GooseFEM Documentation

Tom de Geus

May 16, 2020

INTRODUCTION

CHAPTER 1

Introduction

1.1 Overview

This header-only module provides C++ classes (including Python wrappers) to write simulations using the Finite Element Method.

Note: This library is free to use under the GPLv3 license. Any additions are very much appreciated, in terms of suggested functionality, code, documentation, testimonials, word of mouth advertisement, ... Bug reports or feature requests can be filed on GitHub. As always, the code comes with no guarantee. None of the developers can be held responsible for possible mistakes.

1.2 Hallmark feature

The hallmark feature of GooseFEM is that data is not stored in GooseFEM's classes but only in multi-dimensional arrays with certain *storage conventions*. Consequently one is entirely free to mix and match routines, from GooseFEM or even from somewhere else. Consequently the readability of the end-user's code and of the library remains high: Only the main function combines different ingredients, there is no interdependence between GooseFEM's classes.

1.3 Documentation

This document should be considered as a quick-start guide. A lot effort has been spent on the readability of the code itself (in particular the `.h` files should be instructive). One is highly encouraged to answer more advanced questions that arise from this guide directly using the code. Download buttons to the relevant files are included throughout this reader.

A compact reader covering the basic theory is available as `PDF`.

CHAPTER 2

Getting GooseFEM

2.1 Using conda

The easiest is to use *conda* to install *GooseFEM*:

```
conda install -c conda-forge goosefem
```

2.2 From source

Download the package:

```
git checkout https://github.com/tdegeus/GooseFEM.git
cd GooseFEM
```

Install headers, *CMake* and *pkg-config* support:

```
cmake .
make install
```

2.3 Python interface

2.3.1 Using conda

The quickest (but not the most efficient!) is to use *conda* to install *GooseFEM*:

```
conda install -c conda-forge python-goosefem
```

Warning: This package does not benefit from *xsimd* optimisation, as it is not compiled on your hardware. Therefore compiling by hand is recommended.

2.3.2 From source

Start by installing the dependencies, for example using *conda*:

```
conda install -c conda-forge pyxtensor eigen xsimd
```

Note that *xsimd* is optional, but recommended.

Note: You can also use:

```
python -m pip install pyxtensor pybind11
```

for use without *conda*. Note that you install *Eigen* and *xsimd* yourself in such a way that Python can find it in order to use it.

Then, download the package:

```
git checkout https://github.com/tdegeus/GooseFEM.git
cd GooseFEM
```

Install the package using:

```
python -m pip install .
```

Note: The following will give more readable output:

```
python setup.py build
python setup.py install
```

CHAPTER 3

Compiling

3.1 Introduction

This module is header only. So one just has to `#include <GooseFEM/GooseFEM.h>`, and to tell the compiler where the header-files are.

3.2 Using CMake

The following structure can be used for your project's `CMakeLists.txt`:

```
find_package(GooseFEM REQUIRED)

add_executable(myexec mymain.cpp)

target_link_libraries(myexec PRIVATE
    GooseFEM
    xtensor::optimize
    xtensor::use_xsimd)
```

See the documentation of xtensor concerning optimisation.

Note: There are additional targets available to expedite your `CMakeLists.txt`:

- `GooseFEM::assert`: enable GooseFEM assertions.
- `GooseFEM::debug`: enable GooseFEM and xtensor assertions (slow).
- `GooseFEM::compiler_warnings`: enable compiler warnings.

3.3 By hand

Presuming that the compiler is `c++`, compile using:

```
c++ -I/path/to/GooseFEM/include ...
```

Note that you have to take care of the *xtensor* and *Eigen* dependencies, the C++ version, optimisation, enabling *xsimd*, ...

3.4 Using pkg-config

Find the location of the headers can be automatised using *pkg-config*:

```
pkg-config --cflags GooseFEM
```

CHAPTER 4

Linear elastic

Consider a uniform linear elastic bar that is extended by a uniform fixed displacement on both sides. This problem can be modelled and discretised using symmetry as show below. In this example we will furthermore assume that the bar is sufficiently thick in the out-of-plane direction to be modelled using two-dimensional plane strain.

Below an example is described line-by-line. The full example can be downloaded:

`CMakeLists.txt`
`example.cpp`
`plot.py`

Note: This example is also available using the Python interface (`example.py`). Compared to the C++ API, the Python API requires more data-allocation, in particular for the functions *AsElement* and *AssembleNode*. See: *Data allocation*.

4.1 Include library

```
#include <GooseFEM/GooseFEM.h>
#include <GooseFEM/MatrixPartitioned.h>
#include <GMatElastic/Cartesian3d.h>
#include <highfive/H5Easy.hpp>
```

The first step is to include the header-only library. Note that for this example we also make use of a material model (GMatElastic) and a library to write (and read) HDF5 files (HighFive).

4.2 Define mesh

```
    // define mesh
    GooseFEM::Mesh::Quad4::Regular mesh(5, 5);

    // mesh dimensions
    size_t nelem = mesh.nelem();
    size_t nne = mesh.nne();
    size_t ndim = mesh.ndim();

    // mesh definitions
    xt::xtensor<double,2> coor = mesh.coor();
    xt::xtensor<size_t,2> conn = mesh.conn();
    xt::xtensor<size_t,2> dofs = mesh.dofs();

    // node sets
    xt::xtensor<size_t,1> nodesLft = mesh.nodesLeftEdge();
    xt::xtensor<size_t,1> nodesRgt = mesh.nodesRightEdge();
    xt::xtensor<size_t,1> nodesTop = mesh.nodesTopEdge();
    xt::xtensor<size_t,1> nodesBot = mesh.nodesBottomEdge();
```

A mesh is defined using GooseFEM. As observed the *mesh* is a class that has methods to extract the relevant information such as the nodal coordinates (*coor*), the connectivity (*conn*), the degrees-of-freedom per node (*dofs*) and several node-sets that will be used to impose the sketched boundary conditions (*nodesLeft*, *nodesRight*, *nodesTop*, *nodesBottom*).

Note that:

- The connectivity (*conn*) contains information of which nodes, in which order, belong to which element.
- The degrees-of-freedom per node (*dofs*) contains information of how a nodal vector (a vector stored per node) can be transformed to a list of degrees-of-freedom as used in the linear system (although this can be mostly done automatically as we will see below).

See also:

- *Terminology*
- Details: *Mesh::Quad4*

4.3 Define partitioning

```
    xt::xtensor<size_t,1> iip = xt::concatenate(xt::xtuple(
        xt::view(dofs, xt::keep(nodesRgt), 0),
        xt::view(dofs, xt::keep(nodesTop), 1),
        xt::view(dofs, xt::keep(nodesLft), 0),
        xt::view(dofs, xt::keep(nodesBot), 1)));
```

We will reorder such that degrees-of-freedom are ordered such that

$$\mathtt{u} = \begin{bmatrix} \mathtt{u}_u \\ \mathtt{u}_p \end{bmatrix}$$

where the subscript u and p respectively denote *Unknown* and *Prescribed* degrees-of-freedom. To achieve this we start by collecting all prescribed degrees-of-freedom in *iip*.

4.4 (Avoid) Book-keeping

```
GooseFEM::VectorPartitioned vector(conn, dofs, iip);
```

To switch between the three of GooseFEM's data-representations, an instance of the *Vector* class is used. This instance, *vector*, will enable us to switch between a vector field (e.g. the displacement)

1. collected per node,
2. collected per degree-of-freedom, or
3. collected per element.

Note: The *Vector* class collects most, if not all, the burden of book-keeping. It is thus here that *conn*, *dofs*, and *iip* are used. In particular,

- 'nodevec' $\leftrightarrow$ 'dofval' using *dofs* and *iip*,
- 'nodevec' $\leftrightarrow$ 'elemvec' using *conn*.

By contrast, most of GooseFEM's other methods receive the relevant representation, and consequently require no problem specific knowledge. They thus do not have to supplied with *conn*, *dofs*, or *iip*.

See also:

- *Vector representation*
- *Data storage*
- Details: *Vector*

4.5 System matrix

```
GooseFEM::MatrixPartitioned<> K(conn, dofs, iip);
```

We now also allocate the system/stiffness system (stored as sparse matrix). Like vector, it can accept and return different vector representations, in addition to the ability to assemble from element system matrices.

Note: Here, the default solver is used (which is the default template, hence the "<>"). To use other solvers see: *Linear solver*.

See also:

- *Matrix representation*
- Details: *Matrix*

4.6 Allocate nodal vectors

```
xt::xtensor<double,2> disp = xt::zeros<double>(coor.shape());
xt::xtensor<double,2> fint = xt::zeros<double>(coor.shape());
xt::xtensor<double,2> fext = xt::zeros<double>(coor.shape());
xt::xtensor<double,2> fres = xt::zeros<double>(coor.shape());
```

- *disp*: nodal displacements
- *fint*: nodal internal forces
- *fext*: nodal external forces
- *fres*: nodal residual forces

4.7 Allocate element vectors

```
xt::xtensor<double,3> ue = xt::empty<double>({nelem, nne, ndim});
xt::xtensor<double,3> fe = xt::empty<double>({nelem, nne, ndim});
xt::xtensor<double,3> Ke = xt::empty<double>({nelem, nne * ndim, nne * ndim});
```

- *ue*: displacement
- *fe*: force
- *Ke*: tangent matrix

Warning: Upsizing (e.g. *disp* → *ue*) can be done uniquely, but downsizing (e.g. *fe* → *fint*) can be done in more than one way, see *Conversion*. We will get back to this point below.

4.8 Element definition

```
GooseFEM::Element::Quad4::QuadraturePlanar elem(vector.AsElement(coor));
size_t nip = elem.nip();
```

At this moment the interpolation and quadrature is allocated. The shape functions and integration points (that can be customised) are stored in this class. As observed, no further information is needed than the number of elements and the nodal coordinates per element. Both are contained in the output of "vector.AsElement(coor)", which is an 'elemvec' of shape "[nelem, nne, ndim]". This illustrates that problem specific book-keeping is isolated to the main program, using *Vector* as tool.

Note: The shape functions are computed when constructing this class, they are not recomputed when evaluating them. One can recompute them if the nodal coordinates change using ".update_x(...)", however, this is only relevant in a large deformation setting.

See also:

- *Vector representation*
- *Data storage*
- Details: *Vector*

- Details: *Element::Quad4*

4.9 Material definition

```
    GMatElastic::Cartesian3d::Matrix mat(nelem, nip, 1.0, 1.0);
```

We now define a uniform linear elastic material, using an external library that is tuned to GooseFEM. This material library will translate a strain tensor per integration point to a stress tensor per integration point and a stiffness tensor per integration point.

See also:

Material libraries tuned to GooseFEM include:

- GMatElastic
- GMatElastoPlastic
- GMatElastoPlasticFiniteStrainSimo
- GMatElastoPlasticQPot
- GMatElastoPlasticQPot3d
- GMatNonLinearElastic

But other libraries can also be easily used with (simple) wrappers.

4.10 Integration point tensors

```
    xt::xtensor<double,4> Eps = xt::empty<double>({nelem, nip, 3ul, 3ul});
    xt::xtensor<double,4> Sig = xt::empty<double>({nelem, nip, 3ul, 3ul});
    xt::xtensor<double,6> C = xt::empty<double>({nelem, nip, 3ul, 3ul, 3ul, 3ul});
```

These strain, stress, and stiffness tensors per integration point are now allocated. Note that these tensors are 3-d while our problem was 2-d. This is thanks to the plane strain assumption, and the element definition that ignores all third-dimension components.

4.11 Compute strain

```
    vector.asElement(disp, ue);
    elem.symGradN_vector(ue, Eps);
```

The strain per integration point is now computed using the current nodal displacements (stored as 'elemvec' in *ue*) and the gradient of the shape functions.

Note: *ue* is the output of "vector.asElement(disp, ue)". Using this syntax re-allocation of *ue* is avoided. If this optimisation is irrelevant for you problem (or if you are using the Python interface), please use the same function, but starting with a capital:

```
ue = vector.AsElement(disp);
```

Note that this allows the one-liner

```
Eps = elem.SymGradN_vector(vector.AElement(disp));
```

4.12 Compute stress and tangent

```
    mat.tangent(Eps, Sig, C);
```

The stress and stiffness tensors are now computed for each integration point (completely independently) using the external material model.

Note: *Sig* and *C* are the output variables that were preallocated in the main.

4.13 Assemble system

```
    // internal force
    elem.int_gradN_dot_tensor2_dV(Sig, fe);
    vector.assembleNode(fe, fint);

    // stiffness matrix
    elem.int_gradN_dot_tensor4_dot_gradNT_dV(C, Ke);
    K.assemble(Ke);
```

The stress stored per integration point (*Sig*) is now converted to nodal internal force vectors stored per element (*fe*). Using *vector* this 'elemvec' representation is then converted of a 'nodevec' representation in *fint*. Likewise, the stiffness tensor stored for integration point (*C*) are converted to system matrices stored per element ('elemmat') and finally assembled to the global stiffness matrix.

Warning: Please note that downsizing ($fe \rightarrow fint$ and $Ke \rightarrow K$) can be done in two ways, and that "assemble..." is the right function here as it adds entries that occur more than once. In contrast "as..." would not result in what we want here.

Note: Once more, *fe*, *fint*, and *Ke* are output variables. Less efficient, but shorter, is:

```
// internal force
fint = vector.AssembleNode(elem.Int_gradN_dot_tensor2_dV(Sig));

// stiffness matrix
K.assemble(elem.Int_gradN_dot_tensor4_dot_gradNT_dV(C));
```

4.14 Solve

```
    // set fixed displacements
    xt::view(disp, xt::keep(nodesRgt), 0) = +0.1;
    xt::view(disp, xt::keep(nodesTop), 1) = -0.1;
    xt::view(disp, xt::keep(nodesLft), 0) = 0.0;
    xt::view(disp, xt::keep(nodesBot), 1) = 0.0;

    // residual
    xt::noalias(fres) = fext - fint;

    // solve
    K.solve(fres, disp);
```

We now prescribe the displacement of the Prescribed degrees-of-freedom directly in the nodal displacements *disp* and compute the residual force. This is follows by partitioning and solving, all done internally in the *MatrixPartitioned* class.

4.15 Post-process

4.15.1 Strain and stress

```
    vector.asElement(disp, ue);
    elem.symGradN_vector(ue, Eps);
    mat.stress(Eps, Sig);
```

The strain and stress per integration point are recomputed for post-processing.

4.15.2 Residual force

```
    // internal force
    elem.int_gradN_dot_tensor2_dV(Sig, fe);
    vector.assembleNode(fe, fint);

    // apply reaction force
    vector.copy_p(fint, fext);

    // residual
    xt::noalias(fres) = fext - fint;

    // print residual
    std::cout << xt::sum(xt::abs(fres))[0] / xt::sum(xt::abs(fext))[0] << std::endl;
```

We convince ourselves that the solution is indeed in mechanical equilibrium.

4.15.3 Store & plot

```
    // average stress per node
    xt::xtensor<double,4> dV = elem.DV(2);
    xt::xtensor<double,3> SigAv = xt::average(Sig, dV, {1});

    // write output
    H5Easy::File file("output.h5", H5Easy::File::Overwrite);
```

(continues on next page)

(continued from previous page)

```
    H5Easy::dump(file, "/coor", coor);
    H5Easy::dump(file, "/conn", conn);
    H5Easy::dump(file, "/disp", disp);
    H5Easy::dump(file, "/Sig", SigAv);
```

Finally we store some fields for plotting using `plot.py`.

4.16 Manual partitioning

To verify how partitioning and solving is done internally using the *MatrixPartitioned* class, the same example is provided where partitioning is done manually:

`manual_partition.cpp`

CHAPTER 5

Semi-periodic

Consider a "rheometer" which corresponds to a torus, in which a linear elastic material is placed consists of two phases. In terms of boundary conditions this implies taking the geometry periodic in horizontal direction, while the displacements of the top and bottom boundary are controlled. For simplicity two-dimensional plane strain is considered.

Below the important difference with respect to the previous example are discussed. The full example can be downloaded:

`CMakeLists.txt`
`example.cpp`
`plot.py`

Note: The same example is available using the Python interface: `example.py`

5.1 Node sets

```
    xt::xtensor<size_t,1> nodesLft = mesh.nodesLeftOpenEdge();
    xt::xtensor<size_t,1> nodesRgt = mesh.nodesRightOpenEdge();
    xt::xtensor<size_t,1> nodesTop = mesh.nodesTopEdge();
    xt::xtensor<size_t,1> nodesBot = mesh.nodesBottomEdge();
```

We will apply periodicity to all the nodes along the left and right boundary of the geometry. The corner nodes are thereby assigned the fixed displacement (that is itself taken periodic).

5.2 Apply periodicity

```
    for (size_t j = 0; j < coor.shape(1); ++j) {
        xt::view(dofs, xt::keep(nodesRgt), j) = xt::view(dofs, xt::keep(nodesLft), j);
    }

    dofs = GooseFEM::Mesh::renumber(dofs);
```

Applying periodicity in this case is rather straightforward. In particular the degrees-of-freedom along the right edge are eliminated, and replaced by the degrees-of-freedom of the left edge. The size of the actually solved system is therefore reduced, while the response vectors are simply assembled to both sides of the geometry.

5.3 Fixed displacement

```
    xt::xtensor<size_t,1> iip = xt::concatenate(xt::xtuple(
        xt::view(dofs, xt::keep(nodesBot), 0),
        xt::view(dofs, xt::keep(nodesBot), 1),
        xt::view(dofs, xt::keep(nodesTop), 0),
        xt::view(dofs, xt::keep(nodesTop), 1)));
```

The degrees-of-freedom of which the displacement is controlled are finally extracted from the renumbered list of degrees-of-freedom.

CHAPTER 6

Inertia

6.1 Time discretisation: Verlet

```cpp
// position, velocity, acceleration (and history: last increment)
xt::xtensor<double,2> u   = xt::zeros<double>({nnode, ndim});
xt::xtensor<double,2> v   = xt::zeros<double>({nnode, ndim});
xt::xtensor<double,2> a   = xt::zeros<double>({nnode, ndim});
xt::xtensor<double,2> v_n = xt::zeros<double>({nnode, ndim});
xt::xtensor<double,2> a_n = xt::zeros<double>({nnode, ndim});

// residual force
xt::xtensor<double,2> fres = xt::zeros<double>({nnode, ndim});

// compute mass matrix
// (often assumed constant & diagonal, remove either assumption if needed)
GooseFEM::MatrixDiagonal M(...);

...

// time increments
for ( ... )
{
    // store history
    xt::noalias(v_n) = v;
    xt::noalias(a_n) = a;

    // new displacement
    xt::noalias(u) = u + dt * v + 0.5 * std::pow(dt, 2.0) * a;

    // new residual force (and mass matrix if needed)
    ...

    // new acceleration
    M.solve(fres, a);
```

(continues on next page)

(continued from previous page)

```
    // new velocity
    xt::noalias(v) = v_n + 0.5 * dt * (a_n + a);
}
```

6.2 Example

main.cpp
CMakeLists.txt
plot.py

CHAPTER 7

Inertia + damping

7.1 Velocity Verlet

```cpp
// position, velocity, acceleration (and history: last increment)
xt::xtensor<double,2> u   = xt::zeros<double>({nnode, ndim});
xt::xtensor<double,2> v   = xt::zeros<double>({nnode, ndim});
xt::xtensor<double,2> a   = xt::zeros<double>({nnode, ndim});
xt::xtensor<double,2> v_n = xt::zeros<double>({nnode, ndim});
xt::xtensor<double,2> a_n = xt::zeros<double>({nnode, ndim});

// residual force
xt::xtensor<double,2> fr = xt::zeros<double>({nnode, ndim});

// compute mass matrix
// (often assumed constant & diagonal, remove either assumption if needed)
GooseFEM::MatrixDiagonal M(...);

...

// time increments
for ( ... )
{
  // store history
  xt::noalias(v_n) = v;
  xt::noalias(a_n) = a;

  // new displacement
  xt::noalias(u) = u + dt * v + 0.5 * std::pow(dt,2.) * a;

  // update residual force (and mass matrix if needed)
  ...

  // estimate 1: new velocity
  xt::noalias(v) = v_n + dt * a_n;
```

(continues on next page)

(continued from previous page)

```
  // estimate 1: new residual force (and mass matrix if needed)
  ...

  // estimate 1: new acceleration
  M.solve(fr, a);

  // estimate 2: new velocity
  xt::noalias(v) = v_n + .5 * dt * ( a_n + a );

  // estimate 2: new residual force (and mass matrix if needed)
  ...

  // estimate 2: new acceleration
  M.solve(fr, a);

  // new velocity
  xt::noalias(v) = v_n + .5 * dt * ( a_n + a );

  // new residual force (and mass matrix if needed)
  ...

  // new acceleration
  M.solve(fr, a);
}
```

7.2 Example

main.cpp
CMakeLists.txt
plot.py

CHAPTER 8

Overdamped

8.1 Forward Euler

```
// position and velocity
xt::xtensor<double,2> u = xt::zeros<double>({nnode, ndim});
xt::xtensor<double,2> v = xt::zeros<double>({nnode, ndim});

// time increments
for ( ... )
{
  // new displacement
  xt::noalias(u) = u + dt * v;

  // new velocity based on residual force
  ...
}
```

CHAPTER 9

Terminology

9.1 Sizes

Alias	Description
nnode	number of nodes
ndim	number of dimensions
nelem	number of elements
nne	number of nodes per element
tdim	number of dimensions of a tensor

9.2 Arrays

Alias	Description
dofval	degrees of freedom
nodevec	nodal vector
elemvec	nodal vector stored per element
elemmat	matrix stored per element
qscalar	scalar stored per integration point
qtensor	tensor stored per integration point

9.3 Names

Alias	Description
coor	nodal coordinates
conn	connectivity

9.4 Elements

Alias	Description
Tri3	2-d triangular element (3 nodes)
Quad4	2-d quadrilateral element (4 nodes)
Hex8	3-d hexahedral element (8 nodes)

9.5 Coordinates

- Nodal coordinates: each node has one row
- Shape [*nnode*, *ndim*]
- Denoted: *coor*

9.6 Connectivity

- Node numbers per element: each element has one row
- Shape [*nelem*, *nne*]
- Denoted: *conn*

CHAPTER 10

Data storage

10.1 Glossary

Alias	Shape	Description
dofval	[ndof]	degrees of freedom
nodevec	[nnode, ndim]	nodal vector
elemvec	[nelem, nne, ndim]	nodal vector stored per element
elemmat	[nelem, nne*ndim, nne*ndim]	matrix stored per element
qscalar	[nelem, nip]	scalar stored per integration point
qtensor	[nelem, nip, tdim, tdim, . . .]	tensor stored per integration point

10.2 dofval

- Degrees-of-freedom
- Shape [ndof]
- `xt::xtensor<double,1>`

10.3 nodevec

- Nodal vectors
- Shape [nnode, ndim]
- `xt::xtensor<double,2>`

10.4 elemvec

- Nodal vectors stored per element
- Allows treatment of all elements independently, no connectivity needed
- Shape [nelem, nne, ndim]
- `xt::xtensor<double,3>`

10.5 elemmat

- Matrices stored per element
- Shape [nelem, nne*ndim, nne*ndim]
- `xt::xtensor<double,3>`

10.6 qscalar

- Scalars stored per integration point
- Shape [nelem, nip]
- `xt::xtensor<double,2>`

10.7 qtensor (2nd order)

- 2nd-order tensors stored per integration point
- For certain elements, the number of dimensions of the tensor can be larger than the number of dimensions of the element (tdim >= ndim)
- Shape [nelem, nip, tdim, tdim]
- `xt::xtensor<double,4>`

10.8 qtensor (4th order)

- 4th-order tensors stored per integration point
- For certain elements, the number of dimensions of the tensor can be larger than the number of dimensions of the element (tdim >= ndim)
- Shape [nelem, nip, tdim, tdim, tdim, tdim]
- `xt::xtensor<double,4>`

CHAPTER 11

Vector representation

In GooseFEM there are three ways to represent vectors. In particular, a vector field (e.g. the displacement) can be collected:

- per node (denoted "nodevec", *see below*),
- per degree-of-freedom (denoted "dofval", *see below*),
- per element (denoted "elemvec", *see below*).

Warning: Watch out with the conversion from one representation to the other as downsizing can be done in more than one way, see *Conversion*.

Consider a simple two-dimensional mesh of just two elements, and a displacement vector per node:

11.1 Collected per node (nodevec)

$$\texttt{disp} = \begin{bmatrix} u_x^{(0)} & u_y^{(0)} \\ u_x^{(1)} & u_y^{(1)} \\ u_x^{(2)} & u_y^{(2)} \\ u_x^{(3)} & u_y^{(3)} \\ u_x^{(4)} & u_y^{(4)} \\ u_x^{(5)} & u_y^{(5)} \end{bmatrix}$$

11.2 Collected per degree-of-freedom (dofval)

The following definition

$$\texttt{dofs} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \\ 10 & 11 \end{bmatrix}$$

gives:

$$\texttt{u} = \left[u_x^{(0)}\ u_y^{(0)}\ u_x^{(1)}\ u_y^{(1)}\ u_x^{(2)}\ u_y^{(2)}\ u_x^{(3)}\ u_y^{(3)}\ u_x^{(4)}\ u_y^{(4)}\ u_x^{(5)}\ u_y^{(5)}\right]^T$$

Whereby "dofs" can be used to:

- **Reorder** "u" such that is can be easily (even directly) partitioned. For example, consider that all x-coordinates are *Prescribed* and all y-coordinates are *Unknown*. In particular,

  $$\texttt{dofs} = \begin{bmatrix} 6 & 0 \\ 7 & 1 \\ 8 & 2 \\ 9 & 3 \\ 10 & 4 \\ 11 & 5 \end{bmatrix}$$

  gives

  $$\texttt{u} = \left[u_y^{(0)}\ u_y^{(1)}\ u_y^{(2)}\ u_y^{(3)}\ u_y^{(4)}\ u_y^{(5)}\ \ u_x^{(0)}\ u_x^{(1)}\ u_x^{(2)}\ u_x^{(3)}\ u_x^{(4)}\ u_x^{(5)}\right]^T = \left[\texttt{u}_u\ \texttt{u}_p\right]^T$$

  which allows

  $$\texttt{u}_u = \texttt{u[:6]}$$
  $$\texttt{u}_p = \texttt{u[6:]}$$

- **Eliminate** dependent nodes. For example, suppose that the displacement of all top nodes is equal to that of the bottom nodes. In this one could:

  $$\texttt{dofs} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} \quad \rightarrow \quad \texttt{u} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} \quad \leftrightarrow \quad \texttt{disp} = \begin{bmatrix} u_0 & u_1 \\ u_2 & u_3 \\ u_4 & u_5 \\ u_0 & u_1 \\ u_2 & u_3 \\ u_4 & u_5 \end{bmatrix}$$

Note: *Vector* applies the reordering itself. One does not need to change "dofs", but one simply supplies "iip".

11.3 Collected per element (elemvec)

For this example:

$$\texttt{conn} = \begin{bmatrix} 0 & 1 & 4 & 3 \\ 1 & 2 & 5 & 4 \end{bmatrix}$$

The storage per node proceeds in

$$\begin{aligned}\texttt{shape(ue)} &= [n_{\text{elements}} \times n_{\text{nodes-per-element}} \times n_{\text{dim}}] \\ &= [2 \times 4 \times 2]\end{aligned}$$

In particular:

$$\texttt{ue[0,:,:]} = \begin{bmatrix} u_x^{(0)} & u_y^{(0)} \\ u_x^{(1)} & u_y^{(1)} \\ u_x^{(4)} & u_y^{(4)} \\ u_x^{(3)} & u_y^{(3)} \end{bmatrix}$$

and

$$\texttt{ue[1,:,:]} = \begin{bmatrix} u_x^{(1)} & u_y^{(1)} \\ u_x^{(2)} & u_y^{(2)} \\ u_x^{(5)} & u_y^{(5)} \\ u_x^{(4)} & u_y^{(4)} \end{bmatrix}$$

11.4 Conversion

Conversion to a larger representation (upsizing) can always be done uniquely, however, conversion to a more compact representation (downsizing) can be done in two ways. In particular:

From	To	Function	Remarks
dofval	nodevec	asNode(...)	unique
dofval	elemvec	asElement(...)	unique
nodevec	elemvec	asElement(...)	unique
nodevec	dofval	asDofs(...)	overwrites reoccurring items
elemvec	dofval	asDofs(...)	overwrites reoccurring items
elemvec	nodevec	asNode(...)	overwrites reoccurring items
nodevec	dofval	assembleDofs(...)	adds reoccurring items
elemvec	dofval	assembleDofs(...)	adds reoccurring items
elemvec	nodevec	assembleNode(...)	adds reoccurring items

CHAPTER 12

Matrix representation

12.1 Element system matrix

The element system matrix collects individual system matrices as a multi-dimensional array of shape $[n_{\text{elements}} \times n_{\text{nodes-per-element}} n_{\text{dim}} \times n_{\text{nodes-per-element}} n_{\text{dim}}]$. An element system matrix

```
Ke = K[e,:,:]
```

obeys the following convention:

$$\underline{f} = \underline{\underline{K}}\underline{u}$$

where

$$f_{(n+id)} \equiv f_i^{(n)}$$

with n the node number, i the dimension, and d the number of dimensions. For example for a two-dimensional quadrilateral element

$$\underline{f} = \left[f_x^{(0)}, f_y^{(0)}, f_x^{(1)}, f_y^{(1)}, f_x^{(2)}, f_y^{(2)}, f_x^{(3)}, f_y^{(3)}\right]^T$$

CHAPTER 13

Data allocation

Most of GooseFEM's functions provided an interface to:

1. Allocate output arrays: names start with a **upper-case** letter. For example:

```
ue = GooseFEM::Vector::AsElement(disp);
```

2. Directly write to output arrays, without allocation them and copying them, by taking a pointer the externally allocated array as the last input argument(s): names start with a **lower-case** letter. For example:

```
GooseFEM::Vector::asElement(disp, ue);
```

Note: The Python API only provides option 1. Option 2 is only available in the C++ API.

CHAPTER 14

Macros

14.1 GOOSEFEM_ENABLE_ASSERT

To enable assertions use

```
#define GOOSEFEM_ENABLE_ASSERT
```

Note: Assertions are always turned on in the Python API.

CHAPTER 15

Mesh

`GooseFEM/Mesh.h`
`GooseFEM/Mesh.hpp`

15.1 Mesh::dofs

Get a sequential list of DOF-numbers for each vector-component of each node. For example for 3 nodes in 2 dimensions the output is

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

15.2 Mesh::Renumber

Renumber (DOF) indices to lowest possible indices. For example:

$$\begin{bmatrix} 0 & 1 \\ 5 & 4 \end{bmatrix}$$

is renumbered to

$$\begin{bmatrix} 0 & 1 \\ 3 & 2 \end{bmatrix}$$

Or, in pseudo-code, the result of this function is that:

```
dofs = renumber(dofs)

sort(unique(dofs[:])) == range(max(dofs+1))
```

Tip: One can use the wrapper function "GooseFEM::reorder" or the class "Mesh::Reorder" to get more advanced features.

15.3 Mesh::Reorder

Reorder (DOF) indices such to the lowest possible indices, such that some items are at the beginning or the end. For example:

$$\text{dofs} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

with

$$\text{idx} = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

Implies that `dofs` is renumbered such that 0 becomes the one-before-last index ($0 \rightarrow 4$), and the 1 becomes the last index ($1 \rightarrow 5$). The remaining items are renumbered to the lowest index while keeping the same order. The result:

$$\begin{bmatrix} 4 & 5 \\ 0 & 1 \\ 2 & 3 \end{bmatrix}$$

Tip: One can use the wrapper function "GooseFEM::reorder" or the class "Mesh::Reorder" to get more advanced features.

15.4 Mesh::coordination

Get the number of elements connected to each node.

15.5 Mesh::elem2node

Get the element numbers (columns) that are connected to each node (rows).

CHAPTER 16

Mesh::Tri3

```
GooseFEM/MeshTri3.h
GooseFEM/MeshTri3.hpp
```

16.1 Naming convention

16.2 Mesh::Tri3::Regular

Regular mesh.

See also:

```
Python - example
Python/ParaView - example
Python/ParaView - nodesets
```

16.2.1 Mesh::Tri3::Regular::nelem()

Return number of elements.

16.2.2 Mesh::Tri3::Regular::nnode()

Return number of nodes.

16.2.3 Mesh::Tri3::Regular::nne()

Return number of nodes-per-element (= 3).

16.2.4 Mesh::Tri3::Regular::ndim()

Return number of dimensions (= 2).

16.2.5 Mesh::Tri3::Regular::getElementType()

Return element-type. For example used in the *ParaView* module.

16.2.6 Mesh::Tri3::Regular::coor()

Return nodal coordinates [nnode, ndim].

16.2.7 Mesh::Tri3::Regular::conn()

Return connectivity [nelem, nne].

16.2.8 Mesh::Tri3::Regular::nodesXXXEdge()

Node numbers along the "Bottom", "Top", "Left", or "Right" edge.

16.2.9 Mesh::Tri3::Regular::nodesXXXOpenEdge()

Node numbers along the "Bottom", "Top", "Left", or "Right" edge, excluding the corners.

16.2.10 Mesh::Tri3::Regular::nodesXXXCorner()

Node number of one of the corners (e.g. "BottomLeft").

16.2.11 Mesh::Tri3::Regular::nodesPeriodic()

Periodic node pairs. Each row contains on pair of (independent, dependent) node numbers. The output shape is thus [n_pairs, 2].

16.2.12 Mesh::Tri3::Regular::nodesOrigin()

Bottom-left node, used as reference for periodicity.

16.2.13 Mesh::Tri3::Regular::dofs()

DOF-numbers for each component of each node (sequential). The output shape is thus [nnode, ndim].

16.2.14 Mesh::Tri3::Regular::dofsPeriodic()

DOF-numbers for each component of each node, for the case that the periodicity if fully eliminated. The output shape is thus [nnode, ndim].

16.3 Mesh::Tri3::getOrientation(...)

Get the orientation (-1 or +1) of all the elements.

16.4 Mesh::Tri3::setOrientation(...)

Set the orientation (-1 or +1) of all the elements.

CHAPTER 17

Mesh::Quad4

```
GooseFEM/MeshQuad4.h
GooseFEM/MeshQuad4.hpp
```

17.1 Naming convention

17.2 Mesh::Quad4::Regular

Regular mesh.

See also:

```
Python - example
Python/ParaView - example
Python/ParaView - nodesets
```

17.2.1 Mesh::Quad4::Regular::nelem()

Return number of elements.

17.2.2 Mesh::Quad4::Regular::nnode()

Return number of nodes.

17.2.3 Mesh::Quad4::Regular::nne()

Return number of nodes-per-element (= 3).

17.2.4 Mesh::Quad4::Regular::ndim()

Return number of dimensions (= 2).

17.2.5 Mesh::Quad4::Regular::getElementType()

Return element-type. For example used in the *ParaView* module.

17.2.6 Mesh::Quad4::Regular::coor()

Return nodal coordinates [nnode, ndim].

17.2.7 Mesh::Quad4::Regular::conn()

Return connectivity [nelem, nne].

17.2.8 Mesh::Quad4::Regular::nodesXXXEdge()

Node numbers along the "Bottom", "Top", "Left", or "Right" edge.

17.2.9 Mesh::Quad4::Regular::nodesXXXOpenEdge()

Node numbers along the "Bottom", "Top", "Left", or "Right" edge, excluding the corners.

17.2.10 Mesh::Quad4::Regular::nodesXXXCorner()

Node number of one of the corners (e.g. "BottomLeft").

17.2.11 Mesh::Quad4::Regular::nodesPeriodic()

Periodic node pairs. Each row contains on pair of (independent, dependent) node numbers. The output shape is thus [n_pairs, 2].

17.2.12 Mesh::Quad4::Regular::nodesOrigin()

Bottom-left node, used as reference for periodicity.

17.2.13 Mesh::Quad4::Regular::dofs()

DOF-numbers for each component of each node (sequential). The output shape is thus [nnode, ndim].

17.2.14 Mesh::Quad4::Regular::dofsPeriodic()

DOF-numbers for each component of each node, for the case that the periodicity if fully eliminated. The output shape is thus [nnode, ndim].

17.2.15 Mesh::Quad4::Regular::elementMatrix()

Return element numbers as matrix [nely, nelx].

17.3 Mesh::Quad4::FineLayer

Mesh with a fine layer in the middle, and that becomes course away from this plane (see image below). Note coursening can only be done if the number of elements in horizontal direction is dividable by 3, and that it is only optimal if the number of elements in horizontal direction is a factor of 3. Note that the number of elements in the vertical direction is specified as the number of times the unit element (the number of times "h" the height should be), and that this number is only a target: the algorithm chooses in accordance with the applied coursing.

See also:

```
Python - example
Python/ParaView - example
Python/ParaView - nodesets
Python - behaviour 'nx'
Python - element numbers
```

17.3.1 Mesh::Quad4::FineLayer::nelem()

Return number of elements.

17.3.2 Mesh::Quad4::FineLayer::nnode()

Return number of nodes.

17.3.3 Mesh::Quad4::FineLayer::nne()

Return number of nodes-per-element (= 3).

17.3.4 Mesh::Quad4::FineLayer::ndim()

Return number of dimensions (= 2).

17.3.5 Mesh::Quad4::FineLayer::nelx()

Number of elements in horizontal direction (along the weak layer) (matches input).

17.3.6 Mesh::Quad4::FineLayer::nely()

Actual number of elements unit elements in vertical direction (actual number of times "h" the mesh is heigh).

17.3.7 Mesh::Quad4::FineLayer::h()

Unit edge size (matches input).

17.3.8 Mesh::Quad4::FineLayer::getElementType()

Return element-type. For example used in the *ParaView* module.

17.3.9 Mesh::Quad4::FineLayer::coor()

Return nodal coordinates [nnode, ndim].

17.3.10 Mesh::Quad4::FineLayer::conn()

Return connectivity [nelem, nne].

17.3.11 Mesh::Quad4::FineLayer::nodesXXXEdge()

Node numbers along the "Bottom", "Top", "Left", or "Right" edge.

17.3.12 Mesh::Quad4::FineLayer::nodesXXXOpenEdge()

Node numbers along the "Bottom", "Top", "Left", or "Right" edge, excluding the corners.

17.3.13 Mesh::Quad4::FineLayer::nodesXXXCorner()

Node number of one of the corners (e.g. "BottomLeft").

17.3.14 Mesh::Quad4::FineLayer::nodesPeriodic()

Periodic node pairs. Each row contains on pair of (independent, dependent) node numbers. The output shape is thus [n_pairs, 2].

17.3.15 Mesh::Quad4::FineLayer::nodesOrigin()

Bottom-left node, used as reference for periodicity.

17.3.16 Mesh::Quad4::FineLayer::dofs()

DOF-numbers for each component of each node (sequential). The output shape is thus [nnode, ndim].

17.3.17 Mesh::Quad4::FineLayer::dofsPeriodic()

DOF-numbers for each component of each node, for the case that the periodicity if fully eliminated. The output shape is thus [nnode, ndim].

17.3.18 Mesh::Quad4::FineLayer::elementsMiddleLayer()

Element numbers of the middle, fine, layer

17.4 Mesh::Quad4::Map::RefineRegular

Refine a "Regular" mesh.

17.4.1 Mesh::Quad4::Map::RefineRegular::getCoarseMesh()

Return course mesh as "Mesh::Quad4::Regular".

17.4.2 Mesh::Quad4::Map::RefineRegular::getFineMesh()

Return fine mesh as "Mesh::Quad4::Regular".

17.4.3 Mesh::Quad4::Map::RefineRegular::getMap()

Elements of the fine mesh per element of the coarse mesh (rows).

17.4.4 Mesh::Quad4::Map::RefineRegular::mapToCoarse(...)

Map field to the course mesh:

- Scalar per element.
- Scalar per integration point.
- Tensor per integration point.

17.4.5 Mesh::Quad4::Map::RefineRegular::mapToFine(...)

Map field to the fine mesh:

- Scalar per element.
- Scalar per integration point.
- Tensor per integration point.

17.5 Mesh::Quad4::Map::FineLayer2Regular

Map "Regular" mesh to "FineLayer" mesh.

See also:

```
Python - map
Python - element numbers
```

17.5.1 Mesh::Quad4::Map::FineLayer2Regular::getCoarseMesh()

Return course mesh as "Mesh::Quad4::Regular".

17.5.2 Mesh::Quad4::Map::FineLayer2Regular::getFineMesh()

Return fine mesh as "Mesh::Quad4::Regular".

17.5.3 Mesh::Quad4::Map::FineLayer2Regular::getMap()

Elements of the fine mesh per element of the coarse mesh (rows).

17.5.4 Mesh::Quad4::Map::FineLayer2Regular::getMapFraction()

Get the fraction of overlap for the output of "getMap()".

17.5.5 Mesh::Quad4::Map::FineLayer2Regular::mapToRegular(...)

Map field to the course mesh:

- Scalar per element.
- Scalar per integration point.
- Tensor per integration point.

CHAPTER 18

Mesh::Hex8

`GooseFEM/MeshHex8.h`
`GooseFEM/MeshHex8.hpp`

18.1 Naming convention

The following naming convention is used:

- **Front**: all nodes whose coordinates $0 \leq x \leq L_x$, $0 \leq y \leq L_y$, $z = 0$.
- **Back**: all nodes whose coordinates $0 \leq x \leq L_x$, $0 \leq y \leq L_y$, $z = L_z$.
- **Bottom**: all nodes whose coordinates $0 \leq x \leq L_x$, $0 \leq z \leq L_z$, $y = 0$.
- **Top**: all nodes whose coordinates $0 \leq x \leq L_x$, $0 \leq z \leq L_z$, $y = L_y$.
- **Left**: all nodes whose coordinates $0 \leq y \leq L_y$, $0 \leq z \leq L_z$, $x = 0$.
- **Right**: all nodes whose coordinates $0 \leq y \leq L_y$, $0 \leq z \leq L_z$, $x = L_x$.

The edges and corners follow from the intersections, i.e.

- **FrontBottomEdge**: all nodes whose coordinates $0 \leq x \leq L_x$, $y = 0$, $z = 0$.
- ...
- **FrontBottomLeftCorner**: the node whose coordinate $x = 0$, $y = 0$, $z = 0$.
- ...

18.2 Mesh::Hex8::Regular

Regular mesh.

See also:

```
Python/ParaView - example
Python/ParaView - nodesets
```

18.2.1 Mesh::Hex8::Regular::nelem()

Return number of elements.

18.2.2 Mesh::Hex8::Regular::nnode()

Return number of nodes.

18.2.3 Mesh::Hex8::Regular::nne()

Return number of nodes-per-element (= 3).

18.2.4 Mesh::Hex8::Regular::ndim()

Return number of dimensions (= 2).

18.2.5 Mesh::Hex8::Regular::getElementType()

Return element-type. For example used in the *ParaView* module.

18.2.6 Mesh::Hex8::Regular::coor()

Return nodal coordinates [nnode, ndim].

18.2.7 Mesh::Hex8::Regular::conn()

Return connectivity [nelem, nne].

18.2.8 Mesh::Hex8::Regular::nodesXXXEdge()

Node numbers along the “Bottom”, “Top”, “Left”, or “Right” edge.

18.2.9 Mesh::Hex8::Regular::nodesXXXOpenEdge()

Node numbers along the “Bottom”, “Top”, “Left”, or “Right” edge, excluding the corners.

18.2.10 Mesh::Hex8::Regular::nodesXXXCorner()

Node number of one of the corners (e.g. “BottomLeft”).

18.2.11 Mesh::Hex8::Regular::nodesPeriodic()

Periodic node pairs. Each row contains on pair of (independent, dependent) node numbers. The output shape is thus [n_pairs, 2].

18.2.12 Mesh::Hex8::Regular::nodesOrigin()

Bottom-left node, used as reference for periodicity.

18.2.13 Mesh::Hex8::Regular::dofs()

DOF-numbers for each component of each node (sequential). The output shape is thus [nnode, ndim].

18.2.14 Mesh::Hex8::Regular::dofsPeriodic()

DOF-numbers for each component of each node, for the case that the periodicity if fully eliminated. The output shape is thus [nnode, ndim].

18.2.15 Mesh::Hex8::Regular::elementMatrix()

Return element numbers as matrix [nely, nelx].

18.3 Mesh::Hex8::FineLayer

Mesh with a fine layer in the middle, and that becomes course away from this plane (see image below). Note coursening can only be done if the number of elements in x- and y-direction is dividable by 3, and that it is only optimal if the number of elements in x- and y-direction is a factor of 3. Note that the number of elements in the vertical direction is specified as the number of times the unit element (the number of times "h" the height should be), and that this number is only a target: the algorithm chooses in accordance with the applied coursing.

See also:

```
Python/ParaView - example
Python/ParaView - nodesets
```

18.3.1 Mesh::Hex8::FineLayer::nelem()

Return number of elements.

18.3.2 Mesh::Hex8::FineLayer::nnode()

Return number of nodes.

18.3.3 Mesh::Hex8::FineLayer::nne()

Return number of nodes-per-element (= 3).

18.3.4 Mesh::Hex8::FineLayer::ndim()

Return number of dimensions (= 2).

18.3.5 Mesh::Hex8::FineLayer::nelx()

Number of elements in horizontal direction (along the weak layer) (matches input).

18.3.6 Mesh::Hex8::FineLayer::nely()

Actual number of elements unit elements in vertical direction (actual number of times "h" the mesh is heigh).

18.3.7 Mesh::Hex8::FineLayer::h()

Unit edge size (matches input).

18.3.8 Mesh::Hex8::FineLayer::getElementType()

Return element-type. For example used in the *ParaView* module.

18.3.9 Mesh::Hex8::FineLayer::coor()

Return nodal coordinates [nnode, ndim].

18.3.10 Mesh::Hex8::FineLayer::conn()

Return connectivity [nelem, nne].

18.3.11 Mesh::Hex8::FineLayer::nodesXXXEdge()

Node numbers along the "Bottom", "Top", "Left", "Right", "Front", or "Back" edge.

18.3.12 Mesh::Hex8::FineLayer::nodesXXXOpenEdge()

Node numbers along the "Bottom", "Top", "Left", "Right", "Front", or "Back" edge, excluding the corners.

18.3.13 Mesh::Hex8::FineLayer::nodesXXXCorner()

Node number of one of the corners (e.g. "FrontBottomLeft").

18.3.14 Mesh::Hex8::FineLayer::nodesPeriodic()

Periodic node pairs. Each row contains on pair of (independent, dependent) node numbers. The output shape is thus [n_pairs, 2].

18.3.15 Mesh::Hex8::FineLayer::nodesOrigin()

Bottom-left node, used as reference for periodicity.

18.3.16 Mesh::Hex8::FineLayer::dofs()

DOF-numbers for each component of each node (sequential). The output shape is thus [nnode, ndim].

18.3.17 Mesh::Hex8::FineLayer::dofsPeriodic()

DOF-numbers for each component of each node, for the case that the periodicity if fully eliminated. The output shape is thus [nnode, ndim].

18.3.18 Mesh::Hex8::FineLayer::elementsMiddleLayer()

Element numbers of the middle, fine, layer

CHAPTER 19

Element

```
GooseFEM/Element.h
GooseFEM/Element.hpp
```

19.1 Element::asElementVector

Convert nodal vector “[nnode, ndim]” to nodal vector stored per element “[nelem, nne, ndim]”.

19.2 Element::assembleNodeVector

Assemble nodal vector stored per element “[nelem, nne, ndim]” to nodal vector “[nnode, ndim]”.

19.3 Element::isSequential

Check that DOFs leave no holes.

19.4 Element::isDiagonal

Check structure of the matrices stored per element “[nelem, nne*ndim, nne*ndim]” to be diagonal (check that all off-diagonal entries have a value lower than a small numerical tolerance).

CHAPTER 20

Element::Quad4

```
GooseFEM/ElementQuad4.h
GooseFEM/ElementQuad4.hpp
GooseFEM/ElementQuad4Planar.h
GooseFEM/ElementQuad4Planar.hpp
GooseFEM/ElementQuad4Axisymmetric.h
GooseFEM/ElementQuad4Axisymmetric.hpp
```

20.1 Element::Quad4::Quadrature

Element definition to numerically interpolate and integrate.

Note: This function evaluates the shape function gradients upon construction, they are not recomputed upon evaluation. To evaluate them with respect to updated coordinates (e.g. to do updated Lagrange), use ".update_x(...)" to update the nodal coordinates and re-evaluate the shape function gradients and integration volumes.

Note: By default integration is done using Gauss points. To use a different scheme one has to supply the position (in isoparametric coordinates) and weight of the integration points (their number is inferred from the input).

Note: Most functions take the output as the last input-argument, as to write directly to a pre-allocated array, avoiding their re-allocation. All these functions have a wrapper that does the allocation for you (and thus returns the output rather than taking it as input). All function of this kind are indicated here with a *

20.1.1 Element::Quad4::Quadrature::update_x(...)

Update the nodal coordinates (elemvec: [nelem, nne, ndim]).

20.1.2 Element::Quad4::Quadrature::nelem()

Number of elements.

20.1.3 Element::Quad4::Quadrature::nne()

Number of nodes per element.

20.1.4 Element::Quad4::Quadrature::ndim()

Number of dimensions.

20.1.5 Element::Quad4::Quadrature::nip()

Number of integration points.

20.1.6 Element::Quad4::Quadrature::GradN()

(Current) Shape function gradient (w.r.t. real coordinates): [nelem, nip, nne, ndim]

20.1.7 Element::Quad4::Quadrature::dV(...)*

(Current) Volume of each integration point (qscalar: [nelem, nip]). An overload is available to get the same result as a tensor per integration point (qtensor: [nelem, nip, tdim, tdim]) with all tensor-components having the same value.

20.1.8 Element::Quad4::Quadrature::gradN_vector(...)*

Implementation of

$$\varepsilon = \vec{\nabla} N_m \vec{u}_m$$

or in index notation

$$\varepsilon_{ij} = \frac{\partial N_m}{\partial x_i} u_{mj}$$

20.1.9 Element::Quad4::Quadrature::gradN_vector_T(...)*

Implementation of

$$\varepsilon = \left(\vec{\nabla} N_m \vec{u}_m \right)^T$$

or in index notation

$$\varepsilon_{ji} = \frac{\partial N_m}{\partial x_i} u_{mj}$$

20.1.10 Element::Quad4::Quadrature::symGradN_vector(...)*

Implementation of

$$\varepsilon = \frac{1}{2}\left(\vec{\nabla} N_m \vec{u}_m + \left(\vec{\nabla} N_m \vec{u}_m\right)^T\right)$$

20.1.11 Element::Quad4::Quadrature::int_N_scalar_NT_dV(...)*

Implementation of

$$M_{mn} = \int_{\Omega^h} N_m \ \rho \ N_n \ \mathrm{d}\Omega^h \equiv \sum_q \ N_m \ \rho \ N_n \ \delta\Omega_q$$

Note that the output is an "elemmat", which has shape [nelem, nne*ndim, nne*ndim]. This implies that all dimensions are the same.

20.1.12 Element::Quad4::Quadrature::int_gradN_dot_tensor2_dV(...)*

Implementation of:

$$\vec{f}_m = \int_{\Omega^h} (\vec{\nabla} N_m) \cdot \sigma \ \mathrm{d}\Omega^h$$

or in index notation

$$f_{mj} = \sum_q \ \frac{\partial N_m}{\partial x_i} \sigma_{ij} \ \delta\Omega_q$$

20.1.13 Element::Quad4::Quadrature::int_gradN_dot_tensor4_dot_gradNT_dV(...)*

Implementation of:

$$K_{mn} = \int_{\Omega^h} (\vec{\nabla} N_m) \cdot \mathbb{C} \cdot \vec{\nabla} N_n \ \mathrm{d}\Omega^h$$

or in index notation

$$K_{m+id,n+kd} = \sum_q \ \frac{\partial N_m}{\partial x_i} C_{ijkl} \frac{\partial N_n}{\partial x_l} \ \delta\Omega_q$$

Note that the output is an "elemmat", which has shape [nelem, nne*ndim, nne*ndim].

20.2 Element::Quad4::QuadraturePlanar

Element definition to numerically interpolate and integrate under a planar assumption. This implies that all the tensors are 3-d, but that the third dimension is ignored by all functions (although for output these components are zero-initialised).

Note: This function evaluates the shape function gradients upon construction, they are not recomputed upon evaluation. To evaluate them with respect to updated coordinates (e.g. to do updated Lagrange), use ".update_x(...)" to update the nodal coordinates and re-evaluate the shape function gradients and integration volumes.

Note: By default integration is done using Gauss points. To use a different scheme one has to supply the position (in isoparametric coordinates) and weight of the integration points (their number is inferred from the input).

Note: Most functions take the output as the last input-argument, as to write directly to a pre-allocated array, avoiding their re-allocation. All these functions have a wrapper that does the allocation for you (and thus returns the output rather than taking it as input). All function of this kind are indicated here with a *

20.2.1 Element::Quad4::QuadraturePlanar::update_x(...)

Update the nodal coordinates (elemvec: [nelem, nne, ndim]).

20.2.2 Element::Quad4::QuadraturePlanar::nelem()

Number of elements.

20.2.3 Element::Quad4::QuadraturePlanar::nne()

Number of nodes per element.

20.2.4 Element::Quad4::QuadraturePlanar::ndim()

Number of dimensions.

20.2.5 Element::Quad4::QuadraturePlanar::nip()

Number of integration points.

20.2.6 Element::Quad4::QuadraturePlanar::GradN()

(Current) Shape function gradient (w.r.t. real coordinates): [nelem, nip, nne, ndim]

20.2.7 Element::Quad4::QuadraturePlanar::dV(...)*

(Current) Volume of each integration point (qscalar: [nelem, nip]). An overload is available to get the same result as a tensor per integration point (qtensor: [nelem, nip, tdim, tdim]) with all tensor-components having the same value.

20.2.8 Element::Quad4::QuadraturePlanar::gradN_vector(...)*

Implementation of

$$\varepsilon = \vec{\nabla} N_m \vec{u}_m$$

or in index notation

$$\varepsilon_{ij} = \frac{\partial N_m}{\partial x_i} u_{mj}$$

20.2.9 Element::Quad4::QuadraturePlanar::gradN_vector_T(...)*

Implementation of

$$\varepsilon = \left(\vec{\nabla} N_m \vec{u}_m \right)^T$$

or in index notation

$$\varepsilon_{ji} = \frac{\partial N_m}{\partial x_i} u_{mj}$$

20.2.10 Element::Quad4::QuadraturePlanar::symGradN_vector(...)*

Implementation of

$$\varepsilon = \tfrac{1}{2} \left(\vec{\nabla} N_m \vec{u}_m + \left(\vec{\nabla} N_m \vec{u}_m \right)^T \right)$$

20.2.11 Element::Quad4::QuadraturePlanar::int_N_scalar_NT_dV(...)*

Implementation of

$$M_{mn} = \int_{\Omega^h} N_m \; \rho \; N_n \; \mathrm{d}\Omega^h \equiv \sum_q \; N_m \; \rho \; N_n \; \delta\Omega_q$$

Note that the output is an "elemmat", which has shape [nelem, nne*ndim, nne*ndim]. This implies that all dimensions are the same.

20.2.12 Element::Quad4::QuadraturePlanar::int_gradN_dot_tensor2_dV(...)*

Implementation of:

$$\vec{f}_m = \int_{\Omega^h} (\vec{\nabla} N_m) \cdot \sigma \; \mathrm{d}\Omega^h$$

or in index notation

$$f_{mj} = \sum_q \; \frac{\partial N_m}{\partial x_i} \sigma_{ij} \; \delta\Omega_q$$

20.2.13 Element::Quad4::QuadraturePlanar::int_gradN_dot_tensor4_dot_gradNT_dV(...)*

Implementation of:

$$K_{mn} = \int_{\Omega^h} (\vec{\nabla} N_m) \cdot \mathbb{C} \cdot \vec{\nabla} N_n \, \mathrm{d}\Omega^h$$

or in index notation

$$K_{m+id,n+kd} = \sum_q \frac{\partial N_m}{\partial x_i} C_{ijkl} \frac{\partial N_n}{\partial x_l} \, \delta\Omega_q$$

Note that the output is an "elemmat", which has shape [nelem, nne*ndim, nne*ndim].

20.3 Element::Quad4::QuadratureAxisymmetric

Element definition to numerically interpolate and integrate in an axisymmetric cylindrical coordinate system. This implies that all tensors (stress, strain, stiffness, ...) are fully three dimensional, but the discretisation is two-dimensional.

Note: This function evaluates the shape function gradients upon construction, they are not recomputed upon evaluation. To evaluate them with respect to updated coordinates (e.g. to do updated Lagrange), use ".update_x(...)" to update the nodal coordinates and re-evaluate the shape function gradients and integration volumes.

Note: By default integration is done using Gauss points. To use a different scheme one has to supply the position (in isoparametric coordinates) and weight of the integration points (their number is inferred from the input).

Note: Most functions take the output as the last input-argument, as to write directly to a pre-allocated array, avoiding their re-allocation. All these functions have a wrapper that does the allocation for you (and thus returns the output rather than taking it as input). All function of this kind are indicated here with a *

20.3.1 Element::Quad4::QuadratureAxisymmetric::update_x(...)

Update the nodal coordinates (elemvec: [nelem, nne, ndim]).

20.3.2 Element::Quad4::QuadratureAxisymmetric::nelem()

Number of elements.

20.3.3 Element::Quad4::QuadratureAxisymmetric::nne()

Number of nodes per element.

20.3.4 Element::Quad4::QuadratureAxisymmetric::ndim()

Number of dimensions.

20.3.5 Element::Quad4::QuadratureAxisymmetric::nip()

Number of integration points.

20.3.6 Element::Quad4::QuadratureAxisymmetric::GradN()

(Current) Shape function gradient (w.r.t. real coordinates): [nelem, nip, nne, ndim]

20.3.7 Element::Quad4::QuadratureAxisymmetric::dV(...)*

(Current) Volume of each integration point (qscalar: [nelem, nip]). An overload is available to get the same result as a tensor per integration point (qtensor: [nelem, nip, tdim, tdim]) with all tensor-components having the same value.

20.3.8 Element::Quad4::QuadratureAxisymmetric::gradN_vector(...)*

Implementation of

$$\varepsilon = \vec{\nabla} N_m \vec{u}_m$$

or in index notation

$$\varepsilon_{ij} = \frac{\partial N_m}{\partial x_i} u_{mj}$$

20.3.9 Element::Quad4::QuadratureAxisymmetric::gradN_vector_T(...)*

Implementation of

$$\varepsilon = \left(\vec{\nabla} N_m \vec{u}_m \right)^T$$

or in index notation

$$\varepsilon_{ji} = \frac{\partial N_m}{\partial x_i} u_{mj}$$

20.3.10 Element::Quad4::QuadratureAxisymmetric::symGradN_vector(...)*

Implementation of

$$\varepsilon = \tfrac{1}{2} \left(\vec{\nabla} N_m \vec{u}_m + \left(\vec{\nabla} N_m \vec{u}_m \right)^T \right)$$

20.3.11 Element::Quad4::QuadratureAxisymmetric::int_N_scalar_NT_dV(...)*

Implementation of

$$M_{mn} = \int_{\Omega^h} N_m \; \rho \; N_n \; \mathrm{d}\Omega^h \equiv \sum_q \; N_m \; \rho \; N_n \; \delta\Omega_q$$

Note that the output is an "elemmat", which has shape [nelem, nne*ndim, nne*ndim]. This implies that all dimensions are the same.

20.3.12 Element::Quad4::QuadratureAxisymmetric::int_gradN_dot_tensor2_dV(...)*

Implementation of:

$$\vec{f}_m = \int_{\Omega^h} (\vec{\nabla} N_m) \cdot \sigma \, \mathrm{d}\Omega^h$$

or in index notation

$$f_{mj} = \sum_q \frac{\partial N_m}{\partial x_i} \sigma_{ij} \, \delta\Omega_q$$

20.3.13 Element::Quad4::QuadratureAxisymmetric::int_gradN_dot_tensor4_dot_gradNT_dV(...

Implementation of:

$$K_{mn} = \int_{\Omega^h} (\vec{\nabla} N_m) \cdot \mathbb{C} \cdot \vec{\nabla} N_n \, \mathrm{d}\Omega^h$$

or in index notation

$$K_{m+id,n+kd} = \sum_q \frac{\partial N_m}{\partial x_i} C_{ijkl} \frac{\partial N_n}{\partial x_l} \, \delta\Omega_q$$

Note that the output is an "elemmat", which has shape [nelem, nne*ndim, nne*ndim].

20.4 Element::Quad4::Gauss

Integration points according to exact integration using Gauss points.

20.4.1 Element::Quad4::Gauss::nip()

Returns the number of integration points.

20.4.2 Element::Quad4::Gauss::xi()

Returns the position of the integration points in isoparametric coordinates [nip, ndim] (with ndim = 3).

20.4.3 Element::Quad4::Gauss::w()

Returns the weights of the integration points [nip].

20.5 Element::Quad4::Nodal

Integration points that coincide with the nodes (equally weight). This scheme can for example be used to obtain a diagonal mass matrix.

20.5.1 Element::Quad4::Nodal::nip()

Returns the number of integration points.

20.5.2 Element::Quad4::Nodal::xi()

Returns the position of the integration points in isoparametric coordinates [nip, ndim] (with ndim = 3).

20.5.3 Element::Quad4::Nodal::w()

Returns the weights of the integration points [nip].

20.6 Element::Quad4::MidPoint

Single integration point in the middle of the element.

20.6.1 Element::Quad4::MidPoint::nip()

Returns the number of integration points.

20.6.2 Element::Quad4::MidPoint::xi()

Returns the position of the integration points in isoparametric coordinates [nip, ndim] (with ndim = 3).

20.6.3 Element::Quad4::MidPoint::w()

Returns the weights of the integration points [nip].

CHAPTER 21

Element::Hex8

`GooseFEM/ElementHex8.h`
`GooseFEM/ElementHex8.hpp`

21.1 Element::Hex8::Quadrature

Element definition to numerically interpolate and integrate.

Note: This function evaluates the shape function gradients upon construction, they are not recomputed upon evaluation. To evaluate them with respect to updated coordinates (e.g. to do updated Lagrange), use ".update_x(...)" to update the nodal coordinates and re-evaluate the shape function gradients and integration volumes.

Note: By default integration is done using Gauss points. To use a different scheme one has to supply the position (in isoparametric coordinates) and weight of the integration points (their number is inferred from the input).

Note: Most functions take the output as the last input-argument, as to write directly to a pre-allocated array, avoiding their re-allocation. All these functions have a wrapper that does the allocation for you (and thus returns the output rather than taking it as input). All function of this kind are indicated here with a *

21.1.1 Element::Hex8::Quadrature::update_x(...)

Update the nodal coordinates (elemvec: [nelem, nne, ndim]).

21.1.2 Element::Hex8::Quadrature::nelem()

Number of elements.

21.1.3 Element::Hex8::Quadrature::nne()

Number of nodes per element.

21.1.4 Element::Hex8::Quadrature::ndim()

Number of dimensions.

21.1.5 Element::Hex8::Quadrature::nip()

Number of integration points.

21.1.6 Element::Hex8::Quadrature::GradN()

(Current) Shape function gradient (w.r.t. real coordinates): [nelem, nip, nne, ndim]

21.1.7 Element::Hex8::Quadrature::dV(...)*

(Current) Volume of each integration point (qscalar: [nelem, nip]). An overload is available to get the same result as a tensor per integration point (qtensor: [nelem, nip, tdim, tdim]) with all tensor-components having the same value.

21.1.8 Element::Hex8::Quadrature::gradN_vector(...)*

Implementation of

$$\varepsilon = \vec{\nabla} N_m \vec{u}_m$$

or in index notation

$$\varepsilon_{ij} = \frac{\partial N_m}{\partial x_i} u_{mj}$$

21.1.9 Element::Hex8::Quadrature::gradN_vector_T(...)*

Implementation of

$$\varepsilon = \left(\vec{\nabla} N_m \vec{u}_m \right)^T$$

or in index notation

$$\varepsilon_{ji} = \frac{\partial N_m}{\partial x_i} u_{mj}$$

21.1.10 Element::Hex8::Quadrature::symGradN_vector(...)*

Implementation of

$$\varepsilon = \frac{1}{2}\left(\vec{\nabla} N_m \vec{u}_m + \left(\vec{\nabla} N_m \vec{u}_m\right)^T\right)$$

21.1.11 Element::Hex8::Quadrature::int_N_scalar_NT_dV(...)*

Implementation of

$$M_{mn} = \int_{\Omega^h} N_m \; \rho \; N_n \; \mathrm{d}\Omega^h \equiv \sum_q \; N_m \; \rho \; N_n \; \delta\Omega_q$$

Note that the output is an "elemmat", which has shape [nelem, nne*ndim, nne*ndim]. This implies that all dimensions are the same.

21.1.12 Element::Hex8::Quadrature::int_gradN_dot_tensor2_dV(...)*

Implementation of:

$$\vec{f}_m = \int_{\Omega^h} (\vec{\nabla} N_m) \cdot \sigma \; \mathrm{d}\Omega^h$$

or in index notation

$$f_{mj} = \sum_q \; \frac{\partial N_m}{\partial x_i} \sigma_{ij} \; \delta\Omega_q$$

21.1.13 Element::Hex8::Quadrature::int_gradN_dot_tensor4_dot_gradNT_dV(...)*

Implementation of:

$$K_{mn} = \int_{\Omega^h} (\vec{\nabla} N_m) \cdot \mathbb{C} \cdot \vec{\nabla} N_n \; \mathrm{d}\Omega^h$$

or in index notation

$$K_{m+id,n+kd} = \sum_q \; \frac{\partial N_m}{\partial x_i} C_{ijkl} \frac{\partial N_n}{\partial x_l} \; \delta\Omega_q$$

Note that the output is an "elemmat", which has shape [nelem, nne*ndim, nne*ndim].

21.2 Element::Hex8::Gauss

Integration points according to exact integration using Gauss points.

21.2.1 Element::Hex8::Gauss::nip()

Returns the number of integration points.

21.2.2 Element::Hex8::Gauss::xi()

Returns the position of the integration points in isoparametric coordinates [nip, ndim] (with ndim = 3).

21.2.3 Element::Hex8::Gauss::w()

Returns the weights of the integration points [nip].

21.3 Element::Hex8::Nodal

Integration points that coincide with the nodes (equally weight). This scheme can for example be used to obtain a diagonal mass matrix.

21.3.1 Element::Hex8::Nodal::nip()

Returns the number of integration points.

21.3.2 Element::Hex8::Nodal::xi()

Returns the position of the integration points in isoparametric coordinates [nip, ndim] (with ndim = 3).

21.3.3 Element::Hex8::Nodal::w()

Returns the weights of the integration points [nip].

CHAPTER 22

Vector

```
GooseFEM/Vector.h
GooseFEM/Vector.hpp
GooseFEM/VectorPartitioned.h
GooseFEM/VectorPartitioned.hpp
GooseFEM/VectorPartitionedTyings.h
GooseFEM/VectorPartitionedTyings.hpp
```

22.1 Vector

Vector definition allowing transforming between "dofval", "nodevec", and "elemvec" representations. See *Vector representation*.

22.1.1 Vector::nelem()

Return the number of elements.

22.1.2 Vector::nne()

Return the number of nodes-per-element.

22.1.3 Vector::nnode()

Return the number of nodes.

22.1.4 Vector::ndim()

Return the number of dimensions.

22.1.5 Vector::ndof()

Return the number of DOFs.

22.1.6 Vector::dofs()

Return the DOF-numbers per node [nnode, ndim].

22.1.7 Vector::copy(...)

Copy "nodevec" to "nodevec".

22.1.8 Vector::asDofs(...)

Convert "nodevec" or "elemvec" to "dofval".

Warning: Verify that you don't need "assembleDofs(...)"

22.1.9 Vector::asNode(...)

Convert "dofval" or "elemvec" to "nodevec".

Warning: Verify that you don't need "assembleNode(...)"

22.1.10 Vector::asElement(...)

Convert "dofval" or "nodevec" to "elemvec".

22.1.11 Vector::assembleDofs(...)

Convert "nodevec" or "elemvec" to "dofval".

Warning: Verify that you don't need "asDofs(...)"

22.1.12 Vector::assembleNode(...)

Convert "dofval" or "elemvec" to "nodevec".

Warning: Verify that you don't need "asNode(...)"

22.2 VectorPartitioned

Partitioned vector definition allowing transforming between "dofval", "nodevec", and "elemvec" representations. See *Vector representation*. The partitioning is such that the DOFs are ordered as "[iiu, iip]" with "iiu" the unknown DOFs and "iip" the prescribed DOFs.

22.2.1 VectorPartitioned::nelem()

Return the number of elements.

22.2.2 VectorPartitioned::nne()

Return the number of nodes-per-element.

22.2.3 VectorPartitioned::nnode()

Return the number of nodes.

22.2.4 VectorPartitioned::ndim()

Return the number of dimensions.

22.2.5 VectorPartitioned::ndof()

Return the number of DOFs.

22.2.6 VectorPartitioned::nnu()

Return the number of unknown DOFs.

22.2.7 VectorPartitioned::nnp()

Return the number of prescribed DOFs.

22.2.8 VectorPartitioned::dofs()

Return the DOF-numbers per node [nnode, ndim].

22.2.9 VectorPartitioned::iiu()

Return the unknown DOF-numbers per node [nnu].

22.2.10 VectorPartitioned::iip()

Return the prescribed DOF-numbers per node [nnp].

22.2.11 VectorPartitioned::copy(...)

Copy "nodevec" to "nodevec".

22.2.12 VectorPartitioned::copy_u(...)

Copy the unknown DOFs from a "nodevec" to the unknown DOFs from another "nodevec".

22.2.13 VectorPartitioned::copy_p(...)

Copy the prescribed DOFs from a "nodevec" to the prescribed DOFs from another "nodevec".

22.2.14 VectorPartitioned::asDofs(...)

Convert "nodevec" or "elemvec" to "dofval".

Warning: Verify that you don't need "assembleDofs(...)"

22.2.15 VectorPartitioned::asDofs_u(...)

Convert "nodevec" or "elemvec" to "dofval" and extract the unknown DOFs "iiu".

Warning: Verify that you don't need "assembleDofs(...)"

22.2.16 VectorPartitioned::asDofs_p(...)

Convert "nodevec" or "elemvec" to "dofval" and extract the prescribed DOFs "iip".

Warning: Verify that you don't need "assembleDofs(...)"

22.2.17 VectorPartitioned::asNode(...)

Convert "dofval" or "elemvec" to "nodevec".

Warning: Verify that you don't need "assembleNode(...)"

22.2.18 VectorPartitioned::asElement(...)

Convert "dofval" or "nodevec" to "elemvec".

22.2.19 VectorPartitioned::assembleDofs(...)

Convert "nodevec" or "elemvec" to "dofval".

Warning: Verify that you don't need "asDofs(...)"

22.2.20 VectorPartitioned::assembleDofs_u(...)

Convert "nodevec" or "elemvec" to "dofval" and extract the unknown DOFs "iiu".

Warning: Verify that you don't need "asDofs(...)"

22.2.21 VectorPartitioned::assembleDofs_p(...)

Convert "nodevec" or "elemvec" to "dofval" and extract the prescribed DOFs "iip".

Warning: Verify that you don't need "asDofs(...)"

22.2.22 VectorPartitioned::assembleNode(...)

Convert "dofval" or "elemvec" to "nodevec".

Warning: Verify that you don't need "asNode(...)"

22.3 VectorPartitionedTyings

Partitioned vector definition with nodal tyings allowing transforming between "dofval", "nodevec", and "elemvec" representations. See *Vector representation*. The partitioning is such that the DOFs are ordered as "[iiu, iip, iid]" with "iiu" the unknown DOFs and "iip" the prescribed DOFs and "iid" the dependent DOFs.

22.3.1 VectorPartitionedTyings::nelem()

Return the number of elements.

22.3.2 VectorPartitionedTyings::nne()

Return the number of nodes-per-element.

22.3.3 VectorPartitionedTyings::nnode()

Return the number of nodes.

22.3.4 VectorPartitionedTyings::ndim()

Return the number of dimensions.

22.3.5 VectorPartitionedTyings::ndof()

Return the number of DOFs.

22.3.6 VectorPartitionedTyings::nnu()

Return the number of unknown DOFs.

22.3.7 VectorPartitionedTyings::nnp()

Return the number of prescribed DOFs.

22.3.8 VectorPartitionedTyings::nni()

Return the number of independent DOFs.

22.3.9 VectorPartitionedTyings::nnd()

Return the number of dependent DOFs.

22.3.10 VectorPartitionedTyings::dofs()

Return the DOF-numbers per node [nnode, ndim].

22.3.11 VectorPartitionedTyings::iiu()

Return the unknown DOF-numbers per node [nnu].

22.3.12 VectorPartitionedTyings::iip()

Return the prescribed DOF-numbers per node [nnp].

22.3.13 VectorPartitionedTyings::iii()

Return the independent DOF-numbers per node [nni].

22.3.14 VectorPartitionedTyings::iid()

Return the dependent DOF-numbers per node [nnd].

22.3.15 VectorPartitionedTyings::copy(...)

Copy "nodevec" to "nodevec".

22.3.16 VectorPartitionedTyings::copy_u(...)

Copy the unknown DOFs from a "nodevec" to the unknown DOFs from another "nodevec".

22.3.17 VectorPartitionedTyings::copy_p(...)

Copy the prescribed DOFs from a "nodevec" to the prescribed DOFs from another "nodevec".

22.3.18 VectorPartitionedTyings::asDofs(...)

Convert "nodevec" or "elemvec" to "dofval".

Warning: Verify that you don't need "assembleDofs(...)"

22.3.19 VectorPartitionedTyings::asDofs_i(...)

Convert "nodevec" or "elemvec" to "dofval" and extract the independent DOFs "iii". Choose to apply the tyings:

$$u_i = C_{di}^T u_d$$

Warning: Verify that you don't need "assembleDofs(...)"

22.3.20 VectorPartitionedTyings::asNode(...)

Convert "dofval" or "elemvec" to "nodevec".

> **Warning:** Verify that you don't need "assembleNode(...)"

22.3.21 VectorPartitionedTyings::asElement(...)

Convert "dofval" or "nodevec" to "elemvec".

22.3.22 VectorPartitionedTyings::assembleDofs(...)

Convert "nodevec" or "elemvec" to "dofval".

> **Warning:** Verify that you don't need "asDofs(...)"

22.3.23 VectorPartitionedTyings::assembleNode(...)

Convert "dofval" or "elemvec" to "nodevec".

> **Warning:** Verify that you don't need "asNode(...)"

CHAPTER 23

Matrix

```
GooseFEM/Matrix.h
GooseFEM/Matrix.hpp
GooseFEM/MatrixPartitioned.h
GooseFEM/MatrixPartitioned.hpp
GooseFEM/MatrixPartitionedTyings.h
GooseFEM/MatrixPartitionedTyings.hpp
GooseFEM/MatrixDiagonal.h
GooseFEM/MatrixDiagonal.hpp
GooseFEM/MatrixDiagonalPartitioned.h
GooseFEM/MatrixDiagonalPartitioned.hpp
```

23.1 Matrix

Matrix definition.

Note: A solver has to be chosen, see *Linear solver*.

23.1.1 Matrix::nelem()

Return the number of elements.

23.1.2 Matrix::nne()

Return the number of nodes-per-element.

23.1.3 Matrix::nnode()

Return the number of nodes.

23.1.4 Matrix::ndim()

Return the number of dimensions.

23.1.5 Matrix::ndof()

Return the number of DOFs.

23.1.6 Matrix::dofs()

Return the DOF-numbers per node [nnode, ndim].

23.1.7 Matrix::assemble(...)

Assemble matrix from element matrices stored as "elemmat".

23.1.8 Matrix::solve(...)

Solve linear system.

23.2 MatrixPartitioned

Partitioned matrix definition.

Note: A solver has to be chosen, see *Linear solver*.

23.2.1 MatrixPartitioned::nelem()

Return the number of elements.

23.2.2 MatrixPartitioned::nne()

Return the number of nodes-per-element.

23.2.3 MatrixPartitioned::nnode()

Return the number of nodes.

23.2.4 MatrixPartitioned::ndim()

Return the number of dimensions.

23.2.5 MatrixPartitioned::ndof()

Return the number of DOFs.

23.2.6 MatrixPartitioned::nnu()

Return the number of unknown DOFs.

23.2.7 MatrixPartitioned::nnp()

Return the number of prescribed DOFs.

23.2.8 MatrixPartitioned::dofs()

Return the DOF-numbers per node [nnode, ndim].

23.2.9 MatrixPartitioned::iiu()

Return the unknown DOF-numbers per node [nnu].

23.2.10 MatrixPartitioned::iip()

Return the prescribed DOF-numbers per node [nnp].

23.2.11 MatrixPartitioned::assemble(...)

Assemble matrix from element matrices stored as "elemmat".

23.2.12 MatrixPartitioned::solve(...)

Solve linear system.

23.2.13 MatrixPartitioned::solve_u(...)

Solve linear system.

23.2.14 MatrixPartitioned::reaction(...)

Compute reaction forces (part of "b" that corresponds to "x_p").

23.2.15 MatrixPartitioned::reaction_p(...)

Compute reaction forces (part of "b" that corresponds to "x_p").

23.3 MatrixPartitionedTyings

Partitioned matrix definition with nodal tyings.

Note: A solver has to be chosen, see *Linear solver*.

23.3.1 MatrixPartitionedTyings::nelem()

Return the number of elements.

23.3.2 MatrixPartitionedTyings::nne()

Return the number of nodes-per-element.

23.3.3 MatrixPartitionedTyings::nnode()

Return the number of nodes.

23.3.4 MatrixPartitionedTyings::ndim()

Return the number of dimensions.

23.3.5 MatrixPartitionedTyings::ndof()

Return the number of DOFs.

23.3.6 MatrixPartitionedTyings::nnu()

Return the number of unknown DOFs.

23.3.7 MatrixPartitionedTyings::nnp()

Return the number of prescribed DOFs.

23.3.8 MatrixPartitionedTyings::nni()

Return the number of independent DOFs.

23.3.9 MatrixPartitionedTyings::nnd()

Return the number of dependent DOFs.

23.3.10 MatrixPartitionedTyings::dofs()

Return the DOF-numbers per node [nnode, ndim].

23.3.11 MatrixPartitionedTyings::iiu()

Return the unknown DOF-numbers per node [nnu].

23.3.12 MatrixPartitionedTyings::iip()

Return the prescribed DOF-numbers per node [nnp].

23.3.13 MatrixPartitionedTyings::iii()

Return the independent DOF-numbers per node [nni].

23.3.14 MatrixPartitionedTyings::iid()

Return the dependent DOF-numbers per node [nnd].

23.3.15 MatrixPartitionedTyings::assemble(...)

Assemble matrix from element matrices stored as "elemmat".

23.3.16 MatrixPartitionedTyings::solve(...)

Solve linear system.

23.3.17 MatrixPartitionedTyings::solve_u(...)

Solve linear system.

23.4 MatrixDiagonal

Diagonal matrix definition.

23.4.1 MatrixDiagonal::nelem()

Return the number of elements.

23.4.2 MatrixDiagonal::nne()

Return the number of nodes-per-element.

23.4.3 MatrixDiagonal::nnode()

Return the number of nodes.

23.4.4 MatrixDiagonal::ndim()

Return the number of dimensions.

23.4.5 MatrixDiagonal::ndof()

Return the number of DOFs.

23.4.6 MatrixDiagonal::dofs()

Return the DOF-numbers per node [nnode, ndim].

23.4.7 MatrixDiagonal::assemble(...)

Assemble matrix from element matrices stored as "elemmat".

23.4.8 MatrixDiagonal::dot(...)

Dot-product:

$$b_i = A_{ij} x_j$$

23.4.9 MatrixDiagonal::solve(...)

Solve linear system.

23.4.10 MatrixDiagonal::AsDiagonal(...)

Return matrix as diagonal matrix (column)

23.5 MatrixDiagonalPartitioned

Diagonal and partitioned matrix definition.

23.5.1 MatrixDiagonalPartitioned::nelem()

Return the number of elements.

23.5.2 MatrixDiagonalPartitioned::nne()

Return the number of nodes-per-element.

23.5.3 MatrixDiagonalPartitioned::nnode()

Return the number of nodes.

23.5.4 MatrixDiagonalPartitioned::ndim()

Return the number of dimensions.

23.5.5 MatrixDiagonalPartitioned::ndof()

Return the number of DOFs.

23.5.6 MatrixDiagonalPartitioned::nnu()

Return the number of unknown DOFs.

23.5.7 MatrixDiagonalPartitioned::nnp()

Return the number of prescribed DOFs.

23.5.8 MatrixDiagonalPartitioned::dofs()

Return the DOF-numbers per node [nnode, ndim].

23.5.9 MatrixDiagonalPartitioned::iiu()

Return the unknown DOF-numbers per node [nnu].

23.5.10 MatrixDiagonalPartitioned::iip()

Return the prescribed DOF-numbers per node [nnp].

23.5.11 MatrixDiagonalPartitioned::assemble(...)

Assemble matrix from element matrices stored as "elemmat".

23.5.12 MatrixDiagonalPartitioned::dot(...)

Dot-product:

$$b_i = A_{ij} x_j$$

23.5.13 MatrixDiagonalPartitioned::dot_u(...)

Dot-product:

$$b_i = A_{ij} x_j$$

23.5.14 MatrixDiagonalPartitioned::dot_p(...)

Dot-product:

$$b_i = A_{ij} x_j$$

23.5.15 MatrixDiagonalPartitioned::solve(...)

Solve linear system.

23.5.16 MatrixDiagonalPartitioned::solve_u(...)

Solve linear system.

23.5.17 MatrixDiagonalPartitioned::reaction(...)

Compute reaction forces (part of "b" that corresponds to "x_p").

23.5.18 MatrixDiagonalPartitioned::reaction_p(...)

Compute reaction forces (part of "b" that corresponds to "x_p").

23.5.19 MatrixDiagonalPartitioned::AsDiagonal(...)

Return matrix as diagonal matrix (column)

23.6 Linear solver

The classes `GooseFEM::MatrixPartitioned` and `GooseFEM::MatrixPartitionedTyings` make use of a library to solver the linear system (stored as a sparse matrix). In particular the Eigen library and its plug-ins are used. To use the library's default solver:

```
#include <Eigen/Eigen>
#include <GooseFEM/GooseFEM.h>

int main()
{
    ...

    GooseFEM::MatrixPartitioned<> K(...);
```

(continues on next page)

(continued from previous page)

```
    ...

    return 0;
}
```

The default solver can, however, be quite slow. Therefore Eigen has quite some plug-ins for the solver. GooseFEM allows the use of Eigen's Sparse Solver Concept to use such plug-ins. For example, to use SuiteSparse:

```
#include <Eigen/Eigen>
#include <Eigen/CholmodSupport>
#include <GooseFEM/GooseFEM.h>

int main()
{
    ...

    GooseFEM::MatrixPartitioned<Eigen::CholmodSupernodalLLT<Eigen::SparseMatrix
↪<double>>> K(...);

    ...

    return 0;
}
```

CHAPTER 24

Tyings

`GooseFEM/TyingsPeriodic.h`
`GooseFEM/TyingsPeriodic.hpp`

24.1 Tyings::Periodic

Periodic nodal tyings: partition the system (renumber the DOFs) in the following order [iii, iid]: first the independent DOFs and the the dependent DOFs. If, in addition, independent DOFs are prescribed the partitioning is [iiu, iip, iid], where iii = [iiu, iip]: first the unknown and then the prescribed DOFs.

24.1.1 Tyings::Periodic::nnd()

Return the dependent DOF-numbers.

24.1.2 Tyings::Periodic::nni()

Return the independent DOF-numbers.

24.1.3 Tyings::Periodic::nnu()

Return the unknown DOF-numbers.

24.1.4 Tyings::Periodic::nnp()

Return the prescribed DOF-numbers.

24.1.5 Tyings::Periodic::dofs()

Renumbered DOFs per node [nnode, ndim].

24.1.6 Tyings::Periodic::control()

Control DOF, that should be fixed [ndim].

24.1.7 Tyings::Periodic::iid()

Return the dependent DOFs.

24.1.8 Tyings::Periodic::iii()

Return the independent DOFs.

24.1.9 Tyings::Periodic::iiu()

Return the unknown DOFs.

24.1.10 Tyings::Periodic::iip()

Return the prescribed DOFs.

24.1.11 Tyings::Periodic::Cdi()

Return the tying matrix, such that

$$u_d = C_{di} u_i$$

In addition, the tying matrix in terms of the partitioned system can be obtained:

$$u_d = [C_{du}, C_{dp}]^T [u_u, u_p] = C_{du} u_u + C_{dp} u_p$$

24.1.12 Tyings::Periodic::Cdu()

Return the tying matrix, such that:

$$u_d = [C_{du}, C_{dp}]^T [u_u, u_p] = C_{du} u_u + C_{dp} u_p$$

24.1.13 Tyings::Periodic::Cdp()

Return the tying matrix, such that:

$$u_d = [C_{du}, C_{dp}]^T [u_u, u_p] = C_{du} u_u + C_{dp} u_p$$

24.2 Tyings::Control

Add virtual control nodes to the system.

24.2.1 Tyings::Control::coor()

Nodal coordinates, including the virtual control nodes [nnode, ndim].

24.2.2 Tyings::Control::dofs()

DOFs, including the virtual control nodes [nnode, ndim].

24.2.3 Tyings::Control::controlDofs()

Virtual control DOFs [ndim, ndim].

24.2.4 Tyings::Control::controlNodes()

Virtual control nodes [ndim].

CHAPTER 25

ParaView

`GooseFEM/ParaView.h`
`GooseFEM/ParaView.hpp`

Note: This header relies on HDF5 and HighFive as dependencies. If you wish to use ParaView support without making use of HDF5 and HighFive, you have to define `GOOSEFEM_NO_HIGHFIVE` before including `ParaView.h` for the first time:

```
#define GOOSEFEM_NO_HIGHFIVE
#include <GooseFEM/ParaView.h>
```

In this case the library does not automatically read the shapes of the datasets. Instead you'll have to provide them as `std::vector<size_t>`.

25.1 HDF5

25.1.1 TimeSeries

A TimeSeries is constructed from a number of Increments. The consecutive Increments are added to the TimeSeries using push_back. The order in which this is done will define the order of the TimeSeries. Each Increment is constructed from a mesh (using Connectivity and Coordinates) and a number of nodal or cell Attributes.

Consider this example

`figures/ParaView/HDF5/main.cpp`

```
#include <GooseFEM/GooseFEM.h>
#include <GooseFEM/ParaView.h>

```

(continues on next page)

(continued from previous page)

```
namespace PV = GooseFEM::ParaView::HDF5;

int main()
{
  // define mesh
  GooseFEM::Mesh::Quad4::FineLayer mesh(6,18);

  // extract mesh fields
  xt::xtensor<double,2> coor = mesh.coor();
  xt::xtensor<double,2> conn = mesh.conn();
  xt::xtensor<double,2> disp = xt::zeros<double>(coor.shape());

  // vector definition:
  // provides methods to switch between dofval/nodeval/elemvec, or to manipulate a␣
↪part of them
  GooseFEM::Vector vector(conn, mesh.dofs());

  // FEM quadrature
  GooseFEM::Element::Quad4::Quadrature elem(vector.AsElement(coor));

  // open output file
  H5Easy::File data("output.h5" , H5Easy::File::Overwrite);

  // initialise ParaView metadata
  PV::TimeSeries xdmf;

  // save mesh to output file
  H5Easy::dump(data, "/coor", coor);
  H5Easy::dump(data, "/conn", conn);

  // define strain history
  xt::xtensor<double,1> gamma = xt::linspace<double>(0, 1, 100);

  // loop over increments
  for (size_t inc = 0; inc < gamma.size(); ++inc)
  {
    // apply fictitious strain
    for (size_t node = 0; node < disp.shape(0); ++node)
      disp(node,0) = gamma(inc) * (coor(node,1) - coor(0,1));

    // compute strain tensor
    xt::xtensor<double,4> Eps = elem.SymGradN_vector(vector.AsElement(disp));
    xt::xtensor<double,1> Eps_xy = xt::view(Eps, xt::all(), 0, 0, 1);

    // store data to output file
    H5Easy::dump(data, "/disp/"   + std::to_string(inc), PV::as3d(disp));
    H5Easy::dump(data, "/eps_xy/" + std::to_string(inc), Eps_xy);

    // add increment to ParaView metadata
    xdmf.push_back(PV::Increment(
      PV::Connectivity(data, "/conn", mesh.getElementType()),
      PV::Coordinates (data, "/coor"),
      {
        PV::Attribute(data, "/disp/"   + std::to_string(inc), "Displacement",␣
↪PV::AttributeType::Node),
        PV::Attribute(data, "/eps_xy/" + std::to_string(inc), "Eps_xy"      ,␣
↪PV::AttributeType::Cell)
```

(continues on next page)

(continued from previous page)

```
      }
    ));
  }

  // write ParaView metadata
  xdmf.write("output.xdmf");

  return 0;
}
```

Tip: A displacement vector in must be always 3-d in ParaView, even when the mesh is in 2-d. Use the `GooseFEM::ParaView::HDF5::as3d(...)` function to convert a matrix of 2-d displacements to a matrix of 3-d displacements.

Note: The Python interface avoids the HDF5 and HighFive dependencies. One therefore has to provide the datasets' shapes. Consider the following Python example:

`figures/ParaView/HDF5/main.py`

```
import h5py
import numpy                 as np
import GooseFEM              as gf
import GooseFEM.ParaView.HDF5 as pv

# define mesh
mesh = gf.Mesh.Quad4.FineLayer(6, 18)

# extract mesh fields
coor = mesh.coor();
conn = mesh.conn();
disp = np.zeros(coor.shape)

# vector definition:
# provides methods to switch between dofval/nodeval/elemvec, or to manipulate a part␣
↪of them
vector = gf.Vector(conn, mesh.dofs())

# FEM quadrature
elem = gf.Element.Quad4.Quadrature(vector.AsElement(coor))

# open output file
data = h5py.File("output.h5", "w")

# initialise ParaView metadata
xdmf = pv.TimeSeries()

# save mesh to output file
data["/coor"] = coor
data["/conn"] = conn

# define strain history
gamma = np.linspace(0, 1, 100);

# loop over increments
```

(continues on next page)

(continued from previous page)

```
for inc in range(len(gamma)):

  # apply fictitious strain
  for node in range(disp.shape[0]):
    disp[node,0] = gamma[inc] * (coor[node,1] - coor[0,1])

  # compute strain tensor
  Eps = elem.SymGradN_vector(vector.AsElement(disp));
  Eps_xy = Eps[:, 0, 0, 1]

  # store data to output file
  data["/disp/"   + str(inc)] = pv.as3d(disp)
  data["/eps_xy/" + str(inc)] = Eps_xy

  # ParaView metadata
  # - initialise Increment
  xdmf_inc = pv.Increment(
    pv.Connectivity(data.filename, "/conn", pv.ElementType.Quadrilateral, conn.shape),
    pv.Coordinates (data.filename, "/coor"                              , coor.shape),
  )
  # - add attributes to Increment
  dataset = "/disp/" + str(inc)
  xdmf_inc.push_back(pv.Attribute(
    data.filename, dataset, "Displacement", pv.AttributeType.Node, data[dataset].
↪shape))
  # - add attributes to Increment
  dataset = "/eps_xy/" + str(inc)
  xdmf_inc.push_back(pv.Attribute(
    data.filename, dataset, "Eps_xy", pv.AttributeType.Cell, data[dataset].shape))
  # - add Increment to TimeSeries
  xdmf.push_back(xdmf_inc)

# write ParaView metadata
xdmf.write("output.xdmf");
```

CHAPTER 26

Iterate

```
GooseFEM/Iterate.h
GooseFEM/Iterate.hpp
```

26.1 Iterate::StopList

Convergence check to check that a residual is smaller than a certain value of a certain number of consecutive steps. The class is constructed with the number of consecutive to enforce the residual.

26.1.1 Iterate::StopList::reset(...)

Reset all residuals to infinity.

26.1.2 Iterate::StopList::stop(...)

Append the list with residuals, return “true” if all residuals are below the tolerance (argument).

CHAPTER 27

Notes for developers

27.1 Testing

Please add relevant tests to `test/`. To run all tests:

```
mkdir build/
cmake .. -DBUILD_TESTS=ON
./test/test
```

CHAPTER 28

Indices and tables

- genindex
- modindex
- search