
GooseFEM Documentation

Tom de Geus

Dec 08, 2018

USAGE

1	Data-types	3
2	Data-storage	5
3	Contents	7
4	Indices and tables	19

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, ... Bugs 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.

Tip: 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.

This header-only module provides C++ classes and several accompanying methods to work with n-d arrays and/or tensors. It's usage, programmatically and from a compilation perspective, is really simple. One just has to `#include <GooseFEM/GooseFEM.h>` and tell your compiler where GooseFEM is located (and to use the C++14 or younger standard). Really, that's it!

[GooseFEM/GooseFEM.h]

Beyond the default C++ types, GooseFEM use Eigen columns/matrices and cppmat multi-dimensional arrays. In particular:

Type	Description
MatD	row-major <code>Eigen::matrix</code>
ColD	(column-major) <code>Eigen::Matrix(column)</code>
SpMatD	row-major <code>Eigen::SparseMatrix</code>
ArrD	multi-dimensional <code>cppmat::array</code>

The last letter thereby indicates the type specialisation:

Indicator	Description
D	double
S	size_t
I	int

CHAPTER 2

Data-storage

Alias	Description	Type	Shape
“dofval”	degrees-of-freedom	ColD	[ndof]
“nodevec”	nodal vectors	MatD	[nnode, ndim]
“elemvec”	nodal vectors stored per element	ArrD	[nelem, nne, ndim]
“elemmat”	matrices stored per element	ArrD	[nelem, nne*ndim, nne*ndim]
“qtensor”	tensors stored (as list) per integration point	ArrD	[nelem, nip, #tensor-components]
“qscalar”	scalars stored per integration point	ArrD	[nelem, nip]

3.1 GooseFEM::Mesh

3.1.1 Generic methods

[GooseFEM/Mesh.h, GooseFEM/Mesh.hpp]

GooseFEM::Mesh::dofs

```
GooseFEM::MatS GooseFEM::Mesh::dofs(size_t nnode, size_t ndim)
```

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

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

GooseFEM::Mesh::renumber

```
GooseFEM::MatS GooseFEM::Mesh::renumber(const GooseFEM::MatS &dofs)
```

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: A generic interface using iterator is available if you do not wish to use the default Eigen interface.

GooseFEM::Mesh::reorder

```
GooseFEM::MatS GooseFEM::Mesh::reorder(const GooseFEM::MatS &dofs, const ColS &idx,
↳std::string location="end")
```

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: A generic interface using iterator is available if you do not wish to use the default Eigen interface.

GooseFEM::Mesh::elem2node

```
GooseFEM::SpMatS GooseFEM::Mesh::elem2node(const GooseFEM::MatS &conn)
```

Return a sparse matrix which contains the element numbers (columns) that are connected to each node (rows).

Warning: One should not confuse the element 0 when this matrix is converted to a dense matrix. When this is done all the ‘missing’ items are filled in as zero, which does not have a meaning here.

3.1.2 Predefined meshes

GooseFEM::Mesh::Tri3

[GooseFEM/MeshTri3.h, GooseFEM/MeshTri3.hpp]

GooseFEM::Mesh::Tri3::Regular

No description yet, please consult the code.

GooseFEM::Mesh::Quad4

[GooseFEM/MeshQuad4.h, GooseFEM/MeshQuad4.hpp]

Naming convention

GooseFEM::Mesh::Quad4::Regular

```
GooseFEM::Mesh::Quad4::Regular(size_t nelx, size_t nely, double h=1.);
```

Regular mesh of linear quadrilaterals in two-dimensions. The element edges are all of the same size h (by default equal to one), optional scaling can be applied afterwards. For example the mesh shown below that consists of 21 x 11 elements. In that image the element numbers are indicated with a color, and likewise for the boundary nodes.

Methods:

```
// A matrix with on each row a nodal coordinate:
// [ x , y ]
MatD = GooseFEM::Mesh::Quad4::Regular.coor();

// A matrix with the connectivity, with on each row to the nodes of each element
MatS = GooseFEM::Mesh::Quad4::Regular.conn();

// A list of boundary nodes
ColS = GooseFEM::Mesh::Quad4::Regular.nodesBottom();
ColS = GooseFEM::Mesh::Quad4::Regular.nodesTop();
ColS = GooseFEM::Mesh::Quad4::Regular.nodesLeft();
ColS = GooseFEM::Mesh::Quad4::Regular.nodesRight();

// A matrix with periodic node pairs on each row:
// [ independent nodes, dependent nodes ]
MatS = GooseFEM::Mesh::Quad4::Regular.nodesPeriodic();

// The node at the origin
size_t = GooseFEM::Mesh::Quad4::Regular.nodeOrigin();

// A matrix with DOF-numbers: two per node in sequential order
MatS = GooseFEM::Mesh::Quad4::Regular.dofs();

// A matrix with DOF-numbers: two per node in sequential order
// All the periodic repetitions are eliminated from the system
MatS = GooseFEM::Mesh::Quad4::Regular.dofsPeriodic();
```

GooseFEM::Mesh::Quad4::FineLayer

Regular mesh with a fine layer of quadrilateral elements, and coarser elements above and below.

Note: The coarsening depends strongly on the desired number of elements in horizontal elements. The becomes clear from the following example:

```
mesh = GooseFEM::Mesh::Quad4::FineLayer(6*9 ,51); // left image : 546 elements
mesh = GooseFEM::Mesh::Quad4::FineLayer(6*9+3,51); // middle image : 703 elements
mesh = GooseFEM::Mesh::Quad4::FineLayer(6*9+1,51); // right image : 2915 elements
```

Methods:

```
// A matrix with on each row a nodal coordinate:
// [ x , y ]
MatD = GooseFEM::Mesh::Quad4::Regular.coor();

// A matrix with the connectivity, with on each row to the nodes of each element
MatS = GooseFEM::Mesh::Quad4::Regular.conn();

// A list of boundary nodes
ColS = GooseFEM::Mesh::Quad4::Regular.nodesBottom();
ColS = GooseFEM::Mesh::Quad4::Regular.nodesTop();
ColS = GooseFEM::Mesh::Quad4::Regular.nodesLeft();
ColS = GooseFEM::Mesh::Quad4::Regular.nodesRight();

// A matrix with periodic node pairs on each row:
// [ independent nodes, dependent nodes ]
MatS = GooseFEM::Mesh::Quad4::Regular.nodesPeriodic();

// The node at the origin
size_t = GooseFEM::Mesh::Quad4::Regular.nodeOrigin();

// A matrix with DOF-numbers: two per node in sequential order
MatS = GooseFEM::Mesh::Quad4::Regular.dofs();

// A matrix with DOF-numbers: two per node in sequential order
// All the periodic repetitions are eliminated from the system
MatS = GooseFEM::Mesh::Quad4::Regular.dofsPeriodic();

// A list with the element numbers of the fine elements in the center of the mesh
// (highlighted in the plot below)
ColS = GooseFEM::Mesh::Quad4::FineLayer.elementsFine();

.. image:: figures/MeshQuad4/FineLayer/example_elementsFine.svg
   :width: 500px
   :align: center
```

GooseFEM::Mesh::Hex8

[MeshHex8.h, MeshHex8.hpp]

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 $x = 0, y = 0, z = 0$.
- ...

GooseFEM::Mesh::Hex8::Regular

Regular mesh.

GooseFEM::Mesh::Hex8::FineLayer

Mesh with a middle plane that is fine the middle, and becomes course away from this plane.

3.1.3 Type specific methods

GooseFEM::Mesh::Tri3

GooseFEM::Mesh::Tri3::Regular

[GooseFEM/MeshTri3.h, GooseFEM/MeshTri3.hpp]

GooseFEM::Mesh::Tri3::getOrientation

No description yet, please consult the code.

GooseFEM::Mesh::Tri3::setOrientation

No description yet, please consult the code.

GooseFEM::Mesh::Tri3::retriangulate

No description yet, please consult the code.

GooseFEM::Mesh::Tri3::TriUpdate

No description yet, please consult the code.

GooseFEM::Mesh::Tri3::Edge

No description yet, please consult the code.

3.2 GooseFEM::Element

3.2.1 Introduction

Provides routines to perform numerical quadrature. The philosophy is that the in- and output are multi-dimensional arrays (all of type `GooseFEM::ArrD`) that contain:

Alias	Description	Shape
“elemmat”	matrices stored per element	[nelem, nne*ndim, nne*ndim]
“elemvec”	nodal vectors stored per element	[nelem, nne, ndim]
“qtensor”	tensors stored (as list) per integration point, per element	[nelem, nip, #tensor-components]
“qscalar”	scalars stored per integration point, per element	[nelem, nip]

As a result of this choice, the routines here need to know nothing of your choice how to organise your data, and thus remain flexible on code-unspecific. Also, evaluation in parallel (with OpenMP) has been trivially implemented, without there being any pitfalls.

The different elements each have a class `Quadrature` that is used to take gradients and to integrate. The idea is that you supply the nodal coordinates as “elemvec” to the constructor, and if you wish customize the integration points and their weights. The shape functions and their gradients are immediately evaluated and stored for reuse.

A small example to get you started is to compute the displacement gradients at each integration point:

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

int main()
{
    // some mesh
    GooseFEM::Mesh::Quad4::Regular mesh(3,3);

    // get relevant fields
    GooseFEM::MatD coor = mesh.coor();
    GooseFEM::MatS conn = mesh.conn();
    GooseFEM::MatS dofs = mesh.dofs();

    // define quadrature
    GooseFEM::Element::Quad4::Quadrature quad(GooseFEM::Element::asElementVector(conn,
    ↪ coor));
```

(continues on next page)

(continued from previous page)

```

// define some displacement field: simple shear
// - zero-initialize displacements
GooseFEM::MatD disp = GooseFEM::MatD::Zero(mesh.nnode(), mesh.ndim());
// - shear stain
double gamma = 0.1;
// - update displacement field
for ( size_t n = 0 ; n < mesh.nnode() ; ++n )
    disp(n,0) = gamma * ( coor(n,1) - coor(0,1) );

// compute the displacement gradient of each integration point of each element
GooseFEM::ArrD Gradu = quad.gradN_vector_T(GooseFEM::Element::asElementVector(conn,
↪disp));

// view result
// - tensor to interpret the tensor components of "Gradu": [nelem, nip, #tensor-
↪components]
cppmat::view::cartesian::tensor2<double,2> gradu;
// - view for all integration points of all elements
for ( size_t e = 0 ; e < mesh.nelem() ; ++e ) {
    for ( size_t k = 0 ; k < quad.nip() ; ++k ) {
        // -- interpret sub-matrix
        gradu.setMap(&Gradu(e,k));
        // -- print element and integration point number
        std::cout << "e = " << e << ", k = " << k << std::endl;
        // -- print the displacement gradient
        std::cout << std::setw(5) << std::setprecision(3) << gradu << std::endl;
    }
}

return 0;
}

```

Tip: `GooseFEM::Element::asElementVector` converts nodal coordinates and displacement to the corresponding “elemvec”. In addition:

- `GooseFEM::Vector`: switch between “nodevec”, “elemvec”, “dofval”, ...

Tip: To take the gradients and integral with respect to updated coordinates (i.e. to do updated Lagrange), use the `.update_x(...)` method to update the nodal coordinates and re-evaluate the shape function gradients and integration volumes.

Note: All routines that take or return a “qtensor” have been templated such that you can supply a type to interpret it. For example:

Note: The code and headers for the different elements are quite similar. They have been kept as parallel implementations to allow flexible adaption. One can inspect or deploy changes easily using an editor that highlight the differences between files.

3.2.2 GooseFEM::Element::Quad4

No description yet, please consult the code.

GooseFEM::Element::Quad4::Quadrature

No description yet, please consult the code.

GooseFEM::Element::Quad4::Quadrature::gradN_vector

No description yet, please consult the code.

GooseFEM::Element::Quad4::Nodal

No description yet, please consult the code.

GooseFEM::Element::Quad4::Gauss

No description yet, please consult the code.

3.2.3 General routines

GooseFEM::Element::asElementVector

No description yet, please consult the code.

3.3 GooseFEM::Vector

3.4 Compiling

3.4.1 Introduction

This module is header only. So one just has to `#include <GooseFEM/GooseFEM.h>`, somewhere in the source code, and to tell the compiler where the header-files are. For the latter, several ways are described below.

Before proceeding, a words about optimization. Of course one should use optimization when compiling the release of the code (`-O2` or `-O3`). But it is also a good idea to switch off the assertions in the code (mostly checks on size) that facilitate easy debugging, but do cost time. Therefore, include the flag `-DNDEBUG`. Note that this is all C++ standard. I.e. it should be no surprise, and it always a good idea to do.

Note: This code depends on `eigen3` and `cppmat`. Both are also header-only libraries. Both can be ‘installed’ using identical steps as described below.

3.4.2 Manual compiler flags

GNU / Clang

Add the following compiler's arguments:

```
-I${PATH_TO_GOOSEFEM}/src -std=c++14
```

Note: (Not recommended)

If you want to avoid separately including the header files using a compiler flag, `git submodule` is a nice way to go:

1. Include this module as a submodule using `git submodule add https://github.com/tdegeus/GooseFEM.git`.
2. Replace the first line of this example by `#include "GooseFEM/src/GooseFEM/GooseFEM.h"`.

If you decide to manually copy the header file, you might need to modify this relative path to your liking.

Or see *(Semi-)Automatic compiler flags*. You can also combine the `git submodule` with any of the below compiling strategies.

3.4.3 (Semi-)Automatic compiler flags

Install

To enable (semi-)automatic build, one should 'install' GooseFEM somewhere.

Install system-wide (root)

1. Proceed to a (temporary) build directory. For example

```
$ cd /path/to/GooseFEM/src/build
```

2. 'Build' GooseFEM

```
$ cmake ..
$ make install
```

(If you've used another build directory, change the first command to `$ cmake /path/to/GooseFEM/src`)

Install in custom location (user)

1. Proceed to a (temporary) build directory. For example

```
$ cd /path/to/GooseFEM/src/build
```

2. 'Build' GooseFEM to install it in a custom location

```
$ mkdir /custom/install/path
$ cmake .. -DCMAKE_INSTALL_PREFIX:PATH=/custom/install/path
$ make install
```

(If you've used another build directory, change the first command to `$ cmake /path/to/GooseFEM/src`)

3. Add the following path to your `~/.bashrc` (or `~/.zshrc`):

```
export PKG_CONFIG_PATH=/custom/install/path/share/pkgconfig:$PKG_CONFIG_PATH
```

Note: (Not recommended)

If you do not wish to use CMake for the installation, or you want to do something custom. You can of course. Follow these steps:

1. Copy the file `src/GooseFEM.pc.in` to `GooseFEM.pc` to some location that can be found by `pkg-config` (for example by adding `export PKG_CONFIG_PATH=/path/to/GooseFEM.pc:$PKG_CONFIG_PATH` to the `.bashrc`).
 2. Modify the line `prefix=@CMAKE_INSTALL_PREFIX@` to `prefix=/path/to/GooseFEM`.
 3. Modify the line `Cflags: -I${prefix}/@INCLUDE_INSTALL_DIR@` to `Cflags: -I${prefix}/src`.
 4. Modify the line `Version: @GOOSEFEM_VERSION_NUMBER@` to reflect the correct release version.
-

Compiler arguments from 'pkg-config'

Instead of `-I...` one can now use

```
`pkg-config --cflags GooseFEM` -std=c++14
```

as compiler argument.

Compiler arguments from 'cmake'

Add the following to your `CMakeLists.txt`:

```
set(CMAKE_CXX_STANDARD 14)

find_package(PkgConfig)

pkg_check_modules(GOOSEFEM REQUIRED GooseFEM)
include_directories(${GOOSEFEM_INCLUDE_DIRS})
```

3.5 Notes for developers

3.5.1 Create a new release

1. Update the version number as follows in `src/GooseFEM/Macros.h`. The C++ and Python distributions will read from this.

2. Upload the changes to GitHub and create a new release there (with the correct version number).
3. Upload the package to PyPi:

```
$ python3 setup.py bdist_wheel --universal
$ twine upload dist/*
```

3.6 Dynamics

3.6.1 Time discretisation

Verlet

```
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});

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

GooseFEM::MatrixDiagonal M(...);

...

while ( ... )
{
    // 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;

    ...

    // new acceleration

    M.solve(fres, a);

    // new velocity

    xt::noalias(v) = v_n + .5 * dt * ( a_n + a );
}
```

Velocity Verlet

```
while ( ... )
{
    // variables & history
```

(continues on next page)

```
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;

...

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

...

M.solve(fres, a);

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

...

// new velocity
M.solve(fres, a);

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

...

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

}
```

Forward Euler

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

...

while ( ... )
{
    xt::noalias(u) = u + dt * v;

    ...
}
```

Tip: A compact reader covering the basic theory is available [here](#)

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- `search`