sparselizard

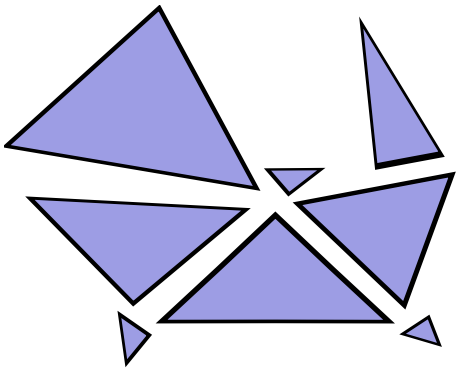
the user friendly

finite element

c++ library

Alexandre Halbach

December 7, 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 (a working example solving an electrostatic problem on a 3D disk is shown below) 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.

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.

```
int vol = 1, sur = 2;             // The domain regions as defined in 'circle.geo'
mesh mymesh("circle.msh");        // The elements in the mesh can be curved!

field v("h1");                    // Nodal shape functions for the electric potential
v.setorder(vol, 4);               // Interpolation order 4 on the whole domain
v.setconstraint(sur, 2);          // Force 2 V on the disk external boundary

formulation electrostatics;       // Electrostatics with 1 nC/m^3 charge density
electrostatics += integral(vol, -8.85e-12 * grad(dof(v)) * grad(tf(v)) + 1e-9 * tf(v));
electrostatics.generate();
vec solv = solve(electrostatics.A(), electrostatics.b());

v.getdata(vol, solv);             // Transfer data from solution vector to v field
(-grad(v)).write(vol, "E.pos");   // Write the electric field
```

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.

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 sparselizard, the external libraries below must be installed. For that make sure you have the gcc, g++ and the **standard** gfortran compilers. On Ubuntu linux install them 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 involving large sparse algebraic matrices. 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.
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 'Target visualization error' to the smallest possible and 'Maximum recursion level' to 4 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.

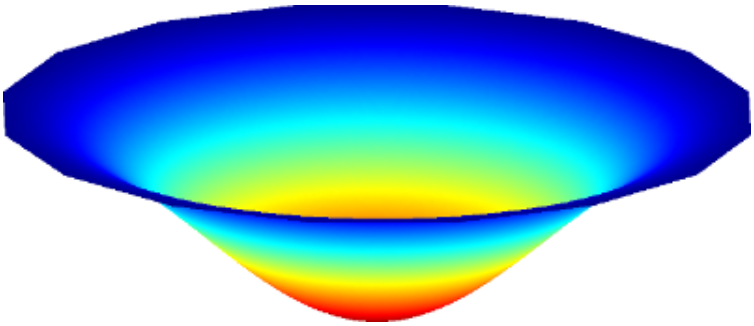


Figure 1: Exaggerated deflection of the 3D disk

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 (and namespaces) that you can use in your simulations. In any case we recommend to follow the examples in the **examples folder** to get used to sparselizard.

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

The densematrix object stores a row-major array of doubles that corresponds to a dense matrix. Only the functions required to use the other objects are detailed.

```
densematrix(int numberofrows, int numberofcolumns)
```

```
densematrix B(2,3);
```

This creates an all zero matrix with 2 rows and 3 columns.

```
densematrix(int numberofrows, int numberofcolumns, double initvalue)
```

```
densematrix B(2,3, 12);
```

This creates a matrix with 2 rows and 3 columns. All entries are assigned to value 'initvalue'.

```
densematrix(int numberofrows, int numberofcolumns, const std::vector<double> valvec)
```

```
densematrix B(2,3, {1,2,3,4,5,6});
```

This creates a matrix with 2 rows and 3 columns. The entries are assigned to the values of 'valvec'. As an example B at row 0, column 2 is set to 3.

```
densematrix(int numberofrows, int numberofcolumns, int init, int step)
```

```
densematrix B(2,3, 0,1);
```

This creates a matrix with 2 rows and 3 columns whose values increase by steps of 'step' (first is 'init').

```
int countrows(void)
```

```
densematrix B(2,3);
int numrows = B.countrows();
```

This counts the number of rows in the dense matrix (2 here).

```
int countcolumns(void)
```

```
densematrix B(2,3);
int numcols = B.countcolumns();
```

This counts the number of columns in the dense matrix (3 here).

```
void print(void)
```

```
densematrix B(2,3, 0,1);
B.print();
```

This prints the matrix values to the console.

```
void printsize(void)
```

```
densematrix B(2,3);
B.printsize();
```

This prints the matrix size to the console.

```
double* getvalues(void)
```

```
densematrix B(2,3);
double* vals = B.getvalues();
```

This returns a double array pointer that lets you read or write any value to the array (stay in the array to avoid a segmentation fault). The array is deallocated when the matrix goes out of scope.

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

```
mesh mymesh("circle.msh")
field v("h1");
expression myexpression(2,3,{12,v,v*(1-v),3,14-v,0});
myexpression.print();
```

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;
// Mesh with curved elements at order 3 to accurately
// capture the cylinder geometry and get value 3.7699!
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)

```
...
field u("h1xyz");
double integralvaluedeformedmesh = 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, 10, "order3.pos", 3);
abs(v).write(vol, 10, "order1.pos");
```

This writes an expression to a file, here with either an order 3 interpolation or, in the latter case with a default order 1 interpolation. The 10 means the expression is treated as multiharmonic, nonlinear in the time variable and an FFT is performed to get the 10 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(u).write(vol, 10, 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 computed. 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 requires 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();
```

Prints 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.

4.3 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)
```

```
mesh mymesh("circle.msh");
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 are two extra bubble modes 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)`

```
mesh mymesh("circle.msh");
field v("h1", {1,4,5,6});
field v4 = v.harmonic(4);
```

Consider the infinite Fourier series of a field that is 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 third line in the example gets harmonic V_4 from field v. It can then be used like any other field.

`int countcomponents(void)`

```
mesh mymesh("circle.msh");
field E("hcurl");
int numcomp = E.countcomponents();
```

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

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

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

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

Print a string showing the harmonics in the field.

`void setname(std::string name)`

```
mesh mymesh("circle.msh");
field v("h1");
v.setname("v");
```

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

```
void print(void)
```

```
mesh mymesh("circle.msh");
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. +3 or -1) 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 12 until w is set to a non-zero value). An extra int argument (e.g. +3 or -1) 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)
```

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

```
mesh mymesh("circle.msh");
field u("h1xyz");
field ux = u.compx();
```

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

```
field compy(void)
```

```
mesh mymesh("circle.msh");
field u("h1xyz");
field uy = u.compy();
```

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

```
field compz(void)
```

```
mesh mymesh("circle.msh");
field u("h1xyz");
field uz = u.compz();
```

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

```
field harmonic(int harmonicnumber)
```

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

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

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

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

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;

// Nine valid += calls are listed below

// Basic version:
```

```
projection += integral(vol, dof(v)*tf(v) );
projection += integral(vol, 2*dof(v)*tf(v,sur), +1 );
projection += integral(vol, compx(u)*dof(v,sur)*tf(v) - 2*tf(v), +1, 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), -1 );
projection += integral(vol, u, dof(v,sur)*tf(v,sur), +3, 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), +3 );
projection += integral(vol, 20, vh*dof(vh,sur)*tf(vh), +1, 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), +1, 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 done 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.

```
int countdofs(void)
```

```
...
int numdofs = projection.countdofs();
```

This returns the number of degrees of freedom defined in the formulation.

```
void generate(void)
```

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

projection += integral(vol, dof(v)*tf(v), 0, 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)
```

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

4.5 The *intdensematrix* object (/src/intdensematrix.h):

The intdensematrix object stores a row-major array of ints that corresponds to a dense matrix. Only the functions required to use the other objects are detailed.

`intdensematrix(int numberofrows, int numberofcolumns)`

```
intdensematrix B(2,3);
```

This creates an all zero matrix with 2 rows and 3 columns.

`intdensematrix(int numberofrows, int numberofcolumns, int initvalue)`

```
intdensematrix B(2,3, 12);
```

This creates a matrix with 2 rows and 3 columns. All entries are assigned to value 'initvalue' (12 here).

`intdensematrix(int numberofrows, int numberofcolumns, const std::vector<int> valvec)`

```
intdensematrix B(2,3, {1,2,3,4,5,6});
```

This creates a matrix with 2 rows and 3 columns. The entries are assigned to the value of 'valvec'. As an example B at row 0, column 2 is set to 3.

`intdensematrix(int numberofrows, int numberofcolumns, int init, int step)`

```
intdensematrix B(2,3, 0,1);
```

This creates a matrix with 2 rows and 3 columns whose values increase by steps of 'step' (first is 'init').

`int countrows(void)`

```
intdensematrix B(2,3);
int numrows = B.countrows();
```

This counts the number of rows in the dense matrix (2 here).

`int countcolumns(void)`

```
intdensematrix B(2,3);
int numcols = B.countcolumns();
```

This counts the number of columns in the dense matrix (3 here).

```
void print(void)
```

```
intdensematrix B(2,3, 0,1);
B.print();
```

This prints the matrix values to the console.

```
void printsize(void)
```

```
intdensematrix B(2,3);
B.printsize();
```

This prints the matrix size to the console.

```
int* getvalues(void)
```

```
intdensematrix B(2,3);
int* vals = B.getvalues();
```

This returns an int array pointer that lets you read or write any value to the array (stay in the array to avoid a segmentation fault). The array is deallocated when the matrix goes out of scope.

4.6 The *mat* object (/src/formulation/mat.h):

The mat object holds a sparse algebraic matrix.

```
mat(formulation myformulation, intdensematrix rowadresses, intdensematrix coladresses,
densematrix vals)
```

```
mesh mymesh("circle.msh");
int vol = 1, sur = 2;
field v("h1");
formulation projection;
projection += integral(vol, dof(v)*tf(v) - 2*tf(v));

intdensematrix adresses(projection.countdofs(), 1, 0, 1);
densematrix vals(projection.countdofs(), 1, 12);

mat A(projection, adresses, adresses, vals);
```

This creates a matrix object whose dof structure is the one in 'projection'. The matrix non zero values are obtained from the input (in format row of value - column of value - value). Here a diagonal matrix is created with value 12 everywhere.

```
int countrows(void)
```

```
...
int numrows = A.countrows();
```

This counts the number of rows in the matrix.

```
int countcolumns(void)
```

```
...
int numcols = A.countcolumns();
```

This counts the number of columns in the matrix.

```
int countnnz(void)
```

```
...
int numnnz = A.countnnz();
```

This counts the number of non zero entries in the matrix.

```
void removeconstraints(void)
```

```
...
v.setconstraint(sur);
A.removeconstraints();
```

This removes the Dirichlet constraint-entries of the matrix. The new matrix has a structure based on a copy of the one defined in formulation 'projection' but without the Dirichlet constraint entries.

```
void reuselu(void)
```

```
...
A.reuselu();
```

The LU decomposition of the matrix will be reused in mathop::solve.

```
Mat getpetsc(void)
```

```
...
Mat petscmat = A.getpetsc();
```

This outputs the petsc object corresponding to the matrix. It can be used as any other petsc object.

```
void print(void)

...
A.print();
```

This prints the matrix size and values.

```
mat copy(void)

...
mat copiedmat = A.copy();
```

This creates a full copy of the matrix. Only the values are copied (e.g. the LU reuse property is set back to the default no reuse).

```
operators + - *

...
vec b(projection);
vec x(projection);

vec residual = b - A*x;
mat AA = A*A;
```

Any valid +, - and * operation between vectors and matrices is permitted.

4.7 The *mathop* namespace (/src/expression/operation/mathop.h):

The mathop namespace provides a collection of mathematical tools.

```
int regionunion(const std::vector<int> physregs)

mesh mymesh("circle.msh");
int sur = 2, top = 3;
int surandtop = regionunion({sur, top});
```

This creates a new physical region that is the union of surfaces 2 and 3.

```
int regionintersection(const std::vector<int> physregs)

mesh mymesh("circle.msh");
int sur = 2, top = 3;
int line = regionintersection({sur, top});
```

This creates a new physical region that is the intersection of surfaces 2 and 3. Here the new region is a line.

```
expression normal(int surfphysreg)
```

```
mesh mymesh("circle.msh");
int sur = 2;
normal(sur).write(sur, "normal.pos");
```

This defines the vector normal to a surface in 3D or a line in 2D. Depending on the face orientation in the mesh the normal is flipped or not.

```
void setfundamentalfrequency(double f)
```

```
setfundamentalfrequency(50);
```

This defines the fundamental frequency (in Hz) required for multiharmonic problems.

```
void settime(double t)
```

```
settime(1e-3);
```

This sets the time variable t, here to 1 ms.

```
double gettime(void)
```

```
settime(1e-3);
double gettime();
```

This gets the value of the time variable t.

```
expression t(void)
```

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1");
v.setconstraint(vol, sin(2*t()));
```

This gives the time variable in form of an expression. The evaluation gives a value equal to gettime().

```
expression dx(expression input)
```

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1");
dx(v).write(vol, "dxv.pos");
```

This is the x space derivative.

```
-------------------------------------------------------------
expression dy(expression input)

...
dy(v).write(vol, "dyv.pos");
```

This is the y space derivative.

```
-------------------------------------------------------------
expression dz(expression input)

...
dz(v).write(vol, "dzv.pos");
```

This is the z space derivative.

```
-------------------------------------------------------------
expression dt(expression input)

...
setfundamentalfrequency(50);
field vmh("h1", {2,3});
dt(vmh).write(vol, "dtv.pos");
dt(dt(vmh)).write(vol, "dt-dtv.pos");
```

This is the first order time derivative.

```
-------------------------------------------------------------
expression dtdt(expression input)

...
setfundamentalfrequency(50);
dtdt(vmh).write(vol, "dtdtv.pos");
```

This is the second order time derivative.

```
-------------------------------------------------------------
expression sin/cos/abs/sqrt/log10(expression input)

expression expr1 = sin(2);
expression expr2 = cos(2);
expression expr3 = abs(2);
expression expr4 = sqrt(2);
expression expr5 = log10(2);
```

This is the sin/cos/abs/sqrt/log10 function.

```
-------------------------------------------------------------
expression pow(expression base, expression exponent)
```

```
expression expr = pow(2, 2);
```

This is the power expression (^ notation is not supported).

---

expression comp(int selectedcomp, expression input)

```
mesh mymesh("circle.msh");
int vol = 1;
field u("h1xyz");
comp(0, 2*(u+u)).write(vol, "expr.pos");
```

This returns the selected component of a column vector expression. For a matrix expression the whole row is returned in form of an expression. Select component 0 for the first (x) component, 1 for the second (y) and 2 for the third one (z).

---

expression compx(expression input)

```
...
compx(2*(u+u)).write(vol, "compx.pos");
```

This is equivalent to comp(0, expression).

---

expression compy(expression input)

```
...
compy(2*(u+u)).write(vol, "compy.pos");
```

This is equivalent to comp(1, expression).

---

expression compz(expression input)

```
...
compz(2*(u+u)).write(vol, "compz.pos");
```

This is equivalent to comp(2, expression).

---

expression entry(int row, int col, expression input)

```
...
expression arrayentryrow2col0 = entry(2, 0, u);
```

This gets the (row, col) entry in the vector or matrix expression.

---

expression transpose(expression input)

```
...
expression utransposed = transpose(2*u);
```

This transposes the vector or matrix expression.

```
expression inverse(expression input)
```

```
expression matexpr(3, 3, {1,2,3,4,5,6,7,8,9});
expression inversedmat = inverse(matexpr);
```

This gets the inverse of a square matrix.

```
expression determinant(expression input)
```

```
expression matexpr(3, 3, {1,2,3,4,5,6,7,8,9});
expression detmat = determinant(matexpr);
```

This gets the determinant of a square matrix.

```
expression grad(expression input)
```

```
mesh mymesh("circle.msh");
int vol = 1;
field v("h1");
grad(v).write(vol, "gradv.pos");
```

This computes the gradient of a scalar expression.

```
expression curl(expression input)
```

```
...
field u("h1xyz");
curl(u).write(vol, "curlu.pos");
```

This computes the curl of a vector expression.

```
expression dof(expression input, int physreg = -1)
```

```
mesh mymesh("circle.msh");
int vol = 1, sur = 2;
field v("h1");
formulation projection;
projection += integral(vol, dof(v)*tf(v) - 2*tf(v));
projection += integral(vol, dof(v, 1)*tf(v) - 2*tf(v));
```

This declares an unknown field (dof for degree of freedom). The dofs are defined only on region 'physreg', which when not provided is set to the element integration region (here to vol).

```
expression tf(expression input, int physreg = -1)
```

```
...
projection += integral(vol, dof(v)*tf(v, 1)  - 2*tf(v));
```

This declares a test function field. The test functions are defined only on region 'physreg', which when not provided is set to the element integration region (here to vol).

---

expression array1x2(expression term11, expression term12) and the like

```
expression myarray = array2x3(1,2,3,4,5,6);
```

This defines a vector or matrix expression of up to 3×3 size. The array is populated in a row major way. In the example the first row is (1,2,3) and the second row is (4,5,6).

---

vec solve(mat A, vec b)

```
mesh mymesh("circle.msh");
int vol = 1, sur = 2;
field v("h1");
v.setconstraint(sur);
formulation projection;
projection += integral(vol, dof(v)*tf(v)  - 2*tf(v));

vec sol = solve(projection.A(), projection.b());
```

This solves an algebraic problem with a (possibly reused) LU decomposition by calling the parallel mumps solver via petsc.

---

expression predefinedelasticity(expression u, expression Eyoung, expression nupoisson)

```
mesh mymesh("circle.msh");
int vol = 1;
field u("h1xyz");
formulation elasticity;
elasticity += integral(vol, predefinedelasticity(u, 150e9, 0.3));
elasticity += integral(vol, -2330*dtdt(dof(u))*tf(u));
```

This defines a classical linear elasticity formulation. Field u is the mechanical displacement. Expression 'Eyoung' is Young's modulus while 'nupoisson' is Poisson's ratio. The second term in the formulation is the inertia term.

---

expression predefinedelectrostaticforce(expression gradtfu, expression gradv, expression epsilon)

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

```
int vol = 1;
field v("h1"), u("h1xyz");
formulation elasticity;
elasticity += integral(vol, predefinedelasticity(u, 150e9, 0.3));
elasticity += integral(vol, predefinedelectrostaticforce(grad(tf(u)),
grad(v), 8.854e-12);
```

This defines the formulation term to compute the electrostatic forces (obtained with the virtual work principle). The first argument is the gradient of the mechanical displacement test function (a column vector), the second is the gradient of the scalar electric potential field and the third is the electric permittivity.

4.8 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("circle.msh");
mymesh.write("circle2.msh");
```

This writes the mesh in object 'mymesh' to file 'circle2.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'.

4.9 The *parameter* object (/src/expression/parameter.h):

The parameter object can hold different expression objects on different geometrical regions.

```
parameter(void)
```

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

This creates parameter E (not yet defined).

```
parameter(int numrows, int numcols)
```

```
mesh mymesh("circle.msh");
parameter E(3,3);
```

This creates a 3 by 3 matrix parameter E (not yet defined).

```
int countrows(void)
```

```
mesh mymesh("circle.msh");
parameter E(3,3);
int numrows = E.countrows();
```

This returns the number of rows in the parameter.

```
int countcolumns(void)
```

```
mesh mymesh("circle.msh");
parameter E(3,3);
int numcols = E.countcolumns();
```

This returns the number of columns in the parameter.

```
parameterselectedregion operator|(int physreg)
```

```
mesh mymesh("circle.msh");
int vol = 1;
parameter E;
E|vol = 150e9;
```

This sets the parameter expression on region 1. Since surface region 2 and 3 are part of volume region 1 in "circle.msh" the parameter is also defined on these two surface regions.

```
void print(void)
```

```
mesh mymesh("circle.msh");
int vol = 1;
parameter E;
E|vol = 150e9;
E.print();
```

This prints information on the parameter.

4.10 The *vec* object (/src/formulation/vec.h):

The vec object holds a vector, be it the solution vector of an algebraic problem or its right handside.

```
vec(formulation formul)
```

```
mesh mymesh("circle.msh");
int vol = 1, sur = 2;
field v("h1");
formulation projection;
projection += integral(vol, dof(v)*tf(v) - 2*tf(v));

vec b(projection);
```

This creates an all zero vector whose structure and size is the one of formulation 'projection'.

```
int size(void)
```

```
...
int mysize = b.size();
```

This returns the size of the vector. It is equal to the number of dofs in formulation 'projection'.

```
void removeconstraints(void)
```

```
...
v.setconstraint(sur);
b.removeconstraints();
```

This removes the Dirichlet constraint-entries of the vector. The new vector has a structure based on a copy of the one defined in formulation 'projection' but without the Dirichlet constraint entries.

```
void updateconstraints(void)
```

```
...
v.setconstraint(sur,1);
b.updateconstraints();
```

This updates the value of all Dirichlet constraint-entries in the vector.

```
void setvalues(intdensematrix adresses, densematrix valsmat, std::string op = "set")
```

```
mesh mymesh("circle.msh");
int vol = 1, sur = 2;
field v("h1");
formulation projection;
projection += integral(vol, dof(v)*tf(v) - 2*tf(v));

vec myvec(projection);

intdensematrix adresses(myvec.size(), 1, 0, 1);
densematrix vals(myvec.size(), 1, 12);

myvec.setvalues(adresses, vals);
// or .setvalues(adresses, vals, "set")
// or .setvalues(adresses, vals, "add")
```

This replaces the values in 'myvec' at the adresses in 'adresses' by the values in 'vals'. With "add" the values are not replaced but added. Here all entries are replaced by value 12.

```
densematrix getvalues(intdensematrix adresses)
```

```
...
densematrix vecvals = myvec.getvalues(adresses);
```

This gets the values in the vector that are at the adresses given in 'adresses'.

`Vec getpetsc(void)`

```
...
Vec petscvec = myvec.getpetsc();
```

This gives the petsc object coresponding to the vector. It can be used like any other petsc object.

`void print(void)`

```
...
myvec.print();
```

This prints the vector values.

`vec copy(void)`

```
...
vec copiedvec = myvec.copy();
```

This creates a full copy of the vector.

`double norm(std::string type = "2")`

```
...
double mynorm2 = myvec.norm(); // or .norm("2")
double mynorm1 = myvec.norm("1");
double mynorminfinity = myvec.norm("infinity");
```

This outputs the 1, 2 or infinity norm of the vector (default is the 2 norm).

`operators + - *`

```
...
vec b(projection);
vec x(projection);
mat A = projection.A();

vec residual = b - A*x;
```

Any valid +, - and * operation between vectors and matrices is permitted.

4.11 The *wallclock* object (/src/wallclock.h):

The wallclock object provides an easy way to measure wall execution time.

`wallclock(void)`

```
wallclock myclock();
```

This initialises the wall clock object.

`void tic(void)`

```
wallclock myclock();
myclock.tic();
```

This resets the clock.

`double toc(void)`

```
wallclock myclock();
double timelapsed = myclock.toc();
```

This returns the time elapsed (in ns).

`void print(void)`

```
wallclock myclock();
myclock.print();
```

This prints the time elapsed in the most appropriate format (ns, us, ms or s).

`void pause(void)`

```
wallclock myclock();
myclock.pause();
// Do something
myclock.resume();
myclock.print();
```

This pauses the clock. The pause() and resume() functions allow to time selected operations in loops.

`void resume(void)`

```
wallclock myclock();
myclock.pause();

for (int i = 0; i < 10; i++)
{
        myclock.resume();
        // Do something and time it
```

```
        myclock.pause();
        // Do something else
}
myclock.print();
```

This resumes the clock. The pause() and resume() functions allow to time selected operations in loops.