DOLFIN

Dynamic Object-oriented Library for FINite element computation

Johan Hoffman and Anders Logg

`hoffman@math.chalmers.se` `logg@math.chalmers.se`

Chalmers Finite Element Center

Introduction

- An adaptive finite element solver for PDEs
- Written by people at the Department of Computational Mathematics (Hoffman/Logg)
- Written in C++
- Highly modularized
- DOLFIN is a solver. No grid generation. No visualisation.
- Licensed under the GNU GPL
- `http://www.phi.chalmers.se/dolfin`

GNU and the GPL

- Makes the software free for all users
- “Free” as in “free speech”, not “free beer”
- Free to modify, change, copy, redistribute
- Derived work must also use the GPL license
- Enables sharing of code
- Simplifies distribution of the program
- Linux is distributed under the GPL license
- See `http://www.gnu.org`

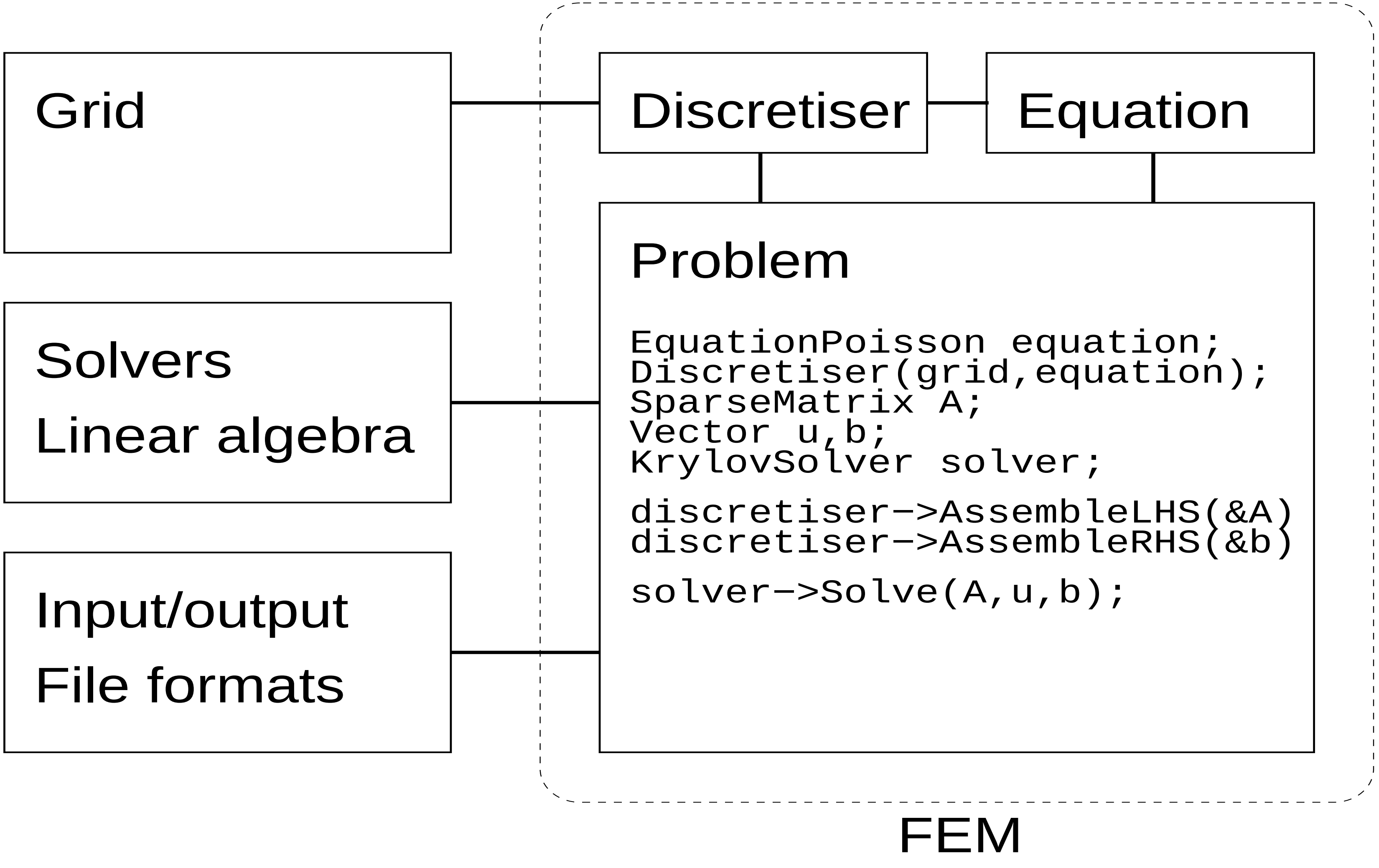
Features

- 3D or 2D
- Automatic assembling
- Tetrahedrons or triangles
- Linear elements
- Algebraic solvers: LU, GMRES, CG
- Missing: mesh refinement, adaptivity, multi-grid

DOLFIN examples

- *Start movie 1 (driven cavity, solution)*
- *Start movie 2 (driven cavity, dual)*
- *Start movie 3 (driven cavity, dual)*
- *Start movie 4 (bluff body, solution)*
- *Start movie 5 (bluff body, dual)*
- *Start movie 6 (jet, solution)*
- *Start movie 7 (transition to turbulence)*

Internal structure



The finite element

Following Ciarlet and Brenner-Scott a *finite element* $(K, \mathcal{P}, \mathcal{N})$ is defined by

1. A domain $K \subseteq \mathbb{R}^N$ with piecewise smooth boundary, typically a triangle or a tetrahedron;
2. A finite-dimensional space $\mathcal{P}$ of functions on K together with a set of basis functions (the *shape functions*);
3. A basis $\mathcal{N} = \{N_1, N_2, \ldots, N_k\}$ for the dual space $\mathcal{P}^*$ (the *degrees of freedom*).

The finite element: implementation

`FiniteElement`	The *finite element*, containing geometry and the local function space.
`FunctionSpace`	A finite-dimensional space of functions on the domain of a finite element.
`TriLinSpace`, `TetLinSpace`	Sub-classes derived from `FunctionSpace`.
`ShapeFunction`	Member of the basis for a local function space.
`TriLinFunction`, `TetLinFunction`	Sub-classes derived from `ShapeFunction`.
`LocalField`	A member of the local function space on the domain of a finite element, i.e. a linear combination of shape functions. Can also be viewed as the restriction of a `GlobalField` to the domain of a finite element.
`GlobalField`	A function (possibly vector-valued) defined on the whole of the computational domain.

Automatic assembling

- Automatic assembling is handled by *operator overloading*.
- The symmetric binary operator `'*'` is defined for the following classes:

$$
\begin{aligned}
* : \texttt{ShapeFunction} \times \texttt{ShapeFunction} &\longrightarrow \texttt{real} \\
* : \texttt{ShapeFunction} \times \texttt{real} &\longrightarrow \texttt{real} \\
* : \texttt{ShapeFunction} \times \texttt{LocalField} &\longrightarrow \texttt{real} \\
* : \texttt{LocalField} \times \texttt{LocalField} &\longrightarrow \texttt{real} \\
* : \texttt{LocalField} \times \texttt{real} &\longrightarrow \texttt{real}
\end{aligned}
$$

Automatic assembling

For two `ShapeFunctions` `v` and `w`, representing two shape functions v and w on K, the operator `'*'` is defined by

$$\texttt{v * w} = \frac{1}{|K|}\int_K v \cdot w \, dx, \tag{1}$$

$$\texttt{a * v} = \frac{1}{|K|}\int_K \alpha \cdot v \, dx, \tag{2}$$

where $|K|$ is the volume (area) of the domain K and a represents the real number α.

Automatic assembling

$$\dot{u} - \nabla \cdot (c \nabla u) = f$$

$$\int_\Omega \left(\frac{U^n - U^{n-1}}{k_n} \right) v \, dx + \int_\Omega c(\cdot, t_n) \nabla U^n \cdot \nabla v \, dx = \int_\Omega f(\cdot, t_n) v \, dx$$

```
(u*v-up*v)/k + c*(u.dx*v.dx+u.dy*v.dy+u.dz*v.dz)
```

```
f*v
```

Automatic assembling

Exact evaluation of integrals for linear elements on triangles and tetrahedrons:

$$\frac{1}{|K|}\int_K \lambda_1^{m_1}\lambda_2^{m_2}\lambda_3^{m_3}\,dx = \frac{2\cdot m_1!m_2!m_3!}{(m_1+m_2+m_3+2)!}$$

$$\texttt{a * v} = \frac{1}{|K|}\int_K \alpha\lambda_i\,dx = \alpha/3$$

$$\frac{1}{|K|}\int_K \lambda_1^{m_1}\lambda_2^{m_2}\lambda_3^{m_3}\lambda_4^{m_4}\,dx = \frac{3!\cdot m_1!m_2!m_3!m_4!}{(m_1+m_2+m_3+m_4+3)!}$$

$$\texttt{a * v} = \frac{1}{|K|}\int_K \alpha\lambda_i\,dx = \alpha/4$$

Three levels

- Simple C/C++ interface for the *user* who just wants to solve an equation with specified geometry and boundary conditions.
- New algorithms are added at *module level* by the developer or advanced user.
- Core features are added at *kernel level*.

Poisson's equation: user level

```
#include <dolfin.h>
int main(int argc, char **argv)
{
    dolfin_set_problem("poisson");

    dolfin_set_parameter("output file",     "poisson.dx");
    dolfin_set_parameter("grid file",       "tetgrid.inp");
    dolfin_set_parameter("space dimension", 2)

    dolfin_set_boundary_conditions(my_bc);
    dolfin_set_function("source",f);

    dolfin_init(argc,argv);
    dolfin_solve();
    dolfin_end();

    return 0;
}
```

Poisson's equation: module level

```
class EquationPoisson: public Equation{
public:
    EquationPoisson():Equation(3){
        AllocateFields(1);
        field[0] = &f;
    }

    real IntegrateLHS(ShapeFunction &u, ShapeFunction &v){
        return ( u.dx*v.dx + u.dy*v.dy + u.dz*v.dz );
    }

    real IntegrateRHS(ShapeFunction &v){
        return ( f * v );
    }

private:
    LocalField f;
};
```

Poisson's equation: module level

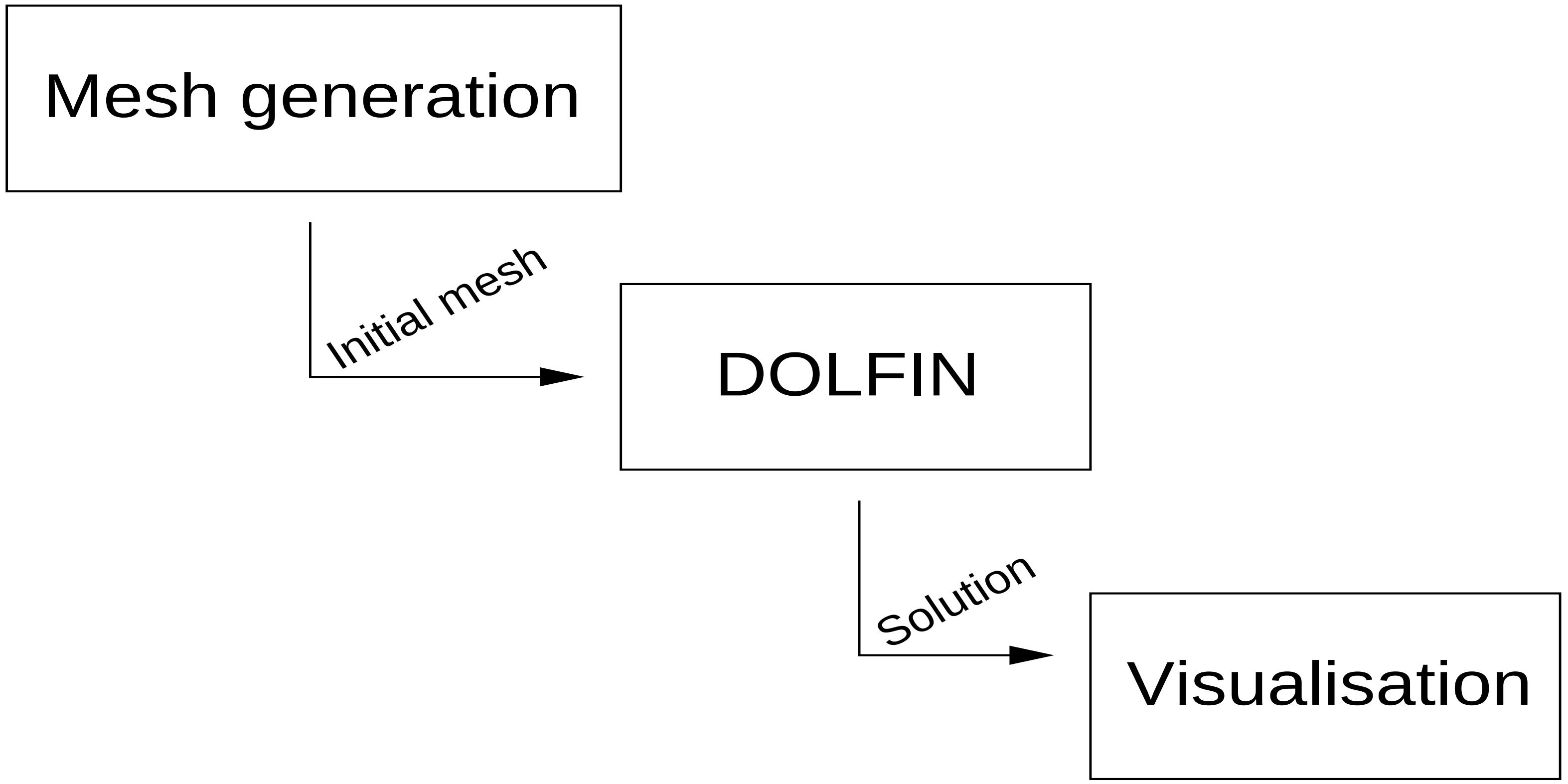
```
void ProblemPoisson::Solve()
{
    EquationPoisson equation;
    SparseMatrix A;
    Vector x,b;
    KrylovSolver solver;
    GlobalField u(grid,&x);
    GlobalField f(grid,"source");
    Discretiser discretiser(grid,equation);

    equation.AttachField(0,&f);
    discretiser.Assemble(&A,&b);
    solver.Solve(&A,&x,&b);

    u.SetLabel("u","temperature");
    u.Save();
}
```

Poisson's equation: kernel level

```
class Equation{
public:

    Equation(int nsd);
    ~Equation();

    virtual real IntegrateLHS(ShapeFunction &u, ShapeFunction &v)
    virtual real IntegrateRHS(ShapeFunction &v) = 0;

    void UpdateLHS(FiniteElement *element);
    void UpdateRHS(FiniteElement *element);
    void AttachField(int i, GlobalField *globalfield);

    ...
```

Input / output

The solver (DOLFIN) is the key part in the larger system containing also pre- and post-processing:



Input / output

- OpenDX: free open-source visualisation program based on IBM:s *Visualization Data Explorer*.
- MATLAB: commercial software (2000 Euros)
- GiD: commercial software (570 Euros)

Input / output: examples

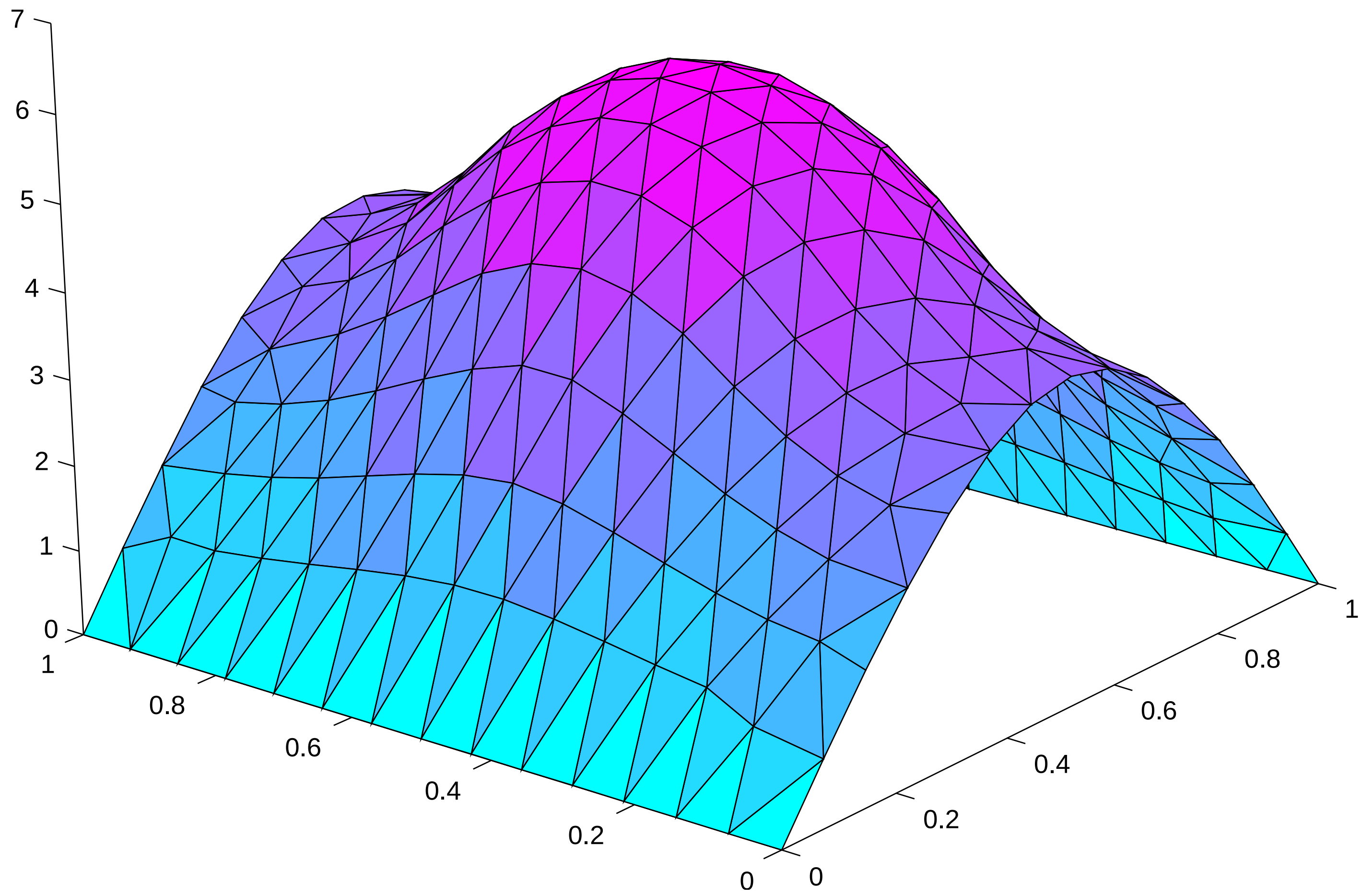
Poisson's equation:

$$-\Delta u(x) = f(x), \quad x \in \Omega, \tag{3}$$

on the unit square $\Omega = (0,1) \times (0,1)$ with the source term f localised to the middle of the domain.

Grid generation with **GiD** and visualisation using the `pdesurf` command in **MATLAB**.

Input / output: examples



Input / output: examples

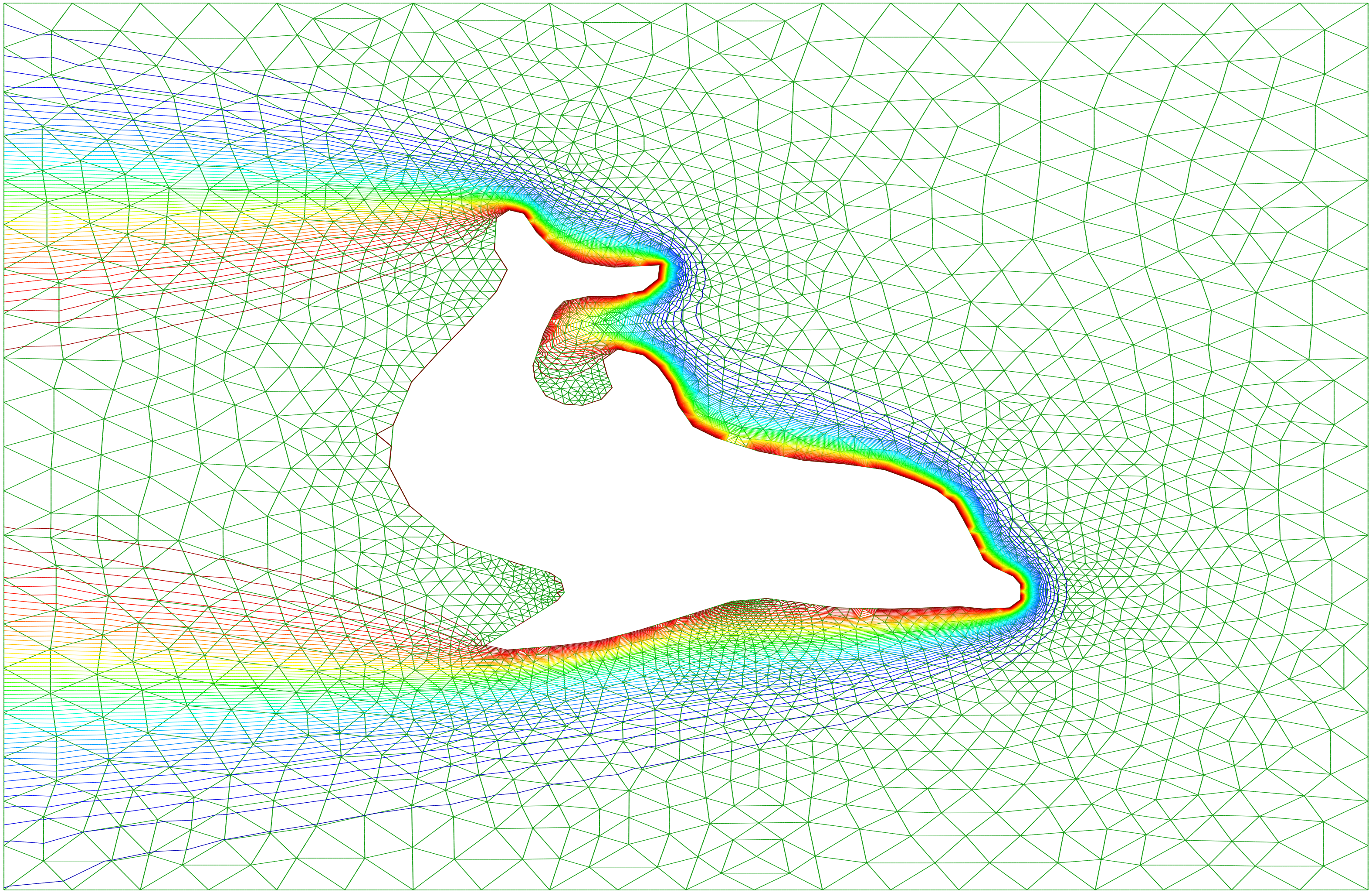
Convection–diffusion:

$$\dot{u} + b \cdot \nabla u - \nabla \cdot (\epsilon \nabla u) = f, \qquad (4)$$

with $b = (-10, 0)$, $f = 0$ and $\epsilon = 0.1$ around a warm dolphin.

Grid generation with **MATLAB** and visualisation using *contour lines* in **GiD**.

Input / output: examples



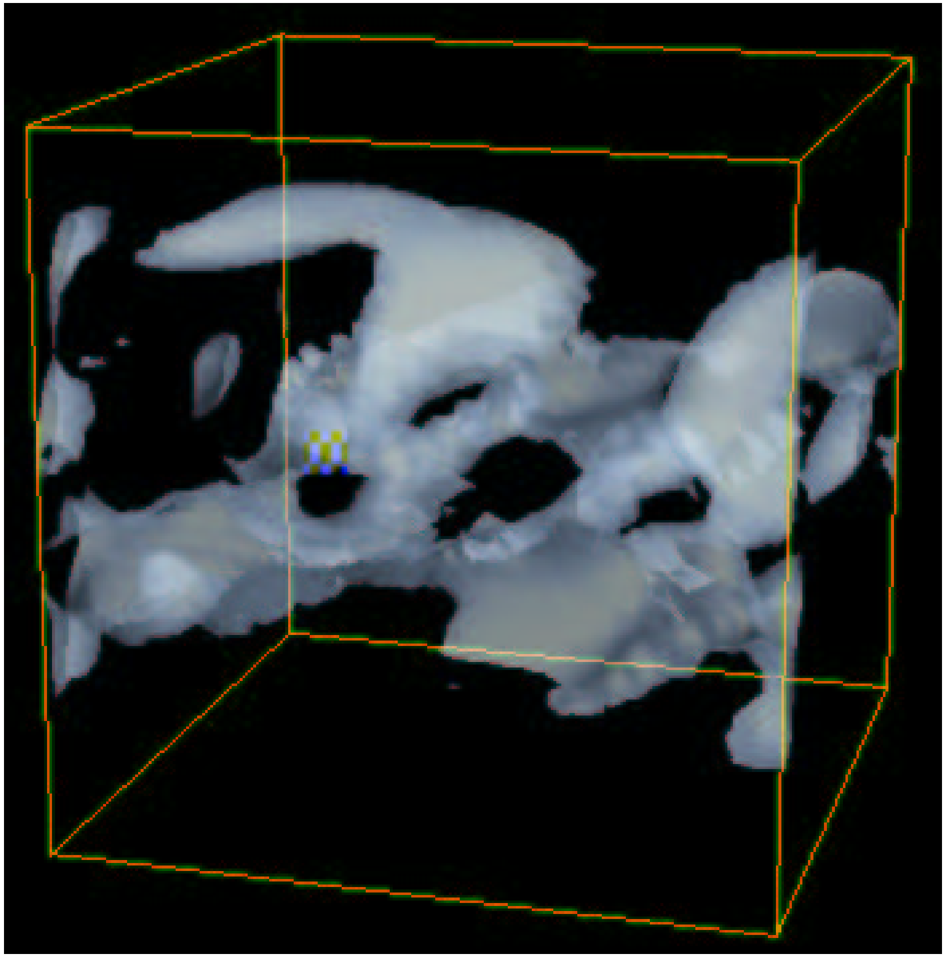
Input / output: examples

Incompressible Navier–Stokes:

$$\begin{aligned} \dot{u} + u \cdot \nabla u - \nu \Delta u + \nabla p &= f, \\ \nabla \cdot u &= 0, \end{aligned} \tag{5}$$

Visualisation in **OpenDX** of the isosurface for the velocity in a computation of transition to turbulence in shear flow on a mesh consisting of 1,600,000 tetrahedral elements.

Input / output: examples



Web page

- www.phi.chalmers.se/dolfin
- www.freshmeat.net/projects/dolfin

Organisation of the code

```
doc
src/io
src/la
src/fem
src/grid
src/init
src/test
src/utils
src/common
src/config
src/modules/navier-stokes
src/modules/poisson
src/problems/navier-stokes/benchmark
src/problems/navier-stokes/jet
src/problems/navier-stokes/...
src/problems/poisson
data/grids
```

src/io

Display

Terminal

Curses

Value

Input

Output

inp.h

opendx.h

matlab.h

gid.h

«

src/la

Vector

DenseMatrix

SparseMatrix

DirectSolver

SISolver

KrylovSolver

«

src/fem

```
Discretiser
Equation
EquationSystem
FiniteElement
FunctionSpace
GlobalField
LocalField
Problem
ShapeFunction
TetLinFunction
TetLinSpace
TriLinFunction
TriLinSpace
```

«

src/grid

Grid

Cell

CellType

Node

Point

Tetrahedron

Triangle

«

src/init

`dolfin.h`
`dolfin.C`

«

src/common

Parameter

ParameterList

Settings

Globals

«

src/modules/poisson

```
EquationPoisson
ProblemPoisson
```

«

src/problems/poisson

main.C

«

data/grids

```
tetgrid_1_1_1.inp
tetgrid_4_4_4.inp
tetgrid_8_8_8.inp
tetgrid_24_8_8.inp
...
```

«

Grid

```
class Grid{
public:
  ...
  void Init();
  void Clear();

  int GetNoNodes();
  int GetNoCells();

  Node* GetNode(int node);
  Cell* GetCell(int cell);

  void Read(const char *file);
  void Write(const char *file);
  ...
```

«

KrylovSolver

```
class KrylovSolver{
public:

  KrylovSolver();
  ~KrylovSolver(){}

  void SetMethod( KrylovMethod km );

  void Solve(Vector* x, Vector* b);
  void SolveCG(Vector* xvec, Vector* b);
  void SolveGMRES(Vector* xvec, Vector* b);

   ...
```

«

SISolver

```
class SISolver{
public:


  SISolver();
  ~SISolver(){}

  void Solve(SparseMatrix *A, Vector *x, Vector *b);

private:

  void IterateRichardson  (SparseMatrix *A, Vector *x, Vector *b);
  void IterateJacobi      (SparseMatrix *A, Vector *x, Vector *b);
  void IterateGaussSeidel (SparseMatrix *A, Vector *x, Vector *b);
  void IterateSOR         (SparseMatrix *A, Vector *x, Vector *b);
  ...
```

Vector

```
class Vector{
public:

  Vector  (int n);
  ~Vector ();

  void Add  (real a, Vector *v);
  real Dot  (Vector *v);
  real Norm ();
  ...
```

«

SparseMatrix

```
class SparseMatrix{
public:

  SparseMatrix  (int m, int n, int *ncols);
  ~SparseMatrix ();

  void Mult(Vector* x, Vector* Ax);

  ...
```

«

Equation

```
class Equation{
public:

  Equation(int noeq, int nsd);

  virtual real IntegrateLHS(TrialFunction &u, TestFunction &v) = 0;
  virtual real IntegrateRHS(TestFunction &v) = 0;

  ...
```

«

Discretiser

```
class Discretiser{
public:

  Discretiser(Grid *grid, Equation *equation);
  ˜Discretiser();

  void AssembleLHS(SparseMatrix *A);
  void AssembleRHS(Vector *b);

  ...
```

«

Problem

```
class Problem{
public:

  Problem(Grid *grid);
  ~Problem();

  virtual const char *Description() = 0;

  virtual void Solve() = 0;

  ...
```

«

ParameterList

```
class ParameterList{
public:

  ...

  void Add(const char *identifier, Type type, ...);
  void Set(const char *identifier, ...);
  void Get(const char *identifier, ...);

  void Save(const char *filename);
  void Load(const char *filename);

  ...
```

OpenDX

```
class OpenDX{
public:

  OpenDX  (const char *filename, int n, ...);
  ~OpenDX ();

  void SetLabel (int component, const char *label);
  void AddFrame (Grid *grid, Vector *u, real t);

  ...
```

«

dolfin.h

```
void dolfin_init  (int argc, char **argv);
void dolfin_end   ();
void dolfin_solve ();

void dolfin_set_problem     (const char *problem);
void dolfin_set_parameter   (const char *identifier, ...);
void dolfin_get_parameter   (const char *identifier, ...);
void dolfin_save_parameters (const char *filename);
void dolfin_load_parameters (const char *filename);

void dolfin_set_boundary_conditions
      (dolfin_bc (*bc)(real x, real y, real z, int node, int componen

void dolfin_set_function
      (const char *identifier, real (*f)(real x, real y, real z, real
```

«

main.C

```
#include <dolfin.h>

int main(int argc, char **argv)
{
  kw_set_problem("poisson");

  kw_set_parameter("problem description", "Poisson’s equation on the
  kw_set_parameter("grid file",            "../../../data/grids/tetgri
  kw_set_parameter("output file prefix",  "poisson");

  kw_init(argc,argv);
  kw_solve();
  kw_end();

  return 0;
}
```