This tutorial program is another one in the series on the elasticity problem that we have already started with step-8 and step-17. It extends it into two different directions: first, it solves the quasistatic but time dependent elasticity problem for large deformations with a Lagrangian mesh movement approach. Secondly, it shows some more techniques for solving such problems using parallel processing with PETSc's linear algebra. In addition to this, we show how to work around the main bottleneck of step-17, namely that we generated graphical output from only one process, and that this scaled very badly with larger numbers of processes and on large problems. Finally, a good number of assorted improvements and techniques are demonstrated that have not been shown yet in previous programs.

As before in step-17, the program runs just as fine on a single sequential machine as long as you have PETSc installed. Information on how to tell deal.II about a PETSc installation on your system can be found in the deal.II README file, which is linked to from the main documentation page `doc/index.html` in your installation of deal.II, or on the deal.II webpage `http://www.dealii.org/`.

Quasistatic elastic deformation

Motivation of the model

In general, time-dependent small elastic deformations are described by the elastic wave equation

$$\rho \frac{\partial^2 \mathbf{u}}{\partial t^2} + c\frac{\partial \mathbf{u}}{\partial t} - \text{div }(C\varepsilon(\mathbf{u})) = \mathbf{f} \qquad \text{in } \Omega, \tag{1}$$

where $\mathbf{u} = \mathbf{u}(\mathbf{x}, t)$ is the deformation of the body, ρ and c the density and attenuation coefficient, and $\mathbf{f}$ external forces. In addition, initial conditions

$$\mathbf{u}(\cdot, 0) = \mathbf{u}_0(\cdot) \qquad \text{on } \Omega, \tag{2}$$

and Dirichlet (displacement) or Neumann (traction) boundary conditions need to be specified for a unique solution:

$$\begin{aligned} \mathbf{u}(\mathbf{x}, t) &= \mathbf{d}(\mathbf{x}, t) && \text{on } \Gamma_D \subset \partial\Omega, && (3) \\ \mathbf{n}\ C\varepsilon(\mathbf{u}(\mathbf{x}, t)) &= \mathbf{b}(\mathbf{x}, t) && \text{on } \Gamma_N = \partial\Omega \backslash \Gamma_D. && (4) \end{aligned}$$

In above formulation, $\varepsilon(\mathbf{u}) = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$ is the symmetric gradient of the displacement, also called the *strain*. C is a tensor of rank 4, called the *stress-strain tensor* that contains knowledge of the elastic strength of the material; its symmetry properties make sure that it maps symmetric tensors of rank 2 ("matrices" of dimension d, where d is the spatial dimensionality) onto symmetric tensors of the same rank. We will comment on the roles of the strain and stress tensors more below. For the moment it suffices to say that we interpret the term div $(C\varepsilon(\mathbf{u}))$ as the vector with components $\frac{\partial}{\partial x_j} C_{ijkl}\varepsilon(\mathbf{u})_{kl}$, where summation over indices j, k, l is implied.

The quasistatic limit of this equation is motivated as follows: each small perturbation of the body, for example by changes in boundary condition or the forcing function, will result in a corresponding change in the configuration of the body. In general, this will be in the form of waves radiating away from the location of the disturbance. Due to the presence of the damping term, these waves will be attenuated on a time scale of, say, τ. Now, assume that all changes in external forcing happen on times scales that are much larger than τ. In that case, the dynamic nature of the change is unimportant: we can consider the body to always be in static equilibrium, i.e. we can assume that at all times the body satisfies

$$\begin{aligned} -\text{div}\ (C\varepsilon(\mathbf{u})) &= \mathbf{f} && \text{in } \Omega, && (5)\\ \mathbf{u}(\mathbf{x},t) &= \mathbf{d}(\mathbf{x},t) && \text{on } \Gamma_D, && (6)\\ \mathbf{n}\ C\varepsilon(\mathbf{u}(\mathbf{x},t)) &= \mathbf{b}(\mathbf{x},t) && \text{on } \Gamma_N. && (7) \end{aligned}$$

Note that the differential equation does not contain any time derivatives any more – all time dependence is introduced through boundary conditions and a possibly time-varying force function $\mathbf{f}(\mathbf{x},t)$.

While these equations are sufficient to describe small deformations, computing large deformations is a little more complicated. To do so, let us first introduce a stress variable σ, and write the differential equations in terms of the stress:

$$\begin{aligned} -\text{div}\ \sigma &= \mathbf{f} && \text{in } \Omega(t), && (8)\\ \mathbf{u}(\mathbf{x},t) &= \mathbf{d}(\mathbf{x},t) && \text{on } \Gamma_D \subset \partial\Omega(t), && (9)\\ \mathbf{n}\ C\varepsilon(\mathbf{u}(\mathbf{x},t)) &= \mathbf{b}(\mathbf{x},t) && \text{on } \Gamma_N = \partial\Omega(t)\backslash\Gamma_D. && (10) \end{aligned}$$

Note that these equations are posed on a domain $\Omega(t)$ that changes with time, with the boundary moving according to the displacements $\mathbf{u}(\mathbf{x},t)$ of the points on the boundary. To complete this system, we have to specify the relationship between the stress and the strain, as follows:

$$\dot{\sigma} = C\varepsilon(\dot{\mathbf{u}}), \tag{11}$$

where a dot indicates a time derivative. Both the stress σ and the strain $\varepsilon(\mathbf{u})$ are symmetric tensors of rank 2.

Time discretization

Numerically, this system is solved as follows: first, we discretize the time component using a backward Euler scheme. This leads to a discrete equilibrium of force at time step n:

$$-\text{div}\ \sigma^n = f^n, \tag{12}$$

where

$$\sigma^n = \sigma^{n-1} + C\varepsilon(\Delta\mathbf{u}^n), \tag{13}$$

and $\Delta\mathbf{u}^n$ the incremental displacement for time step n. This way, if we want to solve for the displacement increment, we have to solve the following system:

$$
\begin{aligned}
-\mathrm{div}\ C\varepsilon(\Delta\mathbf{u}^n) &= \mathbf{f} + \mathrm{div}\ \sigma^{n-1} && \text{in } \Omega(t_{n-1}), && (14)\\
\Delta\mathbf{u}^n(\mathbf{x},t) &= \mathbf{d}(\mathbf{x},t_n) - \mathbf{d}(\mathbf{x},t_{n-1}) && \text{on } \Gamma_D \subset \partial\Omega(t_{n-1}), && (15)\\
\mathbf{n}\ C\varepsilon(\Delta\mathbf{u}^n(\mathbf{x},t)) &= \mathbf{b}(\mathbf{x},t_n) - \mathbf{b}(\mathbf{x},t_{n-1}) && \text{on } \Gamma_N = \partial\Omega(t_{n-1})\backslash\Gamma_D. && (16)
\end{aligned}
$$

The weak form of this set of equations, which as usual is the basis for the finite element formulation, reads as follows: find $\Delta\mathbf{u}^n \in \{v \in H^1(\Omega(t_{n-1}))^d : v|_{\Gamma_D} = \mathbf{d}(\cdot,t_n) - \mathbf{d}(\cdot,t_{n-1})\}$ such that

$$
\begin{aligned}
(C\varepsilon(\Delta\mathbf{u}^n), \varepsilon(\varphi))_{\Omega(t_{n-1})} = (\mathbf{f},\varphi)_{\Omega(t_{n-1})} - (\sigma^{n-1}, \varepsilon(\varphi))_{\Omega(t_{n-1})} \\
+ (\mathbf{b}(\mathbf{x},t_n) - \mathbf{b}(\mathbf{x},t_{n-1}), \varphi)_{\Gamma_N} \qquad (17)\\
\forall\varphi \in \{\mathbf{v} \in H^1(\Omega(t_{n-1}))^d : \mathbf{v}|_{\Gamma_D} = 0\}.
\end{aligned}
$$

We note that, for simplicity, in the program we will always assume that there are no boundary forces, i.e. $\mathbf{b} = 0$, and that the deformation of the body is driven by body forces $\mathbf{f}$ and prescribed boundary displacements $\mathbf{d}$ alone. It is also worth noting that when integrating by parts, we would get terms of the form $(C\varepsilon(\Delta\mathbf{u}^n), \nabla\varphi)_{\Omega(t_{n-1})}$, but that we replace it with the term involving the symmetric gradient $\varepsilon(\varphi)$ instead of $\nabla\varphi$. Due to the symmetry of C, the two terms are equivalent, but the symmetric version avoids a potential for round-off to render the resulting matrix slightly non-symmetric.

The system at time step n, to be solved on the old domain $\Omega(t_{n-1})$, has exactly the form of a stationary elastic problem, and is therefore similar to what we have already implemented in previous example programs. We will therefore not comment on the space discretization beyond saying that we again use lowest order continuous finite elements.

There are differences, however:

1. We have to move the mesh after each time step, in order to be able to solve the next time step on a new domain;
2. We need to know σ^{n-1} to compute the next incremental displacement, i.e. we need to compute it at the end of the time step to make sure it is available for the next time step. Essentially, the stress variable is our window to the history of deformation of the body.

These two operations are done in the functions `move_mesh` and `update_quadrature_point_history` in the program. While moving the mesh is only a technicality, updating the stress is a little more complicated and will be discussed in the next section.

Updating the stress variable

As indicated above, we need to have the stress variable σ^n available when computing time step $n+1$, and we can compute it using

$$\sigma^n = \sigma^{n-1} + C\varepsilon(\Delta\mathbf{u}^n). \qquad (18)$$

There are, despite the apparent simplicity of this equation, two questions that we need to discuss. The first concerns the way we store σ^n: even if we compute the incremental updates $\Delta\mathbf{u}^n$ using lowest-order finite elements, then its symmetric gradient $\varepsilon(\Delta\mathbf{u}^n)$ is in general still a function that is not easy to describe. In particular, it is not a piecewise constant function, and on general meshes (with cells that are not rectangles parallel to the coordinate axes) or with non-constant stress-strain tensors C it is not even a bi- or trilinear function. Thus, it is a priori not clear how to store σ^n in a computer program.

To decide this, we have to see where it is used. The only place where we require the stress is in the term $(\sigma^{n-1}, \varepsilon(\varphi))_{\Omega(t_{n-1})}$. In practice, we of course replace this term by numerical quadrature:

$$(\sigma^{n-1}, \varepsilon(\varphi))_{\Omega(t_{n-1})} = \sum_{K\subset\mathbb{T}} (\sigma^{n-1}, \varepsilon(\varphi))_K \approx \sum_{K\subset\mathbb{T}} \sum_q w_q \ \sigma^{n-1}(\mathbf{x}_q) : \varepsilon(\varphi(\mathbf{x}_q)), \tag{19}$$

where w_q are the quadrature weights and $\mathbf{x}_q$ the quadrature points on cell K. This should make clear that what we really need is not the stress σ^{n-1} in itself, but only the values of the stress in the quadrature points on all cells. This, however, is a simpler task: we only have to provide a data structure that is able to hold one symmetric tensor of rank 2 for each quadrature point on all cells (or, since we compute in parallel, all quadrature points of all cells that the present MPI process "owns"). At the end of each time step we then only have to evaluate $\varepsilon(\Delta\mathbf{u}^n(\mathbf{x}_q))$, multiply it by the stress-strain tensor C, and use the result to update the stress $\sigma^n(\mathbf{x}_q)$ at quadrature point q.

The second complication is not visible in our notation as chosen above. It is due to the fact that we compute Δu^n on the domain $\Omega(t_{n-1})$, and then use this displacement increment to both update the stress as well as move the mesh nodes around to get to $\Omega(t_n)$ on which the next increment is computed. What we have to make sure, in this context, is that moving the mesh does not only involve moving around the nodes, but also making corresponding changes to the stress variable: the updated stress is a variable that is defined with respect to the coordinate system of the material in the old domain, and has to be transferred to the new domain. The reason for this can be understood as follows: locally, the incremental deformation $\Delta\mathbf{u}$ can be decomposed into three parts, a linear translation (the constant part of the displacement field in the neighborhood of a point), a dilational component (that part of the gradient if the displacement field that has a nonzero divergence), and a rotation. A linear translation of the material does not affect the stresses that are frozen into it – the stress values are simply translated along. The dilational or compressional change produces a corresponding stress update. However, the rotational component does not necessarily induce a nonzero stress update (think, in 2d, for example of the situation where $\Delta\mathbf{u} = (y, -x)^T$, which which $\varepsilon(\Delta\mathbf{u}) = 0$). Nevertheless, if the the material was pre-stressed in a certain direction, then this direction will be rotated along with the material. To this end, we have to define a rotation matrix $R(\Delta\mathbf{u}^n)$ that describes, in each point the rotation due to the displacement increments. It is not hard to see that the actual dependence of R on $\Delta\mathbf{u}^n$ can only be through the curl of the displacement, rather than the displacement itself or its full gradient (as mentioned above, the constant components of the increment describe translations, its divergence the dilational modes, and the curl the rotational modes). Since the exact form of R is cumbersome, we only state it in the program code, and note that the correct updating formula for the stress variable is then

$$\sigma^n = R(\Delta\mathbf{u}^n)^T[\sigma^{n-1} + C\varepsilon(\Delta\mathbf{u}^n)]R(\Delta\mathbf{u}^n). \tag{20}$$

Both stress update and rotation are implemented in the function `update_-quadrature_point_history` of the example program.

Parallel graphical output

In the step-17 example program, the main bottleneck for parallel computations was that only the first processor generated output for the entire domain. Since generating graphical output is expensive, this did not scale well when large numbers of processors were involved. However, no viable ways around this problem were implemented in the library at the time, and the problem was deferred to a later version.

This functionality has been implemented in the meantime, and this is the time to explain its use. Basically, what we need to do is let every process generate graphical output for that subset of cells that it owns, write them into separate files and have a way to merge them later on. At this point, it should be noted that none of the graphical output formats known to the author of this program allows for a simple way to later re-read it and merge it with other files corresponding to the same simulation. What deal.II therefore offers is the following: When you call the `DataOut::build_patches` function, an intermediate format is generated that contains all the information for the data on each cell. Usually, this intermediate format is then further processed and converted into one of the graphical formats that we can presently write, such as gmv, eps, ucd, gnuplot, or a number of other ones. Once written in these formats, there is no way to reconstruct the necessary information to merge multiple blocks of output. However, the base classes of `DataOut` also allow to simply dump the intermediate format to a file, from which it can later be recovered without loss of information.

This has two advantages: first, simulations may just dump the intermediate format data during run-time, and the user may later decide which particular graphics format she wants to have. This way, she does not have to re-run the entire simulation if graphical output is requested in a different format. One typical case is that one would like to take a quick look at the data with gnuplot, and then create high-quality pictures using GMV or OpenDX. Since both can be generated out of the intermediate format without problem, there is no need to re-run the simulation.

In the present context, of more interest is the fact that in contrast to any of the other formats, it is simple to merge multiple files of intermediate format, if they belong to the same simulation. This is what we will do here: we will generate one output file in intermediate format for each processor that belongs to this computation (in the sequential case, this will simply be a single file). They may then later be read in and merged so that we can output a single file in whatever graphical format is requested.

The way to do this is to first instruct the `DataOutBase` class to write intermediate format rather than in gmv or any other graphical format. This is simple: just use `data_out.write_deal_II_intermediate`. We will write to a file called `solution-TTTT.TTTT.d2` if there is only one processor, or files `solution-TTTT.TTTT.NNN.d2` if this is really a parallel job. Here, `TTTT.TTTT` denotes the time for which this output has been generated, and `NNN` the number of the MPI process that did this.

The next step is to convert this file or these files into whatever format you like. The program that does this is the step-19 tutorial program: for example, for the first time step, call it through

```
../step-19/step-19 solution-0001.0000.*.d2 solution-0001.0000.gmv
```

to merge all the intermediate format files into a single file in GMV format. More details on the parameters of this program and what it can do for you can be found in the documentation of the step-19 tutorial program.

Overall structure of the program

The overall structure of the program can be inferred from the `run()` function that first calls `do_initial_timestep()` for the first time step, and then `do_timestep()` on all subsequent time steps. The difference between these functions is only that in the first time step we start on a coarse mesh, solve on it, refine the mesh adaptively, and then start again with a clean state on that new mesh. This procedure gives us a better starting mesh, although we should of course keep adapting the mesh as iterations proceed – this isn't done in this program, but commented on below.

The common part of the two functions treating time steps is the following sequence of operations on the present mesh:

- `assemble_system ()` [via `solve_timestep ()`]: This first function is also the most interesting one. It assembles the linear system corresponding to the discretized version of equation (**??**). This leads to a system matrix $A_{ij} = \sum_K A^K_{ij}$ built up of local contributions on each cell K with entries

  $$A^K_{ij} = (C\varepsilon(\varphi_i), \varepsilon(\varphi_j))_K; \tag{21}$$

  In practice, A^K is computed using numerical quadrature according to the formula

  $$A^K_{ij} = \sum_q w_q [\varepsilon(\varphi_i(\mathbf{x}_q)) : C : \varepsilon(\varphi_j(\mathbf{x}_q))], \tag{22}$$

with quadrature points $\mathbf{x}_q$ and weights w_q. We have built these contributions before, in step-8 and step-17, but in both of these cases we have done so rather clumsily by using knowledge of how the rank-4 tensor C is composed, and considering individual elements of the strain tensors $\varepsilon(\varphi_i), \varepsilon(\varphi_j)$. This is not really convenient, in particular if we want to consider more complicated elasticity models than the isotropic case for which C had the convenient form $c_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})$. While we in fact do not use a more complicated form than this in the present program, we nevertheless want to write it in a way that would easily allow for this. It is then natural to introduce classes that represent symmetric tensors of rank 2 (for the strains and stresses) and 4 (for the stress-strain tensor C). Fortunately, deal.II provides these: the `SymmetricTensor<rank,dim>` class template provides a full-fledged implementation of such tensors of rank `rank` (which needs to be an even number) and dimension `dim`.

What we then need is two things: a way to create the stress-strain rank-4 tensor C as well as to create a symmetric tensor of rank 2 (the strain tensor) from the gradients of a shape function φ_i at a quadrature point $\mathbf{x}_q$ on a given cell. At the top of the implementation of this example program, you will find such functions. The first one, `get_stress_strain_tensor`, takes two arguments corresponding to the Lamé constants λ and μ and returns the stress-strain tensor for the isotropic case corresponding to these constants (in the program, we will choose constants corresponding to steel); it would be simple to replace this function by one that computes this tensor for the anisotropic case, or taking into account crystal symmetries, for example. The second one, `get_strain` takes an object of type `FEValues` and indices i and q and returns the symmetric gradient, i.e. the strain, corresponding to shape function $\varphi_i(\mathbf{x}_q)$, evaluated on the cell on which the `FEValues` object was last reinitialized.

Given this, the innermost loop of `assemble_system` computes the local contributions to the matrix in the following elegant way (the variable `stress_strain_tensor`, corresponding to the tensor C, has previously been initialized with the result of the first function above):

```
for (unsigned int i=0; i<dofs_per_cell; ++i)
  for (unsigned int j=0; j<dofs_per_cell; ++j)
    for (unsigned int q_point=0; q_point<n_q_points;
         ++q_point)
      {
        const SymmetricTensor<2,dim>
          eps_phi_i = get_strain (fe_values, i, q_point),
          eps_phi_j = get_strain (fe_values, j, q_point);

        cell_matrix(i,j)
          += (eps_phi_i * stress_strain_tensor * eps_phi_j
              *
          fe_values.JxW (q_point));
    }
```

It is worth noting the expressive power of this piece of code, and to compare it with the complications we had to go through in previous examples for the elasticity problem. (To be fair, the `SymmetricTensor` class template did not exist when these previous examples were written.) For simplicity, `operator*` provides for the (double summation) product between symmetric tensors of even rank here.

Assembling the local contributions

$$\begin{aligned} f_i^K &= (\mathbf{f}, \varphi_i)_K - (\sigma^{n-1}, \varepsilon(\varphi_i))_K \\ &\approx \sum_q w_q \left\{ \mathbf{f}(\mathbf{x}_q) \cdot \varphi_i(\mathbf{x}_q) - \sigma_q^{n-1} : \varepsilon(\varphi_i(\mathbf{x}_q)) \right\} \end{aligned} \tag{23}$$

to the right hand side of (**??**) is equally straightforward (note that we do not consider any boundary tractions **b** here). Remember that we only had to store the old stress in the quadrature points of cells. In the program, we will provide a variable `local_quadrature_points_data` that allows to access the stress σ_q^{n-1} in each quadrature point. With this the code for the right hand side looks as this, again rather elegant:

```
for (unsigned int i=0; i<dofs_per_cell; ++i)
  {
    const unsigned int
      component_i = fe.system_to_component_index(i).first;

    for (unsigned int q_point=0; q_point<n_q_points; ++q_point)
      {
        const SymmetricTensor<2,dim> &old_stress
          = local_quadrature_points_data[q_point].old_stress;

        cell_rhs(i) += (body_force_values[q_point](component_i) *
                         fe_values.shape_value (i,q_point)
                         -
                         old_stress *
                         get_strain (fe_values,i,q_point))
                        *
                        fe_values.JxW (q_point);
      }
  }
```

Note that in the multiplication $\mathbf{f}(\mathbf{x}_q) \cdot \varphi_i(\mathbf{x}_q)$, we have made use of the fact that for the chosen finite element, only one vector component (namely `component_i`) of φ_i is nonzero, and that we therefore also have to consider only one component of $\mathbf{f}(\mathbf{x}_q)$.

This essentially concludes the new material we present in this function. It later has to deal with boundary conditions as well as hanging node constraints, but this parallels what we had to do previously in other programs already.

- `solve_linear_problem ()` [via `solve_timestep ()`]: Unlike the previous one, this function is not really interesting, since it does what similar functions have done in all previous tutorial programs – solving the linear system using the CG method, using an incomplete LU decomposition as a preconditioner (in the parallel case, it uses an ILU of each processor's block separately). It is virtually unchanged from step-17.
- `update_quadrature_point_history ()` [via `solve_timestep ()`]: Based on the displacement field $\Delta\mathbf{u}^n$ computed before, we update the stress values in all quadrature points according to (**??**) and (**??**), including the rotation of the coordinate system.
- `move_mesh ()`: Given the solution computed before, in this function we deform the mesh by moving each vertex by the displacement vector field evaluated at this particular vertex.
- `output_results ()`: This function simply outputs the solution based on what we have said above, i.e. every processor computes output only for its own portion of the domain, and this can then be later merged by an external program. In addition to the solution, we also compute the norm of the stress averaged over all the quadrature points on each cell.

With this general structure of the code, we only have to define what case we want to solve. For the present program, we have chosen to simulate the quasistatic deformation of a vertical cylinder for which the bottom boundary is fixed and the top boundary is pushed down at a prescribed vertical velocity. However, the horizontal velocity of the top boundary is left unspecified – one can imagine this situation as a well-greased plate pushing from the top onto the cylinder, the points on the top boundary of the cylinder being allowed to slide horizontally along the surface of the plate, but forced to move downward by the plate. The inner and outer boundaries of the cylinder are free and not subject to any prescribed deflection or traction. In addition, gravity acts on the body.

The program text will reveal more about how to implement this situation, and the results section will show what displacement pattern comes out of this simulation.

Possible directions for extensions

The program as is does not really solve an equation that has many applications in practice: quasi-static material deformation based on a purely elastic law is almost boring. However, the program may serve as the starting point for more interesting experiments, and that indeed was the initial motivation for writing it. Here are some suggestions of what the program is missing and in what direction it may be extended:

Plasticity models. The most obvious extension is to use a more realistic material model for large-scale quasistatic deformation. The natural choice for this would be plasticity, in which a nonlinear relationship between stress and strain replaces equation (**??**). Plasticity models are usually rather complicated to program since the stress-strain dependence is generally non-smooth. The material can be thought of being able to withstand only a maximal stress (the yield stress) after which it starts to "flow". A mathematical description to this can be given in the form of a variational inequality, which alternatively can be treated as minimizing the elastic energy

$$E(\mathbf{u}) = (\varepsilon(\mathbf{u}), C\varepsilon(\mathbf{u}))_\Omega - (\mathbf{f}, \mathbf{u})_\Omega - (\mathbf{b}, \mathbf{u})_{\Gamma_N}, \tag{24}$$

subject to the constraint

$$f(\sigma(\mathbf{u})) \leq 0 \tag{25}$$

on the stress. This extension makes the problem to be solved in each time step nonlinear, so we need another loop within each time step.

Without going into further details of this model, we refer to the excellent book by Simo and Hughes on "Computational Inelasticity" for a comprehensive overview of computational strategies for solving plastic models. Alternatively, a brief but concise description of an algorithm for plasticity is given in an article by S. Commend, A. Truty, and Th. Zimmermann, titled "Stabilized finite elements applied to elastoplasticity: I. Mixed displacement-pressure formulation" (Computer Methods in Applied Mechanics and Engineering, vol. 193, pp. 3559–3586, 2004).

Stabilization issues. The formulation we have chosen, i.e. using piecewise (bi-, tri-)linear elements for all components of the displacement vector, and treating the stress as a variable dependent on the displacement is appropriate for most materials. However, this so-called displacement-based formulation becomes unstable and exhibits spurious modes for incompressible or nearly-incompressible materials. While fluids are usually not elastic (in most cases, the stress depends on velocity gradients, not displacement gradients, although there are exceptions such as electro-rheologic fluids), there are a few solids that are nearly incompressible, for example rubber. Another case is that many plasticity models ultimately let the material become incompressible, although this is outside the scope of the present program.

Incompressibility is characterized by Poisson's ratio

$$\nu = \frac{\lambda}{2(\lambda + \mu)},$$

where λ, μ are the Lamé constants of the material. Physical constraints indicate that $-1 \leq \nu \leq \frac{1}{2}$. If ν approaches $\frac{1}{2}$, then the material becomes incompressible. In that case, pure displacement-based formulations are no longer appropriate for the solution of such problems, and stabilization techniques have to be employed for a stable and accurate solution. The book and paper cited above give indications as to how to do this, but there is also a large volume of literature on this subject; a good start to get an overview of the topic can be found in the references of the paper by H.-Y. Duan and Q. Lin on "Mixed finite elements of least-squares type for elasticity" (Computer Methods in Applied Mechanics and Engineering, vol. 194, pp. 1093–1112, 2005).

Refinement during timesteps. In the present form, the program only refines the initial mesh a number of times, but then never again. For any kind of realistic simulation, one would want to extend this so that the mesh is refined and coarsened every few time steps instead. This is not hard to do, in fact, but has been left for future tutorial programs or as an exercise, if you wish. The main complication one has to overcome is that one has to transfer the data that is stored in the quadrature points of the cells of the old mesh to the new mesh, preferably by some sort of projection scheme. This is only slightly messy in the sequential case. However, it becomes complicated once we run the program in parallel, since then each process only stores this data for the cells it owned on the old mesh, and it may need to know the values of the quadrature point data on other cells if the corresponding cells on the new mesh are assigned to this process after subdividing the new mesh. A global communication of these data elements is therefore necessary, making the entire process a little more unpleasant.

Ensuring mesh regularity. At present, the program makes no attempt to make sure that a cell, after moving its vertices at the end of the time step, still has a valid geometry (i.e. that its Jacobian determinant is positive and bounded away from zero everywhere). It is, in fact, not very hard to set boundary values and forcing terms in such a way that one gets distorted and inverted cells rather quickly. Certainly, in some cases of large deformation, this is unavoidable with a mesh of finite mesh size, but in some other cases this should be preventable by appropriate mesh refinement and/or a reduction of the time step size. The program does not do that, but a more sophisticated version definitely should employ some sort of heuristic defining what amount of deformation of cells is acceptable, and what isn't.

Compiling the program

Finally, just to remind everyone: the program runs in 3d (see the definition of the `elastic_problem` variable in `main()`, unlike almost all of the other example programs. While the compiler doesn't care what dimension it compiles for, the linker has to know which library to link with. And as explained in other places, this requires slight changes to the Makefile compared to the other tutorial programs. In particular, everywhere where the 2d versions of libraries are mentioned, one needs to change this to 3d, although this is already done in the distributed version of the Makefile. Conversely, if you want to run the program in 2d (after making the necessary changes to accommodate for a 2d geometry), you have to change the Makefile back to allow for 2d.