sparselizard

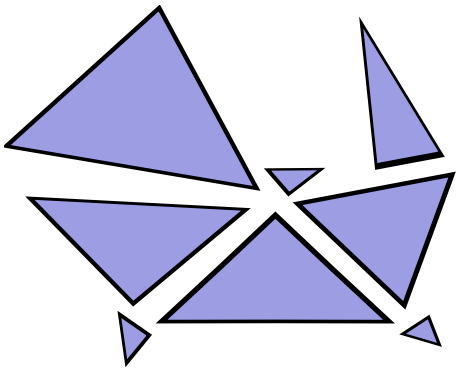
the user friendly

finite element

c++ library

Alexandre Halbach

December 1, 2017



Contents

1 What is sparselizard

Sparselizard (Copyright (C) 2017- Alexandre Halbach and Christophe Geuzaine, University of Liege, Belgium) is an open source C++ finite element library provided under the terms of the GNU General Public License (GPL), version 2 or later.
The library is meant to be user friendly while decently fast and parallelised. It can handle a rather general set of problems in 1D, 2D and 3D such as mechanical, acoustic, thermal, electric and electromagnetic problems (provided in form of a weak formulation as detailed in `https://en.wikipedia.org/wiki/Weak_formulation`). Multiphysics problems, nonlinear problems or nonlinear multiphysics problems can be simulated as well. The problems can be readily solved in time with a time-stepping resolution or with the natively supported multiharmonic resolution method. In the latter case the steady-state solution of a time-periodic problem can be obtained in a single step, for linear as well as for general nonlinear problems. The library comes with hierarchical high order shape functions so that high order interpolations can be used with an interpolation order adapted to every unknown field and geometrical region.
For now sparselizard has been successfully tested on Linux and Mac (but not on Windows). Working examples can be found in the 'examples' folder in the project.

The widely-used open-source GMSH meshing software (`www.gmsh.info`) is recommended to mesh the geometry and generate the .msh file required in the finite element simulation. The result files output by sparselizard are in .pos format supported by GMSH.

We hope you appreciate this library and wish you all the best with it!

2 How to install sparselizard

Sparselizard can be obtained at the following adress:

`https://gitlab.onelab.info/halbux/sparselizard.git`

The files can be either downloaded as an archive or by running in a terminal:

```
git clone https://gitlab.onelab.info/halbux/sparselizard.git
```

Before compiling it, the external libraries listed below must be installed. For that make sure you have the gcc, g++ and the **standard** gfortran compilers. On Ubuntu linux these can be installed with:

```
sudo apt-get install gfortran
sudo apt-get install gcc
sudo apt-get install g++
```

Once the compilers are available the required external libraries must be installed. This can be done easily by running in the provided order all bash scripts in folder 'install_external_libs'. Each script installs with the right options the corresponding external library in the 'SLlibs' folder in the home directory. In case this does not work for a given library, please install it yourself with the configuration options detailed in the bash script. In case you do not want to use the standard installation directory or want to use an already available library do not forget to change the library path accordingly in the makefile and in 'run_sparselizard.sh'.

The external libraries used are the following:

- OpenBLAS: is used for optimised and multithreaded operations on dense matrices and vectors. More information at `www.openblas.net`.
- FFTW: is used for fast Fourier transforms. More information at `www.fftw.org`.
- PETSc: in combination with MUMPS is mainly used to solve the large sparse algebraic problems. More information at `www.mcs.anl.gov/petsc` and `mumps.enseeiht.fr`.
- SLEPc: in combination with PETSc is used to solve eigenvalue problems for large sparse algebraic problems. More information at `slepc.upv.es`.

Once all external libraries are successfully installed sparselizard can be compiled by simply running 'make' or 'make -j4' if you have 4 computing cores.

3 How to use and run sparselizard

One way of using sparselizard is with the following steps:

1. Edit the 'sparselizard' function in the 'main.cpp' file for your simulation.
2. Run make in the terminal. This should be much quicker this time since only the 'main.cpp' file has to be recompiled.
3. Run your simulation by entering './run_sparselizard.sh' in the terminal.

As an example let us simulate the static deflection of a mechanical disk with some volume force applied to it. This requires to have the original 'main.cpp' and 'circle.geo' files that are available after having downloaded the sparselizard project. This also requires the binary of the open source GMSH meshing software that can be downloaded at `www.gmsh.info`.
Copy the binary to the sparselizard folder then mesh the 'circle.geo' geometry by running './gmsh circle.geo -3' (3 because it is a 3D problem) or with './gmsh circle.geo' to mesh graphically. This creates a 'circle.msh' file which contains the mesh. Now run './run_sparselizard.sh' in the terminal. This runs the code in 'main.cpp' that has just been compiled.

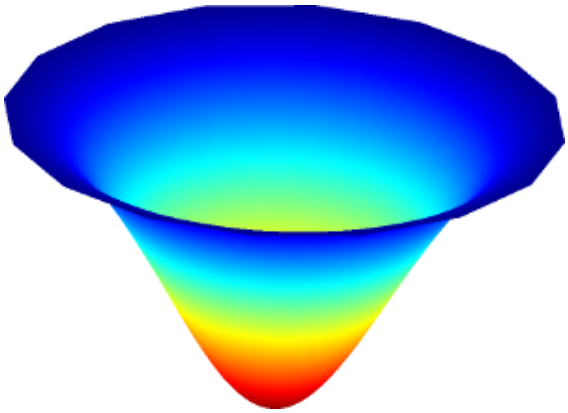


Figure 1: Exaggerated deflection of the 3D disk

The last step has created the 'u.pos' output file, which gives the exaggerated displacement of the top surface in the thin cylinder geometry when the sides are clamped and a volume force is applied downwards. Open it with './gmsh u.pos'. You don't see anything or it looks weird? Don't worry, this is just because the simulation was performed using very few hexahedra in the mesh but with an order 3 interpolation! To visualise high order interpolations in GMSH do this:

- Double click in the middle of the window then select 'All view options' at the bottom of the box that appeared go to the 'General' tab and tick the 'Adapt visualization grid' box.
- Set 'Maximum recursion level' to 3 and 'Target visualization error' to the smallest possible value then press enter. Now you have a finer solution!
- Since the solution is a mechanical displacement you might want to see the (exaggerated) deflection in 3D by double clicking in the middle of the window then selecting 'View vector display' >> 'Displacement' with factor 1.
- In case you see strange lighting effects double click in the middle of the window then select 'All view options' at the bottom of the box that appeared, go to the 'Color' tab and untick the 'Enable lighting' box.

Figure 1 is what you should see. Congratulations for your first simulation with the sparselizard library! In the 'examples' folder you will find more sparselizard examples.

4 Objects and functions available in the library

Much of sparselizard is written in c++ in an object-oriented way. It should thus be no surprise that most of the code you write consists in creating and managing objects. Below is the list of the main objects (as well as the few namespaces) that you can use in your simulations. More objects and functions are available but are not meant to be directly called by the user.

4.1 The *mesh* object (/src/mesh/mesh.h):

The mesh object holds the information of the finite element mesh of the geometry.

```
mesh(std::string meshname)
```

```
mesh mymesh("circle.msh");
```

This creates the mesh object based on the 'circle.msh' mesh file created by GMSH.

```
void load(std::string meshname)
```

```
mesh mymesh;
mymesh.load("circle.msh");
```

This loads the 'circle.msh' mesh file created by GMSH.

```
void write(std::string meshname)
```

```
mesh mymesh;
mymesh.write("circle.msh");
```

This writes the mesh in object 'mymesh' to file 'circle.msh'.

```
void shift(double x, double y, double z)
```

```
mesh mymesh("circle.msh");
mymesh.shift(1.0, 2.0, 3.0);
```

This translates the mesh in object 'mymesh' by 1, 2 and 3 respectively in the x, y and z direction.

```
void rotate(double ax, double ay, double az)
```

```
mesh mymesh("circle.msh");
mymesh.rotate(20, 60, 90);
```

This rotates the mesh in object 'mymesh' by 20, 60 and 90 degrees respectively around the x, y and z axis.

```
int getmeshdimension(void)
```

```
mesh mymesh("circle.msh");
int dim = mymesh.getmeshdimension();
```

This returns the dimension of the highest dimension element in the mesh in object 'mymesh'.

4.2 The *field* object (/src/field/field.h):

The field object holds the information of the finite element fields. The field object itself only holds a pointer to a 'rawfield' object.

```
field(std::string fieldtypename)
```

```
field v("h1");
```

This creates a field v with nodal (i.e. "h1") shape functions.

The full list of shape functions that are available is:

- Nodal shape functions "h1" e.g. for the electrostatic potential or acoustic pressure field.
- Two-components nodal shape functions "h1xy" e.g. for 2D mechanical displacements.
- Three-components nodal shape functions "h1xyz" e.g. for 3D mechanical displacements.
- Nedelec's edge shape functions "hcurl" e.g. for the electric and magnetic fields in the E-formulation of electromagnetic wave propagation.
- "q6" ("q6xy" or "q6xyz") shape functions (only for rectangular elements) identical to order one "h1" ("h1xy" or "h1xyz") shape functions except that there is an extra bubble mode for more accurate mechanical bending computations.
- "h11" ("h11xy" or "h11xyz") shape functions (only for hexahedral elements) identical to order one "h1" ("h1xy" or "h1xyz") shape functions except that there are three extra bubble modes for more accurate mechanical bending computations.

```
field(std::string fieldtypename, const std::vector<int> harmonicnumbers)
```

```
field v("h1", {1,4,5,6});
field v4 = v.getharmonic(4);
```

Consider the infinite Fourier series of a field periodic in time:
$v(x,t) = V_1 + V_2\sin(2\pi f_o t) + V_3\cos(2\pi f_o t) + V_4\sin(2\cdot 2\pi f_o t) + V_5\cos(2\cdot 2\pi f_o t) + V_6\sin(3\cdot 2\pi f_o t) + ...$
where t is the time variable, x the space variable and f_0 the fundamental frequency of the periodic field. The V_i coeffcients only depend on the space variable, not on the time variable which has now moved to the sines and cosines.
In the example above field v is a *multiharmonic* "h1" type field that includes 4 monoharmonic fields: the V_1, V_4, V_5 and V_6 fields in the truncated Fourier series above. All other harmonics in the infinite Fourier series are supposed equal zero so that field v can be rewritten as:
$v(x,t) = V_1 + V_4\sin(2\cdot 2\pi f_o t) + V_5\cos(2\cdot 2\pi f_o t) + V_6\sin(3\cdot 2\pi f_o t)$. This is the truncated multiharmonic representation of field v (which must be periodic in time).

The second line in the example gets from field v the 4th harmonic, which can now be used like any other field.

```
int countcomponents(void)
```

```
field E("hcurl");
int numcomp = E.countcomponents();
```

This returns the number of components of field E (3 here).

```
std::vector<int> getharmonics(void)
```

```
field v("h1", {1,4,5,6});
std::vector<int> myharms = v.getharmonics();
```

This returns the harmonics of field v ({1,4,5,6} here).

```
void printharmonics(void)
```

```
field v("h1", {1,4,5,6});
v.printharmonics();
```

Print a string showing the harmonics in the field.

```
void setname(std::string name)
```

```
field v("h1");
v.setname("v");
```

This gives a name to the field (usefull e.g. when printing expressions including fields).

```
void print(void)
```

```
field v("h1");
v.setname("v");
v.print();
```

Print the field name (“v” here).

```
void setorder(int physreg, int interpolorder)
```

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1");
v.setorder(vol, 3);
```

Sets interpolation order 3 on region number 1. The default interpolation order is 1.

```
void setvalue(int physreg, expression input, int extraintegrationdegree = 0)
```

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1");
v.setvalue(vol, 12 );
```

Sets the field value on region 1 to expression "12". An extra int argument (e.g. +2) can be used to increase (or decrease) the default integration order when computing the projection of the expression on field v. Increasing it can give a more accurate computation of the expression but might take longer. The default integration order is the v field order + 2.

```
void setvalue(int physreg)
```

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1");
v.setvalue(vol);
```

Sets the field value on region 1 to 0.

```
void setconstraint(int physreg, expression input, int extraintegrationdegree = 0)
```

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1"), w("h1");
v.setconstraint(vol, 12*w*w );
```

Forces the field value (i.e. Dirichlet condition) on region 1 to expression "12*w*w" (this gives 0 untill w is set to a non-zero value). An extra int argument (e.g. +2) can be used to increase (or decrease) the default integration order when computing the projection of the expression on field v. Increasing it can give a more accurate computation of the expression but might take longer. The default integration order is the v field order + 2.

```
void setconstraint(int physreg)
```

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1");
v.setconstraint(vol);
```

Forces the field value (i.e. Dirichlet condition) on region 1 to value 0.

```
void getdata(int physreg, vectorfieldselect myvec)
```

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1"), w("h1");
formulation projection;
projection += integral(vol, dof(v)*tf(v) - 2*tf(v) );
projection.generate();
vec sol = solve(projection.A(), projection.b());

w.getdata(vol, sol|v);
```

The last line transfers the data corresponding to field v in the solution vector 'sol' to field w on the region number 1.
This only works if v and w are of the same type (here they are both "h1" type). In case v has a higher interpolation order than w the higher order dofs are not transferred to w. In the opposite case the higher order dofs of w are zeroed.

```
void getdata(int physreg, vec myvec)
```

```
...
v.getdata(vol, sol);
```

This function does the same as the above except that data from field v in the 'sol' vector is transferred to field v, not to an other field.

```
field comp(int component)
```

```
field u("h1xyz");
field ux = u.comp(0);
field uy = u.comp(1);
field uz = u.comp(2);
```

This function gets the x, y or z component of a field with subfields.

```
field compx(void)
```

```
field u("h1xyz");
field ux = u.compx();
```

This function gets the x component of a field with multiple subfields.

```
field compy(void)
```

```
field u("h1xyz");
field uy = u.compy();
```

This function gets the y component of a field with multiple subfields.

`field compz(void)`

```
field u("h1xyz");
field uz = u.compz();
```

This function gets the z component of a field with multiple subfields.

`field harmonic(int harmonicnumber)`

```
field u("h1xyz", {1,2,3});
field u2 = u.harmonic(2);
```

This function gets a "h1xyz" type field that is the harmonic 2 of field u.

`field harmonic(const std::vector<int> harmonicnumbers)`

```
field u("h1xyz");
field u23 = u.harmonic({2,3});
```

This function gets a "h1xyz" type field that includes the harmonics 2 and 3 of field u.

`field sin(int freqindex)`

```
field u("h1xyz", {1,2,3,4,5});
field us = u.sin(2);
```

This function gets a "h1xyz" type field that is the sin harmonic at 2 times the fundamental frequency in field u, i.e. it is harmonic 4.

`field cos(int freqindex)`

```
field u("h1xyz", {1,2,3,4,5});
field uc = u.cos(0);
```

This function gets a "h1xyz" type field that is the cos harmonic at 0 times the fundamental frequency in field u, i.e. it is harmonic 1.

`integrate and write`: please refer to the expression object

`operators + - * /` : please refer to the expression object

4.3 The *expression* object (/src/expression/expression.h):

The expression object holds a mathematical expression made of $+ - */$ operators, fields, parameters, square operators, abs operators...

`expression(void)`

```
expression myexpression;
```

This creates an empty expression object.

`expression(field)`

```
field v("h1");
expression myexpression(v);
```

This creates an expression object from a field.

`expression(double)`

```
expression myexpression(12.0);
```

This creates an expression object from a double.

`expression(parameter&)`

```
mesh mymesh("circle.msh");
int vol = 1;
parameter E;
E|vol = 10;
expression myexpression(E);
```

This creates an expression object from a parameter.

`expression(int numrows, int numcols, std::vector<expression>)`

```
field v("h1");
expression myexpression(2,3,{12,v,v*(1-v),3,14-v,0});
```

This creates a 2×3 sized expression object (2 rows, 3 columns) and sets every entry to the corresponding expression in the expression vector. The expression vector is row-major. As an example entry (1,0) in the created expression is set to 3 and entry (1,2) to 0.

`int countrows(void)`

```
expression myexpression(2,1,{0,1});
int numrows = myexpression.countrows();
```

This counts the number of rows in an expression (here 2).

```
int countcolumns(void)
```

```
expression myexpression(2,1,{0,1});
int numcolumns = myexpression.countcolumns();
```

This counts the number of columns in an expression (here 1).

```
double integrate(int physreg, int integrationorder)
```

```
mesh mymesh("circle.geo");
int vol = 1;
expression myexpression = 12.0;
double integralvalue = myexpression.integrate(vol, 4);
```

This integrates the expression over the geometrical region 1. The integration is exact for up to 4th order polynomials.

```
double integrate(int physreg, expression meshdeform, int integrationorder)
```

```
mesh mymesh("circle.geo");
int vol = 1;
field u("h1xyz");
u.setvalue(vol, 12);
expression myexpression = 12.0;
double integralvalue = myexpression.integrate(vol, u, 4);
```

This integrates the expression over the geometrical region 1. The integration is exact for up to 4th order polynomials. It is performed on the geometry deformed by field u (possibly curved mesh).

```
void write(int physreg, int numfftharms, std::string filename, int lagrangeorder = 1)
```

```
mesh mymesh("circle.geo");
int vol = 1;
field u("h1xyz"), v("h1",{1,2,3});
(u*u).write(vol, 20, "uorder3.pos", 3);
abs(v).write(vol, 20, "uorder1.pos");
```

This writes any expression to a file, here with either an order 3 interpolation or, in the latter case with a default order 1 interpolation. The 20 means the expression is treated as multiharmonic, nonlinear in the time variable and an FFT is performed to get the 20 first harmonics. All harmonics whose magnitude are above a threshold are saved with the '_harm i' extension (except the time-constant harmonic).

```
void write(int physreg, int numfftharms, expression meshdeform, std::string filename,
int lagrangeorder = 1)
```

```
...
abs(v).write(vol, 20, u, "u.pos", 2);
```

Same as above but here the expression in evaluated and written on a mesh deformed by field u (possibly curved mesh).

```
void write(int physreg, std::string filename, int lagrangeorder = 1,
int numtimesteps = -1)
```

```
...
(1e8*u).write(vol, "uorder1.pos");
(1e8*u).write(vol, "uorder3.pos",3);
(1e8*u).write(vol, "uintime.pos", 2, 50);
```

Same as two above except that here no FFT is performed. In case the expression is nonlinear and multiharmonic the FFT is required and an error is thus thrown. If 'numtimesteps' is set to a positive value n then the (multiharmonic) expression is saved at n equidistant timesteps in the fundamental period and can then be visualised in time.

```
void write(int physreg, expression meshdeform, std::string filename,
int lagrangeorder = 1, int numtimesteps = -1)
```

```
...
(1e8*v).write(vol, u, "uintime.pos", 2, 50);
```

Same as above but here the expression in evaluated and written on a mesh deformed by field u (possibly curved mesh).

```
void reuseit(bool istobereused = true)
```

```
expression myexpression(12.0);
myexpression.reuseit();
```

In case you have an expression that appears multiple times e.g. in a formulation and require too much time to be computed you can 'reuse' that expression. With this the expression will only be computed once to assemble a formulation block and reused as long as it is impossible that its value has changed.

```
bool isscalar(void)
```

```
expression myexpression(12.0);
bool a = myexpression.isscalar();
```

True if the expression is a scalar (i.e. has a single row and column).

```
bool iszero(void)
```

```
expression myexpression(2,1,{0,0});
bool a = myexpression.iszero();
```

True if the expression is zero (here it is true).

```
void print(void)
```

```
expression myexpression(12.0);
myexpression.print();
```

Print the expression to the console.

```
expression getarrayentry(int row, int col)
```

```
expression myexpression(2,2,{1,2,3,4});
expression myentry = myexpression.getarrayentry(0,1);
```

This returns the entry at the requested row and column (here it returns expression 2).

```
operators + - * / :
```

```
mesh mymesh("circle.msh");
int vol = 1;
expression expr(12.0);
parameter E;
E|vol = 10;
double dbl = -3.2;
field v("h1");

expression plus = expr + E + dbl + v;
expression minus = -expr - E - dbl - v;
expression product = expr * E * dbl * v;
expression divided = expr / E / dbl / v;

expression mixed = expr * (-2.0 * v) - E / dbl;
```

The sum, difference, product and division between any two of 'expression', 'field', 'double' and 'parameter' is allowed.

4.4 The *formulation* object (/src/formulation/formulation.h):

The formulation object holds the terms of the weak formulation of the problem to simulate.

`formulation(void)`

```
formulation myformulation;
```

This creates an empty formulation object.

`void operator+=(integration integrationobject)`

```
mesh mymesh("circle.msh");
int vol = 1, sur = 2;
field v("h1"), vh("h1",{1,2,3}), u("h1xyz");
formulation projection;

// Basic version:
projection += integral(vol, dof(v)*tf(v) - 2*tf(v) );
projection += integral(vol, dof(v)*tf(v,sur) - 2*tf(v), +2 );
projection += integral(vol, dof(v,sur)*tf(v) - 2*tf(v), +2, 2 );
// Assemble on the mesh deformed by field u:
projection += integral(sur, u, dof(v)*tf(v) - 2*tf(v) );
projection += integral(vol, u, dof(v)*tf(v) - 2*tf(v), -1 );
projection += integral(vol, u, dof(v)*tf(v,sur) - 2*tf(v), +2, 1 );

// Assemble with a call to FFT to compute the 20 first harmonics:
projection += integral(vol, 20, (1-vh)*dof(vh)*tf(vh) + vh*tf(vh) );
projection += integral(sur, 20, vh*vh*dof(vh)*tf(vh) - tf(vh), +2 );
projection += integral(vol, 20, vh*dof(vh,sur)*tf(vh), +2, 2 );
// Same as above but assemble on the mesh deformed by field u:
projection += integral(vol, 20, u, vh*vh*dof(vh)*tf(vh) );
projection += integral(vol, 20, u, vh*vh*dof(vh)*tf(vh) - vh*tf(vh), -1 );
projection += integral(sur, 20, u, dof(vh)*tf(vh,sur), +2, 1 );
```

This adds a term to the formulation. All terms are added together and their sum equals zero.
For the first line in the basic version the term is assembled for unknowns (dof) and test functions (tf) defined on region 'vol' and for an integration on all elements in region 'vol' as well. When no region is specified for the dof or the tf then the element integration region (first argument) is used by default. Otherwise, e.g. on line 2 and 3 the dof or tf region used is the one requested. In other words on line 2 unknowns are defined on region 'vol' but test functions only on region 'sur' while on line 3 it is the opposite.

On line 2 of the basic version an extra int is added at the end compared to line 1. This extra int gives the extra number that should be added to the default integration order to perform the numerical integration in the assembly process. The default integration order is order of the unknown + order of the test function + 2. By increasing the integration order a more accurate assembly can be obtained, at the expense of an increased assembling time.
On line 3 there is another extra int which specifies the 'block number' of the term. The default value is 0. Here it is set to 2. This can be of interest when the formulation is generated since one can choose exactly which block numbers to generate and which ones not.
Line 4 through 6 are similar to the first 3 ones except that an extra argument is added. All calculations performed when assembling these 3 terms will be performed on the mesh deformed by field u (possibly on a curved mesh).
The last six lines are similar to the first 6 ones except that an extra int has been added (20 here). When the int is positive an FFT will be called during the assembly and the first 20 harmonics will be computed (the harmonics whose magnitude are below a threshold are disregarded). This must be called when assembling a multiharmonic formulation term that is nonlinear in the time variable.

`void generate(void)`

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1");
formulation projection;

projection += integral(vol, dof(v)*tf(v), +2, 2 );
projection += integral(vol, dt(dof(v))*tf(v) - 2*tf(v) );
projection += integral(vol, dtdt(dof(v))*tf(v) );

projection.generate();
```

This assembles all terms in the formulation.

`void generatestiffnessmatrix(void)`

```
...
projection.generatestiffnessmatrix();
```

This assembles only the terms in the formulation which have a dof and that dof has no time derivative applied to it. For multiharmonic formulations it generates all terms.
Here it only generates 'dof(v) $\cdot$ tf(v)'.

`void generatedampingmatrix(void)`

```
...
projection.generatedampingmatrix();
```

This assembles only the terms in the formulation which have a dof and that dof has an order one time derivative applied to it (i.e. dt). For multiharmonic formulations it generates nothing.
Here it only generates 'dt(dof(v)) $\cdot$ tf(v)'.

```
void generatemassmatrix(void)
```

```
...
projection.generatemassmatrix();
```

This assembles only the terms in the formulation which have a dof and that dof has an order two time derivative applied to it (i.e. dtdt). For multiharmonic formulations it generates nothing.
Here it only generates 'dtdt(dof(v)) $\cdot$ tf(v)'.

```
void generaterhs(void)
```

```
...
projection.generaterhs();
```

This assembles only the terms in the formulation which have no dof.
Here it only generates '$-2 \cdot$ tf(v)'.

```
void generate(std::vector<int> contributionnumbers)
```

```
...
projection.generate({0,2});
```

This generates all terms with block number 0 or 2. Here it means all terms are generated since there are only 2 and 0 (default) block numbers.

```
void generate(int contributionnumber)
```

```
...
projection.generate(2);
```

This generates only the block number 2 (i.e. the first integral term).

```
vec b(bool keepvector = false)
```

```
int vol = 1;
field v("h1");
formulation projection;
projection += integral(vol, dof(v)*tf(v) - 2*tf(v) );
```

```
projection.generate();
vec b = projection.b(); // Or vec b = projection.b(false);
```

This gives the rhs vector b that was assembled during the 'generate' call. If you select 'true' it means everything that has been assembled during the 'generate' call will be added to what is assembled in another 'generate' call later on. If you select 'false' (default) then all the generated data is no more in the formulation, only in vector b.
All entries in vector b that correspond to Dirichlet-constrained fields are set to the constraint value.

```
mat A(bool keepfragments = false)
```

...

```
mat A = projection.A(); // Or mat A = projection.A(false);
```

This gives the matrix A (of $Ax = b$) that was assembled during the 'generate' call. If you select 'true' it means everything that has been assembled during the 'generate' call will be added to what is assembled in another 'generate' call later on. If you select 'false' then all the generated data is no more in the formulation, only in matrix A.

```
vec rhs(bool keepvector = false)
```

...

```
vec rhs = projection.rhs();
```

Does the same as .b().

```
mat K(bool keepfragments = false)
```

...

```
mat K = projection.K();
```

Gives the stiffness matrix, i.e. the matrix that is assembled with all terms where the unknown (dof) has no time derivative. For multiharmonic formulations this holds everything. Refer to .A() for the boolean argument.

```
mat C(bool keepfragments = false)
```

...

```
mat C = projection.C();
```

Gives the damping matrix, i.e. the matrix that is assembled with all terms where the unknown (dof) has an order one time derivative. For multiharmonic formulations C is empty. Refer to .A() for the boolean argument.

```
mat M(bool keepfragments = false)
```

```
...
mat M = projection.M();
```

Gives the mass matrix, i.e. the matrix that is assembled with all terms where the unknown (dof) has an order two time derivative. For multiharmonic formulations M is empty. Refer to .A() for the boolean argument.

```
mat getmatrix(int KCM, bool keepfragments = false)
```

```
...
mat K = projection.getmatrix(0);
mat C = projection.getmatrix(1);
mat M = projection.getmatrix(2);
```

The first line is equivalent to .K(), the second to .C() and the third to .M(). Refer to .A() for the boolean argument.