The FEniCS Project

Anders Logg

Simula Research Laboratory
University of Oslo

NOTUR 2011

2011–05–23
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/Ubuntu GNU/Linux
- Licensed under the GNU LGPL

http://www.fenicsproject.org/

Collaborators

University of Chicago, Argonne National Laboratory, Delft University of Technology, Royal Institute of Technology KTH, Simula Research Laboratory, Texas Tech University, University of Cambridge, ...
FEniCS is new technology combining generality, efficiency, simplicity and reliability.

- Generality through *abstraction*
- Efficiency through *code generation, adaptivity, parallelism*
- Simplicity through *high level scripting*
- Reliability through *adaptive error control*
FEniCS is automated FEM

- Automated generation of basis functions
- Automated evaluation of variational forms
- Automated finite element assembly
- Automated adaptive error control
What has FEniCS been used for?
Computational hemodynamics

- Low wall shear stress may trigger aneurysm growth
- Solve the incompressible Navier–Stokes equations on patient-specific geometries

\[ \dot{u} + \nabla u \cdot u - \nabla \cdot \sigma(u, p) = f \]
\[ \nabla \cdot u = 0 \]

Valen-Sendstad, Mardal, Logg, *Computational hemodynamics* (2011)
The Navier–Stokes solver is implemented in Python/FEniCS
FEniCS allows the solver to be implemented in a minimal amount of code
Hyperelasticity

- CBC.Solve is a collection of FEniCS-based solvers developed at the CBC
- CBC.Twist, CBC.Flow, CBC.Swing, CBC.Beat, ...

Fluid–structure interaction

- The FSI problem is a computationally very expensive coupled multiphysics problem

- The FSI problem has many important applications in engineering and biomedicine

Images courtesy of the Internet
Fluid–structure interaction (contd.)

- Fluid governed by the incompressible Navier–Stokes equations
- Structure modeled by the St. Venant–Kirchhoff model
- Adaptive refinement in space and time

Computational geodynamics

\[- \text{div} \sigma' - \nabla p = (Rb \phi - Ra T) e\]

\[\text{div} u = 0\]

\[\frac{\partial T}{\partial t} + u \cdot \nabla T = \Delta T\]

\[\frac{\partial \phi}{\partial t} + u \cdot \nabla \phi = k_c \Delta \phi\]

\[\sigma' = 2\eta \dot{\varepsilon}(u)\]

\[\dot{\varepsilon}(u) = \frac{1}{2} \left( \nabla u + \nabla u^T \right)\]

\[\eta = \eta_0 \exp \left( -b T / \Delta T + c (h - x_2) / h \right)\]
The mantle convection simulator is implemented in Python/FEniCS.

Images show a sequence of snapshots of the temperature distribution.

Vynnytska, Clark, Rognes, *Dynamic simulations of convection in the Earth’s mantle* (2011)
How to use FEniCS?
Installation

- Official packages for Debian and Ubuntu
- Drag and drop installation on Mac OS X
- Binary installer for Windows

- Automated building from source for a multitude of platforms
- VirtualBox / VMWare + Ubuntu!
Hello World in FEniCS: problem formulation

Poisson’s equation

\[-\Delta u = f \quad \text{in } \Omega\]
\[u = 0 \quad \text{on } \partial\Omega\]

Finite element formulation
Find \( u \in V \) such that

\[\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f \, v \, dx \quad \forall \, v \in \hat{V}\]
Hello World in FEniCS: problem formulation

Poisson’s equation

\[-\Delta u = f \quad \text{in } \Omega\]
\[u = 0 \quad \text{on } \partial \Omega\]

Variational formulation
Find \( u \in V \) such that

\[
\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx \quad \forall \, v \in \hat{V}
\]

\[
\left\{ \begin{array}{c}
\text{a}(u,v) \\
\text{L}(v)
\end{array} \right\}
\]
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())

problem = VariationalProblem(a, L, bc)
u = problem.solve()
plot(u)
Implementation of advanced solvers in FEniCS

Implementation of advanced solvers in FEniCS

# 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 - u))*dx +
  inner(epsilon(v), sigma(U, p0))*dx +
  inner(v, p0*n)*ds - mu*inner(grad(U)*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, lmbda] = parameters
    return lmbda/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
Key features

- Simple and intuitive object-oriented API, C++ or Python
- Automatic and efficient evaluation of variational forms
- Automatic and efficient assembly of linear systems
- Distributed (clusters) and shared memory (multicore) parallelism
- General families of finite elements, including arbitrary order continuous and discontinuous Lagrange elements, BDM, RT, Nédélec, ...
- Arbitrary mixed elements
- High-performance parallel linear algebra
- General meshes, adaptive mesh refinement
- mcG(q)/mdG(q) and cG(q)/dG(q) ODE solvers
- Support for a range of input/output formats
- Built-in plotting
Basic API

- Mesh, MeshEntity, Vertex, Edge, Face, Facet, Cell
- FiniteElement, FunctionSpace
- TrialFunction, TestFunction, Function
- grad(), curl(), div(), ...
- Matrix, Vector, KrylovSolver
- assemble(), solve(), plot()

- Python interface generated semi-automatically by SWIG
- C++ and Python interfaces almost identical
DOLFIN class diagram
FEniCS under the hood
Automated Scientific Computing

Input

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

Output

$$\|u - u_h\| \leq \epsilon$$
Automated Scientific Computing: a blueprint
Automatic code generation

**Input**
Equation (variational problem)

**Output**
Efficient application-specific code
Assembler interfaces

Variational Form  Finite element

Form compiler

UFC interface

Mesh

Finite element assembler

Linear algebra interface

PETSc  Epetra  uBLAS  MTL4  User

Linear algebra backend
Linear algebra in DOLFIN

- Generic linear algebra interface to
  - PETSc
  - Trilinos/Epetra
  - uBLAS
  - MTL4

- Eigenvalue problems solved by SLEPc for PETSc matrix types
- Matrix-free solvers ("virtual matrices")

Linear algebra backends

```python
>>> from dolfin import *
>>> parameters["linear_algebra_backend"] = "PETSc"
>>> A = Matrix()
>>> parameters["linear_algebra_backend"] = "Epetra"
>>> B = Matrix()
```
from dolfin import *

mesh = UnitSquare(32, 32)

V = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(V)
v = TestFunction(V)
f = Expression("sin(x[0])\cdot sin(x[1])")
a = (grad(u), grad(v)) + (u, v)
L = (f, v)

A = assemble(a, mesh)
b = assemble(L, mesh)

u = Function(V)
solve(A, u.vector(), b)
plot(u)

(Python, C++ – SWIG – Python, Python – JIT – C++ – GCC – SWIG – Python)
Code generation system

from dolfin import *

mesh = UnitSquare(32, 32)

V = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(V)
v = TestFunction(V)
f = Expression("sin(x[0])*sin(x[1])")
a = (grad(u), grad(v)) + (u, v)
L = (f, v)

A = assemble(a, mesh)
b = assemble(L, mesh)

u = Function(V)
solve(A, u.vector(), b)
plot(u)

(Python, C++ – SWIG – Python, Python – JIT – C++ – GCC – SWIG – Python)
Quality assurance by continuous testing

<table>
<thead>
<tr>
<th>fenics-buildbot</th>
<th>lucid-amd64</th>
<th>maverick-i386</th>
<th>mac-osx</th>
<th>linux64-exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>ferari</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>fiat</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>ufc</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>instant</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>ufl</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>ffc</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
<td>building</td>
</tr>
<tr>
<td>viper</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>dolfin</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>syfi</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
<td>Success</td>
</tr>
</tbody>
</table>

9 (9) / 9
Closing remarks
The state of FEniCS

- Parallelization (2009)
- Automated error control (2010)
- Debian/Ubuntu (2010)
- Documentation (2010)
- Latest release: 0.9.11 (May 2011)
- Release of 1.0 (2011)
- Book (2011)
- New web page (2011)
Summary

- Automated solution of differential equations
- Simple installation
- Simple scripting in Python
- Efficiency by automated code generation
- Free/open-source (LGPL)

Upcoming events

- Release of 1.0 (2011)
- Book (2011)
- New web page (2011)
- Mini courses / seminars (2011)

http://www.fenicsproject.org/

http://www.simula.no/research/acdc/