

# Automated Solution of Differential Equations with Application to Fluid–Structure Interaction on Cut Meshes

*Presented by* Anders Logg\*  
Simula Research Laboratory, Oslo

Biomechanics Seminar, UCSD  
2012–10–24

\* Credits: <http://fenicsproject.org/about/team.html>

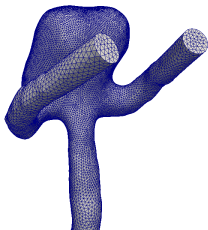


# Topics

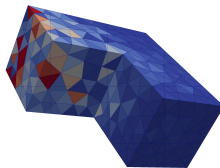
The FEniCS Project



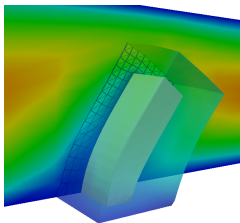
Meshing



Adaptivity



FSI



What is FEniCS?

# FEniCS is an automated programming environment for differential equations

- C++/Python library
- Initiated 2003 in Chicago
- 1000–2000 monthly downloads
- Part of Debian and Ubuntu
- Licensed under the GNU LGPL



<http://fenicsproject.org/>

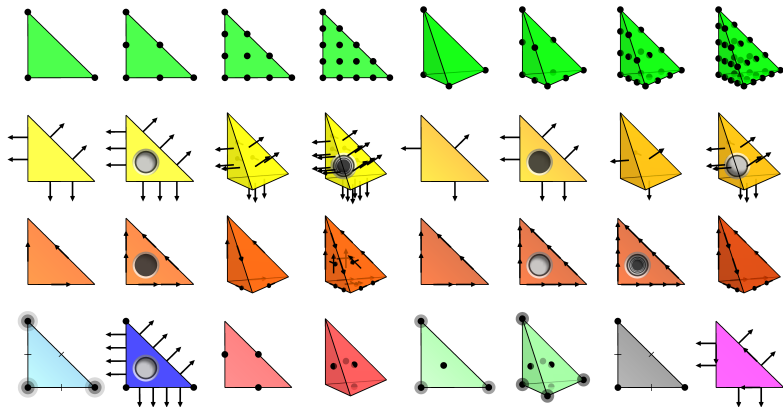
## Collaborators

*Simula Research Laboratory, University of Cambridge,  
University of Chicago, Texas Tech University, University of  
Texas at Austin, KTH Royal Institute of Technology, ...*



# FEniCS is automated FEM

- Automated generation of basis functions
- Automated evaluation of variational forms
- Automated finite element assembly
- Automated adaptive error control



# FEniCS is automated scientific computing

## Input

- $A(u) = f$
- $\epsilon > 0$

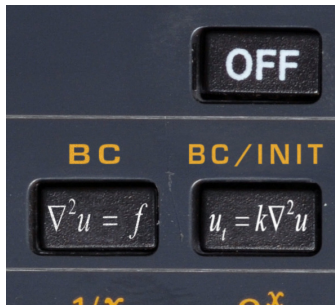
## Output

- Approximate solution:

$$u_h \approx u$$

- Guaranteed accuracy:

$$\|u - u_h\| \leq \epsilon$$



# How to use FEniCS?

# Installation



Official packages for Debian and Ubuntu



Drag and drop installation on Mac OS X



Binary installer for Windows



Automated installation from source

# Hello World in FEniCS: problem formulation

## Poisson's equation

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega \\ u &= 0 && \text{on } \partial\Omega \end{aligned}$$

## Finite element formulation

Find  $u \in V$  such that

$$\underbrace{\int_{\Omega} \nabla u \cdot \nabla v \, dx}_{a(u,v)} = \underbrace{\int_{\Omega} f v \, dx}_{L(v)} \quad \forall v \in V$$

# Hello World in FEniCS: implementation

```
from dolfin import *

mesh = UnitSquare(32, 32)

V = FunctionSpace(mesh, "Lagrange", 1)
u = TrialFunction(V)
v = TestFunction(V)
f = Expression("x[0]*x[1]")

a = dot(grad(u), grad(v))*dx
L = f*v*dx

bc = DirichletBC(V, 0.0, DomainBoundary())

u = Function(V)
solve(a == L, u, bc)
plot(u)
```

# Linear elasticity

## Differential equation

Differential equation:

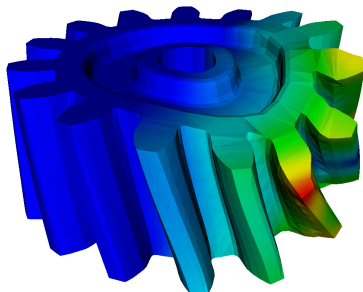
$$-\nabla \cdot \sigma(u) = f$$

where

$$\sigma(v) = 2\mu\epsilon(v) + \lambda\text{tr}\epsilon(v)I$$

$$\epsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^\top)$$

- Displacement  $u = u(x)$
- Stress  $\sigma = \sigma(x)$



# Linear elasticity

## Variational formulation

Find  $u \in V$  such that

$$a(u, v) = L(v) \quad \forall v \in \hat{V}$$

where

$$a(u, v) = \langle \sigma(u), \epsilon(v) \rangle$$

$$L(v) = \langle f, v \rangle$$



# Linear elasticity

## Implementation

```
element = VectorElement("Lagrange", "tetrahedron", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element)

def epsilon(v):
    return 0.5*(grad(v) + grad(v).T)

def sigma(v):
    return 2.0*mu*epsilon(v) + lambda*tr(epsilon(v))*I

a = inner(sigma(u), epsilon(v))*dx
L = dot(f, v)*dx
```

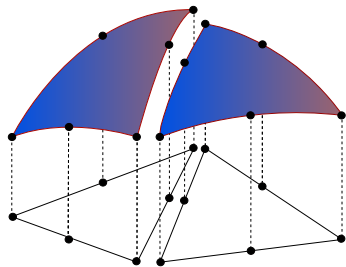
# Poisson's equation with DG elements

## Differential equation

Differential equation:

$$-\Delta u = f$$

- $u \in L^2$
- $u$  discontinuous across element boundaries



# Poisson's equation with DG elements

## Variational formulation (interior penalty method)

Find  $u \in V$  such that

$$a(u, v) = L(v) \quad \forall v \in V$$

where

$$\begin{aligned} a(u, v) &= \int_{\Omega} \nabla u \cdot \nabla v \, dx \\ &+ \sum_S \int_S -\langle \nabla u \rangle \cdot \llbracket v \rrbracket_n - \llbracket u \rrbracket_n \cdot \langle \nabla v \rangle + (\alpha/h) \llbracket u \rrbracket_n \cdot \llbracket v \rrbracket_n \, dS \\ &+ \int_{\partial\Omega} -\nabla u \cdot \llbracket v \rrbracket_n - \llbracket u \rrbracket_n \cdot \nabla v + (\gamma/h) uv \, ds \\ L(v) &= \int_{\Omega} f v \, dx + \int_{\partial\Omega} g v \, ds \end{aligned}$$

# Poisson's equation with DG elements

## Implementation

```
V = FunctionSpace(mesh, "DG", 1)

u = TrialFunction(V)
v = TestFunction(V)

f = Expression(...)
g = Expression(...)
n = FacetNormal(mesh)
h = CellSize(mesh)

a = dot(grad(u), grad(v))*dx
  - dot(avg(grad(u)), jump(v, n))*dS
  - dot(jump(u, n), avg(grad(v)))*dS
  + alpha/avg(h)*dot(jump(u, n), jump(v, n))*dS
  - dot(grad(u), jump(v, n))*ds
  - dot(jump(u, n), grad(v))*ds
  + gamma/h*u*v*ds
```

# Simple prototyping and development in Python

```
# Tentative velocity step (sigma formulation)
U = 0.5*(u0 + u)
F1 = rho*(1/k)*inner(v, u - u0)*dx +
rho*inner(v, grad(u0)*(u0 - w))*dx \
+ inner(epsilon(v), sigma(U, p0))*dx \
+ inner(v, p0*n)*ds - mu*inner(grad(U).T*n, v)*ds \
- inner(v, f)*dx
a1 = lhs(F1)
L1 = rhs(F1)
```

```
class StVenantKirchhoff(MaterialModel):

    def model_info(self):
        self.num_parameters = 2
        self.kinematic_measure = \
            "GreenLagrangeStrain"

    def strain_energy(self, parameters):
        E = self.E
        [mu, lambda] = parameters
        return lambda/2*(tr(E)**2) + mu*tr(E**E)
```

```
class GentThomas(MaterialModel):

    def model_info(self):
        self.num_parameters = 2
        self.kinematic_measure = \
            "CauchyGreenInvariants"

    def strain_energy(self, parameters):
        I1 = self.I1
        I2 = self.I2

        [C1, C2] = parameters
        return C1*(I1 - 3) + C2*ln(I2/3)
```

```
# Time-stepping loop
while True:

    # Fixed point iteration on FSI problem
    for iter in range(maxiter):

        # Solve fluid subproblem
        F.step(dt)

        # Transfer fluid stresses to structure
        Sigma_F = F.compute_fluid_stress(u_F0, u_F1,
                                         p_F0, p_F1,
                                         U_M0, U_M1)
        S.update_fluid_stress(Sigma_F)

        # Solve structure subproblem
        U_S1, P_S1 = S.step(dt)

        # Transfer structure displacement to fluidmesh
        M.update_structure_displacement(U_S1)

        # Solve mesh equation
        M.step(dt)

        # Transfer mesh displacement to fluid
        F.update_mesh_displacement(U_M1, dt)
```

```
# Fluid residual contributions
R_F0 = w*inner(EZ_F - Z_F, Dt_U_F - div(Sigma_F))*dx_F
R_F1 = avg(w)*inner(EZ_F('+') - Z_F('+'),
                   jump(Sigma_F, N_F))*dS_F
R_F2 = w*inner(EZ_F - Z_F, dot(Sigma_F, N_F))*ds
R_F3 = w*inner(EY_F - Y_F,
               div(J(U_M)*dot(inv(F(U_M)), U_F)))*dx_F
```

# Simple development of specialized applications



```
# Define Cauchy stress tensor
def sigma(v,w):
    return 2.0*mu*0.5*(grad(v) + grad(v).T) -
w*Identity(v.cell().d)

# Define symmetric gradient
def epsilon(v):
    return 0.5*(grad(v) + grad(v).T)

# Tentative velocity step (sigma formulation)
U = 0.5*(u0 + u)
F1 = rho*(1/k)*inner(v, u - u0)*dx +
rho*inner(v, grad(u0)*(u0 - w))*dx \
+ inner(epsilon(v), sigma(U, p0))*dx \
+ inner(v, p0*n)*ds - mu*inner(grad(U).T*n, v)*ds \
- inner(v, f)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Pressure correction
a2 = inner(grad(q), k*grad(p))*dx
L2 = inner(grad(q), k*grad(p0))*dx - q*div(u1)*dx

# Velocity correction
a3 = inner(v, u)*dx
L3 = inner(v, u1)*dx + inner(v, k*grad(p0 - p1))*dx
```

- The Navier–Stokes solver is implemented in Python/FEniCS
- FEniCS allows solvers to be implemented in a minimal amount of code
- Simple integration with application specific code and data management

# FEniCS under the hood

# Automatic code generation

## Input

Equation (variational problem)

## Output

Efficient application-specific code

$$\rho \left( \frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} \right) = -\nabla p + \mu \nabla^2 \vec{u} + \rho \vec{b}$$
$$\nabla \cdot \vec{u} = 0$$

```
// Extract vertex coordinates
const double * const c = c.coordinates;

// Compute Jacobian of affine map from reference cell
const double J_00 = *11001 - *01001;
const double J_01 = *12101 - *02001;
const double J_10 = *11111 - *01111;
const double J_11 = *12111 - *02111;

// Compute determinant of Jacobian
double detJ = J_00*J_11 - J_01*J_10;

// Compute inverse of Jacobian
const double Jinv_00 = J_11 / detJ;
const double Jinv_01 = -J_01 / detJ;
const double Jinv_10 = -J_10 / detJ;
const double Jinv_11 = J_00 / detJ;

// Take absolute value of determinant
detJ = std::abs(detJ);

// Set scale factor
const double det = detJ;

// Compute geometry tensors
const double G0_0_0 = det*(Jinv_00*Jinv_00 + Jinv_01*Jinv_01);
const double G0_0_1 = det*(Jinv_00*Jinv_10 + Jinv_01*Jinv_11);
const double G0_1_0 = det*(Jinv_10*Jinv_00 + Jinv_11*Jinv_01);
const double G0_1_1 = det*(Jinv_10*Jinv_10 + Jinv_11*Jinv_11);
```

Equation (variational form)

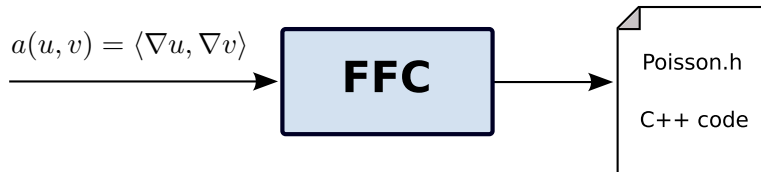
Form compiler

Application-specific code



# Code generation framework

- UFL - Unified Form Language
- UFC - Unified Form-assembly Code
- Form compilers: FFC, SyFi



# Form compiler interfaces

## Command-line

```
>> ffc poisson.ufl
```

## Just-in-time

```
V = FunctionSpace(mesh, "CG", 3)
u = TrialFunction(V)
v = TestFunction(V)
A = assemble(dot(grad(u), grad(v))*dx)
```

# Geometry and meshing

# Geometry and meshing

## Built-in meshing

```
mesh = UnitSquare(64, 64)
mesh = UnitCube(64, 64, 64)
```

## External mesh generators

```
mesh = Mesh("mesh.xml")
```

```
dolfin-convert mesh.inp mesh.xml
```

Conversion from Gmsh, Medit, Tetgen, Diffpack, Abaqus, ExodusII, Star-CD

## Extensions / work in progress

- Constructive solid geometry (CSG)
- Meshing from biomedical image data using VMTK

# Constructive solid geometry (CSG)

## Boolean operators

$A \cup B$      $A + B$

$A \cap B$      $A * B$

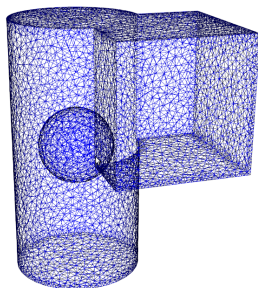
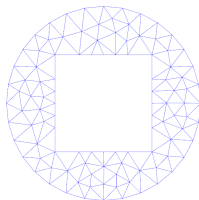
$A \setminus B$      $A - B$

## Implementation

- Modeled after NETGEN
- Implemented using CGAL

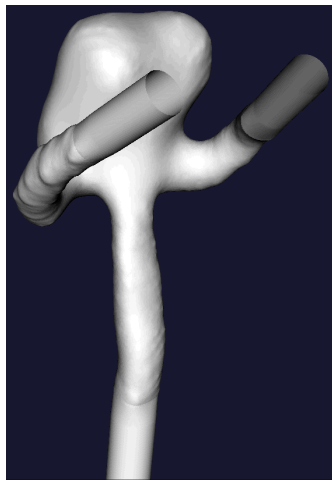
## Example

```
r = Rectangle(-1, -1, 1, 1)
c = Circle(0, 0, 1)
g = c - r
mesh = Mesh(g)
```



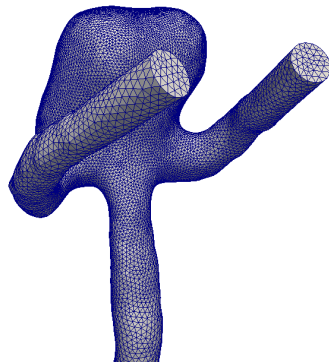
# Meshing from biomedical images

- Biomedical image data (DICOM)
- VMTK generates high quality FEniCS meshes
- Adaptive *a priori* graded meshes
- Simple specification of boundary markers
- Resolution of boundary layers



# Meshing from biomedical images

- Biomedical image data (DICOM)
- VMTK generates high quality FEniCS meshes
- Adaptive *a priori* graded meshes
- Simple specification of boundary markers
- Resolution of boundary layers



# Meshing from biomedical images

- Biomedical image data (DICOM)
- VMTK generates high quality FEniCS meshes
- Adaptive *a priori* graded meshes
- Simple specification of boundary markers
- Resolution of boundary layers





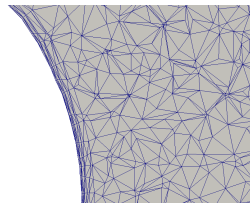
# Meshing from biomedical images

- Biomedical image data (DICOM)
- VMTK generates high quality FEniCS meshes
- Adaptive *a priori* graded meshes
- Simple specification of boundary markers
- Resolution of boundary layers



# Meshing from biomedical images

- Biomedical image data (DICOM)
- VMTK generates high quality FEniCS meshes
- Adaptive *a priori* graded meshes
- Simple specification of boundary markers
- Resolution of boundary layers



# Automated error control

# Automated goal-oriented error control

## Input

- Variational problem: Find  $u \in V$ :  $a(u, v) = L(v) \quad \forall v \in V$
- Quantity of interest:  $\mathcal{M} : V \rightarrow \mathbb{R}$
- Tolerance:  $\epsilon > 0$

## Objective

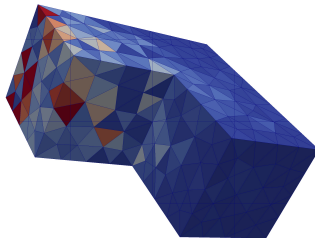
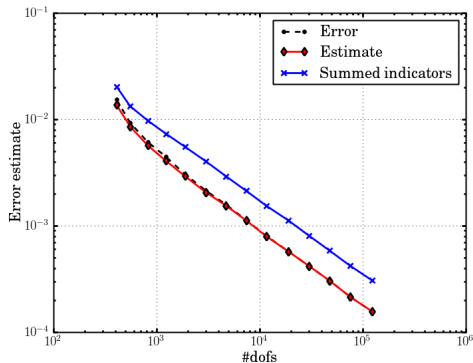
Find  $V_h \subset V$  such that  $|\mathcal{M}(u) - \mathcal{M}(u_h)| < \epsilon$  where

$$a(u_h, v) = L(v) \quad \forall v \in V_h$$

## Automated in FEniCS (for linear and nonlinear PDE)

```
solve(a == L, u, M=M, tol=1e-3)
```

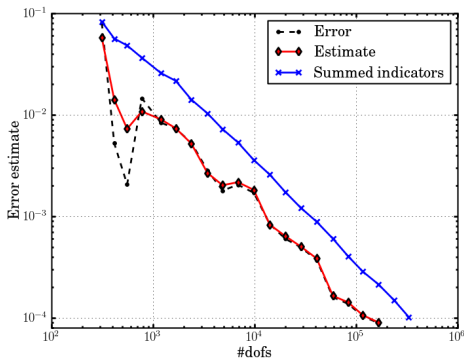
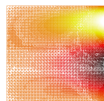
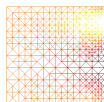
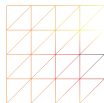
# Poisson's equation



$$a(u, v) = \langle \nabla u, \nabla v \rangle$$

$$\mathcal{M}(u) = \int_{\Gamma} u \, ds, \quad \Gamma \subset \partial\Omega$$

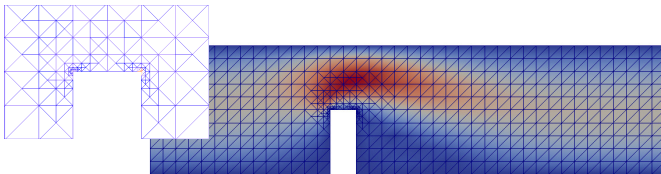
# A three-field mixed elasticity formulation



$$a((\sigma, u, \gamma), (\tau, v, \eta)) = \langle A\sigma, \tau \rangle + \langle u, \operatorname{div} \tau \rangle + \langle \operatorname{div} \sigma, v \rangle + \langle \gamma, \tau \rangle + \langle \sigma, \eta \rangle$$

$$\mathcal{M}((\sigma, u, \eta)) = \int_{\Gamma} g \sigma \cdot n \cdot t \, ds$$

# Incompressible Navier–Stokes



Outflux  $\approx 0.4087 \pm 10^{-4}$

**Uniform**

1.000.000 dofs,  $N$  hours

**Adaptive**

5.200 dofs, 127 seconds

```
from dolfin import *

class Noslip(SubDomain): ...

mesh = Mesh("channel-with-flap.xml.gz")
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V*Q

# Define test functions and unknown(s)
(v, q) = TestFunctions(W)
w = Function(W)
(u, p) = split(w)

# Define (non-linear) form
n = FacetNormal(mesh)
p0 = Expression("(4.0 - x[0])/4.0")
F = (0.02*inner(grad(u), grad(v)) + inner(grad(u)*u, v)*dx
     - p*div(v) + div(u)*q + dot(v, n)*p0*ds

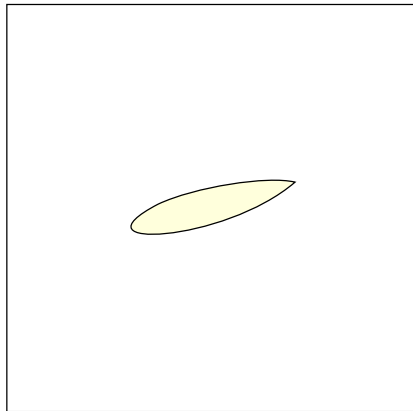
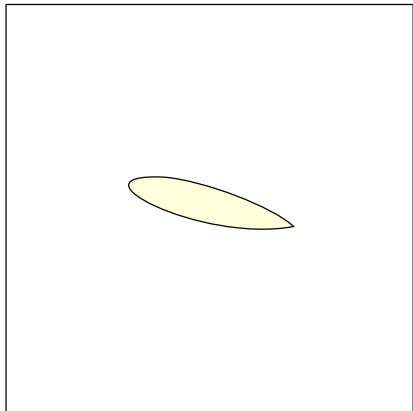
# Define goal functional
M = u[0]*ds(0)

# Compute solution
tol = 1e-4
solve(F == 0, w, bcs, M, tol)
```

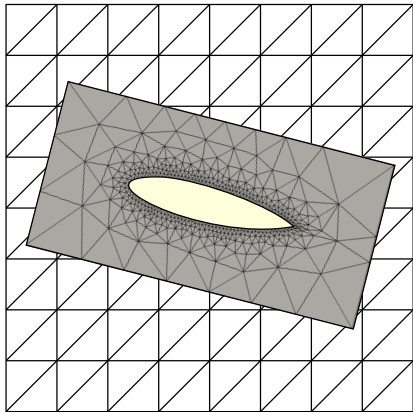
## Cut finite elements



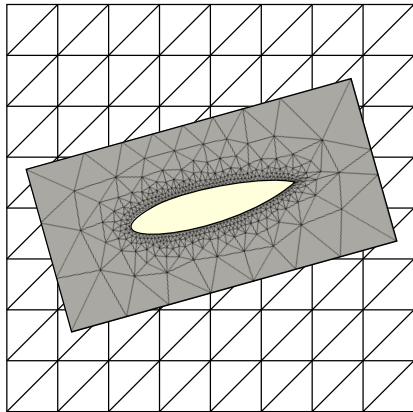
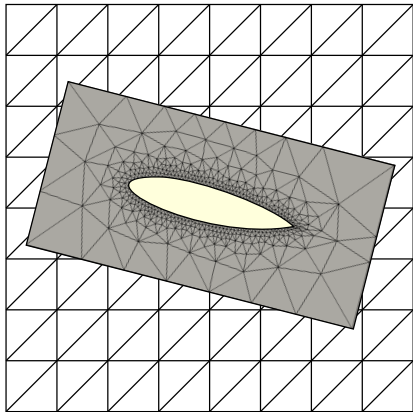
# Multiple geometries – multiple meshes



# Multiple geometries – multiple meshes



# Multiple geometries – multiple meshes

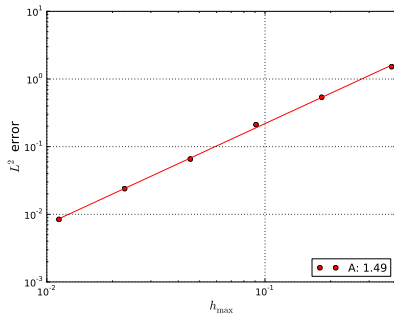
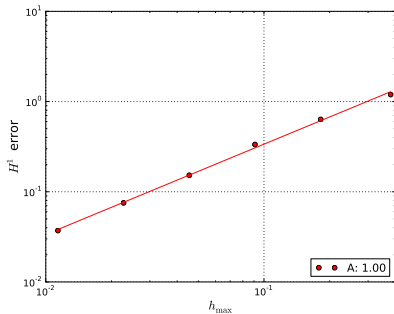


# Optimal *a priori* estimates

## Theorem

Let  $k, l \geq 1$  and assume that  $(\mathbf{u}, p) \in [H^{k+1}(\Omega)]^d \times H^{l+1}(\Omega)$  is a (weak) solution of the Stokes problem. Then the finite element solution  $(\mathbf{u}_h, p_h) \in V_h^k \times Q_h^l$  satisfies the following error estimate:

$$\|(\mathbf{u} - \mathbf{u}_h, p - p_h)\| \lesssim h^k |\mathbf{u}|_{k+1} + h^{l+1} |p|_{l+1}.$$

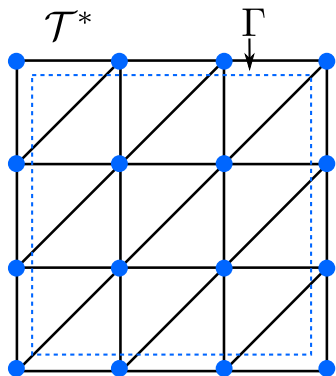


# Bounded condition numbers

## Theorem

There is a constant  $C > 0$  *independent* of the position of  $\Gamma$ , s.t. the condition number of the stiffness matrix  $\mathcal{A}$  associated with the Nitsche fictitious domain method satisfies

$$\kappa(\mathcal{A}) \leq Ch^{-2},$$

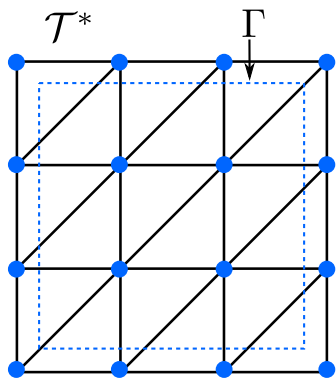


# Bounded condition numbers

## Theorem

There is a constant  $C > 0$  *independent* of the position of  $\Gamma$ , s.t. the condition number of the stiffness matrix  $\mathcal{A}$  associated with the Nitsche fictitious domain method satisfies

$$\kappa(\mathcal{A}) \leq Ch^{-2},$$

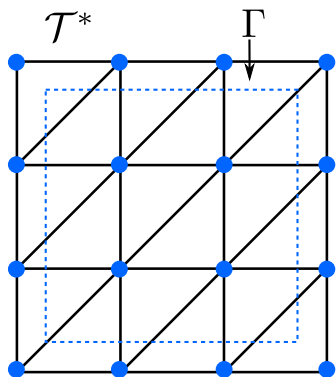


# Bounded condition numbers

## Theorem

There is a constant  $C > 0$  *independent* of the position of  $\Gamma$ , s.t. the condition number of the stiffness matrix  $\mathcal{A}$  associated with the Nitsche fictitious domain method satisfies

$$\kappa(\mathcal{A}) \leq Ch^{-2},$$

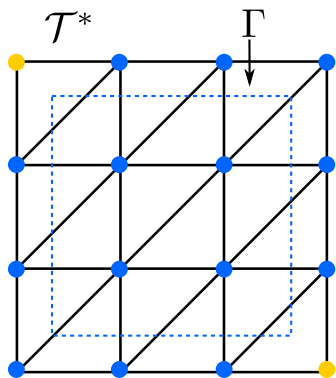


# Bounded condition numbers

## Theorem

There is a constant  $C > 0$  *independent* of the position of  $\Gamma$ , s.t. the condition number of the stiffness matrix  $\mathcal{A}$  associated with the Nitsche fictitious domain method satisfies

$$\kappa(\mathcal{A}) \leq Ch^{-2},$$



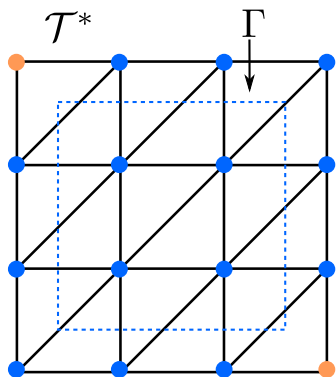


# Bounded condition numbers

## Theorem

There is a constant  $C > 0$  *independent* of the position of  $\Gamma$ , s.t. the condition number of the stiffness matrix  $\mathcal{A}$  associated with the Nitsche fictitious domain method satisfies

$$\kappa(\mathcal{A}) \leq Ch^{-2},$$

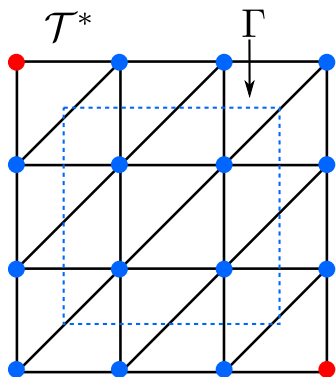


# Bounded condition numbers

## Theorem

There is a constant  $C > 0$  *independent* of the position of  $\Gamma$ , s.t. the condition number of the stiffness matrix  $\mathcal{A}$  associated with the Nitsche fictitious domain method satisfies

$$\kappa(\mathcal{A}) \leq Ch^{-2},$$

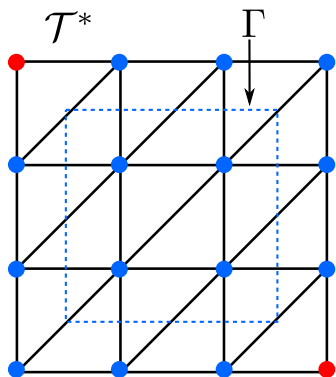


# Bounded condition numbers

## Theorem

There is a constant  $C > 0$  *independent* of the position of  $\Gamma$ , s.t. the condition number of the stiffness matrix  $\mathcal{A}$  associated with the Nitsche fictitious domain method satisfies

$$\kappa(\mathcal{A}) \leq Ch^{-2},$$

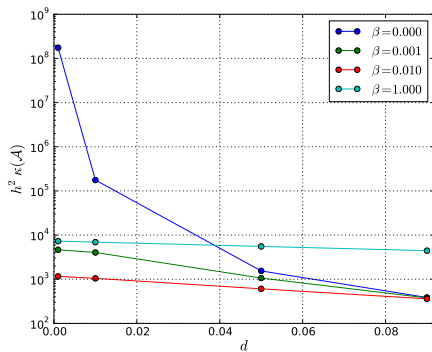
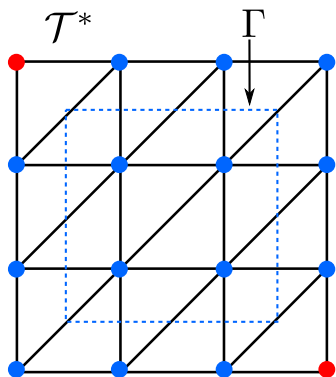


# Bounded condition numbers

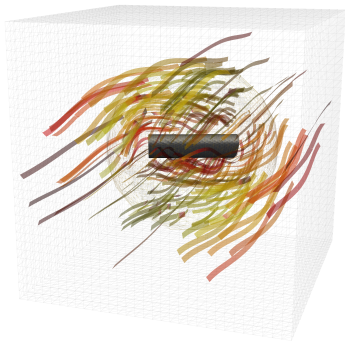
## Theorem

There is a constant  $C > 0$  *independent* of the position of  $\Gamma$ , s.t. the condition number of the stiffness matrix  $\mathcal{A}$  associated with the Nitsche fictitious domain method satisfies

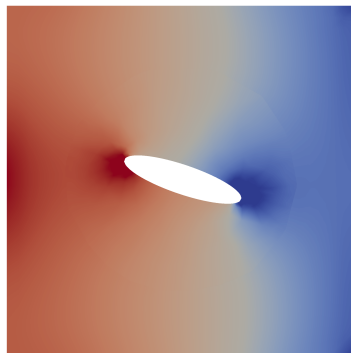
$$\kappa(\mathcal{A}) \leq Ch^{-2},$$



# Stokes flow for different angles of attack

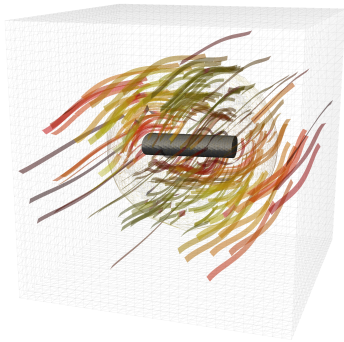


Velocity streamlines

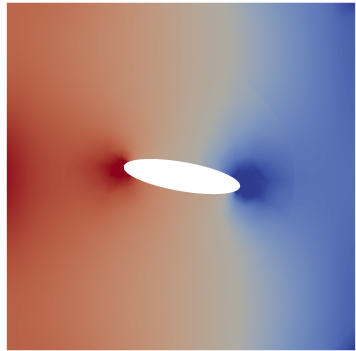


Pressure

# Stokes flow for different angles of attack

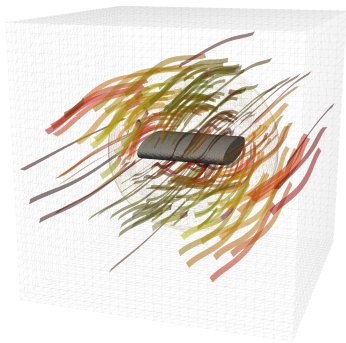


Velocity streamlines

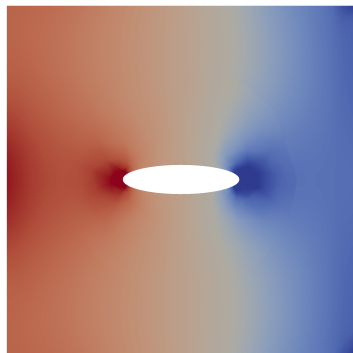


Pressure

# Stokes flow for different angles of attack

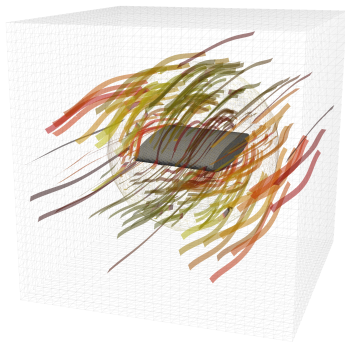


Velocity streamlines

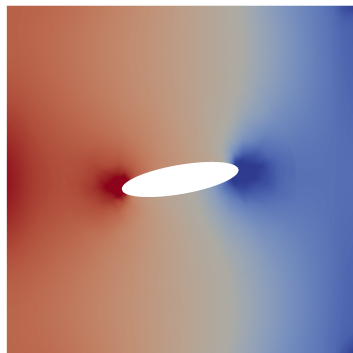


Pressure

# Stokes flow for different angles of attack



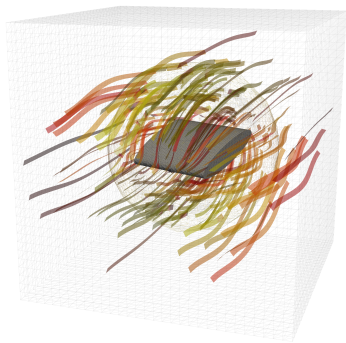
Velocity streamlines



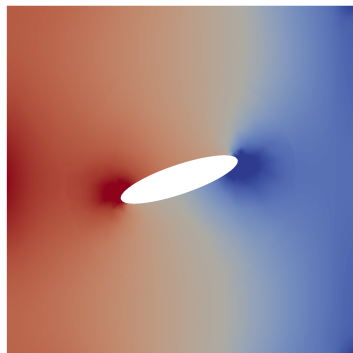
Pressure



# Stokes flow for different angles of attack

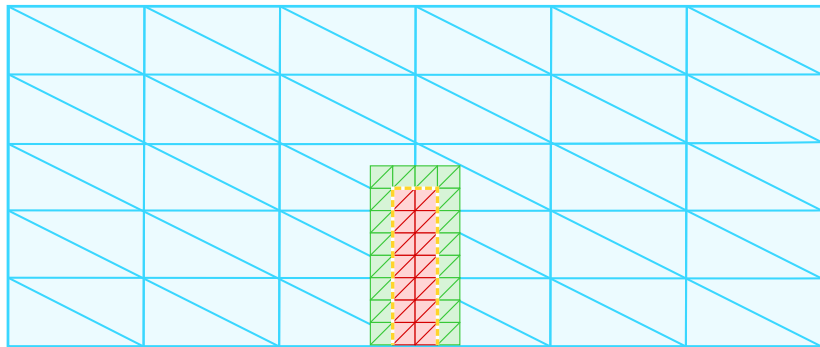


Velocity streamlines



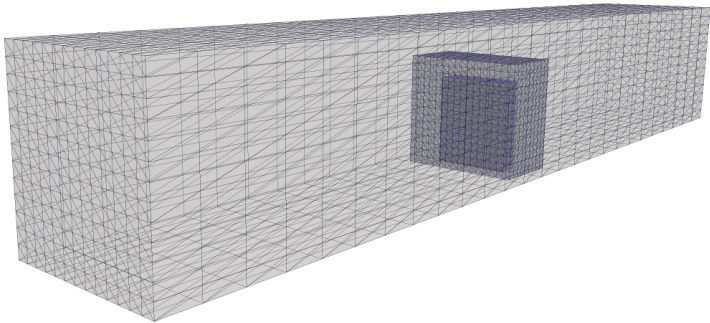
Pressure

# Fluid–structure interaction on cut meshes

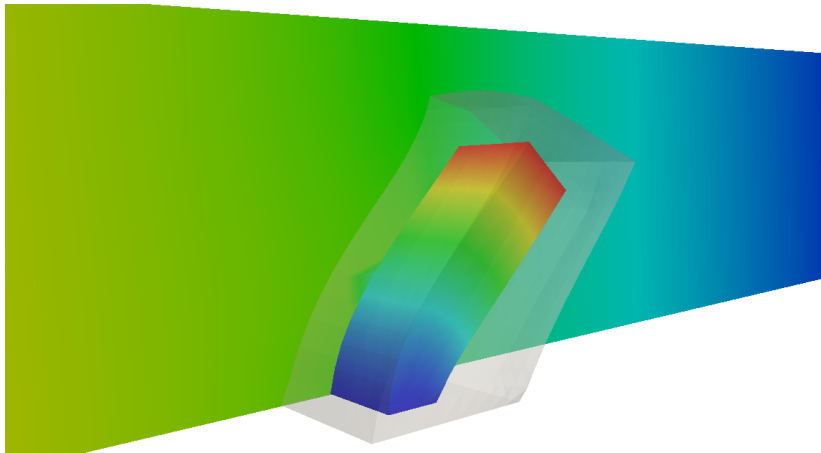


 *Fluid*     *Mesh + Fluid*     *Structure*     *Interface*

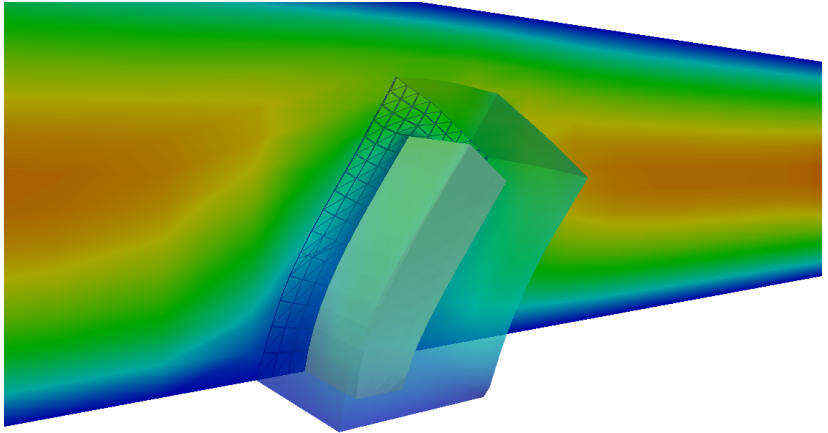
# Fluid–structure interaction on cut meshes



# Fluid–structure interaction: displacement

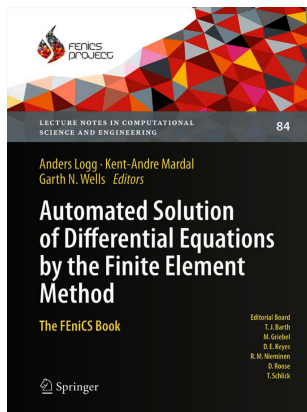


# Fluid–structure interaction: velocity magnitude



## Closing remarks

# Ongoing activities



- Parallelization (2009)
- Automated error control (2010)
- Debian/Ubuntu (2010)
- Documentation (2011)
- FEniCS 1.0 (2011)
- The FEniCS Book (2012)
  
- **FEniCS'13**  
Cambridge March 2013
- Visualization, mesh generation
- Parallel AMR
- Hybrid MPI/OpenMP
- Overlapping/intersecting meshes

# Summary

- Automated solution of PDE
- Easy install
- Easy scripting in Python
- Efficiency by automated code generation
- Free/open-source (LGPL)

<http://fenicsproject.org/>

