Flexible Specification of Large Systems of Nonlinear PDEs

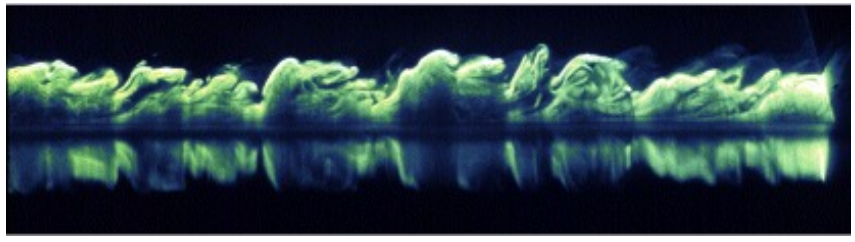
Hans Petter Langtangen[1,2] Mikael Mortensen[3,1]

Center for Biomedical Computing
Simula Research Laboratory[1]

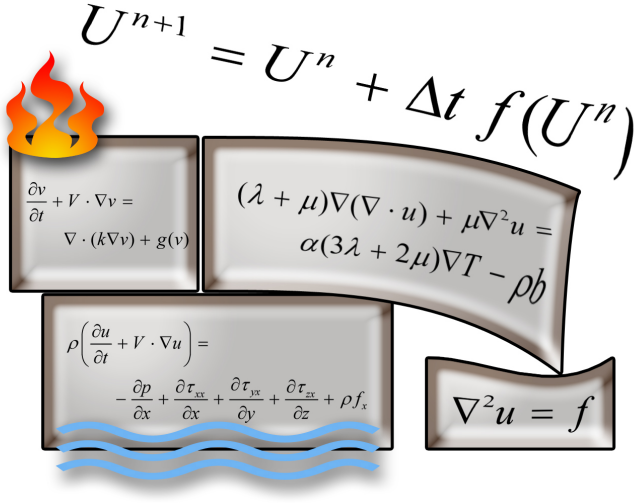
Dept. of Informatics, University of Oslo[2]

Dept. of Mathematics, University of Oslo[3]

[HPC]^3 at KAUST, Feb 6, 2012



FEniCS PDE tools



PDE systems tools and CFD

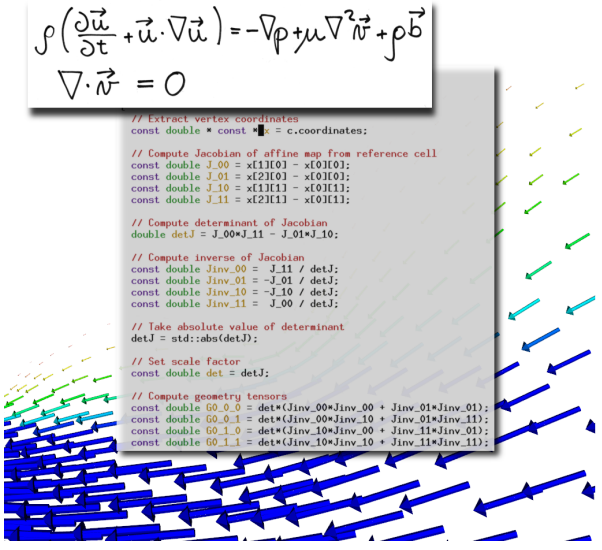
FEniCS solves PDEs by the finite element method

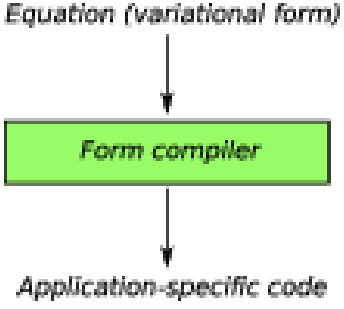
Input: finite element formulation of the PDE problem

$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x + \int_\Omega f v \, \mathrm{d}x$ in Python

Output: C++ code loaded back in Python

Python module with C++ def. of element matrix/vector, linked to finite element and linear algebra libraries





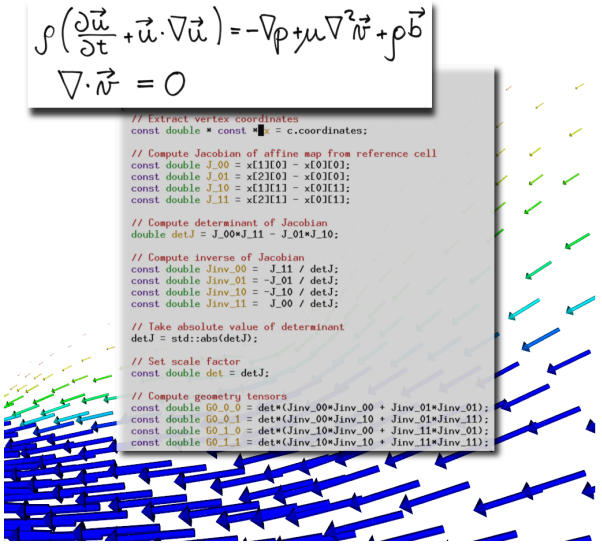
FEniCS solves PDEs by the finite element method

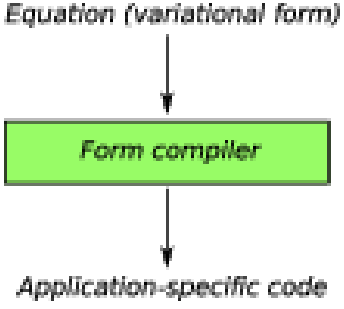
Input: finite element formulation of the PDE problem

$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x + \int_\Omega f v \, \mathrm{d}x$ in Python

Output: C++ code loaded back in Python

Python module with C++ def. of element matrix/vector, linked to finite element and linear algebra libraries





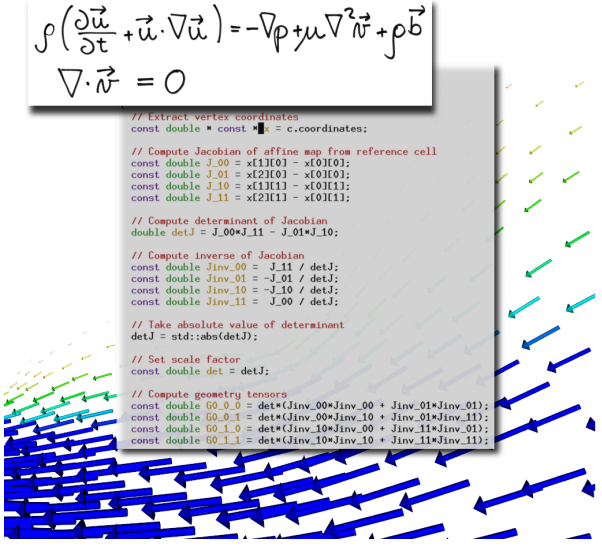
FEniCS solves PDEs by the finite element method

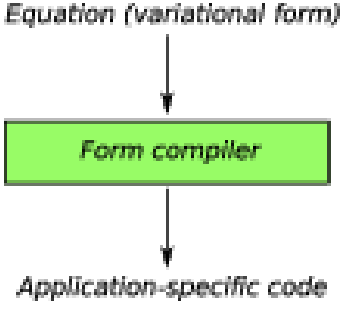
Input: finite element formulation of the PDE problem

$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x + \int_\Omega f v \, \mathrm{d}x$ in Python

Output: C++ code loaded back in Python

Python module with C++ def. of element matrix/vector, linked to finite element and linear algebra libraries





FEniCS tries to combine four contradictory goals

Simplicity

$\int_\Omega a \nabla u \cdot \nabla v \, \mathrm{d}x \rightarrow$ `inner(a*grad(u), grad(v))*dx`

Generality

Linear $a(u, v) = L(v)$ or nonlinear $F(u; v) = 0$ variational problem

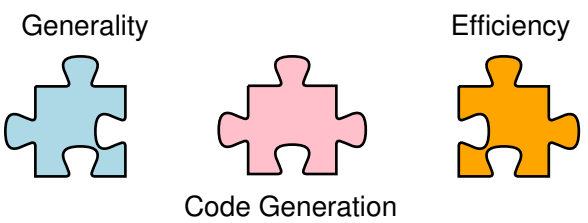
Efficiency

Generated C++ code tailored to the problem + efficient third-party libraries (PETSc, Trilinos, ...)

Reliability

Given a goal $\mathcal{M}(u)$ and tolerance ϵ, compute u such that

$$|\mathcal{M}(u_e) - \mathcal{M}(u)| \leq \epsilon \quad (u_e\text{: exact sol.})$$



FEniCS tries to combine four contradictory goals

Simplicity

$\int_\Omega a\nabla u \cdot \nabla v \, \mathrm{d}x \rightarrow$ `inner(a*grad(u), grad(v))*dx`

Generality

Linear $a(u, v) = L(v)$ or nonlinear $F(u; v) = 0$ variational problem

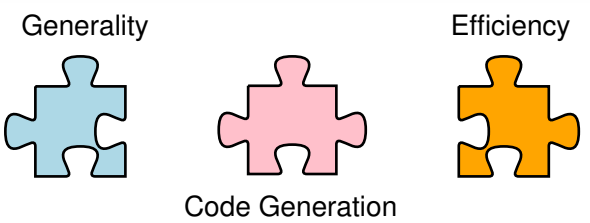
Efficiency

Generated C++ code tailored to the problem + efficient third-party libraries (PETSc, Trilinos, ...)

Reliability

Given a goal $\mathcal{M}(u)$ and tolerance ϵ, compute u such that

$$||\mathcal{M}(u_e) - \mathcal{M}(u)|| \leq \epsilon \quad (u_e\text{: exact sol.})$$



FEniCS tries to combine four contradictory goals

Simplicity

$\int_\Omega a\nabla u \cdot \nabla v \, \mathrm{d}x \rightarrow$ `inner(a*grad(u), grad(v))*dx`

Generality

Linear $a(u, v) = L(v)$ or nonlinear $F(u; v) = 0$ variational problem

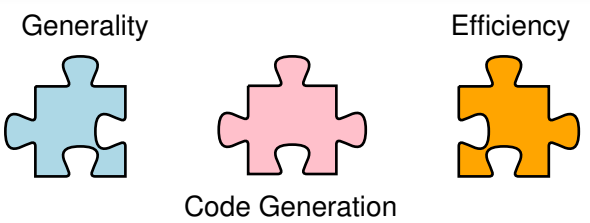
Efficiency

Generated C++ code tailored to the problem + efficient third-party libraries (PETSc, Trilinos, ...)

Reliability

Given a goal $\mathcal{M}(u)$ and tolerance ϵ, compute u such that

$$||\mathcal{M}(u_e) - \mathcal{M}(u)|| \leq \epsilon \quad (u_e\text{: exact sol.})$$



FEniCS tries to combine four contradictory goals

Simplicity

$\int_\Omega a \nabla u \cdot \nabla v \, \mathrm{d}x \rightarrow$ `inner(a*grad(u), grad(v))*dx`

Generality

Linear $a(u, v) = L(v)$ or nonlinear $F(u; v) = 0$ variational problem

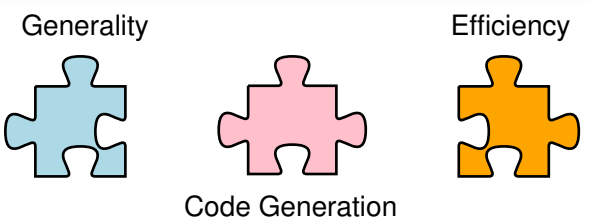
Efficiency

Generated C++ code tailored to the problem + efficient third-party libraries (PETSc, Trilinos, ...)

Reliability

Given a goal $\mathcal{M}(u)$ and tolerance ϵ, compute u such that

$$||\mathcal{M}(u_\mathrm{e}) - \mathcal{M}(u)|| \leq \epsilon \quad (u_\mathrm{e}\text{: exact sol.})$$



FEniCS tries to combine four contradictory goals

Simplicity

$\int_\Omega a\nabla u \cdot \nabla v \, \mathrm{d}x \rightarrow$ `inner(a*grad(u), grad(v))*dx`

Generality

Linear $a(u, v) = L(v)$ or nonlinear $F(u; v) = 0$ variational problem

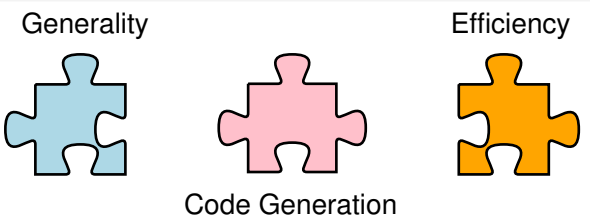
Efficiency

Generated C++ code tailored to the problem + efficient third-party libraries (PETSc, Trilinos, ...)

Reliability

Given a goal $\mathcal{M}(u)$ and tolerance ϵ, compute u such that

$$||\mathcal{M}(u_\mathrm{e}) - \mathcal{M}(u)|| \leq \epsilon \quad (u_\mathrm{e}\text{: exact sol.})$$



"Hello, world!" for PDEs: $-\nabla \cdot (k \nabla u) = f$

"Hello, world!" for HPC^3: $u_t + \nabla \cdot F(u) = 0$

"Hello, world!" for PDEs: $-\nabla \cdot (k\nabla u) = f$

$$-\nabla \cdot (k\nabla u) = f \text{ in } \Omega$$
$$u = g \text{ on } \partial\Omega_D$$
$$-k\frac{\partial u}{\partial n} = \alpha(u - u_0) \text{ on } \partial\Omega_R$$

Variational problem: find $u \in V$ such that

$$F = \int_\Omega k\nabla u \cdot \nabla v dx - \int_\Omega fv dx + \int_{\partial\Omega_R} \alpha(u - u_0)v ds = 0 \quad \forall\, v \in V$$

Implementation:

```
F = inner(k*grad(u), grad(v))*dx - f*v*dx + alpha*(u-u0)*v*ds
```

"Hello, world!" for PDEs: $-\nabla \cdot (k\nabla u) = f$

$$-\nabla \cdot (k\nabla u) = f \text{ in } \Omega$$
$$u = g \text{ on } \partial\Omega_D$$
$$-k\frac{\partial u}{\partial n} = \alpha(u - u_0) \text{ on } \partial\Omega_R$$

Variational problem: find $u \in V$ such that

$$F = \int_\Omega k\nabla u \cdot \nabla v dx - \int_\Omega fv dx + \int_{\partial\Omega_R} \alpha(u - u_0)v ds = 0 \quad \forall\, v \in V$$

Implementation:

```
F = inner(k*grad(u), grad(v))*dx - f*v*dx + alpha*(u-u0)*v*ds
```

"Hello, world!" for PDEs: $-\nabla \cdot (k\nabla u) = f$

$$
\begin{aligned}
-\nabla \cdot (k\nabla u) &= f \text{ in } \Omega \\
u &= g \text{ on } \partial\Omega_D \\
-k\frac{\partial u}{\partial n} &= \alpha(u - u_0) \text{ on } \partial\Omega_R
\end{aligned}
$$

Variational problem: find $u \in V$ such that

$$F = \int_\Omega k\nabla u \cdot \nabla v dx - \int_\Omega f v dx + \int_{\partial\Omega_R} \alpha(u - u_0) v ds = 0 \quad \forall\, v \in V$$

Implementation:

```
F = inner(k*grad(u), grad(v))*dx - f*v*dx + alpha*(u-u0)*v*ds
```

"Hello, world!" for PDEs: $-\nabla \cdot (k\nabla u) = f$

$$
\begin{aligned}
-\nabla \cdot (k\nabla u) &= f \text{ in } \Omega \\
u &= g \text{ on } \partial\Omega_D \\
-k\frac{\partial u}{\partial n} &= \alpha(u - u_0) \text{ on } \partial\Omega_R
\end{aligned}
$$

Variational problem: find $u \in V$ such that

$$F = \int_\Omega k\nabla u \cdot \nabla v dx - \int_\Omega f v dx + \int_{\partial\Omega_R} \alpha(u - u_0) v ds = 0 \quad \forall\, v \in V$$

Implementation:

```
F = inner(k*grad(u), grad(v))*dx - f*v*dx + alpha*(u-u0)*v*ds
```

The complete "Hello, world!" program

```
from dolfin import *
mesh = Mesh('mydomain.xml.gz')
V = FunctionSpace(mesh, 'Lagrange', degree=1)

dOmega_D = MeshFunction('uint', mesh, 'myboundary.xml.gz')
g = Constant(0.0)
bc = DirichletBC(V, g, dOmega_D)

u = TrialFunction(V)
v = TestFunction(V)
f = Constant(2.0)
k = Expression('A*x[1]*sin(pi*q*x[0])', A=4.5, q=1)
alpha = 10; u0 = 2

F = inner(k*grad(u), grad(v))*dx - f*v*dx + alpha*(u-u0)*v*ds

a = lhs(F); L = rhs(F)
u = Function(V)          # finite element function to compute
solve(a == L, u, bc)
plot(u)
```

Example of an autogenerated element matrix routine

```
void tabulate_tensor(double* A, ...)
{
...
const double G0_0_0 = det*Jinv00*Jinv00 + det*Jinv00*Jinv00
+ det*Jinv00*Jinv00 + det*Jinv00*Jinv00
+ det*Jinv01*Jinv01 + det*Jinv02*Jinv02
+ det*Jinv01*Jinv01 + det*Jinv02*Jinv02;
const double G0_0_1 = det*Jinv00*Jinv10 + det*Jinv00*Jinv10
+ det*Jinv00*Jinv10 + det*Jinv00*Jinv10
+ det*Jinv01*Jinv11 + det*Jinv02*Jinv12
+ det*Jinv01*Jinv11 + det*Jinv02*Jinv12;
...
const double G8_2_1 = det*Jinv21*Jinv12 + det*Jinv21*Jinv12;
const double G8_2_2 = det*Jinv21*Jinv22 + det*Jinv21*Jinv22;

const real tmp0_13 = 4.166666666666660e-02*G0_0_0;
const real tmp0_38 = 4.166666666666662e-02*G0_2_1;
const real tmp0_1 = -tmp0_13 + -4.166666666666661e-02*G0_1_0
- 4.166666666666660e-02*G0_2_0;
const real tmp0_37 = 4.166666666666661e-02*G0_2_0;
const real tmp0_25 = 4.166666666666661e-02*G0_1_0;
const real tmp0_26 = 4.166666666666662e-02*G0_1_1;

...
const real tmp8_139 = 4.166666666666662e-02*G8_2_2;
const real tmp8_125 = 4.166666666666661e-02*G8_1_0;
const real tmp8_100 = -tmp8_101 + 4.166666666666661e-02*G8_0_1
+ 4.166666666666661e-02*G8_0_2 + 4.166666666666662e-02*G8_1_1
+ 4.166666666666662e-02*G8_1_2 + 4.166666666666662e-02*G8_2_1
+ 4.166666666666662e-02*G8_2_2;
const real tmp8_126 = 4.166666666666662e-02*G8_1_1;
const real tmp8_113 = 4.166666666666660e-02*G8_0_0;
const real tmp8_138 = 4.166666666666662e-02*G8_2_1;
A[0] = tmp0_0;
A[1] = tmp0_1;
A[2] = tmp0_2;
...
A[141] = tmp6_141;
A[142] = tmp6_142;
A[143] = tmp6_143;
}
```

Mixed formulation of $-\nabla \cdot (k\nabla u) = f$

PDE problem:

$$\nabla \cdot q = f \text{ in } \Omega$$
$$-k^{-1}q = \nabla u \text{ in } \Omega$$

Variational problem: find $(u, q) \in V \times Q$ such that

$$F = \int_\Omega \nabla \cdot q\, v \, \mathrm{d}x - \int_\Omega fv \, \mathrm{d}x + \int_\Omega k^{-1} q \cdot p \, dx + \int_\Omega \nabla \cdot p \, u \, dx$$
$$\forall (v, p) \in V \times Q$$

Principal implementation line:

```
F = div(q)*v*dx - f*u*dx + (1./k)*inner(q,p)*dx + div(p)*u*dx
```

The program

```
mesh = UnitCube(N, N, N)
Q = FunctionSpace(mesh, "BDM", 1)
V = FunctionSpace(mesh, "DG", 0)
W = Q * V

q, u = TrialFunctions(W)
p, v = TestFunctions(W)

f = Expression('x[0] > L/2 ? a : 0', L=1, a=2)

F = div(q)*v*dx - f*u*dx + (1./k)*inner(q,p)*dx + div(p)*u*dx

a = lhs(F);  L = rhs(F)
A = assemble(a)
b = assemble(L)
qu = Function(W)    # compound (q,u) field to be solved for
solve(A, uq.vector(), b, 'gmres', 'ilu')
q, u = qu.split()
```

Discontinuous Galerkin method for $-\nabla \cdot (k \nabla u) = f$

Variational problem:

$$F = \int_\Omega \nabla u \cdot \nabla v \, dx - \int_\Gamma \langle \nabla u \rangle \cdot [vn] \, dS$$
$$- \int_\Gamma [un] \cdot \langle \nabla v \rangle \, dS + \frac{\alpha}{h} \int_\Gamma [un] \cdot [vn] \, dS - \int_\Omega fv \, \mathrm{d}x = 0$$

Implementation:

```
F = inner(grad(v), grad(u))*dx \
  - inner(avg(grad(v)), jump(u, n))*dS \
  - inner(jump(v, n), avg(grad(u)))*dS \
  + alpha/avg*inner(jump(v, n), jump(u, n))*dS \
  - f*v*dx

a = lhs(F);  L = rhs(F)
```

CFD "Hello, world!": Stokes problem'

Stokes' problem for slow viscous flow:

$$-\nabla^2 u + \nabla p = f$$
$$\nabla \cdot u = 0$$

Variational problem: find $(u, p) \in V \times Q$ such that

$$F = \int_\Omega (\nabla v \cdot \nabla u - \nabla \cdot v\, p + v \cdot f)\, \mathrm{d}x +$$
$$\int_\Omega q \nabla \cdot u\, \mathrm{d}x = 0 \quad \forall\, (v, q) \in V \times Q$$

CFD "Hello, world!" code

```
V = VectorFunctionSpace(mesh, 'Lagrange', 2)
Q = FunctionSpace(mesh, 'Lagrange', 1)
W = V * Q    # Taylor-Hood mixed finite element

v, q = TestFunctions(W)
u, p = TrialFunctions(W)

f = Constant((0, 0))

F = (inner(grad(v), grad(u)) - div(v)*p + q*div(u))*dx + \
    inner(v, f)*dx

a = lhs(F);  L = rhs(F)
up = Function(W)
solve(a == L, up, bc)      # solve variational problem

# or

A = assemble(a);  b = assemble(L)
solve(A, up.vector(), b)   # solve linear system

u, p = up.split()
```

Again, code ≈ math

Key mathematical formula:

$$F = \int_\Omega (\nabla v \cdot \nabla u - \nabla \cdot v\, p + v \cdot f)\,\mathrm{d}x + \int_\Omega q \nabla \cdot u\,\mathrm{d}x$$

Key code line:

```
F = (inner(grad(v), grad(u)) - div(v)*p + inner(f,v)*dx + \
    q*div(u))*dx
```

Hyperelasticity (Fung model for biological tissues)

Mathematical problem:

$$F = I + (\nabla \boldsymbol{u})$$ $\boldsymbol{u}$: unknown displacement

$$C = F^T : F$$

$$E = (C - I)/2$$

$$\psi = \frac{\lambda}{2}\,\mathrm{tr}(E)^2 + K \exp((EA, E))$$ material law

$$P = \frac{\partial \psi}{\partial E}$$ stress tensor

$$F = \int_\Omega P : (\nabla \boldsymbol{v})\, dx$$ nonlinear variational form

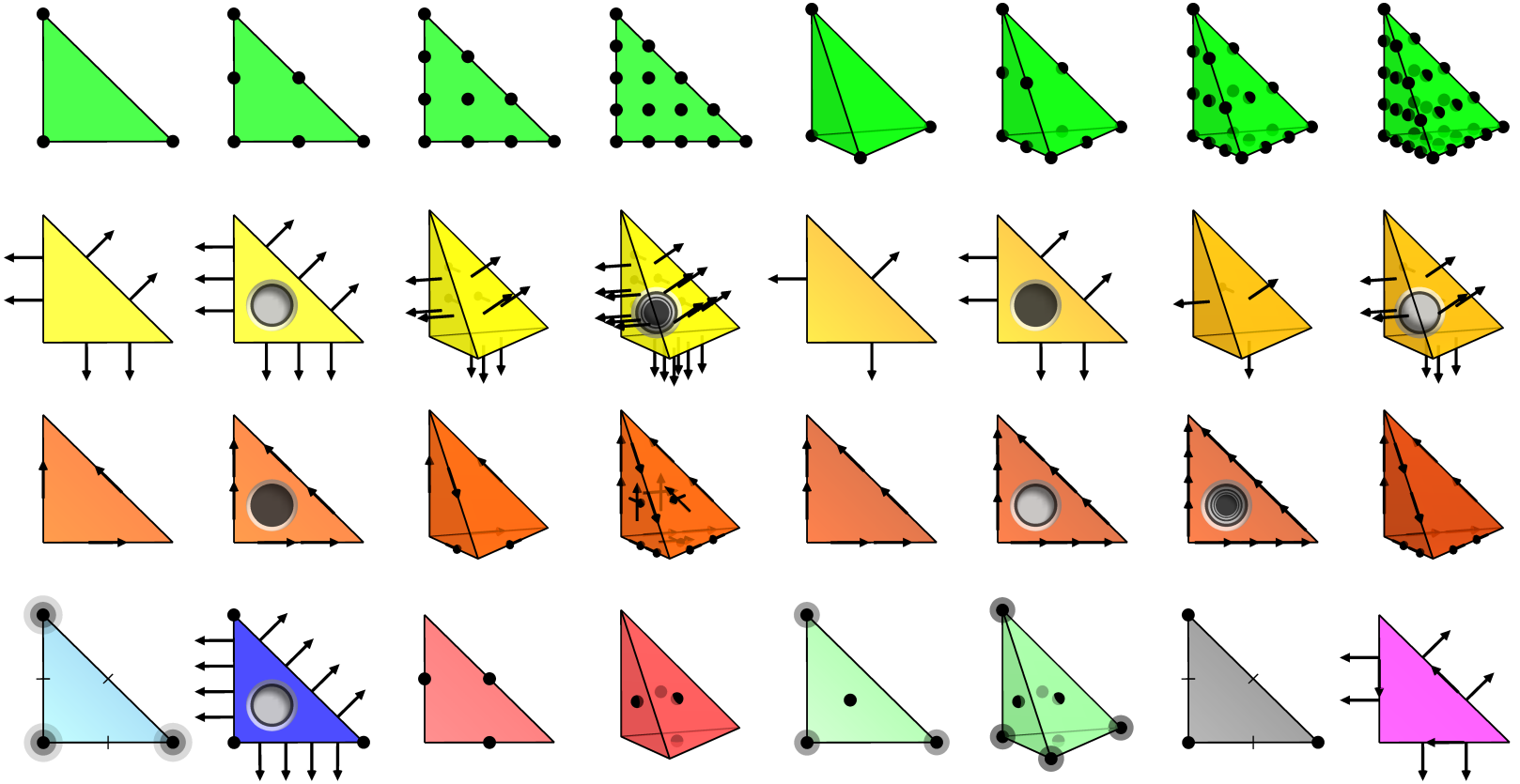
$$J = \frac{\partial F}{\partial u}$$ Jacobian ("tanget stiffness")

Hyperelasticity implementation

```
V = VectorFunctionSpace(mesh, 'Lagrange', order)

v = TestFunction(V)
u = TrialFunction(V)
u_ = Function(V)      # computed solution

I = Identity(u_.cell().d)
F = I + grad(u_)
J = det(F)

C = F.T * F
E = (C-I)/2

# Material law
lamda = Constant(1.0)
A = Expression ([['1.0 + x[0]', '0.3'], ['0.3', '2.3']])
K = Constant(1.0)
psi = lamda/2 * tr(E)**2  +  K*exp(inner(A*E,E))

P = F*diff(psi, E)          # symbolic differentiation
F = inner(P, grad(v))*dx
J = derivative(F, u_, u)    # symbolic differentiation
A = assemble(J)
```

FEniCS supports a rich set of finite elements

- $\mathrm{Lagrange}_q$ (P_q), DG_q, BDM_q, BDFM_q, RT_q, Nedelec 1st/2nd kind, Crouzeix–Raviart, Arnold-Winther, $\mathcal{P}_q\Lambda^k$, $\mathcal{P}_q^-\Lambda^k$, Morley, Hermite, Argyris, Bell, ...



Parallel computing

Distributed computing via MPI:

```
mpirun -n 32 python myprog.py
```

Shared memory via OpenMP:

```
# In program
parameters['num_threads'] = Q
```

Automated error control

Input

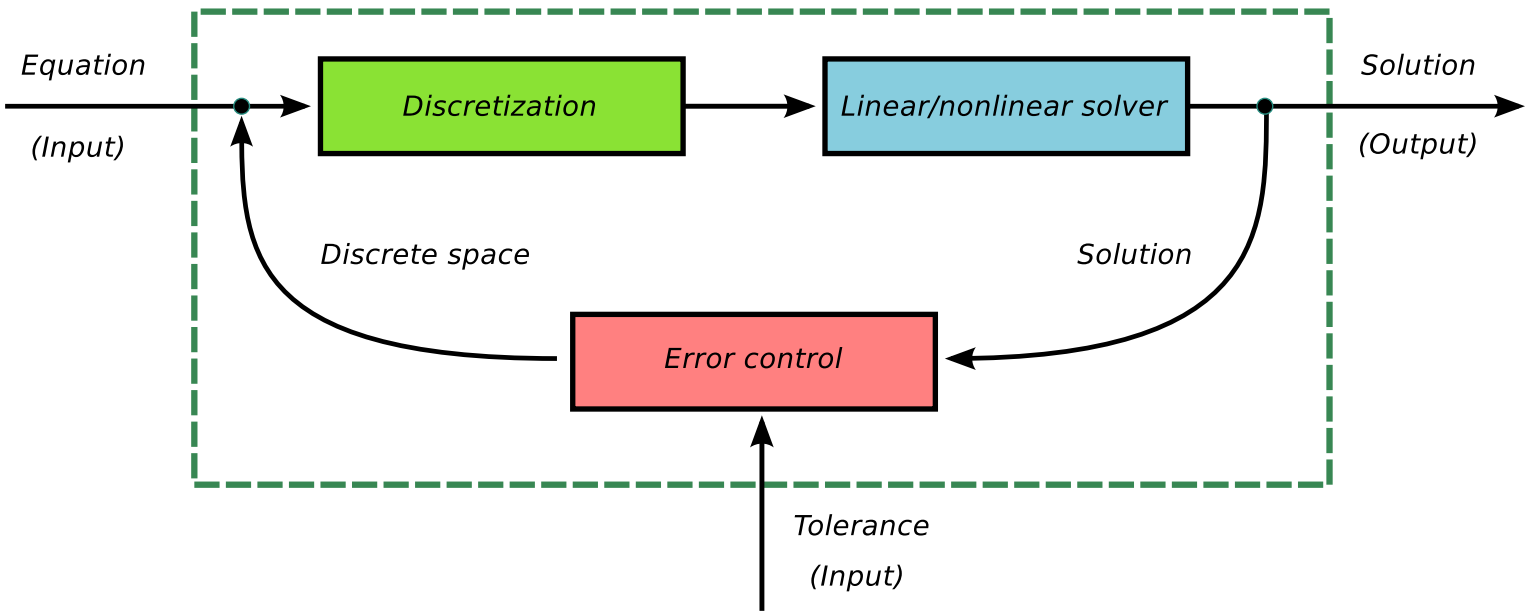
- $a(u, v) = L(v)$ or $F(u; v) = 0$
- Goal $\mathcal{M}(u)$
- $\epsilon > 0$

Output

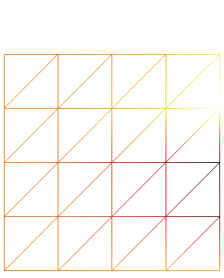
u such that

$$\|\mathcal{M}(u_e) - \mathcal{M}(u)\| \leq \epsilon$$

(u_e: exact solution)



FEniCS automatatically generates a posteriori error estimators and refinement indicators

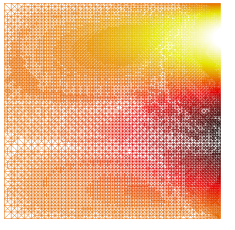


Solve

u_h

Estimate

$\eta_h < \epsilon$



Indicate

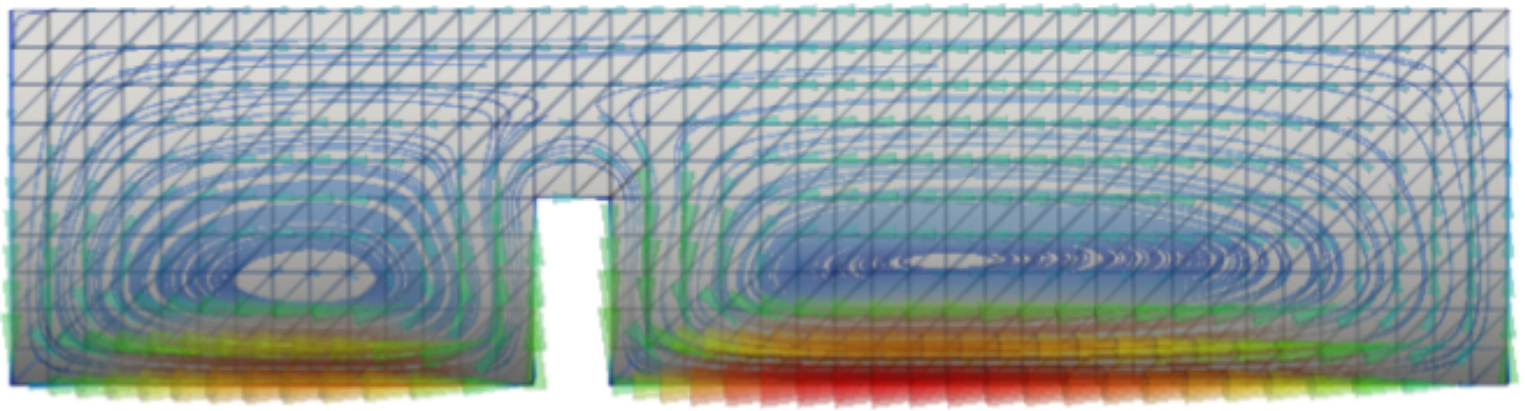
$\{\eta_T\}_{T \in \mathcal{T}_h}$

Refine

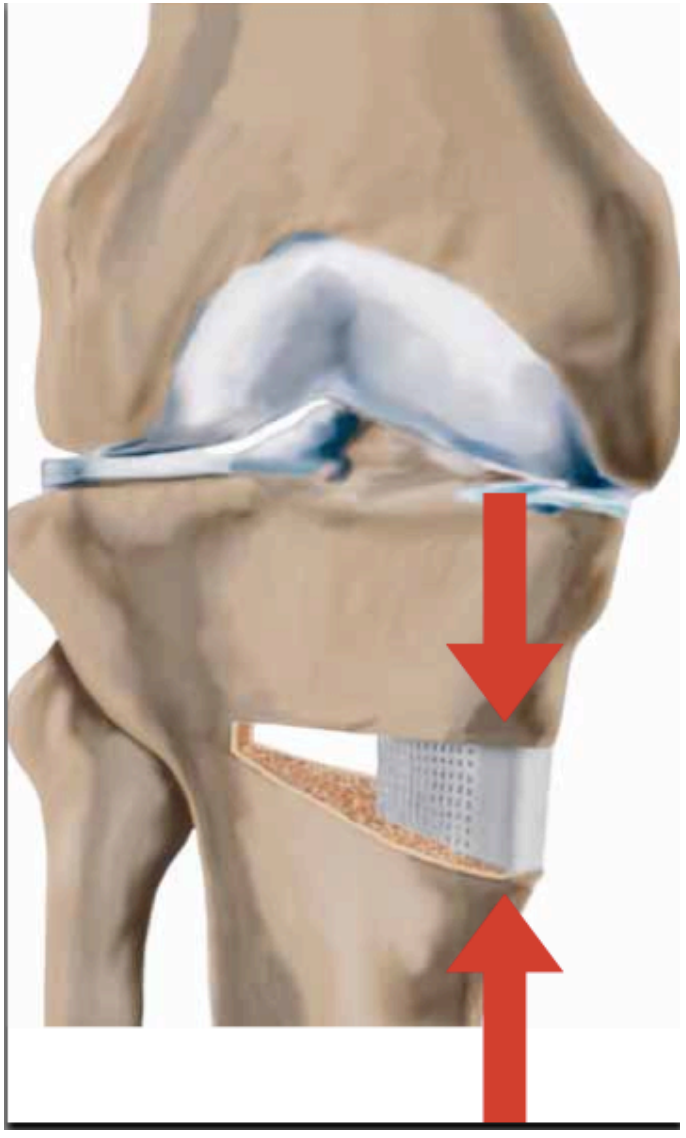
Dual

Error control: just define the goal and the tolerance

```
# Define variational form as usual
F = ....

# Define goal functional
M = dot(mu*(grad(u) + grad(u).T), n)*ds(FLAP)
tol = 1E-3

solve(F == 0, u, bc, M=M, tol=tol)
```

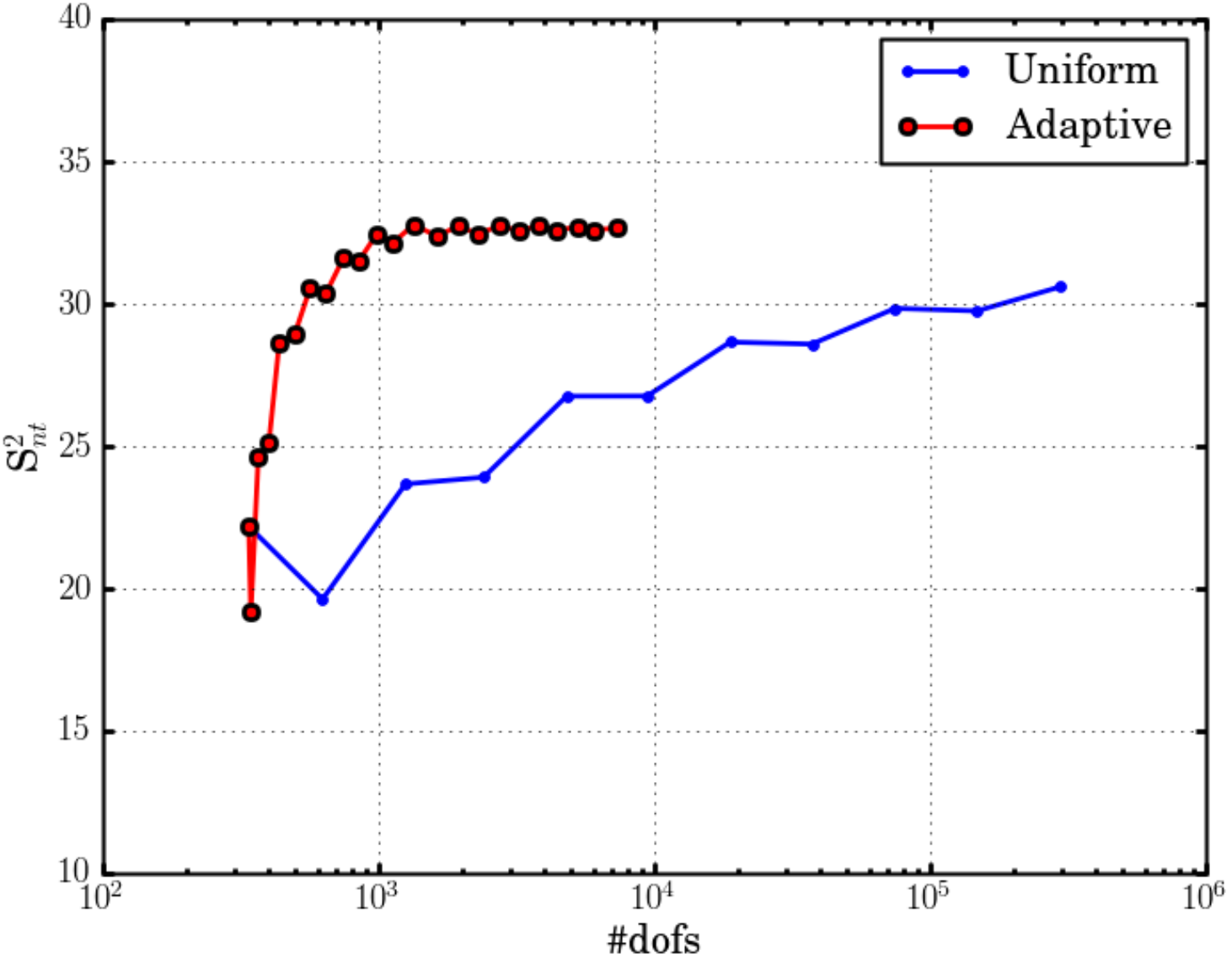


Example: compute shear stress in a bone implant

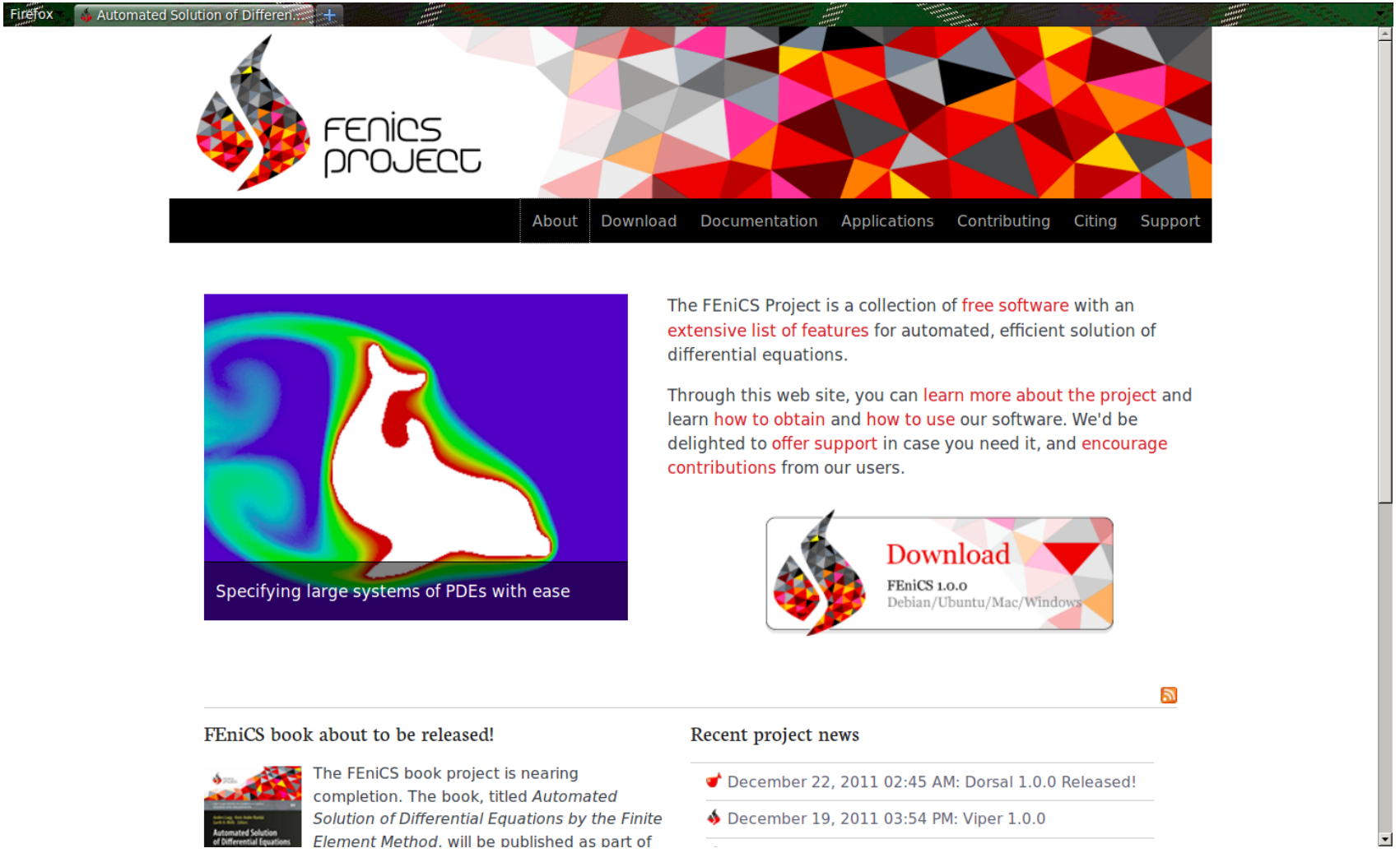


- Polymer-fluid mixture
- Nonlinear hyperelasticity
- Complicated constitutive law
- Novel mixed displacement-stress discretization via Arnold-Winther element

Adaptivity pays off – but would be really difficult to implement by hand in this case



fenicsproject.org



FEniCS is easy to install

- Easiest on Ubuntu (Debian):
  `sudo apt-get install fenics`
- Mac OS X drag and drop installation (.dmg file)
- Windows binary installer
- Automated installation from source (compile & link)

FEniCS is a multi-institutional project

- Initiated 2003 by Univ. of Chicago and Chalmers Univ. of Technology (Ridgway Scott and Claes Johnson)
- Important contributions from
  - Univ.of Chicago (Rob Kirby, Andy Terrel, Matt Knepley, R. Scott)
  - Chalmers Univ. of Technology (Anders Logg, Johan Hoffman, Johan Janson)
  - Delft Univ. of Technology (Garth Wells, Kristian Oelgaard)
- Current key institutions:
  - Simula Research Laboratory (Anders Logg, Marie Rognes, Martin Alnæs, Johan Hake, Kent-Andre Mardal, ...)
  - Cambridge University (Garth Wells, ...)
- About 20 active developers
- Lots of application developers

What are the problems with computing turbulent flows?

Three classes of models:

- Direct Numerical Simulation
- Large Eddy Simulation
- The jungle of Reynolds-Averaged Navier-Stokes (RANS) models: k-ϵ, k-ω, v^2-f, various tensor models, ...

Should be easy to implement and compare...

- Various models
- Various linearizations
- Coupled vs. segregated solution
- Picard vs. Newton iteration

Example: k-ϵ model (unknowns: $\boldsymbol{u}$, p, k, ϵ)

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{1}{\varrho} \nabla p + \nu \nabla^2 \boldsymbol{u} + \boldsymbol{f} - \nabla \cdot \overline{\boldsymbol{u}' \boldsymbol{u}'}$$

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

$$\nabla \cdot \overline{\boldsymbol{u}' \boldsymbol{u}'} = -2 \frac{k^2}{\epsilon} \frac{1}{2} (\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T) + \frac{2}{3} k \mathbf{I}$$

$$\frac{\partial k}{\partial t} + \boldsymbol{u} \cdot \nabla k = \nabla \cdot (\nu_k \nabla k) + P_k - \epsilon - D,$$

$$\frac{\partial \epsilon}{\partial t} + \boldsymbol{u} \cdot \nabla \epsilon = \nabla \cdot (\nu_\epsilon \nabla \epsilon) + (C_{\epsilon 1} P_k - C_{\epsilon 2} f_2 \epsilon) \frac{\epsilon}{k} + E$$

$$\epsilon = 2\nu \overline{\boldsymbol{s} : \boldsymbol{s}}, \quad \boldsymbol{s} = \frac{1}{2} (\nabla \boldsymbol{u}' + (\nabla \boldsymbol{u}')^T)$$

$$\nu_k = \nu + \frac{\nu_T}{\sigma_k}$$

$$\vdots$$

Multi-physics problems with large systems of PDEs

Solve a system of PDEs, e.g.,

$$\mathcal{L}(u_1, u_2, ..., u_6) = 0$$

in some grouping into subsystems, e.g.,

$$\begin{aligned} \mathcal{L}_1(u_1, u_2) &= 0 \\ \mathcal{L}_2(u_3) &= 0 \\ \mathcal{L}_3(u_4, u_5, u_6) &= 0 \end{aligned}$$

Segregeted solve (iteration) between subsystems

- One scalar/vector PDE solver is compact in FEniCS
- Large PDE systems require tedious, repetitive code
- Let's automate!

Multi-physics problems with large systems of PDEs

Solve a system of PDEs, e.g.,

$$\mathcal{L}(u_1, u_2, ..., u_6) = 0$$

in some grouping into subsystems, e.g.,

$$\begin{aligned} \mathcal{L}_1(u_1, u_2) &= 0 \\ \mathcal{L}_2(u_3) &= 0 \\ \mathcal{L}_3(u_4, u_5, u_6) &= 0 \end{aligned}$$

Segregeted solve (iteration) between subsystems

- One scalar/vector PDE solver is compact in FEniCS
- Large PDE systems require tedious, repetitive code
- Let's automate!

Multi-physics problems with large systems of PDEs

Solve a system of PDEs, e.g.,

$$\mathcal{L}(u_1, u_2, ..., u_6) = 0$$

in some grouping into subsystems, e.g.,

$$\begin{aligned} \mathcal{L}_1(u_1, u_2) &= 0 \\ \mathcal{L}_2(u_3) &= 0 \\ \mathcal{L}_3(u_4, u_5, u_6) &= 0 \end{aligned}$$

Segregeted solve (iteration) between subsystems

- One scalar/vector PDE solver is compact in FEniCS
- Large PDE systems require tedious, repetitive code
- Let's automate!

Download from launchpad.net: bzr branch lp:cbcpdesys

CBC.PDESys - Specify large systems of PDEs with ease

Overview | Code | Bugs | Blueprints | Translations | Answers

Registered 2011-10-13 by Mikael Mortensen

CBC.PDESys is a Python package, built on top of FEniCS, for specifying and solving large systems of nonlinear Partial Differential Equations (PDEs) with very compact, flexible and reusable code (see, e.g., http://fenicsproject.org/featured/2011/pdesys.html). The package is completely general and targets any system of PDEs. One specific feature of CBC.PDESys is that all PDESystem classes (e.g., Navier-Stokes) integrate seamlessly and can be solved together simply by adding them to a Problem class. The primary focus of CBC.PDESys today is on Computational Fluid Dynamics and turbulence modeling (RANS) through the subpackage CBC.CFD. Implemented turbulence models are currently the standard k-epsilon, Spalart Allmaras, V2F and elliptic relaxation models. CBC.CFD also comes with some highly optimized incompressible Navier-Stokes solvers. The software packages are developed at the Center for Biomedical Computing at Simula Research Laboratory in Oslo, Norway. CBC.PDESys is distributed freely in the hope that it will be useful, but without any warranty.

Project information

Part of:
The FEniCS Apps Project

Maintainer:
Mikael Mortensen

Driver:
Not yet selected

Development focus:
trunk series

lp:cbcpdesys
Browse the code

Programming Languages:
Python

Licenses:
GNU GPL v3

RDF metadata

Series and milestones

View full history

trunk
2011-12-09
1.0

CBC.PDESys trunk series is the current focus of development

Principles for solving a system of PDEs $\mathcal{L}(u_1, u_2, ...) = 0$

- List the names of unknowns: `u1`, `u2`, ...
- Automatically create standard FEniCS objects:

```
FunctionSpace V_u1 (mesh, element_u1, degree_u1)
Function      u1_  (V_u1)
TrialFunction u1   (V_u1)
TestFunction  v_u1 (V_u1)
```

- Let the user supply the form:

```
inner(u1*grad(u2), grad(v_u2)) ...
```

- Let the user specify degree of implicitness:

```
[['u1', 'u2'], ['u3'], ['u4', 'u5', 'u6']]
```

  = 3 PDE systems: 2×2, scalar, 3×3

- Automatically create mixed function spaces, linear systems, Jacobians, call up nonlinear solves, etc.
- Easy to optimize (precomputed matrices, etc.)

What needs to be programmed by a user?

- Name the unknowns
- `Problem` instance containing problem-specific parameters
- `PDESystem` instance with list of unknowns grouped into subsystems
- For each subsystem, a `PDESubSystem` subclass with method `form` to specify a variational form

```
class NavierStokes(PDESubSystem):
    def form(self, u, v_u, u_, u_1, p, v_p, nu, dt, f,
             **kwargs):
        return (1/dt)*inner(u - u_1, v_u)*dx +   \
               inner(u_1*nabla_grad(u_1), v_u) + \
               nu*inner(grad(u), grad(v_u))*dx - \
               inner(p, div(v_u))*dx + inner(div(U), v_p)*dx
```

Inner details & ideas

- The user defines the notation (names of unknowns + forms)
- The library must be general
- Callbacks to the user must have argument names corresponding to the user's notation

- No classical code generation (just Python)
- Heavy use of string operations, `eval` and `**kwargs`
- Store data as dicts (and attributes for convenience)
- Send the namespaces (user's and generated) around
- Rely on simple naming conventions: `u`, `u_`, `v_k`, `V_u`, ...

Currently implemented solvers for CFD

Navier-Stokes solvers

- Fully coupled
- Segregated
  - Chorin
  - Incremental pressure correction

RANS models

- Spalart-Allmaras
- $k - \epsilon$
  - Low-Reynolds models
  - Standard $k - \epsilon$
- $v^2 - f$ model
  - Original
  - Lien-Kalizin
- Elliptic relaxation
  - LRR-IP
  - SSG

A k equation in a turbulence model

PDE for turbulent kinetic energy k (unknowns: $\mathbf{u}, k, \epsilon$)

$$0 = -\mathbf{u} \cdot \nabla k + \nabla \cdot (\nu_k(\mathbf{u}, k, \epsilon) \nabla k) + P_k(\mathbf{u}, k, \epsilon) - \epsilon$$

Typical linearization (underscore subscript: old value)

$$0 = -\mathbf{u}_- \cdot \nabla k + \nabla \cdot (\nu_{k-} \nabla k) + P_{k-} - \epsilon$$

Variational form and Python code: code $\approx$ math

Variational form (k and v_k are trial and test functions)

$$F_k = -\int_\Omega \mathbf{u}\_ \cdot \nabla k \, v_k \, \mathrm{d}x - \int_\Omega \nu_k\_ \nabla k \cdot \nabla v_k \, \mathrm{d}x + \int_\Omega (P_k\_ - \epsilon) \, v_k \, \mathrm{d}x$$

Corresponding code

```
F_k = - inner(dot(u_, grad(k)), v_k)*dx \
      - nu_k_*inner(grad(k), grad(v_k))*dx \
      + (P_k_ - e)*v_k*dx
```

Linearization, i.e., implicit vs explicit treatment is a matter of inserting or removing an underscore

Implicit treatment of ϵ in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e*v_k*dx`

Explicit treatment of ϵ for decoupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon\_ v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*v_k*dx`

Explicit treatment of ϵ, but implicit term in k eq.:

$F_k = \ldots + \int_\Omega \epsilon\_ \frac{k}{k\_} v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*k/k_*v_k*dx`

Weighted combination in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega ((1-w)\epsilon\_k + w\epsilon k\_)\frac{1}{k\_} v_k \, \mathrm{d}x \quad \rightarrow$

`(1/k_)*((1-w)*e_*k + w*e*k_)*v_k*dx`

Linearization, i.e., implicit vs explicit treatment is a matter of inserting or removing an underscore

Implicit treatment of ϵ in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e*v_k*dx`

Explicit treatment of ϵ for decoupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon\_ v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*v_k*dx`

Explicit treatment of ϵ, but implicit term in k eq.:

$F_k = \ldots + \int_\Omega \epsilon\_ \frac{k}{k\_} v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*k/k_*v_k*dx`

Weighted combination in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega ((1-w)\epsilon\_ k + w\epsilon k\_) \frac{1}{k\_} v_k \, \mathrm{d}x \quad \rightarrow$

`(1/k_)*((1-w)*e_*k + w*e*k_)*v_k*dx`

Linearization, i.e., implicit vs explicit treatment is a matter of inserting or removing an underscore

Implicit treatment of ϵ in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e*v_k*dx`

Explicit treatment of ϵ for decoupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon\_ v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*v_k*dx`

Explicit treatment of ϵ, but implicit term in k eq.:

$F_k = \ldots + \int_\Omega \epsilon\_ \frac{k}{k\_} v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*k/k_*v_k*dx`

Weighted combination in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega ((1-w)\epsilon\_ k + w\epsilon k\_) \frac{1}{k\_} v_k \, \mathrm{d}x \quad \rightarrow$

```
(1/k_)*((1-w)*e_*k + w*e*k_)*v_k*dx
```

Linearization, i.e., implicit vs explicit treatment is a matter of inserting or removing an underscore

Implicit treatment of ϵ in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e*v_k*dx`

Explicit treatment of ϵ for decoupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon\_ v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*v_k*dx`

Explicit treatment of ϵ, but implicit term in k eq.:

$F_k = \ldots + \int_\Omega \epsilon\_ \frac{k}{k\_} v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*k/k_*v_k*dx`

Weighted combination in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega ((1-w)\epsilon\_ k + w\epsilon k\_) \frac{1}{k\_} v_k \, \mathrm{d}x \quad \rightarrow$

`(1/k_)*((1-w)*e_*k + w*e*k_)*v_k*dx`

Linearization, i.e., implicit vs explicit treatment is a matter of inserting or removing an underscore

Implicit treatment of ϵ in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e*v_k*dx`

Explicit treatment of ϵ for decoupled k-ϵ system:

$F_k = \ldots + \int_\Omega \epsilon\_ v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*v_k*dx`

Explicit treatment of ϵ, but implicit term in k eq.:

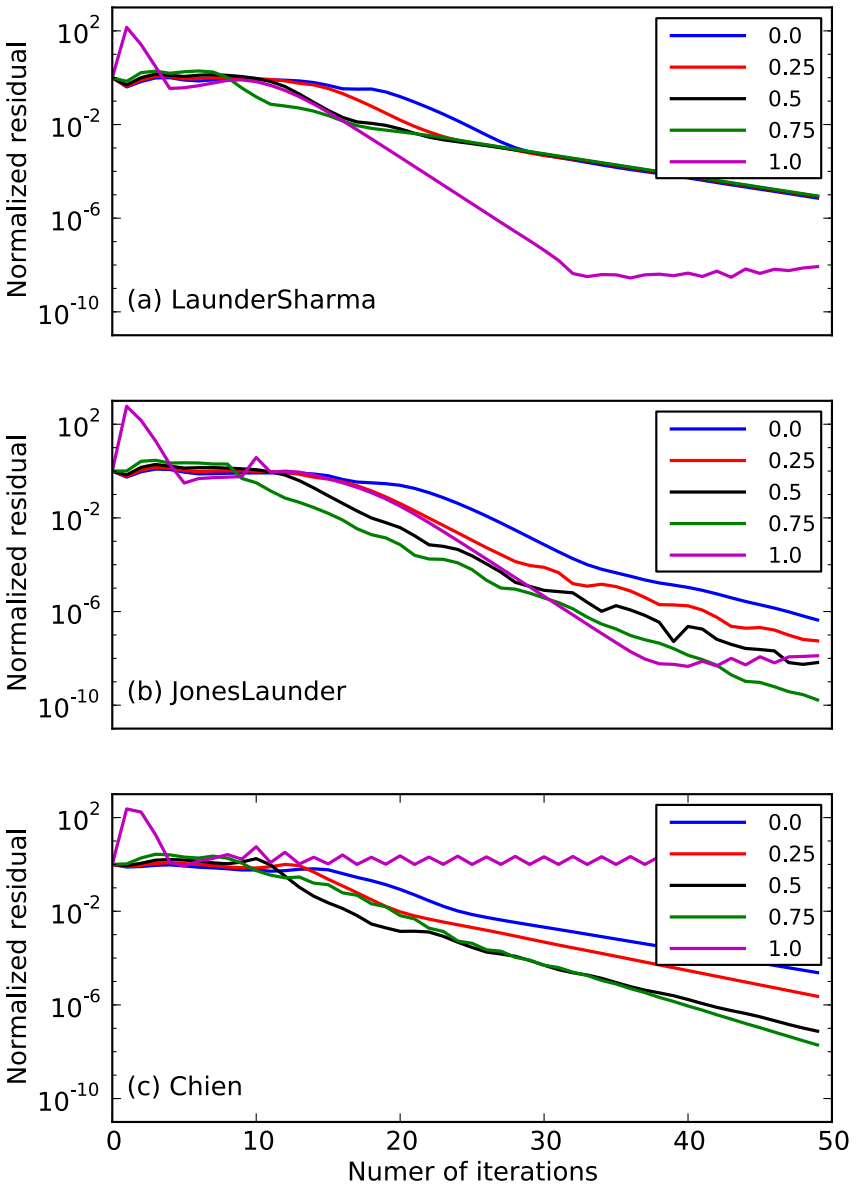
$F_k = \ldots + \int_\Omega \epsilon\_ \frac{k}{k\_} v_k \, \mathrm{d}x \quad \rightarrow \quad$ `e_*k/k_*v_k*dx`

Weighted combination in coupled k-ϵ system:

$F_k = \ldots + \int_\Omega ((1 - w)\epsilon\_ k + w\epsilon k\_) \frac{1}{k\_} v_k \, \mathrm{d}x \quad \rightarrow$

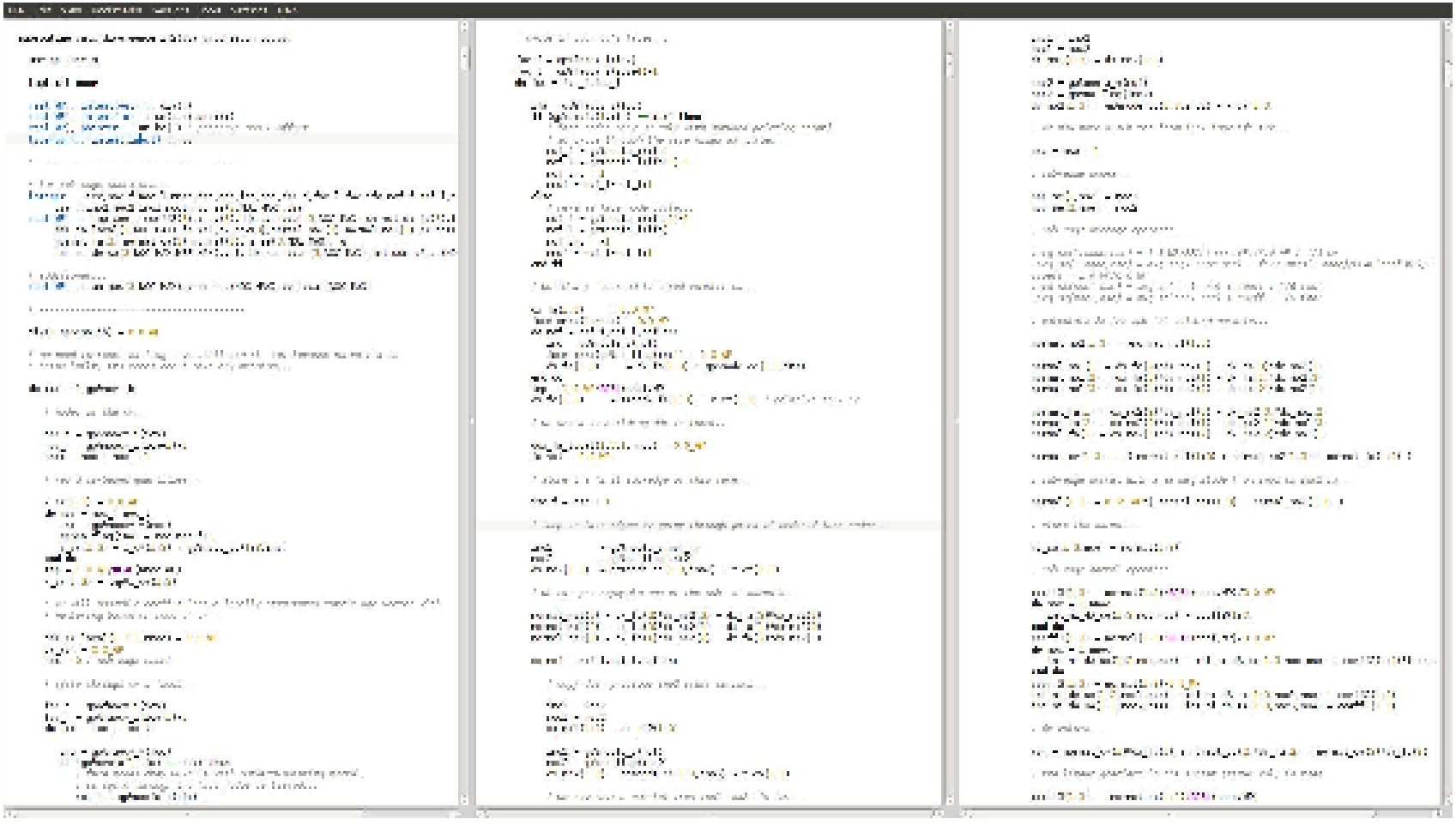
```
(1/k_)*((1-w)*e_*k + w*e*k_)*v_k*dx
```

Impact of w on convergence of nonlinear iterations



Traditional CFD codes require much more programming

- Compute $\nabla \cdot \boldsymbol{u}$ using Fortran90 in CDP (Stanford):
- FEniCS vs CDP efficiency: $\times 2$



Proof of concept: 18 highly coupled nonlinear PDEs

The elliptic relaxation model:

$$\frac{\partial R_{ij}}{\partial t} + u_k \frac{\partial R_{ij}}{\partial x_k} + \frac{\partial T_{kij}}{\partial x_k} = G_{ij} + P_{ij} - \varepsilon_{ij}$$

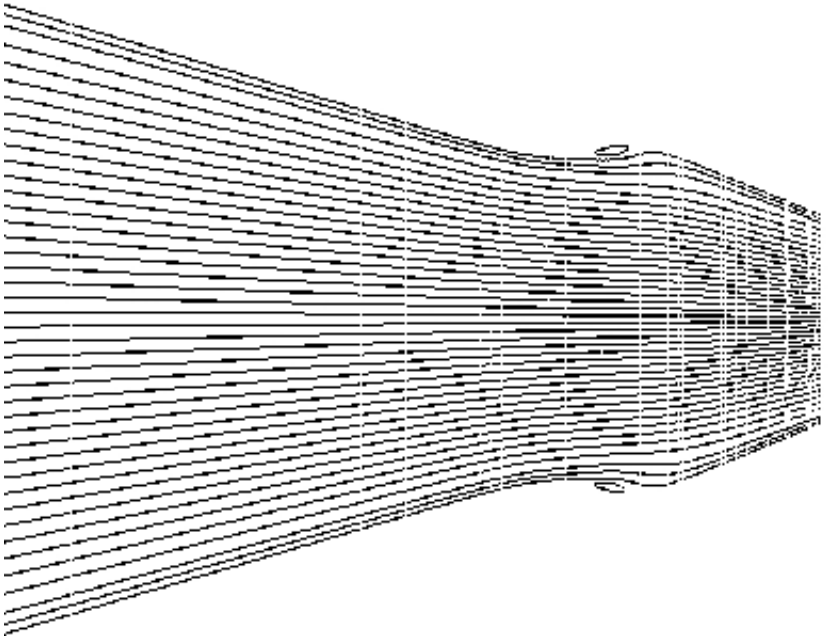
$$L^2 \nabla^2 f_{ij} - f_{ij} = -\frac{G^h_{ij}}{k} - \frac{2A_{ij}}{T}$$

$+$ standard eqs. for $\mathbf{u}, p, k, \epsilon$

Coupled implementation of variational forms:

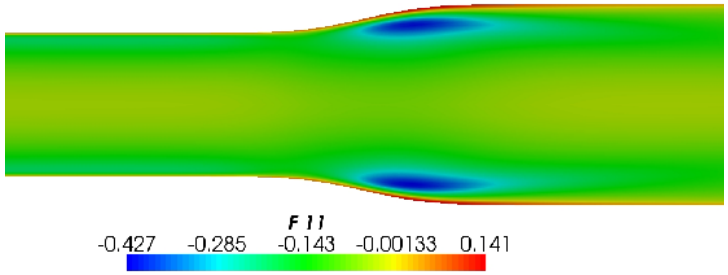
```
class RF_1(TurbModel):

    def form(self, R, R_, v_R, k_, e_, P_, nu, u_, f, f_, v_f,
             A_, Gh, Cmu, T_, L_, **kwargs):

        Fr = inner(dot(grad(R), u_), v_R)*dx + nu*inner(grad(R), grad(v_R))*dx  \
             + inner(Cmu*T_*dot(grad(R), R_), grad(v_R) )*dx
             - inner(k_*f, v_R)*dx - inner(P_, v_R )*dx + inner(R*e_*(1./k_), v_R)*dx \

        Ff = inner(grad(f), grad(L_**2*v_f))*dx + inner(f , v_f)*dx \
             - (1./k_)*inner(Gh , v_f)*dx - (2./T_)*inner(A_ , v_f)*dx

        return Fr + Ff
```

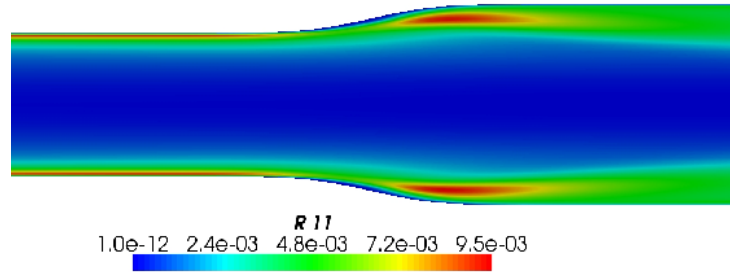
Diffusor as validation problem

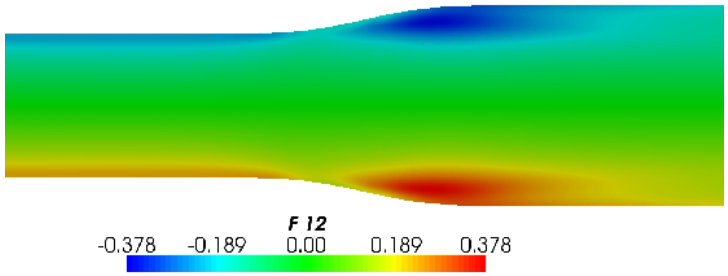


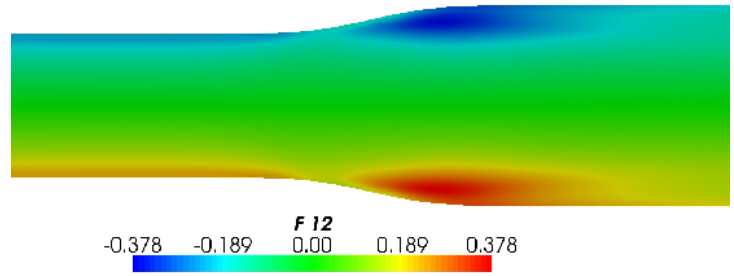
Consider a diffusor with flow through an expanding channel:

- $Re_\tau = 395$
- Separation and recirculation
- Elliptic relaxation model









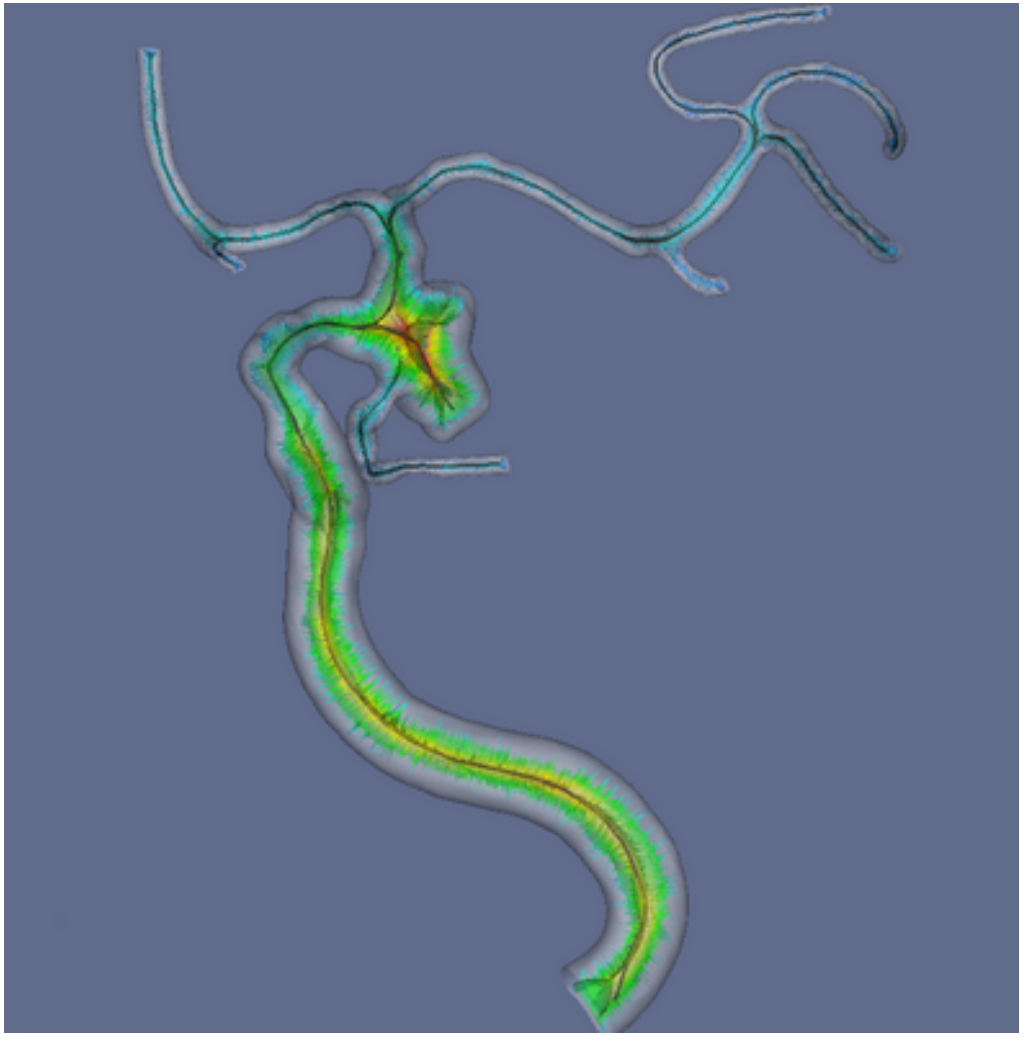
Redistribution parameter

Reynolds stresses

Ongoing work: libraries built on FEniCS

- Uncertainty quantification: (generalized) multivariate polynomial chaos *with dependent variables*
  - non-intrusive
  - intrusive (new PDEs :-)
- PDE constrained optimization for inverse problems (define Lagrangian, autogenerate the rest)
- Block preconditioning
- Automated adaptive time integration for PDEs

Cerebral blood flow and the impact on stroke



movie

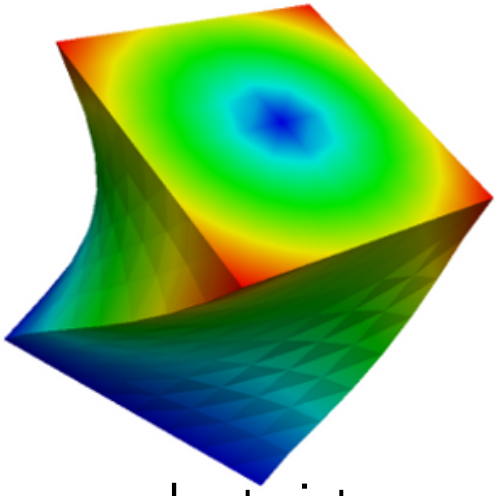
Summary

Follow links and read more

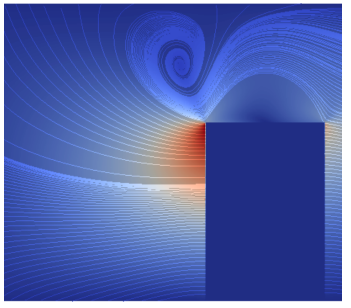
- FEniCS: Python FEM math syntax with HPC
- `pdesys` package: flexible syntax for PDE systems
- Vast collection of CFD solvers
- More info
- The FEniCS tutorial
- The FEniCS book on launchpad.net and Springer

cbc.solve: collection of FEniCS solvers/examples

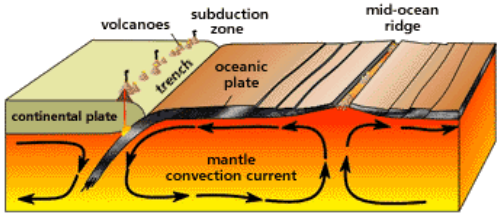
cbc.flow



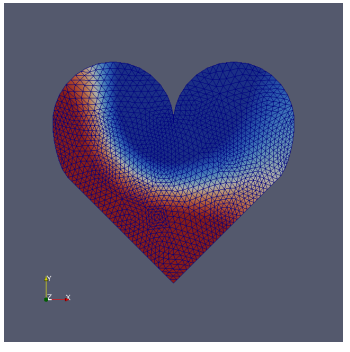
cbc.twist



cbc.swing



cbc.rock



cbc.beat

$$\begin{pmatrix} A & B^\top \\ B & 0 \end{pmatrix}$$

cbc.block

Download from launchpad.net: bzr branch lp:cbcpdesys

CBC.PDESys - Specify large systems of PDEs with ease

Overview | Code | Bugs | Blueprints | Translations | Answers

Registered 2011-10-13 by Mikael Mortensen

CBC.PDESys is a Python package, built on top of FEniCS, for specifying and solving large systems of nonlinear Partial Differential Equations (PDEs) with very compact, flexible and reusable code (see, e.g., http://fenicsproject.org/featured/2011/pdesys.html). The package is completely general and targets any system of PDEs. One specific feature of CBC.PDESys is that all PDESystem classes (e.g., Navier-Stokes) integrate seamlessly and can be solved together simply by adding them to a Problem class. The primary focus of CBC.PDESys today is on Computational Fluid Dynamics and turbulence modeling (RANS) through the subpackage CBC.CFD. Implemented turbulence models are currently the standard k-epsilon, Spalart Allmaras, V2F and elliptic relaxation models. CBC.CFD also comes with some highly optimized incompressible Navier-Stokes solvers. The software packages are developed at the Center for Biomedical Computing at Simula Research Laboratory in Oslo, Norway. CBC.PDESys is distributed freely in the hope that it will be useful, but without any warranty.

Project information

Part of:
The FEniCS Apps Project

Maintainer:
Mikael Mortensen

Driver:
Not yet selected

Development focus:
trunk series

lp:cbcpdesys
Browse the code

Programming Languages:
Python

Licenses:
GNU GPL v3

RDF metadata

Series and milestones

View full history

trunk
2011-12-09
1.0

CBC.PDESys trunk series is the current focus of development