sparselizard

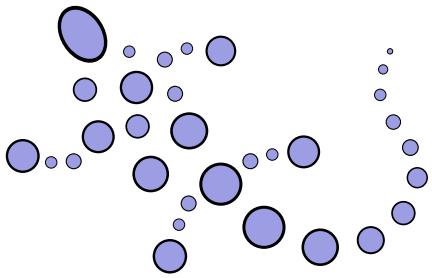
the user friendly

finite element

c++ library

Alexandre Halbach

January 17, 2023



Contents

1 What is sparselizard

Sparselizard (GNU GPL, Copyright (C) 2020-2021 A. Halbach, Copyright (C) 2018-2019 A. Halbach, IMEC, Copyright (C) 2017 A. Halbach and C. Geuzaine, University of Liege) is a high-performance, multiphysics, open source, *hp-adaptive* c++ finite element library. It is meant to be user-friendly as illustrated below. The following example solves an electrostatic problem on a 3D disk with volume charges and grounded at its boundary:

```
int vol = 1, sur = 2;              // Disk volume and boundary as set in 'disk.geo'
mesh mymesh("disk.msh");           // The elements in the GMSH mesh can be curved!

field v("h1");                     // Nodal shape functions for the electric potential
v.setorder(vol, 2);                // Interpolation order 2 on the whole domain
v.setconstraint(sur, 0);           // Force 0 V on the disk boundary

formulation elec;                  // Electrostatics with 1 nC/m^3 charge density
elec += integral(vol, -8.854e-12 * grad(dof(v)) * grad(tf(v)) + 1e-9 * tf(v));

elec.solve();                      // Generate, solve and save solution to field v

(-grad(v)).write(vol, "E.vtk", 2); // Write the electric field to ParaView format
```

The equations are provided in a weak formulation as described in `https://en.wikipedia.org/wiki/Weak_formulation`.

For a detailed list of capabilities and examples refer to the official website `www.sparselizard.org`.

2 How to compile sparselizard

Get the sparselizard source code from github at `https://github.com/halbux/sparselizard`. Before compiling it, the external libraries below must be installed. For that make sure you have git, cmake and the gcc, g++ and the standard gfortran compilers. On Ubuntu linux install them with:

sudo apt−get install git
sudo apt−get install cmake
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 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.

The external libraries used are the following:

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

Once all external libraries are successfully installed, sparselizard can be compiled as follows:

mkdir build && cd build
cmake ..
cmake −−build . −j$(nproc)

To provide a custom path to the libraries use:

cmake .. −DPETSC_PATH=/path/petsc −DGMSH_PATH=/path/gmsh −DMPI_PATH=/path/mpi

or use the cmake GUI.

3 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.

3.1 The *densemat* object (/src/densemat.h):

The densemat 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.

`densemat::densemat(int numberofrows, int numberofcolumns)`

densemat B(2,3);

This creates a matrix with 2 rows and 3 columns. Its values may be undefined.

```
densemat::densemat(int numberofrows, int numberofcolumns, double initvalue)
```

densemat B(2,3, 12);

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

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

densemat 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.

```
densemat::densemat(int numberofrows, int numberofcolumns, double init, double step)
```

densemat 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').

```
densemat::densemat(std::vector<densemat> input)
```

densemat A(2,3, 0,1);
densemat B(2,3, 10,1);
densemat AB({A,B});
AB.print();

This creates a matrix that is the vertical concatenation of matrices.

```
int densemat::countrows(void)
```

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

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

```
int densemat::countcolumns(void)
```

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

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

```
int densemat::count(void)
```

densemat B(2,3);
int numentries = B.count();

This counts the number of entries in the dense matrix (number of rows x number of columns, 6 here).

`void densemat::print(void)`

densemat B(2,3, 0,1);
B.print ();

This prints the matrix values to the console.

`void densemat::printsize(void)`

densemat B(2,3);
B. printsize ();

This prints the matrix size to the console.

`double* densemat::getvalues(void)`

densemat 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.

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

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

`expression::expression(void)`

expression myexpression;

This creates an empty expression object.

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

mesh mymesh("disk.msh");
field v("h1");
expression myexpression(2,3,{12,v,v*(1-v),3,14-v,0});
myexpression.print ();
expression myexpressionsym (3,3,{1,2,3,4,5,6});
myexpressionsym.print();

```
expression myexpressiondiag(3,3,{1,2,3});
myexpressiondiag.print();
```

The first call creates a 2×3 sized expression array (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.
The second call creates a symmetric expression array where the provided vector corresponds to the lower triangular part (top to bottom and left to right ordered).
The third call creates a diagonal expression array where the diagonal is filled with the expressions provided in the input vector.

`expression::expression(const std::vector<std::vector<expression>> input)`

```
mesh mymesh("disk.msh");
field v("h1");
expression blockleft(3,1,{1,2*v,3});
expression blockrighttop(1,2,{4,5});
expression blockrightbottom(2,2,{6,7,8,9});
expression exprconcatenated({{blockleft},{blockrighttop,blockrightbottom}});
exprconcatenated.print();
```

This creates an expression obtained from the rowwise and columnwise concatenation of expressions. Every expression vector input[i] is concatenated columnwise with the others. Every expression in a given expression vector input[i] is concatenated rowwise with the other expressions in that vector.

`expression::expression(expression condexpr, expression exprtrue, expression exprfalse)`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
field x("x"), y("y");
expression expr(x+y, 2*x, 0);
expr.print();
expr.write(top, "conditionalexpr.vtk", 1);
```

This creates a conditional expression. If the first argument is greater or equal to zero then the expression is equal to the expression provided as second argument. If smaller than zero it is equal to the third argument expression.

`expression::expression(spline spl, expression arg)`

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

std::vector<double> temperature = {273,300,320,340};
```

```
std::vector<double> youngsmodulus = {5e9,4e9,2.5e9,1e9};
spline spl(temperature, youngsmodulus);
// The spline object can also be created from a measurement data file:
// spline spl("smoothedmeasurements.txt");

// Temperature field:
field T("h1");
// Young's modulus as a cubic (natural) spline interpolation of the experimental data:
expression E(spl, T);
```

This creates a continuous expression based on discrete data samples. As an application example, if measurements of a material stiffness (Young's modulus E) have been performed for a set of temperatures T (as illustrated in the example above) then this constructor allows to define expression E that provides a 3rd order (natural) spline interpolation of Young's modulus in the measured discrete temperature range. Field T can contain any space-dependent temperature profile as long as it is in the temperature data range provided.
Refer to the 'spline' object for more details on how to create it.

`expression::expression(std::vector<double> pos, std::vector<expression> exprs, expression var)`

```
// Define an electric supply voltage profile in time that is:
// - 0V for time range [0,1] sec.
// - increases linearly from 0V to 1V for time range [1,3] sec.
// - 1V for time range [3,5] sec.
expression vsupply({1.0,3.0}, {0,0.5*(t()-1),1.0}, t());
```

This creates an expression defined by intervals for values *var* in range $]-\infty,+\infty[$.

`expression::expression(int m, int n, std::vector<densemat> customfct(std::vector<densemat>), std::vector<expression> exprs)`

```
std::vector<densemat> customfunction(std::vector<densemat> evaledexprs)
{
        int nrows = evaledexprs[0].countrows();
        int ncols = evaledexprs[0].countcolumns();
        densemat outx(nrows,ncols), outy(nrows,ncols);
        double* ox = outx.getvalues();
        double* oy = outy.getvalues();
        double* a = evaledexprs[0].getvalues();
```

```
        double* b = evaledexprs[1].getvalues();
        double* c = evaledexprs[2].getvalues();
        double* d = evaledexprs[3].getvalues();
        double* e = evaledexprs[4].getvalues();
        for (int i = 0; i < nrows*ncols; i++)
        {
                ox[i] = a[i]+b[i]+c[i];
                oy[i] = e[i];
        }
        return {outx,outy};
}

int main(void)
{
        int vol = 1, sur = 2, top = 3;
        mesh mymesh("disk.msh");
        field u("h1xyz"), x("x"), y("y");
        u.setorder(vol, 1);
        u.setvalue(vol, array3x1(1,2,3));
        // Custom expression whose value is defined by 'customfunction':
        expression customexpr(2, 1, customfunction, {x,x,u});
        compx(customexpr).write(vol, "customx.pos", 1);
        compy(customexpr).write(vol, "customy.pos", 1);
}
```

This creates an $m \times n$ expression based on a custom function provided by the user. The value of the expression entry (i, j) is $customfct(exprs)[i \cdot n + j]$. The custom function is called at every evaluation of the expression. Its argument provides the numerical values of all expressions in 'exprs' in the order provided (non scalar expressions are flattened row by row). The function must return $m \cdot n$ densemat objects of same size as the argument densemat objects.

`int expression::countrows(void)`

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

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

`int expression::countcolumns(void)`

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

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

`expression expression::getrow(int rownum)`

expression myexpression (2,2,{0,1,2,3});
expression subexpr = myexpression.getrow(1);
subexpr.print ();

This returns a row of a matrix expression.

`expression expression::getcolumn(int colnum)`

expression myexpression (2,2,{0,1,2,3});
expression subexpr = myexpression.getcolumn(0);
subexpr.print ();

This returns a column of a matrix expression.

`void expression::reorderrows(std::vector<int> neworder)`

expression expr (2,2,{0,1,2,3});
expr.print ();
expr.reorderrows({1,0});
expr.print ();

This reorders the rows of a matrix expression.

`void expression::reordercolumns(std::vector<int> neworder)`

expression expr (2,2,{0,1,2,3});
expr.print ();
expr.reordercolumns({1,0});
expr.print ();

This reorders the columns of a matrix expression.

`std::vector<double> expression::max/min(int physreg, int refinement,`
`std::vector<double> xyzrange = {})`

mesh mymesh("disk.msh");
int vol = 1;
field x("x");
std :: vector<double> maxdata = (2∗x).max(vol, 1);
std :: vector<double> maxdatainbox = (2∗x).max(vol, 5, {−2,0, −2,2, −2,2});

This gives the max/min value of the expression over the geometric region 1. The search of the max/min can be restricted to a box delimited by the last argument (optional) whose form is {xboxmin xboxmax, yboxmin, yboxmax, zboxmin, zboxmax}. The output is {maxvalue, xcoordmax, ycoordmax, zcoordmax} or an empty vector if the argument region is empty or not in the box provided.
The max/min is obtained by splitting all elements 'refinement' times in each direction. Increasing the refinement will thus lead to a more accurate max/min value, but at an increasing computational cost. The max/min value is exact when the refinement nodes added to the elements correspond to the position of the max/min. For a first order nodal shape function interpolation on a mesh that is not curved the max/min is always exact to machine precision.

```
std::vector<double> expression::max/min(int physreg, expression meshdeform,
int refinement, std::vector<double> xyzrange = {})
```

```
...
field u("h1xyz");
u.setorder(vol, 1);
std::vector<double> maxdatadeformedmesh = (2*x).max(vol, u, 1);
```

Same as the previous function but the expression is evaluated on the geometry deformed by field u (possibly curved mesh). The max/min location and delimiting box are on the undeformed mesh.

```
void expression::interpolate(int physreg, std::vector<double>& xyzcoord,
std::vector<double>& interpolated, std::vector<bool>& isfound)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field x("x"), y("y"), z("z");
std::vector<double> xyzcoord = {0.5,0.6,0.05, 0.1,0.1,0.1};
std::vector<double> interpolated;
std::vector<bool> isfound;
array3x1(x,y,z).interpolate(vol, xyzcoord, interpolated, isfound);
printvector(interpolated);
```

This interpolates the expression at multiple points whose {x, y, z} coordinates are provided in 'xyzcoord'. The flattened interpolated expression values are concatenated for every interpolation point and put in 'interpolated'. If the ith point was not found in the elements of the physical region 'physreg' then 'isfound[i]' is set to false.

```
void expression::interpolate(int physreg, expression meshdeform,
std::vector<double>& xyzcoord, std::vector<double>& interpolated,
std::vector<bool>& isfound)
```

```
...
field u("h1xyz");
u.setorder(vol, 1);
array3x1(x,y,z).interpolate(vol, u, xyzcoord, interpolated, isfound);
printvector(interpolated);
```

Same as the previous function but the expression is computed on the mesh deformed by field 'u'.

```
std::vector<double> expression::interpolate(int physreg,
const std::vector<double> xyzcoord)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field x("x"), y("y"), z("z");
std::vector<double> interpolated = array3x1(x,y,z).interpolate(vol, {0.5,0.6,0.05});
printvector(interpolated);
```

This interpolates the expression at a single point whose {x, y, z} coordinates are provided as argument. The flattened interpolated expression values are returned if the point was found in the elements of the physical region 'physreg'. If not found an empty vector is returned.

```
std::vector<double> expression::interpolate(int physreg, expression meshdeform,
const std::vector<double> xyzcoord)
```

```
...
field u("h1xyz");
u.setorder(vol, 1);
interpolated = array3x1(x,y,z).interpolate(vol, u, {0.5,0.6,0.05});
printvector(interpolated);
```

Same as the previous function but the expression is computed on the mesh deformed by field 'u'.

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

```
mesh mymesh("disk.msh");
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 geometric region 1. The integration is exact for up to 4th order polynomials. Integrate *expression(1)* to calculate a volume/area/length. For axisymmetric problems

the value returned is the integral of the requested expression times the coordinate change Jacobian. In case of axisymmetry the volume/area/length of the 3D shape corresponding to the physical region on which to integrate can be obtained by integrating *expression(1)* and multiplying the output by 2π.

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

```
...
field u("h1xyz");
u.setorder(vol, 1);
double integralvaluedeformedmesh = myexpression.integrate(vol, u, 4);
```

This integrates the expression over the geometric 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 expression::write(int physreg, int numfftharms, std::string filename,
int lagrangeorder)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field u("h1xyz"), v("h1 ",{1,2,3});
u.setorder(vol, 1);
v.setorder(vol, 1);
(u*u).write(vol, 10, "order3.vtk", 3);
abs(v).write(vol, 10, "order1.vtk", 1);
```

This writes an expression to a file, here with either an order 3 or 1 interpolation order. 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 expression::write(int physreg, int numfftharms, expression meshdeform,
std::string filename, int lagrangeorder)
```

```
...
abs(u).write(vol, 10, u, "u.vtk", 2);
```

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

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

```
...
```

```
(1e8*u).write(vol, "uorder1.vtk", 1);
(1e8*u).write(vol, "uorder3.vtk", 3);
(1e8*u).write(vol, "uintime.vtk", 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 expression::write(int physreg, expression meshdeform, std::string filename,
int lagrangeorder, int numtimesteps = -1)
```

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

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

```
void expression::streamline(int physreg, std::string filename,
const std::vector<double>& startcoords, double stepsize, bool downstreamonly = false)
```

```
int vol = 1;
mesh mymesh("disk.msh");
field x("x"), y("y"), z("z");
std::vector<double> startcoords(12*3,0.05);
for (int i = 0; i < 12; i++)
        startcoords[3*i+0] += 0.1+0.05*i;
array3x1(y+2*x,-y+2*x,0).streamline(vol, "streamlines.vtk", startcoords, 1.0/100.0);
```

This follows and writes to disk all paths tangent to the expression vector that are starting at a set of points whose x, y and z coordinates are provided in 'startcoords' (these coordinates can for example be obtained via .getcoords() on a shape object). A fourth order Runge-Kutta algorithm is used ('stepsize' is related to the distance between two vector direction updates, decrease it to more accurately follow the paths). The paths will be followed as long as they remain in physical region 'physreg'. In case the vector norm is zero somewhere on the paths or a path is a closed loop the function might enter in an **infinite loop** and never return.
To use this function on closed loops (for example to get the magnetic field lines of a permanent magnet) a solution is to break the loops by excluding the permanent magnet domain from the physical region (the 'selectexclusion' function can be called for that) and set the starting coordinates on the boundary of the magnet.

```
void expression::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 expression::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 expression::iszero(void)`

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

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

`vec expression::atbarycenter(int physreg, field onefield)`

```
mesh mymesh("disk.msh");
int vol = 1;
field barycentervaluefield("one"), x("x");
vec myvec = (12*x).atbarycenter(vol, barycentervaluefield);
barycentervaluefield.setdata(vol, myvec);
(12*x).write(vol, "expression.vtk", 1);
barycentervaluefield.write(vol, "barycentervalues.vtk", 1);
```

This outputs a vector whose structure is based on the argument field and which contains the expression evaluated at the **barycenter of each reference element** of physical region 'physreg'. The **barycenter of the reference element might not be identical to the barycenter of the actual element in the mesh** (for curved elements, for general quadrangles, hexahedra and prisms).

`void expression::print(void)`

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

Prints the expression to the console.

`void expression::rotate(double ax, double ay, double az,`
`std::string leftop = "default", std::string rightop = "default")`

// Diagonal relative permittivity matrix for PZT:
expression P(3,3,{1704,1704,1433});
P = P * 8.854e−12;
// Coupling matrix [C/m^2] for PZT (6 rows, 3 columns):
expression C(6,3,{0,0,−6.62, 0,0,−6.62, 0,0,23.24, 0,17.03,0, 17.03,0,0, 0,0,0});
// Anisotropic elasticity matrix [Pa] for PZT. Ordering is [exx,eyy,ezz,gyz,gxz,gxy] (Voigt form).
// Lower triangular part (top to bottom and left to right) provided since it is symmetric.
expression H(6,6, {1.27e11, 8.02e10,1.27e11, 8.46e10,8.46e10,1.17e11, 0,0,0,2.29 e10,
0,0,0,0,2.29 e10, 0,0,0,0,0,2.34 e10});
// Rotate the PZT crystal 45 degrees around z:
H.rotate (0,0,45," K","KT"); C.rotate (0,0,45," K","RT"); P.rotate (0,0,45);

Let us call R (3×3) the classical 3D rotation matrix and K (6×6) the rotation matrix such that σ_V, the mechanical stress tensor in Voigt form (σ_{xx}, σ_{yy}, σ_{zz}, σ_{yz}, σ_{xz},σ_{xy}), is rotated as $K\sigma_V$. Both matrices correspond to a rotation first of ax degrees around the x axis then ay degrees around the y axis and finally az degrees around the z axis. Matrix R is orthogonal ($R^{-1} = R^T$) but matrix $\boldsymbol{K}$ **is not orthogonal** ($K^{-1} \neq K^T$).
This function left-multiplies the calling expression by 'leftop' and right-multiplies it by 'rightop' (if any). Options for 'leftop'/'rightop' are "", "R", "RT", "R-1", "R-T", "K", "KT", "K-1", "K-T" respectively for a left/right multiplication by nothing, R, R^T, R^{-1}, R^{-T}, K, K^T, K^{-1}, K^{-T}. The function can be called without providing the string arguments ("default") to rotate 3×3 tensors and 3×1 vectors (both with x, y, z component ordering).
The stress tensor in Voigt form σ_V is rotated as $K\sigma_V$ while the strain tensor in Voigt form ϵ_V is rotated as $(K^{-1})^T\epsilon_V$. Only the strain Voigt form ϵ_V is rotated with a different formula than $K\sigma_V$, because of the factor 2 added to the off-diagonal strain terms. In any case $(K^{-1})^T = (K^T)^{-1}$. Denoting the rotated quantities with the ' symbol one can deduce as an illustration the rotation formulas below.

The elasticity matrix H is such that $\sigma_V = H\epsilon_V$ and thus

$$K^{-1}\sigma_V^{'} = HK^T\epsilon_V^{'} \Rightarrow \sigma_V^{'} = KHK^T\epsilon_V^{'} \Rightarrow H^{'} = KHK^T$$

The compliance matrix C is such that $\epsilon_V = C\sigma_V$ and thus

$$K^T\epsilon_V^{'} = CK^{-1}\sigma_V^{'} \Rightarrow \epsilon_V^{'} = (K^{-1})^TCK^{-1}\sigma_V^{'} \Rightarrow C^{'} = (K^{-1})^TCK^{-1}$$

The 6×3 piezoelectric coupling matrix $[C/m^2]$ relating the electric field E to induced stresses is such that $\sigma_V = CE$ and thus

$$K^{-1}\sigma_V^{'} = C\,R^{-1}E^{'} \Rightarrow \sigma_V^{'} = KCR^TE^{'} \Rightarrow C^{'} = KCR^T$$

The 6×6 piezoresistivity matrix $[m^4/(sA^2)] \equiv [\Omega\, m/Pa]$ relating the stresses to the change in resistivity

$\Delta\rho_V$ (in Voigt form) is such that $\Delta\rho_V = \Pi\,\sigma_V$ and thus

$$K^{-1}\Delta\rho_V^{'} = \Pi K^{-1}\sigma_V^{'} \Rightarrow \Delta\rho_V^{'} = K\Pi K^{-1}\sigma_V^{'} \Rightarrow \Pi^{'} = K\Pi K^{-1}$$

`expression expression::at(int row, int col)`

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

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

`double expression::evaluate(void)`

settime (0.5);
expression expr = 2∗abs(−5∗t())+3;
std :: cout << "Value is " << expr.evaluate() << std::endl;

This evaluates a scalar, space-independent expression.

`expression expression::resize(int numrows, int numcols)`

expression myexpression (2,2,{1,2,3,4});
expression resizedexpr = myexpression.resize (1,3);
resizedexpr . print ();

This resizes an expression. Any newly created expression entry is set to zero.

`expression expression::operators + - * /`

mesh mymesh("disk.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.

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

mesh mymesh("disk.msh");
field v("h1");

This creates field v with nodal shape functions. The full list of shape functions 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 field in the E-formulation of electromagnetic wave propagation (here order 0 is allowed).
- "one", "one0", "one1", "one2", "one3" (trailing "xy" or "xyz" allowed) shape functions have a single shape function equal to a constant one on respectively an n, 0, 1, 2, 3 dimensional element (n is the geometry dimension).
- "h1d", "h1d0", "h1d1", "h1d2", "h1d3" (trailing "xy" or "xyz" allowed) shape functions are elementwise-"h1" shape functions which allow to store fields that are fully discontinuous between elements.

Additionally, types "x", "y" and "z" can be used to define the x, y and z coordinate fields.

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

mesh mymesh("disk.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.

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

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
spanningtree spantree({sur,top});
field a("hcurl", spantree);
field aharmonic("hcurl", {2,3}, spantree);
```

This adds to the constructors above the spanning tree input argument needed when the field has to be gauged (e.g. for the magnetic vector potential formulation of the magnetostatic problem in 3D).

`int field::countcomponents(void)`

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

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

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

```
mesh mymesh("disk.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 field::printharmonics(void)`

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

Print a string showing the harmonics in the field.

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

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

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

`void field::print(void)`

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

Print the field name ("v" here).

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

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

Sets interpolation order 3 on region number 1.
When using different interpolation orders on different physical regions for a given field it is only allowed to set the interpolation orders in a decreasing way, i.e. starting with the physical region with highest order and ending with the physical region with lowest order. This is required to enforce field continuity and is due to the fact that the interpolation order on the interface between multiple physical regions must be the one of the lowest touching region.

`void field::setorder(expression criterion, int loworder, int highorder)`

```
int all = 1, in = 2, out = 3;
int n = 20;
shape q0("quadrangle", all, {0,0,0, 1,0,0, 1,0.3,0, 0,0.3,0}, {n,n,n,n});
shape q1("quadrangle", all, {1,0,0, 2,0,0, 2,0.3,0, 1,0.3,0}, {n,n,n,n});
shape linein = q0.getsons()[3];
linein.setphysicalregion(in);
shape lineout = q1.getsons()[0];
lineout.setphysicalregion(out);
mesh mymesh({q0,q1,linein,lineout});
```

```
field v("h1");
v.setname("v");
v.setorder(all, 1);
v.setorder(norm(grad(v)), 1, 5);
v.setconstraint(in, 1);
v.setconstraint(out);
formulation electrostatics ;
electrostatics += integral(all, 8.854e-12*grad(dof(v))*grad(tf(v)));

for (int i = 0; i < 5; i++)
{
        electrostatics.solve();
        v.write(all, "v"+std::to_string(100+i)+".pos", 5);
        (-grad(v)).write(all, "E"+std::to_string(100+i)+".pos", 5);
        fieldorder(v).write(all, "vorder"+std::to_string(100+i)+".pos", 1);

        adapt(2);
}
```

The field interpolation order will be adapted on each mesh element (of the entire geometry) based on the value of a **positive** criterion (p-adaptivity). The max range of the criterion is split into a number of intervals equal to the number of orders in range 'loworder' to 'highorder'. All intervals have the same size. The barycenter value of the criterion on each element is considered to select the interval, and therefore the corresponding interpolation order to assign to the field on each element. As an example, for a criterion with highest value 900 over the entire domain and a low/high order requested of 1/3 the field on elements with criterion value in range 0 to 300, 300 to 600, 600 to 900 will be assigned order 1, 2, 3 respectively.

`void field::setorder(double targeterror, int loworder, int highorder, double absthres)`

```
int sur = 1;
shape q("quadrangle", sur, {0,0,0, 1,0,0, 1,1,0, 0,1,0}, {20,20,20,20});
mesh mymesh({q});
field v("h1xy"), x("x"), y("y");
v.setorder(sur, 1);
v.setorder(1e-5, 1, 5, 0.001);
for (int i = 0; i < 5; i++)
{
    v.setvalue(sur, array2x1(0, cos(10*x*y)));
    adapt();
```

```
    v.write(sur, "v"+std::to_string(i)+".vtu", 5);
    fieldorder(v).write(sur, "fov"+std::to_string(i)+".vtu", 1);
}
```

The field interpolation order will be adapted on each mesh element (of the entire geometry) based on a criterion measuring the Legendre expansion decay. The target error gives the fraction of the total shape function weight that does not need to be captured. The low order is used on all elements where the total weight is lower than the absolute threshold provided.

```
void field::setport(int physreg, port primal, port dual)

int sur = 1, left = 2, right = 3;
shape q("quadrangle", sur, {0,0,0, 1,0,0, 1,1,0, 0,1,0}, {10,10,10,10});
shape ll = q.getsons()[3];
ll.setphysicalregion(left);
shape rl = q.getsons()[1];
rl.setphysicalregion(right);
mesh mymesh({q, ll, rl});

field v("h1"), y("y");
v.setorder(sur, 2);

// Electric conductivity increasing with the y coordinate:
expression sigma = 0.01*(1+2*y);

// Ground the right electrode:
v.setconstraint(right);

port V, I;
v.setport(left, V, I);
//                |  |
//    primal port  dual port
//
// The dual port holds the global Neumann term on the port region.
// For an electrokinetic formulation this equals the total current.

formulation electrokinetic;

// Set a 1 A current flowing in through the left electrode with port relation I - 1.0 = 0:
electrokinetic += I - 1.0;
```

```
// Define the weak formulation for the DC current flow:
electrokinetic  += integral(sur, -sigma * grad(dof(v)) * grad(tf(v)));

electrokinetic.solve();

v.write(sur, "v.pos", 2);
(-grad(v)*sigma).write(sur, "j.pos", 2);

double resistance = V.getvalue()/I.getvalue();
std::cout << "Resistance is " << resistance << " Ohm" << std::endl;
```

This function associates a primal-dual pair of ports to the field on the requested physical region. As a side effect it lowers the field order on that region to the minimum possible. **Ports have priority over Dirichlet constraints, conditional constraints and gauge conditions**. Defining any of these on a port region has no effect.

Ports that have been associated to a field with a *setport* call and unassociated ports are visible to a formulation only if they appear in a port relation (in the example with *electrokinetic += I - 1.0*). The primal and dual of associated ports are always made visible together even if only one of them appears in a port relation. Unassociated ports are not connected to the weak form terms in the formulation. Associated ports provide the link to the weak form terms: the primal can be used as the lumped field value on the associated region while the dual can be used as the total contribution over that region of the Neumann term in the formulation. The field value is considered constant by the formulation over the region of each associated port visible to it.

To illustrate the meaning of the dual port let us consider the above DC current flow simulation code. The strong form to solve is

$$\nabla \cdot (\sigma \, \nabla v) = 0$$

where σ is the electric conductivity and v the electric potential field. The corresponding weak form is

$$\int_{\Omega} \nabla \cdot (\sigma \, \nabla v) \, v' d\Omega = 0$$

which can be rewritten as

$$-\int_{\Omega} \sigma \, \nabla v \cdot \nabla v' d\Omega + \int_{\partial\Omega} \sigma \, \partial_{\boldsymbol{n}} v \, v' \, d\partial\Omega = 0$$

where $\partial\Omega$ is the boundary of Ω, $\boldsymbol{n}$ is the normal pointing outward from Ω and $\sigma \, \partial_{\boldsymbol{n}} v = \sigma \, \nabla v \cdot \boldsymbol{n} = -\sigma \boldsymbol{E} \cdot \boldsymbol{n} = -\boldsymbol{J} \cdot \boldsymbol{n}$. The Neumann term is the second term of the weak formulation. The dual port I therefore equals the total current flowing through the electrode so that I can be used to impose a total current source condition on the electrode. More details about the associated mathematics can

be found in paper *'Coupling of local and global quantities in various finite element formulations and its application to electrostatics, magnetostatics and magnetodynamics'*, Dular et al.

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

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1");
v.setorder(vol, 1);
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 $\times$ 2 + 2.

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

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1"), u("h1xyz");
v.setorder(vol, 1);
u.setorder(vol, 1);
v.setvalue(vol, u, expression(12));
```

Same as above but here the 'input' expression is computed on a mesh deformed by 'meshdeform'.

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

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1", {2,3}), v2("h1", {1,4,5});
v.setorder(vol, 1);
v2.setorder(vol, 1);
v2.setvalue(vol, 5, v*v);
v2.write(vol, "v2.vtk", 1);
```

This calls an FFT for the calculation (required for nonlinear multiharmonic expressions). The FFT is computed at 'numfftharms' timesteps.

```
void field::setvalue(int physreg, int numfftharms, expression meshdeform,
```

`expression input, int extraintegrationdegree = 0)`

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1", {2,3}), v2("h1", {1,4,5}), u("h1xyz");
v.setorder(vol, 1);
v2.setorder(vol, 1);
u.setorder(vol, 1);
v2.setvalue(vol, 5, u, v*v);
```

This calls an FFT for the calculation and the expression is evaluated on a mesh deformed by 'meshdeform'.

`void field::setvalue(int physreg)`

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

Sets the field value on region 1 to 0.

`void field::setnodalvalues(indexmat nodenumbers, densemat values)`

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1");
v.setorder(vol, 1);
indexmat nodenums(5, 1, {0,1,2,3,4});
densemat nodevals(5, 1, {10,11,12,13,14});
v.setnodalvalues(nodenums, nodevals);
```

Sets the value of a 'h1' type field at a set of nodes.

`densemat field::getnodalvalues(indexmat nodenumbers)`

```
...
densemat outvals = v.getnodalvalues(nodenums);
outvals.print();
```

Gets the value of a 'h1' type field at a set of nodes.

`void field::setconstraint(int physreg, expression input,`

```
int extraintegrationdegree = 0)
```

```
mesh mymesh("disk.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 nonzero 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 $\times$ 2 + 2. **Dirichlet constraints have priority over conditional constraints and gauge conditions**. Defining any of these on a Dirichlet constrained region has no effect.

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

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1"), u("h1xyz");
v.setconstraint(vol, u, expression(12));
```

Same as above but here the 'input' expression is computed on a mesh deformed by 'meshdeform'.

```
void field::setconstraint(int physreg, std::vector<expression> input,
int extraintegrationdegree = 0)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1", {2,3});
v.setconstraint(vol, {1,0});
```

Sets a Dirichlet condition with given value for every field harmonic.

```
void field::setconstraint(int physreg, expression meshdeform,
std::vector<expression> input, int extraintegrationdegree = 0)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1", {2,3}), u("h1xyz");
v.setconstraint(vol, u, {1,0});
```

Same as above but here the 'input' expressions are computed on a mesh deformed by 'meshdeform'.

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

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1", {2,3}), v2("h1", {1,4,5});
v2.setconstraint(vol, 5, v*v);
```

This calls an FFT for the calculation (required for nonlinear multiharmonic expressions). The FFT is computed at 'numfftharms' timesteps.

```
void field::setconstraint(int physreg, int numfftharms, expression meshdeform,
expression input, int extraintegrationdegree = 0)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1", {2,3}), v2("h1", {1,4,5}), u("h1xyz");
v2.setconstraint(vol, 5, u, v*v);
```

This calls an FFT for the calculation and the expression is evaluated on a mesh deformed by 'meshdeform'.

```
void field::setconstraint(int physreg)
```

```
mesh mymesh("disk.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 field::setconditionalconstraint(int physreg, expression condexpr,
expression valexpr)
```

```
mesh mymesh("disk.msh");
int vol = 1, top = 3;
field v("h1"), x("x"), y("y");
v.setorder(vol, 1);
v.setconditionalconstraint(vol, x+y, 12);
formulation form;
form += integral(vol, dof(v)*tf(v) - 1*tf(v));
form.generate();
```

```
vec sol = solve(form.A(), form.b());
v.setdata(vol, sol);
v.write(top, "v.vtk", 1);
```

Forces the field value (i.e. Dirichlet condition) on region 'physreg' to value 'valexpr' for all node-associated degrees of freedom respecting the condition 'condexpr' greater or equal to zero at the nodes. This should only be used for fields with 'h1' like shape functions.

`void field::setgauge(int physreg)`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
spanningtree spantree({sur,top});
field a("hcurl", spantree);
a.setgauge(vol);
```

This sets a gauge condition on region 'vol'. It must be used e.g. for the magnetic vector potential formulation of the magnetostatic problem in 3D since otherwise the algebraic system to solve is singular. It is only defined for edge shape functions ('hcurl'). Its effect is to constrain to zero all degrees of freedom corresponding to:

- gradient type shape functions
- lowest order edge-based shape functions for all edges on the spanning tree provided

---

`void field::setdata(int physreg, vectorfieldselect myvec, std::string op = "set")`

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

w.setdata(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. The (optional) third argument makes it possible

to either *set* the value of field w with the data in the vector (with "set", the default choice) or *add* the data in the vector to field w (with "add").

```
void field::setdata(int physreg, vec myvec, std::string op = "set")
```

...
v.setdata(vol, sol);

This function is equivalent to v.setdata(vol, sol|v).

```
void field::setcohomologysources(std::vector<int> cohomologyphysregs,
std::vector<double> cohomologycoefficients)
```

mesh mymesh("disk.msh");
field v("hcurl");
...
v.setcohomologysources({chreg1, chreg2}, {100, 50});
v.write(chreg1, "v.pos", 1);

This function assigns cohomology coefficients to the field. The field value is reset to zero on the cohomology regions before the coefficients are added on their respective regions.

```
void field::noautomaticupdate(void)
```

...
v.noautomaticupdate();

After this call the field and all its subfields will not have their value automatically updated after hp-adaptivity. If the automatic update is not needed then this call is recommended to avoid a possibly costly field value update.

```
void field::setupdateaccuracy(int extraintegrationorder)
```

...
v.setupdateaccuracy(2);

Allows to tune the integration order in the projection used to update the field value after hp-adaptivity. A positive/negative argument increases/decreases the accuracy but slows down/speeds up the update.

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

mesh mymesh("disk.msh");
field u("h1xyz");
field ux = u.comp(0);
field uy = u.comp(1);

field uz = u.comp(2);

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

`field field::compx(void)`

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

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

`field field::compy(void)`

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

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

`field field::compz(void)`

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

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

`field field::harmonic(int harmonicnumber)`

mesh mymesh("disk.msh");
field u("h1xyz", {1,2,3});
field u2 = u.harmonic(2);

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

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

mesh mymesh("disk.msh");
field u("h1xyz", {1,2,3});
field u23 = u.harmonic({2,3});

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

`field field::sin(int freqindex)`

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

This 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 field::cos(int freqindex)`

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

This 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.

`void field::writeraw(int physreg, std::string filename, bool isbinary = false, std::vector<double> extradata = {})`

```
int vol = 1;
mesh mymesh("disk.msh");
field v("h1xyz"), x("x"), y("y"), z("z");
v.setorder(vol, 2);
v.setvalue(vol, array3x1(x*x,y*y,z*z));
v.writeraw(vol, "v.slz.gz", true);
std::cout << norm(v).max(vol,5)[0] << std::endl;
```

This writes a (possibly multiharmonic) field on a given region to disk in the **compact .slz sparselizard format**. The output format can be ASCII .slz (with 'isbinary' set to false, default option) or binary .slz format. In the latter case the .slz.gz extension can also be used to write to gz compressed .slz format (most compact version). While the binary file is more compact on disk it might be less portable across different platforms than the ASCII version.
The last input argument allows to store extra data (timestep, parameter values,...) that can be loaded back from the 'loadraw' output.

`std::vector<double> field::loadraw(std::string filename, bool isbinary = false)`

```
int vol = 1;
mesh mymesh("disk.msh");
field u("h1xyz");
u.loadraw("v.slz.gz", true);
std::cout << norm(u).max(vol,5)[0] << std::endl;
```

This loads the gz compressed .slz binary file created with the above 'writeraw' function example. After the call the order of field u is set to 2 on region vol. In case 'isbinary' is set to true the filename can point to a remote location (e.g. *"http://www.sparselizard.org/misc/u.slz.gz"*) to **download via ftp or http** the file from a remote server. The vector returned is the extra data vector provided as argument to 'writeraw'.

The **exact same mesh** must be used when loading with 'loadraw' as the one that was used during the corresponding 'writeraw' call.

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

The formulation object holds the port relations and the weak form terms of the problem to solve.

`formulation::formulation(void)`

```
mesh mymesh("disk.msh");
formulation myformulation;
```

This creates an empty formulation object.

`void formulation::operator+=(expression expr)`

```
mesh mymesh("disk.msh");
port A, B;
formulation linearsystem;
linearsystem += A + B - 1.0; // A + B - 1 = 0
linearsystem += B - 5.0; // B - 5 = 0
linearsystem.solve();
std::cout << "Solution is (A, B) = (" << A.getvalue() << ", " << B.getvalue() << ")" << std::endl;
```

This adds a port relation to the formulation. It can be coupled to weak form terms.

`void formulation::operator+=(integration integrationobject)`

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

// Twelve 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 or order of the test function $\times$ 2 + 2 in case there is no unknown. 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 formulation::countdofs(void)`

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

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

`long long int formulation::allcountdofs(void)`

```
...
long long int allnumdofs = projection.allcountdofs();
```

This is a collective MPI operation (must be called by all ranks). It returns on every rank the global number of degrees of freedom defined in the scattered formulation. The count is exact if for each field the number of unknowns associated to each element matches across touching ranks. It is an estimation otherwise.

`void formulation::generate(void)`

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1");
v.setorder(vol, 1);
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 formulation::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) · tf(v)'.

`void formulation::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)) · tf(v)'.

`void formulation::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)) · tf(v)'.

`void formulation::generaterhs(void)`

...
projection.generaterhs();

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

`void formulation::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 formulation::generate(int contributionnumber)`

...
projection.generate(2);

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

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

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1");
v.setorder(vol, 1);
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 formulation::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 set 'keepfragments' to 'true' 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' all the generated data is no more in the formulation, only in matrix A.

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

...

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

Does the same as .b().

`mat formulation::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 formulation::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 formulation::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 formulation::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.

```
void formulation::solve(std::string soltype = "lu", bool diagscaling = false)
```

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

This generates the formulation, solves the algebraic problem Ax = b with a direct solver then saves all data in vector x to the fields defined in the formulation.

3.5 The *indexmat* object (/src/indexmat.h):

The indexmat 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.

`indexmat::indexmat(int numberofrows, int numberofcolumns)`

indexmat B(2,3);

This creates a matrix with 2 rows and 3 columns. Its values may be undefined.

`indexmat::indexmat(int numberofrows, int numberofcolumns, int initvalue)`

indexmat B(2,3, 12);

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

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

indexmat 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.

`indexmat::indexmat(int numberofrows, int numberofcolumns, int init, int step)`

indexmat 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').

`indexmat::indexmat(std::vector<indexmat> input)`

indexmat A(2,3, 0,1);
indexmat B(2,3, 10,1);
indexmat AB({A,B});
AB.print();

This creates a matrix that is the vertical concatenation of matrices.

`int indexmat::countrows(void)`

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

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

`int indexmat::countcolumns(void)`

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

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

`int indexmat::count(void)`

indexmat B(2,3);
int numentries = B.count();

This counts the number of entries in the dense matrix (number of rows x number of columns, 6 here).

`void indexmat::print(void)`

indexmat B(2,3, 0,1);
B.print ();

This prints the matrix values to the console.

`void indexmat::printsize(void)`

indexmat B(2,3);
B. printsize ();

This prints the matrix size to the console.

`int* indexmat::getvalues(void)`

indexmat 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.

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

The mat object holds a sparse algebraic matrix.

`mat::mat(int matsize, indexmat rowaddresses, indexmat coladdresses, densemat vals)`

indexmat rows(7, 1, {0,0,1,1,1,2,2});
indexmat cols(7, 1, {0,1,0,1,2,1,2});
densemat vals(7, 1, {11,12,13,14,15,16,17});

mat A(3, rows, cols, vals);
A.print ();

This creates a matsize×matsize sparse matrix object. The matrix nonzero values are obtained from the input (in format row of value - column of value - value).

`mat::mat(formulation myformulation, indexmat rowaddresses, indexmat coladdresses, densemat vals)`

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

indexmat addresses(projection.countdofs(), 1, 0, 1);
densemat vals(projection.countdofs(), 1, 12);

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

This creates a sparse matrix object whose dof structure is the one in 'projection'. The matrix nonzero 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 mat::countrows(void)`

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

This counts the number of rows in the matrix.

`int mat::countcolumns(void)`

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

This counts the number of columns in the matrix.

`int mat::countnnz(void)`

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

This counts the number of nonzero entries in the matrix with eliminated Dirichlet constraints (-1 if the information is not available).

`void mat::reusefactorization(void)`

...
A. reusefactorization ();

The matrix factorization will be reused in sl::solve.

`indexmat mat::getainds(void)`

...
indexmat ainds = A.getainds();

Let us call *dinds* the set of unknowns that have a Dirichlet constraint and *ainds* the remaining unknowns. The mat object $\boldsymbol{A}$ holds matrices $\boldsymbol{Aa}$ and $\boldsymbol{Ad}$ such that

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{Aa} & \boldsymbol{Ad} \\ \mathbf{0} & \mathbf{1} \end{bmatrix}$$

where $\boldsymbol{Aa}$ is a square matrix equal to $\boldsymbol{A}$ with eliminated Dirichlet constraints, $\mathbf{0}$ is an all zero matrix and $\mathbf{1}$ is the square identity matrix of all Dirichlet constraints. Matrices $\boldsymbol{Aa}$ and $\boldsymbol{Ad}$ are stored with their local indexing. Objects *ainds* and *dinds* give the index in $\boldsymbol{A}$ of each local index in $\boldsymbol{Aa}$ and $\boldsymbol{Ad}$.

`indexmat mat::getdinds(void)`

...
indexmat dinds = A.getdinds();

This outputs *dinds* (see above).

`Mat mat::getapetsc(void)`

...
Mat apetscmat = A.getapetsc();

This outputs the petsc object corresponding to the $\boldsymbol{Aa}$ matrix (see above). It can be used like any other petsc object.

`Mat mat::getdpetsc(void)`

...
Mat dpetscmat = A.getdpetsc();

This outputs the petsc object corresponding to the $\boldsymbol{Ad}$ matrix (see above). It can be used like any other petsc object.

`void mat::print(void)`

...
A.print ();

This prints the matrix size and values.

`mat mat::copy(void)`

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

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

`mat/vec mat::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.

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

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

`mesh::mesh(std::string filename, int verbosity = 1)`
`mesh::mesh(std::string filename, int globalgeometryskin, int numoverlaplayers, int verbosity = 1)`

// Load with the native reader:
mesh mymesh("disk.msh");

// Load with the GMSH API:
mesh mymesh("gmsh:disk.msh");

// Load from the GMSH API:
...
#include "gmsh.h"
...
gmsh:: initialize ();
gmsh::open("disk.msh");
mesh mymesh("gmsh:api");
gmsh:: finalize ();

// Load with petsc:
mesh mymesh("petsc:disk.msh");

This creates the mesh object based on the 'disk.msh' mesh file. See the example code above for the list of tools that can be used to load various mesh formats. Set verbosity to 2 to get information on the physical regions in the mesh.

In the second version of this constructor the mesh is treated as part of a global mesh in the domain decomposition framework (each MPI rank owns a part of the mesh and all ranks must perform the call together). Argument 'globalgeometryskin' is the part of the global mesh skin that belongs to the current rank. It can only hold elements of dimension one lower than the geometry dimension (use -1 if empty). **The global mesh skin cannot intersect itself**. The mesh parts are overlapped by the number of layers requested. More than one overlap layer cannot be guaranteed everywhere because the overlap is limited to the direct neighbouring domains.

```
mesh::mesh(std::vector<shape> inputshapes, int verbosity = 1)
mesh::mesh(std::vector<shape> inputshapes, int globalgeometryskin,
int numoverlaplayers, int verbosity = 1)
```

int faceregionnumber = 1, lineregionnumber = 2;
shape quadface("quadrangle", faceregionnumber, {0,0,0, 1,0,0, 1,1,0, 0,1,0}, {10,6,10,6});
shape leftline = quadface.getsons()[3];
leftline.setphysicalregion(lineregionnumber);
mesh mymesh({quadface, leftline});
mymesh.write("quadmesh.msh");

This creates the mesh object based on the quadrangle shape and its left side line. Set verbosity to 2 to get information on the physical regions in the mesh.

In the second version of this constructor the mesh is treated as part of a global mesh in the domain decomposition framework (each MPI rank owns a part of the mesh and all ranks must perform the call together). Argument 'globalgeometryskin' is the part of the global mesh skin that belongs to the current rank. It can only hold elements of dimension one lower than the geometry dimension (use -1 if empty). **The global mesh skin cannot intersect itself**. The mesh parts are overlapped by the number of layers requested. More than one overlap layer cannot be guaranteed everywhere because the overlap is limited to the direct neighbouring domains.

```
mesh::mesh(bool mergeduplicates, std::vector<std::string> meshfiles,
int verbosity = 1)
```

mesh mymesh("disk.msh", 0);

mymesh.shift(2,1,0);
mymesh.write("shifted.msh", 0);
mesh mergedmesh(true, {"disk.msh","shifted.msh"});
mergedmesh.write("merged.msh", 0);

This combines multiple meshes. Elements shared by the input meshes can be either merged or not. For every input mesh a new physical region containing all elements is created. Set verbosity to 2 to get information on the physical regions in the mesh.

```
void mesh::load(std::string filename, int verbosity = 1)
void mesh::load(std::string filename, int globalgeometryskin, int numoverlaplayers,
int verbosity = 1)
```

mesh mymesh;
mymesh.load("disk.msh");

See corresponding constructor for details.

```
void mesh::load(std::vector<shape> inputshapes, int verbosity = 1)
void mesh::load(std::vector<shape> inputshapes, int globalgeometryskin,
int numoverlaplayers, int verbosity = 1)
```

// See corresponding constructor

See corresponding constructor for details.

```
void mesh::load(bool mergeduplicates, std::vector<std::string> meshfiles,
int verbosity = 1)
```

// See corresponding constructor

See corresponding constructor for details.

```
void mesh::write(int physreg, std::string name)
```

int vol = 1;
mesh mymesh("disk.msh");
mymesh.write(vol, "out.msh");

This writes the part of the mesh in object 'mymesh' that is included in the argument physical region to file 'out.msh'.

```
void mesh::write(std::string name, std::vector<int> physregs = {-1}, int option = 1)
```

```
mesh mymesh("disk.msh");
mymesh.write("disk2.msh");
mymesh.write("disk2.msh", {4}, −1);
```

This writes the mesh in object 'mymesh' to file 'disk2.msh'. If only the first argument is provided then all physical regions are written. If the other arguments are also provided then setting the option to 1 writes only the argument physical regions while setting the option to -1 writes all but the argument physical regions.

```
void mesh::setadaptivity(expression criterion, int lownumsplits, int highnumsplits)
```

```
int all = 1;
shape q("quadrangle", all, {0,0,0, 1,0,0, 1.2,1,0, 0,1,0}, {5,5,5,5});
mesh mymesh({q});
field x("x"), y("y");
expression criterion = 1+sin(10*x)*sin(10*y);

mymesh.setadaptivity(criterion, 0, 5);

for (int i = 0; i < 5; i++)
{
        criterion.write(all, "criterion"+std::to_string(100+i)+".vtk", 1);
        adapt(1);
}
```

Each element in the mesh will be adapted (refined/coarsened) based on the value of a **positive** criterion (h-adaptivity). The max range of the criterion is split into a number of intervals equal to the number of refinement levels in range 'lownumsplits' to 'highnumsplits'. All intervals have the same size. The barycenter value of the criterion on each element is considered to select the interval, and therefore the corresponding refinement of each mesh element. As an example, for a criterion with highest value 900 over the entire domain and a low/high refinement level requested of 1/3 the refinement on mesh elements with criterion value in range 0 to 300, 300 to 600, 600 to 900 will be 1, 2, 3 levels respectively.

```
void mesh::split(int n = 1)
```

```
mesh mymesh;
mymesh.split();
mymesh.load("disk.msh");
mymesh.write("splitdisk.msh");
```

Split each element in the mesh n times. Element quality is maximized and element curvature is taken into account. Each element is split recursively n times as follows:

- point → 1 point
- line → 2 lines
- triangle → 4 triangles
- quadrangle → 4 quadrangles
- tetrahedron → 8 tetrahedra
- hexahedron → 8 hexahedra
- prism → 8 prisms
- pyramid → 6 pyramids + 4 tetrahedra

`void mesh::move(int physreg, expression u)`

```
int vol = 1;
mesh mymesh("disk.msh");
field x("x"), y("y");
mymesh.move(vol, array3x1(0,0,sin(x*y)));
mymesh.write("moved.msh");
```

This moves the mesh in object 'mymesh' on region 'physreg' by the x, y and z component of expression u in the x, y and z direction.

`void mesh::move(expression u)`

```
int vol = 1;
mesh mymesh("disk.msh");
field x("x"), y("y");
mymesh.move(array3x1(0,0,sin(x*y)));
mymesh.write("moved.msh");
```

This moves the whole mesh in object 'mymesh' by the x, y and z component of expression u in the x, y and z direction.

`void mesh::shift(int physreg, double x, double y, double z)`

```
int vol = 1;
mesh mymesh("disk.msh");
mymesh.shift(vol, 1.0, 2.0, 3.0);
```

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

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

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

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

```
void mesh::rotate(int physreg, double ax, double ay, double az)
```

```
int vol = 1;
mesh mymesh("disk.msh");
mymesh.rotate(vol, 20, 60, 90);
```

This rotates the mesh in object 'mymesh' on region 'physreg' first 20 degrees around the x axis then 60 degrees around the y axis then 90 degrees around the z axis.

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

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

This rotates the whole mesh in object 'mymesh' first 20 degrees around the x axis then 60 degrees around the y axis then 90 degrees around the z axis.

```
void mesh::scale(int physreg, double x, double y, double z)
```

```
int vol = 1;
mesh mymesh("disk.msh");
mymesh.scale(vol, 0.1, 0.3, 1.0);
```

This scales the mesh in object 'mymesh' on region 'physreg' by a factor 0.1, 0.3 and 1.0 respectively in the x, y and z direction.

```
void mesh::scale(double x, double y, double z)
```

```
mesh mymesh("disk.msh");
mymesh.scale(0.1, 0.3, 1.0);
```

This scales the whole mesh in object 'mymesh' by a factor 0.1, 0.3 and 1.0 respectively in the x, y and z direction.

```
int mesh::getdimension(void)
```

```
mesh mymesh("disk.msh");
int dim = mymesh.getdimension();
```

This returns the dimension of the highest dimension element in the mesh (0D, 1D, 2D or 3D).

`std::vector<double> mesh::getdimensions(void)`

```
mesh mymesh("disk.msh");
std::vector<double> dims = mymesh.getdimensions();
```

This returns the x, y and z mesh dimensions in meters.

`std::vector<double> mesh::printdimensions(void)`

```
mesh mymesh("disk.msh");
mymesh.printdimensions();
```

This prints and returns the x, y and z mesh dimensions.

`std::vector<int> mesh::getphysicalregionnumbers(int dim = -1)`

```
mesh mymesh("disk.msh");
std::vector<int> allphysregs = mymesh.getphysicalregionnumbers();
```

This returns all physical region numbers of a given dimension. Use -1 or no argument to get the regions for all dimensions.

`void mesh::selectskin(int newphysreg, int physregtoselectfrom)`

```
int vol = 1, skin = 12;
mesh mymesh;
mymesh.selectskin(skin, vol);
mymesh.load("disk.msh");
field v("h1"), x("x"), y("y"), z("z");
v.setorder(vol, 1);
v.setvalue(vol, x*y*z);
v.write(vol, "v.vtk", 1);
v.write(skin, "vskin.vtk", 1);
```

This tells the mesh object to create physical region number 12 that contains the elements in the skin of physical region number 1. The region is created when the *.load* function is called on the mesh object. Because region 1 is a volume the skin region contains surface elements. This can be advantageously used to write the field to a small-sized file on disk. Note that space derivatives or 'hcurl' field evaluations on a surface do not usually lead to the same values as a volume evaluation.

`void mesh::selectskin(int newphysreg)`
Same as the previous function but 'physregtoselectfrom' is the whole domain.

`void mesh::selectbox(int newphysreg, int physregtoselectfrom, int selecteddim, std::vector<double> boxlimit)`

```
int vol = 1, boxregion = 12;
mesh mymesh;
mymesh.selectbox(boxregion, vol, 3, {0,1,0,1,0,0.1});
mymesh.load("disk.msh");
field v("h1"), x("x"), y("y"), z("z");
v.setorder(vol, 1);
v.setvalue(vol, x*y*z);
v.write(vol, "v.vtk", 1);
v.write(boxregion, "vboxregion.vtk", 1);
```

This tells the mesh object to create physical region number 12 that contains the elements of physical region number 1 that are in the box delimited by $\{x_1, x_2, y_1, y_2, z_1, z_2\}$. The region is created when the *.load* function is called on the mesh object. The new region is populated with the elements of region 1 that are of dimension 'selecteddim' (points, lines, faces or volumes).

`void mesh::selectbox(int newphysreg, int selecteddim, std::vector<double> boxlimit)`
Same as the previous function but 'physregtoselectfrom' is the whole domain.

`void mesh::selectsphere(int newphysreg, int physregtoselectfrom, int selecteddim, std::vector<double> centercoords, double radius)`

```
int vol = 1, sphereregion = 12;
mesh mymesh;
mymesh.selectsphere(sphereregion, vol, 3, {1.0,0.0,0.0}, 1);
mymesh.load("disk.msh");
field v("h1"), x("x"), y("y"), z("z");
v.setorder(vol, 1);
v.setvalue(vol, x*y*z);
v.write(vol, "v.vtk", 1);
v.write(sphereregion, "vsphereregion.vtk", 1);
```

This tells the mesh object to create physical region number 12 that contains the elements of physical region number 1 that are in the sphere of center $\{x_c, y_c, z_c\}$ and of prescribed radius. The region is created when the *.load* function is called on the mesh object. The new region is populated with the elements of region 1 that are of dimension 'selecteddim' (points, lines, faces or volumes).

`void mesh::selectsphere(int newphysreg, int selecteddim,`
`std::vector<double> centercoords, double radius)`
Same as the previous function but 'physregtoselectfrom' is the whole domain.

`void mesh::selectlayer(int newphysreg, int physregtoselectfrom,`
`int physregtostartgrowth, int numlayers)`

int vol = 1, sur = 2, top = 3, layerregion = 12;
mesh mymesh;
mymesh.selectlayer(layerregion, vol, sur, 1);
mymesh.load("disk.msh");
mymesh.write("out.msh");

This tells the mesh object to create physical region number 12 that contains the layer of elements of physical region number 1 that touches physical region number 2. When multiple layers are requested they are grown on top of each other. The region is created when the *.load* function is called on the mesh object.

`void mesh::selectlayer(int newphysreg, int physregtostartgrowth, int numlayers)`
Same as the previous function but 'physregtoselectfrom' is the whole domain.

`void mesh::selectexclusion(int newphysreg, int physregtoexcludefrom,`
`std::vector<int> physregstoexclude)`

int vol = 1, sur = 2, top = 3, excluded = 12;
mesh mymesh;
mymesh.selectbox(11, vol, 3, {0,2, −2,2, −2,2});
mymesh.selectexclusion(excluded, vol, {11});
mymesh.load("disk.msh");
mymesh.write("out.msh");

This tells the mesh object to create physical region number 12 that contains the elements in the physical region 1 that are not in physical region 11. The region is created when the *.load* function is called on the mesh object.

`void mesh::selectexclusion(int newphysreg, std::vector<int> physregstoexclude)`
Same as the previous function but 'physregtoexcludefrom' is the whole domain.

`void mesh::selectanynode(int newphysreg, int physregtoselectfrom)`

int vol = 1, anynode = 12;

```
mesh mymesh;
mymesh.selectanynode(anynode, vol);
mymesh.load("disk.msh");
mymesh.write("out.msh");
```

This tells the mesh object to create physical region number 12 that contains a single node arbitrarily chosen in physical region 1. The region is created when the *.load* function is called on the mesh object.

`void mesh::selectanynode(int newphysreg)`

Same as the previous function but 'physregtoselectfrom' is the whole domain.

`void mesh::use(void)`

```
mesh finemesh;
finemesh. split (2);
finemesh.load("disk.msh");
mesh coarsemesh("disk.msh");
finemesh.use();
```

This allows to select which mesh to use in case multiple meshes are available. This call invalidates all objects that are based on the previously selected mesh for as long as the latter is not selected again.

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

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

`parameter::parameter(void)`

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

This creates parameter E (not yet defined).

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

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

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

`int parameter::countrows(void)`

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

This returns the number of rows in the parameter.

`int parameter::countcolumns(void)`

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

This returns the number of columns in the parameter.

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

```
mesh mymesh("disk.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 "disk.msh" the parameter is also defined on these two surface regions.

`void parameter::setvalue(int physreg, expression input)`

```
mesh mymesh("disk.msh");
int vol = 1;
parameter E;
E.setvalue(vol, 150e9);
```

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

`void parameter::print(void)`

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

This prints information on the parameter.

3.9 The *port* object (/src/expression/port.h):

The port object represents a scalar lumped quantity.

`port::port(void)`

port V;

This creates port V with initial zero value.

`port::port(std::vector<int> harmonicnumbers)`

port V({2,3});

This creates a multiharmonic port V with initial zero value. Refer to the multiharmonic field constructor for the meaning of the harmonic numbers.

`void port::setvalue(double portval)`

port V({2,3});
V.harmonic(2).setvalue(1);
V.harmonic(3).setvalue(0.5);

This sets the value of both harmonics in port V.

`double port::getvalue(void)`

port V;
V.setvalue(−1.2);
double val = V.getvalue();

This gets the value of port V.

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

port V;
V.setname("V");
V.print();

This sets the name of port V.

`std::string port::getname(void)`

port V;
V.setname("V");
std::cout << "Port name is " << V.getname() << std::endl;

This gets the name of port V.

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

port V({1,2,3});
std::vector<int> harms = V.getharmonics();

This returns the harmonics of port V ({1,2,3} here).

`port port::harmonic(int harmonicnumber)`

port V({1,2,3});
port V3 = V.harmonic(3);

This returns a port that is the harmonic 3 of port V.

`port port::harmonic(std::vector<int> harmonicnumbers)`

port V({1,2,3});
port V23 = V.harmonic({2,3});

This returns a port that includes the harmonics 2 and 3 of port V.

`port port::sin(int freqindex)`

port V ({1,2,3,4,5});
port Vs = V.sin(2);

This gets a port that is the sin harmonic at 2 times the fundamental frequency in port V, i.e. it is harmonic 4.

`port port::cos(int freqindex)`

port V ({1,2,3,4,5});
port Vc = V.cos(0);

This gets a port that is the cos harmonic at 0 times the fundamental frequency in port V, i.e. it is harmonic 1.

`void port::print(void)`

port V({1,2,3});
V.setname("V");
V.print ();

This prints the information of port V.

3.10 The *resolution* objects (/src/resolution):

The resolution objects provide high level tools for convenient solving of classical problems.

3.10.1 The *eigenvalue* object (/src/resolution/eigenvalue.h):

The eigenvalue object allows to solve classical, generalized and polynomial eigenvalue problems. The computation is done by SLEPc, a scalable library for eigenvalue problem computation.

`eigenvalue::eigenvalue(mat A)`
This defines a classical eigenvalue problem $Ax = \lambda x$.

`eigenvalue::eigenvalue(mat A, mat B)`
This defines a generalized eigenvalue problem $Ax = \lambda Bx$.
Undamped mechanical resonance modes and resonance frequencies can be calculated with this since an **undamped mechanical problem** can be written in the form

$$M\ddot{x} + Kx = 0$$

which for a harmonic excitation at angular frequency ω can be rewritten as

$$Kx = \omega^2 Mx$$

so that the generalized eigenvalue λ is equal to ω^2.
To visualize the resonance frequencies of all calculated undamped modes the function *printeigenfrequencies()* can be called.

`eigenvalue::eigenvalue(mat K, mat C, mat M)`
This defines a second order polynomial eigenvalue problem $(M\lambda^2 + C\lambda + K)x = 0$ which allows to get the resonance modes and resonance frequencies for **damped mechanical problems**. The input arguments are respectively the mechanical stiffness, damping and mass matrix.
A second order polynomial eigenvalue problem attempts to find a solution of the form

$$x(t) = ue^{\lambda t},\ \lambda = \alpha + \beta i = -\zeta\omega - i\omega\sqrt{1-\zeta^2}$$

which corresponds to a damped oscillation at frequency $f_{damped} = \frac{\beta}{2\pi}$ with a damping ratio

$$\zeta = \frac{-\alpha}{\sqrt{\alpha^2 + \beta^2}}$$

In case of proportional damping (if and only if $KM^{-1}C$ is symmetric) the oscillation of the undamped system is at ω. The undamped oscillation frequency can then be calculated as

$$f_{undamped} = \frac{\sqrt{\alpha^2 + \beta^2}}{2\pi}$$

To visualize all relevant resonance information for the computed eigenvalues the function *printeigenfrequencies()* can be called.

`eigenvalue::eigenvalue(std::vector<mat> inmats)`
This defines an arbitrary order polynomial eigenvalue problem
$(inmats[0] + inmats[1]\lambda + inmats[2]\lambda^2 + inmats[3]\lambda^3 + ...)x = 0$.

`void eigenvalue::compute(int numeigenvaluestocompute,`
`double targeteigenvaluemagnitude = 0.0)`
This attempts to compute the first *numeigenvaluestocompute* eigenvalues whose magnitude is closest to a target magnitude (0.0 by default). There is no guarantee that SLEPc will return the exact number of eigenvalues requested.

`int eigenvalue::count(void)`
Get the number of eigenvalues found by SLEPc.

`std::vector<double> eigenvalue::geteigenvaluerealpart(void)`
Get the real part of all eigenvalues found.

`std::vector<double> eigenvalue::geteigenvalueimaginarypart(void)`
Get the imaginary part of all eigenvalues found.

`std::vector<vec> eigenvalue::geteigenvectorrealpart(void)`
Get the real part of all eigenvectors found.

`std::vector<vec> eigenvalue::geteigenvectorimaginarypart(void)`
Get the imaginary part of all eigenvectors found.

`void eigenvalue::printeigenvalues(void)`
Print the eigenvalues found.

`void eigenvalue::printeigenfrequencies(void)`
This function provides a convenient way to print the eigenfrequencies associated to all eigenvalues calculated for a mechanical resonance problem. In case a generalized eigenvalue problem is used to calculate the resonance modes of an undamped mechanical problem this function displays the resonance frequency of each calculated resonance mode. In case a second order polynomial eigenvalue problem is used to calculate the resonance modes of a damped mechanical problem this function displays not only the **damped resonance frequency** of each resonance mode but also the **undamped resonance**

frequency (only valid in case of proportional damping), the **bandwidth**, the **damping ratio** and the **quality factor**.

3.10.2 The *genalpha* object (/src/resolution/genalpha.h):

The genalpha object allows to perform a generalized alpha time resolution for a problem of the form $M\ddot{x}+C\dot{x}+Kx = b$, be it linear or nonlinear. The solutions for x as well as $\dot{x}$ and $\ddot{x}$ are made available. For nonlinear problems a fixed-point iteration is performed at every timestep, for as long as the L2 norm of the relative solution vector change is larger than the prescribed tolerance.
The generalized alpha method comes with four parameters (β, γ, α_f and α_m) that can be tuned to adjust the properties of the time resolution method (convergence order, stability, high frequency damping,...). When both α parameters are set to zero a classical Newmark iteration is obtained. By default the parameters are set to (0.25, 0.5, 0.0, 0.0), which corresponds to an unconditionally stable Newmark iteration.
A convenient way proposed below to set the four parameters is to specify a high-frequency dissipation level and let the four parameters be deduced accordingly. This gives a set of parameters leading to an unconditionally stable, second-order accurate algorithm possessing an optimal combination of high-frequency and low-frequency dissipation.
More information on the generalized alpha method can be found in paper "A time integration algorithm for structural dynamics with improved numerical dissipation: the generalized-alpha method".

`genalpha::genalpha(formulation formul, vec dtxinit, vec dtdtxinit, int verbosity = 3, std::vector<bool> isrhskcmconstant = {false, false, false, false})`
This defines the genalpha object to solve in time formulation *formul* with the fields state as initial solution x, with *dtxinit* for $\dot{x}$ and *dtdtxinit* for $\ddot{x}$.
Set *isrhskcmconstant[i]* to true and the corresponding matrix/vector is supposed constant in time and will only be generated once then reused (if K, C and M are all constant the factorization of the algebraic problem is also reused). In *isrhskcmconstant[i]*:

- $i = 0$ corresponds to the rhs vector
- $i = 1$ corresponds to the K matrix
- $i = 2$ corresponds to the C matrix
- $i = 3$ corresponds to the M matrix

Note: even if the rhs vector can be reused the Dirichlet constraints will nevertheless be recomputed at each time step.

`void genalpha::setverbosity(int verbosity)`
This sets the verbosity level.

`void genalpha::setparameter(double b, double g, double af, double am)`
This sets the four parameters of the generalized alpha method.

`void genalpha::setparameter(double rinf)`
This specifies a high-frequency dissipation level ρ_∞ in range

$$0 \leq \rho_\infty \leq 1$$

and lets the four generalized alpha parameters be optimally deduced. The deduced parameters lead to an unconditionally stable, second-order accurate algorithm possessing an optimal combination of high-frequency and low-frequency dissipation. Lower ρ_∞ values lead to more dissipation.

`void genalpha::settolerance(double nltol)`
This sets the tolerance for the fixed-point nonlinear iteration performed at every timestep for nonlinear problems. If the tolerance is not set it is 10^{-3} by default.

`void genalpha::setrelaxationfactor(double relaxfact)`
This sets the relaxation factor (default is 1.0) for the fixed-point nonlinear iteration performed at every timestep for nonlinear problems. The new solution x_{new} is updated as

$$x_{new} := \eta\, x_{new} + (1 - \eta)\, x_{old}$$

where η is the relaxation factor.

`std::vector<vec> genalpha::gettimederivative(void)`
This returns in output[0] the current solution for $\dot{x}$ and in output[1] the current solution for $\ddot{x}$.

`void genalpha::settimederivative(std::vector<vec> sol)`
This sets the current solution for $\dot{x}$ to sol[0] and $\ddot{x}$ to sol[1].

`void genalpha::settimestep(double timestep)`
This sets the current timestep.

`double genalpha::gettimestep(void)`
This returns the current timestep.

`int genalpha::count(void)`
This counts the total number of steps computed.

`std::vector<double> genalpha::gettimes(void)`
This returns all time values stepped through.

`void genalpha::setadaptivity(double tol, double mints, double maxts,`
`double reffact = 0.5, double coarfact = 2.0, double coarthres = 0.5)`
This sets the settings for automatic time-adaptivity. The timestep Δt will be adjusted between *mints* and *maxts* to reach the requested relative error *tol*. The relative error is defined as

$$\Delta t \cdot \frac{||\dot{x}_{n+1} - \dot{x}_n||_2}{||x_{n+1}||_2}$$

to measure the relative deviation from a constant time derivative.
Argument *reffact*/*coarfact* gives the factor to use when the timestep is refined/coarsened. The timestep is refined when the relative error is above *tol* or when the max number of nonlinear iterations is reached. The timestep is coarsened when the relative error is below $coarthres \cdot tol$ and the nonlinear loop has converged in less than the max number of nonlinear iterations.

`void genalpha::presolve(std::vector<formulation> formuls)`
This defines the set of formulations that must be solved *before* every resolution of the formulation provided to the genalpha constructor. The formulations provided here must lead to a system of the form $Ax = b$ (no damping or mass matrix allowed).

`void genalpha::postsolve(std::vector<formulation> formuls)`
This defines the set of formulations that must be solved *after* every resolution of the formulation provided to the genalpha constructor. The formulations provided here must lead to a system of the form $Ax = b$ (no damping or mass matrix allowed).

`void genalpha::next(double timestep)`
This runs the generalized alpha algorithm for one timestep (**use -1 for automatic time-adaptivity**). This call is for linear problems since a single iteration of the inner nonlinear loop is performed. After the call the time and the field values are updated.

`int genalpha::next(double timestep, int maxnumnlit)`
This runs the generalized alpha algorithm for one timestep (use -1 for automatic time-adaptivity). This call is for nonlinear problems. A nonlinear fixed-point iteration is performed for at most *maxnumnlit* iterations (use -1 for unlimited). After the call the time and the field values are updated. The returned value is the number of nonlinear iterations performed.

3.10.3 The *impliciteuler* object (/src/resolution/impliciteuler.h):

The impliciteuler object allows to perform an implicit (backward) Euler time resolution for a problem of the form $C\dot{x} + Kx = b$, be it linear or nonlinear. The solutions for x as well as $\dot{x}$ are made available. For nonlinear problems a fixed-point iteration is performed at every timestep, for as long as the L2 norm of the relative solution vector change is larger than the prescribed tolerance.

`impliciteuler::impliciteuler(formulation formul, vec dtxinit, int verbosity = 3, std::vector<bool> isrhskcconstant = {false, false, false})`
This defines the impliciteuler object to solve in time formulation *formul* with the fields state as initial solution x and *dtxinit* for $\dot{x}$.
Set *isrhskcconstant[i]* to true and the corresponding matrix/vector is supposed constant in time and will only be generated once then reused (if K and C are constant the factorization of the algebraic problem is also reused). In *isrhskcconstant[i]*:

- $i = 0$ corresponds to the rhs vector
- $i = 1$ corresponds to the K matrix
- $i = 2$ corresponds to the C matrix

Note: even if the rhs vector can be reused the Dirichlet constraints will nevertheless be recomputed at each time step.

`void impliciteuler::setverbosity(int verbosity)`
This sets the verbosity level.

`void impliciteuler::settolerance(double nltol)`
This sets the tolerance for the fixed-point nonlinear iteration performed at every timestep for nonlinear problems. If the tolerance is not set it is 10^{-3} by default.

`void impliciteuler::setrelaxationfactor(double relaxfact)`
This sets the relaxation factor (default is 1.0) for the fixed-point nonlinear iteration performed at every timestep for nonlinear problems. The new solution x_{new} is updated as

$$x_{new} := \eta\, x_{new} + (1 - \eta)\, x_{old}$$

where η is the relaxation factor.

`vec impliciteuler::gettimederivative(void)`
This returns the current solution for $\dot{x}$.

`void impliciteuler::settimederivative(vec sol)`
This sets the current solution for $\dot{x}$ to sol.

`void impliciteuler::settimestep(double timestep)`
This sets the current timestep.

`double impliciteuler::gettimestep(void)`
This returns the current timestep.

`int impliciteuler::count(void)`
This counts the total number of steps computed.

`std::vector<double> impliciteuler::gettimes(void)`
This returns all time values stepped through.

`void impliciteuler::setadaptivity(double tol, double mints, double maxts,`
`double reffact = 0.5, double coarfact = 2.0, double coarthres = 0.5)`
This sets the settings for automatic time-adaptivity. The timestep Δt will be adjusted between *mints* and *maxts* to reach the requested relative error *tol*. The relative error is defined as

$$\Delta t \cdot \frac{||\dot{x}_{n+1} - \dot{x}_n||_2}{||x_{n+1}||_2}$$

to measure the relative deviation from a constant time derivative.
Argument *reffact*/*coarfact* gives the factor to use when the timestep is refined/coarsened. The timestep is refined when the relative error is above *tol* or when the max number of nonlinear iterations is reached. The timestep is coarsened when the relative error is below $coarthres \cdot tol$ and the nonlinear loop has converged in less than the max number of nonlinear iterations.

`void impliciteuler::presolve(std::vector<formulation> formuls)`
This defines the set of formulations that must be solved *before* every resolution of the formulation provided to the impliciteuler constructor. The formulations provided here must lead to a system of the form $Ax = b$ (no damping or mass matrix allowed).

`void impliciteuler::postsolve(std::vector<formulation> formuls)`
This defines the set of formulations that must be solved *after* every resolution of the formulation provided to the impliciteuler constructor. The formulations provided here must lead to a system of the form $Ax = b$ (no damping or mass matrix allowed).

`void impliciteuler::next(double timestep)`

This runs the implicit Euler algorithm for one timestep (**use -1 for automatic time-adaptivity**). This call is for linear problems since a single iteration of the inner nonlinear loop is performed. After the call the time and the field values are updated.

`int impliciteuler::next(double timestep, int maxnumnlit)`
This runs the implicit Euler algorithm for one timestep (use -1 for automatic time-adaptivity). This call is for nonlinear problems. A nonlinear fixed-point iteration is performed for at most *maxnumnlit* iterations (use -1 for unlimited). After the call the time and the field values are updated. The returned value is the number of nonlinear iterations performed.

3.11 The *shape* object (/src/geometry/shape.h):

The shape objects are meshed geometric entities. The mesh created based on shapes can be written in .msh format (check the 'mesh' object for that) at any time for visualization in GMSH (it might be needed to change the 'color' and 'visibility' options in the menu 'all mesh options' of GMSH).

`shape::shape(void)`

shape myshape;

This creates an empty shape object.

`shape::shape(std::string shapename, int physreg, std::vector<double> coords)`

int pointphysicalregion = 1, linephysicalregion = 2;
shape mypoint("point", pointphysicalregion, {1.2, 1.5, −0.2});
shape myline("line", linephysicalregion, {0,0,0, 0.5,0.5,0, 1,1,0, 1.5,1,0, 2,1,0});
mesh mymesh({myline, mypoint});
mymesh.write("meshed.msh");

This constructor can be used to create:

- a point at given x, y, z coordinates (1.2,1.5,-0,2)
- a line going through a list of nodes (0,0,0), (0.5,0.5,0),... whose x, y, z coordinates are provided

A physical region number is also provided to have access to the geometric regions of interest in the finite element simulation.

`shape::shape(std::string shapename, int physreg, std::vector<double> coords, int nummeshpts)`

int linephysicalregion = 1, arcphysicalregion = 1;
shape myline("line", linephysicalregion, {0,0,0, 1,−1,1}, 10);

shape myarc("arc", arcphysicalregion , {1,0,0, 0,1,0, 0,0,0}, 8);
mesh mymesh({myline, myarc});
mymesh.write("meshed.msh");

This constructor can be used to create:

- a straight line between the first (0,0,0) and last point (1,-1,1) provided
- a circle arc between the first point (1,0,0) and the second point (0,1,0) whose center is the third point (0,0,0)

The 'nummeshpts' argument corresponds to the number of nodes in the meshed shape.

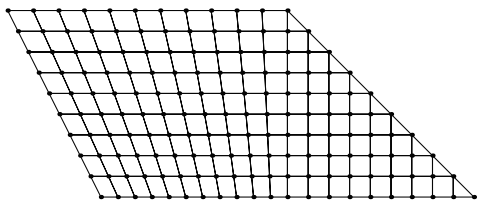
`shape::shape(std::string shapename, int physreg, std::vector<double> coords, std::vector<int> nummeshpts)`

int quadranglephysicalregion = 1, trianglephysicalregion = 2;
shape myquadrangle("quadrangle", quadranglephysicalregion, {0,0,0, 1,0,0, 1,1,0, −0.5,1,0}, {12,10,12,10});
shape mytriangle("triangle ", trianglephysicalregion , {1,0,0, 2,0,0, 1,1,0}, {10,10,10});
mesh mymesh({myquadrangle, mytriangle});
mymesh.write("meshed.msh");

This constructor can be used to create:

- a straight-edge quadrangle with a full quadrangle structured mesh
- a straight-edge triangle with a structured mesh made of triangles along the edge linking the second and third node and quadrangles everywhere else

The 'coords' argument provides the x, y, z coordinates of the corner nodes, e.g. (0,0,0), (1,0,0), (1,1,0) and (-0.5,1,0) for the quadrangle. The 'nummeshpts' argument gives the number of nodes to mesh each of the edges. All edges must have the same number of nodes for the triangle shape while for the quadrangle shape the edges facing each other must have the same number of nodes. The mesh created by the above shapes is displayed below for illustration.



`shape::shape(std::string shapename, int physreg, std::vector<shape> subshapes, int nummeshpts)`

```
int linephysicalregion = 1, arcphysicalregion = 1;
shape point1("point", -1, {0,0,0});
shape point2("point", -1, {1,0,0});
shape point3("point", -1, {0,1,0});
shape point4("point", -1, {1,-1,1});
shape myline("line", linephysicalregion, {point1, point4}, 10);
shape myarc("arc", arcphysicalregion, {point2, point3, point1}, 8);
mesh mymesh({myline, myarc});
mymesh.write("meshed.msh");
```

This constructor can be used to create:

- a straight line between the first (0,0,0) and last point (1,-1,1) provided
- a circle arc between the first point (1,0,0) and the second point (0,1,0) whose center is the third point (0,0,0)

The 'nummeshpts' argument corresponds to the number of nodes in the meshed shape.

`shape::shape(std::string shapename, int physreg, std::vector<shape> subshapes, std::vector<int> nummeshpts)`

```
int quadranglephysicalregion = 1, trianglephysicalregion = 2;
shape point1("point", -1, {0,0,0});
shape point2("point", -1, {1,0,0});
shape point3("point", -1, {1,1,0});
shape point4("point", -1, {0,1,0});
shape point5("point", -1, {2,0,0});
shape myquadrangle("quadrangle", quadranglephysicalregion, {point1, point2, point3, point4}, {6,8,6,8});
shape mytriangle("triangle", trianglephysicalregion, {point2, point5, point3}, {8,8,8});
mesh mymesh({myquadrangle, mytriangle});
mymesh.write("meshed.msh");
```

This constructor can be used to create:

- a straight-edge quadrangle with a full quadrangle structured mesh
- a straight-edge triangle with a structured mesh made of triangles along the edge linking the second and third node and quadrangles everywhere else

The 'subshapes' argument provides the corner point shapes. The 'nummeshpts' argument gives the number of nodes to mesh each of the contour lines. All edges must have the same number of nodes for the triangle shape while for the quadrangle shape the edges facing each other must have the same number of nodes.

`shape::shape(std::string shapename, int physreg, std::vector<shape> subshapes)`

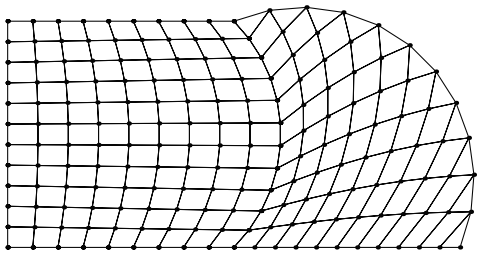
```
int quadranglephysicalregion = 1, trianglephysicalregion = 2, unionphysicalregion = 3;
shape line1("line", -1, {-1,-1,0, 1,-1,0}, 10);
shape arc2("arc", -1, {1,-1,0, 1,1,0, 0,0,0}, 12);
shape line3("line", -1, {1,1,0, -1,1,0}, 10);
shape line4("line", -1, {-1,1,0, -1,-1,0}, 12);
shape line5("line", -1, {1,-1,0, 3,-1,0}, 12);
shape arc6("arc", -1, {3,-1,0, 1,1,0, 1.6,-0.4,0}, 12);
shape myquadrangle("quadrangle", quadranglephysicalregion, {line1, arc2, line3, line4});
shape mytriangle("triangle", trianglephysicalregion, {line5, arc6, arc2});
shape myunion("union", unionphysicalregion, {line1, arc2, line3, line4});

mesh mymesh({myquadrangle, mytriangle, myunion});
mymesh.write("meshed.msh");
```

This constructor can be used to create:

- a curved quadrangle with a full quadrangle structured mesh
- a curved triangle with a structured mesh made of triangles along the edge linking the second and third node and quadrangles everywhere else
- a shape that is the union of several shapes of the same dimension

The 'subshapes' argument provides the contour shapes (clockwise or anti-clockwise). All edges must have the same number of nodes for the triangle shape while for the quadrangle shape the edges facing each other must have the same number of nodes. The mesh created by the above shapes is displayed below for illustration.



`shape::shape(std::string shapename, int physreg, std::vector<double> centercoords, double radius, int nummeshpts)`

```
int diskphysicalregion = 1;
shape mydisk("disk", diskphysicalregion, {1,0,0}, 2, 40);
mesh mymesh({mydisk});
mymesh.write("meshed.msh");
```

This constructor creates a radius 2 disk (with a structured mesh) centered around coordinates (1,0,0). The 'nummeshpts' argument corresponds to the number of nodes in the contour circle of the disk. Because the disk has a structured mesh the number of mesh nodes must be a multiple of 4.

`shape::shape(std::string shapename, int physreg, shape centerpoint, double radius, int nummeshpts)`

```
int diskphysicalregion = 1;
shape centerpoint("point", -1, {1,0,0});
shape mydisk("disk", diskphysicalregion, centerpoint, 2, 40);
mesh mymesh({mydisk});
mymesh.write("meshed.msh");
```

This constructor creates a radius 2 disk (with a structured mesh) centered around point 'centerpoint'. The 'nummeshpts' argument corresponds to the number of nodes in the contour circle of the disk. Because the disk has a structured mesh the number of mesh nodes must be a multiple of 4.

`void shape::setphysicalregion(int physreg)`

```
int quadphysicalregion = 1, linephysicalregion = 2;
shape myquadrangle("quadrangle", quadphysicalregion, {0,0,0, 1,0,0, 1,1,0, 0,1,0}, {6,8,6,8});
shape myline = myquadrangle.getsons()[0];
myline.setphysicalregion( linephysicalregion );
mesh mymesh({myquadrangle, myline});
mymesh.write("meshed.msh");
```

This sets the physical region number for a given shape. Subshapes are not affected. The physical region number is used in the finite element simulation to identify a region.

`int shape::getcurvatureorder(void)`

```
shape q("quadrangle", 1, {0,0,0, 1,0,0, 1,1,0, 0,1,0}, {2,2,2,2});
int co = q.getcurvatureorder();
```

This gets the curvature order of a given shape.

`void shape::move(expression u)`

```
field x("x"), y("y"), z("z");
```

shape myquadrangle("quadrangle", 1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {12,16,12,16});
myquadrangle.move(array3x1(0,x,sin(x∗y)));
mesh mymesh({myquadrangle});
mymesh.write("meshed.msh");

This moves the shape (and all its subshapes recursively) in the x, y and z direction by a value provided as an expression that can include the x, y and z coordinate fields. When moving multiple shapes that share common subshapes make sure the subshapes are not moved multiple times.

`void shape::shift(double shiftx, double shifty, double shiftz)`

shape myquadrangle("quadrangle", 1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {6,8,6,8});
myquadrangle.shift(1,1,2);
mesh mymesh({myquadrangle});
mymesh.write("meshed.msh");

This shifts the shape (and all its subshapes recursively) in the x, y and z direction by a double value. When shifting multiple shapes that share common subshapes make sure the subshapes are not shifted multiple times.

`void shape::rotate(double alphax, double alphay, double alphaz)`

shape myquadrangle("quadrangle", 1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {6,8,6,8});
myquadrangle.rotate(0,0,45);
mesh mymesh({myquadrangle});
mymesh.write("meshed.msh");

This rotates the shape (and all its subshapes recursively) first 'alphax' degrees around the x axis then 'alphay' degrees around the y axis then 'alphaz' degrees around the z axis. When rotating multiple shapes that share common subshapes make sure the subshapes are not rotated multiple times.

`void shape::scale(double scalex, double scaley, double scalez)`

shape myquadrangle("quadrangle", 1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {6,8,6,8});
myquadrangle.scale(2,0.5,1);
mesh mymesh({myquadrangle});
mymesh.write("meshed.msh");

This scales the shape (and all its subshapes recursively) in the x, y and z direction by a given factor (1 keeps it unchanged). When scaling multiple shapes that share common subshapes make sure the subshapes are not scaled multiple times.

`shape shape::extrude(int physreg, double height, int numlayers,`
`std::vector<double> extrudedirection = {0,0,1})`

int volumephysicalregion = 1;
shape myquadrangle("quadrangle", −1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {6,8,6,8});
shape myvolume = myquadrangle.extrude(volumephysicalregion, 1.4, 6);
mesh mymesh({myvolume});
mymesh.write("meshed.msh");

This outputs a shape of higher dimension that is the extrusion in the requested direction (z by default) of the initial shape. There are 'numlayers' node layers in the extruded mesh. The physical region number of the extruded shape is set to 'physreg'. The extrude function works for 0D, 1D and 2D shapes.

`std::vector<shape> shape::extrude(std::vector<int> physreg, std::vector<double> height, std::vector<int> numlayers, std::vector<double> extrudedirection = {0,0,1})`

shape mytri("triangle", 1, {0,0,0, 1,0,0, 0,1,0}, {6,6,6});
std::vector<shape> extruded = mytri.extrude({11,12}, {0.5,0.3}, {3,5});
mesh mymesh(extruded);
mymesh.write("extruded.msh");

This extends the previous function to multilayer extrusions.

`shape shape::duplicate(void)`

shape myquadrangle("quadrangle", 1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {6,8,6,8});
shape otherquadrangle = myquadrangle.duplicate();

This outputs a shape that is the duplicate of the initial shape. All subshapes are duplicated recursively as well but the object equality relations between subshapes are identical between a shape and its duplicate.

`int shape::getdimension(void)`

shape myquadrangle("quadrangle", 1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {6,8,6,8});
int dim = myquadrangle.getdimension();
std::cout << dim << std::endl;

This gives the shape dimension (0D, 1D, 2D or 3D).

`std::vector<double> shape::getcoords(void)`

shape myquadrangle("quadrangle", 1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {2,2,2,2});
std::vector<double> mycoords = myquadrangle.getcoords();

This returns the coordinates of all nodes in the shape mesh.

`std::string shape::getname(void)`

shape myquadrangle("quadrangle", 1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {6,8,6,8});
std :: string name = myquadrangle.getname();
std :: cout << name << std::endl;

This gives the shape name ("quadrangle" here).

`std::vector<shape> shape::getsons(void)`

shape myquadrangle("quadrangle", 1, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {6,8,6,8});
std :: vector<shape> mylines = myquadrangle.getsons();
mylines [0]. setphysicalregion (2);
mylines [1]. setphysicalregion (2);
mylines [2]. setphysicalregion (2);
mylines [3]. setphysicalregion (2);
mesh mymesh(mylines);
mymesh.write("meshed.msh");

This outputs a vector containing the direct subshapes of the initial shape. Here it provides the 4 contour lines of the quadrangle.

`int shape::getphysicalregion(void)`

shape myquadrangle("quadrangle", 111, {−1,−1,0, 1,−1,0, 1,1,0, −1,1,0}, {6,8,6,8});
int physreg = myquadrangle.getphysicalregion();
std :: cout << physreg << std::endl;

This gives the physical region number of a given shape. The physical region number is used in the finite element simulation to identify a region.

3.12 The *sl* namespace (/src/expression/operation/sl.h):

The sl namespace provides a collection of tools.

`void sl::printversion(void)`

printversion ();

This prints information on the sparselizard version.

`int sl::getversion(void)`

int versionnumber = getversion();

This returns the sparselizard version number in format $YYYYMM$. The version number is defined by the year and month of the release.

`int sl::getsubversion(void)`

int subversionnumber = getsubversion();

This returns the sparselizard subversion number (0 for a release).

`std::string sl::getversionname(void)`

std :: string versionname = getversionname();

This returns the sparselizard version name.

`void sl::setmaxnumthreads(int mnt)`

setmaxnumthreads(2);

This sets the maximum number of threads allowed.

`int sl::getmaxnumthreads(void)`

int mnt = getmaxnumthreads();

This returns the maximum number of threads allowed.

`field sl::getx(void)`

expression x = getx();

This returns the x coordinate.

`field sl::gety(void)`

expression y = gety();

This returns the y coordinate.

`field sl::getz(void)`

expression z = getz();

This returns the z coordinate.

`double sl::getpi(void)`

double pi = getpi();

This returns the value of pi.

`double sl::getmu0(void)`

double mu0 = getmu0();

This returns the vacuum permeability.

`double sl::getepsilon0(void)`

double eps0 = getepsilon0();

This returns the vacuum permittivity.

`double sl::getrandom(void)`

double rnd = getrandom();

This returns a random value uniformly distributed between 0.0 and 1.0.

`int sl::selectunion(std::vector<int> physregs)`

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

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

`int sl::selectintersection(std::vector<int> physregs, int intersectdim)`

```
mesh mymesh("disk.msh");
int sur = 2, top = 3;
int line = selectintersection ({sur, top}, 1);
```

This creates a new/returns an existing line physical region that is the intersection of surfaces 2 and 3.

`int sl::selectall(void)`

```
int rega = 1, regb = 2;
shape qa("quadrangle", rega, {0,0,0, 1,0,0, 1,1,0, 0,1,0}, {5,5,5,5});
shape qb("quadrangle", regb, {1,0,0, 2,0,0, 2,1,0, 1,1,0}, {5,5,5,5});
mesh mymesh({qa,qb});
int wholedomain = selectall();
mymesh.write("mesh.msh");
```

This creates a new/returns an existing physical region that covers the entire domain.

`bool sl::isdefined(int physreg)`

mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
bool issurdefined = isdefined(sur);

This checks if a region is defined.

`bool sl::isempty(int physreg)`

...
bool issurempty = isempty(sur);

This checks if a region is empty.

`bool sl::isinside(int physregtocheck, int physreg)`

...
bool issurinsidevol = isinside(sur, vol);

This checks if a region is fully included in another region.

`bool sl::istouching(int physregtocheck, int physreg)`

...
bool issurtouchingtop = istouching(sur, top);

This checks if a region is touching another region.

`void sl::printvector(std::vector<double/int/bool> input)`

std::vector<double> v = {2.4, 3.14, −0.1};
printvector(v);

This prints the size of the vector as well as its values.

`void sl::writevector(std::string filename, std::vector<double> towrite,`
`char delimiter = ',', bool writesize = false)`

std::vector<double> v = {2.4, 3.14, −0.1};
writevector("vecvals.txt", v);

This writes to disk all double values with the requested delimiter (use '\n' for newline delimiter). If requested, the vector size can be written at the beginning of the file.

`std::vector<double> sl::loadvector(std::string filename, char delimiter = ',', bool sizeincluded = false)`

...
std::vector<double> vloaded = loadvector("vecvals.txt");
printvector(vloaded);

This loads from disk a vector with a given delimiter (use '\n' for newline delimiter). Set 'sizeincluded' to true if the first number in the file is the vector length integer.

`std::string sl::allpartition(std::string meshfile)`

slmpi::initialize();
std::string partfilename = allpartition("disk.msh");
slmpi::finalize();

This is a collective MPI operation (must be called by all ranks). This function partitions the requested mesh into a number of parts equal to the number of ranks. All parts are saved to disk and the part file name for each rank is returned. The GMSH API must be available to partition into more than one part.

`expression sl::norm(expression expr)`

expression myvector = array3x1(1,2,3);
norm(myvector).print();

This gives the L2 norm of an expression.

`expression sl::normal(void)`
`expression sl::normal(int pointoutofphysreg)`

mesh mymesh("disk.msh");
int vol = 1, sur = 2;
normal(vol).write(sur, "normal.pos", 1);

This defines a normal vector with unit norm. If a physical region is provided as argument then the normal points out of it. If no physical region is provided then the normal can be flipped depending on the element orientation in the mesh.

`expression sl::tangent(void)`

mesh mymesh("disk.msh");
int top = 3;
tangent().write(top, "tangent.pos", 1);

This defines a tangent vector with unit norm.

`void sl::scatterwrite(std::string filename, std::vector<double> xcoords, std::vector<double> ycoords, std::vector<double> zcoords, std::vector<double> compxevals, std::vector<double> compyevals = {}, std::vector<double> compzevals = {})`

std::vector<double> coordx = {0.0, 1.0, 2.0};
std::vector<double> coordy = {0.0, 1.0, 2.0};
std::vector<double> coordz = {0.0, 0.0, 0.0};
std::vector<double> vals = {10.0, 20.0, 30.0};
scatterwrite("values.vtk", coordx, coordy, coordz, vals);

This writes to the output .vtk file a set of values at given coordinates. If at least one of 'compyevals' or 'compzevals' is not empty then the values saved are vectors and not scalars.

`void sl::setaxisymmetry(void)`

setaxisymmetry();

This call should be placed at the very beginning of the code. After the call everything will be solved assuming axisymmetry (works for 2D meshes in the xy plane only). All equations should be written in their 3D form.
Please note that in order to correctly take into account the cylindrical coordinate change the appropriate space derivative operators should be used. As an example the *gradient of a vector* operator required in the mechanical strain calculation to compute the gradient of a mechanical displacement should not be defined manually using the *dx*, *dy* and *dz* space derivatives. The *grad* operator should instead be called on the mechanical displacement vector.

`void sl::setfundamentalfrequency(double f)`

setfundamentalfrequency(50);

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

`void sl::settime(double t)`

settime(1e−3);

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

`double sl::gettime(void)`

settime(1e−3);
double gettime();

This gets the value of the time variable t.

`expression sl::meshsize(int integrationorder)`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
expression ms = meshsize(2);
ms.write(top, "meshsize.pos", 1);
```

This returns an expression whose value is the length/area/volume for each 1D/2D/3D mesh element. The value is constant on each element.

`expression sl::fieldorder(field input, double alpha = -1.0, double absthres = 0.0)`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
field v("h1");
v.setorder(vol, 2);
expression fo = fieldorder(v);
fo.write(vol, "fieldorder.pos", 1);
```

This returns an expression whose value is the interpolation order on each element for the provided field. The value is constant on each element. When argument α is set the value returned is the lowest order required to include α % of the total shape function coefficient weight. An additional optional argument can be set to provide a minimum total weight below which the lowest possible field order is returned.

`expression sl::getharmonic(int harmnum, expression input, int numfftharms = -1)`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
field v("h1", {2});
v.setorder(vol, 1);
v.harmonic(2).setvalue(vol, 1);
expression constcomp = getharmonic(1, abs(v), 10);
constcomp.write(vol, "constcomp.pos", 1);
```

Get a single harmonic from a multiharmonic expression. Set a positive last argument to use an FFT to compute the harmonic. The returned expression is on harmonic 1.

`expression sl::makeharmonic(std::vector<int> harms, std::vector<expression> exprs)`

```
...
makeharmonic({1,2,4}, {11,v.harmonic(2),14}).write(vol, "harmexpr.pos", 1);
```

Return a multiharmonic expression whose harmonic numbers and values are provided as argument. The argument expressions must be on harmonic 1.

```
expression sl::moveharmonic(std::vector<int> origharms, std::vector<int> destharms,
expression input, int numfftharms = -1)
```

...
moveharmonic({1,2}, {5,3}, 11+v).write(vol, "moved.pos", 1);

Return an expression equal to the input expression with a selected and moved harmonic content. Set a positive last argument to use an FFT to compute the harmonics of the input expression.

```
std::vector<double> sl::gettotalforce(int physreg, expression EorH,
expression epsilonormu, int extraintegrationorder = 0)
```

mesh mymesh("disk.msh");
int vol = 1;
field phi("h1");
phi.setorder(vol, 2);
double mu0 = 4*getpi()*1e-7;
parameter mu;
mu|vol = mu0;
std::vector<double> totforce = gettotalforce(vol, -grad(phi), mu);

This returns the components of the total magnetostatic/electrostatic force acting on a given region. In the axisymmetric case zero x and z components are returned and the y component includes a 2π factor to provide the force acting on the corresponding 3D shape. Units are N per unit depth in 2D and N in 2D axisymmetry and 3D.

```
std::vector<double> sl::gettotalforce(int physreg, expression meshdeform,
expression EorH, expression epsilonormu, int extraintegrationorder = 0)
```

...
field u("h1xyz");
u.setorder(vol, 1);
std::vector<double> totforcedef = gettotalforce(vol, u, -grad(phi), mu);

Similar to the above function but the total force is computed on the mesh deformed by field 'u'.

```
std::vector<double> sl::printtotalforce(int physreg, expression EorH,
expression epsilonormu, int extraintegrationorder = 0)
```

...
printtotalforce (vol, -grad(phi), mu);

Prints the total force and its unit. The total force value is returned.

```
std::vector<double> sl::printtotalforce(int physreg, expression meshdeform,
expression EorH, expression epsilonormu, int extraintegrationorder = 0)
```

...
printtotalforce (vol, u, −grad(phi), mu);

Similar to the above function but the total force is computed on the mesh deformed by field 'u'.

```
void sl::setphysicalregionshift(int shiftamount)
```

setphysicalregionshift (1000);

This shifts the physical region numbers by shiftamount × (1 + physical region dimension) when loading a mesh. In the example the point/line/face/volume (0D/1D/2D/3D) physical region numbers will be shifted by 1000/2000/3000/4000 when a mesh is loaded.

```
void sl::writeshapefunctions(std::string filename, std::string sftypename,
int elementtypenumber, int maxorder, bool allorientations = false)
```

writeshapefunctions("sf.pos", "hcurl", 2, 2);

This writes to file all shape functions up to a requested order. It is a convenient tool to visualize the shape functions.

```
expression sl::t(void)
```

mesh mymesh("disk.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().

```
void sl::grouptimesteps(std::string filename, std::vector<std::string> filestogroup,
std::vector<double> timevals)
```

mesh mymesh("disk.msh");
int vol = 1;
field v("h1");
v.setorder(vol, 1);
v.write(vol, "v1.vtu", 1);
v.write(vol, "v2.vtu", 1);
grouptimesteps("v.pvd", {"v1.vtu", "v2.vtu"}, {0.0, 10.0});

This writes a .pvd ParaView file to group a set of .vtu files that are time solutions at the time values provided in 'timevals'.

`void sl::grouptimesteps(std::string filename, std::string fileprefix, int firstint, std::vector<double> timevals)`

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1");
v.setorder(vol, 1);
v.write(vol, "v1.vtu", 1);
v.write(vol, "v2.vtu", 1);
grouptimesteps("v.pvd", "v", 1, {0.0, 10.0});
```

This is similar to the previous function except that the full list of file names to group does not have to be provided. The file names are constructed from the file prefix with an appended integer starting from 'firstint' by steps of 1. The file names are ended with .vtu.

`std::vector<std::vector<shape>> sl::loadshape(std::string meshfile)`

```
std::vector<std::vector<shape>> diskshapes = loadshape("disk.msh");
// Add a thin slice on top of the disk (diskshapes[2][1] is the disk top face):
shape thinslice = diskshapes[2][1].extrude(5, 0.02, 2);
mesh mymesh({diskshapes[2][0], diskshapes[2][1], diskshapes[3][0], thinslice});
mymesh.write("editeddisk.msh");
```

This function loads a mesh file to shapes. The output 'diskshapes[d]' holds a shape for every physical region of dimension d (0D, 1D, 2D, 3D) defined in the mesh file. The loaded shapes can be edited (extruded, deformed,...) and grouped with other shapes to create a new mesh. Note that the usage of loaded shapes might be more limited than other shapes.

`void sl::settimederivative(vec dtx)`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2;
field v("h1");
v.setorder(vol, 1);
v.setconstraint(sur);
formulation poisson;
poisson += integral(vol, grad(dof(v))*grad(tf(v)));
vec solt1(poisson), solt2(poisson);
vec dtsol = 0.1*(solt2-solt1);
```

settimederivative(dtsol);
dt(v).write(vol, "dtv.pos", 1);

This provides the first order time derivative vector to the universe and removes the second order time derivative vector.

`void sl::settimederivative(vec dtx, vec dtdtx)`

...
settimederivative(dtsol, vec(poisson));
dtdt(v).write(vol, "dtdtv.pos", 1);

This provides the first and second order time derivative vectors to the universe.

`expression sl::dx(expression input)`

mesh mymesh("disk.msh");
int vol = 1;
field v("h1");
v.setorder(vol, 1);
dx(v).write(vol, "dxv.vtk", 1);

This is the x space derivative.

`expression sl::dy(expression input)`

...
dy(v).write(vol, "dyv.vtk", 1);

This is the y space derivative.

`expression sl::dz(expression input)`

...
dz(v).write(vol, "dzv.vtk", 1);

This is the z space derivative.

`expression sl::dt(expression input)`

...
setfundamentalfrequency(50);
field vmh("h1", {2,3});
vmh.setorder(vol, 1);
dt(vmh).write(vol, "dtv.vtk", 1);
dt(dt(vmh)).write(vol, "dt−dtv.vtk", 1);

This is the first order time derivative.

`expression sl::dtdt(expression input)`

...
dtdt(vmh).write(vol, "dtdtv.vtk", 1);

This is the second order time derivative.

`expression sl::dtdtdt(expression input)`

...
dtdtdt(vmh).write(vol, "dtdtdtv.vtk", 1);

This is the third order time derivative.

`expression sl::dtdtdtdt(expression input)`

...
dtdtdtdt(vmh).write(vol, "dtdtdtdtv.vtk", 1);

This is the fourth order time derivative.

`expression sl::dt(expression input, double initdt, double initdtdt)`

expression dtapprox = dt(t()∗t(), 0, 2);

This gives the transient approximation of the first order time derivative of a space-independent expression. The initial values must be provided when using generalized alpha and are ignored otherwise.

`expression sl::dtdt(expression input, double initdt, double initdtdt)`

expression dtdtapprox = dtdt(t()∗t(), 0, 2);

This gives the transient approximation of the second order time derivative of a space-independent expression. The initial values must be provided when using generalized alpha and are ignored otherwise.

`expression sl::sin/cos/tan/asin/acos/atan/abs/sqrt/log/exp(expression input)`

expression expr1 = sin(2);
expression expr2 = cos(2);
expression expr3 = tan(2);
expression expr4 = asin(0.5);
expression expr5 = acos(0.5);
expression expr6 = atan(0.5);
expression expr7 = abs(−2);

```
expression expr8 = sqrt(2);
expression expr9 = log(2);
expression expr10 = exp(2);
```

This is the sin/cos/tan/asin/acos/atan/abs/sqrt/log/exp function.

`expression sl::pow(expression base, expression exponent)`

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

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

`expression sl::mod(expression input, double modval)`

```
expression expr = mod(2.55, 0.6);
```

This is the modulo function.

`expression sl::ifpositive(expression condexpr, expression trueexpr, expression falseexpr)`

```
mesh mymesh("disk.msh");
int vol = 1;
field x("x"), y("y"), z("z");
ifpositive (x+y, 1, -1).write(vol, "cond.pos", 1);
```

This returns a conditional expression. The expression value is 'trueexpr' for all evaluation points where 'condexpr' is larger or equal to zero and its value is 'falseexpr' otherwise.

`expression sl::andpositive(std::vector<expression> exprs)`

```
mesh mymesh("disk.msh");
int vol = 1;
field x("x"), y("y"), z("z");
andpositive({x,y}).write(vol, "and.pos", 1);
```

This returns an expression whose value is 1 for all evaluation points where the value of all input expressions is larger or equal to zero and -1 otherwise.

`expression sl::orpositive(std::vector<expression> exprs)`

```
mesh mymesh("disk.msh");
int vol = 1;
field x("x"), y("y"), z("z");
orpositive({x,y}).write(vol, "or.pos", 1);
```

This returns an expression whose value is 1 for all evaluation points where at least one input expression has a value larger or equal to zero and -1 otherwise.

```
expression sl::max(expression a, expression b)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field x("x"), y("y"), z("z");
max(x,y).write(vol, "max.pos", 1);
```

This returns an expression whose value is the max of the argument expressions.

```
expression sl::min(expression a, expression b)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field x("x"), y("y"), z("z");
min(x,y).write(vol, "min.pos", 1);
```

This returns an expression whose value is the min of the argument expressions.

```
expression sl::on(int physreg, expression expr, bool errorifnotfound = true)
```

This function allows to use fields, unknown *dof* fields or general expressions across physical regions with possibly non-matching meshes by evaluating the expression 'expr' using a (x,y,z) coordinate interpolation. It makes it straightforward to setup the **mortar finite element method** in order to enforce general relations, such as field equality $u_1 = u_2$, at the interface Γ of non-matching meshes. This can for example be achieved with a Lagrange multiplier λ such that

$$\int_\Gamma \lambda(u_1^{'} - u_2^{'}) + (u_1 - u_2)\lambda' d\Gamma = 0$$

holds for any appropriate field u_1', u_2' and λ'. The example below illustrates the formulation terms needed to implement the Lagrange multiplier between interfaces Γ_1 and Γ_2.

```
...
field u1("h1xyz"), u2("h1xyz"), lambda("h1xyz");
...
formulation elasticity ;
...
elasticity += integral(gamma1, dof(lambda) * tf(u1) );
elasticity += integral(gamma2, -on(gamma1, dof(lambda)) * tf(u2) );
elasticity += integral(gamma1, (dof(u1) - on(gamma2, dof(u2))) * tf(lambda) );
```

When setting flag 'errorifnotfound' to false any point in Γ_1 without a relative in Γ_2 (and vice versa) does not contribute to the assembled matrix.

The case where there is no unknown *dof* term in the expression 'expr' is explicited below.

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
field x("x"), y("y"), z("z"), v("h1");
v.setorder(vol, 1);
formulation projection;
projection += integral(top, dof(v)*tf(v) - on(vol, dz(z))*tf(v));
//projection += integral(top, dof(v)*tf(v) - dz(z)*tf(v));
projection.solve();
v.write(top, "dzz.pos", 1);
```

With the 'on' expression in the first formulation term of the above example, the (x,y,z) coordinates corresponding to each Gauss point of the integral are first calculated then the dz(z) expression is evaluated through interpolation at these (x,y,z) coordinates on region 'vol'. With 'on' here dz(z) is correctly evaluated as 1 because the z derivative calculation is performed on the volume region 'vol'. Without the 'on' operator the z derivative would be wrongly calculated on the 'top' face of the disk (a plane perpendicular to the z axis).
If a requested interpolation point cannot be found (because it is outside of 'physreg' or because the interpolation algorithm fails to converge, as can happen on curved 3D elements) then an error occurs unless 'errorifnotfound' is set to false. In the latter case the value returned at any non-found coordinate is zero, without error.

```
expression sl::on(int physreg, expression coordshift, expression expr,
bool errorifnotfound = true)
```

...

```
projection += integral(top, dof(v)*tf(v) - on(vol, array3x1(2*x,2*y,2*z), dz(z))*tf(v));
```

This is similar to the previous function but here the (x,y,z) coordinates at which to interpolate the expression are shifted as (x+compx(coordshift), y+compy(coordshift), z+compz(coordshift)).

```
expression sl::comp(int selectedcomp, expression input)
```

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

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 sl::compx(expression input)`

...
compx(2∗(u+u)).write(vol, "compx.vtk", 1);

This is equivalent to comp(0, expression).

`expression sl::compy(expression input)`

...
compy(2∗(u+u)).write(vol, "compy.vtk", 1);

This is equivalent to comp(1, expression).

`expression sl::compz(expression input)`

...
compz(2∗(u+u)).write(vol, "compz.vtk", 1);

This is equivalent to comp(2, expression).

`expression sl::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 sl::eye(int size)`

...
expression II = eye(2);
II.print();

This returns a size × size identity matrix.

`expression sl::transpose(expression input)`

...
expression utransposed = transpose(2∗u);

This transposes the vector or matrix expression.

`expression sl::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 sl::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 sl::grad(expression input)`

```
mesh mymesh("disk.msh");
int vol = 1;
field v("h1");
v.setorder(vol, 1);
grad(v).write(vol, "gradv.vtk", 1);
grad(v).print();
```

For a scalar input expression this is mathematically treated as the gradient of a scalar and the output is a column vector with one entry per space derivative. For a vector input expression this is mathematically treated as the gradient of a vector and the output has one row per component of the input expression and one column per space derivative.

`expression sl::div(expression input)`

```
...
field u("h1xyz");
u.setorder(vol, 1);
div(u).write(vol, "divu.vtk", 1);
```

This computes the divergence of a vector expression.

`expression sl::curl(expression input)`

```
...
curl(u).write(vol, "curlu.vtk", 1);
```

This computes the curl of a vector expression.

`expression sl::crossproduct(expression a, expression b)`

```
mesh mymesh("disk.msh");
int vol = 1, top = 3;
field E("hcurl");
expression n = normal(vol);
expression cp = crossproduct(E, n);
```

This computes the cross product of two vector expressions.

`expression sl::doubledotproduct(expression a, expression b)`

```
expression a = array2x2(1,2,3,4);
expression b = array2x2(11,12,13,14);
expression addotb = doubledotproduct(a, b);
addotb.print();
```

This computes the double dot product of two matrix expressions:

$$\boldsymbol{A} : \boldsymbol{B} = \sum_{i,j} A_{ij} B_{ij}$$

`expression sl::elementwiseproduct(expression a, expression b)`

```
expression a = array2x2(1,2,3,4);
expression b = array2x2(11,12,13,14);
expression atimesb = elementwiseproduct(a, b);
atimesb.print();
```

This computes the element-wise product of two matrix expressions.

`expression sl::trace(expression a)`

```
expression a = array2x2(1,2,3,4);
expression tracea = trace(a);
tracea.print();
```

This computes the trace of a square matrix expression.

`expression sl::dof(expression input)`
`expression sl::dof(expression input, int physreg)`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2;
```

```
field v("h1");
v.setorder(vol, 1);
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 sl::tf(expression input)
expression sl::tf(expression input, int physreg)
```

```
...
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).

```
bool sl::adapt(int verbosity = 0)
```

```
...
adapt();
```

Perform a h/p/hp adaptation according to the defined h/p/hp adaptivity settings. To define the h-adaptivity use function *.setadaptivity* on a mesh object. To define p-adaptivity for a field use function *.setorder(expression criterion,...)* on that field.
The function returns true/false if the mesh or any field order was changed/unchanged by the adaptation.

```
bool sl::alladapt(int verbosity = 0)
```

```
...
alladapt();
```

This is a collective MPI operation (must be called by all ranks). It replaces the *adapt* function in the DDM framework.

```
expression sl::zienkiewiczzhu(expression input)
```

```
int sur = 1;
shape q("quadrangle", sur, {0,0,0, 5,0,0, 5,1,0, 0,1,0}, {10,3,10,3});
mesh mymesh({q});

field v("h1"), x("x"), y("y");
```

```
v.setorder(sur, 1);
expression criterion = zienkiewiczzhu( grad(v) );
// Target max criterion is 0.05:
expression maxcrit = ifpositive( criterion - 0.05, 1, 0);
mymesh.setadaptivity(maxcrit, 0, 3);
v.setorder(maxcrit, 1, 3);

for (int i = 0; i < 10; i++)
{
        expression fct = sin(3*x)/(x*x+1)*sin(getpi()*y);
        v.setvalue(sur, fct );
        v.write(sur, "v"+std::to_string(i+100)+".vtk", 3);
        fieldorder(v).write(sur, "fieldorderv"+std::to_string(i+100)+".vtk", 1);
        criterion.write(sur, "zienkiewiczzhu"+std::to_string(i+100)+".vtk", 1);
        double relL2err = std::sqrt(pow(v-fct, 2).integrate(sur, 5)/pow(fct, 2).integrate(sur, 5));
        std::cout << "Relative L2 error @" << i << " is " << relL2err << std::endl;
        adapt(2);
}
```

This defines a Zienkiewicz-Zhu type error indicator for the argument expression. The value of the returned expression is constant over each element. It equals the maximum of the argument expression value jump between that element and any neighbour. In the above example the *zienkiewiczzhu(grad(v))* expression quantifies the discontinuity of the field derivative. For non-scalar arguments the function is applied to each entry and the norm is returned.

`expression sl::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 sl::solve(mat A, vec b, std::string soltype = "lu", bool diagscaling = false)`

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

```
v.setdata(vol, sol);
v.write(vol, "sol.pos", 1);
```

This solves an algebraic problem with a (possibly reused) LU or Cholesky factorization by calling the mumps parallel direct solver via petsc. The matrix can be diagonally scaled for an improved conditioning (especially in multiphysics problems). In case of diagonal scaling A is **modified** after the call.

```
std::vector<vec> sl::solve(mat A, std::vector<vec> b, std::string soltype = "lu")
```

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2;
field v("h1"), x("x");
v.setorder(vol, 1);
formulation projection;
projection += integral(vol, dof(v)*tf(v) - x*tf(v));
projection.generate();
vec b0 = projection.b();
vec b1 = 2*b0;
vec b2 = 3*b0;
std::vector<vec> sol = solve(projection.A(), {b0, b1, b2});
v.setdata(vol, sol[0]);
v.write(vol, "sol0.pos", 1);
v.setdata(vol, sol[1]);
v.write(vol, "sol1.pos", 1);
v.setdata(vol, sol[2]);
v.write(vol, "sol2.pos", 1);
```

This allows to efficiently solve $Ax = b$ for multiple right handside vectors b.

```
void sl::solve(mat A, vec b, vec sol, double& relrestol, int& maxnumit,
std::string soltype = "bicgstab", std::string precondtype = "sor", int verbosity = 1,
bool diagscaling = false)
```

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

```
double relativeresidualtol = 1e−8;
int maxnumit = 200;
vec initsol(projection);
solve(projection.A(), projection.b(), initsol, relativeresidualtol, maxnumit);

std::cout << "Reached residual " << relativeresidualtol << " at iteration " << maxnumit << std::endl;

v.setdata(vol, initsol);
std::cout << "Max solution value is " << v.max(vol,5)[0] << std::endl;
```

This solves an algebraic problem with a preconditioned (*ilu*, *sor* or *gamg*) iterative solver (*gmres* or *bicgstab*). Vector *sol* is used as initial guess and holds the solution at the end of the call. Values *relrestol* and *maxnumit* give the relative residual tolerance and the maximum number of iterations to be performed by the iterative solver. Upon return these two values provide the achieved residual and the number of iterations actually performed. The matrix can be diagonally scaled for an improved conditioning (especially in multiphysics problems). In case of diagonal scaling A is **modified** after the call.

```
std::vector<double> sl::gmres(densemat (*mymatmult)(densemat), densemat b,
densemat x, double relrestol, int maxnumit, int verbosity = 1)
```

```
densemat custommatmult(densemat x)
{
        double* xptr = x.getvalues();
        for (int i = 0; i < x.count(); i++)
                xptr[i] = 2*xptr[i];
        return x;
}

int main(void)
{
        densemat b(4,1, 0,1);
        densemat x(4,1, 0);
        std::vector<double> resvec = gmres(custommatmult, b, x, 1e−8, 4);
        x.print();
}
```

This is a MPI based gmres with custom matrix vector product (no restart). The initial guess and solution are in x. The relative residual at each iteration is returned (the vector length is the number of iterations + 1, the first element is the initial residual).

`std::vector<double> sl::linspace(double a, double b, int num)`

std :: vector<double> vals = linspace(0.5, 2.5, 5);
printvector (vals); // {0.5,1.0,1.5,2.0,2.5}

This gives a vector of five equally spaced values from a to b.

`std::vector<double> sl::logspace(double a, double b, int num, double basis = 10.0)`

std :: vector<double> vals = logspace(1, 3, 3);
printvector (vals); // {10.0,100.0,1000.0}

This gives the basis to the power of each of the three values in the linspace.

`expression sl::dbtoneper(expression toconvert)`

expression neperattenuation = dbtoneper(100.0);

This converts a dB value to Nepers.

`void sl::setdata(vec invec)`

vec x;
setdata(x);

This transfers the vector data to all fields and ports defined in the formulation associated to the vector.

`expression sl::strain(expression input)`

```
mesh mymesh("disk.msh");
int vol = 1;
field u("h1xyz");
expression engstrain = strain(u);
engstrain.print();
```

This defines the (linear) engineering strains in Voigt form (ϵ_{xx}, ϵ_{yy}, ϵ_{zz}, γ_{yz}, γ_{xz},γ_{xy}). The input can either be the displacement field or its gradient.

`expression sl::greenlagrangestrain(expression input)`

```
mesh mymesh("disk.msh");
int vol = 1;
field u("h1xyz");
expression glstrain = greenlagrangestrain(u);
glstrain.print();
```

This defines the (nonlinear) Green-Lagrange strains in Voigt form (ϵ_{xx}, ϵ_{yy}, ϵ_{zz}, γ_{yz}, γ_{xz},γ_{xy}). The input can either be the displacement field or its gradient.

```
expression sl::vonmises(expression stress)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field u("h1xyz");
u.setorder(vol, 1);
double E = 150e9, nu = 0.3;
// Elasticity matrix for isotropic materials:
expression H(6,6,{1−nu,nu,nu,0,0,0, nu,1−nu,nu,0,0,0, nu,nu,1−nu,0,0,0,
0,0,0,0.5∗(1−2∗nu),0,0, 0,0,0,0,0.5∗(1−2∗nu),0, 0,0,0,0,0,0.5∗(1−2∗nu)});
H = E/(1+nu)/(1−2∗nu) ∗ H;
vonmises( H∗strain(u) ).write(vol,"vonmises.vtk", 1);
double maxvonmises = vonmises( H∗strain(u) ).max(vol,5)[0];
```

This returns the von Mises stress expression corresponding to the 3D stress tensor provided as argument. The stress tensor should be provided in Voigt form (σ_{xx}, σ_{yy}, σ_{zz}, σ_{yz}, σ_{xz},σ_{xy}).
For 2D plane stress problems all z related components of the stress tensor are 0. For plane strain problems do not forget the term $\sigma_{zz} = \nu \cdot (\sigma_{xx} + \sigma_{yy})$. Please note that in the example above the strains must be calculated on a volume in 3D and on a surface in 2D to get correct results (surface or line derivatives are considered otherwise, which leads to erroneous strains).

```
std::vector<integration> sl::continuitycondition(int gamma1, int gamma2, field u1,
field u2, int lagmultorder, bool errorifnotfound = true)
```

```
...
field u1("h1xyz"), u2("h1xyz");
...
formulation elasticity ;
...
elasticity += continuitycondition(gamma1, gamma2, u1, u2, 1);
```

This returns the formulation terms required to enforce $u_1 = u_2$ between boundary region Γ_1 and Γ_2 (with $\Gamma_1 \subseteq \Gamma_2$, meshes can be non-matching). The condition is based on a Lagrange multiplier of same type and same harmonic content as field u_1 and u_2. The mortar finite element method is used to link the unknown *dof* fields on Γ_1 and Γ_2 so that there is no restriction on the mesh used for both regions. In case Γ_2 is larger than Γ_1 ($\Gamma_1 \subset \Gamma_2$) the boolean flag must be set to false.

Note that the **Lagrange multiplier interpolation order cannot be arbitrarily chosen**. It might be required to be at least one lower than the field order.

`std::vector<integration> sl::continuitycondition(int gamma1, int gamma2, field u1, field u2, std::vector<double> rotcent, double rotangz, double angzmod, double factor, int lagmultorder)`

```
...
// Rotor−stator interface:
int rotorside = 11, statorside = 12;
...
// Rotor rotation around z axis:
double alpha = 30.0;
...
field az("h1");
...
formulation magnetostatics;
...
magnetostatics += continuitycondition(statorside, rotorside, az, az, {0,0,0}, alpha, 45.0, −1.0, 1);
```

This returns the formulation terms required to enforce field continuity across an *angzmod* degrees slice of a rotor-stator interface where the rotor geometry is rotated *rotangz* degrees around the z axis (rotation center is *rotcent*). This situation arises for example in electric motor simulations when (anti)periodicity can be considered and thus only a slice of the entire 360 degrees needs to be simulated. Use a factor -1 for antiperiodicity. Boundary Γ_1 is the rotor-stator interface on the (non-moving) stator side while boundary Γ_2 is the interface on the rotor side. In the unrotated position the bottom boundary of the stator and rotor slice must be aligned with the x axis.
The condition is based on a Lagrange multiplier of same type and same harmonic content as field u_1 and u_2. The mortar finite element method is used to link the unknown *dof* fields on Γ_1 and Γ_2 so that there is no restriction on the mesh used for both regions.

Note that the **Lagrange multiplier interpolation order cannot be arbitrarily chosen**. It might be required to be at least one lower than the field order.

`std::vector<integration> sl::periodicitycondition(int gamma1, int gamma2, field u, std::vector<double> dat1, std::vector<double> dat2, double factor, int lagmultorder)`

```
...
field u("h1xyz");
...
formulation elasticity ;
...
// In case gamma2 is gamma1 rotated (ax, ay, az) degrees around first the x then y then z axis.
```

```
// Rotation center is (cx, cy, cz).
double cx = 0, cy = 0, cz = 0, ax = 0, ay = 0, az = 60;
elasticity += periodicitycondition(gamma1, gamma2, u, {cx, cy, cz}, {ax, ay, az}, 1.0, 1);

// In case gamma2 is gamma1 translated by a distance d in direction (nx, ny, nz).
double d = 0.8, nx = 1, ny = 0, nz = 0;
elasticity += periodicitycondition(gamma1, gamma2, u, {nx, ny, nz}, {d}, 1.0, 1);
```

This returns the formulation terms required to enforce on field u a rotation or translation periodic condition between boundary region Γ_1 and Γ_2 (meshes can be non-matching). A factor different than one can be provided to scale the field on Γ_2 (use -1 for antiperiodicity). The condition is based on a Lagrange multiplier of same type and same harmonic content as field u. The mortar finite element method is used to link the unknown *dof* fields on Γ_1 and Γ_2 so that there is no restriction on the mesh used for both regions.
More advanced periodic conditions can be implemented easily using the *on()* function.

Note that the **Lagrange multiplier interpolation order cannot be arbitrarily chosen**. It might be required to be at least one lower than the field order.

`expression sl::predefinedelasticity(expression dofu, expression tfu, expression Eyoung, expression nupoisson, std::string myoption = "")`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
field u("h1xyz");
u.setorder(vol, 2);
u.setconstraint(sur);
formulation elasticity ;
elasticity += integral(vol, predefinedelasticity (dof(u), tf(u), 150e9, 0.3));
// elasticity += integral(vol, -2330*dtdt(dof(u))*tf(u));
// Atmospheric pressure load on top face deformed by field u (might require a nonlinear iteration ):
elasticity += integral(top, u, 1e5*-normal(vol)*tf(u));
elasticity .solve ();
u.write(top, "u.vtk", 2);
```

This defines a classical isotropic linear elasticity formulation whose strong form is:

$$\begin{cases} \nabla \cdot \boldsymbol{\sigma} - \rho\, \ddot{\boldsymbol{u}} + \boldsymbol{F} = \boldsymbol{0} \\ \boldsymbol{\epsilon} = \frac{1}{2}[\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T] \\ \boldsymbol{\sigma} = \boldsymbol{H} : \boldsymbol{\epsilon} \end{cases}$$

Field u is the mechanical displacement. Expression 'Eyoung' is Young's modulus $[Pa]$ while 'nupoisson' is Poisson's ratio. In 2D the option string must be either set to "planestrain" or to "planestress" for respectively a plane strain or plane stress assumption.

```
expression sl::predefinedelasticity(expression dofu, expression tfu,
expression elasticitymatrix, std::string myoption = "")
```

```
mesh mymesh("disk.msh");
int vol = 1;
field u("h1xyz");
u.setorder(vol, 2);
// It is enough to only provide the lower triangular part of the elasticity matrix (symmetric):
expression H(6,6,{195e9, 36e9,195e9, 64e9,64e9,166e9, 0,0,0,80e9, 0,0,0,0,80e9, 0,0,0,0,0,51e9});
formulation elasticity ;
elasticity += integral(vol, predefinedelasticity (dof(u), tf(u), H));
elasticity += integral(vol, -2330*dtdt(dof(u))*tf(u));
```

This extends the previous function to general anisotropic materials.
The elasticity matrix $[Pa]$ must be provided in Voigt form (ϵ_{xx}, ϵ_{yy}, ϵ_{zz}, γ_{yz}, γ_{xz},γ_{xy}).

```
expression sl::predefinedelasticity(expression dofu, expression tfu, field u,
expression Eyoung, expression nupoisson, expression prestress,
std::string myoption = "")
```

```
mesh mymesh("disk.msh");
int vol = 1;
field u("h1xyz");
u.setorder(vol, 2);
expression prestress(6,1,{10e6 ,0,0,0,0,0});
formulation elasticity ;
elasticity += integral(vol, predefinedelasticity (dof(u), tf(u), u, 150e9, 0.3, prestress ));
elasticity += integral(vol, -2330*dtdt(dof(u))*tf(u));
```

This defines an isotropic linear elasticity formulation with geometric nonlinearity taken into account (full-Lagrangian formulation using the Green-Lagrange strain tensor). Problems with large displacements and rotations can be simulated with this equation but **strains must always remain small**. Buckling, snap-through and the like or eigenvalues of prestressed structures can be simulated with the above equation in combination with a nonlinear iteration loop.
Field u is the mechanical displacement. Expression 'Eyoung' is Young's modulus $[Pa]$ while 'nupoisson' is Poisson's ratio. The prestress vector $[Pa]$ must be provided in Voigt form (σ_{xx}, σ_{yy}, σ_{zz}, σ_{yz}, σ_{xz},σ_{xy}). Set the prestress expression to '0.0' for no prestress.

In 2D the option string must be either set to “planestrain” or to “planestress” for respectively a plane strain or plane stress assumption.

```
expression sl::predefinedelasticity(expression dofu, expression tfu, field u,
expression elasticitymatrix, expression prestress, std::string myoption = "")
```

```
mesh mymesh("disk.msh");
int vol = 1;
field u("h1xyz");
u.setorder(vol, 2);
// It is enough to only provide the lower triangular part of the elasticity matrix (symmetric):
expression H(6,6,{195e9, 36e9,195e9, 64e9,64e9,166e9, 0,0,0,80e9, 0,0,0,0,80e9, 0,0,0,0,0,51e9});
expression prestress(6,1,{10e6 ,0,0,0,0,0});
formulation elasticity ;
elasticity += integral(vol, predefinedelasticity (dof(u), tf(u), u, H, prestress ));
elasticity += integral(vol, −2330∗dtdt(dof(u))∗tf(u));
```

This extends the previous function to general anisotropic materials.
The elasticity matrix and the prestress vector $[Pa]$ must be provided in Voigt form (ϵ_{xx}, ϵ_{yy}, ϵ_{zz}, γ_{yz}, γ_{xz},γ_{xy} for the elasticity matrix and σ_{xx}, σ_{yy}, σ_{zz}, σ_{yz}, σ_{xz},σ_{xy} for the prestress vector).

```
expression sl::predefinedelectrostaticforce(expression input, expression E,
expression epsilon)
```

```
mesh mymesh("disk.msh");
int vol = 1, top = 3;
field v("h1"), u("h1xyz");
v.setorder(vol, 1);
u.setorder(vol, 2);
formulation elasticity ;
elasticity += integral(vol, predefinedelasticity (dof(u), tf(u), 150e9, 0.3));
elasticity += integral(vol, predefinedelectrostaticforce (tf(u,top), −grad(v), 8.854e−12));
```

This function defines the weak formulation term for electrostatic forces. The first argument is the mechanical displacement test function or its gradient, the second is the electric field while the third argument is the electric permittivity (must be a scalar).
Let us call $\boldsymbol{T}$ $[N/m^2]$ the electrostatic Maxwell stress tensor:

$$\boldsymbol{T} = \epsilon\, \boldsymbol{E} \otimes \boldsymbol{E} - \frac{1}{2}\epsilon\, (\boldsymbol{E} \cdot \boldsymbol{E})\, \boldsymbol{I}$$

where ϵ is the electric permittivity, $\boldsymbol{E}$ is the electric field and $\boldsymbol{I}$ is the identity matrix. The electrostatic force density is $\nabla \cdot \boldsymbol{T}$ $[N/m^3]$ so that the loading for a mechanical problem can be obtained by adding

the following formulation term:

$$\int_{\Omega} (\nabla \cdot \boldsymbol{T}) \cdot \boldsymbol{u}' \, d\Omega$$

where $\boldsymbol{u}$ is the mechanical displacement. The term can be rewritten in the form that is provided by this function:

$$-\int_{\Omega} \boldsymbol{T}\, \boldsymbol{\epsilon}' \, d\Omega$$

where $\boldsymbol{\epsilon}$ is the infinitesimal strain tensor. This is identical to what is obtained using the virtual work principle (for details refer to *'Domain decomposition techniques for the nonlinear, steady state, finite element simulation of MEMS ultrasonic transducer arrays*', page 40).
In this function a region should be provided to the test function argument to compute the force only for the degrees of freedom associated to that specific region (in the example above with *tf(u,top)* the force only acts on surface region 'top'). In any case a **correct force calculation requires to include** in the integration domain all elements in the region where the force acts and in the **element layer around** it (in the example above 'vol' includes all volume elements touching surface 'top').

`expression sl::predefinedmagnetostaticforce(expression input, expression H, expression mu)`

```
mesh mymesh("disk.msh");
int vol = 1, top = 3;
field phi("h1"), u("h1xyz");
phi.setorder(vol, 1);
u.setorder(vol, 2);
formulation elasticity ;
elasticity += integral(vol, predefinedelasticity (dof(u), tf(u), 150e9, 0.3));
elasticity += integral(vol, predefinedmagnetostaticforce(tf(u,top), −grad(phi), 4∗getpi()∗1e−7));
```

This is similar to the previous function but considering the magnetostatic Maxwell stress tensor:

$$\boldsymbol{T} = \mu\, \boldsymbol{H} \otimes \boldsymbol{H} - \frac{1}{2}\mu\, (\boldsymbol{H} \cdot \boldsymbol{H})\, \boldsymbol{I}$$

where μ is the magnetic permeability, $\boldsymbol{H}$ is the magnetic field and $\boldsymbol{I}$ is the identity matrix.

`expression sl::predefinedacousticwave(expression dofp, expression tfp, expression soundspeed, expression neperattenuation)`

```
int sur = 1, left = 2, wall = 3;
double h = 10e−3, l = 50e−3;
shape q("quadrangle", sur, {0,0,0, l,0,0, l,h,0, 0,h,0}, {250,50,250,50});
```

```
shape ll = q.getsons()[3];
ll.setphysicalregion(left);
shape lwall("union", wall, {q.getsons()[0], q.getsons()[1], q.getsons()[2]});
mesh mymesh({q,ll,lwall});

setfundamentalfrequency(40e3);
// Wave propagation requires both the in−phase (2) and quadrature (3) harmonics:
field p("h1", {2,3}), y("y");
p.setorder(sur, 2);
// In−phase only pressure source:
p.harmonic(2).setconstraint(left, y*(h−y)/(h*h/4));
p.harmonic(3).setconstraint(left, 0);
p.setconstraint(wall);

formulation acoustics;
acoustics += integral(sur, predefinedacousticwave(dof(p), tf(p), 340, dbtoneper(500)));
acoustics.solve();
p.write(sur, "p.vtu", 2);
// Write 50 timesteps for a time visualization:
// p.write(sur, "p.vtu", 2, 50);
```

This function defines the equation for (linear) acoustic wave propagation:

$$\nabla^2 p - \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = 0$$

An acoustic attenuation value can be provided (in Neper/m) in case of harmonic problems. For convenience use the function 'dbtoneper' to convert dB/m attenuation values to Np/m.
In the illustrative example above a highly-attenuated acoustic wave propagation in a rectangular 2D box is simulated.

`expression sl::predefinedacousticradiation(expression dofp, expression tfp, expression soundspeed, expression neperattenuation)`

```
...
acoustics += integral(sur, predefinedacousticradiation(dof(p), tf(p), 340, dbtoneper(500)));
```

This function defines the equation for the Sommerfeld acoustic radiation condition

$$\partial_{\boldsymbol{n}} p + \frac{1}{c}\frac{\partial p}{\partial t} = 0$$

which forces outgoing pressure waves at infinity: a pressure field of the form

$$p(r,t) = P\,\cos(\omega t - kr)$$

propagating in direction $\boldsymbol{e}_r$ perpendicular to the truncation boundary indeed satisfies the Sommerfeld radiation condition since

$$\partial_{\boldsymbol{n}}\, p = \partial_{\boldsymbol{e}_r} p = k\,P\,\sin(\omega t - kr) = \frac{\omega}{c}\,P\,\sin(\omega t - kr) = -\frac{1}{c}\,\frac{\partial\, P\,\cos(\omega t - kr)}{\partial t}$$

Zero artificial wave reflection at the truncation boundary happens only if it is perpendicular to the outgoing waves. In practical applications however the truncation boundary is not at an infinite distance from the acoustic source and the wave amplitude is not constant so that some level of artificial reflection cannot be avoided. To minimize it the truncation boundary should be placed as far as possible from the acoustic source (at least a few wavelength away).
An acoustic attenuation value can be provided to the function (in Neper/m) in case of harmonic problems. For convenience use the function 'dbtoneper' to convert dB/m attenuation values to Np/m.

```
expression sl::predefinedacousticstructureinteraction(expression dofp, expression tfp,
expression dofu, expression tfu, expression soundspeed, expression fluiddensity,
expression normal, expression neperattenuation, double scaling = 1.0)
```

...
field u("h1xy", {2,3});
u.setorder(sur, 2);
acoustics += integral(left, predefinedacousticstructureinteraction(dof(p), tf(p), dof(u), tf(u), 340, 1.2, array2x1(1,0), dbtoneper(500), 1e10));

This function defines the bi-directional coupling for acoustic-structure interaction at the medium interface. Field p is the acoustic pressure and field $\boldsymbol{u}$ is the mechanical displacement. Calling $\boldsymbol{n}$ the **normal** to the interface, **pointing out of the solid region**, the bi-directional coupling is obtained by adding the fluid pressure loading to the structure

$$\boldsymbol{f}_{pressure} = -p\,\boldsymbol{n}$$

as well as linking the structure acceleration to the fluid pressure normal derivative using Newton's law:

$$\partial_{\boldsymbol{n}}\, p = -\rho_{fluid}\,\frac{\partial^2 \boldsymbol{u}}{\partial t^2}\cdot \boldsymbol{n}$$

To have a good matrix conditionning a scaling factor s (e.g. $s = 10^{10}$) can be provided. In this case the pressure source is divided by s and, to compensate, the pressure force is multiplied by s. This leads to the correct membrane deflection but the pressure field is divided by the scaling factor.
An acoustic attenuation value can be provided to the function (in Neper/m) in case of harmonic problems. For convenience use the function 'dbtoneper' to convert dB/m attenuation values to Np/m.

```
expression sl::predefinedstokes(expression dofv, expression tfv, expression dofp,
expression tfp, expression mu, expression rho, expression dtrho, expression gradrho,
bool includetimederivs = false, bool isdensityconstant = true,
bool isviscosityconstant = true)
```

```
mesh mymesh("microvalve.msh");
int fluid = 2;
field v("h1xy"), p("h1");
v.setorder(fluid,2); p.setorder(fluid,1); // Satisfies the LBB condition
formulation stokesflow;
stokesflow += integral(fluid, predefinedstokes(dof(v), tf(v), dof(p), tf(p), 8.9e-4, 1000, 0, 0));
```

This defines the weak formulation for the Stokes (creeping) flow, a linear form of Navier-Stokes where the advective inertial forces are small compared to the viscous forces:

$$\begin{cases} -\rho\frac{\partial \mathbf{v}}{\partial t} - \nabla p + \nabla \cdot (\mu(\nabla \mathbf{v} + (\nabla \mathbf{v})^T) - \frac{2}{3}\mu(\nabla \cdot \mathbf{v})\mathbf{I}) = 0 \\ \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \end{cases}$$

where ρ $[kg/m^3]$ is the density, μ $[Pa \cdot s]$ is the dynamic viscosity of the fluid, p $[Pa]$ is the pressure and $\mathbf{v}$ $[m/s]$ is the flow velocity. This formulation is valid to simulate the flow of **Newtonian fluids** (air, water,...) with a very small Reynolds number ($Re \ll 1$):

$$Re = \frac{\rho v L}{\mu}$$

where L $[m]$ is the characteristic length of the flow. Low flow velocities, high viscosities or small dimensions can lead to a valid Stokes flow approximation. Flows in microscale devices such as microvalves are also good candidates for Stokes flow simulations.
Arguments *dtrho* and *gradrho* are respectively the time derivative and the gradient of the density while *includetimederivs* gives the option to include or not the time-derivative terms in the formulation. In case the density constant argument is set to true the fluid is supposed incompressible and the Navier-Stokes equations are further simplified since the divergence of the velocity is zero. If the viscosity is constant **in space** (it does not have to be constant in time) the constant viscosity argument can be set to true. By default the density and viscosity are supposed constant and the time-derivative terms are not included. Please note that to simulate the Stokes flow the **LBB condition has to be satisfied**. This is achieved by using nodal (h1) type shape functions with an interpolation order at least one higher for the velocity field than for the pressure field. Alternatively, an additional isotropic diffusive term or other stabilization techniques can be used to overcome the LBB limitation.

```
expression sl::predefinednavierstokes(expression dofv, expression tfv, expression v,
expression dofp, expression tfp, expression mu, expression rho, expression dtrho,
```

```
expression gradrho, bool includetimederivs = false, bool isdensityconstant = true,
bool isviscosityconstant = true)
```

```
mesh mymesh("microvalve.msh");
int fluid = 2;
field v("h1xy"), p("h1");
v.setorder(fluid,2); p.setorder(fluid,1); // Satisfies the LBB condition
formulation lamflow;
lamflow += integral(fluid, predefinednavierstokes(dof(v), tf(v), v, dof(p), tf(p), 8.9e-4, 1000, 0, 0));
```

This defines the weak formulation for the general (nonlinear) flow of **Newtonian fluids** (air, water,...):

$$\begin{cases} -\rho(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}) - \nabla p + \nabla \cdot (\mu(\nabla \mathbf{v} + (\nabla \mathbf{v})^T) - \frac{2}{3}\mu(\nabla \cdot \mathbf{v})\mathbf{I}) = 0 \\ \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \end{cases}$$

where ρ $[kg/m^3]$ is the density, μ $[Pa \cdot s]$ is the dynamic viscosity of the fluid, p $[Pa]$ is the pressure and $\mathbf{v}$ $[m/s]$ is the flow velocity. The formulation is provided in a form leading to a quadratic (Newton) convergence when solved iteratively in a loop.

This formulation is valid to simulate laminar as well as turbulent flows. Using it to simulate turbulent flows leads to a so-called **DNS** method (direct numerical simulation). DNS does not require any turbulence model since it takes into account the whole range of spatial and temporal scales of the turbulence. It therefore requires a spatial and time refinement that for industrial applications typically exceeds the computing power of the most advanced supercomputers. As an alternative, RANS and LES methods can be used for turbulent flow simulations.

The transition from a laminar to a turbulent flow is linked to a threshold value of the Reynolds number (defined above). For a flow in pipes the typical Reynolds number below which the flow is laminar is about 2000.

Arguments *dtrho* and *gradrho* are respectively the time derivative and the gradient of the density while *includetimederivs* gives the option to include or not the time-derivative terms in the formulation. In case the density constant argument is set to true the fluid is supposed incompressible and the Navier-Stokes equations are simplified since the divergence of the velocity is zero. If the viscosity is constant **in space** (it does not have to be constant in time) the constant viscosity argument can be set to true. By default the density and viscosity are supposed constant and the time-derivative terms are not included. Please note that for the simulation the **LBB condition has to be satisfied**. This is achieved by using nodal (h1) type shape functions with an interpolation order at least one higher for the velocity field than for the pressure field. Alternatively, an additional isotropic diffusive term or other stabilization techniques can be used to overcome the LBB limitation.

```
expression sl::predefinedadvectiondiffusion(expression doff, expression tff,
expression v, expression alpha, expression beta, expression gamma,
bool isdivvzero = true)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field T("h1"), v("h1xyz");
T.setorder(vol, 1);
v.setorder(vol, 1);
formulation advdiff;
advdiff += integral(vol, predefinedadvectiondiffusion(dof(T), tf(T), v, 1.0e-4, 1.0, 1.0, true));
```

This defines the weak formulation for the generalized advection-diffusion equation:

$$\beta \frac{\partial c}{\partial t} - \nabla \cdot (\boldsymbol{\alpha} \nabla c) + \gamma \nabla \cdot (c\boldsymbol{v}) = 0$$

where c is the scalar quantity of interest and $\boldsymbol{v}$ $[m/s]$ is the velocity that the quantity is moving with. With β and γ set to unit the classical advection-diffusion equation with diffusivity tensor $\boldsymbol{\alpha}$ $[m^2/s]$ is obtained. Set 'isdivvzero' to true if $\nabla \cdot \boldsymbol{v}$ is zero (for incompressible fluids).

`expression sl::predefineddiffusion(expression doff, expression tff, expression alpha, expression beta)`

```
mesh mymesh("disk.msh");
int vol = 1, top = 3;
// Temperature field [K]:
field T("h1");
T.setorder(vol, 1);
T.setconstraint(top, 298);
// Thermal conductivity of aluminium [W/(m*K)]:
double k = 237;
// Heat capacity of aluminium [J/(kg*K)]:
double cp = 897;
// Density of aluminium [kg/m^3]:
double rho = 2700;
formulation heatequation;
heatequation += integral(vol, predefineddiffusion(dof(T), tf(T), k, rho*cp));
```

This defines the weak formulation for the generalized diffusion equation:

$$\beta \frac{\partial c}{\partial t} - \nabla \cdot (\boldsymbol{\alpha} \nabla c) = 0$$

where c is the scalar quantity of interest. With β set to unit the classical diffusion equation with diffusivity tensor $\boldsymbol{\alpha}$ $[m^2/s]$ is obtained.

`expression sl::predefinedstabilization(std::string stabtype, expression delta,`

```
expression f, expression v, expression diffusivity, expression residual)
```

```
mesh mymesh("disk.msh");
int vol = 1;
field c("h1"), v("h1xyz");
c.setorder(vol, 1);
v.setorder(vol, 1);
// Diffusivity alpha (can be a tensor):
expression alpha = 0.001;
formulation advdiff;
advdiff += integral(vol, predefinedadvectiondiffusion(dof(c), tf(c), v, alpha, 1.0, 1.0));

// Tuning factor:
double delta = 0.5;

// Isotropic diffusion:
advdiff += integral(vol, predefinedstabilization ("iso", delta, c, v, 0.0, 0.0));
// Streamline anisotropic diffusion:
advdiff += integral(vol, predefinedstabilization ("aniso", delta, c, v, 0.0, 0.0));
// Crosswind diffusion:
advdiff += integral(vol, predefinedstabilization ("cw", delta, c, v, 0.0, 0.0));

// The following residual-based stabilizations require the strong-form residual.
// Neglecting the second order space-derivative still leads to a good residual approximation.
// The flow is supposed incompressible, that is with zero div(v).
expression dofresidual = dt(dof(c)) + v*grad(dof(c));
// For crosswind shockwave the residual at the previous iteration must be considered.
expression residual = dt(c) + v*grad(c);

// Crosswind shockwave diffusion:
advdiff += integral(vol, predefinedstabilization ("cws", delta, c, v, alpha, residual));

// Streamline Petrov-Galerkin diffusion:
advdiff += integral(vol, predefinedstabilization ("spg", delta, c, v, alpha, dofresidual));
// Streamline upwind Petrov-Galerkin diffusion:
advdiff += integral(vol, predefinedstabilization ("supg", delta, c, v, alpha, dofresidual));
```

This defines the isotropic, streamline anisotropic, crosswind, crosswind shockwave, SPG and SUPG stabilization methods for the advection-diffusion problem:

$$\frac{\partial c}{\partial t} - \nabla \cdot (\boldsymbol{\alpha} \nabla c) + \nabla \cdot (c \boldsymbol{v}) = 0$$

where c is the scalar quantity of interest, $\boldsymbol{v}$ $[m/s]$ is the velocity that the quantity is moving with and $\boldsymbol{\alpha}$ $[m^2/s]$ is the diffusivity tensor.
A characteristic number for advection-diffusion problems is the Peclet number:

$$P_e = \frac{h \,||\, \boldsymbol{v} \,||}{\alpha}$$

where h is the length of each mesh element. It quantifies the relative importance of advective and diffusive effects. When the Peclet number is large ($P_e \gg 1$) the problem is dominated by advection and prone to spurious oscillations in the solution. Although lowering the Peclet number can be achieved by refining the mesh, a classical alternative is to add stabilization terms to the original equation. A proper choice of stabilization should remove the oscillations while changing as little as possible the original problem. In the most simple method proposed (isotropic diffusion) the diffusivity α is artificially increased to lower the Peclet number. The more advanced methods proposed attempt to add artificial diffusion only where it is needed. In the crosswind shockwave, SPG and SUPG methods the residual of the advection-diffusion equation is used to quantify the local amount of diffusion to add.
The terms provided by the proposed stabilization methods have the following form:

- Isotropic diffusion: $\delta h \,||\, \boldsymbol{v} \,|| \, \nabla c \, \nabla c'$
- Streamline anisotropic diffusion: $\frac{\delta h}{||\, \boldsymbol{v} \,||} (\boldsymbol{v} \cdot \nabla c)(\boldsymbol{v} \cdot \nabla c')$
- Crosswind diffusion: $\delta h^{1.5} (\nabla c)^T \boldsymbol{T} \, \nabla c'$
- Crosswind shockwave: $\frac{1}{2} \max(0, \delta - \frac{1}{\gamma}) \, h \, \frac{|\text{residual}|}{||\, \nabla c \,||} (\nabla c)^T \boldsymbol{T} \, \nabla c'$, $\gamma = \frac{||\, \boldsymbol{v}_{/\!/} \,||\, h}{2\alpha}$, $\boldsymbol{v}_{/\!/} = \frac{\boldsymbol{v} \cdot \nabla c}{||\, \nabla c \,||^2} \nabla c$
- Streamline Petrov-Galerkin diffusion: $\frac{\delta h}{||\, \boldsymbol{v} \,||} (\text{residual}) \, \boldsymbol{v} \cdot \nabla c'$
- Streamline upwind Petrov-Galerkin diffusion: $\lambda (\lambda > 0) (\text{residual}) \, \boldsymbol{v} \cdot \nabla c'$, $\lambda = \frac{\delta h}{||\, \boldsymbol{v} \,||} - \frac{\alpha}{||\, \boldsymbol{v} \,||^2}$

where c' is the test function associated to field c and $\boldsymbol{T} = \mathbb{I} - \frac{1}{||\, \boldsymbol{v} \,||^2} \boldsymbol{v} \otimes \boldsymbol{v}$.
To understand the effect of the crosswind diffusion one can notice that for a 2D flow in the x direction only, tensor $\boldsymbol{T}$ becomes

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} \frac{v_x^2}{v_x^2} & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

and artificial diffusion is only added at places where ∇c has a component in the direction perpendicular to the flow.

How to use the predefined stabilization methods:
Due to the large amount of artificial diffusion added by the isotropic diffusion method it should only be considered as a fallback option. In practice a pair of one streamline, one crosswind method should

be used with the smallest possible tuning factor δ. If the problem allows, SUPG should be preferred over SPG and crosswind shockwave should be preferred over crosswind because the amount of diffusion added tends to be lower.

3.13 The *slmpi* namespace (/src/slmpi.h):

The slmpi namespace provides an interface to MPI (message passing interface).

`bool slmpi::isavailable(void)`

bool ismpiavailable = slmpi:: isavailable ();

Returns true if sparselizard was compiled with MPI and false otherwise.

`void slmpi::initialize(void)`

slmpi:: initialize ();
slmpi:: finalize ();

This initializes MPI. It must be called before any MPI operation.

`void slmpi::finalize(void)`

// See initialize example

This finalizes MPI. It must be called before the end of the program.

`int slmpi::getrank(void)`

slmpi:: initialize ();
int rank = slmpi::getrank();
slmpi:: finalize ();

This returns the rank of the current process.

`int slmpi::count(void)`

slmpi:: initialize ();
int numranks = slmpi::count();
slmpi:: finalize ();

This returns the number of processes in the universe.

`void slmpi::barrier(void)`

```
slmpi:: initialize ();
slmpi:: barrier ();
slmpi:: finalize ();
```

This is a collective MPI operation (must be called by all ranks). Processes will be waiting until all have reached this call.

`void slmpi::send(int destination, int tag, std::vector<int/double>& data)`

```
slmpi:: initialize ();
int rank = slmpi::getrank();
if (rank == 0)
{
        std :: vector<int> senddata = {1,2,3};
        slmpi::send(1, 0, senddata);
}
if (rank == 1)
{
        std :: vector<int> recvdata(3);
        slmpi:: receive (0, 0, recvdata);
        printvector(recvdata);
}
slmpi:: finalize ();
```

This sends a vector to the destination rank. A message tag can be provided to identify the send call.

`void slmpi::receive(int source, int tag, std::vector<int/double>& data)`

```
// See send example
```

This receives a vector from the source rank. The vector must be preallocated to the size to receive. The same tag used by the corresponding send call must be used.

`void slmpi::sum(std::vector<int/double>& data)`

```
slmpi:: initialize ();
std :: vector<int> data = {2,5};
slmpi::sum(data);
slmpi:: finalize ();
```

This is a collective MPI operation (must be called by all ranks). Each value in the vector is summed on all ranks and returned to all ranks.

`void slmpi::max(std::vector<int/double>& data)`

```
slmpi:: initialize ();
std::vector<int> data = {slmpi::getrank(), 4};
slmpi::max(data);
slmpi:: finalize ();
```

This is a collective MPI operation (must be called by all ranks). Each value in the vector is maxed on all ranks and returned to all ranks.

`void slmpi::min(std::vector<int/double>& data)`

```
slmpi:: initialize ();
std::vector<int> data = {slmpi::getrank(), 4};
slmpi::min(data);
slmpi:: finalize ();
```

This is a collective MPI operation (must be called by all ranks). Each value in the vector is mined on all ranks and returned to all ranks.

`void slmpi::broadcast(int broadcaster, std::vector<int/double>& data)`

```
slmpi:: initialize ();
int rank = slmpi::getrank();
std::vector<int> data(5);
if (rank == 0)
{
        for (int i = 0; i < data.size(); i++)
                data[i] = i;
}
slmpi::broadcast(0, data);
if (rank == 1)
        printvector(data);
slmpi:: finalize ();
```

This is a collective MPI operation (must be called by all ranks). The data vector on the broadcaster rank is sent to all other ranks. It must be preallocated on each rank to the size on the broadcaster.

`void slmpi::gather(int gatherer, std::vector<int/double>& fragment, std::vector<int/double>& gathered)`

```
slmpi:: initialize ();
int rank = slmpi::getrank();
std::vector<int> fragment = {rank, rank+1};
std::vector<int> gathered;
```

```
slmpi::gather(0, fragment, gathered);
if (rank == 0)
        printvector(gathered);
slmpi:: finalize ();
```

This is a collective MPI operation (must be called by all ranks). This gathers fixed-size fragments on the gatherer rank.

`void slmpi::gather(int gatherer, std::vector<int/double>& fragment, std::vector<int/double>& gathered, std::vector<int>& fragsizes)`

```
slmpi:: initialize ();
int rank = slmpi::getrank();
int numranks = slmpi::count();
std::vector<int> fragment(0);
if (rank == 0)
        fragment = {rank};
if (rank == 1)
        fragment = {rank, rank+1};
std::vector<int> fragsizes;
if (rank == 0)
{
        fragsizes = std::vector<int>(numranks, 0);
        fragsizes [0] = 1;
        fragsizes [1] = 2;
}
std::vector<int> gathered;
slmpi::gather(0, fragment, gathered, fragsizes );
if (rank == 0)
        printvector(gathered);
slmpi:: finalize ();
```

This is a collective MPI operation (must be called by all ranks). This gathers variable-size fragments on the gatherer rank. The fragment size vector is ignored on all but the gatherer rank.

`void slmpi::allgather(std::vector<int/double>& fragment, std::vector<int/double>& gathered)`

```
slmpi:: initialize ();
int rank = slmpi::getrank();
std::vector<int> fragment = {rank, rank+1};
```

```
std :: vector<int> gathered;
slmpi :: allgather (fragment, gathered);
if (rank == 1)
        printvector (gathered);
slmpi :: finalize ();
```

This is a collective MPI operation (must be called by all ranks). This gathers fixed-size fragments from all ranks on all ranks.

```
void slmpi::allgather(std::vector<int/double>& fragment,
std::vector<int/double>& gathered, std::vector<int>& fragsizes)
```

```
slmpi :: initialize ();
int rank = slmpi::getrank();
int numranks = slmpi::count();
std :: vector<int> fragment;
if (rank == 0)
        fragment = {rank};
if (rank == 1)
        fragment = {rank, rank+1};
std :: vector<int> fragsizes(numranks, 0);
fragsizes [0] = 1;
fragsizes [1] = 2;
std :: vector<int> gathered;
slmpi :: allgather (fragment, gathered, fragsizes );
if (rank == 1)
        printvector (gathered);
slmpi :: finalize ();
```

This is a collective MPI operation (must be called by all ranks). This gathers variable-size fragments from all ranks on all ranks. The fragment size vector must be provided on all ranks.

```
void slmpi::scatter(int scatterer, std::vector<int/double>& toscatter,
std::vector<int/double>& fragment)
```

```
slmpi :: initialize ();
int rank = slmpi::getrank();
int numranks = slmpi::count();
std :: vector<int> toscatter(2*numranks);
if (rank == 0)
{
```

```
        for (int i = 0; i < toscatter.size(); i++)
                toscatter[i] = i;
}
std::vector<int> fragment(2);
slmpi::scatter(0, toscatter, fragment);
if (rank == 1)
        printvector(fragment);
slmpi::finalize();
```

This is a collective MPI operation (must be called by all ranks). This scatters fixed-size fragments from the vector to scatter on the scatterer rank to all ranks. The vector to scatter is ignored on all but the scatterer rank. The fragment vector must be preallocated to the correct size on each rank.

`void slmpi::scatter(int scatterer, std::vector<int/double>& toscatter, std::vector<int/double>& fragment, std::vector<int>& fragsizes)`

```
slmpi::initialize();
int rank = slmpi::getrank();
int numranks = slmpi::count();
std::vector<int> toscatter, fragsizes;
if (rank == 0)
{
        toscatter = {1,2,3};
        fragsizes = std::vector<int>(numranks, 0);
        fragsizes[0] = 1;
        fragsizes[1] = 2;
}
std::vector<int> fragment(0);
if (rank == 0)
        fragment = std::vector<int>(1);
if (rank == 1)
        fragment = std::vector<int>(2);
slmpi::scatter(0, toscatter, fragment, fragsizes);
if (rank == 1)
        printvector(fragment);
slmpi::finalize();
```

This is a collective MPI operation (must be called by all ranks). This scatters variable-size fragments from the vector to scatter on the scatterer rank to all ranks. The vector to scatter and the fragment size vector are ignored on all but the scatterer rank. The fragment vector must be preallocated to the correct size on each rank.

`void slmpi::exchange(std::vector<int> targetranks, std::vector<int/double>& sendvalues, std::vector<int/double>& receivevalues)`

```
slmpi:: initialize ();
int rank = slmpi::getrank();
int numranks = slmpi::count();
std::vector<int> sendvalues = {rank, rank};
std::vector<int> receivevalues;
slmpi::exchange({(rank−1+numranks)%numranks, (rank+1)%numranks}, sendvalues, receivevalues);
if (rank == 1)
        printvector( receivevalues );
slmpi:: finalize ();
```

This MPI operation must be called by all ranks. This exchanges a fixed number of values with each **unique** target rank. The number of targets can change from a rank to another.

`void slmpi::exchange(std::vector<int> targetranks, std::vector<std::vector<int/double>>& sends, std::vector<std::vector<int/double>>& receives)`

```
slmpi:: initialize ();
int rank = slmpi::getrank();
int numranks = slmpi::count();
std::vector<std::vector<int>> sendvalues = {{rank}, {rank}};
std::vector<std::vector<int>> receivevalues(2, std::vector<int>(1));
slmpi::exchange({(rank−1+numranks)%numranks, (rank+1)%numranks}, sendvalues, receivevalues);
if (rank == 1)
        printvector( receivevalues [0]);
slmpi:: finalize ();
```

This MPI operation must be called by all ranks. This exchanges a variable number of values with each **unique** target rank. The send and receive vectors must be preallocated to the correct size on each rank for each target. The number of targets can change from a rank to another.

3.14 The *spanningtree* object (/src/mesh/spanningtree.h):

The spanningtree object holds a spanning tree whose edges go through every node of the mesh without forming a loop.

`spanningtree::spanningtree(std::vector<int> physregs)`

mesh mymesh("disk.msh");
int vol = 1, sur = 2, top = 3;
spanningtree spantree({sur,top});

This creates a spanning tree whose edges go through all nodes in the mesh without forming a loop. The tree growth starts on the physical regions provided as argument. Here the tree is first fully grown on face regions 'sur' and 'top' before being extended everywhere.
For large trees it might be required to increase the stack size (e.g. ulimit -s 131072 on Linux) because the tree is created using recursive calls leading to segmentation faults when the stack size is exceeded.

`int spanningtree::countedgesintree(void)`

...
std::cout << spantree.countedgesintree() << std::endl;

This returns the number of edges in the spanning tree.

`void spanningtree::write(std::string filename)`

...
spantree.write("spantree.vtk");

This writes the tree to a file for visualization.

3.15 The *spline* object (/src/shapefunction/spline.h):

The spline object allows to interpolate in a discrete data set using cubic (natural) splines.

`spline::spline(std::string filename, char delimiter = '\n')`

spline spl("measureddata.txt");

This creates the spline object from x-y data contained in a text file. Using a ',' delimiter (default is newline) the text file format must be:
x1,y1,x2,y2,...,xn,yn (e.g. 273,5e9,300,4e9,320,2.5e9,340,1e9 for measured temperature-Young's modulus samples).
Data requested outside of the provided sample range (i.e. temperatures between 273 and 340 Kelvins in the example) is extrapolated linearly using the spline-end slope. The samples ordering does not matter.

`spline::spline(std::vector<double> xin, std::vector<double> yin)`

std::vector<double> temperature = {273,300,320,340};
std::vector<double> youngsmodulus = {5e9,4e9,2.5e9,1e9};
spline spl(temperature, youngsmodulus);

This creates the spline object from x-y data provided in two separate vectors.
Data requested outside of the provided sample range (i.e. temperatures between 273 and 340 Kelvins in the example) is extrapolated linearly using the spline-end slope. The samples ordering does not matter.

`spline spline::getderivative(void)`

...
spline dspl = spl.getderivative();

This returns the spline derivative.

`double spline::getxmin(void)`

...
double xminval = spl.getxmin();

This returns the min value that the temperature takes in the data provided.

`double spline::getxmax(void)`

...
double xmaxval = spl.getxmax();

This returns the max value that the temperature takes in the data provided.

`double spline::evalat(double input)`

...
double val = spl.evalat(340);

This allows to interpolate at a given point.

`std::vector<double> spline::evalat(std::vector<double> input)`

...
std::vector<double> temperat = {298,304,275};
std::vector<double> youngsmodevaled = spl.evalat(temperat);
printvector(youngsmodevaled);

This allows to interpolate at a large number of points.

`void spline::write(std::string filename, int numsplits, char delimiter = '\n')`

...
spl.write("finersampling.txt", 5);

This writes to file a refined version of the original data samples in the same format as the format to load. It can be used to visualize the interpolation obtained with the cubic splines.

3.16 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::vec(formulation formul)`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2;
field v("h1");
v.setorder(vol, 1);
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'.

`vec::vec(int vecsize, indexmat addresses, densemat vals)`

```
indexmat addresses(3, 1, {0,1,2});
densemat vals(3, 1, {5,10,20});

vec b(3, addresses, vals);
b.print();
```

This creates a vector with given values at given addresses.

`int vec::size(void)`

```
mesh mymesh("disk.msh");
int vol = 1, sur = 2;
field v("h1");
v.setorder(vol, 1);
formulation projection;
projection += integral(vol, dof(v)*tf(v) - 2*tf(v));
vec b(projection);
int mysize = b.size();
```

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

`void vec::updateconstraints(void)`

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

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

`void vec::setvalues(indexmat addresses, densemat valsmat, std::string op = "set")`

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

vec myvec(projection);

indexmat addresses(myvec.size(), 1, 0, 1);
densemat vals(myvec.size(), 1, 12);

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

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

`void vec::setallvalues(densemat valsmat, std::string op = "set")`

```
...
myvec.setallvalues(vals);
```

This is similar to the previous one but works on all entries.

`void vec::setvalue(int address, double value, std::string op = "set")`

```
...
myvec.setvalue(2, 2.32);
```

This is similar to the previous one but works on a single entry.

`densemat vec::getvalues(indexmat addresses)`

```
...
```

densemat vecvals = myvec.getvalues(addresses);

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

`densemat vec::getallvalues(void)`

...
densemat allvecvals = myvec.getallvalues();

This is similar to the previous one but works on all entries.

`double vec::getvalue(int address)`

...
double vecval = myvec.getvalue(2);

This is similar to the previous one but works on a single entry.

`void vec::setvalue(port prt, double value, std::string op = "set")`

mesh mymesh("disk.msh");
port A, B;
formulation linearsystem;
linearsystem += A − 1.0;
linearsystem += B − 1.0;

vec myvec(linearsystem);
myvec.setvalue(B, 5.5);
myvec.print();

This replaces the value in 'myvec' at the address of the port by the value provided. With "add" the value is not replaced but added.

`double vec::getvalue(port prt)`

...
double val = myvec.getvalue(B);

This gets the value in the vector that is at the address of the port.

`void vec::setdata(int physreg, field myfield, std::string op = "set")`

mesh mymesh("disk.msh");
int vol = 1;
field v("h1"), w("h1");

```
v.setorder(vol, 1);
w.setorder(vol, 1);
formulation projection;
projection += integral(vol, dof(v)*tf(v) - 2*tf(v) );
vec sol(projection);

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

The last line transfers the data of field v to the addresses of field v in vector sol on region number 1. The line before allows to transfer data with different origin and destination fields: the data of field w is transferred to the addresses in vector sol that correspond to field v. This only works if v and w are of the same type (here they are both "h1" type). In case w has a higher interpolation order than v the higher order dofs of w are not transferred to the vector. In the opposite case the addresses in vector sol of the higher order dofs of v are zeroed. The (optional) third argument makes it possible to either *set* the value of the vector to the field value (with "set", the default choice) or *add* the field value to the vector (with "add").

`void vec::setdata(void)`

```
...
sol.setdata();
```

This transfers to the vector the data from all fields and ports defined in the formulation associated to the vector.

`void vec::noautomaticupdate(void)`

```
...
sol.noautomaticupdate();
```

After this call the vec will not have its value automatically updated after hp-adaptivity. If the automatic update is not needed then this call is recommended to avoid a possibly costly vec value update.

`Vec vec::getpetsc(void)`

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

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

`void vec::write(std::string filename)`

```
...
```

sol.write("vecdata.bin");

This writes all data in a vec object to disk in a lossless, compact form. The file can be written in binary .bin format (extremely compact but less portable) or in ASCII .txt format (portable).

`void vec::load(std::string filename)`

...
vec loadedvec(projection);
loadedvec.load("vecdata.bin");

This loads the data of the vec object from disk (from .bin binary or .txt ASCII format). This only works correctly if the dof structure of the calling vec object is the exact same as the dof structure of the vec object from which the file was written to disk. In other words the same set of formulation contributions must be defined in the same order (but all generation and resolution steps can be skipped).

`void vec::print(void)`

...
sol.print();

This prints the vector values.

`vec vec::copy(void)`

...
vec copiedvec = sol.copy();

This creates a full copy of the vector.

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

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

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

`double vec::sum(void)`

...
double mysum = sol.sum();

This outputs the sum of all values in the vector.

`vec vec::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.

`void vec::permute(indexmat rowpermute, bool invertit = false)`

indexmat rows(6,1, { 0,1,2,3,4,5 });
densemat vals(6,1, { 10,20,30,40,50,60 });
vec v(6, rows, vals);
v.print();
indexmat permuterows(6,1, { 5,4,3,2,1,0 });
v.permute(permuterows);
v.print();

This permutes the vector. The inverse permutation is used if the boolean flag is set to true.

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

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

`wallclock::wallclock(void)`

wallclock myclock;

This initialises the wall clock object.

`void wallclock::tic(void)`

wallclock myclock;
myclock.tic();

This resets the clock.

`double wallclock::toc(void)`

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

This returns the time elapsed (in ns).

`void wallclock::print(std::string toprint = "")`

```
wallclock myclock;
myclock.print("Time elapsed:"); // or myclock.print()
```

This prints the time elapsed in the most appropriate format (ns, us, ms or s).
It also prints the message in the argument string (if any).

`void wallclock::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 wallclock::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.