OBJECT ORIENTED DESIGN OF A
THERMO-MECHANICAL FEM CODE

A.H. van den Boogaard, N. van Vliet and J. Huétink
University of Twente
Faculty of Mechanical Engineering
P.O. Box 217, 7500 AE Enschede, The Netherlands

Key Words: object oriented software, finite element method, thermo-mechanical processes

Abstract. An object oriented design is presented for a computer program that can perform thermo-mechanically coupled analyzes. The target of the design is a flexible and robust computer program. It should be easy to adapt and extend, re-using existing code, without interfering with already established algorithms.

The program uses publicly available toolkits that are currently emerging as C++ packages. First of all the Standard C++ Library (formerly Standard Template Library) is used for packing items in container classes. Secondly the matrix and vector operations are derived from the Template Numerical Toolkit (TNT) and finally (not essentially for the numerical part) a graphical user interface is made, based on the wxWindows package, that can generate a GUI for Motif and MS-Windows with the same code.

Attention is given to the design of classes such as specific elements and material classes based on more general classes. A hierarchy of classes is constructed where general behavior is put high in the hierarchy and specific behavior low. The choice between inheritance and aggregation is made at several levels.
1 INTRODUCTION

Finite element analysis has become an important engineering tool in many engineering disciplines. Stress, thermal, fluid, electro magnetic and other types of problems can be modeled by finite elements. Also multi-physics problems, where a combination of these types are involved can be modeled, e.g. thermo-mechanical behavior.\(^1\)\(^-\)\(^3\) On top of that, there is a range of solution methods in each problem type e.g. static, dynamic, steady state, stability etc. The computer programs supporting these analysis have evolved over the last 4 decades.

With the growth of a computer program, maintainability, extendibility and quality assurance becomes an increasing problem, mainly because of the many possible interrelations between different parts of the code. This is especially true for e.g. COMMON blocks in Fortran. This is not only a problem for the commercial software vendor, but also for researchers doing more than simple two-element tests. In the last decade the object oriented design of software has become more and more popular.\(^4\) Regarding finite element analysis, a number of publications describe the use of object oriented design (OOD) or object oriented programming (OOP) for program development.\(^5\)\(^-\)\(^7\) An interesting follow up is the development of symbolic derivation of FEM procedures.\(^8\)\(^,\)\(^9\) In\(^10\) an object oriented environment is presented that focuses on adaptive mesh refinement.

In object oriented codes data can easily be protected against unwanted side effects, by denying access from other routines than the object’s routines itself. Although an object oriented design can be implemented in a procedural language the benefits of this strategy are fully exploited only with a language that supports object oriented programming. With the adoption of the ANSI C++ standard and the widespread availability of C++ compilers for different architectures and the efficiency of the underlying C code, C++ is currently the most obvious choice for an object oriented programming language.

A number of toolkits are publicly available in the C++ language. For FEM programs interesting areas are linear algebra routines, container classes (to store elements, nodes etc.) and graphical user interfaces. Although linear algebra routines are also available in Fortran, the object oriented paradigm makes their use much more flexible. The advantage of using toolkits is that the computational mechanics expert does not have to write code that was generated many times before, but can use e.g. linear algebra code that was made by experts in that field. Usually these toolkits are more efficient than the general ‘home-brewed’ subroutines and are prepared for e.g. parallel processing.

For container classes the recently adopted ANSI standard for the Standard C++ Library (formerly Standard Template Library STL) is a good choice. Since this toolkit is adopted as a standard it can be expected that future compilers will handle this code efficiently. For linear algebra the successor of LAPACK++, Sparselib++, IML++ and MV++ is emerging, called: Template Numerical Toolkit (TNT).\(^11\) This is a promising development. An excellent free toolkit for generating graphical user interfaces for Motif and MS-Windows is the wxWindows toolkit.\(^12\)
In this presentation an example is given of the object oriented design and implementation of a thermo-mechanically coupled finite element code for large deformation processes. Attention is given to the design of objects such as specific elements and material behavior based on more general objects. A hierarchy of objects is constructed where general behavior is put high in the hierarchy and specific behavior low. This yields objects with the highest degree of re-usability.

2 FINITE ELEMENT FORMULATION

In this section some basic equations are presented for thermo-mechanically coupled finite element analysis. To demonstrate the ability for a coupled analysis a material model is used in which the thermal part of the equations is purely linear and the mechanical part can be linear elastic or elastoplastic. The coupling is due to the thermal expansion only. Although simple, these models include all the ingredients for more general behavior: solution of nonlinear equations with a coupled thermal and mechanical part at element and integration point level.

In a nonlinear finite element analysis of thermo-mechanically coupled behavior the values for displacements and temperatures are calculated that realize mechanical and thermodynamic equilibrium. Often, and certainly if the material behavior is path-dependent, the external load is applied in increments. The nonlinear equations are solved by iterations e.g. a Newton–Raphson process. Collecting the displacements and temperatures in a vector $\mathbf{u}$ and the forces and fluxes in a vector $\mathbf{f}$ the Newton–Raphson process for one increment can be represented as:

```plaintext
$\Delta \mathbf{u}_0 = 0$
while ($|f_{ext} - f_{int}| > \delta$) do
$\Delta \Delta \mathbf{u}_i = \mathbf{K}^{-1}(f_{ext} - f_{int,i})$
$\Delta \mathbf{u}_{i+1} = \Delta \mathbf{u}_i + \Delta \Delta \mathbf{u}_i$
end while
```

where $\mathbf{K}$ represents the stiffness matrix and $f_{ext}$ and $f_{int}$ respectively the external force vector (with prescribed forces and fluxes) and the internal force vector.

The internal force vector and stiffness matrix are assembled from the contributions of the finite elements. The following equations determine the contribution of 1 element. The mechanical parts can be described as:

$$f_{int}^M = \int_V \mathbf{B}^T \sigma \, dV$$  \hspace{1cm} (1)

$$\mathbf{K}^{MM} = \int_V \mathbf{B}^T \mathbf{D} \mathbf{B} \, dV$$  \hspace{1cm} (2)
where $\mathbf{B}$ represents the relation between nodal displacements and strains, $\sigma$ the Cauchy stress vector and $\mathbf{D}$ the material stiffness matrix.

For a non-steady-state solution of the thermal equations, the thermal capacity of the material must be included. Choosing an Euler backward integration of the rate equations, the thermal parts of the force vector and stiffness matrix can be described as:

$$ f_{int}^{T} = \int_V \mathbf{N}^T \frac{\partial}{\partial x} \mathbf{q}\Delta t + \rho c \mathbf{N}^T \Delta T \, dV $$

$$ K^{TT} = \int_V \mathbf{N}^T \mathbf{N} \frac{\partial}{\partial x} \Delta t + \rho c \mathbf{N}^T \mathbf{N} \, dV $$

with $\mathbf{N}$ the interpolation vector, $\mathbf{q}$ the flux vector, $\Delta T$ the temperature increment and $\mathbf{A}$ the conductivity matrix (usually only $\lambda$ on the diagonal).

In the material models considered here, the only coupling between the thermal and mechanical part is the thermal expansion. The thermal strains influence the stress $\sigma$. In the stiffness matrix this results in a coupling part, where rows of the mechanical degrees of freedom are coupled to the columns of the thermal degrees of freedom:

$$ K^{MT} = \int_V -\mathbf{B}^T \mathbf{D} \alpha \mathbf{N} \, dV $$

Here $\alpha$ is a vector representation of the thermal expansion coefficient ($\alpha$ at the normal strain positions and 0 at the shear strain positions). If for every element the displacement degrees of freedom are placed before the thermal degrees of freedom, the stiffness matrix and internal force vector can be composed of the presented (sub-)matrices:

$$ f_{int} = \begin{bmatrix} f_{int}^M \\ f_{int}^T \end{bmatrix} $$

$$ K = \begin{bmatrix} K^{MM} & K^{MT} \\ K^{TM} & K^{TT} \end{bmatrix} $$

The sub-matrix $K^{TM}$ represents the influence of the displacements on the temperature, e.g. due to dissipation. In this paper dissipation is neglected and $K^{TM} = 0$.

The material behavior is determined by a linear relation between the Cauchy stress and the elastic strain:

$$ \sigma = \mathbf{D} \varepsilon^{e} = \mathbf{D} (\varepsilon - \varepsilon^{p} - \alpha \Delta T) $$

and a linear isotropic conduction relation:

$$ \mathbf{q} = -\mathbf{A} \cdot \nabla T $$

4
The equilibrium equations:

\[ \sigma \cdot \nabla + \rho b = 0 \]  \hspace{1cm} (10)

with \( b \) the body force, and

\[ \nabla \cdot q + \rho c \dot{T} = Q \]  \hspace{1cm} (11)

with \( Q \) the local heat production, are already accounted for in (1) and (3).

In the FEM program all integrals are determined by numerical integration. This yields a summation over the contribution of integration points.

3 GENERAL PROGRAM LAYOUT

In this section the object oriented environment is described in which the elements and materials 'live'. The element and material classes are described in more detail in sections 4 and 5.

3.1 Object oriented programming

Before describing the general program layout, a few introductory remarks are made about OOP. Most important is the concept of abstraction. Data and algorithms are not separated as in a typical procedural approach but are contained together in objects. As an easy example we can consider matrix-vector multiplication, where the algorithm depends on the data format for the matrix and vector e.g. storage by row, by column or a sparse storage scheme. Since the data and algorithm are so closely connected the two are contained together in a matrix–vector class. The class is a description of data and algorithms. In object oriented terminology these are called: ‘attributes’ and ‘methods’ or ‘member-functions’. An instantiation of a class is an object. In this terminology an integer variable is an object of the class ‘integer’.

If we use the matrix objects only by the ‘methods’ and not by the ‘data’ (data hiding), we can change the internal representation of the data and only have to adapt the methods of the class itself. This behavior can be enforced in C++ by defining the data in the class as ‘private’. The compiler will not accept external routines (not belonging to this class) to read from or write to the private data. For external use the internal data of a class is not important. The ‘public’ functions of a class define the behavior of the class, such as ‘multiplication with a vector’. For the user of a class it is sufficient to know that he can ask objects to perform a task, it does not matter how the objects realize this behavior. A nice feature of C++ is the ability to assign functions to operator symbols. This can be used e.g. to write matrix–matrix multiplication as:  

\[ A = B \cdot C; \]

Another important feature of OOP is ‘inheritance’. It is possible to derive a class from another class. The derived class inherits the behavior of the parent class without further declarations. It is possible to extend the derived class with specific behavior that was not available in the parent class and it is possible to overrule parent class behavior.
with specific derived behavior. This feature is used by programming generic behavior high up in the hierarchy and specific behavior low. It is also used to adapt objects to specific behavior, without any recoding in the original code, not even the addition of an IF–THEN structure. In this paper, inheritance is heavily used in the definition of element and material classes.

The crux of object oriented programming is the determination of objects that can relatively independent of its neighborhood contain data and perform well defined tasks.

### 3.2 Typical FEM program objects

Thinking about a finite element program, a number of entities can be distinguished that could be implemented as separate classes. Elements and nodes are the most obvious. In\textsuperscript{5,7,13,14} several classes are presented. The main classes that we use in our development are as follows.

**Domain** A domain collects all model data like lists of elements, nodes, loads, materials and geometries. It can setup the stiffness matrix and calculate the internal force vector.

**Node** A node has an initial location and zero or more degrees of freedom (dof). The nodes are the intermediate objects between elements and dofs.

**FiniteElement** A FiniteElement has an array of node-pointers, i.e. it does not refer to a node number, but directly to the memory location of the node. It must at least be able to give its own stiffness matrix and internal force vector. The sub-class of numerically integrated elements contains a variable number of material-points, representing the material at an integration point.

**Dof** A degree of freedom has a type, such as displacement, rotation or temperature. Depending on the type it can have a direction. Furthermore it has a restriction that can be: free, fixed or prescribed. This determines whether the degree of ‘freedom’ is really free or that its value is kept constant at a value of ‘0’ or prescribed to a non-zero possibly non-constant value. A ‘dof’ has a dof-number to represent the dof in a set of equations.

**Matrix** At many levels of abstraction matrices appear. In a material point e.g. $3 \times 3$ matrices can represent the deformation gradient and at domain level we have the sparse, possibly symmetric, system matrix. For an efficient code, it is important to use specific sub-classes of the matrix class that make use of the specific features of the matrix under consideration. In sub-classes an efficient implementation must exist for full and sparse and for symmetric and non-symmetric matrices. The matrix class must be able to perform the usual linear algebra manipulations as matrix addition, multiplication and deriving the solution of a set of equations. The Matrix class is derived from the public domain ‘Template Numerical Toolkit’\textsuperscript{11}.
Material  A material class contains many sub-classes that can represent material-models. Two sub-classes used in this paper are the class of structural materials and the class of thermal materials. An important feature of the material class is that it can dynamically create ‘Material_Point’s. This feature is used during the initiation of elements. For every material used in the model only one material object is generated that contains the material parameters.

Material_Point  The Material_Point class implements the algorithms for stress-strain relations or temperature-flux relations. It uses the parameters of the models described in the Material class and stores relevant data at material point level e.g. the equivalent plastic strain or the current temperature. For every (integration-)point in the model a Material_Point is generated. Again two important sub-classes are defined: a Structural_Point class and a Thermal_Point class. For each separate routines are required e.g. to give (to the element) the material stiffness matrix or the material conductivity matrix.

Shape  In the Shape class, the shape functions and its partial derivatives are defined. This class is only used by numerically integrated elements. The Shape class also contains the numerical integration scheme and can give the locations of the integration points. Sub-classes of the Shape class are e.g. the Triangular, Quadrilateral and Hexahedral classes. The sub-classes are further refined in sub-sub-classes, defining the interpolation order.

Loads  The load class contains the loads as given by the model description. It can transform this load-definition to force-vectors during initialization of the analysis.

Control  The Control class works together with the Domain class. The domain knows about the model description, while the control class knows what must be done with it e.g. a nonlinear analysis with a full Newton–Raphson process. It contains data like the number of increments, step-size, maximum number of iterations and convergence criteria. It can not directly specify a task for an element or a node, because only the domain knows about these items. The control class can ask the domain to give the global stiffness matrix, the internal force vector and the unit load vector. The control class does not know the meaning of any degree of freedom, whether it is a displacement, rotation, temperature or whatever.

The general process flow now is the following.

1. Create the model by adding nodes, elements, materials, geometries, loads and supports to a domain object.

2. Create a control flow, stored as part of a control object.
3. Send a message to the control object, with the domain object as an argument, to execute itself with this domain.

4. Send a message to the domain object to print or plot selected data.

The execution of a nonlinear analysis is defined at several levels in the control class. At the highest level it sends a message to the domain to initialize itself and then it sends repetitive messages to itself to perform an increment until the end load is achieved. In reaction to the ‘perform increment’ message it sends a message to the domain object to initialize an increment, it performs some initialization itself and then it sends repetitive messages to perform an iteration, until convergence is achieved (see section 2).

For every iteration the domain is asked to give its internal force vector and if necessary its stiffness matrix. The control object subtracts the internal force vector from the external force vector and determines the delta-incremental ‘displacements’. Finally the delta-incremental displacements are added to the incremental displacements.

The domain mainly reacts on messages from the control object by executing a loop over all elements and assembling the element matrices and vectors in the global matrix and vector. In turn, numerically integrated elements react on messages from the domain object by executing a loop over all material points. The interaction between domain and elements and between element and material-points is designed for reuse in thermo-mechanical analyzes. This topic is described in the next two sections.

4 THERMO-MECHANICAL ELEMENTS

This section focuses on the FiniteElement class and its interaction with the Domain class and the MaterialPoint class. Special attention is given to the creation of a thermo-mechanically coupled element sub-class.

4.1 The Generic Element

A Domain object can contain a number of elements. These elements are collected in a linked list, that is derived from the Standard C++ Library. To define a generic element behavior we first determine which tasks must be performed by all elements, irrespective of the element type, shape etc. These tasks must be performed by public member functions, so that the domain object can use these functions.

As we have seen in the previous section the Domain object can be asked to initialize itself for a new analysis, initialize a new increment and it can be asked to give its global stiffness matrix and internal force vector. The domain object must do this, using the elements it contains. It can be foreseen that in an iterative process, the internal force vector depends on the current displacement increments, therefore the element status must be updated every iteration. A generic element is devised that can satisfy the requests from a domain object (see figures 1 and 2). In the graphical representation of the class structure the Booch notation is used.

The dashed cloud represents a class with its name. Beneath
a line some of its attributes and functions can be given. Arrows point from a derived class to its parent. A downward pointing triangle with an ‘A’ inside means that the class is an abstract class. This is a class that cannot be instantiated itself, but can only be used as the parent of some derived class. The actual behavior of the element must be specified in derived classes.

![Class Structure Diagram]

Figure 1: The element class structure.

In the initialization of the analysis, the nodes must be filled with degrees of freedom that are necessary for the elements that are connected to each node. In this way a thermal element will generate only temperature dofs, a 2-dimensional structural element will generate 2 displacements per node and a 3-dimensional element will generate 3 displacements per node. A thermo-mechanically coupled element will generate displacement dofs and temperature dofs as needed. A virtual function add_dofs_to_nodes() is defined that
A.H. van den Boogaard, N. van Vliet, J. Huétink

**class** FiniteElement

{

private:
    Matrix transmat;
    Dof *dof_array;

public:
    void add_dofs_to_nodes();
    virtual void init_increment()=0;
    virtual void iteration_update( const Vector& free, const Vector& prescribed )=0;
    void PutLHS_in_Global( Vector& LHS );
    void PutStiff_in_Global( Matrix *system );

protected:
    virtual Matrix GetLocalStiff() const=0;
    virtual Vector GetLocalLHS() const=0;

}

**Figure 2:** Some functions of the FiniteElement class.

performs this task. The dof-numbers and the transformation matrix from the local dofs to the global dofs are stored in the element. The transformation matrix accounts e.g. for the difference between a local element coordinate system and the coordinate systems of the nodes.

The initialization of the increment and the update per iteration can be defined for specific elements. In the generic element it is defined that the functions init\_increment and iteration\_update must exist. The functions are defined pure virtual, that means that the code for this function must be provided in derived classes.

The remaining public functions are PutLHS\_in\_Global and PutStiff\_in\_Global. The data and functions that are in the protected part of the declaration can only be used in the class itself and its derived classes. The pure virtual functions GetLocalStiff and GetLocalLHS are provided by derived classes. Because they are declared here, the function PutLHS\_in\_Global can use GetLocalLHS even before any derived class is defined. The correct code for GetLocalLHS is the responsibility of the derived classes. In run-time PutLHS\_in\_Global calls the correct GetLocalLHS function, transforms it to the global dofs and adds the contribution to the internal force vector. The same relations hold for PutStiff\_in\_Global and GetLocalStiff.
4.2 Numerically integrated elements

Only for very simple elements (geometry and material) can the integrals of section 2 be derived analytically. An important class of elements is the class of numerically integrated elements. As can be seen in figure 1 the class NumInt_Element is still an abstract class. It contains a Shape object and an array of Mat(Point objects. In the shape object the element shape and interpolation (linear triangular, quadratic quadrilateral etc.) are contained as well as the integration scheme (natural coordinates and weight factors).

The classes that are derived from the NumInt_Element class are the structural and the thermal element classes: NumInt_Structural and NumInt_Thermal. The generic class FiniteElement puts the stiffness matrix and the force vector in the global system of equations, based on the stiffness matrix and force vector in local coordinate systems that are returned by the functions GetLocalStiff() and GetLocalLHS(). These functions are implemented in the derived classes. For the NumInt_Structural class the local stiffness matrix and local force vector are derived by numerical integration according to equations (1) and (2). Based on the partial derivatives of the shape functions to the natural coordinates, this class is capable of generating the B-matrix. For the NumInt_Thermal class the local stiffness matrix and local force vector are derived by numerical integration according to (3) and (4). This class is capable of generating the $\frac{\partial N}{\partial x}$-matrix. With the function ‘iteration_update()’ the new values for the strain increment or the temperature increment are passed to the material points. It is the task of the material points to process this data and be prepared to give the current stress or current flux.

The subject of this paper is the derivation of the NumInt_ThermoMech class. After all the preparatory work this is easy. The NumInt_ThermoMech class uses multiple inheritance. It inherits the behavior of two parent classes: NumInt_Structural and NumInt_Thermal. The functions GetLocalStiff(), GetLocalLHS() and iteration_update() are overwritten with new ones. However these new routines can use the routines from the parent classes and store the structural and thermal parts as described in (6) and (7). Only the calculation of the not yet existing coupling part from (5) must be programmed in this class. The function ‘iteration_update()’ merely has the function of passing the temperature increment to the NumInt_Thermal parent class and passing the strain increment to the NumInt_Structural parent class, after it has subtracted the thermal strain. As an example the code for GetLocalStiff() for the NumInt_ThermoMech class is presented in appendix A.

5 THERMO-MECHANICAL MATERIAL

For the description of the material behavior two class hierarchies are used: Material_type and Material_Point. The first defines the material model with its parameters, the second stores e.g. deformation history data and implements the actual algorithm.
5.1 The Material_type class

In the Material_type hierarchy, the material models are defined with their model parameters. The class structure is presented in figure 3. In a model description it can e.g. be stated that the material is represented by a Von Mises elastoplastic model with specific parameters for the Young’s modulus, Poisson’s ratio and hardening curve. For this material one object of class VonMises_type is instantiated. The object stores the material parameters. The intermediate classes Structural_type and Thermal_type are useful for the identification of the material model, they do not contribute any attribute or function. However, the Material_type that is passed to a NumInt_Structural element must have a Structural_type part and the Material_type that is passed to a NumInt_ThermoMech el-

Figure 3: The material-type class structure.
ement must have both a Structural type part and a Thermal type part. In the presented design this can be checked easily.

As can be seen in the Booch diagram, a ThermoElastic type object inherits the attributes Young and Poisson from the Elastic type and it inherits lambda, capacity and density from the LinearThermal type. The expansion coefficient ‘alpha’ is stored additionally. An important function that is redefine in every sub-class that can be instantiated is the ‘new Material Point()’ function. This function forms the relation with the MaterialPoint hierarchy.

5.2 The MaterialPoint class

The MaterialPoint class hierarchy exists next to the Material type classes. As can be seen in figure 4 the class structure maps to the structure of Material type. In the MaterialPoint objects the state of a material point e.g. in an integration point is stored. In case of numerical integration, for every integration point one MaterialPoint is instantiated. During initialization of the element, the element ‘has’ a pointer to the Material type object that represents the material type of this element. When it initializes the integration points it sends a message to the Material type object to create a new MaterialPoint. This is performed by the public member function ‘new MaterialPoint()’ of the Material type.

If e.g. a visco-plastic material model is developed, a class ViscoPlastic type must be programmed that inherits from the Structural type class and a ViscoPlasticPoint that inherits from the StructuralPoint class. Passing the ViscoPlastic type to a NumInt-Structural element during initialization will result in the creation of ViscoPlasticPoint objects in the integration points, without any change in the existing code.

In the StructuralPoint class the functions are declared that are needed by a NumInt-Structural element to initialize an iteration (set strain increment), to generate the stiffness matrix (get D mat) and to generate the internal force vector (get stress). The implementation of these functions are left to the derived classes. E.g. the VonMisesPoint stores the elastic strain, equivalent plastic strain etc. and can with this (private) data implement the function get stress().

In the ThermalPoint class the functions are declared that are needed by a NumInt-Thermal element (get flux, get conduction and set potential incr). Again the implementation of these functions is left to the derived class.

As with the elements and material type classes, the creation of a thermo-mechanical material point is easy since the thermal part and the mechanical part already exist. Only the thermal expansion must be treated by an additional part of the code. The additional code for a ThermoPlasticPoint is given in appendix B. The set strain increment() function is redefined. First the thermal strains are subtracted from the total strain increments and subsequently this new strain is passed to the original VonMisesPoint. The thermal strain is calculated when the temperature increment is added to the material point. This needs a redefinition of the set potential incr() function.

13
6 CONCLUSIONS

After the creation of separate mechanical and thermal elements, coupled thermo-mechanical elements can be created, mainly by inheritance from parent classes. The material behavior for thermo-mechanically coupled analysis is based on mechanical behavior and thermal behavior. This can directly be used in the object-oriented design.

The presented implementation in C++ makes it possible to add new material models by deriving them from a parent class. This can be done in a separate file, without any change to the existing code. By passing the new material type to the elements at initialization, the new material behavior is introduced in the model.

Future developments are the inclusion of large deformation kinematics. It is supposed that this can be performed by defining a layer between integration points and material
points. The integration points could, e.g. based on the deformation gradient, calculate the deformations in a co-rotating coordinate system.

A STIFFNESS MATRIX CALCULATION OF THE COUPLED ELEMENT

Matrix NumInt_ThermoMech::GetLocalStiff() const
{
    double DetJ, dV, alpha;

    Matrix Stiffmat(nr_of_local_dofs,nr_of_local_dofs);
    Stiffmat.set_zero();
    // get the structural and thermal parts of the stiffness matrix
    Matrix K_MM( NumIntStructElement::GetLocalStiff() );
    Matrix K_TT( NumIntThermalElement::GetLocalStiff() );

    // coupling terms
    Matrix K_MT( nr_struct_dofs, nr_thermal_dofs );
    K_MT.set_zero();

    Int_Locs intlocs( get_int_locs() ); \ integration point locations

    for ( int ip=1; ip<=get_nr_int_points(); ip++ )
    {
        // integration according to equation 5

        Matrix B( get_B_mat( intlocs.get_coor(ip), &DetJ ) );
        Matrix N( shape->get_N_mat( intlocs.get_coor(ip) ) );

        dV = DetJ*intlocs.get_weight( ip );

        alpha = dynamic_cast<Structural_Point*> ( mat_point[ip-1] ) -> get_expansion_coefficient();

        FPVector D_alpha_dV( dynamic_cast<Structural_Point*> ( mat_point[ip-1] ) -> get_D_times_iso_unit( alpha*dV ) );

        K_MT -= ~B*D_alpha_dV*N;
    }

    Stiffmat.add_SubMat( 1, 1, K_MM );
    Stiffmat.add_SubMat( nr_struct_dofs+1, nr_struct_dofs+1, K_TT );
    Stiffmat.add_SubMat( 1, nr_struct_dofs+1, K_MT );

    return Stiffmat;
}
void ThermoPlastic_Point::set_potential_increm( double Tinc )
{
    newTemp = Temp + Tinc;
    Delta_eps_th = Tinc * get_expansion();
}

void ThermoPlastic_Point::set_strain_increment( const FPVector& strain_increment )
{
    FPVector elpl_strain_incr(strain_increment);
    elpl_strain_incr[1] -= Delta_eps_th;
    switch ( stress_type )
    {
        case pss:
        case psn: elpl_strain_incr[2] -= Delta_eps_th;
                    break;
        case axi:
        case gen: elpl_strain_incr[2] -= Delta_eps_th;
                  elpl_strain_incr[3] -= Delta_eps_th;
    }
    VonMises_Point::set_strain_increment( elpl_strain_incr );
}

REFERENCES


