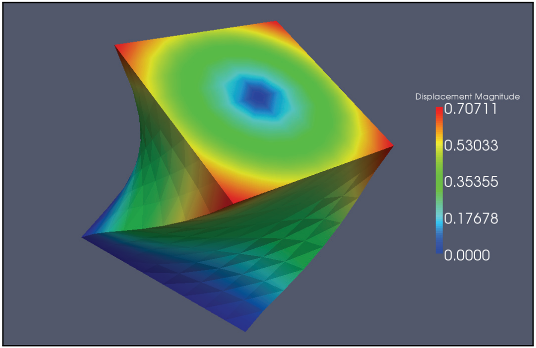
An automated computational framework for hyperelasticity

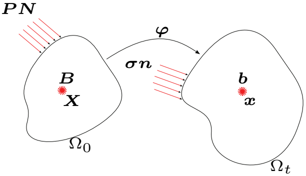
Harish Narayanan

Center for Biomedical Computing
Simula Research Laboratory

May 20th, 2010



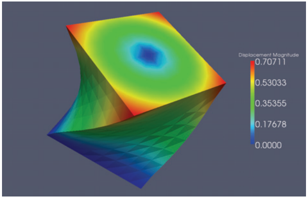
This talk will examine the motivation, design and use of our general framework for hyperelasticity



A review of relevant topics from continuum mechanics

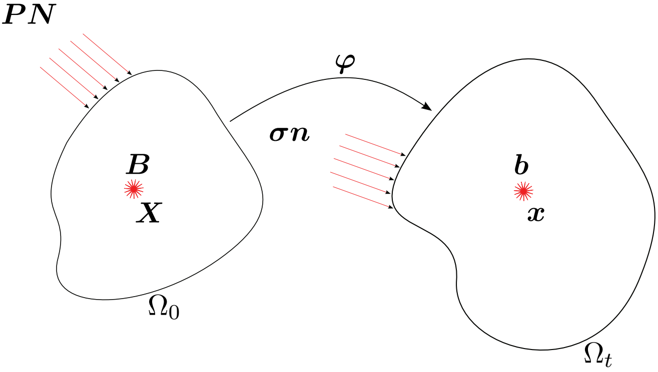
```
def SecondPiolaKirchhoffStress(self, u):
    self._construct_local_kinematics(u)
    psi = self.strain_energy(MaterialModel._parameters_as_fu

    if self.kinematic_measure == "InfinitesimalStrain":
        epsilon = self.epsilon
        S = diff(psi, epsilon)
    elif self.kinematic_measure == "RightCauchyGreen":
        C = self.C
        S = 2*diff(psi, C)
    elif self.kinematic_measure ==  "GreenLagrangeStrain":
        E = self.E
        S = diff(psi, E)
```

A brief look at numerical and computational aspects



Examples demonstrating the use of the framework

Recall, from elementary continuum mechanics . . .



The body idealised as a continuous medium

Reference and current configurations, body forces and tractions

. . . that the motion of solid bodies can be described using different strain measures

- Infinitesimal strain: $\boldsymbol{\epsilon} = \frac{1}{2}\left(\mathrm{Grad}(\boldsymbol{u}) + \mathrm{Grad}(\boldsymbol{u})^{\mathrm{T}}\right)$
- Deformation gradient: $\boldsymbol{F} = \mathbf{1} + \mathrm{Grad}(\boldsymbol{u})$
- Right Cauchy-Green: $\boldsymbol{C} = \boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}$
- Green-Lagrange: $\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{C} - \mathbf{1}\right)$
- Left Cauchy-Green: $\boldsymbol{b} = \boldsymbol{F}\boldsymbol{F}^{\mathrm{T}}$
- Euler-Almansi: $\boldsymbol{e} = \frac{1}{2}\left(\mathbf{1} - \boldsymbol{b}^{-1}\right)$
- Volumetric and isochoric splits: e.g.
  $J = \mathrm{Det}(\boldsymbol{F}), \quad \bar{\boldsymbol{C}} = J^{-\frac{2}{3}}\boldsymbol{C}$
- Invariants of the tensors: I_1, I_2, I_3
- Principal stretches and directions: λ_1, λ_2, λ_3; $\quad \hat{\boldsymbol{N}}_1, \hat{\boldsymbol{N}}_2, \hat{\boldsymbol{N}}_3$

And the UFL syntax for defining these measures is almost identical to the mathematical notation

```
# Infinitesimal strain tensor
def InfinitesimalStrain(u):
    return variable(0.5*(Grad(u) + Grad(u).T))

# Second order identity tensor
def SecondOrderIdentity(u):
    return variable(Identity(u.cell().d))

# Deformation gradient
def DeformationGradient(u):
    I = SecondOrderIdentity(u)
    return variable(I + Grad(u))

# Determinant of the deformation gradient
def Jacobian(u):
    F = DeformationGradient(u)
    return variable(det(F))

# Right Cauchy-Green tensor
def RightCauchyGreen(u):
    F = DeformationGradient(u)
    return variable(F.T*F)

# Green-Lagrange strain tensor
def GreenLagrangeStrain(u):
    I = SecondOrderIdentity(u)
    C = RightCauchyGreen(u)
    return variable(0.5*(C - I))
```

```
# Left Cauchy-Green tensor
def LeftCauchyGreen(u):
    F = DeformationGradient(u)
    return variable(F*F.T)

# Euler-Almansi strain tensor
def EulerAlmansiStrain(u):
    I = SecondOrderIdentity(u)
    b = LeftCauchyGreen(u)
    return variable(0.5*(I - inv(b)))

# Invariants of an arbitrary tensor, A
def Invariants(A):
    I1 = tr(A)
    I2 = 0.5*(tr(A)**2 - tr(A*A))
    I3 = det(A)
    return [I1, I2, I3]

# Invariants of the (right/left) Cauchy-Green tensor
def CauchyGreenInvariants(u):
    C = RightCauchyGreen(u)
    [I1, I2, I3] = Invariants(C)
    return [variable(I1), variable(I2), variable(I3)]

# Isochoric part of the deformation gradient
def IsochoricDeformationGradient(u):
    F = DeformationGradient(u)
    J = Jacobian(u)
    return variable(J**(-1.0/3.0)*F)
```

Stress responses of hyperelastic materials are specified using constitutive relationships involving strain energy functions

- Strain energy functions: $\Psi(\boldsymbol{F}), \Psi(\boldsymbol{C}), \Psi(\boldsymbol{E}), \ldots$
- First Piola Kirchhoff: $\boldsymbol{P} = \frac{\partial \Psi(\boldsymbol{F})}{\partial \boldsymbol{F}} = 2\boldsymbol{F}\frac{\partial \Psi(\boldsymbol{C})}{\partial \boldsymbol{C}} = \ldots$
- Second Piola Kirchhoff: $\boldsymbol{S} = 2\frac{\partial \Psi(\boldsymbol{C})}{\partial \boldsymbol{C}} = \frac{\partial \Psi(\boldsymbol{E})}{\partial \boldsymbol{E}} = 2\left[\left(\frac{\partial \Psi}{\partial I_1} + I_1\frac{\partial \Psi}{\partial I_2}\right)\boldsymbol{1} - \frac{\partial \Psi}{\partial I_2}\boldsymbol{C} + I_3\frac{\partial \Psi}{\partial I_3}\boldsymbol{C}^{-1}\right] = \sum_{a=1}^{3}\frac{1}{\lambda_a}\frac{\partial \Psi}{\partial \lambda_a}\hat{\boldsymbol{N}}_a \otimes \hat{\boldsymbol{N}}_a = \ldots$
- e.g. $\Psi_{\mathrm{St.Venant-Kirchhoff}} = \frac{\lambda}{2}\mathrm{tr}(\boldsymbol{E})^2 + \mu\mathrm{tr}(\boldsymbol{E}^2)$
  $\Psi_{\mathrm{Ogden}} = \sum_{p=1}^{N}\frac{\mu_p}{\alpha_p}\left(\lambda_1^{\alpha_p} + \lambda_2^{\alpha_p} + \lambda_3^{\alpha_p} - 3\right)$
  $\Psi_{\mathrm{Mooney-Rivlin}} = c_1(I_1 - 3) + c_2(I_2 - 3)$
  $\Psi_{\mathrm{Arruda-Boyce}} = \mu\left[\frac{1}{2}(I_1 - 3) + \frac{1}{20n}(I_1^2 - 9) + \frac{11}{1050n^2}(I_1^3 - 27) + \ldots\right]$
  $\Psi_{\mathrm{Yeoh}}, \Psi_{\mathrm{Gent-Thomas}}, \Psi_{\mathrm{neo-Hookean}}, \Psi_{\mathrm{Ishihara}}, \Psi_{\mathrm{Blatz-Ko}}, \ldots$

Again, the UFL syntax for defining different materials is almost identical to the mathematical notation

```
class StVenantKirchhoff(MaterialModel):
    """Defines the strain energy function for a St. Venant-Kirchhoff
    material"""

    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 MooneyRivlin(MaterialModel):
    """Defines the strain energy function for a (two term)
    Mooney-Rivlin material"""

    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*(I2 - 3)
```

Again, the UFL syntax for defining different materials is almost identical to the mathematical notation

```
def SecondPiolaKirchhoffStress(self, u):
    self._construct_local_kinematics(u)
    psi = self.strain_energy(MaterialModel._parameters_as_functions(self, u))

    if self.kinematic_measure == "InfinitesimalStrain":
        epsilon = self.epsilon
        S = diff(psi, epsilon)
    elif self.kinematic_measure == "RightCauchyGreen":
        C = self.C
        S = 2*diff(psi, C)
    elif self.kinematic_measure ==  "GreenLagrangeStrain":
        E = self.E
        S = diff(psi, E)
    elif self.kinematic_measure == "CauchyGreenInvariants":
        I = self.I; C = self.C
        I1 = self.I1; I2 = self.I2; I3 = self.I3
        gamma1 = diff(psi, I1) + I1*diff(psi, I2)
        gamma2 = -diff(psi, I2)
        gamma3 = I3*diff(psi, I3)
        S = 2*(gamma1*I + gamma2*C + gamma3*inv(C))
    elif self.kinematic_measure == "IsochoricCauchyGreenInvariants":
        I = self.I; Cbar = self.Cbar
        I1bar = self.I1bar; I2bar = self.I2bar; J = self.J
        gamma1bar = diff(psibar, I1bar) + I1bar*diff(psibar, I2bar)
        gamma2bar = -diff(psibar, I2bar)
        Sbar = 2*(gamma1bar*I + gamma2bar*C_bar)
        ...
```

The equations that need to be solved are the balance laws in the reference configuration

- Balance of mass: $\frac{\partial \rho_0}{\partial t} = 0$
- Balance of linear momentum: $\rho_0 \frac{\partial^2 \boldsymbol{u}}{\partial t^2} = \mathrm{Div}(\boldsymbol{P}) + \boldsymbol{B}$
- Balance of angular momentum: $\boldsymbol{P}\boldsymbol{F}^{\mathrm{T}} = \boldsymbol{F}\boldsymbol{P}^{\mathrm{T}}$

The weak form thus reads: Find $\boldsymbol{u} \in \mathrm{V}$, such that $\forall\ \boldsymbol{v} \in \hat{\mathrm{V}}$:

$$\int_{\Omega_0} \rho_0 \frac{\partial^2 \boldsymbol{u}}{\partial t^2} \cdot \boldsymbol{v}\, dx + \int_{\Omega_0} \boldsymbol{P} : \mathrm{Grad}(\boldsymbol{v})\, dx \ = \int_{\Omega_0} \boldsymbol{B} \cdot \boldsymbol{v}\, dx + \int_{\Gamma_{\mathrm{N}}} \boldsymbol{P}\boldsymbol{N} \cdot \boldsymbol{v}\, dx$$

with suitable initial conditions, and Dirichlet and Neumann boundary conditions.

UFL's automatic differentiation capabilities allows for easy specification of such a problem

```
# Get the problem mesh
mesh = problem.mesh()

# Define the function space
vector = VectorFunctionSpace(mesh, "CG", 1)

# Test and trial functions
v = TestFunction(vector)
u = Function(vector)
du = TrialFunction(vector)

# Get forces and boundary conditions
B = problem.body_force()
PN = problem.surface_traction()
bcu = problem.boundary_conditions()

# First Piola-Kirchhoff stress tensor based on the material
# model
P = problem.first_pk_stress(u)

# The variational form corresponding to static hyperelasticity
L = inner(P, Grad(v))*dx - inner(B, v)*dx - inner(PN, v)*ds
a = derivative(L, u, du)

# Setup and solve problem
equation = VariationalProblem(a, L, bcu, nonlinear = True)
equation.solve(u)
```

UFL's automatic differentiation capabilities allows for easy specification of such a problem

- Spatial derivatives:
```
df_i = Dx(f, i)
```
- With respect to user-defined variables:
```
g = variable(cos(cell.x[0]))
f = exp(g**2)
h = diff(f, g)
```
- Forms with respect to coefficients of a discrete function:
```
a = derivative(L, w, u)
```
- Computing expressions and automatic differentiation:

  for $i = 1, \ldots, m$:

  $$y_i = t_i = \text{terminal expression}$$
  $$\frac{dy_i}{dv} = \frac{dt_i}{dv} = \text{terminal differentiation rule}$$

  for $i = m+1, \ldots, n$:

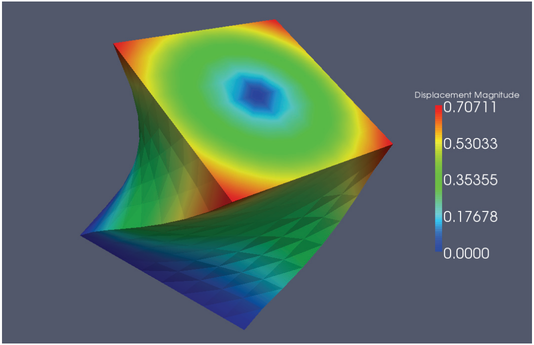
  $$y_i = f_i(<y_j>_{j \in \mathcal{J}_i})$$
  $$\frac{dy_i}{dv} = \sum_{k \in \mathcal{J}_i} \frac{\partial f_i}{\partial y_k} \frac{dy_k}{dv}$$

  $$z = y_n$$
  $$\frac{dz}{dv} = \frac{dy_n}{dv}$$

A simple static calculation involving a twisted block

```
class Twist(StaticHyperelasticity):

    def mesh(self):
        n = 8
        return UnitCube(n, n, n)

    def dirichlet_conditions(self):
        clamp = Expression(("0.0", "0.0", "0.0"))
        twist = Expression(("0.0",
                            "y0 + (x[1] - y0) * cos(theta) - (x[2] - z0) * sin(theta) - x[1]",
                            "z0 + (x[1] - y0) * sin(theta) + (x[2] - z0) * cos(theta) - x[2]"))
        twist.y0 = 0.5
        twist.z0 = 0.5
        twist.theta = pi/3
        return [clamp, twist]

    def dirichlet_boundaries(self):
        return ["x[0] == 0.0", "x[0] == 1.0"]

    def material_model(self):
        # Material parameters can either be numbers or spatially
        # varying fields. For example,
        mu       = 3.8461
        lmbda    = Expression("x[0]*5.8 + (1 - x[0])*5.7")
        C10 = 0.171; C01 = 4.89e-3; C20 = -2.4e-4; C30 = 5.e-4

        #material = MooneyRivlin([mu/2, mu/2])
        material = StVenantKirchhoff([mu, lmbda])
        #material = Isihara([C10, C01, C20])
        #material = Biderman([C10, C01, C20, C30])
        return material

# Setup and solve the problem
twist = Twist()
u = twist.solve()
```

A simple static calculation involving a twisted block



A solid block twisted by 60 degrees

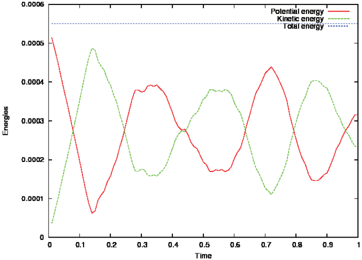
Iteration	Res. Norm
1	2.397e+00
2	6.306e-01
3	1.495e-01
4	4.122e-02
5	4.587e-03
6	8.198e-05
7	4.081e-08
8	1.579e-14

Newton scheme convergence

The dynamic release of the twisted block

```
class Release(Hyperelasticity):

        ...

    def end_time(self):
        return 10.0

    def time_step(self):
        return 2.e-3

    def reference_density(self):
        return 1.0

    def initial_conditions(self):
        """Return initial conditions for displacement field, u0, and
        velocity field, v0"""
        u0 = "twisty.txt"
        v0 = Expression(("0.0", "0.0", "0.0"))
        return u0, v0

    def dirichlet_conditions(self):
        clamp = Expression(("0.0", "0.0", "0.0"))
        return [clamp]

    def dirichlet_boundaries(self):
        return ["x[0] == 0.0"]

    def material_model(self):
        material = StVenantKirchhoff([3.8461, 5.76])
        return material

# Setup and solve the problem
release = Release()
u = release.solve()
```

The dynamic release of the twisted block



The relaxation of the released block

Conservation of energy

A silly hyperelastic fish being forced by a “flow”

```
class FishyFlow(Hyperelasticity):

    def mesh(self):
        mesh = Mesh("dolphin.xml.gz")
        return mesh

    def end_time(self):
        return 10.0

    def time_step(self):
        return 0.1

    def neumann_conditions(self):
        flow_push = Expression(("force", "0.0"))
        flow_push.force = 0.05
        return [flow_push]

    def neumann_boundaries(self):
        everywhere = "on_boundary"
        return [everywhere]

    def material_model(self):

        material = MooneyRivlin([6.169, 10.15])
        return material

# Setup and solve the problem
fishy = FishyFlow()
u = fishy.solve()
```

A silly hyperelastic fish being forced by a “flow”

The tumbling of the hyperelastic fish!

Concluding remarks, and where you can obtain the code

- We have a general framework for isotropic, dynamic hyperelasticity
- The following extensions are being worked on:
  - Implementing other specific material models
  - Allow for multiple materials and anisotropy
  - Goal-oriented adaptivity
  - Introducing coupling with other physics (including FSI)

- FEniCS Project: `http://fenics.org/`
- FEniCS Project Installer: `https://launchpad.net/dorsal/`
  `bzr get lp:dorsal`
- cbc.solve: `https://launchpad.net/cbc.solve/`
  `bzr get lp:cbc.solve`