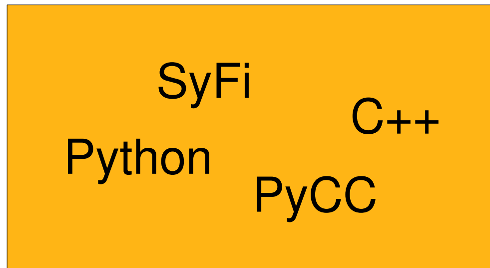
Application of symbolic finite element tools to nonlinear elasticity

Martin Sandve Alnæs,
Kent-André Mardal and Joakim Sundnes

MekIT'07
24/05 2007

[simula . research laboratory]

Outline



- Software concepts and components

- Hyperelasticity equations

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega_0} \mathbf{FS} : \nabla \mathbf{v} \, d\mathbf{x}$$

$$S_{ij} = \frac{\partial \Psi}{\partial E_{ij}}$$

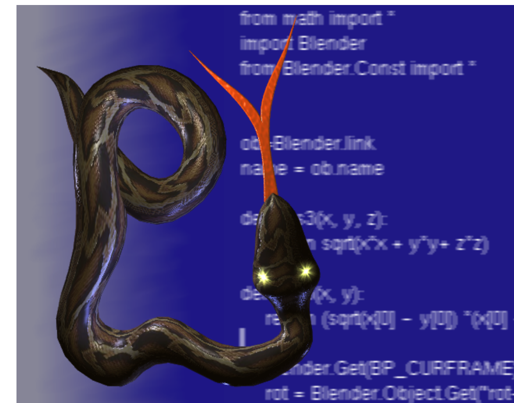
```
elasticity_form = compile_form(?)
A = asm.assemble_matrix(elasticity_form, ...)
```

- Implementation

PyCC is a high level framework for writing computational software

```
mesh = UnitSquare(10,10)
asm = Assembler(mesh)
A = asm.assemble_matrix(Ae)
B = MLPrec(A)
u = conjgrad(A, b, u, B)
```

- Assembling matrices/vectors
- Linear algebra solvers
- Gluing existing software together (f.ex. using ML)



Unified Form-assembly Code (UFC)

```
mesh = UnitSquare(10,10)
asm = Assembler(mesh)
A = asm.assemble_matrix(Ae)
B = MLPrec(A)
u = conjgrad(A, b, u, B)
```

- *Ae* is an implementation of the *UFC* interface
- *Ae* knows how to compute the element matrix for a given problem with a given choice of elements

SyFi uses symbolic computations to define finite elements

```
triangle = ReferenceTriangle()
fe = VectorLagrange(triangle)
print fe.N(0)
print grad(fe.N(0))
```

→

```
[[1-x-y],[0]]
[[-1,0],[-1,0]]
```

- Lets you compose a symbolic expression for a weak form using *grad(u)*, *div(u)* etc.
- Supports many advanced elements

The SyFi Form Compiler generates equation-specific code (1)

- The integrand of a weak form is written using the symbolic tools of SyFi

$$a(u, v) = \int_\Omega \nabla u \nabla v \, dx$$

→

```
# user code:
def a(u, v, G, GinvT):
    Du = grad(u, GinvT)
    Dv = grad(v, GinvT)
    return inner(Du, Dv)
```

The SyFi Form Compiler generates equation-specific code (2)

- Given a symbolic weak form, SFC generates UFC-compatible C++ code

```
// generated code:
void ...::tabulate_tensor(...) const
{
  ...
  A[3*0 + 0] = detG*( (Ginv00*Ginv00)/2.0 + ... )
  A[3*0 + 1] = detG*(-(Ginv00*Ginv00)/2.0 - ... )
  A[3*0 + 2] = detG*(-(Ginv01*Ginv01)/2.0 - ... )
  ...
}
```

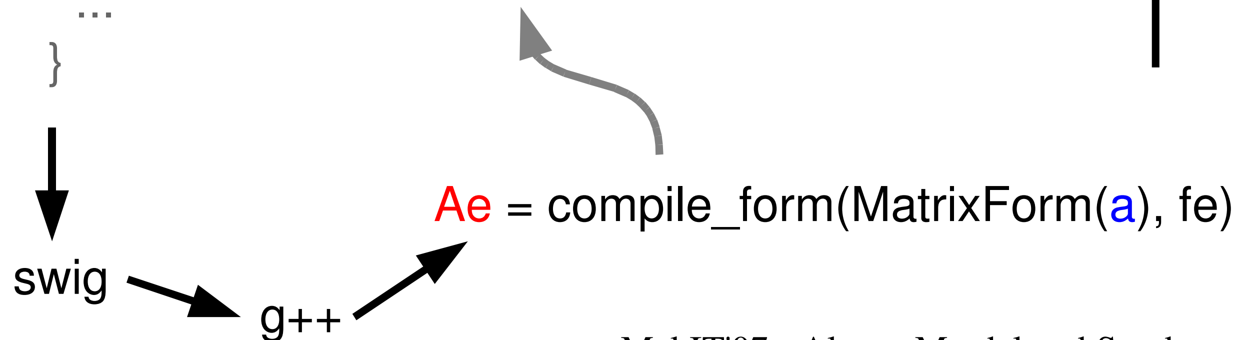
```
# user code:
def a(u, v, G, GinvT):
    Du = grad(u, GinvT)
    Dv = grad(v, GinvT)
    return inner(Du, Dv)
```

```
compile_form(MatrixForm(a), fe)
```

Automated Python-wrapping and compilation

```
// generated code:
void ...::tabulate_tensor(...) const
{
  ...
  A[3*0 + 0] = detG*( (Ginv00*Ginv00)/2.0 + ... )
  A[3*0 + 1] = detG*(-(Ginv00*Ginv00)/2.0 - ... )
  A[3*0 + 2] = detG*(-(Ginv01*Ginv01)/2.0 - ... )
  ...
}
```

```
# user code:
def a(u, v, G, GinvT):
    Du = grad(u, GinvT)
    Dv = grad(v, GinvT)
    return inner(Du, Dv)
```

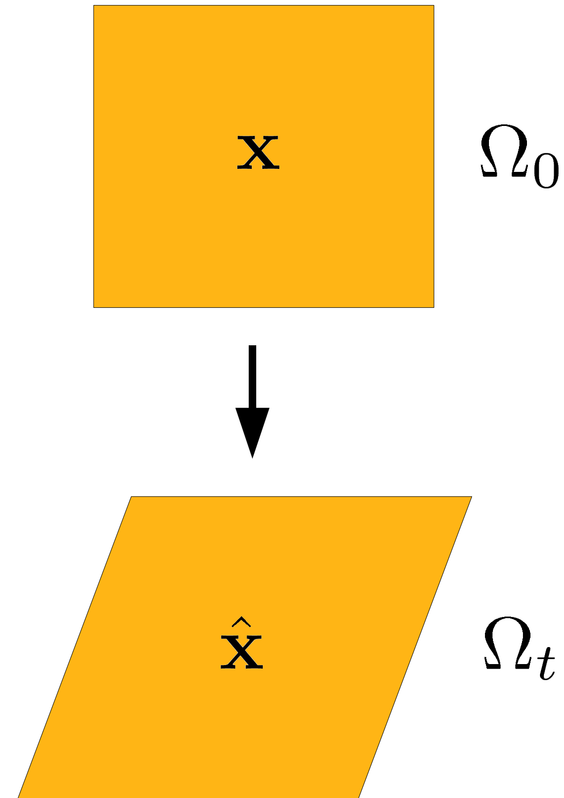


Large deformation elasticity basics

$$\mathbf{u} = \hat{\mathbf{x}} - \mathbf{x}$$

$$\mathbf{F} = \frac{\partial \hat{\mathbf{x}}}{\partial \mathbf{x}} = \mathbf{I} + (\nabla \mathbf{u})^T$$

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^T \mathbf{F} - \mathbf{I})$$



Full Lagrangian hyperelasticity (simplified)

Equilibrium equations:

$$\nabla \cdot (\mathbf{FS}) = 0$$

Weak formulation:

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega_0} \mathbf{FS} : \nabla \mathbf{v} \, d\mathbf{x}$$

General hyperelastic stress-strain relation:

$$S_{ij} = \frac{\partial \Psi}{\partial E_{ij}}$$

Heart tissue can be described by a Fung type material law

A transversely isotropic Fung type material law:

$$Q = b_{ff} E_{ff}^2 + b_{xx}(E_{nn}^2 + \cdots) + b_{fx}(E_{fn}^2 + \cdots)$$

$$\Psi(\mathbf{E}) = \frac{1}{2} K (e^Q - 1)$$

In the linearized system, we also need:

$$C_{ijkl} = \frac{\partial^2 \Psi}{\partial E_{ij} \partial E_{kl}}$$

Tedious differentiation of the stress tensor can be automated

$$S_{ij} = \frac{\partial \Psi}{\partial E_{ij}}$$

```
Es = symbolic_matrix(nsd, nsd, "E")
p = psi.value(Es)
S = diff(p, Es)
```

```
class Fung(StrainEnergy):
    def value(self, E):
        ...
        Q = bff*Eff**2 + bxx*(Enn**2 + ...) + bfx*(Efn**2 + ...)
        return K * (exp(Q) - 1) / 2
```

The SFC implementation stays close to the weak form

```
def finite_elasticity_B(v, u, f, G, GinvT, psi):
    ...
    Fs, Es, Ss = symbolic_matrices(nsd, ["F", "E", "S"])
    Du   = grad(u, GinvT)
    Dv   = grad(v, GinvT)
    F    = I + Du.transpose()
    FTs  = Fs.transpose()
    E    = (FTs * Fs - I) / 2
    p    = psi.value(Es)
    S    = diff(p, Es)
    integrand = contract(Fs*Ss, Dv) - inner(f, v)
    ...
```

Automated optimization techniques can reduce code size

- «Compile time computations»
- Factorization of symbolic expressions
- Finding common subexpressions
- Work in progress!

Conclusions

- Code generation can make the application code abstract while keeping runtime efficiency
- Some more work is needed to handle complex problems more efficiently

- Get SyFi at www.fenics.org/syfi/
- PyCC is currently not distributed

Questions?

- Code generation can make the application code abstract while keeping runtime efficiency
- Some more work is needed to handle complex problems more efficiently

- Get SyFi at www.fenics.org/syfi/
- PyCC is currently not distributed