FeaTure

a Finite Element Analysis Toolkit for Use in a Research Environment

release 0.5

Ton van den Boogaard

January 13, 2006

Contents

Part I

Tutorial

Chapter 1

Introduction

1.1 Objectives of FeaTure

The FeaTure toolkit provides a framework for finite element programming in C++. It is a Finite Element Analysis Toolkit for Use in a Research Environment. Hence it is not a finite element program itself. With the use of FeaTure it should be quite straightforward to program a finite element program in an object oriented way, even without knowing all details of C++. It is the intention of FeaTure that, while building ones own FEM program using the supplied building blocks, it will be relatively easy to exchange e.g. elements, material descriptions or control algorithms with others who use the same building blocks.

To achieve this goal, it is necessary to define the behavior of the basic classes and specially their interfaces. Here a delicate balance is needed between an interface that is general enough to allow many different physical phenomena to be modeled and an interface that uses names that are meaningful enough for an expert in a certain field of application.

1.2 Overview

'All classes are equal, but some classes are more equal than others'. The base classes of the FeaTure framework are almost all abstract classes. Obvious classes in finite element programming are elements and nodes. Together with the concept of degrees of freedom (Dof class), these classes form a module.

System vectors and matrices typically represent a domain. This concept is represented by the Domain class. Usually algorithms at this level, like deriving the linear solution, extracting buckling modes or performing a nonlinear transient analysis, do not interfere directly with individual elements, but only with system vectors and matrices. Therefore the Domain class interacts heavily with the Control class. Often the action taken by the Domain class is performing a loop over all elements, performing action on an element by element basis. The Control and Domain classes form a second module.

In most elements, material behavior plays an important role. The material is represented by two classes, the material type and the material point class. This distinction is made to distinguish between the material parameters that represent a typical material model and the data and algorithms that represent the behavior of one material point. In the latter e.g. state variables can be supported. Defining these classes as a module, separate from the elements, facilitates the development of material classes separately

in a prototyping system, without any knowledge of element behavior.

Additional classes are made available to ease the programming of finite element codes. An indispensable concept is the linear algebra concept of matrices and vectors. In fact they are used already in the base classes of domains, elements, control algorithms and nodes. Several types of matrices and vectors can be distinguished e.g. symmetric and non-symmetric matrices, sparse or dense matrices, vectors in 3-D geometric descriptions and (large) vectors as system vectors. It can be expected that the implementation of these classes highly influences the efficiency of the resulting FEM code. Therefore it is necessary to utilize the optimization options like symmetry and sparsity that the different types of matrices and vectors can give.

The design of FeaTure was influenced by a number of articles, published in the open literature or on internet. The bibliography at the end of this manual presents an overview of the articles I am aware of and that consciously or sub-consciously attributed to the software. An application of FeaTure was published in [10].

1.3 The release system

As all software FeaTure is released every now and then, improving on old and introducing new bugs and other features. Since FeaTure is a framework, the interface to application programs should be quite stable. Two maturity levels are defined in the release system. The classes in level 1 are considered to be mature in the sense that their public member functions are unlikely to change in the future. If a change is needed, a new member function will be defined, leaving the old function unchanged (if possible). FeaTure Classes that are not in level 1 are placed in level 2. A class should be at least one year in level 2 with unchanged public member functions, before it *can* be promoted to level 1. As intended with object oriented programs, the internals of level 1 classes can of course change completely, as long as the public interface remains the same. (This maturity indexing will start when releasing version 1.0)

Together with the distribution of FeaTure other classes or even complete examples of working finite element programs, built on FeaTure may be distributed. They do not belong however to the FeaTure framework as such and can be changed or even missing in a next release.

1.4 Future plans

Before announcing a 1.0 release the following plans should be realized:

1. improve this manual e.g. add more examples;
2. redesign output generating functions (post-processing, restart-data, printed output) explore the use of pointers to member functions;
3. redesign the Dof class, to be more flexible in adding new types of Dofs;
4. make some iterative solvers available e.g. conjugate gradient;
5. set up a naming convention and implement this;
6. re-think the membership types (public, protected, private) of variables and functions and the virtual or non-virtual declaration of functions;

7. ... improve the code based on your comments.

Chapter 2

The basic idea

In this chapter the basic relations between a number of essential classes are described. The main classes, defining the behavior of one element are the ElementBase, Node and Dof classes, relating elements, nodes and degrees of freedom. At system level, the Domain and Control classes have important relations, while the Domain class primarily performs tasks by iterating over all elements or nodes.

For efficient programming, a number of matrix and vector classes are defined. Main classes are the general (dense) Matrix and Vector class, the transformation matrix class TransformationMatrix, the three-dimensional Cartesian vector CartVect and the (sparse) system matrix Sysmat.

All classes and functions, including the matrix and vector classes are defined in the namespace *FeaTure*.

2.1 Elements

The base class **ElementBase** performs a number of actions that are equal for all element types. It relies on some virtual member functions that are to be defined by derived element classes. The base element determines the relation between the local definitions in elements and the global definitions, in particular the potentially different interpretation of local and global directions in the elements and in the nodes. For this purpose a transformation matrix is set up to transform the local stiffness matrix and local force vectors to the global directions and to transform the global (incremental) displacements to the local displacements:

$$\mathbf{K}_{\text{global}} = \mathbf{T}^{\text{T}}\mathbf{K}_{\text{local}}\mathbf{T}$$

$$\mathbf{F}_{\text{global}} = \mathbf{T}^{\text{T}}\mathbf{F}_{\text{local}}$$

$$\mathbf{U}_{\text{local}} = \mathbf{T}\mathbf{U}_{\text{global}}$$

where $\mathbf{T}$ is the transformation matrix. The transformation matrix transforms the inherent local degrees of freedom of the element to the global degrees of freedom as stored in the base element. The transformation matrix is stored efficiently and performs the multiplications efficiently, e.g. it 'knows' if it is a unit diagonal matrix.

For the determination of the global stiffness matrix and the global internal force vector, the ElementBase relies on the functions **GetLocalStiff** and **GetLocalLHS**.

To connect an element to the nodes, the function **add_dofs_to_nodes** calls the virtual function **GetLocalDofs**. This function returns all needed degrees of freedom

for each node. The set of dofs need not be the same for every node. The ElementBase takes care that for every node the necessary degrees of freedom are generated if they are not yet available. If previously processed boundary conditions or elements required degrees of freedom in different directions (not necessarily orthogonal) only additional degrees of freedom are generated, orthogonal to the already existing ones, that are necessary to represent the current degree of freedom completely. In these cases the transformation matrix will have values that differ from 0 and 1.

After that all degrees of freedom have been generated, the function **GreateGlobalDofs** is used to generate a set of pointers to global dofs that define the global dofs to which this element is attached. Since pointers to dofs are used a later renumbering (for efficient solution of the set of equations) does not influence this set.

Isoparametric elements can make use of the **shape** class to derive shape functions and their derivatives and by generating locations of integration points.

2.2 Nodes

A node is an entity of the finite element model that is the owner of degrees of freedom. It has a location in three-dimensional space and for external reference it has a node number.

Two important functions are **get_dof** and **create_dof**. Both functions have *Dof::Type* and *Direction* as arguments. The function **get_dof** returns a set of degrees of freedom and weight factors that together represent the required degree of freedom. If e.g. the required dof is in the direction of the line represented by $y = x$ with $z = 0$ and the local degrees of freedom are defined in the x- and y-direction, then these two dofs will be returned, both with a weight factor of $\frac{1}{2}\sqrt{2}$. The function **create_dof** only creates additional degrees of freedom if the required degree of freedom can not be represented by already available dofs. Newly created dofs are always orthogonal to existing dofs.

New dofs are created if a potential degree of freedom can not be generated within a fraction of ϵ of the existing dofs. The fraction ϵ is 10^{-6} by default and can be changed with the function **set_new_dof_limit**.

The initial location can be derived with the function **get_location** and the current location with **get_current_location**. For the graphical presentation of displacements the location with exaggerated displacements can be derived with the function **draw_location**.

For the use in incremental-iterative algorithms, some special functions are provided in the class **IncNode**.

2.3 The Domain and Control classes

A domain contains all elements, nodes, materials, loads and (other) boundary conditions. The implementation, provided by FeaTure is a template that can use different Element and Node classes. The domain owns the elements, nodes, etc. These objects are deleted if the domain is deleted.

The functions that are available for a domain typically setup system vectors and matrices, by iterating over all elements. These functions are used by the Control object. Since in an incremental algorithm, the total displacements and for an incremental-iterative algorithm also the incremental displacements are stored in the degrees of

freedom, the domain class is the link between the solution vector from the control object and the storage in the degrees of freedom from the nodes.

The **IncDomain** class provides some extra functions for use in an incremental-iterative algorithm.

Several Control classes are provided. The hierarchy of **IncrementalControl** and **IterativeIncrementalControl** can be used as a basis for many specialized algorithms. They do have facilities for intermediate stopping and later continuation of an analysis.

Part II

Reference manual

Chapter 3

Domain classes

In this chapter, the domain classes are described. The Domain class or its derived classes are usually controlled by the Control classes. The domain classes contain the elements and nodes and the control class manipulates the system vectors and matrices.

The domain classes presented here are template classes. They are used to contain model data such as elements, nodes, materials, boundary conditions et cetera. The template generalizes on the element and node classes that are to be stored. If these classes are used, the header file `domain.h` must be included. If classes are derived from the domain classes, also the definitions of the template functions must be included from `domain.cpp`.

Two domain classes are described: the base class **Domain** and the derived class **IncDomain** that is an extension for use in incremental and iterative algorithms.

3.1 Domain

A domain is an object that has knowledge of the (finite element) model. It contains the elements and nodes, boundary conditions, load sets and the material types. A domain is used by a Control object to perform actions on system matrices and vectors.

template <class Element, class Node>
class **FeaTure::Domain**

constructors and destructor

Domain()

Constructor for the Domain object.

~Domain()

Destructor for the Domain object.

creating the model

int **add_element**(*const Element_ptr*)

Adds the element to the domain. The pointer is copied, not the element itself. The element will be deleted upon destruction of the domain.

int **add_node**(*const Node_ptr*)

Adds the node to the domain. The pointer is copied, not the node itself. The node will be deleted upon destruction of the domain.

int **add_load_set**(*LoadSet∗* **load**)

Adds the loadset to the domain. The pointer is copied, not the load set itself. The load set will be deleted upon destruction of the domain.

void **set_title**(*const std::string&* **text**)

Gives this model a title.

void **add_material**(*Material_type∗* **mat**)

Add the material type to the domain. The pointer is copied, not the material type itself. The material type will be deleted upon destruction of the domain.

void **add_geometry**(*Geometry∗* **geom**)

Add the geometry to the domain. The pointer is copied, not the geometry itself. The geometry type will be deleted upon destruction of the domain.

void **make_global_dofs**()

Gives all degrees of freedom a number, ranking them in the system vectors and matrices. This function must be called if all degrees of freedom have been created.

query

Material_type∗ **get_mat_ptr**(*int* **matnr**) *const*

Get the pointer to the material with reference number **matnr**. If not found, return 0.

Geometry∗ **get_geom_ptr**(*int* **geomnr**) *const*

Get the pointer to the geometry with reference number **geomnr**. If not found, return 0.

Node_ptr **get_node_ptr**(*int* **nodenr**) *const*

Get the pointer to the node with reference number **nodenr**. If not found, return 0.

Element_ptr **get_element_ptr**(*int* **elmnr**) *const*

Get the pointer to the element with reference number **elmnr**. If not found, return 0.

int **get_nr_nodes**() *const*

Return the number of nodes currently in the domain.

int **get_nr_elements**() *const*

Return the number of elements currently in the domain.

int **get_nr_free**() *const*

Return the number of (really free) degrees of freedom.

int **get_nr_prescribed**() *const*

Return the number of prescribed degrees of freedom.

int **get_nr_fixed**() *const*

Return the number of fixed (value=0) degrees of freedom.

bool **need_non_symmetric**() *const*

Returns true or false, depending on whether a non-symmetric matrix is (going to be) set up.

preparing an analysis

void **init_system**(*SystemMat&* **system**)

Initializes the system matrix. This can be used to initialize e.g. the storage system of a sparse system matrix. It will iterate over all non-zero elements of the system matrix. It depends on the matrix routine **SystemMat::init** what is done with this data.

analysis control

void **make_LHS**(*SystemVector&* **lhs**) *const*

Generate the internal load vector, split in a part for the really free degrees of freedom (**free**), the degrees of freedom that have a prescribed value (**prescribed**) and the degrees of freedom that will always have a value of zero (**fixed**).

void **make_system**(*SystemMat&* **System**, *Vector&* **rhs**)

Generate the system matrix and any possible contribution of the elements to the right-hand-side vector (equivalent loads).

void **make_mass_system**(*SystemMat&* **System**, *bool* **lumped**)

Generate the system mass matrix. If **lumped** is true, a lumped mass matrix is generated.

void **update_system**(*SystemMat&* **System**, *Vector&* **rhs**)

Update the system matrix with the current values. If necessary, contributions to the right hand side can be added to **rhs**. The right hand side is added to the external load vector.

void **get_load_vector**(*SystemVector&* **load_vector**) *const*

Generates the load vector corresponding to all loads. The vector is split in a part for the really free degrees of freedom (**vec_free**), the degrees of freedom that have a prescribed value (**vec_prescribed**) and the degrees of freedom that will always have a value of zero (**vec_fixed**).

void **set_load_size**(*int* **set**, *double* **size**)

Sets the magnitude of load set **set** to the value **size**.

void **add_load_size**(*int* **set**, *double* **size**)

Increments the magnitude of load set **set** with a value of **size**.

virtual void **print_results**(*ostream&* **out**) *const*

Prints model results for the end-of-current-step state.

Protected member functions

void **make_transmats**()

Initializes the transformation of local to global degrees of freedom. The transformation is usually represented by a transformation matrix. This function must be called after all degrees of freedom have been numbered (by **make_global_dofs**

void **create_element_dofs**()

Runs over all elements to create the necessary degrees of freedom in the connected nodes. This function must be called after the addition of elements and before the analysis is started.

3.2 IncDomain

The incremental domain is derived from **Domain** and adds some functions to use in incremental and iterative solution methods.

template <class Element, class Node>
class **FeaTure::IncDomain**:*public FeaTure::Domain<Element,Node>*

void **finish_increment**()

Should be called if an increment is successfully finished. It will update the start values of the domain (mostly within nodes and elements) with the end-of-step values of the current increment. After this, the end-of-step values are undefined. This leaves open the possibility to swap the start and end values instead of copying them. If start values and increments are stored, the startvalues are updated and the increments are reset to zero. Analysis data should be printed before the call to **finish_increment()** because for some data the incremental values are needed. Restart data can be stored after **finish_increment()** because only start of (new) increment data is needed.

void **iteration_update**()

Iterates over all elements to update the elements and probably the material points to the currently active displacement (increments).

void **add_increments_to_totals**()

Adds the increments of all degree of freedom values to the total values. This should at the end of an increment, e.g. after convergence of an iterative process.

void **reset_increments**()

Sets the incremental values of all degrees of freedom to zero. This is done at the start of an increment.

void **add_to_increments**(*const Vector&* **free**, *const Vector&* **prescribed**)

Add values contained in the vectors **free** and **prescribed** to the respective degrees of freedom, by iterating over all nodes in this domain. The fixed dofs are left unchanged.

void **add_to_increments**(*const Vector&* **free**)

Add values contained in the vector **free** to the respective degrees of freedom, by iterating over all nodes in this domain. The prescribed and fixed dofs are left unchanged.

void **get_incremental_solution_vector**(*Vector&* **increments_free**)

Returns in **increments_free** the current incremental values of the really free degrees of freedom.

void **WriteRestartData**(*ostream&* **restart**) *const*

Writes restart data to the **restart** stream.

void **ReadRestartData**(*istream&* **restart**)

Reads restart data from the **restart** stream.

Chapter 4

Control classes

The hierarchy of control classes control the analysis flow of a finite element analysis. It includes a base class **Control** that can be used e.g. for a linear static analysis, a class **IncrementalControl** for incremental algorithms e.g. for explicit transient algorithms and a class **IncrementalIterativeControl** for incremental–iterative algorithms. The control classes are templated on the type of domain class that is used.

If control classes are used, the header file `control.h` must be included. If classes are derived from the control classes, also the definitions of the template functions must be included from `control.cpp`.

4.1 Control

The Control class is an abstract class that can not be instantiated directly. It is a template class to be able to use different types of Domains. Derived Control classes can in this way use Domain features that are not available for the most basic domains.

template <class DomainT>
class **FeaTure::Control**

virtual **~Control**()

Virtual destructor.

virtual void **solve**(*Domain&* **model**, *ostream&* **out**, *ostream&* **post**)=0

A virtual function, that solves the actual problem by calling the virtual functions **initialize** and **continuate**.

virtual void **initialize**(*Domain&* **model**, *ostream&* **out**)=0

A pure virtual function, to initialize an analysis. A call to this function must precede the call to **continuate**.

virtual void **continuate**(*Domain&* **model**, *ostream&* **out**, *ostream&* **post**, *int* **steps**)=0

A pure virtual function, to perform an already initialized analysis or continue an already started analysis. The **steps** parameter determines the 'amount of progress' that must be made e.g. the number of increments in an incremental analysis. With **steps**=-1, the

analysis will be performed untill the end. A call to **initialize** and **continuate** should perform the required analysis.

void **set_basename**(*const std::string&* **filename**)

Sets the basename for several output files.

void **set_postproc**(*const PostProc&* **post**)

Copies a **PostProc** object into the control to specify what post processing data must be written.

bool **need_post_file**()

A function, to determine whether a post-processing file is required.

4.2 IncrementalControl

template <class DomainT>
IncrementalControl::*public Control<DomainT>*

The IncrementalControl class can not be constructed by its own, but only through derived classes. It defines an incremental procedure but not necessarily an iterative procedure. The IncrementalControl class is a base class for the ExplicitControl and the IncrementalIterativeControl class and provides much of their functionality.

void **set_increments**(*int* **set**, *double* **size**, *int* **steps**)

Sets the number of **steps** for loadset **set** to be performed with magnitude **size**.

void **set_time_increment**(*double* **dtime**)

Sets the time increment to be used for one increment.

float **get_fraction_executed**(*int* **flag**)

Returns the fraction (from 0 to 1) that is executed. The fraction of what is specified by the **flag**. At the moment only the fraction of number of steps is implemented. This can be used e.g. to draw a progress bar.

void **set_Rayleigh_damping**(*double* **alpha**, *double* **beta**)

Sets the parameters alpha and beta for Rayleigh damping (if required).

void **set_max_resize**(*int* **i**)

Sets the maximum number of resizes (on divergence) before stopping.

bool **is_last**()

Returns whether the current increment is the last increment. This can be used if e.g. for the last step post processing data must be written.

void **set_restart**(*const std::string&* **filename**)

Sets the name for the restart file if a restart is necessary.

void **solve**(*Domain&* **model**, *ostream&* **out**, *ostream&* **post**)

Performs the incremental solution process.

virtual bool **continuate**(*Domain&* **model**, *ostream&* **out**, *ostream&* **post**, *int* **steps**)

A function to continuate a possibly stopped process for a number of steps.

virtual void **WriteRestartData**(*Domain&* **model**)

Writes data to a restart file so that the analysis can be continued from the current increment on. The name depends on the name specified by **set_basename** and the current increment number.

virtual void **ReadRestartData**(*Domain&* **model**)

Read data from the restart file specified by **set_restart**. After this the analysis can be continued with **continuate**

void **set_restart_select**(*const StepSelect&* **restart_select**)

Copies a **StepSelect** class into the control to specify the steps for which a restart file must be made.

4.3 IncrementalIterativeControl

IncrementalIterativeControl::*public* **IncrementalControl**

This class is a base class for incremental–iterative procedures. It can not be instantiated by itself.

~IncrementalIterativeControl()

Destructor.

void **set_max_iter**(*int* **maxiter**)

Sets the maximum number of iterations allowed per increment.

void **set_unbalance_crit**(*double* **eps**)

Sets the convergence criterion on the norm of the unbalance vector.

void **set_iteration_method**(*int* **method**)

Sets the iteration method.

4.4 Exception handling

not_converged_error::*public* **feature_error**

Error class, thrown if no convergence was reached in an iterative method.

divergence_error::*public* **not_converged_error**

Error class, thrown if divergence occurred in an iterative method.

Chapter 5

Node classes

A node has a location in 3D space and a node number. It can contain degrees of freedom. Degrees of freedom must be added to a node explicitly. If a node is deleted, also the degrees of freedom of this node are deleted. Usually a number of nodes are 'owned' by a domain (object) and an element (object) contains copies of pointers to the nodes that are connected to that element. The **Node** class is a base class. The derived class **IncNode** is adapted to use in an incremental (and possibly iterative) algorithm.

5.1 Node

The Node class is a concrete class. If this class is used, the header file `nodes.h` must be included.

Node()

Default constructor. The node number is set to '0', the coordinates are not initialized.

Node(*int* **nodenr**)

Node constructor. The node number is set to **nodenr**, the coordinates are not initialized.

Node(*int* **nodenr**, *double* **x**, *double* **y**, *double* **z**)

Constructor of a Node with reference number **nodenr** and initial location (**x,y,z**).

~Node()

Node destructor. Deletes all attached degrees of freedom.

void **set_nr**(*int* **nr**)

Sets the node number to **nr**.

int **get_nr**() *const*

Returns the node number.

void **add_dof**(*Dof_ptr* **plus_dof**)

Adds **plus_dof** unconditionally to the set of degrees of freedom for this node. The pointer to the dof is copied, not the dof itself.

Dof_ptr **get_dof**(*int* **idof**) *const*

Returns the pointer to the degree of freedom that was stored as number **idof** (starting with 0). An exception is thrown if **idof** is larger than the maximum stored dofs - 1.

void **get_dof**(*Dof::Type* **phys_type**, *Direction* **dir**, *dof_sets** **SystDofs**) *const*

Returns a combination of degrees of freedom that represent the type **phys_type** and direction **dir** in **SystDofs**.

void **set_location**(*Location* **loc**)

Initializes or resets the location of the node with **loc**.

void **set_location**(*double* **x**, *double* **y**, *double* **z**)

Initializes or resets the location of the node to (**x,y,z**).

Location **get_location**() *const*

Returns the location of the node as stored. This is probably the initial location, unless the location is updated during the analysis.

void **get_location**(*double&* **x**, *double&* **y**, *double&* **z**) *const*

Returns the location of the node as stored in (**x,y,z**). This is probably the initial location, unless the location is updated during the analysis.

void **get_current_location**(*double&* **x**, *double&* **y**, *double&* **z**) *const*

Returns the current location in (**x,y,z**) i.e. the initial location plus total displacements plus incremental displacements. The values are valid after the moment that the iterative displacements are added to the increments, but before that the increments are added to the total displacements.

void **draw_location**(*double&* **x**, *double&* **y**, *double&* **z**, *double* **magnif**) *const*

Returns in **x,y,z** the initial location plus the total displacements multiplied with **magnif**. This can be used to make an exaggerated displacement plot.

int **create_dof**(*dof_fixed* **fix_type**, *Dof::Type* **phys_type**, *const Direction&* **dir**)

This function takes care that a degree of freedom with type **fix_type** and **Dof::Type** (e.g. prescribed rotation) and direction **dir** for this node exists. It only creates the dof (or only the missing direction) is it not already exists.

void **make_global_dofs**(*int&* **nr_free**, *int&* **nr_fixed**, *int&* **nr_prescribed**)

This function gives the consecutive degrees of freedom for this node the following free dof numbers. The dof numbers are used for the global system vectors and matrices. The arguments **nr_free**, **nr_fixed** and **nr_prescribed** are the current maxima on input and are incremented as needed.

void **add_to_rhs**(*load_type* **type**, *Direction* **dir**, *double* **magn**, *SystemVector&* **rhs**) *const*

Add a contribution to the load vector for the free degrees of freedom that correlate with the **type** and **dir**ection as specified. If necessary e.g. a displacement is decomposed in

contributions to several degrees of freedom. If the load is not fully transferred to one or more dofs a warning is printed.

void **add_to_prescribed**(*Dof::Type* **type**, *Direction* **dir**, *double* **magn**) *const*

As **add_to_rhs** but now for prescribed dofs. The load is added to get correct reaction forces.

CartVector **get_total_displacement**() *const*

Returns the total displacement of the current node as a Cartesian vector. If no displacement dofs are present (0,0,0) is returned.

double **get_temperature**() *const*

Returns the temperature of the current node. If no temperature dof is present it returns 0.

virtual void **WriteRestartData**(*ostream&* **restart**) *const*

Writes restart data for this node. Must be compatible with **ReadRestartData**. Since this function is virtual its implementation can be redefined.

virtual void **ReadRestartData**(*istream&* **restart**)

Reads restart data for this node. Must be compatible with **WriteRestartData**. Since this function is virtual its implementation can be redefined.

5.2 IncNode

The IncNode class is derived from the Node class. This class can be used in incremental and iterative algorithms.

IncNode()

Default constructor. The node number is set to '0', the coordinates are not initialized.

IncNode(*int* **nodenr**)

IncNode constructor. The node number is set to **nodenr**, the coordinates are not initialized.

IncNode(*int* **nodenr**, *double* **x**, *double* **y**, *double* **z**)

Constructor of a IncNode with reference number **nodenr** and initial location (**x,y,z**).

~IncNode()

IncNode destructor. Deletes all attached degrees of freedom.

void **add_increments_to_totals**()

Add the increments of dof values to the total values for this node. The increments are not reset to 0, to be able to print the increments before the next increment starts.

void **reset_increments**()

Resets all incremental dof values for this node to zero.

void **add_to_increments**(*const Vector&* **free**, *const Vector&* **prescribed**)

Iterates over the Dofs for this node and adds the corresponding values in **free** and **prescribed** to the incremental values of the Dofs. Corresponding values means: the value of **free[dofnr-1]** if the dof is really free or **prescribed[dofnr-1]** if the dof is prescribed.

void **add_to_increments**(*const Vector&* **free**)

Iterates over the Dofs for this node and adds the corresponding values in **free** to the incremental values of the Dofs.

void **get_incremental_solution_vector**(*Vector&* **free**) *const*

Returns the incremental solution vector in **free** (also if this vector is not explicitly stored).

CartVector **get_incremental_displacement**() *const*

Returns the incremental displacement of the current node as a Cartesian vector. If no displacement dofs are present (0,0,0) is returned.

double **get_temperature_increment**() *const*

Returns the temperature increment of the current node. If no temperature dof is present it returns 0.

Chapter 6

Element classes

The ElementBase class is an abstract class. If this class is used, the header file `elembase.h` must be included.

Elements are usually 'owned' by a domain (object). An element contains pointers to nodes and pointers to degrees of freedom, but does not own these objects (so they are not deleted if the element is deleted).

More interesting elements are presented in the demo. In the future some of them could migrate to this place.

6.1 ElementBase

This class is a base class for all element types. It is an abstract class and can not be constructed by itself.

virtual **~ElementBase**()

Destructor.

virtual void **iteration_update**()=0

Pure virtual function that is called at the end of an iteration, to let the element update itself with the current iterative values.

virtual void **finish_increment**()=0

Pure virtual function that is called at the end of a (successful) increment, to let the element update itself and get ready for the next increment.

virtual void **AddNodalPostContribution**() *const*

static void **SetTimeIncrement**(*double* **dtime**)

Static function to set the current time increment. Need only be called once if the time increment changes.

static void **SetPostItem**(*PostItem* **post**)

virtual bool **symmetric**() *const*=0

Pure virtual function to specify whether a symmetric (true) or a nonsymmetric (false) stiffness matrix will result. Probably this depends on the material and the algorithm used.

int **get_nr**() *const*

Returns the reference number of this element.

void **add_dofs_to_nodes**() *const*

Adds the necessary degrees of freedom to the nodes of the element. The 'necessary' dofs are taken from the protected pure virtual function **GetLocalDofs**() that must be supplied for every derived concrete element type.

void **CreateGlobalDofs**()

Searches for the dof numbers, corresponding to the local dofs as specified by the pure virtual function **GetLocalDofs**() and initializes the transformation matrix between the local and global dofs.

Vector **GetGlobalLHS**() *const*

Returns the internal load vector in global degrees of freedom. This function fully depends on the pure virtual function **GetLocalLHS** and transforms the local to the global vector.

void **PutLHS_in_Global**(*SystemVector&* **lhs**) *const*

This function adds the contribution of the element internal load vector to the system internal load vectors.

void **InitStiff_in_Global**(*SystemMat∗* **System**) *const*

Initializes the storage for a system matrix **System** for all combinations of element global degrees of freedom.

void **PutStiff_and_RHS_in_Global**(*SystemMat∗* **System**, *Vector&* **rhs**) *const*

Calculates the element system matrix and adds the contribution to the system matrix **System**. If needed also a contribution to the load vector for the really free degrees of freedom is added to **rhs**.

void **PutStiff_in_Global**(*SystemMat∗* **System**, *Vector&* **rhs**) *const*

Calculates the element system matrix and adds the contribution to the system matrix **System**.

void **InitMass_in_Global**(*SystemMat∗* **System**, *bool* **lumped**) *const*

Initializes the storage for a mass matrix **System**. If **lumped** is true, storage for a lumped mass matrix is initialized.

void **PutMass_in_Global**(*SystemMat∗* **System**, *bool* **lumped**) *const*

Calculates the element mass matrix and adds the contribution to the system matrix **System**. If **lumped** is true, a lumped mass matrix is calculated.

void **AddPres2Load**(*int* **surface**, *double* **magnitude**, *SystemVector&* **load**) *const*

Calculates the contribution to the load vectors of a pressure on one of the faces of an element (**surface**) with pressure **magnitude**.

void **WriteRestartData**(*ostream&* **restart**) *const*

Write data for this element to **restart**, to be able to restart from the current increment onwards.

void **ReadRestartData**(*istream&* **restart**)

Read data for this element from **restart**, to be able to restart from the saved increment onwards.

Chapter 7

Dof classes

The Dof classes represent the concept of degrees of freedom. Usually the Dof objects are created and owned by Node objects. The element objects contain copies of pointers to degrees of freedom.

First a helper class for creating sets of Dofs is described, then the base class **Dof**, followed by **incDof** and **itincDof** for use in incremental and iterative incremental algorithms respectively. Finally some global routines are presented that are of special interest for Dof classes.

The classes and functions described in this chapter are declared in the header file `dofs.h`.

7.1 helper class

class **FeaTure::dof_sets**

The dof_sets class contains sets of global degrees of freedom and contributions of that global dof.

dof_sets()

Default constructor.

dof_sets(*const dof_sets&* **in_set**)

Copy constructor.

~dof_sets()

Destructor.

void **add_set**(*Dof_ptr* **p**, *double* **fraction**)

Adds the pointer to Dof **p** with a contribution **fraction** to the set of dofs.

int **nr_of_dofs**()

Returns the number of global dofs, contained in this set.

Dof_ptr **dof**(*int* **i**)

Returns the pointer to the dof, stored as number **i**.

double **contribution**(*int* **i**)

Returns the contribution of the global dof, stored an number **i**.

7.2 Dof

enum **Dof::Type { u, r, t }**

Public enumeration to represent the type of degree of freedom (displacement, rotation, temperature).

enum **Load{ f, m, q }**

Public enumeration to represent the type of nodal load (force, moment, flux).

enum **dof_fixed{ prescribed, fixed, free_dof }**

Public enumeration to represent the value type of the dof. A fixed dof will never get a value different from 0, a prescribed dof will get a prescribed value and is not really free and a free_dof will get a value depending on the response of the system.

If many dofs are fixed, it can be efficient to use the knowledge about this, to avoid many useless multiplications by zero.

Dof()

Default constructor.

void **set_direction**(*Direction* **dir**)

Sets the direction of this degree of freedom to **dir** (only for dofs of vector type, like displacements and rotations).

void **set_Dof::Type**(*Dof::Type* **phys_type**)

Sets the type of the dof, e.g. 'u' for displacement, 'r' for rotation and 't' for temperature.

void **set_dof_fixed**(*dof_fixed* **fix_type**) *Direction* **get_direction**() *const*

Sets the freedom type of the dof to 'free', 'fixed' or 'prescribed'. The difference between fixed and prescribed is that a fixed degree of freedom will always have a value of '0' and a prescribed degree of freedom can have a non-zero value. This difference can be used to facilitate an efficient implementation.

Dof::Type **get_dof_type**() *const*

Returns the type of this dof.

int **get_global_dof**() *const*

Returns the global dof number for reference in system matrix and vector.

void **set_global_dof**(*int* **dofnr**)

Sets the global dof number for reference in system matrix and vector.

void **print**() *const*

Prints the type of dof (free, fixed, prescribed) and the global number. This routine is just for debugging purposes.

bool **is_vector**() *const*

Returns true if this dof represents a vector, false otherwise.

bool **is_scalar**() *const*

Returns true if this dof represents a scalar, false otherwise.

bool **is_free**() *const*

Returns true if this dof is really free, false otherwise.

bool **is_fixed**() *const*

Returns true if this dof is fixed, false otherwise.

bool **is_prescribed**() *const*

Returns true it this dof is prescribed, false otherwise.

7.3 incDof

Degree of freedom class for incremental analysis, derived from **Dof**. The solution value of a **Dof** is considered to be the incremental value.

incDof:*public* **Dof**

incDof()

Default constructor.

void **set_total**(*double* **value**)

Sets the total value of this dof to **value**.

void **add_to_total**(*double* **value**)

Adds **value** to the total value of this dof.

double **get_total**() *const*

Returns the total value of this dof.

7.4 itincDof

Degree of freedom class for iterative incremental analysis. Derived from **incDof**. The solution in the **Dof** part is considered to be the iterative increment.

itincDof:*public* **incDof**

itincDof()

Default constructor.

void **set_increment**(*double* **value**)

Sets the incremental value to **value**.

void **add_to_increment**(*double* **value**)

Adds **value** to the incremental value.

void **add_increment_to_total**()

Adds the incremental value to the total value.

double **get_increment**() *const*

Returns the incremental value.

7.5 Global functions

bool **is_vector_load**(*Dof::Load* **type**)

Returns true if the load type **type** is a vector valued type (e.g. a force), false otherwise.

bool **is_scalar_load**(*Dof::Load* **type**)

Returns true if the load type **type** is a scalar valued type (e.g. a production), false otherwise.

void **print_load_type**(*Dof::Load* **type**, *ostream&* **out**)

Prints a shortname for the load type, e.g. 'f' for force and 'm' for moment.

bool **is_vector_dof**(*Dof::Type* **phys_type**)

Returns true if **phys_typ** is a vector type of dof, e.g. a displacement, false otherwise.

bool **is_scalar_dof**(*Dof::Type* **phys_type**)

Returns true if **phys_typ** is a scalar type of dof, e.g. a temperature, false otherwise.

void **print_dof_type**(*Dof::Type* **type**, *ostream&* **out**)

Prints a shortname for the dof type, e.g. 'u' for displacement and 'r' for rotation.

Dof::Type **conjugate_type**(*load_type* **type**)

Returns the dof type conjugate to the load type **type**, e.g. a displacement to a force or a rotation to a moment.

typedef **itincDof*** **Dof_ptr**

Chapter 8

Material classes

All concrete material classes come in pairs: a Material Type and a Material Point. The Material Type defines the material model with its parameters. An element has only one material type. The Material Point contains the actual implementation of the algorithm and the possibly history dependent (state) variables.

The classes and functions described in this chapter are declared in the header file `material.h`.

8.1 The Material_type and Material_Point classes

8.1.1 Material_type

Material_type()

Default constructor.

Material_type(*int* **mat_nr**)

Constructor for Material_type, sets the reference number to **mat_nr**.

~Material_type()

Destructor.

int **set_density**(*double* **rho**)

Sets the density for this material.

double **get_density**() *const*

Returns the density for this material.

virtual Material_Point∗ **new_Material_Point**(*StressType* **st**)=0

Pure virtual function, returns a pointer to a new Material_Point. In this way material points can be created e.g. in the integration points by calling the Material_type. Every derived class should implement this function in its own way.

virtual bool **symmetric**() *const*=0

Pure virtual function to indicate whether a symmetric or non-symmetric stiffness matrix will be derived.

int **get_nr**() *const*

Returns the reference number of this material.

8.1.2 Material_Point

Abstract class for generation of material points.

virtual **~Material_Point**()

Virtual destructor.

virtual void **finish_increment**()=0

Pure virtual function to indicate that an increment is finished. The material point should prepare itself for the next increment.

virtual void **GetRestartData**(*CollectArray&* **data**) *const*=0

Get all relevant data from array **data** to be able to restart from this data. Should correspond to **PutRestartData**.

virtual void **PutRestartData**(*const CollectArray&* **data**)=0

Assemble all relevant data in an array **data** to be able to restart from the current point on. Should correspond to **GetRestartData**.

static void **SetTimeIncrement**(*double* **dtime**)

8.2 The Structural_type and Structural_Point classes

This couple of classes is intended for structural analysis. They are derived virtual public from Material_type and Material_Point respectively. This makes it possible to derive a thermal class set also and later derive a thermo-mechanical class from both the Structural and Thermal classes.

8.2.1 Structural_type

Structural_type()

Default constructor.

~Structural_type()

Destructor.

virtual bool **NeedExpandStrain**(*StressType* **st**) *const*

If the material model needs an expanded strain, the thickness strain is added to the strain vector, even for plane strain and plane stress situation. The same applies to the stress vector.

8.2.2 Structural_Point

Structural_Point:*public virtual* **Material_Point**

Structural_Point(*StressType* **st**)

protected constructor. The StressType **st** defines the structural behavior of the Point: uniaxial, plane strain, plane stress, axisymmetric or full 3D solid.

virtual Matrix **get_D_mat**() *const* =0

Pure virtual function. Should return the material stiffness matrix $\mathbf{D} = \frac{\partial \boldsymbol{\sigma}}{\partial \boldsymbol{\varepsilon}}$.

Vector **get_D_times_iso_unit**(*double* **eps_iso=1.0**) *const*

virtual void **set_strain_increment**(*const Vector&* **strain_increment**)=0

Sets the strain increment.

virtual Vector **get_stress**() *const* =0

Returns the current stress (at the end of an increment). The size of the stress vector is the minimum needed for the applicable stress state (e.g. size=3 for plane stress and plane strain).

virtual Vector **get_xp_stress**() *const*

Returns the current expanded stress (at the end of an increment). The size of the stress vector is the minimum needed for the applicable stress state (e.g. size=4 for plane stress and plane strain).

virtual double **get_eps_plast_eq**(*bool&* **has_epq**) *const*

Returns the equivalent plastic strain.

void **get_D_and_stress_correction**(*Matrix&* **D**,
Vector& **stress_correction**) *const*

Gets the material stiffness matrix and if necessary a stress correction that is taken into account in the load vector.

virtual double **get_expansion_coefficient**() *const*

Chapter 9

Loads and Boundary Conditions

The classes and functions described in this chapter are declared in the header file `loads.h`.

9.1 Load set class

A load set determines a collection of loads that can be incremented or decremented only as a complete set. A load set is referenced by a set number.

LoadSet()

Default constructor.

LoadSet(*int* **nr**)

Constructor with initialization of number of the set.

~LoadSet()

Destructor.

int **get_nr**() *const*

Returns the reference number of the set.

void **set_nr**(*int* **nr**)

Sets the reference number of this load set to **nr**.

void **add_load**(*Nodal_Loads*∗ **load**)

Adds a **Nodal_Loads** object to this load set.

void **add_load**(*Nodal_Prescribed*∗ **load**)

Adds a **Nodal_Prescribed** object to this load set.

void **add_load**(*Pressure_Loads*∗ **load**)

Adds a **Pressure_Loads** object to this load set.

void **get_load_vector**(*SystemVector&* **load_vec**) *const*

Returns the external load vector, build up from the loads of this load set, taking the current multiplication factor into account.

void **set_load_size**(*double* **size**)

Sets the magnitude of this load set to **size**.

void **add_load_size**(*double* **size**)

Increments the magnitude of this load set by **size**.

void **print**(*ostream&* **out**) *const*

Prints the input to the **out** stream. This function is virtual, so you can override it with your own version.

void **WriteRestartData**(*ostream&* **restart**) *const*

Writes restart data to the **restart** stream.

void **ReadRestartData**(*istream&* **restart**)

Reads restart data from the **restart** stream.

9.2 Nodal loads

A nodal load is a load of the 'force' type, that directly related to the right-hand side of the matrix equations. A **Nodal_Loads** object contains a load on a set of nodes, possibly in a set of directions.

Nodal_Loads()

Default constructor.

Nodal_Loads(*int* **nrloads**, *Load_dir** **lods**, *int* **nrnodes**, *NodePtr** **nods**)

Constructor which prepares for a load on a set of nodes in a number of directions.

~Nodal_Loads()

Destructor.

void **add_load**(*Dof::Load* **type**, *Direction* **dir**, *double* **mag**)

Adds a load of size **mag** of a particular **type** and in a particular direction **dir** to this object.

void **add_load**(*Dof::Load* **type**, *double* **mag**)

Adds a scalar load of size **mag** of a particular **type** to this object.

void **add_node**(*NodePtr* **node**);

Adds a node to the selection of nodes to which this object applies.

int **get_nr_loads**() *const*

Returns the number of loads in this object.

Dof::Load **get_type**(*int* **load_nr**) *const*

Returns the type of load number **nr** in this object.

Direction **get_dir**(*int* **load_nr**) *const*

Returns the direction of load number **nr** in this object.

double **get_magnitude**(*int* **load_nr**) *const*

Returns the magnitude of load number **nr** in this object.

int **get_nr_nodes**() *const*

Returns the number of nodes in this object.

NodePtr **get_node**(*int* **node_nr**) *const*

Returns a pointer to the node, specified by **node_nr** in this object.

void **print**(*std::ostream&* **out**) *const*

Prints an overview of this nodal load object.

void **add_to_rhs**(*SystemVector&* **rhs**) *const*

Adds the contribution of this nodal load object to the right-hand side vector.

9.3 Prescribed Nodal values

A prescribed load is a load of the 'displacement' type, that directly prescribes the value of a primary nodal variable.

Nodal_Prescribed()

Default constructor.

Nodal_Prescribed(*int* **nrloads**, *Load_dir** **lods**, *int* **nrnodes**, *NodePtr** **nods**)

Constructor which prepares for an equal load on a set of nodes in a number of directions.

~Nodal_Prescribed()

Destructor.

void **add_load**(*Dof::Type* **type**, *Direction* **dir**, *double* **mag**)

Adds a prescribed load of size **mag** of a particular **type** and in a particular direction **dir** to this object.

void **add_load**(*Dof::Type* **type**, *double* **mag**)

Adds a scalar prescribed load of size **mag** of a particular **type** to this object.

void **add_node**(*NodePtr* **node**);

Adds a node to the selection of nodes to which this object applies.

int **get_nr_loads**() *const*

Returns the number of loads in this object.

Dof::Type **get_type**(*int* **load_nr**) *const*

Returns the type of load number **nr** in this object.

Direction **get_dir**(*int* **load_nr**) *const*

Returns the direction of load number **nr** in this object.

double **get_magnitude**(*int* **load_nr**) *const*

Returns the magnitude of load number **nr** in this object.

int **get_nr_nodes**() *const*

Returns the number of nodes in this object.

NodePtr **get_node**(*int* **node_nr**) *const*

Returns a pointer to the node, specified by **node_nr** in this object.

void **print**(*std::ostream&* **out**) *const*

Prints an overview of this prescribed nodal load object.

void **add_to_prescribed**(*double* **inc_size**) *const*

Adds the prescribed nodal value times **inc_size** to the nodes, which will pass it on to the respective degrees of freedom.

9.4 Pressure loads

A pressure load describes a force per unit area perpendicular to an element surface.

Pressure_Loads()

Default constructor.

~Pressure_Loads()

Destructor.

void **add_load**(*Element_ptr* **elmptr**, *int* **surface**, *double* **pres**)

Adds a pressure load of size **pres** to the defined **surface** of the element **elmptr** to this object.

void **set_factor**(*double* **fac**)

Sets a multiplication factor with which all pressures will be multiplied, before they are added to the load vector.

void **add_factor**(*double* **fac**)

Increments a multiplication factor with which all pressures will be multiplied, before they are added to the load vector.

void **add_to_vec**(*SystemVector&* **load_vec**) *const*

Adds the contribution of this pressure load object to the load vector.

Chapter 10

Matrix and Vector classes

FeaTure relies on a number of classes that represent the concepts of matrices and vectors. The most general matrix and vectors classes the dense matrix and vector classes **Matrix** and **Vector**. All classes are defined for double precision arithmetic. For special purposes some other matrix and vector classes are defined within FeaTure. For vectors in 3-dimensional space it was found that the general vector has too much overhead. The **CartVector** (Cartesian vector) contains 3 double precision variables and does not use dynamic memory allocation internally. From this base class the **Direction** and **Location** classes are derived. Simple addition, subtraction and scalar multiplication are defined for the Cartesian vectors.

For transformations between local and global coordinates a transformation matrix can be defined. The **TransformationMatrix** class can be very efficient because it uses only a flag in case it is a unit diagonal matrix. The **SystemMat** classes are used for storage of the system matrix.

The dense matrices and vectors use both indices with a 0 offset as with a 1 offset. Here 0 offset means that the first item is indexed 0 and the last item n-1 (if n is the size). This is the traditional C-style indexing and is used when square brackets are used e.g. `A[0][4]`, indicates the matrix element that is usually named A_{15} in mathematics. Offset 1 is the traditional Fortran-style (and linear algebra style). The first item is 1 and the last item is n. This indexing can be used with parenthesis e.g. `A(1,5)` for a matrix and `b(3)` for a vector.

10.1 Vector

The Vector class is declared in `ftmatrix.h`

class **FeaTure::Vector**

Vector()

Constructs an empty vector.

Vector(*size_t* **N**)

Constructs an uninitialized vector of size **N**.

Vector(*size_t* **N**, *double* **val**)

Constructs a vector of size **N**, filled with values **val**.

Vector(*const Vector&* **a**)

Copy constructor.

explicit **Vector**(*const Matrix&* **A**)

Constructor that copies a matrix **A**, that must have only 1 column.

~Vector()

Destructor.

Vector& **operator=**(*const Vector&* **a**)

Assignment operator.

size_t **size**() *const*

Returns the number of rows.

double& **operator[]**(*size_t* **i**)

Reference to the i-th element (zero-based).

const double& **operator[]**(*size_t* **i**) *const*

Const reference to the i-th element (zero-based).

double& **operator()**(*size_t* **i**)

Reference to the i-th element (one-based).

const double& **operator()**(*size_t* **i**) *const*

Const reference to the i-th element (one-based).

void **clear**()

(Re)sets the values of this vector to zero.

void **resize**(*size_t* **N**)

(Re)sets the size of this vector to **N**.

Vector **add_SubVec**(*int* **row**, *const Vector&* **SubVec**)

Adds sub-vector **SubVec** starting at position **row** (1-based).

double **norm**() *const*

Returns the Euclidean norm of this vector:

$$\|v\| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

void **print_row**(*std::ostream&* **os**) *const*

Prints this vector on stream **os** as a row.

Vector& **operator**$*$ =(*const double&* **a**)

Multiplies this vector with scalar **a**.

Vector& **operator**+ =(*const Vector&* **a**)

Adds **a** to this vector.

Vector& **operator**− =(*const Vector&* **a**)

Subtracts **a** from this vector.

10.2 Matrix

The matrix class is declared in `ftmatrix.h`.

class **FeaTure::Matrix**

Matrix()

Default constructor for an empty matrix.

Matrix(*size_t* **M**, *size_t* **N**)

Constructs an uninitialized (M,N) matrix.

Matrix(*int* **M**, *int* **N**, *double* **val**)

Constructs an (M,N) matrix of which all elements have the value **val**.

Matrix(*const Matrix&* **A**)

Copy constructor; initializes this matrix with a Matrix **A**.

Matrix(*const Vector&* **a**)

Initializes this matrix with vector **a**. The size of the matrix will become (n,1), where n is the size of the vector.

~Matrix()

Destructor.

Matrix& **operator=**(*const Matrix&* **a**)

Assignment operator.

size_t **size1**() *const*

Returns the number of rows.

size_t **size2**() *const*

Returns the number of columns.

double$*$ **operator[]**(*size_t* **i**)

Return a pointer to the first element of row **i** (zero-based), such that A[i][j] will refer to A_{ij}. In this case index j is not checked to be in the range of the matrix columns.

*const double** **operator[]**(*size_t* **i**) *const*

Return a const pointer to the first element of row **i** (zero-based), such that A[i][j] will refer to A_{ij}. In this case index j is not checked to be in the range of the matrix columns.

double& **operator()**(*size_t* **i**,*size_t* **j**)

Reference to the i, j-th element (one-based).

const double& **operator()**(*size_t* **i**, *size_t* **j**) *const*

Const reference to the i, j-th element (one-based).

void **clear**()

Fills this matrix with zeros.

void **resize**(*size_t* **M**, *size_t* **N**)

(Re)sets the size of this matrix to (**M**,**N**).

Matrix **without_rowcol**(*size_t* **rowcol**) *const*

Returns a matrix that is identical to this matrix, but without the row and column number **rowcol** (1-based).

Matrix **add_SubMat**(*size_t* **row**, *size_t* **col**, *const Matrix&* **SubMat**)

Adds **SubMat** as a sub-matrix to this matrix, starting at position (**row**,**col**) (1-based).

double **determinant**()

Returns the determinant for this matrix if it is a square matrix of maximum 3 rows and columns. If the matrix is not square or too large it throws an error. This routine should be generalized in the future, without affecting the efficiency for small matrices.

Matrix **inv**() *const*

Returns the inverse of this matrix if it is a square matrix, otherwise throws an error. The matrix is inverted using the method described at page 171 of 'Introduction to numerical analysis' by J. Stoer and R. Bulirsch. For large matrices, this is a costly operation.

Matrix& **operator**$*$ =(*const double&* **a**)

Multiplies this matrix with scalar **a**.

Matrix& **operator**+ =(*const Matrix&* **A**)

Adds **A** to this matrix.

Matrix& **operator**− =(*const Matrix&* **A**)

Subtracts **A** from this matrix.

std::ostream& **operator**<<(*std::ostream&* **s**, *const Vector&* **A**)

Puts vector **A** on the output stream **s** as a column. The **setw()** manipulator can be used to set the width for each vector-element.

std::ostream& **operator<<**(*std::ostream&* **s**, *const Matrix&* **A**)

Puts matrix **A** on the output stream **s**. The **setw()** manipulator can be used to set the width for each matrix-element.

10.3 Additional functions for full matrices and vectors

10.3.1 Matrix and vector addition, subtraction and multiplication

The following functions between Matrix and Vector objects are defined. They are declared in `ftmatrix.h`.

double **inner_product**(*const Vector&* **A**, *const Vector&* **B**, *double* **v=0**)

Returns the inner product added to the initial value $v + \mathbf{A}^{\mathrm{T}}\mathbf{B}$.

Matrix **mult**(*const Matrix&* **A**, *const Matrix&* **B**)

Returns $\mathbf{R} = \mathbf{AB}$.

Vector **mult**(*const Matrix&* **A**, *const Vector&* **B**)

Returns $\mathbf{r} = \mathbf{AB}$.

Matrix **tmult**(*const Matrix&* **A**, *const Matrix&* **B**)

Returns $\mathbf{R} = \mathbf{A}^{\mathrm{T}}\mathbf{B}$.

Vector **tmult**(*const Matrix&* **A**, *const Vector&* **B**)

Returns $\mathbf{r} = \mathbf{A}^{\mathrm{T}}\mathbf{B}$.

Matrix **tmult**(*const Vector&* **A**, *const Matrix&* **B**)

Returns $\mathbf{R} = \mathbf{A}^{\mathrm{T}}\mathbf{B}$.

void **Add_sAtBA**(*Matrix&* **C**, *double* **s**, *const Matrix&* **A**, *const Matrix&* **B**)

Optimized function for the triple matrix product summation $\mathbf{C} = \mathbf{C} + s\mathbf{A}^{\mathrm{T}}\mathbf{BA}$. Typically used for numerical integration of the stiffness matrix.

Matrix **operator~** (*const Vector&* **A**)

Returns the transpose of vector **A** as a Matrix (row-vector).

Matrix **operator~** (*const Matrix&* **A**)

Returns the transpose of matrix **A**: result = $\mathbf{A}^{\mathrm{T}}$.

Vector **operator+**(*const Vector&* **A**, *const Vector&* **B**)

Return the result of $\mathbf{A} + \mathbf{B}$.

Matrix **operator+**(*const Matrix&* **A**, *const Matrix&* **B**)

Return the result of $\mathbf{A} + \mathbf{B}$.

Vector **operator+**(*const Matrix&* **A**, *const Vector&* **B**)

Returns the result of $\mathbf{A} + \mathbf{B}$ if the matrix $\mathbf{A}$ has same size as vector $\mathbf{B}$.

Vector **operator**+(*const Vector&* $\mathbf{A}$, *const Matrix&* $\mathbf{B}$)

Returns the result of $\mathbf{A} + \mathbf{B}$ if the matrix $\mathbf{B}$ has same size as vector $\mathbf{A}$.

Vector **operator**−(*const Vector&* $\mathbf{A}$, *const Vector&* $\mathbf{B}$)

Return the result of $\mathbf{A} - \mathbf{B}$.

Matrix **operator**−(*const Matrix&* $\mathbf{A}$, *const Matrix&* $\mathbf{B}$)

Return the result of $\mathbf{A} - \mathbf{B}$.

Vector **operator**−(*const Matrix&* $\mathbf{A}$, *const Vector&* $\mathbf{B}$)

Return the result of $\mathbf{A} - \mathbf{B}$ if the matrix $\mathbf{A}$ has the same size as vector $\mathbf{B}$.

Vector **operator**−(*const Vector&* $\mathbf{A}$, *const Matrix&* $\mathbf{B}$)

Return the result of $\mathbf{A} - \mathbf{B}$ if the matrix $\mathbf{B}$ has the same size as vector $\mathbf{A}$.

Vector **operator**∗(*const Vector&* $\mathbf{A}$, *const double&* $\mathbf{B}$)

Returns vector $\mathbf{A} * B$ (B is a scalar).

Vector **operator**∗(*const double&* $\mathbf{A}$, *const Vector&* $\mathbf{B}$)

Returns vector $A * \mathbf{B}$ (A is a scalar).

Matrix **operator**∗(*const Matrix&* $\mathbf{A}$, *const double&* $\mathbf{B}$)

Returns $\mathbf{A} * B$ (B is a scalar).

Matrix **operator**∗(*const double&* $\mathbf{A}$, *const Matrix&* $\mathbf{B}$)

Returns matrix $A * \mathbf{B}$ (A is a scalar).

Matrix **operator**∗(*const Matrix&* $\mathbf{A}$, *const Matrix&* $\mathbf{B}$)

Returns matrix $\mathbf{A} * \mathbf{B}$.

Vector **operator**∗(*const Matrix&* $\mathbf{A}$, *const Vector&* $\mathbf{B}$)

Returns vector $\mathbf{A} * \mathbf{B}$.

Matrix **operator**∗(*const Vector&* $\mathbf{A}$, *const Matrix&* $\mathbf{B}$)

Returns the result of $\mathbf{A} * \mathbf{B}$ only if matrix $\mathbf{B}$ has only 1 row (error otherwise).

10.3.2 Triangular decompositions and multiplications

void **LDLtDecomposition**(*Matrix∗* **Ap**)

Calculates the $\mathbf{LDL}^{\mathrm{T}}$ decomposition of the symmetric matrix $\mathbf{A}$. The lower left part of $A(i,j)$ with $j < i$ is substituted by $L(i,j)$. The diagonal $A(i,i)$ is substituted by the diagonal $D(i,i)$. The upper right part of $A(i,j)$ with $j > i$ is left unchanged.

void **LDLtSubstitution**(*const Matrix&* **LDLt**, *Vector∗* **xp**)

Solve $\mathbf{x}$ from $\mathbf{LDL}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$, were $\mathbf{b}$ is initially stored in $\mathbf{x}$. The set of equations is efficiently solved as:

$$\mathbf{Ly} = \mathbf{b}$$
$$\mathbf{Dz} = \mathbf{y}$$
$$\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{z}$$

void **CholeskyDecomposition**(*Matrix∗* **Ap**)

Calculates the Cholesky decomposition $\mathbf{A} = \mathbf{LL}^{\mathrm{T}}$ of the symmetric matrix $\mathbf{A}$. The lower left part of $A(i,j)$ with $j \leq i$ is substituted by $L(i,j)$. The upper right part of $A(i,j)$ with $j > i$ is left unchanged.

void **CholeskySubstitution**(*const Matrix&* **LDLt**, *Vector∗* **xp**)

Solve $\mathbf{x}$ from $\mathbf{LL}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$, were $\mathbf{b}$ is initially stored in $\mathbf{x}$. The matrix $\mathbf{L}$ is a lower left triangular matrix and L_{ij} is not used for $i < j$.

$$\mathbf{Ly} = \mathbf{b}$$
$$\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y}$$

void **PostMultLowerTInv**(*Matrix∗* **Ap**, *const Matrix&* **L**)

Calculate $\mathbf{R} = \mathbf{AL}^{-\mathrm{T}}$ where $\mathbf{L}$ is a lower left triangular matrix and return the result in **Ap**. The part L_{ij} with $i < j$ is not used.

void **PreMultLowerInv**(*const Matrix&* **L**, *Matrix∗* **Ap**)

Calculate $\mathbf{R} = \mathbf{L}^{-1}\mathbf{A}$ where $\mathbf{L}$ is a lower left triangular matrix and return the result in **Ap**. The part L_{ij} with $i < j$ is not used.

void **PreMultLowerTInv**(*const Matrix&* **L**, *Matrix∗* **Ap**)

Calculate $\mathbf{R} = \mathbf{L}^{-\mathrm{T}}\mathbf{A}$ where $\mathbf{L}$ is a lower left triangular matrix and return the result in **Ap**. The part L_{ij} with $i < j$ is not used.

Matrix **PreMultLower**(*const Matrix&* **L**, *const Matrix&* **A**)

Return $\mathbf{R} = \mathbf{LA}$ where $\mathbf{L}$ is a lower left triangular matrix. The part L_{ij} with $i < j$ is not used.

Matrix **PostMultLowerT**(*const Matrix&* **A**, *const Matrix&* **L**)

Return $\mathbf{R} = \mathbf{AL}^{\mathrm{T}}$ where $\mathbf{L}$ is a lower left triangular matrix. The part L_{ij} with $i < j$ is not used.

Matrix **PreMultLowerT**(*const Matrix&* **L**, *const Matrix&* **A**)

Return $\mathbf{R} = \mathbf{L}^{\mathrm{T}}\mathbf{A}$ where $\mathbf{L}$ is a lower left triangular matrix. The part L_{ij} with $i < j$ is not used.

10.3.3 Eigenvalue problems

void **householder**(*Matrix*∗ **A**, *Matrix*∗ **V=0**)

Transforms the symmetric matrix **A** into a tridiagonal matrix. If **V** equals 0, the transformation matrix is not calculated, else **V** on exit contains the orthogonal transformation such that the tridiagonal matrix **T** is obtained from: $\mathbf{T} = \mathbf{V}^{\mathrm{T}}\mathbf{A}\mathbf{V}$.

void **symmetricQR**(*Matrix*∗ **Tp**, *Matrix*∗ **V=0**)

Transforms the symmetric tridiagonal matrix **Tp** into a diagonal matrix. If **V** equals 0, the transformation matrix is not calculated, else **V** on exit contains the orthogonal transformation such that the diagonal matrix **D** is obtained from: $\mathbf{D} = \mathbf{V}^{\mathrm{T}}\mathbf{A}\mathbf{V}$. If **V** already contained a matrix from a previous transformation e.g. a Householder transformation, the matrix **V** contains the combined transformation on output. The columns of **V** contain the eigenvectors corresponding to the eigenvalues in the same column of **D**, which is stored in **Tp** on exit.

int **SturmSequence**(*const Matrix&* **A**, *double* **shift=0**)

Calculates the number of sign changes in the calculation of subdeterminants of matrix $\mathbf{A} - \mu\mathbf{I}$, where μ is the eigenvalue shift. The Sturm sequence number corresponds to the number of eigenvalues less than μ.

void **GenEigen**(*Matrix*∗ **Ap**, *Matrix*∗ **Bp**, *Vector*∗ **lambda**, *Matrix*∗ **Vp=0**)

Determines the eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ (if $\mathbf{Vp} \neq 0$) of the generalized eigenproblem:

$$(\mathbf{A} - \lambda\mathbf{B})\mathbf{v} = \mathbf{0}$$

The eigenvectors are stored as columns in matrix **Vp**.

void **InverseIteration**(*const Matrix&* **A**, *Vector** **xp**, *double** **lambda**)

Perform inverse iteration on matrix **A** with starting vector **xp**. The result is the lowest eigenvalue **lambda** and corresponding eigenvector **xp**.

10.4 Cartesian vectors

10.4.1 CartVector

The CartVector class is declared in the header file `cartvect.h`. This class is an efficient implementation of Cartesian vectors in 3-dimensional space. It is especially useful as base class of the **Direction** and **Location** class.

class **CartVector**

CartVector()

Default constructor.

CartVector(*const CartVector&* **vec**)

Copy constructor.

CartVector(*double* **x**, *double* **y**, *double* **z**)

Constructor, initializes the vector with (**x,y,z**).

double& **operator[]**(*int* **i**)
Zero-based reference to coordinate i.

const double& **operator[]**(*int* **i**) *const*
Zero-based const reference to coordinate i.

double& **operator()**(*int* **i**)
One-based reference to coordinate i.

const double& **operator()**(*int* **i**) *const*
One-based const reference to coordinate i.

void **print**() *const*
Prints the vector.

CartVector **add**(*const CartVector&* **src2**) *const*
The addition of **src2** and this vector is returned.

CartVector **selfadd**(*const CartVector&* **src2**)
The vector **src2** is added to this vector and this vector is returned.

CartVector **sub**(*const CartVector&* **src2**) *const*
Returns this vector minus **src2**.

CartVector **selfsub**(*const CartVector&* **src2**)
Subtracts **src2** from this vector and returns this vector.

CartVector **mul**(*double* **val**) *const*
Returns the multiplication of **val** and this vector.

CartVector **selfmult**(*double* **scalar**)
Multiplies this vector by **scalar** and returns it.

void **getValues**(*double&* **x**, *double&* **y**, *double&* **z**)
Returns the coordinates of this vector in **x,y,z**.

double **dot**(*const CartVector&* **vec2**) *const*
Returns the dot product of this vector and **vec2**.

double **normsq**() *const*
Returns the square of the Euclidean norm of this vector.

double **norm**() *const*
Returns the Euclidean norm of this vector.

friend CartVector **outer**(*const CartVector&* **a**, *const CartVector&* **b**)

Returns the outer product **a** × **b** (not a member function).

The following friend functions perform the obvious operations by calling the member functions described above.

friend ostream& **operator**<<(*ostream&* **os**, *const CartVector&* **vec**)

friend CartVector **operator+**(*const CartVector&* **a**, *const CartVector&* **b**)

friend CartVector **operator+=**(*CartVector&* **a**, *const CartVector&* **b**)

friend CartVector **operator-**(*const CartVector&* **a**, *const CartVector&* **b**)

friend CartVector **operator−=**(*CartVector&* **a**, *const CartVector&* **b**)

friend CartVector **operator∗**(*const CartVector&* **a**, *double* **b**)

friend CartVector **operator∗**(*double* **b**, *const CartVector&* **a**)

friend double **operator∗**(*const CartVector&* **a**, *const CartVector&* **b**)

10.4.2 Direction

The Direction class is declared in the header file `cartvect.h`. This class is derived from the **CartVector** class and always contains a normalized vector, representing a direction. Normalized, means that the Euclidean norm of the vector is 1.

class **Direction**:*public* **CartVector**

Direction()

Default constructor. This one leaves the data uninitialized and should be avoided if possible. If used, the vector should be initialized shortly after creation.

Direction(*const Direction&* **dir**)

Copy constructor.

Direction(*const CartVector&* **vect**)

Constructor that initializes with the normalized version of **vect**.

void **set_dir**(*double* **x**, *double* **y**, *double* **z**)

Sets the direction to (x,y,z) and normalizes the vector if not yet done.

void **get_dir**(*double&* **x**, *double&* **y**, *double&* **z**) *const*

Returns the direction in **x,y,z**.

void **print**() *const*

Prints the direction to standard output.

10.4.3 Location

The Location class is declared in the header file `cartvect.h`.

class **Location**:*public* **CartVector**

Location(*double* **x=0**, *double* **y=0**, *double* **z=0**)

Initializes the location with the vector **x,y,z**.

Location(*const CartVector&* **vect**)

Initializes the location with the vector **vect**.

void **set_location**(*double* **x**, *double* **y**, *double* **z**)

Sets the location to **x,y,z**.

void **get_location**(*double&* **x**, *double&* **y**, *double&* **z**) *const*

Returns the location to **x,y,z**.

void **print**() *const*

Prints the location to the standard output.

10.5 Transformation Matrix

The **TransformationMatrix** class is a separate matrix class that can perform the actions that are usually required for coordinate transformations (pre-multiplying with the transpose and post-multiplying with full matrices and multiplication with vectors). It can perform these actions efficiently. Often the transformation matrix is just a unit diagonal matrix or a permutation matrix. In these cases multiplication is trivial (unit matrix) or concerns just a replacement of rows and/or columns (permutation). Even in the general case, only the non-zeros are stored and used in multiplications.

The TransformationMatrix class is declared in `transmat.h`.

class **FeaTure::TransformationMatrix**

TransformationMatrix()

Default constructor.

~TransformationMatrix()

Destructor.

void **setUnit**(*int* **dim**)

Sets this matrix to be a unit diagonal matrix of dimension **dim**. No dynamic memory allocation will take place and multiplication just 'pipes' through the vector or matrix that is multiplied. A unit diagonal matrix can also be deduced from a sparse matrix (see **setSparse()**).

void **setPermutation**(*int* **dimen1**, *int* **dimen2**, *int∗* **perm**)

Sets this matrix to be a permutation matrix of size (**dimen1,dimen2**). A permutation matrix only contain the values 0 and one value of 1 in every row. The integer array **perm** contains for each row the column number that has the value 1. A permutation matrix can also be deduced from a sparse matrix (see **setSparse()**).

void **setSparse**(*int* **dimen1**, *int* **dimen2**, *int* **nz**)

Sets the matrix (initially) to be a general sparse matrix of dimension (**dimen1,dimen2**) with **nz** number of nonzeros. This number of nonzeros is needed to initialize the storage efficiently and it will be used to evaluate the matrix after **nz** numbers have been added by **put(i,j,val)**. If the matrix appears to be a unit matrix or a permutation matrix during evaluation, the storage scheme and functions are optimized for that.

void **put**(*int* **i**, *int* **j**, *double* **val**)

Put the value **val** at location (**i,j**) (1-indexed). Only put non-zero values in the matrix. Non-initialized values are automatically recognized as zero.

double **get**(*int* **i**, *int* **j**) *const*

Returns the value of T_{ij} regardless of the storage scheme (1-indexed).

Vector<double> **mult**(*const Vector<double>&* **v**) *const*

Returns the multiplication of this matrix by vector **v**: $\mathbf{r} = \mathbf{T}\mathbf{v}$.

Vector<double> **tmult**(*const Vector<double>&* **v**) *const*

Returns the multiplication of the transpose of this matrix by vector **v** $\mathbf{r} = \mathbf{T}^{\mathrm{T}}\mathbf{v}$.

Matrix<double> **preTmult_postMult**(*const Matrix<double>&* **M**) *const*

Returns the matrix result of **M** pre-multiplied with the transposed of this matrix and post-multiplied with this matrix $\mathbf{R} = \mathbf{T}^{\mathrm{T}}\mathbf{M}\mathbf{T}$.

Subscript **num_rows**() *const*

Returns the number of rows in this matrix.

Subscript **num_cols**() *const*

Returns the number of columns in this matrix.

Global functions

ostream& **operator**<<(*ostream&* **s**, *const TransformationMatrix&* **A**);

Puts the transformation matrix **A** on the output stream **s**.

Vector<double> **operator***(*const TransformationMatrix&* **A**,
const Vector<double>& **B**)

Multiplies the transformation matrix **A** with vector **B**.

10.6 System Matrix classes

The classes in this section are specially designed to work with system matrices efficiently. The storage schemes and solution algorithms take the sparsity of the matrix into account.

10.6.1 SystemMat

The SystemMat class is declared in `sysmat.h`.

virtual **~SystemMat**()

Destructor.

virtual void **init**(*int* **row**, *int* **col**)=0

Pure virtual function that reserves a storage place for matrix element A(row,col). This function can be used by derived classes and should be called for all non-zero elements before the elements are really added to the matrix.

virtual void **set_zero**()=0

Pure virtual function that sets all elements of the matrix to zero.

virtual void **add**(*int* **row**, *int* **col**, *double* **value**)=0

Pure virtual function that increments the matrix element (row,col) with **value**. Do not forget to use set_zero() before starting to fill the matrix.

virtual void **print**(*char** **name**) *const*=0

Pure virtual function to print this matrix with **name** as header.

virtual void **solve**(*const Vector&* **rhs**, *Vector&* **x**)=0

Pure virtual function to solve the linear set of equations given by $\mathbf{Ax} = \mathbf{rhs}$ where **A** is this system matrix.

10.6.2 SparseSysMat

The SparseSysMat class is declared in `sysmat.h`. This matrix is derived from a compressed sparse row storage defined in `sprsmat.h`. Two multiplications are defined for the base class with a *Vector* and with a *Matrix*:

Vector **operator***(*const CR_Matrix&* **A**, *const Vector&* **x**)

Defines the multiplication of the compressed row matrix with a vector.

Matrix **operator***(*const CR_Matrix&* **A**, *const Matrix&* **X**)

Defines the multiplication of the compressed row matrix with a matrix.

class **FeaTure::SparseSysMat**:*public FeaTure::SystemMat, public FeaTure::CR_Matrix*

The SparseSysMat class derived publicly from SystemMat and CR_Matrix.

SparseSysMat()

Default constructor.

SparseSysMat(*Subscript* **N**, *bool* **symm=false**)

Constructor for a (square) system matrix of size **N**. If **symm** is true then only the lower triangular part has to be stored.

SparseSysMat(*const SparseSysMat&* **S**)

Copy constructor.

~SparseSysMat()

Destructor.

void **init**(*int* **row**, *int* **col**)

This function only checks whether (**row,col**) fits within this matrix (1-indexed). The matrix-elements can be added at will.

void **set_zero**()

Sets all the matrix elements to zero.

void **add**(*int* **row**, *int* **col**, *double* **value**)

Increments matrix element (**row,col**) with **value**.

void **add**(*const SparseSysMat&* **A**, *double* **s=1.0**)

Adds to this matrix the matrix **A** multiplied by **s**.

$$\mathbf{M} = \mathbf{M} + s\mathbf{A}$$

void **print**(*char** **name**) *const*

Print this matrix with header **name**.

void **solve**(*const Vector&* **rhs**, *Vector&* **x**)

Solves the set of equations $\mathbf{Ax} = \mathbf{rhs}$. At this moment a direct skyline solver is used. In the future more solvers should be made available.

A non-member function **subspace_iteration** exists to derive the lowest eigenvalues and corresponding eigenvectors of a symmetric **SparseSysMat** matrix.

void **subspace_iteration**(*SparseSysMat&* **A**, *const SparseSysMat&* **B**, *Vector** **lambda**, *Matrix** **V**, *bool* **init=true**)

Solves the generalized eigen problem

$$(\mathbf{A} - \lambda\mathbf{B})\mathbf{v} = \mathbf{0}$$

The number of columns in **V** defines the dimension of the subspace, the number of rows should be equal to the dimension of **A** and **B**. If **init** is true, the first starting vector will be initialized with the diagonal elements of **B** and the other vectors as unit vectors with value +1 on the row (i) with the smallest ratios of the diagonal elements of **A** and **B** (A_{ii}/B_{ii}). If **init** is false, **V** should be initialized with independent starting vectors. On exit, **lambda** contains the eigenvalues and **V** the eigenvectors.

10.6.3 SkySymMat

The SkySymMat class is defined in `sysmat.h`.

class **FeaTure::SkySysMat**:*public FeaTure::SystemMat*

This class uses a skyline storage format for symmetric matrices, storing only the lower left triangular part.

SkySymMat()

Default constructor.

SkySymMat(*int* **size**)

Constructs a symmetric skyline matrix of size **size**.

SkySymMat(*int* **size**, *int*∗ **p**)

Constructs a symmetric skyline matrix of size **size** and for each row a first non-zero element as indicated by **p** (0-indexed).

SkySymMat(*const SkySymMat&* **S**)

Copy constructor.

~SkySymMat()

Destructor.

void **init**(*int* **row**, *int* **col**)

Initializes element (**row,col**) in the storage scheme. This function generates the storage for the row-elements after (and including) the first non-zero element in the row. This storage scheme must be available before the first items are really stored.

void **set_zero**()

(Re)sets the matrix to zero.

void **add**(*int* **row**, *int* **col**, *double* **value**)

Adds **value** to the matrix element (**row,col**). This function is ignored if **row**<**col**.

void **print**(*char*∗ **name**) *const*

Print this matrix with **name** as header.

void **solve**(*const Vector&* **rhs**, *Vector&* **x**)

Solve the linear set of equations given by $\mathbf{Ax} = \mathbf{rhs}$ by a Cholesky decomposition.

10.6.4 DiagonalSysMat

The DiagonalSysMat class is declared in `sysmat.h`.

class **FeaTure::DiagonalSysMat**:*public FeaTure::SystemMat*

This class can only store diagonal elements of a matrix. It can be used advantageously for e.g. a lumped mass matrix.

DiagonalSysMat()

Default constructor.

DiagonalSysMat(*int* **size**)

Constructs a diagonal matrix of given **size**.

DiagonalSysMat(*const DiagonalSysMat&* **D**)

Copy constructor.

~DiagonalSysMat()

Destructor.

void **init**(*int* **row**, *int* **col**)

Checks whether (**row,col**) is legal for this matrix, i.e. **row**=**col** and $1 \leq$ row $\leq$ size (1-indexed).

void **set_zero**()

(Re)sets this matrix to zero.

void **add**(*int* **row**, *int* **col**, *double* **value**)

Adds **value** to matrix element (**row,col**), the element is checked to be in the legal range.

void **print**(*char** **name**) *const*

Prints this matrix with **name** as header.

void **solve**(*const Vector&* **rhs**, *Vector&* **x**)

Solve the linear set of equations given by $\mathbf{Ax} = \mathbf{rhs}$.

void **selfmult**(*double* **s**)

Multiplies this matrix with the scalar s.

void **mult**(*const Vector&* **x**, *Vector** **b**) *const*

Returns the multiplication of itself with vector **x** in **b**:

$$\mathbf{Ax} = \mathbf{b}$$

10.6.5 SkyNonSymMat

The SkyNonSymMat class is declared in `nonsym.h`.

class **FeaTure::SkyNonSymMat**:*public FeaTure::SystemMat*

This class uses a skyline storage scheme for nonsymmetric matrices with a symmetric storage structure i.e. if A_{ij} is stored, also A_{ji} is stored, but they do not have to have the same value.

SkyNonSymMat()

Default constructor.

SkyNonSymMat(*int* **n**)

Constructs a sparse square matrix of size **n**.

~SkyNonSymMat()

Destructor.

void **init**(*int* **row**, *int* **col**)

Initializes the matrix element (**row,col**) (1-indexed). Space is reserved for the row starting at the first non-zero element to the diagonal and the corresponding column elements. This function must be called for at least all rows with their first non-zero element, before the matrix is really stored.

void **set_zero**()

(Re)sets this matrix to zero.

void **add**(*int* **row**, *int* **col**, *double* **value**)

Adds **value** to the element (**row,col**) of this matrix (1-indexed).

void **print**(*char∗* **name**) *const*

Print this matrix with the header **name**.

void **solve**(*const Vector&* **rhs**, *Vector&* **x**)

Solve the linear set of equations given by $\mathbf{Ax} = \mathbf{rhs}$.

10.7 System Vector

A vector that contains three *Vector* subvectors that refer to the free, fixed and prescribed degrees of freedom.

SystemVector()

Default constructor.

SystemVector(*size_t* **nr_free**, *size_t* **nr_prescribed**, *size_t* **nr_fixed**)

Constructor that initializes the number of free, prescribed and fixed degrees of freedom.

SystemVector(*const SystemVector&* **A**)

Copy constructor.

~SystemVector()()

Destructor.

void **resize**(*size_t* **nr_free**, *size_t* **nr_prescribed**, *size_t* **nr_fixed**)

Resizes the length of the subvectors. The contents are not re-initialized, use **clear** for that subsequently.

void **clear**()

Resets the values of all subvectors to zero.

SystemVector& **operator+=**(*const SystemVector&* **A**)

System vector addition.

SystemVector& **operator−=**(*const SystemVector&* **A**)

System vector subtraction.

SystemVector& **operator∗=**(*double* **s**)

Scalar system vector multiplication.

double& **operator()**(*const Dof&* **d**)

Returns a reference to the value in the system vector, referred to by the degree of freedom **d**. The d.o.f. can be free, prescribed or fixed. The value can be changed e.g.: `V(d) += 1.5`.

const double& **operator()**(*const Dof&* **d**) *const*

Returns a const reference to the value in the system vector, referred to by the degree of freedom **d**. The d.o.f. can be free, prescribed or fixed.

const Vector& **free**()*const*

Returns a const reference to the vector with free degrees of freedom. A non-const version is not supplied, to be able to change the data storage later.

const Vector& **prescribed**()*const*

Returns a const reference to the vector with prescribed degrees of freedom. A non-const version is not supplied, to be able to change the data storage later.

const Vector& **fixed**()*const*

Returns a const reference to the vector with fixed degrees of freedom. A non-const version is not supplied, to be able to change the data storage later.

Chapter 11

Utility classes

11.1 PostProc

A class for selection of post processing data and step numbers.

enum **PostItem { coordinates, stresses, eps_plast_eq }**

PostProc()

Default constructor.

~PostProc()

Destructor.

void **add_step**(*int* **step**)

Add step number **step** for post processing.

void **add_all_steps**()

All steps should generate post processing data.

void **add_last_step**()

The last step must generate post processing data.

bool **process**(*int* **step**, *bool* **is_last**) *const*

Returns true if this **step** needs post processing. The argument **is_last** indicates whether this step is the last in a sequence.

bool **any_selected**() *const*

Returns true if at least one item and one step was selected.

void **set_displa**()

Select displacements to be written.

void **set_inddis**()

Select incremental displacements to be written.

void **set_temp**()

Select temperatures to be written.

void **set_inctemp**()

Select incremental temperatures to be written.

void **set_stress**()

Select stresses to be written.

void **set_epspeq**()

Select equivalent plastic strains to be written.

bool **need_displa**() *const*

Returns true if the displacements are to be written.

bool **need_incdis**() *const*

Returns true if the incremental displacements are to be written.

bool **need_temp**() *const*

Returns true if the temperatures are to be written.

bool **need_inctemp**() *const*

Returns true if the incremental temperatures are to be written.

bool **need_stress**() *const*

Returns true if the stresses are to be written.

bool **need_epspeq**() *const*

Returns true if the equivalent plastic strains are to be written.

11.2 StepSelect

StepSelect()

Default constructor.

~StepSelect()

Destructor.

void **add_step**(*int* **step**)

Add step number **step** for processing.

void **add_all_steps**()

All steps should be processed.

void **add_last_step**()

The last step must be processed.

bool **is_selected**(*int* **step**, *bool* **is_last**) *const*

Returns true if this **step** must be processed. The argument **is_last** indicates whether this step is the last in a sequence.

bool **any_selected**() *const*

Returns true if at least one step was selected.

11.3 CollectArray

A class for collecting and retracting double precision scalars, Vector and Matrix objects to be stored e.g. for a restart. The file `collect.h` contains the declarations of the **CollectArray** class.

class **FeaTure::CollectArray**::*public* **std::vector<double>**

CollectArray()

Default constructor.

CollectArray(*int* **size**)

Constructor that reserves storage for **size** items.

~CollectArray()

Destructor.

void **reset**()

Resets the array. Subsequent reading from the array will start at the beginning.

void **add**(*double* **value**)

Add **value** to the array.

void **add**(*Vector* **vector**)

Add **vector** to the array.

void **add**(*Matrix* **matrix**)

Add **matrix** to the array.

double **extract**() *const*

Extracts the next double precision scalar from the array.

void **extract**(*Vector&* **v**) *const*

Extracts the next Vector **v** from the array.

void **extract**(*Matrix&* **m**) *const*

Extracts the next Matrix **m** from the array.

std::ostream& **operator**<<(*std::ostream&* **s**, *const CollectArray&* **A**)

Puts array **A** on the output stream **s**.

std::istream& **operator**>>(*std::istream&* **s**, *CollectArray&* **A**)

Reads array **A** from the input stream **s**.

11.4 Integration point data

The file `intpoint.h` contains the declarations of the class **IntLocs** for defining a set of integration point locations and weight factors and the declaration of the routines **Gauss** and **Hammer** for one-dimensional and triangular integration rules respectively.

void **FeaTure::Gauss**(*int* **ip**, *int* **nip**, *double&* **xi**, *double&* **weight**)

This routine returns the natural coordinate **xi** (between -1 and 1) and the corresponding weight factor for a **nip**-point Gauss integration rule.

void **FeaTure::Hammer**(*int* **ip**, *int* **nip**, *double&* **L1**, *double&* **L2**, *double&* **L3**, *double&* **weight**)

This routine returns the natural coordinates (area coordinates) **L1**, **L2** and **L3** (where $L_1 + L_2 + L_3 = 1$) of a triangular area and the corresponding weight factor for a **nip**-point integration rule according to Hammer.

11.4.1 The IntLocs class

class **FeaTure::IntLocs**

IntLocs()

Default constructor.

IntLocs(*const IntLocs&* **il**)

Copy constructor.

IntLocs(*int* **nr**)

Constructor that reserves space for **nr** integration points.

~IntLocs()

Destructor.

void **set_nr_intpt**(*int* **nr**)

Sets the number of integration points to **nr**.

int **get_nr_intpt**() *const*

Returns the number of integration points in this set.

void **set_values**(*int* **ip**, *Vector* **coor**, *double* **weight**)

Returns the natural coordinates of integration point **ip** from this set in **coor** and its weight factor in **weight**

Vector **get_coor**(*int* **ip**) *const*

Returns the natural coordinates of integration point **ip** from this set.

double **get_weight**(*int* **ip**) *const*

Returns the weight factor of integration point **ip** from this set.

11.5 Shape functions

The declarations of the class **shape** and its derivatives is contained in the file `shape.h`. These classes are intended to be used in numerically integrated continuum elements. They contain, among others, functions to get the interpolation vector and the matrix with derivatives to the local coordinates.

11.5.1 Shape

The **Shape** class is an abstract class that only contains pure virtual functions. The actual implementation must be defined in the derived classes.

virtual **~Shape**()

Destructor for the Shape objects.

virtual Matrix **get_N_vec**(*const Vector&* **natcor**) *const*=0

Returns the interpolation vector **N** for the particular shape at location **natcor** (in natural coordinates).

virtual Matrix **getPartialDerivatives**(*const Vector&* **natcor**) *const*=0

Returns the vector of partial derivatives of the shape functions to the applicable natural coordinates e.g. $\frac{\partial \mathbf{N}}{\partial \xi}$ and $\frac{\partial \mathbf{N}}{\partial \eta}$ for a quadrilateral shape at location **natcor** (in natural coordinates). The first row contains the derivatives to the first natural coordinate, the second row to the second etc.

virtual IntLocs **get_int_locs**() *const*=0

Returns a set of natural coordinates, representing the locations of all integration points.

virtual int **get_nr_int_points**() *const*=0

Returns the number of integration points for the actual shape.

virtual IntLocs **get_int_locs_face**(*int* **face_nr**) *const*=0

Returns a set of natural coordinates, representing the locations of integration points in face (or edge for 2D) number **face_nr** of the actual shape. This can be used e.g. for the analysis of face loads.

virtual Matrix **dVdA**(*int* **face_nr**) *const*=0

Returns a matrix that relates the natural volume coordinates to the natural area coordinates on face number **face_nr** (or the area coordinates to the line coordinates for 2D). This can be used e.g. for the analysis of face loads.
Example:
For a triangle the derivatives of an item to L_1 and L_2 along an edge can be translated to the derivatives to the edge coordinate ξ by post-multiplication with **dVdA**:

$$\frac{\partial \phi}{\partial \xi} = \frac{\partial \phi}{\partial \mathbf{L}} \cdot \frac{\partial \mathbf{L}}{\partial \xi}.$$

Where in this case **dVdA** represents $\frac{\partial \mathbf{L}}{\partial \xi}$.

11.5.2 Triangular

class **FeaTure::Triangular**::*public* **FeaTure::Shape**

Specification for a triangular shape.

~Triangular()

Destructor.

IntLocs **get_int_locs**() *const*

Returns the natural coordinates L_1, L_2 and L_3 and the corresponding weight factors for a triangular region.

int **get_nr_int_points**() *const*

Returns the total number of integration points for this shape. The total number can be given during initialization.

Matrix **dVdA**(*int* **face_nr**) *const*

Relates the directions of the local coordinates L_1, L_2 and L_3 to the coordinate ξ along edge number **face_nr**. The faces (edges actually) are numbered counter clockwise, starting from node 1.

IntLocs **get_int_locs_face**(*int* **face_nr**) *const*

Returns the location of integration points along edge number **face_nr**.

11.5.3 Quadrilateral

class **FeaTure::Quadrilateral**::*public* **FeaTure::Shape**

Specification for a quadrilateral shape.

~Quadrilateral()

Destructor.

IntLocs **get_int_locs**() *const*

Returns the natural coordinates ξ and η and the corresponding weight factors for a quadrilateral region.

int **get_nr_int_points**() *const*

Returns the total number of integration points for this shape. The total number can be given during initialization.

Matrix **dVdA**(*int* **face_nr**) *const*

Relates the directions of the local coordinates ξ and η to the coordinate ξ' along edge number **face_nr**. The faces (edges actually) are numbered counter clockwise, starting from node 1.

IntLocs **get_int_locs_face**(*int* **face_nr**) *const*

Returns the location of integration points along edge number **face_nr**.

11.5.4 Hexagonal

class **FeaTure::Hexagonal**::*public* **FeaTure::Shape**

Specification for a hexagonal shape.

~Hexagonal()

Destructor.

IntLocs **get_int_locs**() *const*

Returns the natural coordinates ξ, η and ζ and the corresponding weight factors for a hexahedral region.

int **get_nr_int_points**() *const*

Returns the total number of integration points for this shape. The total number can be given during initialization.

Matrix **dVdA**(*int* **face_nr**) *const*

Relates the directions of the local (volume) coordinates ξ, η and ζ to the (area) coordinates ξ' and η' along face number **face_nr**. The face numbers are numbered: 1 if $\eta = -1$, 2 if $\eta = 1$, 3 if $\zeta = -1$, 4 if $\xi = 1$, 5 if $\zeta = 1$ and 6 if $\xi = -1$.

IntLocs **get_int_locs_face**(*int* **face_nr**) *const*

Returns the location of integration points along face number **face_nr**.

11.5.5 Concrete shapes

A number of actual shape classes are derived from the previously defined intermediate classes. Only the derivation and the constructors are described below, because the classes further only define the specific behavior of previously declared virtual functions. The constructors require the definition of the selected integration scheme.

class **FeaTure::Tri_lin**:*public* **FeaTure::Triangular**

Three noded triangular element with a linear interpolation function. The nodes are numbered counter-clockwise.

Tri_lin(*int* **intscheme**)

Constructor, defining the number of integration points.

class **FeaTure::Tri_quad**:*public* **FeaTure::Triangular**

Six noded triangular element with a quadratic interpolation function. The nodes are numbered counter-clockwise.

Tri_quad(*int* **intscheme**)

Constructor, defining the number of integration points.

class **FeaTure::Tri_quad_alt**:*public* **FeaTure::Triangular**

Six noded triangular element with a quadratic interpolation function. The nodes are numbered counter-clockwise for all corners first and then for all midside nodes.

Tri_quad_alt(*int* **intscheme**)

Constructor, defining the number of integration points.

class **FeaTure::Quad_lin**:*public* **FeaTure::Quadrilateral**

Four noded quadrilateral element with a bi-linear interpolation function. The nodes are numbered counter-clockwise.

Quad_lin(*int* **nxi**, *int* **neta**)

Constructor, defining the number of integration points in ξ and η direction respectively.

class **FeaTure::Quad_quad**:*public* **FeaTure::Quadrilateral**

Eight noded quadrilateral element with a quadratic (serendipity) interpolation function. The nodes are numbered counter-clockwise.

Quad_quad(*int* **nxi**, *int* **neta**)

Constructor, defining the number of integration points in ξ and η direction respectively.

class **FeaTure::Quad_quad_alt**:*public* **FeaTure::Quadrilateral**

Eight noded quadrilateral element with a quadratic (serendipity) interpolation function. The nodes are numbered counter-clockwise for the corner nodes, followed by the midside nodes.

Quad_quad_alt(*int* **nxi**, *int* **neta**)

Constructor, defining the number of integration points in ξ and η direction respectively.

class **FeaTure::Hex_lin**:*public* **FeaTure::Hexagonal**

Eight nodes hexahedral element (cube) with a tri-linear interpolation function. The nodes are numbered in the base plane first ($\zeta = -1$), counter clockwise, looking from the positive ζ-direction, followed by the top plane ($\zeta = 1$).

Hex_lin(*int* **nxi**, *int* **neta**, *int* **nzeta**)

Constructor, defining the number of integration points in ξ, η and ζ direction respectively.

11.6 Exceptions

In the file `ftexcept.h`, a number of classes are defined for use with exception handling. The base-exception for FeaTure is

class **FeaTure::feature_error**:*public* **exception**

Since **feature_error** is derived from the standard exception: **exception** exceptions from FeaTure and standard exceptions can be catched with **catch**(*const exception&* **e**). The FeaTure exceptions can also be catched separately by **catch**(*const feature_error&* **e**) or even more specific catches. The *control* class has two exception classes for specific use (section 4.4).

feature_error(*const std::string&* **what_arg**)

The constructor for the base exception.

const char∗ **what** *const*()

Returns the message with which the exception was generated.

A number of derived classes are used for specific errors. Below, they are defined by showing their constructors:

file_error(*const std::string&* **what_arg**)

Derived from *feature_error* to indicate some error in file opening, reading or writing.

program_error(*const std::string&* **what_arg**)

Derived from *feature_error* to indicate some program error. Program errors are considered to be errors that should not occur in final releases of programs. They can be compared with well known **assert** macros.

not_implemented_error(*const std::string&* **what_arg**)

Derived from *program_error* to indicate a not yet implemented feature.

dimension_error(*const std::string&* **what_arg**)

Derived from *program_error* e.g. to indicate non-matching vector or matrix dimensions.

matrix_error(*const std::string&* **what_arg**)

Derived from *feature_error* for runtime matrix errors e.g. when an inverse matrix is calculated from a singular matrix.

not_found(*const std::string&* **what_arg**)

Derived from *feature_error* for exceptions that indicate that a requested item is not found in the model.

Bibliography

[1] J. Besson and R. Foerch. Large scale object-oriented finite element code design. *Comput. Methods Appl. Mech. Engrg.*, 142:165–187, 1997.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 2nd edition, 1994.

[3] Y. Dubois-Pèlerin and Th. Zimmermann. Object-oriented finite element programming: III. an efficient implementation in C++. *Comput. Methods Appl. Mech. Engrg.*, 108:165–183, 1993.

[4] D. Eyheramendy and Th. Zimmermann. Object-oriented finite elements: II. a symbolic environment for automatic programming. *Comput. Methods Appl. Mech. Engrg.*, 132:277–304, 1996.

[5] X. A. Kong and D. P. Chen. An object-oriented design for FEM programs. *Comput. Struct.*, 57:157–166, 1995.

[6] Ph. R. B. Devloo. PZ: An object oriented environment for scientific programming. *Comput. Methods Appl. Mech. Engrg.*, 150:133–153, 1997.

[7] R. Pozo. Template numerical toolkit for linear algebra: High performance programming with C++ and the standard template library. `http://math.nist.gov/tnt`.

[8] J. Smart, R. Roebling, V. Zeitlin, R. Dunn, et al. Reference manual for wxWindows 2.2: a portable C++ and Python GUI toolkit. `http://www.wxwindows.org`, 2000.

[9] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading etc., 3rd edition, 1997.

[10] A. H. van den Boogaard, N. van Vliet, and J. Huétink. Object oriented design of a thermo-mechanical fem code. In S. R. Idelsohn, E. Oñate, and E. N. Dvorkin, editors, *Computational Mechanics, New Trends and Applications*, Barcelona, 1998. CIMNE.

[11] L. Walterthum and J. C. Gelin. Design of an object oriented software for the computer aided simulation of complex forming processes. In Shen and Dawson, editors, *Simulation of Materials Processing: Theory, Methods and Applications*, pages 507–512, Rotterdam, 1995. Balkema.

[12] Th. Zimmermann, Y. Dubois-Pèlerin, and P. Bomme. Object-oriented finite element programming: I. governing principles. *Comput. Methods Appl. Mech. Engrg.*, 98:291–303, 1992.

[13] Th. Zimmermann and D. Eyheramendy. Object-oriented finite elements: I. principles of symbolic derivations and automatic programming. *Comput. Methods Appl. Mech. Engrg.*, 132:259–276, 1996.

Index