Contents

1

fem.h

Names

1.1

#define **SET_ELEMSET_TYPE_ALIAS** (alias,elemset_type)

If user enters an elemset type `alias` in the data file, associate with real elemset `elemset_type`.

Parameters:

`elem_set_type_alias`	elemset type alias
`elem_set_type`	real elemset type

Author: M. Storti

1.2

#define **SET_ELEMSET_TYPE** (elemset_type)

Use this macro to include elemset types in the function bless_elemset. NOTE: This is somewhat incorrect. NewElemset in the future will will replace Elemset. An in that case we will no need the explicit caset `(Elemset *)`. This is due to the fact that `Elemset` has private members for NewElemset and derived classes, so that we can not make polymorphism between these classes.

Parameters: `elemset_type` new elemset type to be included
Author: M. Storti

1.3

#define **GETOPTDEF_HOOK** (name)

This is prepended in some of the GETOPT macros to the name of the variable. Remember to first "undef" the macro

1.4

#define **TGETOPTDEF** (thash,type,name,default)

Gets a value of type int or double from any hash table. Example: TGETOPTDEF(thash,int,n,10)

Parameters:
`thash` the TextHashTable from where to get the value
`type` may be 'int' or 'double'
`name` name of the variable
`default` default value.
Author: M. Storti

1.5

#define **TGETOPTNDEF** (thash,type,name,default)

Gets a value of type int or double from any hash table. No default value assumed. Example: GETOPTDEF(thash,int,n,none)

Parameters:

`thash`	the TextHashTable from where to get the value
`type`	may be 'int' or 'double'
`name`	name of the variable
`default`	(none) for use with 'odoc.pl' default value.

Author: M. Storti

1.6

#define **TGETOPTDEF_S** (thash,type,name,default)

Gets a value of type string. Example: `TGETOPTDEF_S(thash,string,preco_type,"Jacobi")`

Parameters:

`thash`	the TextHashTable from where to get the value
`type`	should be string
`name`	name of the variable
`default`	default value (not in parentheses)

Author: M. Storti

1.7

#define **TGETOPTDEF_S_ND** (thash,type,name,default)

Gets a value of type string (doesn't declare the variable). Example: `TGETOPTDEF_S_ND(thash,string,preco_type,"Jacobi")`

Parameters:

`thash`	the TextHashTable from where to get the value
`type`	should be string
`name`	name of the variable (should be declared previously)
`default`	default value (not in parentheses)

Author: M. Storti

1.8

#define **NGETOPTDEF_S** (type,name,default)

Gets a value of type string from the thash of the elemset Example: `NTGETOPTDEF_S(string,preco_type,"Jacobi")`

Parameters:

`type`	should be string
`name`	name of the variable
`default`	default value (not in parentheses)

Author: M. Storti

1.9

#define **GETOPTDEF** (type,name,default)

Gets a value of type int or double from the global hash table. Example: GETOPTDEF(int,n,10)

Parameters:

`type`	may be 'int' or 'double'
`name`	name of the variable
`default`	default value.

Author: M. Storti

1.10

#define **GGETOPTDEF** (type,name,default)

Gets a value of type int or double from the global options hash table. Example: GGETOPTDEF(int,n,10)

Parameters:

`type`	may be 'int' or 'double'
`name`	name of the variable
`default`	default value.

Author: M. Storti

1.11

#define **SGETOPTDEF** (type,name,default)

Gets a value of type int or double from the general element hash table. Example: SGETOPTDEF(int,n,10)

Parameters:

`type`	may be 'int' or 'double'
`name`	name of the variable
`default`	default value.

Author: M. Storti

1.12

#define **SGETOPTDEF_ND** (type,name,default)

Gets a value of type int or double from the elemset hash table. Doesn't define the variable. Example: SGETOPTDEF(int,n,10)

Parameters:

`type`	may be 'int' or 'double'
`name`	name of the variable
`default`	default value.

Author: M. Storti

1.13

#define **EGETOPTDEF** (elemset,type,name,default)

Gets a value of type int or double from an elemset hash table. Example: EGETOPTDEF(elemset,int,n,10)

Parameters:

`type`	may be 'int', 'double' or 'string'
`name`	name of the variable
`default`	default value.

Author: M. Storti

1.14

#define **EGETOPTDEF_ND** (elemset,type,name,default)

Gets a value of type int or double from an elemset hash table. Doesn't define the variable. Example: EGETOPTDEF_ND(elemset,int,n,10)

Parameters:

`type`	may be 'int', 'double' or 'string'
`name`	name of the variable
`default`	default value.

Author: M. Storti

1.15

#define **TGETOPTDEF_ND** (thash,type,name,default)

Gets a value of type int or double from any hash table. Doesn't define the variable. Example: GGETOPTDEF(thash,int,n,10)

Parameters:

`thash`	the TextHashTable from where to get the value
`type`	may be 'int' or 'double'
`name`	name of the variable
`default`	default value.

Author: M. Storti

1.16

#define **NSGETOPTDEF** (type,name,default)

Gets a value of type int or double from the general element hash table. OOP version. Example: NSGETOPTDEF(int,n,10)

Parameters:

`type`	may be 'int' or 'double'
`name`	name of the variable
`default`	default value.

Author: M. Storti

1.17

#define **NSGETOPTDEF_ND** (type,name,default)

Gets a value of type int or double from the general element hash table. OOP version. Doesn't define the variable. Example: NSGETOPTDEF_ND(int,n,10)

Parameters:

`type`	may be 'int' or 'double'
`name`	name of the variable
`default`	default value.

Author: M. Storti

1.18

#define **PFEMERRQ** (s)

Sets an error condition and back-traces (routines). Based on PETSc 'PFEMERRQ' macro. Use this macro in internal routines. Use PFEMERRA in the main.

Parameters:

`s`	string of error message

Author: M. Storti

1.19

#define **PFEMERRA** (s)

Sets an error condition and back-traces (main). Based on PETSc 'PFEMERRA' macro. Use this macro in the main. Use PFEMERRQ in internal routines.

Parameters: `s` string of error message
Author: M. Storti

1.20

#define **PFEMERRCQ** (ierr,s)

Sets an error condition depending on error code and back-traces (routines). Like PFEMERRQ, but checks integer 'ierr' and sets error if ierr>0. Based on PETSc 'PFEMERRQ' macro. Use this macro in internal routines. Use PFEMERRA in the main.

Parameters: `ierr` error code (input)
`s` string of error message
Author: M. Storti

1.21

#define **PFEMERRCA** (ierr,s)

Sets an error condition depending on error code and back-traces (routines). Like PFEMERRQ, but checks integer 'ierr' and sets error if ierr>0. Based on PETSc 'PFEMERRQ' macro. Use this macro in the main. Use PFEMERRA in internal routines.

Parameters: `ierr` error code (input)
`s` string of error message
Author: M. Storti

1.22

#define **USE_VARARG_MACROS**

Issues an error (with `PetscPrintf())` and calls `PetscFinalize()`. Argument are as for 'printf()' usage: PETSCFEM_ERROR(template,args).

1.23

#define **PETSCFEM_ASSERT0** (bool_cond,templ)

If `bool_cond` evaluates to false issues an error (with `PETSCFEM_ERROR`). usage: `PETSCFEM_ASSERT(bool_cond,printf_args);`

1.24

class **Nodedata**

Contains (constant) data relative to nodes (may be coordinates and other).

Parameters:
- `nnod` number of nodes
- `nu` number of real quantities per node

Author: M. Storti

1.25

class **Mesh**

Public Members

Contains a list of the elemsets, node information and a hash table of global options.

Parameters:
- `elemsetlist` list of elemsets
- `nodedata` constant data per node
- `global\_options` hash table with global options

Author: M. Storti

1.25.1

Darray* **elemsetlist**

The list of elemsets

1.25.2

Nodedata* **nodedata**

The array of nodes

1.25.3

TextHashTable* **global_options**

The option table

1.25.4

Elemset* **find** (const string &name)

Finds an elemset give its name

1.25.5

Mesh ()

Default ctor

1.26

#define **ijob**

Determines what kind of action is performed on the data

1.27

#define **EPSILON_FDJ**

parametros numericos

1.28

Utility definitions

Names

1.28.1

#define **SHV_OPTIONS**

Prints a variable along with his name (useful for debugging).

1.28.2

#define **SHM** (x)

Prints a matrix variable along with his name (useful for debugging).

1.28.3

#define **TRACE** (s)

Prints a trace

1.28.4

#define **SHVS** (name,f)

Prints a variable with printf()

1.29

int
print_vector (const char* filename, const Vec x, const Dofmap* dofmap, const TimeData* time_data=NULL, const int append=0)

Converts a state vector (reduced form) to node/field form and prints it.

Parameters:	`filename`	(input) file where to write the vector. May containrelative directories.
	`x`	(input) PETSc MPI vector to be written
	`dofmap`	(input) corresponding dofmap
	`time_data`	(input, def=NULL) an external parameter in order to computeexternal boundary conditions, etc...
	`append`	(input, def=0) appending mode (append if'append==0')
Author:	M. Storti	

1.30

int
state2fields (double* fields, const Vec x, const Dofmap* dofmap, const TimeData* time_data=NULL)

Converts a state vector (reduced form) to node/field form.

Return Value:	`PETSc`	error code
Parameters:	`fields`	(output) double array where the node/fields form is put.
	`x`	(input) PETSc MPI vector to be written
	`dofmap`	(input) corresponding dofmap
	`time_data`	(input, def=NULL) an external parameter in order to computeexternal boundary conditions, etc...

Author: M. Storti

1.31

int **read_vector** (const char* filename, Vec x, Dofmap* dofmap, int myrank)

Reads a vector from a file. This can be used for initialization for instance.

Parameters:

`filename`	file from where to read the vector. May containrelative directories.
`x`	PETSc MPI vector to be read
`x`	PETSc sequantial working vector
`dofmap`	corresponding dofmap

Author: M. Storti

1.32

int
compute_prof (Darray* da, Dofmap* dofmap, int myrank, Mat* A, int debug_compute_prof)

Computes the sparse profile for defining matrices. PETSc needs the number of non null entries in the diagonal and off diagonal blocks. Calling 'assemble' with option 'COMP_MAT_PROF' computes 'da' and this function defines a matrix prototype.

Parameters:

`da`	(input) the dynamic array computed by assemble
`dofmap`	(input) the corresponding dofmap
`myrank`	(input) the index of the current processor
`A`	(output) PETSc matrix prototype

Author: M. Storti

1.33

int **zeroe_mat** (Mat A, int & ass_flag)

Sets a Matrix to zero. This is obsolete. We should call MatZeroEntries() (PETSc)

Parameters:

`A`	matrix to set to zero
`ass_flag`	a flag indicating wether the matrix was alreadybuilt or not.

Author: M. Storti

1.34

int **opt_read_vector** (Mesh* mesh, Vec x, Dofmap* dofmap, int myrank)

1.35

void **print_copyright** (void)

1.36

#define **DONT_SKIP_JOBINFO** (name)

Process this jobinfo for this elemset.

Parameters: `name` (input) the jobinfo to be processed
Author: M. Storti

1.37

inline double **square** (double x)

Returns the square of a double

Parameters: `x` (input) the base

1.38

inline double **cube** (double x)

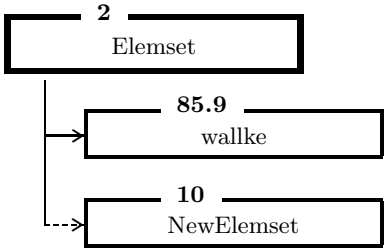
Returns the cube of a double

Parameters: `x` (input) the base

2

class **Elemset**

Inheritance



Public Members

Private Members

Stores sets of elements with similar types and properties.

Author: M. Storti

2.1

Elemset ()

ctor

2.2

virtual **~Elemset** ()

dtor

2.3

char* **type**

type of element

2.4

int* **icone**

table of connectivities

2.5

int **nelem**

number of elements in the elemset

2.6

int **nel**

number of nodes per element

2.7

int **ndof**

number of degrees of freedom per node

2.8

int* **epart**

mesh partitioning as computed by Metis

2.9

int* **epart2**

Contains the position of element in this processor. It is obtained by first numbering all the elements in processor 0, then on 1, and so on. Old 'epart' vector could be replaced by this in a future.

2.10

vector<int> **epart_p**

Element 'k' is in processor 'p' if 'epart2[k]' is in range epart2_p[p] and epart2_p[p+1]

2.11

int **e1**

Particularly elements e1 = epart2_p[MY_RANK], e2 = epart2_p[MY_RANK+1]

2.12

int **isfat**

flag indicating whether this is the fat elemset or not

2.13

int **nelem_here**

number of elements in this processor

2.14

int **nelprops**

number of double properties in the per-element properties table

2.15

int **neliprops**

number of integer properties in the per-element properties table

2.16

int **nelprops_add**

number of additional double properties

2.17

int **neliprops_add**

number of additional integer properties

2.18

double* **elemprops**

double per-element properties table

2.19

int* **elemiprops**

int per-element properties table

2.20

double* **elemprops_add**

additional double per-element properties table

2.21

int* **elemiprops_add**

additional int per-element properties table

2.22

void** **local_store**

This is a “loccker” for each element

2.23

TextHashTable* **thash**

properties hash table

2.24

GHashTable* **elem_prop_names**

hash of properties in the per element double properties table

2.25

GHashTable* **elem_iprop_names**

hash of properties in the per element int properties table

2.26

string **name_m**

The name of the elemset

2.27

virtual void **initialize** ()

Makes some initialization from the hash table

2.28

virtual int **real_nodes** (int iele, const int* &nodes)

Returns the list of "real" nodes (this is used for computing the graph)

2.29

virtual void
before_assemble (arg_data_list &arg_datav, Nodedata* nodedata, Dofmap* dofmap, const char* jobinfo, int myrank, int el_start, int el_last, int iter_mode, const TimeData* time_data)

This called only once before calling assemble for each chunk. The arguments are the same as for the assemble function.

Parameters:

`nodedata`	(input) vector with properties per node
`locst`	(input) state vectors localized to elements
`locst2`	(input) same for alternative state vector (this may beused in temporal integration), etc...
`dofmap`	(input) maps the node/field representation to the vectorstate
`jobinfo`	(input) tells the routine element what kind ofmatrix or vector has to be assembled
`myrank`	(input) identifies this processor
`el_start`	(input) low end of the range of elements to beprocessed
`el_last`	(input) high end of the range of elements to beprocessed
`iter_mode`	(input) include or not ghost elements

2.30

virtual void **after_assemble** (const char* jobinfo)

This called only once after calling assemble for each chunk.

2.31

virtual int
assemble (arg_data_list &arg_datav, Nodedata* nodedata, Dofmap* dofmap, const char* jobinfo, int myrank, int el_start, int el_last, int iter_mode, const TimeData* time_data)

Assembles residuals, matrices, scalars and other quantities.

Parameters:

`retval`	(output) here returns contributions vectors(residuals), and matrices
`nodedata`	(input) vector with properties per node
`locst`	(input) state vectors localized to elements
`locst2`	(input) same for alternative state vector (this may beused in temporal integration), etc...
`dofmap`	(input) maps the node/field representation to the vectorstate
`jobinfo`	(input) tells the routine element what kind ofmatrix or vector has to be assembled
`myrank`	(input) identifies this processor
`el_start`	(input) low end of the range of elements to beprocessed
`el_last`	(input) high end of the range of elements to beprocessed
`iter_mode`	(input) include or not ghost elements

2.32

virtual int **ask** (const char* jobinfo, int &skip_elemset)

Ask the elemset if it should be processed for this jobinfo.

Parameters:
`jobinfo` (input) The name of the task.
`answer` (output) 1 = process (0 = do not process) thiselemset.

Author: M. Storti

2.33

int **size** ()

Returns the number of elements in the elemset.

Return Value: `number` of elements

2.34

Darray* **ghost_elems**

dynamic aray with ghost elements

2.35

void **print** ()

print info of this elemset

2.36

virtual double **weight** ()

Returns the amount of work needed to process this element.

Return Value: `the` weight of the processor

2.37

int
download_vector (int nel, int ndof, Dofmap* dofmap, arg_data &argd, int myrank, int el_start, int el_last, int iter_mode, const TimeData* time_data=NULL)

Builds localized vectors (in node/field) representation of state vectors.

Parameters:

`nel`	(input) number of nodes per element
`ndof`	(input) number of degrees of freedom per node
`dofmap`	(input) the dofmap of the mesh
`locst`	(output) localized vector to be assembled
`myrank`	(input) identifies this processor
`el_start`	(input) low end of the range of elements to beprocessed
`el_last`	(input) high end of the range of elements to beprocessed
`iter_mode`	(input) include or not ghost elements
`time_data`	(input, def=NULL) an external parameter in order to computeexternal boundary conditions, etc...

Author: M. Storti

2.42

int
upload_vector (int nel, int ndof, Dofmap* dofmap, int options, arg_data &argd, int myrank, int el_start, int el_last, int iter_mode, int klocc=0, int kdofc=0)

Put localized values (nod/field representation) returned by elemset assembles in the global residual vector.

Parameters:

`nel`	(input) number of nodes per element
`ndof`	(input) number of degrees of freedom per node
`dofmap`	(input) the dofmap of the mesh
`retval`	(input) localized values (nod/fieldrepresentation) returned by elemset assembles
`A`	(input/output) assemble matrix contributions on it (ifijob/jobinfo indicates so).
`da`	(output) assemble matrix profile on this (ifijob/jobinfo indicates so).
`vec`	(output) assemble vector contributions on it (ifijob/jobinfo indicates so).
`myrank`	(input) identifies this processor
`el_start`	(input) low end of the range of elements to beprocessed
`el_last`	(input) high end of the range of elements to beprocessed
`iter_mode`	(input) include or not ghost elements
`klocc`	(input) if computing difference finite jacobian, isthe local node perturbed
`klocc`	(input) if computing difference finite jacobian, isthe local d.o.f. perturbed

Author: M. Storti

2.43

void*& **local_store_address** (int global_elem)

Return the "locker" for the element, that is a (void *) where to put data. Only for elements local to this processor.

Return Value: `the` address (void *) of the locker
Parameters: `global_elem` (input) the index of the element in the elemset
Author: M. Storti

2.44

void
element_node_data (const ElementIterator &element, const Nodedata* nodedata, double* xloc, double* Hloc) const

Returns coordinates and node data for the element

Parameters:
`element` (input) iterator to the element for which to return the data
`nodedata` (input) the node coordinate info
`xloc` (output) the coordinates of the element nodes
`Hloc` (output) the auxiliary data for the nodes
Author: M. Storti

2.45

double* **element_props** (const ElementIterator &element) const

Returns element properties

Return Value: `pointer` to an array of nelprops doubles for the element
Parameters: `element` (input) iterator to the element for which to return the data
Author: M. Storti

2.46

void **element_connect** (const ElementIterator &element, int* connect) const

Returns node indices for the element

Parameters:
- `element` (input) iterator to the element for which to return the data
- `connect` (input) indices of the element nodes

Author: M. Storti

2.47

const double*
element_vector_values (const ElementIterator &element, arg_data &ad) const

Returns localized vector values for a given element

Return Value: `a` pointer to the element values

Parameters:
- `element` (input) iterator to the element for which to return the data
- `ad` (input) the argument in the arglist.

Author: M. Storti

2.48

double*
element_ret_vector_values (const ElementIterator &element, arg_data &ad) const

Returns localized element vector contributions for a given element

Return Value: `a` pointer to the element values

Parameters:
- `element` (input) iterator to the element for which to return the data
- `ad` (input) the argument in the arglist.

Author: M. Storti

2.49

double*
element_ret_fdj_values (const ElementIterator &element, arg_data &ad) const

Returns localized element contributions to FD jacobian for a given element

Return Value: `a` pointer to the element values
Parameters: `element` (input) iterator to the element for which to return the data
`ad` (input) the argument in the arglist.
Author: M. Storti

2.50

double*
element_ret_mat_values (const ElementIterator &element, arg_data &ad) const

Returns localized element contributions to FD jacobian for a given element

Return Value: `a` pointer to the element values
Parameters: `element` (input) iterator to the element for which to return the data
`ad` (input) the argument in the arglist.
Author: M. Storti

2.51

void **elem_params** (int &nel_, int &ndof_, int &nelprops_) const

Returns element values

Parameters: `nel_` (output) the number of nodes connected to an element
`ndof_` (output) the number of dofs for each node
Author: M. Storti

2.52

static string **anon**

Default name (the elemset is renamed if this name is passed

2.53

int* **elem_conne**

Stores temporarily element connectivities

2.54

static map<string,Elemset *> **elemset_table**

A table where all the elemsets are registered

2.55

void **register_name** (const string &name, const char* type)

register the elemset in the table and generates a unique name

Parameters: `type` (input) the type of the elemset

2.56

virtual void **read** (FileStack* fstack)

Pass to the elemset the filestack in order to read data, if needed

2.58

void **clear_error** ()

Clear the error code in the elemset

2.59

void **set_error** (int error)

Set an error at the element level

2.60

void **check_error** ()

Check the actual error code (collective)

2.61

virtual void **handle_error** (int error)

This can be modified by the user in order to handle different errors

2.62

int **error_code** ()

Return the actual error code

2.63

TextHashTable* **option_table** ()

Return the option table for this elemset

2.65

virtual void **dx** (Socket* sock, Nodedata* nd, double* field_state)

Generates fields for processing with DX.

Parameters:

`sock`	(input) socket where to send data following protocol-understand by DX `ExtProgImport` module
`nd`	(input) coordinates object
`field_state`	(input) array of values (node/field) representation

2.66

virtual int **dx_types_n** ()

Number of sub-types in the splitting.

Return Value: `number` of subelements

2.67

virtual void **dx_type** (int j, string &dx_type, int &subnel, vector<int> &nodes)

Returns the description of the `j`-th type.

Parameters:

`j`	(input) 0-based type index
`dx_type`	(output) the DX type
`subnel`	(input) the number of nodes for this type
`nodes`	(input) the nodes connected to this subelement. Lengthmust be multiple of subnel

2.38

int
upload_vector_fast (int nel, int ndof, Dofmap* dofmap, int options, arg_data &argd, int myrank, int el_start, int el_last, int iter_mode, int klocc=0, int kdofc=0)

This is the fast version, calls only MateSetValues once per element with a block. However it is constrained to have a whole block. Wether this or `upload_vector_slow` are used depends on the `fast_uploading` option for the elemset.

2.39

int
upload_vector_fast_mb (int nel, int ndof, Dofmap* dofmap, int options, arg_data &argd, int myrank, int el_start, int el_last, int iter_mode, int klocc=0, int kdofc=0)

Fast version, like `upload_vector_fast` but allows mutiple blocks by introducing a different mask value for each block.

2.40

int
upload_vector_fast_1b (int mask_val, int nel, int ndof, Dofmap* dofmap, int options, arg_data &argd, int myrank, int el_start, int el_last, int iter_mode, int klocc=0, int kdofc=0)

Auxiliary function for `upload_vector_fast_mb` loads blocked values for a specific value of the mask.

2.41

int
upload_vector_slow (int nel, int ndof, Dofmap* dofmap, int options, arg_data &argd, int myrank, int el_start, int el_last, int iter_mode, int klocc=0, int kdofc=0)

Makes a MatsSetValue for each element of the matrix. Slower but accept arbitrary masks.

2.57

int **error_code_m**

The actual error code

2.64

void
dx_send_connectivities (Socket* sock, int nsubelem, int subnel, vector<int> &node_indices)

Auxiliary function that sends connectivities to DX

3

int
assemble (Mesh* mesh, arg_list argl, Dofmap* dofmap, const char* jobinfo, const TimeData* time_data=NULL)

Global assemble function. Loops over all elemsets and assembles element contributions to amtrices, vectors and profiles. Same as assemble, but for computine profiles.

Return Value: `error` code

Parameters:

`mesh`	(input) the mesh to be processed
`argl`	(input/output) list of arguments, (vector matrices) on inputand output.
`dofmap`	(input) the dofmap of the mesh
`jobinfo`	(input) tells the routine element what kind ofmatrix or vector has to be assembled
`time_data`	(input, def=NULL) an external parameter in order to computeexternal boundary conditions, etc...

Author: M. Storti

4

class **ElementList**

Public Members

Private Members

Const iterator for looping over elements in an elemset

4.5

ElementList (const Elemset* elemset_, int first_, int last_, ElemsetIteratorMode mode_)

Constructor

4.6

ElementIterator **begin** (void) const

Begin of chunk

4.7

ElementIterator **end** (void) const

End of chunk

4.1

const Elemset* **elemset**

Pointer to the elemset

4.2

int **first**

Start range of values while iterating

4.3

int **last**

Ends range of values while iterating

4.4

ElemsetIteratorMode **mode**

Iteration mode

5

class **ElementIterator**

Public Members

Private Members

Iterators for looping over elements in an elemset

5.4

ElementIterator ()

Default Constructor

5.5

ElementIterator (const ElementList* el, const int rie, const int ric)

Constructor

5.6

ElementIterator& **operator++** (void)

Prefix increment operator

5.7

ElementIterator **operator++** (int)

Postfix Increment operator

5.8

int **operator!=** (const ElementIterator &other) const

Not equal operator

5.9

void **position** (int &pos_in_elemset, int &pos_in_chunk) const

Return position in chunk

5.10

Returns true/false whether the current element is

Returns true/false whether the current element is

5.11

Advances iterator to

Advances iterator to

5.12

void **node_data** (const Nodedata* nodedata, double* xloc, double* Hloc)

Returns coordinates and node data for the element

Parameters:
`nodedata` (input) the node coordinate info
`xloc` (input) the coordinates of the element nodes
`Hloc` (input) the auxiliary data for the nodes

Author: M. Storti

5.13

const double* **vector_values** (arg_data &ad) const

Returns localized vector values for a given element

Return Value:
`a` pointer to the element values

Parameters:
`ad` (input) the argument in the arglist.

Author: M. Storti

5.14

double* **ret_vector_values** (arg_data &ad) const

Returns localized vector element contributions for a given element

Return Value: `a` pointer to the element values
Parameters: `element` (input) iterator to the element for which to return the data
`ad` (input) the argument in the arglist.
Author: M. Storti

5.15

double* **ret_fdj_values** (arg_data &ad) const

Returns localized vector element contributions to FD jacobian for a given element

Return Value: `a` pointer to the element values
Parameters: `ad` (input) the argument in the arglist.
Author: M. Storti

5.16

double* **ret_mat_values** (arg_data &ad) const

Returns localized matrix element contributions for a given element

Return Value: `a` pointer to the element values
Parameters: `ad` (input) the argument in the arglist.
Author: M. Storti

5.17

double* **props** ()

Returns element properties

Return Value: `pointer` to an array of nelprops doubles for the element
Author: M. Storti

5.1

const ElementList* **elemlist**

The list to which it belongs

5.2

int **rank_in_chunk**

Rank of element in the chunk

5.3

int **rank_in_elemset**

Rank of element in the elemset

6

compute_prof package

Names

6.1

class **Node**

Public Members

nodes to store the profile of a matrix.

Author: M. Storti

6.1.1

int **next**

points to next dof connected to this one

6.1.2

val

the connected dof

6.1.3

Node (int next_=-1, int val_=0)

constructor

6.1.4

void **print** (void)

printer

6.2

void **node_insert** (Darray* da, int j, int k)

Inserts a node in the profile.

Parameters:

`da`	(input/output) dynamic array containing the profile
`j`	(input) row index
`k`	(output) column index

Author: M. Storti

6.3

int **int_cmp** (const void* left, const void* right, void* args)

Compare function for sort and then binary search on the profile.

Parameters:

`left`	(input) pointer to left element to be compared
`right`	(input) pointer to right element to be compared
`args`	(not used) as required by libretto routines

Author: M. Storti

7

int
compute_this_elem (const int & iele, const Elemset* elemset, const int & myrank, int iter_mode)

Flags if the current element is to be computed his contribution. Basically it process all elements in the current processor. However, depending on iter_mode it may include also ghot-elements.

Return Value:	`boolean`	flag indicating whether process or not this element
Parameters:	`iele`	(input) element to be tested
	`elemset`	(input) this elemset
	`myrank`	(input) this processor index
	`iter\_mode`	(input) flags whether or not includeghost-elements.
Author:	M. Storti	

8

int
measure_performance_fun (Mesh* mesh, arg_list argl, Dofmap* dofmap, const char* jobinfo, const TimeData* time_data=NULL)

Loops many times executing assmble() with the same arguments and issuing a performance indication.

Return Value:	`error`	code
Parameters:	`mesh`	(input) the mesh to be processed
	`argl`	(input/output) list of arguments, (vector matrices) on inputand output.
	`dofmap`	(input) the dofmap of the mesh
	`jobinfo`	(input) tells the routine element what kind ofmatrix or vector has to be assembled
	`time_data`	(input, def=NULL) an external parameter in order to computeexternal boundary conditions, etc...
Author:	M. Storti	

9

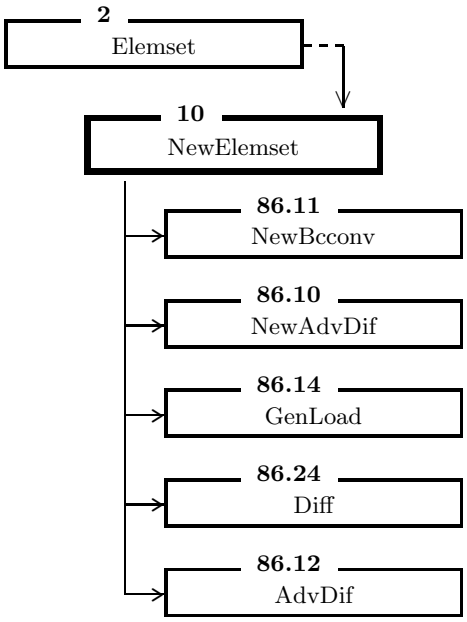
class **Property**

The generic ElemProperty class

10

class **NewElemset** : private Elemset

Inheritance



Public Members

Private Members

This is an adaptor to the old Elemset class

Author: M. Storti

10.3

virtual void
new_assemble (arg_data_list &arg_datav, const Nodedata* nodedata, const Dofmap* dofmap, const char* jobinfo, const ElementList &elemlist, const TimeData* time_data)

The new assemble function

10.4

int
get_vec_double (const char* name, vector<double> &retval, int defval=0) const

Gets a vector of double properties whose length is unknown.

Parameters:

`name`	(input) the name of the property
`retval`	(input) the returned vector
`defval`	(input) Controls the action if no entry is foundin the hash table. If `defval==0`, let retval unchanged, so thatyou have to set it to its default value. Give an error if`defval!=0`

Author: M. Storti

10.5

void **get_prop** (Property &prop, const char* prop_name, int n=1) const

Creates a Property object from his name

10.1

int
assemble (arg_data_list &arg_datav, Nodedata* nodedata, Dofmap* dofmap, const char* jobinfo, int myrank, int el_start, int el_last, int iter_mode, const TimeData* time_data)

This is the adaptor to the old assemble function

10.2

The new get-prop methods

Names

10.2.1

vector<double> **propel**

The vector containing the double values

10.2.2

double* **begin_propel**

The first position in propel

10.2.3

vector<int> **elprpsindx**

The indices

11

enum **arg_options**

Type of arguments and processing information for items in the arglists's

12

class **arg_entry**

Public Members

Individual entry in the argument

Parameters:

`arg`	(input)
`arginfo`	(input)

Author: M. Storti

12.1

void* **arg**

Generic pointer to the argument being passed.

12.2

int **options**

other options in the form of bitfields.

12.3

string **arginfo**

string containing info about what the argument contains and how has to be processed

12.4

arg_entry (void* arg_, int options_, string arginfo_="")

Constructor

13

class **arg_data**

Public Members

For each argument the quantities passed to the element routine are stored in a list of this structs. For instance for a MPI PETSc vector, we store a pointer to it, a pointer to the 'got' array of doubles in 'sstate' and also a pointer to an array with ghost values. For an associative vector we store a pointer to the vector and a flag called 'was_set' that flags if an element has been inspected or not.

Author: M. Storti

13.1

string **arginfo**

string containing info about what the argument contains and how has to be processed

13.2

int **options**

A copy of the options for the corresponding arg_entry value.

13.3

Vec* **x**

The MPI vector

13.4

double* **sstate**

Vector of doubles where the MPI vector is 'got'.

13.5

Vec* **ghost_vec**

Sequential vector with ghost values.

13.6

double* **ghost_vals**

Vector of doubles where the ghost vector is 'got' (with VecGetArray).

13.7

double* **locst**

Local values (one row per element).

13.8

Mat* **A**

PETSc-MPI Matrix

13.9

PFMat* **pfA**

PETSc-FEM Matrix

13.10

double* **profile**

Profile of matrix

13.11

double* **retval**

Returned local values(one row per element) .

13.12

double* **refres**

reference state for finite difference calculation of jacobians

13.13

Darray* **da**

Libretto dynamic array for storing the profile.

13.14

vector<double> * **vector_assoc**

Vector for associative operationes (max, min, and add operations)

13.15

int **was_set**

For vector_assoc arguments with MIN and MAX operations, flags if the actual min or max value has been already set.

13.16

void* **user_data**

This is passed to the element routines and nothing specific is done. It is assumed that is managed by the user.

13.17

int **must_flush**

Indicates if assembly state must be flushed with assembly_end()

13.18

arg_data ()

Default constructor.

13.19

~arg_data ()

Destructor

14

class **FastMat**

Public Members

Fast matrices.

14.1

FastMat (int m=0, int n=0, double* s = NULL)

Default constructor. Constructor from dimensions and from optional vector

14.2

FastMat (const Matrix & B)

Constructor from a Newmat matrix

14.3

~FastMat ()

destructor

14.4

void **eye** (int m)

set to the identity matrix

14.5

void **set_size** (int m, int n)

sets correct size (only if not set previously)

14.6

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

sets element i,j to val

14.7

void **set** (int i1, int i2, int j1, int j2, double* val)

sets submatrix i1 to i2, j1 to j2 to val

14.8

void **set** (int i1, int i2, int j1, int j2, const FastMat & B)

sets submatrix i1 to i2, j1 to j2 to matrix B

14.9

void **add** (const int i, const int j, const double val)

adds val to element i,j

14.10

void **add** (int i1, int i2, int j1, int j2, double* val)

adds submatrix in array val to submatrix i1 to i2, j1 to j1

14.11

void **add** (int i1, int i2, int j1, int j2, const FastMat & B)

adds submatrix from another matrix in array val to submatrix i1 to i2, j1 to j1

14.12

void **set** (double* val, int n=0)

copies from array. Only n first elements if n
neq 0

14.13

void **set** (const FastMat & A, int n=0)

copies from other matrix. Only n first elements if n
neq 0

14.14

void **set** (double val)

sets to all elements to a constant

14.15

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

returns element i,j (obsolete!!)

14.16

void **get** (const int i, const int j, double & val) const

returns element i,j

14.17

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

returns element i,j

14.18

void **get** (const int i1, const int i2, const int j1, const int j2, const FastMat &A)

returns block (i1,i2,j1,j2) of matrix A a *this

14.19

void **print** (char* s)

prints the matrix

14.20

void **copy** (double* val, int n=0) const

copies from matrix to external array. only first n elements if n.neq.0

14.21

void **row** (const FastMat & B, int i)

copies a row to *this

14.22

void **rows** (const FastMat & B, int i1, int i2)

copies some rows to *this

14.23

void **column** (const FastMat & B, int j)

copies some columns to *this

14.24

void **columns** (const FastMat & B, int j1, int j2)

copies some columns to *this

14.25

double* **location1** (const int i, const int j) const

return pointer to location (i,j) (base 1)

14.26

double* **location0** (const int i, const int j) const

return pointer to location (i,j) (base 0)

14.27

void **reshape** (int m, int n)

reshapes the matrix

14.28

int **as_scalar** (double & val) const

converts to scalar

14.29

double **sum** () const

sum of all terms

14.30

int **sum_square** (double & val) const

sum of squares

14.31

int **transpose** (const FastMat & A)

transpose

14.32

int **scale** (const double c)

scales the matrix

14.33

int **trace_of_product** (const FastMat & B, double & trace) const

Trace of product

14.34

double **norm1** () const

Norm 1

14.35

double* **store**

the storage

14.36

int **m**

matrix dimensions

14.37

static int **count**

number of existing matrices

14.38

int **defined**

flags if matrix was already defined

14.39

int **is_defined** (void) const

flags if matrix was already defined

15

FastMat operations

Names

15.1

int **FMp** (FastMat & A, const FastMat & B, const FastMat & C)

Product of matrices. A=B*C

Parameters:

- `A` (output) result of product
- `B` (input) first matrix
- `C` (input) second matrix

Author: M. Storti

15.2

int **FMa** (FastMat & A, const FastMat & B, const FastMat & C)

Addition of matrices. A = B + C

Parameters:
- `A` (output) result of sum
- `B` (input) first matrix
- `C` (input) second matrix

Author: M. Storti

15.3

int **FMaxpy** (FastMat & Y, double a, FastMat & X)

AXPY operation for fastmat matrices y = a x + y

Parameters:
- `Y` (input/output) matrix to be modified
- `a` (input) scalar
- `(input)` matrix to be added

Author: M. Storti

15.4

int **FMaxpy_t** (FastMat & Y, double a, const FastMat & X)

AXPY operation for fastmat matrices y = a x' + y

Parameters:
- `Y` (input/output) matrix to be modified
- `a` (input) scalar
- `(input)` matrix to be added (transposed)

Author: M. Storti

15.5

int **FMinv** (FastMat & invA, FastMat & A)

Inverse of a matrix

Parameters: `A` (input) the matrix to be inverted
`invA` (output) the inverted matrix
Author: M. Storti

15.6

int **FMtr** (FastMat & At, const FastMat & A)

Transpose of a matrix

Parameters: `A` (input) the matrix to be transposed
`At` (output) the transposed matrix
Author: M. Storti

15.7

int **FMdet** (double & det, const FastMat & A)

Determinant of a Fastmat matrix

Parameters: `A` (input) the matrix
`det` (ouput) the determinant
Author: M. Storti

15.8

int **NM2FM** (FastMat & A, const Matrix & B)

Converts Newmat matrices to Fastmat matrices

Parameters: `B` (input) matrix in Newmat form
`A` (output) matrix in Fastmat form
Author: M. Storti

15.9

int **kron** (FastMat & C, FastMat const &A, FastMat const & B)

Clon of Matlab's kron. For given matrices A (nxm) and B (pxq) returns a matrix C (np x mq) formed with blocks [A(1,1)*B A(1,2)*B .... ; A(2,1)*B A(2,2)*B ; .... A(n,m)*B]

Return Value: `C=cr/n(A,B)`
Parameters:
`A` first matrix argument
`B` second matrix argument
Author: M. Storti

16

To be used in variable argument functions.

To be used in variable argument functions. Ex: fun(double,INT_ARG_LIST)

17

#define **INT_ARG_LIST_ND** (int,arg)

In definitions (without the default value)

18

#define **READ_INT_ARG_LIST** (indx)

To access the elements in the INT_ARG_LIST

19

#define **INT_VAR_ARGS**

Defines whether uses standard variadic argument lists or that defined rhtough readlisth

20

#define **FMSHV** (a)

Prints a FastMat2 matrix

21

#define **INDX_CHUNK_SIZE**

num IndxOPT LIST ;

22

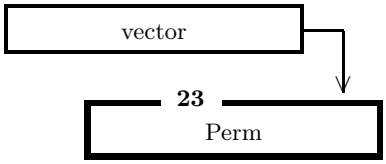
typedef FastVector<int> **Indx**

The type of indices used internally in FastMat2

23

class **Perm** : public vector<int>

Inheritance



An object of class perm stores the permutation of indices inside FastMat2

24

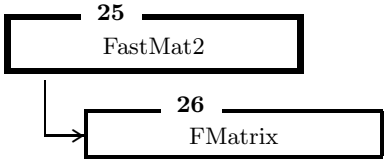
struct **OperationCount**

This stores the counters for the various kind of operations

25

class **FastMat2**

Inheritance



Public Members

Private Members

Fast matrices.

25.2

FastMat2 (void)

Default constructor

Author: M. Storti

25.3

FastMat2 (const int m, INT_VAR_ARGS)

Constructor from indices (reading with varargs).

Parameters:

`m`	(input) the number of indices.
`INT\_VAR\_ARGS`	(input) the list of dimensions (m values)

Author: M. Storti

25.4

FastMat2 (const Indx & dims_)

Constructor from indices

Parameters:

`dims\_`	(input) the vector of dimensions.

Author: M. Storti

25.5

FastMat2 (CacheCtx* ctx)

Default constructor

Author: M. Storti

25.6

FastMat2 (CacheCtx* ctx, const int m, INT_VAR_ARGS)

Constructor from indices (reading with varargs).

Parameters:

`m`	(input) the number of indices.
`INT\_VAR\_ARGS`	(input) the list of dimensions (m values)

Author: M. Storti

25.7

FastMat2 (CacheCtx* ctx, const Indx & dims_)

Constructor from indices

Parameters:

`dims\_`	(input) the vector of dimensions.

Author: M. Storti

25.8

~FastMat2 ()

Destructor

Author: M. Storti

25.9

int **size** () const

Returns the size of the matrix, ie. the product of its indices.

Return Value: `size` of matrix

25.10

int **dim** (const int jd) const

Returns the jd-th dimension

Return Value: `the` corresponding dimension
Parameters: `jd` (input) returns the jd-th dimension
Author: M. Storti

25.11

int **n** () const

Returns the number of indices

Return Value: `the` number of indices
Author: M. Storti

25.12

int **is_defined** (void) const

Flags if matrix was already defined

Return Value: `the` correponding logical value
Parameters: `void` (input) Used internally, flags if the matrix wasalready defined or not.
Author: M. Storti

25.13

void **print** (const char* s=NULL) const

Prints the matrix.

Return Value: `a` reference to the matrix.
Parameters: `s` (input) An optional string to print.
Author: M. Storti

25.14

void **print** (int rowsz, const char* s=NULL) const

Prints the matrix with a fixed rowsize.

Return Value: `a` reference to the matrix.
Parameters: `rowsz` (input) the row size
`s` (input) An optional string to print.
Author: M. Storti

25.15

void **dump** (FILE* stream=stdout, int rowsz=0) const

Prints the matrix with a fixed rowsize.

Return Value: `a` reference to the matrix.
Parameters: `rowsz` (input) the row size
`s` (input) An optional string to print.
Author: M. Storti

25.16

void **printd** (char* s=NULL)

Prints dimensions of the matrix.

Return Value: `a` reference to the matrix.
Parameters: `s` (input) An optional string to print.

Author: M. Storti

25.17

int **is_nan** ()

Check if any of the elements in the matrix are NAN.

Return Value: `1` if any element is NAN, 0 otherwise.
Author: M. Storti

25.18

FastMat2&
is (const int index, const int start=0, const int finish=0, const int step=1)

Add a range to the specified index filter. Adds the range (for j=start; j<finish; j+=step) of indices to the current filter list. If start is undefined resets the filter for this index.

Return Value: `a` reference to the matrix.
Parameters:
- `index` (input) The index to which a range in the filter isadded.
- `start` (input) the start of the range
- `finish` (input) the end of the range. Def: start
- `step` (input) the step to be used. Def: 1

Author: M. Storti

25.19

void **get_dims** (Indx & indx) const

Get filtered dims.

Parameters: `indx` (output) The dimensions of the filtered matrix.
Author: M. Storti

25.20

FastMat2& **r** (const int i, const int j=0, const int step=1)

Adds a range to the row filter. Same as is() but acts on the first index.

Return Value: `a` reference to the matrix.
Parameters: `start` (input) the start of the range
`finish` (input) the end of the range
`step` (input) the step to be used.
Author: M. Storti

25.21

FastMat2& **c** (const int i, const int j=0, const int step=1)

Adds a range to the column filter. Same as is() but acts on the second index.

Return Value: `a` reference to the matrix.
Parameters: `start` (input) the start of the range
`finish` (input) the end of the range
`step` (input) the step to be used.
Author: M. Storti

25.22

FastMat2& **ir** (const int indx, const int j=0)

Sets an index to a fixed value and reducing one dimension. The resulting filtered matrix is one dimension lower, and the corresponding index 'indx' is set to 'j'.

Return Value: `a` reference to the matrix.
Parameters: `indx` (input) the index to be restricted.
`j` (input) the value to which it is restricted.
Author: M. Storti

25.23

FastMat2& **d** (const int j1, const int j2)

Sets an index to mirror another, so that it creates a mask that lets see the diagonal part of the matrix.

Return Value: `a` reference to the matrix.
Parameters: `j1` (input) the first index
`j2` (input) the second index
Author: M. Storti

25.24

FastMat2& **rs** ()

Reset index filters for all dimensions.

Return Value: `a` reference to the matrix.
Author: M. Storti

25.25

Operations on individual elements

Names

These operations act on individual elements of the matrix.

25.25.1

FastMat2& **setel** (const Indx & indx, const double val)

Sets value at filtered position indx.

Return Value: `a` reference to the matrix.
Parameters: `indx` (input) the vector of indices defining the positionto be set.
`val` (input) the value to be set.
Author: M. Storti

25.25.2

FastMat2& **addel** (const Indx & indx, const double val)

Adds value at filtered position indx.

Return Value: `a` reference to the matrix.
Parameters: `indx` (input) the vector of indices defining the positionto be set.
`val` (input) the value to be added.
Author: M. Storti

25.25.3

FastMat2& **setel** (const double val, INT_VAR_ARGS)

Sets value at filtered position i,j,k. (INT_VAR_ARGS)

Return Value: `a` reference to the matrix.
Parameters: `val` (input) the value to be set.
`INT\_VAR\_ARGS` (input) the indices of the position to be set.
Author: M. Storti

25.25.4

FastMat2& **addel** (const double val, INT_VAR_ARGS)

Adds value at filtered position i,j,k. (INT_VAR_ARGS)

Return Value: `a` reference to the matrix.
Parameters: `val` (input) the value to be added.
`INT\_VAR\_ARGS` (input) the indices of the position to be set.
Author: M. Storti

25.25.5

FastMat2& **multel** (const double val, INT_VAR_ARGS)

Multiplies element at filtered position i,j,k. (INT_VAR_ARGS) by val.

Return Value: `a` reference to the matrix.
Parameters: `val` (input) the coefficient to multiply.
`INT\_VAR\_ARGS` (input) the indices of the position to be multiplied.
Author: M. Storti

25.25.6

double **get** (INT_VAR_ARGS) const

Returns the value at a given position.

Return Value: `the` value at that position.
Parameters: `i,j,k,l` (input) the indices of the position
Author: M. Storti

25.25.7

double **get** (const Indx & indx) const

Gets value at filtered position

Return Value: `the` value at that position.
Parameters: `indx` (input) the vector of indices to retrieve element

Author: M. Storti

25.26

FastMat2& **exc** (const int i1, const int i2)

Exchange indices (Obsolete)

Return Value: `a` reference to the matrix.
Parameters: `i1` (input) One of the indices to be exchanged
`i2` (input) The other index to be exchanged
Author: M. Storti

25.27

FastMat2& **t** (void)

Transpose (for matrix with two indices)

Return Value: `a` reference to the matrix.
Author: M. Storti

25.28

One to one operations.

Names

These operations act on pair of matrices transforming from one to the other element by element.

25.28.1

FastMat2& **set** (const FastMat2 & A)

Copy matrix.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to copy from
Author: M. Storti

25.28.2

FastMat2& **add** (const FastMat2 & A)

Add matrix.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to add
Author: M. Storti

25.28.3

FastMat2& **rest** (const FastMat2 & A)

Substract a matrix.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to substract
Author: M. Storti

25.28.4

FastMat2& **mult** (const FastMat2 & A)

25.28.5

FastMat2& **div** (const FastMat2 & A)

25.28.6

FastMat2& **rcp** (const FastMat2 & A, double c=1.)

25.28.7

FastMat2& **axpy** (const FastMat2 & A, double alpha)

Axpy operation (element by element): `(*this) += alpha * A`

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to add.
`alpha` (input)
Author: M. Storti

25.29

In-place operations.

Names

25.29.1 FastMat2&
set (const double val=0.)
Sets all the element of a matrix to a constant value. 104

25.29.2 FastMat2&

These operations perform an action on all the elements of a matrix.

25.29.1

FastMat2& **set** (const double val=0.)

Sets all the element of a matrix to a constant value.

Return Value: `a` reference to the matrix.
Parameters: `val` (input) the value to be set
Author: M. Storti

25.29.2

FastMat2& **scale** (const double val)

Scale by a constant value.

Return Value: `a` reference to the matrix.
Parameters: `val` (input) the scale factor
Author: M. Storti

25.29.3

FastMat2& **add** (const double val)

Adds constant val

Return Value: `a` reference to the matrix.
Parameters: `val` (input) the value to be added
Author: M. Storti

25.29.4

FastMat2& **rcp** (const double c=1.)

Computes reciprocal of elements. In Matlab notation `A.rcp(c)` is equivalent to `A = c./A`.

Return Value: `a` reference to the matrix.
Parameters: `c` (input) scales the reciprocal
Author: M. Storti

25.29.5

FastMat2& **fun** (scalar_fun_t* function)

Apply a function to all elements

Return Value: `a` reference to the matrix.
Parameters: `function` (input) the function to be applied
Author: M. Storti

25.29.6

FastMat2& **fun** (scalar_fun_with_args_t* function, void* user_args)

Apply a function with optional arguments to all elements

Return Value: `a` reference to the matrix.
Parameters: `function` (input) the function to be applied
`user_args` (input) additional arguments to the scalar function
Author: M. Storti

25.30

Miscellaneous utilities.

Names

25.30.1

FastMat2& **cross** (const FastMat2 & a, const FastMat2 & b)

Cross product of 'a' and 'b'. (all must be vectors of length 3).

Return Value: `a` reference to the matrix.
Parameters: `a` (input) first vector
`b` (input) second vector
Author: M. Storti

25.30.2

FastMat2& **eps_LC** ()

Sets to the Levi-Civita density tensor. This is a third order tensor ϵ_{ijk} $(3 \times 3 \times 3)$ such that ϵ_{ijk} is 1 if ijk is an even permutation of 123, -1 if it is an odd permutation and 0 otherwise. The vector cross-product and determinant p can be computed in terms of this tensor.

Return Value: `a` reference to the matrix.
Parameters: `a` (input) the value to be set in the diagonal
Author: M. Storti

25.30.3

FastMat2& **eye** (const double a=1.)

Sets to a multiple of the identity matrix.

Return Value: `a` reference to the matrix.
Parameters: `a` (input) the value to be set in the diagonal
Author: M. Storti

25.31

Product and contraction operation.

Names

25.31.1

FastMat2& **ctr** (const FastMat2 & A, const int m, INT_VAR_ARGS)

Contraction operation. Contracts pairs of indices. (generalized trace).

Return Value: `a` reference to the matrix.
Parameters: `A` (input) The matrix to take the trace
`i,j,k,l....` (input) The indices to contract and remap indices.
Author: M. Storti

25.31.2

double **trace** ()

Trace of tensor for a two index tensor.

Return Value: `the` trace of `A`
Parameters: `A` (input) The matrix to take the trace

25.31.3

FastMat2&
prod (const FastMat2 & A, const FastMat2 & B, const int m, INT_VAR_ARGS)

Product operation. `(*this) = A * B` (generalized by index contraction)

Return Value: `a` reference to the matrix.
Parameters: `A` (input) first natrix
`B` (input) second natrix
`i,j,k,l...` (input) indices to contract and remap.
Author: M. Storti

25.31.4

FastMat2& **kron** (const FastMat2 & A, const FastMat2 & B)

Kronecker product (also called Schur product) If A is n x m and B is p x q, then kron returns a matrix which is np x mq, and where each p x q block is proportional to B. Only implemented for two-dimensional matrices so far.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) first matrix
`B` (input) second matrix
Author: M. Storti

25.32

Sum operations (sum over indices), `max`, `sum_`, ...

Names

These operations give a reduced matrix by applying some reduce operation (sum, max, min etc...) over all the contracted indices. For instance if A has 4 indices then `B.sum(A,2,-1,-1,1)` is equivalent to $B_{ij} = \sum_{kl} A_{jkli}$

25.32.1

FastMat2& **sum** (const FastMat2 & A, const int m=0, INT_VAR_ARGS)

Sum over all selected indices.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to contract
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.2

FastMat2& **sum_square** (const FastMat2 & A, const int m=0, INT_VAR_ARGS)

Sum of squares over all selected indices.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to contract
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.3

FastMat2& **sum_abs** (const FastMat2 & A, const int m=0, INT_VAR_ARGS)

Sum of absolute values all selected indices.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to contract
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.4

FastMat2&
norm_p (const FastMat2 & A, double p, const int m=0, INT_VAR_ARGS)

Norm p of matrix (per column). $(\sum_j |a_j|^p)^{\frac{1}{p}}$

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to contract
`p` (input) exponent of norm
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.5

FastMat2& **norm_p** (const FastMat2 & A, int p, const int m=0, INT_VAR_ARGS)

Norm `p` of matrix (per column) for `p` integer (more efficient than the version for `p` real) $(\sum_j |a_j|^p)^{\frac{1}{p}}$

Return Value: `a` reference to the matrix.
Parameters:
`A` (input) matrix to contract
`p` (input) exponent of norm
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.6

FastMat2& **norm_2** (const FastMat2 & A, const int m=0, INT_VAR_ARGS)

Norm `2` of matrix (per column) (more efficient than the version for `p` generic) $sqrt(\sum_j |a_j|^2)$

Return Value: `a` reference to the matrix.
Parameters:
`A` (input) matrix to contract
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.7

FastMat2& **min** (const FastMat2 & A, const int m=0, INT_VAR_ARGS)

Minimum over all selected indices.

Return Value: `a` reference to the matrix.
Parameters:
`A` (input) matrix to contract
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.8

FastMat2& **max** (const FastMat2 & A, const int m=0, INT_VAR_ARGS)

Maximum over all selected indices.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to contract
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.9

FastMat2& **min_abs** (const FastMat2 & A, const int m=0, INT_VAR_ARGS)

Min of absolute value over all selected indices.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to contract
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.10

FastMat2& **max_abs** (const FastMat2 & A, const int m=0, INT_VAR_ARGS)

Min of absolute value over all selected indices.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) matrix to contract
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.32.11

class **Fun2**

Public Members

Private Members

Generic associative two arg function.

25.32.11.2

virtual void **pre** ()

Called prior to a stride is processed

25.32.11.3

virtual void **pre_all** ()

Called prior to all strides

25.32.11.4

virtual double **fun2** (double x, double v)

called after iteration `val = fun2(a_ij, val)` for all elements in a stride

25.32.11.5

virtual void **post** ()

May be run after each stride is processed

25.32.11.6

virtual void **post_all** ()

May be run after all strides are processed

25.32.11.7

double **v** ()

Returns the cumulated value

25.32.11.1

double **val**

Cumulated value

25.32.12

FastMat2&
assoc (const FastMat2 & A, Fun2 &f, const int m=0, INT_VAR_ARGS)

Associated value over all selected indices.

Return Value: `a` reference to the result matrix.
Parameters: `A` (input) matrix to contract
`i,j,k,l...` (input) indices that define indices to be contracted
Author: M. Storti

25.33

Sum operations over all indices, max, sum_, ...

Names

These operations give a scalar by applying some reduced operation (sum, max, min etc...) over all indices. For instance if A has 4 indices then `b = sum_all(A)` returns a double $b = \sum_{kl} A_{jkli}$

25.33.1

double **sum_all** () const

Sum over all indices.

Return Value: `the` result of the operation

Author: M. Storti

25.33.2

double **sum_square_all** () const

Sum of squares over all indices.

Return Value: `the` result of the operation
Author: M. Storti

25.33.3

double **sum_abs_all** () const

Sum of absolute values over all indices.

Return Value: `the` result of the operation
Author: M. Storti

25.33.4

double **norm_p_all** (const double p) const

Norm p over all indices.

Return Value: `the` result of the operation
Author: M. Storti

25.33.5

double **norm_2_all** () const

Norm 2 over all indices.

Return Value: `the` result of the operation
Author: M. Storti

25.33.6

double **norm_p_all** (const int p=2) const

Norm p over all indices for integer p.

Return Value: `the` result of the operation
Author: M. Storti

25.33.7

double **min_all** () const

Minimum over all indices.

Return Value: `the` result of the operation
Author: M. Storti

25.33.8

double **max_all** () const

Maximum over all indices.

Return Value: `the` result of the operation
Author: M. Storti

25.33.9

double **min_abs_all** () const

Minimum absolute value over all indices.

Return Value: `the` result of the operation
Author: M. Storti

25.33.10

double **max_abs_all** () const

Maximum absolute value over all indices.

Return Value: `the` result of the operation
Author: M. Storti

25.34

FastMat2& **reshape** (const int m, INT_VAR_ARGS)

Reshapes a matrix. New and old dimensions must be compatible.

Return Value: `a` reference to the matrix.
Parameters: `i,j,k,l` (input) the new dimensions
Author: M. Storti

25.35

FastMat2& **resize** (const int m, INT_VAR_ARGS)

Resizes the matrix (destroying the information). New and old dimensions might not be the same.

Return Value: `a` reference to the matrix.
Parameters: `i,j,k,l` (input) the new dimensions
Author: M. Storti

25.36

FastMat2& **resize** (const Indx &indx)

Resizes the matrix (destroying the information). New and old dimensions might not be the same.

Return Value: `a` reference to the matrix.
Parameters: `indx` (input) the shape of the matrix
Author: M. Storti

25.37

FastMat2& **clear** ()

Resizes to one dimension ans zero elements

25.38

FastMat2& **diag** (FastMat2 & A, const int m, INT_VAR_ARGS)

Sets vector to diagonal part

Return Value: `a` reference to the matrix.
Parameters: `A` (input) the matrix to take the diagonal
`i,j,k,l...` (input) the indices defining the operation
Author: M. Storti

25.39

Export/Import operations

Names

25.39.1

FastMat2& **set** (const Matrix & A)

Copies to argument from Newmat matrix.

Return Value: `a` reference to the matrix.
Parameters: `A` (input) the matrix to be copied
Author: M. Storti

25.39.2

FastMat2& **set** (const double* a)

Copy from array of doubles.

Return Value: `a` reference to the matrix.
Parameters: `a` (input) the array of indices from where to copy
Author: M. Storti

25.39.3

const FastMat2& **export_vals** (double* a) const

exports to a double vector

Return Value: `a` reference to the matrix.
Parameters: `a` (output) array doubles to where export
Author: M. Storti

25.39.4

FastMat2& **export_vals** (double* a)

exports to a double vector

Return Value: `a` reference to the matrix.
Parameters: `a` (output) array doubles to where export
Author: M. Storti

25.39.5

const FastMat2& **export_vals** (Matrix & A) const

Exports to a Newmat matrix.

Return Value: `a` reference to the matrix.
Parameters: `A` (output) the Newmat matrix to where export to
Author: M. Storti

25.39.6

FastMat2& **export_vals** (Matrix & A)

Exports to a Newmat matrix.

Return Value: `a` reference to the matrix.
Parameters: `A` (output) the Newmat matrix to where export to
Author: M. Storti

25.39.7

double* **storage_begin** ()

Returns a pointer to the start of the storage matrix.

Return Value: `s` as mentioned
Author: M. Storti

25.40

double **det** (void) const

Determinant

Return Value: `the` determinant of the matrix.
Author: M. Storti

25.41

double **detsur** (FastMat2* nor=NULL)

For a matrix of size `A` of `(n-1)* n` computes the determinant `sqrt(det(A*A'))`. This is useful when integrating on surfaces in 3D and lines in 2D. If `nor` is not null then computes `nor` as the normal (not necessarily a unit vector) to the surface.

Return Value: `determinant` of the surface Jacobian
Parameters: `nor` (ouput) the normal to the surface

25.42

FastMat2& **inv** (const FastMat2 & A)

The inverse of a matrix. `(*this) = inverse of A`

Return Value: `a` reference to the matrix.
Parameters: `A` (input) the matrix to take the inverse
Author: M. Storti

25.43

FastMat2& **eig** (const FastMat2 & A, FastMat2 &VR)

Solve the eigenvalue problem for non-symmetric matrix `A` and compute right eigenvectors. (see `eig(const FastMat2 & A, FastMat2 *VR=NULL,...)` below).

Return Value: `s` a reference to a vector containing the eigenvalues.
Parameters: `A` (input) the matrix to take the eigenvalues
`VR` (output) the matrix with the right eigenvectors

Author: M. Storti

25.44

FastMat2& **eig** (const FastMat2 & A, FastMat2 &VR, FastMat2 &VL)

Solve the eigenvalue problem for non-symmetric matrix `A` and compute right and left eigenvectors. (see `eig(const FastMat2 & A, FastMat2 *VR=NULL,...)` below).

Return Value: `s` a reference to a vector containing the eigenvalues.
Parameters:
`A` (input) the matrix to take the eigenvalues
`VR` (output) the matrix with the right eigenvectors
`VL` (output) the matrix with the left eigenvectors
Author: M. Storti

25.45

FastMat2&
eig (const FastMat2 & A, FastMat2* VR=NULL, FastMat2* VL=NULL)

Solve the eigenvalue problem for non-symmetric matrix `A`. `A` has to be square. If `A` is of size `m x m` then it returns the eigenvalues in a matrix of size `2 x m` containing the real and imaginary part of the eigenvalues. Both right and left eigenvectors can be computed independently. Based on the `DGEEV LAPACK` routine.

Return Value: `s` a reference to a vector containing the eigenvalues.
Parameters:
`A` (input) the matrix to take the eigenvalues
`VR` (output) the matrix with the right eigenvectors
`VL` (output) the matrix with the left eigenvectors
`crev` (input) compute the right eigenvectors flag
`clev` (input) compute the left eigenvectors flag
Author: M. Storti

25.46

FastMat2& **seig** (const FastMat2 & A)

Solve the eigenvalue problem for symmetric matrix `A`. Note: `A` is NOT checked for symmetry. `A` has to be square. This routine may be more faster and accurate tha the non-symmetric version. Based on the `DSYEV LAPACK` routine.

Return Value: `s` a reference to a vector containing the eigenvalues.
Parameters:
`A` (input) the matrix to take the eigenvalues

Author: M. Storti

25.47

FastMat2&
seig (const FastMat2 & A, FastMat2 &V, int compute_eigen_vectors=1)

Solve the eigenvalue problem for symmetric matrix `A`. Note: `A` is NOT checked for symmetry. `A` has to be square. Based on the `DSYEV LAPACK` routine.

Return Value: `s` a reference to a vector containing the eigenvalues.
Parameters:

`A`	(input) the matrix to take the eigenvalues
`V`	(output) the matrix with the eigenvectors
`compute_eigen_vectors`	(input) flags whetherto compute the eigenvectors or not

Author: M. Storti

25.48

operator double () const

Converts to double, for zero dimension matrices.

Return Value: `a` double that is the only element of the matrices.
Author: M. Storti

25.49

Static cache operations

Names

25.49.1

static void **activate_cache** (FastMatCacheList* cache_list_=NULL)

Activates use of the cache

Parameters: `cache_list_` (input) the cache list root to activate.
Author: M. Storti

25.49.2

static void **deactivate_cache** (void)

Deactivates use of the cache

Author: M. Storti

25.49.3

static void **reset_cache** (void)

Resets the cache. To be used after each iteration loop.

Author: M. Storti

25.49.4

static void **void_cache** (void)

Voids the cache. Frees the memory used by the cahe list after processing.

Author: M. Storti

25.49.5

static void **branch** (void)

Creates a branch point.

Parameters: `(input)`
Author: M. Storti

25.49.6

static void **choose** (const int j)

Follows a branch.

Parameters: `j` (input) the number of branch to follow.
Author: M. Storti

25.49.7

static void **leave** (void)

Leaves the current branch.

Author: M. Storti

25.49.8

static double **operation_count** (void)

Computes the total number of operations in the cache list. Currently all operations counts as one. In the future we will give weights to each type of operation.

Return Value: `the` total number of operations.
Author: M. Storti

25.49.9

static void **print_count_statistics** ()

Print statistics about the number of operations of each type in the current cache list.

Author: M. Storti

25.49.10

static void **get_cache_position** (FastMatCachePosition & pos)

Gets actual position in cache.

Author: M. Storti

25.49.11

static void **jump_to** (FastMatCachePosition &pos)

Jumps to a given position in the cache-list.

Author: M. Storti

25.49.12

static void **resync_was_cached** (void)

Recomputes the was_cached variable

Author: M. Storti

25.50

Initializes the matrix entries with integers

Initializes the matrix entries with integers

25.51

identifying positions, ie.

identifying positions, ie. a 2x3 matrix would

25.52

be initialized to [11,12,13;21,22,23].

be initialized to [11,12,13;21,22,23]. Usually

25.1

static int **cache_dbg**

Controls debugging

25.53

int **storage**

Total storage. Should be the product of 'dims'.

25.54

double* **store**

The array of values. Should be of size 'storage'.

25.55

DimArray **dims**

dimensions

25.56

IndexFilter* **dims_p**

Pointer to dims, as an array

25.57

int **n_dims**

Size of dims array

25.58

Perm **perm**

Permutation of indices(transposing)

25.59

Indx **set_indx**

Fixed indices

25.60

Indx **absindx**

auxiliary vector to put absolute indices

25.61

int **defined**

was the matriz defined?

25.62

void **define_matrix** (void)

creates storage and freezes dimensions

25.63

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

returns value at position i,j

25.64

double* **location_abs** (const Indx & indx) const

returns address of absolute position indx[0],indx[1],

25.65

double* **location** (const Indx & indx) const

returns address of filtered position indx[0],indx[1],

25.66

void **create_from_indx** (const Indx & dims_)

used in constructors

25.67

void **print2** (const Indx & indxp, const Indx & fdims) const

auxiliary. prints matrices with 2 indices.

25.68

void **print1** (const Indx & indxp, const Indx & fdims) const

auxiliary. prints matrices with 1 indices.

25.69

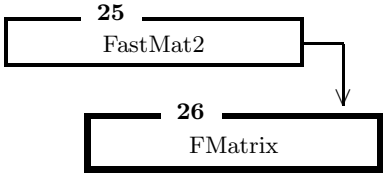
void **get_addresses** (Indx perm, Indx Afdims, vector<double* > &ap) const

Internally used by prod()

26

class **FMatrix** : public FastMat2

Inheritance



For 2 indices matrices

27

int **inc** (Indx & indx, const Indx &dims)

increment an index

28

void **read_int_list** (const int m, va_list v, Indx* indx)

Read a list of integers from variable arguments in a vector

29

#define **FSHV** (name)

prints a FastMat2 matrix for debugging

30

Obtains the amount of memory used by a matrix from its vector of

Obtains the amount of memory used by a matrix from its vector of

31

class **FileStack**

Public Members

Private Members

Allows reading from a set of files with preprocessing capabilities. Supports file inclusion, comments and continuation lines.

Author: M. Storti

31.6

int **get_line** (char* & line)

Reads a line from the file stack.

Return Value: `error` code
Parameters: `line` the line that has been read
Author: M. Storti

31.7

const char* **line_read** (void) const

Gives a pointer to the last line read.

Parameters: `line` the line that has been read
Author: M. Storti

31.8

int **line_number** () const

Position in file at top.

Return Value: `line` number

31.9

const char* **file_name** () const

Name of current file at top.

Return Value: `name` of file at top

31.10

void **print** (void) const

Prints the current file stack.

31.11

int **unread_line** (const char* line)

Unreads a line

Author: M. Storti

31.12

void **close** ()

Closes the file-stack.

Author: M. Storti

31.13

int **open** (const char* filename)

Opens a file.

Author: M. Storti

31.14

FileStack (const char* filename)

Constructor from the name of the main file.

Parameters: `filename` the name of the main file.
Author: M. Storti

31.15

~FileStack (void)

Destructor.

Author: M. Storti

31.16

int **ok** (void)

Check if the file has been opened correctly

Return Value: `error` code

31.17

void **set_echo_stream** (FILE* echo_s)

Set stream where to write echo lines

Parameters: `echo_s` (input) the output stream

31.1

Darray* **file_stack**

stack of file pointers

31.2

FILE* **file_at_top**

the file at the top of the stack

31.3

int **echo**

Flag indicates whether lines are output to the echo stream

31.4

Darray* **read_buffer**

the pile of unread lines

31.5

Autostr* **buf**

The current line buffers

32

getprop package

Names

32.1

void **load_props** (double* propel, int* elprpsindx, int nprops, double* elemprops)

Fast load of element properties. It transparently loads properties defined 'per element' or globally for the elemenset in the properties hash table. You should call DEFPROP(prop1); DEFPROP(prop2); etc.. before entering the element loop, and inside the element loop you call load_props(...), then you can use the values as propel[prop1_indx], propel[prop2_indx], etc...

Parameters:

`propel`	a double working array defined by the user.
`elprpsindx`	integer working array defined by the user.
`nprops`	number of properties to be 'fast loaded'
`elemprops`	array of 'per element' properties

Author: M. Storti

32.2

int
get_double (TextHashTable* thash, const char* name, double* retval, int defval=0, int n=1)

Gets a double value from the hash table.

Parameters:

`thash`	pointer to the elemset hash table.
`name`	name of the variable
`retval`	the value read from the table. Set to the defaultvalue if 'defval=0'.
`n`	number of doubles to be read

Author: M. Storti

32.3

int
get_string (const TextHashTable* thash, const char* name, string &ret, int defval=0, int n=1)

Gets a string value from the hash table.

Parameters:

`thash`	pointer to the elemset hash table.
`name`	name of the variable
`retval`	the value read from the table. Set to the defaultvalue if 'defval=0'. Strings may be enclosed in double quotes to further
`n`	number of integers to be read

Author: M. Storti

32.4

int
get_int (const TextHashTable* thash, const char* name, int* retval, int defval=0, int n=1)

Gets an integer value from the hash table.

Parameters:

`thash`	pointer to the elemset hash table.
`name`	name of the variable
`retval`	the value read from the table. Set to the defaultvalue if 'defval=0'. If the entry is found but not value assigned,then assign 1 (to be used as a 'get_flag()' function).
`n`	number of integers to be read

Author: M. Storti

32.5

int
get_prop (int & iprop, GHashTable* props, TextHashTable* thash, int* elprpsindx, double* propel, const char* name, int n)

Prepares for a 'fast load' of properties.

Author: M. Storti

33

class **GPdata**

Public Members

Contains information on Gauss integration points (GP).

Author: M. Storti

33.1

int **initialized**

Flags if the object has been initialized

33.2

Matrix* **dshapexi**

Gradient of shape functions with respect to master element coordiantes at each GP

33.3

* **dshapex**

gradient of shape functions with respect to global coordiantes at each GP

33.4

FastMat** **FM_dshapexi**

For the FastMat version

33.5

FastMat2** **FM2_dshapexi**

For the FastMat2 version

33.6

RowVector* **shape**

Shape functions

33.7

double* **wpg**

Integrations weights

33.8

GPdata (const char* geom, int ndim, int nel, int npg, int mat_version=GP_NEWMAT)

Constructor.

Parameters:	`geom`	(input) string defining the geometry, currently may becartesian<n>d, with n=1,2,3, triangle or tetra.
	`ndim`	(input) number of dimensions. (This may bedifferent from the problem dimension. Typically it may be onedimension lower for surface elements.)
	`nel`	(input) number of nodes per element
	`npg`	(input) number of gauss points
	`mat_version`	(input)
Author:	M. Storti	

33.9

~GPdata (void)

Destructor

33.10

int **npg**

number of Gauss points.

33.11

int **mat_version**

flags whether Newmat or FastMat

33.12

double **master_volume**

The volume of the reference (master) element

33.13

double **wpg_sum**

Sum of Integrations weights

33.14

DXSplit **splitting**

This represents how the element is mapped onto Data Explorer connections.

34

int
cartesian_2d_shape (RowVector &shape, Matrix &dshapexi, double xipg, double etapg)

Provides the shape functions and gradients for the bilinear quadrangle.

Parameters:	`shape`	(output) the shape function of the different nodes atthe Gauss Points.
	`dshapexi`	(output) the gradients of the shape functions withrespect to master element coordinates.
	`xipg`	(input) xi coordinate of the Gauss point.
	`etapg`	(input) eta coordinate of the Gauss point.
Author:	M. Storti	

35

idmap package

Names

35.1

typedef map<int,double> **row_t**

Stores rows of the Q matrix

35.2

void **axpy** (row_t &y, double const a, const row_t &x)

'daxpy' type operation (y <- y + a*x) for 'idmap' rows.

Parameters:

`y` (input/output) row to be modified.
`a` (input) scalar coeficient.
`x` (input) row to be added.

Author: M. Storti

35.3

typedef set<int> **col_t**

Stores columns of the Q matrix

36

class **idmap**

Public Members

Private Members

"idmap" class stores sparse m x n matrices that are "close" to a permutation.

36.1

idmap (int m, int n, int* ident, int* iident)

Constructor from an initial permutation given by ident and iident.

Parameters:

`m`	row dimension of Q
`n`	column dimension of Q
`ident`	integer array (permutation).
`iident`	integer array (permutation).

Author: M. Storti

36.2

idmap (int m, map_type mt)

Constructor for a square (m=n) map to identity or null.

Parameters:
`m` (input) dimension of matrix
`ival` (input) may be IDENTITY_MAP or NULL_MAP

Author: M. Storti

36.3

void **set_elem** (const int i, const int j, const double val=0.)

Sets element i,j to val.

Parameters:
`i` row index
`j` column index
`val` coefficient to be entered at position i,j. (default=0)

Author: M. Storti

36.4

void **del_row** (const int i)

Deletes the i-th row.

Parameters:
`i` index of row to be deleted

Author: M. Storti

36.5

void **row_set** (const int i, const row_t &row)

Sets the i-th row to row.

Parameters:
`i` row index
`row` row to be entered in the i-th position

Author: M. Storti

36.6

void **get_row** (const int i, row_t &row)

Gets the i-th row.

Parameters:
- `i` index of row to be retrieved
- `row` retrieved row (output)

Author: M. Storti

36.7

void **get_row** (const int i, IdMapRow &row)

overloaded in order to run faster

36.8

void **get_col** (const int j, row_t &col)

Gets the j-th column to col.

Parameters:
- `j` column index
- `col` column to be insert in j-th position

Author: M. Storti

36.9

void **get_val** (const int i, const int j, double &val)

Gets the i,j value Q_ij.

Parameters:
- `i` row index
- `j` column index
- `val` coefficient Q_ij

Author: M. Storti

36.10

void **del_col** (const int j)

Deletes a column.

Parameters: `j` (input) column indx
Author: M. Storti

36.11

void **column_set** (const int j, row_t &col)

Sets the j-th column to col.

Parameters: `j` (input) column index
`col` (input) column to be entered in the j-th position
Author: M. Storti

36.12

void **print** (const char* s = NULL)

Prints the Q matrix by rows (in sparse form).

Parameters: `s` optional string
Author: M. Storti

36.13

void **print_by_col** (const char* s=NULL)

Prints the Q matrix by cols.

Author: M. Storti

36.14

void **check** ()

Checks internal consistency of idmap. Checks that any entry (by row) has the corresponding entry by column, and viceversa. Also, that all entries in 'special rows' are non-zero.

Parameters: `s` optional string
Author: M. Storti

36.15

void **solve** (double* x, double* y)

Solves an equation of the form Q x = y for x. Assumes that rank(Q) = n = dim(x).

Author: M. Storti

36.16

void **get_block_matrix** (int j, set<int> &iindx, set<int> &jindx, Matrix &qq)

Extracts the matrix connected to a given column 'j'. This routine is the key part for solving a linear system associated with the underlying matrix. (In a future I will write a similar one for index row).

Parameters:
`j` (input) column index to get the connected matrix
`iindx` set of row indices connected to `j`
`jindx` set of column indices connected to `j`
`q` Newmat matrix returned. Entry (k,l) in qcorresponds to the k-th entry in iindx and the l-th entry in jindx

Author: M. Storti

36.17

void **remap_cols** (int* perm)

Redefines the matrix through a reordering of the columns.

Parameters: `perm` (input) The vector that remaps columns. new_j_indx = perm[old_j_indx-1]

Author: M. Storti

36.18

int **m**

row and column dimension

36.19

int* **ident**

pointers row <-> column

36.20

row_map_t* **row_map**

for 'special' rows, maps row index to a pointer to a 'special row'

36.21

col_map_t* **col_map**

for 'special' columns, maps column index to a pointer 'special column'

37

Class 'idmap' utility functions

Names

37.1

void **print** (col_t &col, const char* s=NULL)

prints a column with an optional string

37.2

void **print** (row_t &row, const char* s=NULL)

prints a row with an optional string

37.3

void **erase_null** (row_t &row)

voids a row

38

#define **ARG_LIST** (type,name,default)

Defines and argument list to be used as a variable argument list. Example: ARG_LIST(int,arg,0) expands to int arg_0 = 0, int arg_1 = 0, int arg_2 = 0 ... The current maximum number of args is $maxargs=. This is set in 'readlist.eperl'.

Parameters:

`type`	(input) the type of the variable argument list (int,double, etc...)
`name`	(input) the name of the arguments.
`default`	(input) the default value.

Author: M. Storti

39

#define **ARG_LIST_ND** (type,name)

Same, but without the default value

40

#define **READ_ARG_LIST** (name,indx,default,exit_label)

Reads the argument list in a FastVector of the corresponding type. Example: READ_ARG_LIST(name,indx,default,exit_label) This is set in 'readlist.eperl'.

Parameters:

`name`	(input) the name of the arguments.
`indx`	(input) the FastVector where arguments are stored.
`default`	(input) the default value.
`exit\_label`	(input) the exit label to be generated. Usually: EXIT

Author: M. Storti

41

read_mesh package

Names

41.1

struct **props_hash_entry**

This struct contains the info for the 'props' hash table which gives the position in the per-element props table for a given text.

Parameters:

`position`	position (column) in the table
`width`	number of scalars

Author: M. Storti

41.2

void **bless_elemset** (char* type, Elemset* & elemset)

Bless elemset with the given type (derived class).

Parameters:

`type`	type of elemset
`elemset`	to be blessed

Author: M. Storti

41.3

void **bless_elemset0** (char* type, Elemset* & elemset)

Bless elemset with the given type (derived class) for generic classes (not belonging to a specific application).

Parameters:

`type`	type of elemset
`elemset`	to be blessed

Author: M. Storti

41.4

int
read_mesh (Mesh* & mesh, char* fcase, Dofmap* & dofmap, int & neq, int size, int myrank)

reads mesh and returns the corresponding mesh and dofmap.

Parameters:

`mesh`	(output) the mesh that has been read
`fcase`	(input) name of the file to be read
`dofmap`	(output) the dofmap read
`neq`	(output) total number of unknowns.
`size`	(input) number of processors running
`myrank`	(input) this processor number (base 0)
`x`	(output) reduced (MPI) vector prototype
`xseq`	(output) reduced (sequential) vector prototype

Author: M. Storti

42

int
print_some (const char* filename, const State &s, Dofmap* dofmap, set<int> & node_list)

For a given state vector, prints the state for some nodes.

Parameters:	`(input)`	filename file where to write the vector. May contain relative directories.
	`s`	(input) State vector
	`dofmap`	(input) corresponding dofmap
	`node_list`	(input) set of nodes to print.
Author:	M. Storti	

43

int
print_some_file_init (TextHashTable* thash, const char* print_some_file, const char* save_file_some, set<int> &node_list, int save_file_some_append)

Initializes the print_some saving mechanism. Basically reads the nodes to be printed at each time step.

Parameters:	`thash`	(input) the text hash from where to get properties.
	`print_some_file`	(input) the file where to read the list ofnodes
	`save_file_some`	(input) the file where to write the valuesat those nodes (this is not used currently)
	`node_list`	(output) the list of nodes
	`save_file_some_append`	(input) Access mode to the "some"file. If 0 rewind file. If 1 append to previous results.
Author:	M. Storti	

44

int **readval** (int &rflag, char* line, double &val)

Reads a double value token from a string.

Return Value: `boolean` value indicating wether a value has been read ordnot.

Parameters:

`rflag`	(input/output) First time pass initially 0 and thensets to 1.
`line`	(input) the string to be read
`val`	(input) the double value to be read

Author: M. Storti

45

int **readval** (int &rflag, char* line, int &val)

Reads an int value token from a string.

Return Value: `boolean` value indicating wether a value has been read ordnot.

Parameters:

`rflag`	(input/output) First time pass initially 0 and thensets to 1.
`line`	(input) the string to be read
`val`	(output) the int value to be read

Author: M. Storti

46

void
print_vector_rota (const char* filenamepat, const Vec x, const Dofmap* dofmap, const TimeData* time_data, const int j, const int nsave, const int nrec, const int nfile)

Prints a vector with "rotary save" mechanism. Prints a vector to a file

Parameters:	`filenamepat`	(input) The pattern that generates thefilenames to which the vectors are written. Must contain `%d`
	`x`	(input) The vector to be written
	`dofmap`	(input) The dofmap of the problem. (Gives values forfixed node/field combinations - Dirichlet boundary conditions.)
	`time_data`	(input) The corresponding time instant. (Gives values forfixed node/field combinations - Time dependent Dirichlet boundaryconditions .)
	`j`	(input) The actual time step
	`nrec`	(input) The number of records allowed in each file.
	`nfile`	(input) The number of files allowed.
	`filenamepat`	(input) pattern to generate the filename (must contain a %d)
	`x`	(input) the vector to be printed
	`dofmap`	(input) the Dofmap of the problem
	`time_data`	(input) an external parameter in order to computeexternal boundary conditions, etc...
	`j`	(input) the time step
	`nsave`	(input) save each nsave time steps
	`nrec`	(input) save nrec records per file
	`nfile`	(input) rotate through 'nfile' files
Author:	M. Storti	

47

void
parse_props_line (const char* line, vector<string> &prop_name, vector<int> &prop_len)

Parses a line of props of the form 'prop1[len1] prop2[len2] ' Some properties may not have a length in which case we asume 1.

Parameters:

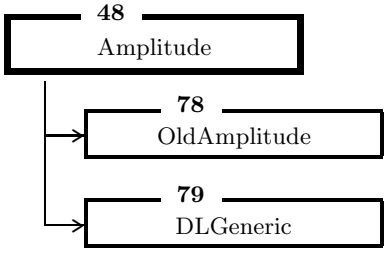
`line`	(input) the props line
`prop_name`	(output) a vector of strings containing theproperty names
`prop_len`	(output) a vector of ints containing the length ofthe property

Author: M. Storti

48

class **Amplitude**

Inheritance



Public Members

An amplitude is basically a function object that returns the Dirichlet value for a given time. For instance function 'sine' with constant parameters 'omega' and 'phase', depending on time.

Author: M. Storti

48.1

virtual double **eval** (const TimeData* time_data)

Eval the amplitude of the function at this time.

48.2

virtual double **eval** (const TimeData* time_data, int node, int field)

Eval the amplitude of the function at this time (needs node and field)

48.3

virtual int **needs_dof_field_q** ()

Callback function defined by the user – returns whether this amplitude function needs to be passed the node/field combination.

48.4

virtual void **init** (TextHashTable* t)

Initializes the object. Table t should be deleted if not incorporated in the created object. If it is deleted set it to NULL.

48.5

virtual void **print** (void) const

prints the amplitude entry.

49

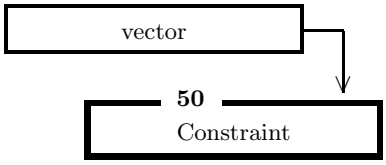
dofmap.

This is not documented since it will suffer from major revision when using the idmap class

50

class **Constraint** : public vector<constraint_entry>

Inheritance



Public Members

Constraints are set passing a list of node, field, coefficients. This class implements such lists.

50.1

void **add_entry** (int node, int field, double coef)

Add an entry to the list.

Parameters:

`node`	(input) node number
`field`	(input) field number
`coef`	(input) coefficient in the linear combination

Author: M. Storti

50.2

void **empty** ()

Empties the list of the linear combination.

Author: M. Storti

51

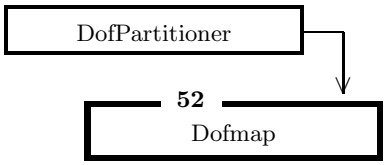
class **fixation_entry**

Defines a fixation (typically Dirichlet bc's). The double is the "spatial" amplitude and the int is a pointer (or something) that describes the temporal evolution.

52

class **Dofmap** : public DofPartitioner

Inheritance



Public Members

Private Members

Stores the mapping between the node/field representation and unknowns in MPI vectors and matrices.

Author: M. Storti

52.2

MPI_Comm **comm**

MPI communicator (default: PETSCFEM_COMM_WORLD)

52.3

int **nnod**

number of nodes

52.4

int **neq**

number of equations

52.5

int **dof1**

start range of dof's for this processor. Dof's on this processor are dof1<= dof <=dof2. (dof in base 1).

52.6

int **dof2**

end range of dof's for this processor. (See arg dof1).

52.7

int **neqf**

number of independent fixations (entries in the 'fixed' table)

52.8

int **neqtot**

total number of independent variables (fixed + free) = column dimension of Q

52.9

int **ndof**

number of degrees of freedom per node

52.10

int* **ident**

identifier of whether the node/field pair is free, fixed, or special. (This will be included in an idmap object in the future.)

52.11

vector<int> * **ghost_dofs**

The STL vector containing ghost_dofs (ordered, base 0)

52.12

idmap* **id**

this will replace ident in a future

52.13

Darray* **fixa**

points to fixations

52.14

Darray* **q**

points to special rows

52.15

vector<fixation_entry> **fixed**

stores fixations. 'fixa' will change to this. In the Q matrix, column indices greater than 'neq' point to fixation j-neq-1

52.16

flags whether the dofmap is synchronized or not

flags whether the dofmap is synchronized or not

52.17

int* **neqproc**

number of lines (unknowns) in processor 'myrank'

52.18

int **size**

number of processors

52.19

float* **tpwgts**

weight per processor

52.20

int* **npart**

node partitioning (as returned by Metis)

52.21

VecScatter* **ghost_scatter**

Scatter to convert to sequential vector with ghost values

52.22

VecScatter* **scatter_print**

scatter to print

52.23

void **get_row_free** (int const & node, int const & kdof, row_t &row) const

Gets a row of the mapping matrix, only free (jeq<=neq) dofs are returned.

Parameters:

`node`	(input) the node number to get the row
`kdof`	(input) the field number to get the row
`row`	(output) the rwtrieved row

Author: M. Storti

52.24

void **get_row** (int const & node, int const & kdof, row_t &row) const

Gets a row of the maping matrix.

Parameters:

`node`	(input) the node number to get the row
`kdof`	(input) the field number to get the row
`row`	(output) the retrieved row

Author: M. Storti

52.25

void
get_row (int node, int kdof, int &ndof, const int** dof, const double**coef) const

Fast version. Uses an internal array. ADD ARG DOC ...

52.26

void **row_set** (const int & node, const int & kdof, const row_t &row)

Sets a row of the maping matrix.

Parameters:
- `node` (input) the node number to get the row
- `kdof` (input) the field number to get the row
- `row` (input) the retrieved row

Author: M. Storti

52.27

double
get_dofval (const int & jeq, double const* sstate, double const* ghost_vals, const TimeData* time_data) const

Gets a dof value from a state vector scattered to this processor.

Return Value:
- `the` double value corresonding to that dof

Parameters:
- `jeq` (input) the column index to be retrieved. May be free(1 < jeq <= neq) or fixed (neq < jeq <= neqtot)
- `sstate` (input) the state vector
- `ghost_vals` (input) double array containing scattered ghost values
- `time_data` (input) a pointer to a struct that tipically is a time

Author: M. Storti

52.28

double
get_dofval (const int & jeq, const double* sstate, const TimeData* time_data) const

Gets a dof value from state vector and scattered ghost values.

Return Value: `the` double value corresonding to that dof

Parameters:

`jeq`	(input) the column index to be retrieved. May be free(1 < jeq <= neq) or fixed (neq < jeq <= neqtot)
`sstate`	(input) the state vector containing all values,scattered to processor 0.
`time_data`	(input) a pointer to a struct that tipically is a time

Author: M. Storti

52.29

int
get_nodal_value (const int & node, const int & kdof, double const* sstate, double const* ghost_vals, const TimeData* time_data, double & val) const

Gets a node/field value from state vector.

Parameters:

`node`	(input) the node number to get the row
`kdof`	(input) the field number to get the row
`sstate`	(input) the state vector
`ghost_vals`	(input) double array containing scattered ghost values
`time_data`	(input) a pointer to a struct that tipically is a time
`val`	(output) the double value corresonding to thisnode/field pair.

Author: M. Storti

52.30

int
get_nodal_value (const int & node, const int & kdof, const double* sstate, const TimeData* time_data, double & value) const

Gets a node/field value from state vector.

Parameters:

`node`	(input) the node number to get the row
`kdof`	(input) the field number to get the row
`sstate`	(input) the state vector (scattered to processor0.
`time_data`	(input) a pointer to a struct that tipically is a time
`val`	(output) the double value corresonding to thisnode/field pair.

Author: M. Storti

52.31

void **set_fixation** (int node, int kdof, double val)

Sets a fixation.

Parameters:
`node` (input) the node number to set
`kdof` (input) the field number to set
`val` (input) the value to be set.

Author: M. Storti

52.32

int **edof** (const int node, const int field) const

Returns the unique number corresponding to the node/field pair. This is the inverse of nodf().

Return Value:
`the` unique number

Parameters:
`node` (input) the node
`kdof` (input) the field

Author: M. Storti

52.33

void **nodf** (int edof, int &node, int &field) const

Transforms nodf to (node,field). This is the inverse of edof().

Parameters:
`edof` (input) the nodf unique value
`node` (output) the node
`kdof` (output) the field

Author: M. Storti

52.34

int **col_is_null** (int j)

Returns true if col j is null.

Parameters: `j` (input) the column index.
Author: M. Storti

52.35

int **set_constraint** (const Constraint &constraint)

Sets a constraint from the list of node/field/coefficients.

Return Value: `0/1` if the linear constraint has been detected to belinearly dependent with the preexisting constraints.
Parameters: `constraint` (input) list of node/field/coefficients.
Author: M. Storti

52.36

void **dof_range** (int myrank, int &dof1, int &dof2)

Returns the range of dofs that belong to a given processor. The dofs in this processor (base 0) are dof1 <= dof <= dof2.

Parameters: `myrank` (input) the identifier of this processor.
`dof1` (input) the start of the dof range.
`dof2` (input) the end of the dof range.
Author: M. Storti

52.37

int **create_MPI_vector** (Vec &v)

Creates an MPI vector.

Parameters: `v` (output) the vector to be created.
Author: M. Storti

52.38

int **create_MPI_ghost_vector** (Vec &v)

Creates an MPI vector with the ghost values.

Parameters: `v` (output) the vector to be created.
Author: M. Storti

52.39

void **solve** (double* y, double* x)

Solves an equation of the form Q x = y for x. Assumes that rank(Q) = n = dim(x).

Author: M. Storti

52.40

void **qxpy** (double* x, double* y, double alpha)

Makes `y=y+alpha*Q*x`.

Parameters:
- `y` (input/output) the vector to add `Q*x` (size `nnod*ndof`).
- `x` (input) the vector to multiply by `Q` (size `neq`).
- `alpha` (input) the scalar to scale the term `Q*x`.

52.41

void **qtxpy** (double* x, double* y, double alpha)

Makes `y=y+alpha*Q*x`.

Parameters:
- `y` (input/output) the vector to add `Q'*x` (size `neq`).
- `x` (input) the vector to multiply by `Q'` (size `nnod*ndof`).
- `alpha` (input) the scalar to scale the term `Q*x`.

52.1

dvector<int> **idmap2_dv**

This maps a node/dof to an eq. or either an entry in the non-regular list.

53

Vec macros to allow Fortran-like access to array elements.

Names

53.1

#define **VEC_ADDR_2** (j,k,dk)

adressing a 1-dimensional array as 2 dimensional.

53.2

#define **VEC_ADDR_3** (j,k,dk,l,dl)

adressing a 1-dimensional array as 3 dimensional.

53.3

#define **VEC_ADDR_4** (j,k,dk,l,dl,p,dp)

adressing a 1-dimensional array as 4 dimensional.

53.4

#define **VEC_ADDR_5** (j,k,dk,l,dl,p,dp,q,dq)

adressing a 1-dimensional array as 5 dimensional.

53.5

#define **VEC2** (name,j,k,dk)

Accessing a 1-dimensional array as 2 dimensional. Typical use: #define MATRIX(j,k) VEC2(j,k,dk)

Parameters:

`name`	(input) name of the array
`j`	(input) row index
`k`	(input) column index (running faster)
`dk`	(input) column dimension

Author: M. Storti

53.6

#define **VEC3** (name,j,k,dk,l,dl)

Accessing a 1-dimensional array as 3 dimensional. Last index runs faster. Typical use: #define MATRIX(j,k,l) VEC2(j,k,dk,l,dl)

Parameters:

`name`	(input) name of the array
`j`	(input) first index
`k`	(input) 2nd index
`dk`	(input) 2nd index dimension
`l`	(input) 3rd index
`dl`	(input) 3rd index dimension

Author: M. Storti

53.7

#define **VEC4** (name,j,k,dk,l,dl,p,dp)

Accessing a 1-dimensional array as 4 dimensional. Last index runs faster. Typical use: #define MATRIX(j,k,l) VEC2(j,k,dk,l,dl)

Parameters:

`name`	(input) name of the array
`j`	(input) first index
`k`	(input) 2nd index
`dk`	(input) 2nd index dimension
`l`	(input) 3rd index
`dl`	(input) 3rd index dimension
`p`	(input) 4th index
`dp`	(input) 4th index dimension

Author: M. Storti

53.8

#define **VEC5** (name,j,k,dk,l,dl,p,dp,q,dq)

Accessing a 1-dimensional array as 4 dimensional. Last index runs faster. Typical use: #define MATRIX(j,k,l,p,q) VEC2(j,k,dk,l,dl,p,dp,q,dq)

Parameters:

`name`	(input) name of the array
`j`	(input) 1st index
`k`	(input) 2nd index
`dk`	(input) 2nd index dimension
`l`	(input) 3rd index
`dl`	(input) 3rd index dimension
`p`	(input) 4th index
`dp`	(input) 4th index dimension
`q`	(input) 5th index
`dq`	(input) 5th index dimension

Author: M. Storti

54

char* **local_copy** (const char* cstr)

Makes a temporary copy of a string.

Return Value: `a` pointer to the copied string
Parameters: `cstr` (input) the string to be copied
Author: M. Storti

55

Text hash functions

Names

55.1

void **delete_hash_entry** (void* p, void* q, void* u)

Remove entry from hash. To be passed to Glib traversal functions to build the hash destructor.

Parameters:
- `p` (input) key string
- `q` (input) value string
- `u` (not used) as required by Glib

Author: M. Storti

56

class **TextHashTableVal**

TextHashTable's are a map string -> TextHashTableVal objects

57

extern TextHashTable* **GLOBAL_OPTIONS**

The GLOBAL hash table function

58

class **TextHashTable**

Public Members

Private Members

Text hash tables (key and value are strings) are used to store elemset properties.

58.1

void **print** (const char* = NULL) const

Prints the entire hash.

Parameters: `s` (input) optional string

Author: M. Storti

58.2

void **set_entries** (const std::map<std::string, std::string>& entries)

Sets many entries to the hash (no warn if already there).

Parameters: `entries` (input) map : key->value
Author: L. Dalcin

58.3

void **set_entry** (const char* key, const char* value)

Sets an entry to the hash (no warn if already there).

Parameters: `key` (input) key of the entry
`value` (input) value of the entry
Author: L. Dalcin

58.4

void **add_entry** (const char* key, const char* value, int warn=1)

Adds an entry to the hash.

Parameters: `key` (input) key of the entry
`value` (input) value of the entry
Author: M. Storti

58.5

void **add_entry** (const char* key, const int* value, int n=1)

Adds an entry to the hash.

Parameters:
`key` (input) key of the entry
`value` (input) value of the entry

Author: M. Storti

58.6

void **add_entry** (const char* key, const double* value, int n=1)

Adds an entry to the hash.

Parameters:
`key` (input) key of the entry
`value` (input) value of the entry

Author: M. Storti

58.7

void **include_table** (const string &s, const TextHashTable* t = NULL)

Adds an included table.

Parameters:
`s` (input) its name
`t` (input) a pointer to the TextHashTable to be included(optional). If not given, look for 's' in the'included_tables_names' table

Author: M. Storti

58.8

void **register_name** (const string &s)

Registers the table by its name.

Parameters:
`s` (input) the name of the table.

Author: M. Storti

58.9

void **set_as_global** ()

Sets this table as the global one.

Author: M. Storti

58.10

void **get_entries** (std::map<std::string, std::string>&) const

Fills a map<string,string> with hash table contents. Read values are appended to the map so you perhaps have to clear() it before calling this method. Access counter for entries are not incremented.

Parameters:
`M` (output) value of the entry

Author: L. Dalcin

58.11

void **get_entry** (const char* , const char* &) const

Searches an entry in the hash.

Parameters:
`key` (input) key of the entry
`value` (output) value of the entry

Author: M. Storti

58.12

void **get_entry** (const char* , vector<double> &v)

Searches an entry in the hash and reads doubles from it

Parameters:
`key` (input) key of the entry
`v` (output) A vector of doubles. Read valuesare appended to the vector so you perhaps have to clear() it before calling this method.

Author: M. Storti

58.13

void **del_entries** ()

Removes all entries in the hash

Author: L. Dalcin

58.14

int **access_count** (const char*)

Returns the number of times a particular key was accessed.

Return Value: `the` number of access to the key
Parameters: `key` (input) key of the entry
Author: M. Storti

58.15

TextHashTable ()

Constructs a void hash table.

Author: M. Storti

58.16

~TextHashTable ()

Destructor.

58.17

void **read** (FileStack* & fstack)

Reads a text hash table from a filestack.

Parameters: `fstack` (input) The filestack from which the hash table isread.
Author: M. Storti

58.18

static void **print_stat** ()

Print all the text-hash-tables

Author: M. Storti

58.19

static const TextHashTable* **find** (const string &name)

Returns a pointer to the table given his name.

Return Value: `a` pointer to the table, `NULL` if it wasn't found.
Parameters: `name` (input) the name of the table

58.20

GHashTable* **hash**

The underlying Glib hash.

58.21

vector<const TextHashTable *> **included_tables**

A list of pointers to other (included) hashes

58.22

vector<const string *> **included_tables_names**

A list of the names of the included hashes

58.23

static THashTable **thash_table**

The global register table

58.24

static TextHashTable* **global_options**

The global hash table

58.25

void
get_entry_recursive (const char* , TextHashTableVal* &, int &glob_was_visited) const

Searches an entry in the hash recursively. Returns the whole entry (a struct) instead of the plain string.

Parameters:

`key`	(input) key of the entry
`value`	(output) value of the entry

Author: M. Storti

58.26

void **get_entry** (const char* , TextHashTableVal* &) const

Searches an entry in the hash. This returns the whole entry (a struct) instead of the plain string.

Parameters:

`key`	(input) key of the entry
`value`	(output) value of the entry

Author: M. Storti

59

Utils

Names

59.1

double **mydet** (Matrix A)

Computes the determinant of a newmat matrix. Currently only for n x n with n<=3 matrices.

Return Value: `deterinant` of A
Parameters: `A` matrix to compute the determinant
Author: M. Storti

59.2

double **mydetsur** (Matrix &A, ColumnVector & S)

Computes the absolute value of the normal vector (differential of surface). When integrating over a surface (or a line in 2D) we need the differential of surface, which plays the role of the differential of volume in 3D which is related with the determinant of the Jacobian from master to spatial coordinates. In this case we have `ndim` spatial dimensions and `ndim-1` coordinates in the master element. The jacobian has dimensions `ndim x (ndim-1)` and computing the minors of this matrix we obtain a vector of dimension ndim. The norm of this vector is the differential of surface. If you need the unit vector normal to the surface, then simply normalize this vector.

Return Value: `norm` of vector formed with the minors of A (norm of normalvector in 3D)
Parameters: `A` matrix to compute the determinant
`S` vector normal to the surface (with absolute value equalto the relative area)
Author: M. Storti

59.3

double **mydetsur** (FastMat2 &A, FastMat2 &S)

Overloaded for Newmat matrices

59.4

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

Clon of Matlab's kron. For given matrices A (nxm) and B (pxq) returns a matrix C (np x mq) formed with blocks [A(1,1)*B A(1,2)*B .... ; A(2,1)*B A(2,2)*B ; .... A(n,m)*B]

Return Value: `C=kron(A,B)`
Parameters: `A` first matrix argument
`B` second matrix argument
Author: M. Storti

59.5

inline double **drand** ()

Double random function based on rand(). Random value uniformly distributed in [0,1].

Return Value: `random` value
Author: M. Storti

59.6

int **irand** (int imin, int imax)

Integer random number uniformly distributed in [imin,imax]

Return Value: `integer` random value
Parameters: `imin` low end of the range
`imin` high end of the range
Author: M. Storti

59.7

int **irand** (int n)

Integer random number uniformly distributed in [0,n-1]

Return Value: `integer` random value
Parameters: `n` number of objects

Author: M. Storti

59.8

int **mini** (int n, ...)

Minimum value of a set of integers.

Return Value:	`the`	minimum of these integers
Parameters:	`n`	number of integer values
	`...`	set of integers
Author:	M. Storti	

59.9

int **maxi** (int n, ...)

Maximum value of a set of integers.

Return Value:	`the`	maximum of these integers
Parameters:	`n`	number of integer values
	`...`	set of integers
Author:	M. Storti	

59.10

double **mind** (int n, ...)

Minimum value of a set of doubles.

Return Value:	`the`	minimum of these doubles
Parameters:	`n`	number of double values
	`...`	set of double
Author:	M. Storti	

59.11

double **maxd** (int n, ...)

Maximum value of a set of doubles.

Return Value: `the` maximum of these integers
Parameters: `n` number of double values
`...` set of doubles
Author: M. Storti

59.12

int **reshape** (Matrix &A, int m, int n)

Reshapes a matrix to be m x n. (Clon of matlab reshape()).

Parameters: `A` (input/output) matrix to be reshaped.
`m` (input) new row dimension
`n` (input) new column dimension
Author: M. Storti

59.13

template<typename T> T **random_pop** (set<T> &Tset)

Generic algorithm to return a random item from a set.

Return Value: `random` element in set
Parameters: `T` class of elements in the set
`Tset` the set
Author: M. Storti

59.14

#define **VOID_IT** (x)

Voids a generic container.

Parameters: `x` generic container (set, list, etc...)
Author: M. Storti

60

inline int **modulo** (int k, int n, int* div=NULL)

Performs the modulo operation, in the sense of number theory. Given integers `k` and `n>0`, we find `m` and `div` such that `k = m * div + n` with `0 <= m < n`

Return Value:	`the`	result of the modulo operation
Parameters:	`k`	(input) the number to take the modulo
	`n`	(input) the modulo
	`div`	(input) pointer to an integer where to put the divisor

61

inline double **modulo** (double a, double b, int &m)

Performs the modulo operation, in the sense of number theory for doubles. Given doubles `a` and `b>0`, we find `r` and `m` such that `a = m * b + r` with `0 <= r < |b|`

Return Value:	`the`	result of the modulo operation
Parameters:	`k`	(input) the number to take the modulo
	`n`	(input) the modulo
	`div`	(input) pointer to an integer where to put the divisor

62

int **string_bcast** (string &s, int master, MPI_Comm comm)

Broadcasts a string from the master to the slaves

Return Value:	`error`	code
Parameters:	`s`	(input/output) the string to be broadcasted
	`master`	(input) the index of the master
	`comm`	(input) the MPI communicator

63

Manage chronometers.

Manage chronometers.

Author: M. Storti

64

High precision cronometer

High precision cronometer

Author: M. Storti

65

int **read_hash_table** (FileStack* & fstack, TextHashTable* & thash)

Reads a hash table from a filestack. Reads a text hash table from a filestack.

Parameters:	`fstack`	(input) the file stack from where the hash will be read
	`thash`	(output) the text hash table that is read
	`fstack`	(input) The filestack from which the hash table isread.
	`(output)`	The hash table that has been read.
Author:	M. Storti	

66

void **read_double_array** (vector<double> &v, const char* s)

Reads a series of doubles from a string into a vector<double>. Useful when reading hash-tables with a number of double arguments.

Return Value: `s` (input) the string where the doubles are read.
Parameters: `v` (output) the array of doubles returned
Author: M. Storti

67

void **read_int_array** (vector<int> &v, const char* s)

Reads a series of ints from a string into a vector<int>. Useful when reading hash-tables with a number of int arguments.

Return Value: `s` (input) the string where the ints are read.
Parameters: `v` (output) the array of ints returned
Author: M. Storti

68

double **int_pow** (double base, int exp)

Equivalent to 'pow' but for integer powers. (May be more efficient).

Return Value: `a` reference to the matrix.
Parameters: `(input)`

69

int **crem** (int j, int m)

Cyclic 'rem'. `crem(n,m)` computes the remainder of division of 'n' by 'm' but (in contrast with the C builtin % operator) is extended cyclically to negative numbers so that for instance, for the numbers `n` -4 to 4 we have: `rem (*,3) =` {`-1,0,-2,-1,0,1,2,0,1`} whereas `crem (*,3) =` {`2,0,1,2,0,1,2,0`} [Note: this is duplicated with 'modulo' defined in 'utils.h']

70

Wrapper to PETSc destroy functions

Names

Check if the pointer is NULL.

70.1

int **VecDestroy_maybe** (Vec &v)

VecDestroy wrapper

70.2

int **MatDestroy_maybe** (Mat &v)

MatDestroy wrapper

70.3

int **KSPDestroy_maybe** (KSP &v)

KSPDestroy wrapper

71

class **State**

Public Members

Protected Members

States are a state vectr plus a time, so that they can be passed to a dofmap and we can have all the nodal values from it. In a future we can have a generic StateFilter (double arrays) class so that Filters can get their values from them.

Author: M. Storti

71.3

State ()

Default constructor

71.4

State (Vec &vec, Time t)

Constructor from state vector and time

71.5

State (const State &v)

Constructor from another state

71.6

virtual **~State** ()

Destructor

71.7

Operations on the time part

Names

71.7.1

State& **set_time** (const Time& t)

Changes the time of the state

71.7.2

State& **inc** (double dt)

Increments the time part

71.7.3

const Time& **t** () const

Const access to the time part

71.8

Operations on the state vector part

Names

71.8.1

State& **axpy** (double gamma, const State &v)

axpy operation on the state vector part: `*this += gamma * v`

71.8.2

State& **scale** (double gamma)

scales state vector

71.8.3

State& **set_cnst** (double a)

sets to a constant

71.8.4

const Vec& **v** () const

Const acces to the vector part

71.8.5

operator Vec & ()

Converts to a Petsc vector

71.9

Utilities

Names

71.9.1

const State&
print_some (const char* filename, Dofmap* dofmap, set<int> & node_list) const

Print some node/field combinations

71.9.2

const State& **print** () const

Print the whole state vector

71.9.3

Prints only the first n items of the local part of the state

Prints only the first n items of the local part of the state

71.1

Vec* **vec**

The state vector

71.2

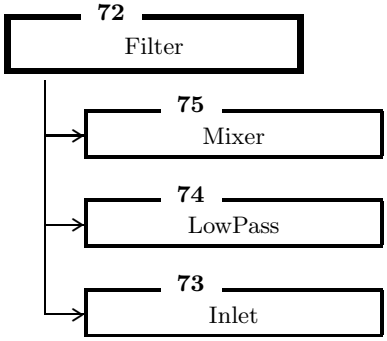
The time corresponding to this state vector.

The time corresponding to this state vector. Values on Dirichlet

72

class **Filter**

Inheritance



Public Members

Private Members

This is the basic, virtual filter class.

Author: M. Storti

72.2

int **step** ()

Returns the time_step

72.3

Filter (int ts=0)

Default constructor

72.4

virtual void **update** (Time time)

Updates the filter from its inputs

72.5

virtual const State& **state** () const

Returns the state of the filter

72.6

virtual **operator const State &** () const

Converts to state

72.1

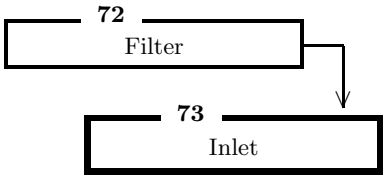
int **time_step**

The actual time step

73

class **Inlet** : public Filter

Inheritance



Public Members

Private Members

This is the basic filter that connects a external state to the filter chain. It has no internal state.

Author: M. Storti

73.2

Inlet (State &st)

Constructor from the external signal

73.3

const State& **state** () const

Instantiation of the state function

73.4

virtual **~Inlet** ()

Destructor

73.1

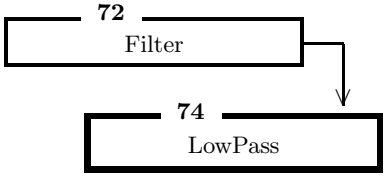
const State* **state_**

The external state from where the signal is received

74

class **LowPass** : public Filter

Inheritance



Public Members

This is a recursive filter of the form $\hat{u}_{j+1} = \gamma\hat{u}_j + (1-\gamma)u_{j+1}$. One usually enters α such that $\gamma = \exp -\alpha\Delta t$.

Author: M. Storti

74.1

LowPass (double gamma_, Filter &input_, State &state_)

Constructor

74.2

virtual ~**LowPass** ()

Destructor

74.3

void **update** (Time time)

Updates the filter (and its input)

74.4

State& **state** ()

Allows access to the internal state

74.5

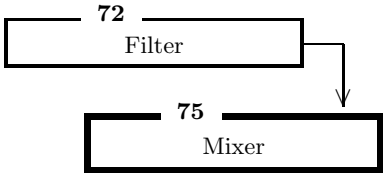
const State& **state** () const

Allows const access to the internal state

75

class **Mixer** : public Filter

Inheritance



Public Members

Private Members

The Mixer class should not have an internal state and is a linear combination of its inputs.

Author: M. Storti

75.4

Mixer (State &st)

Constructor from a typical state

75.5

virtual ~**Mixer** ()

Destructor

75.6

Mixer& **add_input** (Filter &input, double g)

Adds an input to the list of inputs

75.7

void **update** (Time time)

Updates the state and all its inputs.

75.8

State& **state** ()

Gives access to the internal state

75.9

const State& **state** () const

Gives const access to the internal state

75.1

vector<Filter *> **filter_l**

List of inputs

75.2

vector<double> **gain_l**

List of gain factors. The output is `output = sum_j {gain_l[j] * filter_l[j].state}`

75.3

State **i_state**

Internal state.

76

class **LPFilterGroup**

Public Members

Private Members

Family of LowPass filters. Contains typically several chains of low pass filters at different orders.

76.6

LPFilterGroup (TextHashTable* thash, State &x, double Dt)

Constructor from the pointer to the general options

76.7

~LPFilterGroup ()

Destructor

76.8

void **update** (Time time)

Updates of the filters in the chain

76.9

Filter& **filter** (int j, int k=1)

Returns the k-th filter in the j-th chain

76.10

void **print_some** (const char* filename, Dofmap* dofmap, set<int> & node_list)

Prints filtered states

76.1

Inlet **input**

Input to the filter chain

76.2

vector<vector < LowPass *> > **lp_filters**

List of lists of filters

76.3

vector<double> **gamma_v**

List of relaxation factors

76.4

vector<int> **n_v**

Orders (length) of each chain

76.5

int **nalpha**

Number of different chains

77

typedef double
AmplitudeFunction (TextHashTable* thash, const TimeData* time_data, void* & fun_data)

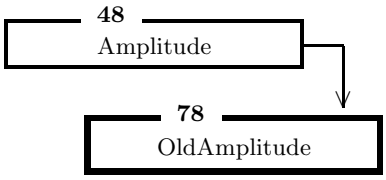
This is the type of temporal functions. A TextHashTable stores various parameters, and the time (or time like data) is passed to the function.

Author: M. Storti

78

class **OldAmplitude** : public Amplitude

Inheritance



Public Members

Private Members

An amplitude is basically a pointer to a function that depends on some fixed parameters passed through a hash table and a variable parameter (typically time). For instance function 'sine' with constant parameters omega and phase, depending on time. Note: All these hould be rewritten using polymorphism.

Author: M. Storti

78.4

static FunctionTable* **function_table**

A static table from the key to a pointer to the corresponding function.

78.5

OldAmplitude (char* & s_, TextHashTable* tht_=NULL)

Constructor

78.6

double **eval** (const TimeData* time_data)

Eval the amplitude of the function at this time.

78.7

static void **add_entry** (const char* s, AmplitudeFunction* f)

Adds an entry to the static table.

78.8

static void **initialize_function_table** (void)

Initializes the function table.

78.9

void **read_hash_table** (FileStack* fstack)

Reads the table from a fstack

78.10

void **print** (void) const

prints the amplitude entry.

78.1

char* **amp_function_key**

The key, ie. the string identifying the function (e.g. 'sine')

78.2

TextHashTable* **thash**

A table with parameters for the function (amplitude, starting time, for instance.)

78.3

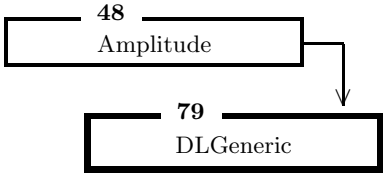
void* **fun_data**

A place where to store things

79

class **DLGeneric** : public Amplitude

Inheritance



Public Members

Private Members

Generic amplitude function that dynamically loads functions

79.17

DLGeneric ()

Constructor (initializes 'fun_data')

79.18

void **print** () const

Prints information about the function

79.19

void **init** (TextHashTable* thash)

Initializes values

79.20

virtual void **clear** ()

Clears the object

79.21

double **eval** (const TimeData* time_data, int node, int field)

Computes the value of the Dirichlet bc. at the specified time

79.22

virtual **~DLGeneric** ()

Destructor

79.1

TextHashTable* **thash**

Options table

79.2

typedef void **InitFun** (TextHashTable* , void* &)

Initialization function prototype

79.3

typedef double **EvalFun** (double, void*)

Evaluation function prototype

79.4

typedef void **ClearFun** (void*)

Cleanup function prototype

79.5

InitFun* **init_fun**

Initialization function

79.6

EvalFun* **eval_fun**

Evaluation function

79.7

ClearFun* **clear_fun**

Cleanup function

79.8

struct **FunHandle**

Contains pointers for a given set of functions

79.9

typedef map<string,FunHandle> **FunTable**

This is the internal table that is kept for each file

79.10

struct **FileHandle**

Members

For each file we keep a handle (from 'dlopen()') table of pointers to functions

79.10.1

void* **handle**

handle obtained from 'dlopen()'

79.10.2

FunTable* **fun_table**

table of functions for each file

79.11

typedef map<string,FileHandle> **FileHandleTable**

We keep a static table filename -> file_handle. This is the generic type for this table.

79.12

void* **handle**

The actual 'dlopen' handle for this instance

79.13

FileHandle **fh**

The actual handle for this instance

79.14

FunHandle **fuh**

The actual function handle for this instance

79.15

void* **fun_data**

This generic pointer may be used to store internal values for the functions

79.16

static FileHandleTable **file_handle_table**

This is the actual table

80

Useful macros for defining extended functions for amplitude

Names

temporal functions.

80.1

#define **INIT_FUN** (TextHashTable *thash,void *&fun_data)

Defines the init_fun to be called before the loop

80.2

#define **INIT_FUN1** (name)

Same as `INIT_FUN` but for the case where a prefix is used.

80.3

#define **EVAL_FUN** (double t,void *fun_data)

Evaluation function to be called for each time

80.4

#define **EVAL_FUN1** (name)

Same as `EVAL_FUN` but with prefix

80.5

#define **CLEAR_FUN** (void *fun_data)

Clean-up function to be called after all calls to the eval function.

80.6

#define **CLEAR_FUN1** (name)

Same as `CLEAR_FUN` but with prefix

80.7

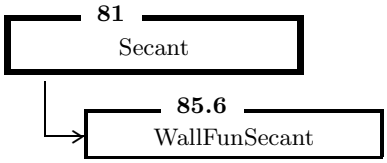
#define **DEFINE_EXTENDED_AMPLITUDE_FUNCTION** (fun_obj_class)

The simple way is to define a class with methods `init(TextHashTable *)` and `eval_fun(double time)`. This is wrapper to this class. Clean up is done in the destructor of the class.

81

class **Secant**

Inheritance



Public Members

Solves a 1D non-linear eq. with the secant method $f(x) = 0$. Very simple.

81.1

double **x0**

Initial value

81.2

double **epsilon**

Initial increment

81.3

double **tol**

Tolerance for the solution of the equation

81.4

double **f**

function value at the solution

81.5

int **maxits**

maximum number of iterations

81.6

int **its**

iterations actually perfomed

81.7

double **omega**

relaxation factor

81.8

Default constructor

Default constructor

81.9

virtual double **residual** (double x, void* user_data=NULL)

This should be defined by the user

81.10

double **sol** ()

Computes the solution

82

class **Debug**

Private Members

82.1 map<string, int> **active_flags**
Flags whether the system should stop or not at each call to 'wait' or not 252
Puts barriers so that all processes are synchronized

82.1

map<string,int> **active_flags**

Flags whether the system should stop or not at each call to 'wait' or not

83

The distributed container and graph class

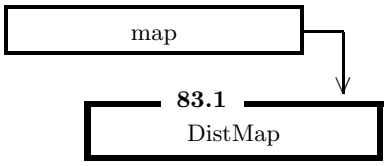
Names

The distributed container and graph class

83.1

template<class Key,class Val,class Partitioner> class
DistMap : public map<Key, Val>

Inheritance



Public Members

Protected Members

Distributed map class. Elements can be assigned as for a standard 'map' and, after, a 'scatter' operation allows items in the map to be passed from one processor to other. The `Partitioner` object determines to which processor belongs each dof.

83.1.4

DistMap<Key,Val,Partitioner>
Constructor from a communicator (Partitioner* pp=NULL, MPI_Comm comm_=PETSCFEM_COMM_WORLD)

Constructor from a communicator

Return Value: `a` reference to the matrix.

Parameters: `comm_` (input) MPI communicator

83.1.5

int **processor** (kv_iterator k) const

User defines this function that determine to which processor belongs each entry

Return Value: `the` number of processor where these matrix should go.
Parameters: `k` (input) iterator to the considered entry.

83.1.6

int **size_of_pack** (kv_iterator k) const

Computes the size of data needed to pack this entry

Return Value: `the` size in bytes of the packed object
Parameters: `k` (input) iterator to the entry

83.1.7

void **pack** (const Key &k, const Val &v, char* &buff) const

Packs the entry `(k,v)` in buffer `buff`. This function should be defined by the user.

Parameters:
`k` (input) key of the entry
`v` (input) value of the entry
`buff` (input/output) the position in the buffer where thepacking is performed

83.1.8

void **unpack** (Key &k, Val &v, const char* & buff)

Does the reverse of `pack`. Given a buffer `buff` recovers the corresponding key and val. This function should be defined by the user.

Parameters:

`k`	(output) key of the entry
`v`	(output) value of the entry
`buff`	(input/output) the position in the buffer from where theunpacking is performed

83.1.9

void **scatter** ()

perform the scatter of elements to its corresponding processor.

83.1.10

void **combine** (const pair<Key, Val> &p)

This function should be defined by the user. Merges a pair key, value in the container.

Parameters:

`p`	(input) the pair to be inserted.

83.1.1

MPI_Comm **comm**

MPI communicator

83.1.2

Partitioner* **part**

This returns the number of processor for a given dof

83.1.3

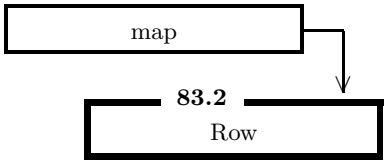
int **size**

size and rank in the comunicator

83.2

class **Row** : public map<int, double>

Inheritance



Public Members

Private Members

This the row of the matrix.

83.2.3

void **print** () const

print the row

83.2.1

int **size_of_pack** () const

define their own members for packing and unpacking

83.2.2

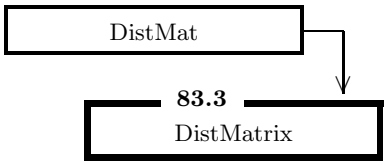
void **pack** (char* &buff) const

define their own members for packing and unpacking

83.3

class **DistMatrix** : public DistMat

Inheritance



Public Members

A distributed map<int,int,double> class

83.3.1

void **insert_val** (int i, int j, double v)

Specific "insert" routine.

83.3.2

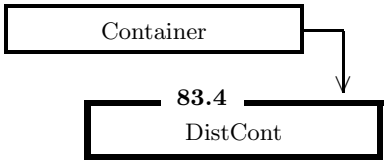
double **val** (int i, int j)

Specific function for retrieving values.

83.4

template<typename Container,typename ValueType,typename Partitioner> class
DistCont : public Container

Inheritance



Public Members

Protected Members

Distributed map class. Elements can be assigned as for a standard 'map' and, after, a 'scatter' operation allows items in the map to be passed from one processor to other. The `Partitioner` object determines to which processor belongs each dof.

83.4.1

enum **iter_mode_t**

Iteration is different for non-associative containers ('erase()' doesn't remove the object, ie. random-access containers like vectors-deques.) than for associative containers ('erase()' does remove the object, for instance lists and maps). (fixme:= this should be done better).

83.4.6

DistCont<Container, ValueType,Partitioner>
Constructor. (Partitioner* part=NULL, MPI_Comm comm_=PETSCFEM_COMM_WORLD, iter_mode_t iter_mode = associative_iter_mode)

Constructor.

Parameters:

`part`	(input) partitioner, defines to which processor belongs each iterator
`comm_`	(input) MPI communicator
`iter_mode`	(input) tells what kindof container the class 'Container' is.

83.4.7

int **size_of_pack** (const ValueType &p) const

Computes the size of data needed to pack this entry

Return Value: `the` size in bytes of the packed object

Parameters: `k` (input) iterator to the entry

83.4.8

void **pack** (const ValueType &p, char* &buff) const

Packs the entry `(k,v)` in buffer `buff`. This function should be defined by the user.

Parameters:

`k`	(input) key of the entry
`v`	(input) value of the entry
`buff`	(input/output) the position in the buffer where thepacking is performed

83.4.9

void **unpack** (ValueType &p, const char* & buff)

Does the reverse of `pack`. Given a buffer `buff` recovers the corresponding key and val. This function should be defined by the user.

Parameters:

`k`	(output) key of the entry
`v`	(output) value of the entry
`buff`	(input/output) the position in the buffer from where theunpacking is performed

83.4.10

void **scatter** ()

perform the scatter of elements to its corresponding processor.

83.4.11

void **combine** (const ValueType &p)

This function should be defined by the user. Merges a pair key, value in the container.

Parameters:

`p`	(input) the pair to be inserted.

83.4.2

MPI_Comm **comm**

MPI communicator

83.4.3

Partitioner* **part**

This returns the number of processor for a given dof

83.4.4

int **nprocs**

nprocs and rank in the comunicator

83.4.5

iter_mode_t **iter_mode**

Type of iteration mode

83.5

typedef map<int, GSet, less<int> > **GMap**

The storage area type

83.6

typedef pair<int, GSet > **GRow**

An individual set of the storage map.

83.7

typedef Partitioner< GSet > **GPartitioner**

Partitioner for the scatter operation.

83.8

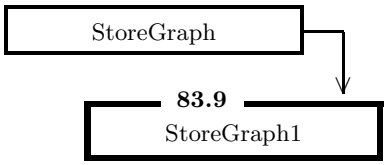
typedef DistCont<GMap,GRow,GPartitioner> **DGMap**

The basic container for the distributed graph class

83.9

class **StoreGraph1** : public StoreGraph

Inheritance



Public Members

Private Members

This 'Graph' class has internal storage, which you can fill with 'set'. Furthermore, it is distributed.

83.9.3

void **add** (int i, int j)

Adds an edge to the graph

83.9.4

void **set_ngbrs** (int j, GSet &ngbrs_v)

callback function, returns the set of neighbors to `j` vertex.

83.9.5

~StoreGraph1 ()

Clean all memory related

83.9.6

StoreGraph1 (int N=0, const DofPartitioner* dp=NULL, MPI_Comm comm_=PETSCFEM_COMM_WORLD)

Constructor

83.9.7

void **scatter** ()

perform the scatter of elements to its corresponding processor.

83.9.8

void **clear** ()

Clean all memory related

83.9.1

MPI_Comm **comm**

The MPI compunicator

83.9.2

DGMap **lgraph**

This is the storage area. The Graph is stored as a correspondence between an integer (one vertex) and a set of integers (those vertices that are connected to him).

83.10

typedef set<int> **GSet**

A set of neighbors.

84

The PFMat abstract Matrix class

Names

The PFMat abstract Matrix class

84.1

class **PFMat**

Public Members

Protected Members

Private Members

This is the generic matrix class

84.1.7

static PFMat* **dispatch** (int N, DofPartitioner &part, const char* s)

This is the 'factory' of 'PFMat' matrices

84.1.8

virtual int **size** (int j)

Returns the size of the j-th dimension

84.1.9

void **size** (int &m, int&n)

Return both sizes

84.1.10

virtual **~PFMat** ()

Virtual destructor

84.1.11

int **set_profile** (int row, int col)

Adds an element to the matrix profile

84.1.12

int **create** ()

Creates the matrix from the profile graph entered with 'profile'

84.1.13

int
set_value (int row, int col, PetscScalar value, InsertMode mode=ADD_VALUES)

Sets individual values on the operator `A(row,col) = value`

Parameters:

`row`	(input) first index
`col`	(input) second index
`value`	(input) the value to be set
`mode`	(input) either `ADD_VALUES` (default) or `INSERT_VALUES`

84.1.14

int
set_values (int nrows, int* idxr, int ncols, int* idxc, PetscScalar* values, InsertMode mode=ADD_VALUES)

Sets an array of values on the operator . For `mode=INSERT_VALUES` it is equivalent to `for (j=0; j<nrows; j++) for (k=0; k<ncols; k++) A(idxr[j],idxc[k]) = values[j*ncols+k]`. For `mode=ADD_VALUES` the assignment operator is replaced by `+=`.

Parameters:

`nrows`	(input) number of row indices
`idxr`	(input) array of row indices of size `nrows`
`ncols`	(input) number of column indices
`idxc`	(input) array of column indices of size `ncols`
`values`	(input) array of values of size `nrows*ncols` stored by row (C style).
`mode`	(input) either `ADD_VALUES` (default) or `INSERT_VALUES`

84.1.15

int **assembly_begin** (MatAssemblyType type)

calls MatAssemblyBegin on internal matrices, see PETSc doc

84.1.16

int **assembly_end** (MatAssemblyType type)

calls MatAssemblyEnd on internal matrices, see PETSc doc

84.1.17

int **assembly** (MatAssemblyType type)

This calls both

84.1.18

int **solve** (Vec &res, Vec &dx)

Solve the linear system

Parameters:

`res`	(input) the rhs vector
`dx`	(input) the solution vector

84.1.19

int **factor_and_solve** (Vec &res, Vec &dx)

Factorizes matrix and solves linear system. Args are passed via pointers.

84.1.20

int **solve_only** (Vec &res, Vec &dx)

Solves (matrix should be factorized). Args are passed via pointers.

84.1.21

int **clean_factor** ()

Cleans the factored part

84.1.22

int **clean_mat** ()

Sets all values of the operator to zero

84.1.23

int **clean_prof** ()

Cleans the profile related part

84.1.24

int **clear** ()

clear memory (almost destructor)

84.1.25

virtual int **monitor** (int n, double rnorm)

Defines how to report convergence in the internal loop. Derive this function to obtain a different effect from the default one.

Return Value: `A` PETSc error code
Parameters: `n` (input) current iteration number
`rnorm` (input) norm of the residual for the currentiteration

84.1.26

virtual int **its** ()

returns the number of iterations spent in the last solve

84.1.27

virtual int **view** (PetscViewer viewer=PETSC_VIEWER_STDOUT_WORLD)

Prints the matrix to a PETSc viewer

84.1.28

int **duplicate** (MatDuplicateOption op, const PFMat &A)

Duplicate matrices (currently not implemented for IISDMat)

84.1.2

int **print_fsm_transition_info**

Print Finite State Machine transitions

84.1.3

int **ierr**

Allows to pass PETSc error codes through the FSM layer

84.1.4

Actions of the finite state machine.

Names

84.1.4.1 virtual int
set_profile_a (int j, int k)
The action corresponding to 'set_profile' 274

84.1.4.2 virtual int
create_a ()
The action corresponding to 'create' 274

84.1.4.3 virtual int
set_value_a (int row, int col, PetscScalar value,
InsertMode mode=ADD_VALUES)
The action corresponding to 'set_value' 274

84.1.4.4 virtual int
set_values_a (int nrows, int* idxr, int ncols, int* idxc, PetscScalar* values,
InsertMode mode=ADD_VALUES)
The action corresponding to 'set_values' 274

84.1.4.5 virtual int
assembly_begin_a (MatAssemblyType type)
calls MatAssemblyBegin on internal matrices, see PETSc doc 275

84.1.4.6 virtual int
assembly_end_a (MatAssemblyType type)
calls MatAssemblyEnd on internal matrices, see PETSc doc 275

84.1.4.7 virtual int
factor_and_solve_a (Vec &res, Vec &dx)
Factorizes matrix and solves linear system. 275

84.1.4.8 virtual int
solve_only_a (Vec &res, Vec &dx)
Solves (matrix should be factorized). 275

84.1.4.9 virtual int

84.1.4.1

virtual int **set_profile_a** (int j, int k)

The action corresponding to 'set_profile'

84.1.4.2

virtual int **create_a** ()

The action corresponding to 'create'

84.1.4.3

virtual int
set_value_a (int row, int col, PetscScalar value, InsertMode mode=ADD_VALUES)

The action corresponding to 'set_value'

84.1.4.4

virtual int
set_values_a (int nrows, int* idxr, int ncols, int* idxc, PetscScalar* values, InsertMode mode=ADD_VALUES)

The action corresponding to 'set_values'

84.1.4.5

virtual int **assembly_begin_a** (MatAssemblyType type)

calls MatAssemblyBegin on internal matrices, see PETSc doc

84.1.4.6

virtual int **assembly_end_a** (MatAssemblyType type)

calls MatAssemblyEnd on internal matrices, see PETSc doc

84.1.4.7

virtual int **factor_and_solve_a** (Vec &res, Vec &dx)

Factorizes matrix and solves linear system. Args are passed via pointers.

84.1.4.8

virtual int **solve_only_a** (Vec &res, Vec &dx)

Solves (matrix should be factorized). Args are passed via pointers.

84.1.4.9

virtual int **clean_factor_a** ()

Cleans the factored part

84.1.4.10

virtual int **clean_mat_a** ()

Sets all values of the operator to zero

84.1.4.11

virtual int **clean_prof_a** ()

clear profile memory

84.1.4.12

virtual int **duplicate_a** (MatDuplicateOption op, const PFMat &A)

duplicate matrix

84.1.5

int **factor_and_solve_A** ()

Factorizes matrix and solves linear system. Args are passed via pointers.

84.1.6

int **solve_only_A** ()

Solves linear system. Args are passed via pointers.

84.1.1

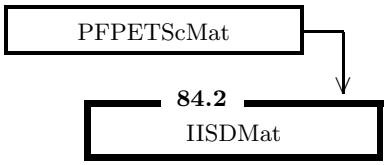
Vec* **res_p**

Pointers to pass args to solver routines through the FSM layer

84.2

class **IISDMat** : public PFPETScMat

Inheritance



Public Members

84.2.55 enum **LocalSolver**
Local solver type 281

84.2.56 int
create_a ()
Creates the matrix from the profile computed in `da` 281

84.2.57 int
mult (Vec x, Vec y)
Applies the Schur operator `y = S * x` 281

84.2.58 int
mult_trans (Vec x, Vec y)
Applies the traspose of the Schur operator (see `mult`*).* 282

84.2.59 int
set_value_a (int row, int col, PetscScalar value,
InsertMode mode=ADD_VALUES)
Sets individual values on the operator `A(row, col) = value` 282

84.2.60 int
set_values_a (int nrows, int* idxr, int ncols, int* idxc, PetscScalar* values,
InsertMode mode=ADD_VALUES)
The action corresponding to 'set_values' 282

84.2.61 void
clear ()
Clear the object (almost destructor) 282

84.2.62 int
clean_mat_a ()
Sets the underlying matrices to zero 282

84.2.63 int
assembly_begin_a (MatAssemblyType type)
Calls MatAssemblyBegin on internal matrices, see PETSc doc 283

84.2.64 int
assembly_end_a (MatAssemblyType type)
calls MatAssemblyEnd on internal matrices, see PETSc doc 283

84.2.65 int
view (PetscViewer viewer=PETSC_VIEWER_STDOUT_WORLD)
Prints the matrix to a PETSc viewer 283

84.2.66 int
set_preco (const string & preco_type)
Derive this if you want to manage directly the preconditioning. 283

84.2.67 **IISDMat** (int MM, int NN, const DofPartitioner &pp,
MPI_Comm comm_ = PETSCFEM_COMM_WORLD)

Private Members

Solves iteratively on the 'interface' (between subdomain) nodes and solving by a direct method in the internal nodes

84.2.55

enum **LocalSolver**

Local solver type

84.2.56

int **create_a** ()

Creates the matrix from the profile computed in `da`

Parameters:

`da`	(input) dynamic array containing the adjacency matrixof the operator
`dofmap`	(input) the dofmap of the operator (containsinformation about range of dofs per processor.
`debug_compute_prof`	(input) flag for debugging the processof building the operator.

84.2.57

int **mult** (Vec x, Vec y)

Applies the Schur operator `y = S * x`

Parameters:

`x`	(input) a given interface vector
`y`	(output) the result of applying the Schur operator on`x`. Usually involves a solution (by a direct method) on eachsubdomain.

84.2.58

int **mult_trans** (Vec x, Vec y)

Applies the traspose of the Schur operator (see `mult`).

Parameters:

`x`	(input) a given interface vector
`y`	(output) the result of applying the Schur operator on`x`.

84.2.59

int
set_value_a (int row, int col, PetscScalar value, InsertMode mode=ADD_VALUES)

Sets individual values on the operator `A(row,col) = value`

Parameters:

`row`	(input) first index
`col`	(input) second index
`value`	(input) the value to be set
`mode`	(input) either `ADD_VALUES` (default) or `INSERT_VALUES`

84.2.60

int
set_values_a (int nrows, int* idxr, int ncols, int* idxc, PetscScalar* values, InsertMode mode=ADD_VALUES)

The action corresponding to 'set_values'

84.2.61

void **clear** ()

Clear the object (almost destructor)

84.2.62

int **clean_mat_a** ()

Sets the underlying matrices to zero

84.2.63

int **assembly_begin_a** (MatAssemblyType type)

Calls MatAssemblyBegin on internal matrices, see PETSc doc

84.2.64

int **assembly_end_a** (MatAssemblyType type)

calls MatAssemblyEnd on internal matrices, see PETSc doc

84.2.65

int **view** (PetscViewer viewer=PETSC_VIEWER_STDOUT_WORLD)

Prints the matrix to a PETSc viewer

84.2.66

int **set_preco** (const string & preco_type)

Derive this if you want to manage directly the preconditioning.

84.2.67

IISDMat (int MM, int NN, const DofPartitioner &pp, MPI_Comm comm_ = PETSCFEM_COMM_WORLD)

Ctor

84.2.68

int **ksp_ll_view** (PetscViewer viewer)

The PETSc wrapper function calls this

84.2.69

~IISDMat ()

Destructor

84.2.1

int **M**

Matrix dimensions

84.2.2

static const int **D**

Type of dofs: L: local, I: interface Type of block (PETSc sense) D: diagonal, O: off-diagonal.

84.2.3

int **n_int**

Number of interface nodes

84.2.4

int **n_loc**

Number of local nodes

84.2.5

int **n_int_tot**

Number of interface dof's in all processors

84.2.6

int **n_loc_tot**

Number of local dof's in all processors

84.2.7

int **n_locp**

Local dof's in this processor start at this position in the MPI local vector.

84.2.8

int **n_intp**

Interface nodes in this processor start at this position in the MPI local vector.

84.2.9

int **neqp**

Number of dof's in this processor.

84.2.10

int **iisd_subpart**

The number of subdomains in which local nodes to each processor are subdivided

84.2.11

int **use_interface_full_preco**

Flags whether we precodition with `A_ii` (`use_interface_full_preco=1`) or with `diag(A_ii)` (`=0`).

84.2.12

int **nlay**

Number of iters in solving the preconditioning for the interface problem when using `use_interface_full_preco`

84.2.13

int **interface_full_preco_maxits**

Number of iters in solving the preconditioning for the interface problem when using `use_interface_full_preco`

84.2.14

string **interface_full_preco_pc**

Defines the preconditioning to be used for the solution of the diagonal interface problem (not the Schur problem)

84.2.15

double **interface_full_preco_fill**

The ILU fill to be used for the A_II problem if the ILU preconditioning is used

84.2.16

int **print_interface_full_preco_conv**

Flags whether or not print the convergence when solving the preconditioning for the interface problem when using `use_interface_full_preco`.

84.2.17

double **interface_full_preco_relax_factor**

The problem on the interface is solved with Richardson method with few iterations (normally 5). Richardon iteration may not converge in some cases and then we can help convergence using a relaxation factor <1.

84.2.18

vector<int> **map_dof**

Maps old numbering in new numbering

84.2.19

vector<int> **n_loc_v**

Local dof's in this processor are in the range `n_loc_v[myrank] <= dof <` n_loc_v[myrank+1]

84.2.20

vector<int> **n_int_v**

Interface dof's in this processor are in the range `n_loc_v[myrank] <= dof <` n_loc_v[myrank+1]

84.2.21

vector<int> **d_nnz_LL**

The PETSc nnz vector for the local part

84.2.22

Sparse::SuperLUMat **A_LL_SLU**

Version of the local matrix

84.2.23

Mat **A_LL**

Local-Local matrix (sequential matrix on each processor).

84.2.24

Mat **A_LI**

Local-Interface matrix (MPI matrix)

84.2.25

Mat **A_IL**

Interface-Local matrix (MPI matrix)

84.2.26

Mat **A_II**

Interface-Interface matrix (MPI matrix)

84.2.27

Mat* **AA** [2][2]

Shortcuts to the `A_LL`, `A_IL`, `A_LI` and `A_II` matrices.

84.2.28

DistMatrix* **A_LL_other**

Here we put all non-local things that are in the loca-local block on other processors

84.2.29

InsertMode **insert_mode**

The mode we are inserting values

84.2.30

Vec **x_loc**

Auxiliar MPI vector that contains all local dof's

84.2.31

Vec **x_loc_seq**

Auxiliar sequential vector that contains local dof's in this processor

84.2.32

Vec **y_loc_seq**

Auxiliar sequential vector that contains local dof's in this processor

84.2.33

PC **pc_ii**

PC for the for the interface preconditioning problem

84.2.34

KSP **ksp_ii**

KSP for the interface preconditioning problem

84.2.35

PC **pc_ll**

PC for local solution (en each processor)

84.2.36

KSP **ksp_ll**

KSP for local solution (en each processor)

84.2.37

static int **warn_iisdmat**

Warn if not appropriate setting for preconditioning type

84.2.38

void **map_dof_fun** (int gdof, int &block, int &ldof)

For a global dof `gdof` gives the `block` ('local' or 'interface') and the number in that block.

Parameters:

`gdof`	(input) dof in old numbering
`block`	(output) the block to which the dof belongs
`ldof`	(input) number of dof in new ordering relative toits block

84.2.39

int **local_solve** (Vec x_loc, Vec y_loc, int trans=0, double s=1.)

Solves the local problem `A_LL x_loc = s * y_loc` for `x_loc`. `x_loc` and `y_loc` may be aliased.

Parameters:

`x_loc`	(output) the solution vector
`y_loc`	(input) the right hand side
`trans`	(input) if non null solves with the traspose of `A_LL_`
`s`	(input) scale factor (usually for changing sign)

84.2.40

int **local_solve_SLU** (Vec x_loc, Vec y_loc, int trans=0, double c=1.)

Idem to `local_solve` but for SuperLU (Sparse::Mat) representation of the local problem.

84.2.41

Vec **A_II_diag**

Diagonal of Interface matrix to use as preconditioning

84.2.42

double **pc_lu_fill**

PETSc LU fill parameter

84.2.43

int **print_Schur_matrix**

Prints the Schur matrix

84.2.44

vector< set<int> > **int_layers**

Layers of nodes of the preconditioning

84.2.45

int **maybe_factor_and_solve** (Vec &res, Vec &dx, int factored)

Factors (maybe) the linear system and solves.

Parameters:

`res`	(input) the rhs vector
`dx`	(input) the solution vector
`factored`	(input) The matrix has been already factored

84.2.46

int **factor_and_solve_a** (Vec &res, Vec &dx)

Factors (maybe) the linear system and solves.

Parameters:

`res`	(input) the rhs vector
`dx`	(input) the solution vector

84.2.47

int **solve_only_a** (Vec &res, Vec &dx)

Solve the linear system

Parameters:

`res`	(input) the rhs vector
`dx`	(input) the solution vector

84.2.48

int **clean_prof_a** ()

Clean all data related to factorization

84.2.49

int **clean_factor_a** ()

Clean all data related to factorization

84.2.50

vector<int> **dofs_proc**

Maps dofs in this processors to global dofs

84.2.51

int* **dofs_proc_v**

Poitner to the storage area in 'dofs_proc'

84.2.52

map<int,int> **proc2glob**

Maps dof's in this processor to global ones. The inverse of 'dofs_proc'.

84.2.53

ISP preconditioning

Names

84.2.53.1

Mat **A_II_isp**

Band-Band matrix (MPI matrix for ISP preco)

84.2.53.2

int **n_isp_tot**

Dofs in band region

84.2.53.3

vector<int> **isp_map**

`isp_map[j]` is the position in the PETSc `A_isp` matrix

84.2.53.4

int **isp_set_value** (int row, int col, PetscScalar value, InsertMode mode)

Set value in the `A_II_isp` matrix (for ISP preconditioning)

84.2.53.5

vector<int> **n_lay1_p**

Dof's in interface/processor `p`: are in range `n_lay1_p[p] <= keq < n_band_p[p]`

84.2.53.6

vector<int> **n_band_p**

Dof's in band/processor `p`: are in range `n_band_p[p] <= keq < n_lay1_p[p+1]`

84.2.53.7

Vec **wb**

Auxiliay vectors for solving the interface ISP preconditioning problem

84.2.53.8

Vec **xb**

Auxiliay vectors for solving the interface ISP preconditioning problem

84.2.54

For fast loading of PETSc matrices

Names

84.2.54.1

dvector<int> **indxr_L**

Indices of submatrix in LL, LI, . blocks

84.2.54.2

dvector<int> **jndxr_L**

Indices of submatrices in submatrix to be loaded in LL, LI, . blocks

84.2.54.3

dvector<double> **v_LL**

Values of submatrix in LL, LI, . blocks

84.2.54.4

dvector<double> * **v** [2][2]

Pointers to v_LL, v_LI,

84.2.54.5

dvector<int> * **indxr** [2]

Pointers to indxr_L, indxr_I

84.3

int **PFPETScMat_default_monitor** (KSP ksp, int n, double rnorm, void*)

Wrapper monitor. You customize the monitor by deriving the `monitor` function member.

84.4

class **SparseDirect** : public ThashPFMat

Inheritance

84.4 SparseDirect

Public Members

84.4.1 int
size (int j)
Returns size 298

84.4.2 int
set_profile_a (int j, int k)
Not used 298

84.4.3 **~SparseDirect** ()
destructor 298

84.4.4 int
assembly_begin_a (MatAssemblyType type)
does nothing here (only sequential use) 298

84.4.5 int
assembly_end_a (MatAssemblyType type)
does nothing here (only sequential use) 298

84.4.6 int
create_a ()
Resizes the underlying `A` *matrix.* 298

84.4.7 int
clean_mat_a ()
Sets all values of the operator to zero 299

84.4.8 **Solve the linear system** 299

84.4.9 int
solve_only_a (Vec &res, Vec &dx)
Solve the linear system 299

84.4.10 int
its ()
returns the number of iterations spent in the last solve 299

84.4.11 int
view (PetscViewer viewer=NULL)
Prints the matrix to a PETSc viewer 299

84.4.12 int
set_preco (const string & preco_type)
Derive this if you want to manage directly the preconditioning. 300

84.4.13 int
duplicate_a (MatDuplicateOption op, const PFMat &B)
Duplicate matrices (currently not implemented for IISDMat) 300

Direct solver. (May be PETSc or SuperLU)

84.4.1

int **size** (int j)

Returns size

84.4.2

int **set_profile_a** (int j, int k)

Not used

84.4.3

~SparseDirect ()

destructor

84.4.4

int **assembly_begin_a** (MatAssemblyType type)

does nothing here (only sequential use)

84.4.5

int **assembly_end_a** (MatAssemblyType type)

does nothing here (only sequential use)

84.4.6

int **create_a** ()

Resizes the underlying `A` matrix. ????

84.4.7

int **clean_mat_a** ()

Sets all values of the operator to zero

84.4.8

Solve the linear system

Solve the linear system

Parameters:

`res`	(input) the rhs vector
`dx`	(input) the solution vector

84.4.9

int **solve_only_a** (Vec &res, Vec &dx)

Solve the linear system

Parameters:

`res`	(input) the rhs vector
`dx`	(input) the solution vector

84.4.10

int **its** ()

returns the number of iterations spent in the last solve

84.4.11

int **view** (PetscViewer viewer=NULL)

Prints the matrix to a PETSc viewer

84.4.12

int **set_preco** (const string & preco_type)

Derive this if you want to manage directly the preconditioning.

84.4.13

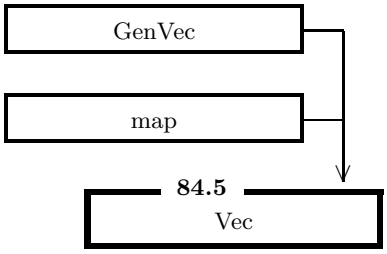
int **duplicate_a** (MatDuplicateOption op, const PFMat &B)

Duplicate matrices (currently not implemented for IISDMat)

84.5

class **Vec** : public map<int, double>, public GenVec

Inheritance



Public Members

Private Members

A simple sparse vector class (0 indexed).

84.5.4

Vec (int l=0)

Constructor from the length

84.5.5

int **length** () const

Return length of the vector

84.5.6

Vec (const Vec &v)

Constructor from another vector

84.5.7

double **get** (int j) const

Get element at specified position

84.5.8

void **get** (const Indx &I, Vec &v) const

Get a subvector of elements at position I

84.5.9

Vec& **copy** (const Vec &v)

Copy

84.5.10

Vec& **set** (int j, double v)

Set element at position j

84.5.11

Vec& **set** (const Indx & I, const Vec & v)

Set elements at subvector at position I. w[I] = v

84.5.12

Vec& **set** (const Vec & v, const Indx & I)

Set vector to elements of v at position I. w = v[I]

84.5.13

Vec& **set** (const Indx & I, const Vec & v, const Indx & K)

Set vector elements at I to elements f of v at position J. w[I] = v[J]

84.5.14

Vec& **set** (const Mat & a, int j)

Set to row of a matrix

84.5.15

Vec& **setc** (const Mat &a, int k)

Set vector to k-th column of matrix a. w = a(:,k)

84.5.16

Vec& **scale** (double c)

Scale elements

84.5.17

Vec& **axpy** (double a, const Vec & v)

Sets w += a * v

84.5.18

Vec& **axpy** (double c, const Indx & I, double a, const Vec & v)

Sets w[I] = c * w[I] + a * v

84.5.19

double **dot** (const Vec & w) const

Dot product

84.5.20

double **dot** (const GenVec & w) const

Dot product with generic vector

84.5.21

void
print_g (int l, const char* s, const char* psep, const char* isep, const char* lsep) const

print elements (generic version)

84.5.22

void **print** (const char* s = NULL)

print elements

84.5.23

Vec& **grow** (int g)

Set mode if can grow automatically or not

84.5.24

Vec& **clear** ()

Clears all elements

84.5.25

Vec& **resize** (int n)

Resize vectors, truncates elements if greater than this value

84.5.26

int **empty** () const

Flags if the vector is empty or not

84.5.27

Vec& **purge** (double tol = 1e-10)

Purge elements below a given tolerance value

84.5.28

Vec& **random_fill** (double fill=0.1, Generator & g = uniform)

Fill with random values

84.5.29

Vec& **apply** (const ScalarFunObj & fun)

Apply a function object to all elements

84.5.30

Vec& **apply** (ScalarFunD* fun, void* user_data = NULL)

Apply a scalar function with args to all elements

84.5.31

Vec& **apply** (ScalarFun* fun)

Apply a scalar function with no args to all elements

84.5.32

double **assoc** (BinAssoc & op) const

perform an asscociative function on all non-null elements

84.5.33

double **sum** () const

Sum of all elements

84.5.34

double **sum_abs** () const

Sum of absolute value of all elements

84.5.35

double **max** () const

Max of all elements

84.5.36

double **max_abs** () const

Max of absolute value of all elements

84.5.37

double **min** () const

Min of all elements

84.5.38

void **accum** (double &v, Accumulator & acc) const

perform an asscociative function on all non-null elements

84.5.39

double **sum_sq** () const

Sum of squares of all elementsSum of squares of all elements

84.5.40

double **sum_pow** (double n) const

Sum of power of all absolute value of all elementsSum of power of all absolute value of all elements

84.5.1

int **len**

Length of the sparse vector

84.5.2

int **grow_m**

Flag indicating where you can add values past the specified length or not

84.5.3

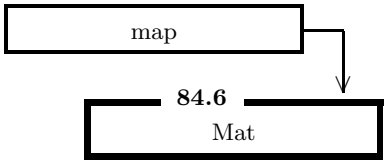
double **get_nc** (int j) const

Get element at specified position (do not check bounds)

84.6

class **Mat** : public map< int, Vec >

Inheritance



Public Members

Protected Members

Private Members

Simple sparse matrix class.

84.6.8

Mat (int m=0, int n=0, TextHashTable* t=NULL)

Constructor from the length

84.6.9

virtual ~**Mat** ()

Destructor (invokes clear() FSM)

84.6.10

int **rows** () const

Return row dimension

84.6.11

int **cols** () const

Return column dimension

84.6.12

Mat (const Mat &a)

Constructor from another vector

84.6.13

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

Get element at specified position: v = w(j,k)

84.6.14

void **getr** (int j, Vec & v) const

Get row: v = w(j,:)

84.6.15

Mat& **set** (int j, int k, double v)

Set element at position j: w(j,k) = v

84.6.16

Mat& **setr** (int j, Vec &v)

Set row j: w(j,:) = v;

84.6.17

Mat& **setr** (const Mat & a, Indx &J)

set to J rows of a: w = a(J,:)

84.6.18

Mat& **setc** (int j, const Vec &v)

Set col j: w(:,j) = v;

84.6.19

Mat& **setc** (const Mat & a, Indx &J)

set to J cols of a: w = a(:,J)

84.6.20

Mat& **id** (double a=1.)

Set to multiple of Identity matrix

84.6.21

Mat& **diag** (const Vec & v)

Diagonal from a vector

84.6.22

Mat& **random_fill** (double fill=0.1, Generator & g=uniform)

Fill with random values

84.6.23

Mat& **prod** (const Mat & a, const Mat & b, double c=1.)

Product of sparse matrices

84.6.24

Mat& **apply** (const ScalarFunObj & fun)

Apply a function object to all elements

84.6.25

Mat& **apply** (ScalarFunD* fun, void* user_data = NULL)

Apply a scalar function with args to all elements

84.6.26

Mat& **apply** (ScalarFun* fun)

Apply a scalar function with no args to all elements

84.6.27

Mat& **axpy** (double c, const Mat & a)

Sets w += a * v

84.6.28

Mat& **scale** (double c)

Scale elements

84.6.29

double **assoc** (BinAssoc & op) const

perform an asscociative function on all non-null elements

84.6.30

double **sum** () const

Sum of all elements

84.6.31

double **sum_abs** () const

Sum of absolute value of all elements

84.6.32

double **max** () const

Max of all elements

84.6.33

double **max_abs** () const

Max of absolute value of all elements

84.6.34

double **min** () const

Min of all elements

84.6.35

void **accum** (double &v, Accumulator & acc) const

perform an asscociative function on all non-null elements

84.6.36

double **sum_sq** () const

Sum of squares of all elements

84.6.37

double **sum_pow** (double n) const

Sum of power of all absolute value of all elements

84.6.38

void **print** (const char* s = NULL) const

print elements (sparse version)

84.6.39

void **print_f** (const char* s = NULL) const

print elements (full version)

84.6.40

Mat& **resize** (int m, int n)

Resize vectors, truncates elements if greater than this value

84.6.41

Mat& **clear** ()

Clears all elements

84.6.42

Mat& **grow** (int g)

Set mode if can grow automatically or not

84.6.43

int **empty** () const

Flags if the vector is empty or not

84.6.44

int **size** () const

Number of non null elements

84.6.45

void **solve** (FullVec &b)

Solve the linear system

84.6.46

void **solve** (double* b)

Solve the linear system

84.6.47

void **duplicate** (const Mat &B)

Duplicate content (not factored state)

84.6.48

virtual void **clean_factor** ()

FSM actions

84.6.7

TextHashTable **thash**

The options hash table

84.6.1

int **nrows**

Dimensions

84.6.2

int **grow_m**

Flag indicating where you can add values past the specified dimensions

84.6.3

static double **not_represented_val**

Value of those elements that are not represented

84.6.4

void **init_fsm** (Mat*)

The internal options hash table. If no external hash table is

84.6.5

void **print_compact** (const char* s = NULL) const

print elements (compact sparse version)

84.6.6

void **print_matlab** (const char* s = NULL) const

print elements (matlab sparse version)

85

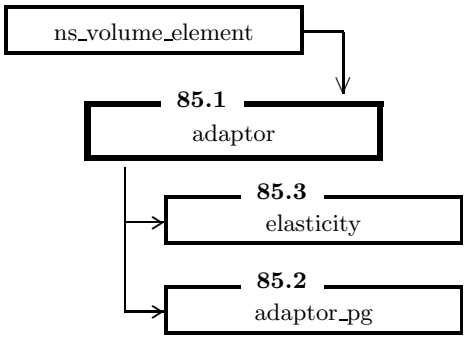
Navier-Stokes and general non-linear module

Names

85.1

class **adaptor** : public ns_volume_element

Inheritance



Public Members

Private Members

Generic class that allows the user not make explicitly the element loop, but instead he has to code the 'element_connector' function that computes the residual vector and jacobian of the element.

85.1.4

ASSEMBLE_FUNCTION

This should not be defined by the user

85.1.5

double **rec_Dt**

the reciprocal of the time step. (May be null for steady problems)

85.1.6

double **alpha**

The trapezoidal rule parameter

85.1.7

int **npg**

The number of Gauss points

85.1.8

int **ndim**

The dimension of the space

85.1.9

int **ndimel**

The dimension of the element

85.1.10

int **elem**

The element number (may be used for printing errors, for instance)

85.1.11

int **fem_interpolation**

Use FEM interpolation (shape, dshapexi members)

85.1.12

FastMat2 **shape**

Shape function at this GP (size `nel`).

85.1.13

FastMat2 **dshapexi**

Gradient with respect to master element coordinates (size `ndimel x nel`)

85.1.14

FastMat2 **dshapex**

Gradient with respect to global coordinates. It is only dimensioned but it should be computed by the user. (size `ndim x nel`)

85.1.15

FastMat2 **wpg**

Gauss points weights (size `nel`)

85.1.16

GlobParam* **glob_param**

Parameters passed to the element from the main

85.1.17

virtual void **init** ()

User defined callback function to be defined by the user. Called **before** the element loop. May be used for initialization operations.

85.1.18

virtual void **clean** ()

User defined callback function to be defined by the user. Called **after** the element loop. May be used for clean-up operations.

85.1.19

virtual void
element_connector (const FastMat2 &xloc, const FastMat2 &state_old, const FastMat2 &state_new, FastMat2 &res, FastMat2 &mat)

Callback that computes the residual and jacobian of the element contribution.

Parameters:

`xloc`	(input) Coordinates of the nodes.
`state_old`	(input) The state at time t^n
`state_new`	(input) The state at time t^{n+1}
`res`	(output) residual vector
`mat`	(input) jacobian of the residual with respect tox^{n+1}

85.1.20

virtual void
element_connector_analytic (const FastMat2 &xloc, const FastMat2 &state_old, const FastMat2 &state_new, FastMat2 &resa, FastMat2 &mata)

Callback similar to the `element_connector()`, the difference cames when the option `use_jacobian_fdj` to compute Jacobians of the residual using finite differences is activated. The idea is that each of the callbacks compute a term in the residual. In the following we use the convention to call `resd`, `matd` the values returned by `element_connector()` and `resa`, `mata` those returned by `element_connector_analytic()`. The residual as seen from the FEM program is the sum of the two, i.e. `res=resd+resa`. If the Jacobians are computed analytically (i.e. `use_jacobian_fdj=0`) then the Jacobian will be simply `matd+mata`. If the Jacobians are computed by finite differences (i.e. `use_jacobian_fdj=1`) then the Jacobian will be `matd_fdj+mata`, where `matd_fdj` is the Jacobian of the `resd` part computed by perturbation of the `element_connector()` analytic function. Note that `matd` is irrelevant when `use_jacobian_fdj=1`. The idea is that the computation of FD Jacobians is expensive, so that if some terms of the residual are expensive to compute but their Jacobians can be easily computed analytically, then it is cheaper to separate this part, and leave the FD computation only for the rest. Another optimization may be to compute expensive stuff that do not depend on the state only once (see doc for `prtb_index()`).

Parameters:

`xloc`	(input) Coordinates of the nodes.
`state_old`	(input) The state at time t^n
`state_new`	(input) The state at time t^{n+1}
`res`	(output) residual vector
`mat`	(input) jacobian of the residual with respect tox^{n+1}

85.1.21

virtual void **element_init** ()

This is called only once for each element after calling initialize().

85.1.22

int **prtb_index** ()

Returns the stage we are performinf when computing Jacobians by finite differences. The stage number returned may be `j=-1` (no stage), `j=0` (initialization, reference values), `1<=j<=nen` computing residual for the perturbation of the `j` component of the element state vector.

Return Value: `stage` number

85.1.23

FastMat2 **Hloc**

Contains constant fields for the element

85.1.1

int **elem_init_flag**

Flags whether the elements have been initialized or not

85.1.2

int **ielh**

Index of element local to chunk

85.1.3

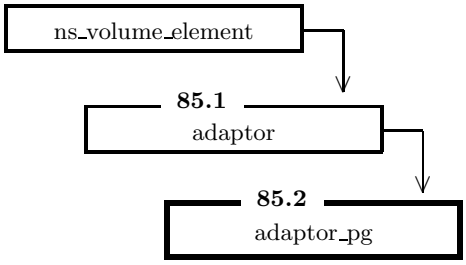
Perturbed index, if=-1: not in FDJ loop,

Perturbed index, if=-1: not in FDJ loop,

85.2

class **adaptor_pg** : public adaptor

Inheritance



Public Members

Private Members

Allows to define elements only by its contributions at Gauss points (GP). The user is passed the GP coordinates, the state and gradient of state at the GP, for both the state at this time step and the next time step, and should return the residual and contribution to the Jacobian matrix. One has access also to the shape function, This adaptor may be used also for `ndimel<ndim`, for instance quad panels in 3D. In that case the gradients of the states and shape functions are also of size `ndim*...` and are parallel to the surface. Also the user has acces to the normal to the element.

85.2.4

Call back functions.

Names

pg_connector (const FastMat2 &xpg, const FastMat2 &state_old_pg,
const FastMat2 &grad_state_old_pg,
const FastMat2 &state_new_pg,
const FastMat2 &grad_state_new_pg, FastMat2 &res_pg,
FastMat2 &mat_pg)
Callback function that defines the residual and matrix at the Gauss point. 331

85.2.4.1

virtual void **elemset_init** ()

Callback hook to be executed before a chunk of elements

85.2.4.2

virtual void **elem_init** ()

Callback hook to be executed before a specific element

85.2.4.3

virtual void **elem_end** ()

Callback hook to be executed after a specific element

85.2.4.4

virtual void **elemset_end** ()

Callback hook to be executed after a chunk of elements

85.2.4.5

virtual void
pg_connector (const FastMat2 &xpg, const FastMat2 &state_old_pg, const FastMat2 &grad_state_old_pg, const FastMat2 &state_new_pg, const FastMat2 &grad_state_new_pg, FastMat2 &res_pg, FastMat2 &mat_pg)

Callback function that defines the residual and matrix at the Gauss point. This shouldn't scale by the Gauss

weight, neither by the Gauss point volume. Also this function should **not** accumulate on `mat` and `res`. It should **set** those variables. (Eventually reset to 0.)

Parameters:

`xpg`	(input) coordinates of the Gauss point (size `ndim`)
`state_old_pg`	(input) state vector at the GP (size `ndof`)
`grad_state_old_pg`	(input) gradient of state vector at theGP (size `ndim*ndof`).
`state_new_pg`	(input) state vector at the GP (size `ndof`)
`grad_state_new_pg`	(input) gradient of state vector at theGP (size `ndim*ndof`).
`res_pg`	(output) Residual computed by the routine. (size `nel*ndof`)
`mat_pg`	(output) Jacobian of the residual (sign reverted, i.e. `-dR/dU`),(size `nel*ndof*nel*ndof`).

85.2.5

FastMat2& **normal** ()

Returns the normal to the element (in the case `ndimel<ndim`).

85.2.1

void **init** ()

User defined callback function for the 'adaptor' class. Implemented in this class.

85.2.2

void **clean** ()

User defined callback function for the 'adaptor' class. Implemented in this class.

85.2.3

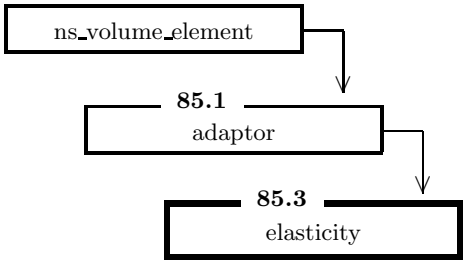
void
element_connector (const FastMat2 &xloc, const FastMat2 &state_old, const FastMat2 &state_new, FastMat2 &res, FastMat2 &mat)

User defined callback function for the 'adaptor' class. Implemented in this class.

85.3

class **elasticity** : public adaptor

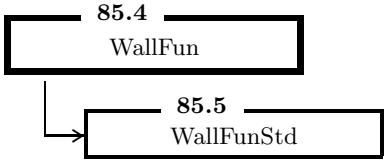
Inheritance



85.4

class **WallFun**

Inheritance



Public Members

85.4.1 virtual void
init ()
Initialize the object 334

85.4.2 virtual void
w (double yp, double &f, double &fp)
Compute the function $f = f(y^+)$ and $fp = f'(y^+)$. 334

85.4.3 virtual
~WallFun ()
The destructor 334

This is the generic wall function

85.4.1

virtual void **init** ()

Initialize the object

85.4.2

virtual void **w** (double yp, double &f, double &fp)

Compute the function $f = f(y^+)$ and $fp = f'(y^+)$. `yp`=y^+ is the nondimesional coordinate normal to the wall.

85.4.3

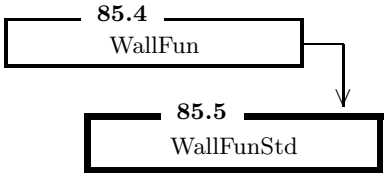
virtual **~WallFun** ()

The destructor

85.5

class **WallFunStd** : public WallFun

Inheritance



Public Members

Private Members

The standard wall function, composed of a linear laminar profile, a buffer region and the logarithmic region.

85.5.3

void **w** (double yp, double &f, double &fprime)

The specific wall function

85.5.4

WallFunStd (Elemset* e)

Constructor

85.5.5

~WallFunStd ()

Destructor

85.5.1

Elemset* **elemset**

A pointer to the elemeset in order to get the physical data.

85.5.2

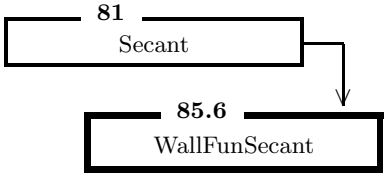
double **c1**

Constants of the law

85.6

class **WallFunSecant** : public Secant

Inheritance



Public Members

This class provides the solution of the equation for the friction velocity with the secant method.

85.6.1

double **nu**

viscosity

85.6.2

double **y_wall**

coordinate normal to the wall

85.6.3

double **u**

Velocity at the point near the wall

85.6.4

double **rho**

Density

85.6.5

WallFun* **wf**

Specific wall function

85.6.6

double **residual** (double ustar, void* user_data=NULL)

Provides the residual of the function to be solved

85.6.7

WallFunSecant (WallFun* wf_)

Constructor from the wall function

85.6.8

void
solve (double u_, double &ustar, double &tau_w, double &yplus, double &fwall, double &fprime, double &dustar_du)

Solve the nonlinear equation in the friction velocity for a given velocity at the near wall. The wall law $u/u_* = f(y_w u_*/\nu)$ gives a non-linear equation in u_* for a given u. So that we can eliminate $u = u(u_*)$.

Parameters:

`u`	(input) velocity at the near wall
`ustar`	(output) friction velocity
`tau_w`	(input) traction (friction) at the wall
`yplus`	(output) non-dimensional normal coordinate atthe near wall
`fwall`	(output) wall function at the near wall
`fprime`	(output) slope of the wall function at the nearwall
`dustar_du`	(output) is du/du_*.

85.7

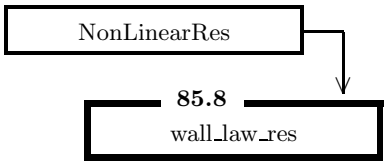
#define **NonLinearRes**

Generic nonlinear restriction element. It may not work for restrictions that involve fields in more that one node.

85.8

class **wall_law_res** : public NonLinearRes

Inheritance



Public Members

85.8.18 int
nres ()
Number of restrictions (=2) 340

85.8.19 void

init ()
Initialize the elemset (maybe reads hash table) 340

85.8.20 void
res (int k, FastMat2 &U, FastMat2 & r, FastMat2 & lambda, FastMat2 & jac)
computes the residual and jacobian of the function to be imposed 340

85.8.21 **wall_law_res** ()
Contructor 340

85.8.22 **~wall_law_res** ()
Destructor 340

Private Members

85.8.1 int **nk**
Index (field numbers) of k and ϵ, number of dimensions 341

85.8.2 double **y_wall_plus**
Normal coordinate where the wall law is imposed (non-dimensional) 341

85.8.3 double **y_wall**
Normal coordinate where the wall law is imposed (dimensional) 341

85.8.4 double **fwall**
Wall function at the near wall. 341

85.8.5 double **fprime**
Slope of wall function at the near wall. 341

85.8.6 double **C_mu**
Turbulence model constant 341

85.8.7 double **von_Karman_cnst**
Turbulence model constant 342

85.8.8 double **viscosity**
Viscosity 342

85.8.9 double **turbulence_coef**
Deactivate turbulence 342

85.8.10 WallFun* **wf**
Specific wall function 342

85.8.11 WallFunSecant* **wfs**
Wall function with secant solver if use y =cnst instead of y^+ =cnst 342

85.8.12 int **u_wall_indx**
Index for `u_wall` *property* 342

85.8.13 int **nprops**
Number of properties 343

85.8.14 FastMat2 **u_wall**
Absolute velocity at the near wall 343

85.8.15 FastMat2 **du_wall**
Relative velocity at the near wall 343

85.8.16 int

elprpsindx [MAXPROP_WLR]
Array of property indices 343

85.8.17 double
propel [MAXPROP_WLR]
Array of properties 343

This elemset provides the restriction (via Lagrange multipliers) of the nonlinear Dirichlet type bc. at the wall of the form $k_w = u_*^2/C_\mu$ and similarly for ϵ_w.

85.8.18

int **nres** ()

Number of restrictions (=2)

85.8.19

void **init** ()

Initialize the elemset (maybe reads hash table)

85.8.20

void **res** (int k, FastMat2 &U, FastMat2 & r, FastMat2 & lambda, FastMat2 & jac)

computes the residual and jacobian of the function to be imposed

85.8.21

wall_law_res ()

Contructor

85.8.22

~wall_law_res ()

Destructor

85.8.1

int **nk**

Index (field numbers) of k and ϵ, number of dimensions

85.8.2

double **y_wall_plus**

Normal coordinate where the wall law is imposed (non-dimensional)

85.8.3

double **y_wall**

Normal coordinate where the wall law is imposed (dimensional)

85.8.4

double **fwall**

Wall function at the near wall.

85.8.5

double **fprime**

Slope of wall function at the near wall.

85.8.6

double **C_mu**

Turbulence model constant

85.8.7

double **von_Karman_cnst**

Turbulence model constant

85.8.8

double **viscosity**

Viscosity

85.8.9

double **turbulence_coef**

Deactivate turbulence

85.8.10

WallFun* **wf**

Specific wall function

85.8.11

WallFunSecant* **wfs**

Wall function with secant solver if use y =cnst instead of y^+ =cnst

85.8.12

int **u_wall_indx**

Index for `u_wall` property

85.8.13

int **nprops**

Number of properties

85.8.14

FastMat2 **u_wall**

Absolute velocity at the near wall

85.8.15

FastMat2 **du_wall**

Relative velocity at the near wall

85.8.16

int **elprpsindx** [MAXPROP_WLR]

Array of property indices

85.8.17

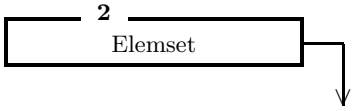
double **propel** [MAXPROP_WLR]

Array of properties

85.9

class **wallke** : public Elemset

Inheritance



85.9

wallke

Wall elemset. Imposes traction as a function of velocity vial universal wall law functions.

85.10

double **ctff** (double x, double & diff_ctff, double tol=1e-5)

Cutoff function used in turbulence calculations. It is very near to $\mathrm{ctff}(x) \approx \mathrm{tol}$ for $x < 0$ and $\mathrm{ctff}(x) = x$ for $x \gg \mathrm{tol}$.

Return Value: `the` cutoff'ed value

Parameters:

`x`	(input) the argument where to compute the cutoff fuction
`(output)`	the derivative of the cutoff function at x
`(input)`	cuttoff scale parameter.

85.11

void
read_cond_matrix (TextHashTable* thash, const char* s, int ndof, FastMat2 &cond)

Reads a `FastMat2` matrix from a `TextHashTable` entry.

Return Value: `a` reference to the matrix.

Parameters:

`thash`	(input) the options table
`s`	(input) the key in the table where to extract the matrix coefficients
`ndof`	(input) Let `coef[]` be the list of coefficients entered inthe line and `ncoef` the length of `coef`. If `ncoef=1` then `cond = v[0] * Id(ndof)`, else if `ncoef=ndof` then, `cond = diag(v)`, else if `ncoef=ndof*ndof` then `cond(i,j) = v[ndof*i+j]` (rowwise). Any other causes an error.
`ndof`	(input) the number of fields
`cond`	(output) the matrix to be read

85.12

BasicObject_factory_t **BasicObject_ns_factory**

This is the factory of BasicObjects (Objects that are read from the user data file), specific for the NS module

85.13

void **detj_error** (double &detJaco, int elem)

Fixes the jacobian of the element.

85.14

Hook* **ns_hook_factory** (const char* name)

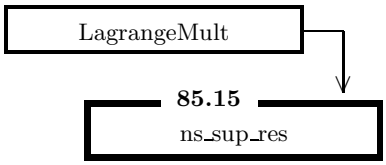
Creates hooks depending on the name.

Return Value: `a` pointer to the hook.
Parameters: `name` (input) the name of the hook.

85.15

class **ns_sup_res** : public LagrangeMult

Inheritance



Public Members

85.15.4 int
nres ()
Number of restrictions 346

85.15.5 void

Private Members

Restriction for linearized free surface boundary condition

85.15.4

int **nres** ()

Number of restrictions

85.15.5

void **lag_mul_dof** (int jr, int &node, int &dof)

Return the node/dof pair to be used as lagrange multiplier for the `jr`-th restriction.

85.15.6

void **init** ()

Initialize the elemset (maybe reads hash table)

85.15.7

void **res** (int k, FastMat2 &U, FastMat2 & r, FastMat2 & lambda, FastMat2 & jac)

computes the residual and jacobian of the function to be imposed

85.15.8

void **close** ()

Called after the loop over all elements

85.15.9

void **lm_initialize** ()

Initializes the elemset even if there are not elements in this processor

85.15.1

int **ndim**

The dimension of the problem

85.15.2

int **p_indx**

The index (number of degree of freedom) for pressure

85.15.3

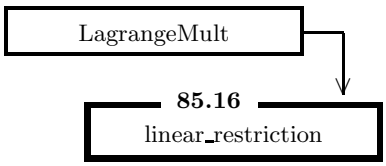
double **gravity**

Gravity acceleration

85.16

class **linear_restriction** : public LagrangeMult

Inheritance



Public Members

85.16.6 int
nres ()
Number of restrictions 349

85.16.7 void
lag_mul_dof (int jr, int &node, int &dof)
Return the node/dof pair to be used as lagrange multiplier for the `jr`*-th restriction.* 349

85.16.8 void
init ()
Initialize the elemset (maybe reads hash table) 349

85.16.9 void
res (int k, FastMat2 &U, FastMat2 & r, FastMat2 & lambda, FastMat2 & jac)
computes the residual and jacobian of the function to be imposed 349

Private Members

85.16.1 FastMat2 **coef**
Coefficients of the restrictions. 349

85.16.2 int **nres_m**
Number of restrictions 350

85.16.3 int **was_loaded**
Flags whether the coefs. 350

85.16.4 vector<int> **node_lm**

General restriction elemset

85.16.6

int **nres** ()

Number of restrictions

85.16.7

void **lag_mul_dof** (int jr, int &node, int &dof)

Return the node/dof pair to be used as lagrange multiplier for the `jr`-th restriction.

85.16.8

void **init** ()

Initialize the elemset (maybe reads hash table)

85.16.9

void **res** (int k, FastMat2 &U, FastMat2 & r, FastMat2 & lambda, FastMat2 & jac)

computes the residual and jacobian of the function to be imposed

85.16.1

FastMat2 **coef**

Coefficients of the restrictions.

85.16.2

int **nres_m**

Number of restrictions

85.16.3

int **was_loaded**

Flags whether the coefs. have been read

85.16.4

vector<int> **node_lm**

Local nodes, dofs to be used as Lagrange multipliers

85.16.5

vector<int> **dofs_lm**

Local nodes, dofs to be used as Lagrange multipliers

86

AdvDif module

Names

86.1

#define **AJAC** (jd)

The jacobians of the flux functions. This is an array of ndim matrices of ndof x ndof entries each.

86.2

#define **FLUX_FUN_ARGS**

Standard arguments for the typical flux function. Newmat version.

86.3

#define **FLUX_FUN_CALL_ARGS_GENER** (ML_)

Standard arguments for the typical flux function (in the function call)

86.4

typedef int **AdvDifFluxFunction** (AD_FLUX_FUN_ARGS)

Type of flux function.

86.5

class **FastMat2Shell**

Public Members

86.5.1 virtual
apply (FastMat2 & A, FastMat2 & B) const
*Performs A = (*this) * B;* 355

Generic FastMat2 matrix

86.5.1

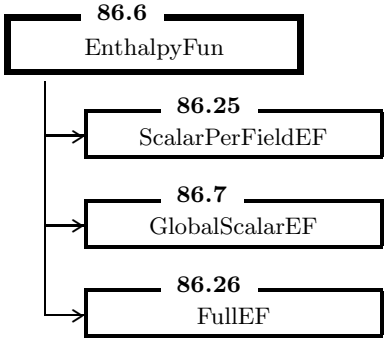
virtual **apply** (FastMat2 & A, FastMat2 & B) const

Performs A = (*this) * B;

86.6

class **EnthalpyFun**

Inheritance



Public Members

Virtual class thet defines the relation between vector state and enthalpy content.

86.6.1

FastMat2 **UU**

The actual state

86.6.2

virtual void **update** (const double* e)

Allows updating the data for the object.

Parameters: `e` (input) cofficients for updating the object

86.6.3

virtual void **set_state** (const FastMat2 &U)

Allows setting the state of the object

Parameters: `U` (input) the state of the object

86.6.4

virtual void **enthalpy** (FastMat2 &H, const FastMat2 &U)

Computes the enthalpy vector from the state vector

Parameters:

`H`	(output) the enthalpy content vector
`U`	(input) the state vector

86.6.5

virtual void **enthalpy** (FastMat2 &H)

Computes the enthalpy vector from the state vector. Uses the state previously set with 'set_state'

Parameters:

`H`	(output) the enthalpy content vector

86.6.6

virtual void
comp_W_Cp_N (FastMat2 &W_Cp_N, const FastMat2 &W, const FastMat2 &N, double w)

Computes the product `(W_Cp_N)_(p,mu,q,nu) = W_p N_q Cp_(mu,nu)`

Parameters:

`W_Cp_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`W`	(input) weight function, size `nel`
`N`	(input) interpolation function, size `nel`
`w`	(input) scalar weight

86.6.7

virtual void **comp_P_Cp** (FastMat2 &P_Cp, const FastMat2 &P_supg)

Computes the product `(P_Cp)_(mu,nu) = (P_supg)_(mu,lambda) Cp_(lambda,nu)`

Parameters:

`P_Cp`	(output) size `ndof` x `ndof`
`P_supg`	(input) matricial weight function, size `ndof` x `ndof`

86.6.8

virtual void **get_Cp** (FastMat2 &Cp)

Gets the `Cp` jacobian

86.6.9

virtual void **comp_P_Gamma** (FastMat2 &P_Ga, const FastMat2 &P_supg)

Computes the product `(P_Gamma)_(mu,nu) = (P_supg)_(mu,lambda) preco_(lambda,nu)`

Parameters:

`P_Gamma`	(output) size `ndof` x `ndof`
`P_supg`	(input) matricial weight function, size `ndof` x `ndof`

86.6.10

virtual void
comp_W_Gamma_N (FastMat2 &W_Ga_N, const FastMat2 &W, const FastMat2 &N, double w)

Computes the product `(W_Gamma_N)_(p,mu,q,nu) = W_p N_q preco_(mu,nu)`

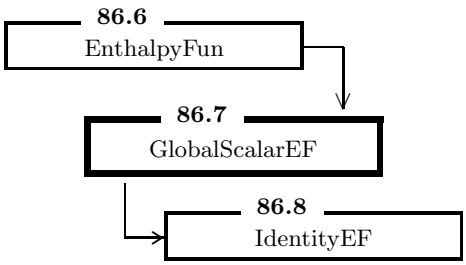
Parameters:

`W_Ga_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`W`	(input) weight function, size `nel`
`N`	(input) interpolation function, size `nel`
`w`	(input) scalar weight

86.7

class **GlobalScalarEF** : public EnthalpyFun

Inheritance



Public Members

Protected Members

Private Members

Constant Cp for all fields

86.7.3

void **init** (int ndim, int ndof, int nel, double Cp=1.)

initializes dimensions and sets Cp

86.7.4

void **update** (const double* Cp_)

Sets Cp from elemset data

86.7.5

void **enthalpy** (FastMat2 &H)

Multiplies U by Cp with 'scale'

86.7.6

void
comp_W_Cp_N (FastMat2 &W_Cp_N, const FastMat2 &W, const FastMat2 &N, double w)

Scales at the same time by w*Cp

86.7.7

void **comp_P_Cp** (FastMat2 &P_Cp, const FastMat2 &P_supg)

Scales `P_supg` by `Cp`

86.7.2

double **Cp**

The actual Cp

86.7.1

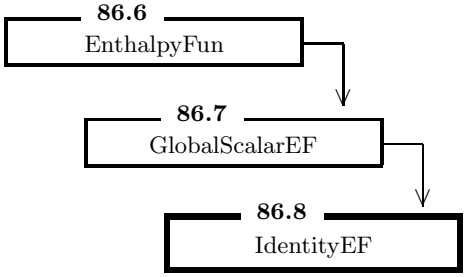
FastMat2 **eye_ndof**

Aux var. identity of size ndof

86.8

class **IdentityEF** : public GlobalScalarEF

Inheritance



Public Members

Constant Cp=1 for all the fields. Identity relation between H and T

86.8.1

void **update** (const double* Cp_)

Does nothing

86.8.2

void **enthalpy** (FastMat2 &H)

Copies `U` in `H`

86.8.3

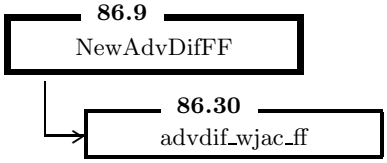
void **comp_P_Cp** (FastMat2 &P_Cp, FastMat2 &P_supg)

Copies `P_supg` in `P_Cp`

86.9

class **NewAdvDifFF**

Inheritance



Public Members

Private Members

This is the flux function for a given physical problem.

86.9.3

const NewElemset* **elemset**

The elemset associated with the flux function

86.9.4

const NewAdvDif* **new_adv_dif_elemset**

Pointer to NewAdvDif elemset (FIXME:= 'elemset' doesn't serve because it is a pointer to 'NewElemset')

86.9.5

EnthalpyFun* **enthalpy_fun**

The enthalpy function for this flux function

86.9.6

NewAdvDifFF (const NewElemset* elemset_=NULL)

Constructor from the elemset

86.9.7

virtual void **get_log_vars** (int &nlog_vars, const int* & log_vars)

Define the list of variables that are treated logarithmically. Reads from the options `nlog_vars` and `log_vars`.

86.9.8

virtual void **start_chunk** (int &ret_options)

This is called before any other in a loop and may help in optimization

Parameters:

`ret_options`	(input/output) this is used by the fluxfunction writer for returning some options. Currently the onlyoption used is `SCALAR_TAU`. This options tells the elemsetwhether the flux function returns a scalar or matrix`tau_supg`.

86.9.9

virtual void **element_hook** (ElementIterator &element)

This is called before entering the Gauss points loop and may help in optimization.

Parameters:

`element`	(input) an iterator on the elemlist.

86.9.10

Advective jacobians related

Names

86.9.10.1

virtual void **comp_A_grad_N** (FastMat2 & A_grad_N, FastMat2 & grad_N)

Computes the product `(A_grad_N)_(p,mu,nu) = A_(i,mu,nu) (grad_N)_(i,p)`

Parameters:

`A_grad_N`	(output, size `nel` x `nd` x `nd`)
`grad_N`	(input, size `nel` x `ndof`)

86.9.10.2

virtual void **comp_A_jac_n** (FastMat2 &A_jac_n, FastMat2 &normal)

Computes the product `(A_jac_n)_(mu,nu) = A_(i,mu,nu) normal_i`

Parameters:

`A_jac_n`	(output, size `ndof` x `ndof`)
`normal`	(input, size `ndim`)

86.9.10.3

virtual void **set_state** (const FastMat2 &U)

This sets the local state of the flow and is called before to call all the enthalpy functions functions and the like (those that don't need the state gradient `grad_U`.

Parameters:

`U`	(input) the local state of the fluid

86.9.10.4

virtual void **set_state** (const FastMat2 &U, const FastMat2 &grad_U)

This sets the local state of the flow and is called before to call all the 'comp_grad_N_D_grad_N' functions and alike.

Parameters:

`U`	(input) the local state of the fluid
`grad_U`	(input) the local gradient of state of the fluid

86.9.10.5

virtual void **compute_flux** (COMPUTE_FLUX_ARGS)

Computes fluxes, upwind parameters etc. fixme:= more doc here ...

86.9.11

Diffusive jacobians related

Names

86.9.11.1 virtual void
comp_grad_N_D_grad_N (FastMat2 &grad_N_D_grad_N, FastMat2 & dshapex, double w)
Computes the product `(grad_N_D_grad_N)_(p, mu, q, nu) = D_(i, j, mu, nu) (grad_N)_(i, p) (grad_N)_(j, q)` 366

86.9.11.1

virtual void
comp_grad_N_D_grad_N (FastMat2 &grad_N_D_grad_N, FastMat2 & dshapex, double w)

Computes the product `(grad_N_D_grad_N)_(p,mu,q,nu) = D_(i,j,mu,nu) (grad_N)_(i,p) (grad_N)_(j,q)`

Parameters:

`grad_N_D_grad_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`grad_N`	(input) size `nel` x `ndof`

86.9.12

Diffusive jacobians related to dependences of diffusion coeff to state variable

Names

86.9.12.1 virtual int
comp_grad_N_dDdU_N (FastMat2 &grad_N_dDdU_N, FastMat2 &grad_U, FastMat2 &dshapex, FastMat2 &N, double w)
Computes the product `(grad_N_dDdU_N)_(p, mu, q, nu) = dDdU_(i, mu, nu) (grad_N)_(i, p) (N)_(q)` 367

86.9.12.1

virtual int
comp_grad_N_dDdU_N (FastMat2 &grad_N_dDdU_N, FastMat2 &grad_U, FastMat2 &dshapex, FastMat2 &N, double w)

Computes the product `(grad_N_dDdU_N)_(p,mu,q,nu) = dDdU_(i,mu,nu) (grad_N)_(i,p) (N)_(q)`

Return Value: `1` if flux function truly implements this, 0 otherwise.
Parameters:

`grad_N_dDdU_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`grad_N`	(input) size `nel` x `ndof`
`N`	(input) size `nel`

86.9.13

Reactive jacobians related

Names

86.9.13.1 virtual void
comp_N_N_C (FastMat2 &N_N_C, FastMat2 &N, double w)
Computes the product `(N_N_C)_(p, mu, q, nu) = w C_(mu, nu) N_p N_q` 368

86.9.13.2 virtual void
comp_N_P_C (FastMat2 &N_P_C, FastMat2 &P_supg, FastMat2 &N, double w)
Computes the product `(N_P_C)_(mu, q, nu) = w (P_supg)_(mu, lambda) C_(lambda, nu) N_q` 368

86.9.13.3 virtual void

86.9.13.1

virtual void **comp_N_N_C** (FastMat2 &N_N_C, FastMat2 &N, double w)

Computes the product `(N_N_C)_(p,mu,q,nu) = w C_(mu,nu) N_p N_q`

Parameters:

`N_N_C`	(output, size `nel` x `ndof` x `nel` x `ndof`
`N`	(input) FEM interpolation function size `nel`
`w`	(input) a scalar coefficient

86.9.13.2

virtual void
comp_N_P_C (FastMat2 &N_P_C, FastMat2 &P_supg, FastMat2 &N, double w)

Computes the product `(N_P_C)_(mu,q,nu) = w (P_supg)_(mu,lambda) C_(lambda,nu) N_q`

Parameters:

`N_P_C`	(output) size `ndof` x `nel` x `ndof`
`P_supg`	(input) SUPG perturbation function size `ndof` x `ndof`
`N`	(input) FEM interpolation function size `nel`
`w`	(input) a scalar coefficient

86.9.13.3

virtual void **comp_P_supg** (FastMat2 &P_supg)

Computes the SUPG perturbation function from the gradient of the shape function. If you want to define a formulation for the SUPG perturbation function that can't be cast in the standard form (i.e. it is not of the form `tau * A * grad_N` then you should define `tau` as scalar (may be 0) and then compute your own expression for `P_supg`.

Parameters: `P_supg` (output) the SUPG perturbation function for allnodes at this Gauss point, has dimensions `nel` x `ndof` x`ndof`

86.9.13.4

virtual int **dim** () const

Returns the dimension of the element (May be different from the dimension space). For instance, a river may be a 1D elemset in a 2D space. If this is equal to the space dimension (the most common case) then return -1.

Return Value: `the` dimension of the advective elemset.

86.9.13.5

virtual void
Riemann_Inv (const FastMat2 &U, const FastMat2 &normal, FastMat2 &Rie, FastMat2 &drdU, FastMat2 &C_)

Returns the Riemann Invariants and jacobians for Adv-Diff absorbent condition

86.9.13.6

virtual void **set_Ufluid** (FastMat2 &Uref, FastMat2 &Ufluid)

Sets "Ufluid" state extracted from Uref state and related to particular flux function in order to use ULSAR absorb boundary conditions

86.9.1

vector<int> **log_vars_v**

The list of variables to be logarithmically transformed

86.9.2

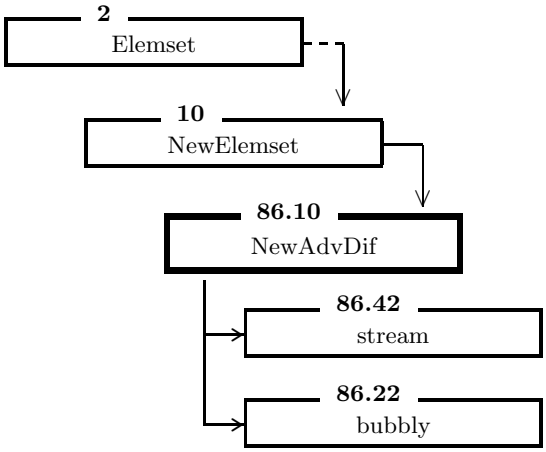
FastMat2 **tmp_P_supg_ALE_1**

Needed for ALE computations

86.10

class **NewAdvDif** : public NewElemset

Inheritance



Public Members

86.10.6 **NewAdvDif** (NewAdvDifFF* adv_diff_ff_=NULL)
Contructor from the pointer to the fux function 371

86.10.7 ~**NewAdvDif** ()
Destructor. 372

86.10.8 void
before_assemble (arg_data_list &arg_datav, Nodedata* nodedata, Dofmap* dofmap, const char* jobinfo, int myrank, int el_start, int el_last, int iter_mode, const TimeData* time_data)
Prepare variables for report of error on flux advective jacobians 372

86.10.9 void

Protected Members

Generic elemset for advective diffusive problems. Several physical problems may be solved by defining the corresponding flux function object.

86.10.6

NewAdvDif (NewAdvDifFF* adv_diff_ff_=NULL)

Contructor from the pointer to the fux function

86.10.7

~NewAdvDif ()

Destructor. Destroys the flux function object. fixme:= Warning: this is not good!! We cannot destroy the flux function object here if it is built in the derived class, because it may happen, for instance, that we pass a pointer to a global object. To be fixed later...

86.10.8

void
before_assemble (arg_data_list &arg_datav, Nodedata* nodedata, Dofmap* dofmap, const char* jobinfo, int myrank, int el_start, int el_last, int iter_mode, const TimeData* time_data)

Prepare variables for report of error on flux advective jacobians

86.10.9

void **after_assemble** (const char* jobinfo)

Report erros on jacobian fluxes

86.10.10

NewAssembleFunction **new_assemble**

The assemble function for the elemset.

86.10.11

NewAssembleFunction **new_assemble_ALE_formulation**

The assemble function for the elemset (version with ALE+GCL formulation)

86.10.12

NewAssembleFunction **new_assemble_BDF**

The assemble function for the elemset (version with BDF time integration)

86.10.13

NewAssembleFunction **new_assemble_GCL_compliant**

The assemble function for the elemset (version with GCL)

86.10.14

NewAssembleFunction **new_assemble_preco**

The assemble function for the elemset (version with PRECO)

86.10.15

ASK_FUNCTION

The ask function for the elemset.

86.10.16

int **axi**

axisymmetric key

86.10.17

double **time** () const

Returns the actual time.

Return Value: `actual` time

86.10.18

int **ALE_form** () const

sto se deberia sacar de todas las FF y poner esto

86.10.1

NewAdvDifFF* **adv_diff_ff**

A pointer to the flux function. Different physics are obtained by redefining the flux function

86.10.2

double **rec_Dpt**

Declared for psuedo-time discretization

86.10.3

int **ff_options**

Options returned by the flux function

86.10.4

double **time_m**

The actual time

86.10.5

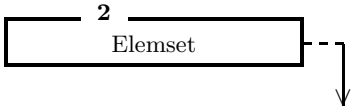
int **compute_fd_adv_jacobian**

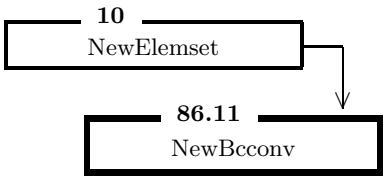
All these are for checking advective jacobians with numerical

86.11

class **NewBcconv** : public NewElemset

Inheritance





Public Members

Private Members

This is the companion elemset to advdif that computes the boundary term when using the weak-form option.

86.11.2

NewBcconv (NewAdvDifFF* adv_diff_ff_=NULL)

Contructor from the pointer to the fux function

86.11.3

~NewBcconv ()

Destructor. Destroys the flux function object.

86.11.4

NewAssembleFunction **new_assemble**

The assemble function for the elemset.

86.11.5

The assemble function modified for being GCL compliant

The assemble function modified for being GCL compliant

86.11.1

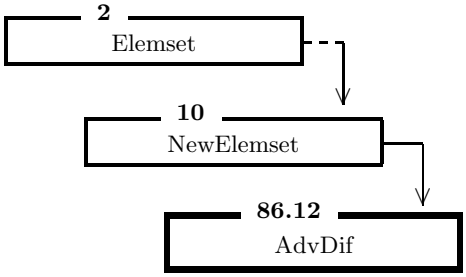
NewAdvDifFF* **adv_diff_ff**

A pointer to the flux function. Different physics are obtained by redefining the flux function

86.12

class **AdvDif** : public NewElemset

Inheritance



The class AdvDif is a NewElemset class plus a advdif flux function object

86.13

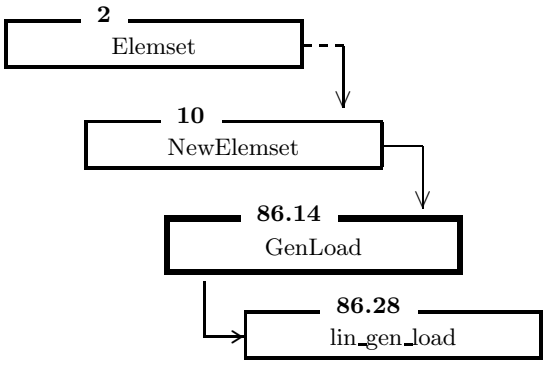
#define **ADVDIF_ELEMSET** (name)

Euler equations for inviscid (Gas dynamics eqs)

86.14

class **GenLoad** : public NewElemset

Inheritance



Private Members

Generic surface flux element

86.14.1

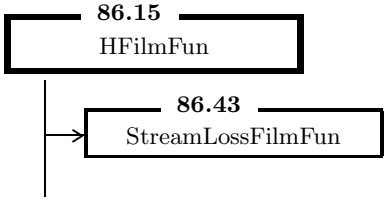
FastMat2 **H_m**

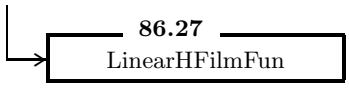
These are to pass the state of the 'H' quantities on the internal and external layers. 'Hin' is an alias for 'H'

86.15

class **HFilmFun**

Inheritance





Generic surface flux function (film function) element

86.16

void
log_transf (FastMat2 &true_lstate, const FastMat2 &lstate, const int nlog_vars, const int* log_vars)

Transforms state vector from logarithmic. The indices of fields logarithmically tranformed are listed in `log_vars`.

Parameters:

`true_lstate`	(output) Transformed from logarithmto positive variable.
`lstate`	(ouput) input state logarithmically transformed(only those fields in `log_vars`).
`nlog_vars`	(input) number of fields logarithmicallytransformed
`log_vars`	(input) list of fields logarithmicallytransformed.

Author: M. Storti CORREGIR:=

86.17

void **detj_error** (double &detJaco, int elem)

Sets an error for the negative jacobian error case.

Parameters:

`detjaco`	(input) the determinant of the jacobian
`elem`	(input) the element number

86.18

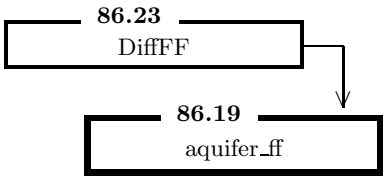
double **ctff** (double x, double & diff_ctff, double tol)

Cutoff function. It is very near to ctff(x) $\approx$ tol for $x < 0$ and ctff$(x) = x$ for $x \gg$ tol.

86.19

class **aquifer_ff** : public DiffFF

Inheritance



Public Members

86.19.6 void
start_chunk ()
This is called before any other in a loop and may help in optimization .. 380

86.19.7 void
element_hook (ElementIterator &element)
This is called before entering the Gauss points loop and may help in optimization. 380

86.19.8 void
gp_hook (int ipg, const FastMat2 &U, const FastMat2 &grad_U)
This is called before each Gauss point. 380

86.19.9 void
compute_flux (const FastMat2 &U, const FastMat2 &grad_U,
FastMat2 &fluxd, FastMat2 &G, FastMat2 &H,
FastMat2 &grad_H)
Computes fluxes, upwind parameters etc. 381

86.19.10 void
comp_grad_N_D_grad_N (FastMat2 &grad_N_D_grad_N, FastMat2 & dshapex,
double w)
Computes the product `(grad_N_D_grad_N)_(p, mu, q, nu) = D_(i, j, mu, nu) (grad_N)_(i, p) (grad_N)_(j, q)` 381

86.19.11 void
enthalpy (FastMat2 &H, FastMat2 &U)
Computes the enthalpy vector from the state vector 381

86.19.12 void
comp_N_Cp_N (FastMat2 &N_Cp_N, FastMat2 &N, double w)
Computes the product `(W_Cp_N)_(p, mu, q, nu) = W_p N_q Cp_(mu, nu)` 381

86.19.13 virtual void
end_chunk ()
This is called after all elements in a chunk 382

Private Members

This is the flux function for a quasi-harmonic equation with a conductivity proportional to the difference between the free surface of the aquifer and its bottom (a known quantity dependent on coordinates)

86.19.6

void **start_chunk** ()

This is called before any other in a loop and may help in optimization

86.19.7

void **element_hook** (ElementIterator &element)

This is called before entering the Gauss points loop and may help in optimization.

Parameters: `element` (input) an iterator on the elemlist.

86.19.8

void **gp_hook** (int ipg, const FastMat2 &U, const FastMat2 &grad_U)

This is called before each Gauss point.

Parameters: `ipg` (input) the Gauss point number (may be used to report errors)

86.19.9

void
compute_flux (const FastMat2 &U, const FastMat2 &grad_U, FastMat2 &fluxd, FastMat2 &G, FastMat2 &H, FastMat2 &grad_H)

Computes fluxes, upwind parameters etc. fixme:= more doc here ...

86.19.10

void
comp_grad_N_D_grad_N (FastMat2 &grad_N_D_grad_N, FastMat2 & dshapex, double w)

Computes the product `(grad_N_D_grad_N)_(p,mu,q,nu) = D_(i,j,mu,nu) (grad_N)_(i,p) (grad_N)_(j,q)`

Parameters:

`grad_N_D_grad_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`grad_N`	(input) size `nel` x `ndof`

86.19.11

void **enthalpy** (FastMat2 &H, FastMat2 &U)

Computes the enthalpy vector from the state vector

Parameters:

`H`	(output) the enthalpy content vector
`U`	(input) the state vector

86.19.12

void **comp_N_Cp_N** (FastMat2 &N_Cp_N, FastMat2 &N, double w)

Computes the product `(W_Cp_N)_(p,mu,q,nu) = W_p N_q Cp_(mu,nu)`

Parameters:

`W_Cp_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`W`	(input) weight function, size `nel`
`N`	(input) interpolation function, size `nel`
`w`	(input) scalar weight

86.19.13

virtual void **end_chunk** ()

This is called after all elements in a chunk

86.19.1

Property **eta_pr**

Basic properties, vertical position of aquifer bottom, hydraulic permeability, storativity

86.19.2

double **phi**

Values of properties at an element

86.19.3

int **ndim**

Dimension of the problem (should be always 2)

86.19.4

int **nel**

Number of nodes per element, number of dof's per node, number of propoerties per element

86.19.5

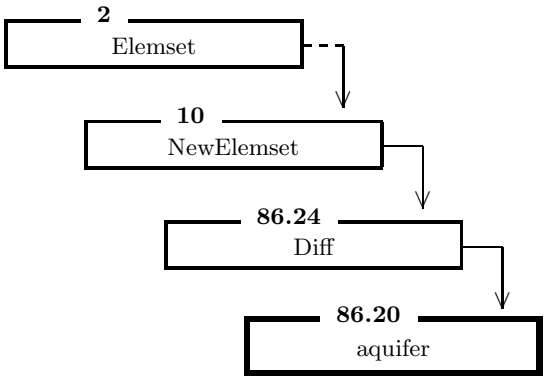
FastMat2 **tmp**

Auxiliary matrices

86.20

class **aquifer** : public Diff

Inheritance



Public Members

This is the elenset derived from the `aquifer_ff` flux function

86.20.1

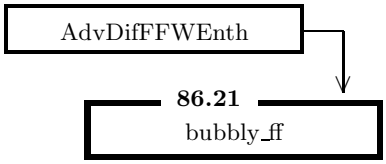
aquifer ()

Constructor, creates the flux function object.

86.21

class **bubbly_ff** : public AdvDifFFWEnth

Inheritance



Public Members

Flux function for multi-phase flow

86.21.1

void **start_chunk** (int &ret_options)

This is called before any other in a loop and may help in optimization

Parameters:

`ret_options` (input/output) this is used by the fluxfunction writer for returning some options. Currently the onlyoption used is `SCALAR_TAU`. This options tells the elemsetwhether the flux function returns a scalar or matrix`tau_supg`.

86.21.2

void **element_hook** (ElementIterator &element)

This is called before entering the Gauss points loop and may help in optimization.

Parameters:

`element` (input) an iterator on the elemlist.

86.21.3

void **set_state** (const FastMat2 &U, const FastMat2 &grad_U)

Basically stores 'U(1)' in the water depth 'u' and computes geometric parameters of the channel

Parameters:

`U`	(input) the state of the fluid
`grad_U`	(input) gradient of the state of the fluid

86.21.4

void **set_state** (const FastMat2 &U)

Basically stores 'U(1)' in the water depth 'u' and computes geometric parameters of the channel

Parameters:

`U`	(input) the state of the fluid

86.21.5

Advective jacobians related

Names

86.21.5.1

void **comp_A_grad_N** (FastMat2 & A_grad_N, FastMat2 & grad_N)

Computes the product `(A_grad_N)_(p,mu,nu) = A_(i,mu,nu) (grad_N)_(i,p)`

Parameters:

`A_grad_N`	(output, size `nel` x `nd` x `nd`)
`grad_N`	(input, size `nel` x `ndof`)

86.21.5.2

void **comp_A_jac_n** (FastMat2 &A_jac_n, FastMat2 &normal)

Computes the product `(A_jac_n)_(mu,nu) = A_(i,mu,nu) normal_i`

Parameters:

`A_jac_n`	(output, size `ndof` x `ndof`)
`normal`	(input, size `ndim`)

86.21.5.3

void **compute_flux** (COMPUTE_FLUX_ARGS)

Computes fluxes, upwind parameters etc. fixme:= include more doc here ...

86.21.6

Diffusive jacobians related

Names

86.21.6.1

void
comp_grad_N_D_grad_N (FastMat2 &grad_N_D_grad_N, FastMat2 & dshapex, double w)

Computes the product `(grad_N_D_grad_N)_(p,mu,q,nu) = D_(i,j,mu,nu) (grad_N)_(i,p) (grad_N)_(j,q)`

Parameters:

`grad_N_D_grad_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`grad_N`	(input) size `nel` x `ndof`

86.21.7

Reactive jacobians related

Names

86.21.7.1

void **comp_N_N_C** (FastMat2 &N_N_C, FastMat2 &N, double w)

Computes the product `(N_N_C)_(p,mu,q,nu) = w C_(mu,nu) N_p N_q`

Parameters:

`N_N_C`	(output, size `nel` x `ndof` x `nel` x `ndof`
`N`	(input) FEM interpolation function size `nel`
`w`	(input) a scalar coefficient

86.21.7.2

void
comp_N_P_C (FastMat2 &N_P_C, FastMat2 &P_supg, FastMat2 &N, double w)

Computes the product `(N_P_C)_(mu,q,nu) = w (P_supg)_(mu,lambda) C_(lambda,nu) N_q`

Parameters:

`N_P_C`	(output) size `ndof` x `nel` x `ndof`
`P_supg`	(input) SUPG perturbation function size `ndof` x `ndof`
`N`	(input) FEM interpolation function size `nel`
`w`	(input) a scalar coefficient

86.21.7.3

void **enthalpy** (FastMat2 &H)

Computes the enthalpy

86.21.7.4

void
comp_W_Cp_N (FastMat2 &W_Cp_N, const FastMat2 &W, const FastMat2 &N, double w)

Computes the product of the enthalpy jacobian matrix with a shape function and a weight function

86.21.7.5

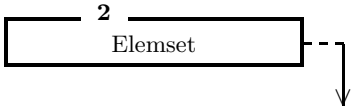
void **comp_P_Cp** (FastMat2 &P_Cp, const FastMat2 &P_supg)

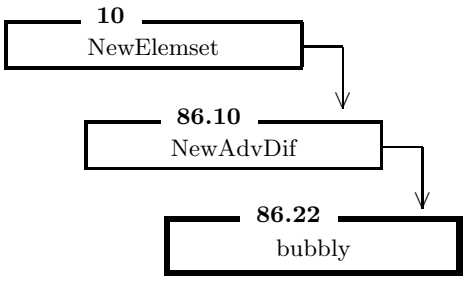
Computes the product of the enthalpy jacobian matrix with an SUPG weight function

86.22

class **bubbly** : public NewAdvDif

Inheritance





Public Members

86.22.1 **bubbly** ()
Constructor, creates the fluc function object. 389

The elemset corresponding to the 'bubbly_ff' flux function

86.22.1

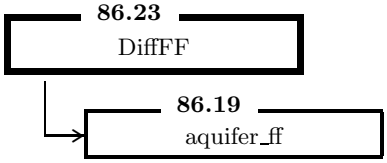
bubbly ()

Constructor, creates the fluc function object. fixme:= should destroy the flux functin.

86.23

class **DiffFF**

Inheritance



Public Members

86.23.1 const Diff* **elemset**
The elemset associated with the flux function 390

86.23.2 **DiffFF** (const Diff* elemset_=NULL)
Constructor from the elemset 390

86.23.3 int
ask (const char* jobinfo, int &skip_elemset)
This flags whether a given task is processed or not 391

86.23.4 virtual void

This is the flux function for a given physical problem.

86.23.1

const Diff* **elemset**

The elemset associated with the flux function

86.23.2

DiffFF (const Diff* elemset_=NULL)

Constructor from the elemset

86.23.3

int **ask** (const char* jobinfo, int &skip_elemset)

This flags whether a given task is processed or not

86.23.4

virtual void **start_chunk** ()

This is called before any other in a loop and may help in optimization

86.23.5

virtual void **element_hook** (ElementIterator &element)

This is called before entering the Gauss points loop and may help in optimization.

Parameters: `element` (input) an iterator on the elemlist.

86.23.6

virtual void **gp_hook** (int ipg, const FastMat2 &U, const FastMat2 &grad_U)

This is called before each Gauss point.

Parameters: `ipg` (input) the Gauss point number (may be used to report errors)

86.23.7

virtual void
compute_flux (const FastMat2 &U, const FastMat2 &grad_U, FastMat2 &fluxd, FastMat2 &G, FastMat2 &H, FastMat2 &grad_H)

Computes fluxes, upwind parameters etc. fixme:= more doc here ...

86.23.8

virtual void
comp_grad_N_D_grad_N (FastMat2 &grad_N_D_grad_N, FastMat2 & dshapex, double w)

Computes the product `(grad_N_D_grad_N)_(p,mu,q,nu) = D_(i,j,mu,nu) (grad_N)_(i,p) (grad_N)_(j,q)`

Parameters:

`grad_N_D_grad_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`grad_N`	(input) size `nel` x `ndof`

86.23.9

virtual void **enthalpy** (FastMat2 &H, FastMat2 &U)

Computes the enthalpy vector from the state vector

Parameters:

`H`	(output) the enthalpy content vector
`U`	(input) the state vector

86.23.10

virtual void **comp_N_Cp_N** (FastMat2 &N_Cp_N, FastMat2 &N, double w)

Computes the product `(W_Cp_N)_(p,mu,q,nu) = W_p N_q Cp_(mu,nu)`

Parameters:

`W_Cp_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`N`	(input) interpolation function, size `nel`
`w`	(input) scalar weight

86.23.11

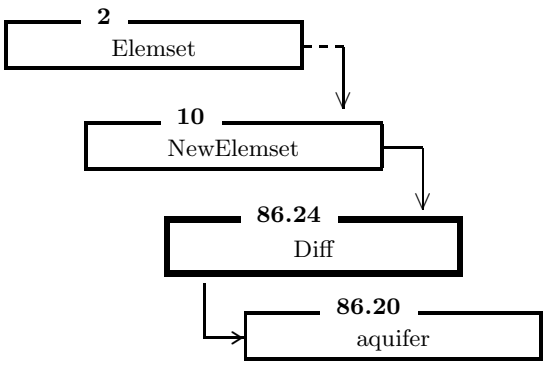
virtual **~DiffFF** ()

Destructor

86.24

class **Diff** : public NewElemset

Inheritance



Public Members

Private Members

Generic elemset for advective diffusive problems. Several physical problems may be solved by defining the corresponding flux function object.

86.24.3

Diff (DiffFF* diff_ff_=NULL)

Contructor from the pointer to the fux function

86.24.4

~**Diff** ()

Destructor. Destroys the flux function object.

86.24.5

NewAssembleFunction **new_assemble**

The assemble function for the elemset.

86.24.6

ASK_FUNCTION

The ask function for the elemset.

86.24.7

double **time** () const

Returns the actual time.

Return Value: `actual` time

86.24.1

DiffFF* **diff_ff**

A pointer to the flux function. Different physics are obtained by redefining the flux function

86.24.2

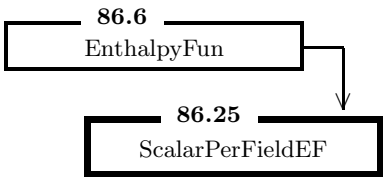
double **time_m**

The actual time

86.25

class **ScalarPerFieldEF** : public EnthalpyFun

Inheritance



Public Members

Private Members

Linear Constant Cp, the same for all fields

86.25.3

void **init** (int ndim, int ndof, int nel)

initializes the object

86.25.4

void **update** (const double* ejac)

sets the Cp coefficients for each field

86.25.5

void **enthalpy** (FastMat2 &H)

Scales each U value by the corresponding C_p coefficient

86.25.6

void
comp_W_Cp_N (FastMat2 &W_Cp_N, const FastMat2 &W, const FastMat2 &N, double w)

Efficient implementation

86.25.7

void **comp_P_Cp** (FastMat2 &P_Cp, const FastMat2 &P_supg)

Efficient implementation

86.25.1

FastMat2 **Cp**

Stores a cp for each field

86.25.2

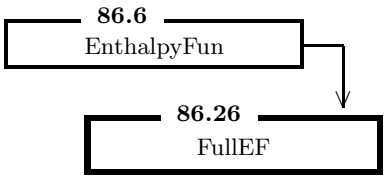
int **ndof**

the number of fields

86.26

class **FullEF** : public EnthalpyFun

Inheritance



Public Members

A general Cp matrix. See base class documentation.

86.26.1

void **init** (int ndim, int ndof, int nel)

Initializes

86.26.2

void **update** (const double* ejac)

Full implementation

86.26.3

void **enthalpy** (FastMat2 &H)

Full implementation

86.26.4

void
comp_W_Cp_N (FastMat2 &W_Cp_N, const FastMat2 &W, const FastMat2 &N, double w)

Full implementation

86.26.5

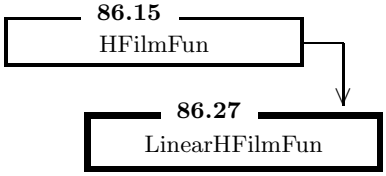
void **comp_P_Cp** (FastMat2 &P_Cp, const FastMat2 &P_supg)

Full implementation

86.27

class **LinearHFilmFun** : public HFilmFun

Inheritance

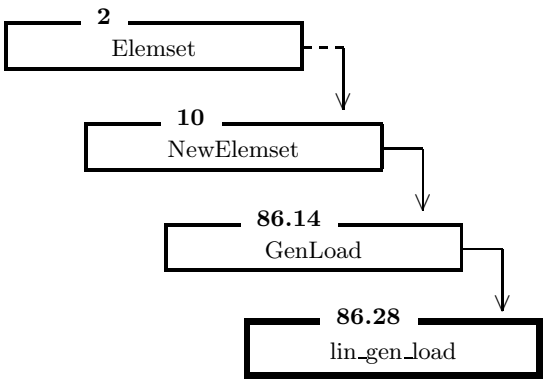


Generic surface flux function (film function) element

86.28

class **lin_gen_load** : public GenLoad

Inheritance

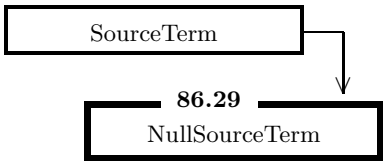


Linear surface flux element

86.29

class **NullSourceTerm** : public SourceTerm

Inheritance

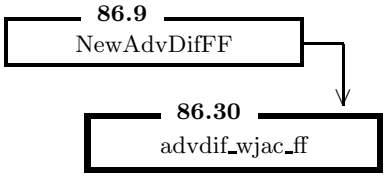


Null source term

86.30

class **advdif_wjac_ff** : public NewAdvDifFF

Inheritance



Public Members

In this class we have only to define the advective, diffusive and reactive jacobians, and source term

86.30.1

AJac* **a_jac**

The advective jacobian

86.30.2

DJac* **d_jac**

The diffusive jacobian

86.30.3

CJac* **c_jac**

the reactive jacobian

86.30.4

SourceTerm* **source_term**

The source term

86.30.5

advdif_wjac_ff (NewElemset* elemset_, DJac* d=NULL, AJac* a_jac_=NULL, CJac* c=NULL, SourceTerm* st=NULL)

Contructor from the objects

86.30.6

void **comp_A_jac_n** (FastMat2 &A_jac_n, FastMat2 &normal)

Compute $!A\hat{\mathbf{n}} = \sum_j A_j n_j$. Often, $\hat{\mathbf{n}}$ is a normal vector, but you can't assume that it is of unit length.

Parameters:

`A_jac_n`	(output) The contraction of the advectivejacobian with the `nornal` vector.
`normal`	(input) The vector onto which to make theprojection.

86.30.7

void **comp_A_grad_N** (FastMat2 & A_grad_N, FastMat2 & grad_N)

Compute $\sum_j A_j \partial_j N$ where N are the interpolation functions.

Parameters:

`A_grad_N`	(output) The contraction.
`grad_N`	(input) the gradient of the interpolation function.

86.30.8

void
comp_grad_N_D_grad_N (FastMat2 &grad_N_D_grad_N, FastMat2 &grad_N, double w)

Compute $\sum_{ij} wD_{ij}\partial_i N \partial_j N$. w is a scalar weight, normally the weight of the Gauss point times the determinant of the transformation.

Parameters:

`grad_N_D_grad_N`	(output) the result.
`grad_N`	(input) the gradient of the interpolationfunctions
`w`	(input) the scalar weight

86.30.9

void **comp_N_N_C** (FastMat2 &N_N_C, FastMat2 &N, double w)

Comput $\sum_{ij} C_{ij} N_i N_j$.

Parameters:

`N_N_C`	(output) the result
`N`	(input) The interpolation function
`w`	(input) the scalar weight

86.30.10

void
comp_N_P_C (FastMat2 &N_P_C, FastMat2 &P_supg, FastMat2 &N, double w)

Compute $\sum_{ij} C_{ij} N_i P_j$

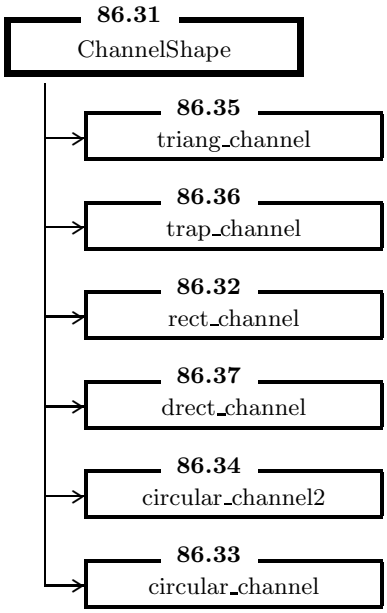
Parameters:

`N_P_C`	(output) the result
`N`	(input) The interpolation function
`w`	(input) the scalar weight

86.31

class **ChannelShape**

Inheritance



Public Members

Defines the shape of the channel

86.31.1

static ChannelShape* **factory** (const NewElemset* e)

All objects should be crated with this

86.31.2

virtual void **init** ()

Initialize properties (perhaps) from the elemset table

86.31.3

virtual void **element_hook** (ElementIterator element)

Read local element properties

86.31.4

virtual void **geometry** (double h, double &A, double &w, double &P)

For a given water depth (with respect to the bottom of the channel) give the fluid area, cross sectional water-line and wet channel perimeter.

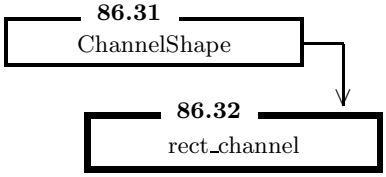
Parameters:

`h`	(input) the water depth
`A`	(ouput) the fluid cross sectional area
`w`	(ouput) the fluid cross sectional water line
`P`	(ouput) the fluid cross sectional perimeter

86.32

class **rect_channel** : public ChannelShape

Inheritance



Public Members

86.32.1 void

Rectangular shaped channel

86.32.1

void **init** ()

Initializes the object

86.32.2

void **element_hook** (ElementIterator element)

Read local element properties

86.32.3

void **geometry** (double h, double &area, double &wl_width, double &perimeter)

For a given water depth (with respect to the bottom of the channel) give the fluid area, cross sectional water-line and wet channel perimeter.

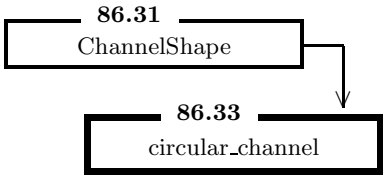
Parameters:

`h`	(input) the water depth
`area`	(ouput) the fluid cross sectional area
`wl_width`	(ouput) the fluid cross sectional water line
`perimeter`	(ouput) the fluid cross sectional perimeter

86.33

class **circular_channel** : public ChannelShape

Inheritance



Public Members

Circular shaped channel

86.33.1

void **init** ()

Initializes the object

86.33.2

void **element_hook** (ElementIterator element)

Read local element properties

86.33.3

void **geometry** (double h, double &area, double &wl_width, double &perimeter)

For a given water depth (with respect to the bottom of the channel) give the fluid area, cross sectional water-line and wet channel perimeter.

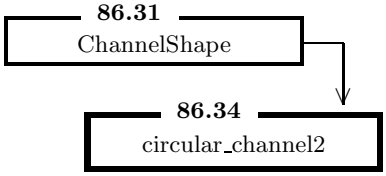
Parameters:

`h`	(input) the water depth
`area`	(ouput) the fluid cross sectional area
`wl_width`	(ouput) the fluid cross sectional water line
`perimeter`	(ouput) the fluid cross sectional perimeter

86.34

class **circular_channel2** : public ChannelShape

Inheritance



Public Members

Circular shaped channel 2

86.34.1

void **init** ()

Initializes the object

86.34.2

void **element_hook** (ElementIterator element)

Read local element properties

86.34.3

void **geometry** (double h, double &area, double &wl_width, double &perimeter)

For a given water depth (with respect to the bottom of the channel) give the fluid area, cross sectional water-line and wet channel perimeter.

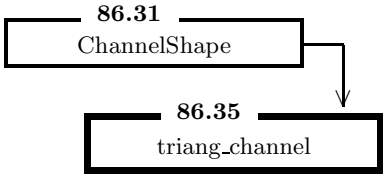
Parameters:

`h`	(input) the water depth
`area`	(ouput) the fluid cross sectional area
`wl_width`	(ouput) the fluid cross sectional water line
`perimeter`	(ouput) the fluid cross sectional perimeter

86.35

class **triang_channel** : public ChannelShape

Inheritance



Public Members

86.35.1 void
init ()
Initializes the object 409

86.35.2 void
element_hook (ElementIterator element)
Read local element properties 409

Triangular shaped channel

86.35.1

void **init** ()

Initializes the object

86.35.2

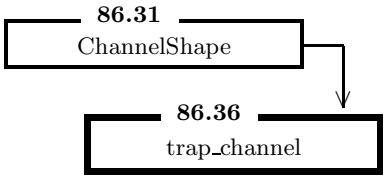
void **element_hook** (ElementIterator element)

Read local element properties

86.36

class **trap_channel** : public ChannelShape

Inheritance



Public Members

Trapezoidal shaped channel

86.36.1

void **init** ()

Initializes the object

86.36.2

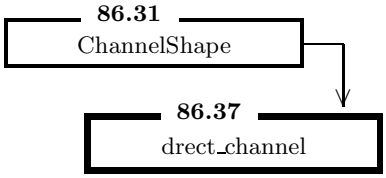
void **element_hook** (ElementIterator element)

Read local element properties

86.37

class **drect_channel** : public ChannelShape

Inheritance



Public Members

Rectangular shaped channel

86.37.1

void **init** ()

Initializes the object

86.37.2

void **element_hook** (ElementIterator element)

Read local element properties

86.37.3

void **geometry** (double h, double &area, double &wl_width, double &perimeter)

For a given water depth (with respect to the bottom of the channel) give the fluid area, cross sectional water-line and wet channel perimeter.

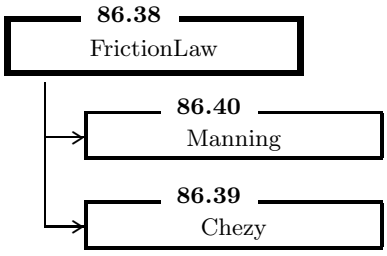
Parameters:

`h`	(input) the water depth
`area`	(ouput) the fluid cross sectional area
`wl_width`	(ouput) the fluid cross sectional water line
`perimeter`	(ouput) the fluid cross sectional perimeter

86.38

class **FrictionLaw**

Inheritance



Public Members

Abstract class representing all friction laws

86.38.1

FrictionLaw (const NewElemset* e)

fixme:= This should be private

86.38.2

static FrictionLaw* **factory** (const NewElemset* e)

All objects should be crated with this

86.38.3

virtual void **init** ()

Initialize properties (perhaps) from the elemset table

86.38.4

virtual void **element_hook** (ElementIterator element)

Read local element properties

86.38.5

virtual void
flow (double area, double perimeter, double S, double &Q, double &C) const

Should return the volumetric flow 'Q' for a given area 'A' and the derivative 'dQ/dA'.

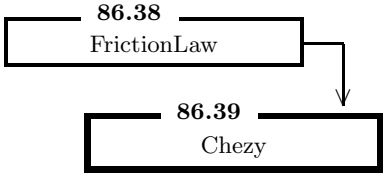
Parameters:

`area`	(input) tranversal area
`perimeter`	(input) wetted perimeter
`S`	(input) bottom slope
`Q`	(output) volumetric flow
`C`	(output) derivative of volumetric flow w.r.t. A,i.e. `C = dQ/dQ`

86.39

class **Chezy** : public FrictionLaw

Inheritance



Public Members

Private Members

This implements the Chezy friction law

86.39.3

void **element_hook** (ElementIterator element)

Read local element properties

86.39.4

void **flow** (double area, double perimeter, double S, double &Q, double &C) const

Should return the volumetric flow 'Q' for a given area 'A' and the derivative 'dQ/dA'.

Parameters:

`area`	(input) tranversal area
`perimeter`	(input) wetted perimeter
`S`	(input) bottom slope
`Q`	(output) volumetric flow
`C`	(output) derivative of volumetric flow w.r.t. A,i.e. `C = dQ/dQ`

86.39.1

Property **Ch_prop**

Chezy friction coefficient property

86.39.2

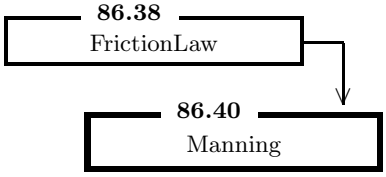
double **Ch**

Chezy friction coefficient value

86.40

class **Manning** : public FrictionLaw

Inheritance



Public Members

Private Members

This implements the Manning friction law

86.40.4

void **element_hook** (ElementIterator element)

Read local element properties

86.40.5

void **flow** (double area, double perimeter, double S, double &Q, double &C) const

Should return the volumetric flow 'Q' for a given area 'A' and the derivative 'dQ/dA'.

Parameters:

`area`	(input) tranversal area
`perimeter`	(input) wetted perimeter
`S`	(input) bottom slope
`Q`	(output) volumetric flow
`C`	(output) derivative of volumetric flow w.r.t. A,i.e. `C = dQ/dQ`

86.40.1

Property **roughness_prop**

Manning friction coefficient property

86.40.2

double **roughness**

Manning friction coefficient value

86.40.3

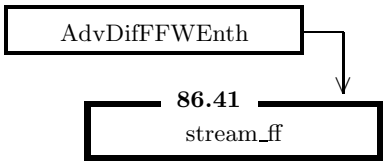
double **a_bar**

Conversion factor

86.41

class **stream_ff** : public AdvDifFFWEnth

Inheritance



Public Members

Private Members

The flux function for flow in a channel with arbitrary shape and using the Kinematic Wave Model

86.41.6

stream_ff (const NewAdvDif* e)

Properties related to friction

86.41.7

void **start_chunk** (int &ret_options)

This is called before any other in a loop and may help in optimization

Parameters:

`ret_options`	(input/output) this is used by the fluxfunction writer for returning some options. Currently the onlyoption used is `SCALAR_TAU`. This options tells the elemsetwhether the flux function returns a scalar or matrix`tau_supg`.

86.41.8

void **element_hook** (ElementIterator &element)

This is called before entering the Gauss points loop and may help in optimization.

Parameters: `element` (input) an iterator on the elemlist.

86.41.9

void **set_state** (const FastMat2 &U, const FastMat2 &grad_U)

Basically stores 'U(1)' in the water depth 'u' and computes geometric parameters of the channel

Parameters:
`U` (input) the state of the fluid
`grad_U` (input) gradient of the state of the fluid

86.41.10

void **set_state** (const FastMat2 &U)

Basically stores 'U(1)' in the water depth 'u' and computes geometric parameters of the channel

Parameters: `U` (input) the state of the fluid

86.41.11

Advective jacobians related

Names

86.41.11.1

void **comp_A_grad_N** (FastMat2 & A_grad_N, FastMat2 & grad_N)

Computes the product `(A_grad_N)_(p,mu,nu) = A_(i,mu,nu) (grad_N)_(i,p)`

Parameters:

`A_grad_N`	(output, size `nel` x `nd` x `nd`)
`grad_N`	(input, size `nel` x `ndof`)

86.41.11.2

void **comp_A_jac_n** (FastMat2 &A_jac_n, FastMat2 &normal)

Computes the product `(A_jac_n)_(mu,nu) = A_(i,mu,nu) normal_i`

Parameters:

`A_jac_n`	(output, size `ndof` x `ndof`)
`normal`	(input, size `ndim`)

86.41.11.3

void **compute_flux** (COMPUTE_FLUX_ARGS)

Computes fluxes, upwind parameters etc. fixme:= include more doc here ...

86.41.12

Diffusive jacobians related

Names

86.41.12.1

void
comp_grad_N_D_grad_N (FastMat2 &grad_N_D_grad_N, FastMat2 & dshapex, double w)

Computes the product `(grad_N_D_grad_N)_(p,mu,q,nu) = D_(i,j,mu,nu) (grad_N)_(i,p) (grad_N)_(j,q)`

Parameters:

`grad_N_D_grad_N`	(output) size `nel` x `ndof` x `nel` x `ndof`
`grad_N`	(input) size `nel` x `ndof`

86.41.13

Reactive jacobians related

Names

86.41.13.1

void **comp_N_N_C** (FastMat2 &N_N_C, FastMat2 &N, double w)

Computes the product `(N_N_C)_(p,mu,q,nu) = w C_(mu,nu) N_p N_q`

Parameters:

`N_N_C`	(output, size `nel` x `ndof` x `nel` x `ndof`
`N`	(input) FEM interpolation function size `nel`
`w`	(input) a scalar coefficient

86.41.13.2

void
comp_N_P_C (FastMat2 &N_P_C, FastMat2 &P_supg, FastMat2 &N, double w)

Computes the product `(N_P_C)_(mu,q,nu) = w (P_supg)_(mu,lambda) C_(lambda,nu) N_q`

Parameters:

`N_P_C`	(output) size `ndof` x `nel` x `ndof`
`P_supg`	(input) SUPG perturbation function size `ndof` x `ndof`
`N`	(input) FEM interpolation function size `nel`
`w`	(input) a scalar coefficient

86.41.13.3

int **dim** () const

This stream elemset is essentially 1D.

Return Value:

`the`	dimension of this element that is 1

86.41.1

double **h**

The local depth of the fluid

86.41.2

The slope of the channel

The slope of the channel

86.41.3

double **C**

The local wave velocity C:= dQ/dA

86.41.4

FrictionLaw* **friction_law**

Type of friction law used

86.41.5

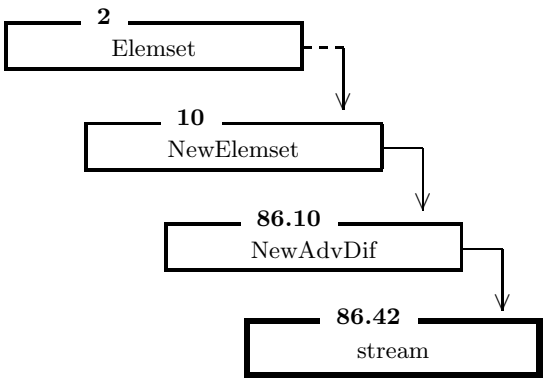
ChannelShape* **channel**

Pointer to the channel shape object

86.42

class **stream** : public NewAdvDif

Inheritance

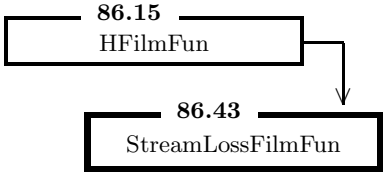


The 'stream' (river or channel) element

86.43

class **StreamLossFilmFun** : public HFilmFun

Inheritance



Private Members

Losses from a stream to the aquifer. Here the 'in' side of the element represents the stream and the 'out' side the aquifer.

86.43.1

Property **Rf_prop**

The property corresponding to the resistance of the stream bottom surface

86.43.2

double **k**

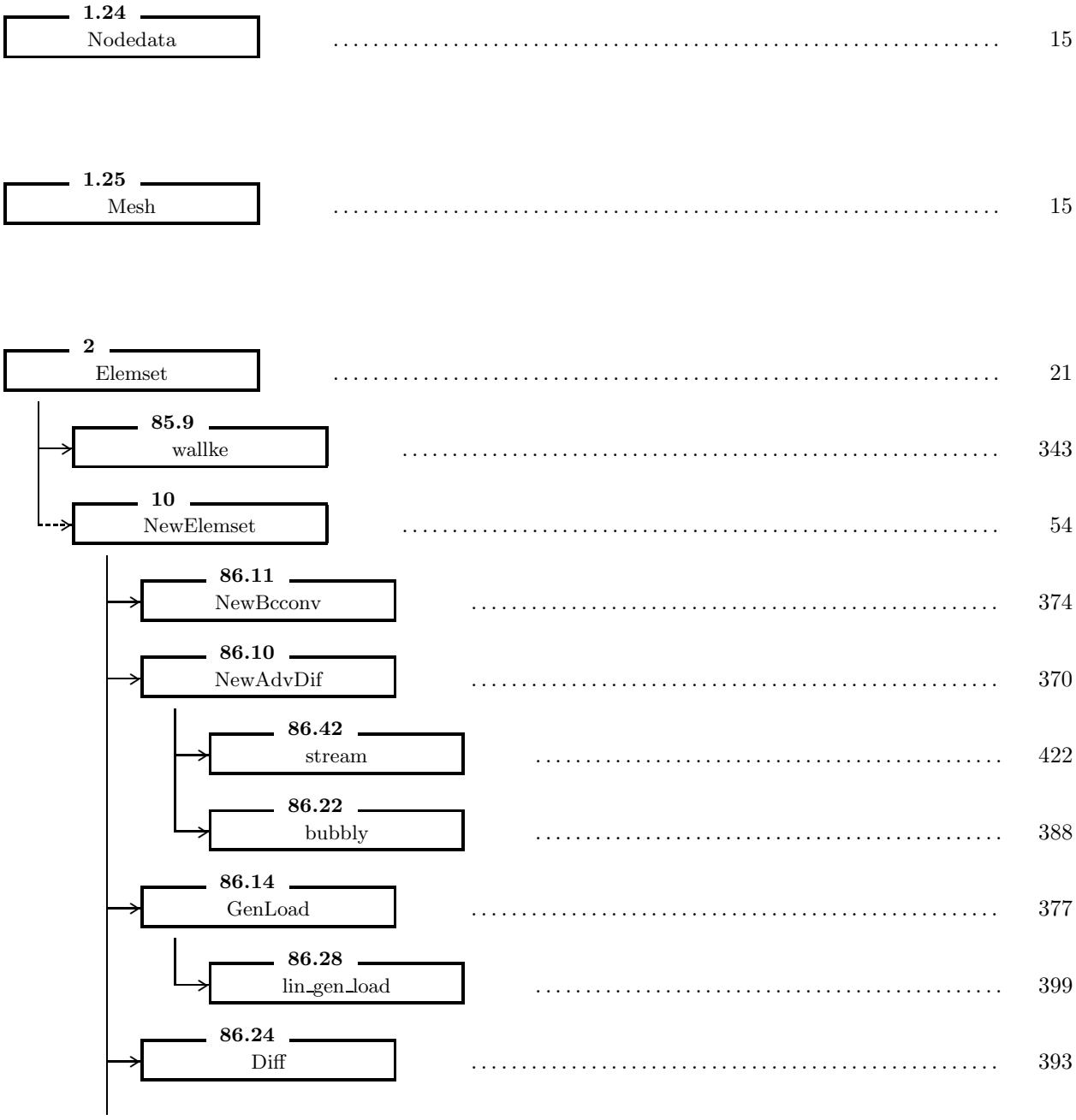
The inverse of resistance of the stream bottom surface 'k=1/Rf'

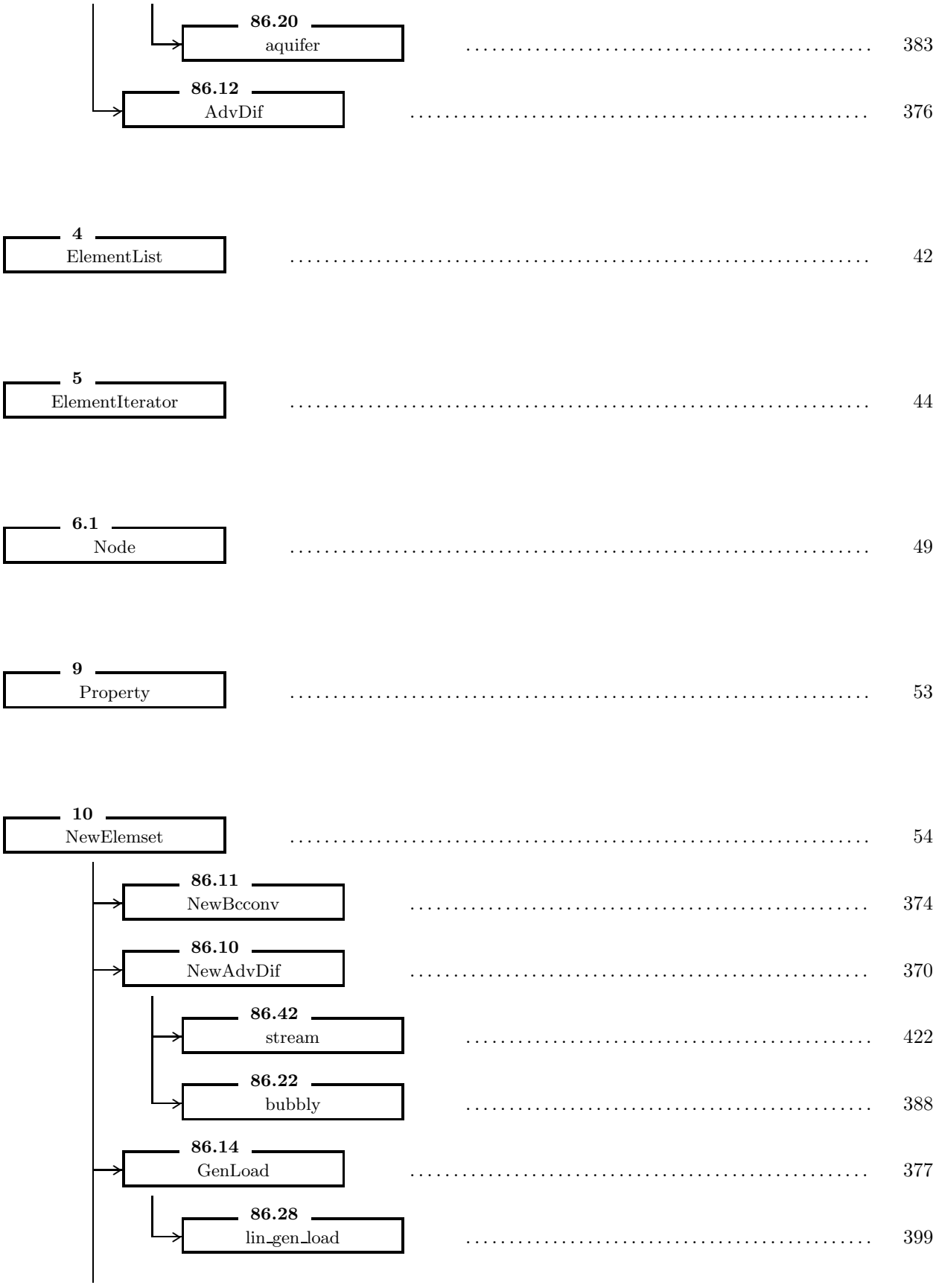
86.43.3

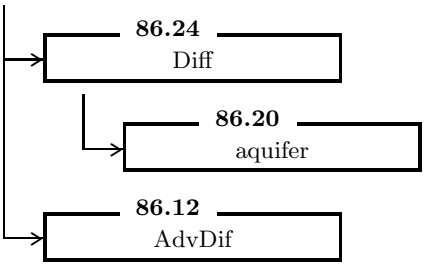
int **impermeable**

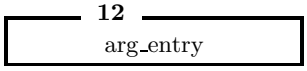
If set to 1 then Rf = infty

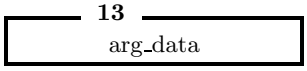
Class Graph

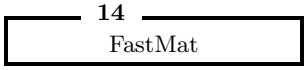

1.24
Nodedata
15
1.25
Mesh
15
2
Elemset
21
85.9
wallke
343
10
NewElemset
54
86.11
NewBcconv
374
86.10
NewAdvDif
370
86.42
stream
422
86.22
bubbly
388
86.14
GenLoad
377
86.28
lin_gen_load
399
86.24
Diff
393

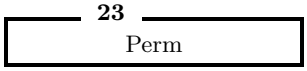

86.20
aquifer
383
86.12
AdvDif
376
4
ElementList
42
5
ElementIterator
44
6.1
Node
49
9
Property
53
10
NewElemset
54
86.11
NewBcconv
374
86.10
NewAdvDif
370
86.42
stream
422
86.22
bubbly
388
86.14
GenLoad
377
86.28
lin_gen_load
399

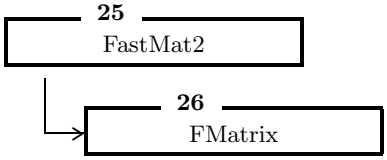


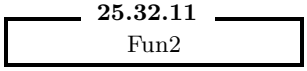


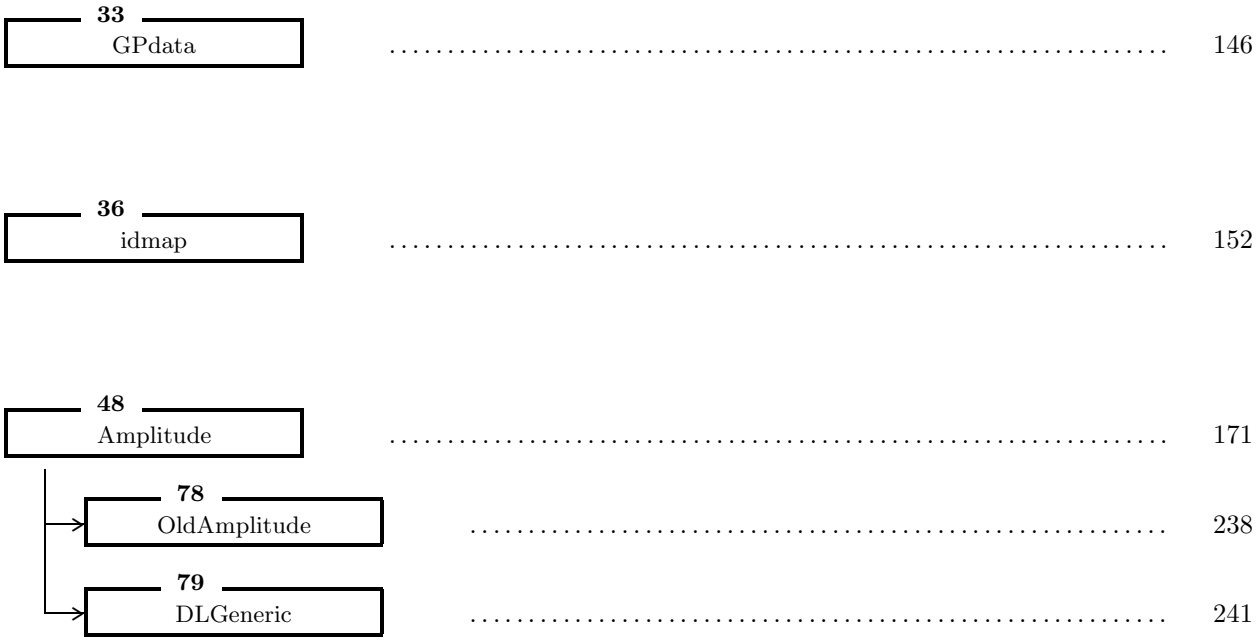


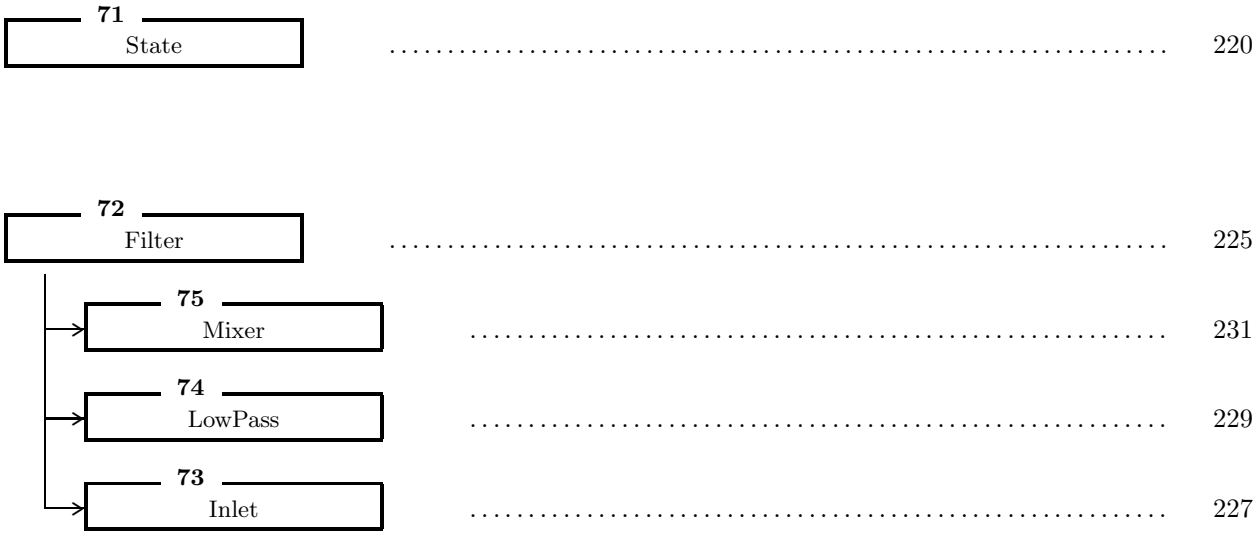


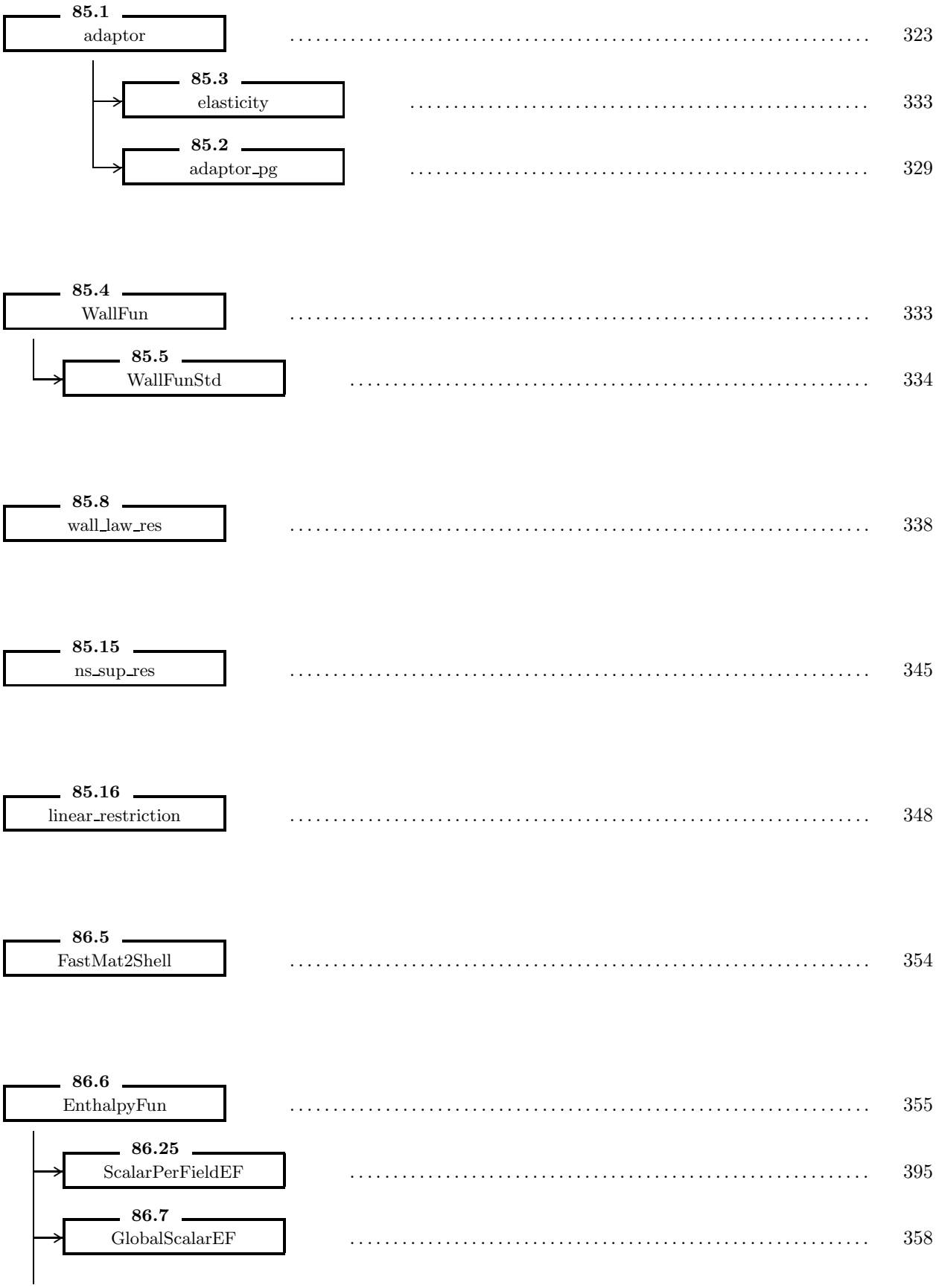


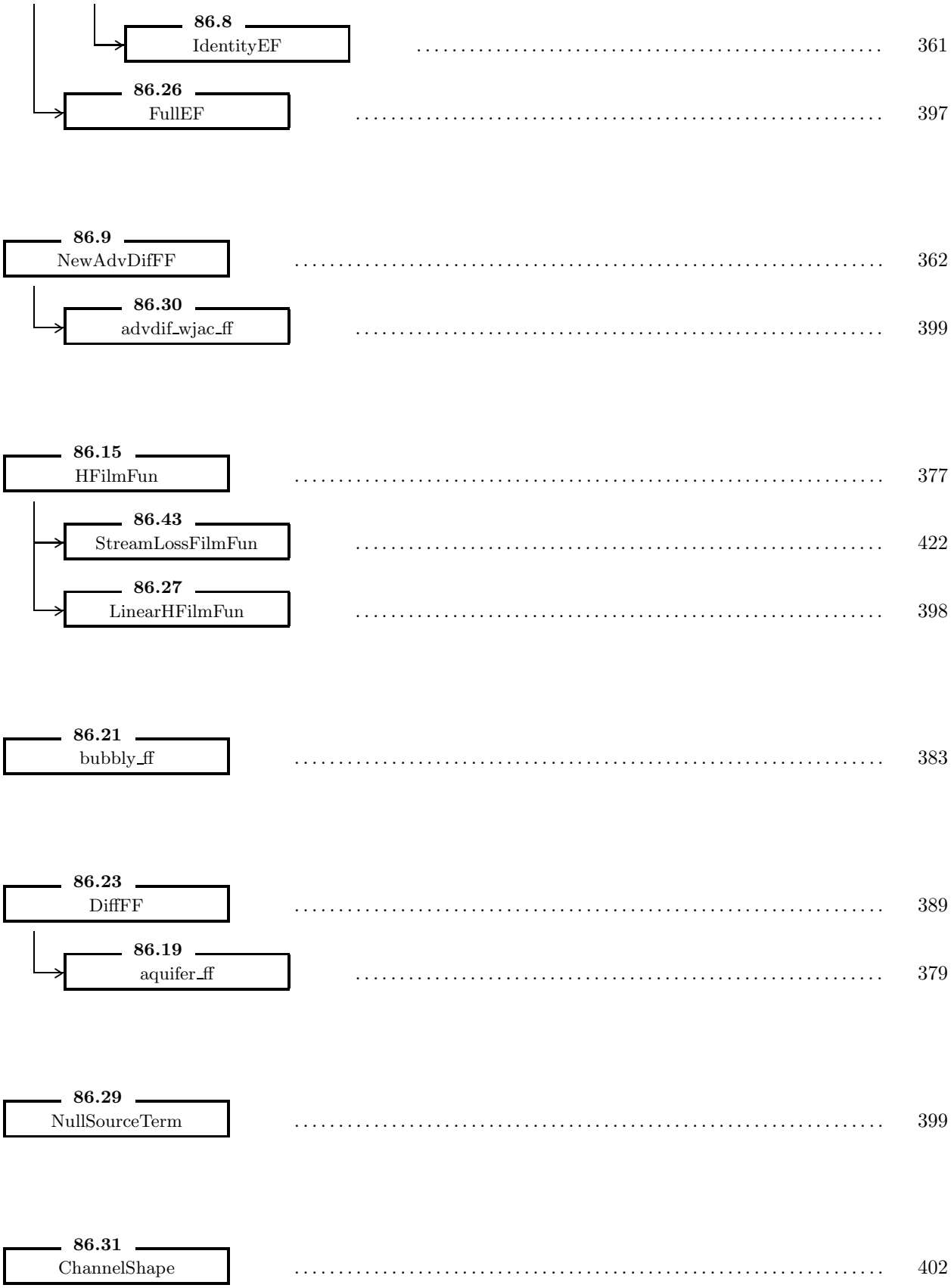





33
GPdata
146
36
idmap
152
48
Amplitude
171
78
OldAmplitude
238
79
DLGeneric
241
50
Constraint
174
51
fixation_entry
175
52
Dofmap
176
56
TextHashTableVal
194
58
TextHashTable
196


71
State
220
72
Filter
225
75
Mixer
231
74
LowPass
229
73
Inlet
227
76
LPFilterGroup
234
81
Secant
249
85.6
WallFunSecant
336
82
Debug
252
83.1
DistMap
253
83.2
Row
257




86.8
IdentityEF
361
86.26
FullEF
397
86.9
NewAdvDifFF
362
86.30
advdif_wjac_ff
399
86.15
HFilmFun
377
86.43
StreamLossFilmFun
422
86.27
LinearHFilmFun
398
86.21
bubbly_ff
383
86.23
DiffFF
389
86.19
aquifer_ff
379
86.29
NullSourceTerm
399
86.31
ChannelShape
402