GooseFEM Documentation

Release

Tom de Geus

Nov 21, 2017

Contents

CHAPTER 1

Introduction

GooseFEM provides several standard finite element simulations, some common element definitions, and some simple finite element meshes. A basic documentation is provided here, whereby one is highly urged to look into the code itself, at minimum the function decelerators and their comments.

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.

Download: .zip file | .tar.gz file.

(c - GPLv3) T.W.J. de Geus (Tom) | tom@geus.me | www.geus.me | github.com/tdegeus/GooseFEM

CHAPTER 2

Contents

2.1 GooseFEM documentation

2.1.1 Compiling

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.

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.

(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})
```

2.1.2 GooseFEM::Mesh

Note: Source:

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

GooseFEM::Mesh::dofs

Get a sequential list of DOF-numbers for each vector-component of each node.

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

For example for 3 nodes in 2 dimensions the output is

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

GooseFEM::Mesh::renumber

- Renumber indices to lowest possible indices (returns a copy, input not modified).

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

For example:

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

is renumbered to

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

- Reorder indices such that some items are at the beginning or the end (returns a copy, input not modified).

```
MatS GooseFEM::Mesh::renumber ( const MatS &in , const ColS &idx, std::string␣
↪location="end" );
```

For example:

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

with

$$\text{idx} = \begin{bmatrix} 6 & 4 \end{bmatrix}$$

Implies that `in` is renumbered such that the 6th item becomes the one-before-last item ($5 \rightarrow 4$), and the 4th item become the last ($3 \rightarrow 5$). The remaining items are renumbered to the lowest index while keeping the same order. The result:

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

Consider also [`source: figures/Mesh/renumber.cpp`]

2.1.3 GooseFEM::Mesh::Tri3

Note: Source:

```
#include <GooseFEM/MeshTri3.h>
```

2.1.4 GooseFEM::Mesh::Quad4

Note: Source:

```
#include <GooseFEM/MeshQuad4.h>
```

GooseFEM::Mesh::Quad4::Regular

```
GooseFEM::Mesh::Quad4::Regular(size_t nx, size_t ny, 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
```

2.1.5 GooseFEM::Dynamics::Diagonal

Note: Source:

```
#include <GooseFEM/DynamicsDiagonalPeriodic.h>
#include <GooseFEM/DynamicsDiagonalSemiPeriodic.h>
#include <GooseFEM/DynamicsDiagonalLinearStrainQuad4.h>
```

Overview

Principle

The philosophy is to provide some structure to efficiently run a finite element simulation which remains customizable. Even more customization can be obtained by copy/pasting the source and modifying it to your need. The idea that is followed involves a hierarchy of three classes, whereby a class that is higher in hierarchy writes to some field of the class that is lower in hierarchy, runs a function, and reads from some field. In general:

- **Discretized system** (`GooseFEM::Dynamics::Diagonal::Periodic`, `GooseFEM::Dynamics::Diagonal::SemiPeriodic`).

  Defines the discretized system, and assembles the (inverse) mass matrix, the displacement dependent forces, and the velocity dependent forces from element arrays that are provided by the element definition.

- **Element definition** (`GooseFEM::Dynamics::Diagonal::LinearStrain::Qaud4`)

  Provides the element arrays by performing numerical quadrature. At the integration point the constitutive response is probed from the quadrature point definition.

- **Quadrature point definition**

  This class is not provided, and should be provided by the user.

Example

A simple example is:

[source: examples/DynamicsDiagonalPeriodic/laminate/no_damping/Verlet/main.cpp]

```
#include <Eigen/Eigen>
#include <cppmat/cppmat.h>
#include <GooseFEM/GooseFEM.h>
#include <GooseMaterial/AmorphousSolid/LinearStrain/Elastic/Cartesian2d.h>

// -----------------------------------------------------------------------

using      MatS = GooseFEM::MatS;
using      MatD = GooseFEM::MatD;
using      ColD = GooseFEM::ColD;

using      T2   = cppmat::cartesian2d::tensor2 <double>;
using      T2s  = cppmat::cartesian2d::tensor2s<double>;

namespace GM    = GooseMaterial::AmorphousSolid::LinearStrain::Elastic::Cartesian2d;

// =======================================================================

class Quadrature
{
public:
  T2s eps, epsdot, sig;

  size_t nhard;
  GM::Material hard, soft;

  double Ebar, Vbar;
```

```
  Quadrature(size_t nhard);

  double density             (size_t elem, size_t k, double V);
  void   stressStrain        (size_t elem, size_t k, double V);
  void   stressStrainRate    (size_t elem, size_t k, double V);
  void   stressStrainPost    (size_t elem, size_t k, double V);
  void   stressStrainRatePost(size_t elem, size_t k, double V);
};

// ------------------------------------------------------------------------

Quadrature::Quadrature(size_t _nhard)
{
  nhard    = _nhard;
  hard     = GM::Material(100.,10.);
  soft     = GM::Material(100., 1.);
}

// ------------------------------------------------------------------------

double Quadrature::density(size_t elem, size_t k, double V)
{
  return 1.0;
}
// ------------------------------------------------------------------------

void Quadrature::stressStrain(size_t elem, size_t k, double V)
{
  if ( elem < nhard ) sig = hard.stress(eps);
  else                sig = soft.stress(eps);
}
// ------------------------------------------------------------------------

void Quadrature::stressStrainRate(size_t elem, size_t k, double V)
{
}
// ------------------------------------------------------------------------

void Quadrature::stressStrainPost(size_t elem, size_t k, double V)
{
  Vbar += V;

  if ( elem < nhard ) Ebar += hard.energy(eps) * V;
  else                Ebar += soft.energy(eps) * V;
}

// ------------------------------------------------------------------------

void Quadrature::stressStrainRatePost(size_t elem, size_t k, double V)
{
}

// ========================================================================

int main()
{
  // class which provides the mesh
```

```
  GooseFEM::Mesh::Quad4::Regular mesh(40,40,1.);

  // class which provides the constitutive response at each quadrature point
  auto  quadrature = std::make_shared<Quadrature>(40*40/4);

  // class which provides the response of each element
  using Elem = GooseFEM::Dynamics::Diagonal::LinearStrain::Quad4<Quadrature>;
  auto  elem = std::make_shared<Elem>(quadrature);

  // class which provides the system and an increment
  GooseFEM::Dynamics::Diagonal::Periodic<Elem> sim(
    elem,
    mesh.coor(),
    mesh.conn(),
    mesh.dofsPeriodic(),
    1.e-2,
    0.0
  );

  // loop over increments
  for ( ... )
  {
    // - set displacement of fixed DOFs
    ...

    // - compute time increment
    sim.Verlet();

    // - post-process
    quadrature->Ebar = 0.0;
    quadrature->Vbar = 0.0;

    sim.post();

    ...
  }

  return 0;
}
```

Pseudo-code

What is happening inside `Verlet` is evaluating the forces (and the mass matrix), and updating the displacements by solving the system. In pseudo-code:

- Mass matrix:

```
sim.computeMinv():
{
  for e in elements:

    sim->elem->xe(i,j) = ...
    sim->elem->ue(i,j) = ...

    sim->elem->computeM(e):
    {
```

```
      for k in integration-points:

        sim->elem->M(...,...) += ... * sim->elem->quad->density(e,k,V)
    }

    M(...) += sim->elem->M(i,i)
}
```

- Displacement dependent force:

```
sim.computeFu():
{
  for e in elements:

    sim->elem->xe(i,j) = ...
    sim->elem->ue(i,j) = ...

    sim->elem->computeFu(e):
    {
      for k in integration-points:

        sim->elem->quad->eps(i,j) = ...

        sim->elem->quad->stressStrain(e,k,V)

        sim->elem->fu(...) += ... * sim->elem->quad->sig(i,j)
    }

    Fu(...) += sim->elem->fu(i)
}
```

- Velocity dependent force:

```
sim.computeFv():
{
  for e in elements:

    sim->elem->xe(i,j) = ...
    sim->elem->ue(i,j) = ...
    sim->elem->ve(i,j) = ...

    sim->elem->computeFv(e):
    {
      for k in integration-points:

        sim->elem->quad->epsdot(i,j) = ...

        sim->elem->quad->stressStrainRate(e,k,V)

        sim->elem->fv(...) += ... * sim->elem->quad->sig(i,j)
    }

    Fv(...) += sim->elem->fu(i)
}
```

Signature

From this it is clear that:

- `GooseFEM::Dynamics::Diagonal::Periodic` requires the following minimal signature from `GooseFEM::Dynamics::Diagonal::LinearStrain::Qaud4`:

```
class Element
{
public:
  matrix M;                    // should have operator(i,j)
  column fu, fv;               // should have operator(i)
  matrix xe, ue, ve;           // should have operator(i,j)

  void computeM (size_t elem); // mass matrix                      <- quad->density
  void computeFu(size_t elem); // displacement dependent forces    <- quad->
↪stressStrain
  void computeFv(size_t elem); // displacement dependent forces    <- quad->
↪stressStrainRate
  void post     (size_t elem); // post-process                     <- quad->
↪stressStrain(Rate)
}
```

- `GooseFEM::Dynamics::Diagonal::LinearStrain::Qaud4` requires the minimal signature from `Quadrature`

```
class Quadrature
{
public:
  tensor eps, epsdot, sig;     // should have operator(i,j)

  double density             (size_t elem, size_t k, double V);
  void   stressStrain        (size_t elem, size_t k, double V);
  void   stressStrainRate    (size_t elem, size_t k, double V);
  void   stressStrainPost    (size_t elem, size_t k, double V);
  void   stressStrainRatePost(size_t elem, size_t k, double V);
}
```

2.1.6 Notes for developers

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/*
```

2.2 Theory

2.2.1 Finite Element Method

Contents

In the sequel the theory of the Finite Element is discussed in a compact way. This discussion is by no means comprehensive. Therefore one is invited to dive in more complete textbooks. The key contribution of this reader is that it is supported by many examples that can be easily extended and customized into efficient, production ready code. To this end, the examples are in C++14, and are specifically written such that they are mostly 'what you see is what you get'. The entire structure is in the main-file, and not hidden somewhere in a library. To simplify your life we do use several libraries, each of which however only with a dedicated task, which can be understood, used, and checked independently of the Finite Element Method or any specific application. More specifically we use:

- GooseMaterial

  Provides the constitutive response (and optionally the constitutive tangent) of several materials.

- cppmat

  Provides tensor classes and operations. (The amount of tensor-operations is limited in the main program, and even non-standard, but this library is crucial to compute the material response implemented in GooseMaterial.)

- Eigen3

  A linear algebra library. As you will notice, Eigen plays an important role in GooseFEM, and glues everything together since in the end the Finite Element Method is just a way to cast a problem into a set linear or linearized equations. Most of the efficiency of the final program will depend on the efficiency of the implementation of the linear algebra. In several examples we will simplify the structure by using dense matrices together with a simple solver which solves the resulting linear system. In reality one should always use sparse matrices combined with a more efficient solver. As you will notice, many examples need only few changes to be transformed in a production code.

Note: Compilation

Unless otherwise mentioned, the examples can be compiled as follows. Provided that `pkg-config` is set-up correctly one can use

```
clang++ `pkg-config --cflags Eigen3 cppmat GooseMaterial GooseFEM` -std=c++14 -o␣
↪example example_name.cpp
```

(whereby `clang++` can be replaced by for example `g++`). If one does not want to use `pkg-config`, one has to specify `-I/path/to/library` for each of the libraries.

For further development it is strongly advised to include the options `-Wpedantic -Wall` to get on top of mistakes. Once the code is ready, one should compile with optimization (`-O3`) and without assertions (`-DNDEBUG`). The Eigen3 documentation further recommends the option `-march=native` to enable vectorization optimized for you architecture.

Statics

The conceptual idea

We begin our discussion by considering a static, solid mechanics, problem. Loosely speaking the the goal is to find a deformation map, $\vec{x} = \varphi(\vec{X}, t)$, that maps a body Ω_0 to a deformed state Ω that satisfies equilibrium and the boundary conditions applied on Γ.

As is the case in the illustration, the body can be subjected to two kinds of boundary conditions:

- *Essential* or *Dirichlet* boundary conditions on Γ_p, whereby the displacements are prescribed.
- *Natural* or *Neumann* boundary conditions on Γ_u, whereby the tractions are prescribed. (Whereby traction-free is also perfectly acceptable.)

In practice, we are not explicitly looking for the map $\vec{x} = \varphi(\vec{X}, t)$, but for the deformation gradient F, or in fact for a displacement field $\vec{u} = \vec{x} - \vec{X}$. To make things a bit more explicit, the deformation gradient is defined as follows:

$$\vec{x} = F \cdot \vec{X}$$

hence

$$F = \frac{\partial \varphi}{\partial \vec{X}} = \left(\vec{\nabla}_0 \, \vec{x}\right)^T = I + \left(\vec{\nabla}_0 \, \vec{u}\right)^T$$

Momentum balance

We start from the linear momentum balance:

$$\vec{\nabla} \cdot \sigma(\vec{x}) = \vec{0} \qquad \vec{x} \in \Omega$$

where σ is the Cauchy stress which depends on the new position $\vec{x}$ and thus on the displacement $\vec{u}$. It has been assumed that all actions are instantaneous (no inertia) and, for simplicity, that there are no body forces. Loosely speaking the interpretation of this equation is that *the sum of all forces vanishes* everywhere in the domain Ω

Note: The following nomenclature has been used

$$\vec{\nabla} \cdot \sigma = \frac{\partial \sigma_{ij}}{\partial x_i}$$

The crux of the Finite Element Method is that this non-linear differential equation is solved in a weak sense. I.e.

$$\int_\Omega \vec{\phi}(\vec{X}) \cdot \left[\vec{\nabla} \cdot \sigma(\vec{x}) \right] \, \mathrm{d}\Omega = 0 \qquad \forall \, \vec{\phi}(\vec{X}) \in \mathbb{R}^d$$

where $\vec{\phi}$ are test functions. For reasons that become obvious below, we apply integration by parts, which results in

$$\int_\Omega \left[\vec{\nabla}\vec{\phi}(\vec{X}) \right] : \sigma(\vec{x}) \, \mathrm{d}\Omega = \int_\Omega \vec{\nabla} \cdot \left[\vec{\phi}(\vec{X}) \cdot \sigma(\vec{x}) \right] \, \mathrm{d}\Omega \qquad \forall \, \vec{\phi}(\vec{X}) \in \mathbb{R}^d$$

Note: Use has been made of the following chain rule

$$\vec{\nabla} \cdot \left[\vec{\phi} \cdot \sigma^T \right] = \left[\vec{\nabla}\vec{\phi} \right] : \sigma^T + \vec{\phi} \cdot \left[\vec{\nabla} \cdot \sigma \right]$$

together with the symmetry of the Cauchy stress

$$\sigma = \sigma^T$$

and the following nomenclature:

$$C = A : B = A_{ij} B_{ji}$$

The right-hand-side of this equation can be reduced to an area integral by employing Gauss' divergence theorem. The result reads

$$\int_\Omega \left[\vec{\nabla}\vec{\phi}(\vec{X}) \right] : \sigma(\vec{x}) \, \mathrm{d}\Omega = \int_\Gamma \vec{\phi}(\vec{X}) \cdot \underbrace{\vec{n}(\vec{x}) \cdot \sigma(\vec{x})}_{\vec{t}(\vec{x})} \, \mathrm{d}\Gamma \qquad \forall \, \vec{\phi}(\vec{X}) \in \mathbb{R}^d$$

Note: Gauss' divergence theorem states that

$$\int_\Omega \vec{\nabla} \cdot \vec{a}(\vec{x}) \, \mathrm{d}\Omega = \int_\Gamma \vec{n}(\vec{x}) \cdot \vec{a}(\vec{x}) \, \mathrm{d}\Gamma$$

where $\vec{n}$ is the normal along the surface Γ.

Discretization

The problem is now discretized using n *nodes* that are connected through *elements*, which define the discretized domain Ω_0^h. *Shape functions* $N_i(\vec{X})$ are used to extrapolate the nodal quantities throughout the domain Ω_0^h (and Ω^h), as follows:

$$\vec{x}(\vec{X}, t) \approx \vec{x}^h(\vec{X}, t) = \sum_{i=1}^{n} N_i(\vec{X}) \, \vec{x}_i(t) = \underline{N}^{\mathsf{T}}(\vec{X}) \, \underline{\vec{x}}(t)$$

Following standard Galerkin

$$\vec{\phi}(\vec{X}) \approx \vec{\phi}^h(\vec{X}) = \underline{N}^{\mathsf{T}}(\vec{X}) \, \underline{\vec{\phi}}$$

Note: Applied to our problem sketch, a discretization might look like this. The nodes are clearly marked as circles. The lines connecting the nodes clearly mark the elements which are in this case three-node triangles (Tri3 in GooseFEM)

Applied to the balance equation we obtain

$$\vec{\underline{\phi}}^{\mathsf{T}} \cdot \int_{\Omega^h} \left[\vec{\nabla} \underline{N}(\vec{X}) \right] \cdot \sigma(\vec{x}) \, \mathrm{d}\Omega = \vec{\underline{\phi}}^{\mathsf{T}} \cdot \int_{\Gamma^h} \underline{N}(\vec{X}) \cdot \vec{t}(\vec{x}) \, \mathrm{d}\Gamma \qquad \forall \, \vec{\underline{\phi}} \in \mathbb{R}^d_n$$

from which the dependency on $\vec{\underline{\phi}}$ can be dropped:

$$\int_{\Omega^h} \left[\vec{\nabla} \underline{N}(\vec{X}) \right] \cdot \sigma(\vec{x}) \, \mathrm{d}\Omega = \int_{\Gamma^h} \underline{N}(\vec{X}) \cdot \vec{t}(\vec{x}) \, \mathrm{d}\Gamma$$

This corresponds to (non-linear) set of nodal balance equations:

$$\vec{\underline{f}}(\vec{x}) = \vec{\underline{t}}(\vec{x})$$

with:

- *Internal forces*

$$\vec{\underline{f}}(\vec{x}) = \int_{\Omega^h} \left[\vec{\nabla} \underline{N}(\vec{X}) \right] \cdot \sigma(\vec{x}) \, \mathrm{d}\Omega$$

- *Boundary tractions*

$$\vec{\underline{t}}(\vec{x}) = \int_{\Gamma^h} \underline{N}(\vec{X}) \cdot \vec{t}(\vec{x}) \, \mathrm{d}\Gamma$$

  which is zero in the interior of the domain, i.e. in $\Omega^h \bigcap \Gamma^h$, while they can be zero or non-zero in Γ^h depending on the problem details.

Iterative solution – small strain

A commonly used strategy to solve the non-linear system, is the iterative Newton-Raphson scheme (see inset below). The idea is thereby to formulate an initial guess for the solution, determine possible residual forces, and use these forces to come to a better guess for the solution. This is continued until the solution has been found, i.e. when the residual vanishes.

This solution technique is discussed here in the context of small deformations, while it is later generalized. Assuming the deformations to be small allows us to assume that $\Omega = \Omega_0$, and thus that $\nabla = \nabla_0$. Also we define a strain tensor

$$\varepsilon = \tfrac{1}{2} \left[\nabla_0 \vec{u} + \left[\nabla_0 \vec{u} \right]^T \right] = \mathbb{I}_s : \left[\nabla_0 \vec{u} \right]$$

and use some non-linear relationship between it and the stress

$$\sigma = \sigma\big(\varepsilon\big)$$

To simplify our discussion we assume the boundary tractions to be some known constant. Our nodal equilibrium equations now read

$$\underline{\vec{r}}(\vec{x}) = \underline{\vec{t}} - \underline{\vec{f}}(\vec{x}) = \underline{\vec{0}}$$

with

$$\underline{\vec{f}}(\vec{x}) = \int_{\Omega_0^h} \left[\vec{\nabla}_0 \underline{N}(\vec{X}) \right] \cdot \sigma(\vec{x}) \, \mathrm{d}\Omega$$

To come to an iterative solution, we linearize as this point. This results in

$$\int_{\Omega_0^h} \left[\vec{\nabla}_0 \underline{N}(\vec{X}) \right] \cdot \mathbb{K}\big(\vec{x}_{(i)}\big) \cdot \left[\vec{\nabla}_0 \underline{N}(\vec{X}) \right]^{\mathsf{T}} \mathrm{d}\Omega \cdot \delta\underline{\vec{x}} = \underline{\vec{t}} - \int_{\Omega_0^h} \left[\vec{\nabla}_0 \underline{N}(\vec{X}) \right] \cdot \sigma\big(\vec{x}_{(i)}\big) \, \mathrm{d}\Omega$$

where

$$\mathbb{K}\big(\vec{x}_{(i)}\big) = \left. \frac{\partial \sigma}{\partial \varepsilon} \right|_{\vec{x}_{(i)}} : \mathbb{I}_s$$

where the left part is the *constitutive tangent operator* and the right part comes from the strain definition. Note that this right part, the symmetrization using $\mathbb{I}_s$, can often be omitted as many *constitutive tangent operators* already symmetrize.

In a shorter notation, this is our iterative update:

$$\underline{\underline{\mathbb{K}}}_{(i)} \cdot \delta\underline{\vec{x}} = \underline{\vec{t}} - \underline{\vec{f}}_{(i)}$$

with

$$\underline{\underline{\mathbb{K}}}_{(i)} = \int_{\Omega_0^h} \left[\vec{\nabla}_0 \underline{N} \right] \cdot \mathbb{K}\big(\vec{x}_{(i)}\big) \cdot \left[\vec{\nabla}_0 \underline{N} \right]^{\mathsf{T}} \mathrm{d}\Omega$$

and

$$\underline{\vec{f}}_{(i)} = \int_{\Omega_0^h} \left[\vec{\nabla}_0 \underline{N} \right] \cdot \sigma\big(\vec{x}_{(i)}\big) \, \mathrm{d}\Omega$$

Note: This is a good point to study some examples:

- *Linear statics – small strain*

  We slowly work up to an iterative scheme starting from a linear problem, written however in such a way that the step towards a non-linear problem is small.

- *Non-linear statics – small strain*

  Here we employ Newton-Raphson to solve the non-linear equilibrium equation. It is easy to see that once the above examples have been understood this step is indeed trivial.

Note: Newton-Raphson in one dimension

We try to find x such that

$$r(x) = 0$$

We will make a guess for x and (hopefully) iteratively improve this guess. This iterative value is denoted using $x_{(i)}$. Therefore we will make use of the following Taylor expansion

$$r\big(x_{(i+1)}\big) = r\big(x_{(i)}\big) + \left.\frac{\mathrm{d}r}{\mathrm{d}x}\right|_{x=x_{(i)}} \delta x + \mathcal{O}\big(\delta x^2\big) \approx 0$$

where

$$\delta x = x_{(i+1)} - x_{(i)}$$

We now determine δx by neglecting higher order terms, which results in

$$r\big(x_{(i)}\big) + \left.\frac{\mathrm{d}r}{\mathrm{d}x}\right|_{x=x_{(i)}} \delta x = 0$$

From which we obtain δx as

$$\delta x = -\left[\left.\frac{\mathrm{d}r}{\mathrm{d}x}\right|_{x=x_{(i)}}\right]^{-1} r\big(x_{(i)}\big)$$

Thereafter we set

$$x_{(i+1)} = x_{(i)} + \delta x$$

And check if we are have reached our solution within a certain accuracy ϵ:

$$\big|r\big(x_{(i+1)}\big)\big| < \epsilon$$

If not, we repeat the above steps until we do.

The iterative scheme is well understood from the following illustration:

Dynamics

Momentum balance

We continue with our balance equation and add inertia an damping to it:

$$\rho\, \ddot{\vec{x}} = \vec{\nabla} \cdot \sigma(\vec{x}) + \eta\, \nabla^2 \dot{\vec{x}} \qquad \vec{x} \in \Omega$$

where ρ is the density and η the viscosity (a.k.a. the damping coefficient). The first and second time derivative of the position $\vec{x}$ are respectively the velocity $\vec{v} = \dot{\vec{x}}$ and the acceleration $\vec{a} = \ddot{\vec{x}}$.

We can generalize this as follows (which will also simplify our proceedings below)

$$\rho(\vec{x})\, \ddot{\vec{x}} = \vec{\nabla} \cdot \big[\, \sigma(\vec{x}) + \sigma_\eta(\dot{\vec{x}}) \,\big] \qquad \vec{x} \in \Omega$$

Note: To retrieve the original form

$$\sigma_\eta = \eta\, \vec{\nabla} \dot{\vec{x}}$$

But, we can now use also other expressions. For example the damping equivalent of linear elasticity:

$$\sigma_\eta(\vec{x}) = \mathbb{C}_\eta(\vec{x}) : \dot{\varepsilon}(\vec{x})$$

with

$$\mathbb{C}_\eta(\vec{x}) = \kappa(\vec{x}) I \otimes I + 2\gamma(\vec{x}) \mathbb{I}_d$$

where κ is the bulk viscosity while γ is the shear viscosity. Furthermore

$$\dot{\varepsilon}(\vec{x}) = \tfrac{1}{2}\left[\, \vec{\nabla}\dot{\vec{x}} + [\, \vec{\nabla}\dot{\vec{x}}]^T \,\right]$$

Our original form is retrieved when $\kappa = \frac{2}{3}\gamma$, both are independent of $\vec{x}$, and $\dot{\vec{x}}$ possesses the necessary symmetries.

Like before, we will solve this equation in a weak sense

$$\int_\Omega \rho(\vec{x})\ \vec{\phi}(\vec{X}) \cdot \ddot{\vec{x}}\ \mathrm{d}\Omega = \int_\Omega \vec{\phi}(\vec{X}) \cdot \Big[\, \vec{\nabla} \cdot \left[\, \sigma(\vec{x}) + \sigma_\eta(\dot{\vec{x}}) \,\right] \Big]\ \mathrm{d}\Omega \qquad \forall\ \vec{\phi}(\vec{X}) \in \mathbb{R}^d$$

Integration by parts results in

$$\int_\Omega \rho(\vec{x})\ \vec{\phi}(\vec{X}) \cdot \ddot{\vec{x}}\ \mathrm{d}\Omega = \int_\Gamma \vec{\phi}(\vec{X}) \cdot \left[\, \vec{t}(\vec{x}) + \vec{t}_\eta(\vec{x}) \,\right]\ \mathrm{d}\Gamma - \int_\Omega \left[\, \vec{\nabla}\vec{\phi}(\vec{X}) \,\right] : \left[\, \sigma(\vec{x}) + \sigma_\eta(\dot{\vec{x}}) \,\right]\ \mathrm{d}\Omega \qquad \forall\ \vec{\phi}(\vec{X}) \in \mathbb{R}^d$$

Which we will discretize as before:

$$\underline{\vec{\phi}}^{\mathsf{T}} \cdot \int_\Omega \rho(\vec{x})\ \underline{N}(\vec{X})\ \underline{N}^{\mathsf{T}}(\vec{X})\ \mathrm{d}\Omega\ \ddot{\underline{\vec{x}}} = \underline{\vec{\phi}}^{\mathsf{T}} \cdot \int_\Gamma \underline{N}(\vec{X})\ \left[\, \vec{t}(\vec{x}) + \vec{t}_\eta(\vec{x}) \,\right]\ \mathrm{d}\Gamma - \underline{\vec{\phi}}^{\mathsf{T}} \cdot \int_\Omega \left[\, \vec{\nabla}\underline{N}(\vec{X}) \,\right] : \left[\, \sigma(\vec{x}) + \sigma_\eta(\dot{\vec{x}}) \,\right]\ \mathrm{d}\Omega \qquad \forall\ \underline{\vec{\phi}} \in \mathbb{R}^d_n$$

Which is independent of the test functions, hence:

$$\int_\Omega \rho(\vec{x})\ \underline{N}(\vec{X})\ \underline{N}^{\mathsf{T}}(\vec{X})\ \mathrm{d}\Omega\ \ddot{\underline{\vec{x}}} = \int_\Gamma \underline{N}(\vec{X})\ \left[\, \vec{t}(\vec{x}) + \vec{t}_\eta(\vec{x}) \,\right]\ \mathrm{d}\Gamma - \int_\Omega \left[\, \vec{\nabla}\underline{N}(\vec{X}) \,\right] : \left[\, \sigma(\vec{x}) + \sigma_\eta(\dot{\vec{x}}) \,\right]\ \mathrm{d}\Omega$$

Which we can denote as follows

$$\underline{\underline{M}}(\vec{x})\ \ddot{\underline{\vec{x}}} = \underline{\vec{t}}(\vec{x}) + \underline{\vec{t}}_\eta(\vec{x}) - \underline{\vec{f}}(\vec{x}) - \underline{\vec{f}}_\eta(\vec{x})$$

whereby we have introduced:

- *Mass matrix*

$$\underline{\underline{M}}(\vec{x}) = \int_\Omega \rho(\vec{x})\ \underline{N}(\vec{X})\ \underline{N}^{\mathsf{T}}(\vec{X})\ \mathrm{d}\Omega$$

- *Boundary tractions*

$$\underline{\vec{t}}(\vec{x}) = \int_\Gamma \underline{N}(\vec{X})\ \vec{t}(\vec{x})\ \mathrm{d}\Gamma \qquad \text{and} \qquad \underline{\vec{t}}_\eta(\vec{x}) = \int_\Gamma \underline{N}(\vec{X})\ \vec{t}_\eta(\vec{x})\ \mathrm{d}\Gamma$$

- *Internal forces*

$$\underline{\vec{f}}(\vec{x}) = \int_\Omega \left[\, \vec{\nabla}\underline{N}(\vec{X}) \,\right] : \sigma(\vec{x})\ \mathrm{d}\Omega \qquad \text{and} \qquad \underline{\vec{f}}(\vec{x}) = \int_\Omega \left[\, \vec{\nabla}\underline{N}(\vec{X}) \,\right] : \sigma_\eta(\dot{\vec{x}})\ \mathrm{d}\Omega$$

Note: In many problems it make sense to assume the mass matrix constant, as any change of volume results in an equivalent change of the density, i.e.

$$\int_{\Omega} \rho(\vec{x}) \, \mathrm{d}\Omega = \int_{\Omega_0} \rho(\vec{X}) \, \mathrm{d}\Omega_0$$

This results in the following expression for the mass matrix:

$$\underline{\underline{M}}(\vec{X}) = \int_{\Omega_0} \rho(\vec{X}) \, \underline{N}(\vec{X}) \, \underline{N}^{\mathsf{T}}(\vec{X}) \, \mathrm{d}\Omega_0 = \text{constant}$$

Time discretization

Here we will discuss several common time discretization steps. To simplify notation we will denote the velocity $\vec{v} = \dot{\vec{x}}$ and the acceleration $\vec{a} = \ddot{\vec{x}}$.

Note: Most time integration schemes result is some form like

$$\underline{\underline{M}} \, \underline{\vec{a}}_{n+1} = \underline{\vec{q}}_n$$

where $\underline{\vec{q}}_n$ contains the boundary tractions and internal forces, including their damping equivalents. The subscript n indicates that the variable is a known quantity, while $n+1$ indicates that it is an unknown quantity. To enhance computational efficiency, it may be a good option to approximate the mass matrix in such a way that it becomes diagonal. Consequently, no system has be solved to find $\underline{\vec{a}}_{n+1}$. One only has to invert an array of scalars. Since in addition the mass matrix is almost often assumed constant, this factorization has to be performed only once for the entire simulation.

Physically one can interpret this assumption as assuming the damping to be concentrated on the nodes.

See: *Diagonal mass matrix*.

Note: References

Syllabus of the course "Computational Physics (PY 502)" by Anders Sandvik, Department of Physics, Boston University.

Velocity Verlet with damping

1. Compute the position at $t_{n+1} = t_n + \Delta_t$:

$$\vec{x}_{n+1} = \vec{x}_n + \Delta_t \vec{v}_n + \tfrac{1}{2} \Delta_t^2 \vec{a}_n$$

2. Estimate the velocity at $t_{n+1} = t_n + \Delta_t$:

$$\hat{\vec{v}}_{n+1} = \vec{v}_n + \tfrac{1}{2} \Delta_t \Big[\vec{a}_n + \vec{a}(\vec{x}_{n+1}, \vec{v}_n + \Delta_t \vec{a}_n, t_{n+1}) \Big]$$

3. Correct $\hat{\vec{v}}_{n+1}$:

$$\vec{v}_{n+1} = \vec{v}_n + \tfrac{1}{2} \Delta_t \Big[\vec{a}_n + \vec{a}(\vec{x}_{n+1}, \hat{\vec{v}}_{n+1}, t_{n+1}) \Big]$$

Shape functions

In the Finite Element Method a geometry is discretized using nodes. The nodes are grouped in elements which define the domain Ω_0^h. The crux of the method is that nodal quantities, for example $\vec{u}_i$, are extrapolated throughout the discretized domain Ω_0^h using shape functions $N_i(\vec{X})$. Each shape function is globally supported, however in such a way that $N_i(\vec{X}) \neq 0$ only in the elements containing node i. It is furthermore imposed that $N_i(\vec{X}_j) = \delta_{ij}$, i.e. it is one in the node i, and zero in all other nodes.

For a one-dimensional problem comprising four linear elements and five nodes the shape functions are sketched below (whereby the node numbers are in color, while the element numbers are in black, in between the nodes).

From this it becomes obvious that $N_i(\vec{X})$ is polynomial through each of the nodes, and that $\partial N_i / \partial \vec{X}$ is discontinuous across element boundaries. Note once more that each of the shape functions $N_i(X)$ is globally supported, but zero outside the elements that contain the node i. For node 2, the shape function is thus:

As we can see, node 2 is only non-zero in elements 1 and 2, while it is zero everywhere else. To evaluate $\vec{f}_2$ we therefore only have to integrate on these elements (using *Isoparametric transformation and quadrature*):

$$\vec{f}_2 = \int_{\Omega^1} \left[\vec{\nabla} N_2^1(\vec{X}) \right] \cdot \sigma(\vec{x}) \, \mathrm{d}\Omega + \int_{\Omega^2} \left[\vec{\nabla} N_2^2(\vec{X}) \right] \cdot \sigma(\vec{x}) \, \mathrm{d}\Omega$$

By now it should be clear that the above allows us assemble $\underline{f}$ element-by-element. For this example, graphically this corresponds to the following sum:

where the indices show that the *shape functions* are evaluated compared to some generic element definition (see *Isoparametric transformation and quadrature*).

Isoparametric transformation and quadrature

A very important concept in the Finite Element Method is the isoparametric transformation. It allows us to map an arbitrarily shaped element with volume Ω^e onto a generic *isoparametric element* of constant volume Q. By using this mapping it is easy to perform numerical quadrature while even reusing an existing implementation (for example the one of GooseFEM).

The mapping between the generic domain Q and the physical domain Ω^e is as follows

$$\vec{x}(\vec{\xi}) = \left[\underline{N}^e \right]^{\mathsf{T}} \underline{x}^e$$

where the column $\underline{x}^e$ contains the real position vectors of the element nodes. In order to perform the quadrature on Q we must map also the gradient operator:

$$\vec{\nabla}_\xi = \vec{e}_i \frac{\partial}{\partial \xi_i} = \vec{e}_i \frac{\partial x_j(\vec{\xi})}{\partial \xi_i} \frac{\partial}{\partial x_j} = \vec{e}_i \frac{\partial x_j(\vec{\xi})}{\partial \xi_i} \vec{e}_j \cdot \vec{e}_k \frac{\partial}{\partial x_k} = \left[\vec{\nabla}_\xi \, \vec{x}(\vec{\xi}) \right] \cdot \vec{\nabla} = J(\vec{\xi}) \cdot \vec{\nabla}$$

or

$$\vec{\nabla} = J^{-1}(\vec{\xi}) \cdot \vec{\nabla}_\xi$$

with

$$J(\vec{\xi}) = \vec{\nabla}_\xi \, \vec{x}(\vec{\xi}) = \left[\vec{\nabla}_\xi \, \underline{N}^e \right]^\mathsf{T} \underline{x}^e$$

Using the above:

$$\vec{\nabla} \underline{N}^e = J^{-1}(\vec{\xi}) \cdot \left[\vec{\nabla}_\xi \, \underline{N}^e \right]$$

We can now determine the mapping between the real and the master volume:

$$\mathrm{d}\Omega = \mathrm{d}\vec{x}_0 \times \mathrm{d}\vec{x}_1 \cdot \mathrm{d}\vec{x}_2 = \left[\mathrm{d}\vec{x}_0 \cdot J(\vec{\xi})\right] \times \left[\mathrm{d}\vec{x}_1 \cdot J(\vec{\xi})\right] \cdot \left[\mathrm{d}\vec{x}_2 \cdot J(\vec{\xi})\right] = \det\left(J(\vec{\xi})\right) \mathrm{d}\vec{\xi}_0 \times \mathrm{d}\vec{\xi}_1 \cdot \mathrm{d}\vec{\xi}_2 = \det\left(J(\vec{\xi})\right) \mathrm{d}Q$$

For example for the internal force this implies

$$\underline{\vec{f}^e} = \int_{\Omega^e} \left[\vec{\nabla} \underline{N} \right] \cdot \sigma(\vec{x}) \, \mathrm{d}\Omega = \int_Q \left[\vec{\nabla} \underline{N} \right] \cdot \sigma(\vec{x}) \; \det\left(J(\vec{\xi})\right) \, \mathrm{d}Q$$

Numerical quadrature can be formulated (exactly) on the master element. It corresponds to taking the weighted sum of the integrand evaluated at specific *quadrature points* (or *integration-points*). Again, for our internal force:

$$\underline{\vec{f}^e} = \sum_k^{n_k} w_k \left[\vec{\nabla} \underline{N} \right]_{\vec{\xi}=\vec{\xi}_k} \cdot \sigma\left(\vec{x}(\vec{\xi}_k)\right) \; \det\left(J(\vec{\xi}_k)\right)$$

Note: To obtain $\vec{X}(\vec{\xi})$, $\vec{\nabla}_0$, and $\int_{\Omega_0} . \, \mathrm{d}\Omega$, simply replace $\underline{x}^e$ with $\underline{X}^e$ in the first equation. For this reason the same element implementation (of for example GooseFEM) can be used in small strain and finite strain (total Lagrange and updated Lagrange), proving either $\underline{X}^e$ or $\underline{X}^e + \underline{u}^e$ as input.

Note: The details depend on the element type. Several standard elements types are implemented in GooseFEM.

2.3 Examples

2.3.1 Linear statics – small strain

Contents

Mixed boundary conditions

[source: fixedbnd-basic.cpp]

[source: fixedbnd.cpp]

Problem description

In this first example we will look at a linear elastic, 2-D plane strain problem, with imposed boundary displacements. The problem is sketched in the figure below. From which it is observed that:

1. The sample is homogeneous.
2. Symmetry conditions are assumed in x- and y-direction.
3. The outer (right) edge is displaced by λ in x-direction (whereby the displacement is constant in y-direction).
4. The boundary tractions are zero everywhere, except for where the displacement is prescribed. There a non-zero reaction force may appear.
5. (Not shown) We assume that $\Omega = \Omega_0$, and thus $\nabla = \nabla_0$. By doing this we can make the problem truly linear.

The geometry is discretized in 2-D linear quadrilateral elements, which have four nodes per element. To keep things simple we use only three, equi-sized, elements in each direction. The mesh is shown below, whereby the element numbers have a regular font while the node numbers are italic.

At this point we focus our attention on the internal force. Thereby we first consider the constitutive response:

$$\sigma = \mathbb{C} : \varepsilon$$

I.e the double contraction between the fourth-order stiffness

$$\mathbb{C} = KI \otimes I + 2G(\mathbb{I}_\mathrm{s} - \tfrac{1}{3} I \otimes I) = KI \otimes I + 2G\,\mathbb{I}_\mathrm{d}$$

(with K and G the bulk and the shear modulus), and the linear strain

$$\varepsilon = \tfrac{1}{2}\left[\,\vec{\nabla}_0 \vec{u} + [\,\vec{\nabla}_0 \vec{u}]^T\,\right]$$

(more information in the documentation of GooseSolid). Because of the symmetries in $\mathbb{C}$ we can simplify the constitutive expression as follows

$$\sigma = \mathbb{C} : \left[\,\vec{\nabla}_0 \vec{u}\,\right]$$

The displacement of the final configuration, $\vec{u}$, is now decomposed some known *pre-strain* $\vec{u}_{(0)}$ plus an unknown update $\delta\vec{u}$:

$$\vec{u} = \vec{u}_{(0)} + \delta\vec{u}$$

Note: For our simple problem, which is initially stress and strain free, we find that

$$\underline{\vec{f}}_{(0)} = \underline{\vec{t}}_{(0)} = \underline{\vec{u}}_{(0)} = \vec{0}$$

Some of the above expressions could thus be simplified further, while also part of the implementation can be omitted. We keep it here to build on this problem later on.

For the internal force this implies that

$$\underline{\vec{f}}(\vec{u}) = \underline{\vec{f}}(\vec{u}_{(0)}) + \underline{\vec{f}}(\delta\vec{u}) \qquad \text{or} \qquad \underline{\vec{f}} = \underline{\vec{f}}_{(0)} + \delta\underline{\vec{f}}$$

Since $\vec{u}_{(0)}$ is known we easily evaluate the original (in principle non-linear) expression of the internal force:

$$\underline{\vec{f}}_{(0)} = \int\limits_{\Omega_0^h} \left[\, \vec{\nabla}_0 \underline{N} \,\right] \cdot \sigma\big(\vec{u}_{(0)}\big) \, \mathrm{d}\Omega$$

For the update we use the explicit relation for the stress

$$\delta\underline{\vec{f}} = \int\limits_{\Omega_0^h} \left[\, \vec{\nabla}_0 \underline{N} \,\right] \cdot \sigma\big(\delta\vec{u}\big) \, \mathrm{d}\Omega = \int\limits_{\Omega_0^h} \left[\, \vec{\nabla}_0 \underline{N} \,\right] \cdot \mathbb{C} : \left[\, \vec{\nabla}_0 \, \delta\vec{u} \,\right] \, \mathrm{d}\Omega$$

We then again apply our discretization scheme to obtain

$$\delta\underline{\vec{f}} = \int\limits_{\Omega_0^h} \left[\, \vec{\nabla}_0 \underline{N} \,\right] \cdot \mathbb{C} \cdot \left[\, \vec{\nabla}_0 \underline{N} \,\right]^{\mathsf{T}} \mathrm{d}\Omega \cdot \delta\underline{u}$$

Hence:

$$\underline{\vec{f}} = \underline{\vec{f}}_{(0)} + \underline{\underline{K}}_{(0)} \cdot \delta\underline{\vec{u}}$$

The tractions $\underline{\vec{t}}_{(0)}$ are fully given by the boundary conditions, which also can be used to show that $\delta\underline{\vec{t}} = \underline{\vec{0}}$ – or at least for the relevant components (see below). The final force balance then reads:

$$\underline{\underline{K}}_{(0)} \cdot \delta\underline{\vec{u}} = \underline{\vec{t}}_{(0)} - \underline{\vec{f}}_{(0)}$$

We continue, by writing the problem in terms of scalar degrees-of-freedoms (DOFs). Each node has two DOFs, the two vector directions. For our mesh we define the DOFs as follows (whereby the cyan colored DOF-numbers correspond to the x-direction, while the magenta ones correspond to the y-direction).

By doing this, a little bit of book-keeping allows us to write our balance equation as the following system of equations

$$\underline{\underline{K}}_{(0)} \, \delta\underline{u} = \underline{t}_{(0)} - \underline{f}_{(0)}$$

In short

$$\underline{\underline{K}}_{(0)} \, \delta\underline{u} = \underline{r}_{(0)}$$

Next, we have to impose the boundary conditions. Given the fact that we impose displacements on part of the boundary here, a part of $\delta\underline{\vec{u}}$ will be prescribed. More specifically, our mesh looks as follows

where the yellow nodes are prescribed DOFs while the blue ones are yet unknown. We are now ready to solve the balance equation in two steps. First we will partition the system is unknown and prescribed DOFs:

$$\begin{bmatrix} \underline{\underline{K}}_{uu} & \underline{\underline{K}}_{up} \\ \underline{\underline{K}}_{pu} & \underline{\underline{K}}_{pp} \end{bmatrix} \begin{bmatrix} \delta\underline{u}_u \\ \delta\underline{u}_p \end{bmatrix} = \begin{bmatrix} \underline{r}_u \\ \underline{r}_p \end{bmatrix}$$

We are now ready to solve the former part:

$$\delta\underline{u}_u = \underline{\underline{K}}_{uu}^{-1} \big(\underline{r}_u - \underline{\underline{K}}_{up} \delta\underline{u}_p \big)$$

which gives us that

$$\delta \underline{u} = \begin{bmatrix} \delta \underline{u}_u \\ \delta \underline{u}_p \end{bmatrix}$$

and finally that

$$\underline{u} = \underline{u}_{(0)} + \delta \underline{u}$$

which can be reassembled as displacement vectors per node, $\underline{\vec{u}}$.

Should you be interested, one can compute the *reaction force*, i.e. the boundary tractions there where the displacement has been prescribed. One has to do the following:

$$\underline{t}_p = \underline{f}_p$$

For this, one thus has to compute the new $\underline{\vec{f}}(\vec{u})$. Because this specific model is linear we can however obtain the reaction forces without having to re-evaluate $\underline{\vec{f}}$. Specifically

$$\underline{t}_p = \underline{f}_p = (\underline{f}_{(0)})_p + \underline{\underline{K}}_{pu} \delta \underline{u}_u + \underline{\underline{K}}_{pp} \delta \underline{u}_p$$

Basic implementation

[source: fixedbnd-basic.cpp]

Our first attempt of an implementation literately follows the steps above: it constructs $\underline{\underline{K}}$ and $\underline{f}$, which are then partitioned. The prescribed displacements are then set. Thereafter the problem is solved, and the displacements are reconstructed to nodal vector for easy post-processing.

Pre-partitioning

[source: fixedbnd.cpp]

One of the things that made the previous examples not very suitable for becoming a production code is the fact that the stiffness matrix was first fully assembled and afterwards partitioned. Besides costing a lot of memory for storing the matrix twice, it might cost a lot of time since partitioning might become a costly operation in the case that sparse matrices are used. To avoid this, the system may be pre-partioned. In that case we renumber the DOFs such that we end up with first all the unknown DOFs (denoted `iiu` in the code), and then all the known DOFs (denoted `iip`). For our example this results in:

This allows us to consider four different matrices (denoted using `_uu`, `_up`, `_pu`, and `_pp`) and two different columns (denoted using `_u` and `_p`) to which the internal force and the stiffness are directly assembled. The rows (and columns) of these matrices and columns follow from introducing separate indices for `iiu` and `iip`:

Periodic boundary conditions

Note: Some additional notes on the theory discussed on a simplified scalar system, for the same mesh as presented here, are included in a separate document. One is invited to study this document before continuing.

[source: readme.pdf]

Prescribed macroscopic deformation

[source: periodic.cpp]

In our first example we will consider the same material and mesh as above. However, now we will assume periodicity is both spatial directions and prescribe a change in the macroscopic deformation gradient, equal to

$$\Delta F = \begin{bmatrix} 0 & 0.01 \\ 0 & 0 \end{bmatrix}$$

First of all we will specify periodicity for our mesh. It applies that the following equalities hold (in terms of node numbers:

$$\begin{aligned} \vec{u}^\star_3 &= \vec{u}^\star_0 \\ \vec{u}^\star_7 &= \vec{u}^\star_4 \\ \vec{u}^\star_{11} &= \vec{u}^\star_8 \\ \vec{u}^\star_{12} &= \vec{u}^\star_0 \\ \vec{u}^\star_{13} &= \vec{u}^\star_1 \\ \vec{u}^\star_{14} &= \vec{u}^\star_2 \\ \vec{u}^\star_{15} &= \vec{u}^\star_0 \end{aligned}$$

where $\vec{u}^\star$ are the microscopic fluctuations, that do not affect the macroscopic affine deformation. In terms of DOFs this is can be illustrated as follows:

where the red DOFs are said to be dependent (i.e. they directly follow from the equalities listed above). The simplest this that we can do is construct a system with only the independent DOFs (in blue above) by directly assembling to the independent DOFs. To this end we employ the following DOF numbers:

where the yellow color of the lower left corner indicates that this node is used as reference. Firstly it is used to suppress rigid body deformation. Secondly we apply the macroscopic deformation as the initial condition.

$$\underline{\vec{u}}_{(0)} = \Delta F \cdot \left[\underline{\vec{X}} - \vec{X}_{\mathrm{ref}} \right]$$

Final equilibrium is then obtained by solving

$$\underline{\underline{K}}^\star_{(0)} \, \delta \underline{u}^\star = -\underline{f}^\star_{(0)}$$

(which has the dimensions of the number of independent DOFs), and then assembling $\delta \underline{u}^\star$ to the entire system (including the dependent nodes). This is done in the first example, whereby the resulting system is partitioned to deal with the zero-displacement of the reference node.

Mixed macroscopic boundary conditions

[source: periodic-virtual-basic.cpp]

Here we will enable mixed macroscopic boundary conditions by introduction extra DOFs for the macroscopic deformation gradient tensor, and its antagonist stress response. Since we work in two dimensions we introduce two virtual nodes, each with two DOFs:

We will now employ the following tying relation

$$\vec{x}_d = \vec{x}_i + F \cdot (\vec{X}_d - \vec{X}_i)$$
$$\vec{u}_d = \vec{u}_i + (F - I) \cdot (\vec{X}_d - \vec{X}_i)$$

To this end we first renumber the system to have all the dependent DOFs at the end

And then partition the system in independent and dependent DOFs

Finally, we obtain the following tying relations for the DOFs

$$\underline{\underline{C}}_{di} = \begin{bmatrix}
1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & L_x & 0 & . & . \\
. & 1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & L_x & 0 \\
. & . & . & . & . & . & 1 & . & . & . & . & . & . & . & . & . & . & . & L_x & 0 & . & . \\
. & . & . & . & . & . & . & 1 & . & . & . & . & . & . & . & . & . & . & . & . & L_x & 0 \\
. & . & . & . & . & . & . & . & . & . & . & . & 1 & . & . & . & . & . & L_x & 0 & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . & . & . & . & . & . & L_x & 0 \\
1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 0 & L_y & . & . \\
. & 1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 0 & L_y \\
. & . & 1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 0 & L_y & . & . \\
. & . & . & 1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 0 & L_y \\
. & . & . & . & 1 & . & . & . & . & . & . & . & . & . & . & . & . & . & 0 & L_y & . & . \\
. & . & . & . & . & 1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 0 & L_y \\
1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & L_x & L_y & . & . \\
. & 1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & L_x & L_y
\end{bmatrix}$$

We then employ the nodal dependency to obtain a system of the independent DOFs only:

$$\left[\underline{\underline{K}}^{(0)}_{ii} + \underline{\underline{C}}^{\mathsf{T}}_{di}\underline{\underline{K}}^{(0)}_{di} + \underline{\underline{C}}^{\mathsf{T}}_{di}\underline{\underline{K}}^{(0)}_{dd}\underline{\underline{C}}_{di}\right] \delta\underline{u}^{\star}_{i} = -\left[\underline{f}^{(0)}_{i} + \underline{\underline{C}}^{\mathsf{T}}_{di}\underline{f}^{(0)}_{d}\right]$$

which, after solving, we can reconstruct to the dependent DOFs using

$$\delta\underline{u}^{\star}_{d} = \underline{\underline{C}}_{di}\delta\underline{u}^{\star}_{i}$$

Note: Towards a production code

Although not (yet) pursued here, it would make sense to partition the system as follows:

$$\begin{bmatrix}
\underline{\underline{K}}_{uu} & \underline{\underline{K}}_{up} & \underline{\underline{K}}_{ud} \\
\underline{\underline{K}}_{pu} & \underline{\underline{K}}_{pp} & \underline{\underline{K}}_{pd} \\
\underline{\underline{K}}_{du} & \underline{\underline{K}}_{dp} & \underline{\underline{K}}_{dd}
\end{bmatrix}
\begin{bmatrix}
\delta\underline{u}_{u} \\
\delta\underline{u}_{p} \\
\delta\underline{u}_{d}
\end{bmatrix} = -
\begin{bmatrix}
\underline{f}_{u} \\
\underline{f}_{p} \\
\underline{f}_{d}
\end{bmatrix}$$

Accompanied with the tying relations

$$\begin{bmatrix}
\underline{\underline{C}}_{du} & \underline{\underline{C}}_{dp}
\end{bmatrix}$$

And use the following condensation

$$\begin{bmatrix}
\underline{\underline{K}}^{(0)}_{uu} + \underline{\underline{C}}^{\mathsf{T}}_{du}\underline{\underline{K}}^{(0)}_{du} + \underline{\underline{C}}^{\mathsf{T}}_{du}\underline{\underline{K}}^{(0)}_{dd}\underline{\underline{C}}_{du} & \underline{\underline{K}}^{(0)}_{up} + \underline{\underline{C}}^{\mathsf{T}}_{du}\underline{\underline{K}}^{(0)}_{dp} + \underline{\underline{C}}^{\mathsf{T}}_{du}\underline{\underline{K}}^{(0)}_{dd}\underline{\underline{C}}_{dp} \\
\underline{\underline{K}}^{(0)}_{pu} + \underline{\underline{C}}^{\mathsf{T}}_{dp}\underline{\underline{K}}^{(0)}_{du} + \underline{\underline{C}}^{\mathsf{T}}_{dp}\underline{\underline{K}}^{(0)}_{dd}\underline{\underline{C}}_{du} & \underline{\underline{K}}^{(0)}_{pp} + \underline{\underline{C}}^{\mathsf{T}}_{dp}\underline{\underline{K}}^{(0)}_{dp} + \underline{\underline{C}}^{\mathsf{T}}_{dp}\underline{\underline{K}}^{(0)}_{dd}\underline{\underline{C}}_{dp}
\end{bmatrix}
\begin{bmatrix}
\delta\underline{u}^{\star}_{u} \\
\delta\underline{u}^{\star}_{p}
\end{bmatrix} = -
\begin{bmatrix}
\underline{f}^{(0)}_{u} + \underline{\underline{C}}^{\mathsf{T}}_{du}\underline{f}^{(0)}_{d} \\
\underline{f}^{(0)}_{p} + \underline{\underline{C}}^{\mathsf{T}}_{dp}\underline{f}^{(0)}_{d}
\end{bmatrix}$$

with the following reconstruction:

$$\delta\underline{u}^{\star}_{d} = \underline{\underline{C}}_{du}\delta\underline{u}^{\star}_{u} + \underline{\underline{C}}_{dp}\delta\underline{u}^{\star}_{p}$$

2.3.2 Non-linear statics – small strain

Contents

Here we extend the example from *Linear statics – small strain* to a non-linear constitutive response, which is however still subjected to a **small deformations** assumption. We treat the constitutive response as a black-box here:

$$\varepsilon \ \rightarrow \ \sigma, \mathbb{K}$$

where it must be emphasized that $\mathbb{K}$ symmetrizes. To understand more about the constitutive response, please consult the documentation of GooseSolid

Mixed boundary conditions

[source: fixedbnd.cpp]

In summary, our iterative update reads

$$\underline{\underline{K}}_{(i)} \cdot \delta\vec{\underline{x}} = \vec{\underline{t}} - \vec{\underline{f}}_{(i)}$$

with

$$\underline{\underline{K}}_{(i)} = \int_{\Omega_0^h} \left[\vec{\nabla}_0 \underline{N} \right] \cdot K\left(\vec{x}_{(i)}\right) \cdot \left[\vec{\nabla}_0 \underline{N} \right]^{\mathsf{T}} \, \mathrm{d}\Omega$$

and

$$\vec{\underline{f}}_{(i)} = \int_{\Omega_0^h} \left[\vec{\nabla}_0 \underline{N} \right] \cdot \sigma\left(\vec{x}_{(i)}\right) \, \mathrm{d}\Omega$$

We will use this to iteratively update

$$\vec{\underline{x}}_{(i+1)} = \vec{\underline{x}}_{(i)} + \delta\vec{\underline{x}}$$

From an 'initial guess'

$$\vec{\underline{x}}_{(0)} = \vec{\underline{0}}$$

Note: This is a bit of particular case. We need this iteration to get things going, however typically

$$\vec{\underline{t}} - \vec{\underline{f}}_{(0)} = \vec{\underline{0}}$$

One should not be fooled, this does not mean to an equilibrium has been obtained.

Like before, we introduce DOFs to make the system scalar. We then need to partition the system to deal with the prescribed displacement-components:

$$\begin{bmatrix} \underline{\underline{K}}_{uu}^{(i)} & \underline{\underline{K}}_{up}^{(i)} \\ \underline{\underline{K}}_{pu}^{(i)} & \underline{\underline{K}}_{pp}^{(i)} \end{bmatrix} \cdot \begin{bmatrix} \delta\underline{x}_u^{(i)} \\ \delta\underline{x}_p^{(i)} \end{bmatrix} = \begin{bmatrix} \underline{t}_u^{(i)} \\ \underline{t}_p^{(i)} \end{bmatrix} - \begin{bmatrix} \underline{f}_u^{(i)} \\ \underline{f}_p^{(i)} \end{bmatrix}$$

From which the update of the unknown DOFs as follows

$$\delta \underline{x}_u^{(i)} = \left[\underline{\underline{K}}_{uu}^{(i)} \right]^{-1} \left[\underline{t}_u^{(i)} - \underline{f}_u^{(i)} - \underline{\underline{K}}_{up}^{(i)} \delta \underline{x}_p^{(i)} \right]$$

There a very important concept hidden here. Because we prescribe $\delta \underline{x}_p$ directly to the correct value – which we do not iteratively update – we need to set

$$\delta \underline{x}_p^{(i)} = \begin{cases} \delta \underline{x}_p & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases}$$

(Which means that one does not have to compute the product $\underline{\underline{K}}_{up}^{(i)} \delta \underline{x}_p^{(i)}$ for $i > 0$, as it will be zero.)

Note: Reaction forces

To obtain the reaction forces on the prescribed DOFs simply use that

$$\underline{t}_p^{(i)} = \underline{f}_p^{(i)}$$

Which should be evaluated once convergence has been reached (before that, this has no meaning).

Periodic problem

[source: periodic.cpp]

In some ways the periodic example is even easier than the example above. As discussed previously $\delta \underline{\vec{x}}$ only leads to periodic fluctuations. In summary we begin by setting

$$\underline{\vec{x}}_{(0)} = \left[\bar{F} - I \right] \cdot \left[\underline{\vec{X}} - \underline{\vec{X}}_{\text{ref}} \right]$$

the update

$$\underline{\vec{x}}_{(i+1)} = \underline{\vec{x}}_{(i)} + \delta \underline{\vec{x}}$$

then only affects the fluctuations, and not the average. Also we do not need to worry about the above discussion on the prescribed displacements since they are always zero and traction free – as they merely suppress rigid body modes.

Note: The periodic implementation with macroscopic DOFs is identically extended to the non-linear case as above. Here one does have care about properly adding the prescribed displacements.

2.3.3 Diagonal mass matrix

'Theory'

[source: Tri3/complete.cpp]

[source: Tri3/lumped.cpp]

[source: Tri3/diagonal.cpp]

[source: Quad4/complete.cpp]

[source: Quad4/lumped.cpp]

[source: Quad4/diagonal.cpp]

Note: Should you still want to use the full (sparse) matrix in conjunction with a solver, please consider using the Intel compile with the Intel MKL library (which includes the Pardiso solver). Consider this example.

Because dynamic computations need so many increments, it may be worth to cut down the computational cost of a simulation. The best way to start cutting is on the solver, which is often the most costly of the entire Finite Element Program. Remember that

$$\underline{\underline{M}}(\vec{x})\ \underline{\vec{a}} = \underline{\vec{t}}(\vec{x}) - \underline{\vec{f}}(\vec{x}) - \underline{\underline{H}}(\vec{x})\ \underline{\vec{v}}$$

If we make the mass matrix diagonal, $\underline{\vec{a}}$ can be obtained without solving a system. Instead we merely needed the component wise inverse of the diagonal terms (which are the only remaining non-zero terms).

There are essentially four possible definitions of the mass matrix:

1. The *consistent* (or *complete*) mass matrix. Here the quadrature of the mass matrix is identical to that of the internal force in the system. It is the result of numerical integration of

$$\underline{\underline{M}}(\vec{x}) = \int_\Omega \rho(\vec{x})\ \underline{N}(\vec{X})\ \underline{N}^\mathsf{T}(\vec{X})\ \mathrm{d}\Omega$$

2. The *lumped* mass matrix. In which

$$M_{ii}^{\mathrm{lumped}} = \sum_j M_{ij}$$

3. The *lumped* mass matrix with *diagonal scaling*:

$$M_{ii}^{\mathrm{lumped}} = c \sum_j M_{ij}$$

   where the constant c is set such that:

$$\sum_i M_{ii}^{\mathrm{lumped}} = \int_\Omega \rho(\vec{x})\ \mathrm{d}\Omega$$

   A trick to accomplish this is to set

$$c = \frac{\sum_i \sum_j M_{ij}}{\sum_i M_{ii}}$$

   where the denominator is the trace of $\underline{\underline{M}}$.

4. Evaluating of $\underline{\underline{M}}$ using a quadrature rule involving only the nodal points and thus automatically yielding a diagonal matrix. It relies on the fact that when the quadrature-points coincide with the nodes, a local support is obtained (since the shape functions are zero in all the other nodes). The integration point volume is that part of the element's volume that can be associated to that node.

   For 2-D linear triangles, which have three nodes and normally one Gauss point in the middle of the element, the number of integration points is increased to three (whose positions correspond to that of the nodes), and their weight is set to 1/3 of the normal weight. Interestingly, for this case all possible definitions of the diagonalized mass matrix are the same.

   For 2-D quads only the position of the integration points changes to that of the nodes.

Note: References

- This answer on Computational Science StackExchange.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search