INTRODUZIONE A FENICS

Milano 03/02/2015

OUTLINE

- The FEniCS project
- Features
- UFL – Simple mathematical language
- Applicative Examples
  - Poisson Equation
  - Compressible Hyperelasticity
  - Incompressible Hyperelasticity

The FEniCS project

The FEniCS Project is a collaborative project set out in 2003 for the development of innovative concepts and tools for automated scientific computing, with a particular focus on **automated solution of differential equations by finite element method**.

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

FEniCS is a user-friendly tool for solving partial differential equations (PDEs).

The FEniCS Project is developed by researchers from a number of research institutes from around the world. The research institutes mainly involved are:

- Simula Research Laboratory (Oslo)
- University of Cambridge
- University of Chicago
- Baylor University
- KTH Royal Institute of Technology

The FEniCS project – Useful Information

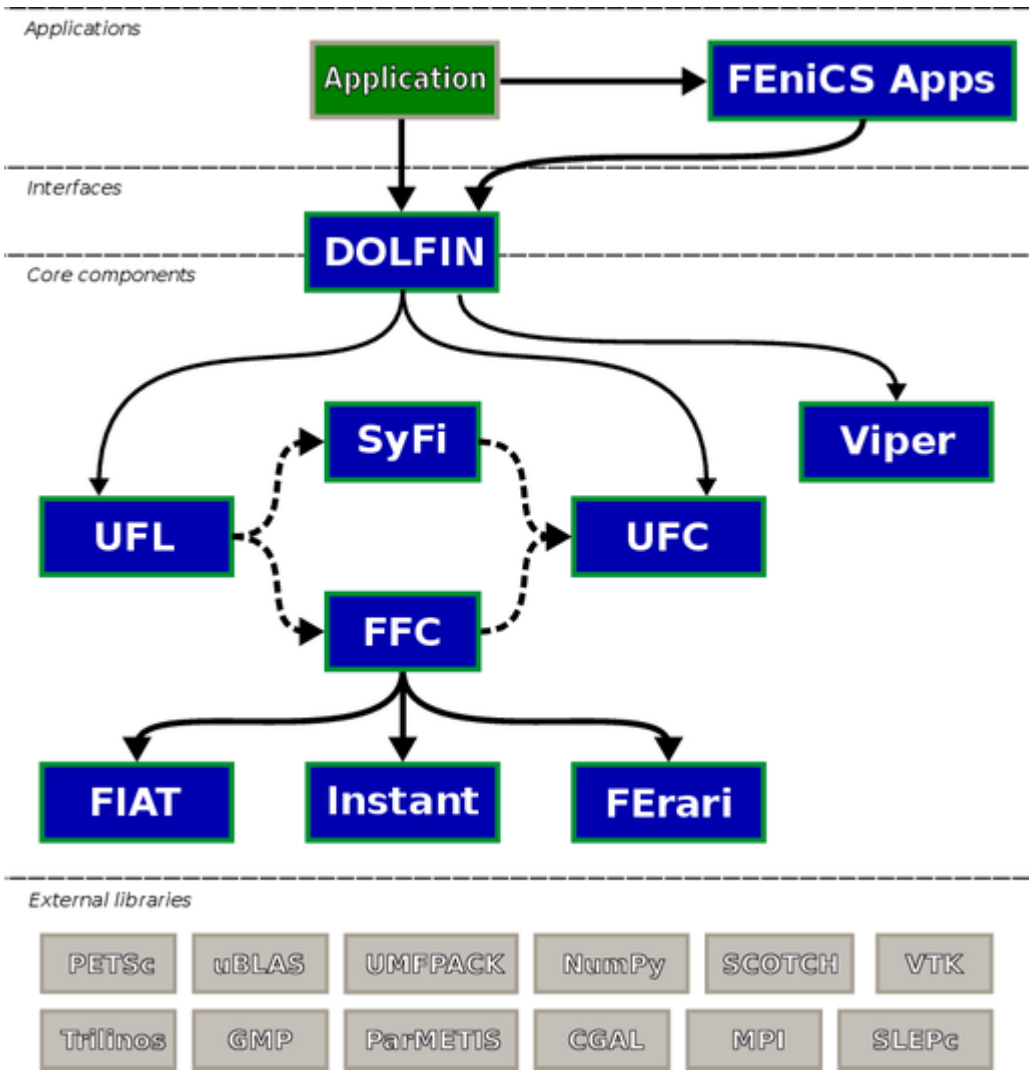
Website FEniCS project:
http://fenicsproject.org/

Installation:
FEniCS is included as part of Ubuntu GNU/Linux (starting with 10.04/Lucid). Installing procedure from the Ubuntu Software Center.
FEniCS can also be installed by running the following command in a terminal:
sudo apt-get install fenics

Documentation:
Python and C++ demos already installed in the system
FEniCS book "**Automated Solution of Differential Equations by the Finite Element Method - The FEniCS Book**"
pydoc dolfin.X to look up documentation of a Python class X from the DOLFIN library

Question and Answers:
(old) https://launchpad.net/fenics
(new) **http://fenicsproject.org/qa/**

STRUCTURE



FEniCS is organized as a collection of interoperable components that together form the FEniCS Project.

DOLFIN: the problem-solving environment; it is a C++ library, with a Python interface.

FFC: form compiler.

FIAT: the finite element tabulator.

Instant: the just-in-time compiler.

UFC: the code generation interface.

UFL: the form language; a component implementing the unified form language for specifying finite element forms.

Viper: a component for quick visualization of finite element meshes and solutions.

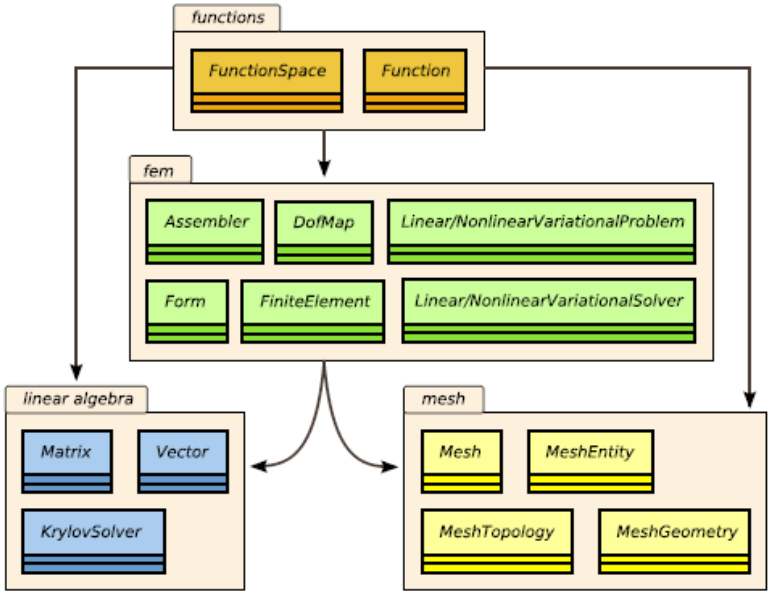
Python Interface

CLASS: a programming construction for creating objects containing a set of variables and functions. Most types of FEniCS objects are defined through the class concept.

INSTANCE: an object of a particular type, where the type is implemented as a class.
For instance, mesh = UnitInterval(10) creates an instance of class UnitInterval, which is reached by the name mesh. (Class UnitInterval is actually just an interface to a corresponding C++ class in the DOLFIN C++ library.)

CLASS ATTRIBUTE: a variable in a class, reached by dot notation: instance_name.attribute_name.

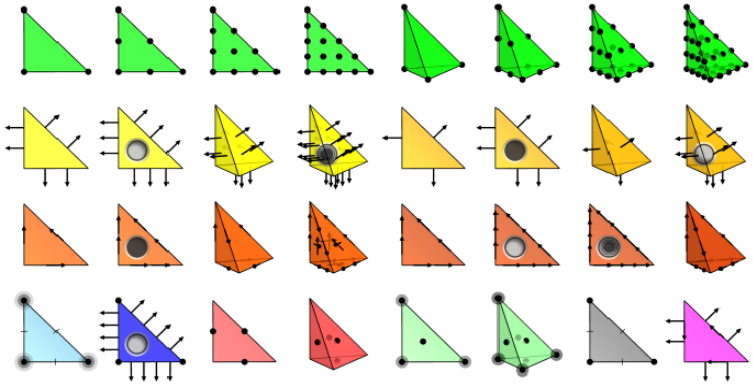
Most important components and classes in FEniCS, along with their dependencies indicated by arrows.



FEATURES

ONLY TRIANGLE AND TETRAHEDRON
FULLY SUPPORTED in BLACK
PARTLY SUPPORTED in GREY

Name	Symbol
Argyris	ARG
Arnold–Winther	AW
Brezzi–Douglas–Marini	BDM
Crouzeix–Raviart	CR
Discontinuous Lagrange	DG
Hermite	HER
Lagrange	CG
Mardal–Tai–Winther	MTW
Morley	MOR
Nédélec 1st kind H(curl)	N1curl
Nédélec 2nd kind H(curl)	N2curl
Raviart–Thomas	RT



FEATURES

THIRD-PARTS LIBRARY for
LINEAR ALGEBRA BACKENDS

- vectors;
- dense/sparse matrices;
- direct/iterative solvers;
- eigenvalue solver;

Linear algebra backends
PETSc
Trilinos/Epetra
uBLAS
MTL4

Depending on the specific algebra backend
list_lu_solver_methods()
list_krylov_solver_methods()
list_krylov_solver_preconditioners()

```
>>> list_lu_solver_methods()
LU method   |  Description
-----------------------------------------------------------------------
default     |  default LU solver
umfpack     |  UMFPACK (Unsymmetric MultiFrontal sparse LU factorization)
mumps       |  MUMPS (MUltifrontal Massively Parallel Sparse direct Solver)
spooles     |  SPOOLES (SParse Object Oriented Linear Equations Solver)
petsc       |  PETSc builtin LU solver
>>> list_krylov_solver_methods()
Krylov method  |  Description
-----------------------------------------------------------
default        |  default Krylov method
cg             |  Conjugate gradient method
gmres          |  Generalized minimal residual method
minres         |  Minimal residual method
tfqmr          |  Transpose-free quasi-minimal residual method
richardson     |  Richardson method
bicgstab       |  Biconjugate gradient stabilized method
```

FEATURES

Linear solver		Preconditioners	
lu	Sparse lu factorization	ilu	Incomplete LU factorization
cholesky	Sparse Cholesky factorization	icc	Incomplete Cholesky factorization
cg	Conjugate gradient method	jacobi	Jacobi iteration
gmres	Generalized minimal residual method	bjacobi	Block Jacobi iteration
bicgstab	Biconjugate gradient stabilized method	sor	Successive over-relaxation
minres	Minimal residual method	amg	Algebraic multigrid (BoomerAMG or ML)
tfgmr	Transpose-free quasi-minimal residual method	additive_schwartz	Additive Schwarz
richardson	Richardson method	hypre_amg	Hypre algebraic multigrid (BoomerAMG)
		hypre_euclid	Hypre parallel incomplete LU factorization
		hypre_parasails	Hypre parallel sparse approximate inverse
		ml_amg	ML algebraic multigrid

UFL – Unified Form Language

The Unified Form Language – UFL (Alnæs and Logg, 2009) – is a domain specific language for the declaration of finite element discretizations of variational forms and functionals. More precisely, the language defines a flexible user interface for defining finite element spaces and expressions for weak forms in a notation close to mathematical notation.

- A richer form language, especially for expressing nonlinear PDEs.
- Automatic differentiation of expressions and forms.
- Improving the performance of the form compiler technology to handle more complicated equations efficiently.

UFL – Unified Form Language

$F = I + \text{grad}u$
$C = F^T F$
$I_C = tr(C)$
$II_C = \frac{1}{2}[I_C^2 - tr(CC)]$
$W = c_1(I_C - 3) + c_2(II_C - 3)$
$S = 2\frac{\partial W}{\partial C}$

Find u such that
$div(FS) = 0 \; in \; \Omega$
+ boundary conditions

The non linear variational problem reads
$L(u, \phi) = \int_\Omega FS : grad \; \phi dx \; \forall \; \phi \in V$

```
# Form arguments
phi0 = TestFunction(element)
phi1 = TrialFunction(element)
u = Coefficient(element)
c1 = Constant(cell)
c2 = Constant(cell)

# Deformation gradient Fij = dXi/dxj
I = Identity(cell.d)
F = I + grad(u)

# Right Cauchy-Green strain tensor C with invariants
C = variable(F.T*F)
I_C = tr(C)
II_C = (I_C**2 - tr(C*C))/2

# Mooney-Rivlin constitutive law
W = c1*(I_C-3) + c2*(II_C-3)

# Second Piola-Kirchoff stress tensor
S = 2*diff(W, C)

# Weak forms
L = inner(F*S, grad(phi0))*dx
```

dot(a,b): $\text{v} \cdot u = v_i u_i$, $\text{A} \cdot u = A_{ij} u_j e_i$, $\text{A} \cdot B = A_{ik} B_{kj} e_i e_j$, $\text{C} \cdot A = C_{ijk} A_{kl} e_i e_j e_l$

inner(a,b): $\text{v}: u = v_i u_i$, $\text{A}: B = A_{ij} B_{ij}$, $\text{C}: D = C_{ijkl} D_{ijkl}$

outer(a,b): $\text{v} \otimes u = v_i u_j e_i e_j$, $\text{A} \otimes u = A_{ij} u_k e_i e_j e_k$, $\text{A} \otimes B = A_{ij} B_{kl} e_i e_j e_k e_l$

cross(u,v), transpose(A), tr(A), det(A), inv(A), cofac(A), dev(A), skew(A), sym(A), …

UFL – Unified Form Language

Consider Poisson's equation with two different boundary conditions on $\partial\Omega_0$ and $\partial\Omega_1$, the variational problem reads:

$$a(w;u,v) = L(f,g,h;v)$$

$$a(w;u,v) = \int_\Omega w\ gradu\ \cdot gradv\ dx$$

$$L(f,g,h;v) = \int_\Omega fvdx + \int_{\partial\Omega_0} g^2 vds + \int_{\partial\Omega_1} hvds$$

These forms can be expressed in UFL as

$$a\ =\ w * inner(grad(u), grad(v)) * dx$$
$$L\ =\ f * v * dx\ +\ g ** 2 * v * ds(0)\ +\ h * v * ds(1)$$

Integrals:

$\int_\Omega (.)dx$ →*dx defines a cell integral

$\int_{\partial\Omega}(.)ds$ →*ds defines an exterior facet integral

Subdomains:

```
class Boundary(SubDomain):
  def inside(self,x,on_boundary):
    return x[0]>0.5 + DOLFIN+EPS

boundary_parts = MeshFunction("size_t", mesh, 1)
boundary_parts.set_all(0)
boundary = Boundary()
boundary. mark(boundary_parts, 1)
```

UFL – Unified Form Language

Spatial derivative: $^{\partial f}/_{\partial x_i}$. Derivative of f in the spatial direction x_i

$$df = Dx(f, i)$$
$$df = f.dx(i)$$

The operator **diff** can be applied to expressions to differentiate w.r.t designated variables.

$$g = \sin(cell.x[0])$$
$$v = variable(g)$$
$$f = \exp(v ** 2)$$
$$h = diff(f, v)$$

The form operator **derivative** declares the derivative of a form w.r.t. coefficients of a discrete function. In case of non-trivial equations is useful.

- to linearize your nonlinear residual equation (linear form) automatically for use with the Newton-Raphson method.

Pi is energy; u: function; du: trial function; v: test function

$$F = derivative(Pi, u, v)$$

Compute Jacobian of F for Newton-Raphson method

$$J = derivative(F, u, du)$$

- if applied multiple times, to derive a linear system from a convex functional, in order to find the function that minimizes the functional.

EXAMPLE – Poisson Equation

For a domain $\Omega \subset R^n$ with boundary $\partial\Omega = \Gamma_D \cup \Gamma_N$, the Poisson equation is

$$-\nabla^2 u = f \ in\ \Omega$$
$$u = 0\ in\ \Gamma_D$$
$$\nabla u \cdot n = g\ in\ \Gamma_N$$

where f and g are input and n is the outward directed boundary normal.
The variational form reads

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

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v\, dx\, , L(v) = \int_\Omega fv\, dx + \int_{\Gamma_N} gvds$$

V is a function space that satisfy the Dirichlet boundary conditions

Data in the example
$\Omega = [0,1]x[0,1]$ (unit square)
$\Gamma_D = \{(0,y) \cup (1,y) \subset \partial\Omega\}$ (Dirichlet boundary)
$\Gamma_N = \{(x,0) \cup (x,1) \subset \partial\Omega\}$ (Neumann boundary)
$g = \sin 5x$
$f = 10e^{\left[-\frac{2(x-0.5)+2(y-0.5)}{0.02}\right]}$

EXAMPLE – Poisson Equation

```
from dolfin import *
# Create mesh and define function space
mesh = UnitSquareMesh(32, 32)
V = FunctionSpace(mesh, "Lagrange", 1)
# Define Dirichlet boundary (x = 0 or x = 1)
def boundary(x):
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS
# Define boundary condition
u0 = Constant(0.0)
bc = DirichletBC(V, u0, boundary)
# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Expression("10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)")
g = Expression("sin(5*x[0])")
a = inner(grad(u), grad(v))*dx
L = f*v*dx + g*v*ds
# Compute solution
u = Function(V)
solve(a == L, u, bc)
# Save solution in VTK format
file = File("poisson.pvd") file << u
# Plot solution
plot(u, interactive=True)
```

EXAMPLE – Poisson Equation

from dolfin import *

To use DOLFIN from Python, users need to import functionality from the DOLFIN Python module.

call to required external library. Es. import numpy

parameters["linear_algebra_backend"] = "PETSc"

 # Create mesh and define function space

mesh = **UnitSquareMesh**(32, 32)

implemented class for simple mesh: Square, Circle, Cube, Cone, Cylinder, Sphere
dolfin-convert filename.* filename.msh
gmsh, Abaqus…

V = **FunctionSpace**(mesh, "Lagrange", 1)

FunctionSpace: Class to create a Function Space. It requires the mesh, the element type, the degree

EXAMPLE – Poisson Equation

```
# Define Dirichlet boundary (x = 0 or x = 1)
def boundary(x):
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS
```

Or label from gmsh

```
# Define boundary condition
u0 = Constant(0.0)
bc = DirichletBC(V, u0, boundary)
```

A Python function, returning a boolean, can be used to define the subdomain for the Dirichlet boundary condition. The function should return True for those points inside the subdomain and False for the points outside.

```
# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
```

EXAMPLE – Poisson Equation

```
f = Expression("10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)")
g = Expression("sin(5*x[0])")
```

strings defining f and g use C++ syntax since, for efficiency, DOLFIN will generate and compile C++ code for these expressions at run-time.
Add parameter

```
a = inner(grad(u), grad(v))*dx
L = f*v*dx + g*v*ds
```

Variational form

```
# Compute solution
u = Function(V)
solve(a == L, u, bc)
```

u is automatically initialized at null
Solving the variational problem; Matrices automatically assembled in the form Ax=b

```
# Save solution in VTK format
file = File("poisson.pvd") file << u
# Plot solution
plot(u, interactive=True)
```

Saving and Visualization.
vtk format available

SOLVING A x=b

```
solve(A, x, b)
```

This is the simplest approach to solving the linear system Ax = b:

- default methods are considered
- straightforward, but little control
- it discriminates between linear and nonlinear problems

```
solve(A, x, b,''lu'')
solve(A, x, b, ''gmres'', ''ilu'')
```

Here you provide methods and preconditioners

```
solver = LUSolver(A)
solver.solve(x,b)

solver = KrylovSolver(A)
solver.solve(x,b)
```

More specific choices

SOLVING A x=b

```
>>> solver = KrylovSolver()
>>> info(solver.parameters,1)
<Parameter set "krylov_solver" containing 9 parameter(s) and 2 nested parameter set(s)>

  krylov_solver            |    type  value          range  access  change
  ---------------------------------------------------------------------
  absolute_tolerance       |  double  1e-15             []       1       0
  divergence_limit         |  double  10000             []       1       0
  error_on_nonconvergence  |    bool   true  {true, false}       1       0
  maximum_iterations       |     int  10000             []       1       0
  monitor_convergence      |    bool  false  {true, false}       1       0
  nonzero_initial_guess    |    bool  false  {true, false}       1       0
  relative_tolerance       |  double  1e-06             []       1       0
  report                   |    bool   true  {true, false}       1       0
  use_petsc_cusp_hack      |    bool  false  {true, false}       1       0

  <Parameter set "gmres" containing 1 parameter(s) and 0 nested parameter set(s)>

    gmres    |  type  value  range  access  change
    ---------------------------------------------
    restart  |   int     30     []       1       0

  <Parameter set "preconditioner" containing 4 parameter(s) and 2 nested parameter set(s)>

    preconditioner        |    type  value          range  access  change
    ---------------------------------------------------------------------
    report                |    bool  false  {true, false}       1       0
    reuse                 |    bool  false  {true, false}       1       0
    same_nonzero_pattern  |    bool  false  {true, false}       1       0
    shift_nonzero         |  double      0             []       1       0

    <Parameter set "ilu" containing 1 parameter(s) and 0 nested parameter set(s)>
```

EXAMPLE – Compressible Hyperelasticity

Boundary value problems for hyperelastic media can be expressed as minimisation problems.
For a domain $\Omega \subset R^n$, the task is to find the displacement field $u: \Omega \to R^n$ that minimises the total potential energy Π:

$$\min_{v \in V} \Pi$$

where V is a suitable function space that satisfies boundary conditions on u.
The total potential energy is given by

$$\Pi = \int_\Omega \Phi(u)\, dx - \int_\Omega B \cdot u\, dx - \int_\Gamma T \cdot u ds$$

where $\Phi(u)$ is the elastic stored energy density, B is a body force (per unit reference volume) and T is a traction force (per unit reference area).
At minimum point, the directional derivative of Π along the test function v with respect to change in function u is equal to zero

$$L(u; v) = D_v \Pi = \left.\frac{d\Pi(u + \epsilon v)}{d\epsilon}\right|_{\epsilon=0} = 0 \qquad \forall v \in V$$

If $\Phi(u)$ is nonlinear, the Jacobian is required for the Newton's method. It is

$$Jac(u; du, v) = D_{du} L = \left.\frac{d\mathrm{L}(u + \epsilon du; v)}{d\epsilon}\right|_{\epsilon=0}$$

EXAMPLE – Compressible Hyperelasticity

```
from dolfin import *

# Create mesh and define function space
mesh = UnitCubeMesh(24, 16, 16)
V = VectorFunctionSpace(mesh, "Lagrange", 1)
```

A Vector Function Space is required

```
bcl = DirichletBC(V, c, left)
bcr = DirichletBC(V, r, right)
bcs = [bcl, bcr]
```

More than one boundary condition are collected together in a list

```
B = Constant((0.0, -0.5, 0.0))  # Body force per unit volume
T = Constant((0.1, 0.0, 0.0)) # Traction force on the boundary
```

In place of Constant, it is also possible to use as_vector, e.g. B = as_vector([0.0, -0.5, 0.0]). The advantage of Constant is that its values can be changed without requiring re-generation and re-compilation of C++ code. On the other hand, using as_vector can eliminate some function calls during assembly.

EXAMPLE – Compressible Hyperelasticity

```
# Kinematics
I = Identity(V.cell().d) # Identity tensor
F = I + grad(u) # Deformation gradient
C = F.T*F # Right Cauchy-Green tensor

# Invariants of deformation tensors
Ic = tr(C) J = det(F)

# Elasticity parameters
E, nu = 10.0, 0.3
mu, lmbda = Constant(E/(2*(1 + nu))), Constant(E*nu/((1 + nu)*(1 - 2*nu)))

# Stored strain energy density (compressible neo-Hookean model)
psi = (mu/2)*(Ic - 3) - mu*ln(J) + (lmbda/2)*(ln(J))**2 # Strain Energy Function
Pi = psi*dx - dot(B, u)*dx - dot(T, u)*ds # Total potential energy
```

Kinematics and Energy are derived with a formalism very similar to the analytical one

EXAMPLE – Compressible Hyperelasticity

```
# Compute first variation of Pi (directional derivative about u in the direction of v)
F = derivative(Pi, u, v)
# Compute Jacobian of F
J = derivative(F, u, du)
```

Function *derivative* performs the directional derivatives automatically.
NB: it is not possible to modify the output of *derivative*

```
# Solve variational problem
solve(F == 0, u, bcs, J=J, form_compiler_parameters=ffc_options)
# Save solution in VTK format
file = File("displacement.pvd"); file << u;
# Plot and hold solution
plot(u, mode = "displacement", interactive = True)
```

Solve with customized options

EXAMPLE – Incompressible Hyperelasticity

```
# Periodic Boundary conditions
class PeriodicBoundary(SubDomain):
 #Left boundary is "targeted domain" G
 def inside(self,x,on_boundary) :
    return bool(x[0] < 0 + DOLFIN_EPS and x[0] > -1 and on_boundary)
 #map right boundary (H) to left boundary (G)
 def map(self,x,y) :
    if (x[0] > 20 - DOLFIN_EPS and x[0] < 20 + DOLFIN_EPS and on_boundary) :
       y[0] = x[0] - 20
       y[1] = x[1]
pbc = PeriodicBoundary()

# Definition of function spaces
P2 = VectorFunctionSpace(mesh, "CG", user_par.fe_order_u, constrained_domain =
PeriodicBoundary()) # Space for displacement
P1 = FunctionSpace(mesh, "CG", user_par.fe_order_p) # Space for pressure
V = MixedFunctionSpace([P1,P2])
```

Kinematics and Energy are derived with a formalism very similar to the analytical one

EXAMPLE– Incompressible Hyperelasticity

```
# Create functions to define the energy and store the results
up = Function(V)
(p,u)=split(up)
```

Function that extracts one function for each subspace

```
# Create test and trial functions for the variational formulation
dup = TrialFunction(V)
vq = TestFunction(V)
(q,v) = TestFunctions(V)
```

```
bc1 = DirichletBC(V.sub(0).sub(0), zero_scalar, X())
bc1 = DirichletBC(V.sub(0), zero_vector, X())
```

Boundary conditions can be applied component by component of the subspace

EXAMPLE – Incompressible Hyperelasticity

```
# Energy (Incompressible neo Hookean)
psi = mu/2 * (Ic - 3) + p * (J - 1)
Pi = psi*dx

F=derivative(Pi, up,vq)
dF=derivative(F,up,dup)
```

Lie Derivative is automatically applied to the functions belonging to different subspaces

```
# Setup the variational problem
varproblem = NonlinearVariationalProblem(F, up, bc_u,
J=dF,form_compiler_parameters=ffc_options)
solver = NonlinearVariationalSolver(varproblem)
```

In this case, the NonlinearVariationalSolver is called
Info(NonlinearVariationalSolver.default_parameters(),1)

EXAMPLE – Incompressible Hyperelasticity

```
# surface energy
gamma = Constant(36.5)
N = FacetNormal(mesh)
NansonOp = transpose(cofac(F))
deformed_N = dot(NansonOp,N)
current_element_of_area = (1+u[1]/X[1])*sqrt(dot(deformed_N,deformed_N))
surface_energy_density = gamma*current_element_of_area
surface_energy = 2*math.pi*X[1]*surface_energy_density*ds(1)
```

FEniCS, by default, works in a Material Reference System and with a Cartesian framework

```
# Total potential energy
Pi= Psi + surface_energy
```

(as before)

EXAMPLE – Incompressible Hyperelasticity

```
# Implemented Newton Raphson
while ttt <= T:
  fnorm = 1
  eps = 1
  mu.assign(value)
  nIter = 0
  cicle = 0
  while ok == 0 :
      while fnorm > 1e-7 or eps > 1e-6 and nIter < nIter_max :
      nIter += 1
      bcs_du = homogenize(bc_u)
      A, b = assemble_system(dF, -F, bcs_du)
      solve (A, dw_inc.vector(), b)
      eps_ok = np.linalg.norm(dw_inc.vector().array(), ord = 2)
      up.vector()[:] = up.vector() + omega*dw_inc.vector()
…
```

Kinematics and Energy are derived with a formalism very similar to the analytical one

- Adaptive meshing
- Dofmaps
- Numpy, Scipy
- Fem- Fenics