Tutorial

Nicolai Schoch, Fabian Kißler

Elasticity Tutorial for Soft Tissue Simulation

modified on November 16, 2017

Version 1.5

http://www.hiflow3.org/

Contents

Applying HiFlow^3 for solving an Elasticity Problem for Soft Tissue Simulation

1 Introduction

HiFlow^3 is a multi-purpose finite element software providing powerful tools for efficient and accurate solution to a wide range of problems modeled by partial differential equations (PDEs). Based on object-oriented concepts and the full capabilities of C++ the HiFlow^3 project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules dealing with the mesh setup, finite element spaces, degrees of freedom, linear algebra routines, numerical solvers, and output data for visualization. Parallelism - as the basis for high performance simulations on modern computing systems - is introduced on two levels: coarse-grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends.

1.1 How to Use the Tutorial?

You find the example code (elasticity_tutorial.cc, elasticity_tutorial.h) and a parameter file for the first numerical example (elasticity_tutorial.xml) in the folder `/hiflow/examples/elasticity`. The geometry data (stanford_bunny.inp) is stored in the folder
`/hiflow/examples/elasticity/SimInput`.

1.1.1 Using HiFlow^3 as a Developer

First build and compile HiFlow^3. Go to the directory `/build/example/elasticity`, where the binary `elasticity_tutorial` is stored. Type

```
./elasticity_tutorial /"path_to"/elasticity_tutorial.xml
```

to execute the program in sequential mode. To execute in parallel mode with `X` processes, type

```
mpirun -np X elasticity_tutorial /"path_to"/elasticity_tutorial.xml
```

In both cases, you need to make sure that the default parameter file `elasticity_tutorial.xml` is stored in the same directory as the binary, and that the geometry data specified in the parameter file is stored in `/hiflow/examples/data`. Alternatively, you can specify the path of your own xml-file, i.e.

```
./elasticity_tutorial /"path_to_parameter_file"/"name_of_parameter_file".xml
```

2 Mathematical Setup

2.1 Problem

We want to investigate the deformation of an elastic solid subject to external forces. In the following, the solid will be represented by a bounded subset $\Omega \subset \mathbb{R}^3$. The Cauchy stress tensor and the infinitesimal strain tensor will be denoted by σ and ϵ, respectively. Given any point $p \in \Omega$

and any surface passing through p, let n denote a unit normal vector to this surface at p. Then, the stress vector at p w.r.t. n is given by

$$T^{(n)} = n\sigma. \tag{1}$$

According to this formula, the computation of the stress vector does not depend on the choice of the surface passing through p. If we apply pressure to the surface of the solid, that is, given any surface force density $g = n\sigma^{\partial}$, we have

$$g = n\sigma \tag{2}$$

on the boundary $\partial\Omega$ with unit normal vector field n. Generally, pressure will only act on a certain subset of $\partial\Omega$, the effective Neumann boundary $\Gamma_{\mathsf{N}}^{\neq 0}$. Furthermore, there is a Dirichlet boundary Γ_{D} with $\Gamma_{\mathsf{D}} \cap \Gamma_{\mathsf{N}}^{\neq 0} = \emptyset$. Let u be the displacement vector field associated to the deformation, that is, for any $p \in \Omega$,

$$p' = p + u(p) \tag{3}$$

represents the position of the particle p after the deformation. By u^{∂} we denote a given displacement on the Dirichlet boundary. We define the Neumann boundary $\Gamma_{\mathsf{N}} := \partial\Omega \setminus \Gamma_{\mathsf{D}}$ with $g = 0$ on $\Gamma_{\mathsf{N}} \setminus \Gamma_{\mathsf{N}}^{\neq 0}$ (for illustration, see figure 1).

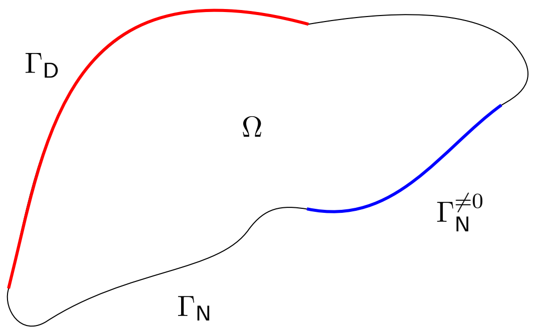

Figure 1: Partition of the boundary of Ω: The red line represents the Dirichlet boundary. The remaining part is the Neumann boundary with the effective Neumann boundary, represented by the blue line, as a subset.

The infinitesimal strain tensor ϵ relates the deformation gradient ∇u to the deformation u of Ω. If we assume that the displacement gradients are small compared to the diameter of Ω, we have

$$\epsilon(u) = \frac{1}{2}\left(\nabla u + \nabla u^{T}\right). \tag{4}$$

Note that the infinitesimal strain tensor is the linearized version of the finite strain tensor. Furthermore, we require Hooke's Law to hold, or, to put it in another way, we want that there is a linear relation between the Cauchy stress tensor and the infinitesimal strain tensor. If C denotes the fourth-order stiffness tensor, using Einstein summation, we get

$$\sigma_{ij} = C_{ijkl}\epsilon_{kl}. \tag{5}$$

When using Voigt notation, we obtain

$$\sigma = \lambda\,\mathrm{tr}(\epsilon)I + 2\mu\epsilon, \tag{6}$$

for homogeneous and isotropic materials, where λ and μ are Lamé's parameters. Note that other types of engineering constants such as Poisson's ratio ν and Young's modulus E can be expressed in terms of Lamé's parameters and vice versa:

$$\nu = \frac{\lambda}{2(\lambda+\mu)} \qquad E = \frac{\mu(3\lambda+2\mu)}{\lambda+\mu}$$
$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \qquad \mu = \frac{E}{2(1+\nu)}$$

Enforcing physical reality, we require $\mu, \lambda > 0$. In terms of E and ν we have $E > 0$ and $0 < \nu < 0.5$. For more details we refer to [1] or [2]. As mentioned before, deformation of the solid can be caused by forces acting on its surface. However, we also want to consider forces acting on all the particles, e.g. gravity, which are represented by the body force density f. Assuming that the body Ω is in a state of equilibrium, all forces and momentums acting on the surface and volume of an arbitrary subset $V \subseteq \Omega$ add up to zero:

$$0 = \underbrace{\int_V f(x)\,\mathrm{d}x}_{\text{body force}} + \underbrace{\int_{\partial V} \sigma(x) n \,\mathrm{d}S}_{\text{surface force}}. \tag{7}$$

Using the divergence theorem, we get

$$0 = \int_V f(x) + \operatorname{div}(\sigma(x))\,\mathrm{d}x, \tag{8}$$

which, since V is an arbitrary subset and f and $\operatorname{div}\sigma$ are supposed to be continuous functions, yields the differential equation

$$-\operatorname{div}\sigma = f. \tag{9}$$

A detailed derivation of equation (9) can be found in [1]. Furthermore, we require the stiffness tensor to be symmetric, positive definite on symmetric dyads and sufficiently smooth:

$$C_{ijkl} \in C^2(\overline{\Omega}), \quad C_{ijkl} = C_{klij}, \quad C_{ijkl}\epsilon_{ij}\epsilon_{kl} > 0, \text{ if } \epsilon \neq 0. \tag{10}$$

All things considered, assuming that f, σ^∂ and u^∂ are sufficiently smooth, we need to find functions u, ϵ and σ, whose components need to satisfy the following regularity conditions

$$u_i \in C(\Omega \cup \Gamma_{\mathsf{D}}) \cap C^1(\Omega), \quad \epsilon_{ij} \in C(\Omega) \cap L^2(\Omega), \quad \sigma_{ij} \in C(\Omega \cup \Gamma_{\mathsf{N}}) \cap C^1(\Omega) \cap L^2(\Omega) \tag{11}$$

and the equilibrium equations of linear elasticity:

$$\boxed{\begin{aligned} -\operatorname{div}\sigma &= f && \text{on } \Omega, && (12)\\ u &= u^\partial && \text{on } \Gamma_{\mathsf{D}}, && (13)\\ n\sigma &= g && \text{on } \Gamma_{\mathsf{N}}, && (14)\\ \frac{1}{2}\left(\nabla u + \nabla u^T\right) &= \epsilon && \text{on } \Omega, && (15)\\ \sigma &= C\epsilon && \text{on } \Omega. && (16)\end{aligned}}$$

Since the existence of a classical solution to this problem cannot be guaranteed, we will derive the so-called weak formulation of the problem in the succeeding section. However, the following theorem by Cosserat states that the difference of two solutions describes the motion of a rigid body and gives a uniqueness criterion based on the geometry of Ω.

Cosserat's Theorem. *Let $u^{(1)}$ and $u^{(2)}$ denote two classical solutions to the foregoing system of partial differential equations. Then there is a vector $b \in \mathbb{R}^3$ and a skew symmetric matrix $B \in \mathbb{R}^{3\times 3}$ such that*

$$u^{(1)}(x) - u^{(2)}(x) = b + Bx \tag{17}$$

for all $x \in \Omega$. Additionally, if Γ_D contains three linearly independent vectors, we have $b = 0$ and $B = 0$ and the solution to the equilibrium equations of linear elasticity is unique.

2.2 Weak Formulation

We begin by introducing some notation and basic definitions. Let $L^2(\Omega)$ denote the set of square integrable functions on Ω with the following inner product and norm:

$$(f, g)_\Omega = \int_\Omega fg \, \mathrm{d}x, \quad \|f\|_\Omega = \sqrt{\int_\Omega |f|^2 \, \mathrm{d}x}. \tag{18}$$

The Sobolev space $H^1(\Omega)$ is the completion of $C^\infty(\overline{\Omega})$ with respect to the norm given by

$$\|f\|_{1;\Omega} = \sqrt{\|f\|_\Omega^2 + \int_\Omega |\nabla f|^2 \, \mathrm{d}x}, \tag{19}$$

and let $H_0^1(\Gamma_\mathsf{D}; \Omega)$ be the closure of the subset

$$\{f \in C^\infty(\overline{\Omega}) : f = 0 \text{ on } \Gamma_\mathsf{D}\} \subseteq H^1(\Omega). \tag{20}$$

Moreover, in order to ensure that the following propositions and declarations are well-defined, we need the statements (A1) to (A4) to hold:

(A1) The subset $\Omega \subset \mathbb{R}^3$ satisfies the strong cone condition.

(A2) The measure of $\Gamma_\mathsf{D} \subset \partial\Omega$ is positive and its relative interior contains three linearly independent vectors.

(A3) On Ω, the stiffness tensor $C = (C_{ijkl})$ is symmetric and uniformly bounded,

$$C_{ijkl} = C_{klij} \in L^\infty(\Omega), \tag{21}$$

as well as positive definite, i.e.,

$$C_{ijkl}\epsilon_{ij}\epsilon_{kl} > 0 \tag{22}$$

for all symmetric tensors $\epsilon \neq 0$.

(A4) The external forces acting on the solid and the displacement of the boundary are given by

$$f \in L^2(\Omega)^3, \quad \sigma^\partial \in H^1(\partial\Omega)^{3\times3}_{\mathsf{sym}}, \quad u^\partial \in H^1(\Omega)^3. \tag{23}$$

Now, we can derive the weak formulation of the equilibrium equations of linear elasticity. Using previous notation, we multiply equation (12) by an arbitrary test function $v \in C^\infty(\Omega)^3$ and integrate over Ω. This yields

$$-\int_\Omega \operatorname{div}(\sigma(u)) v \, \mathrm{d}V = \int_\Omega f v \, \mathrm{d}V. \tag{24}$$

Integrating the left hand side by parts gives

$$-\int_{\partial\Omega} \sigma(u) v \, \mathrm{d}\vec{S} + \int_\Omega \sigma_{ij} \partial_i v_j \, \mathrm{d}V = \int_\Omega f v \, \mathrm{d}V. \tag{25}$$

Note that the Cauchy stress tensor $\sigma(u)$, given by equation (6), is symmetric. Hence, we can rewrite the second term on the left hand side of equation (25):

$$\begin{aligned} \int_\Omega \sigma_{ij} \partial_i v_j \, \mathrm{d}V &= \frac{1}{2} \int_\Omega \sigma_{ij} \partial_i v_j + \sigma_{ji} \partial_i v_j \, \mathrm{d}V && (26) \\ &= \int_\Omega \sigma_{ij} \left(\frac{1}{2} \left(\partial_i v_j + \partial_j v_i \right) \right) \mathrm{d}V && (27) \\ &= \int_\Omega \sigma(u) : \epsilon(v) \, \mathrm{d}V, && (28) \end{aligned}$$

which denotes the Frobenius inner product, where $A : B = A_{ij} B_{ij}$. We hence obtain

$$\int_\Omega \sigma(u) : \epsilon(v) \, \mathrm{d}V = \int_\Omega f v \, \mathrm{d}V + \int_{\partial\Omega} \sigma(u) v \, \mathrm{d}\vec{S}. \tag{29}$$

We can rewrite the second summand on the right-hand side in terms of the surface force density g by using the fact that $\partial\Omega$ is the disjoint union of Γ_D and Γ_N:

$$\begin{aligned} \int_{\partial\Omega} \sigma(u) v \, \mathrm{d}\vec{S} &= \int_{\partial\Omega} n\sigma(u) v \, \mathrm{d}S && (30) \\ &= \int_{\Gamma_\mathsf{D}} n\sigma(u) v \, \mathrm{d}S + \int_{\Gamma_\mathsf{N}} g v \, \mathrm{d}S. && (31) \end{aligned}$$

All things considered, we get

$$\int_\Omega \sigma(u) : \epsilon(v) \, \mathrm{d}V = \int_\Omega f v \, \mathrm{d}V + \int_{\Gamma_\mathsf{D}} n\sigma(u) v \, \mathrm{d}S + \int_{\Gamma_\mathsf{N}} g v \, \mathrm{d}S. \tag{32}$$

In order to shorten notation, we define the bilinear form a and the linear form l:

$$a(u,v) := \int_\Omega \sigma(u) : \epsilon(v) \, \mathrm{d}V, \quad l(v) := \int_\Omega f v \, \mathrm{d}V + \int_{\Gamma_\mathsf{N}} g v \, \mathrm{d}S \tag{33}$$

Note that in the definition of l we omitted the second term on the right-hand side of equation (32), as if we considered homogeneous Dirichlet boundary conditions, i.e. $u^\partial = 0$ on Γ_D. We will correct this error by requiring the difference $u - u^\partial$ of the solution u and the given translation vector field u^∂ to vanish on the boundary Γ_D, i.e. $u - u^\partial \in H^1_0(\Gamma_\mathsf{D}; \Omega)^3$. You will find the same method being used in the implementation below, where the Dirichlet boundary conditions are considered separately in the assembly of the system, see 3.5.5. The weak formulation of the equilibrium equations of linear elasticity now reads as follows:

$$\boxed{\begin{gathered} \text{Find } u \in H^1(\Omega)^3 \text{ such that } u - u^\partial \in H^1_0(\Gamma_\mathsf{D}; \Omega)^3 \text{ and} \\ a(u,\varphi) = l(\varphi) \quad \forall \varphi \in H^1_0(\Gamma_\mathsf{D}; \Omega)^3. \end{gathered}} \tag{34}$$

As stated in Theorem 1.10 in [2], assuming (A1) to (A4) hold, there is a unique weak solution to the equilibrium equations of linear elasticity. According to the derivation of the weak formulation of the problem, every classical solution is also a weak solution and vice versa, if the weak solution satisfies the necessary regularity conditions.
One can directly recognize the terms of the above weak formulation in the implementation, see sections 3.5 and 3.6. Please note that our above derivation represents the stationary consideration of an elastic deformation process. In order to describe an instationary simulation, we would need to consider additional terms accounting for inertia and damping. This results in a more complicated assembly, and can best be solved using e.g. the Newmark time integration scheme, which guarantees for stable and efficient simulation results, see [3].

3 The Commented Program

3.1 Preliminaries

HiFlow^3 is designed for high performance computing on massively parallel machines. So it applies the Message Passing Interface (MPI) library specification for message-passing, see sections 3.4 and 3.5.1, as well as [4] and [5] . The elasticity tutorial needs the following two input files:

- A parameter file: The parameter file is an xml-file, which contains all parameters needed to execute the program. It is read in by the program. It is not necessary to recompile the program, when parameters in the xml-file are changed. By default the elasticity tutorial reads in the parameter file `elasticity_tutorial.xml`, see section 3.2, which contains the parameters of the included example, see section 5. This file is stored in `/hiflow/examples/elasticity/`.
- Geometry data: The file containing the geometry is specified in the parameter file. In the example in section 5 we used `stanford_bunny.inp`.

HiFlow^3 does not generate meshes for the domain Ω. Meshes in *.inp and *.vtu format can be read in. It is possible to extend the reader for other formats. Furthermore, it is possible to generate other geometries by using external programs (Mesh generators) or by hand.

3.2 Parameter File

The needed parameters are initialized in the parameter file `elasticity_tutorial.xml`.

```
<Param>
  <OutputPathAndPrefix>elasticity_test</OutputPathAndPrefix>
  <Mesh>
    <Filename>stanford_bunny.inp</Filename>
    <InitialRefLevel>0</InitialRefLevel>
  </Mesh>

  <LinearAlgebra>
    <Platform>CPU</Platform>
    <Implementation>Naive</Implementation>
    <MatrixFormat>CSR</MatrixFormat>
  </LinearAlgebra>

  <ElasticityModel>
    <density>1070</density>
    <lambda>259259</lambda> <!-- 259259(PR=.35), 49329(PR=.49) -->
    <mu>111111</mu> <!-- 111111(PR=.35), 1007(PR=.49) -->
    <gravity>.0</gravity>
  </ElasticityModel>

  <QuadratureOrder>2</QuadratureOrder>

  <FiniteElements>
    <DisplacementDegree>1</DisplacementDegree>
  </FiniteElements>

  <Instationary>
    <SolveInstationary>0</SolveInstationary> <!-- boolean 0 or 1 -->
    <DampingFactor>1.0</DampingFactor> <!-- should remain 1.0 -->
    <RayleighAlpha>0.1</RayleighAlpha> <!-- MassFactor -->
    <RayleighBeta>0.2</RayleighBeta> <!-- StiffnessFactor -->
    <Method>Newmark</Method> <!-- ImplicitEuler, CrankNicolson, ExplicitEuler, Newmark, ...
        -->
    <DeltaT>0.1</DeltaT> <!-- smaller: 0.05 -->
    <MaxTimeStepIts>20</MaxTimeStepIts> <!-- higher: 20 -->
  </Instationary>

  <Boundary>
    <DirichletMaterial1>110</DirichletMaterial1> <!-- fixed boundary -->
    <DirichletMaterial2>112</DirichletMaterial2> <!-- displaced boundary -->
    <DirichletMaterial3>114</DirichletMaterial3> <!-- displaced boundary -->
  </Boundary>

  <LinearSolver>
    <SolverName>CG</SolverName> <!-- CG (+ SGAUSS_SEIDEL etc) or GMRES (+ ILU2) or ... -->
    <MaximumIterations>1000</MaximumIterations>
    <AbsoluteTolerance>1.e-8</AbsoluteTolerance>
    <RelativeTolerance>1.e-20</RelativeTolerance>
    <DivergenceLimit>1.e6</DivergenceLimit>
    <BasisSize>1000</BasisSize>
    <Preconditioning>1</Preconditioning> <!-- boolean 0 or 1 -->
    <PreconditionerName>SGAUSS_SEIDEL</PreconditionerName> <!-- NOPRECOND = 0, JACOBI = 1,
        GAUSS_SEIDEL = 2, SGAUSS_SEIDEL = 3, SOR, SSOR, ILU, ILU2, ILU_P, ILUpp -->
    <Omega>2.5</Omega>
    <ILU_p>2.5</ILU_p>
  </LinearSolver>
  <ILUPP> <!-- use GaussSeidel / Jacobi / SSOR instead -->
    <PreprocessingType>0</PreprocessingType>
    <PreconditionerNumber>11</PreconditionerNumber>
    <MaxMultilevels>20</MaxMultilevels>
    <MemFactor>0.8</MemFactor>
    <PivotThreshold>2.75</PivotThreshold>
    <MinPivot>0.05</MinPivot>
  </ILUPP>

  <Backup>
    <Restore>0</Restore>
    <LastTimeStep>160</LastTimeStep>
    <Filename>backup.h5</Filename>
  </Backup>
</Param>
```

Since, for reasons of simplicity and better understandability, we do not consider effective (i.e. non-zero) Neumann force or pressure boundary conditions in the tutorial, this part is not included in the sample parameter file either. However, we want to shortly outline the integration of Neumann boundary conditions here: First of all, one has to declare an effective Neumann boundary $\Gamma_{\mathsf{N}}^{\neq 0}$. This can be done by simply including an additional line <NeumannMaterialX> in the <Boundary> part of the parameter file. An additional Neumann boundary assembler class similar to the class StationaryElasticityAssembler in 3.6.2 must loop over all these Neumann boundary elements and compute the effective force or pressure. Creating an object of the Neumann boundary class in 3.5.5 then allows for integrating it according to the weak formulation.

3.3 Structure of the Elasticity Tutorial

Following member functions are defined in the class Elasticity. The survey reflects the access relations between the functions, classes and structures.

- run()
  - setup_linear_algebra()
  - read_mesh()
  - prepare()
    - prepare_bc_stationary()
      - StationaryElasticityAssembler_DirichletBC_3D
  - assemble_system()
      - StationaryElasticityAssembler
  - solve_system()
  - visualize()

You can find the source code of every member function in an extra section below.

3.4 Main Function

The main function starts the simulation of the elasticity problem (elasticity_tutorial.cc).

```cpp
int main ( int argc, char** argv )
{
    MPI_Init ( &argc, &argv );

    std::string param_filename ( PARAM_FILENAME );
    if ( argc > 1 )
    {
        param_filename = std::string ( argv[1] );
    }

    try
    {
        Elasticity app ( param_filename );
        app.run ( );
    }
    catch ( std::exception& e )
    {
        std::cerr << "\nProgram ended with uncaught exception.\n";
        std::cerr << e.what ( ) << "\n";
        return -1;
    }
    MPI_Finalize ( );
    return 0;
}
```

3.5 Member Functions

3.5.1 run()

The member function `run()` is defined in the class `Elasticity` (`elasticity_tutorial.cc`).

```
    virtual void run ( )
    {
        simul_name_ = params_["OutputPathAndPrefix"].get<std::string>( );
        u_deg = params_["FiniteElements"]["DisplacementDegree"].get<int>( );

        MPI_Comm_rank ( comm_, &rank_ );
        MPI_Comm_size ( comm_, &num_partitions_ );

        // Turn off INFO log except on master proc.
        if ( rank_ != MASTER_RANK )
        {
            INFO = false;
            CONSOLE_OUTPUT_ACTIVE = false;
        }

        std::ofstream info_log ( ( simul_name_ + "_info_log" ).c_str ( ) );
        LogKeeper::get_log ( "info" ).set_target ( &info_log );
        std::ofstream debug_log ( ( simul_name_ + "_debug_log" ).c_str ( ) );
        LogKeeper::get_log ( "debug" ).set_target ( &debug_log );

        CONSOLE_OUTPUT ( 0, "============================================================" );
        CONSOLE_OUTPUT ( 0, "==== Elasticity Simulation ===" );
        CONSOLE_OUTPUT ( 0, "==== built using HiFlow3. ===" );
        CONSOLE_OUTPUT ( 0, "==== ===" );
        CONSOLE_OUTPUT ( 0, "==== Engineering Mathematics and Computing Lab (EMCL) ===" );
        CONSOLE_OUTPUT ( 0, "============================================================" );
        CONSOLE_OUTPUT ( 0, "" );

        // output parameters for debugging
        LOG_INFO ( "parameters", params_ );

        // setup timing report
        TimingScope::set_report ( &time_report_ );

        {
            TimingScope tscope ( "Setup" );

            setup_linear_algebra ( );

            read_mesh ( );

            // The simulation has two modes: stationary and
            // instationary. Which one is used, depends on the parameter
            // Instationary.SolveInstationary. In stationary mode, solve()
            // is only called once, whereas in instationary mode, it is
            // called several times via the time-stepping method implemented in run_time_loop
                () .
            solve_instationary_ = params_["Instationary"]["SolveInstationary"].get<bool>( );
        prepare ( );

    }

    if ( solve_instationary_ )
    {
        TimingScope tscope ( "Complete_instationary_simulation_loop" );
        // See private source code by Nicolai Schoch.

    }
    else
    {
        TimingScope tscope ( "Complete_stationary_simulation_loop" );

        LOG_INFO ( "simulation", "Solving stationary problem" );

        assemble_system ( );

        solve_system ( );

        visualize ( );
    }

    CONSOLE_OUTPUT ( 0, "" );

    if ( rank_ == MASTER_RANK )
    {
        // Output time report
        TimingReportOutputVisitor visitor ( std::cout );
        time_report_.traverse_depth_first ( visitor );
    }

    LogKeeper::get_log ( "info" ).flush ( );
    LogKeeper::get_log ( "debug" ).flush ( );
    LogKeeper::get_log ( "info" ).set_target ( 0 );
    LogKeeper::get_log ( "debug" ).set_target ( 0 );

    CONSOLE_OUTPUT ( 0, "============================================================\n" )
        ;
}
```

3.5.2 setup_linear_algebra()

The method `setup_linear_algebra()` reads in the needed parameters `Platform`, `Implementation`, `MatrixFormat` and `PreconditionerName` from the parameter file 3.2.

```
void Elasticity::setup_linear_algebra ( )
{

    TimingScope tscope ( "setup_linear_algebra" );

    const std::string platform_str = params_["LinearAlgebra"]["Platform"].get<std::string>( )
        ;
    if ( platform_str == "CPU" )
    {
        la_sys_.Platform = CPU;
    }
```

```
else if ( platform_str == "GPU" )
{
    la_sys_.Platform = GPU;
}
else
{
    throw UnexpectedParameterValue ( "LinearAlgebra.Platform", platform_str );
}
init_platform ( la_sys_ );

const std::string impl_str = params_["LinearAlgebra"]["Implementation"].get<std::string>(
      );
if ( impl_str == "Naive" )
{
    la_impl_ = NAIVE;
}
else if ( impl_str == "BLAS" )
{
    la_impl_ = BLAS;
}
else if ( impl_str == "MKL" )
{
    la_impl_ = MKL;
}
else if ( impl_str == "OPENMP" )
{
    la_impl_ = OPENMP;
}
else if ( impl_str == "SCALAR" )
{
    la_impl_ = SCALAR;
}
else if ( impl_str == "SCALAR_TEX" )
{
    la_impl_ = SCALAR_TEX;
}
else
{
    throw UnexpectedParameterValue ( "LinearAlgebra.Implementation", impl_str );
}

const std::string matrix_str = params_["LinearAlgebra"]["MatrixFormat"].get<std::string>(
      );
if ( matrix_str == "CSR" )
{
    la_matrix_format_ = CSR;
}
else if ( matrix_str == "COO" )
{
    la_matrix_format_ = COO;
}
else
{
    throw UnexpectedParameterValue ( "LinearAlgebra.MatrixFormat", impl_str );
}

// the following part is needed to initialize class member "matrix_precond_".
const std::string precond_str = params_["LinearSolver"]["PreconditionerName"].get<std::
     string>( );
    if ( precond_str == "NOPRECOND" )
    {
        matrix_precond_ = NOPRECOND;
    }
    else if ( precond_str == "JACOBI" )
    {
        matrix_precond_ = JACOBI;
    }
    else if ( precond_str == "GAUSS_SEIDEL" )
    {
        matrix_precond_ = GAUSS_SEIDEL;
    }
    else if ( precond_str == "SGAUSS_SEIDEL" )
    {
        matrix_precond_ = SGAUSS_SEIDEL;
    }
    else if ( precond_str == "SOR" )
    {
        matrix_precond_ = SOR;
    }
    else if ( precond_str == "SSOR" )
    {
        matrix_precond_ = SSOR;
    }
    else if ( precond_str == "ILU" )
    {
        matrix_precond_ = ILU;
    }
    else if ( precond_str == "ILU2" )
    {
        matrix_precond_ = ILU2; // ILU2 = ILUpp;
    }
    else
    {
        throw UnexpectedParameterValue ( "LinearSolver.PreconditionerName", precond_str );
    }
}
```

3.5.3 read_mesh()

The member function `read_mesh()` reads in the initial mesh, partitions, distributes and writes out the mesh.

```
void Elasticity::read_mesh ( )
{
    TimingScope tscope ( "read_mesh" );

    MeshPtr master_mesh;
    MeshPtr local_mesh;

    // get mesh_filename:
    const std::string mesh_name = params_["Mesh"]["Filename"].get<std::string>( );
    std::string mesh_filename = std::string ( DATADIR ) + mesh_name;

    // find position of last '.' in mesh_filename:
    int dot_pos = mesh_filename.find_last_of ( "." );
    assert ( dot_pos != std::string::npos );
    // get suffix of mesh_filename:
    std::string suffix = mesh_filename.substr ( dot_pos );

    // switch cases according to the suffix of mesh_filename --> pvtu or vtu/inp:
    // this tutorial allows for ".vtu" or ".inp" only.

    if ( rank ( ) == MASTER_RANK )
    {

        master_mesh = read_mesh_from_file ( mesh_filename, DIMENSION, DIMENSION, &comm_ );
        CONSOLE_OUTPUT ( 1, "Read mesh with " << master_mesh->num_entities ( DIMENSION ) << "
             cells." );

        if ( suffix == std::string ( ".vtu" ) )
        {
            // in case the input mesh does not specify material_IDs for the boundary facets,
            // the function set_default_material_number_on_bdy(MeshPtr*, int) sets default
                bdy_facet_IDs
            // in order for (among others) the GridGeometricSearch to work properly
            // especially w.r.t. find_closest_point() -> see also: documentation.
            set_default_material_number_on_bdy ( master_mesh, 1 );
        }
        refinement_level_ = 0;
        const int initial_ref_lvl = params_["Mesh"]["InitialRefLevel"].get<int>( );

        for ( int r = 0; r < initial_ref_lvl; ++r )
        {
            master_mesh = master_mesh->refine ( );
            ++refinement_level_;
        }
        LOG_INFO ( "mesh", "Initial refinement level = " << refinement_level_ );
        CONSOLE_OUTPUT ( 1, "Refined mesh (level " << refinement_level_ << ") has "
                         << master_mesh->num_entities ( DIMENSION ) << " cells." );

    }

    if ( num_partitions ( ) > 1 )
    {
#ifdef WITH_METIS
        MetisGraphPartitioner partitioner;
#else
        NaiveGraphPartitioner partitioner;
#endif

        const GraphPartitioner* p = &partitioner;

        local_mesh = partition_and_distribute ( master_mesh, MASTER_RANK, comm_, p );
    }
    else
    {
        NaiveGraphPartitioner partitioner;
        local_mesh = partition_and_distribute ( master_mesh, MASTER_RANK, comm_, &partitioner
            );
    }
    assert ( local_mesh != 0 );

    SharedVertexTable shared_verts;
    mesh_ = compute_ghost_cells ( *local_mesh, comm_, shared_verts );
    std::ostringstream rank_str;
    rank_str << rank ( );

    PVtkWriter writer ( comm_ );
    std::string output_file = simul_name_ + std::string ( "_initial_mesh_local.pvtu" );
    writer.add_all_attributes ( *mesh_, true );
    writer.write ( output_file.c_str ( ), *mesh_ );
}
```

3.5.4 prepare()

The member function `prepare()` reads in parameters regarding the physical properties of the body like `density`, `mu`, `lambda` and `gravity`, see the remark about Lamé's parameters in section 2.1 for details. Furthermore, we initialize the FE space, the system matrix, the solution and right-hand side vector and execute the setup of the linear solver and preconditioner. Since we solve the stationary elasticity problem, the function calls `prepare_bc_stationary()`, see 3.5.4.1.

```
void Elasticity::prepare ( )
{
    TimingScope tscope ( "prepare" );

    // prepare modelling problem parameters
    rho_ = params_["ElasticityModel"]["density"].get<double>( );
    mu_ = params_["ElasticityModel"]["mu"].get<double>( );
    lambda_ = params_["ElasticityModel"]["lambda"].get<double>( );
    gravity_ = params_["ElasticityModel"]["gravity"].get<double>( );

    // prepare space
    std::vector< int > degrees ( DIMENSION );
    for ( int c = 0; c < DIMENSION; ++c )
    {
        degrees.at ( c ) = u_deg;
    }

    // Initialize the VectorSpace object
    space_.Init ( degrees, *mesh_ );

    CONSOLE_OUTPUT ( 1, "Total number of dofs = " << space_.dof ( ).ndofs_global ( ) );

    for ( int p = 0; p < num_partitions ( ); ++p )
    {
        CONSOLE_OUTPUT ( 2, "Num dofs on process " << p << " = " << space_.dof ( ).ndofs_on_sd
             ( p ) );
    }

    // prepare visualization structures
    visu_names_.push_back ( "u1" );
    visu_names_.push_back ( "u2" );
    visu_names_.push_back ( "u3" );

    // Setup couplings object and prepare linear algebra structures
    couplings_.Clear ( );
    couplings_.Init ( communicator ( ), space_.dof ( ) );

    // prepare global assembler
    QuadratureSelection q_sel ( params_["QuadratureOrder"].get<int>( ) );
    global_asm_.set_quadrature_selection_function ( q_sel );
```

```
// compute matrix graph
SparsityStructure sparsity;
global_asm_.compute_sparsity_structure ( space_, sparsity );

couplings_.InitializeCouplings ( sparsity.off_diagonal_rows, sparsity.off_diagonal_cols )
    ;

// Initialize system matrix (K_eff), solution vector and rhs vector (R_eff) [and possibly
     nbc vector].
matrix_.Init ( communicator ( ), couplings_, la_platform ( ), la_implementation ( ),
    la_matrix_format ( ) ); // System (Stiffness) Matrix.
sol_.Init ( communicator ( ), couplings_, la_platform ( ), la_implementation ( ) ); //
    Solution Vector.
rhs_.Init ( communicator ( ), couplings_, la_platform ( ), la_implementation ( ) ); //
    RHS Vector.

matrix_.InitStructure ( vec2ptr ( sparsity.diagonal_rows ),
                        vec2ptr ( sparsity.diagonal_cols ),
                        sparsity.diagonal_rows.size ( ),
                        vec2ptr ( sparsity.off_diagonal_rows ),
                        vec2ptr ( sparsity.off_diagonal_cols ),
                        sparsity.off_diagonal_rows.size ( ) );
matrix_.Zeros ( );

sol_.InitStructure ( );
sol_.Zeros ( );

rhs_.InitStructure ( );
rhs_.Zeros ( );

// setup linear solver parameters
solver_name_ = params_["LinearSolver"]["SolverName"].get<std::string>( );
lin_max_iter = params_["LinearSolver"]["MaximumIterations"].get<int>( );
lin_abs_tol = params_["LinearSolver"]["AbsoluteTolerance"].get<double>( );
lin_rel_tol = params_["LinearSolver"]["RelativeTolerance"].get<double>( );
lin_div_tol = params_["LinearSolver"]["DivergenceLimit"].get<double>( );
basis_size = params_["LinearSolver"]["BasisSize"].get<int>( );

// Setup Solver and Preconditioner
    ////////////////////////////////////////////////////////
if ( solver_name_ == "CG" )
{

    // Setup CG Preconditioner /////////////////////////////////////////
    if ( matrix_precond_ != NOPRECOND )
    { // With Preconditioning.

        if ( matrix_precond_ == JACOBI )
        {
            preconditioner_.Init_Jacobi ( rhs_ );
        }
        if ( matrix_precond_ == GAUSS_SEIDEL )
        {
            preconditioner_.Init_GaussSeidel ( );
        }
        if ( matrix_precond_ == SGAUSS_SEIDEL )
        {
            preconditioner_.Init_SymmetricGaussSeidel ( );
            }
            if ( matrix_precond_ == SOR )
            {
                omega_ = params_["LinearSolver"]["Omega"].get<double>( );
                preconditioner_.Init_SOR ( omega_ );
            }
            if ( matrix_precond_ == SSOR )
            {
                omega_ = params_["LinearSolver"]["Omega"].get<double>( );
                preconditioner_.Init_SSOR ( omega_ );
            }
            if ( matrix_precond_ == ILU )
            {
                std::cout << "By definition: CG is not supposed to pair with ILU as solver-
                    preconditioner-combination." << std::endl;
            }
            if ( matrix_precond_ == ILU2 )
            { // Note that: ILU2 = ILUpp; handled for GMRES only.
                std::cout << "By definition: CG is not supposed to pair with ILU2 (a.k.a.
                    ILUpp) as solver-preconditioner-combination." << std::endl;
            }

            cg_.InitParameter ( "Preconditioning" );
            cg_.SetupPreconditioner ( preconditioner_ );

        }
        else
        { // Without Preconditioning.

            cg_.InitParameter ( "NoPreconditioning" );

        }

        // Setup CG Solver ///////////////////////////////////////////////////
        cg_.InitControl ( lin_max_iter, lin_abs_tol, lin_rel_tol, lin_div_tol );
        cg_.SetupOperator ( matrix_ );

    } // end of CG Setup.
    else /*if (solver_name_ == "GMRES")*/
    { // if not CG then GMRES by default.

        // Setup GMRES Preconditioner //////////////////////////////////////////
        use_ilupp_ = 0; // default.
        if ( matrix_precond_ == ILU2 )
        { // Note that: ILU2 = ILU_pp.
            use_ilupp_ = 1; // otherwise: no preconditioner for GMRES.
        }
#ifdef WITH_ILUPP
        if ( use_ilupp_ )
        {
            ilupp_.InitParameter ( params_["ILUPP"]["PreprocessingType"].get<int>( ),
                                   params_["ILUPP"]["PreconditionerNumber"].get<int>( ),
                                   params_["ILUPP"]["MaxMultilevels"].get<int>( ),
                                   params_["ILUPP"]["MemFactor"].get<double>( ),
                                   params_["ILUPP"]["PivotThreshold"].get<double>( ),
                                   params_["ILUPP"]["MinPivot"].get<double>( ) );

            gmres_.SetupPreconditioner ( ilupp_ );
            gmres_.InitParameter ( basis_size, "RightPreconditioning" );
        }
        else
        {
            gmres_.InitParameter ( basis_size, "NoPreconditioning" );
        }
#else
        gmres_.InitParameter ( basis_size, "NoPreconditioning" );
#endif

        // Setup GMRES Solver ////////////////////////////////////////////////////
        gmres_.InitControl ( lin_max_iter, lin_abs_tol, lin_rel_tol, lin_div_tol );
        gmres_.SetupOperator ( matrix_ );

    } // end of GMRES Setup.

    // prepare dirichlet BC
    if ( !solve_instationary_ )
    {
        prepare_bc_stationary ( );
    }

}
```

3.5.4.1 prepare_bc_stationary()

The Boundary parameters in the parameter file determine whether a point will be shifted. We create a StationaryElasticityAssembler_DirichletBC_3D object, see section 3.6.1.

```
void Elasticity::prepare_bc_stationary ( )
{
    TimingScope tscope ( "prepare_bc_stationary" );

    dirichlet_dofs_.clear ( );
    dirichlet_values_.clear ( );

    // >---------------------------------
    // Set DirichletBCs for prescribed facets (i.e. material_IDs given in hiflow3_scene.xml-
        File):
    const int dir_bdy1 = params_["Boundary"]["DirichletMaterial1"].get<int>( );
    const int dir_bdy2 = params_["Boundary"]["DirichletMaterial2"].get<int>( );
    const int dir_bdy3 = params_["Boundary"]["DirichletMaterial3"].get<int>( );

    // create InstationaryElasticity_DirichletBC_3D-Object.
    StationaryElasticity_DirichletBC_3D bc[3] = { StationaryElasticity_DirichletBC_3D ( 0,
        dir_bdy1, dir_bdy2, dir_bdy3 ),
                                              StationaryElasticity_DirichletBC_3D ( 1, dir_bdy1
                                                  , dir_bdy2, dir_bdy3 ),
                                              StationaryElasticity_DirichletBC_3D ( 2, dir_bdy1
                                                  , dir_bdy2, dir_bdy3 ) };

    // and compute Dirichlet values for pre-set dofs.
    for ( int var = 0; var < DIMENSION; ++var )
    {
        compute_dirichlet_dofs_and_values ( bc[var], space_, var,
                                          dirichlet_dofs_, dirichlet_values_ );
    }
}
```

3.5.5 assemble_system()

In the member function `assemble_system()`, we create the `StationaryElasticityAssembler` object `local_asm`, see section 3.6.2, and call methods for the assembly of the system matrix and the right-hand side vector. In case an effective non-zero Neumann boundary condition should be considered, this is the above mentioned place to do so, too.

```
void Elasticity::assemble_system ( )
{

    if ( !solve_instationary_ )
    {
        TimingScope tscope ( "Assemble_system_stationary" );

        StationaryElasticityAssembler local_asm ( lambda_, mu_, rho_, gravity_ );

        global_asm_.assemble_matrix ( space_, local_asm, matrix_ );

        global_asm_.assemble_vector ( space_, local_asm, rhs_ );

        rhs_.UpdateCouplings ( );

        if ( !dirichlet_dofs_.empty ( ) )
        {
            // Correct Dirichlet dofs.
            matrix_.diagonalize_rows ( vec2ptr ( dirichlet_dofs_ ), dirichlet_dofs_.size ( ),
                 1.0 );
            rhs_.SetValues ( vec2ptr ( dirichlet_dofs_ ), dirichlet_dofs_.size ( ),
                             vec2ptr ( dirichlet_values_ ) );
            sol_.SetValues ( vec2ptr ( dirichlet_dofs_ ), dirichlet_dofs_.size ( ),
                             vec2ptr ( dirichlet_values_ ) );
        }

        rhs_.UpdateCouplings ( );
        sol_.UpdateCouplings ( );

        // Setup Operator for Preconditioner/Solver:
        // (needs to be done AFTER DirichletBC-Setup)
        if ( solver_name_ == "CG" && matrix_precond_ != NOPRECOND )
        {
            preconditioner_.SetupOperator ( matrix_ );
        }
        else if ( solver_name_ == "CG" && matrix_precond_ == NOPRECOND )
        {
            cg_.SetupOperator ( matrix_ );
        }
        else
        {
            // GMRES (and ILUPP) by default.
            if ( use_ilupp_ )
            {
#ifdef WITH_ILUPP
                ilupp_.SetupOperator ( matrix_ );
#endif
            }
        }

    }
}
```

3.5.6 solve_system()

After the setup of the system matrix, solution and right-hand side vector is done, we can compute the solution.

```
void Elasticity::solve_system ( )
{

    // Solver setup is done in prepare_system() and assemble_system().

    if ( !solve_instationary_ )
    {
        TimingScope tscope ( "Solve_system_stationary" );

        // Solve linear system.
        if ( solver_name_ == "CG" )
        {
            cg_.Solve ( rhs_, &sol_ );
        }
        else /*if (solver_name_ == "GMRES")*/
        {
            gmres_.Solve ( rhs_, &sol_ );
        }

        sol_.UpdateCouplings ( );

    }

    if ( solver_name_ == "CG" )
    {
        if ( matrix_precond_ == 0 )
        {
            CONSOLE_OUTPUT ( 3, "Linear solver (CG) not using preconditioner." );
        }
        else
        {
            CONSOLE_OUTPUT ( 3, "Linear solver (CG) using the following preconditioner: " <<
                matrix_precond_ << "." );
            CONSOLE_OUTPUT ( 3, "Note: Jacobi = 1, GaussSeidel = 2, SymmetricGaussSeidel = 3,
                SOR = 4, SSOR = 5, ILU = 6, ILU2 = ILU_pp = 7, ILU_P = 8." );
        }
        CONSOLE_OUTPUT ( 3, "Linear solver (CG) computed solution in " << cg_.iter ( ) << "
            iterations." );
        CONSOLE_OUTPUT ( 3, "Residual norm for solution = " << cg_.res ( ) );
    }
    else
    {
        if ( matrix_precond_ == 0 )
        {
            CONSOLE_OUTPUT ( 3, "Linear solver (GMRES) not using preconditioner." );
        }
        else
        {
            CONSOLE_OUTPUT ( 3, "Linear solver (GMRES) using the following preconditioner: "
                << matrix_precond_ << "." );
            CONSOLE_OUTPUT ( 3, "Note: Jacobi = 1, GaussSeidel = 2, SymmetricGaussSeidel = 3,
                SOR = 4, SSOR = 5, ILU = 6, ILU2 = ILU_pp = 7, ILU_P = 8." );
        }
        CONSOLE_OUTPUT ( 3, "Linear solver (GMRES) computed solution in " << gmres_.iter ( )
            << " iterations." );
        CONSOLE_OUTPUT ( 3, "Residual norm for solution = " << gmres_.res ( ) );
    }
}
```

3.5.7 visualize()

The member function visualize() writes out data for visualization.

```
void Elasticity::visualize ( /* int ts_*/ )
{
    TimingScope tscope ( "Visualization" );

    // Setup visualization object.
    int num_intervals = u_deg; // num_intervals for CellVisualization is reasonably less or
        equal the FE degree.
    ParallelCellVisualization<double> visu ( space_, num_intervals, comm_, MASTER_RANK );

    // Generate filename for mesh which includes solution-array.
    std::stringstream input;
    input << simul_name_ << "_solution";
    const int xml_init_ref_lvl = params_["Mesh"]["InitialRefLevel"].get<int>( );

    input << "_np" << num_partitions ( ) << "_RefLvl" << xml_init_ref_lvl << "_Tstep";
    input << "_stationary";

    if ( num_partitions ( ) > 1 )
        input << ".pvtu";
    else
        input << ".vtu";

    std::vector<double> remote_index ( mesh_->num_entities ( mesh_->tdim ( ) ), 0 );
    std::vector<double> sub_domain ( mesh_->num_entities ( mesh_->tdim ( ) ), 0 );
    std::vector<double> material_number ( mesh_->num_entities ( mesh_->tdim ( ) ), 0 );

    for ( mesh::EntityIterator it = mesh_->begin ( mesh_->tdim ( ) ); it != mesh_->end (
        mesh_->tdim ( ) ); ++it )
    {
        int temp1, temp2;
        mesh_->get_attribute_value ( "_remote_index_", mesh_->tdim ( ),
                                     it->index ( ),
                                     &temp1 );
        mesh_->get_attribute_value ( "_sub_domain_", mesh_->tdim ( ),
                                     it->index ( ),
                                     &temp2 );
        remote_index.at ( it->index ( ) ) = temp1;
        sub_domain.at ( it->index ( ) ) = temp2;
        material_number.at ( it->index ( ) ) = mesh_->get_material_number ( mesh_->tdim ( ),
            it->index ( ) );
    }

    sol_.UpdateCouplings ( );

    visu.visualize ( EvalFeFunction<LAD>( space_, sol_, 0 ), "u0" );
visu.visualize ( EvalFeFunction<LAD>( space_, sol_, 1 ), "u1" );
visu.visualize ( EvalFeFunction<LAD>( space_, sol_, 2 ), "u2" );

visu.visualize_cell_data ( material_number, "Material Id" );
visu.visualize_cell_data ( remote_index, "_remote_index_" );
visu.visualize_cell_data ( sub_domain, "_sub_domain_" );
visu.write ( input.str ( ) );

// Create visualization helper vector from (post-processing/solution) vector (pp_sol_/
    sol_).
std::vector<LAD::DataType> visu_vec ( sol_.size_global ( ), 1.e20 );
std::vector<int> dof_ids;
std::vector<double> values;
sol_.GetAllDofsAndValues ( dof_ids, values ); // solution values (u1,u2,u3) from "sol_"
    are given into the "visu" object.
for ( int i = 0; i < static_cast < int > ( values.size ( ) ); ++i )
{
    visu_vec.at ( dof_ids[i] ) = values.at ( i ); // values.at(i) correspond to
        displacement values (as in (pp_)sol_).
    // Note: visu_vec does not contain the body's deformed state
    // (the deformed state would correspond to orig values + displacement values)
}

// --------------------------------------------------------
// START THE ADDITIONAL DEFORMATION VISUALIZATION PART HERE:
// Compute "deformed state"
// as sum of initial coords (= mesh input) and displacement values (= solution vector u).

std::vector<double> deformation_vector ( 3 * mesh_->num_entities ( 0 ) ); // This vector
    contains the mesh/geometry in deformed state.

std::vector<int> local_dof_id;

AttributePtr sub_domain_attr;
// if attribute _sub_domain_ exists, there are several subdomains, and hence ghost cells,
     too.
const bool has_ghost_cells = mesh_->has_attribute ( "_sub_domain_", mesh_->tdim ( ) );
if ( has_ghost_cells )
{
    sub_domain_attr = mesh_->get_attribute ( "_sub_domain_", mesh_->tdim ( ) );
}

for ( EntityIterator cell = mesh_->begin ( mesh_->tdim ( ) ), end_c = mesh_->end ( mesh_
    ->tdim ( ) ); cell != end_c; ++cell )
{

    int vertex_num = 0;

    if ( has_ghost_cells && sub_domain_attr->get_int_value ( cell->index ( ) ) != rank_ )
    {
        continue; // in order not to write values twice/thrice/... for parallel processes
    }

    for ( IncidentEntityIterator vertex_it = cell->begin_incident ( 0 ); vertex_it != cell
        ->end_incident ( 0 ); ++vertex_it )
    {
        //for (int vertex = 0; vertex < cell->num_vertices(); ++vertex) {
        for ( int var = 0; var < 3; ++var )
        {
                space_.dof ( ).get_dofs_on_subentity ( var, cell->index ( ), 0, vertex_num,
                    local_dof_id );
                deformation_vector.at ( 3 * vertex_it->index ( ) + var ) = visu_vec.at (
                    local_dof_id[0] );
            }
            ++vertex_num;
        }
    }

    // Generate output filename and write output mesh/geometry of deformed state.
    std::stringstream input2;
    input2 << simul_name_ << "_deformedSolution";
    input2 << "_np" << num_partitions ( ) << "_RefLvl" << xml_init_ref_lvl << "_Tstep";
    input2 << "_stationary";

    if ( num_partitions ( ) == 1 )
    {
        input2 << ".pvtu";
        PVtkWriter deformation_writer ( comm_ );
        deformation_writer.set_deformation ( &deformation_vector );
        deformation_writer.add_all_attributes ( *mesh_, true );
        deformation_writer.write ( input2.str ( ).c_str ( ), *mesh_ );

    }
    else if ( num_partitions ( ) > 1 )
    {

        mesh::MeshPtr mesh_ptr;
        mesh::TDim tdim = mesh_->tdim ( );

        if ( mesh_->has_attribute ( "_sub_domain_", tdim ) )
        {
            // create new mesh without ghost layers
            mesh::MeshDbViewBuilder builder ( ( static_cast < mesh::MeshDbView* > ( mesh_.get
                ( ) ) )->get_db ( ) );

            // cell index map
            std::vector<int> index_map;

            // get __remote_indices__ attribute
            mesh::AttributePtr subdomain_attr = mesh_->get_attribute ( "_sub_domain_", tdim );

            // loop over cells that are not ghosts
            for ( mesh::EntityIterator cell_it = mesh_->begin ( tdim ); cell_it != mesh_->end
                ( tdim ); ++cell_it )
            {
                if ( subdomain_attr->get_int_value ( cell_it->index ( ) ) == rank_ )
                {
                    // loop over vertices of cell
                    std::vector<mesh::MeshBuilder::VertexHandle> vertex_handle;
                    for ( mesh::IncidentEntityIterator inc_vert_it = cell_it->begin_incident (
                        0 ); inc_vert_it != cell_it->end_incident ( 0 ); ++inc_vert_it )
                    {
                        std::vector<double> coord ( 3, 0.0 );
                        inc_vert_it->get_coordinates ( coord );
                        vertex_handle.push_back ( builder.add_vertex ( coord ) );
                    }
                    builder.add_entity ( tdim, vertex_handle );
                    index_map.push_back ( cell_it->index ( ) );
                }
            }

            mesh_ptr = builder.build ( );
            mesh::AttributePtr index_map_attr ( new mesh::IntAttribute ( index_map ) );
            mesh_ptr->add_attribute ( "__index_map__", mesh_ptr->tdim ( ), index_map_attr );

            // transfer cell attributes
            std::vector< std::string > cell_attr_names = mesh_->get_attribute_names ( mesh_ptr
                ->tdim ( ) );
            for ( std::vector< std::string >::const_iterator it = cell_attr_names.begin ( ),
                  end_it = cell_attr_names.end ( ); it != end_it; ++it )
            {
                mesh::AttributePtr mapped_attr (
                                                 new mesh::InheritedAttribute ( mesh_->
                                                     get_attribute ( *it, mesh_->tdim ( ) ),
                                                     index_map_attr ) );
                mesh_ptr->add_attribute ( *it, mesh_->tdim ( ), mapped_attr );
            }
        }
        else
        {
            std::cout << "Error in Visualization Routine! Missing subdomain attribute!\n";
        }

        // Since deformation_vector still contains some vertices twice/thrice/... (according
             to num processes, and ghost cells),
        // every vertex is now (by means of an EntityIterator) set/mapped (once only) into a
             mapped_deformation_vector.
        // Thus, finally, the mapped_deformation_vector has the size 3*mesh_ptr->num_entities
             (0).
        std::vector<double> mapped_deformation_vector ( 3 * mesh_ptr->num_entities ( 0 ) ); //
              dim = number of nodes in mesh * dimension
        for ( EntityIterator v_it = mesh_ptr->begin ( 0 ), end_v = mesh_ptr->end ( 0 ); v_it
             != end_v; ++v_it )
        {
            mesh::Id v_id = v_it->id ( );
            mesh::EntityNumber v_index = -1;
            bool found = mesh_->find_entity ( 0, v_id, &v_index );
            if ( !found )
            {
                std::cout << "Error in Visualization Routine! Missing vertex!\n";
            }
            for ( int c = 0; c < 3; ++c )
            {
                mapped_deformation_vector[3 * v_it->index ( ) + c] = deformation_vector[3 *
                     v_index + c];
            }
        }

        input2 << ".pvtu";
        PVtkWriter p_deformation_writer ( comm_ );
        p_deformation_writer.set_deformation ( &mapped_deformation_vector );
        p_deformation_writer.add_all_attributes ( *mesh_ptr, true );
        p_deformation_writer.write ( input2.str ( ).c_str ( ), *mesh_ptr );

    }
    else
    {
        std::cout << "Error in Visualization Routine! Wrong counting and managing the number
            of partitions: num_partitions_ <= 0 is wrong." << std::endl;
    }
}
```

3.6 Assembly Functions

3.6.1 StationaryElasticityAssembler_DirichletBC_3D

In 3.5.4.1 we create an object of the structure `StationaryElasticityAssembler_DirichletBC_3D`, which is part of the header file `elasticity_tutorial.h`. This structure realizes the Dirichlet boundary conditions, i.e. the displacement of the points. In the example in section 5, we modify the Dirichlet boundary conditions and the respective output.

```
struct StationaryElasticity_DirichletBC_3D
{
    // Parameters:
    // bdy - material number of boundary

    StationaryElasticity_DirichletBC_3D ( int var, int bdy1, int bdy2, int bdy3 )
    : var_ ( var ), bdy1_ ( bdy1 ), bdy2_ ( bdy2 ), bdy3_ ( bdy3 )
    {
        assert ( var_ == 0 || var_ == 1 || var_ == 2 );
        assert ( DIMENSION == 3 );
    }

    std::vector<double> evaluate ( const Entity& face, const std::vector<Coord>&
        coords_on_face ) const
    {
        //Return array with Dirichlet values for dof:s on boundary face.
        std::vector<double> values;

        const int material_num = face.get_material_number ( );

        if ( material_num == bdy1_ )
        { // NOTE: this is the fixed Dirichlet BC part, and holds during the whole simulation.
            values.resize ( coords_on_face.size ( ) );

            // loop over dof points on the face
            for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
            {
                // evaluate dirichlet function at each point
                values[i] = 0.;
            }
        }
        else if ( material_num == bdy2_ )
        { // NOTE: this is a displacement Dirichlet BC, and holds ONLY for the time dt*
            timestep <= 1.0.
            values.resize ( coords_on_face.size ( ) );

            // loop over dof points on the face
            for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
            {
                // evaluate dirichlet function at each point
                values[i] = 0.;
                if ( var_ == 0 )
                {
                    values[i] = 0.;
                }
                else if ( var_ == 1 )
                {
                    values[i] = 0.04;
                }
                else if ( var_ == 2 )
                {
                    values[i] = 0.02;
                }
            }

        }
        else if ( material_num == bdy3_ )
        { // NOTE: this is a displacement Dirichlet BC, and holds ONLY for the time dt*
             timestep <= 1.0.
            values.resize ( coords_on_face.size ( ) );

            // loop over dof points on the face
            for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
            {
                // evaluate dirichlet function at each point
                values[i] = 0.;
                if ( var_ == 0 )
                {
                    values[i] = 0.;
                }
                else if ( var_ == 1 )
                {
                    values[i] = 0.04;
                }
                else if ( var_ == 2 )
                {
                    values[i] = 0.;
                }
            }

        }
        else
        {
            assert ( 0 );
        }

        return values;
    }

    const int var_;
    const int bdy1_, bdy2_, bdy3_;
};
```

3.6.2 StationaryElasticityAssembler

This class implements the system matrix and right-hand side vector locally for each cell. It can be found in `elasticity_tutorial.h`. We create an object of this class in 3.5.5.

```
class StationaryElasticityAssembler : private AssemblyAssistant<DIMENSION, double>
{
  public:
```

```
StationaryElasticityAssembler ( double lambda, double mu, double rho, double gravity )
: lambda_ ( lambda ), mu_ ( mu ), rho_ ( rho ), gravity_ ( gravity )
{
}

void operator() ( const Element<double>& element, const Quadrature<double>& quadrature,
    LocalMatrix& lm )
{
   AssemblyAssistant<DIMENSION, double>::initialize_for_element ( element, quadrature );

   const int num_q = num_quadrature_points ( );

   // loop over quadrature points.
   for ( int q = 0; q < num_q; ++q )
   {
      const double wq = w ( q );
      const double dJ = std::abs ( detJ ( q ) );

      // assemble \int {lambda \div(u) \div(phi)}
      for ( int test_var = 0; test_var < DIMENSION; ++test_var )
      {
         for ( int trial_var = 0; trial_var < DIMENSION; ++trial_var )
         {
            for ( int i = 0; i < num_dofs ( test_var ); ++i )
            {
               for ( int j = 0; j < num_dofs ( trial_var ); ++j )
               {
                  lm ( dof_index ( i, test_var ), dof_index ( j, trial_var ) ) +=
                        wq * lambda_ * grad_phi ( j, q, trial_var )[trial_var] *
                            grad_phi ( i, q, test_var )[test_var] * dJ;
               }
            }
         }
      }

      // assemble \int {mu [ \frob(\nabla(u)\nabla(\phi)) + \frob(\nabla(u)^T\nabla(\phi
          )) ] }
      for ( int var = 0; var < DIMENSION; ++var )
      {
         for ( int i = 0; i < num_dofs ( var ); ++i )
         {
            for ( int j = 0; j < num_dofs ( var ); ++j )
            {
               for ( int var_frob = 0; var_frob < DIMENSION; ++var_frob )
               {
                  lm ( dof_index ( i, var ), dof_index ( j, var ) ) +=
                        wq * mu_ * ( grad_phi ( j, q, var )[var_frob] ) * grad_phi (
                             i, q, var )[var_frob] * dJ;
                  lm ( dof_index ( i, var ), dof_index ( j, var_frob ) ) +=
                        wq * mu_ * ( grad_phi ( j, q, var_frob )[var] ) * grad_phi (
                             i, q, var )[var_frob] * dJ;
               }
            }
         }
      }

   }
}
```

```
    void operator() ( const Element<double>& element, const Quadrature<double>& quadrature,
        LocalVector& lv )
    {
        AssemblyAssistant<DIMENSION, double>::initialize_for_element ( element, quadrature );

        const int num_q = num_quadrature_points ( );

        const double source[3] = { 0., gravity_ * rho_, 0. };

        // loop over quadrature points.
        for ( int q = 0; q < num_q; ++q )
        {
            const double wq = w ( q );
            const double dJ = std::abs ( detJ ( q ) );

            // assemble l(v) = \rho * \int( f_ext \phi )
            for ( int test_var = 0; test_var < DIMENSION; ++test_var )
            {
                for ( int i = 0; i < num_dofs ( test_var ); ++i )
                {
                    lv[dof_index ( i, test_var )] +=
                            wq * source[test_var] * phi ( i, q, test_var ) * dJ;
                }
            }
        }
    }

  private:
    double lambda_, mu_, rho_, gravity_;
};
```

4 Program Output and Visualization

HiFlow^3 can be executed in a parallel or sequential mode which influences the generated output data. Note that the log files can be viewed by any editor. Executing the program in parallel mode with `X` processes by

```
mpirun -np X elasticity_tutorial /"path_to"/elasticity_tutorial.xml
```

or in sequential mode by

```
./elasticity_tutorial /"path_to"/elasticity_tutorial.xml
```

generates following output data.

- Mesh/geometry data:
  - `elasticity_tutorial_initial_mesh_local.pvtu` Global mesh of initial refinement level 0 (parallel vtk-format). It combines the local meshes of sequential vtk-format owned by the different processes to the global mesh.
  - `elasticity_tutorial_initial_mesh_local_Y.vtu` Local mesh of refinement level 0 owned by process `Y` for `Y=0,1,2,...,X-1` (vtk-format).
- Solution data:
  - `elasticity_tutorial_deformedSolution_npX_RefLvl0_Tstep_stationary.pvtu` and `elasticity_tutorial_solution_npX_RefLvl0_Tstep_stationary.pvtu`

Solution of the elasticity problem for refinement level 0 in (parallel) vtk-format. It combines the local solutions owned by the different processes to a global solution. The deformed solution is directly suitable for visualization of the deformed state of the object. The solution contains all information available through the computation process and is suitable for more complex visualization tasks for more advanced users.

  - `elasticity_tutorial_deformedSolution_npX_RefLvl0_Tstep_stationary_Y.vtu` and `elasticity_tutorial_solution_npX_RefLvl0_Tstep_stationary_Y.vtu` Local solution of the elasticity problem for refinement level 0 owned by process `Y`, for `Y=0,1,2,...,X-1` (vtk-format).

- Log files:
  - `elasticity_tutorial_debug_log` Log file listing errors helping to simplify the debugging process. This file is empty if the program runs without errors.
  - `elasticity_tutorial_info_log` Log file listing parameters and some helpful information to control the program.

- Terminal output:
  - Number of cells of the mesh.
  - Number of cells of the refined mesh.
  - Total number of dofs.
  - Number of dofs assigned to each process.
  - Timing report containing information about the duration of the setup, assembly and solution of the system, and visualization.

HiFlow^3 only generates output data but does not visualize. The mesh/geometry data as well as the solution data can be visualized by any external program which can handle the vtk data format, e.g. the program ParaView [6].

5 Example

We present two examples that show the deformation of the Stanford Bunny [7], which is included in the tutorial (stanford_bunny.inp), by altering the gravity in the parameter file 3.2 and the displacement of the tips of the bunny's ears in the file 3.6.1. The initial state can be seen in figure 2. The images were created with ParaView [6].
In figure 3 and 4 the wireframe model shows the initial state of the bunny. Its deformed state is represented by the solid model. We obtain the deformation as shown in figure 3 if we set the gravity in the parameter file 3.2 to -9.81

```
Parameter File -> ElasticityModel -> gravity -> -9.81
```

and change the displacement of the tips of the ears in the file 3.6.1 as follows:

```
if ( material_num == bdy1_ )
      {
          values.resize ( coords_on_face.size ( ) );

          // loop over dof points on the face
          for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
          {
```

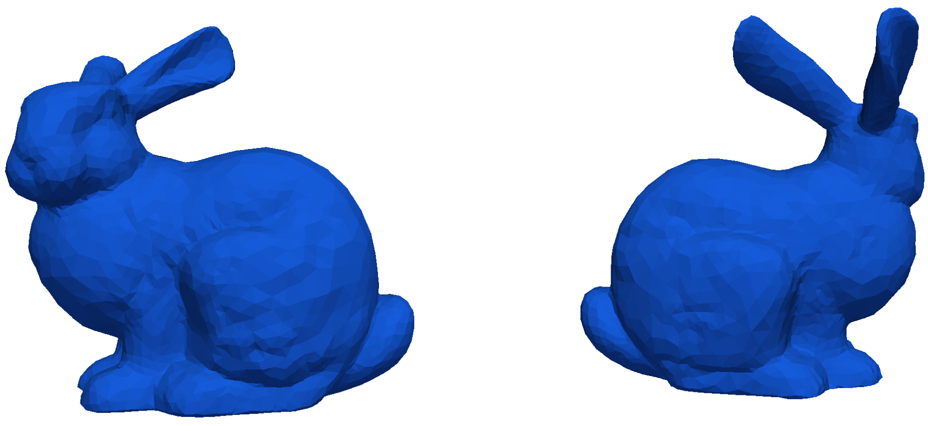
Figure 2: Initial state of the bunny.

```
            // evaluate dirichlet function at each point
            values[i] = 0.;
        }
    }
    else if ( material_num == bdy2_ )
    {
        values.resize ( coords_on_face.size ( ) );

        // loop over dof points on the face
        for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
        {
            // evaluate dirichlet function at each point
            values[i] = 0.;
            if ( var_ == 0 )
            {
                values[i] = 0.;
            }
            else if ( var_ == 1 )
            {
                values[i] = 0.04;
            }
            else if ( var_ == 2 )
            {
                values[i] = 0.02;
            }
        }

    }
    else if ( material_num == bdy3_ )
    {

        // loop over dof points on the face
        for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
        {
            // evaluate dirichlet function at each point
            values[i] = 0.;
            if ( var_ == 0 )
            {
                values[i] = 0.;
            }
            else if ( var_ == 1 )
            {
                values[i] = 0.04;
            }
            else if ( var_ == 2 )
            {
                values[i] = 0.;
            }
        }

    }
```

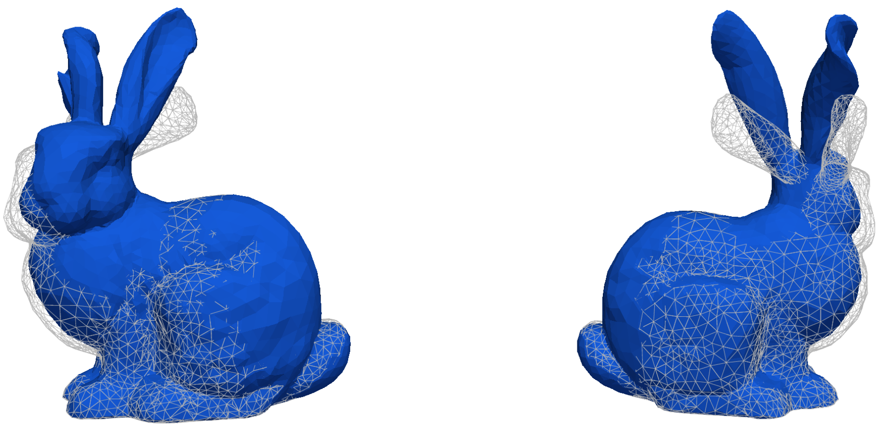

Figure 3: Fixed displacement of the tips of the bunny's ears.

Setting the gravity to -300.00

```
Parameter File -> ElasticityModel -> gravity -> -300.00
```

and the displacement of the ears as shown in the code below, leads to the deformation displayed in figure 4.

```
if ( material_num == bdy1_ )
        {
            values.resize ( coords_on_face.size ( ) );

            // loop over dof points on the face
            for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
            {
                // evaluate dirichlet function at each point
                values[i] = 0.;
            }
        }
        else if ( material_num == bdy2_ )
        {
            values.resize ( coords_on_face.size ( ) );

            // loop over dof points on the face
            for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
            {
                // evaluate dirichlet function at each point
                values[i] = 0.;
                if ( var_ == 0 )
                {
```

```
                values[i] = 0.;
            }
            else if ( var_ == 1 )
            {
                values[i] = 0.;
            }
            else if ( var_ == 2 )
            {
                values[i] = 0.;
            }
        }

    }
    else if ( material_num == bdy3_ )
    {
        values.resize ( coords_on_face.size ( ) );

        // loop over dof points on the face
        for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
        {
            // evaluate dirichlet function at each point
            values[i] = 0.;
            if ( var_ == 0 )
            {
                values[i] = 0.;
            }
            else if ( var_ == 1 )
            {
                values[i] = 0.;
            }
            else if ( var_ == 2 )
            {
                values[i] = 0.;
            }
        }

    }
```

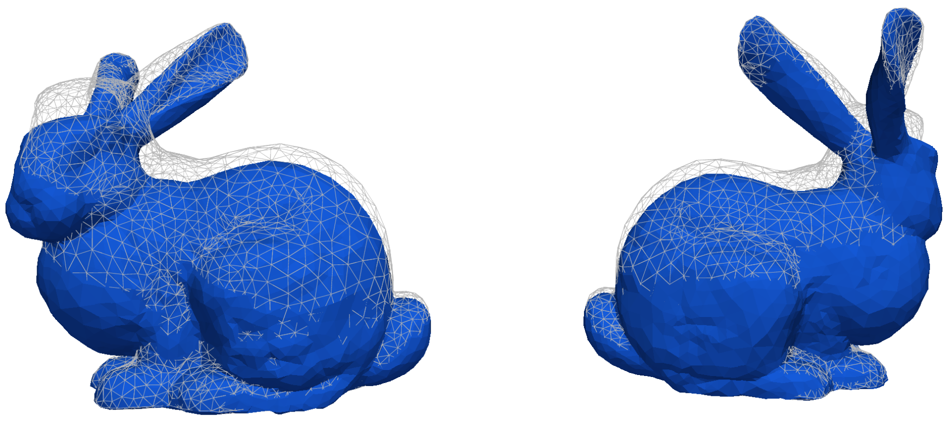
Figure 4: Bunny under the influence of increased gravity.

6 Acknowledgements

This work was carried out with the support of the German Research Foundation (DFG) within the project I03 of the Collaborative Research Center SFB/TRR 125 'Cognition-Guided Surgery'.

References

[1] Dietrich Braess. *Finite Elemente : Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Springer, Berlin, 4., überarb. und erw. aufl. edition, 2007.

[2] Rolf Rannacher. *Numerische Mathematik 3*. Lecture notes, 2004. Available at `http://www.numerik.uni-hd.de`.

[3] Klaus-Jürgen Bathe. *Finite-Elemente-Methoden*. Springer, 2002.

[4] William D. Gropp. *MPI - Eine Einführung*. Oldenbourg, 2007.

[5] The message passing interface (mpi) standard. `http://www.mcs.anl.gov/research/projects/mpi`.

[6] Amy Henderson Squillacote. *The ParaView guide: a parallel visualization application*. Kitware, Clifton Park, NY, 2007.

[7] The stanford bunny. `http://www.gvu.gatech.edu/people/faculty/greg.turk/bunny/bunny.html`.