Numerical methods for partial differential equations

Programming Exam

by

Enes Witwit and Marcel Duerr

Examined by

Prof. Dr. Kanschat

Faculty for math and computer science

University of Heidelberg

Contents

0.1 Introduction

The programming exam states the following task.

> Write a finite element code in your preferred programming language solving the boundary value problem:
>
> $$\text{Find } u \in C^2(\Omega) : -\Delta u + u = \cos(\pi x)\cos(\pi y) \quad \text{in } \Omega \tag{1}$$
> $$\partial_n u = 0 \quad \text{on } \partial\Omega. \tag{2}$$
>
> Here, $\Omega = [0,1]^2$.

First of all, we want to examine the given BVP. In order to produce meaningful results, we have to proof that a unique solution exists. Furthermore, this document will explain some of the specific formulars used for computation. These formulars depend heavily on the structure of the problem and can not be generalized.

In the second chapter we want to give details and explanation of our implementation. We will discuss relevant functions and give an overview of our programm.

1 Theory

1.1 Weak formulation

In order to apply some of the results from the lecture, we need to derive the weak formulation of the given problem

$$\text{Find } u \in C^2(\Omega) : -\Delta u + u = \cos(\pi x)\cos(\pi y) \qquad \text{in } \Omega \tag{1.1}$$
$$\partial_n u = 0 \qquad \text{on } \partial\Omega. \tag{1.2}$$

Multiplying with an arbitary $v \in C^2(\Omega)$ and integrating over Ω gives us

$$-\int_\Omega \Delta u v \,\mathrm{d}\boldsymbol{x} + \int_\Omega uv \,\mathrm{d}\boldsymbol{x} = \int_\Omega fv \,\mathrm{d}\boldsymbol{x}$$

where $f = \cos(\pi x)\cos(\pi y)$ and $\boldsymbol{x} = (x, y)$. Using Green's first formula and (2) we can obtain the weak formulation

$$\int_\Omega \nabla u \nabla v \,\mathrm{d}\boldsymbol{x} + \int_\Omega uv \,\mathrm{d}\boldsymbol{x} = \int_\Omega fv \,\mathrm{d}\boldsymbol{x}$$

From now on, we will denote the left hand side of the equation by $a(u, v)$ and the right hand side by $F(v)$. Thus, we obtain the weak formulation

$$\text{Find } u \in H^1(\Omega) : \quad a(u, v) = F(v) \quad \forall v \in H^1(\Omega) \tag{1.3}$$

1.2 Existence and uniqueness of a solution

We can already see, that our bilinear form $a(\cdot\,, \cdot)$ is the inner product associated with the norm on our function space $H^1(\Omega)$. We want to use the Riesz representation theorem to prove existence and uniqueness of a solution. In order to do so, it remains to show that our functional $F(\cdot)$ is linear and bounded. Linearity follows from the properties of integration. Using Hoelder's inequality, we show that

$$\begin{aligned} F(u) &= \|fu\|_{L^1(\Omega)} \\ &\overset{\text{Hld.}}{\leq} \|f\|_{L^2(\Omega)} \|u\|_{L^2(\Omega)} \\ &\leq \|1\|_{L^2(\Omega)} \left(\|u\|_{L^2(\Omega)} + \|u\|_{L^2(\Omega)}\right) \\ &\leq c \, \|u\|_{H^1(\Omega)}, \end{aligned}$$

where c depends on our domain Ω. For our case, we have $\Omega = [0, 1]^2$, in particular this means that Ω is bounded and our constant c is finite. Therefore, $F(\cdot)$ is a bounded, linear functional and we can apply the Riesz representation theorem.

1.3 Analytical solution

Now that we know that a unique solution exists, we want to actually compute it. We will use the ansatz $u = C\cos(\pi x)\cos(\pi y)$, with its gradient $\Delta u = 2\pi^2 u$. Inserting in (1) gives us

$$-2C\pi^2\cos(\pi x)\cos(\pi y) + C\cos(\pi x)\cos(\pi y) = \cos(\pi x)\cos(\pi y) \tag{1.4}$$

$$\Rightarrow -2C\pi^2 + C = 1 \tag{1.5}$$

$$\Leftrightarrow C = \frac{1}{1-2\pi^2} \tag{1.6}$$

This leaves us with the solution $u = \frac{1}{1-2\pi^2}\cos(\pi x)\cos(\pi y)$.

1.4 Integrals and transformations

The transformations we have to use for our FEM-code are very simple, since our mesh will be generated uniformly. Ultimately, we will only have to scale and displace our reference cell. We decided, that we will hard-code this property, since it will heavily simplifiy the computation of the local stiffness-matrices and the right-hand-side of the linear system, as we will see in brief.

Let T be a cell of the mesh with vertices v_1, v_2, v_3, v_4, where v_1 is the vertex on the bottom left and h is the length of every edge. $\hat{T} = [0,1]^2$ shall be our reference cell. Then, our transformation F will look like

$$F\colon \hat{T} \to T, \quad \hat{\boldsymbol{x}} = \begin{pmatrix}\hat{x}\\ \hat{y}\end{pmatrix} \mapsto \begin{pmatrix}h & 0\\ 0 & h\end{pmatrix}\begin{pmatrix}\hat{x}\\ \hat{y}\end{pmatrix} + v_1 := \begin{pmatrix}x\\ y\end{pmatrix} := \boldsymbol{x}$$

We immediately can see, that the jacobian J is given by

$$J(\hat{\boldsymbol{x}}) = J = \begin{pmatrix}h & 0\\ 0 & h\end{pmatrix}, \tag{1.7}$$

in particular, it is independet of $\hat{\boldsymbol{x}}$. Consequently, we also get

$$|\det(J(\hat{\boldsymbol{x}}))| = |\det(J)| = h^2 \text{and } J^{-1} = h^{-1}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \tag{1.8}$$

Using the results from the lecture on the transformation of our bilinear form $a(\cdot\,,\cdot)$ and using (7) and (8), we can compute the stiffness matrix in the following way

$$a_{ij}^{(T)} = \int_{\hat{T}} \nabla p_i \cdot \nabla p_j \,\mathrm{d}\hat{\boldsymbol{x}} + h^2 \int_{\hat{T}} p_i p_j \,\mathrm{d}\hat{\boldsymbol{x}},$$

where p_i and p_j are our shape functions.

Remark: Note that the ∇ in our formula refers to the gradient with respect to $\hat{\boldsymbol{x}} = \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix}$. *Remark:* Note that for affine transformations, the local stiffness matrices do not depend on the cell T, for which we want to compute it. Therefore, every local matrix looks the same and we only have to compute it once.

For the right-hand side, we have to look at the substitution formula for higher dimension integrals. But first of all, we need do split the intagral up. Let ϕ_i be a basis function of our domain Ω, $\mathcal{T}_h$ our meshcells and $\mathcal{T}_i$ the mesh cells, on which $\phi_i \neq 0$. Then we get

$$\begin{aligned} \int_\Omega f(\boldsymbol{x})\phi_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} &= \sum_{T \in \mathcal{T}_h} \int_T f(\boldsymbol{x})\phi_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \\ &= \sum_{T \in \mathcal{T}_i} \int_T f(\boldsymbol{x})\phi_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \end{aligned}$$

$$\begin{aligned} &= \sum_{T \in \mathcal{T}_i} \int_T f(\boldsymbol{x}) p_{i^{(T)}}^{(T)}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \\ &= \sum_{T \in \mathcal{T}_i} \int_{\hat{T}} f(F^{(T)}(\hat{\boldsymbol{x}}) \hat{p}_{i^{(T)}}(\hat{\boldsymbol{x}}) |\det(J)| \, \mathrm{d}\hat{\boldsymbol{x}} \\ &= h^2 \sum_{T \in \mathcal{T}_i} \int_{\hat{T}} f(h\hat{\boldsymbol{x}} + v_1^{(T)}) \hat{p}_{i^{(T)}}(\hat{\boldsymbol{x}}) \, \mathrm{d}\hat{\boldsymbol{x}} \end{aligned}$$

1.5 Linear solver

1.6 Error analysis

In this section, we will discuss our expectations of the error and our method to compute the convergence order.

1.6.1 A posteriori

For the estimation of the error we are going to implement an error control method by C. Runge. It seemed very appropriate to use a heuristic rule, which proved itself to be sufficiently good and most importantly easy and cheap for our case.

Heuristic rule of Runge *If the difference between two approximate solutions computed on the mesh with mesh size h and mesh size $h/2$ has become small, then both approximated solutions are probably close to the exact solution*

We are going to use this heuristic rule to compute $|u_h - u_{h/2}|$, which we will view as a certain a posteriori error indicator. Unfortunatley it will not have the property modern a posteriori error indicators have, which is to be a upper bound. Still with empirical investigations it seems that our error indicator is quite well.

1.6.2 A priori

In lecture, we had the following theorem
Theorem 4.29 *Assume that the solution u of the given problem (5) satisfies the regularity condition $u \in V = H^1(\Omega) \cap H^{k+1}(\Omega)$ with $k \geq m$. Then, The error in the solution of the finite element method satisfies the following bound:*

$$\|u - u_h\|_{m,\Omega} \leq ch^{k+1-m}|u|_{k,\Omega},$$

where c is a positive constant and k is the polynomial degree.

We chose V to be a subspace of $H^1(\Omega)$, therefore $m = 1$. The estimate in theorem 4.29 still contains a constant c and can not be used directly. We will revise the computation of a convergence order from last semesters lecture:
Let k be the polynomial degree, assume that the equality holds

$$\|u - u_h\|_{1,\Omega} = ch^{k+1}|u|_{k+1,\Omega} \tag{1.9}$$

$$\|u - u_{\frac{h}{2}}\|_{1,\Omega} = c\frac{h^{k+1}}{2^{k+1}}|u|_{k+1,\Omega} \tag{1.10}$$

Dividing (11) by (12) yields

$$\frac{\|u - u_h\|_{1,\Omega}}{\|u - u_{\frac{h}{2}}\|_{1,\Omega}} = 2^{k+1}$$

This is the convergence we would like to see. Suppose we use polynomials of degree 1. Using half the mesh size should reduce the error by a factor of 4.

2 Implementation

In this chapter, we will recapitulate our code and provide further explanations on selected excerpts of our code

2.1 Mesh

We start of with our mesh generator 'mesh_generate'. Our domain $\Omega = [0,1]^2$ will not be subject to change for this task, therefore our only input will be the mesh size h. The result will be two matrices, namely *vertex_matrix* and *cell_matrix*. *vertex_matrix* will store the coordinates of each vertex, while *cell_matrix* only stores the numbers of the four vertices of each cell.
Example: $h = 0.5$

$$vertex\_matrix = \begin{pmatrix} 0 & 0 \\ 0.5 & 0 \\ 1 & 0 \\ 0 & 0.5 \\ 0.5 & 0.5 \\ 1 & 0.5 \\ 0 & 1 \\ 0.5 & 1 \\ 1 & 1 \end{pmatrix}, \quad cell\_matrix = \begin{pmatrix} 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \end{pmatrix}$$

2.2 Shape functions

An essential concept for finite element methods would be shape functions. We have to set up certain points, so called *nodes* on a reference cell $\hat{T} = [0,1]^2$ such that our shape functions p_i are uniquely defined by two properties:

1. $p_i(x_j) = \delta_{ij},\ x_j \text{node}$
2. p_i is a polynomial on each edge.

A way to achieve this, is to choose the location of the nodes in the same way as our mesh. Afterwards, we choose appropriate base-polynomials. If we want polynomials of degree k on the edges, the appropriate basis would be $\{x^i y^j,\ i,j = 0...k\}$. Although the resulting polynomials are of order $2k$, they are only of order k in each variable and therefore only of order k on each edge. A mesh generated with mesh-size $\frac{1}{k}$ will yield $k+1$ nodes on each edge; the interpolation is well-posed.
Remark: The way we order our basis has to be consistent for the entire code, because we only store the coefficients when we want to store a polynomial.

In particular, this means that one entry of the coefficient vector must refer to the same polynomial basis element, even for different orders of polynomials. Effectively, this means that the basis for a higher order polynomial can only add basis elements at the end of any lower order polynomial basis.

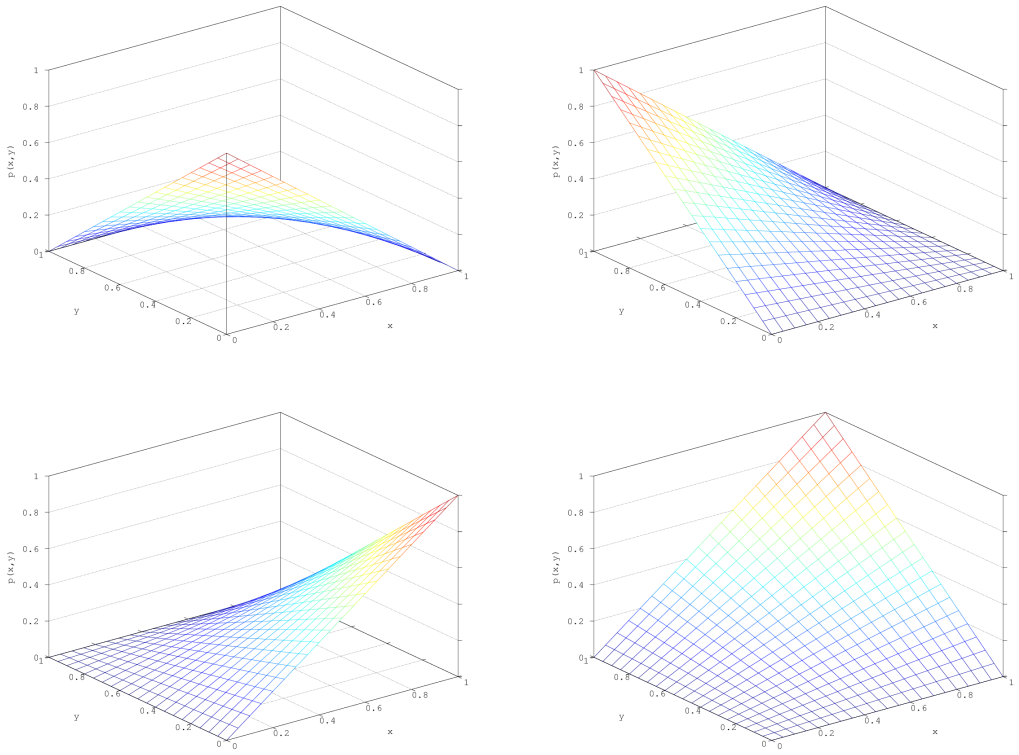
sf_generate follows this scheme closely. Since the shape functions only depend on the polynomial degree k we want to have on the edges, our only input will be the integer k. The function will create the nodes using *mesh_generate* with mesh-size $\frac{1}{k}$. Next, it will create a data-set, containing the coordinates and the value at these coordinates for each node. In total, we get $(k+1)^2 := n$ nodes, their values are computed using the Kronecker Delta. This will ensure, that property 1) holds true when interpolating over all these points. The interpolation is done by solving

$$\begin{pmatrix} 1 & x_0 & \cdots & x_0^k y_0^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^k y_n^k \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} b_0 \\ \vdots \\ b_n \end{pmatrix}$$

The output of our function is the matrix SF, which stores the coefficient vectors of each shape function in its rows. Again, we see that a consistent numbering is needed when we want to evaluate a shape function. Let's look at some shape functions:
Example: Let the polynomial degree on the edges be 1. The coefficient matrix SF will look like this:

$$\begin{pmatrix} 1 & -1 & -1 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The four resulting shape functions:



You can clearly see, that each shape function has the value 1 at one vertex, and 0 at all the other vertices

2.3 Stiffness matrix

Next up is the assembly of the stiffness matrix, this will be the left hand side of our linear system. The original idea behind the stiffness matrix was to build every combination of our basis functions in our bilinear form, namely $A_{ij} = a(\phi_i, \phi_j)$. As seen in the lecture, it is more efficient to compute a *local* stiffness matrix for each cell, and then insert these values in the right way in a global stiffness matrix. This approach proves to be even more efficient, considering that in our case, the local stiffness matrices are identical for each cell (*see also*: Theoretical Construct, Chapter 'Integrals and Transformations').

The computation of the local matrix is done by *sm_assemble_local*. As already shown, the computation will not depend on a specific cell, therefore we only need the mesh-size h and the shape function coefficients SF. Furthermore, we need several help functions which will compute the required

integrals

- *hf_eval_poly* will evaluate a given polynomal at a given point, by using our specific polynomial basis and a coefficient vector
- *sf_derivate* will return the coefficient vectors for the derivative in x and y direction for a given polynomial.
- *int_gauss_weights* will return quadrature points and weigths for a given order and a given quadratic domain.
- *int_gauss* will evaluate a given function at the quadrature points and sum over the product of the quadrature weigths.

Note that the computation of the quadrature points and the actual quadrature is split up into two function. The transformations we use enable us to only integrate over the unit square. Therefore, we only have to compute the quadrature points and weights once. This will save alot of recources.
Having these functions at hand, the local matrix can be easily computed. The symmetric bilinear form will result in a symmetric local matrix, we will only compute on half if the matrix.

The next step will be the assembly of the global stiffness matrix. Let $(a_{ij})^T$ be the local stiffness matrix for a cell T. For any given i and j, the entry in the global stiffness matrix can be obtained by

$$A_{ij} = \sum_{T \in \mathcal{T}_h} a^T_{\rho^{-1}(i)\rho(j)^{-1}}$$

where ρ will map the local node numbering to the global numbering. The most efficient way to implement this, is to iterate over all cells, and add the entries of the local matrix at the right spot.

Key to this operation is the mapping ρ. Our function *mesh_nodes* will handle the renumbering by generating a matrix. The i-th row corresponds to the i-th cell, the j-th entry in this row corresponds to the j-th local node within the cell. In this entry, the matrix will store the global number of the degree of freedom, which corresponds to the j-th local node of the i-th cell. For each iteration, we will select the corresponding entries of the global matrix by using *mesh_nodes* and simply add the local stiffness matrix.

2.4 Right hand side

The linear system is still missing its right hand side. We already discussed the computation of it in the 'Theoretical Construct', Chapter 'Integrals and Transformations'. The function *rhs_integration* will compute the right hand side.
This function works in a similar fashion to the assembly of the global stiffness matrix. Given a cell, this function will integrate over every shape function at once, the function *mesh_nodes* will add the resulting vector at the correct spot. We use octave's built-in function *repmat* to multiplay our rigth hand side f to every entry of *hf_eval_poly*. The function will now be compatible with our gauss quadrature function for matrix/vector valued functions.

2.5 Linear solvers

The global stiffness matrix will be sparse, due to the way we chose the basis functions (zero on the vast majority of the cells). The prefered solvers for linear systems with sparse matrices are iterative solvers. We implemented two common algorithms, namely the *minimal residual method* and the *conjugate gradient method*. We are left with a coefficient vector u. We will get our solution, by building the linear combination of the basis ϕ_i with u:

$$u_h = \sum \phi_i u_i$$

Remember, that we do not actually have the basis functions ϕ_i at hand, since we accessed the needed values by using shape functions and transforming the integral. We will need an auxiliary function to evaluate u_h

2.6 Evaluation of the solution

The idea behind *hf_eval_solution* is to find the indices of the basis functions, that do not vanish at a given point (x, y). Similar to the calculation of the right hand side, we will need to know which shape function should be evaluated at which cell. Additionally, we need to know to which entry of the coefficient vector u the computation correlates.
In the special case, that (x, y) lies on a node, the function will return the corresponding entry of u_h.If not, *hf_eval_solution* starts searching for adjacent cells to the given point (x, y), at which we want to evaluate our solution. Once we have the numbers of the active cells, we want to know, which nodes are located on these cells. We will store the local and the global number of the

node. This leaves us with a matrix with rows consisting of the local node number, the corresponding global node number and the cell, on which the node lies. If a node lies on an edge between two cells, only one cell will be stored, the other one will be ommited. It remains to evaluate the correct shapefunction over the correct cell and multiply with the corresponding entry of u_h.

2.7 Presentation of our results

- Our function *m_plot_solution* will create three plots: The analytical solution. The right hand side of the strong formulation (to show, that it is a multiple of the solution). The approximation, using *hf_eval_solution* and the coefficient vector u_h
- Our functions *hf_vtk* and *m_plot_paraview* will create .vtk files and .vtk file series for simulation investigations.
- Our function *linear_solver_analytics* will compare three linear solvers, namely GMRES, CG and MINRES. More over it will plot the accuracy and the runtime for these solvers for varying mesh sizes.
- Our function *error_map* will print out the L2 errors for varying mesh size and polynomial degree into a table.
- Our function *error_runge_evaluation* will show the accuracy of our a posteriori error indicator.

2.8 Runtime optimization

This section will highlight some function, that were programmed in a very specific way to suit our needs. Since matrix and vector operations are optimized in Octave, we avoided for-loops with many iterations and tried utilizing matrix or vector operations.

Evaluation of a Polynomial

Normaly, you would want to evaluate a polynomial p only at a single point $(x, y) \in R$. For our problem, it will be benificial to multiple polynomials at once. We designed *hf_eval_poly* in a way, that we can have the whole coefficient matrix SF as an input, and therefore get a vector with a value for each shape function as an output. To utilize Octave's full potential, we

also wanted to be able to have two matrices as inputs, instead of only single points. This will create a matrix, containing a column of blockmatrices. Each blockmatrix will resemble one shape function, evaluated at the given set of points by the two matrices.

Gauss quadrature

Given the sample points $(x_1, ..., x_n)^T$ and weights $(w_1, ..., w_n)^T$, the gauss quadrature of a function f in 2-D is given by

$$\sum_{i,j} w_i w_j f(x_i, x_j)$$

Instead of using for-loops, we use Octave's built-in function *meshgrid* to create rank one matrices W_1, W_2 for the weigths w_i and X, Y for sample points x_i. Evaluating $f(X, Y)$ will create the matrix

$$\begin{pmatrix} f(x_1, x_1) & f(x_1, x_2) & \cdots & f(x_1, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ f(x_n, x_1) & f(x_n, x_2) & \cdots & f(x_n, x_n) \end{pmatrix}$$

Using component-wise multiplication, we get

$$\begin{pmatrix} w_1 w_1 f(x_1, x_1) & w_1 w_2 f(x_1, x_2) & \cdots & w_1 w_n f(x_1, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ w_n w_1 f(x_n, x_1) & w_n w_2 f(x_n, x_2) & \cdots & w_n w_n f(x_n, x_n) \end{pmatrix}$$

Adding all entries of the matrix will give the desired quadrature. Our function *int_gauss_vectorized* does exactly this. The function will fail when confronted with vector or even matrix-valued functions. We had to write a more complex function *int_gauss_vectorized_matrices*, that has the ability to sum over submatrices, and return the correct quadrature for every component. We will explain this function with an example:
Let f be a matrix-valued function

$$f(x, y) = \begin{pmatrix} f_{11}(x, y) & f_{12}(x, y) \\ f_{21}(x, y) & f_{22}(x, y) \\ f_{31}(x, y) & f_{32}(x, y) \end{pmatrix} \tag{2.1}$$

Using Octave's *repmat* function, we can obtain a matrix, consisting of submatrices,

$$M = \begin{pmatrix} A & B \\ C & D \\ E & F \end{pmatrix}$$

Where each submatrix has the form of (1). Let $A, B, C, D, E, F \in R^{3\times 3}$ for this example. We will generate two matrices and multiply them to the

matrix M:

$$\begin{pmatrix} \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} & \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix} \\ \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix} & \begin{pmatrix} D_{11} & D_{12} & D_{13} \\ D_{21} & D_{22} & D_{23} \\ D_{31} & D_{32} & D_{33} \end{pmatrix} \\ \begin{pmatrix} E_{11} & E_{12} & E_{13} \\ E_{21} & E_{22} & E_{23} \\ E_{31} & E_{32} & E_{33} \end{pmatrix} & \begin{pmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{pmatrix} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \bar{A}_1 & \bar{B}_1 \\ \bar{A}_2 & \bar{B}_2 \\ \bar{A}_3 & \bar{B}_3 \\ \bar{C}_1 & \bar{D}_1 \\ \bar{C}_2 & \bar{D}_2 \\ \bar{C}_3 & \bar{D}_3 \\ \bar{E}_1 & \bar{F}_1 \\ \bar{E}_2 & \bar{F}_2 \\ \bar{E}_3 & \bar{F}_3 \end{pmatrix}$$

Where $\bar{A}_i$ is the sum of the entries of the i-th row of A. We can now transpose this matrix and multiply with another auxiliary matrix

$$\begin{pmatrix} \bar{A}_1 & \bar{A}_2 & \bar{A}_3 & \bar{C}_1 & \bar{C}_2 & \bar{C}_3 & \bar{E}_1 & \bar{E}_2 & \bar{E}_3 \\ \bar{B}_1 & \bar{B}_2 & \bar{B}_3 & \bar{D}_1 & \bar{D}_2 & \bar{D}_3 & \bar{F}_1 & \bar{F}_2 & \bar{F}_3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} A^* & C^* & E^* \\ B^* & D^* & F^* \end{pmatrix}$$

Where A^* is the desired sum of the submatrix A. Transposing this matrix yields the succesfull quadrature.

Vectorization of sm_local_assembly

Let $p_1, ..., p_n$ our shape function, which are defined on our reference square. The local stiffness matrix SM_local consists of the bilinear form of two shape functions

$$SM\_local_{ij} = a(p_i, p_j)$$

As usual, we want to compute the integral at once, the approach using rank one matrices comes to mind. Let P_i be the matrix, which store the values f the shape function p_i at the sample points. We would want to have terms of the form

$$\begin{pmatrix} P_1. * P_1 & \cdots & P_1. * P_n \\ \vdots & \ddots & \vdots \\ P_n. * P_1 & \cdots & P_n. * P_n \end{pmatrix} \tag{2.2}$$

where the operator ".$*$" means Octave's built-in component-wise multiplication. The vector $(P_1, ..., P_n)^T$ is easily obtained through *hf_eval_poly* as already said. Using *repmat*, we can get the matrix

$$\begin{pmatrix} P_1 & \cdots & P_1 \\ P_2 & \cdots & P_2 \\ \vdots & \vdots & \vdots \\ P_n & \cdots & P_n \end{pmatrix} \tag{2.3}$$

very easily. The other part of the "pseudo rank one" matrix is obtained by "cheating": We can get the transposed matrix P_1^T by swapping the arguments x and y in our evaluation function of the polynomials. This leaves us with the matrix in (3), but each P_i will be transposed. Transposing the whole matrix will now yield the matrix

$$\begin{pmatrix} P_1 & P_2 & \cdots & P_n \\ \vdots & \vdots & \vdots & \vdots \\ P_1 & P_2 & \cdots & P_n \end{pmatrix} \tag{2.4}$$

The matrices from (3) and (4) can now be multiplied component-wise to recieve (2). The same steps can be taken for the derivatives of the polynomials, the exact computations are explained in detail in our theoretical analysis. The function is now ready to be integrated by our gauss quadrature.

3 Result