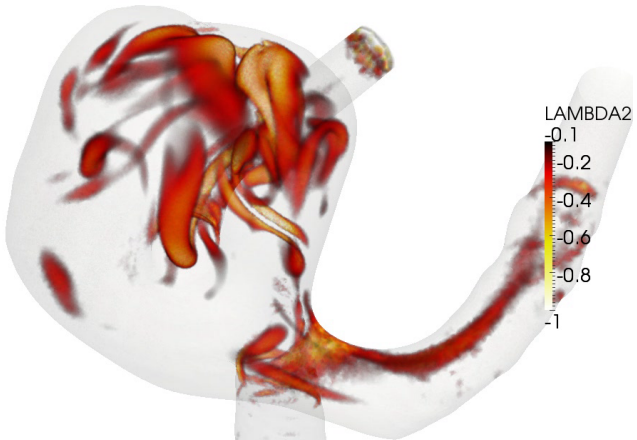
# The FEniCS Project and dolfin-adjoint

Marie E. Rognes

Biomedical Computing Department

Center for Biomedical Computing

Simula Research Laboratory

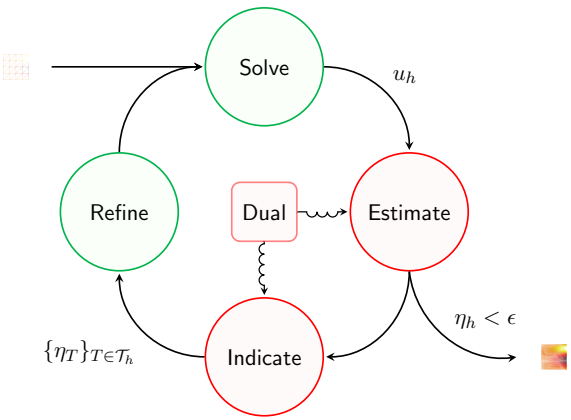


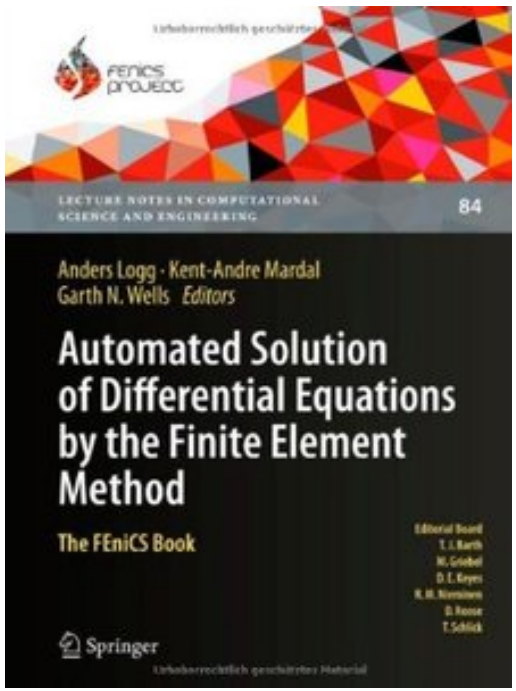

Center for Biomedical Computing

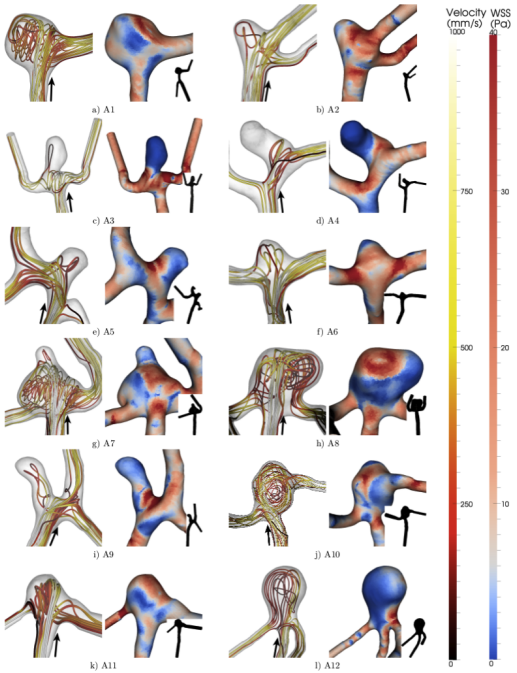
[ simula . research laboratory ]

# Developing the next generation simulation technology to solve problems affecting human health and disease

## The Biomedical Computing department @ Simula



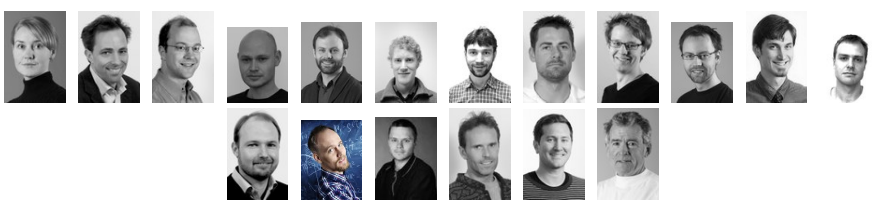




[Evju, Valen-Sendstad, Mardal; 2013]

Center for Biomedical Computing

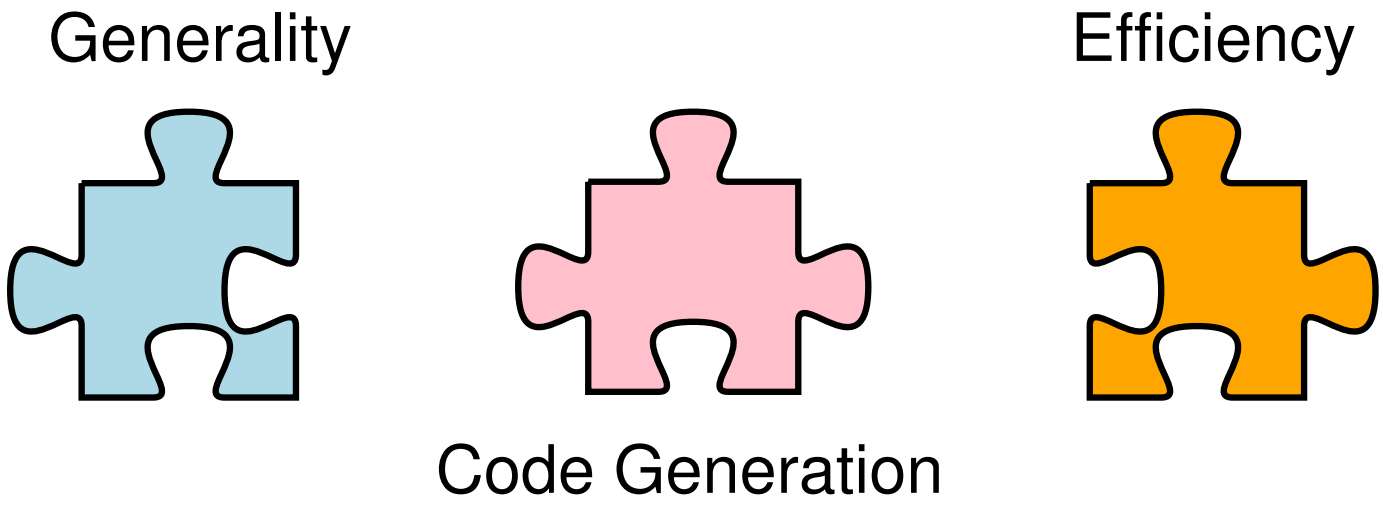
Norwegian Centre of Excellence




# An introduction to FEniCS by example

www.fenicsproject.org

# Automated code generation solves long-standing dilemma of how to combine generality, efficiency, simplicity and reliability




- Generality through *abstraction*
- Efficiency through *code generation*, *adaptivity*, *parallelism*
- Reliability through *error control* and *uncertainty quantification*
- Simplicity through *high level scripting*, *automation*

# The FEniCS Project provides an automated programming environment: fenicsproject.org


www.fenicsproject.org

# FEniCS code can be readable, scale with mathematical complexity, and provide high-performance

## Stokes with nonlinear viscosity

Given temperature $T$, find velocity $u$ and pressure $p$ such that

$$-\operatorname{div}(2\nu(u,T)\varepsilon(u) + pI) = \operatorname{Ra} T\, g$$
$$\operatorname{div} u = 0$$

in $\Omega$ with (for instance)

$$\nu(u,T) = e^{-\alpha T}\,(u \cdot u).$$

## Finite element formulation

Given temperature $T$, find $(u,p) \in W = V \times Q$ such that

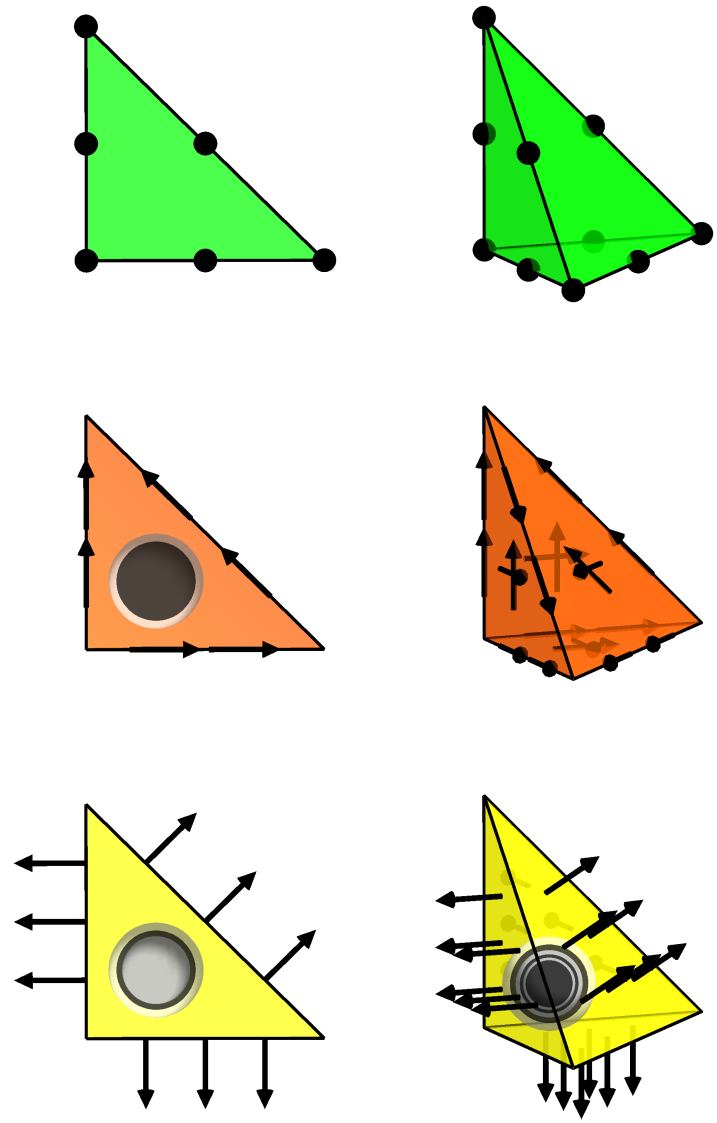
$$\int_\Omega 2\nu(u,T)\varepsilon(u) \cdot \varepsilon(v) + \operatorname{div}(v)\, p + \operatorname{div}(u)\, q - \operatorname{Ra} T g \cdot v \,\mathrm{d}x = 0$$

for all $(v,q) \in W$.

```
from dolfin import *

# Define viscosity
def nu(u, T):
  return exp(-10.0*T)*dot(u, u)

# Define element spaces
mesh = Mesh(...)
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# Define functions
T = Expression("...")
w = Function(W)
(u, p) = split(w)
(v, q) = TestFunctions(W)

# Define equation
F = (2*nu(u, T)*inner(eps(u), eps(v))
     + div(v)*p + div(u)*q
     + Ra*T*v[1])*dx
bcs = ...

# Solve F = 0 w.r.t w
solve(F == 0, w, bcs)
```

# FEniCS provides a wide range of (mixed) finite element spaces

```
from dolfin import *

# Define viscosity
def nu(u, T):
  return exp(-10.0*T)*dot(u, u)

# Define element spaces
mesh = Mesh(...)
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# Define functions
T = Expression("...")
w = Function(W)
(u, p) = split(w)
(v, q) = TestFunctions(W)

# Define equation
F = (2*nu(u, T)*inner(eps(u), eps(v))
     + div(v)*p + div(u)*q
     + Ra*T*v[1])*dx
bcs = ...

# Solve F = 0 w.r.t w
solve(F == 0, w, bcs)
```

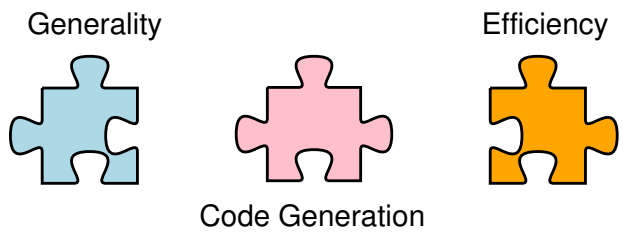


Kirby 2004; Ølgaard, Logg, and Wells 2008; R., Kirby and Logg 2009

# FEniCS provides an expressive form language close to mathematical syntax

```
from dolfin import *

# Define viscosity
def nu(u, T):
  return exp(-10.0*T)*dot(u, u)

# Define element spaces
mesh = Mesh(...)
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# Define functions
T = Expression("...")
w = Function(W)
(u, p) = split(w)
(v, q) = TestFunctions(W)

# Define equation
F = (2*nu(u, T)*inner(eps(u), eps(v))
     + div(v)*p + div(u)*q
     + Ra*T*v[1])*dx
bcs = ...

# Solve F = 0 w.r.t w
solve(F == 0, w, bcs)
```

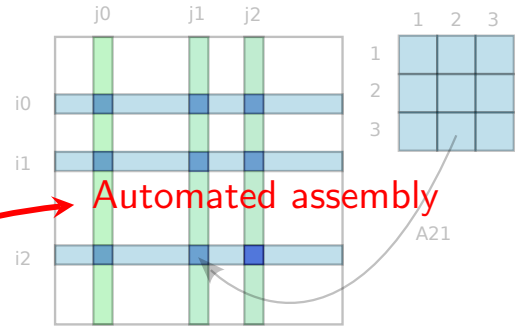
Language for variational forms



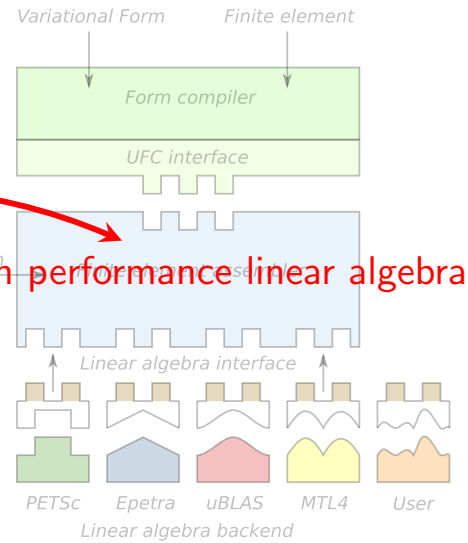

Kirby and Logg 2006; Alnæs 2012

# FEniCS provides automated form assembly over finite element meshes and numerical linear algebra

```
from dolfin import *

# Define viscosity
def nu(u, T):
  return exp(-10.0*T)*dot(u, u)

# Define element spaces
mesh = Mesh(...)
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# Define functions
T = Expression("...")
w = Function(W)
(u, p) = split(w)
(v, q) = TestFunctions(W)

# Define equation
F = (2*nu(u, T)*inner(eps(u), eps(v))
     + div(v)*p + div(u)*q
     + Ra*T*v[1])*dx
bcs = ...

# Solve F = 0 w.r.t w
solve(F == 0, w, bcs)
```




Automated assembly




High performance linear algebra

Logg and Wells, 2010; Hake, Logg and Wells 2012

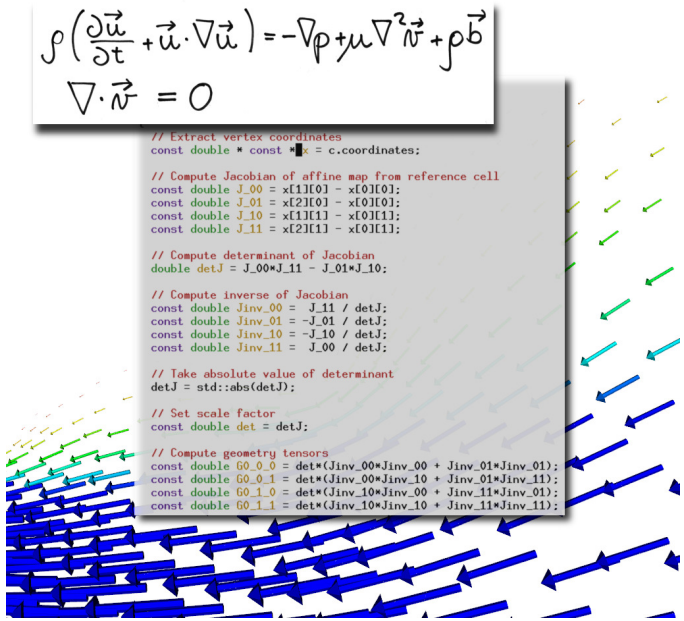
# Generating optimized code from high-level input allows for rapid deployment in new application areas

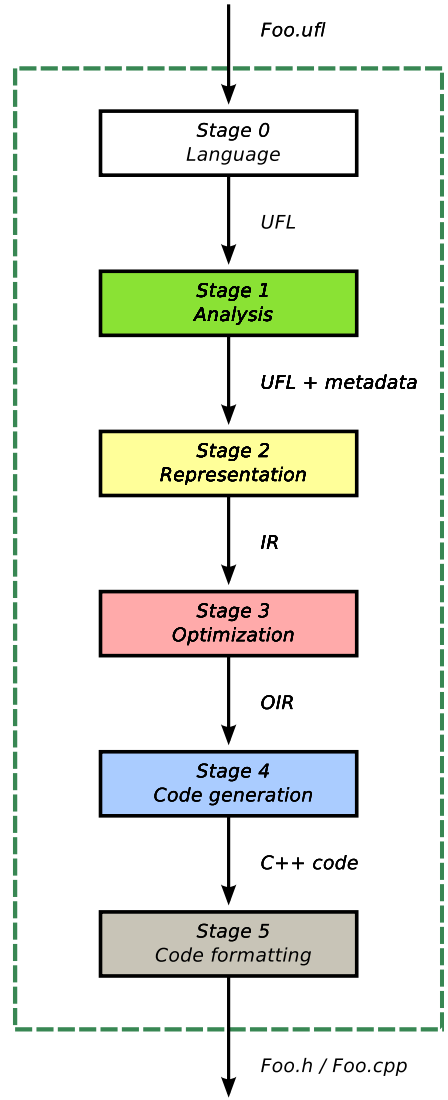
## Input

Equation (variational form(s))

## Output

Efficient application-specific code



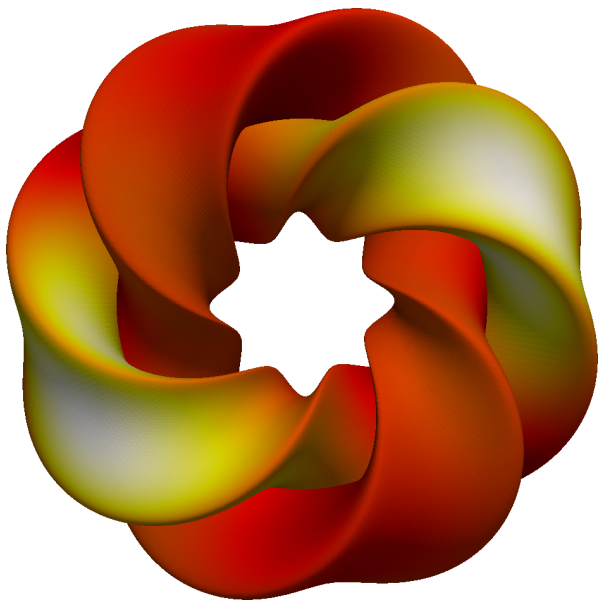
## Automated derivation of transient adjoint models

www.dolfin-adjoint.org

# High-level abstractions allow for optimally efficient large-scale optimization

## Features

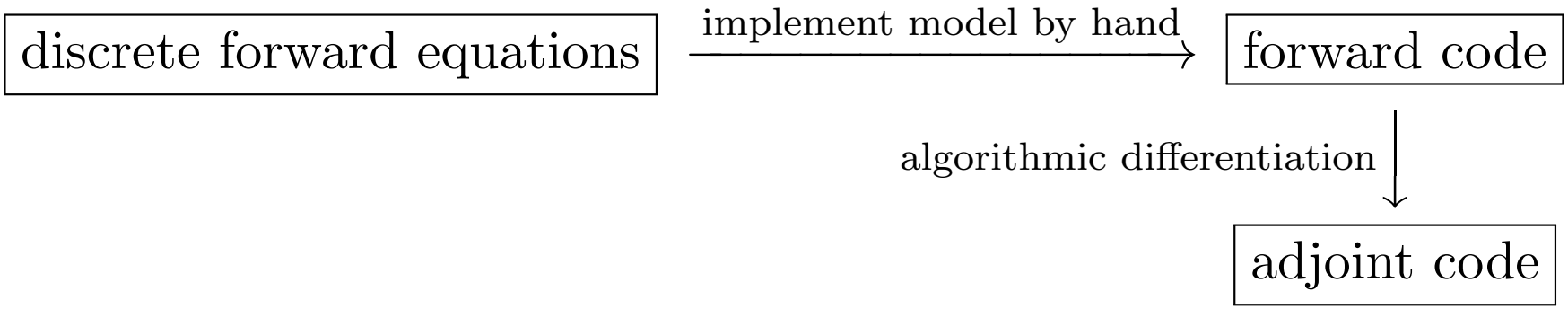
- a non-intrusive interface;
- seamless parallel functionality;
- optimal checkpointing algorithms;
- built-in verification routines
- documentation: `www.dolfin-adjoint.org`
- open source (LGPL)



```
from dolfin import *
from dolfin_adjoint import *

# Define the forward model
mesh = Mesh("klein.xdmf")
V = FunctionSpace(mesh, "CG", 1)
u_old = Function(V)
u = Function(V)
v = TestFunction(V)

g = interpolate(Expression("sin(x[2])*cos(x[1])"), V)
nu = 1.0; T = 1.; t = 0.0; step = 0.1

F = u*v*dx - u_old*v*dx \
    + step*nu*inner(grad(v), grad(u))*dx

u_old.assign(g, annotate=True)
while t <= T:
    solve(F == 0, u)
    u_old.assign(u)
    t += step

# Define the functional and control
J = Functional(inner(u, u)*dx*dt[FINISH_TIME])
m = Control(g)

# Compute and plot the gradient
dJdm = compute_gradient(J, m, project=True)
plot(dJdm, title="Sensitivity", interactive=True)
```

[Farrell, Ham, Funke and R., Automated Derivation of the Adjoint of High-Level Transient Finite Element Programs, SISC, 2013.]

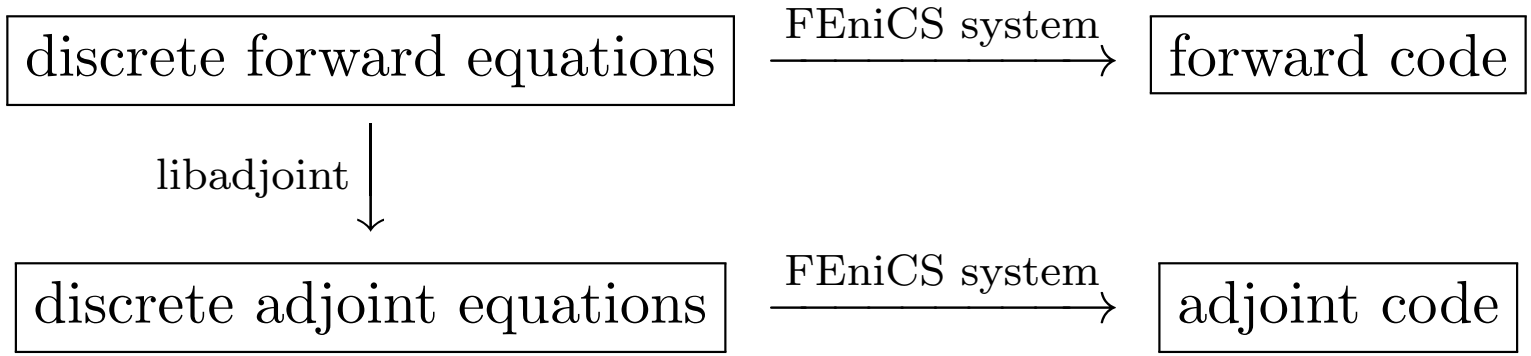
# One traditional approach to developing adjoint models




## Pros/Cons

- Its much easier than doing it by hand
- All of the problems can be solved (with effort)
- Major investment of labour (semi-automatic)
- Adjoint can be very slow (Naumann (2011): $3 - 30\times$ slower)
- Does not naturally work in parallel (manual intervention)
- Checkpointing requires large amount of intervention

# A novel approach to developing adjoint models for forward models implemented in FEniCS




## Cons/Pros

- The problem must be representable in the high-level language
- It does not apply to legacy code.
- The adjoint derivation is totally automatic ($3 - 10$ lines)
- The adjoint is efficient
- The adjoint works naturally in parallel (MPI and OpenMP) and can use checkpointing

[Farrell, Ham, Funke, R.: Automated derivation of the adjoint of high-level transient finite element programs, SISC, 2013]

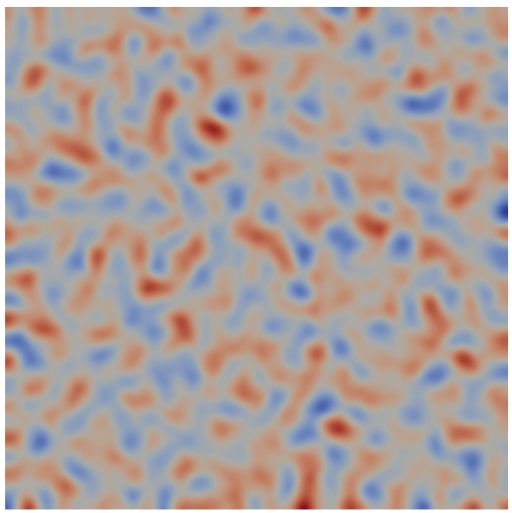
# A (nontrivial) example: the Cahn-Hilliard equation

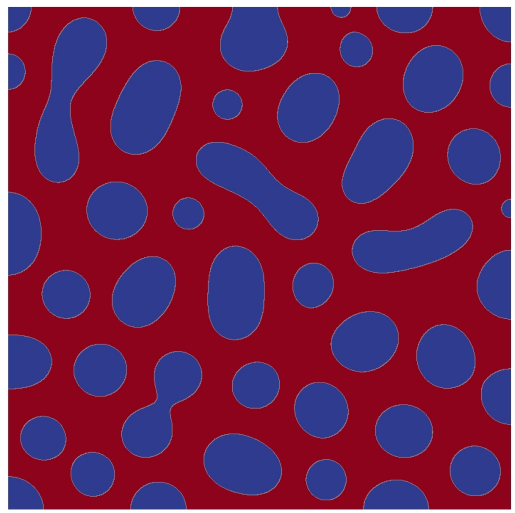
Given an initial concentration $c_0$, find the concentration field $c$ such that

$$\frac{\partial c}{\partial t} - \nabla \cdot M\nabla\left(\frac{df}{dc} - \epsilon^2\nabla^2 c\right) = 0 \quad \text{in } \Omega,$$

$$M\nabla\left(\frac{df}{dc} - \epsilon^2\nabla^2 c\right) = 0, \quad M\epsilon^2\nabla c \cdot \hat{n} = 0 \quad \text{on } \partial\Omega,$$

$$f = 100c^2(1-c)^2, \quad c(t=0) = c_0 \quad \text{on } \Omega.$$





# Tools are available for testing the correctness of the computed adjoint and functional gradient

Figure : Python code snippet demonstrating how to run Taylor test to corroborate gradient and adjoint correctness

```
dJdic = compute_gradient(J, Control(ic))
minimal_rate = taylor_test(J, Control(ic), Jic, dJdic)
```

Table : Taylor test for the Cahn-Hilliard example with Willmore functional. Convergence orders computed without adjoint are $1$ as expected, convergence orders computed with adjoint are $2$ as expected if the gradient is correct.

| $h$ | $\left\|\widehat{W}(\tilde{c_0}) - \widehat{W}(c_0)\right\|$ | order | $\left\|\widehat{W}(\tilde{c_0}) - \widehat{W}(c_0) - \tilde{c_0}^T \nabla \widehat{W}\right\|$ | order |
|---|---|---|---|---|
| $1 \times 10^{-7}$ | $3.4826 \times 10^{-6}$ | | $3.0017 \times 10^{-9}$ | |
| $5 \times 10^{-8}$ | $1.7405 \times 10^{-6}$ | $1.0006$ | $7.4976 \times 10^{-10}$ | $2.0013$ |
| $2.5 \times 10^{-8}$ | $8.7009 \times 10^{-7}$ | $1.0003$ | $1.8737 \times 10^{-10}$ | $2.0005$ |
| $1.25 \times 10^{-8}$ | $4.3499 \times 10^{-7}$ | $1.0002$ | $4.6829 \times 10^{-11}$ | $2.0004$ |
| $6.25 \times 10^{-9}$ | $2.1749 \times 10^{-7}$ | $1.0001$ | $1.1716 \times 10^{-11}$ | $1.9989$ |

# This approach to creating the adjoint model has been observed to be efficient

Table : Comparing runtime of forward and adjoint model: (Nonlinear) Cahn-Hilliard example with Willmore functional

| | Runtime (s) | Ratio |
|---|---|---|
| Forward model | 103.93 | |
| Forward model + annotation | 104.24 | 1.002 |
| Forward model + annotation + adjoint model | 127.07 | 1.22 |

Table : Comparing runtime of forward and adjoint model: (Linear) Viscoelasticity example with average partial stress as functional

| | Runtime (s) | Ratio |
|---|---|---|
| Forward model | 119.93 | |
| Forward model + annotation | 120.24 | 1.002 |
| Forward model + annotation + adjoint model | 243.99 | 2.029 |

Contact NAG Blog NAGnews Case Studies Help Me Site Map Purchasing Search

nag®

Numerical Services | HPC and Supercomputing | Numerical Software | Collaboration | Industries | Support | About

# THE WILKINSON PRIZE FOR NUMERICAL SOFTWARE 2015

The Wilkinson Prize was established to honour the outstanding contributions of Dr James Hardy Wilkinson to the field of numerical software. It is awarded every four years at the International Congress on Industrial and Applied Mathematics by Argonne National Laboratory, the National Physical Laboratory, and the Numerical Algorithms Group. The recipients are authors of an outstanding piece of numerical software, judged on:

- the clarity of the software implementation and documentation;
- the importance of the application(s) addressed by the software;
- the portability, reliability, efficiency and usability of the software implementation;
- the clarity and depth of analysis of the algorithms and the software in the submission;
- the quality of the test software.

The 2015 prize is awarded to P.E. Farrell (University of Oxford), S.W. Funke (Simula Research Laboratory), D.A. Ham (Imperial College London), and M.E. Rognes (Simula Research laboratory) for the development of dolfin-adjoint, a package which automatically derives and solves adjoint and tangent linear equations from high-level mathematical specifications of finite element discretisations of partial differential equations. The prize will be presented at ICIAM 2015 and will consist of $3000 plus a commemorative plaque for each winner.