FFW DOCUMENTATION

A. BYFUT, C. CARSTENSEN, J. GEDICKE, D. GÜNTHER, H. MELLMANN, H. RABUS, J. REININGHAUS, AND S. WIEDEMANN

Contents

1. Introduction

This document describes how to use and extend the **F**inite element **F**rame **W**ork, from here on referred to as FFW . The goal of this software package is to provide our target audience, students and researchers in the field of finite element research, a tool which presents various methods in a reference implementation and to provide a platform for future research and development. The following design goals guided the development decisions:

- Clean and readable implementation precedes performance,
- Good extensibility,
- Easy to debug,
- Providing mechanisms for interpreting and visualizing the numerical results.

Following these design goals we have chosen the MATLAB programming language, as it is relatively wide known in our target audience and provides a coherent setting in which one can focus on the problem at hand. For simplicity we only consider methods with triangular elements in $2D$. The FFW currently features:

- Methods
  - P_1-FEM, a standard conforming discretization for elliptic PDE's,
  - CR-FEM, a non-conforming discretization for elliptic PDE's,
  - RT_0-P_0-MFEM, a mixed FEM for elliptic PDE's,
  - $P_1 \times P_1$, a standard conforming discretization for elasticity problems,
  - $P_1 \times CR$, a non-conforming and locking free discretization for elasticity problems,
  - AW, a mixed, higher order, locking free FEM for elasticity problems.
- Adaptivity
  - A newest vertex bisection like algorithm for edge oriented mesh refinement that produces shape regular, nested triangulations with no hanging nodes and allows for the H^1 stability of the L^2 projection,
  - Graded meshes using the above algorithm,
  - Reliable and efficient a posteriori error estimators for almost all of the above methods,
  - Two marking strategies, maximum and bulk, controlling the mesh refinement based on a posteriori error estimators.
- A global data structure containing all computed data, including complete mesh information like edge enumeration, normals, tangents, etc.
- A simple reference multigrid implementation for P_1-FEM.
- Simple integration routines for efficient calculation of boundary integrals, error norms, etc.
- Automatic problem creation to reliably test new methods using symbolic differentiation.

- A general framework for output routines to analyse the methods and results.
- Various test problems to illustrate performance of adaptivity and test correctness.
- Full script ability for automatic computation with different parameters.

2. Quick Start

2.1. **Installation of the FFW.** There is no need to install the FFW. As long as any script is started from the root folder of the FFW the function `initFFW` adds all subfolders to the MATLAB searchpath. Once any script has run `initFFW` successfully, it is also possible to start scripts from any subfolder until the MATLAB session is finished. The folders are not permanently added to the searchpath. Alternatively you can manually add the folder of the FFW with all subfolder to the MATLAB searchpath.

2.2. **Getting Started with the FFW.** This subsection will describe how to execute an existing script for a problem and how to manipulate it.

In the following we are interested in the solution u of the elliptic PDE

$$\begin{aligned} -\operatorname{div}(\kappa\,\nabla u) + \lambda\,\nabla u + \mu\, u &= f && \text{in } \Omega,\\ u &= u_D && \text{on } \Gamma,\\ \frac{\partial u}{\partial n} &= g && \text{on } \partial\Omega\setminus\Gamma. \end{aligned}$$

As model problem we consider the above problem with $\kappa = I$, $\lambda = 0$, $\mu = 0$, $f \equiv 1$, $u_D \equiv 0$ and $g \equiv 0$, on a L-shaped domain Ω with Dirichlet boundary only (Neumann boundary $\partial\Omega \setminus \Gamma = \emptyset$).

The script `gettingStarted.m` in the root folder shows the basic usage of the FFW. It's code is printed in Figure 2.1.

```
function p = gettingStarted

problem = 'Elliptic_Lshape';
pdeSolver = 'P1';
maxNrDoF = 100;
mark = 'bulk';

p = initFFW(pdeSolver,problem,mark,maxNrDoF,'elliptic');
p = computeSolution(p);

figure(1);
set(gcf,'Name','Displacement');
p = show('drawU',p);
view(-30,20)

figure(2);
set(gcf,'Name','Estimated Error');
p = show('drawError_estimatedError',p);
```

FIGURE 2.1. Content of `gettingStarted.m`

To start the script, set MATLAB's current directory to the root of the FFW and execute it.

Line 3: The name of a file that describes the problem definition. This file is located at `.\problems\elliptic\`. For more details on the geometry refer to Section 5. For more details on the problem definition see also Subsection 2.5.

Line 4: The type of finite element method is chosen. This can possibly also be `'CR'`.

Line 5: This is the indicator when to stop the refinement, i.e., the calculations. For more details refer to Section 7.

Line 6: The mark algorithm is chosen. The argument can also be `'maximum'` or `'uniform'`. For more details refer to Section 7.

Line 8: The FFW is initialized with the above arguments, i.e., the structure `p` is created. The last argument describes what problem type is to be computed. This argument can possibly also be `'elasticity'`. See also Section 4.

Line 9: The actual calculations are started. See also Section 3 for details.

Line 12: The solution is plotted with the standard parameters. More details about the `show` function as well as its parameters can be found in Section 8.

Line 15: The estimated error is plotted with the standard parameters.

For the results, i.e. the solution and the estimated error, of the `gettingStarted.m` script, see Figure 2.2.

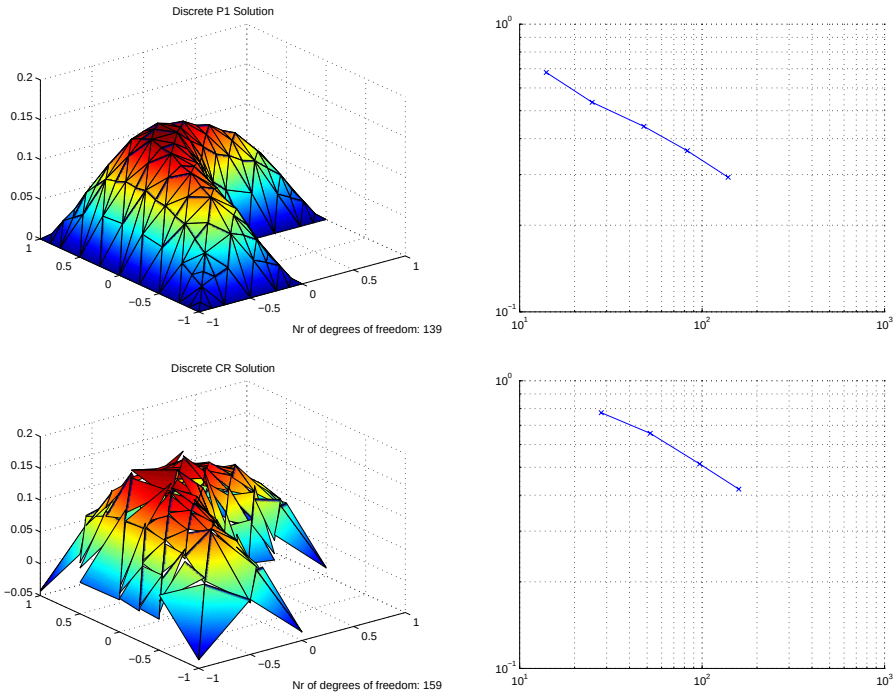


FIGURE 2.2. Plots obtained from the `gettingStarted.m` script for `pdeSolver = 'P1'` on top and `pdeSolver = 'CR'` below.

2.3. **Initialization of the FFW.** The initialization of the FFW is called as follows:

```
p = initFFW(pdeSolver,problem,mark,maxNrDoF,problemType)
```

or with optional Parameter

```
p = initFFW(pdeSolver,problem,mark,maxNrDoF,problemType,solver,refine,estimate)
```

After the initialization of the structure, in the following refered to as `p`, it is still possible to change specific parameters in the structure if needed. In the initialization progress at first all parameters given with the function call as well as the default parameters are stored in `p.params.`, then the function handles to the files needed in the AFEM loop are stored in `p.statics.` and the problem definition is loaded. The geometry is stored in `p.level(end).geom`, see section 4.4. After that the problem type specific initialisation in `.\PDESolver\<problemType>\<problemType>_init.m` as well as the pde solver specific initialization routine in `.\PDESolver\<problemType>\<pdeSolver>\<pdeSolver_init.m>` are executed. The following table gives an overview of the possible parameters for `initFFW.m`.

token	parameter	description
`pdeSolver`	`'P1'`	P1-Elliptic
	`'P2'`	P2-Elliptic
	`'P3'`	P3-Elliptic
	`'CR'`	CR-Elliptic
	`'P1P0'`	P0-P1-mixed-Elliptic (unstable element, works only with uniform refinement)
	`'RT0P0'`	RT0-P0-mixed-Elliptic
		choose different Finite Element Methods
`problem`	<problem name>	a string which represents the name of a file in the folder `\problems\elliptic` or `\problems\elasticity`
`mark`	`'bulk'`	a string which represents
	`'graded'`	a marking algorithm
	`'maximum'`	specified in `\algorithms\mark`
	`'uniform'`	
`maxNrDoF`	a non negative integer	there will be no more refinement if this number of degrees of freedom is reached
`problemType`	`'elliptic'`	defines the type of problem
	`'elasticity'`	
`solver` (optional)	`'direct'`	method how the resulting linear system will be solved (name of a file in `\algorithms\linSysSolvers\`)
	`'multigrid'`	
`refine` (optional)	`'bisection'`	method how to refine (name of a file in `\algorithms\refine\`)
	`'redGreenBlue'`	

`estimate` (optional)	`'estimate'`	a string which represents an error estimating routine, for future implementation of other error estimators

2.4. **Start Scripts.** To make it more convenient to work with the FFW , there are 'startscripts' with almost all available PDE solvers, already implemented problems and different draw routines, for elliptic and elasticity problems respectively. Those scripts can be found in the root folder (`startElasticity.m` and `startElliptic.m`).

Let's have a closer look at the start script for elliptic problems. At first the various parameters for `initFFW` are set. Then the solution is computed. At last the solution, the mesh and the convergence history for the estimated error, the energy error and the L^2 error are plotted. To solve different problems one has to change the given parameters by comment and uncomment the different defined problems.

In the following the graphical output for some examples using P_1 finite elements is presented.

2.4.1. *P_1 FEM for the Elliptic-Square Problem.*

$$-\Delta u = 1 \text{ in } \Omega$$
$$u = 0 \text{ on } \partial\Omega$$

In this example Ω is the unit square. The error estimator has convergence rate $O(h)$ thus $N^{1/2}$, see Figure 2.3.

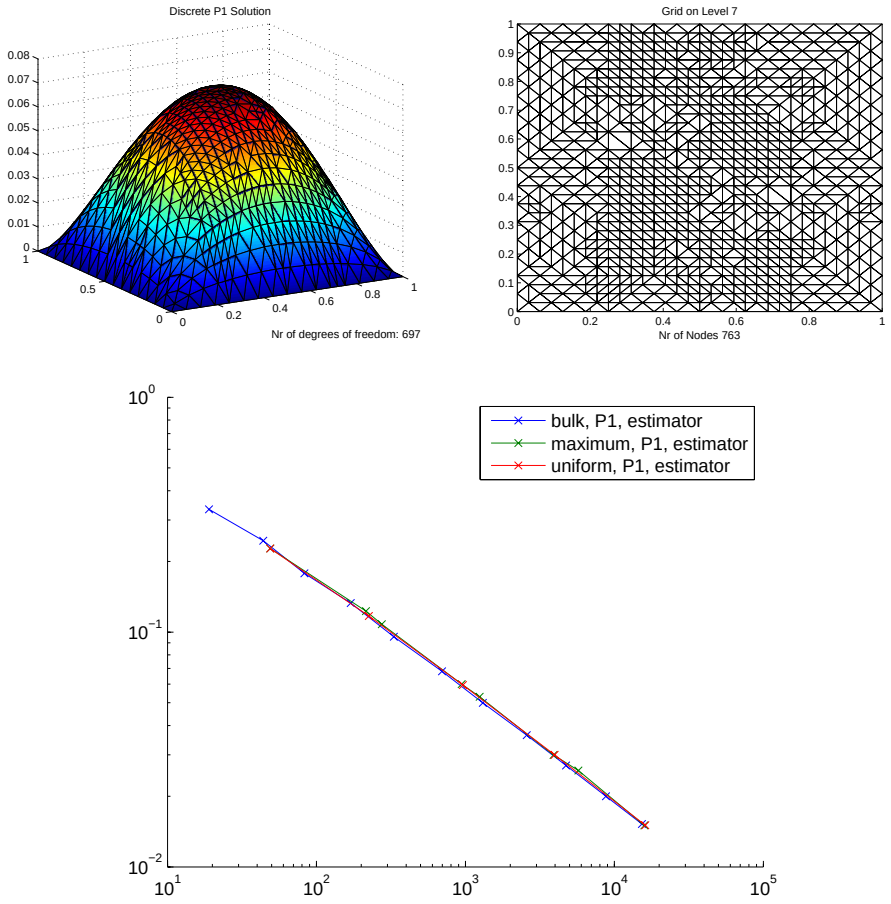


FIGURE 2.3. Plots for the example `Elliptic-Square`

2.4.2. *P_1 FEM for the Elliptic-Lshape Problem.*

$$-\Delta u = 1 \text{ in } \Omega$$
$$u = 0 \text{ on } \partial\Omega$$

In this example Ω is the L-shaped domain. The solution is at least in $H^{5/3}(\Omega)$, therefore the expected convergence rate of the H^1 semi-error is $O(h^{2/3})$. Thus the convergence rate with respect to the degrees of freedom should be $N^{1/3}$ using uniform refined meshes, see Figure 2.4.

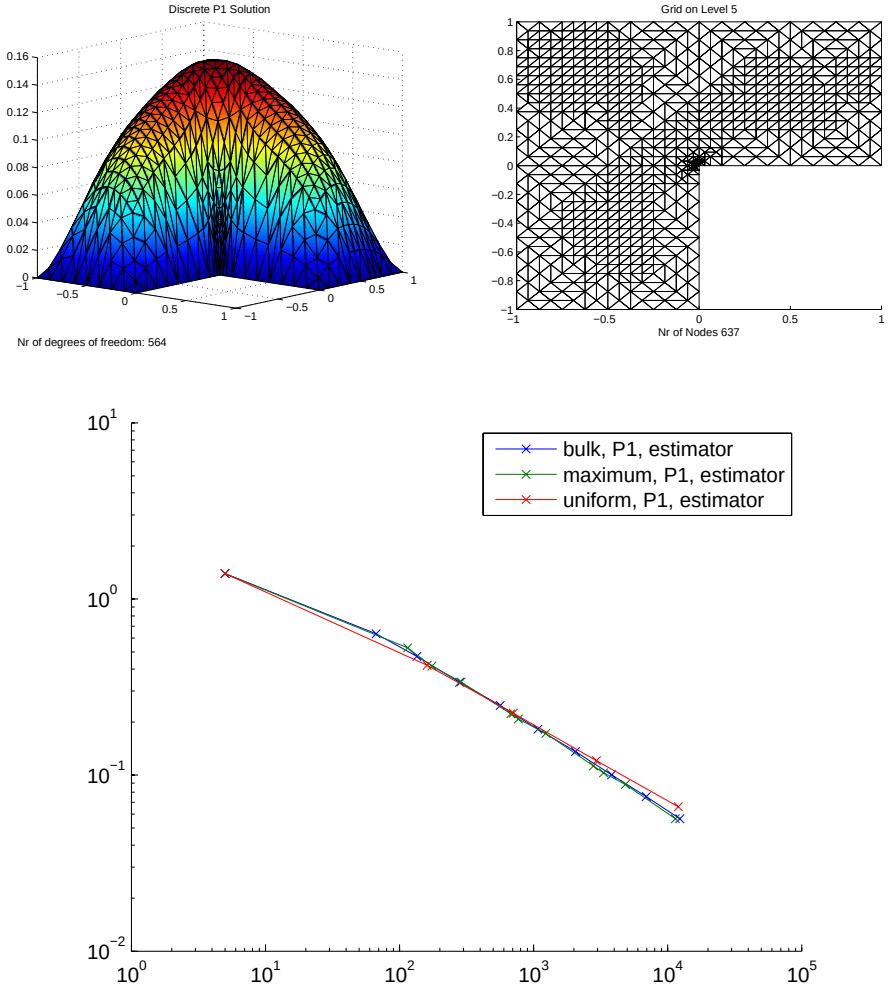


FIGURE 2.4. Plots for the example `Elliptic-Lshape`

2.4.3. *P_1 FEM for the Elliptic-Square-exact Problem.*

$$-\Delta u = 2(x(1-x)+y(1-y)) \text{ in } \Omega$$
$$u = 0 \text{ on } \partial\Omega$$

The exact solution for the unit square is given by $u = x(1-x)y(1-y)$. The Convergence rate is $O(h)$ for the energy error and because of Aubin-Nietsche $O(h^2)$ for the L^2-error, see Figure 2.5.

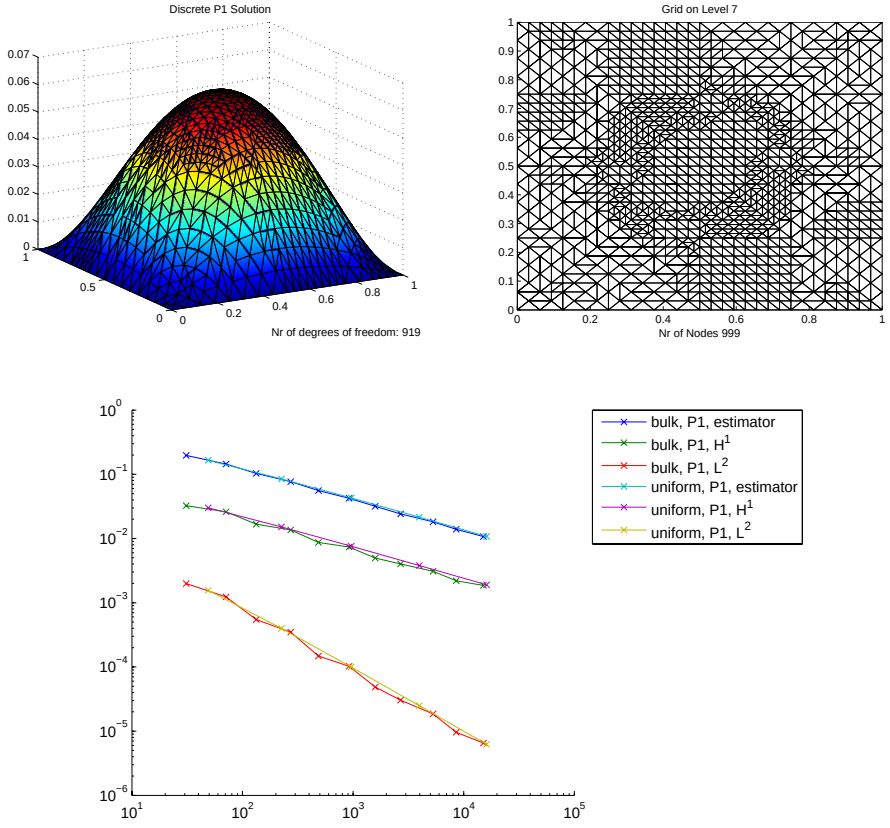


FIGURE 2.5. Plots for the example `Elliptic-Square-exact`

2.4.4. *P_1 FEM for the Elliptic-Lshape-exact Problem.*

$$-\Delta u = 0 \text{ in } \Omega$$
$$u = 0 \text{ on } \Gamma_D = \{(x = 0 \wedge y \in [0, -1]) \vee (y = 0 \wedge x \in [0, 1])\}$$
$$\partial u \cdot \nu = g(x) \text{ on } \Gamma_N = \partial\Omega \backslash \Gamma_D$$

The exact solution given in polar coordinates is $u = r^{2/3} \sin(2/3\phi)$ for the L-shaped domain. Uniform meshes yield a convergence rate of $O(h^{2/3})$ for the energy error and $O(h^{4/3})$ for the L^2 error, see Figure 2.6.

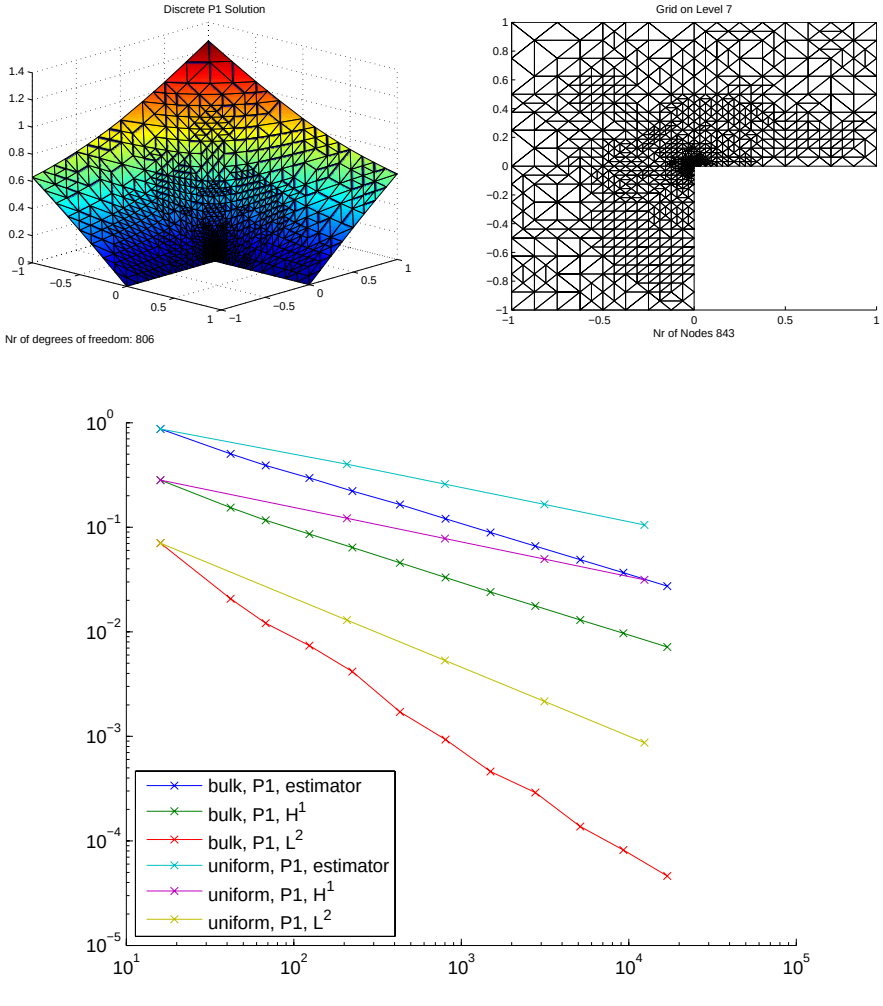


FIGURE 2.6. Plots for the example `Elliptic-Lshape-exact`

2.4.5. *P_1 FEM for the Elliptic-Waterfall Problem.*

$$-\Delta u = f(x,y) \text{ in } \Omega$$
$$u = 0 \text{ on } \partial\Omega$$

The exact solution is given by

$$u = xy(1-x)(1-y)\text{atan}(k(\sqrt{(x-5/4)^2+(y+1/4)^2}-1))$$

for the unit square. The convergence rate for the energy error is $O(h)$ whereas for the L^2-error it is $O(h^2)$. For $k \to \infty$ the slope of the function in special points tends to infinity, see Figure 2.7.

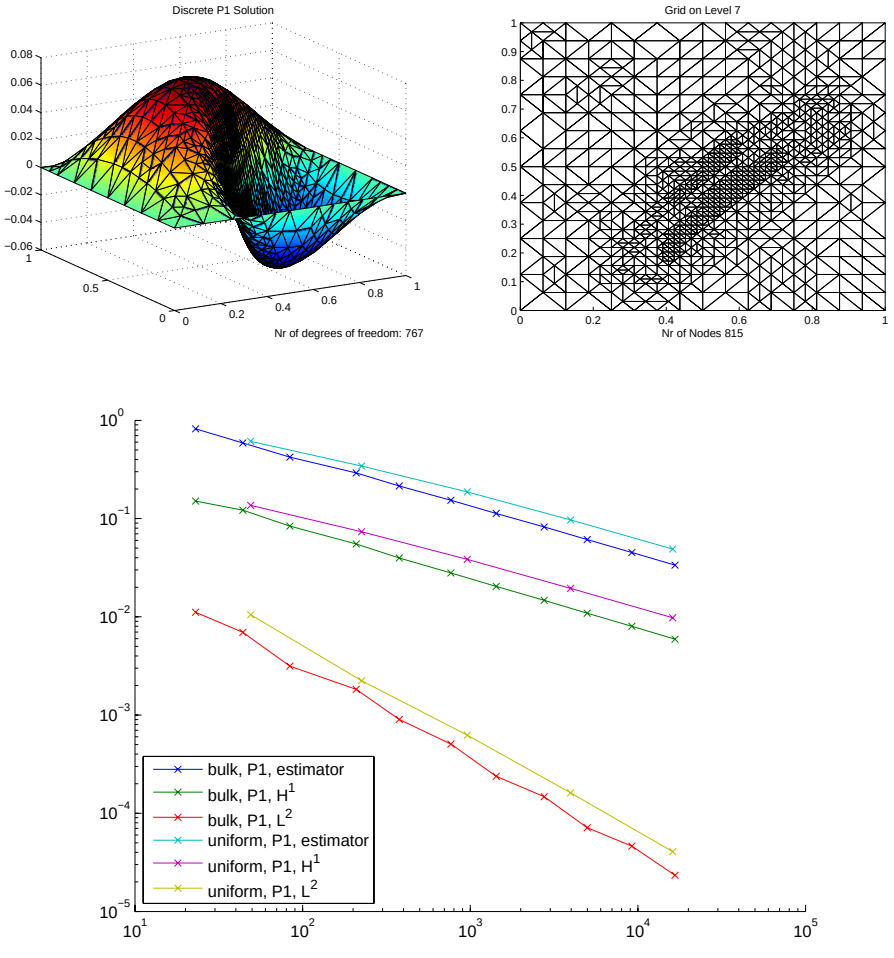


FIGURE 2.7. Plots for the example `Elliptic-Waterfall`

2.5. Problem Definition.

2.5.1. *Problem Definition for given right hand side.* This Subsection will try to explain the problem definition files for given right hand side (RHS). To make this more practical we will consider our elliptic example from Subsection 2.2. It was created from the template `.\problems\Elliptic\Elliptic_Template.m`.

The actual problem definition looks as follows:

```
function p = <name>(p)

p.problem.geom = <shape>;
p.problem.f = @<f>;
p.problem.g = @<g>;
p.problem.u_D = @<u_D>;
p.problem.kappa = @<kappa>;
p.problem.lambda = @<lambda>;
p.problem.mu = @<mu>;
```

In the following table we will explain this code. `<f>`, `<g>`, `<u_D>`, `<kappa>`, `<lambda>`, `<mu>` are functions that represent the corresponding functions of the problem. In general these have the following form: `function z = <name>(x,y,p)`, where (`x`,`y`) are the coordinates for all nodes (here supposed to be N), and `p` is the structure.

	Description
`<name>`	Any function name valid in MATLAB (should describes the problem definition).
`<shape>`	This is the name of the folder that contains the data for a geometry. Those folders have to be located at `.\problems\geometries\`. For more information about what defines a geometry see Section 5
`<f>`	Returns a vector with function values of f of length N calculated at (x, y).
`<g>`	Returns a vector of length n, where n is the number of points (x, y) passed.
`<u_D>`	Returns a vector of length n, where n is the number of points (x, y) passed.
`<kappa>`	Returns a $(2 \times 2 \times N)$ matrix, where the entry $(:, :, i)$ is the value of κ at node i, i.e., $\kappa(x_i, y_i) = z(:, :, i)$.
`<lambda>`	Returns a $(2 \times N)$ matrix, where the entry $(:, i)$ is the value of λ at node i, i.e., $\lambda(x_i, y_i) = z(:, i)$.
`<mu>`	Returns a vector of length N that contains the function values of μ for every node (x_i, y_i).

If one wants to solve a given problem, one uses the template `Elliptic_Template.m` in `.\problems\elliptic`. In the renamed copy one has to change the function definitions of the data. The relevant part of the template `Elliptic_Template.m` looks as follows:

```
% Volume force
function z = f(x,y,p)
z = ones(length(x),1);

% Dirichlet boundary values
function z = u_D(x,y,p)
z = zeros(length(x),1);
```

```
% Neumann boundary values
function z = g(x,y,n,p)
z = zeros(length(x),1);

% elliptic PDE coefficent kappa ( div(kappa*grad_u) )
function z = kappa(x,y,p)
nrPoints = length(x);
z = zeros(2,2,nrPoints);
for curPoint = 1:nrPoints
    z(:,:,curPoint) = [1 0;
                       0 1];
end

% elliptic PDE coefficent lambda ( lambda*grad_u )
function z = lambda(x,y,p)
nrPoints = length(x);
z = zeros(nrPoints,2);
for curPoint = 1:nrPoints
    z(curPoint,:) = [0 , 0];
end

% elliptic PDE coefficent mu ( mu*u )
function z = mu(x,y,p)
z = zeros(length(x),1);
```

2.5.2. *Problem definition for given exact solution.* In this subsection we describe to use the template file `Elliptic_Exact_Template.m` in `.\problems\elliptic` for the problem definition with given exact solution. The idea is that using the symbolic toolbox MATLAB by itself computes the correct right hand side and boundary values for a given exact solution. To explain how to change the data we look at the example of a elliptic PDE of the form

$$\begin{aligned} -\operatorname{div}(\kappa\,\nabla u) + \lambda\,\nabla u + \mu\, u &= f && \text{in } \Omega, \\ u &= u_D && \text{on } \Gamma, \\ \frac{\partial u}{\partial n} &= g && \text{on } \partial\Omega \setminus \Gamma, \end{aligned}$$

One has to copy the file `Elliptic_Exact_Template.m` and rename the copy. In the copy of the template one can change the symbolic expressions for u, κ, λ and μ then the right hand side f is calculated automatically such that u is the solution of the PDE. The relevant part of the template `.\Elliptic_Exact_Template.m` looks as follows:

```
% Specification of exact solution and differential Operator
u = sin(x^3)*cos(y^pi)+x^8-y^9+x^6*y^10;
lambda = [0, 0];
mu = 0;
kappa = [1 0; 0 1];
```

2.5.3. *Definition of geometry.* If one wants to change the geometry of Ω, one can use one of the implemented geometries in `.\problems\geometries` by changing the geometry in the PDE definition in the copy of the template file to the name of the folder containing the data. p.problem.geom = 'Lshape';

To use a new geometry one has to create a new folder in `.\problems\geometries` which contains the data files `<geometryname>_n4e.dat`, `<geometryname>_c4n.dat`, `<geometryname>_Db.dat` and `<geometryname>_Nb.dat`. Here `<geometryname>`

is the name of the created folder. To see how the geometry data must be structured see Section 5 and the already existing folders.

2.5.4. *Choose new problem.* In `start_elliptic.m` the new Problem is chosen by

```
problem = '<name of file>'
```

where `<nameoffile>` is the name of the copy from the template.

3. Flow Chart

This Section is devoted to the structure of the FFW and illustrates to flow of the framework. In Figure 3.1 one can see this structure in a flow chart.

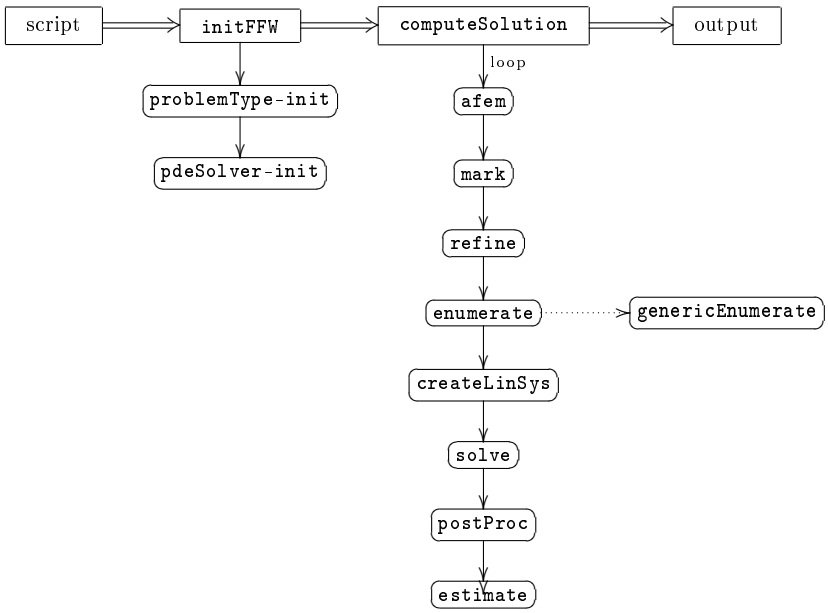


FIGURE 3.1. Flow chart of the FFW .

function name	description
script	control script, examples are `startScalar` and `startElasticity`
`initFFW`	sets paths, loads default parameters, sets supplied parameters and loads problem definition and geometry
`problemType-init`	creates method specific function handles, e.g. a function handle to evaluate the discrete solution
`pdeSolver-init`	creates finite element space specific function handles, e.g. a function handle to eval the basis functions
`computeSolution`	calls `afem` as long as some abort criteria defined in the configuration file, e.g., `maxNrDoF`, is not fulfilled and supplies method type specific function handles, e.g. a function to evaluate the energy error
`afem`	calls `mark`, `refine`, `enumerate`, `createLinSys`, `solve`, `postProc` and `estimate`
`mark`	marks edges and triangles for refinement based on an error estimator (`bulk`, `max`) with additional edges to maintain shape regularity, or marks all edges (`uniform`)
`refine`	refines the list of edges supplied by mark
`enumerate`	calls `genericEnumerate` and computes method specific data structures from the triangulation, e.g., the number of degrees of freedom
`genericEnumerate`	creates useful data structures from the current triangulation, e.g., normals, tangents, areas
`createLinSys`	assembles the global linear system of equations $Ax = b$

`solve`	computes x
`postProc`	post processing of the solution, e.g., separating Lagrange multipliers form the discrete solution
`estimate`	locally estimates the error based on the discrete solution
output	user specific data evaluation

4. Implemented Problems

In this section we give an overview on the implemented problems. At first, we present the strong and weak formulation for the full elliptic PDE and for a problem in elasticity. We define the spaces for the conforming and non-conforming discretizations. Additionally, we take a look on the mixed formulation with the corresponding spaces.

There are several problems and geometries already defined in the FFW. The geometry definitions are stored in `.\problems\geometries`, where as the problem definitions for elliptic and elasticity problems are stored in `.\problems\elliptic` and `.\problems\elasticity`, respectively.

4.1. FEM for Elliptic PDEs.

We consider the following elliptic PDE

$$\begin{aligned} -\operatorname{div}(\kappa \cdot \nabla u) + \lambda \cdot \nabla u + \mu u &= f && \text{in } \Omega, \\ u &= u_D && \text{on } \Gamma_D, \\ \nabla u \cdot \nu &= g && \text{on } \Gamma_N, \end{aligned} \tag{4.1}$$

with $u_D \in H^1(\Omega;\mathbb{R})$, $f \in L^2(\Omega;\mathbb{R})$, $g \in L^2(\Gamma_N;\mathbb{R})$, $\kappa \in L^\infty(\Omega;\mathbb{R}^{2\times 2})$, $\lambda \in L^\infty(\Omega,\mathbb{R}^2)$ and $\mu \in L^\infty(\Omega,\mathbb{R})$.

The weak form of (4.1) reads: Find $u \in V$ such that

$$\int_\Omega (\nabla u \cdot (\kappa \cdot \nabla v) + \lambda \cdot \nabla u\, v + \mu u\, v)\, dx = \int_\Omega f\, v\, dx + \int_{\Gamma_N} g\, v\, ds_x. \tag{4.2}$$

for all $v \in V$.

If the main interest is on an accurate stress or flux approximation and some strict equilibration condition rather than the displacement, it might be advantageous to consider an operator split: Instead of one partial differential equation of order $2m$ one considers two equations of order m. To be more precise, given one equation in an abstract form $\mathcal{L}u = G$ with some differential operator $\mathcal{L} = \mathcal{A}\mathcal{B}$ composed of $\mathcal{A}$ and $\mathcal{B}$, define $p := \mathcal{B}u$ and solve the two equations $\mathcal{A}p = G$ and $\mathcal{B}u = p$.

The mixed formulation of (4.1) reads

$$\begin{aligned} -\operatorname{div}\sigma + \lambda \cdot (\kappa^{-1} \cdot \sigma) + \mu u &= f && \text{in } \Omega, \\ \sigma &= \kappa \cdot \nabla u && \text{in } \Omega, \\ u &= u_D && \text{on } \Gamma_D, \\ (\kappa^{-1} \cdot \sigma) \cdot \nu &= g && \text{on } \Gamma_N \end{aligned} \tag{4.3}$$

with the corresponding weak form

$$\begin{aligned} \int_\Omega \left(-\operatorname{div}\sigma\, v + \lambda \cdot (\kappa^{-1} \cdot \sigma)\, v + \mu u\, v\right)\, dx &= \int_\Omega f\, v\, dx, \\ \int_\Omega \left((\kappa^{-1} \cdot \sigma) \cdot \tau + \operatorname{div}\tau\, u\right)\, dx &= \int_{\Gamma_D} u_D (\tau \cdot \nu)\, ds_x, \end{aligned} \tag{4.4}$$

for all $v \in V$ and all $\tau \in \Sigma$.

4.2. **FEM for Elasticity.** We consider the following elliptic system of PDE's modelling linear elasticity.

$$\begin{aligned} -\operatorname{div} \mathbb{C}\,\varepsilon(u) &= f && \text{in } \Omega, \\ u &= u_D && \text{on } \Gamma_D, \\ (\mathbb{C}\,\varepsilon(u))\cdot\nu &= g && \text{on } \Gamma_N, \end{aligned} \tag{4.5}$$

with $u_D \in H^1(\Omega;\mathbb{R}^2)$, $f \in L^2(\Omega;\mathbb{R}^2)$, $g \in L^2(\Gamma_N;\mathbb{R}^2)$. Here and throughout, $\varepsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^T)$ denotes the linearized Green strain tensor, $\mathbb{C}$ is the reduced symmetric fourth order bounded and positive definite elasticity tensor defined by the Lamé parameters λ and μ. In Voigt notation the tensor $\mathbb{C}$ and its inverse is given through

$$\mathbb{C} := \begin{pmatrix} 2\mu+\lambda & \lambda & 0 \\ \lambda & 2\mu+\lambda & 0 \\ 0 & 0 & \mu \end{pmatrix} \quad \text{and} \quad \mathbb{C}^{-1} := \begin{pmatrix} \frac{\lambda+2\mu}{4\mu(\lambda+\mu)} & \frac{-\lambda}{4\mu(\lambda+\mu)} & 0 \\ \frac{-\lambda}{4\mu(\lambda+\mu)} & \frac{\lambda+2\mu}{4\mu(\lambda+\mu)} & 0 \\ 0 & 0 & \frac{1}{\mu} \end{pmatrix}.$$

The weak form of (4.5) reads: Seek $u \in V$

$$\int_\Omega \varepsilon(u) : \mathbb{C}\varepsilon(v)\,dx = \int_\Omega f\cdot v\,dx + \int_{\Gamma_N} g\cdot v\,ds_x \quad \text{for all } v \in V. \tag{4.6}$$

If the main interest is on an accurate stress or flux approximation and some strict equilibration condition rather than the displacement, it might be advantageous to consider an operator split: Instead of one partial differential equation of order $2m$ one considers two equations of order m. To be more precise, given one equation in an abstract form $\mathcal{L}u = G$ with some differential operator $\mathcal{L} = \mathcal{A}\mathcal{B}$ composed of $\mathcal{A}$ and $\mathcal{B}$, define $p := \mathcal{B}u$ and solve the two equations $\mathcal{A}p = G$ and $\mathcal{B}u = p$.
The mixed form of (4.5) reads:

$$\begin{aligned} -\operatorname{div}\sigma &= f && \text{and} && \sigma = \mathbb{C}\,\varepsilon(u) && \text{in } \Omega, \\ u &= u_D \text{ on } \Gamma_D && \text{and} && \sigma\nu = g \text{ on } \Gamma_N. \end{aligned}$$

with the corresponding weak form

$$\begin{aligned} \int_\Omega \sigma : \mathbb{C}^{-1}\tau\,dx + \int_\Omega u\cdot\operatorname{div}\tau\,dx &= \int_{\Gamma_D} u_D\cdot(\tau\,\nu)\,ds_x && \text{for all } \tau \in \Sigma, \\ \int_\Omega v\cdot\operatorname{div}\sigma\,dx &= -\int_\Omega f\cdot v\,dx && \text{for all } v \in V. \end{aligned}$$

4.3. **Predefined Problem Definitions.** There are various problem definitions for elliptic PDEs and for elasticity already defined, for which we give an overview in the following.

All problem definitions contain functions for κ, λ, μ, f, u_D and g.

Some of the problem-definition-filenames contain a suffix `_exact`. Here a function u_{exact} that represent the exact solution is given and the data-functions f, u_D and g are computed automatically. With those problem definitions it is possible to compute the exact errors, e.g. energy error between u_h and u_{exact}.

Problem definitions for elasticity additionally contain standard parameters for nu and E.

If you want to create your own problem definition, create an `.m`-file named `Elliptic_<problemName>` in `.\problems\elliptic` or `Elasticity_<problemName>` in (`.\problems\elasticity`), which contains all necessary information. For details concerning the structure of the problem definition, see Section 2.

4.3.1. *Predefined Problem Definitions for Elliptic PDEs.*

Elliptic_Lshape

Model example $-\Delta u = 1$, $u_D = 0$ and $g = 0$ on an L-shaped domain.

Elliptic_Lshape_exact

The exact solution $u(r,\phi) = r^{2/3}\sin(2/3\phi)$ in polar coordinates on an L-shaped domain with Neumann boundary is given. The corresponding right hand side f is zero and induced boundary data. The coefficients of the elliptic PDE correspond to the Laplacian operator.

Elliptic_Square

Model example $-\Delta u = 1$, $u_D = 0$ and $g = 0$ on a squared domain.

Elliptic_Square_exact

The exact solution $u(x,y) = x(1-x)\,y(1-y)$ as well as its gradient and the right-hand side f are given with the PDE-coefficients κ, λ and μ all zero, on a squared domain is given.

Elliptic_SquareFullElliptic_exact

The exact solution $u(x,y) = \sin(x^3)\cos(y^\pi) + x^8 - y^9 + x^6y^{10}$ with the PDE-coefficients κ being the identity, $\lambda = \begin{pmatrix} 5\sin(x+y) \\ 6\cos(x+y) \end{pmatrix}$ and $\mu = 7$ on a squared domain is given. The symbolic toolbox of MATLAB computes all necessary information, i.e., f, u_D and g.

Elliptic_HexagonalSlit_exact

The exact solution $u(r,\phi) = r^{1/4}\sin(1/4\phi)$ in polar coordinates with the load $f \equiv 0$, the Dirichlet function $u_D = u|_{\Gamma_D}$, and coefficients belonging to the Laplacian are given on a slitted hexagon.

Elliptic_Waterfall_exact

The waterfall function

$$u(x,y) = xy(1-x)(1-y)\arctan\left(k(\sqrt{(x-5/4)^2+(y+1/4)^2}-1)\right)$$

is given. The parameter k controls the slope of u. For $k \to \infty$ the slope of the function tends to infinity. This parameter is stored in the structure `p` at `p.PDE.k` and can be changed in the starting scripts. The domain is a square with homogeneous Dirichlet boundary. We look at the problem $-\Delta u = f$. The load f is computed from u.

Elliptic_Template

A predefined template for generating problem definitions. For given data f, u_D and g one can compute the corresponding discrete solution u_h.

Elliptic_Template_Exact

A predefined template for generating problem definitions. For a given function $u_{\text{exact}}(x,y)$ in cartesian coordinates and coefficients κ, λ, μ, the symbolic toolbox of MATLAB computes all necessary information, i.e., f, u_D and g.

4.3.2. *Predefined Problem Definitions for Elasticity.*

Elasticity_ Cooks
A tapered panel is clamped on one end and subjected to a surface load in vertical direction on the opposite end with $f = 0$ and $g(x, y) = (0, 1000)$ if $(x, y) \in \Gamma_N$ with $x = 48$ and $g = 0$ on the remaining part of Γ_N, the Young modulo $E = 2900$, and the Possion ratio $\nu = 0.3$.

Elasticity_ Square_ exact
The unit square with the given function $u(x, y) = 10^{-5} \begin{pmatrix} \cos((x+1)(y+1)^2) \\ \sin((x+1))\cos(y+1) \end{pmatrix}$. Young modulo and the Possion ratio are set to $E = 10^5$ and $\nu = 0.3$.

Elasticity_ Square_ Neumann_ exact
Besides the geometry we have the same problem definition as in *Elasticity_ Square*. The domain changes from `Square` to `SquareNeumann`.

Elasticity_ Lshape_ exact
Using polar coordinates (r, θ), $-\pi < \theta \leq \pi$ u with radial component u_r, u_θ reads

$$u_r(r, \theta) = \frac{r^\alpha}{2\mu}(-(\alpha + 1)\cos((\alpha + 1)\theta) + (C_2 - (\alpha + 1))C_1 \cos((\alpha - 1)\theta)),$$

and

$$u_\theta(r, \theta) = \frac{r^\alpha}{2\mu}((\alpha + 1)\sin((\alpha + 1)\theta) + (C_2 + \alpha - 1)C_1 \sin((\alpha - 1)\theta)).$$

The parameters are $C_1 = -\cos((\alpha + 1)\omega)/\cos((\alpha - 1)\omega)$, $C_2 = 2(\lambda + 2\mu)/(\lambda + \mu)$ where $\alpha = 0.54448...$ is the positive solution of $\alpha \sin 2\omega + \sin 2\omega\alpha = 0$ for $\omega = 3\pi/4$; the Young modulus is $E = 10^5$, Poisson ratio $\nu = 0.3$, and the volume force $f \equiv 0$.

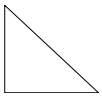
Elasticity_ Template
A predefined template for generating problem definitions. For given data f, u_D and g and coefficients κ, λ, μ one can compute the corresponding discrete solution u_h.

Elasticity_ Exact_ Template
A predefined template for generating problem definitions. For a given function $u_{\text{exact}}(x, y)$ in cartesian coordinates, the symbolic toolbox of MATLAB computes all necessary information, i.e., f, u_D and g.

4.4. **Geometries.** In the following we give an overview about the geometries which are already available.

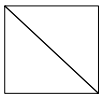
To add a new geometry create a new directory `<newGeometry>` in `.\problems\geometries\`. There you have to define the coordinates of the geometry in `<newGeometry>_c4n.dat`. The nodes for each element, e.g. triangle, have to be defined in `<newGeometry>_n4e.dat`. The Dirichlet- and Neumann-part of the domain boundary is specified in `<newGeometry>_Db.dat` and `<newGeometry>_Nb.dat` respectively.

Triangle



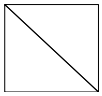
The reference triangle $\Omega = \text{conv}\{(0,0),(1,0),(0,1)\}$ with pure Dirichlet-boundary.

Square



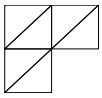
The unit square $\Omega = [0,1]^2$ with pure Dirichlet-boundary.

SquareNeumann



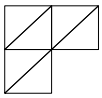
The unit square with Neumann boundary $\Gamma_N = \text{conv}\{(1,0),(1,1)\}$.

Lshape



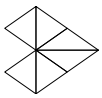
The L-shaped domain $\Omega = [-1,1]^2 \setminus \{(0,1] \times (0,-1]\}$ with pure Dirichlet-boundary.

LshapeNeumann



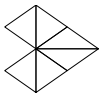
The domain is the L-shape domain with Neumann boundary. $\Gamma_D = \{\text{conv}\{(0,0),(1,0)\} \cup \text{conv}\{(0,0),(0,-1)\}\}$.

Lshape3



Lshape3 is a rotated version of Lshape. Here the boundary of the rotated L-shaped domain is not axis parallel anymore.

Lshape3Neumann



Lshape3Neumann is a rotated version of LshapeNeumann.

Cooks

Geometry for the Cook's membrane problem in elasticity. $\Omega = \text{conv}\{(0,0),(48,44),(48,60),(0,44)\}$. The Dirichlet boundary Γ_D is $\text{conv}\{(0,0),(0,44)\}$. Thus the Neumann boundary is $\partial\Omega \setminus \Gamma_D$.

HexagonalSlit

We have defined a hexagon $\Omega \subset [-1,1]^2$ which is slitted in $\{0\} \times [0,1]$. The slit is approximated by adding an additional node through a small, numerically insignificant, perturbation of the node $(0,1)$. The slitted hexagon is an example for a non-Lipschitz domain with pure Dirichlet boundary.

5. Data Structures

All the functions that calculate the grid information needed in the FFW , e.g. area of elements, length of edges, outer unit normals of the boundary, and the local gradients for P_1 and P_1^{NC} basis functions are located at `.\algorithms\enum`.

The information returned by those functions, the enumerated data, is stored in matrices. In these matrices a row or column number corresponds to the number of an element, node, edge etc. If there is no information on an element, node, edge etc. the corresponding entry is zero. For example a boundary edge will have only one none zero entry in `e4ed`, since there is only one element which contains it.

In the names of the data matrices `e` stands for **e**lements, `n` for **n**odes, `ed` for **ed**ges, `Db` for **D**irichlet**b**oundary, `Nb` for **N**eumann**b**oundary and `4` means '**for**'. For example, the matrix `e4ed` contains elements for an edge. In the following these names are used as well as the abbreviations `nrElems`, `nrNodes` etc., standing for number of elements, number of nodes etc. .

In a triangulation in the FFW the geometric primitives (elements, nodes and edges) are numbered uniquely. Dirichlet and Neummann edges are also numbered in this way. The numbering of elements, nodes, Dirichlet and Neumann edges is defined by `n4e`, `c4n`, `Db` and `Nb`, respectively, whereas the edge numbers are created in `.\enum\getEd4n`. All the other information (e.g. Normals, Tangents) is not numbered and is used as attributes of the information above.

For the geometric primitive one has not only a global number, but most times also a local one. For example the global number of a node is defined by the row number in `c4n`, and for each element its nodes are locally numbered from one to three by there order in the corresponding row in `n4e`.

To calculate all the grid information the only initial data needed is `n4e`, `c4n`, `Db` and `Nb`. Please note that not all information is calculated using directly the initial data. For example `e4n` needs only the information in `n4e`, but `ed4n` is created using the information in `e4n`.

5.1. **Initial Data.**

Name	Dimension	Description
`c4n`	`[nrNodes 2]`	Each row defines a node at the coordinates given by the values. Where the first column contains the x-coordinate and the second the y-coordinate. The number of the defined node is the number of the row.
`n4e`	`[nrElems 3]`	Each row defines an element with the nodes corresponding to the entries as vertices. The vertices have to be entered counter clockwise. The number of the defined element is the number of the row.
`Db`	`[nrDirichletEdges 2]`	Each row defines the edge between the two nodes corresponding to the entries as a Dirichlet edge. In each row the nodes have to be in the same order as in the corresponding element.

Nb	[nrNeumannEdges 2]	Each row defines the edge between the two nodes corresponding to the entries as a Neumann edge. In each row the nodes have to be in the same order as in the corresponding element.

5.2. **Enumerated Data.**

Name	Dimension	Description
e4n	[nrNodes nrNodes]	Each entry (j,k) contains the number of the element which has the nodes j and k counter clockwise as vertices.
ed4n	[nrNodes nrNodes]	Each entry (j,k) contains the number of the edge between the nodes j and k.
ed4e	[nrElems 3]	Each row contains the edge numbers of the corresponding element.
n4ed	[nrEdges 2]	Each row contains the node numbers of the corresponding edge.
e4ed	[nrEdges 2]	Each row contains the element numbers of the elements sharing the corresponding edge in descending order.
DbEdges	[nrDbEdges 1]	Each row contains the number of a Dirichlet edge corresponding to the row in Db.
NbEdges	[nrNbEdges 1]	Each row contains the number of a Neumann edge corresponding to the row in Nb.
area4e	[nrElems 1]	Each row contains the area of the corresponding element.
midpoint4e	[nrElems 2]	Each row contains the coordinates of the midpoint of the corresponding element.
midpoint4ed	[nrEdges 2]	Each row contains the coordinates of the midpoint of the corresponding edge.
tangents4e	[3 2 nrElems]	Each 3×2 matrix contains the coordinates of the three unit tangents of the corresponding element.
normals4e	[3 2 nrElems]	Each 3×2 matrix contains the coordinates of the three outer unit normals of the corresponding element.
normals4DbEd	[nrDbEdges 2]	Each row contains the coordinates of the outer unit normals of the corresponding Dirichlet edge.
normals4NbEd	[nrNbEdges 2]	Each row contains the coordinates of the outer unit normals of the corresponding Neumann edge.
length4ed	[nrEdges 1]	Each row contains the length of the corresponding edge.
area4n	[nrNodes 1]	Each row contains the area of the node patch of the corresponding node.

`angles4e`	`[nrElems 3]`	Each row contains the inner angles at each node of the corresponding element in the order given in the rows of `n4e`.
`angle4n`	`[nrNodes 1]`	Each row contains the sum of all inner angles at the corresponding node.
`grad4e`	`[3 2 nrElems]`	Each 3×2 matrix contains the three local gradients of the three P_1 basis functions not completely zero on the corresponding element.
`gradNC4e`	`[3 2 nrElems]`	Each 3×2 matrix contains the three local gradients of the three CR basis functions not completely zero on the corresponding element.

The following figure shows how one can get from one geometric primitive to another one. To get for example all edges for one element just look at the corresponding row in `ed4e`.

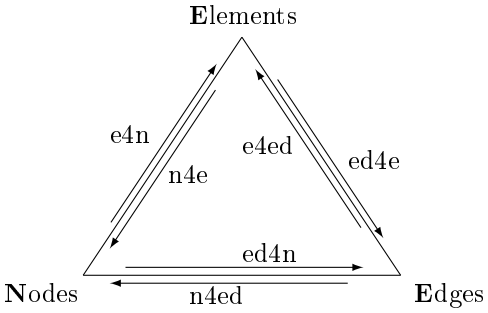


FIGURE 5.1. This diagram illustrates the relations, realized by the enumerated data above, between elements abbreviated by `e`, edges abbreviated by `ed` and nodes abbreviated by `n`.

Figure 5.2 shows a triangulation of two triangles generated by the FFW. The numbering is used to illustrate the structure of the data created in `.\algorithms\enum` by means of some examples below.

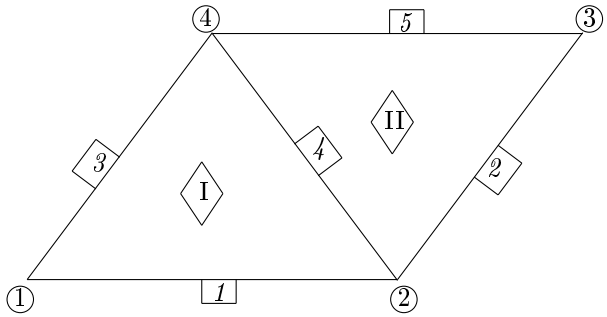


FIGURE 5.2. Enumeration of nodes, edges and elements. The roman numbers in the rhombuses are the element numbers, the numbers in circles are the node numbers and the italic numbers in the rectangles are the edge numbers.

The following `n4e` creates a triangulation as shown in the Figure 5.2.

```
>> n4e
n4e =
     1     2     4
     2     3     4
```

Note that the order of the nodes is important in `e4n` but it is not in `ed4n`.

```
>> e4n(2,4)
ans =
     1
>> e4n(4,2)
ans =
     2
>> ed4n(2,4)
ans =
     4
>> ed4n(4,2)
ans =
     4
```

One can also get the information for more than one item at a time.

```
>> n4ed([3 4],2)
ans =
     4
     4
```

If there is only one element containing the edge, i.e., the edge is a boundary edge, then the second entry is zero.

```
>> e4ed([1 4],:)
ans =
     1     0
     2     1
```

The edges are ordered counter clockwise for each element. The first one in each element is the one between the first end the second node (*not* opposite to the first node).

```
>> ed4e(1,:)
ans =
     1     4     3
```

5.3. **Overview of the Enumeration Functions.**

getE4n.m

The function `getE4n` returns a `[nrNodes nrNodes]` sparse matrix. *Note*: The sparsity constant is bounded due to the used mesh generation. The input is `n4e`. In this matrix each entry (j, k) is the number of the element, whose boundary contains the nodes j and k in counter clockwise order as vertices or zero if the there is no such element. Since the nodes in `n4e` are oriented counter clockwise `e4n` gives you the number of the row in which the sequence $j\ k$ is found. Note that in this context

$k\ i\ j$ also contains the sequence $j\ k$. To find the patch of a node k, i.e, all elements containing node k, just get the non zero entries of the k-th row or column.

getEd4n.m

The function `getEd4n` returns a symmetric `[nrNodes nrNodes]` sparse matrix. *Note*: The sparse constant is bounded due to the used mesh generation. The input is `e4n` generated by the function `getE4n`. The output matrix `ed4n` contains the numbers of the edges between two nodes or zero if the two nodes are not on one edge. In the sense that for node j and node k the entry (j, k), and (k, j) respectively, is the corresponding edge number. The numbering of the edges is arbitrarily generated in this function. The input is `n4e`.

getN4ed.m

The function `getN4ed` returns a `[nrEdges 2]` matrix. the input is `ed4n` generated by the function `getEd4n`. The output matrix contains in each row the number of the two nodes that are the endpoints of the edge corresponding to the row number.

getEd4e.m

The function `getEd4e` returns a `[nrElems 3]` matrix. The input is `n4e` and `ed4n` produced by the function `getEd4n`. This matrix contains in each row j the number of the three edges of element j. The edge numbers are the ones generated in the function `getEd4n`. The edges are ordered counter clockwise beginning with the edge between the first and the second node in `n4e`.

getE4ed.m

The function `getE4ed` returns a `[nrEdges 2]` matrix. The input is `e4n` and `n4ed` produced by the functions `getE4n` and `getN4ed`, respectively. The matrix contains in each row j the element numbers of the elements which share the edge. If the edge is a boundary edge the second entry is zero.

getArea4e.m

The function `getArea4e` returns a `[nrElems 1]` matrix. The input is `n4e` and `c4n`. The matrix contains in each row j the area of the element corresponding to the element number j.

getArea4n.m

The function `getArea4n` returns a `[nrNodes 1]` matrix. The input is `e4n` and `area4e` produced by the functions `getE4n` and `getArea4e`, respectively. The matrix contains in each row j the area of the patch of node j.

getLength4ed.m

The function `getLength4ed` returns a `[nrEdges 1]` matrix. The input is `c4n` and `n4ed` produced by the function `getN4ed`. The matrix contains in each row j the length of the j-th edge, according to the edge numbers created in `getEd4n`.

getDbEdges.m

The function `getDbEdges` returns a `[nrDbEdges 1]` matrix. The input is `Db` and `ed4n` produced by the function `getEd4n`. The Matrix contains the edge numbers of the edges belonging to the Dirichlet boundary.

getNbEdges.m

The function `getNbEdges` returns a `[nrNbEdges 1]` matrix. The input is `Nb` and `ed4n` produced by the function `getEd4n`. The Matrix contains the edge numbers of the edges belonging to the Neumann boundary.

getNormals4DbEd.m

The function `getNormals4DbEd` returns a `[nrDbEdges 2]` matrix. The input is `c4n` and `Db`. The matrix contains in each row j the two coordinates of the outer unit normal at the j-th Dirichlet edge, corresponding to the order in `DbEdges`.

getNormals4NbEd.m

The function `getNormals4NbEd` returns a `[nrNbEdges 2]` matrix. The input is `c4n` and `Nb`. The matrix contains in each row j the two coordinates of the outer unit normal at the j-th Neumann edge, corresponding to the order in `NbEdges`.

getNormals4e.m

The function `getNormnals4e` returns a `[3 2 nrElems]` matrix. The input is `c4n`, `n4e` and `length4ed` and `ed4e` produced by the functions `getLength4ed` and `getEd4e`. The three dimensional matrix contains in the j-th 3×2 matrix the coordinates of the unit outer normals for the three edges of element j. The order of the three rows corresponds two the order in `ed4e`.

getTangents4e.m

The function `getNormnals4e` returns a `[3 2 nrElems]` matrix. the input is `c4n`, `n4e` and `length4ed` and `ed4e` produced by the functions `getLength4ed` and `getEd4e`. The three dimensional matrix contains in the j-th 3×2 matrix the coordinates of the unit tangents in counter clockwise direction for the three edges of element j, i.e., the direction of the edges. The order of the three rows corresponds two the order in `ed4e`.

getMidpoint4e.m

The function `getMidpoint4e` returns a `[nrElems 2]` matrix. The input is `c4n` and `n4e`. The matrix contains in each row j the coordinates of the midpoint of element j.

getMidpoint4ed.m

The function `getMidpoint4ed` returns a `[nrEdges 2]` matrix. The input is `c4n` and `n4ed`. The matrix contains in each row j the coordinates of the midpoint of the j-th edge.

getAngles4e.m

The function `getAngles4e` returns a `[3 nrElems]` matrix. The input is `tangents4e` produced by the function `getTangente4e`. The matrix contains in each column j the three interior angles of element j. The order of the angles correspond to the order of the nodes in `n4e`, i.e. the angles at the first node of each element are in the first row, the one at the second in the second etc.

getAngle4n.m

The function `getAngle4n` returns a `[nrNodes 1]` matrix. The input is `angles4e`, `n4e`, `nrElems` and `nrNodes` where `angles4e` is generated by `getAngles4e`. The matrix contains in each row j the angle at the node j, where the angle is 2π for inner nodes, and the angle inside the domain and between the two boundary edges containing the node for nodes at the boundary.

getGrad4e.m

The function `getGrad4e` returns a `[3 2 nrElems]` matrix. The input is `c4n`, `n4e` and `area4e` produced by the function `getArea4e`. The three dimensional matrix contains in the j-th 3×2 matrix the coordinates of the local gradients of the three P_1 basis function which are not constantly zero on the j-th element. The first row

in each 3×2 matrix contains the local gradient of the nodal basis function for the first node of the element according to the order in `n4e`, the second row the one of the nodal basis function for the second node etc. .

getGradNC4e.m
The function `getGrad4e` returns a `[3 2 nrElems]` matrix. The input is `c4n`, `n4e` and `area4e` produced by the function `getArea4e`. The three dimensional matrix contains in the j-th 3×2 matrix the coordinates of the local gradients of the three P_1^{NC} basis function which are not constantly zero on the j-th element. The first row in each 3×2 matrix contains the local gradient of the basis function corresponding to the first edge in `ed4e`, the second row the one of the second edge etc. .

6. PDE SOLVER

In table 5 the implemented finite element methods are described. Each implemented finite element method has the same folder and file structure in the FFW

```
Folder: .\PDEsolvers\<pdeType>\<pdeSolver>-<pdeType>\
```

```
Files:  <pdeType>createLinSys.m
        <pdeType>enumerate.m
        <pdeType>estimate.m
        <pdeType>init.m
        <pdeType>postProc.m
```

The different finite element methods are initialized by setting the parameter `pdeSolver` in the `initFFW` function call, refer to Section 2.3.

To get a better understanding of what is done in the different files in the following the necessary files for the P_1 finite element method are described in detail. They are located at `.\PDEsolvers\Elliptic\P1-Elliptic\`.

6.1. P1createLinSys. `po = P1createLinSys(pi)`

This function only acts on the last level. Creates the stiffness matrix and the load vector b for the right hand side.

needed data in the input structure pi

`pi.level(end).enum.···`	enumeration for the mesh in the last level for example this data is created by the method `P1enumerate`
`pi.level(end).geom.···`	geometry data for the mesh in the last level for example this data is created by the method `p.statics.refine`
`pi.problem.···`	problem data which specifies coefficients in the PDE and the given boundary Values
`pi.params.` `rhsIntegtrateExactDegree`	an integer which specifies the exactness

	of the approximation of the integrals for the right hand side

output structure po

`po.level(end).A`	stiffness matrix for the last level
`po.level(end).b`	right hand side for the last level
`po.level(end).B`	mass matrix, for future usage
`po.level(end).x`	dummy for solution of linear system
`po.`$\cdots$	every thing else then the above data is identical to the data of the structure `pi`

By using a call like `p = P1createLinSys(p)` the given structure `p` which contains all problem data is changed after the call.

6.2. **P1enumerate.** `po = P1enumerate(pi)`

This function only acts on the last level. It creates enumeration data for edges, elements, nodes and additional data like the local gradients and the area for each element and edge.

needed data in the input structure pi

`pi.level(end).geom.`$\cdots$	geometry data for the mesh in the last level
`pi.params.rhsIntegtrateExactDegree`	an integer which specifies the exactness of the approximation of the integrals which occur

output structure po

`po.level(end).enum.`$\cdots$	enumeration data for the last mesh
`po.`$\cdots$	every thing else then the above data is identical to the data of the structure `pi`

By using a call like `p = P1enumerate(p)` the given structure `p` which contains all problem data is changed after the call.

6.3. **P1estimate.** `po = P1estimate(pi)`

This function only acts on the last level. This function computes a residual based error estimator. The resulting data is usually used by the marking algorithm.

needed data in the input structure pi

`pi.level(end).u4e.`	data of the solution which is computed by postprocessing with `P1postproc`
`pi.problem.···`	problem data which specifies the PDE
`pi.level(end).geom.···`	geometry data for the mesh in the last level
`pi.level(end).enum.···`	enumeration data for the mesh in the last level

output structure po

`po.level(end).etaT···`	double valued m by 1 vector with m=number of elements elementwise approximation of local energy error
`po.level(end).estimatedError···`	one number which represents the approximation of the overall energy error in the whole domain
`po.···`	every thing else then the above data is identical to the data of the structure `pi`

By using a call like `p = P1estimate(p)` the given structure `p` which contains all problem data is changed after the call.

6.4. **P1init.** `po = P1init(pi)`

This function creates the function handles for the evaluation of the basis functions, their gradients and second derivatives.

output structure po

p.statics.basis	function handle to basis functions
p.statics.gradBasis	function handle to gradient of basis functions
p.statics.d2Basis	function handle to second derivatives of basis functions

By using a call like `p = P1postproc(p)` the given structure `p` which contains all problem data is changed after the call.

6.5. **P1postProc.** `po = P1postproc(pi)`

Elliptic	
'P1'	Conforming P_1 finite element method. Solution is piecewise affine and globally continuous.
'CR'	Non-conforming P_1 finite element method. Solution is piecewise affine and continuous in the normal directions of the midpoints of edges bit discontinuous in the midpoints in tangential direction. Note that $P_1^{NC} \subseteq H^1(\mathcal{T})$ but in general $H^1(\Omega) \subset H^1(\mathcal{T})$.
'P2'	Higher order conforming P_2 finite element method using quadratic basis functions.
'P3'	Higher order conforming P_3 finite element method using cubic basis functions.
'P0P1'	Unstable mixed finite element method.
'RT0P0'	Raviart-Thomas mixed finite element method. The gradients are piecewise affine and continuous in their normal component. Note that $RT^0(\mathcal{T}) \subseteq H(\mathrm{div};\Omega)$.

Ellasticity	
'P1P1'	A standard conforming mixed $P_1 - P_1$ finite element method for elasticity problems.
'P1CR'	A non-conforming and locking free $P_1 - P_1^{NC}$ finite element method for elasticity problems.
'AW'	A mixed, higher order, locking free finite element method for elasticity problems, by Arnold and Winther.

TABLE 5. Overview of the implemented finite element methods

This function only acts on the last level. This function computes additional data out of the solution, of the linear system.

needed Data in the input structure pi	
pi.level(end).x	solution from the linear system in last level
pi.problem.enum.$\cdots$	enumeration data of the last mesh

output structure po	
po.level(end).u	double valued m by 1 vector with m=number number of nodes represents the galerkin approximation
po.level(end).u4e	double valued m by 3 matrix with m=number number of elements represents local values of u_h for each element
po.level(end).gradU4e	double valued m by 2 matrix which represents the gradient of u_h on each element
po.$\cdots$	every thing else then the above data is identical to the data of the structure pi

By using a call like `p = P1postproc(p)` the given structure `p` which contains all problem data is changed after the call.

7.2.1. *uniform.*
File: `.\algorithms\mark\uniform.m`

All elements are refined uniformly *red*. Therefore $\mathcal{M}_\ell = \mathcal{E}_\ell$ and no elements are marked for *bisec5* refinement.

```
refineEdges = true(nrEdges,1);
refineElemsBisec5 = false(nrElems,1);
```

7.2.2. *maximum.*
File: `.\algorithms\mark\maximum.m`

The maximum algorithm defines the set $\mathcal{M}_\ell \subseteq \mathcal{E}_\ell$ of marked edges such that for all $E \in \mathcal{M}_\ell$

$$\eta_E > \theta \cdot \max_{K \in \mathcal{E}_\ell} \eta_K,$$

where $\theta \in [0,1]$ is a constant (default: $\theta = 0.5$). The MATLAB code is printed in the next line.

```
refineEdges = (etaEd > thetaEd * max(etaEd));
```

`True` means that the edge is marked and `false` it is not. If elements should be refined all the edges of an element T belong to the set $\mathcal{M}_\ell$ if

$$\eta_T > \theta \cdot \max_{K \in \mathcal{T}_\ell} \eta_K.$$

The code realizes this by first marking the elements and than marking all the edges of marked elements.

```
I = (etaT > thetaT * max(etaT))';
refineElems(I) = true;
refineEdges4e = ed4e(refineElems,:);
refineEdges(refineEdges4e(:)) = true;
```

Elements that have large osscilations (`etaOsc`) are treated analogously. The reason why they are refined with *bisec5* instead of *red* is that *bisec5* generates a new node in the interior of the element.

The maximum algorithm can be very inefficient. Suppose we have the same error on each edge except of very few edges and on those edges the error is very large. Then the maximum algorithm might only refine these few edges (cf. figure 7.1).

7.2.3. *bulk.* The bulk algorithm defines the set $\mathcal{M}_\ell$ of marked edges such that

$$\sum_{E \in \mathcal{M}_\ell} \eta_E^2 \geq \theta \cdot \sum_{K \in \mathcal{E}_\ell} \eta_K^2,$$

or it contains all the edges of elements $T \in \mathcal{K}_\ell$ that satisfy

$$\sum_{T \in \mathcal{K}_\ell} \eta_T^2 \geq \theta \cdot \sum_{K \in \mathcal{T}_\ell} \eta_K^2,$$

It is important to see that the set $\mathcal{M}_\ell$ of edges that are selected by this condition is not unique. Here we use an greedy approach. We iteratively take those edges with the largest error. Using this approach guarantees that we have the smallest possible set of marked edges that satisfy the bulk condition. In the following the corresponding MATLAB lines are printed.

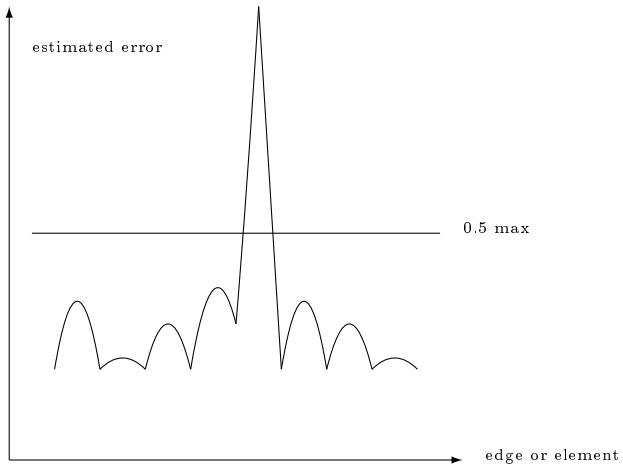


FIGURE 7.1. The maximum algorithm might mark only a few edges.

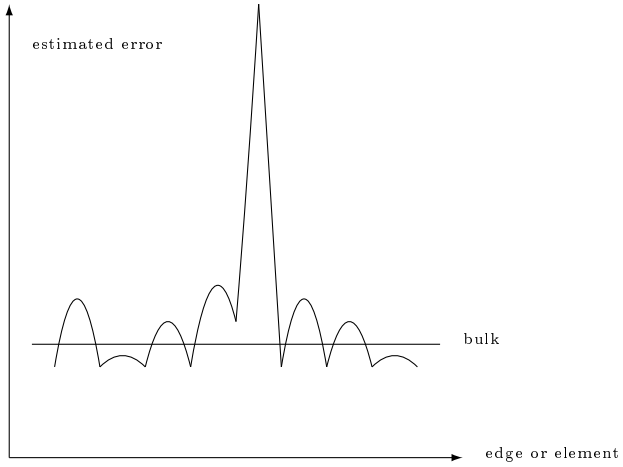


FIGURE 7.2. With the bulk algorithm it cannot happen that only a few edges are marked.

```
[sortedEtaEd,I] = sort(etaEd,'descend');
sumEtaEd = cumsum(sortedEtaEd.^2);
k = find(sumEtaEd >= thetaEd * norm(etaEd,2)^2,1,'first');
[sortedEtaT,I] = sort(etaT,'descend');
sumEtaT = cumsum(sortedEtaT.^2);
k = find(sumEtaT >= thetaT * norm(etaT,2)^2,1,'first');
refineElems(I(1:k)) = true;
refineEdges4e = ed4e((refineElems | refineElemsBisec5),:);
refineEdges(refineEdges4e(:)) = true;
```

The problem that only a few edges or element are marked like with the maximum algorithm cannot happen here, see Figure 7.2.

7.2.4. *graded.* **file:** `.\algorithms\mark\graded.m`

Refinement with graded grids is a a priori mesh refinement toward singular corner points. In the past the so called β-graded grids were very popular. The a priori analysis consists of the following theorem.

Theorem 7.1. *Let $\mathcal{T}$ be a regular triangulation of $\Omega = T_{ref}$ such that for given $N \in \mathbb{N}$ and $\beta > 0$*

(i) $\mathcal{T}$ contains the element $T_0 = conv\{(0,0),(N^{-\beta},0),(0,N^{-\beta})\}$.
(ii) For each $T \in \mathcal{T}\backslash\{T_0\}$ and all $x \in T$ one has $diam(T) \leq c\frac{1}{N}|x|^{1-\beta}$

Then, if $\alpha + \beta > 2$, it follows that

$$||\nabla(u_\alpha - I_{\mathcal{T}} u_\alpha)||_{L^2(\Omega)} \leq cN^{-min\{1,\alpha\beta\}} .$$

where u_α is the corner singularity function and α depends on the opening angle in the corner singularity.

It is known that β-graded grids satisfy this conditions. But here we follow another approach. We simply refine our mesh with *red-green-blue* refinement until these conditions are satisfied. This leads to simpler algorithms since β-graded grids are difficult to implement. Another advantage is that the angles of the triangulation only depend on the initial triangulation. The parameter β can be modified by setting the value

```
p.params.modules.mark.graded.beta
```

For the L-shaped domain the default value 1/3 is optimal. It is important to say that the implemented graded algorithm assumes that the problem has exactly one singularity located at the origin.

7.3. **Closure.** In the process of generating adaptive meshes you have to be careful that the angles of the element are bounded due to the maximum angle condition. To guarantee this you additionally have to refine all reference edges of elements that have marked edges, i.e. compute the smallest subset $\widehat{\mathcal{M}}_\ell$ of $\mathcal{E}_\ell$ which includes $\mathcal{M}_\ell$ such that there holds

$$E \in \widehat{\mathcal{M}}_\ell, \;\; E \subseteq T \Longrightarrow E(T) \in \widehat{\mathcal{M}}_\ell.$$

This is done in the function closure.

File: `.\algorithms\misc\closure.m`

```
function p = closure(p)
input:   p - FFW
output:  p - FFW
```

We briefly say that although the following code can have quadratic runtime it has linear runtime in average. From convergence theory we know that for a sequence of triangulations the number of the additionally refined reference edges are linear in the number of levels. Therefor we know that it is in average constant at each level. Because of that the while loop will be called in average for a constant number of times.

```
I =  refineEdges(ed4e(:,2)) | refineEdges(ed4e(:,3));
while nnz(refineEdges(refEd4e(I))) < nnz(I);
   refineEdges(refEd4e(I)) = true;
   I =  refineEdges(ed4e(:,2)) | refineEdges(ed4e(:,3));
end
```

7.4. **Refine.** Given a triangulation $\mathcal{T}_\ell$ on the level ℓ, let $\mathcal{E}_\ell$ denote its set of interior edges and suppose that $E(T)$ $(E(T) : T \in \mathcal{T}_\ell)$ denotes the given reference edges. There is no need to label the reference edges $E(T)$ by some level ℓ because $E(T)$ will be the same edge of T in all triangulations $\mathcal{T}_m$ which include T. However, once T in $\mathcal{T}_\ell$ is refined, the reference edges will change too. After the closure algorithm each element has either $k = 0, 1, 2$ or 3 of its edges marked for refinement and because of the closure algorithm the reference edge belongs to it if $k \geq 1$. Therefore, exactly one of the four refinement rules of Figure 7.3 is applied. This specifies sub triangles and their reference edges in the new triangulation $\mathcal{T}_{\ell+1}$. In general there are four different cases to refine an element. Elements with no marked edges are not refined, elements with one marked edge are refined *green*, elements with two marked edges are refined *blue* and elements with tree marked edges are refined *red*. Where *blue* refinement is divided into the two cases *blueleft* and *blueright*.

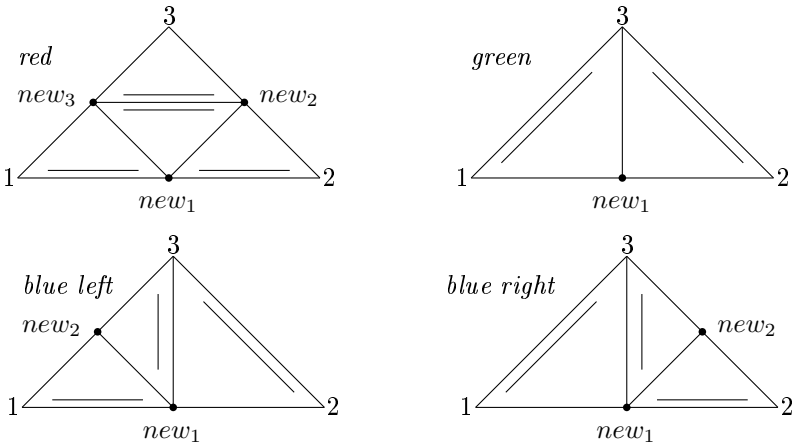


FIGURE 7.3. *Red*, *green* and *blue* refinement. The new reference edge is marked through a second line in parallel opposite the new vertices new_1, new_2 or new_3.

All elements that are marked in refineElemsBisec5 are refined *bisec5* instead of *red*, see Figure 7.4.

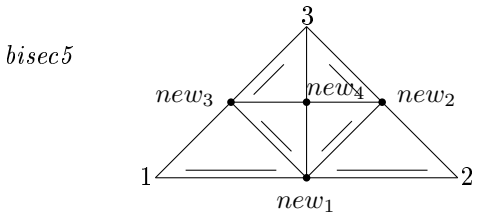


FIGURE 7.4. *bisec5* refinement. The new reference edge is marked through a second line in parallel opposite the new vertices new_1, new_2, new_3 and new_4.

The red-green-blue refinement is implemented in
File: `.\algorithms\refine\redGreenBlue.m`

```
function p = redGreenBlue(p)
% input:   p - FFW
% output:  p - FFW
```

At first we create the new c4n. Therefore we take the old c4n and add the new coordinates at the end of the list. The new coordinates are the midpoints of the marked edges.

```
% Create new node numbers from refineEdges
newNode4ed = zeros(1,nrEdges);
newNode4ed( find(refineEdges) ) = ...
     (nrNodes+1):(nrNodes+nnz(refineEdges));
% Create coordinates of the new nodes
[dontUse,J,S] = find(newNode4ed);
c4n(S,:) = midPoint4ed(J,:);
```

In the next step the new n4e is build. At first we calculate the number of marked edges for each element. All the Elements that will not be refined, e.g. have no marked edge, are fist copied to the new n4e. In the following new elements will be appended to the list.

```
newNode4e = newNode4ed(ed4e);
unrefinedElems = find( all(newNode4e == 0 ,2) );
refineElems = find( any(newNode4e,2) );
nrMarkedEd4MarkedElems = sum(refineEdges( ed4e(refineElems,:) ),2);
newn4e = n4e(unrefinedElems,:);
```

All elements that are to be *red* or *green* refined can be refined simultaneously. In the case of *green* refinement, it is important to know that the first edge of an element is always the reference edge, therefore the marked edge, and that all elements have math. positive orientation, i.e. counter clockwise. Therefore there is only one way to refine an element *green*. Instead we have two different cases with *blue* refinement. Therefore we distinguish between *blueleft* and *blueright* refinement. Again there is only one way to perform *red* refinement. For example the MATLAB code for *green* refinement is printed below. The new elements are $T_1 = \mathrm{conv}(2, 3, new_1)$ and $T_1 = \mathrm{conv}(3, 1, new_1)$. The new reference edge are the edges between the nodes $2, 3$ and $3, 1$.

```
I = find(nrMarkedEd4MarkedElems == 1);
if ~isempty(I)
  gElems = refineElems(I);
  [dontUse,dontUse,newN] = find( newNode4e(gElems,:)' );
  newGreenElems = [n4e(gElems,[2 3]) newN;...
                   n4e(gElems,[3 1]) newN];
  newn4e = [newn4e;newGreenElems];
end
```

At the end the lists for the boundary, Db and Nb, are updated.

```
Db = updateBoundary(Db,DbEd,newNode4ed);
Nb = updateBoundary(Nb,NbEd,newNode4ed);
...
function newBoundary = updateBoundary(oldB,ed4b,newNode4ed)
if(isempty(oldB))
    newBoundary = [];
else
  unrefinedEd = find(~newNode4ed(ed4b));
  refineEd = find(newNode4ed(ed4b));
  newBoundary = [oldB(unrefinedEd,:);...
    oldB(refineEd,1) newNode4ed(ed4b(refineEd))' ;...
    newNode4ed(ed4b(refineEd))' oldB(refineEd,2)];
end
```

7.5. **Properties of the meshes.** This subsection lists a few results on the triangulation $\mathcal{T}_\ell$ obtained by *red-green-blue* refinement under the assumptions on $\mathcal{T}_0$ of subsection 7.1. The non-elementary proofs can be found in [Car04].

(i) $\mathcal{T}_\ell$ is a regular triangulation of Ω into triangles; for each $T \in \mathcal{T}_\ell$ there exists one reference edge $E(T)$ which depends only on T but not on the level ℓ.

(ii) For each $K \in \mathcal{T}_0$, $\mathcal{T}_\ell|_K := \{T \in \mathcal{T}_\ell \,|\, T \subseteq K\}$ is the picture under an affine map $\Phi : K \to T_{ref}$ onto the reference triangle $T_{ref} = \mathrm{conv}\{(0,0),(0,1),(1,0)\}$ by $\Phi(E(K)) = \mathrm{conv}\{(0,0),(1,0)\}$ and $\det D\Phi > 0$. The triangulation $\widehat{T}_K := \{\Phi(T) : T \in \mathcal{T}, T \subseteq K\}$ of K consists of right isosceles triangles. (A right isosceles triangle results from a square halved along a diagonal.)

(iii) The L^2 projection onto V_ℓ is H^1 stable, in the sense that $V_\ell := \mathcal{P}_1(\mathcal{T}_\ell)$ denotes the piecewise affine space, i.e.

$$\mathcal{P}_1(\mathcal{T}_\ell\mathbb{R}^m) := \{v \in C^\infty(T;\mathbb{R}^m) : v \text{ affine on } T\}\,,$$
$$\mathcal{P}_1(\mathcal{T}_\ell\mathbb{R}^m) := \{v \in L^\infty(\Omega;\mathbb{R}^m) : \forall T \in \mathcal{T}_\ell, v|_T \in \mathcal{P}_1(T;\mathbb{R}^m)\}\,.$$

For any $v \in H^1_0(\Omega)$ the L^2 projection Πv on V_ℓ satisfies

$$\|\nabla \Pi v\|_{L^2(\Omega)} \le C_1 \|\nabla v\|_{L^2(\Omega)} \text{ and } \|h_{\mathcal{T}}^{-1}\Pi v\|_{L^2(\Omega)} \le C_2 \|h_{\mathcal{T}}^{-1} v\|_{L^2(\Omega)}.$$

The constants C_1 and C_2 exclusively depend on $\mathcal{T}_0$.

(iv) Approximation property of the L^2 projection

$$\sum_{T\in\mathcal{T}_\ell} \|h_T^{-1}(v-\Pi v)\|^2_{L^2(T)} + \sum_{E\in\mathcal{E}_\ell} \|h_E^{-1/2}(v-\Pi v)\|^2_{L^2(E)} \le C_3 \|\nabla v\|^2_{L^2(\Omega)}.$$

8. Graphical Output

A very convenient way of getting graphical output of any kind from the structure `p` is to use the script `show.m` located at `.\evaluation\`. The function is called with a token, defining what is to be drawn, and the structure `p`, i.e., `p = show('token',p);`. Here *token* is to be replaced by one of the following options:

token	Description
`drawU`	Draw the solution u on the underlying grid of each level of the refinement, saved in structure `p`.
`drawGradU`	Draw the gradient vector field ∇u on the underlying grid of each level of the refinement, saved in structure `p`.
`drawError`	Draw the error development of the discrete solution, as saved in structure `p`. Here one has to specify which error is to be drawn by using the parameters `_estimatedError`, `_L2error` or `_H1semiError` which are added to `drawError`, i.e., `p = show('drawError_L2error',p);`.
`drawErrorOnGrid`	Draw the local error indicators of the discrete solution, as saved in structure `p`. Here one has to specify which error is to be drawn by using the parameters `_estimatedError`, `_L2error` or `_H1semiError` which are added to `drawError`, i.e., `p = show('drawError_L2error',p);`.
`drawGrid`	Draw the grid of each level of the refinement, saved in structure `p`.

Be aware that `drawU` and `drawGradU` depend on the PDE solver used, i.e., these functions can be found in the `script` folder of the `PDEsolver` used; whereas `drawGrid`

and `drawError` are generic.

For all of the above tokens, the `show` function can display the data iteratively from the first level to the last or just show the last level of the refinement by setting `p.params.output.showIteratively` to `true` or `false`, default is `false`. The parameter `p.params.output.pauseTime` defines the pause time between two plots, the value -1 forces a pressed key to continue.

To save the figures, set `p.params.output.saveFigures` to `true`. The resulting files can be found in a subfolder of `.\results\`, which describes the current problem. If you have chosen to show the data iteratively, a figure for each level is saved, except for the `drawError` token, where the error development from the first to the last level is saved in one figure.
The default file type is `fig`. See the following table for more parameters or MATLAB's help for the command `print`.

file type	description
`fig`	Saves a MATLAB figure.
`jpeg`	Saves a JPEG file.
`epsc`	Saves an encapsulated postscript color file.
`epsc2`	Saves an encapsulated postscript level 2 color file.
`png`	Saves a png file.

There are a number of predefined parameters to customize the graphical output for any of the above *token*. Those parameters are also set by `defaultParametersOutput` during initialization of the FFW .

Parameters for `drawU`:

parameter	type	description
`drawInfo`	boolean	Print labels describing the problem, e.g., caption and degree of freedom. Default is `true`.
`drawWalls`	boolean	Relevant only when using the PDE solver `P1P0` and `RT0P0`. Default is `true`.
`myColor`	char	Color of the solution on the grid. Relevant only when solving elasticity problems. Default is `'k'` for black. For the color coding see MATLAB's help.
`lineWidth`	integer	Line width of the solution. Relevant only when solving elasticity problems. Default is `1`.
`factor`	integer	Scales the solution by this factor. Relevant only when solving elasticity problems. Default is `1000`.

Parameters for `drawGradU`:

parameter	type	description
`drawInfo`	boolean	Print labels describing the problem, e.g., caption and degree of freedom. Default is `true`.
`localRes`	integer	Determines the length of the gradients drawn. Default is `10`.

Parameters for `drawError`:

parameter	type	description
`drawInfo`	boolean	Print labels describing the problem, e.g., caption and degree of freedom. Default is `true`.
`myColor`	char	Color of the graph. Default is `'k'` for black. For the color coding see MATLAB's help.
`lineStyle`	char	Determines the line style. Default is `'-'`. For different styles see MATLAB's help.
`marker`	char	Determines the marker style. Default is `'x'`. For different styles see MATLAB's help.
`minDoF`	integer	Minimal number of degrees of freedom to start the plot with. Default is `1`.
`plotGrid`	boolean	Determines whether the axis grid is shown. Default is `true`.
`holdIt`	boolean	Determines whether the graph is to be appended to the current figure or the figure is cleared before drawing. Default is `true`.
`name`	string	Append the submitted name to the legend. Default is `[]`.
`fontSize`	integer	Size of all fonts in the figure. Default is the set font size of the figure.
`setScales`	boolean	Round the log-log plot to integer exponents. Default is `false`.
`getConvRate`	boolean	Calculate the convergence rate and save it in the structure. Default is `true`.

`drawConvRate`	boolean	Draw the convergence rate in the figure (`getConvergenceRate` does not necessarily have to be `true`). Default is `false`.
`degree`	integer	Set the order of the integration routines of the error. Increased value means more accuracy, decreased value means more performance. Default is `19`.

Parameters for `drawGrid`:

parameter	type	description
`drawInfo`	boolean	Print labels describing the problem, e.g., caption and degree of freedom. Default is `true`.
`color`	char	Color of the grid. Default is `'k'` for black. For the color coding see MATLAB's help.
`lineWidth`	integer	Determines the line width of the grid. Default is `1`.

9. Appendix

9.1. **Gauss Quadratur.** To integrate the right hand side of the partial differential equation

$$\text{rhs} = \int_\Omega f\varphi \, dx + \int_{\Gamma_N} g\varphi \, dx$$

we need to use quadrature formulas. On the one hand we have to calculate integrals in 1D for the neumann boundary and on the other hand we have to solve integrals in 2D numerically. In order to do this and for integration various types of integrals we developed the following interface.

9.1.1. *Interface integrand.*
File: `.\algorithms\integrate\integrand.m`

```
function val = integrate(parts, curLvl,  degree, integrand, p)
% input:  parts - nodes for elements or edges
%         curLvl - current level
%         degree - the integration will be exact for all polynomials
%                  up to total degree 'degree'
%         p - FFW
% output: val - values of the integrand per element or edge [nrParts n m]
```

The integration interface can also handle functions $f : \mathbb{R}^k \mapsto \mathbb{R}^{m,n}$ in the sense that it integrates each component separately.
The function handle integrand has to be of the following form.

```
function val = integrand(x,y,curPart,curLvl,p)
```

Where the output has to be of the form `[n m length(x)]`.
For the integration of the right hand side there already exist two function handles.

File: `.\algorithms\integrate\funcHandleRHSVolume.m`
File: `.\algorithms\integrate\funcHandleRHSNb.m`

In both cases the function handle returns a matrix of the form `[nrElements nrBasisFunctions]` and the parameter `degree` is stored in

```
p.params.rhsIntegtrateExactDegree
```

The `degree` parameter for the predefined function handles to calculate the energy and L^2 errors can be modified in

```
p.params.errorIntegtrateExactDegree
```

The integration interface uses the following quadrature formulas.

9.1.2. *Gauss-Legendre formula.*
File: `.\algorithms\integrate\getGaussPoints.m`

```
function [x,w] = getGaussPoints(n)
% input:   n - number of points
% output:  x - Gauss points [n 2]
%          w - Gauss weights [n 1]
```

We want to integrate numerically a given function over the reference edge $\mathrm{conv}\{0,1\}$

$$I(f) = \int_0^1 \omega(x) f(x)\, dx$$

The integration formula has the form

$$\widetilde{I}(f) := \sum_{i=1}^{n} \omega_i f(x_i)\ ,$$

where ω_i are the weights and x_i are the Gauss points.
The Gauss points are the roots of orthogonal polynomials. The weights are chosen such that the formula is optimal for all polynomials up to total degree $p = 2n-1$. When we choose $\omega(x) = 1$ and the interval $[-1,1]$ we get the Gauss-Legendre formula, where the Gauss points are the roots of the n-th Legendre polynomial.
The following two theorems show how to calculate the Gauss points x_i and weights ω_i efficiently for arbitrary $n \in \mathbb{N}$.

Theorem 9.1. *The roots x_i, $i = 1,\dots,n$ of the n-th orthogonal polynomial p_n are the eigenvalues of the tridiagonal matrix*

$$J_n := \begin{bmatrix} \delta_1 & \gamma_2 & & & \\ \gamma_2 & \delta_2 & . & & \\ & . & . & . & \\ & & . & . & \gamma_n \\ & & & \gamma_n & \delta_n \end{bmatrix}.$$

Where the coefficients are recursively defined by δ_i, γ_i

$$\delta_{i+1} := (xp_i, p_i)/(p_i, p_i) \quad \text{for } i \geq 0$$

$$\gamma_{i+1}^2 := \begin{cases} 1 & \text{for } i = 0 \\ (p_i, p_i)/(p_{i-1}, p_{i-1}) & \text{for } i \geq 1 \end{cases}.$$

For the Gauss-Legendre formula the coefficients are

$$\delta_{i+1} = 0 \ \text{ for } i \geq 0 \text{ and } \gamma_{i+1} = \frac{i}{\sqrt{4i^2-1}} \ \text{ for } i \geq 1\ .$$

Therefore we can calculate the Gauss points x_i with the following matlab lines.

```
gamma = [1 : n-1] ./ sqrt(4*[1 : n-1].^2 - ones(1,n-1) );
[V,D] = eig( diag(gamma,1) + diag(gamma,-1) );
x = diag(D);
```

Theorem 9.2. *It holds* $w_k = (v_1^{(k)})^2$, $k = 1, \ldots, n$, *if* $v^{(k)} = (v_1^{(k)}, \ldots, v_n^{(k)})$ *are the eigenvectors to the eigenvalue* x_k *of* J_k *with the norm factor* $|v^{(k)}| = \int_a^b \omega(x)\, dx$.

For the Gauss-Legendre-formula there holds $\int_a^b \omega(x)\, dx = \int_{-1}^1 1\, dx = 2$. E.g. we must multiply the weights with factor 2.

```
w = 2*V(1,:).^2;
```

After that we have to transform the interval $[-1, 1]$ to the reference edge $[0, 1]$. Pay attention to the fact that there holds $\int_0^1 1\, dx = 1$ and therefore we must multiply the weights with factor $1/2$.

```
x = .5 * x + .5;
w = .5 * w';
```

9.1.3. *Conical-Product formula.*
File: `.\algorithms\integrate\getConProdGaussPoints.m`

```
function [x,w] = getConProdGaussPoints(n)
% input:   n - Number of Gauss points in 1D
% output:  x - Gauss points [n^2 2]
%          w - weights [n^2 1]
```

The Conical-Product formula is a quadrature formula especially for triangles. Here we use this formula for the reference triangle with the nodes $(0, 0), (0, 1)$ and $(1, 1)$. The Conical-Product formula is a combination of 1D Gauss points.
The composite formula for the quadrangle is simply the cartesian product of the 1D Gauss-Legendre formula. Therefor it is our aim to transform the quadrangle to an triangle by transforming the coordinates.

$$\begin{aligned} x_1 &= y_1 \\ x_2 &= y_2(1 - y_1) = y_2(1 - x_1)\ . \end{aligned}$$

From $0 \le x_1 \le 1$ and $0 \le x_1 \le 1 - x_2$ we conclude $0 \le y_i \le 1,\ i = 1, 2$. By this transformation we transformed the quadrangle to the triangle.
The Conical-Product formula is therefor a combination from the 1D Gauss formulas for the integrals

$$\begin{aligned} \int_0^1 f(r)\, dr &\approx \sum_{i=1}^n a_i f(r_i)\ , \\ \int_0^1 (1 - s) f(s)\, ds &\approx \sum_{j=1}^n b_j f(s_j)\ . \end{aligned}$$

The first integral can be calculated as above with the Gauss-Legendre formula. For the second integral we need a different Gauss formula. For the weight function $\omega(x) = (1 - x)$ the Gauss-Jacobi formula is the right choice. The roots of the

corresponding Jacobi polynomials can be calculated analogously to the roots of the Gauss-Legendre polynomials. The coefficients for the Jacobi polynomials are

$$\delta_i = \frac{-1}{4i^2-1} \quad \text{and} \quad \gamma_{i+1} = \sqrt{\frac{i(i+1)}{2(i+1)-1}} \; .$$

```
delta = -1./(4*(1 : n).^2-ones(1,n));
gamma = sqrt((2 : n).*(1 : n-1)) ./ (2*(2 : n)-ones(1,n-1));
[V,D] = eig( diag(delta)+diag(gamma,1)+diag(gamma,-1) );
s = diag(D);
b = 2*V(1,:).^2;
```

As above we have to transform the Gauss points to the interval $[0, 1]$. Therefor we must also multiply the weights of the Gauss-Jacobi formula with factor $1/4$.

```
r = .5 * r + .5;
a = .5 * a';
s = .5 * s + .5;
b = .25 * b';
```

The Conical-Product formula therefor has the n^2 Gauss points and weights

$$(s_j, r_i(1-s_j)) \quad a_i b_j \quad i = 1, \ldots, n \quad j = 1, \ldots, n \; .$$

```
s = repmat(s',n,1); s = s(:);
r = repmat(r,n,1);
x = [ s , r.*(ones(n^2,1)-s) ];
w = a*b';
w = w(:);
```

References

[Car04] Carsten Carstensen, *An adaptive mesh-refining algorithm allowing for an H^1 stable L^2 projection onto courant finite element spaces*, Constructive Approximation **20** (2004), 549–564. 30, 36

[Sch97] H.R. Schwarz, *Numerische Mathematik*, 4th ed., Teubner, Stuttgart, 1997.

[Sto99] J. Stoer, *Numerische Mathematik I*, 8th ed., Springer, Berlin, 1999.

[Str71] A.H. Stroud, *Approximate Calculations of Multiple Integrals*, Prentice-Hall, Englewood Cliffs, 1971.