Chair for Computational Analysis of Technical Systems

RWTH AACHEN UNIVERSITY

Isogeometric Analysis

Wintersemester 15/16

Exercise Sheet 4
Matrix Assembly Part I: Quadrature

In this exercise we will start with the implementation of the matrix assembly. Since it is a quite difficult task to write this routine in a whole, it will be split up and distributed over the next exercises. A visualization of what will be done can be seen in Figure 1.

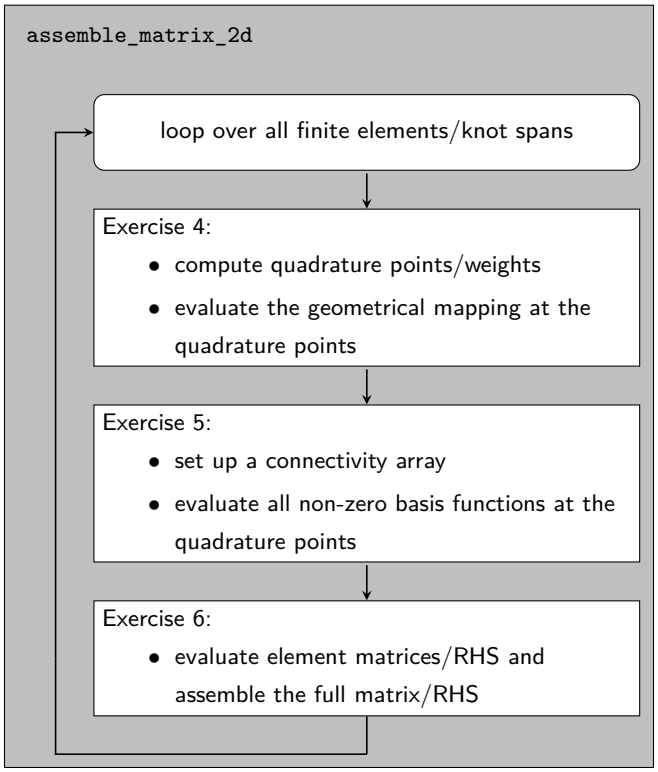

Figure 1: Overview over the next exercises.

The specific objective of this exercise is numerical quadrature on a NURBS surface. The main reason for using quadrature rules is, as in most other FEM applications, that it is too difficult to evaluate the integrals analytically. It might even be that the integral is not expressible in a closed analytical form (e.g. the area integral of an ellipse).

Starting point is the Gaussian quadrature in one dimension. It can be expressed with $w_i \in \mathbb{R}$ and $x_i \in [-1, 1]$, such that

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i \cdot f(x_i) \quad \forall f \in C^0([-1,1])\,. \tag{1}$$

Using Fubini's theorem one can readily deduce a quadrature rule for a two-dimensional rectangle

$$\int_{[-1,1]^2} f(x,y)\,dx\,dy \approx \sum_{i=1}^{n}\sum_{j=1}^{n} w_i \cdot w_j \cdot f(x_i, x_j)\,, \tag{2}$$

which can be recast into the form of equation (1)

$$\begin{aligned}\int_{[-1,1]^2} f(x,y)\,dx\,dy &\approx \sum_{k=1}^{n^2} \tilde{w}_k f(\tilde{\boldsymbol{x}}_k)\\ \tilde{w}_k &= w_{\lfloor (k-1)/n \rfloor + 1} \cdot w_{(k-1) \bmod n+1}\\ \tilde{\boldsymbol{x}}_k &= \begin{pmatrix} x_{\lfloor (k-1)/n \rfloor + 1} \\ x_{(k-1) \bmod n+1} \end{pmatrix}.\end{aligned}$$

Although the last equation with the integer division and the modulo operation looks quite difficult, it can be implemented in a quite simple way as the following table for $n = 3$ exemplifies:

k	1	2	3	4	5	6	7	8	9
$\tilde{x}_{k,1} =$	x_1	x_1	x_1	x_2	x_2	x_2	x_3	x_3	x_3
$\tilde{x}_{k,2} =$	x_1	x_2	x_3	x_1	x_2	x_3	x_1	x_2	x_3
$\tilde{w}_k =$	w_1^2	$w_1 w_2$	$w_1 w_3$	$w_2 w_1$	w_2^2	$w_2 w_3$	$w_3 w_1$	$w_3 w_2$	w_3^2

Now we additionally need to use the transformation formula for integrals in order to define a quadrature rule on the reference domain $[\xi_1, \xi_2] \times [\eta_1, \eta_2]$ of our finite element. For convenience we reuse the symbol w_i:

$$\int_{\xi_1}^{\xi_2}\int_{\eta_1}^{\eta_2} f(\xi,\eta)\,d\xi\,d\eta \approx \sum_{i=1}^{n_{quad}} w_i f(\boldsymbol{u}_i) \tag{3}$$

$$w_i = \frac{\tilde{w}_i}{4} \cdot \underbrace{(\xi_2 - \xi_1)\cdot(\eta_2 - \eta_1)}_{=\texttt{area}} \tag{4}$$

$$\boldsymbol{u}_i = \frac{1}{2}\begin{pmatrix} \xi_2 - \xi_1 & 0 \\ 0 & \eta_2 - \eta_1 \end{pmatrix} \cdot \tilde{\boldsymbol{x}}_i + \frac{1}{2}\begin{pmatrix} \xi_2 + \xi_1 \\ \eta_2 + \eta_1 \end{pmatrix} \tag{5}$$

Any function defined in physical coordinates can now be integrated by evaluating the NURBS function $\boldsymbol{F}$ and its derivatives at the quadrature points $\boldsymbol{u}_i$

$$\int_{\xi_1}^{\xi_2}\int_{\eta_1}^{\eta_2} f(\boldsymbol{F}(\xi,\eta)) \cdot |\det J\boldsymbol{F}(\xi,\eta)|\,d\xi\,d\eta \approx \sum_{i=1}^{n_{quad}} w_i \cdot |\det J\boldsymbol{F}(\boldsymbol{u}_i)| \cdot f(\boldsymbol{F}(\boldsymbol{u}_i))\,. \tag{6}$$

From the implementational point of view we will compute the following arrays for each finite element:

- `quad_weight(1,1:nquad)` holds the weights w_i, which means `quad_weight(1,i)`$= w_i$.
- `quad_points(1:2,1:nquad)` holds the quadrature points in the reference domain $\boldsymbol{u}_i$. More specifically: `quad_points(:,i)`$= \boldsymbol{u}_i$
- `F(1:2,1:nquad)` holds the evaluation of the NURBS function at the quadrature points.
- `dF(1:2,1:2,1:nquad)` holds the values of the NURBS function derivatives at the quadrature points. *Hint:* This implies `dimS`$= 2$. Additionally `dF(:,:,i)` specifies the Jacobian corresponding to the i-th quadrature point.

- `jac_det(1,1:nquad)` holds the absolute value of the Jacobian's determinant.

All this will be embedded into the assemble routine `assemble_matrix_2d`, which will have the following design, where for testing purposes we once again compute the area of our Test-NURBS:

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

% don't worry about this for the moment
%assert(isa(element_matrix, 'function_handle'));
%assert(isa(use_bspline, 'logical'));

% number of shape functions per element
nshp = nurb.order(1) * nurb.order(2);
alldof = ndof*nurb.number(1)*nurb.number(2);
mat = spalloc(alldof,alldof,alldof*nshp);
rhs = zeros(alldof,1);

%remove double entries in the knot arrays
xknots = unique(nurb.knots{1});
yknots = unique(nurb.knots{2});

[quad_weight_1d, quad_points_1d] = quad_rule();
nquad_1d = size(quad_weight_1d, 2);

nquad = nquad_1d * nquad_1d;
quad_weight = zeros(1,nquad);
quad_points = zeros(2,nquad);

nurb_area = 0.0;

% Loop over all elements, which are spaced from knot to knot
for ie = 1:size(xknots,2)-1
    for je = 1:size(yknots,2)-1
        % build quad rules
        xi1 = xknots(ie);
        eta1 = yknots(je);
        deltaXi = xknots(ie+1) - xi1;
        deltaEta = yknots(je+1) - eta1;
        % area of the element in the reference domain
        area = deltaXi * deltaEta;
        % Compute the weights according to equation (4).
        % It is easier if you use the function 'reshape', which makes
        % it possible to avoid the integer division and the modulo operation.
        quad_weight = YOUR_CODE;
        % Compute the quadrature points according to formula (5).
        % If possible use the functions 'repmat' and 'reshape'.
        quad_points(1,:) = YOUR_CODE;
        quad_points(2,:) = YOUR_CODE;
```

```
        % evaluate the geometry mapping and its derivatives
        F = YOUR_CODE;
        dF = YOUR_CODE;
        jac_det = zeros(1,nquad);
        for i=1:nquad
            % evaluate the absolute value of the Jacobian's determinant
            jac_det(i) = YOUR_CODE;
        end

        % Compute the NURBS' area
        nurb_area = nurb_area + YOUR_CODE;
    end
end

% print nurb_area for testing purposes
nurb_area

end
```

Executing

```
assemble_matrix_2d(false,1,generate_testnurb(),1,1);
```

should result in 2.3564. Remember the result of the Monte Carlo integration on the last exercise sheet, where with 5000 evaluation points the result was not as accurate as with the 9 quadrature points we have used now.

The template for the function `assemble_matrix_2d`, as well as the function `quad_rule`, can be found in the L^2P.

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