

*Version 2.0*

**Developer’s Manual**

**Last Updated: July 8, 2013**

**Contact address**

Musculoskeletal Research Laboratories, University of Utah

72 S. Central Campus Drive, Room 2646

Salt Lake City, Utah

**Website**

**MRL:** [http://mrl.sci.utah.edu](http://mrl.sci.utah.edu/)

**FEBio:** <http://www.febio.org>

**Forum:** <http://mrl.sci.utah.edu/forums/>

Development of the FEBio project is supported in part by a grant from the U.S. National Institutes of Health.



**Table of Contents**

[1 Compiling FEBio 4](#_Toc361045149)

[1.1 Compiling for Windows using Visual Studio 2008 4](#_Toc361045150)

[1.2 Compiling for Windows using Visual Studio 2010 4](#_Toc361045151)

[1.3 Compiling for Linux 4](#_Toc361045152)

[2 The FEBio2 framework 5](#_Toc361045153)

[2.1 Introduction 5](#_Toc361045154)

[3 Adding new materials 6](#_Toc361045155)

[3.1 Basic Procedure 6](#_Toc361045156)

[3.1.1 Defining the new material class 6](#_Toc361045157)

[3.1.2 Registering the new material 7](#_Toc361045158)

[3.1.3 Defining the material parameters 8](#_Toc361045159)

[3.1.4 Implementing the initialization function 9](#_Toc361045160)

[3.1.5 Implementing the stress and tangent functions 9](#_Toc361045161)

[3.2 Additional Material Classes 12](#_Toc361045162)

[3.2.1 The FEHydraulicPermeability Class 12](#_Toc361045163)

[3.2.2 The FEOsmoticCoefficientClass 13](#_Toc361045164)

[3.2.3 The FESoluteDiffusivity class 14](#_Toc361045165)

[3.2.4 The FESoluteSolubility class 15](#_Toc361045166)

[3.3 The FEElasticMaterialPoint class 16](#_Toc361045167)

[3.4 Using the new material class 16](#_Toc361045168)

[3.5 Debugging the Material Implementation 17](#_Toc361045169)

[3.6 Advanced Topics 19](#_Toc361045170)

[3.6.1 Array parameters 19](#_Toc361045171)

[3.6.2 Uncoupled Materials 19](#_Toc361045172)

[3.6.3 User-defined material points 21](#_Toc361045173)

[3.6.4 The FEBiphasicMaterialPoint Class 22](#_Toc361045174)

[3.6.5 The FESolutesMaterialPoint class 23](#_Toc361045175)

[4 Tensors in FEBio 25](#_Toc361045176)

[4.1 First-order tensors 25](#_Toc361045177)

[4.2 Second-order tensors 26](#_Toc361045178)

[4.2.1 mat3d class 26](#_Toc361045179)

[4.2.2 mat3ds class 27](#_Toc361045180)

[4.2.3 mat3da class 27](#_Toc361045181)

[4.2.4 mat3dd class 28](#_Toc361045182)

[4.3 Fourth-order tensors 28](#_Toc361045183)

[4.4 Implicit Type conversions 29](#_Toc361045184)

[4.5 Dyadic products 30](#_Toc361045185)

[5 FEBio2 Road Map 31](#_Toc361045186)

[5.1 File Format 31](#_Toc361045187)

[5.1.1 Proposed changed 32](#_Toc361045188)

[5.2 Material Parameters 33](#_Toc361045189)

[5.2.1 A new material interface 33](#_Toc361045190)

[5.2.2 Position dependent parameters 34](#_Toc361045191)

# Compiling FEBio

## Compiling for Windows using Visual Studio 2008

The easiest way to compile FEBio on Windows is using Visual Studio 2008 (VS). The source code directory contains a file titled *febio.sln*. Opening this file in VS opens the FEBio solution which contains the FEBio and FECore project files.

Before you can build FEBio, a configuration needs to be selected. The FEBio solution has several configurations to choose from, but if you do not have additional libraries for sparse linear solvers (e.g. Pardiso, SuperLU, etc) installed on your system the only configurations that you will be able to build successfully are the “Debug – skyline only” and “Release – skyline only”. The first configuration, as the name suggests, is for debugging purposes. It is recommended to use this configuration when writing and testing new code. The release configuration uses several optimization flags to produce increased performance.

Once a configuration is selected, building FEBio is as simple as pressing F7 or by selection the Build/Build Solution menu. First, the FECore library will be built, which is then used to build FEBio.

## Compiling for Windows using Visual Studio 2010

A solution file for Visual Studio 2010 is also provided. It can be found in the FEBio/VS2010 folder. The solution contains the code for FECore and FEBio. The solution can be built from the Build/Build Solution menu. Make sure to first select the appropriate configuration before building.

## Compiling for Linux

For Linux platforms a makefile is provided to facilitate the compilation and linking of FEBio. On Linux platforms the FECore library has to be built separately and before FEBio is built.

# The FEBio2 framework

In this chapter we will take a look at the structure of the FEBio2 framework. Although most of this material is not necessary to understand when creating FEBio2 plugins, it may be helpful in understanding the different classes, how FEBio2 works internally and how to extend the framework. We will first take a bird’s eye view of FEBio2’s class hierarchy and then dive into some of the detailed workings of the most important classes.

## Introduction

FEBio2 is organized into different modules, each module implemented in a separate library. The two most important of these libraries are *FECore* and *FEBioLib*. The first library defines all the core classes that FEBio uses. The latter contains specialized implementations of the core classes, each class implementing a specific feature in FEBio. For instance, the *FEMaterial* class is defined in *FECore* and is used to define the constitutive behavior of a domain. This core class does not define any explicit material behavior. Instead, different material classes are derived from this class in *FEBioLib*, each class defining a specific material behavior.

Other libraries define the different components of FEBio. The following list provides an overview of all the libraries.

|  |  |
| --- | --- |
| **Library** | **Description** |
| *FECore* | Core classes that require specialization. |
| *FEBioLib* | Specialized classes that implement most of FEBio’s features. |
| *FEBioPlot* | Implements support for the xplt database format. |
| *FEBioXML* | Implements a parser for FEBio’s xml-based input format. |
| *NumCore* | Implements numerical algorithms used by FEBio. Mostly hooks to external libraries for solving spare systems of linear equations. |

These libraries are the essential components to build the FEBioLib library. However, the library doesn’t provide a user interface. Instead, the library has to be wrapped inside a program that offers a user interface. Two such programs are provided: a command line interface (FEBio2.exe) and a GUI based application, namely the FEBio Task Manager (FEBioTM.exe).

An advantage of providing FEBio as a library is that developers can integrate FEBio more easily with other existing codes or create a custom UI for FEBio. This offers the possibility of creating specialized code that solves specific problems using FEBio as the underlying FE engine.

# Adding new materials

FEBio allows the user to create new material implementations in a simple and straightforward manner with a minimum of modifications to the code. The new material will be integrated seamlessly in FEBio’s framework so that the user can take immediate advantage of additional functionality such as reading material parameters from the xml-formatted FEBio input file, serialization to the restart archive, parameter optimization and more.

The basic procedure for creating a new material for FEBio requires the following steps.

1. Defining a new material class by deriving it from an available base class.
2. Registering the material with FEBio’s framework.
3. Defining the material parameters.
4. Implementing the initialization function (optional)
5. Defining a MaterialPoint structure (optional)
6. Implementing the stress and tangent functions.

Next, the steps for the basic procedure will be discussed in more detail.

## Basic Procedure

### Defining the new material class

FEBio is written in C++ and therefore the new material implementation must be coded in C++ as well. Each material requires a separate class and hence the first step is to define a new class. The class has to be derived from one of the available material base classes. In this section it is assumed that the new material is derived from *FEElasticMaterial*. Materials that are derived from this base class will be materials that are used to describe isotropic, compressible solid materials. An example of such a material is the neo-Hookean material, which will be used as a case study in this section. The implementation of more advanced models will be discussed later, but they too have to follow most of the same steps.

The definition of the neo-Hookean material class looks as follows. Note that the actual implementation of this class might look a bit different in the code. Only the important aspects are touched upon here.

1. class FENeoHookean : public FEElasticMaterial
2. {
3. public:
4. // material parameters
5. double m\_E;
6. double m\_v;
7. public:
8. // Cauchy-stress calculation
9. virtual mat3ds Stress(FEMaterialPoint& pt);
10. // Spatial elasticity tensor calculation
11. virtual tens4ds Tangent(FEMaterialPoint& pt);
12. // class initialization (optional)
13. virtual void Init();
14. // declare as a registered class
15. DECLARE\_REGISTERED(FENeoHookean);
16. // declare as having a parameter list
17. DECLARE\_PARAMETER\_LIST();
18. };

The class derives, as expected, publicly from *FEElasticMaterial*. It then defines a couple of public variables. These variables will store the material parameters as is discussed below. After this, the class also declares a few member functions. Note that all these functions are virtual. Furthermore, the *Stress* and *Tangent* are declared as abstract in the base class, so they have to be overridden in the derived class. The *Init* function is optional, since a default implementation (which does nothing) is provided.

At this point it is useful to discuss a common practice in adding new classes in C++. Usually the definition of the class is split over two separate files. One file, the so-called header file, declares the new class. The implementation of the function members are placed in a separate file, usually a .cpp file. FEBio follows this practice and therefore the neo-Hookean implementation is spread over two files: the *FENeoHookean.h* contains the class definition and *FENeoHookean.cpp* contains the member function definitions. It is recommended that the implementation of new materials follows this practice.

The last two lines of the class definition contain macros that will help with the registration procedure of the class and its material parameters.

### Registering the new material

In order for FEBio to recognize the new material, the material needs to be registered with the framework. FEBio defines two macros that help with this process. The first one is added to the class definition.

DECLARE\_REGISTERED(FENeoHookean);

This macro takes one parameter, namely the name of the new material class, and informs that the class will be registered with the framework. The second one, which actually registers the class with the framework, is placed in the compilation unit (e.g. the .cpp file). In *FENeoHookean.cpp* it is the first line of the file after the include statements.

REGISTER\_MATERIAL(FENeoHookean, "neo-Hookean");

This macro takes two parameters: the class that needs to be registered and a text parameter that defines the name of the class. The name of the class is used in several places to identify the new material and must be unique (that is, not two materials can have the same name). This name will be used, for instance, as the value of the *type* parameter in the xml-formatted input file. This aspect will be explained further when the use of the new material is discussed.

### Defining the material parameters

Defining the material parameters of the class requires two steps. First, variables need to be defined that will store the values for these parameters. In our example, the *FENeoHookean* class defines two parameters: *m\_E* stores the Young’s modulus and *m\_v* stores the Poisson’s ratio. Note that these variables are defined as *public*. This is important, since only public variables can be used by FEBio’s framework.

The second step is to register the material parameters with FEBio’s framework. Again, a set of macros exist that will facilitate this process. The first one is placed in the class definition.

DECLARE\_PARAMETER\_LIST();

This macro informs FEBio that this material class will be defining a set of material parameters. The actual definition of the material parameter list is placed in the compilation unit (i.e. the .cpp file). In our example, this list is found at the top of the *FENeoHookean.cpp* file.

BEGIN\_PARAMETER\_LIST(FENeoHookean, FEElasticMaterial);

ADD\_PARAMETER(m\_E, FE\_PARAM\_DOUBLE, "E");

ADD\_PARAMETER(m\_v, FE\_PARAM\_DOUBLE, "v");

END\_PARAMETER\_LIST();

The parameter list definition begins with the *BEGIN\_PARAMETER\_LIST* macro, which takes two parameters: the name of the class, and the name of the base class. Next, for each parameter, the *ADD\_PARAMETER* macro can be used to define it. This macro takes three parameters: the variable that will store the parameter’s value, a type identifier and a string name for the variable. The type of the type identifier can be any of the following values.

* *FE\_PARAM\_INT*: defines a parameter of type *int*.
* *FE\_PARAM\_BOOL*: defines a parameter of type *bool*.
* *FE\_PARAM\_DOUBLE*: defines a parameter of type *double*.

Note that the type identifier *must* match the type of the actual variable. For example, a *double* variable must be defined with the *FE\_PARAM\_DOUBLE* type identifier. If the types do not match, the resulting behavior is undefined.

Finally, the parameter list must be ended with the *END\_PARAMETER\_LIST* macro. Note that there a couple of more advanced options to define material parameters, such as vector parameters and load-curve controlled parameters. These will be discussed in the advanced section below.

### Implementing the initialization function

When implementing a new material class the user has the option to override the base class implementation of the *Init* function. This function is called after the material parameters have been read from the input file and can be used to check the values of these parameters. In our example, this function is declared as follows.

1. void FENeoHookean::Init()
2. {
3. FEElasticMaterial::Init();
4. if (m\_E <= 0) throw MaterialError("Invalid value for E");
5. if (!INRANGE(m\_v, -1, 0.5)) throw MaterialError("Invalid ...
6. }

First note line 3 where the base class version of this function is called. It is good practice to always call this function so that the base class has a chance to check itself. Even though the base class implementation may do nothing, it is best not to assume this and always call the base class version.

The next two lines check the values of the parameters as they have been read from the user input file. If an invalid parameter value is encountered the user can report a problem by throwing the *MaterialError* exception. This exception accepts a variable-argument list, similarly to the *printf* class of functions. Throwing this error will cause the run to be aborted with a fatal error. The error message will be printed on the screen as well as to the log file.

Again, it is noted that this function is optional, but it is recommended to implement it so that potential problems due to invalid material parameters can be caught quickly.

### Implementing the stress and tangent functions

Next follows the most important aspect of the implementation: the declaration of the stress and tangent functions. These functions will describe the physical reaction of this material to an applied deformation. Note that FEBio works in the spatial frame. This implies that the stress function needs to return the *Cauchy* stress and the tangent function needs to return the *spatial elasticity tensor*.

The stress function is defined as follows in the material class definition (i.e. the header file).

mat3ds stress(FEMaterialPoint& pt);

This function takes one parameter of type *FEMaterialPoint*. This parameter stores all the important information about the point at which to calculate the stress value. For example, this variable stores the reference and current location of the point, the local deformation gradient, history variables (if defined) and much more. It also defines a bunch of useful functions that can facilitate the implementation of the stress function, such as a function that calculates the left and right Cauchy-Green tensors. There is a lot to say about this class, but in order not to digress, a detailed explanation of this class is postponed and only a few important aspects of it are mentioned here.

The actual definition of the stress function is, as usual, placed in the compilation unit. In our example, this is the *FENeoHookean.cpp* file.

1. mat3ds FENeoHookean::Stress(FEMaterialPoint& mp)
2. {
3. FEElasticMaterialPoint& pt = mp.ExtractData<FEElasticMaterialPoint>();
4. mat3d &F = pt.F;
5. double detF = pt.J;
6. double detFi = 1.0/detF;
7. double lndetF = log(detF);
8. // calculate left Cauchy-Green tensor
9. // (we commented out the matrix components we do not need)
10. mat3ds b = pt.LeftCauchyGreen();
11. // lame parameters
12. double lam = m\_v\*m\_E/((1+m\_v)\*(1-2\*m\_v));
13. double mu = 0.5\*m\_E/(1+m\_v);
14. // Identity
15. mat3dd I(1);
16. // calculate stress
17. mat3ds s = (b - I)\*(mu\*detFi) + I\*(lam\*lndetF\*detFi);
18. return s;
19. }

Although the detailed implementation of this constitutive model will not be explained, a few important points are noted.

On line 3 a perhaps strange construction appears. As mentioned before, the material point stores all the information about the current point at which the stress is required. This class however, stores its data per material type. In this example only elastic materials are mentioned (that is, materials derived from the *FEElasticMaterial* class), but there are other types of materials as will be discussed in the advanced section below. Each material type can define different attributes that need to be stored in the material point class. In order to access the data that corresponds to a particular material class, the user can use the *ExtractData* member function of the material point class. The class returns the subset of data that are relevant for this class of material. In this case the returned data is of the type *FEElasticMaterialPoint*.

On line 5, the local deformation gradient is accessed from the material point data and on line 6, the local Jacobian (which is the determinant of the deformation gradient[[1]](#footnote-1)). These are data members that can be accessed directly. Line 12 illustrates how to obtain additional information using the material point’s member functions. In this case, the left Cauchy-Green tensor is retrieved using the *LeftCauchyGreen* member function. A more detailed description of the available data and function members can be found in the advanced section.

Line 22 shows an example of how the actual stress can be computed. FEBio defines a whole bunch of classes that facilitate the use of tensors. For example, the *mat3ds* class implements a second-order 3D symmetric tensor of doubles (the *d* stands for *double*). The *mat3dd* class implements a second-order 3D diagonal tensor. We will see some examples of fourth-order tensor classes in the tangent function.

Line 24 returns the calculated stress value at the current material point. Note that the variable returned is of type *mat3ds*, that is, a symmetric second-order tensor.

The tangent function is declared in the class definition as well.

tens4ds Tangent(FEMaterialPoint& pt);

This function too takes a single *FEMaterialPoint* variable as input. Note that in this case the return value is of type *tens4ds* which is a class that implements a fourth-order tensor with major and minor symmetries. The definition of the function can be found in the *FENeoHookean.cpp* file.

1. tens4ds FENeoHookean::Tangent(FEMaterialPoint& mp)
2. {
3. FEElasticMaterialPoint& pt = \*mp.ExtractData<FEElasticMaterialPoint>();
4. // deformation gradient
5. mat3d &F = pt.F;
6. double detF = pt.J;
7. // lame parameters
8. double lam = m\_v\*m\_E/((1+m\_v)\*(1-2\*m\_v));
9. double mu = 0.5\*m\_E/(1+m\_v);
10. double lam1 = lam / detF;
11. double mu1 = (mu - lam\*log(detF)) / detF;
12. mat3dd I(1);
13. tens4ds IxI = dyad1s(I);
14. tens4ds I4 = dyad4s(I);
15. return IxI\*lam1 + I4\*(2\*mu1);
16. }

In line 3 the data of the material data that pertains to elastic materials is extracted. The next few lines extract some data from the *FEElasticMaterialPoint* variable and calculate some other parameters.

Lines 16 and following calculate the tangent stiffness. Note the use of the fourth-order tensor class *tens4ds*. This code snippet also illustrates the use of the dyadic products to create fourth-order tensors from second-order tensors. A more detailed explanation of the use of the tensor classes can be found in chapter 3.

## Additional Material Classes

In addition to the material classes defining solid materials, there are several classes defining materials relevant to biphasic and multiphasic mixtures. These are briefly described here.

### The FEHydraulicPermeability Class

This virtual class defines the hydraulic permeability, a material function that relates the interstitial fluid flux to the gradient in fluid pressure. This class provides the following virtual functions that must be defined in any permeability material derived from this class:

* The *Permeability* function: Returns the hydraulic permeability symmetric second-order tensor.
* The *Tangent\_Permeability\_Strain* function: Returns the tangent of the permeability with respect to the strain in the spatial frame. It is a fourth-order tensor.
* The *Tangent\_Permeability\_Concentration* function: Returns the tangent of the permeability with respect to the effective concentration of a solute in the mixture. It is a symmetric second-order tensor.
* The *Init* function: Performs any necessary initialization.

An example of this type of material is the Holmes-Mow permeability:

class FEPermHolmesMow : public FEHydraulicPermeability

{

public:

//! constructor

FEPermHolmesMow();

//! permeability

mat3ds Permeability(FEMaterialPoint& pt);

//! Tangent of permeability

tens4ds Tangent\_Permeability\_Strain(FEMaterialPoint& mp);

//! data initialization and checking

void Init();

public:

double m\_perm; //!< permeability

double m\_M; //!< nonlinear exponential coefficient

double m\_alpha; //!< nonlinear power exponent

// declare as registered

DECLARE\_REGISTERED(FEPermHolmesMow);

// declare parameter list

DECLARE\_PARAMETER\_LIST();

};

The format of this material definition is very similar to that of an elastic solid material as detailed in Section 2.1.1. Material constants associated with this constitutive model are declared here, such as *m\_perm*, *m\_M*, and *m\_alpha* in this example. The material must be registered and its parameter list must be declared. In the corresponding .cpp file, the following lines of code appear near the top of the file:

// register the material with the framework

REGISTER\_MATERIAL(FEPermHolmesMow, "perm-Holmes-Mow");

// define the material parameters

BEGIN\_PARAMETER\_LIST(FEPermHolmesMow, FEHydraulicPermeability)

ADD\_PARAMETER(m\_perm, FE\_PARAM\_DOUBLE, "perm");

ADD\_PARAMETER(m\_M, FE\_PARAM\_DOUBLE, "M");

ADD\_PARAMETER(m\_alpha, FE\_PARAM\_DOUBLE, "alpha");

END\_PARAMETER\_LIST();

### The FEOsmoticCoefficientClass

This material class is needed for mixtures that contain solutes. The osmotic coefficient is needed for the evaluation of the actual fluid pressure. Constitutive relations for the osmotic coefficient must be derived from this virtual class which is defined as follows:

//---------------------------------------------------------------------

//! Base class for osmotic coefficient.

//!

class FEOsmoticCoefficient : public FEMaterial

{

public:

//! osmotic coefficient

virtual double OsmoticCoefficient(FEMaterialPoint& pt) = 0;

//! tangent of osmotic coefficient with respect to strain

virtual double Tangent\_OsmoticCoefficient\_Strain

(FEMaterialPoint& mp) = 0;

//! tangent of osmotic coefficient w.r.t. concentration

virtual double Tangent\_OsmoticCoefficient\_Concentration

(FEMaterialPoint& mp, const int isol) = 0;

};

This class provides the following virtual functions that must be defined in any osmotic coefficient material derived from this class:

* The *OsmoticCoefficient* function: Returns the osmotic coefficient.
* The *Tangent\_OsmoticCoefficient\_Strain* function: Returns the tangent of the osmotic coefficient with respect to the strain in the spatial frame. In this version, the dependence on the strain is only via *J*=det**F**.
* The *Tangent\_OsmoticCoefficient\_Concentration* function: Returns the tangent of the osmotic coefficient with respect to the effective concentration of a solute in the mixture.

### The FESoluteDiffusivity class

The *FESoluteDiffusivity* class defines the diffusivity of a solute in a multiphasic mixture. The virtual class definition is as follows:

//---------------------------------------------------------------------

//! Base class for solute diffusivity.

//!

class FESoluteDiffusivity : public FEMaterial

{

public:

//! solute diffusivity

virtual mat3ds Diffusivity(FEMaterialPoint& pt) = 0;

//! tangent of diffusivity with respect to strain

virtual tens4ds Tangent\_Diffusivity\_Strain

(FEMaterialPoint& mp) = 0;

//! tangent of diffusivity w.r.t. solute concentration

virtual mat3ds Tangent\_Diffusivity\_Concentration

(FEMaterialPoint& mp, const int isol) = 0;

//! solute diffusivity in free solution

virtual double Free\_Diffusivity(FEMaterialPoint& pt) = 0;

//! tangent of free diffusivity w.r.t solute concentration

virtual double Tangent\_Free\_Diffusivity\_Concentration

(FEMaterialPoint& pt, const int isol) = 0;

//! set solute ID

void SetSoluteID(const int ID) {m\_ID = ID;}

private:

int m\_ID; //!< solute ID

};

This class provides the following virtual functions that must be defined in any osmotic coefficient material derived from this class:

* The *Diffusivity* function: Returns the diffusivity of the solute in the mixture, which is a second-order symmetric tensor.
* The *Tangent\_Diffusivity\_Strain* function: Returns the tangent of the diffusivity with respect to the strain in the spatial frame, which is a fourth-order symmetric tensor.
* The *Tangent\_Diffusivity\_Concentration* function: Returns the tangent of the diffusivity with respect to the effective concentration of a solute in the mixture, which is a symmetric second-order tensor.
* The *FreeDiffusivity* function: Returns the free diffusivity of the solute in solution, which is a scalar.
* The *Tangent\_FreeDiffusivity\_Concentration* function: Returns the tangent of the free diffusivity with respect to the effective concentration of a solute in free solution, which is a scalar.

Since the diffusivity is associated with a specific solute, this class also stores the solute identification number as a private class variable. A user-defined diffusivity constitutive relation must be derived from this parent class.

### The FESoluteSolubility class

The *FESoluteSolubility* class defines the solubility of a solute in a multiphasic mixture. The virtual class definition is as follows:

//---------------------------------------------------------------------

//! Base class for solute solubility.

//!

class FESoluteSolubility : public FEMaterial

{

public:

//! solute solubility

virtual double Solubility(FEMaterialPoint& pt) = 0;

//! tangent of solubility w.r.t. strain

virtual double Tangent\_Solubility\_Strain

(FEMaterialPoint& mp) = 0;

//! tangent of solubility w.r.t. concentration

virtual double Tangent\_Solubility\_Concentration

(FEMaterialPoint& mp, const int isol) = 0;

//! cross derivative of solubility w.r.t. strain & concentration

virtual double Tangent\_Solubility\_Strain\_Concentration

(FEMaterialPoint& mp, const int isol) = 0;

//! second derivative of solubility w.r.t. strain

virtual double Tangent\_Solubility\_Strain\_Strain

(FEMaterialPoint& mp) = 0;

//! second derivative of solubility w.r.t. concentration

virtual double Tangent\_Solubility\_Concentration\_Concentration

(FEMaterialPoint& mp, const int isol, const int jsol) = 0;

//! set solute ID

void SetSoluteID(const int ID) {m\_ID = ID;}

private:

int m\_ID; //!< solute ID

};

These functions provide the solubility and its first and second derivatives with respect to strain and concentration. The dependence on strain is only via *J*=det**F**. A solubility material must be associated with each solute in a multiphasic mixture.

## The FEElasticMaterialPoint class

As was shown above, the stress and tangent are evaluated at so-called material points. These are points at which the state of the deformation is tracked. Most often, these points correspond to the integration points of the elements. Material points are implemented by deriving special classes from the FEMaterialPoint base class. A class of particular importance is the FEElasticMaterialPoint class which is used to define the state of hyper-elastic materials. Unless your material implementation requires additional (history-dependent) data, this is the class you will use in the stress and tangent functions.

The FEElasticMaterialPoint class defines the state of an elastic material point, or more precisely, the state of a material derived from the *FEElasticMaterial* class. It defines the following state variables.

* *vec3d r0*: the position of the material point in the reference frame
* *vec3d rt*: the position of the material point in the spatial frame
* *mat3d F*: the deformation gradient
* *double J*: the Jacobian, that is the determinant of the deformation gradient
* *mat3d Q*: a rotation matrix describing the element coordinate system
* *mat3ds s*: the Cauchy stress tensor

If your material derives from *FEElasticMaterial* (or *FEUncoupledMaterial*) and no additional state data is required, you do not need to derive a new material point class. For all elastic materials this is the default material point data class to store the point data.

This class also defines some useful functions that can facilitate the implementation of stress and tangent functions.

* *Strain*: Calculate the Euler-Lagrange strain tensor.
* *RightCauchyGreen*: Calculate the right Cauchy-Green strain tensor.
* *RightCauchyGreen*: Calculate the left Cauchy-Green strain tensor.
* *DevRightCauchyGreen*: Calculate the deviatoric right Cauchy-Green tensor.
* *DevLeftCauchyGreen*: Calculate the deviatoric left Cauchy-Green tensor.
* *pull\_back*: Calculate the pull-back of a 2nd order tensor.
* *push\_back*: Calculates the push-back of a tensor. Two versions are defined. One for 2nd order symmetric tensors, and one for 4th order symmetric tensors.

For materials that require additional information to be stored at the material points (e.g. history-dependent materials) a new material point class needs to be defined, by deriving from the FEElasticMaterialPoint class. See section 2.4.3 for more information.

## Using the new material class

If the steps to register the material and its material parameters have been followed as outlined in the previous sections, the material class will be seamlessly integrated in FEBio’s framework. One of the important consequences of this is that the xml-input file reader will automatically recognize the new material and its parameters. For example, imagine the user created a new class *MyFancyMaterial* and named it *“fancy material”* by registering the material as follows.

REGISTER\_MATERIAL(MyFancyMaterial, "fancy material");

This macro associates the name “*fancy material”* with the *MyFancyMaterial* class. This name will now be used as the type identifier in the xml-input file.

<material id="1" type="fancy material">

...

</material>

FEBio will recognize the type identifier as the name of the *MyFancyMaterial* class and will create an instance of this class. For all the elements in the mesh that have the material ID of this class (in this case “1”), the stress and tangent functions of the new class will be automatically called.

Material parameters are identified in a similar way. For each material parameter, the *ADD\_PARAMETER* macro associates a name with the parameter. For example, imagine that for our new class *MyFancyMaterial* the following parameter is defined,

ADD\_PARAMETER(m\_a, FE\_DOUBLE\_PARAM, "param\_a");

The user can now enter a value for this parameter in the FEBio input file as follows,

<material id="1" type="fancy material">

<param\_a>0.123</param\_a>

</material>

FEBio will now automatically read the value (here 0.123) and store it in the *m\_a* variable which will be defined as a public member variable of the *MyFancyMaterial* class.

## Debugging the Material Implementation

Implementing a new material formulation can be tricky sometimes. Particularly the implementation of the correct tangent stiffness is often quite challenging. For this reason, FEBio offers a few tools that can help in diagnosing a new material implementation.

First, it is highly recommended to use the available tensor classes to implement the stress and tangent stiffness of the material. These classes allow the user to stay as true as possible to the mathematical formulation, facilitating the readability of the code. Obvious mistakes, such as sign errors, will therefore be relatively easy to spot by direct comparison of the code with the mathematical equations.

FEBio also offers a tangent diagnostic tool, which allows the user a more direct inspection of this tangent implementation. The tool basically compares the actual implementation of the tangent with an approximation that is obtained by calculating the finite difference of the residual. To run the diagnostic, a separate FEBio input file needs to be defined. An example for the neo-Hookean material follows.

1. <?xml version="1.0"?>
2. <febio\_diagnostic type="tangent test">
3. <Control>
4. <time\_steps>1</time\_steps>
5. <step\_size>1</step\_size>
6. <plot\_level>PLOT\_DEFAULT</plot\_level>
7. </Control>
8. <Scenario type="uni-axial">
9. <strain>0.15</strain>
10. </Scenario>
11. <Material>
12. <material id="1" name="Solid" type="neo-Hookean">
13. <E>1</E>
14. <v>0.45</v>
15. </material>
16. </Material>
17. </febio\_diagnostic>

The diagnostics input file also takes an xml-formatted input file and is structured similarly as the FEBio input file. The first line is the xml declaration as required by the xml standard. The next line defines the root element of the xml format. In this case, it is defined as *febio\_diagnostics* to indicate that this file is a diagnostics file. The *name* attribute identifies the type of diagnostic this file describes, and in this case this is a “*tangent test*”. Next follows the definition of the three sections of the file.

The first section, the *Control* section, defines some general control settings such as the number of timesteps, time step size and so on.

The second section, the *Scenario* section, defines the type of model and boundary conditions to apply. This section replaces the geometry section in the usual FEBio input file. The geometry is now defined implicitly through the scenario. The *uni-axial* scenario runs a simple uni-axial tension or compression problem on a unit cube. The maximum strain level can be defined through the *strain* parameter.

The third section defines the material that will be assigned to the model. In the uni-axial scenario, only one material needs to be defined with the corresponding material parameters. Note that when the material class is properly registered with the framework as explained above, no additional steps need to be taken to use the tangent diagnostics feature aside from creating the diagnostics input file for the new material.

To run the tangent diagnostic, simply type the following at the command prompt.

>febio –d <filename> [ENTER]

Note that the command option –d needs to be used instead of the usual –i to inform FEBio that you are running a diagnostics problem and not a regular model. Replace *<filename>* with the name of the actual input file.

This diagnostic test outputs a log file that contains the tangent stiffness as calculated from the implementation and a finite difference approximation to this tangent. It also contains the difference between these two matrices and the matrix element where the difference is largest. Although a small difference between the two matrices can be expected due to the finite difference approximation, the difference should be small, e.g. less than 0.01%. If this is not the case, there is probably a mistake in either the *Stress* function or in the *Tangent* function or both. To identify the culprit, the result of the simulation, which is reported as usual in the plot file, can be compared to a known solution (or a solution obtained in a different fashion). If the solutions correspond, then the problem most likely lies with the tangent implementation. If the solutions do not agree, then the implementation of the Cauchy stress is probably also erroneous.

## Advanced Topics

### Array parameters

It is possible to define an array of parameters using a single material parameter declaration. This can be done by first defining a member variable as an array in the class definition. For example, imagine that the new material class has the following variable declared.

double m\_a[3];

To define the variable *m\_a* as the storage for a material parameter, the user can use the *ADD\_PARAMETERV* macro. For example,

ADD\_PARAMETERV(m\_a, FE\_PARAM\_DOUBLE, 3, "a");

This macro requires four parameters. The first parameter is the variable that will store the material parameters. The second is the type of the variable. In this case the variable *m\_a* is declared as an array of doubles, so the *FE\_PARAM\_DOUBLE* has to be used. The third parameter is the size of the array and the fourth parameter is the string name of the variable that will be used to identify the variable in the FEBio input file. In the input file, the parameter’s values can then be defined using a comma-separated list. For example,

<a>0.1, 0.23, -0.73</a>

There is no limitation on the size of array parameters. Currently, the only types that are supported for array parameters are *int* and *double*. These are declared using the *FE\_PARAM\_INT* and *FE\_PARAM\_DOUBLE* identifier respectively.

### Uncoupled Materials

Incompressible materials are an important class of materials since they are dealt with in a very particular manner. FEBio assumes that such materials use a decoupled hyperelastic strain energy function.



Here, **C** is the right Cauchy-Green tensor, is the deviatoric right Cauchy-Green tensor and *J* is the Jacobian. Since the incompressibility constraint can sometimes be hard to enforce for these materials with the usual displacement formulation of FE, a different formulation is used. FEBio uses a three-field formulation that requires a separate integration rule for the dilatational stiffness contribution. We refer to the FEBio theory manual for a more detailed description of the theory of incompressible hyperelasticity. As a consequence of the different formulation, incompressible materials require a few changes to the basic procedure.

First, incompressible materials using a decoupled strain energy function, need to be derived from the base class *FEUncoupledMaterial*. An example of such a class is the *FEMooneyRivlin* material. This class is defined as follows.

1. class FEMooneyRivlin : public FEUncoupledMaterial
2. {
3. ...
4. };

The second important difference relates to the calculation of the stress. For a material with a strain energy function like , the stress is given by,



The pressure *p* is calculated by FEBio. The only thing that the material class needs to implement is the second term. This must be done in the *DevStress* member function which is inherited from *FEUncoupledMaterial*. For example, for the Mooney-Rivlin material, the stress is calculated as follows.

1. mat3ds FEMooneyRivlin::DevStress(FEMaterialPoint& mp)
2. {
3. FEElasticMaterialPoint& pt = \*mp.ExtractData<FEElasticMaterialPoint>();
4. ...
5. mat3ds T = B\*(W1 + W2\*I1) - B2\*W2;
6. return T.dev()\*(2.0/J);
7. }

Similarly, the elasticity tensor requires a slightly different form. It can be shown that it can be decomposed as follows.



Here,



and is the deviatoric tangent stiffness. Again we refer to the FEBio Theory Manual for a more detailed explanation of the elasticity tangent for nearly incompressible materials.

The important thing here is that the *DevTangent* function only needs to return . The terms  and are added automatically by FEBio so the user does not need to do this. Again, the *FEMooneyRivlin* class gives an example.

1. tens4ds FEMooneyRivlin::DevTangent(FEMaterialPoint& mp)
2. {
3. FEElasticMaterialPoint& pt = \*mp.ExtractData<FEElasticMaterialPoint>();
4. mat3ds WCCxC = B\*(W2\*I1) - B2\*W2;
5. // ...
6. tens4ds cw = (BxB - B4)\*(W2\*4.0\*Ji) - dyad1s(WCCxC, I)\*(4.0/3.0\*Ji) + ...
7. tens4ds c = dyad1s(devs, I)\*(-2.0/3.0)+(I4-IxI/3.0)\*(4.0/3.0\*Ji\*WC) + cw;
8. return c;
9. }

It is important to note that at this point, the dilatational function *U*, is defined by FEBio and cannot be specified by the user.



The parameter *K* is referred to as the bulk-modulus. This variable is defined by the *FEUncoupledMaterial* base class so the user does not need to define this parameter in the new material class. This variable is also registered in the framework as can be accessed from the input file using the name “k”.

<material id="1" type="Mooney-Rivlin">

...

<k>1000</k>

</material>

All materials derived from *FEUncoupledMaterial* will automatically inherit this material parameter.

### User-defined material points

In the previous sections we touched briefly on the concept of *material points*. In this section we will elaborate on this important topic and discuss how it can be used to add additional element data, e.g. for use in history-dependent materials.

A *material point* is a point in the material at which the state is stored. Usually these correspond to the integration points of the elements. The state of the material at this point is defined by the position, the deformation gradient, the stress and so forth.

The *FEMaterialPoint* class (defined in *FEMaterialPoint.h*) defines the base class from which all material point classes are derived. It does not define any state data, except for two variables, namely *time*, which defines the current time value and *dt*, which defines the current time increment. To define additional material data, one would need to derive a new material point class from this base class.

Note that this class is an abstract base class since it defines several virtual functions that have no definition. These functions have to be defined in the derived class. They are the following functions: *Init, Copy and Serialize*.

The *Init* function is called before the start of each time step. It has one parameter, *bflag*, a Boolean flag that indicates whether this is the first time this function is called. During the first time this function is called, you can allocate and initialize data variables.

The *Copy* member is called to create a *shallow copy* of the material point class. This is used to store the current state of the material point before the time step begins. When the time step fails this copy function allows FEBio to recover the data from the last converged time step and restart the time step using a smaller time step.

The *Serialize* function is used to store (and read) the material point data to (from) the data archive, a file stored on disk and used for cold restarts.

As discussed above, one particularly important material point class is the FEElasticMaterialPoint class which stores the state data for all elastic materials. If your new material is an elastic material, you probably don’t need to redefine a new material point class. See section 2.2 for a detailed discussion of this material point class. However, if any additional data is required that is not stored in the default material point classes, a new material point class must be defined. For elastic materials, it is best to derive a new material point class from the *FEElasticMaterialPoint* class.

### The FEBiphasicMaterialPoint Class

The *FEBiphasicMaterialPoint* class is needed for biphasic and multiphasic materials. It provides additional data needed for materials that include an interstitial fluid. This class also illustrates how a new material point class may be derived from the *FEElasticMaterialPoint* class.

The *FEBiphasicMaterialPoint* class defines the following state variables.

* *double m\_p*: the effective interstitial fluid pressure
* *vec3d m\_gradp*: the gradient of the effective interstitial fluid pressure
* *vec3d m\_w*: the interstitial fluid volumetric flux relative to the solid
* *double m\_pa*: the actual interstitial fluid pressure
* *double m\_phi0*: the solid volume fraction in the reference configuration

The effective and actual interstitial fluid pressure differ in a mixture that contains solutes; in a biphasic material these two parameters have the same value.

An example of a function that uses the *FEBiphasicMaterialPoint* is the *Permeability* function of a *FEHydraulicPermeability* object. For example, in the case of the Holmes-Mow permeability, this function is as follows:

//---------------------------------------------------------------------

//! Permeability tensor.

mat3ds FEPermHolmesMow::Permeability(FEMaterialPoint& mp)

{

FEElasticMaterialPoint& et =

\*mp.ExtractData<FEElasticMaterialPoint>();

FEBiphasicMaterialPoint& pt =

\*mp.ExtractData<FEBiphasicMaterialPoint>();

// relative volume

double J = et.J;

// referential solid volume fraction

double phi0 = pt.m\_phi0;

// --- strain-dependent isotropic permeability ---

return mat3dd(m\_perm\*pow((J-phi0)/(1.0-phi0),m\_alpha)

\*exp(m\_M\*(J\*J-1.0)/2.0));

}

Note that this function extracts both the *FEElasticMaterialPoint* and the *FEBiphasicMaterialPoint* from the parent *FEMaterialPoint* object. This example illustrates how the information stored in both of these material points may be needed for the evaluation of a particular function.

### The FESolutesMaterialPoint class

The *FESolutesMaterialPoint* class is needed for multiphasic materials. It provides additional data needed for materials that include solutes. This class also illustrates how a new material point class may be derived from the *FEBiphasicMaterialPoint* class.

The *FESolutesMaterialPoint* class defines the following state variables.

* *int m\_nsol*: the number of solutes in this mixture
* *vector<double> m\_c*: the effective concentrations of all the solutes
* *vector<vec3d> m\_gradc*: the gradients of the effective concentration
* *vector<vec3d> m\_j*: the solute molar fluxes
* *vector<double> m\_ca*: the actual concentrations of all the solutes
* *double m\_psi*: the electric potential
* *vec3d m\_Ie*: the electric current density
* *double m\_cF*: the fixed charge density in the current configuration

An example of a function that uses the *FESolutesMaterialPoint* class is *FEMultiphasic::SoluteFlux*, which evaluates the molar flux of a specific solute in a multiphasic material:

//---------------------------------------------------------------------

//! Calculate solute molar flux

vec3d FEMultiphasic::SoluteFlux(FEMaterialPoint& pt, const int sol)

{

FEBiphasicMaterialPoint& bpt =

\*pt.ExtractData<FEBiphasicMaterialPoint>();

FESolutesMaterialPoint& spt =

\*pt.ExtractData<FESolutesMaterialPoint>();

// fluid volume fraction (porosity) in current configuration

double phiw = Porosity(pt);

// concentration

double c = spt.m\_c[sol];

// concentration gradient

vec3d gradc = spt.m\_gradc[sol];

// solute diffusivity in mixture

mat3ds D = m\_pSolute[sol]->m\_pDiff->Diffusivity(pt);

// solute free diffusivity

double D0 = m\_pSolute[sol]->m\_pDiff->Free\_Diffusivity(pt);

// solubility

double khat = m\_pSolute[sol]->m\_pSolub->Solubility(pt);

int z = m\_pSolute[sol]->ChargeNumber();

double zeta = ElectricPotential(pt, true);

double zz = pow(zeta, z);

double kappa = zz\*khat;

// fluid flux w

vec3d w = bpt.m\_w;

// solute flux j

vec3d j = (D\*(w\*(c/D0) - gradc\*phiw))\*kappa;

return j;

}

Note that this function extracts both the *FEBiphasicMaterialPoint* and the *FESolutesMaterialPoint* from the parent *FEMaterialPoint* object. This example illustrates how the information stored in both of these material points may be needed for the evaluation of a particular function.

# Tensors in FEBio

The mathematical language of continuum mechanics is tensor analysis. Most quantities that appear in this branch of mechanics can often be expressed most conveniently as tensors (e.g. stress, strain, elasticity tensors, etc.). Consequently, the finite element method for continuum mechanics requires the numerical implementation of tensor relations. Since managing tensors in a code can be quite tricky, a library was developed that facilitates the implementation of tensor quantities.

FEBio has classes for first-order (vectors), second-order and (symmetric) fourth-order tensors. Each class will be discussed below.

## First-order tensors

First order 3D tensors (or vectors) are defined by the vec3d class. Operators are defined that implement basic vector algebra.

vec3d a, b;

vec3d c = a+b;

vec3d d = a-c;

Vectors can be multiplied or divided by scalars.

vec3d a;

double c;

vec3d b = a\*c;

The inner product (dot product) is defined by the multiplication ‘\*’ symbol.

vec3d a, b;

double c = a\*b;

The outer product (cross product) is defined by the hat ‘^’ symbol.

vec3d a, b;

vec3d c = a^b;

The norm of a vector can be found by the vec3d::norm() member function.

vec3d a;

double L = a.norm();

To make a vector into a unit vector, use the vec3d::unit() member function.

vec3d n;

n.unit();

The dyadic product is defined by the various dyad functions, described in the next section.

This class is defined in the file vec3d.h.

## Second-order tensors

The tensor library in FEBio offers several classes to represent second order tensors.

* mat3d: general 3D second-order tensor.
* mat3ds: symmetric 3D second-order tensor.
* mat3da: skew-symmetric 3D second-order tensor.
* mat3dd: diagonal 3D second-order tensor.

All second order tensor classes are defined in the file math3d.h.

These different classes and how they relate to each later will be discussed next.

### mat3d class

The mat3d class represents a general purpose 3D second-order tensor. All nine components are stored and no assumptions on symmetry are made. For example

mat3d A;

defines a variable that represents a 3D second-order tensor.

Operators are defined that allow basic tensor algebra.

mat3d A, B;

mat3d C = A + B;

mat3d D = A – B;

mat3d E = A\*B;

Tensors can be multiplied or divided by scalars.

mat3d A;

double a;

mat3d B = A\*a;

Tensor-vector multiplication is defined through the multiplication ‘\*’ operator.

mat3d A;

vec3d a, b;

b = A\*a;

To take the trace of a tensor, use the mat3d::tr() member function. The determinant can be calculated with the mat3d::det() member function.

To access the tensor components, use the mat3d::operator() member function.

mat3d A;

double s = A(0,0) + A(1,1) + A(2,2); // use A.tr() instead

Note that the indices are zero-based.

### mat3ds class

The mat3ds class can be used to store a symmetric 3D second-order tensor. For example, the stress tensor calculated by most constitutive models is stored as a mat3ds object.

mat3ds S;

The basic tensor algebra operations defined for mat3d (see above) are also defined for mat3ds.

This class only stores the six independent components of the tensor. In addition to using the mat3ds::operator() to access the matrix components, you can also use the mat3ds::xx(), mat3ds::yy(), mat3ds::zz(), mat3ds::xy(), mat3ds::yz(), mat3ds::xz() member functions.

mat3ds A;

double s = A.xx() + A.yy() + A.zz();

### mat3da class

The mat3da class can be used to store a skew-symmetric 3D second-order tensor. Only three components are stored.

mat3da W;

The same basic tensor algebra operations that are defined for the other tensor classes are defined for mat3da as well.

To convert a 3D vector into its corresponding skew-tensor, use the appropriate mat3da constructor.

vec3d w;

mat3da W(w);

The individual matrix components can be accessed with the mat3da::operator() member function.

### mat3dd class

The mat3dd class can be used to store a diagonal tensor, that is, a tensor whose only non-zero components are the diagonal elements. For example, the 3D second-order identity tensor can be defined as follows.

mat3dd I(1.0);

The same basic tensor algebra operations that are defined for the other second-order tensor classes are also defined for the mat3dd class.

## Fourth-order tensors

A fourth-order tensor that is both major and minor symmetric can be stored in the tens4ds class. This class is defined in the file tens4d.h.

Because of the assumed symmetries only 21 components are stored. For example, the elasticity tangent, calculated by most constitutive models, is stored as a tens4ds variable.

tens4ds C;

The basic tensor algebra is implemented for this class through a set of overloaded operators.

tens4ds A, B;

tens4ds C = A + B;

tens4ds D = A – B;

The tens4ds variables can be multiplied and divided by scalars.

tens4ds C;

double a;

tens4ds D = C\*a;

The contraction with a second order tensor (double-dot product) is defined by the tens4ds::dot() member function.

tens4ds C;

mat3ds S, E;

S = C.dot(E);

This implements the mathematical operation . Note that this operation is only defined for mat3ds tensors, that is second order symmetric tensors.

It is sometimes convenient to work with the 6×6 Voigt representation of a symmetric 4th order tensor. This matrix can be extracted using the tens4ds::extract() member function.

tens4ds C;

double D[6][6];

C.extract(D);

The individual tensor components can be accessed using the tens4ds::operator() member functions. There are two versions, namely one that takes two parameters and one that takes four. The one that takes two parameters, accesses the tensor components as the 6×6 Voigt matrix. The one that takes four parameters accesses the tensor components directly.

tens4ds C;

double a = C(1,3);

double b = C(1,1,0,1); // same component as a

The (symmetric) product of two fourth-order tensors is implemented in the ddots() function.

tens4ds A, B;

tens4ds C = ddots(A,B);

This implements the equation .

Double contraction with a second-order tensor is defined by the dot() function.

tens4ds C;

mat3ds A;

mat3ds B = dot(C,A);

The operation  (or in indicial notation ) is implemented in the vdotTdotv() function.

tens4ds T;

vec3d a, b;

mat3d A = vdotTdotv(a, T, b);

Note that the result is a mat3d object since no symmetry can be assumed for this operation.

## Implicit Type conversions

When using the tensor class described above, one must be aware of the implicit type conversions that are done during tensor algebra. The general rule is that the type of the result will be the type that assumes the least amount of symmetry.

mat3dd I(1.0);

mat3ds E;

mat3ds S = E – I;

In this example, the final result *S* is a mat3ds, since E is a mat3ds.

Another example.

mat3ds S;

mat3da W;

mat3dd I(1.0);

mat3d R = S + W – I;

The final result *R* is of type mat3d, since in general no symmetry can be assumed for the result of this expression.

## Dyadic products

A dyadic product of a tensor of order *n* and a tensor of order *m* creates a tensor of order *n+m*. In FEBio, you can use dyadic products to create a second-order tensor from two first-order tensors and a fourth-order tensor from second-order tensors. However, there are some limitations that one needs to be aware of.

The general dyadic product of two first-order tensors (vectors) is defined by the ampersand operator (&).

vec3d a, b;

mat3d A = a&b;

Note that the result is a mat3d object since in general no symmetry can be implied from this operation.

Two additional dyadic operators are defined for vectors that give a symmetric second-order tensor.

vec3d a, b;

mat3ds A = dyad(a);

mat3ds B = dyads(a,b);

The first one defines the dyadic product of a vector with itself.



The second one calculates the following.



In either case, the result is a mat3ds object, that is, a symmetric second-order tensor.

There are several dyadic products defined for symmetric second order tensors. The function names are not intuitive so it is important to understand the differences between the various definitions. Assume in the following that *A* and *B* are objects of type mat3ds.

tens4ds C1 = dyad1s(A);



tens4ds C2 = dyad1s(A, B);



tens4ds C4 = dyad2s(A);



tens4ds C4 = dyad2s(A, B);



tens4ds C5 = dyad4s(A);



tens4ds C6 = dyad4s(A, B);



Note that all these operations guarantee major and minor symmetries for the resulting fourth-order tensor.

# FEBio2 Road Map

This chapter is a temporary discussion of the planned modifications for FEBio2. It mostly contains discussions on the new program structure. This material can change at any time.

## File Format

The xml-based file format is to undergo several significant changes. The goal is to create a hierarchical structure that defines a one-to-one correspondence with the FEBio2 class structure. The advantage would be that this would make it easier to create an automatic interface to the file format when adding new features (just like it is now with most materials).

Another planned change is making the model components more independent. For instance, in 1.x, each element has a material ID defined. A possible change is that each element is defined inside a domain (or part) and the material assignments are based on the parts.

### Proposed changed

This document lists the changes to the FEBio file format version 2.0.

* The *version* attribute of *febio\_spec* must be *2.0*

<febio\_spec version="2.0">

* The biphasic module is called *biphasic* instead of *poro*.

<Module type="biphasic"/>

* The *Module* tag is replaced as an attribute of the *step* element.

<Step type="biphasic">...</Step>

* Data can be split across multiple files using the *include* attribute.

<Geometry include="file.feb"/>

This command will load the *Geometry* section from the *file.feb* file. The latter file must be a valid FEB file (it must start with the *febio\_spec* tag). It may contain several sections, but only the requestedsection will be loaded.

* The *Geometry* section can contain the optional sections *NodeSet* and *Part*. A node set is defined as follows.

<NodeSet name="set01">1,2,3,...,n</NodeSet>

The value is a list of nodes. A part is defined as follows. (The structure follows loosly the Abaqus structure.)

<Part elem="hex8" mat="1">

<elem id="1">1,2,3,4</elem>

<...>

</Part>

* The *force* and *prescribe* elements now take the *bc* attribute instead of the node lists.

<force bc="x">

* The *Contact* section is now a separate section, instead of a subsection of *Boundary*.

<Contact>

<contact type="sliding\_with\_gaps">

<...>

</contact>

</Contact>

* All surfaces, nodesets and parts should support the *name* attribute.

<contact type="sliding\_with\_gaps">

<master name="surface01"> ... </master>

<slave name="surface02"> ... </slave>

</contact>

* Renaming the *sliding\_with\_gaps* to *node\_to\_facet* to be more consistent with PreView’s contact definition. Same story for *sliding2* and *sliding3*.

## Material Parameters

### A new material interface

I would like to rethink the whole material interface. The idea is that a material simply becomes a container of material properties. Each property corresponds with a class that defines functions related to that parameter. For instance, the *solid* property defines the stress of the material. The *permeability* defines the hydraulic permeability. For instance, a biphasic material would be defined as follows.

<material id="1">

<solid type="neo-Hookean">

<E>1.0</E>

<v>0.3</v>

</solid>

<permeability type="const">1e-4</permeability>

</material>

A material for heat-transfer would then be

<material id="2">

<density>1.0</density>

<conductivity>1.0</conductivity>

<heat\_capacity>0.1</heat\_capacity >

</material>

A material for a thermal-elasticity problem could then be defined as follows.

<material id="2">

<density>1.0</density>

<conductivity>1.0</conductivity>

<heat\_capacity>0.1</heat\_capacity >

<elastic type="neo-Hookean">

<E>1.0</E>

<v>0.3</v>

</elastic>

</material>

In cases where material properties are simple constants, special forms can be defined.

<density>1.0</density>

This would be equivalent to

<density type="const">1.0</density>

Similar shortcuts could be defined for constant vector and tensor parameters.

To access a material property, the code would use the following

double dens = material.get<Density>();

### Position dependent parameters

Can a material parameter be made position dependent? One possible thought would be that each material point keeps a pointer to a list of material properties. The pointer could be owned by either the material point or the corresponding material. By default, the pointer will point to the material’s parameter list, but the list can be overridden for each material point. This would make the parameter list position dependent. Off course, if one parameter is made local, all parameters will be made local.

1. Note that you could obtain the Jacobian by taking the determinant of F directly. However, it is recommended to use the J member of the material point class since for uncoupled materials this value will not match the actual determinant (this is a consequence of the three-field implementation used for uncoupled materials which considers J as an separate field that is independent of the deformation.) [↑](#footnote-ref-1)