Assembling matrices in `deal.II`

Wolfgang Bangerth
ETH Zürich, Switzerland

May 2002

1 Introduction

Assembling the system matrix for finite element discretizations is standard, at least as far as scalar problems are concerned. However, things become a little more complicated in implementations once problems are vector-valued, and in particular if finite elements are used in which different components of vector-valued shape functions are coupled, such as for example for divergence-free elements.

It is this case that we are interested in in this report, and we will discuss the implementational details user programs must follow if they want to use such elements with `deal.II`. In order to explain the problem in a simple way, we start by reviewing the algorithms that are used, first for the scalar case, then for the case of "simple" vector-valued finite elements, and finally for the most general case. However, we do not intend to give an introduction into the derivation of finite element methods, or of posing a partial differential equation in weak form.

The interface for vector-valued finite element shape functions with more than one non-zero component that is described in this report is presently being implemented, and will be merged with the library after version 3.4 is released. Thus, it will be part of version 3.5 or 4.0 of the library, depending on which version number we will assign to the successor of 3.4. By then, the library will also contain an implementation of Nedelec elements, for which these techniques are necessary. The interface for primitive vector-valued shape functions, for which only one vector component is non-zero, has been part of the library since its publication with version 3.0.

2 Linear systems for finite element methods

We start by briefly introducing the way finite element matrices are assembled "on paper". As usual in finite elements, we take the weak form of the partial differential equation. In the most general case, it reads: *find* $u \in V$ *such that*

$$a(u,v) = (f,v)_\Omega \qquad \forall v \in V,$$

where $a(\cdot,\cdot)$ is the bilinear form associated with the partial differential equations, and V is the space of test functions. For simplicity, we have here assumed that the problem is linear and that then $a(\cdot,\cdot)$ is a bilinear form; if the problem is nonlinear, it is usually solved using a sequence of linear problems, so this is no restriction.

In finite elements, we define an approximation of the solution u by choosing a finite dimensional subspace V_h spanned by the basis functions $\{\varphi_i\}$, and searching $u_h \in V_h$ by testing the weak form by the test functions from V_h. The problem then reads: *find* $u_h \in V_h$ *such that*

$$a(u_h,v_h) = (f,v_h)_\Omega \qquad \forall v_h \in V_h.$$

Now, $\{\varphi_i\}$ is a basis of V_h. We denote the dimension of V_h by N, and will henceforth let all sums be over the range $0 \dots N-1$, to keep with the standard notation of the C/C++ programming languages. With this, we can expand the

solution $u_h = \sum_{j=0}^{N-1} U_j \varphi_j$, and by bilinearity of the form $a(\cdot,\cdot)$, the problem above is equivalent to

$$\sum_{j=0}^{N-1} U_j a(\varphi_j, \varphi_i) = (f, \varphi_i) \qquad \forall i = 0 \dots N-1. \tag{1}$$

Denoting

$$A_{ij} = a(\varphi_j, \varphi_i), \qquad F_j = (f, \varphi_j),$$

the equations determining the expansion coefficients U_i are therefore:

$$AU = F. \tag{2}$$

Note that we have taken a reverted order of indices in the definition of A, since we want the linear system (2) with the solution to the right of the matrix, to keep with standard notation, instead of to the left as in (1). For symmetric problems, there is no difference, but for non-symmetric ones this is a common source for problems and a rather common trap.

For partial differential equations, the bilinear form used in (1) involves an integral over the domain Ω on which the problem is posed. For example, for the Laplace equation we have

$$A_{ij} = a(\varphi_j, \varphi_i) = (\nabla\varphi_j, \nabla\varphi_i)_\Omega = \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i \, dx.$$

For practical purposes, we split this equation into integrals over the individual cells K of the triangulation $\mathbb{T}$ we use for the discretization. In `deal.II`, these cells are always lines, quadrilaterals, or hexahedra. With this, we have that

$$A = \sum_{K\in\mathbb{T}} A^K, \qquad A^K_{ij} = a_K(\varphi_j, \varphi_i) \qquad 0 \le i,j \le N-1,$$

where the bilinear form $a_K(\cdot,\cdot)$ only involves an integral over the cell K. The important point is that we do so since for the localized basis functions used in finite elements, A^K is a matrix with almost only zeros. The only elements which are not zero are those corresponding to indices i,j indicating those shape functions that have support also on the element K. For example, in 2d and using the usual bilinear shape functions for a scalar problem, only the four shape functions associated with the vertices of the cell K are nonzero on K, and thus only the entries in A^K are nonzero where the four rows corresponding to these indices and the respective four columns intersect.

In general, assume that there are N_K shape functions with support on cell K, and let the set of their indices be denoted by I_K. Then we can define a matrix $\hat{A}^K$ of (small dimension) $N_K \times N_K$ holding these nonzero entries, and we can obtain back the original contribution A_K to A by the transformation

$$A^K_{ij} = \begin{cases} 0 & \text{if } i \notin I_K \text{ or } j \notin I_K, \\ \hat{A}^K_{local(i),local(j)} & \text{otherwise,} \end{cases} \qquad 0 \le i,j \le N-1.$$

Here, $local(i)$ gives the number of the global degree of freedom i on the cell K, i.e. the position of i in the index set I_K. One could call $\hat{A}^K$ the `reduced` form of A^K, since the many zero rows and column have been stripped.

In general, when assembling the global matrix, the reverse way is used: when adding up A^K to A, we do so only with $\hat{A}^K$ by

$$A_{global(i),global(j)} \mathrel{+}= \hat{A}^K_{ij} \qquad 0 \le i,j \le N_K - 1.$$

Thus, indices only run over the (small) range $0 \dots N_K - 1$ instead of $0 \dots N - 1$. Here, $global(i)$ denotes the global number of the degree of freedom with number i on this cell K, i.e. $global(i) = I_K[i]$, where the bracket operator returns the ith element of the set I_K.

The main part of assembling finite element matrices is therefore to assemble the local matrix $\hat{A}^K$. Before we go on with discussing how this is done in `deal.II`, we would like to comment on the evaluation of the integrals involved. Since the integrals are usually too complex to be evaluated exactly (they may depend on coefficients appearing in the equation, or the solution of previous steps in nonlinear or time-dependent problems), they are approximated by quadrature. Assume we have a quadrature formula with N_q points x_q defined on cell in real space (as opposed to the unit cell) and weights w_q. Then, for example for the Laplace equation, we approximate

$$\hat{A}^K_{ij} \equiv \int_K \nabla\varphi_i \cdot \nabla\varphi_j \, dx \approx \sum_{q=0}^{N_q-1} \nabla\varphi_i(x_q) \cdot \nabla\varphi_j(x_q) \; w_q |\det J(\hat{x}_q)|. \tag{3}$$

For other problems, the integrand is different, but the principle remains the same. $\det J(\hat{x}_q)$ denotes the determinant of the Jacobian of the transformation between the unit cell on which the quadrature weights are defined, and the real cell, and $\hat{x}_q$ is the point on the unit cell corresponding to the quadrature point x_q in real space.

Since all matrices and right hand side vectors only require knowledge of the values and gradients of shape functions at quadrature points, this is all that `deal.II` usually provides. One can see this as a kind of *view* on a finite element, as it only provides a certain perspective on the actual definition of a shape function. Nevertheless, this is entirely sufficient for all purposes of programming finite element programs.

In `deal.II` the `FEValues` class does this: you give it a finite element definition, a quadrature formula object, and an object defining the transformation between unit and real cell, and it provides you with the values, gradient, and second derivatives of shape functions at the quadrature points. It also gives access to the determinant of the Jacobian, although only multiplied with w_q as these two are always used in conjunction. It also provides you with many other fields, such as normal vectors to the outer boundary. In practice you do not need them all computed on each cell; thus, you have to specify explicitly in which data you are interested when constructing `FEValues` objects.

In the following, we provide a list of connections between the symbols introduced above, and the respective functions and variable names used in typical `deal.II` programs. With this, we will subsequently show the basic structure of an assembly routine. If you have already taken a look at the example programs provided with `deal.II`, you will recognize all these names. If you haven't, this would be a good time to look at the first three of them.

A	`system_matrix`
$\hat{A}^K$	`cell_matrix`
K	`cell`
N	`dof_handler.n_dofs()`
N_K	`fe.dofs_per_cell`
I_K	`local_dof_indices`
N_q	`quadrature_formula.n_quadrature_points`
$\varphi_i(x_q)$	`fe_values.shape_value(i,q)`
$\nabla\varphi_i(x_q)$	`fe_values.shape_grad(i,q)`
x_q	`fe_values.quadrature_point(q)`
$\|\det J(\hat{x}_q)\|w_q$	`fe_values.JxW(q)`

With this vocabulary, the typical matrix assembly loop in `deal.II` has the following form: first declare a quadrature object and use it for the initialization of a `FEValues` object as discussed above:

```
QGauss2<2>  quadrature_formula;
FEValues<2> fe_values (fe, quadrature_formula,
                       UpdateFlags(update_values    |
                                   update_gradients |
                                   update_JxW_values));
```

In practice, you may want to use a different set of fields to be updated on each cell. For example if you do not need the values of shape functions on a cell, you may omit `update_values` from the list. Also note that by default a bi- or tri-linear (depending on space dimension) mapping between unit and real cell is used. Other mappings are possible, for example quadratic ones, or a mapping that makes use of the fact that in many cases cells are actually rectangular, rather than arbitrary quadrilaterals; in order to use them, another constructor of the `FEValues` class can be used, which takes a mapping object as first argument, before the other arguments listed above.

Next we define abbreviations for the values of N_K and N_q:

```
const unsigned int
    dofs_per_cell = fe.dofs_per_cell,
    n_q_points    = quadrature_formula.n_quadrature_points;
```

Then have an object to store the matrix $\hat{A}^K$, which is of size $N_K \times N_K$:

```
FullMatrix<double>   cell_matrix (dofs_per_cell, dofs_per_cell);
```

And an object representing the set of global indices of degrees of freedom, previously denoted by I_K, that have support on the present cell, i.e. those degrees of freedom local to the present cell:

```
std::vector<unsigned int> local_dof_indices (dofs_per_cell);
```

The next step is then to loop over all cells:

```
typename DoFHandler<dim>::active_cell_iterator
     cell = dof_handler.begin_active(),
     endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
```

On each cell, first tell the `FEValues` object to compute the values of the various fields for this particular cell, and do not forget to reset the local matrix $\hat{A}^K$ to zero before adding it up:

```
    fe_values.reinit (cell);
    cell_matrix.clear ();
```

Now comes the main part, assembling the local matrix $\hat{A}^K$. It consists of a loop over all indices $0 \le i,j \le N_K$ and all quadrature points $0 \le q \le N_q$, and summing up the contributions. As this is what we will discuss in detail later on, we only denote it here by an ellipse:

```
    for (unsigned int i=0; i<dofs_per_cell; ++i)
      for (unsigned int j=0; j<dofs_per_cell; ++j)
        for (unsigned int q=0; q<n_q_points; ++q)
          cell_matrix(i,j) += ...;
```

After we have $\hat{A}^K$, we still have to sum it into the global matrix A. This is done by first getting the set I_K of the global indices of the shape functions that were active on this cell, and then distributing $\hat{A}^K$:

```
      cell->get_dof_indices (local_dof_indices);

      for (unsigned int i=0; i<dofs_per_cell; ++i)
        for (unsigned int j=0; j<dofs_per_cell; ++j)
          system_matrix.add (local_dof_indices[i],
                             local_dof_indices[j],
                             cell_matrix(i,j));
    };
```

When this is done, we go on to the next cell.

Within this framework, the only open point is assembling $\hat{A}^K$ on one quadrature point. This will be subject of the rest of this report.

3 Assembling scalar problems

For scalar problems, assembling $\hat{A}^K$ is relatively simple. With the terms introduced above, and for the Laplace equation, this looks as follows:

```
  local_matrix(i,j) += (fe_values.shape_grad(i,q_point) *
                        fe_values.shape_grad(j,q_point) *
                        fe_values.JxW (q_point));
```

This term is placed in the innermost loop, i.e. is performed for all indices i, j, and all quadrature points `q_point`.

For nonsymmetric problems, the order of terms has to be considered, as mentioned above. For example, for the advection equation

$$\beta \cdot \nabla u = f,$$

with which the bilinear form

$$a(u,v) = (\beta \cdot \nabla u, v)_\Omega$$

is associated, the local matrix is assembled as

```
  local_matrix(i,j) += (fe_values.shape_values(i,q_point) *
                        ( beta * fe_values.shape_grad(j,q_point) ) *
                        fe_values.JxW (q_point));
```

Here, `beta` is an object of type `Tensor<1,dim>`, which represents a tensor of rank 1 in `dim` space dimensions.

Assembling matrices for scalar problems is also shown from a practical perspective in many of the example programs of `deal.II`, where it is also demonstrated how to do this for the right hand side vectors. Thus, scalar problems are not too interesting, and we now turn to vector-valued problems.

4 Vector-valued problems

Since we need some problems at which we will explain assembling the matrix for the vector-valued case, we now briefly introduce two simple equations. The first are the Lamé equations of elasticity, which are taken for the symmetric case, then we briefly introduce the Stokes equations as a nonsymmetric problem.

4.1 The elastic equations

As first example for the methods we are going to discuss for vector-valued problems, we consider the elastic Lamé equations for isotropic materials, which read in strong form:

$$-\nabla(\lambda+\mu)(\nabla\cdot\mathbf{u}) - (\nabla\cdot\mu\nabla)\mathbf{u} = \mathbf{f}.$$

These equations describe the three-dimensional deformation $\mathbf{u}$ of an elastic body under a body force $\mathbf{f}$. The respective bilinear form associated with this operator is then

$$a(\mathbf{u},\mathbf{v}) = ((\lambda+\mu)\nabla\cdot\mathbf{u}, \nabla\cdot\mathbf{v})_\Omega + \sum_k (\mu\nabla u_k, \nabla v_k)_\Omega\,,$$

or as a sum over components:

$$a(\mathbf{u},\mathbf{v}) = \sum_{k,l} ((\lambda+\mu)\partial_l u_l, \partial_k v_k)_\Omega + \sum_{k,l} (\mu\partial_l u_k, \partial_l v_k)_\Omega\,.$$

When assembling matrices, it is advantageous to write the weak form (i.e. after integration by parts) as a kind of operator. This is since matrix elements after discretization are defined as

$$A_{ij} = a(\varphi_j, \varphi_i)_\Omega,$$

where φ_i, φ_j are two vector-valued trial functions, and it would be nice if we could write the bilinear form $a(\cdot,\cdot)$ as a kind of scalar product $(\varphi_i, Q\varphi_j)$, where Q is a differential operator. This is trivial if we take Q as the operator of the strong form, $Q = -\nabla(\lambda+\mu)\nabla\cdot - (\nabla\cdot\mu\nabla)$, but we wanted to do this for the weak form. For this, we introduce some notation that is used in quantum field theory: for differential operators, we indicate by an arrow placed atop of it whether it shall act on the object to the left or to the right of it. Thus, $\varphi\overleftarrow{\nabla}\cdot\overrightarrow{\nabla}\psi = (\nabla\varphi)\cdot(\nabla\psi)$. With this, a simple computation shows that

$$\begin{aligned} a(\mathbf{u},\mathbf{v}) = (\lambda+\mu)\left(\begin{pmatrix}u_1\\u_2\end{pmatrix} \mid \begin{pmatrix}\overleftarrow{\partial}_1\overrightarrow{\partial}_1 & \overleftarrow{\partial}_1\overrightarrow{\partial}_2\\ \overleftarrow{\partial}_2\overrightarrow{\partial}_1 & \overleftarrow{\partial}_2\overrightarrow{\partial}_2\end{pmatrix} \mid \begin{pmatrix}v_1\\v_2\end{pmatrix}\right)_\Omega \\ + \mu\left(\begin{pmatrix}u_1\\u_2\end{pmatrix} \mid \left(\overleftarrow{\partial}_1\overrightarrow{\partial}_1 + \overleftarrow{\partial}_2\overrightarrow{\partial}_2\right)\begin{pmatrix}1&0\\0&1\end{pmatrix} \mid \begin{pmatrix}v_1\\v_2\end{pmatrix}\right)_\Omega. \end{aligned}$$

The sought operator Q is then

$$Q = (\lambda+\mu)\begin{pmatrix}\overleftarrow{\partial}_1\overrightarrow{\partial}_1 & \overleftarrow{\partial}_1\overrightarrow{\partial}_2\\ \overleftarrow{\partial}_2\overrightarrow{\partial}_1 & \overleftarrow{\partial}_2\overrightarrow{\partial}_2\end{pmatrix} + \mu\begin{pmatrix}\overleftarrow{\partial}_1\overrightarrow{\partial}_1 + \overleftarrow{\partial}_2\overrightarrow{\partial}_2 & 0\\ 0 & \overleftarrow{\partial}_1\overrightarrow{\partial}_1 + \overleftarrow{\partial}_2\overrightarrow{\partial}_2\end{pmatrix},$$

and $a(\mathbf{u},\mathbf{v}) = (\mathbf{u}|Q|\mathbf{v})$. We demonstrate the fact that Q acts to both sides by placing it in the middle of the scalar product, just as in the bra-ket notation of quantum physics.

The advantages of this formulation will become clear when discussing assembling matrices below. For now, we remark that the symmetry of the weak form is equally apparent from the form of Q as well as from the initial definition of $a(\cdot,\cdot)$.

4.2 The Stokes equations

For a nonsymmetric problem, we take the Stokes equations:

$$\begin{aligned} -\Delta\mathbf{u} + \nabla p &= 0,\\ \text{div }\mathbf{u} &= 0. \end{aligned}$$

We denote by $\mathbf{w} = \{\mathbf{u}, p\}$ the entire solution vector. In the weak form, the above equations read

$$a(\mathbf{u}, p; \mathbf{v}, q) = \nu(\nabla \mathbf{u}_1, \nabla \mathbf{v}_1) + \nu(\nabla \mathbf{u}_2, \nabla \mathbf{v}_2) - (p, \nabla \cdot \mathbf{v}) + (\nabla \cdot \mathbf{u}, q).$$

Since we integrated the gradient term in the first equation by parts, but not the divergence term in the second equation, the problem is now nonsymmetric. If we would have liked, we could have made the problem symmetric again by multiplying the entire second equation by -1, but we don't want to do that for now for illustrational purposes.

Again, we introduce the operator Q for this problem, which after some computations turns out to be

$$Q = \begin{pmatrix} \nu(\overleftarrow{\partial}_1 \overrightarrow{\partial}_1 + \overleftarrow{\partial}_2 \overrightarrow{\partial}_2) & 0 & -\overleftarrow{\partial}_1 \\ 0 & \nu(\overleftarrow{\partial}_1 \overrightarrow{\partial}_1 + \overleftarrow{\partial}_2 \overrightarrow{\partial}_2) & -\overleftarrow{\partial}_2 \\ \overrightarrow{\partial}_1 & \overrightarrow{\partial}_2 & 0 \end{pmatrix}.$$

Again, it is clear from this form that we could have made the operator symmetric by multiplying the last row by -1. Note when checking the symmetry of Q that taking the transpose of such an operator means reverting the directions of the arrows over the operators, and exchanging their order. For example, using the first term, these two steps are $\overleftarrow{\partial}_1 \overrightarrow{\partial}_1 \to \overrightarrow{\partial}_1 \overleftarrow{\partial}_1 \to \overleftarrow{\partial}_1 \overrightarrow{\partial}_1$.

5 Assembling vector-valued problems: The simple case

The simple case in assembling vector-valued problems is when the (also vector-valued) shape functions are chosen such that only one component in the vector is nonzero. This is usually the case, if we choose the shape functions to be the outer product of scalar shape functions, such as independent bilinear ansatz spaces for each component of a finite element space.

In this case, each shape function Φ_i has the representation

$$\Phi_i(\mathbf{x}) = (0, \ldots, 0, \varphi_i(\mathbf{x}), 0, \ldots, 0)^T,$$

where Φ_i is the vector-valued shape function, and φ_i its only non-zero component. Let us denote by $c(i)$ the index of this non-zero component, then Φ_i can also be written as

$$(\Phi_i(\mathbf{x}))_l = \varphi_i(\mathbf{x}) \delta_{c(i),l},$$

with the Kronecker delta function δ_{jk}.

With this simple form, the cell matrix on cell K has a simple form:

$$A_{ij}^K = a_K(\Phi_i, \Phi_j) = (\Phi_i \mid Q \mid \Phi_j)_K = \left(\varphi_i \mid Q_{c(i),c(j)} \mid \varphi_j\right)_K.$$

Thus, in assembling the local cell matrices, we only have to determine the single components in which the two shape functions are non-zero, and pick one element from the matrix Q to assemble each entry of the cell matrix with the help of the values of the functions φ_i at the quadrature points. Here, it comes handy that we have written the operator as a matrix operator Q, since this makes it very clear how shape functions i and j couple: if $Q_{c(i),c(j)}$ is zero, then they do not couple at all for this operator. Otherwise, it is easily visible which derivative acts on which shape function for this combination of shape functions.

In `deal.II`, these two actions mentioned above (getting the non-zero component of a shape function, and the value of this component at a given quadrature point) are done as follows:

- *Determining the non-zero component:* Given the shape function with number i (i.e. its index local to the degrees of freedom on the present cell), its only non-zero component is obtained by the function call

  ```
  const unsigned int nonzero_component_i
    = fe.system_to_component_index(i).first;
  ```

  The `FiniteElement::system_to_component_index` returns a pair of numbers for each index $0 \le i < N_K$, the first of which denotes the only non-zero component of the shape function i. Since for the case described in this section, the individual components of the vector-valued finite element are independent, we consider each component as a set of scalar shape functions; the second number of the pair returned by the function then denotes the index of the shape function φ_i within the shape functions corresponding to this component.

  If, for example, our finite element in use is a $Q^2/Q^2/Q^1$ combination (for example for 2d flow computations: bi-quadratic ansatz functions for the velocities, bi-linear for the pressure), then we have a total of 22 shape functions (9+9+4). For each $0 \le i < 22$, the first part of the pair returned by the function described above, $c(i)$, may then either be 0, 1, or 2, denoting the three possible components of the finite element. If $c(i)$ is either 0 or 1, then the component to which the shape function i belongs is a bi-quadratic one, and the second index is between 0 and 8 (inclusive) as the Q^2 element has 9 shape functions. If $c(i) == 2$, then the second part is between 0 and 3 (inclusive).

- *Getting the value of $\varphi_i(\mathbf{x}_q)$:* Since only one component of Φ_i is non-zero, we can use the same function as before, i.e. `FEValues::shape_value(i,q)`, which in the scalar case returned the value of shape function i at quadrature point q. Likewise, the `FEValues::shape_grad(i,q)` function returns the gradient of this particular component.

  In other words, whether the finite element is scalar or not, the two indicated functions return value and gradient of the only non-zero component of a shape function. If the finite element is scalar, then it is of course clear which component this is (since there $c(i) == 0$ for all valid indices i), in the vector-valued case, it is component $c(i)$.

5.1 The elastic equations

With this, and the definition of the "bi-directional" operator Q in Section 4.1, the local matrix assembly function for the elastic equations would then read as follows:

```
for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
  for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
    for (unsigned int q=0; q<n_q_points; ++q)
      {
        const unsigned int
          comp_i = fe.system_to_component_index(i).first,
          comp_j = fe.system_to_component_index(j).first;

                        // first assemble part with lambda+mu
        cell_matrix(i,j) += ((lambda+mu)                      *
                             fe_values.shape_grad(i,q)[comp_i] *
                             fe_values.shape_grad(j,q)[comp_j] *
                             fe_values.JxW(q));
```

```
                    // then part with mu only
      if (comp_i == comp_j)
        cell_matrix(i,j) += (mu                          *
                             (fe_values.shape_grad(i,q) *
                              fe_values.shape_grad(j,q)  )*
                             fe_values.JxW(q));
    };
```

Note that this code works in any space dimension, not only for `dim==2`. Optimization of this is possible by hoisting the computation of `comp_i` and `comp_j`, denoting $c(i)$ and $c(j)$, respectively, out of the inner loops. Also, if the coefficients are non-constant, they need to be computed at each quadrature point; this may be done using this fragment in the innermost loop:

```
const double
  lambda_value = lambda.value(fe_values.quadrature_point(q)),
  mu_value     = mu.value(fe_values.quadrature_point(q));
```

assuming that `lambda,mu` are variables of classes describing space dependent functions, and which are derived from the `Function<dim>` class.

5.2 The Stokes equations

For the Stokes equation, things are slightly more complicated since the three components denote different quantities, and the operator Q does not have such a simple form, but the case is still simple enough. We present its generalization to an arbitrary number of space dimensions, i.e. assume that there are `dim` velocity variables and one scalar pressure:

```
for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
  for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
    for (unsigned int q=0; q<n_q_points; ++q)
      {
        const unsigned int
          comp_i = fe.system_to_component_index(i).first,
          comp_j = fe.system_to_component_index(j).first;

                        // velocity-velocity coupling?
        if ((comp_i<dim) && (comp_j<dim))
          if (comp_i == comp_j)
            cell_matrix(i,j) += (nu *
                                 (fe_values.shape_grad(i,q) *
                                  fe_values.shape_grad(j,q)  ) *
                                 fe_values.JxW(q));

                        // velocity-pressure coupling
        if ((comp_i<dim) && (comp_j==dim))
          cell_matrix(i,j) += (-fe_values.shape_grad(i,q)[comp_i] *
                                fe_values.shape_value(j,q)        *
                               fe_values.JxW(q));

                        // pressure-velocity coupling
        if ((comp_i==dim) && (comp_j<dim))
          cell_matrix(i,j) += (fe_values.shape_value(i,q) *
                               fe_values.shape_grad(j,q)[comp_j] *
```

```
                                        fe_values.JxW(q));
    };
```

Again, optimization is possible by observing that only one of the outer `if`s in the body can be true, for example using `else` clauses, or `break` statements.

6 Assembling vector-valued problems: The complicated case

The more complicated case is when more than one component of a vector-valued shape function is non-zero, i.e. the representation

$$\Phi_i(\mathbf{x}) = (0, \ldots, 0, \varphi_i(\mathbf{x}), 0, \ldots, 0)^T,$$

does not hold any more. The usual case where this happens is when shape functions have to satisfy certain constraints, such as that they should have zero divergence or curl, or when the normal fluxes at some points, e.g. the face centers, are the degrees of freedom: $\mathbf{n} \cdot \Phi_i(x_a) = \delta_{ia}$. In this case, the individual components of a shape function are no more independent, and thus cannot be chosen such that only one component is non-zero.

What happens in this case? First, the function `FiniteElement::system_to_component_index` does not make much sense any more, since a shape function Φ_i cannot be associated with only one vector component any more. Calling this function for basis functions Φ_i that are not restricted to only one non-zero component will thus yield an exception being thrown.

Second, the functions `FEValues::shape_value` and `FEValues::shape_grad` returning the values and gradients of the only non-zero component of a shape function at a quadrature point cannot work any more, since there are now more than only one non-zero components for some or all values of i. For those shape function for which this holds, you will again get an exception upon calling these functions.

Getting information about shape functions

So how do you find out whether calling these functions is ok or not? In other words, how do you know whether shape function Φ_i has only one non-zero component, or more? For this, there are two functions: `FiniteElement::is_primitive(i)` returns as a `bool` whether the shape function has only one non-zero component. For example, for a $Q^2/Q^2/Q^1$ element, this would be `true` for all 22 shape functions. For a finite element for which every shape function is non-zero in more than one component, it would be `false` for all indices i. It might also be `true` for only some shape functions, for example if the velocity components of the Stokes discretization are done using some more complicated element, but the pressure component with a Q^1, then it would be `true` for the pressure shape functions, but `false` otherwise.

Second, the `FiniteElement::n_nonzero_components(i)` function returns in how many components the ith shape function is non-zero. Again, for the $Q^2/Q^2/Q^1$ combination, this would yield the value 1 for all allowed indices i. For coupled elements, it would be greater than 1.

Third, you may sometimes want to know in which components a certain shape function is non-zero. For this, the `FiniteElement::get_nonzero_components(i)` function is the right thing: it returns a reference to a vector of boolean values, one for each component of the vector-valued finite element, and the values indicate whether the shape function is non-zero for each of them.

Note that if you have the result of `FiniteElement::get_nonzero_components(i)`, then the result of `FiniteElement::n_nonzero_components(i)` is simply the number of `true` values in the array returned by the first function. In the same way,

`FiniteElement::is_primitive(i)` is simply whether `FiniteElement::n_nonzero_components(i)` returned a value other than 1. The functions are thus redundant in some way, but useful nevertheless. Of course, the values of the `FiniteElement::is_primitive(i)` and `FiniteElement::n_nonzero_components(i)` functions are not recomputed every time based on the result of some other function, but are cached once at the time of construction of a finite element object.

Evaluating shape functions

Now, we have seen which functions cannot be called for non-primitive shape functions, and also how to find out about shape functions and whether they are primitive and the like. Yet, we don't have replacements for the functions that cannot be called, so here they are: instead of `FEValues::shape_value` and `FEValues::shape_grad`, call `FEValues::shape_value_component` and `FEValues::shape_grad_component`. These functions take as arguments first the number of the shape function, and second the number of the quadrature point (these are also the arguments of the original functions), but now as additional third argument the vector component.

Of course, these functions can be called on primitive shape functions as well. In that case, the following holds:

- The value of `FEValues::shape_value_component(i,q,c)` is equal to `FEValues::shape_value(i,q)` if and only if the component `c` is equal to `fe.system_to_component_index(i).first`, i.e. if `c` is the only non-zero component of the shape function `i`.
- For all other components `c`, the returned value of `FEValues::shape_value_component(i,q,c)` is zero.

The same of course also holds for `FEValues::shape_grad_component(i,q,c)` and `FEValues::shape_2nd_derivative_component(i,q,c)`.

6.1 The elastic equations

With the above, we can now assemble the matrix for the elastic equation in mostly the same way as before. The difference is that for each shape function, we have to loop over all components. The code then looks like this (compare this with the one in Section 5.1):

```
for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
 for (unsigned int comp_i=0; comp_i<fe.n_components(); ++comp_i)
  for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
   for (unsigned int comp_j=0; comp_j<fe.n_components(); ++comp_j)
    for (unsigned int q=0; q<n_q_points; ++q)
      {
                    // first assemble part with lambda+mu
        cell_matrix(i,j)
           += ((lambda+mu)                          *
               fe_values.shape_grad_component(i,q,comp_i)[comp_i] *
               fe_values.shape_grad_component(j,q,comp_j)[comp_j] *
               fe_values.JxW(q));

                    // then part with mu only
        if (comp_i == comp_j)
          cell_matrix(i,j)
            += (mu                              *
                (fe_values.shape_grad_component(i,q,comp_i) *
```

```
                fe_values.shape_grad_component(j,q,comp_j)  )*
                fe_values.JxW(q));
      };
```

If you dislike this particular order of the loops, you can reorder them as you like, as they are independent.

The code as shown above can be optimized. For example, instead of unconditionally performing the loop over all components of shape functions i of j, we might first ask whether these shape functions are primitive, using `fe.is_primitive(i)`, and use the loop only if the result is false; if, on the other hand, the result is true, we only need to set `comp_i` to the fixed value `fe.system_to_component_index(i).first`, and likewise for shape function j.

Another possibility for optimization would be to ask whether a certain component over which we loop is actually non-zero, or if the shape function is a non-primitive one but happens to be zero in the present component nevertheless. For this, we could replace the loop over `comp_i` by this:

```
  for (unsigned int comp_i=0; comp_i<fe.n_components(); ++comp_i)
    if (fe.get_nonzero_components(i)[comp_i] == true)
```

If the if-clause does not succeed then this component of the shape function is definitely zero, and there will not be a contribution to the matrix anyway, so we can also skip the computations.

6.2 The Stokes equations

Likewise, this is now the code for the Stokes equations:

```
for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
 for (unsigned int comp_i=0; comp_i<fe.n_components(); ++comp_i)
  for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
   for (unsigned int comp_j=0; comp_j<fe.n_components(); ++comp_j)
    for (unsigned int q=0; q<n_q_points; ++q)
      {
                    // velocity-velocity coupling?
        if ((comp_i<dim) && (comp_j<dim))
          if (comp_i == comp_j)
            cell_matrix(i,j)
                += (nu *
                    (fe_values.shape_grad_component(i,q,comp_i) *
                     fe_values.shape_grad_component(j,q,comp_j)  ) *
                    fe_values.JxW(q));

                    // velocity-pressure coupling
        if ((comp_i<dim) && (comp_j==dim))
          cell_matrix(i,j)
              += (-fe_values.shape_grad_component(i,q,comp_i)[comp_i] *
                   fe_values.shape_value_component(j,q,comp_j)        *
                   fe_values.JxW(q));

                    // pressure-velocity coupling
        if ((comp_i==dim) && (comp_j<dim))
          cell_matrix(i,j)
              += (fe_values.shape_value_component(i,q,comp_i) *
                   fe_values.shape_grad_component(j,q,comp_j)[comp_j] *
```

```
                    fe_values.JxW(q));
    };
```

Again, the same optimizations as above are possible. Here, they even seem worthwhile, since it is often the case that the velocity variables are discretized using a non-primitive finite element, while the pressure uses a primitive element. In that case, some shape functions are primitive (namely those discretizing the pressure), and of the non-primitive shape functions (those for the velocity variables) some vector components (the pressure components) are always zero. Thus, both optimizations described above would be useful. We leave the implementation of this to the reader.

7 Conclusions

We have shown how finite element matrices are assembled using the functionality of the `deal.II` library. For the scalar case, and, in the vector-valued case, if the finite element shape functions are such that only one vector component of each shape function is non-zero, assembling is relatively simple. In the other case, when there are shape functions with more than one non-zero component, some more care is necessary, but assembling is still straightforward and follows the same pattern as before.