Isogeometric Analysis

Wintersemester 15/16

Exercise Sheet 5
Matrix Assembly Part II: Connectivity Array

In the last exercise we started implementing the quadrature part of the matrix assembly routine. Now we get to the part, where the B-Spline/NURBS basis functions are used as an analysis tool. More specifically, this means that we describe the discretized solution $\boldsymbol{u}^h$ of our problem in a finite dimensional space spanned by the B-Spline/NURBS basis functions N_A

$$\boldsymbol{u}^h(\xi,\eta) = \sum_{i=1}^{n_1}\sum_{j=1}^{n_2} \boldsymbol{u}_{i,j} N_{i,j}(\xi,\eta)\,. \tag{1}$$

Furthermore, `ndof`$=\dim \boldsymbol{u}_h$ denotes the degrees of freedom per basis function. As in normal FEM the assembly is performed element-wise and therefore we will only need to evaluate formula (1) within a given element. Let i_0,j_0 specify the return values of `bspline_findspan` that correspond to this element, such that we can drop the vanishing part of the sum and shift the indices

$$\boldsymbol{u}^h(\xi,\eta) = \sum_{i=1}^{p+1}\sum_{j=1}^{q+1} \boldsymbol{u}_{i_0-p+i-1,j_0-q+j-1} N_{i_0-p+i-1,j_0-q+j-1}(\xi,\eta)\,. \tag{2}$$

From an implementational point of view we now have to ponder on two things:

- We need to evaluate the `nshp`$=(p+1)\cdot(q+1)$ shape functions $N_{i_0-p+i-1,j_0-q+j-1}$ and their derivatives at the quadrature points. Here it once again becomes handy, that the B-Spline/NURBS evaluation functions take the coefficients as a separate argument. Therefore we only need to fill a `coeffs(1:nshp,1:nurb.number(1),1:nurb.number(2))` array and pass it to the existing functions. We will count the non-vanishing shape functions from the bottom left to the upper right corner. To be more specific, `coeffs((j-1)*(p+1)+i,:,:)` should be chosen such that the evaluation routine returns $N_{i_0-p+i-1,j_0-q+j-1}$. *Hint:* The `coeffs`-array entries will be either 0 or 1.
- In order to be able to cast the final system into a matrix form it is necessary to pack all $\boldsymbol{u}_{i,j}$ into a single vector. For this purpose we will introduce a connectivity array `connectivity(1:ndof,1:nshp)`, which assigns to every local degree of freedom the equation number

$$\begin{aligned}&\texttt{connectivity(k,(j-1)*(p+1)+i)}\\ &\quad= \texttt{ndof}\cdot\big((j_0-q+j-2)\cdot\texttt{nurb.number(1)}+i_0-p+i-2\big)+k\,.\end{aligned}$$

  At this point it should be noted that we assemble the matrix regardless of which degrees of freedom are fixed by a Dirichlet boundary condition.

That said, we have to clarify one last point: The functions `nurb_derv_eval` and `bspline_derv_eval` return the derivatives with respect to the parameters ξ, η. Although we want to evaluate the derivatives at certain quadrature points in parameter space, the partial differential equations contain derivatives with respect to the physical coordinates. Consequently we need to transform the derivatives from the parametric to the physical domain. Let $\boldsymbol{F}(\xi, \eta)$ denote the geometrical mapping, then the preceding statement in a mathematical exact form means that the quantities of interest are

$$\underbrace{\nabla_{x,y} N_{i,j}|_{\xi,\eta}}_{=\texttt{dS\_phys}} \overset{\text{def}}{=} \nabla_{x,y}\left(N_{i,j}\left(\boldsymbol{F}^{-1}(x,y)\right)\right)\Big|_{\boldsymbol{x}=\boldsymbol{F}(\xi,\eta)},$$

which can be computed with the chain rule (same procedure as in FEM)

$$\nabla_{x,y}\left(N_{i,j}\left(\boldsymbol{F}^{-1}(x,y)\right)\right)\Big|_{\boldsymbol{x}=\boldsymbol{F}(\xi,\eta)} = \left(\boldsymbol{JF}|_{\xi,\eta}\right)^{-T} \underbrace{\nabla_{\xi,\eta} N_{i,j}}_{=\texttt{dS\_ref}} . \tag{3}$$

Remember that $\boldsymbol{JF}$ has already been computed in the last exercise and stored in the array `dF`. To test the implementation we check for the partition of unity feature at each quadrature point.
Your task is to append the following code to your assembly routine and incorporate the aforementioned considerations to fill the missing gaps. As in the last exercises the code template will also be available in the L^2P.

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

% preamble and quadrature part of the code
...


        % evaluate nurb basis functions and their derivatives
        coeffs = zeros(nshp, nurb.number(1), nurb.number(2));
        p = nurb.order(1) - 1;
        q = nurb.order(2) - 1;
        i0 = YOUR_CODE;
        j0 = YOUR_CODE;
        connectivity = zeros(ndof,nshp);
        % assemble coeffs and connectivity
        YOUR_CODE

        if (use_bspline)
            % use B-Splines basis functions
            S = YOUR_CODE;
            % derivatives in the reference domain
            dS_ref = YOUR_CODE;
        else
            % use NURBS basis functions

            % remember that nurb_(derv_)eval needs the coefficients
            % premultiplied with the weight
            % Remark: This step is only introduced to allow to test for
            % partition of unity of the NURBS basis functions. From an
            % algebraic point of view it does not matter whether we
            % premultiply the coefficients, it only introduces overhead in
            % the sense that the solution of the linear equation system
            % also needs to be mutiplied again with the weights before
            % plotting.
            for i=1:nshp
                coeffs(i,:,:) = coeffs(i,:,:) .* nurb.coeffs(4,:,:);
            end
            S = YOUR_CODE;
            % derivatives in the reference domain
            dS_ref = YOUR_CODE;
        end
        dS_phys = zeros(nshp,2,nquad);

        % derivatives in the physical domain
        for i=1:nquad
            for j=1:nshp
                % compute dS_phys with the help of equation (3)
                % Use the '/' operator to compute the left inverse of the jacobian.
                YOUR_CODE
            end
        end

        % check partition of unity
        testSum = sum(S,1);
        testSumDervRef = sum(dS_ref,1);
        testSumDervPhys = sum(dS_phys,1);
        for i=1:nquad
            assert(abs(testSum(1,i) - 1.0) < 10^-10);
            for j=1:2
                assert(abs(testSumDervRef(1,j,i)) < 10^-10);
                assert(abs(testSumDervPhys(1,j,i)) < 10^-10);
            end
        end

...

end
```

Contact:

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