24 A computational framework for nonlinear elasticity

By Harish Narayanan

Nonlinear elasticity theory plays a fundamental role in modeling the mechanical response of many polymeric and biological materials. Such materials are capable of undergoing finite deformation, and their material response is often characterized by complex, nonlinear constitutive relationships. (See, for example, [Holzapfel, 2000] and [Truesdell and Noll, 1965] and the references within for several examples.) Because of these difficulties, predicting the response of arbitrary structures composed of such materials to arbitrary loads requires numerical computation, usually based on the finite element method. The steps involved in the construction of the required finite element algorithms are classical and straightforward in principle, but their application to non-trivial material models are typically tedious and error-prone. Our recent work on an automated computational framework for nonlinear elasticity, `CBC.Twist`, is an attempt to alleviate this problem.

The focus of this chapter will be to describe the design and implementation of `CBC.Twist`, as well as providing examples of its use. The goal is to allow researchers to easily pose and solve problems in nonlinear elasticity in a straightforward manner, so that they may focus on higher-level modeling questions without being hindered by specific implementation issues.

What follows is the proposed outline for the chapter.

The chapter begins with a summary of some key results from classical nonlinear elasticity theory. This discussion is used to motivate the design of `CBC.Twist`, which is a DOLFIN [Logg and Wells, 2010] module written in UFL syntax [Alnæs and Logg, 2009] that closely resembles how the theory is written down on paper. In particular, we will see how one can easily pose sophisticated material models purely at the level of specifying a strain energy function. The discourse will then turn to the primary equation of interest: the balance of linear momentum of a body posed in the reference configuration. A finite element scheme for this equation will then be presented, pointing out how `CBC.Twist` leverages the automatic linearization capabilities of UFL to implement this scheme in a manner that is independent of the specific material model. The time-stepping schemes that `CBC.Twist` implements will also be discussed. With this in place, we turn to increasingly complex examples to see how initial- boundary-value problems in nonlinear elasticity can be posed and solved in `CBC.Twist` using only a few lines of high-level code. The chapter concludes with some remarks on how one can obtain `CBC.Twist`, along with ideas for its extension.

24.1 Brief overview of nonlinear elasticity theory as it relates to `CBC.Twist`

The goal of this section is to present an overview of the mathematical theory of nonlinear elasticity, which plays an important role in the design of `CBC.Twist`. Readers interested in a more comprehensive treatment of the subject are referred to, for example, the classical treatises of Truesdell and Toupin [1960] and Truesdell and Noll [1965], or more modern works such as Gurtin [1981], Ogden [1997] and Holzapfel [2000].

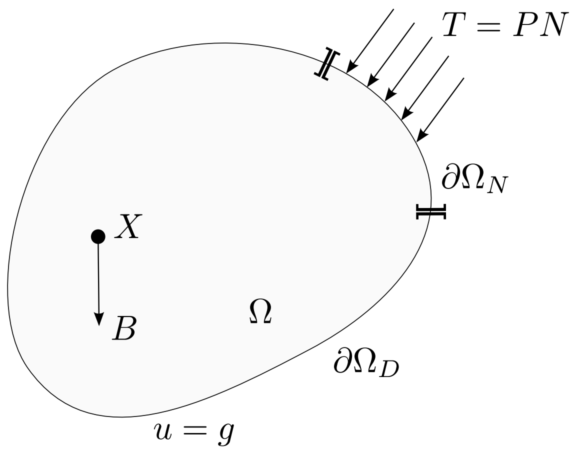


Figure 24.1: An elastic body idealized as a continuum, subjected to body forces, B, surface tractions, T, and prescribed displacement boundary conditions.

Posing the question we aim to answer

The theory begins by idealizing the elastic body of interest as an open subset of $\mathbb{R}^{2,3}$ with a piecewise smooth boundary. At a *reference* placement of the body, Ω, points in the body are identified by their reference positions, $X \in \Omega$. The treatment presented in this chapter is posed in terms of fields which are parametrized by reference positions. This is commonly termed the *material* or *Lagrangian* description.

In its most basic terms, the *deformation* of the body over a time $t \in [0, T]$ is a sufficiently smooth bijective map $\varphi : \overline{\Omega} \times [0, T] \to \mathbb{R}^{2,3}$, where $\overline{\Omega} := \overline{\Omega \cup \partial\Omega}$ and $\partial\Omega$ is the boundary of Ω. The restrictions on the map ensure that the motion it describes is physical (e.g., disallowing the interpenetration of matter or the formation of cracks). From this, we can construct the *displacement field*,

$$u(X, t) = \varphi(X, t) - X, \tag{24.1}$$

which represents the displacement of a point in time relative to its reference position.

With this brief background, we are ready to pose the fundamental question that `CBC.Twist` is designed to answer: Given a body comprised of a specified elastic material, what is the displacement of the body when it is subjected to prescribed:

- *Body forces:* These include forces such as the self-weight of a body, forces on ferromagnetic materials in magnetic fields, etc., which act everywhere in the volume of the body. They are denoted by the vector field $B(X, t)$.
- *Traction forces:* This is the force measured per unit surface area acting on the *Neumann* boundary of the body, $\partial\Omega_{\mathrm{N}}$, and denoted by the vector field $T(X, t)$.
- *Displacement boundary conditions:* These are displacement fields prescribed on the *Dirichlet* boundary of the body, $\partial\Omega_{\mathrm{D}}$.

It is assumed that $\partial\Omega_{\mathrm{N}} \cap \partial\Omega_{\mathrm{D}} = \emptyset$ and $\overline{\partial\Omega_{\mathrm{N}} \cup \partial\Omega_{\mathrm{D}}} = \partial\Omega$. These details are depicted in Figure 24.1.

The basic equation we need to solve

In order to determine the displacement of an elastic body subjected to these specified loads and boundary conditions, we turn to a fundamental law called the *balance of linear momentum*. This is a law which is valid for all materials and must hold for all time. `CBC.Twist` solves the Lagrangian form of this equation, which is presented below in local form that is pertinent to numerical implementation by the finite element method:

$$\rho \frac{\partial^2 u}{\partial t^2} = \mathrm{Div}(P) + B \text{ in } \Omega, \tag{24.2}$$

Infinitesimal strain tensor	$\epsilon = \frac{1}{2}\left(\mathrm{Grad}(u) + \mathrm{Grad}(u)^{\mathrm{T}}\right)$
Deformation gradient	$F = 1 + \mathrm{Grad}(u)$
Right Cauchy–Green tensor	$C = F^{\mathrm{T}}F$
Green–Lagrange strain tensor	$E = \frac{1}{2}(C - 1)$
Left Cauchy–Green tensor	$b = FF^{\mathrm{T}}$
Euler–Almansi strain tensor	$e = \frac{1}{2}\left(1 - b^{-1}\right)$
Volumetric and isochoric decomposition of C	$\bar{C} = J^{-\frac{2}{3}}C, \quad J = \mathrm{Det}(F)$
Principal invariants of C	$I_1 = \mathrm{Tr}(C)$, $I_2 = \frac{1}{2}\left(I_1^2 - \mathrm{Tr}(C^2)\right)$, $I_3 = \mathrm{Det}(C)$
Principal stretches and directions	$C = \sum_{A=1}^{3} \lambda_A^2 N^A \otimes N^A, \quad \lvert\lvert N^A \rvert\rvert = 1$

Table 24.1: Definitions of some common strain measures.

where ρ is the *reference density* of the body, P is the *first Piola–Kirchhoff stress tensor*, $\mathrm{Div}(\cdot)$ is the divergence operator and B is the body force per unit volume. Along with (24.2), we have initial conditions $u(X,0) = u_0(X)$ and $\frac{\partial u}{\partial t}(X,0) = v_0(X)$ in Ω, and boundary conditions $u(X,t) = g(X,t)$ on $\partial\Omega_{\mathrm{D}}$ and $PN = T$ on $\partial\Omega_{\mathrm{N}}$. Here, N is the outward normal on the boundary.

We focus on the balance of linear momentum because, in a continuous sense, the other fundamental balance principles that materials must obey—the balance of mass (continuity equation), balance of angular momentum and balance of energy—are each trivially satisfied[1] in the Lagrangian description by elastic materials with suitably chosen stress responses.

Accounting for different materials

It is important to reiterate that (24.2) is valid for all materials. In order to differentiate between different materials and to characterize their specific mechanical responses, the theory turns to *constitutive relationships*, which are models for describing the real mechanical behavior of matter. In the case of nonlinear elastic (or *hyperelastic*) materials, this description is posed in the form of a stress-strain relationship through an objective and frame-indifferent Helmholtz free energy function called the *strain energy function*, ψ. This is an energy defined per unit reference volume and is solely a function of the local *strain measure*. Comprehensive texts on the subject (e.g. Holzapfel [2000]) cover the motivations for defining different forms of strain measures, but in this chapter we just provide the definitions of some of the most common forms. In what follows, $\mathrm{Grad}(\cdot)$ is the gradient operator, and $\mathrm{Tr}(\cdot)$ and $\mathrm{Det}(\cdot)$ are the trace and determinant of $\cdot$, respectively.

In `CBC.Twist`, each of the forms listed in Table 24.1 have been implemented in the file `kinematics.py` in UFL notation that closely resemble their definitions above. Figure 24.2 presents a section of this file. Notice that it is straightforward to introduce other custom measures as required.

The stress response of isotropic hyperelastic materials (the class of materials `CBC.Twist` restricts its attention to) can be derived from the scalar-valued strain energy function. In particular, the tensor known as the *second Piola–Kirchhoff stress tensor* is defined using the following *constitutive relationship*:

$$S = F^{-1}\frac{\partial\psi(F)}{\partial F}. \tag{24.3}$$

[1]It should be noted that the story is not so simple in the context of numerical approximations. For instance, when modeling (nearly) incompressible materials, it is well known that the ill-conditioned stiffness matrix resulting from the conventional Galerkin approximation (discretizing only the displacement field) can result in *volumetric locking*. One can work around this difficulty by resorting to a mixed formulation of the Hu–Washizu type [Simo and Hughes, 1998], but such a formulation is beyond the scope of the current chapter. `CBC.Twist` can be extended to such a formulation, but for now, we circumvent the problem by restricting our attention to compressible materials.

Python code

```
# 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))

# 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]

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

# Isochoric part of the right Cauchy-Green tensor
def IsochoricRightCauchyGreen(u):
    C = RightCauchyGreen(u)
    J = Jacobian(u)
    return variable(J**(-2.0/3.0)*C)
```

Figure 24.2: Samples of how strain measures are implemented in `CBC.Twist`. Notice that the definitions in the implementation closely resemble the classical forms introduced in Table 24.1.

Python code
```python
def SecondPiolaKirchhoffStress(self, u):

  ...

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

  ...
  return S
```

Figure 24.3: Partial listing of the method that suitably computes the second Piola–Kirchhoff stress tensor based on the strain measure.

The second Piola–Kirchhoff stress tensor is related to the first Piola–Kirchhoff stress tensor introduced earlier through the relation, $P = FS$.

As already mentioned, the strain energy function can be posed in equivalent forms in terms of different strain measures. (Again, the reader is directed to classical texts to motivate this.) In order to then arrive at the second Piola–Kirchhoff stress tensor, we turn to the chain rule of differentiation. For example,

$$\begin{aligned} S &= 2\frac{\partial\psi(C)}{\partial C} = \frac{\partial\psi(E)}{\partial E} \\ &= 2\left[\left(\frac{\partial\psi(I_1,I_2,I_3)}{\partial I_1} + I_1\frac{\partial\psi(I_1,I_2,I_3)}{\partial I_2}\right)1 - \frac{\partial\psi(I_1,I_2,I_3)}{\partial I_2}C + I_3\frac{\partial\psi(I_1,I_2,I_3)}{\partial I_3}C^{-1}\right] \\ &= \sum_{A=1}^{3}\frac{1}{\lambda_A}\frac{\partial\psi(\lambda_1,\lambda_2,\lambda_3)}{\partial\lambda_A}N^A\otimes N^A = \dots \end{aligned} \tag{24.4}$$

Using definitions such as the ones explicitly provided in (24.4), `CBC.Twist` computes the second Piola–Kirchhoff stress tensor from the strain energy function by suitably differentiating it with respect to the appropriate strain measure. This allows the user to easily specify material models in terms of each of the strain measures introduced in Table 24.1. The base class for all material models, `MaterialModel`, encapsulates this functionality. The relevant method of this class is provided in Figure 24.3. The implementation relies heavily on the UFL `diff` operator.

The generality of the material model base class allows for the (almost trivial) specification of a large set of models. To see this in practice, let us consider two popular material models,

- the *St. Venant–Kirchhoff* model: $\psi_{\mathrm{SVK}} = \frac{\lambda}{2}\mathrm{Tr}(E)^2 + \mu\mathrm{Tr}(E^2)$, and
- the two term *Mooney–Rivlin* model: $\psi_{\mathrm{MR}} = c_1(I_1 - 3) + c_2(I_2 - 3)$,

and see how they can be specified in `CBC.Twist`. The relevant blocks of code are shown in Figures 24.4 and 24.5.
Clearly, the code simply contains the strain energy function in classical notation, along with some metadata clarifying the number of material parameters and the strain measure the model relies on. The file `material_models.py` contains several other material models, including *linear elasticity, neo Hookean, Isihara,*

Python code
```
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)
```

Figure 24.4: Definition the strain energy function for a St. Venant–Kirchhoff material.

Python code
```
class MooneyRivlin(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*(I2 - 3)
```

Figure 24.5: Definition the strain energy function for a two term Mooney–Rivlin material.

Biderman, and *Gent–Thomas* that come pre-implemented in `CBC.Twist`. (Refer to the article by Marckmann and Verron [2006] comparing several hyperelastic models for rubber-like materials for their definitions.) But the salient point to note here is that it is straightforward to introduce other additional models, and this is a significant feature of `CBC.Twist`.

24.2 *Numerical methods and further implementation details*

In the preceding section, we saw the functionality that `CBC.Twist` provided to easily specify material models to suitably characterize different materials of interest. In this section, we return to the general form of the balance of linear momentum and look at details of a finite element formulation and implementation for this equation. For further details on the treatment that follows, the interested reader is directed to Simo and Hughes [1998].

The finite element formulation of the balance of linear momentum

By taking the dot product of (24.2) with a test function $v \in \hat{\mathrm{V}}$ and integrating over the reference domain and time, we have

$$\int_0^T \int_\Omega \rho \frac{\partial^2 u}{\partial t^2} \cdot v \,\mathrm{d}x\,\mathrm{d}t = \int_0^T \int_\Omega \mathrm{Div}(P) \cdot v \,\mathrm{d}x\,\mathrm{d}t + \int_0^T \int_\Omega B \cdot v \,\mathrm{d}x\,\mathrm{d}t. \tag{24.5}$$

Noting that the traction vector $T = PN$ on $\partial\Omega_\mathrm{N}$ (N being the outward normal on the boundary) and that by definition $v|_{\partial\Omega_\mathrm{D}} = 0$, we apply the divergence theorem to arrive at the following weak form of the balance of linear momentum:
Find $u \in \mathrm{V}$, such that $\forall\, v \in \hat{\mathrm{V}}$:

$$\int_0^T \int_\Omega \rho \frac{\partial^2 u}{\partial t^2} \cdot v \,\mathrm{d}x\,\mathrm{d}t + \int_0^T \int_\Omega P : \mathrm{Grad}(v) \,\mathrm{d}x\,\mathrm{d}t = \int_0^T \int_\Omega B \cdot v \,\mathrm{d}x\,\mathrm{d}t + \int_0^T \int_{\partial\Omega_\mathrm{N}} T \cdot v \,\mathrm{d}s\,\mathrm{d}t, \tag{24.6}$$

with initial conditions $u(X,0) = u_0(X)$ and $\frac{\partial u}{\partial t}(X,0) = v_0(X)$ in Ω, and boundary conditions $u(X,t) = g(X,t)$ on $\partial\Omega_\mathrm{D}$.

The finite element formulation implemented in `CBC.Twist` follows the Galerkin approximation of the above weak form (24.6), by looking for solutions in a finite solution space $\mathrm{V}_h \subset \mathrm{V}$ and allowing for test functions in a finite approximation of the test space $\hat{\mathrm{V}}_h \subset \hat{\mathrm{V}}$.[2]

Implementation of the static form

We consider first the static weak form (dropping the time derivative term) of the balance of linear momentum which reads

$$\int_\Omega P : \mathrm{Grad}(v)\,\mathrm{d}x - \int_\Omega B \cdot v\,\mathrm{d}x - \int_{\partial\Omega_\mathrm{N}} T \cdot v\,\mathrm{d}s = 0. \tag{24.7}$$

Since `CBC.Twist` provides the necessary functionality to easily compute the first Piola–Kirchhoff stress tensor, P, given a displacement field, u, for arbitrary material models, (24.7) is just a nonlinear functional in terms of u. The automatic differentiation capabilities of UFL[3] make this nonlinear form straightforward to implement, as evidenced by the code listing in Figure 24.6.

This listing provides the relevant section of the static balance of linear momentum solver class, `StaticMomentumBalanceSolver`. The class draws information about the problem (mesh, loading, boundary conditions and form of the stress equation derived from the material model) from the user-specified problem class,[4] and solves the nonlinear momentum balance equation using a Newton solver.

Time-stepping algorithms

`CBC.Twist` implements two time integration algorithms to solve the weak form of the fully dynamic balance of linear momentum (24.6). The first of these is the so-called CG$_1$ method [Eriksson et al., 1996]. In order to derive this method, (24.6), which is a second order differential equation in time, is rewritten as a system of first order equations. We do this by introducing an additional velocity variable, $w = \frac{\partial u}{\partial t}$. Thus, the weak form now reads:

[2]We now note an inherent advantage in choosing the Lagrangian description in formulating the theory. The fact that the integrals in (24.6), along with the various fields and differential operators, are defined over the fixed domain Ω means that one need not be concerned with the complexity associated with calculations on a moving computational domain when implementing this formulation.

[3]An earlier chapter on UFL (17) provides a detailed look at the capabilities of UFL, as well as insights into how it achieves its functionality. Even so, we note the following differentiation capabilities of UFL because of their pivotal relevance to this work:

- Computing spatial derivatives of fields, which allows for the construction of differential operators such as such as $\mathrm{Grad}(\cdot)$ or $\mathrm{Div}(\cdot)$:
  ```
  df_i = Dx(f, i)
  ```
- Differentiating arbitrary expressions with respect to variables they are functions of:
  ```
  g = variable(cos(cell.x[0]))
  f = exp(g**2)
  h = diff(f, g)
  ```
- Differentiating forms with respect to coefficients of a discrete function, allowing for automatic linearizations of nonlinear variational forms:
  ```
  a = derivative(L, w, u)
  ```

[4]Details of how the user can specify problem details are covered in the following section containing examples of `CBC.Twist` usage.

Python code

```python
# 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)
```

Figure 24.6: The relevant section of the class `StaticMomentumBalanceSolver`, the solver for the static balance of linear momentum.

Find $(u, w) \in \mathrm{V}$, such that $\forall\, (v, r) \in \hat{\mathrm{V}}$:

$$\begin{aligned} \int_0^T \int_\Omega \rho \frac{\partial w}{\partial t} \cdot v \,\mathrm{d}x\,\mathrm{d}t \;+ \int_0^T \int_\Omega P : \mathrm{Grad}(v)\,\mathrm{d}x\,\mathrm{d}t \;&= \int_0^T \int_\Omega B \cdot v \,\mathrm{d}x\,\mathrm{d}t \;+ \int_0^T \int_{\partial\Omega_\mathrm{N}} T \cdot v \,\mathrm{d}s\,\mathrm{d}t, \text{ and} \\ \int_0^T \int_\Omega \frac{\partial u}{\partial t} \cdot r \,\mathrm{d}x\,\mathrm{d}t \;&= \int_0^T \int_\Omega w \cdot r \,\mathrm{d}x\,\mathrm{d}t. \end{aligned} \tag{24.8}$$

with initial conditions $u(X, 0) = u_0(X)$ and $w(X, 0) = v_0(X)$ in Ω, and boundary conditions $u(X, t) = g(X, t)$ on $\partial\Omega_\mathrm{D}$.

We now assume that the finite element approximation space V_h is CG_1 (continuous and piecewise linear in time), and $\hat{\mathrm{V}}_h$ is DG_0 (discontinuous and piecewise constant in time). With these assumptions, we arrive at the following scheme:

$$\begin{aligned} \int_\Omega \rho \frac{(w_{n+1} - w_n)}{\Delta t} \cdot v \,\mathrm{d}x \;+ \int_\Omega P(u_\mathrm{mid}) : \mathrm{Grad}(v)\,\mathrm{d}x \;&= \int_\Omega B \cdot v \,\mathrm{d}x \;+ \int_{\partial\Omega_\mathrm{N}} T \cdot v \,\mathrm{d}s, \text{ and} \\ \int_\Omega \frac{(u_{n+1} - u_n)}{\Delta t} \cdot r \,\mathrm{d}x \;&= \int_\Omega w_\mathrm{mid} \cdot r \,\mathrm{d}x, \end{aligned} \tag{24.9}$$

where $(\cdot)_n$ and $(\cdot)_{n+1}$ are the values of a quantity at the current and subsequent time-step, respectively, and $(\cdot)_\mathrm{mid} = \frac{(\cdot)_n + (\cdot)_{n+1}}{2}$. A section of the CG_1 linear momentum balance solver class is presented in Figure 24.7. The code closely mirrors the scheme defined in (24.9), and results in a mixed system that is solved for using a Newton scheme.

The CG_1 scheme defined in (24.9) is straightforward to derive and implement, and it is second order accurate and energy conserving.[5] But it should also be noted that the mixed system that results from the

[5]This is demonstrated in Figure 24.14 as part of the second example calculation.

Python code

```
class CG1MomentumBalanceSolver(CBCSolver):

    # Define function spaces
    vector = VectorFunctionSpace(mesh, "CG", 1)
    mixed_element = MixedFunctionSpace([vector,
                                        vector])
    V = TestFunction(mixed_element)
    dU = TrialFunction(mixed_element)
    U = Function(mixed_element)
    U0 = Function(mixed_element)

    # Get initial conditions, boundary conditions
    # and body forces
    ...

    # Functions
    v, r = split(V)
    u, w = split(U)
    u0, w0 = split(U0)

    # Evaluate displacements and velocities at
    # mid points
    u_mid = 0.5*(u0 + u)
    w_mid = 0.5*(w0 + w)

    # Get reference density
    rho = problem.reference_density()

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

    # The variational form corresponding to
    # dynamic hyperelasticity
    L = rho*inner(w - w0, v)*dx \
        + dt*inner(P, grad(v))*dx \
        - dt*inner(B, v)*dx\
        + inner(u - u0, r)*dx \
        - dt*inner(w_mid, r)*dx

    # Add contributions to the form from the
    # Neumann boundary conditions
    ...

    a = derivative(L, U, dU)
```

Figure 24.7: Relevant portion of the dynamic balance of linear momentum balance solver using the CG_1 time-stepping scheme.

formulation is computationally expensive and memory intensive as the number of variables being solved for have doubled.

CBC.Twist also provides a standard implementation of a finite difference time-stepping algorithm that is commonly used in the computational mechanics community: the Hilber–Hughes–Taylor (HHT) method [Hilber et al., 1977]. The stability and dissipative properties of this method in the case of linear problems have been thoroughly discussed in Hughes [1987]. In particular, the method contains three parameters α, β and γ which control the accuracy, stability and numerical dissipation of the scheme. The default values for these parameters chosen in CBC.Twist ($\alpha = 1$, $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$) ensure that the method is second order accurate, stable for linear problems and introduces no numerical dissipation.

The method is briefly sketched below. For further details about the scheme itself, or its implementation in CBC.Twist, the interested reader is directed to the previously mentioned papers, and the MomentumBalanceSolver class in the file solution_algorithms.py.

Given initial conditions $u(X,0) = u_0(X)$ and $\frac{\partial u}{\partial t}(X,0) = v_0(X)$, we can compute the initial acceleration, a_0, from the weak form:

$$\int_\Omega \rho a_0 \cdot v \,\mathrm{d}x + \int_\Omega P(u_0) : \mathrm{Grad}(v) \,\mathrm{d}x - \int_\Omega B(X,0) \cdot v \,\mathrm{d}x - \int_{\partial\Omega_N} T(X,0) \cdot v \,\mathrm{d}s = 0. \tag{24.10}$$

This provides the complete initial state (u_0, v_0, a_0) of the body. Now, given the solution at time step n, the HHT formulae advance the solution to step $n+1$ as follows. First, we note the following definitions:

$$\begin{aligned} u_{n+1} &= u_n + \Delta t v_n + \Delta t^2 \left[\left(\frac{1}{2} - \beta \right) a_n + \beta a_{n+1} \right] \\ v_{n+1} &= v_n + \Delta t \left[(1-\gamma) a_n + \gamma a_{n+1} \right] \\ u_{n+\alpha} &= (1-\alpha) u_n + \alpha u_{n+1} \\ v_{n+\alpha} &= (1-\alpha) v_n + \alpha v_{n+1} \\ t_{n+\alpha} &= (1-\alpha) t_n + \alpha t_{n+1} \end{aligned} \tag{24.11}$$

Inserting the definitions in (24.11) into the following form of the balance of linear momentum,

$$\int_\Omega \rho a_{n+1} \cdot v \,\mathrm{d}x + \int_\Omega P(u_{n+\alpha}) : \mathrm{Grad}(v) \,\mathrm{d}x - \int_\Omega B(X, t_{n+\alpha}) \cdot v \,\mathrm{d}x - \int_{\partial\Omega_N} T(X, t_{n+\alpha}) \cdot v \,\mathrm{d}s = 0, \tag{24.12}$$

we can solve for the for the only unknown variable, the acceleration at the next step, a_{n+1}. The acceleration solution to (24.12) is then used in the definitions (24.11) to update to new displacement and velocity values, and the problem is stepped through time.

We close this subsection on time-stepping algorithms with one usage detail pertaining to CBC.Twist. By default, when solving a dynamics problem, CBC.Twist assumes that the user wants to use the HHT method. In case one wants to override this behavior, they can do so by returning "CG(1)" in the time_stepping method while specifying the problem. Figure 24.12 is an example showing this.

24.3 *Examples of CBC.Twist usage*

The algorithms discussed thus far serve primarily to explain the computational framework's inner working, and are not at the level at which the user usually interacts with CBC.Twist (unless they are interested in extending it). In practice, the functionality of CBC.Twist is exposed to the user through two primary problem definition classes: StaticHyperelasticity and Hyperelasticity. These classes reside in problem_definitions.py, and contain numerous methods for defining aspects of the nonlinear elasticity problem. As their names suggest, these are respectively used to describe static or dynamic problems in nonlinear elasticity.

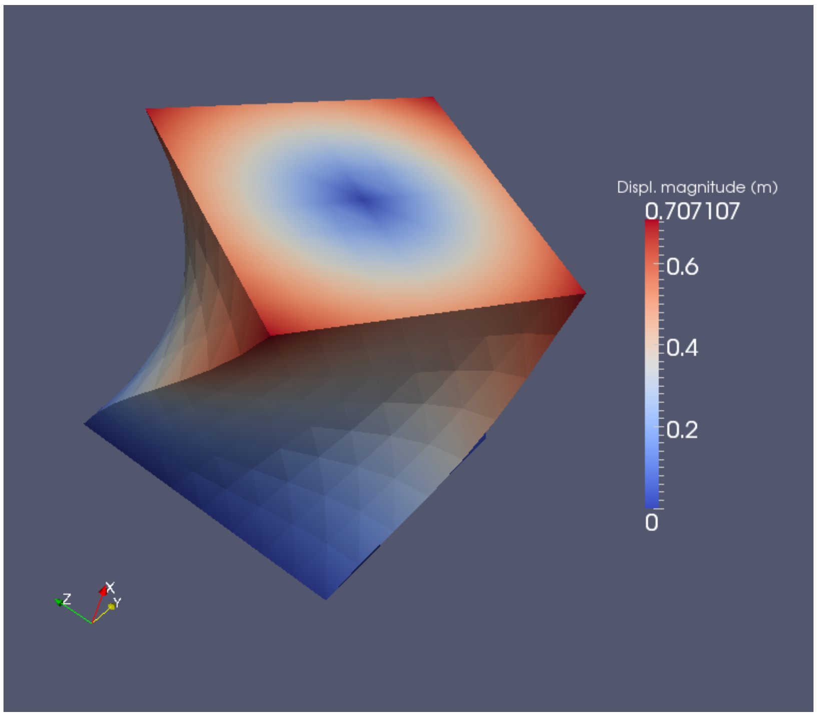


Figure 24.8: A hyperelastic cube twisted by 60 degrees.

Over the course of the following examples, we will see how various problems can be defined in CBC.Twist by suitably deriving from these problem classes and overloading relevant methods.[6] We will also see some results from these calculations. The information defined in the problem classes are internally transferred to the solvers described earlier to actually solve the problem.

The static twisting of a hyperelastic cube

The first problem we are interested in is the twisting of a unit hyperelastic cube (1 m^3). The cube is assumed to be made out of a St. Venant–Kirchhoff material with Lamé's parameters $\mu = 3.8461$ N/m^2 and a spatially varying $\lambda = 5.8x_1 + 5.7(1 - x_1)$ N/m^2. Here, x_1 is the first coordinate of the reference position, X.[7] In order to twist the cube, the face $x_1 = 0$ is held fixed and the opposite face $x_1 = 1$ is rotated 60 degrees using the Dirichlet condition defined in Figure 24.10.

Before getting to the actual specification of the problem in code, we need to import CBC.Twist's functionality.

Python code

```
from cbc.twist import *
```

Figure 24.9: CBC.Twist first needs to be imported to access the functionality that it offers.

The problem is completely specified by defining relevant methods in the user-created class Twist (see Figure 24.10), which derives from the base class StaticHyperelasticity. CBC.Twist only requires relevant methods to be provided, and for the current problem, this includes the computational domain, Dirichlet boundary conditions and material model. The methods are fairly self-explanatory, but the following points are to be noted. Firstly, CBC.Twist supports spatially-varying material parameters. Secondly, Dirichlet

[6]The examples presented in this chapter, along with a few others, reside in the demos/twist/ folder in CBC.Twist's source repository. They can be run by navigating to this folder and typing python demo_name.py on the command-line.

[7]The numerical parameters in this chapter have been arbitrarily chosen for illustration of the framework's use. They do not necessarily correspond to a real material.

Figure 24.10: Problem definition: The static twisting of a hyperelastic cube.

Python code

```python
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):
        mu    = 3.8461
        lmbda = Expression("x[0]*5.8+(1-x[0])*5.7")

        material = StVenantKirchhoff([mu, lmbda])
        return material

    def __str__(self):
        return "A cube twisted by 60 degrees"
```

Figure 24.11: Solving the posed problem.

Python code

```python
twist = Twist()
u = twist.solve()
```

boundary conditions are posed in two parts: the conditions themselves, and the corresponding boundaries along which they act.

In order to solve this problem, an instance of the `Twist` class is created and its `solve` method is called (see Figure 24.11). This triggers a Newton solve which exhibits quadratic convergence (see Table 24.2) and results in the displacement field shown in Figure 24.8.

The dynamic release of a twisted cube

In this problem, we release a unit cube (1 m^3) that has previously been twisted. The initial twist was precomputed in a separate calculation involving a traction force on the top surface and the resulting displacement field was stored in the file `twisty.txt`. The release calculation loads this solution as the initial displacement. It fixes the cube (made of a St. Venant–Kirchhoff material with Lamé's parameters $\mu = 3.8461$ N/m^2 and $\lambda = 5.76$ N/m^2) on the bottom surface, and tracks the motion of the cube over 2 s.

The problem is specified in the user-created class `Release`, which derives from `Hyperelasticity`. This example is similar to the previous one, except that since it is a dynamic calculation, it also provides initial conditions, a reference density and information about time-stepping. Again, the methods listed in Figure 24.12 are straightforward, and the only additional point to note is that `CBC.Twist` provides some convenience utilities to simplify the specification of the problem. For example, one can load initial conditions directly from files, and it allows for the specification of boundaries purely as conditional strings.

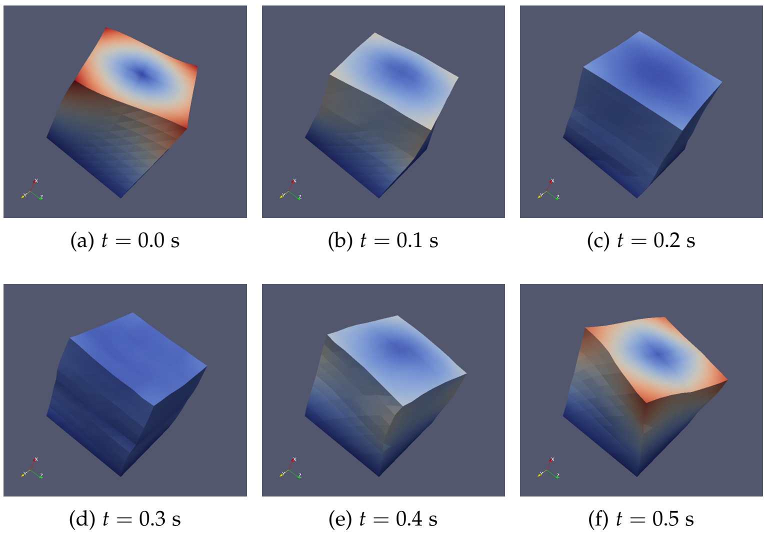
Iteration	Relative Residual Norm
1	5.835e-01
2	1.535e-01
3	3.640e-02
4	1.004e-02
5	1.117e-03
6	1.996e-05
7	9.935e-09
8	3.844e-15

Table 24.2: Quadratic convergence of the Newton method used to solve the hyperelasticity problem. It is interesting to note that this convergence is obtained even though the 60 degree twist condition was imposed in a single step.

Figure 24.12: Problem definition: The dynamic release of a twisted cube.

Python code

```python
class Release(Hyperelasticity):

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

    def end_time(self):
        return 2.0

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

    def time_stepping(self):
        return "CG(1)"

    def reference_density(self):
        return 1.0

    def initial_conditions(self):
        u0 = "twisty.txt"
        v0 = Expression(("0.0", "0.0", "0.0"))
        return u0, v0

    def dirichlet_values(self):
        return [(0, 0, 0)]

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

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

    def __str__(self):
        return "A pretwisted cube being released"
```



(a) $t = 0.0$ s (b) $t = 0.1$ s (c) $t = 0.2$ s

(d) $t = 0.3$ s (e) $t = 0.4$ s (f) $t = 0.5$ s

Figure 24.13: Relaxation and subsequent re-twisting of a released cube over the first 0.5 s of the calculation.

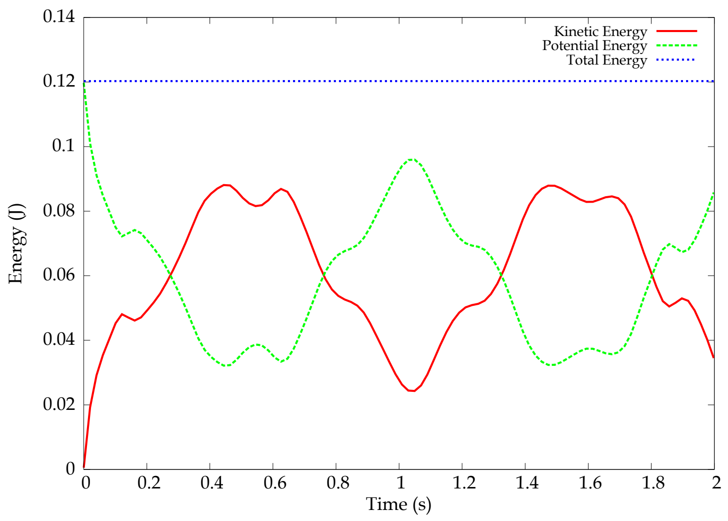


Figure 24.14: Over the course of the computation, the energy in the body is converted between potential and kinetic energy, but the total remains constant.

When `Release` is instantiated and its `solve` method is called, we see the relaxation of the pre-twisted cube. After initial unwinding of the twist, the body proceeds to twist in the opposite direction due to inertia. This process repeats itself, and snapshots of the displacement over the first 0.5 s are shown in Figure 24.13. Figure 24.14 highlights the energy conservation of the CG_1 numerical scheme used to time-step this problem by totaling the kinetic energy and potential energy of the body over the course of the calculation. `CBC.Twist` provides this information through the methods `kinetic_energy(v)` and `potential_energy(u)`, where `v` and `u` are velocity and displacement fields respectively.

A hyperelastic dolphin tumbling through a "flow"

In this final example, we aim to crudely simulate the motion of a dolphin under a flow field. The dolphin is assumed to be made out of a Mooney–Rivlin material ($c_1 = 6.169$ N/m^2, $c_2 = 10.15$ N/m^2), and the flow field is simply modeled by a uniform traction force $T = (0.05, 0)$ N acting everywhere on the surface of the dolphin, pushing it to the right.

This example is constructed to exhibit some additional features of `CBC.Twist`. For one, `CBC.Twist` is

Python code

```
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
```

Figure 24.15: Problem definition: A hyperelastic dolphin being pushed to the right.

Python code

```
problem = FishyFlow()

dt = problem.time_step()
T = problem.end_time()

t = dt
while t <= T:
    problem.step(dt)
    problem.update()
    t = t + dt
```

Figure 24.16: Stepping through time in an external time loop. `step` steps the problem forward by one time step, and `update` updates the values of all time-dependent variables to the current time.

capable of performing dynamic calculations under entirely Neumann boundary conditions. In addition, this calculation points out that the framework can seamlessly handle problems in two dimensions as well.

The problem is specified in the user-created class `FishyFlow` derived from `Hyperelasticity`. There is nothing new to note in the code listing for this problem (Figure 24.15), other than the fact that we now specify Neumann boundary conditions. The specification listing is not very long because `CBC.Twist` assumes meaningful default values for unspecified information.

To demonstrate one final piece of functionality of `CBC.Twist`, we don't solve the problem in the same manner as we did the first two examples; that is, we do not instantiate an object of class `FishyFlow` and call its `solve` method. Instead, we set up our own time loop and manually step through time using the `step` method. This is shown in Figure 24.16.

The advantage of solving the problem in this manner is that, now, one has more control over calculations in `CBC.Twist`. For example, rather than just fixing a traction force on the surface of the dolphin to mimic the effect of flow field, one can instead solve at each time step an actual flow field and use it to correctly drive the solid mechanics. This functionality of `CBC.Twist` is used in a following chapter on adaptive methods for fluid-structure interaction (**??**). In that work, the fluid-structure problem is solved using a staggered approach with the solid mechanics equation being solved by `CBC.Twist`. An external time loop

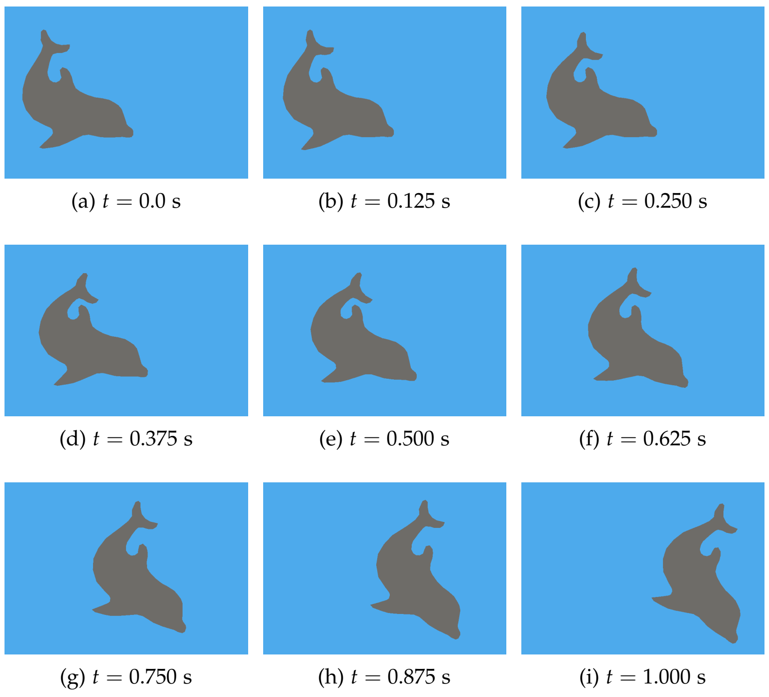

(a) $t = 0.0$ s (b) $t = 0.125$ s (c) $t = 0.250$ s (d) $t = 0.375$ s (e) $t = 0.500$ s (f) $t = 0.625$ s (g) $t = 0.750$ s (h) $t = 0.875$ s (i) $t = 1.000$ s

Figure 24.17: The motion of a hyperelastic dolphin being forced to the right. Careful observation of the tail fin shows deformation of the dolphin in addition to its overall motion toward the right.

similar to the one in Figure 24.16 is set up to individually step through the fluid problem, the solid problem and a mesh equation; a process which is iterated until convergence is reached at each time step. This process involves the systematic transfer of relevant information (such as fluid loading) from other problems to `CBC.Twist`.

But returning to our current example, Figure 24.17 shows time snapshots of the motion of the dolphin over the course of the computation. Notice that the fish deforms elastically as it tumbles toward the right.

24.4 Conclusion

This chapter presented an overview of `CBC.Twist`, an automated computational framework for nonlinear elasticity. Beginning with elements of classical nonlinear elasticity theory to motivate its design, the discourse took a closer look at the algorithms underlying `CBC.Twist`'s implementation. The chapter concluded with some examples, offering a tutorial-like description of how the framework can be used in practice to solve problems.

The discussion aimed to highlight a central feature of `CBC.Twist`: the ease with which different material models can be defined and used. This feature makes `CBC.Twist` immediately applicable to a number of real-world problems in engineering, especially those pertaining to polymer and biological tissue mechanics.

`CBC.Twist` is a collaboratively developed open source project (released under the GNU GPL) that is freely available from its source repository at `https://launchpad.net/cbc.solve/`. Its only dependency is a working FEniCS installation. `CBC.Twist` is released with the goal that it will allow users to easily solve problems in nonlinear elasticity as part of answering specific questions through computational modeling. Everyone is encouraged to fetch and try it. Users are also encouraged to modify the code to better suit their own purposes, and contribute changes that they think are useful to the community. Along these lines, some possible ideas for extending the framework include:

- Implementing other specific material models
- Allowing for bodies composed of multiple materials
- Support for (nearly) incompressible materials

- Support for anisotropic materials
- Support for viscoelastic materials
- Goal-oriented adaptivity

Contributions toward these (or other useful) extensions are welcome.