Chair for Computational Analysis of Technical Systems

RWTH AACHEN UNIVERSITY

Isogeometric Analysis

Wintersemester 15/16

Exercise Sheet 6
Matrix Assembly Part III: The Actual Assembly

In the last two exercises we were mostly concerned with filling helper arrays that are necessary for what we will now finally implement: the matrix assembly. Before we continue, we have to elaborate more on what kind of problems we want to solve. Remember that generally we are seeking for a $\boldsymbol{u}^h \in \mathcal{S}^h$, such that

$$a(\boldsymbol{u}^h, \boldsymbol{v}^h) = s(\boldsymbol{v}^h) \quad \forall \boldsymbol{v}^h \in \mathcal{V}^h \,, \tag{1}$$

where a is a bilinear form and s a functional. Additionally, we make the following assumptions:

- We restrict ourselves to the *standard Galerkin method*, which means that $\mathcal{S}^h = \mathcal{V}^h$ holds.
- Furthermore, we approximate all degrees of freedom within the same underlying function space, meaning we assume

$$\boldsymbol{u}^h(\xi, \eta) = \sum_{B=1}^{n_{np}} \boldsymbol{u}_B R_B(\xi, \eta) \tag{2}$$

  with $\boldsymbol{u}_B \in \mathbb{R}^{ndof}$ and R_B being the basis functions of the B-Splines/NURBS. **CAUTION:** $\boldsymbol{u}_B$ and $\boldsymbol{u}^h$ are both vectors, e.g. representing the velocity in the case of the advection-diffusion equation.
- Since we are trying to solve the weak form of partial differential equations, a and s necessarily involve an integral over the domain $\overline{\Omega} = \bigcup \overline{\Omega}_i$. Therefore we are able to write

$$a(\boldsymbol{u}^h, \boldsymbol{v}^h) = \sum_i a_i(\boldsymbol{u}^h, \boldsymbol{v}^h) \tag{3}$$

$$s(\boldsymbol{v}^h) = \sum_i s_i(\boldsymbol{v}^h) \,, \tag{4}$$

  where a_i and s_i are linear forms/distributions with support restricted to the element $\overline{\Omega}_i$.

Using the representation in equation (2) for $\boldsymbol{u}^h$ and $\boldsymbol{v}^h$ we can reformulate (1)

$$\sum_{A,B} \boldsymbol{v}_A^T \begin{pmatrix} a(\boldsymbol{e}_1 R_B, \boldsymbol{e}_1 R_A) & \cdots & a(\boldsymbol{e}_{ndof} R_B, \boldsymbol{e}_1 R_A) \\ \vdots & \ddots & \vdots \\ a(\boldsymbol{e}_1 R_B, \boldsymbol{e}_{ndof} R_A) & \cdots & a(\boldsymbol{e}_{ndof} R_B, \boldsymbol{e}_{ndof} R_A) \end{pmatrix} \boldsymbol{u}_B = \sum_A \boldsymbol{v}_A^T \begin{pmatrix} s(\boldsymbol{e}_1 R_A) \\ \vdots \\ s(\boldsymbol{e}_{ndof} R_A) \end{pmatrix} ,$$

which has to hold for all $\boldsymbol{v}_A$.

In combination with formula (3) and (4) this can be rewritten as

$$\sum_{A,B} \sum_i \boldsymbol{v}_A^T \boldsymbol{A}_{A,B}^{(i)} \boldsymbol{u}_B = \sum_A \sum_i \boldsymbol{v}_A^T \boldsymbol{s}_A^{(i)} \quad \forall \boldsymbol{v}_A^h \tag{5}$$

with

$$\boldsymbol{A}^{(i)}_{A,B} = \begin{pmatrix} a_i(\boldsymbol{e}_1 R_B, \boldsymbol{e}_1 R_A) & \cdots & a_i(\boldsymbol{e}_{ndof} R_B, \boldsymbol{e}_1 R_A) \\ \vdots & \ddots & \vdots \\ a_i(\boldsymbol{e}_1 R_B, \boldsymbol{e}_{ndof} R_A) & \cdots & a_i(\boldsymbol{e}_{ndof} R_B, \boldsymbol{e}_{ndof} R_A) \end{pmatrix} \tag{6}$$

$$\boldsymbol{s}^{(i)}_{A} = \begin{pmatrix} s_i(\boldsymbol{e}_1 R_A) \\ \vdots \\ s_i(\boldsymbol{e}_{ndof} R_A) \end{pmatrix}. \tag{7}$$

Drawing the same conclusions as in standard FEM, we can deduce:

- The variational form of our problem in (5) only holds if and only if

$$\sum_B \sum_i \boldsymbol{A}^{(i)}_{A,B} \boldsymbol{u}_B = \sum_i \boldsymbol{s}^{(i)}_A \quad \forall A = 1 \dots n_{np}\,. \tag{8}$$

- All $\boldsymbol{u}^h_B$,$\boldsymbol{A}^{(i)}_{A,B}$ and $\boldsymbol{s}^{(i)}_A$ can be packed together respectively

$$\begin{aligned} \boldsymbol{u} &= \left(\boldsymbol{u}_1, \dots, \boldsymbol{u}_{n_{np}}\right)^T \\ \boldsymbol{s} &= \sum_i \left(\boldsymbol{s}^{(i)}_1, \dots, \boldsymbol{s}^{(i)}_{n_{np}}\right)^T \\ \boldsymbol{A} &= \sum_i \begin{pmatrix} \boldsymbol{A}^{(i)}_{1,1} & \cdots & \boldsymbol{A}^{(i)}_{1,n_{np}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{A}^{(i)}_{n_{np},1} & \cdots & \boldsymbol{A}^{(i)}_{n_{np},n_{np}} \end{pmatrix}, \end{aligned}$$

  such that our problem (1) can finally be cast into a matrix equation

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{s}\,. \tag{9}$$

- For a given $\overline{\Omega}_i$ most of the $\boldsymbol{A}^{(i)}_{A,B}$ are zero. Hence, we have only to consider those indices A, B such that $\boldsymbol{A}^{(i)}_{A,B} \neq \boldsymbol{0}$, which is only the case if R_A and R_B are non-zero on $\overline{\Omega}_i$. In our code these non-zero trial/weighting/basis functions are numbered consecutively from `ishp`$=1, \dots,$`nshp`. For adding up these element-level matrices to the final matrix $\boldsymbol{A}$ one additionally needs the mapping from the local index `ishp` to the global index A, which is one part of what the `connectivity` array provides. The difference to a standard connectivity array here is that `connectivity` maps directly to the equation number, which makes it necessary to also include the number of degrees of freedom. More specifically, `connectivity(1:ndof,ishp)` maps to the equation numbers of our DOFs that correspond to the local shape function `ishp`.

From the implementational view it is now important to note that we so far have not specified which partial differential equation we want to solve, but we can say that the problem-specific parts of the assembly are limited to the computations of $\boldsymbol{A}^{(i)}_{A,B}$ and $\boldsymbol{s}^{(i)}_A$. In order to reduce the code bloat it is now convenient to split off this problem-dependent part of the assembly routine, which can be achieved by using the so-called *visitor pattern*. Generally speaking the *visitor pattern* is an object-oriented construct, where a function is passed as an argument to another function, which will then apply the first function to elements of a given data structure (the function is "visiting" the elements). In our code this concept is incorporated in a sense

that we pass two functions `element_matrix` and `rhs_function` to the `assemble_matrix_2d` routine, which we will then call inside the assembly routine to calculate $\boldsymbol{A}_{A,B}^{(i)}$ resp. $\boldsymbol{s}_A^{(i)}$. Therefore, although we have not yet implemented any of these functions, we need to agree on a function declaration:

```
element_matrix(ndof, quad_points, quad_weight, jac_det, F, U, dU, V, dV)
rhs_function(ndof, quad_points, quad_weight, jac_det, F, V, dV)
```

Here, it is implicitly assumed that our weak form of the partial differential equation includes at most first derivatives. Apart from the scalar `ndof` argument, all arguments are arrays:

- `quad_weight, jac_det` are arrays of size `nquad`.
- `quad_points, F` are arrays of size 2×`nquad`.
- `U,V` are arrays of size `nquad`, storing the values of R_B resp. R_A evaluated at the quadrature points.
- `dU,dV` are arrays of size 2×`nquad`, storing the derivatives of R_B resp. R_A with respect to the physical coordinates.

The return values are in the case of `element_matrix` a matrix of size `ndof`×`ndof` and a vector of size `ndof` in the case of `rhs_function`.

Looking at the function declarations above it should become clear that we have computed all these arrays already on a per element basis in the `assemble_matrix_2d` function, where `U,V` and `dU,dV` refer to different entries in `S` resp. `dS_phys`. The only part that is is missing in the routine is the computation of the $(\texttt{nshp})^2$ non-vanishing matrices $\boldsymbol{A}_{A,B}^{(i)}$ for a given element i, which then has to be added to the full matrix $\boldsymbol{A}$. The same has to be done for the `nshp` non-vanishing vectors $\boldsymbol{s}_A^{(i)}$. The final code should look similar to the following snippet:

```
function [mat rhs] = assemble_matrix_2d(use_bspline, ndof, nurb, ...
    element_matrix, rhs_function)

assert(isa(element_matrix, 'function_handle'));
assert(isa(rhs_function, 'function_handle'));
assert(isa(use_bspline, 'logical'));

% part of Exercise 4
...
% part of Exercise 5


        for ishp=1:nshp
            for jshp=1:nshp
                % call element_matrix
                matloc = YOUR_CODE;

                % add matloc to mat. Be aware of the connectivity array.
                mat(YOUR_CODE,YOUR_CODE) = ...
                    mat(YOUR_CODE,YOUR_CODE) + matloc;
            end
            % call rhs_function
            rhsloc = YOUR_CODE;
            % add rhsloc to rhs
```

```
            rhs(YOUR_CODE) = rhs(YOUR_CODE) + rhsloc;
        end
    end
end

end
```

In the L^2P you will also find a function `nurb_knot_refinement` that uses the `assemble_matrix_2d` routine to perform a h-refinement of a NURBS by solving a least-squares problem. Running e.g.

```
nurb = nurb_knot_refinement (generate_testnurb(), 10);
```

should result in a new NURBS with 10 knots inserted in each knot span. Plotting this new NURBS structure gives you a simple test at hand to see whether the assembly routine works or not.

Contact:
Florian Zwicke, M.Sc. · zwicke@cats.rwth-aachen.de