatial dimension \( N = 1, ... 3 \). Note that the boundary conditions is in the class definition not known explicitly.

The driver code is a MATLAB function

```matlab
function testHeat2D()
pde = heatTransfer();
pde.grid = grid2D();
pde.fem = lagrange12D();
pde.grid.lshape;
pde.grid.refineUniformly(4);
pde.grid.makeBoundaryMatrix(pde.grid.dirichletBC('1','0'));
pde.initialize(0.1,1,10);
pde.solve('LINEAR');
pde.grid.plot(pde.y);
end
```

We comment Example 1 line by line

Line | Meaning
--- | ---
1 | Define the “main” function `testHeat2D`.

41
2 Examples

2 Declare the problem. Initialize a heatTransfer object named pde.
3 Declare pde.grid as 2-D grid by calling grid2D. grid2D is the triangle mesh class.
4 Declare the FEM. Here we need lagrange12D.
6 Call pde.grid.lshape method of class grid2D. It creates (without arguments) a mesh with only six elements.
7 Refine the mesh four times uniformly.
9 Create the boundary condition by a encapsulated call of makeBoundaryMatrix and dirichletBC. The arguments of dirichletBC are the coefficients h and r in
\[ hu = r \]
With one argument, makeBoundaryMatrix assumed that on every boundary segment (actually the L-shape geometry has six) holds the same boundary condition.
11 Call initialize with the coefficients c = 0.1, a = 1, and f = 10.
13 Call the linear solver.
15 Use method plot from class grid2D. The argument must be a vector of dimension nx x 1, here the solution stored in pde.y.

Solution of the heat transfer problem.

The single file version of our class consists on four files. We have two declaration blocks with together three methods, implemented in individual MATLAB files.

Content of the file heatTransfer.m. Note that we only declare the methods here, the implementation of all methods will be coded in separate files.

```matlab
1 classdef heatTransfer < pde
2     methods(Access = protected)
3         dy = df(obj,~,y)
4         J = jacobian(obj,~,~)
5     end
6     methods(Access = public)
```
2.2 Solving PDEs

initialize(obj,c,a,f)
end

Content of the file `df.m` for the `df` method.

```matlab
function dy = df(obj,~,y)
    dy = obj.A*y + obj.b;
end
```

Content of the file `jacobian.m` for the `jacobian` method.

```matlab
function J = jacobian(obj,~,~)
    J = obj.A;
end
```

The overwritten `initialize` method, stored in `initialize.m`.

```matlab
function initialize(obj,c,a,f)
    b = zeros(size(obj.grid.p,1),1);
    initialize@pde(obj,1,c,b,a,f);
    s = obj.fem.stiffSpring(obj.K+obj.C);
    obj.A = -(obj.K+s*(obj.H'*obj.H)+obj.Q);
    obj.b = s*obj.H'*obj.R+obj.G;
end
```

2.2.2 Burgers’ equation

We consider Burgers’ equation

\[
\begin{align*}
    u_t - cu_{xx} + \frac{1}{2}(u^2)_x &= 0 \\
    u &= 0 \\
    u(0) &= u_0
\end{align*}
\]

where \( \Omega \subset \mathbb{R} \).

The definition of the class `burgers` is rather simple. We overwrite `initialize` and define the non linearity as a method of the class `burgers`. Because of the homogeneous source and boundary conditions of Burgers’ equation in this example, we have no RHS vector at all. The only non-fix parameter is \( c \), hence we can implement a minimalistic interface for `initialize`. We set \( d = 1, b = 0.5, \) and \( a = f = 0 \)

```matlab
classdef burgers < pde & plotUtilsTimeDependent
    methods(Access=public)
    function initialize(obj,c)
        initialize@pde(obj,1,c,1,0,0);
        s = obj.fem.stiffSpring(obj.K+obj.C);
        obj.A = -(obj.K+s*(obj.H'*obj.H)+obj.Q);
        obj.b = s*obj.H'*obj.R+obj.G;
    end
end
```

43
2 Examples

```matlab
methods(Access=protected)
    function dy = df(obj,~,y)
        dy = obj.A*y+obj.nonLin(y)+obj.b;
    end
end

methods(Access=private)
    function y = nonLin(obj,y)
        y = -0.5*obj.C*(y.*y);
    end
end

We analyze the code line by line

<table>
<thead>
<tr>
<th>Line</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Derive class burgers from pde and plotUtilsTimeDependent parent classes. plotUtilsTimeDependent adds some additional visualization methods</td>
</tr>
<tr>
<td>2</td>
<td>Declare a public methods block.</td>
</tr>
<tr>
<td>3</td>
<td>Overwrite initialize method form pde.</td>
</tr>
<tr>
<td>4</td>
<td>Compute all matrices by calling pde.initialize with the argument list of the overwritten method.</td>
</tr>
<tr>
<td>5</td>
<td>Compute a stiff-spring coefficient s</td>
</tr>
<tr>
<td>6+7</td>
<td>Define properties obj.A and obj.b</td>
</tr>
<tr>
<td>11</td>
<td>Declare a protected methods block</td>
</tr>
<tr>
<td>12-15</td>
<td>Overwrite method df by an abstract definition as linear part plus non-linear part.</td>
</tr>
<tr>
<td>17</td>
<td>Declare a private methods block.</td>
</tr>
<tr>
<td>18-22</td>
<td>Define the method nonLin. It computes the non linear part $\frac{1}{2}(u^2)_x$.</td>
</tr>
</tbody>
</table>

Note, that we do not overwrite jacobian. OOPDE’s solver use the jacobian method from pde class.

We want now solve a well know benchmark example. Let $\Omega = (0, \pi)$, $u(0) = \sin(x)$ and $c = 0.01$. Let the time interval be $[0, 5]$. In the following, we give the main program as a MATLAB function.

```matlab
function testBurgersClass
    problem = burgers();
    problem.fem = lagrange11D;
    problem.grid = grid1D();
    problem.grid.interval([0,pi/2-0.01 pi/2+0.01 pi]);
    problem.grid.refineUniformly(7)
end
```
2.2 Solving PDEs

```matlab
problem.grid.makeBoundaryMatrix(problem.grid.dirichletBC('1','0.0'));

problem.y = y0(problem.grid.p);
problem.initialize(1e-4);
problem.time = linspace(0,10,1000);

problem.solve('EULERI');

figure(1)
problem.animation

function val = y0(x)
    val = sin(2*x);
end
```

We comment the code line by line

<table>
<thead>
<tr>
<th>Line</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define the main function testBurgersClass.</td>
</tr>
<tr>
<td>2</td>
<td>Declare <code>pde</code> as object of <code>burgers</code> class.</td>
</tr>
<tr>
<td>3</td>
<td>Declare <code>pde.fem</code> as <code>lagrange11D</code> object. This is the class that implements 1D first order Lagrange elements.</td>
</tr>
<tr>
<td>4</td>
<td>Declare <code>pde.grid</code>.</td>
</tr>
<tr>
<td>5</td>
<td>Define an interval ((0, \pi)) with maximal mesh width (h &lt; 0.01).</td>
</tr>
<tr>
<td>8</td>
<td>As in the Example 1, we define homogenous Dirichlet boundary conditions.</td>
</tr>
<tr>
<td>10</td>
<td>We initialize (y(0) = \sin(2\pi x)). <code>grid.p</code> contains the coordinates of the mesh points.</td>
</tr>
<tr>
<td>11</td>
<td>We call the overwritten <code>initialize</code> with (c = 0.0001).</td>
</tr>
<tr>
<td>12</td>
<td>Define the time to solve by calling MATLAB's <code>linspace</code> function.</td>
</tr>
<tr>
<td>14</td>
<td>Solve the problem by using <code>pde</code> class' <code>solve</code> with option <code>EULERI</code> which calls the implicit Euler solver.</td>
</tr>
<tr>
<td>16–17</td>
<td>We visualize the solution by calling the <code>burgers.animation</code> method.</td>
</tr>
<tr>
<td>22–24</td>
<td>Define a local function to compute the initial value for Burgers' equation.</td>
</tr>
</tbody>
</table>
2 Examples

Solution of Burgers’ equation with $c = 10^{-4}$

2.2.3 A parabolic PDE with time dependent data

This example shows how to incorporate time dependent boundary conditions and time and space dependent sources.

We choose $\Omega = (0, 2\pi)$, and our PDE should be

$$\frac{d}{dt} u - cu_{xx} + bu_x + au = f(t, x)$$

$$u_x(t, 0) = 0$$

$$u(t, 2\pi) = r(t)$$

$$u(0, x) = u_0$$

The difficulty here is to write a code that handles the time dependent sources efficiently. Obviously, in this example only the RHS of the PDE and the RHS of the Dirichlet boundary condition dependent on time. Hence, we must update the vectors $F$ and $R$ in every time step. This may lead to a code where assemble methods were called within the $df$ method. This is the correct, but not really efficient way to obtain the update of the vectors.

Since the vector $R$ is for $n = 1$ not from an integral, we can write it as $R(t) = (H^T H) \ast r(t)$, where $r(t)$ must be a vector of the length of the number of grid points. The vector $F$ contains values of an integral but we can compute it approximately by $F \approx M \ast f(t, x)$. From that we can prepare the matrices $H^T H$ and $M$ in the offline phase of our program, i.e. in the initialize method and only the rather inexpensive matrix times vector operation must be performed online. We introduce four new properties: $HH$ and $MM$ to store the matrices needed for fast RHS update and $r$ and $f$ to store function handles that implements the time dependent functions $r(t)$ and $f(t)$. Note that $\text{obj}.r(t, \text{obj.grid.p(1,:))}$ and $\text{obj}.f(t, \text{obj.grid.p(1,:))}$ evaluate functions where the time is determined in general by the ODE solver.

The code of the class definition is surprisingly short. In initialize, we only add the definition of $HH$ and $MM$. Note that the time dependent Dirichlet part will be added in $df$. We
2.2 Solving PDEs

```matlab
classdef convectionDiffusionTimedepBC < pde & plotUtilsTimeDependent
    properties (Access=private)
        HH
        MM
    end
    properties (Access=public)
        r
        f
    end
    methods (Access = protected)
        function dy = df(obj,t,y)
            dy = obj.A*y + obj.b +...
            obj.HH*obj.r(t,obj.grid.p(1,:))...
            + obj.MM*obj.f(t,obj.grid.p(1,:));
        end
        function J = jacobian(obj,~,~)
            J = obj.A;
        end
    end
    methods (Access = public)
        function initialize(obj,d,c,b,a,f)
            initialize@pde(obj,d,c,b,a,f);
            s = obj.fem.stiffSpring(obj.K+obj.M+obj.C);
                Q);
            obj.b = obj.G + obj.F;
            obj.MM = obj.mass;
            obj.HH = s*obj.H'*obj.H;
        end
    end

Important note
In two or three dimensional domains the dependence between spatial variable and
time may be more complex. This offline computing technique works only if we have
\( f(t,x) = f_1(t) \cdot f_2(x) \) and \( r(t,x) = r_1(t) \cdot r_2(x) \). If we cannot split \( f \) and \( r \) in this way,
we must assemble the RHS vectors within the df method.

We test the code by the following example.

\[
\frac{d}{dt} u - cu_{xx} + bu_x = f(t,x) \\
u_x(t,0) = 0 \\
u(t,2\pi) = r(t) \\
u(0,x) = u_0
\]
2 Examples

with

\[ f(t, x) = 2e^{-t}\sin(x) \]
\[ u_0(x) = \begin{cases} 
\sin(t) & x < \pi \\
0 & \text{otherwise}
\end{cases} \]
\[ r(t) = \sin(t) \]

and \( c = 1, b = -1 \). The time interval let be \([0, 30]\).

The driver program is given as follows:

```matlab
function testParabolicTimedependentBCs
    pde = convectionDiffusionTimedepBC();
    pde.grid = grid1D();
    pde.grid.interval([0,2*pi],0.05);
    pde.fem = lagrange11D;
    pde.grid.makeBoundaryMatrix(...
        pde.grid.neumannBC('0'),...
        pde.grid.dirichletBC('1'));
    pde.initialize(1,1,-1,0,0);
    pde.r = @r; pde.f = @f;
    pde.time = [0:0.5:30];
    pde.y = y0(pde.grid.p(1,:));
    pde.solve('ODE15S')
    figure(1);
    pde.plotTimeSpace
end

% local functions
function y = r(t,x)
    y = sin(t)*ones(size(x));
    y = y(:);
end
function y = f(t,x)
    y = 2*exp(-t)*sin(x);
    y = y(:);
end
function y = y0(x)
    y = zeros(size(x));
    y(x<=pi) = sin(x(x<=pi));
    y = y(:);
end
```

48
2.2 Solving PDEs

Solution of the transient PDE problem with time dependent sources.

2.2.4 The use of P2 elements

In future releases, OOPDE will give full support for higher order finite elements. However, P2 elements in 1D and 2D —here called lagrange21D and lagrange22D—are already implemented.

The main difficulty of implementing higher order finite elements is the question how to manage the handling of the additional nodes in the mesh. To define a complete polynomial of degree \( n = 2 \) we need in 1D three, in 2D six and in 3D nine coefficients, i.e. every element must define this number of nodes. In 1D, one additional node is added in the center of every sub interval. These nodes we added at the end of the points list grid1D.p. In the elements list grid1D.t we move \( t(2,:) \) to \( t(3,:) \) and add the new points by writing them into grid1D.t(2,:). We also need some information on the original (non-) extended mesh, e.g. the number of its nodes. Also the property grid1D.pointsInElement is now set to three. This all will be performed when the new class method grid1D.extendMesh is called.

We want to solve the problem

\[
-\alpha \frac{\partial^2 u}{\partial x^2} + \beta \frac{\partial u}{\partial x} + \gamma u = f \quad x \in (0, \pi) \\
u = 0 \quad x = \{0, \pi\}
\]

with \( \alpha = 0.1, \beta = 10, \gamma = 1, \) and \( f = \begin{cases} 
10 & \text{if } \frac{\pi}{4} < x < \frac{3}{4} \pi \\
0 & \text{otherwise}
\end{cases} \).

We use an adaption of our heatTransfer class to implements the problem.

```matlab
classdef convectionDiffusion < pde
    methods(Access = protected)
    function dy = df(obj,~,y)
        dy = obj.A*y + obj.b;
    end

    function J = jacobian(obj,~,~)
        J = obj.A;
    end
end
```
2 Examples

methods(Access = public)

function initialize(obj,c,b,a,f)
    initialize@pde(obj,1,c,b,a,f);
    s = obj.fem.stiffSpring(obj.K+obj.M+obj.C);
    Q);
    obj.b = obj.F+s*(obj.H'*obj.R)+obj.G;
end
end

end

Note that we not need to write any element dependent code in the class definition of
convectionDiffusion class at all. Only the main program has to change in a few lines.

Line 4     Extend the mesh.
Line 5       Only for convenience we write the mesh-points of the extended mesh into a
             variable x.
Line 6     Set the fem property to lagrange21D

It its crucial to obtain correct results to call extendMesh method after calling interval or
a 2D geometry building methods like circle, unitSquare etc.

In the following we present the “main program”.

1 function testConvectionDiffusionL21D
2     g = grid1D;
3     g.interval([0,pi],2^(-6));
4     g.extendMesh;
5     x = g.p;
6     fem = lagrange21D;
7     g.makeBoundaryMatrix(g.dirichletBC('0'));
8     pde = convectionDiffusion();
9     pde.fem = fem;
10    pde.grid = g;
11    pde.initialize(.1,10,1,f(x));
12    pde.solve('LINEARGAUSS')
13    pde.grid.plot(pde.y)
14 end

16 function y = f(x)
17     y = zeros(size(x));
18     y(x<pi*3/4&x>pi/4) = 10;
19     y = y(:);
20 end

This works also in 2D:

1 function testHeatL22D
2     s = 0:0.1:2*pi;
3     g = grid2D;
4     g.freeGeometry([2*sin(s)+.5*sin(2*s);-2*cos(s)]) ;
2.2 Solving PDEs

```matlab
5  g.extendMesh;
6  pde = heatTransfer;
7  pde.grid = g;
8  pde.fem = lagrange22D;
9  pde.grid.makeBoundaryMatrix(g.dirichletBC('1', '0'));
10  x1 = pde.grid.p(1,:);
11  x2 = pde.grid.p(2,:);
12  pde.initialize('1','0',f(x1,x2));
13  pde.solve('LINEARGAUSS')
14  pde.grid.plot(pde.y);
15  end
16  function val = f(x1,x2)
17      val = 20*((x1.^2-x1)+(x2.^2-x2))+10*x1.*(1-x1).*x2.*(1-x2);
18  end
```

However, we have in line 13 a hidden issue using higher order finite elements: The discretization error is determined by the projection of the solution into the finite elements space. To obtain the optimal order of this projection, the solution of the PDE has to be sufficiently smooth. In the case of Lagrange-2 elements it must be from $H^2(\Omega)$. On the other hand, this is only possible if the source is also smoother than $L^2(\Omega)$. However, OOPDE assumes that the source term is from $L^2(\Omega)$. Such functions can be approximated by piecewise constant functions. That is the reason why in OOPDE codes the source can be given as a vector of length nElements, i.e. the source $f$ evaluated in the center of each element. $H^1$ function can be approximated also by piecewise constant functions, but not in the optimal order. To overcome this issue, we must consider $H^1$ source function also at least by piecewise linear functions, i.e. the source function must be evaluated in the nodes of the grid instead of the centers. To distinguish between $L^2$ and $H^1$ sources we interpret (in the context with Lagrange-2 elements) a vector of length nElements (and also if the source is given by a function handle) as the coefficients of the approximation of an $L^2$ function by piecewise constant functions and vector of length nPoints as the coefficients of the approximation of an $H^1$ function by the piecewise polynomials of second order.

In our example, using the call
```
5  pde.initialize('1','0',f(x1,x2));
```
the solution converges with order three, but using
```
5  pde.initialize('1','0',@f);
```
the solution converges with order two, where we assume that the source is from $H^1(\Omega)$.

2.2.5 Dirichlet boundary conditions

Dirichlet boundary conditions cannot be handled directly with the finite element method. The next example shows the implementation of the Lagrange multiplier approach for including Dirichlet boundary conditions from Section.
2 Examples

classdef heatTransferD < pde & plotUtilsTimeDependent
    properties(Access = public)
        % Multiplier for the Dirichlet BCs
        multiplier
    end

    methods(Access = protected)
        function dy = df(obj,~,y)
            dy = obj.A*y + obj.b ;
        end

        function J = jacobian(obj,~,~)
            J = obj.A;
        end
    end

    methods(Access = public)
        function initialize(obj,d,c,b,a,f)
            initialize@pde(obj,d,c,b,a,f);
            N = sparse(size(obj.H,1),size(obj.H,1));
                     -obj.H       N];

            obj.b = [obj.F + obj.G
                     obj.R];
        end

        function solve(obj,args)
            % solve method, overwrites pde.solve. We must handle the
            % multiplier. The matrix D must be overwritten, now
            % the system is
            % a DAE.

            if isempty(obj.y)
                obj.y = [zeros(obj.grid.nPoints,1)
                         zeros(size(obj.R))];
            end
            % Only to make the notation shorter...
            N11 = sparse(size(obj.H,1),size(obj.H,1));
            N12 = sparse(size(obj.H,1),size(obj.H,2));
            N21 = sparse(size(obj.H,2),size(obj.H,1));

            obj.D = [obj.D N21
                     N12       N11];

            solve@pde(obj,args);
            obj.multiplier = obj.y(1+obj.grid.nPoints:end,:);
            obj.y = obj.y(1:obj.grid.nPoints,:);
2.3 Systems of PDEs

2.3.1 An optimal control problem without inequality constraints

Our first example is the optimal control problem

$$\min \frac{1}{2} \| y - y_d \|_{L^2(\Omega)}^2 + \frac{\kappa}{2} \| u \|_{L^2(\Omega)}^2$$

subject to

$$-\Delta y + y = u \text{ in } \Omega$$
$$y = 0 \text{ on } \Gamma$$

where $y_d$ is the desired state. In optimal control, the function $u$ is called the control. This problem has a unique solution $\bar{u}$. Its fulfills together with the associated state $\bar{y}$ and the so-called adjoint state $\bar{p}$ the system of coupled linear PDEs

$$-\Delta \bar{y} + \bar{y} - \bar{u} = 0 \text{ in } \Omega$$
$$\bar{y} = 0 \text{ on } \Gamma$$
$$\kappa \bar{u} + \bar{p} = 0 \text{ in } \Omega$$
$$-\Delta \bar{p} + \bar{p} - \bar{y} = -y_d \text{ in } \Omega$$
$$\bar{p} = 0 \text{ on } \Gamma$$

These are the Karush-Kuhn-Tucker (KKT) conditions.

We define a class kkt for defining this system. Again, we derive our class from pde.

The linear system will be stored in the property obj.A. The method kkt.df will be the same as for the scalar linear PDE 4.1.1

The definition of the system will be done in kkt.initialize. The state equation $y$ as well as the adjoint equation for $p$ are up to the source term identical. Hence we need to compute the matrices $K$ and $M$ only one time. Since we fix our problem to homogeneous Dirichlet boundary conditions, we can implement the boundary conditions directly in initialize. The linear system is now a one-to-one implementation of the KKT system. Note that we need here some zero matrices to fill the matrix A with zero blocks and the RHS with zero vectors, respectively. Note further that since we do not use here initialize@pde, we need to set obj.initialized = true manually.
To test our class we use the following small program. To obtain a desired function \( y_d \), we solve first a PDE problem with known source, here the Dirac measure \( u = \delta_{[0,0]} \). The solution of our optimal control problem should converge for \( \lambda \to 0 \) towards the given source in \( L^2 \). We cannot expect that we reach the desired state for \( \lambda > 0 \). That is the price we pay for regularization.
2.3 Systems of PDEs

```matlab
pde.grid.unitCircle
pde.grid.refineUniformly(4);
for k = 1:5
    pde.grid.refineMesh(pde.grid.pointToElementIndex([0 0]));
end

pde.grid.makeBoundaryMatrix(pde.grid.dirichletBC('0'));
pde.initialize(1,0,f(pde));
pde.solve('LINEAR');

kktSystem = kkt();
kktSystem.grid = pde.grid;
kktSystem.fem = pde.fem;

kktSystem.bcState = kktSystem.grid.neumannBC('0.0');

lambda = 1e-6;
kktSystem.initialize(lambda,pde.y);
kktSystem.solve('LINEARGAUSS');

figure(1); clf;
kktSystem.plot

subplot(2,2,4)
pde.grid.plot(pde.y);title('Desired state y_d');view(2)
```

```matlab
function y = f(p)
    y = zeros(size(p.grid.p(1,:)));
    ndx = p.grid.pointToElementIndex([0,0]);
    indx = unique(p.grid.t(1:3,ndx));
    [~,indxindx] = min(sum((p.grid.p(:,indx)-0).^2));
    y(indx(indxindx)) = 1e6;
    y = y/(ones(size(y))*p.mass*abs(y));
end
```

2 Examples

Solutions of the test case for the kkt class.

2.3.2 Reacting Flow—A system of PDEs with non linear coupling

We take this example from the paper MARCELO BUFFONI and KAREN WILLCOX, Projection-based model reduction for reacting flows. [BW10].

The flow field is given as constant. The domain is given by $\Omega = [0, 0.9] \times [0, 1.8]$ cm. Let be given the chemical reaction

$$v_{\text{fuel}} Y_{\text{fuel}} + v_{\text{oxid}} Y_{\text{oxid}} \rightarrow v_{\text{prod}} Y_{\text{prod}}$$

where $Y_{\text{fuel}}, Y_{\text{oxid}},$ and $Y_{\text{prod}}$ are the mass fractions of the fuel, the oxidizer and the product respectively, and $v_{\text{fuel}}, v_{\text{oxid}},$ and $v_{\text{fuel}}$ are their related stoichiometric coefficients. The nonlinear reaction source term is of Arrhenius type, i.e. we have

$$\omega_i = -v_i \left( \frac{(w_i)_{\text{c}}}{{\rho}} \right) \left( \frac{\rho Y_{\text{fuel}}}{{w_{\text{fuel}}}} \right)^{v_{\text{fuel}}} \left( \frac{\rho Y_{\text{oxid}}}{{w_{\text{oxid}}}} \right)^{v_{\text{oxid}}} Ae^{-\frac{E}{RT}} , \quad i = \text{fuel, oxid, prod}.$$ 

Here $A$ is the pre-exponential factor, $w_i$ is the molecular weight for $i = \text{fuel, oxid, prod}$, $E$ is the activation energy and $R$ the universal gas constant, $T$ the temperature. For the temperature we have the source

$$w_T = w_{\text{prod}} Q,$$

where $Q$ is the reaction heat.
2.3 Systems of PDEs

We consider a system of three chemical species and temperature. The system looks like

\[
\frac{\partial}{\partial t} Y_{\text{fuel}} = -c \Delta Y_{\text{fuel}} + \vec{b} \nabla Y_{\text{fuel}} + \omega_{\text{fuel}}(Y_{\text{fuel}}, Y_{\text{oxid}}, T) \quad \text{in } \Omega
\]

\[
\frac{\partial}{\partial t} Y_{\text{oxid}} = -c \Delta Y_{\text{oxid}} + \vec{b} \nabla Y_{\text{oxid}} + \omega_{\text{oxid}}(Y_{\text{fuel}}, Y_{\text{oxid}}, T) \quad \text{in } \Omega
\]

\[
\frac{\partial}{\partial t} Y_{\text{prod}} = -c \Delta Y_{\text{prod}} + \vec{b} \nabla Y_{\text{prod}} + \omega_{\text{prod}}(Y_{\text{fuel}}, Y_{\text{oxid}}, T) \quad \text{in } \Omega
\]

\[
\frac{\partial}{\partial t} T = -c \Delta T + \vec{b} \nabla T + \rho \frac{\partial}{\partial t} Y_{\text{prod}} \quad \text{in } \Omega
\]

\[
\vec{n} c \nabla Y_{\text{fuel,oxid,prod}} = 0 \quad \text{on } \Gamma_{\text{free}}
\]

\[
\vec{n} c \nabla T = 0 \quad \text{on } \Gamma_{\text{free}}
\]

\[
Y_{\text{fuel,oxid}} = Y_{\text{fuel,oxid}}^{\text{in}} \quad \text{in } \Gamma_{\text{inlet}}
\]

\[
Y_{\text{prod}} = 0 \quad \text{on } \Gamma_{\text{inlet}}
\]

\[
T = T^{\text{in}} \quad \text{on } \Gamma_{\text{inlet}}
\]

\[
Y_{\text{fuel,oxid,prod}}(t_0) = Y_{\text{fuel,oxid,prod}}^{0} \quad \text{in } \Omega
\]

This will be the first example with a complex class definition file. However, the methods `df` and `jacobian` are standard except the use of `obj.D` as mass to integrate the chemical source term `obj.w`. The chemical source term implements here the (global) reaction $2H_2 + O_2 \rightarrow 2H_2O$. For that reaction, we have $v_{\text{fuel}} = 2$, $v_{\text{oxid}} = 1$, $v_{\text{prod}} = -2$, $A = 5.5 \cdot 10^{11}$, $E = 4500$, $w_{\text{fuel}} = 2.016$, $w_{\text{oxid}} = 31.9$ and $w_{\text{prod}} = 18.0 \text{ gr/mol}$. The gas constant is $R = 8.314472 \text{ J/mol/K}$, the density $\rho = 1.39 \cdot 10^{-3} \text{ gr/cm}^3$ and the heat $Q = 9800 \text{ K}$. The implementation of the Arrhenius equation is done nearly one-to-one in the method `w`.

Most of the code is needed by the definition of the chemical source term and the definition of the boundary conditions, since every species needs its own. It is easy to see that we need for the fuel, oxidizer and temperature an intake while all other boundary conditions are homogenous Neumann.

The method `initialize` solves the linear part of the system i.e. convection and diffusion to obtain a good choice for an initial guess for the non linear solver.

```matlab
1 classdef reactingFlow < pde
2     methods(Access = protected)
3     function val = df(obj,~,y)
4         val = obj.A*y+obj.D*obj.w(0,y) + obj.b;
5     end
6
7     function J = jacobian(obj,~,y)
8         J = obj.A+obj.D*obj.dw(0,y);
9     end
10
11     methods(Access = public)
12     function initialize(obj,c,b)
13         bcTempInlet = obj.grid.boundaryCondition(...
14             [],[],'1',min(950,max(300,sin(3*pi.*s-pi)*3500)));
15         bcTempOverall = obj.grid.boundaryCondition([],[],
16             '0','0');
17     end
```

57
bcFuel = obj.grid.boundaryCondition(...
    [], [], '1', 'max(0, sin(3*pi.*s-pi)*0.2)');
bc0x = obj.grid.boundaryCondition(...
    [], [], '1', 'max(0, sin(3*pi.*s-pi)*0.02)');
bcProd = obj.grid.boundaryCondition([], [], '1', '0');
bcNeuman = obj.grid.boundaryCondition('0', '0');

obj.grid.makeBoundaryMatrix(bcNeuman, bcNeuman, bcNeuman, bcFuel);
[-, -, H, RF] = obj.fem.assemb(obj.grid);

obj.grid.makeBoundaryMatrix(bcNeuman, bcNeuman, bcNeuman, bc0x);
[-, -, -, RO] = obj.fem.assemb(obj.grid);
obj.grid.makeBoundaryMatrix(bcNeuman, bcNeuman, bcNeuman, bcProd);
[-, -, -, RP] = obj.fem.assemb(obj.grid);

obj.grid.makeBoundaryMatrix(bcTempOverall, bcTempOverall, ...
    bcTempOverall, bcTempInlet);
[-, -, -, RT] = obj.fem.assemb(obj.grid);

[K, -, -] = obj.fem.assema(obj.grid, c, 0, 0);
C = obj.fem.convection(obj.grid, b);

M = obj.mass();
N = sparse(obj.grid.nPoints, obj.grid.nPoints);

obj.A = -[K+C+1e8*(H'*H) N N N;
    N K+C+1e8*(H'*H) N N;
    N N K+C+1e8*(H'*H) N;
    N N N K+C+1e8*(H'*H)];

obj.D = [ M N N N;
    N M N N;
    N N M N;
    N N N M];

obj.b = [1e8*H'*RT;
    1e8*H'*RF;
    1e8*H'*RO;
    1e8*H'*RP];

obj.initialized = true;
end

function d = w(obj, -, Y)
A = 5.5e11;
E = 4.5e3;
nu = [2 1 -2]';
wc = [2.016 31.9 18.0]';
R = 8.314472;
rho = 1.39e-3;
Q = 9800;

np = obj.grid.nPoints;

scalar_rate =
    (rho*Y(np+1:2*np)/wc(1)).^nu(1).*
    (rho*Y(2*np+1:3*np)/wc(2)).^nu(2).*
    .*A.*exp(-E./(R.*Y(1:np)));

dY = -1/rho*[wc(1)*nu(1)*scalar_rate;
    wc(2)*nu(2)*scalar_rate;
    wc(3)*nu(3)*scalar_rate];

dT = dY(2*np+1:3*np)*Q;
d = [dT;dY];

function plot(obj)
    for k = 1:4
        subplot(2,2,k)
        obj.grid.plot(obj.y((k-1)*np:n*obj.grid.nPoints+1:k*obj.grid.nPoints+1:end));
        view(0,90)
    end
end

function initialGuess(obj)
    obj.y = -obj.A\obj.b;
end

methods(Access = private)
    function d = dw(obj, Y)
        A = 5.5e11;
        E = 4.5e3;
        nu = [2 1 -2]';
        wc = [2.016 31.9 18.0]';
        R = 8.314472;
        rho = 1.39e-3;
        Q = 9800;
        np = obj.grid.nPoints;
        d_rate = [...
Again, the main program is now rather short.

```matlab
function testBuffoniWillcox
    pde = reactingFlow();
    pde.fem = lagrange12D();
    pde.grid = grid2D();
    pde.grid.square(0,1.8,0,0.9,1);
    pde.grid.refineUniformly(7);
    pde.initialize(2,[50,0]');
    pde.initialGuess;
    pde.solve('STATIONARY');
    pde.plot
end
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define a function testBuffoniWillcox</td>
</tr>
<tr>
<td>2</td>
<td>Declare pde as reactingFlow object</td>
</tr>
<tr>
<td>3</td>
<td>Declare pde.fem as lagrange12D object.</td>
</tr>
<tr>
<td>4</td>
<td>Declare pde.grid as grid2D object.</td>
</tr>
<tr>
<td>5</td>
<td>Create a square ((0,1.8) \times (0,0.9)) with initial mesh size (h = 1). We obtain a structured grid.</td>
</tr>
<tr>
<td>6</td>
<td>Call the method refineUniformly of grid2D class. The parameter is the number of iterations.</td>
</tr>
<tr>
<td>7</td>
<td>We call initialize from reactingFlow class with parameter (c=1) and (\vec{b} = [50,0]'). This is the constant flow field.</td>
</tr>
</tbody>
</table>
2.3 Systems of PDEs

Call the initialGuess method from reactingFlow class. It computes a solution to the non-reacting flow as a initial guess for the non linear solver.

Call solve method from pde class with option “STATIONARY”. It uses a newton type solver with an “adaptive” update for the Jacobian matrix.

Plot the solution with plot method from reactingFlow class.

2.3.3 KKT system for a problem with control constraints

As mentioned before, readers not familiar with optimal PDE control should see this as an example for a system of coupled nonlinear PDEs. However, we give a short motivation why to consider this problem.

Let be given this optimal control problem:

\[
\min \frac{1}{2} ||y - y_d||^2 + \frac{\lambda}{2} ||u||^2
\]

subject to

\[-c \Delta y + ay = u \text{ in } \Omega \]

\[hy = r \text{ on } \Gamma\]

and to the control constraints

\[u \leq \bar{u} \leq \bar{u} \text{ in } \Omega.\]

The function \(u \in L^2(\Omega)\) is the control and the function \(y \in C(\Omega)\) is the state. This is an optimization problem with equality (the PDE) and inequality constraints (the control


2 Examples

constraints). We eliminate the equality constraints by adding the inner product of the constraints with some function \( p \in L^2(\Omega) \) to the objective function. The control constraints will be eliminated by inserting it in a barrier function and add this also to the objective function. Altogether, this is called Lagrange function. It reads in our case

\[
L(y, u, p) = \frac{1}{2}(y - y_d, y - y_d)_{L^2(\Omega)} + \frac{\lambda}{2}(u, u)_{L^2(\Omega)} - (-c \Delta y + ay - u, p)_{L^2(\Omega)} - (y, p)_{L^2(\Gamma)} - \sigma \int_{\Omega} \ln(u - \bar{u}) - \ln(\bar{u} - u) \, dx
\]

where \( \sigma > 0 \) is a parameter. From Karush-Kuhn-Tucker theory we know, that the set of functions \((\bar{y}, \bar{u}, \bar{p})\) is the (in this case unique) solution of the optimal control problem, if and only if it is a solution of the system

\[
\begin{align*}
\Delta \bar{p} - ap + \bar{y} - y_d &= 0 \quad \text{in } \Omega \\
\bar{h}p &= 0 \quad \text{on } \Gamma \\
\lambda \bar{u} + \bar{p} - \frac{\sigma}{\bar{u} - \bar{u}} + \frac{\sigma}{\bar{u} - \bar{u}} &= 0 \quad \text{in } \Omega \\
\Delta \bar{y} - a \bar{y} + \bar{a} &= 0 \quad \text{in } \Omega \\
h \bar{y} &= r \quad \text{on } \Gamma
\end{align*}
\]

(2.1)

Obviously, we have only to solve this nonlinear system. We should discus briefly the role of the parameter \( \sigma \). Obviously rather large values smooths the nonlinear term in our KKT system, small values allow the control to comes near the constraints. We should apply an algorithms that starts with rather large \( \sigma_0 \) and reduces it within a loop, e.g. \( \sigma_{k+1} = s \sigma_k \) with a number \( s \in (0, 1) \). The iteres depending on \( \sigma_k \) should then used as initial guesses for the next call of the Newton solver. Details you will find in the Literature e.g. under “path following” and “interior point methods”.

To solve the control problem numerically, we write a class kkt. From the problem setting, we identify the additional properties as follows:

- the regularization parameters \( \lambda \) and \( \sigma \), real numbers initialized by ones
- the control constraints \( u \) and \( \bar{u} \), here called u_upper and u_lower, initialized by \( \pm \infty \).
- the boundary conditions for the state equation. However, since the adjoint equation has boundary condition with the same left hand side as the state equation, we do not need to define a boundary condition for the adjoint equation.

We want to include some private properties to store the values of the functions \( y \) and \( u \). Since we are here interested also in the “technical” adjoint \( p \) and in the approximations of the Lagrange multipliers, we add these functions to the private properties of our class.

We will call them state, control, adjoint, multiplier_upper and multiplier_lower.

The code up to now reads
2.3 Systems of PDEs

classdef kktControl < pde
    properties(Access = public)
        bcState
        sigma = 1;
        lambda = 1;
        u_upper = inf
        u_lower = -inf
    end
    properties(SetAccess = private)
        state
        control
        adjoint
        multiplier_upper
        multiplier_lower
    end
end

Within the method df we can split the PDE again in a constant, a linear and a non-
linear part. The constant and the matrix of the linear part can be pre-computed in
initialize method. We will call the non-linear part here penalty, it will be a method of
class kkt with one additional argument. Also the jacobian method is given here. This is
important since the numerically computed Jacobian matrix often not accurate enough.
However, as mentioned before, if the Jacobian is known, it should be implemented also
to increase the performance of the code.

function dy = df(obj,~,y)
    % More formal definition of the lienar pde system. A and
    % be
    % must be implemented in initialize
    % The interface must be obj, time (not used here),
    % and solution
    dy = obj.A*y+obj.penalty(y)+obj.b;
end
function j = jacobian(obj,~,y)
    j = obj.A+obj.dpenalty(y);
end

Within the penalty method we need only the control from the vector obj.y, where we
store the solution of the whole KKT system. We arrange the solution parts state control
adjoint in exactly this order in obj.y. The order of the equations in the KKT system
may be the same as in (2.1), namely adjoint, gradient and state equation, or $\partial_y L(y, u, p)$,
$\partial_y L(y, u, p)$, and $\partial_y L(y, u, p)$. The result giving back by penalty must be of the same
size as obj.y. We initialize it by a zero vector.

The control is now $u = y(obj.grid.nPoints+1:2*obj.grid.nPoints,end)$ and finally
we must take care that the result of the computations must be bring in the position of
the gradient equation. The code is rather simple.

function val = penalty(obj,y)
    % separate u from the system vector
    val = zeros(size(y));
2 Examples

\[ u = y(obj.grid.nPoints+1:2*obj.grid.nPoints,end); \]
\[ val(obj.grid.nPoints+1:2*obj.grid.nPoints) = ... \]
\[ obj.mu*(-1./((u-obj.u_lower) + 1./((obj.u_upper-u)))) ; \]
\end

function val = dpenalty(obj,y)
\[ u = y(obj.grid.nPoints+1:2*obj.grid.nPoints,end); \]
\[ ns = 3*obj.grid.nPoints; \]
\[ d = obj.mu*[obj.n;(1./(u-obj.u_lower).^2)... \]
\[ +(1./(obj.u_upper-u).^2);obj.n]; \]
\[ val = spdiags(d,0,ns,ns); \]
\end

The method initialize is now straight forward. First we apply makeBoundaryMatrix on
the bcState property. Note that for more complex boundary settings one should call this
method outside the initialize method. Since we have only trivial boundary condition
right-hand-sides for the state equation, only one call of obj.fem.assemb is needed. The
same hold for the call of obj.fem.assema. For speed-up the computation of the jacobian,
we should give the jacobian method the sparsity pattern of the Jacobian. The speed-up
depends of the dimension of the problem, but it is typically \( O(nPoints) \). The code is
given by the next Listing. Note that by using assema/b methods instead of pde.initialize
we must set the obj.initialized property here explicitly.

function initialize(obj,y_d)
\[ obj.grid.makeBoundaryMatrix(obj.bcState); \]
\[ [Q,G,H,Ry] = obj.fem.assemb(obj.grid); \]
\[ [K,M,F] = obj.fem.assema(obj.grid,1,0,... \]
\[ -y_d); \]
\[ s = obj.fem.stiffSpring(K+M); \]
\[ N = sparse(obj.grid.nPoints ,obj.grid.nPoints); \]
\[ n = sparse(obj.grid.nPoints,1); \]
\[ obj.A = [obj.mass N -(K+M+s*(H'\*H)+Q);... \]
\[ N obj.lambda*obj.mass obj.mass; \]
\[ ...
\[ -(K+M+s*(H'\*H)+Q) obj.mass N]; \]
\[ obj.b = [F;... \]
\[ n;... \]
\[ s*H'\*Ry+G]; \]
\[ obj.y = ones(obj.grid.nPoints*3,1); \]
\[ obj.pattern = (obj.A~=0); \]
\[ obj.initialized = true; \]
\end

The heart of the solve method is now a call of pde.solve(’STATIONARY’). However, we
need a few further (helper) methods.

(i) For the Newton solver used in pde.solve(’STATIONARY’) we need a feasible
initial guess. We should compute one by an extra method initialGuess.

(ii) The constraints may be given in different formats, e.g. scalar number and
one as a vector of length number of points in the mesh. To bring them into
the same format (vector of length number of points in the mesh) we use
a function expandConstraints, that additionally computes a feasible initial
guess for the control.

(iii) To obtain also a feasible initial guess for the state and the adjoint, i.e. for the
whole KKT system, we use the method initialGuess.

(iv) Further, we need a method that checks if a solution is feasible.

All these helper methods will be described later.

When we used pde.solve('STATIONARY'), we solve our KKT system by Newtons method,
but for fix smoothing $\sigma$. Our aim is to reduce $\sigma$ sequentially, i.e. to construct a set of
$\sigma_0,...,\sigma_k \to 0$ and associated solutions of the KKT system, here $y_{\sigma_i}, u_{\sigma_i}, p_{\sigma_i}$. To construct
such path following we establish a loop where we reduce $\sigma$ by multiplying it by a fixed parameter $s \in (0,1)$. Note that the while loop can be only terminated by using a break
or return statement or, as in our case, by a exception. We use here a unusual to break
the loop. We have three cases:

(i) $\sigma$ is smaller then its smallest allowed value.

(ii) the solution (iterate) is not feasible.

In (ii) we throw in this sense a “good” exception. The code can be studied in detail in
the next Listing.

```matlab
function solve(obj, varargin)
    if ~obj.initialized
        obj.notInitialized.throwAsCaller;
    end
    obj.initialGuess;
    while true
        y = obj.y;
        try
            solve@pde(obj, 'STATIONARY');
            if obj.sigma < 1e-8
                MException('kkt3:sigmaneareps', ...
                            'sigma near eps').throw;
            end
            if ~obj.feasible
                MException('kkt3:notfeasible', ...
                            'solution is not feasible').throw;
            end
            obj.sigma = obj.sigma*0.85;
        catch
            obj.y = y;
            obj.splitSolution;
            fprintf(['Pathfollowing breaks with sigma = ', ...
                     num2str(obj.sigma), '\n']);
            return
        end
    end
```
2 Examples

The Listing below shows the complete class kkt, including the helper methods expandConstraints, feasible, splitSolution, initialGuess and a specialized plot that plots the solution and the approximations on the Lagrange multipliers.

```matlab
classdef kkt3 < pde
  % Example for a kkt system for constrained OCP. Most of code identical
  % with KKT2
  properties(Access = public)
    % to allow the user to change the boundary condition
    % of the state equation we make it public.
    bcState
    mu = 1;
    muFinal% path parameter for log term
    lambda = 1; % Tykhonow parameter
    u_upper = inf % fixed lower and upper control
    constraints.
    u_lower = -inf % For unconstrained problems set it
    +/- inf
  end
  properties(SetAccess = private)
    % propertise to store the state, the control and the
    % adjoint
    % equation. Only methods from kkt should change the
    % values.
    state
    control
    adjoint
    multiplier_upper
    multiplier_lower
    n % zero vector
  end

  methods (Access = protected)
    % These methods must be declared as protected.
    function dy = df(obj,~,y)
      % More formal definition of the lienar pde system
      % A and be
      % must be implemented in initialize
      % The interface must be obj, time (not used here)
      % and solution
      dy = obj.A*y+obj.penalty(y)+obj.b;
    end
    function j = jacobian(obj,~,y)
      j = obj.A+obj.dpenalty(y);
  end
end
```
2.3 Systems of PDEs

```matlab
methods(Access = private)

function val = penalty(obj,y)
% separate u from the system vector
val = zeros(size(y));
val(obj.grid.nPoints+1:2*obj.grid.nPoints) = ...
    obj.mu*(-1./(u-object.u_lower) + 1./(object.u_upper-u));
end

function val = dpenalty(obj,y)
u = y(obj.grid.nPoints+1:2*obj.grid.nPoints,end);
ns = 3*obj.grid.nPoints;
d = obj.mu*[object.n;1/(u-object.u_lower).^2)... +1/(object.u_upper-u).^2];
val = spdiags(d,0,ns,ns);
end

end

methods(Access = public)

function initialize(obj,c,a,y_d,varargin)
% The most important method of all
% We know that the boundary condition for the
% adjoint is always
% homogeneous. Hence, we have only non trivial
% boundary
% conditions for the state
if length(varargin)==1
    u_d = varargin{1};
else
    u_d = 0;
end
obj.grid.makeBoundaryMatrix(obj.bcState{:});
[Q,G,H,Ry] = obj.fem.assemb(obj.grid);
% We have no source for the state equation but a
% source for the
% adjoint. hence we can use the matrices for both
% equations and
% F only for the adjoint.
[K,M,Yd] = obj.fem.assema(obj.grid,c,a,...
    -y_d);

obj.M = obj.mass;
E = speye(obj.grid.nPoints);
% The stiff-spring approximation of Dirichlet boundary
```
% conditions (only needed if applied)
l = obj.fem.stiffSpring(K+M);
%
% Some sparse matrices/ vectors to fill the holes in the
%
% big matrix
N = sparse(obj.grid.nPoints,obj.grid.nPoints);
obj.n = sparse(obj.grid.nPoints,1);

% The pde system matrix. first line adjoint, second gradient
% third state equation. Solution vector sate, control, adjoint
         N   obj.lambda*E   E;...
        -(K+M+l*(H'*H)+Q)   obj.M   N];

% The big rhs, F contains y-yd but no boundary vector
% The state eqn. RHS has no source but both of the boundary
% vectors
obj.b = [Yd;...
        -u_d;...
        l*H'*Ry+G];
% To initialize y we set it to zero. This is very important
% since the solvers need the information that the number of
% grid points in the mesh is not the number of dof.
obj.y = ones(obj.grid.nPoints*3,1);
% At last, we set initialize true. Otherwise the solve method
% returns an error.

obj.initialized = true;
end

function solve(obj,varargin)
    % We iterate the nonlinear solver 'STATIONARY'
    % and decrease mu
    if ~obj.initialized
        obj.notInitialized.throwAsCaller;
    end

    % We need a FEASIBLE initial value for the IP method
% We compute an initial control from the control constraints...
obj.initialGuess;
k = 0;
while true
    y = obj.y;
    try
        solve@pde(obj,'STATIONARY');
        if obj.mu < obj.muFinal
            MException('kkt3:muneareps',...
                'mu near eps').throw;
        end
        if ~obj.feasible
            MException('kkt3:notfeasible',...
                'solution is not feasible').throw
        end
        obj.mu = obj.mu*0.55;
        k = k+1;
    catch ME
        % don't catch the exception
        obj.y = y; % take the last valid solution
        obj.splitSolution;
        fprintf([...'Pathfollowing breaks with mu = ',...
            num2str(obj.mu),'
            ' at ',num2str(k),'
            th iteration\n']);
        fprintf([ME.message,'
']);
        return
    end
end
end

methods(Access = private)
function u = expandConstraints(obj)
    if all(isfinite(obj.u_upper))&all(isfinite(obj.
        u_lower))
        if isscalar(obj.u_upper)
            obj.u_upper = obj.u_upper(ones(obj.grid.
                nPoints,1));
        end
        if isscalar(obj.u_lower)
            obj.u_lower = obj.u_lower(ones(obj.grid.
                nPoints,1));
        end
        u = 0.5*(obj.u_upper+obj.u_lower);
    return
end
2 Examples

if isinf(obj.u_upper)&&isfinite(obj.u_lower)
    if isscalar(obj.u_lower)
        obj.u_lower = obj.u_lower(ones(obj.grid.
        nPoints,1));
    end
    u = obj.u_lower+1;
    return
end
if ~isfinite(obj.u_lower)&&isfinite(obj.u_upper)
    if isscalar(obj.u_upper)
        obj.u_upper = obj.u_upper(ones(obj.grid.
        nPoints,1));
    end
    u = obj.u_upper-1;
    return
end
if isinf(obj.u_upper)&&isinf(obj.u_lower)
    % the stupid case of both inf/-inf
    constraints
    u = zeros(obj.grid.nPoints,1);
    return
end
end

function b = feasible(obj)
    obj.splitSolution;
    b = all(obj.u_lower < obj.control)&&...
        all(obj.control < obj.u_upper);
end

function initialGuess(obj)
    u = expandConstraints(obj);
    % Compute state. First extract that eqn. from
    kkt system
    A = obj.A(2*obj.grid.nPoints+1:3*obj.grid.nPoints
        ,1:obj.grid.nPoints);
    b = obj.b(2*obj.grid.nPoints+1:3*obj.grid.nPoints
        ,1);
    %
    y = A\(obj.M*u+b);
    p = -u*obj.lambda+obj.mu./(u-obj.u_lower)-obj.mu
        ./(obj.u_upper-u);
    obj.y = [y;u;p];
end

function splitSolution(obj)
    % We split the solutions by simple indexing obj.y
    and using
% obj.grid.nPoints
obj.state = obj.y(1:obj.grid.nPoints,end);
obj.control = obj.y(obj.grid.nPoints+1:2*obj.grid
  .nPoints,end);
obj.adjoint = obj.y(2*obj.grid.nPoints+1:3*obj.
    grid.nPoints,end);
obj.multiplier_upper = obj.mu./(obj.u_upper-obj.
    control);
obj.multiplier_lower = obj.mu./(obj.control-obj.
    u_lower);
end
end
end

As an example we solve the problem with $u = -50$, $\bar{u} = 100$ and
\[
y_d = \begin{cases} 
  2 & \text{for } 0.6 \leq x \leq 0.8 \\
  0 & \text{otherwise}
\end{cases}
\]
on the interval $\Omega = (0, 1)$. Further we start at $\sigma = 1$ and we use $\lambda = 10^{-5}$. As boundary condition for the state we choose homogeneous Dirichlet boundary conditions, i.e. $h = 1$ and $r = 0$.

```
function testKKT3
  clf,clc
  % define the kkt system
  kktSystem = kkt3();
  kktSystem.grid = grid1D();
  kktSystem.grid.intervall([0,1],0.01);
  kktSystem.fem = lagrange11D();
  kktSystem.u_lower = -50;
  kktSystem.u_upper = 100;
  kktSystem.sigma = 1;

  % define the boundary condition
  kktSystem.bcState = kktSystem.grid.dirichletBC('0.0');

  % define lambda
  kktSystem.lambda = 1e-5;
  yd = fyd(kktSystem.grid);
  kktSystem.initialize(yd);

  % solve KKT System
  kktSystem.solve();

  kktSystem.grid.plot(kktSystem.control)

  function val = fyd(g)
      val = zeros(size(g.p(1,:)));```
2 Examples

\[
\text{val}(g.p(1,:) >= 0.6 \& g.p(1,:) < 0.8) = 2;
\]

end

end

2.3.4 The Schnakenberg Model

This example is inspired by the collaboration with the group of Hannes Uecker from the University of Oldenburg. It is a problem arising from mathematical biology, cf. [Sch79].

We want to study the long-time behavior, i.e. the stationary points of the non-linear problem

\[
\begin{align*}
\dot{u} &= c_1 \Delta u + u^2 v - u & \text{in} \quad \Omega \times (t_0, t_1) \\
\dot{v} &= c_2 \Delta u - u^2 v + \lambda & \text{in} \quad \Omega \times (t_0, t_1)
\end{align*}
\]

(2.2)

with homogeneous Neumann boundary conditions and initial values \(u(t_0) = u_0\) and \(v(t_0) = v_0\). Let \(c_1 = 1, c_2 = 60\). The domain may be a bounded domain \(\Omega \subset \mathbb{R}^N\). The parameter \(\lambda\) is a positive real number. Altogether, this is a semi-linear PDE that fits in the setting of OOPDE.

We define a class that is independent from the domain dimension. Since the solution depends from \(\lambda\) we define \(\lambda\) as a class property. As usual, the components of the system should be also properties of the class. We define \(y = \left( \begin{array}{c} u \\ v \end{array} \right)\). All (added) properties are from class double, and \(\lambda\) is initialized as \(\lambda = 3\).

For the method \(\text{df}\) we use the abstract formulation
2.3 Systems of PDEs

```matlab
function val = df(obj, ~, y)
    val = obj.A*y + obj.nonlinear(y) + obj.b;
end
```

where \( \text{obj.A} \) will be defined as the linear part matrix in the initialize method as well as the constant part \( \text{obj.b} \) and \( \text{obj.nonlinear} \) will be defined as a private function.

Since the problems (2.2) has homogeneous Neumann boundary conditions, we have only matrices is the system that comes from the domain. The discrete system function \( \text{df} \) reads

\[
    df(u, v; \lambda) = -(K_{c_1} + M_1)u + M_1 u^2 v \\
    -K_{c_2} v - M_1 u^2 v + \lambda F_1
\]

where the subscripts indicate the dependence of the matrices/vectors from the associated coefficients.

The linear part is defined by the matrix

\[
    A = \begin{pmatrix}
        -(K_{c_1} + M_1) & N \\
        N & -K_{c_2}
    \end{pmatrix},
\]

where the constant part is

\[
    b = \begin{pmatrix}
        n \\
        \lambda F_1
    \end{pmatrix}.
\]

The non-linear part is a vector valued function

\[
    \phi(u, v) = \begin{pmatrix}
        M u^2 v \\
        -M u^2 v
    \end{pmatrix}.
\]

It will be defined in a private method

```matlab
function val = nonlinear(obj, y)
    n = obj.grid.nPoints;
    ul = y(1:n);
    vl = y(n+1:end);
    val = [obj.M*(ul.^2.*vl) -obj.M*(ul.^2.*vl)];
end
```

In the first lines we implement the extraction of \( u \) and \( v \) from \( y \). Note that we cannot use here \( \text{obj.u} \) and \( \text{obj.v} \). The jacobian method is defined by

\[
    jacobian(u, v) = A + \begin{pmatrix}
        2M diag(uv) & M diag(u^2) \\
        -2M diag(uv) & -M diag(u^2)
    \end{pmatrix}.
\]

The code is an one-to-one representation of this, where we have to implement of cause the splitting of \( y \) into \( u \) and \( v \).

Initialize is now a method with rather lean code. Note that we can here write the coefficients \( c_1 \) and \( c_2 \) as scalar factors and there is no need to assemble more than on stiffness matrix. Also the \( \lambda \) is a factor that can be multiplied by the source vector \( F \) computed for the constant source one.

A concept that has its premier here is the eventfun. To control the solver STEADYSTATE, we can use the eventfun. It terminates the solver, if an event occurs. In our case,
2 Examples

the event is $\|df(y)\|_\infty - \epsilon \leq 0$. This epsilon can be defined into a (hidden) property `solverOptions`. `solverOptions` is from class `struct` and can be extended by the user. However, it has a predefined field `solverTol` that we use here.

```matlab
function [value,isterminal,direction] = eventfun(obj,t,y)
    value = norm(obj.df(t,y),Inf)-obj.solverOptions.solverTol;
    isterminal = 1;
    direction = 0;
end
```

The complete class definition is given in the next listing.

```matlab
classdef schnakenberg < pde
    properties(Access = public)
        lambda@double = 3;
        u@double;
        v@double;
    end

    methods(Access = protected)
        function val = df(obj,~,y)
            val = obj.A*y+obj.nonlinear(y)+obj.b;
        end

        function J = jacobian(obj,~,y)
            n = obj.grid.nPoints;
            ul = y(1:n);
            vl = y(n+1:end);
                obj.M*sparse(1:n,1:n,ul.^2);...
                -2*obj.M*sparse(1:n,1:n,ul.*vl),...
                -obj.M*sparse(1:n,1:n,ul.^2)];
        end

        function [value,isterminal,direction] = eventfun(obj,t,y)
            value = norm(obj.df(t,y),Inf)-abs(obj.
                solverOptions.solverTol);
            isterminal = 1;
            direction = 0;
        end
    end

    methods(Access = public)
        function initialize(obj)
            [K,M,L] = obj.fem.assema(obj.grid,1,1,obj.lambda);
            N = sparse(obj.grid.nPoints,obj.grid.nPoints);
            n = sparse(obj.grid.nPoints,1);
            obj.A = -[K+M N];
        end
end
```
2.3 Systems of PDEs

\begin{verbatim}
N 60*K];
obj.b = [n
    L];
obj.M = obj.mass;
obj.D = [obj.mass N;N obj.mass];
obj.initialized = true;
end

function splitSolution(obj)
    n = obj.grid.nPoints;
    obj.u = obj.y(1:n,:);
    obj.v = obj.y(n+1:end,:);
end

function solve(obj,varargin)
    solve@pde(obj,varargin{:});
    obj.splitSolution;
end
end

methods(Access = private)
function val = nonlinear(obj,y)
    n = obj.grid.nPoints;
    ul = y(1:n);
    vl = y(n+1:end);
    val = [obj.M*(ul.^2.*vl)
        -obj.M*(ul.^2.*vl)];
end
end
\end{verbatim}

We should run some a test. We want to consider a 3D problem where

\[ \Omega = (-4\pi/\sqrt{2} - 1, 4\pi/\sqrt{2} - 1)^2 \times (-4\pi/\sqrt{3}\sqrt{\sqrt{2} - 1}, 4\pi/\sqrt{3}\sqrt{\sqrt{2} - 1}) \]

is a bar.

The complete program is shown in the next listing.

```matlab
1 system = schnakenberg();
2
3 system.grid = grid3Dpr;
4 kc=sqrt(sqrt(2)-1);
5 lx=4*pi/kc;
6 ly=4*pi/(sqrt(3)*kc);
7 xvec = linspace(-lx,lx,30);
8 yvec = linspace(-ly,ly,10);
9 zvec = linspace(-ly,ly,10);
10 system.grid.bar(xvec,yvec,zvec);
11
12 system.fem = bilinear3D;
75```
We define the domain in lines 3–10. Note that we use here the three argument call of grid3Dpr.bar. For prism elements, we have to use bilinear ansatz functions. In line 12 we call bilinear3D, the constructor method for bi-linear FE on prism elements.

In lines 14–20 we setup the problem, followed by initialize in line 23.

The idea of the solution process is to solve the in-stationary problem by time integration and then to use the solution as an initial value to solve the stationary problem by Newton’s method. The lines 25–29 defines the solver options and we call the solver with STEADYSTATE argument as well as the STATIONARY solver. With the option STEADYSTATE the solver method uses the ode15s method to integrate the time dependent problem until the norm of the RHS is lesser than a given tolerance, here defined in Line 26. The option STATIONARY call the non-linear solver for stationary problems, i.e. with $d = 0$ by Newtons method with the stopping criteria that the norm of the RHS of (2.2) is lesser than $1 \times 10^{-12}$. We change the property system.solverOptions.solverTol to the new value of $1 \times 10^{-12}$. Both solvers read the solverOptions.solverTol field to evaluate the respective stopping criteria.

In lines 31–38 we call some suitable plot methods. Note that grid3Dpr.plot only plots the mesh. All plot methods accept the usual optional arguments, as 'LineStyle', 'EdgeColor', etc.
2.3 Systems of PDEs

The results of the plot methods can be seen in the next figures. Note that a colormap influences the surface plot and the slice plot, but it has no effect to the iso-surfaces plot. The iso-surfaces plot uses a fix set of colors.

For $\lambda = 2.0$ we arrive a the solutions above. Note that the problem has no unique solution and will depend on $\lambda$.

Now we try solve this problem on a tetrahedron grid. We have only to replace

```javascript
system.grid = grid3Dpr;
```

by

```javascript
system.grid = grid3D;
```

and

```javascript
system.fem = bilinear3D;
```

by

```javascript
system.fem = lagrange13D;
```

To make it more interesting, we set now $\lambda = 1$. The grid contains 84.474 elements and 16.000 points compared with 13.804 of the prism grid. Maybe not surprising, the solver finds a different solution, cf. the following plots.
2 Examples

2.4 Adaptive Mesh refinement

OOPDE provides methods for adaptive mesh refinement. There are some predefined methods to estimate the local error as well as methods that implements algorithms to select elements to be refined. However, adaptive mesh refining is very problem specific and hence OOPDE can only provide methods for simple problems. Fortunately, the user can define error indicators for their own problems by overwriting OOPDE methods or by adding their own method to the class defining their specific problem.

2.4.1 General approach

We consider the PDE

\[-\nabla \cdot (c \nabla u) + au = f \text{ in } \Omega\]

\[\vec{n} \cdot c \nabla u + qu = g \text{ on } \Gamma.\]

According to [The95] we use the error indicator

\[E(\Omega_{\Delta_k}) = \alpha \|h^m(f - au)\|_{\Omega_{\Delta_k}} + \beta \left( \frac{1}{2} \sum_{\tau \in \Gamma_{\Delta_k}} h_{\tau}^{2m} (\vec{n}_\tau \cdot \nabla u)^2 \right)^{1/2}\]  

(2.3)

where \(\Omega_{\Delta_k}\) is the k-th element of the triangulation \(\Omega\) and \(\Gamma_{\Delta_k}\) is the boundary of \(\Omega_{\Delta_k}\). Moreover, \(h\) is here the area (or in 3D volume, in 1D length) of \(\Omega_{\Delta_k}\) and \(h_\tau\) the length (or in 3D the area) of \(\Gamma_{\Delta_k}\). The numbers \(\alpha\) and \(\beta\) are weights, and \(m\) is a order.
Note that \( \vec{n}_\tau \cdot \nabla u \) is the jump in flux across the element edge, what suggests the name \texttt{pdejmps} for the corresponding pdetool function. In OOPDE there is a method \texttt{errorInd} that implements the error measure (2.3). This method is from the abstract class \texttt{finiteElements}. It uses the static methods \texttt{fluxThroughEdges}, \texttt{localErrorL2}, and \texttt{fluxJumps}, declared as abstract in \texttt{finiteElements} class. These methods are implemented in \texttt{lagrangeXYD} classes, since the computation of the flux and \( L^2 \)-norms depends on the order of finite elements and the dimension of the domain. However, there is also a method \texttt{errorInd} in class \texttt{pde}, which can be overwritten in user application classes. The, the user may use the already implemented methods or implement their own error indicator methods using the grid data etc.

The method \texttt{selectElements2Refine} of \texttt{pde} class gives back the elements selected to be refined. It implements a criteria by Dörfler, i.e. the elements which error are larger as a factor \( \sigma \) time the accumulated sum of the local errors. This method can be overwritten by the users in their application classes.

### 2.4.2 Ad-hoc adaptive solving

We demonstrate the use of the error handling method at an ad-hoc implementation to solve a diffusion problem in 2D. We use the \texttt{heatTransfer} class form Section 2.2.1. The domain is again the “unit” L-shaped domain, \((0,1)^2 \setminus [0.5,1]^2\). The coefficients are given by \(c = 0.1, a = 1\). The source is defined by \( f(x) = \begin{cases} 10 & \text{on } [0.3,0.8]^2 \\ 0 & \text{otherwise} \end{cases} \).

```matlab
function testHeat2D ()
    pde = heatTransfer() ;
    pde.grid = grid2D() ;
    pde.fem = lagrange12D () ;
    pde.grid.lshape(inf);
    pde.grid.makeBoundaryMatrix(...
        pde.grid.dirichletBC ('1','0'));
    x = pde.grid.p(1,:); y = pde.grid.p(2,:);
    E = inf;
    while max(E)>1e-3
        pde.initialize(0.1,1,f(x,y)) ;
        pde.y = zeros(pde.grid.nPoints,1);
        pde.solve('LINEAR') ;
        E = pde.errorInd(0.1,1,f(x,y),1,1,1);
        indx = pde.selectElements2Refine(E,0.25);
        pde.grid.refineMesh(indx);
        x = pde.grid.p(1,:); y = pde.grid.p(2,:);
    end

    pde.initialize(0.1,1,f(x,y)) ;
    pde.y = zeros(pde.grid.nPoints,1);
    pde.solve('LINEAR') ;

    figure(1); pde.grid.plot(pde.y); drawnow
```

79
function val = f(x,y)
    val = zeros(size(x));
    val(x<0.8&y<0.8&x>0.3&y>0.3) = 10;
end

2.4.3 Writing an adaptive solver

Based on the last example we want to write a method that solves the problem by an adaptive solver. In particular such solver may be a wrapper for the adaption loop from Example 2.4.2.

We slightly adapt the heatTransfer class by adding three properties and one method. Since we need the coefficients and the source of the PDE for calculating the local error, we store them in the new properties \(c\), \(a\), and \(f\), c.f. the properties block in the class definition. The next listing presents the modified code. Note the use of the source function \(f\) as function_handle valued property when it is an argument for initialize. The method errorInd needs the value of \(f\), hence here \(f\) is evaluated at the grid points. Note further that in solveAdaptive the coefficients are properties \(obj.c\), \(obj.a\), and \(obj.f\).

classdef heatTransfer < pde
    properties (Access = public)
        c@double
        a@double
        f@function_handle
    end
    methods (Access = protected )
        function dy = df(obj,~,y)
            dy = obj.A * y + obj.b ;
        end
        function J = jacobian(obj,~,~)
            J = obj.A ;
        end
    end
    methods (Access = public )
        function initialize(obj,c,a,f )
            obj.c = c;
            obj.a = a;
            obj.f = f;
            b = zeros (size(obj.grid.p,1) ,1) ;
            initialize@pde(obj,1,c,b,a,obj.f) ;
            l = obj.fem.stiffSpring(obj.K + obj.M );
                obj.Q ) ;
        end
end
2.4 Adaptive Mesh refinement

```matlab
function solveAdaptive(obj, tol)
    while true
        obj.initialize(obj.c, obj.a, obj.f);
        obj.y = zeros(obj.grid.nPoints, 1);
        obj.solve('LINEAR');
        E = obj.errorInd(obj.c, obj.a, ...
                       obj.f(obj.grid.p(1,:), obj.grid.p(2,:))
               , 1, 1, 1);
        if max(E) < tol
            break
        end
        indx = obj.selectElements2Refine(E, 0.125);
        obj.grid.refineMesh(indx);
    end
    fprintf(['Solve on adaptive refined mesh with ', ...
             num2str(obj.grid.nPoints), ' nodes\n']);
end
end
```

In the main function, the loop is replaced by a call of `solveAdaptive`.

```matlab
function testHeat2D ()
    pde = heatTransfer () ;
    pde.grid = grid2D () ;
    pde.fem = lagrange12D () ;
    pde.grid.lshape(0.1) ;

    pde.grid.makeBoundaryMatrix(...
        pde.grid.dirichletBC ('0'));
    pde.initialize(0.1,1,@f) ;
    pde.solveAdaptive(1e-3);

    figure(1); pde.grid.plot();
    figure(2); pde.grid.plot(pde.y, 'LineStyle', 'none');
end
```

```matlab
function val = f(x, y)
    val = zeros(size(x));
    val(x<0.7&y<0.7&x>0.3&y>0.3) = 10;
end
```
2 Examples

Figure 2.2: Adaptively refined grid of around 6700 points and the associated solution.

2.5 Some special issues

2.5.1 The meaning of pde-initialized

The initialize method computes all matrices used by the solvers. If the initialization was successful, the initialized property will be set true. The solve method will check the property and abort if initialize is not true. However, the system matrices stay valid until the mesh changes. In this case, the meshRefine method will set pde.initialize false. The user must now re-initialize the pde object by calling pde.initialize again.

2.5.2 Resetting pde.fc and pde.g

The hidden property pde.fc is the FAC arguments of numjac. FAC is working storage. It is initialized by empty double array and will be filled by numjac. For further information see the documentation of numjac.

This property will cause some strange errors. For instance, the typical work-flow constructing an adaptive mesh-refinement procedure is

S1 solve the PDE
S2 estimate the error and identify elements to refine
S3 refine the mesh goto S1

A problem occurs if within the solve process the numerical jacobian method, i.e. pde.jacobian is used. After the refinement and re-initialize of the system the two helper properties : pde.fc and pde.g become invalid and must be cleared. Otherwise, numjac will throw an exception.
Thanks

We would like to thank all “early users” for testing the software, especially

- Andreas Himmel (University of Magdeburg) and Pascal Friedrich (UAS Darmstadt) for improving the examples, the (German) manual and the help files by using OOPDE for their thesis,
- Hannes Uecker and Daniel Wetzel (University of Oldenburg) for discussion on code structure and the initial code of the 3D iso-surface method as well as for the Schankenber example,
- Zhi George Lin (Zhejiang University Hangzhou) not only for testing the code on Mac computer and finding a strange bug within the plotting method,
- Jens Seidel for the discussion on the “right or wrong” class structure,
- Marcel Heim and Tobias Höer (both TUBAF Freiberg) for coding the lagrange22D class and the anisotropic diffusion coefficient,
- Martin Ullmann (TUBAF Freiberg) for the real world application demanding the 3D capabilities we now have implemented in OOPDE.
Chapter 3

3 License

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or any later version.

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

For the text of the GNU General Public License V2 see Appendix B.

This distribution of OOPDE contains a adapted version of DistMesh. DistMesh is under GPL. (C) 2004-2011 Per-Olof Persson.

OOPDE used particulary refineRGB by S. Funken, D. Praetorius, P. Wissgott from the program p1afem p1afem is a supplement to the paper "Efficient Implementation of Adaptive P1-FEM in MATLAB" by S. Funken, D. Praetorius, and P. Wissgott. (C) S. Funken, D. Praetorius, P. Wissgott 10-07-08

If you use OOPDE in any program or publication, please acknowledge its author by adding a reference to:

Uwe Prüfert, OOPDE-An Object oriented approach to finite elements in MATLAB. TU Berkakademie Freiberg.
http://www.mathe.tu-freiberg.de/nmo/mitarbeiter/uwe-pruefert/software

3.1 GNU GENERAL PUBLIC LICENSE (GPL)

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free
3.1 GNU GENERAL PUBLIC LICENSE (GPL)

Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change. b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License. c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)
3 License

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or, b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or, c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients’ exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.
If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
Index

A
Abstraction, 12
Attributes, 12

B
boundary condition matrix, 10
boundary conditions, 10
Burger’s equation, 43

C
Class, 12
Class definition, 13
Class heatTransfer, 14
clear all, 15
clear classes, 15
Control constraints, 61

D
Dirichlet boundary conditions, 11

E
Encapsulation, 12

F
Finite Elements, 5

G
Green’s formula, 6

H
Heat transfer problem, 39

I
Inheritance, 12
Installation, 11

K
Karush-Kuhn-Tucker, 53

L
L-shape, 31

M
makeBoundaryMatrix, 10
MATLAB command window, 12
MATLAB special issues, 15

N
Non linear coupling, 56

O
Object oriented programming, 12
Optimal control problem, 53

P
Parabolic PDE, 46
PDE toolbox, 2
Persistence, 13
Polymorphism, 13

R
Reacting Flow, 56
Robin boundary conditions, 11

S
Systems of PDE, 53
Bibliography


