

# FEBio

FINITE ELEMENTS FOR BIOMECHANICS

*Version 1.4*

## Developer's Manual

**Last Updated: September 14, 2011**

### **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>

FEBio: <http://mrl.sci.utah.edu/software/febio>

### **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

<b>1</b>	<b>Compiling FEBio</b> .....	<b>4</b>
1.1	Compiling for Windows using Visual Studio 2008.....	4
1.2	Compiling for Linux .....	4
<b>2</b>	<b>Adding new materials</b> .....	<b>5</b>
2.1	Basic Procedure .....	5
2.1.1	Defining the new material class .....	5
2.1.2	Registering the new material .....	6
2.1.3	Defining the material parameters.....	7
2.1.4	Implementing the initialization function.....	8
2.1.5	Implementing the stress and tangent functions.....	8
2.2	Using the new material class.....	11
2.3	Debugging the Material Implementation .....	11
2.4	Advanced Topics .....	13
2.4.1	Array parameters.....	13
2.4.2	Uncoupled Materials.....	14
2.4.3	Alternative base classes .....	15
2.4.4	User-defined material points.....	15

# 1 Compiling FEBio

## 1.1 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. First, the FECore library will be built, which is then used to build FEBio.

## 1.2 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.

## 2 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. Implementing the stress and tangent functions.

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

### 2.1 Basic Procedure

#### 2.1.1 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:
8: public:
9:     // Cauchy-stress calculation
10:    virtual mat3ds Stress(FEMaterialPoint& pt);

```

```

11:    // Spatial elasticity tensor calculation
12:    virtual tens4ds Tangent(FEMaterialPoint& pt);
13:
14:    // class initialization (optional)
15:    virtual void Init();
16:    // declare as a registered class
17:    DECLARE_REGISTERED(FENeoHookean);
18:
19:    // declare as having a parameter list
20:    DECLARE_PARAMETER_LIST();
21: };

```

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 overwritten 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 declaration of the class is split over two separate files. One file, the so-called header file, defines 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 definition of new materials follow 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.

### 2.1.2 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. For example, it will be used 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.

### 2.1.3 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 defining material parameters, is to register them 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.

### 2.1.4 Implementing the initialization function

When implementing a new material class the user has the option to overwrite 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:
5:   if (m_E <= 0) throw MaterialError("Invalid value for E");
6:   if (!INRANGE(m_v, -1, 0.5)) throw MaterialError("Invalid ...
7: }
```

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 that it is implemented so that potential problems due to invalid material parameters can be caught quickly.

### 2.1.5 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:
5:   mat3d &F = pt.F;
6:   double detF = pt.J;
7:   double detFi = 1.0/detF;
8:   double lndetF = log(detF);
9:
10:  // calculate left Cauchy-Green tensor
11:  // (we commented out the matrix components we do not need)
12:  mat3ds b = pt.LeftCauchyGreen();
13:
14:  // lame parameters
15:  double lam = m_v*m_E/((1+m_v)*(1-2*m_v));
16:  double mu = 0.5*m_E/(1+m_v);
17:
18:  // Identity
19:  mat3dd I(1);
20:
21:  // calculate stress
22:  mat3ds s = (b - I)*(mu*detFi) + I*(lam*lndetF*detFi);
23:
24:  return s;
25: }

```

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 of 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. 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:
5:     // deformation gradient
6:     mat3d &F = pt.F;
7:     double detF = pt.J;
8:
9:     // lame parameters
10:    double lam = m_v*m_E/((1+m_v)*(1-2*m_v));
11:    double mu = 0.5*m_E/(1+m_v);
12:
13:    double lam1 = lam / detF;
14:    double mu1 = (mu - lam*log(detF)) / detF;
15:
16:    mat3dd I(1);
17:    tens4ds IxI = dyad1s(I);
18:    tens4ds I4 = dyad4s(I);
19:
20:    return IxI*lam1 + I4*(2*mu1);
21: }
```

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 the appendix.

## 2.2 Using the new material class

If the steps to register the material and its material parameters have been followed as outlined in the previous section, 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.

## 2.3 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 diagnostics 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.

## 2.4 Advanced Topics

### 2.4.1 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.

## 2.4.2 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.

$$W(\mathbf{C}) = \tilde{W}(\tilde{\mathbf{C}}) + U(J) \quad (0.1)$$

Here,  $\mathbf{C}$  is the right Cauchy-Green tensor,  $\tilde{\mathbf{C}}$  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 (0.1), the stress is given by,

$$\boldsymbol{\sigma} = p\mathbf{I} + \frac{2}{J} \text{dev} \left( \tilde{\mathbf{F}} \frac{\partial \tilde{W}}{\partial \tilde{\mathbf{C}}} \tilde{\mathbf{F}}^T \right) \quad (0.2)$$

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:
7:     return T.dev()*(2.0/J);
8: }
```

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

$$\mathbf{c} = \mathbf{c}_k + \mathbf{c}_p + \tilde{\mathbf{c}}_w \quad (0.3)$$

Here,

$$\mathbf{c}_\kappa = \frac{d^2U}{dJ^2} \mathbf{1} \otimes \mathbf{1}, \quad \mathbf{c}_p = p(\mathbf{1} \otimes \mathbf{1} - 2\mathbf{I}) \quad (0.4)$$

and  $\tilde{\mathbf{c}}_w$  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  $\tilde{\mathbf{c}}_w$ . The terms  $\mathbf{c}_p$  and  $\mathbf{c}_\kappa$  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.

$$U(J) = \frac{1}{2} K (\ln J)^2 \quad (0.5)$$

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.

### 2.4.3 Alternative base classes

(Discuss alternative base classes)

### 2.4.4 User-defined material points

(Discuss user-defined material points)