Application of symbolic finite element tools to nonlinear hyperelasticity

Martin Sandve Alnæs*
Department of Scientific Computing
Simula Research Laboratory
e–mail: martinal@simula.no

Kent-André Mardal and Joakim Sundnes
Department of Scientific Computing
Simula Research Laboratory

Summary The present paper addresses the use of high level languages, symbolic mathematical tools and code generation in an implementation of the finite element method, using a nonlinear hyperelasticity equation as example. Advantages of the software development method that will be demonstrated include closeness to the mathematics, enabling high human efficiency with easy to use high level languages, while still keeping a high computational efficiency by generating tailored inner loop code for the problem at hand. The application we have in mind for the equations presented here is the simulation of the passive elastic properties of heart and blood vessel tissue.

Introduction

Biomechanical modeling often involves coupling of many physical phenomena, including elastic tissue deformations, fluid flow, and electrochemical processes. This often results in complicated mathematical models, for which it is challenging to design and implement efficient numerical methods. Researchers involved in modeling and simulation of biomedical processes are also likely to have very varied backgrounds, and in many cases limited experience with numerical methods and programming. These are good arguments for increasing the abstraction level in the simulator development, by using high level languages and tools that make the computer program closely resemble the mathematical model. Developing the tools that combine these abilities with good numerics and efficient implementation has been an active research topic for decades. One approach that has led to recent progress in this area is to wrap efficient implementations in compiled languages with high level scripting languages such as Python [2], which are currently gaining momentum in the scientific computing communities [6, 21, 23, 17, 18, 5, 19, 22, 3, 24, 20, 16, 9, 8, 14, 12]

Characteristic of most biomaterials is that they can undergo large elastic deformations, often governed by complicated non-linear constitutive laws. In this paper we will demonstrate the implementation of equations for nonlinear anisotropic hyperelasticity with our software SyFi [5, 15] and PyCC [18]. The properties of hyperelastic materials are defined in terms of a strain energy function, see e.g. [10] for details. Within the limits of hyperelastic materials, we want to easily apply new material laws, by simply specifying a new strain energy function. Through application of symbolic mathematics we can use automated differentiation of the material laws where applicable. As the engine for symbolic mathematics we use the C++ library GiNaC [1] and its python bindings swiginac [3]. A primary goal for the work is that the implementation should be close to the mathematics of the problem, in this case close to the weak formulation used for the finite element method. We assume the reader is familiar with continuum mechanics and the finite element method, but understanding the equations in detail should not be necessary to appreciate the ideas of this software development strategy.

The paper is outlined as follows. We first give a brief introduction to the mathematical model, followed by an outline of the applied numerical methods. We then present the corresponding

code that specifies the same equations and solves them. This is followed by a brief explanation of the code generation techniques, and finally some preliminary results.

Mathematical Model

Momentum equations

The general momentum balance equation, or equilibrium equation, reads

$$\frac{\partial^2 \mathbf{u}}{\partial t^2} = \nabla \cdot \boldsymbol{\sigma} + \mathbf{f}, \qquad \frac{\partial^2 u_j}{\partial t^2} = \frac{\partial \sigma_{ij}}{\partial x_i} + f_j, \tag{1}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor, $\mathbf{u}$ is the deformation vector field, and $\mathbf{f}$ is the body forces. In this paper we are primarily concerned with modelling the mechanical behavior of heart and blood vessel tissue, for which the acceleration term may be neglected. We thus focus on the stationary equation

$$\nabla \cdot \boldsymbol{\sigma} = -\mathbf{f}, \qquad \frac{\partial \sigma_{ij}}{\partial x_i} = -f_j, \tag{2}$$

which is here defined relative to the deformed geometry. Although Equation (2) describe a stationary situation, it is still relevant for describing the time dependent movement of the heart and blood vessels during normal physiological function. The equation must then be interpreted as a quasi-stationary equation, where the boundary conditions and source term change with time. More specifically, in a numerical time integration scheme, the work involves solving a number of problems of the kind (2), with a new source term and boundary conditions for each time step.

Finite hyperelasticity

In this section we describe a displacement based formulation of the equilibrium equations for a hyperelastic material. Since we allow large (finite) displacements, it is most convenient to use the Full Lagrangian formulation. With a Full Lagrangian formulation, the equations are formulated on a reference geometry instead of updating the geometry based on the deformation. Let the initial domain at time $t = 0$ be denoted Ω_0, and the deformed domain at time t be Ω_t. If the coordinate of a particle in the reference geometry Ω_0 is $\mathbf{x}$, then the coordinate of the same particle in Ω_t is $\hat{\mathbf{x}}$. With this in mind we can define the displacement $\mathbf{u}$ as

$$\mathbf{u} = \hat{\mathbf{x}} - \mathbf{x}, \qquad u_i = \hat{x}_i - x_i. \tag{3}$$

In the reference coordinates, the equilibrium equation reads

$$\nabla \cdot \mathbf{P} = -\mathbf{f}, \qquad \frac{\partial P_{ij}}{\partial x_i} = -f_j, \tag{4}$$

where $\mathbf{P}$ is known as the first Piola-Kirchoff stress tensor, which is related to the Cauchy stress tensor through a geometry mapping. The rest of this section describes the components needed to formulate material laws for this stress tensor.

We will need the deformation gradient $\mathbf{F}$, defined as

$$\mathbf{F} = \frac{\partial \hat{\mathbf{x}}}{\partial \mathbf{x}} = \mathbf{I} + \frac{\partial \mathbf{u}}{\partial \mathbf{x}} = \mathbf{I} + (\nabla \mathbf{u})^T, \qquad F_{ij} = \frac{\partial \hat{x}_i}{\partial x_j} = \delta_{ij} + \frac{\partial u_i}{\partial x_j}. \tag{5}$$

The deformation gradient can be decomposed into an orthonormal rotation tensor $\mathbf{R}$ and a stretch tensor $\mathbf{U}$, $\mathbf{F} = \mathbf{RU}$. Since $\mathbf{R}$ is orthonormal it follows that that $\mathbf{F}^T\mathbf{F} = \mathbf{U}^T\mathbf{U}$, which is independent of rigid body rotation. We can thus use the right Cauchy-Green strain tensor

$$\mathbf{C} = \mathbf{F}^T\mathbf{F}, \qquad C_{ij} = F_{ki}F_{kj}, \tag{6}$$

as a measure of pure stretch or strain, without rigid body motions. From the right Cauchy-Green tensor another important measure of strain may be derived, namely the Green-Lagrange strain tensor

$$\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I}), \qquad E_{ij} = \frac{1}{2}(C_{ij} - \delta_{ij}), \tag{7}$$

which fulfills $\mathbf{E} = \mathbf{0}$ for $\mathbf{u} = \mathbf{0}$. By assuming small deformations and neglecting higher order terms, the Green-Lagrange strain tensor can be shown to be equal to the strain tensor for small deformations $\boldsymbol{\epsilon}$. The right Cauchy-Green and the Green-Lagrange strain tensor are both defined relative to the reference geometry Ω_0.

It turns out the constitutive laws are easier formulated with the symmetric second Piola-Kirchoff stress tensor $\mathbf{S}$,

$$\mathbf{S} = \mathbf{F}^{-1}\mathbf{P}, \qquad S_{ij} = F_{ik}^{-1}P_{kj}, \tag{8}$$

since it is work-conjugate with the Green-Lagrange strain. This gives the final formulation of the equilibrium equations

$$\nabla\cdot(\mathbf{FS}) = -\mathbf{f}, \qquad \frac{\partial(F_{ik}S_{kj})}{\partial x_i} = -f_j. \tag{9}$$

Material laws

As noted in the introduction, the constitutive law for a hyperelastic material is specified by a strain energy function Ψ. The second Piola-Kirchoff stress tensor is given as partial derivatives of the strain energy with respect to the Green-Lagrange strain tensor components

$$\mathbf{S} = \frac{\partial\Psi}{\partial\mathbf{E}} = 2\frac{\partial\Psi}{\partial\mathbf{C}}, \qquad S_{ij} = \frac{\partial\Psi}{\partial E_{ij}} = 2\frac{\partial\Psi}{\partial C_{ij}}. \tag{10}$$

Many strain energy functions exist for different media, and it is of interest to be able to switch easily between these material laws in the software. In particular, constitutive laws for many biomaterials is an active research topic with few definite answers. Below we will use two different strain energy functions for demonstration.

A simple nonlinear material law for hyperelastic materials is the Saint Vernant-Kirchoff law

$$\Psi(\mathbf{E}) = \frac{1}{2}\lambda(trace\mathbf{E})^2 + \mu\mathbf{E} : \mathbf{E}. \tag{11}$$

One of the material laws that has been applied in modeling of passive heart tissue is the transversely isotropic Fung[7] type law

$$Q = b_{ff}E_{ff}^2 + b_{xx}(E_{nn}^2 + E_{ss}^2 + E_{sn}^2 + E_{ns}^2) + b_{fx}(E_{fn}^2 + E_{nf}^2 + E_{fs}^2 + E_{sf}^2), \tag{12}$$

$$\Psi(\mathbf{E}) = \frac{1}{2}K(e^Q - 1), \tag{13}$$

where f, s, and n refer to a coordinate system oriented with fibers in the material, in the fiber, sheet, and sheet normal directions respectively. See f.ex. [11] for more details. Examples of the implementation of both these constitutive equations and simple test cases are presented below.

```
def finite_elasticity_B(v, u, f, G, GinvT, psi):
    nsd  = v.nops()
    I    = Id(nsd)
    symbol_names = ["F", "E", "S"]
    Fs, Es, Ss = symbolic_matrices(nsd, symbol_names)

    Du  = grad(u, GinvT)
    Dv  = grad(v, GinvT)
    DuT = Du.transpose()
    F   = I + DuT
    FTs = Fs.transpose()
    E   = (FTs * Fs - I) / 2
    p   = psi.value(Es)
    S   = diff(p, Es)
    integrand = contract(Fs*Ss, Dv) - inner(f, v)

    tokens = [ (Fs, F), (Es, E), (Ss, S) ]
    return (integrand, tokens)
```

Figure 1: Implementation of the integrand $((\mathbf{F}(\mathbf{u})\mathbf{S}(\mathbf{u})) : \nabla\mathbf{v} - \mathbf{f}\cdot\mathbf{v})$ for the weak formulation of finite elasticity.

Numerical methods

Weak formulation

The finite element method relies on an equivalent form of the PDE (9) called the weak form. Although this can also be derived from physical principles, we view it here as a purely mathematical step in the formulation of the numerical discretization. The weak form is obtained by multiplying with a test function $\mathbf{v}$ and integrating over Ω_0;

$$\int_{\Omega_0} (\nabla\cdot\mathbf{FS} + \mathbf{f})\cdot\mathbf{v}d\mathbf{x} = 0, \qquad \int_{\Omega_0} (\frac{\partial(F_{ik}S_{kj})}{\partial x_i} + f_j)v_j = 0. \tag{14}$$

Assuming traction free or Dirichlet boundary conditions for simplicity, integration by parts now yields

$$\int_{\Omega_0} (\mathbf{FS} : \nabla\mathbf{v} - \mathbf{f}\cdot\mathbf{v})d\mathbf{x} = 0, \qquad \int_{\Omega_0} (F_{ik}S_{kj}\frac{\partial v_j}{\partial x_i} - f_j v_j) = 0. \tag{15}$$

To simplify the further discussion, we introduce the bilinear form

$$a(\mathbf{u},\mathbf{v}) = \int_{\Omega_0} \mathbf{FS} : \nabla\mathbf{v}d\mathbf{x}. \tag{16}$$

Discretization

We discretize the weak form (15) by the finite element method. Note that the symbol $\mathbf{u}$ will sometimes denote the physical deformation vector field, and sometimes the discrete solution vector from the linear system. The interpretation should be clear from the context. The vector

```
def finite_elasticity_rhs(u, psi):
    nsd  = u.nops()
    I    = Id(nsd)

    Du = grad(u)
    DuT = Du.transpose()

    F  = I + DuT
    FT = F.transpose()

    E = (FT * F - I) / 2

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

    for i in range(Es.nops()):
        S = S.subs( Es.op(i) == E.op(i) )

    f = -div( F*S )
    return f
```

Figure 2: Manufactured analytic right hand side computed from an analytic solution.

field $\mathbf{u}$ is approximated as a superposition of n vector basis functions $\mathbf{v}^j$ with coefficients u^j:

$$\mathbf{u} = \sum_{j=1}^{n} u^j \mathbf{v}^j \tag{17}$$

Applying test functions $\mathbf{v}^i$ for $i = 1, \ldots, n$ we get a system of non-linear algebraic equations

$$\mathbf{B}(\mathbf{u}) = \mathbf{0}, \tag{18}$$

where component i is given by

$$\mathbf{B}_i(\mathbf{u}) = a(\mathbf{u}, \mathbf{v}^i) - \int_{\Omega_0} \mathbf{f} \cdot \mathbf{v}^i \, d\mathbf{x} = 0, \qquad \text{for } i = 1, \ldots, n. \tag{19}$$

This system is to be solved for the unknown coefficients u^j in the vector $\mathbf{u}$.

Linearization

For solving of the nonlinear equations (19), we consider the well-known iterative Newton-Raphson method.

This requires the solution of linear systems of the form

$$\mathbf{J}\Delta\mathbf{u} = \mathbf{B}(\mathbf{u}), \qquad\qquad J_{ij}\delta u_j = B_i(u_k), \tag{20}$$

where $\mathbf{B}$ is the residual at the previous iteration, and $\mathbf{J}$ is the Jacobian with components given by

$$\mathbf{J}_{ij} = \frac{\partial}{\partial u^j} a(\mathbf{u}, \mathbf{v}^i) = \int_{\Omega_0} \frac{\partial(\mathbf{FS})}{\partial u^j} : (\nabla \mathbf{v}^i)^T d\mathbf{x}. \tag{21}$$

```
class StrainEnergy:
    def value(self, E):
        """Evaluate the strain energy Psi(E)."""
        pass

class SaintVenantKirchhoff(StrainEnergy):
    def __init__(self, lambd, mu):
        self.lambd = lambd
        self.mu = mu
    def value(self, E):
        return self.lambd * (trace(E)**2) / 2
                + self.mu * contract(E, E)

class Fung(StrainEnergy):
    def __init__(self, fiber, K, bff, bfx, bxx):
        self.fiber = fiber
        self.K   = K
        self.bff = bff
        self.bfx = bfx
        self.bxx = bxx
    def value(self, E):
        # Missing feature: assuming fiber == delta_ij
        Eff = E[0,0]
        ...
        Q = self.bff*Eff**2 + \
            self.bxx*(Enn**2 + Ess**2 + Esn**2 + Ens**2) + \
            self.bfx*(Efn**2 + Enf**2 + Efs**2 + Esf**2)
        return self.K * (exp(Q) - 1) / 2
```

Figure 3: Strain energy functions.

Recall that the stresses occurring in (21) are defined as partial derivatives of the strain energy function Ψ. For biomaterials, as we will see below, this function can become rather complicated, and performing the differentiation in (21) is a tedious task. A commonly applied solution strategy is to differentiate Ψ by hand to obtain analytical expressions for the stresses S_{ks}, and then do the final differentiation numerically, either by expanding the Jacobian or by differentiating the complete integrand directly. We will apply a different approach, where we apply automatic differentiation software to generate the code for analytic expressions of the entries in the Jacobian. We start by expanding the Jacobian into

$$\mathbf{J}_{ij} = \int_{\Omega_0} \left(\frac{\partial \mathbf{F}}{\partial u^j} \mathbf{S} + \mathbf{F} \frac{\partial \mathbf{S}}{\partial u^j} \right) : (\nabla \mathbf{v}^i)^T d\mathbf{x}, \tag{22}$$

$$J_{ij} = \int_{\Omega_0} \left(\frac{\partial F_{rk}}{\partial u^j} S_{ks} + F_{rk} \frac{\partial S_{ks}}{\partial u^j} \right) \left(\frac{\partial v_s^i}{\partial x_r} \right) dx. \tag{23}$$

In the computation of the element tensor we will fix i and j as one permutation at a time, as a practical step to reduce the order of the tensors we have to deal with. Since the test function v^i is

Algorithm 1 Newton's method.

Given $\mathbf{B}$, $\mathbf{u}$, and tolerance

$\epsilon = \|\mathbf{B}(\mathbf{u})\|$
while $\epsilon >$ tolerance
$\quad \mathbf{e} = \mathbf{J}^{-1}\mathbf{B}(\mathbf{u})$
$\quad \mathbf{u} = \mathbf{u} - \mathbf{e}$
$\quad \epsilon = \|\mathbf{B}(\mathbf{u})\|$

known, the term $\partial v_s^i/\partial x_r$ is trivial to compute in the software. The two terms inside the parenthesis require somewhat more care, and it is convenient to treat these separately. We assume u^j known, and first compute the quantities in $\frac{\partial F_{rk}}{\partial u^j} S_{ks}$ in the following order:

$$\mathbf{F} = \mathbf{I} + (\nabla \mathbf{u})^T, \qquad F_{rs} = \delta_{rs} + \frac{\partial u_r}{\partial x_s}, \tag{24}$$

$$\frac{\partial \mathbf{F}}{\partial u^j} = (\nabla \mathbf{v}^j)^T, \qquad \frac{\partial F_{rs}}{\partial u^j} = \frac{\partial v_r^j}{\partial x_s}, \tag{25}$$

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^T\mathbf{F} - \mathbf{I}), \qquad E_{rs} = \frac{1}{2}(F_{kr}F_{ks} - \delta_{ij}), \tag{26}$$

$$\mathbf{S} = \frac{\partial \Psi}{\partial \mathbf{E}}, \qquad S_{rs} = \frac{\partial \Psi}{\partial E_{rs}}. \tag{27}$$

Having completed these steps, we turn our attention to the second term $F_{rk}\frac{\partial S_{ks}}{\partial u^j}$. The computation of this term is somewhat more complicated, and we introduce two helper variables $\mathbf{H}$ and $\mathbb{C}$, which are tensors of rank two and four, respectively. The computation is then made in the following order

$$\mathbf{H}^j = \frac{1}{2}\left(\mathbf{F}^T\frac{\partial \mathbf{F}}{\partial u^j} + \frac{\partial \mathbf{F}^T}{\partial u^j}\mathbf{F}\right), \qquad H_{rs}^j = \frac{1}{2}\left(F_{kr}\frac{\partial F_{ks}}{\partial u^j} + \frac{\partial F_{kr}}{\partial u^j}F_{ks}\right), \tag{28}$$

$$\mathbb{C}_{pqrs} = \frac{\partial^2 \Psi}{\partial E_{pq}\partial E_{rs}} = \frac{\partial S_{rs}}{\partial E_{pq}}, \tag{29}$$

$$\frac{\partial S_{rs}}{\partial u^j} = H_{pq}^j \mathbb{C}_{pqrs}. \tag{30}$$

In the implementation we do not compute the rank four tensor directly, but use a loop over two indices and handle the "subtensor" as a matrix. With this approach, each component in $\frac{\partial S_{rs}}{\partial u^j}$ can be computed as a contraction of rank 2 tensors. The calculations (29)-(30) are then replaced by

$$\forall (r,s): \quad \frac{\partial \mathbf{S}_{rs}}{\partial u^j} = \mathbf{H}^j : \frac{\partial \mathbf{S}_{rs}}{\partial \mathbf{E}}. \qquad \forall (r,s): \quad \frac{\partial S_{rs}}{\partial u^j} = H_{pq}^j \frac{\partial S_{rs}}{\partial E_{pq}}. \tag{31}$$

We have now computed all the components we need, and the integrand may be calculated with simple matrix products and a tensor contraction using equation (23).

```
def finite_elasticity_J(u, v, w, G, GinvT, psi):
    nsd = u.nops()
    I   = Id(nsd)
    symbol_names = ["F", "E", "H", "S", "dS", "dFS"]
    Fs, Es, Hs, Ss, dSs, dFSs = \
        symbolic_matrices(nsd, symbol_names)

    Du = grad(u, GinvT)
    Dv = grad(v, GinvT)
    Dw = grad(w, GinvT)
    DuT = Du.transpose()
    DwT = Dw.transpose()

    F   = I + DwT
    dF  = DuT
    FTs = Fs.transpose()
    dFT = dF.transpose()

    E = (FTs * Fs - I) / 2

    p = psi.value(Es)
    S = diff(p, Es)

    # stress increment
    dS = zeros(nsd, nsd)
    H = ((dFT * Fs) + (FTs * dF)) / 2
    for r in range(nsd):
        for s in range(nsd):
            dS[r, s] = contract(Hs, diff(S[r,s], Es))

    dFS = ((dF * Ss) + (Fs * dSs))
    integrand = contract( dFSs, Dv )

    tokens = [ (Fs, F), (Es, E),   (Ss, S),
               (Hs, H), (dSs, dS), (dFSs, dFS) ]
    return (integrand, tokens)
```

Figure 4: Implementation of linearized integrand for the application of the Newton-Raphson method to the weak formulation of finite elasticity.

Implementation

In our implementation of the full Lagrangian finite elasticity equations, we strive to stay close to the mathematics. A normal approach often referred to as the engineering formulation is to manually take into account symmetries and sparsity of the fourth order elasticity tensor C_{ijkl} to define a smaller nine by nine matrix, while representing second order tensors as vectors. This formulation is easy to implement in traditional finite element software, but has little similarity with the original mathematical model. With the formulation shown in (31), combined with the finite element library SyFi (Symbolic Finite elements [5, 15]), we avoid this step and stay closer to the mathematical model in the implementation of the weak forms. Furthermore, a part of the process of implementing finite hyperelasticity is often to compute $\frac{\partial \Psi}{\partial \mathbf{E}}$ and $\frac{\partial^2 \Psi}{\partial \mathbf{E} \partial \mathbf{E}}$ manually, or with the help of external symbolic applications.) The expressions are then manually written into the C/C++/Fortran code. We instead utilize a symbolic library to perform the differentiation as part of our application and generate code from these expressions automatically. Therefore, adding a new Ψ is as simple as writing the expression for Ψ. Code for the implementation of the Saint Vernant-Kirchoff and Fung type laws are seen in Figure 3. An implementation of a strain energy function can also be used to specify quantities to compute as part of the postprocessing stage, like strains and stresses.

Code generation

SyFi uses symbolic computations to construct basis functions for various finite elements. It has support for a large set of elements, but in this paper we will stick to regular Lagrange elements on tetrahedra. Based on the explicit basis functions expressions, we can construct symbolic expressions for the integrand of a weak form, using symbolic differentiation for the differential operators. To make the user code close to the mathematics, differential operators like $\nabla \cdot \mathbf{u}$, $\nabla \mathbf{u}$ and $\frac{\partial \Psi}{\partial \mathbf{E}}$ are available as `div(u)`, `grad(u)` and `diff(psi, E)`, and the products $\mathbf{u} \cdot \mathbf{v}$ and $\mathbf{A} : \mathbf{B}$ are simply `inner(u,v)`, `contract(A,B)`.

If the weak form only contains polynomials and regular differential operators, SyFi can also perform the integration over an element symbolically. Since some material laws use exponential and logarithmic functions, this feature cannot be applied to our problem, and the generated code will instead apply quadrature for the integration over a cell.

From the resulting symbolic expressions for the weak form, SyFi can generate C++ code for the computation of the element matrix and element vector. The generated code is in a format specified by the Unified Form-assembly Code [4] (UFC) project. UFC consists of a set of abstract classes in a single header file, providing a predefined interface to the computation of an element matrix, evaluating finite element basis functions, mapping degrees of freedom and related operations. An example of generated low-level code is seen in Figure 6. In this code excerpt, `tabulate_tensor` is a function from the UFC interface, `A` is the element vector for $\mathbf{B}(\mathbf{u})$, and the variable names `Fxx`, `Exx`, `Sxx` etc. should be recognizable from the mathematical formulation, even if the low level expressions are not.

Implementing the weak formulation

Figure 1 shows the implementation of the weak form for $\mathbf{B}(\mathbf{u})$. This user-defined function (`finite_elasticity_B`) will be called by the code generation tools in SyFi at a later stage. The function will then get as input a symbolic expression for a test function `v`, the deformation

```
# <... Imports and initialization>

# Define forms to be compiled:
def elasticity_fung_J(u, v, w, fiber, K, bff, bfx, bxx, G, Ginv):
    psi = Fung(fiber, K, bff, bfx, bxx)
    return finite_elasticity_J(u, v, w, G, Ginv, psi)

def elasticity_fung_B(v, w, f, fiber, K, bff, bfx, bxx, G, Ginv):
    psi = Fung(fiber, K, bff, bfx, bxx)
    return finite_elasticity_B(v, w, f, G, Ginv, psi)

form_J    = MatrixForm(elasticity_fung_J, name="J_fung_3D")
form_B    = VectorForm(elasticity_fung_B, name="B_fung_3D")

# Initialize SyFi
nsd      = 3
order    = 1
qorder   = 6
SyFi.initSyFi(nsd)
polygon  = SyFi.ReferenceTetrahedra()
u_fe     = SyFi.VectorLagrange(polygon, order)
fiber_fe = SyFi.TensorP0(polygon)
fe0      = SyFi.P0(polygon)

# Define the finite elements to use for each argument:
# (ref. arguments to elasticity_fung_* above)
fe_list_J = (u_fe, u_fe, u_fe, fiber_fe, fe0, fe0, fe0, fe0)
fe_list_B = (u_fe, u_fe, u_fe, fiber_fe, fe0, fe0, fe0, fe0)

# Generate code for the forms and compile it
compiled_elasticity_form_J = compile_form(form_J, fe_list_J,
             integration_mode='quadrature', quad_order=qorder)

compiled_elasticity_form_B = compile_form(form_B, fe_list_B,
             integration_mode='quadrature', quad_order=qorder)
```

Figure 5: Compiling an element tensor.

```
void tabulate_tensor(double* A,
                     const double * const * w,
                     const ufc::cell& c) const
{
  ...
  static const double quad_weights[24] = {
      0.00665379170969, 0.00665379170969, ...
    };

  for(int iq=0; iq<24; iq++) {
    const double x = quad_points[iq][0];
    const double y = quad_points[iq][1];
    const double z = quad_points[iq][2];
    const double quad_weight_detG = quad_weights[iq] * detG;

    F00 = Ginv00*(-w[0][0]+w[0][3])+(-w[0][0]+w[0][9])*Ginv02
          +Ginv01*(w[0][6]-w[0][0])+1.0;
    F01 = Ginv12*(-w[0][0]+w[0][9])+...
    ...
    S22 = E22*exp(w[4][0]*((E01*E01)+(E10*E10)+...
    A[0] += ((S20*F12+S00*F10+S10*F11)*...
```

Figure 6: Excerpt of generated code for the computation of the element vector for $\mathbf{B}(\mathbf{u})$

field `u` from the previous iteration as a superposition of symbolic basis functions,

$$\mathbf{u} = \sum_{k=1}^{n_e} u^k \mathbf{v}^k, \tag{32}$$

the body force `f` in the same representation as `u`, symbolic representations of the geometry mappings G and G^{-T} for mapping to a reference element, and a material law definition represented by a `StrainEnergy` object `psi`. Later during the finite element assembly, the coefficients u^k in the symbolic representation will be input values from a finite element vector restricted to one element. The return value is a symbolic representation of the integrand for a single entry in the element vector, along with a list of tokens. The `tokens` list holds symbol/value pairs for variables that will be generated code for, and which the integrand expression depends on. After calling this user-defined function, the code generation tools will generate code for assigments to these variables and wrap this code in a quadrature loop. An excerpt of this generated code is shown in Figure 6.

Stepping through the middle part of `finite_elasticity_B`, each line shows a clear resemblance with equations (24)-(27) and (15). The symbolic variables can be matrices and vectors, greatly reducing error prone index handling. Computing gradients in the reference domain is performed with `grad(u, GinvT)`. To reduce the size of the expressions, symbolic matrices are used for $\mathbf{F}$, $\mathbf{E}$ and $\mathbf{S}$ to represent their values in dependent expressions. Notice in particular how the strain energy function `psi` is evaluated with a symbolic matrix `Es`, and the stress tensor is differentiated with respect to the same symbolic matrix with `diff(p, Es)`.

In Figure 4, similar code is shown for the computation of $\mathbf{J}(\mathbf{u})$. Notice that the fourth order elasticity tensor is never explicitly constructed, it only exists as a step in the algorithm formu-

```
K, bff, bfx, bxx = 876, 18.48, 2.8, 3.58
fiber = (1,0,0, 0,1,0, 0,0,1)

# <... initialize mesh, vectors, matrix, etc.>

while eps > newton_tolerance:
    # Collect coefficients to the form J
    # (ref. fe_list_J in previous code example)
    coeffs_J = [u, fiber, K, bff, bfx, bxx]
    assembler.assemble_matrix(compiled_elasticity_form_J,
                              coeffs_J, J_before_bc)

    # Modify boundary rows and columns in J
    (J, BC, C) = dirichlet_boundarycondition(J_before_bc,
                                             boundary_dofs)
    B = BC*B_before_bc # set boundary components to zero

    # Find and apply the correction
    du.fill(0)
    du = conjgrad(J, du, F)
    u -= du

    coeffs_B = [u, f, fiber, K, bff, bfx, bxx]
    assembler.assemble_vector(compiled_elasticity_form_B,
                              coeffs_B, B_before_bc)

    eps = L2(B_before_bc)
    iter += 1
```

Figure 7: Newton loop with assembly of linear system in each iteration.

lation in equation (31). Zeros and cancelling terms are automatically taken into account by the symbolic code generation tools, so that the resulting generated code for the computation of the element matrix and vector will be partially optimized before it is compiled.

Software verification

Another application of symbolic computations in finite element implementations is to verify the software with the method of manufactured solutions. First we define a set of (possibly unphysical) analytical solutions, and calculate the body force f required to obtain this solution using the strong formulation of the equilibrium equations (9). Figure 2 shows code for these calculations. Next we can solve the discrete equations with this calculated body force $\mathbf{f}$, and compare the computed solution with the original expression to find the error. This approach is particularly convenient when using complicated material laws like the Fung-law described previously.

Application code

PyCC (Python Computing Components [18]) is a high level Python framework for the implementation of PDE solvers in development at Simula. After defining the weak form of the equa-

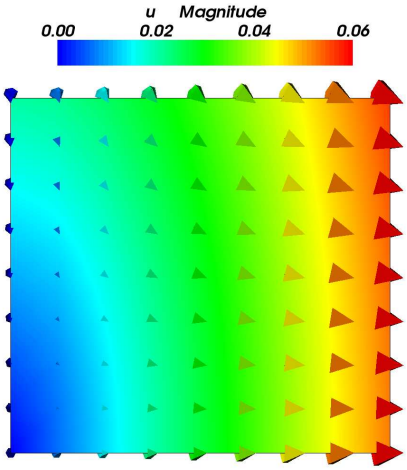

Figure 8: Testcase with SVK material law

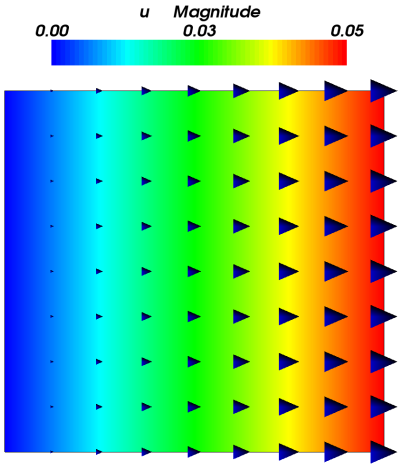

Figure 9: Testcase with Fung material law

tions like described above, and compiling the element matrix and element vector for $\mathbf{J}(\mathbf{u})$ and $\mathbf{B}(\mathbf{u})$ respectively, these compiled UFC forms can be loaded in a Python application and used by a PyCC Assembler object to assemble the global linear system inside a Newton-Raphson iteration, like shown in Figure 7. The linear equations in each iteration are solved with a conjugate gradient method implementation from PyCC. For simple visualization in the application script, a Python module called Viper is used, which is a thin layer on top of VTK [13]. Simulation results are written to file in VTK format, which were loaded in Paraview [20] (v2.9.9) to create the figures.

Test cases

As a simple test case, we apply Dirichlet boundary conditions to the x-component of $\mathbf{u}$ on two opposite sides of a cube, and leave the rest of the boundary traction-free. To remove the

possibility of a rigid body translation, we must also fix all components of $\mathbf{u}$ in one point.

$$\mathbf{u} \cdot \mathbf{e}_x = 0, \qquad x = 0, \tag{33}$$
$$\mathbf{u} \cdot \mathbf{e}_x = \alpha \mathbf{i}, \qquad x = 1, \tag{34}$$
$$\mathbf{u} = 0, \qquad \mathbf{x} = \mathbf{0}, \tag{35}$$
$$\mathbf{t} = 0, \qquad 0 < x < 1. \tag{36}$$

Figures 8 and 9 show the resulting deformation fields as glyphs and color-coded magnitude of $\mathbf{u}$ from one side of the cube with its normal vector in y-direction. The Fung law here yields a fully compressible deformation, while with the SVK law we see the cube is compressed, and the color-coded magnitude shows where the fixed point is.

Discussion

One of the key advantages of this implementation is that it is easy to add new material laws. Since the implementations of the weak forms and postprocessing quantities (not shown) are close to mathematical model formulation, they should be easily readable for people without a background in numerics and programming. For those who are used to the traditional engineering formulation of the elasticity equations this point may not be very important. The user is still subjected to technical implementation details in SyFi, so there is still a need to work more on the user interface of the library. This is work in progress.

Generated code from SyFi is highly efficient for simple equations of similar complexity as mass matrices and stiffness matrices, competing with or even outperforming traditional quadrature-based implementations. But for more complex equations like the finite hyperelasticity presented here there are challenges to overcome in the code generation. The generated code can grow quite large, and great care must be taken to keep the code size small. This problem grows for higher order elements.

In the current implementation and with the tests done so far, the time spent assembling the linear system dominates the Newton iterations for these equations. However, this is expected to improve significantly with future versions of SyFi, when more optimized code can be generated. Since the code generation tools has a more high level overview of the mathematical expressions than the C++ compiler will have at a later stage, it is possible to perform large optimizations by analyzing dependencies in the expressions. This role is traditionally filled by the human code implementer, who chooses the algorithms to use and tunes the code in a manual process. The current code generation tools in SyFi perform only very simple optimization steps, but improving this is work in progress. Quantifying the speedup cannot be done at this stage.

The software has not been tested with the most complicated material laws, only with unidirectional fiber directions and without compressibility constraints. There is also limited support for more advanced boundary conditions, which also must be adressed before truly relevant physiological applications can be attempted.

References

[1] GiNaC, 2006 http://www.ginac.de.

[2] Python, 2006 http://www.python.org/doc/.

[3] Swiginac, 2006 http://swiginac.berlios.de/.

[4] M. S.Alnæs, H. P.Langtangen, A.Logg, K.-A.Mardal and O.Skavhaug UFC, 2007 *http://www.fenics.org/ufc/*.

[5] M. S.Alnæs and K.-A.Mardal Syfi user manual http://www.fenics.org/pub/documents/syfi/syfi-user-manual/syfi-user-manual.pdf.

[6] D.Ascher, P. F.Dubois, K.Hinsen, J.Hugunin and T.Oliphant Numerical Python *http://www.pfdubois.com/numpy/*.

[7] Y.Fung *Biomechanics: mechanical properties of living tissues* Springer-Verlag New York, Inc., 1993.

[8] J.Hoffman, J.Jansson, C.Johnson, M. G.Knepley, R. C.Kirby, A.Logg, L. R.Scott and G. N.Wells *FEniCS*, 2006 *http://www.fenics.org/*.

[9] J.Hoffman, J.Jansson, A.Logg and G. N.Wells *DOLFIN*, 2006 *http://www.fenics.org/dolfin/*.

[10] G.Holzapfel *Nonlinear Solid Mechanics, A Continuum Approach for Engineering* John Wiley& Sons, Ltd, 2001.

[11] J. D.Humphrey *Cardiovascular Solid Mechanics* Springer-Verlag, 2002.

[12] R. C.Kirby *FIAT*, 2006 *http://www.fenics.org/fiat/*.

[13] Kitware The Visualization ToolKit, 2006 *http://www.vtk.org/*.

[14] A.Logg *FFC*, 2006 *http://www.fenics.org/ffc/*.

[15] K.-A.Mardal Syfi - an element matrix factory To appear in the PARA'06 proceedings to be published in the Springer series Lecture Notes in Computer Science (LNCS).

[16] MayaVi package *http://mayavi.sourceforge.net*.

[17] PETSc software package *www.mcs.anl.gov/petsc/*.

[18] PyCC, 2007 Software framework under development. *http://www.simula.no/pycc/*.

[19] PySE software package *http://pyfdm.sf.net*.

[20] Sandia National Laboratories ParaView, 2006 *http://www.paraview.org/*.

[21] SciPy software package *http://www.scipy.org*.

[22] SWIG software package *http://www.swig.org*.

[23] Trilinos software package *http://software.sandia.gov/trilinos*.

[24] Vtk package *http://www.vtk.org*.