

# AMD<sub>i</sub>S – Adaptive Multi-Dimensional Simulations

Eine Einführung in das FEM-Framework

Simon Praetorius\*

`simon.praetorius@tu-dresden.de`

\*Institut für Wissenschaftliches Rechnen, Technische Universität Dresden

September 22, 2015



**TECHNISCHE  
UNIVERSITÄT  
DRESDEN**



# About this Course

- Goals: Introductory course
  - ▶ You know how to use the software AMDiS
  - ▶ You can solve a scalar equation, or a system of elliptic equations
  - ▶ You can handle instationary problems, nonlinearities and complex boundary conditions
  - ▶ (You can run your simulation in parallel)
- References:
  - ▶ Some theoretical background and basic design ideas of AMDiS software: ALBERTA-FEM especially the ALBERT 1.0 documentation.  
<http://goo.gl/ZMI2kA>
  - ▶ The AMDiS-Wiki  
<https://goo.gl/Jy3u1u>
  - ▶ Some (old) PDF Documentation on Fusionforge  
<https://goo.gl/5ngfYd>
- Workshop material can be found on GitHub  
<https://goo.gl/3IxKYx>

# Outline

- ① Scalar linear second order PDEs
- ② Handling data on unstructured grids
- ③ Adaptivity and systems of equations
- ④ Time-dependent and nonlinear problems

# AMDiS Adaptive Multi-Dimensional Simulations

AMDiS developed around 2005. Basis: C-library **ALBERTA**. Now: an object-oriented C++-Framework. Basic concepts

- **High abstraction level:** (Physical) Problems can be formulated with little knowledge about numerical details
- **Generality:** Solve a broad class of PDE problems. Linear and nonlinear problems, stationary and instationary. Multiple dimensions and coupling of different dimensions
- **Extensibility:** Interface to extend AMDiS in several aspects, e.g. own error estimators, linear solvers, preconditioners, time-stepping schemes
- **Efficiency:** Several tools for highly efficient simulations, e.g. adaptive meshes, parallelization, multi-mesh, fast linear solver libraries

# AMDiS Adaptive Multi-Dimensional Simulations

## Short history:

- 2002: Beginning of development (based on C-library ALBERT(A))
- 2005: First release
- 2007: PhD Thesis of Simon Vey: *"Adaptive Finite Elements for Systems of PDEs"*
- 2007: Development in the IWR at TU-Dresden
- 2008: First parallel version
- 2011: Release of stable version 0.9
- 2013: PhD Thesis of Thomas Witkowski: *"Software concepts and algorithms for an efficient and scalable parallel finite element method"*
- 2014: Generic expression terms introduced
- 2016(?): AMDiS 2.0

## Developers

Axel Voigt, Simon Vey, Christina Stöcker, Thomas Wittkowski, Andreas Naumann, Simon Praetorius, ...

# AMDiS Adaptive Multi-Dimensional Simulations

## Some features

- Solve (sequence of) Systems of **stationary linear PDEs of 2nd order**
- **Time integrators**: e.g. Rosenbrock method
- **Nonlinear solvers**: e.g. Newton method
- **Adaptivity** in space and time
- Lagrange **basis functions** (deg. 1–4), and center-bubble function
- **Mixed finite-elements**, e.g.  $P^2/P^1$ , Mini-Element
- **Multi-Mesh** method (i.e. different components with different mesh)
- Sequential and **parallel** (tested with up to 16K cores, and  $10^9$  DOFs)
- **Multi-Grid**: geometric (in AMDiS), algebraic (external library)
- Interface to **linear solvers**: (P)MTL4, PETSc, Hypre
- **FETI-DP** / Schur-complement solvers in parallel

# Session 1

## Second order PDE

General problem formulation:

$$[\partial_t u] + cu - \nabla \cdot (\underline{b}u) - \nabla \cdot (\mathbb{A} \nabla u) = f, \quad \text{in } \Omega \times (0, T]$$

with  $u = u(x)$  the unknown function and coefficients  $c, \underline{b}, \mathbb{A}$  that can depend on space, time, and other quantities  $v$  living in  $\Omega$ :

$$c = c(t, x, v, \nabla v),$$

$$\underline{b} = \underline{b}(t, x, v, \nabla v),$$

$$\mathbb{A} = \mathbb{A}(t, x, v, \nabla v),$$

$$f = f(t, x, v, \nabla v),$$

equipped with (initial- and ) boundary conditions on  $\partial\Omega$ .



## Second order PDE

General problem formulation:

$$[\partial_t u] + cu + \underline{b} \cdot \nabla u - \nabla \cdot (\mathbb{A} \nabla u) = f, \quad \text{in } \Omega \times (0, T]$$

with  $u = u(x)$  the unknown function and coefficients  $c, \underline{b}, \mathbb{A}$  that can depend on space, time, and other quantities  $v$  living in  $\Omega$ :

$$c = c(t, x, v, \nabla v),$$

$$\underline{b} = \underline{b}(t, x, v, \nabla v),$$

$$\mathbb{A} = \mathbb{A}(t, x, v, \nabla v),$$

$$f = f(t, x, v, \nabla v),$$

equipped with (initial- and ) boundary conditions on  $\partial\Omega$ .

# System of second order PDEs

General problem formulation:

$$\sum_j [\mathcal{M}^{i,j} \partial_t u^j] + c^{i,j} u^j + \underline{b}^{i,j} \cdot \nabla u^j - \nabla \cdot (\mathbb{A}^{i,j} \nabla u^j) = f^i, \quad \text{in } \Omega \times (0, T]$$

with  $u^j = u^j(x)$  the unknown functions and coefficients  $c, \underline{b}, \mathbb{A}$  that can depend on space, time, and other quantities  $v$  living in  $\Omega$ :

$$c^{i,j} = c^{i,j}(t, x, \{v_k\}, \{\nabla v_k\}),$$

$$\underline{b}^{i,j} = \underline{b}^{i,j}(t, x, \{v_k\}, \{\nabla v_k\}),$$

$$\mathbb{A}^{i,j} = \mathbb{A}^{i,j}(t, x, \{v_k\}, \{\nabla v_k\}),$$

$$f^i = f^i(t, x, \{v_k\}, \{\nabla v_k\}),$$

equipped with (initial- and ) boundary conditions on  $\partial\Omega$ .

## Second order PDEs

AMDiS is an FEM framework and FEM is based on the **weak formulation** of the equations, thus the basic formulation for problems is:

Find  $u \in L_2(0, T; V^{(1)})$ , s.t.

$$[\langle \partial_t u, \theta \rangle_\Omega] + \langle cu, \theta \rangle_\Omega + \langle \underline{b}u, \nabla \theta \rangle_\Omega + \langle \mathbb{A} \nabla u, \nabla \theta \rangle_\Omega = \langle f, \theta \rangle_\Omega, \quad \forall \theta \in V^{(0)}$$

with  $\langle a, b \rangle_\Omega := \int_\Omega a \cdot b \, dx$  and  $V^{(0)}, V^{(1)}$  compatible spaces, e.g.

- $V^{(0)} = V^{(1)} = H^1(\Omega)$ , or
- $V^{(1)} = V_G := \{u \in H^1(\Omega) : u|_{\partial\Omega} = G\}$ ,  $V^{(0)} = V_0$ .

## Second order PDEs

AMD<sub>i</sub>S is an FEM framework and FEM is based on the **weak formulation** of the equations, thus the basic formulation for problems is:

Find  $u \in L_2(0, T; V^{(1)})$ , s.t.

$$[\langle \partial_t u, \theta \rangle_\Omega] + \langle cu, \theta \rangle_\Omega + \langle \underline{b} \cdot \nabla u, \theta \rangle_\Omega + \langle \mathbb{A} \nabla u, \nabla \theta \rangle_\Omega = \langle f, \theta \rangle_\Omega, \quad \forall \theta \in V^{(0)}$$

with  $\langle a, b \rangle_\Omega := \int_\Omega a \cdot b \, dx$  and  $V^{(0)}, V^{(1)}$  compatible spaces, e.g.

- $V^{(0)} = V^{(1)} = H^1(\Omega)$ , or
- $V^{(1)} = V_G := \{u \in H^1(\Omega) : u|_{\partial\Omega} = G\}$ ,  $V^{(0)} = V_0$ .

## Second order PDEs

AMDiS is an FEM framework and FEM is based on the **weak formulation** of the equations, thus the basic formulation for problems is:

Find  $u \in L_2(0, T; V^{(1)})$ , s.t.

$$\begin{aligned} [\langle \partial_t u, \theta \rangle_\Omega] + \langle cu, \theta \rangle_\Omega + \langle \underline{b} \cdot \nabla u, \theta \rangle_\Omega + \langle \mathbb{A} \nabla u, \nabla \theta \rangle_\Omega + \langle \alpha u, \theta \rangle_{\partial\Omega} \\ = \langle f, \theta \rangle_\Omega + \langle g, \theta \rangle_{\partial\Omega}, \quad \forall \theta \in V^{(0)} \end{aligned}$$

with  $\langle a, b \rangle_\Omega := \int_\Omega a \cdot b \, dx$  and  $V^{(0)}, V^{(1)}$  compatible spaces, e.g.

- $V^{(0)} = V^{(1)} = H^1(\Omega)$ , or
- $V^{(1)} = V_G := \{u \in H^1(\Omega) : u|_{\partial\Omega} = G\}$ ,  $V^{(0)} = V_0$ .

# Basic ingredients for an AMDiS program

What data/information do you need to formulate your problem?

- 1 Description of your **domain**  $\Omega$  + a **triangulation**  $\mathcal{T}_h$  of the domain  
→ Mesh
- 2 Function-space  $V$ , or its **basis-functions** respectively.
  - ▶ P1 := Lagrange elements with polynomial degree  $p = 1$ :

$$V = \{v \in H^1(\Omega) : v|_T \in \mathbb{P}_p(T), \forall T \in \mathcal{T}_h(\Omega)\}$$

- ▶ P1+bubble := P1 + center bubble-function

$V^{(0)}$  is called RowFeSpace and  $V^{(1)}$  is called ColumnFeSpace.

- 3 **Solution** vector  $u$ , based on a numbering of all DOFs → DOFVector

$$u(x) = \sum_i u_i \phi_i(x),$$

with  $\{\phi_i\}$  a basis of  $V^{(1)}$ .

# Basic ingredients for an AMDiS program

What data/information do you need to formulate your problem?

- 1 Description of your **domain**  $\Omega$  + a **triangulation**  $\mathcal{T}_h$  of the domain  
→ Mesh
- 2 Function-space  $V$ , or its **basis-functions** respectively.
  - ▶ P1 := Lagrange elements with polynomial degree  $p = 1$ :

$$V = \{v \in H^1(\Omega) : v|_T \in \mathbb{P}_p(T), \forall T \in \mathcal{T}_h(\Omega)\}$$

- ▶ P1+bubble := P1 + center bubble-function

$V^{(0)}$  is called RowFeSpace and  $V^{(1)}$  is called ColumnFeSpace.

- 3 **Solution** vector  $u$ , based on a numbering of all DOFs → DOFVector

$$u(x) = \sum_i u_i \phi_i(x),$$

with  $\{\phi_i\}$  a basis of  $V^{(1)}$ .

# Basic ingredients for an AMDiS program

What data/information do you need to formulate your problem?

- 1 Description of your **domain**  $\Omega$  + a **triangulation**  $\mathcal{T}_h$  of the domain  
→ Mesh
- 2 Function-space  $V$ , or its **basis-functions** respectively.
  - ▶ P1 := Lagrange elements with polynomial degree  $p = 1$ :

$$V = \{v \in H^1(\Omega) : v|_T \in \mathbb{P}_p(T), \forall T \in \mathcal{T}_h(\Omega)\}$$

- ▶ P1+bubble := P1 + center bubble-function

$V^{(0)}$  is called RowFeSpace and  $V^{(1)}$  is called ColumnFeSpace.

- 3 **Solution** vector  $u$ , based on a numbering of all DOFs → DOFVector

$$u(x) = \sum_i u_i \phi_i(x),$$

with  $\{\phi_i\}$  a basis of  $V^{(1)}$ .



# Basic ingredients for an AMDiS program

What data/information do you need to formulate your problem?

- 1 Description of your **domain**  $\Omega$  + a **triangulation**  $\mathcal{T}_h$  of the domain  
→ Mesh
- 2 Function-space  $V$ , or its **basis-functions** respectively.
  - ▶ P1 := Lagrange elements with polynomial degree  $p = 1$ :

$$V = \{v \in H^1(\Omega) : v|_T \in \mathbb{P}_p(T), \forall T \in \mathcal{T}_h(\Omega)\}$$

- ▶ P1+bubble := P1 + center bubble-function

$V^{(0)}$  is called RowFeSpace and  $V^{(1)}$  is called ColumnFeSpace.

- 3 **Solution** vector  $u$ , based on a numbering of all DOFs → DOFVector

$$u(x) = \sum_i u_i \phi_i(x),$$

with  $\{\phi_i\}$  a basis of  $V^{(1)}$ .

# Basic ingredients for an AMDiS program

## 4 Coefficient functions $c, \underline{b}, \mathbb{A} \rightarrow \text{OperatorTerm}$

Categorized by degree of derivative:

$$cu - \nabla \cdot (\underline{b}u) - \nabla \cdot (\mathbb{A} \nabla u) = f$$

- ▶  $c, f$ : ZeroOrderTerm (ZOT)
- ▶  $\underline{b}$ : FirstOrderTerm (FOT)
  - ★ Derivative is on trial function: GRD\_PHI:  $\langle \underline{b} \cdot \nabla \phi, \psi \rangle$
  - ★ Derivative is on test function: GRD\_PSI:  $\langle \underline{b} \phi, \nabla \psi \rangle$
- ▶  $\mathbb{A}$ : SecondOrderTerm (SOT)

## 5 Boundary terms, i.e. coefficient functions $\alpha, g$ and properties of the pair of function spaces $V^{(0)}, V^{(1)}$ .

# Basic ingredients for an AMDiS program

## 4 Coefficient functions $c, \underline{b}, \mathbb{A} \rightarrow \text{OperatorTerm}$

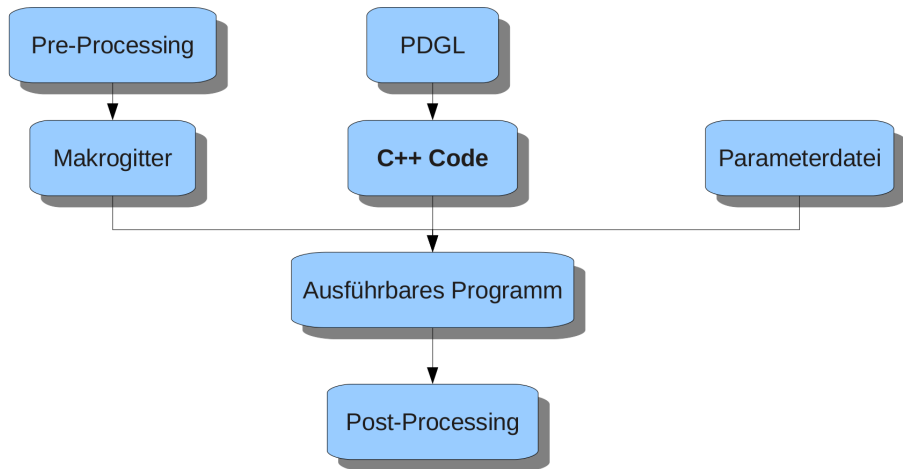
Categorized by degree of derivative:

$$cu - \nabla \cdot (\underline{b}u) - \nabla \cdot (\mathbb{A} \nabla u) = f$$

- ▶  $c, f$ : ZeroOrderTerm (ZOT)
- ▶  $\underline{b}$ : FirstOrderTerm (FOT)
  - ★ Derivative is on trial function: GRD\_PHI:  $\langle \underline{b} \cdot \nabla \phi, \psi \rangle$
  - ★ Derivative is on test function: GRD\_PSI:  $\langle \underline{b} \phi, \nabla \psi \rangle$
- ▶  $\mathbb{A}$ : SecondOrderTerm (SOT)

## 5 Boundary terms, i.e. coefficient functions $\alpha, g$ and properties of the pair of function spaces $V^{(0)}, V^{(1)}$ .

# General procedure



# Basic structure of AMDiS program (C++ code)

- Header file:

```
#include "AMDiS.h"
```

- AMDiS initialization:

```
using namespace AMDiS;  
int main(int argc, char** argv) {  
    AMDiS::init(argc, argv);  
    ...  
    AMDiS::finalize();  
}
```

init(...):

- ▶ initializes *PMTL4*, *PETSc*, *MPI*, *Zoltan*
- ▶ parses command line arguments (run program with `--help` to see available options)
- ▶ reads a parameter-file

finalize():

- ▶ finalizes *PMTL4*, *PETSc*, *MPI*

# Stationary equation

We start with the most simple equation: Poisson equation

$$-\Delta u = f(x) \quad \text{in } \Omega, \quad u|_{\partial\Omega} = 0, \quad f(x) \equiv 1$$

$\Rightarrow$  coefficient functions are:  $c = 0$ ,  $\underline{b} = 0$ ,  $\mathbb{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \equiv 1$ .

Weak formulation: Find  $u \in V_0$ , s.t.

$$\langle \nabla u, \nabla \theta \rangle_{\Omega} = \langle f, \theta \rangle_{\Omega}, \quad \forall \theta \in V_0.$$

- Problem definition

```
ProblemStat prob("poisson");  
prob.initialize(INIT_ALL); // init FeSpace, Mesh, ...
```

- Operator definition

```
Operator opL( prob.getFeSpace(), prob.getFeSpace() );  
addSOT( opL, 1.0 ); //  $\langle \nabla u, \nabla \theta \rangle_{\Omega}$ ,  $\mathbb{A} = 1.0$   
Operator opF( prob.getFeSpace() );  
addZOT( opF, 1.0 ); //  $\langle f, \theta \rangle_{\Omega}$ ,  $f = 1.0$ 
```

- Add Operators to Problem

```
prob.addMatrixOperator( opL, 0, 0 );
```

# Stationary equation

We start with the most simple equation: Poisson equation

$$-\Delta u = f(x) \quad \text{in } \Omega, \quad u|_{\partial\Omega} = 0, \quad f(x) \equiv 1$$

$\Rightarrow$  coefficient functions are:  $c = 0$ ,  $\underline{b} = 0$ ,  $\mathbb{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \equiv 1$ .

Weak formulation: Find  $u \in V_0$ , s.t.  $\langle \nabla u, \nabla \theta \rangle_{\Omega} = \langle f, \theta \rangle_{\Omega}$ ,  $\forall \theta \in V_0$ .

- **Problem definition**

```
ProblemStat prob("poisson");  
prob.initialize(INIT_ALL); // init FeSpace, Mesh, ...
```

- Operator definition

```
Operator opL( prob.getFeSpace(), prob.getFeSpace() );  
addSOT( opL, 1.0 ); //  $\langle \nabla u, \nabla \theta \rangle_{\Omega}$ ,  $\mathbb{A} = 1.0$   
Operator opF( prob.getFeSpace() );  
addZOT( opF, 1.0 ); //  $\langle f, \theta \rangle_{\Omega}$ ,  $f = 1.0$ 
```

- Add Operators to Problem

```
prob.addMatrixOperator( opL, 0, 0 );  
prob.addVectorOperator( opF, 0 );
```

- Define boundary conditions

```
prob.addDirichletBC( nr, 0, 0, new Constant(0.0) );
```

# Stationary equation

We start with the most simple equation: Poisson equation

$$-\Delta u = f(x) \quad \text{in } \Omega, \quad u|_{\partial\Omega} = 0, \quad f(x) \equiv 1$$

$\Rightarrow$  coefficient functions are:  $c = 0$ ,  $\underline{b} = 0$ ,  $\mathbb{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \equiv 1$ .

Weak formulation: Find  $u \in V_0$ , s.t.  $\langle \nabla u, \nabla \theta \rangle_{\Omega} = \langle f, \theta \rangle_{\Omega}$ ,  $\forall \theta \in V_0$ .

- Problem definition

```
ProblemStat prob("poisson");  
prob.initialize(INIT_ALL); // init FeSpace, Mesh, ...
```

- Operator definition

```
Operator opL( prob.getFeSpace(), prob.getFeSpace() );  
addSOT( opL, 1.0 ); //  $\langle \nabla u, \nabla \theta \rangle_{\Omega}$ ,  $\mathbb{A} = 1.0$   
Operator opF( prob.getFeSpace() );  
addZOT( opF, 1.0 ); //  $\langle f, \theta \rangle_{\Omega}$ ,  $f = 1.0$ 
```

- Add Operators to Problem

```
prob.addMatrixOperator( opL, 0, 0 );  
prob.addVectorOperator( opF, 0 );
```

- Define boundary conditions

```
prob.addDirichletBC( nr, 0, 0, new Constant(0.0) );
```



# Stationary equation

Find  $u \in V_0$ , s.t.  $\langle \nabla u, \nabla \theta \rangle_\Omega = \langle f, \theta \rangle_\Omega, \forall \theta \in V_0$ .

- Problem definition

```
ProblemStat prob("poisson");  
prob.initialize(INIT_ALL); // init FeSpace, Mesh, ...
```

- Operator definition

```
Operator opL( prob.getFeSpace(), prob.getFeSpace() );  
addSOT( opL, 1.0 ); //  $\langle \nabla u, \nabla \theta \rangle_\Omega, \mathbb{A} = 1.0$   
Operator opF( prob.getFeSpace() );  
addZOT( opF, 1.0 ); //  $\langle f, \theta \rangle_\Omega, f = 1.0$ 
```

- Add Operators to Problem

```
prob.addMatrixOperator( opL, 0, 0 );  
prob.addVectorOperator( opF, 0 );
```

- Define boundary conditions

```
prob.addDirichletBC( nr, 0, 0, new Constant(0.0) );
```

where `nr` is a boundary number and `Constant` a predefined functor, that returns always 0.

# Stationary equation

Find  $u \in V_0$ , s.t.  $\langle \nabla u, \nabla \theta \rangle_\Omega = \langle f, \theta \rangle_\Omega, \forall \theta \in V_0$ .

- Problem definition

```
ProblemStat prob("poisson");  
prob.initialize(INIT_ALL); // init FeSpace, Mesh, ...
```

- Operator definition

```
Operator opL( prob.getFeSpace(), prob.getFeSpace() );  
addSOT( opL, 1.0 ); //  $\langle \nabla u, \nabla \theta \rangle_\Omega, \mathbb{A} = 1.0$   
Operator opF( prob.getFeSpace() );  
addZOT( opF, 1.0 ); //  $\langle f, \theta \rangle_\Omega, f = 1.0$ 
```

- Add Operators to Problem

```
prob.addMatrixOperator( opL, 0, 0 );  
prob.addVectorOperator( opF, 0 );
```

- Define boundary conditions

```
prob.addDirichletBC( nr, 0, 0, new Constant(0.0) );
```

where  $nr$  is a boundary number and `Constant` a predefined functor, that returns always 0.

# Assemble and solve equation

In order to solve the PDE system: assemble matrix, solve linear system:

- Store Informations about solution process:

```
AdaptInfo adaptInfo("adapt");
```

- Assemble and solve system:

```
prob.assemble(&adaptInfo);  
prob.solve(&adaptInfo);
```

- Write solution to file:

```
prob.writeFiles(&adaptInfo, true);
```

```

// create and init the scalar problem
ProblemStat prob("poisson");
prob.initialize(INIT_ALL);

// define operators
Operator opLaplace(prob.getFeSpace(), prob.getFeSpace());
    addSOT(opLaplace, 1.0); // <grad(u), grad(theta)>
Operator opF(prob.getFeSpace());
    addZOT(opF, 1.0); // <f(x), theta>

// add operators to problem
prob.addMatrixOperator(opLaplace, 0, 0);
prob.addVectorOperator(opF, 0);

// add boundary conditions
BoundaryType nr = 1;
prob.addDirichletBC(nr, 0, 0, new Constant(0.0));

AdaptInfo adaptInfo("adapt");

// assemble and solve linear system
prob.assemble(&adaptInfo);
prob.solve(&adaptInfo);

prob.writeFiles(&adaptInfo, true);

```

# Compile and run AMDiS programs

- CMake configuration:

```
project("workshop")
find_package(AMDIS REQUIRED)
if (AMDIS_FOUND)
    include($AMDIS_USE_FILE)
endif (AMDIS_FOUND)

add_executable("poisson1" src/poisson1.cc)
target_link_libraries("poisson1" $AMDIS_LIBRARIES)
```

- ▶ CMake requires variable `AMDIS_DIR` to point to directory, that contains the file `AMDISConfig.cmake`
- ▶ See <http://goo.gl/kVe0Z2> for documentation of cmake commands.

- Run an AMDiS program by

```
./executable INIT-FILE
```

# The Parameter file

A parameter file (init-file) controls various parameters of the solution process, defines the FiniteElemSpace and sets the mesh and is mandatory.

- Init-files have the suffix `.dat.Xd`, where `X` is in  $\{1, 2, 3\}$ .
- Parameter definition:

```
parameter_name:  parameter_value  % a comment
```

- ▶ The ':' sign is the delimiter between parameter name and value
- ▶ The '%' sign indicates a starting comment

- Read a parameter from init-file:

```
value_type value = value0;  
Parameters::get("parameter_name", value);
```

- ▶ `value0` is default, if parameter not found
- ▶ `value_type` can be number, vector, string
- ▶ arithmetic expressions are parsed, interpreted and cast to `value_type`

The Initfile manual: <https://goo.gl/Dhm9Bx>

# The Parameter file

A parameter file (init-file) controls various parameters of the solution process, defines the FiniteElemSpace and sets the mesh and is mandatory.

- Init-files have the suffix `.dat.Xd`, where `X` is in  $\{1, 2, 3\}$ .
- Parameter definition:

```
parameter_name:  parameter_value  % a comment
```

- ▶ The ':' sign is the delimiter between parameter name and value
- ▶ The '%' sign indicates a starting comment

- Read a parameter from init-file:

```
value_type value = value0;  
Parameters::get("parameter_name", value);
```

- ▶ `value0` is default, if parameter not found
- ▶ `value_type` can be number, vector, string
- ▶ arithmetic expressions are parsed, interpreted and cast to `value_type`

The Initfile manual: <https://goo.gl/Dhm9Bx>

# The Parameter file

A parameter file (init-file) controls various parameters of the solution process, defines the FiniteElemSpace and sets the mesh and is mandatory.

- Init-files have the suffix `.dat.Xd`, where `X` is in  $\{1, 2, 3\}$ .
- Parameter definition:

```
parameter_name:  parameter_value  % a comment
```

- ▶ The ':' sign is the delimiter between parameter name and value
- ▶ The '%' sign indicates a starting comment

- Read a parameter from init-file:

```
value_type value = value0;  
Parameters::get("parameter_name", value);
```

- ▶ `value0` is default, if parameter not found
- ▶ `value_type` can be number, vector, string
- ▶ arithmetic expressions are parsed, interpreted and cast to `value_type`

The Initfile manual: <https://goo.gl/Dhm9Bx>



# The Parameter file

```
dimension of world: 2

mesh->macro file name: ./macro/macro1.2d
mesh->global refinements: 10

poisson->mesh: mesh
poisson->dim: 2
poisson->components: 1
poisson->feSpace[0]: P1
poisson->name[0]: u

poisson->solver: cg
poisson->solver->max iteration: 1000
poisson->solver->tolerance: 1.e-8
poisson->solver->left precon: diag

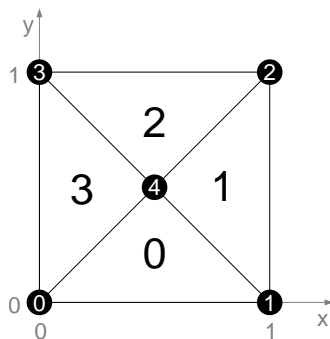
poisson->output->filename: ./output/poisson1.2d
poisson->output->ParaView format: 1
```

# The Macro-Mesh

In AMDiS the geometry and coarse triangulation is called Macro-Mesh

## Header

```
DIM: 2  
DIM_OF_WORLD: 2  
number of elements: 4  
number of vertices: 5
```



ASCII-File, that describes dimension of the elements (DIM), dimension of the world (DIM\_OF\_WORLD), the vertex coordinates, element connectivity, element boundary numbers and a neighborhood relation.

# The Macro-Mesh

In AMDiS the geometry and coarse triangulation is called Macro-Mesh

## Mesh description

element vertices:

0 1 4

1 2 4...

element boundaries:

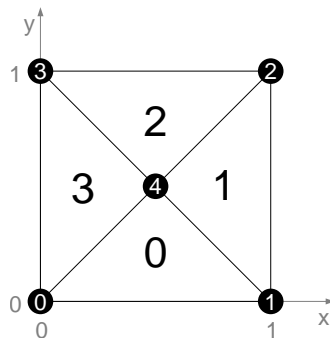
0 0 1

0 0 1...

vertex coordinates:

0.0 0.0

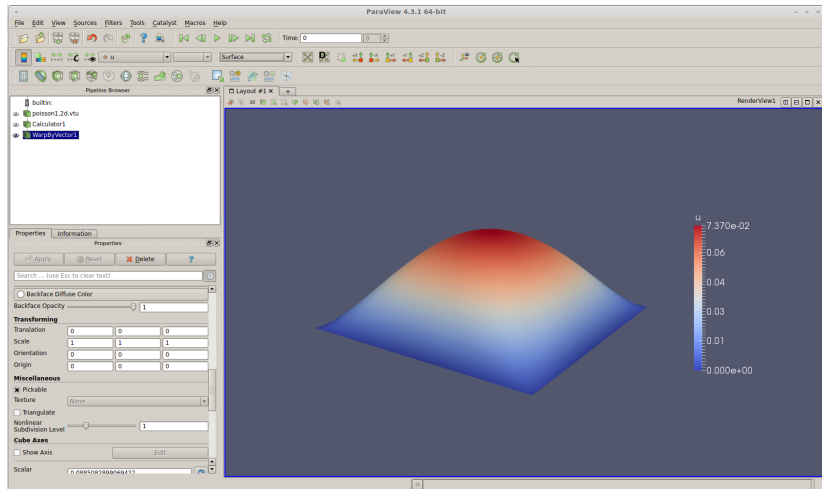
1.0 0.0...



ASCII-File, that describes dimension of the elements (DIM), dimension of the world (DIM\_OF\_WORLD), the vertex coordinates, element connectivity, element boundary numbers and a neighborhood relation.

# Visualization

Plot Solution (.vtu file), using ParaView



## Exercise1: Poisson equation

We want to solve the Poisson equation

$$-\Delta u = f(x) \quad \text{in } \Omega,$$

and (in)homogeneous Dirichlet boundary conditions on  $\partial\Omega$ .

- 1 Setup the CMake configuration:

```
> mkdir build  
> cd build  
> cmake ..
```

- 2 Compile and run the example code exercise1.cc

```
> make exercise1  
> cd ..  
> build/exercise1 init/exercise1.dat.2d
```

- 3 Visualize the file exercise1.2d.vtu created in directory output with the program ParaView.
- 4 What is the concrete expression for  $f$  and dirichlet value  $g$ ?

# Advanced Exercise1: modify the Poisson equation

- ❶ Change the expression for  $f$  and  $g$ .
- ❷ Modify the parameters in the init-file `exercise1.dat.2d`, e.g.
  - ▶ Change the nr. of global refinements
  - ▶ Set a different Finite-Element space
  - ▶ Use another linear solver

See also AMDiS-Wiki/Initfile: <https://goo.gl/Dhm9Bx>

What parameters can be changed, and where do you get an error, or a crash of the program?

- ❸ Solve the same equation in 1d or 3d, by changing the mesh, dimension of world and problem dimension in the init-file.
- ❹ Modify the macro-mesh, e.g.
  - ▶ change the boundary nr for the right edge
  - ▶ change the coordinates of the vertices

# Some hints

## 1 Used functions/classes:

```
addSOT(Operator, EXPRESSION);  
addZOT(Operator, EXPRESSION);  
// Functor class with return-type double and  
// argument-type WorldVector<double>:  
AbstractFunction<double, WorldVector<double>>;
```

## 2 Parameters to modify:

```
mesh->global refinements: INTEGER  
poisson->feSpace[0]: {P1, P2, P3, P4, P1+bubble}  
poisson->solver: {cg, gmres, direct, ...}
```

## 3 Sources for help:

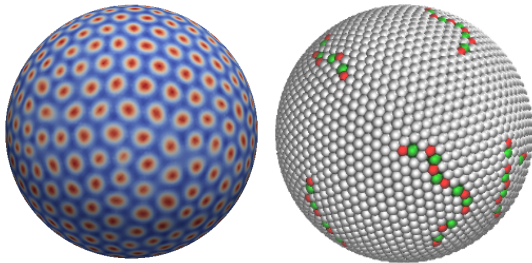
- ▶ AMDiS-Wiki: <https://goo.gl/Jy3u1u>
- ▶ Init-file manual: <https://goo.gl/Dhm9Bx>
- ▶ Expressions manual: <https://goo.gl/JK8EUI>
- ▶ List of init-file parameters: <https://goo.gl/LWJzq9>

# Session 2

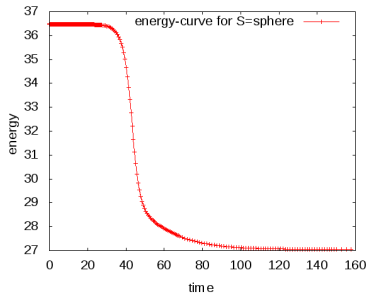


# Processing the data

Extract maxima of solution



Calculate integrals



# Working with the discrete solution

Finite-Element function  $u_h$  expressed as linear combination of basis-functions  $\phi_i$ :

$$u_h(x) = \sum_i u_i \phi_i(x),$$

with  $u_i :=$  Degrees of Freedom, stored in `DOFVector<value_type> U`:

- Evaluate at DOF index: `U[idx]`
- Iterate over all DOFs:

```
DOFIterator<value_type> it(&U [, USED_DOFs]);  
for (it.reset(); !it.end(); ++it)  
    *it = value; // idx = it.getDOFIndex()
```

- Evaluate `DOFVector` at coordinate `x`:

```
WorldVector<double> x;  
x[0] = 0.2; x[1] = 0.3;  
value_type value = U(x);
```

# Working with the discrete solution

Finite-Element function  $u_h$  expressed as linear combination of basis-functions  $\phi_i$ :

$$u_h(x) = \sum_i u_i \phi_i(x),$$

with  $u_i :=$  Degrees of Freedom, stored in `DOFVector<value_type> U`:

- Evaluate at DOF index: `U[idx]`
- Iterate over all DOFs:

```
DOFIterator<value_type> it(&U [, USED_DOFS]);  
for (it.reset(); !it.end(); ++it)  
    *it = value; // idx = it.getDOFIndex()
```

- Evaluate `DOFVector` at coordinate `x`:

```
WorldVector<double> x;  
x[0] = 0.2; x[1] = 0.3;  
value_type value = U(x);
```

## Fill a DOFVector

```
U << EXPRESSION;
```

where EXPRESSION can contain e.g.

- The coordinates: `X()`, `X(i)`
- Scalar values: `1.0`, `constant(1.0)`
- Matrix and vector expressions: `two_norm(...)`, `vec * vec`
- Other DOFVectors: `valueOf(V)`, `gradientOf(V)`

Expressions manual: <https://goo.gl/JK8EUI>

**Example:** Let  $C$  and  $U$  be of type `DOFVector<double>`:

$$U := \frac{1}{2} \left( C^2 (1 - C)^2 + \frac{1}{\epsilon} \|\nabla C\|^2 \right)$$

Assign expression on rhs to DOFVector  $U$ :

```
U << 0.5*( pow<2>(valueOf(C) * (1.0 - valueOf(C))  
           + (1.0/eps) * unary_dot(gradientOf(C)) );
```

## Fill a DOFVector

```
U << EXPRESSION;
```

where EXPRESSION can contain e.g.

- The coordinates:  $X()$ ,  $X(i)$
- Scalar values:  $1.0$ ,  $\text{constant}(1.0)$
- Matrix and vector expressions:  $\text{two\_norm}(\dots)$ ,  $\text{vec} * \text{vec}$
- Other DOFVectors:  $\text{valueOf}(V)$ ,  $\text{gradientOf}(V)$

Expressions manual: <https://goo.gl/JK8EUI>

**Example:** Let  $C$  and  $U$  be of type `DOFVector<double>`:

$$U(x) := \frac{1}{2} \left( C(x)^2 (1 - C(x))^2 + \frac{1}{\epsilon} \|\nabla C(x)\|^2 \right)$$

Assign expression on rhs to DOFVector  $U$ :

```
U << 0.5*( pow<2>(valueOf(C) * (1.0 - valueOf(C))  
          + (1.0/eps) * unary_dot(gradientOf(C)) );
```

# Working with the discrete solution

- Reduce the DOFVector, i.e. calc integrals, norms:

```
integrate( EXPRESSION [, Mesh*] )
```

```
integrate( valueOf(U) );  $\Rightarrow \int_{\Omega} u \, dx$ 
```

```
integrate( two_norm(gradientOf(U)) );  $\Rightarrow \int_{\Omega} \|\nabla u\|_2 \, dx$ 
```

```
integrate(pow<2>(valueOf(U)) + unary_dot(gradientOf(U)));  
 $\Rightarrow \|u\|_{H^1(\Omega)}^2$ 
```

- Other reduction operations (over DOFs):

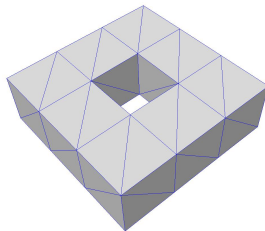
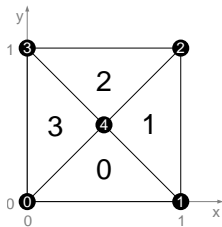
- ▶ `max( EXPRESSION )`  $\Rightarrow \max\{expr(x_i) : x_i \in \Omega\}$

- ▶ `min( EXPRESSION )`  $\Rightarrow \min\{expr(x_i) : x_i \in \Omega\}$

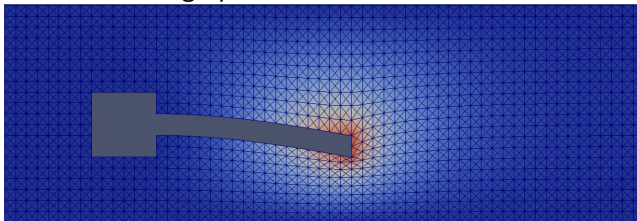
- ▶ `abs_max( EXPRESSION )`  $\Rightarrow \max\{|expr(x_i)| : x_i \in \Omega\}$

- ▶ `abs_min( EXPRESSION )`  $\Rightarrow \min\{|expr(x_i)| : x_i \in \Omega\}$

# Working with the mesh



Change position of mesh vertices:



## Working with the mesh

A Mesh holds all information about a triangulation. Get the mesh from ProblemStat:

```
Mesh& mesh = *prob.getMesh();
```

- Number of vertices: `mesh.getNumberOfVertices()`
- Number of triangles (elements): `mesh.getNumberOfElements()`
- World coordinates are stored in `WorldVector<double>`.
- Get the coordinates of all DOFs:

```
DOFVector<WorldVector<double> >  
  Coords(prob.getFeSpace(), "coords");  
mesh.getDofCoords(Coords);
```

- Change coordinates of the mesh:

```
ParametricSimple parametric(Coords);  
mesh.setParametric(&parametric);
```



# Working with the mesh:

- (Advanced) Traverse the mesh element-wise:

```
Flag traverseFlag = Mesh::CALL_LEAF_EL | (FILL_FLAGS);
TraverseStack stack;
ElInfo *elInfo = stack.traverseFirst(&mesh, -1, traverseFlag);
while (elInfo) {
    // do something with elInfo
    elInfo = stack.traverseNext(elInfo);
}
```

- (Advanced) Global Refinement of the mesh: Use RefinementManager

```
RefinementManager* refManager = prob.getRefinementManager();

Flag f = refManager->globalRefine(&mesh, 5);
if (f == MESH_REFINED) {
    MSG("Mesh globally refined by 5 levels!\n");
}
```

## Example: Print solution with coordinates

Writing out the solution to the screen:

```
DOFVector<WorldVector<double>> C(U.getFeSpace(), "c");
mesh.getDofCoords(C);

DOFIterator<double> it(U);
DOFIterator<WorldVector<double>> c_it(C);
for (it.reset(), c_it.reset(); !it.end(); ++it, ++c_it)
    std::cout << "U(" << *c_it << ") = " << *it << "\n";
```

This will print out:

```
U(0 0) = 0
U(1 0) = 0
U(1 1) = 0
U(0 1) = 0
U(0.5 0.5) = 0.0833333
```

# Input/Output

Write DOFVectors to file, for visualization, serialization, ...

- File writer with automatic file-type detection:

```
io::writeFile(DOFVECTOR, FILENAME);
```

Detection by filename extension: ".vtu", ".arh", ".dat", ".2d", ".gnu", ...

- Use **ParaView** for visualization
- Tool **MeshConv** can convert between mesh formats
- File reader with automatic file-type detection:  

```
io::readFile(FILENAME, DOFVECTOR);
```
- Use ARH format to exchange data (can be converted to VTU by MeshConv.)

## Exercise2: Poisson equation

We want to solve the Poisson equation

$$-\Delta u = f(x) \quad \text{in } \Omega, \quad u|_{\partial\Omega} = g$$

in a rectangular domain  $\Omega$ , with

$$f(x) = 40(1 - 10\|x\|^2)e^{-10.0\|x\|^2}$$

$$g(x) := e^{-10.0\|x\|^2}$$

and with exact solution  $u^* = g$ .

- 1 Assemble and solve the equation.
- 2 Interpolate error  $ERR := |u_h - g|$  to DOFVector and write it to a file.
- 3 Calculate the error-norms  $\|u_h - g\|_{L_2(\Omega)}$  and  $\|u_h - g\|_{H_1(\Omega)}$
- 4 Evaluate the error at the grid-point  $x = (0.5, 0.5)$ .
- 5 Refine the mesh globally and compare error-norms to old error-norms.

## Advanced Exercise2: Mesh adaption

The datastructure `AdaptInfo` provides parameters for tolerance and nr. of iterations for an adaption process:

```
AdaptInfo adaptInfo("adapt");  
adaptInfo.getMaxSpaceIteration(); // => adapt->max iteration  
adaptInfo.getSpaceTolerance(0);   // => adapt[0]->tolerance
```

- 1 Use `AdaptInfo` to write an adaption loop that refines the mesh globally to reduce the error until a tolerance is reached.
- 2 Write the error `DOFVector` in every adaption iteration to a file.
- 3 Calculate the exponent in the error estimate  $\|u_h - I_h u^*\|_{\#} \leq Ch^k$
- 4 (optional) Transform the mesh, by  $x \mapsto 2x$  and solve again.

# Some hints

## 1 Used functions/classes:

```
DOFVector << EXPRESSION;  
integrate( EXPRESSION );  
  
// EXPRESSION: {valueOf(U), gradientOf(U), X(), X(i), +,-,*,/,  
//             unary_dot(EXPR.), pow<2>(EXPR.), absolute(EXPR.)}  
  
RefinementManager* refManager = prob.getRefinementManager();  
refManager->globalRefine(prob.getMesh(), NR_OF_REFINEMENTS);  
  
adaptInfo.getSpaceTolerance(0);  
adaptInfo.getMaxSpaceIteration();  
  
io::writeFile( DOFVECTOR, FILENAME );
```

## 2 Parameters to modify:

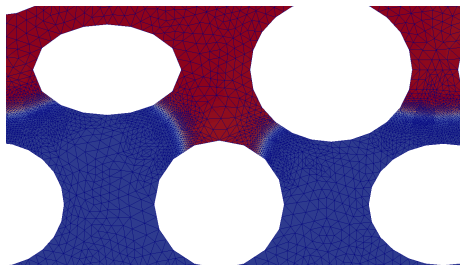
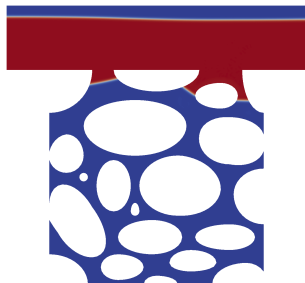
```
adapt[0]->tolerance:  DOUBLE  
adapt->max iteration:  INTEGER
```

# Session 3

# Mesh adaptivity

Motivation:

- Two-phase flow problem
- Steep transition between two phases



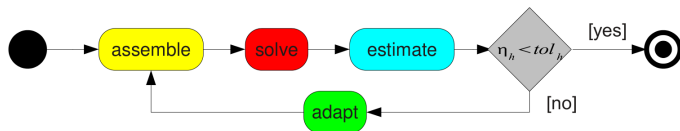
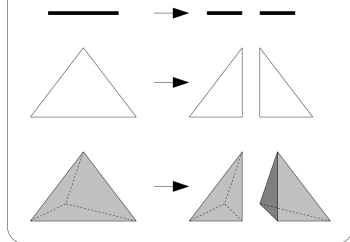


# Adaptivity and error estimators: poisson3.cc

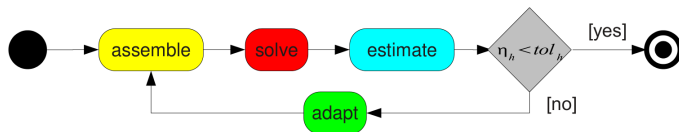
Adaptive solution strategy:

- Refine elements to reduce global error
- Local refinement instead of global refinement
- Repeated solution with adapted mesh:

**bisection of simplices:**



## Adaptivity and error estimators: poisson3.cc

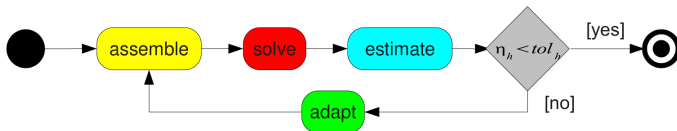


```
prob.assemble(&adaptInfo);
prob.solve(&adaptInfo);
prob.estimate(&adaptInfo); // local error indicator
```

```
Flag adapted;
for (int i = 0; i < MAX_ITER; ++i)
{
    if (adaptInfo.getEstSum(0) < TOL)
        break;
    Flag markFlag = prob.markElements(&adaptInfo); // mesh adaption
    adapted = prob.refineMesh(&adaptInfo);
    adapted |= prob.coarsenMesh(&adaptInfo);

    prob.buildAfterCoarsen(&adaptInfo, markFlag); // assemble
    prob.solve(&adaptInfo);
    prob.estimate(&adaptInfo);
}
```

## Adaptivity and error estimators: poisson3.cc



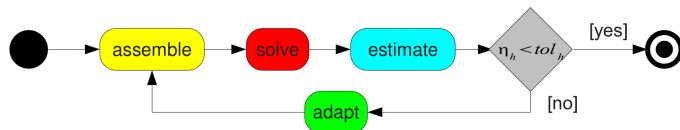
```
prob.assemble(&adaptInfo);
prob.solve(&adaptInfo);
prob.estimate(&adaptInfo); // local error indicator

Flag adapted;
for (int i = 0; i < MAX_ITER; ++i)
{
    if (adaptInfo.getEstSum(0) < TOL)
        break;
    Flag markFlag = prob.markElements(&adaptInfo); // mesh adaption
    adapted = prob.refineMesh(&adaptInfo);
    adapted|= prob.coarsenMesh(&adaptInfo);

    prob.buildAfterCoarsen(&adaptInfo, markFlag); // assemble
    prob.solve(&adaptInfo);
    prob.estimate(&adaptInfo);
}
```

# Better: AdaptStationary

# Adaptivity and error estimators: poisson3.cc



Encapsulation into class:

```
AdaptStationary adaptStat("adapt", prob, adaptInfo);  
adaptStat.adapt(); // start adaptive solution process
```

# Adaptivity and error estimators: poisson3.cc

Solution strategy: assemble  $\rightarrow$  solve  $\rightarrow$  estimate  $\rightarrow$  mark  $\rightarrow$  refine/coarsen

- Estimator type can be set in init-file
- e.g. Residual-Estimator: Element error-indicator  $\eta_T$ :

$$\eta_T := C_0 h_T^2 \|f + \nabla \cdot \mathbb{A} \nabla u_h\|_{L_2(T)} + C_1 \left( \sum_{e \in T} h_e \left[ [\mathbb{A} \nabla u_h \cdot \underline{\nu}_e] \right]^2 \right)^{1/2}$$

- e.g. Recovery Estimator:

$$\eta_T := \|\nabla u_h^* - \nabla u_h\|_{L_2(T)}$$

- Various **marking strategies**: global refinement, maximum strategy, equidistribution strategy, guaranteed error reduction strategy

# Adaptivity and error estimators: poisson3.cc

Solution strategy: assemble  $\rightarrow$  solve  $\rightarrow$  **estimate**  $\rightarrow$  mark  $\rightarrow$  refine/coarsen

- Estimator type can be set in init-file
- e.g. Residual-Estimator: Element error-indicator  $\eta_T$ :

$$\eta_T := C_0 h_T^2 \|f + \nabla \cdot \mathbb{A} \nabla u_h\|_{L_2(T)} + C_1 \left( \sum_{e \in T} h_e \left[ [\mathbb{A} \nabla u_h \cdot \underline{\nu}_e] \right]^2 \right)^{1/2}$$

- e.g. Recovery Estimator:

$$\eta_T := \|\nabla u_h^* - \nabla u_h\|_{L_2(T)}$$

- Various **marking strategies**: global refinement, maximum strategy, equidistribution strategy, guaranteed error reduction strategy

# Adaptivity and error estimators: poisson3.cc

Solution strategy: assemble  $\rightarrow$  solve  $\rightarrow$  estimate  $\rightarrow$  **mark**  $\rightarrow$  refine/coarsen

- Estimator type can be set in init-file
- e.g. Residual-Estimator: Element error-indicator  $\eta_T$ :

$$\eta_T := C_0 h_T^2 \|f + \nabla \cdot \mathbb{A} \nabla u_h\|_{L_2(T)} + C_1 \left( \sum_{e \in T} h_e \left[ [\mathbb{A} \nabla u_h \cdot \underline{\nu}_e] \right]^2 \right)^{1/2}$$

- e.g. Recovery Estimator:

$$\eta_T := \|\nabla u_h^* - \nabla u_h\|_{L_2(T)}$$

- Various **marking strategies**: global refinement, maximum strategy, equidistribution strategy, guaranteed error reduction strategy

# Stationary equation with adaptivity

Source-code: see "poisson4.cc"

```
ProblemStat prob("poisson5");
prob.initialize(INIT_ALL);

Operator opL( prob.getFeSpace(), prob.getFeSpace() );
    addSOT( opL, A ); // e.g. A = 1.0
Operator opF( prob.getFeSpace() );
    addZOT( opF, f ); // e.g. f = X()*X() + 1

prob.addMatrixOperator( opL, 0, 0 );
prob.addVectorOperator( opF, 0 );

prob.addDirichletBC( nr, 0, 0, new G );

AdaptInfo adaptInfo("adapt");
AdaptStationary("adapt", prob, adaptInfo).adapt();

prob.writeFiles(&adaptInfo, true);
```



# Stationary equation with adaptivity

Init-file: see "poisson4.dat.2d"

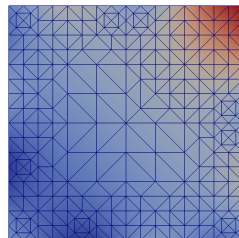
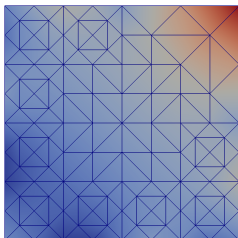
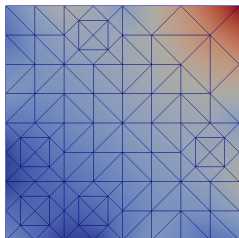
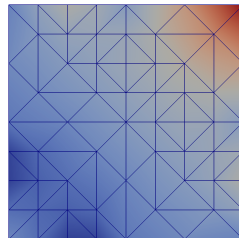
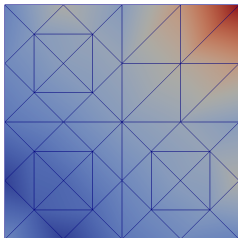
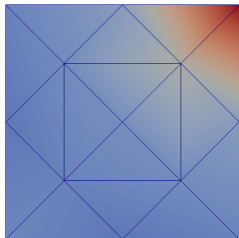
```
poisson->estimator[0]: residual % or recovery
poisson->estimator[0]->C0: 0.1 % weight of element residual
poisson->estimator[0]->C1: 0.1 % weight of jump residual

poisson->marker[0]->strategy: 2
% 1... global refinement
% 2... maximum strategy
% 3... equidistribution strategy
% 4... guaranteed error reduction strategy

adapt[0]->tolerance: 1e-4
adapt->max iteration: 5
```

# Stationary equation with adaptivity

Visualization of the locally refined grid:

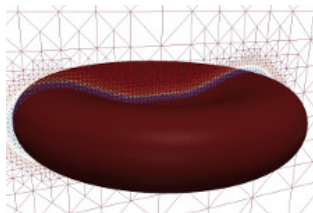
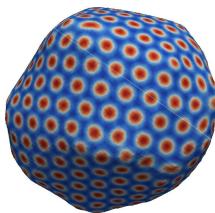
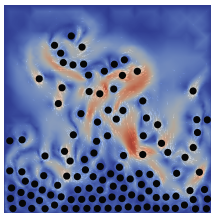


# Lesson 3b

# System of equations

Motivation:

- Navier-Stokes equations
- Phase-Field (Crystal) equation
- Willmore flow



## Example: Biharmonic equation

We consider the stream-function formulation for the Stokes equation (2D):

$$-\nu \Delta^2 \psi = f, \quad \text{in } \Omega, \quad (\mathbf{v} = -\nabla \times \mathbf{e}_z \psi)$$

with  $\psi|_{\partial\Omega} = \Delta\psi|_{\partial\Omega} = 0$  and  $f = (\nabla \times \mathbf{f})_z$ .

## Example: Biharmonic equation

We consider the stream-function formulation for the Stokes equation (2D):

$$-\nu \Delta^2 \psi = f, \quad \text{in } \Omega, \quad (\mathbf{v} = -\nabla \times \mathbf{e}_z \psi)$$

with  $\psi|_{\partial\Omega} = \Delta\psi|_{\partial\Omega} = 0$  and  $f = (\nabla \times \mathbf{f})_z$ . Reformulated as system of second order equation, we obtain:

$$\begin{aligned} -\nu \Delta \phi &= f, & \text{in } \Omega \\ -\Delta \psi + \phi &= 0, \end{aligned}$$

or in weak form:

$$\begin{aligned} \langle \nu \nabla \phi, \nabla \theta \rangle - \langle \nu \partial_n \phi, \theta \rangle_{\partial\Omega} &= \langle f, \theta \rangle, \quad \forall \theta \in V \\ \langle \nabla \psi, \nabla \theta' \rangle - \langle \partial_n \psi, \theta' \rangle_{\partial\Omega} + \langle \phi, \theta' \rangle &= 0, \quad \forall \theta' \in V' \end{aligned}$$

with solution components  $(\psi \in V, \phi \in V')$ .

## Example: Biharmonic equation

We consider the stream-function formulation for the Stokes equation (2D):

$$-\nu \Delta^2 \psi = f, \quad \text{in } \Omega, \quad (\mathbf{v} = -\nabla \times \mathbf{e}_z \psi)$$

with  $\psi|_{\partial\Omega} = \Delta\psi|_{\partial\Omega} = 0$  and  $f = (\nabla \times \mathbf{f})_z$ . Reformulated as system of second order equation, we obtain:

$$\begin{aligned} -\nu \Delta \phi &= f, & \text{in } \Omega \\ -\Delta \psi + \phi &= 0, & \text{with } \psi|_{\partial\Omega} = 0 \text{ and } \phi|_{\partial\Omega} = 0 \end{aligned}$$

or in weak form:

$$\begin{aligned} \langle \nu \nabla \phi, \nabla \theta \rangle - \cancel{\langle \nu \partial_n \phi, \theta \rangle_{\partial\Omega}} &= \langle f, \theta \rangle, \quad \forall \theta \in V_0 \\ \langle \nabla \psi, \nabla \theta' \rangle - \cancel{\langle \partial_n \psi, \theta' \rangle_{\partial\Omega}} + \langle \phi, \theta' \rangle &= 0, \quad \forall \theta' \in V'_0 \end{aligned}$$

with solution components  $(\psi \in V_0, \phi \in V'_0)$ .

## Example: Biharmonic equation

We consider the stream-function formulation for the Stokes equation (2D):

$$-\nu \Delta^2 \psi = f, \quad \text{in } \Omega, \quad (\mathbf{v} = -\nabla \times \mathbf{e}_z \psi)$$

with  $\psi|_{\partial\Omega} = \Delta\psi|_{\partial\Omega} = 0$  and  $f = (\nabla \times \mathbf{f})_z$ . Reformulated as system of second order equation, we obtain:

$$\begin{aligned} -\nu \Delta \phi &= f, & \text{in } \Omega \\ -\Delta \psi + \phi &= 0, & \text{with } \psi|_{\partial\Omega} = 0 \text{ and } \phi|_{\partial\Omega} = 0 \end{aligned}$$

or in weak form:

$$\begin{aligned} \underbrace{\langle \nu \nabla \phi, \nabla \theta \rangle}_{\text{SOT}} &= \underbrace{\langle f, \theta \rangle}_{\text{ZOT}}, & \forall \theta \in V_0 \\ \underbrace{\langle \nabla \psi, \nabla \theta' \rangle}_{\text{SOT}} + \underbrace{\langle \phi, \theta' \rangle}_{\text{ZOT}} &= 0, & \forall \theta' \in V_0' \end{aligned}$$

with solution components  $(\psi \in V_0, \phi \in V_0')$ .



## Example: Biharmonic equation

We consider the stream-function formulation for the Stokes equation (2D):

$$-\nu \Delta^2 \psi = f, \quad \text{in } \Omega, \quad (\mathbf{v} = -\nabla \times \mathbf{e}_z \psi)$$

with  $\psi|_{\partial\Omega} = \Delta\psi|_{\partial\Omega} = 0$  and  $f = (\nabla \times \mathbf{f})_z$ . Reformulated as system of second order equation, we obtain:

$$\begin{aligned} -\nu \Delta \phi &= f, & \text{in } \Omega \\ -\Delta \psi + \phi &= 0, & \text{with } \psi|_{\partial\Omega} = 0 \text{ and } \phi|_{\partial\Omega} = 0 \end{aligned}$$

or in weak form:

$$\begin{aligned} \overbrace{\langle \nu \nabla \phi, \nabla \theta \rangle}^{\text{SOT}(0,1)} &= \overbrace{\langle f, \theta \rangle}^{\text{ZOT}(0)}, \quad \forall \theta \in V_0 \\ \underbrace{\langle \nabla \psi, \nabla \theta' \rangle}_{\text{SOT}(1,0)} + \underbrace{\langle \phi, \theta' \rangle}_{\text{ZOT}(1,1)} &= 0, \quad \forall \theta' \in V'_0 \end{aligned}$$

with solution components  $(\psi \in V_0, \phi \in V'_0)$ .

# The biharmonic equation

Source-code: see "biharmonic1.cc"

$$\underbrace{\langle \nu \nabla \phi, \nabla \theta \rangle}_{\text{opL\_phi}} = \underbrace{\langle f, \theta \rangle}_{\text{opF}},$$
$$\underbrace{\langle \nabla \psi, \nabla \theta' \rangle}_{\text{opL\_psi}} + \underbrace{\langle \phi, \theta' \rangle}_{\text{opM\_phi}} = 0$$

```
Operator opL_phi(prob.getFeSpace(0), prob.getFeSpace(1));
    addSOT(opL_phi, nu);
Operator opF(prob.getFeSpace(0));
    addZOT(opF, f);
Operator opL_psi(prob.getFeSpace(1), prob.getFeSpace(0));
    addSOT(opL_psi, 1.0);
Operator opM_phi(prob.getFeSpace(1), prob.getFeSpace(1));
    addZOT(opM_phi, 1.0);

// ===== add operators to problem =====
prob.addMatrixOperator(opL_phi, 0, 1);    // -nu*laplace(phi)
prob.addMatrixOperator(opL_psi, 1, 0);    // -laplace(psi)
prob.addMatrixOperator(opM_phi, 1, 1);    // phi
prob.addVectorOperator(opF, 0);           // f(x)
/// ...
```

# The biharmonic equation

Source-code: see "biharmonic1.cc"

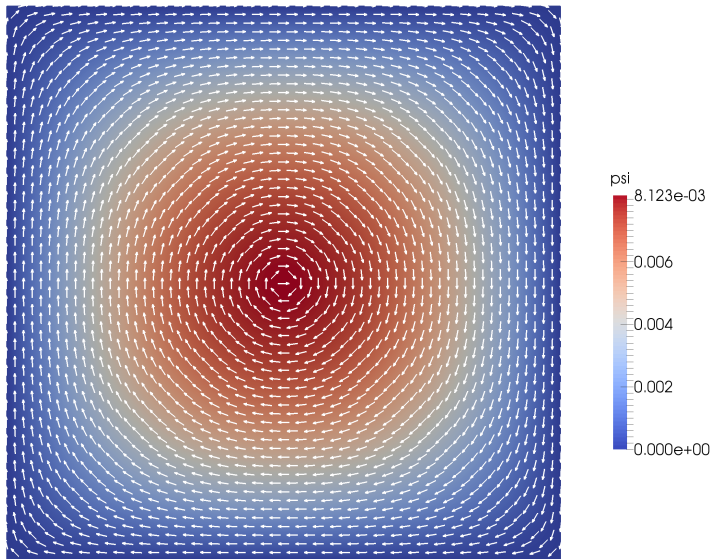
```
/// ...  
BoundaryType nr = 1;  
prob.addDirichletBC(nr, 1, 0, new Constant(0.0));  
prob.addDirichletBC(nr, 0, 1, new Constant(0.0));  
  
AdaptInfo adaptInfo("adapt");  
AdaptStationary("adapt", prob, adaptInfo).adapt();  
  
prob.writeFiles(&adaptInfo, true);
```

Init-file: see "biharmonic1.dat.2d"

```
biharmonic->mesh:      mesh  
biharmonic->components: 2  
biharmonic->feSpace[0]: P1  
biharmonic->feSpace[1]: P1  
biharmonic->dim:      2  
biharmonic->name:      [psi,phi]
```

# The biharmonic equation

Velocity field  $\mathbf{v}$  and stream function  $\psi$  for  $\mathbf{f} = (y, -x)^\top$



## Exercise3: System of equations

Implement the 4th order PDE

$$\begin{aligned}u - \Delta(u - \epsilon \Delta u) &= f(x) \quad \text{in } \Omega, \\ \partial_n u|_{\partial\Omega} &= \partial_n(u - \epsilon \Delta u)|_{\partial\Omega} = 0\end{aligned}$$

in a rectangular domain  $\Omega$ , with  $\epsilon < 1$  and  $f(x) \in [-1, 1]$  (random values).

- 1 Formulate Splitting of equation in system of equations.
- 2 Assemble and solve the system.
- 3 Vary the value  $\epsilon$  from 1 to  $10^{-3}$ . Introduce a parameter and read it from init-file. Change the mesh resolution accordingly.

## Advanced Exercise3: Error estimators

- 1 Modify exercise 1 to allow local mesh adaption by an error estimator.
- 2 Change the init-file correspondingly, i.e. add an entry for estimator and marker.
- 3 Choose a residual estimator/ recovery estimator and vary the estimator parameters
- 4 Vary the marking strategies and visualize the effect.

# Some hints

## ① Used functions/classes:

```
// functor that returns random values in mean +/- amplitude/2
Random(mean, amplitude);

EXPRESSION: eval(&functor)
```

## ② Parameters to modify:

```
poisson->estimator[0]: residual / recovery
poisson->marker[0]->strategy: 1/2/3/4
```

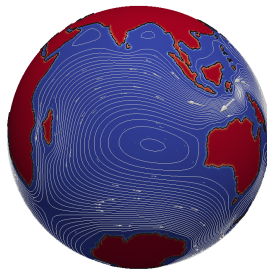
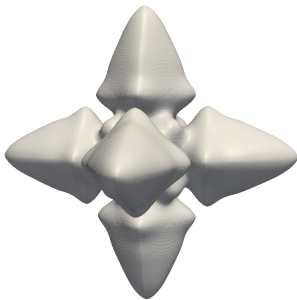
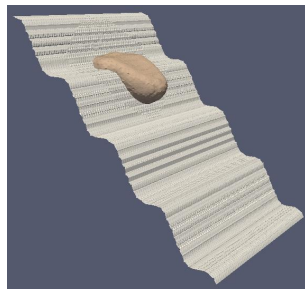
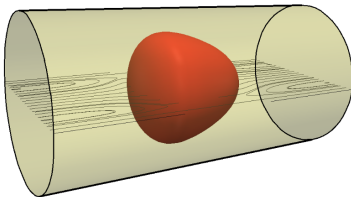
## ③ References:

- ▶ ALBERT manual, Section 1.5.2 (marker strategies)

# Session 4



# Time-dependent problems



# Time-dependent problems

Consider the instationary heat equation

$$\begin{aligned}\partial_t u - \alpha \Delta u &= f, & \text{in } \Omega \times (0, T], \\ \alpha \partial_n u|_{\partial\Omega} &= g\end{aligned}$$

with  $u$ : temperature,  $\alpha > 0$ : thermal diffusivity and  $f$ : heat source.

- Transform equation in a sequence of stationary equations: Let  $0 = t_0 < t_1 < \dots < t_N = T$  and  $\tau_k = t_{k+1} - t_k$ ,  $u_k \equiv u(t_k)$ : For  $k = 0, 1, \dots, N-1$  find  $u_{k+1} \in V$ , s.t.
- Coefficients, like  $\tau_k, \alpha, f, g$  may change over time.

# Time-dependent problems

Consider the instationary heat equation

$$\begin{aligned}\partial_t u - \alpha \Delta u &= f, & \text{in } \Omega \times (0, T], \\ \alpha \partial_n u|_{\partial\Omega} &= g\end{aligned}$$

with  $u$ : temperature,  $\alpha > 0$ : thermal diffusivity and  $f$ : heat source.

- Transform equation in a sequence of stationary equations: Let  $0 = t_0 < t_1 < \dots < t_N = T$  and  $\tau_k = t_{k+1} - t_k$ ,  $u_k \equiv u(t_k)$ : For  $k = 0, 1, \dots, N-1$  find  $u_{k+1} \in V$ , s.t.

$$\frac{1}{\tau_k} u_{k+1} - \alpha \Delta u_{k+1} = \frac{1}{\tau_k} u_k + f(t_{k+1}), \quad \text{in } \Omega$$

- Coefficients, like  $\tau_k, \alpha, f, g$  may change over time.

# Time-dependent problems

Consider the instationary heat equation

$$\begin{aligned}\partial_t u - \alpha \Delta u &= f, & \text{in } \Omega \times (0, T], \\ \alpha \partial_n u|_{\partial\Omega} &= g\end{aligned}$$

with  $u$ : temperature,  $\alpha > 0$ : thermal diffusivity and  $f$ : heat source.

- Transform equation in a sequence of stationary equations: Let  $0 = t_0 < t_1 < \dots < t_N = T$  and  $\tau_k = t_{k+1} - t_k$ ,  $u_k \equiv u(t_k)$ : For  $k = 0, 1, \dots, N-1$  find  $u_{k+1} \in V$ , s.t.

$$\begin{aligned}\left\langle \frac{1}{\tau_k} u_{k+1}, \theta \right\rangle + \langle \alpha \nabla u_{k+1}, \nabla \theta \rangle \\ = \left\langle \frac{1}{\tau_k} u_k + f(t_{k+1}), \theta \right\rangle + \langle g(t_{k+1}), \theta \rangle_{\partial\Omega}, \quad \forall \theta \in V\end{aligned}$$

- Coefficients, like  $\tau_k, \alpha, f, g$  may change over time.

# Time-dependent problems

Consider the instationary heat equation

$$\begin{aligned}\partial_t u - \alpha \Delta u &= f, & \text{in } \Omega \times (0, T], \\ \alpha \partial_n u|_{\partial\Omega} &= g\end{aligned}$$

with  $u$ : temperature,  $\alpha > 0$ : thermal diffusivity and  $f$ : heat source.

- Transform equation in a sequence of stationary equations: Let  $0 = t_0 < t_1 < \dots < t_N = T$  and  $\tau_k = t_{k+1} - t_k$ ,  $u_k \equiv u(t_k)$ : For  $k = 0, 1, \dots, N-1$  find  $u_{k+1} \in V$ , s.t.

$$\begin{aligned}\overbrace{\left\langle \frac{1}{\tau_k} u_{k+1}, \theta \right\rangle}^{\text{ZOT}} + \overbrace{\langle \alpha \nabla u_{k+1}, \nabla \theta \rangle}^{\text{SOT}} \\ = \underbrace{\left\langle \frac{1}{\tau_k} u_k + f(t_{k+1}), \theta \right\rangle}_{\text{ZOT}} + \underbrace{\langle g(t_{k+1}), \theta \rangle_{\partial\Omega}}_{\text{ZOT}_{\partial\Omega}}, \quad \forall \theta \in V\end{aligned}$$

- Coefficients, like  $\tau_k, \alpha, f, g$  may change over time.

# Time-dependent problems: heat1.cc

Formulate the stationary equation, that is solved in each timestep:

```
FiniteElemSpace const* V = prob.getFeSpace(0);
DOFVector<double>* u =      prob.getSolution(0);

Operator opL(V, V);
    addZOT(opL, 1.0/var(tau) );
    addSOT(opL, A ); // e.g. A = 1.0

Operator opF(V);
    addZOT(opF, valueOf(u)/var(tau) );
    addZOT(opF, f ); // e.g. f = X()*X() + 1

Operator opG(V);
    addZOT(opG, g ); // e.g. g = X(0)+X(1)

prob.addMatrixOperator( opL, 0, 0 );
prob.addVectorOperator( opF, 0 );

prob.addBoundaryVectorOperator( nr, opG, 0 ); // <g, theta>_B
```

## Time-dependent problems: heat1.cc

Formulate the stationary equation, that is solved in each timestep:

```
FiniteElemSpace  const* V = prob.getFeSpace(0);
DOFVector<double>* u =      prob.getSolution(0);

Operator opL(V, V);
    addZOT(opL, 1.0/var(tau) );
    addSOT(opL, A ); // e.g. A = 1.0

Operator opF(V);
    addZOT(opF, valueOf(u)/var(tau) );
    addZOT(opF, f ); // e.g. f = X()*X() + 1

Operator opG(V);
    addZOT(opG, g ); // e.g. g = X(0)+X(1)

prob.addMatrixOperator( opL, 0, 0 );
prob.addVectorOperator( opF, 0 );

prob.addBoundaryVectorOperator( nr, opG, 0 ); // <g, theta>_B
```

Get timestep width tau from AdaptInfo:

```
double* tau = adaptInfo.getTimestepPtr();
```

# Time-dependent problems: heat1.cc

## Sequence of timesteps:

```
// ===== set initial solution =====  
u << constant(0.0); // u(x,t=0) := 0  
  
while (adaptInfo.getTime() < adaptInfo.getEndTime()) {  
    adaptInfo.setTime(adaptInfo.getTime()  
                      + adaptInfo.getTimestep()); // t += tau  
  
    // ===== assemble and solve linear system =====  
    prob.assemble(&adaptInfo);  
    prob.solve(&adaptInfo);  
  
    // ===== write solution to file =====  
    prob.writeFiles(&adaptInfo, false);  
}
```



# Time-dependent problems: heat1.cc

Sequence of timesteps:

```
// ===== set initial solution =====  
u << constant(0.0); // u(x,t=0) := 0  
  
while (adaptInfo.getTime() < adaptInfo.getEndTime()) {  
    adaptInfo.setTime(adaptInfo.getTime()  
                      + adaptInfo.getTimestep()); // t += tau  
  
    // ===== assemble and solve linear system =====  
    prob.assemble(&adaptInfo);  
    prob.solve(&adaptInfo);  
  
    // ===== write solution to file =====  
    prob.writeFiles(&adaptInfo, false);  
}
```

Better: AdaptInstationary

# Time-dependent problems: heat2.cc

Sequence of timesteps: encapsulation into class:

```
ProblemInstat instat("heat", prob);  
instat.initialize(INIT_NOTHING); // init (vector of) old solutions  
  
// start adaptive solution process  
AdaptInstationary("adapt", prob, adaptInfo, instat, adaptInfo)  
    .adapt();
```

# Nonlinear problems

- Consider the nonlinear Poisson equation

$$\begin{aligned} L(u, x) &:= -\Delta u + u^4 - f(x) = 0, & \text{in } \Omega, \\ u|_{\partial\Omega} &= 0 \end{aligned}$$

- Newton-Method: provide Jacobian  $J_u L$

$$J_u L(u, x)[\delta u] = -\Delta \delta u + 4u^3 \delta u, \quad \delta u|_{\partial\Omega} = 0$$

- Then we can write the iterative process as

$$\begin{aligned} J_u L(u_k, x)[\delta u] &= -L(u_k, x), \\ u_{k+1} &= u_k + \delta u \end{aligned}$$

for initial solution  $u_0$ . Inserting Jacobian  $J$  and objective fct.  $L$ :

$$\begin{aligned} -\Delta \delta u + 4u_k^3 \delta u &= \Delta u_k - u_k^4 + f(x) \\ \delta u|_{\partial\Omega} &= -u_k|_{\partial\Omega} \end{aligned}$$

# Nonlinear problems

- Consider the nonlinear Poisson equation

$$L(u, x) := -\Delta u + u^4 - f(x) = 0, \quad \text{in } \Omega, \\ u|_{\partial\Omega} = 0$$

- Newton-Method: provide Jacobian  $J_u L$

$$J_u L(u, x)[\delta u] = -\Delta \delta u + 4u^3 \delta u, \quad \delta u|_{\partial\Omega} = 0$$

- Then we can write the iterative process as

$$J_u L(u_k, x)[\delta u] = -L(u_k, x), \\ u_{k+1} = u_k + \delta u$$

for initial solution  $u_0$ . Inserting Jacobian  $J$  and objective fct.  $L$ :

$$-\Delta \delta u + 4u_k^3 \delta u = \Delta u_k - u_k^4 + f(x) \\ \delta u|_{\partial\Omega} = -u_k|_{\partial\Omega}$$

# Nonlinear problems

- Consider the nonlinear Poisson equation

$$\begin{aligned} L(u, x) &:= -\Delta u + u^4 - f(x) = 0, & \text{in } \Omega, \\ u|_{\partial\Omega} &= 0 \end{aligned}$$

- Newton-Method: provide Jacobian  $J_u L$

$$J_u L(u, x)[\delta u] = -\Delta \delta u + 4u^3 \delta u, \quad \delta u|_{\partial\Omega} = 0$$

- Then we can write the iterative process as

$$\begin{aligned} J_u L(u_k, x)[\delta u] &= -L(u_k, x), \\ u_{k+1} &= u_k + \delta u \end{aligned}$$

for initial solution  $u_0$ . Inserting Jacobian  $J$  and objective fct.  $L$ :

$$\begin{aligned} -\Delta \delta u + 4u_k^3 \delta u &= \Delta u_k - u_k^4 + f(x) \\ \delta u|_{\partial\Omega} &= -u_k|_{\partial\Omega} \end{aligned}$$

# Nonlinear problems

In weak form this can be written as

$$\langle \nabla \delta u, \nabla \theta \rangle + \langle 4u_k^3 \delta u, \theta \rangle = -\langle \nabla u_k, \nabla \theta \rangle + \langle -u_k^4 + f(x), \theta \rangle$$

Inserting  $\delta u = u_{k+1} - u_k$  we obtain:

```
Let  $u_0 := u_0$ 
```

```
FOR  $k = 0, 1, 2, \dots, M - 1$ 
```

```
  solve:
```

$$\langle \nabla u_{k+1}, \nabla \theta \rangle + \langle 4u_k^3 u_{k+1}, \theta \rangle = \langle 3u_k^4 + f(x), \theta \rangle \quad \forall \theta \in V_0, \quad u_{k+1}|_{\partial\Omega} = 0$$

```
END-FOR
```

```
set  $u := u_M$ 
```

# Nonlinear problems: nonlin1.cc

Formulate the stationary equation, that is solved in each iteration:

$$\langle \nabla u_{k+1}, \nabla \theta \rangle + \langle 4u_k^3 u_{k+1}, \theta \rangle = \langle 3u_k^4 + f(x), \theta \rangle$$

```
FiniteElemSpace const* V = prob.getFeSpace(0);
DOFVector<double>& u_k = *prob.getSolution(0);

Operator opL(V, V);
  addSOT(opL, 1.0);
  addZOT(opL, 4.0 * pow<3>(valueOf(u_k)));
prob.addMatrixOperator(opL, 0, 0);

Operator opF(V);
  addZOT(opF, 3.0 * pow<4>(valueOf(u_k)) + f );
prob.addVectorOperator(opF, 0);

prob.addDirichletBC(nr, 0, 0, new Constant(0.0));
...
```

# Nonlinear problems: nonlin1.cc

Formulate the stationary equation, that is solved in each iteration:

```
...  
// ===== start adaption loop =====  
AdaptInfo adaptInfo("adapt");  
AdaptStationary adapt("adapt", prob, adaptInfo);  
  
u_k << INITIAL_SOLUTION;  
for (int k = 0; k < MAX_ITER; k++) {  
    adaptInfo.reset();  
    adapt.adapt();  
}
```



## Nonlinear problems: nonlin1.cc

Formulate the stationary equation, that is solved in each iteration:

```
...  
// ===== start adaption loop =====  
AdaptInfo adaptInfo("adapt");  
AdaptStationary adapt("adapt", prob, adaptInfo);  
  
u_k << INITIAL_SOLUTION;  
for (int k = 0; k < MAX_ITER; k++) {  
    adaptInfo.reset();  
    adapt.adapt();  
}
```

Better: ProblemNonlin

# Nonlinear problems: nonlin2.cc

- Implement Newton-formulation directly:

$$J_u L(u_k, x)[\delta u] = -L(u_k, x)$$

- Iterative process implemented/ controlled by ProblemNonlin:

```
ProblemNonLin prob("nonlin");
prob.initialize(INIT_ALL);

FiniteElemSpace const* V = prob.getFeSpace(0);
DOFVector<double>& u_k = *prob.getSolution(0);
...
```

## Nonlinear problems: nonlin2.cc

$$-\Delta \delta u + 4u_k^3 \delta u = \Delta u_k - u_k^4 + f(x)$$

```
...
// J_u L(u_k)[du]
Operator opJL(V, V);
  addSOT(opJL, 1.0);
  addZOT(opJL, 4.0 * pow<3>(valueOf(u_k)) );
prob.addMatrixOperator(opJL, 0, 0);

// -L(u_k)
Operator opL(V);
  addSOT(opL, -1.0);
opL.setUhold(&u_k); // apply operator to DOFVector u_k
prob.addVectorOperator(opL, 0);

Operator opF(V);
  addZOT(opF, -pow<4>(valueOf(u_k)) + f );
prob.addVectorOperator(opF, 0);

prob.addDirichletBC(nr, 0, 0, new Constant(0.0));
...
```

# Nonlinear problems: nonlin2.cc

$$-\Delta \delta u + 4u_k^3 \delta u = \Delta u_k - u_k^4 + f(x)$$

```
...  
// ===== start adaption loop =====  
AdaptInfo adaptInfo("adapt");  
  
u_k << INITIAL_SOLUTION;  
prob.solve();
```

Initfile:

```
nonlin->nonlin solver:  newton % newtonArmijo  
nonlin->nonlin solver->build cycle:  1  
nonlin->nonlin solver->tolerance:  1e-3  
nonlin->nonlin solver->max iteration:  100  
...
```

## (Experimental) Nonlinear problems: nonlin3.cc

$$L(u, x) := -\Delta u + u^4 - f(x)$$

Use symbolic differentiation in Jacobian:

```
// class acts as name for the unknown:
struct _U_;
...
auto coeff = pow<4>(valueOf<_U_>(u_k)) - f;

// J_u L(u_k)[du]
Operator opJL(V, V);
  addSOT(opJL, 1.0);
  addZOT(opJL, diff<_U_>(coeff) );
prob.addMatrixOperator(opJL, 0, 0);

// -L(u_k)
Operator opL(V); opL.setUhOld(&u_k);
  addSOT(opL, -1.0);
prob.addVectorOperator(opL, 0);

Operator opF(V);
  addZOT(opF, -coeff );
prob.addVectorOperator(opF, 0);
...
```

## (Experimental) Nonlinear problems: nonlin3.cc

$$L(u, x) := -\Delta u + u^4 - f(x)$$

Use symbolic differentiation in Jacobian:

```
// class acts as name for the unknown:
struct _U_;
...
auto coeff = pow<4>(valueOf<_U_>(u_k)) - f;

std::cout << coeff << "\n";
std::cout << diff<_U_>(coeff) << "\n";
std::cout << diff<2, _U_>(coeff) << " --> "
          << simplify(diff<2, _U_>(coeff)) << "\n";
```

This prints:

```
(pow<4>(value(u)) - F(X))
([4] * pow<3>(value(u)))
([0] + ([12] * pow<2>(value(u)))) -->
([12] * pow<2>(value(u)))
```

## Exercise4: Navier-Stokes equations

Implement the 2d Navier-Stokes equations

$$\partial_t u_i + (\mathbf{u} \cdot \nabla) u_i = \nu \Delta u_i - \partial_i p, \quad i = 1, 2$$

$$\nabla \cdot \mathbf{u} = 0$$

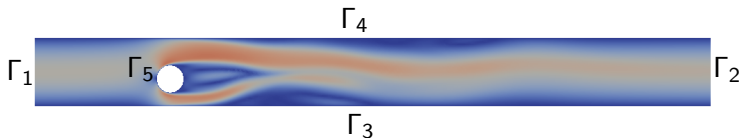
in a rectangular domain  $\Omega = [0, 10] \times [0, 1]$  with boundary conditions

$$\mathbf{u}|_{\Gamma_3} = \mathbf{u}|_{\Gamma_4} = \mathbf{u}|_{\Gamma_5} = 0,$$

$$\mathbf{u}(x_0, x_1)|_{\Gamma_1} = (4u^\infty x_1(1 - x_1), 0)^\top$$

$$\nu \partial_n \mathbf{u} = \mathbf{n} p, \quad \text{at } \Gamma_2$$

with  $\mathbf{n}$  the outer domain normal.



## Exercise4: Navier-Stokes equations

- 1 Implement a simple semi-implicit backward Euler time-discretization
- 2 Linearize the non-linear term:  $(\mathbf{u} \cdot \nabla)u_i \rightarrow (\mathbf{u}^{\text{old}} \cdot \nabla)u_i$
- 3 Take the macro-mesh `kanal.2d` from the macro directory.
- 4 Choose  $\nu = 1$ ,  $u^\infty = 1$  and solve for time  $t \in [0, 10]$  with timestep  $\tau = 0.1$ .



## Advanced Exercise4: Navier-Stokes equations

- 1 Vary the parameters  $\nu, u^\infty$  and timestep  $\tau$ , gridwidth  $h$  to see the karman vortex structures.
- 2 Change the mesh to `kana12.2d` and modify the boundary condition accordingly.
- 3 Instead of a simple linearization, use a Newton-method to solve the non-linearity.
- 4 Implement a higher-order time-discretization. Remark: Use the `old_solution` from `ProblemInstat`

# Some hints

## ① Used functions/classes:

```
int dow = Global::getGeo(WORLD); // dimension of world

// AdaptInfo needs to know the nr. of components
AdaptInfo adaptInfo("adapt", prob.getNumComponents());

// partial derivative:
addFOT(OPERATOR, EXPR, i, GRD_PHI); // < EXPR d_i(u), theta >
addFOT(OPERATOR, EXPR, i, GRD_PSI); // < EXPR*u, d_i(theta) >
```

## ② Parameters for Taylor-Hood element:

```
ns->components: 3
ns->feSpace[0]: P2
ns->feSpace[1]: P2
ns->feSpace[2]: P1
ns->name: [u0, u1, p]
```