P1-AFEM and Computational Geometry Codes

April 2014

"2D war gestern."

JOSEF F.
KEMETMÜLLER

Contents

1 Before you start

`setup` – Compile the sources for the 3D bisection scheme

The 3D bisection scheme provided by this package is programmed in C and needs to be compiled before use by calling the script **`setup`**. You can try out the bisection scheme by starting a 3D example, e.g. `exampleSimplex`.

2 Computational Geometry

2.1 The mesh data structure

Throughout the code we will be using the data structure `mesh`, which contains the arrays `elements` and `coordinates` describing the mesh by standard simplex-vertex format. Additionaly a strucure `mesh` can contain multiple boundaries `mesh.bd(j).elements` for some $j \geq 0$ given also in the simplex-vertex format with respect to the nodes in `coordinates`.

`genMesh` – Constructor for the mesh data structure

`mesh = genMesh(elements, coordinates, dirichlet, neumann)` will construct a mesh containing

- `mesh.elements == elements`,
- `mesh.coordinates == coordinates`,
- `mesh.bd(1).elements == dirichlet`,
- `mesh.bd(2).elements == neumann`.

`cleanMesh` – Removes duplicate and unused points from mesh

Some mesh generators yield data, which contains duplicate or unused points.
`cleanedMesh = cleanMesh(mesh)` removes those points and returns the cleaned mesh. A mesh does not have to be cleaned to be useful for other functions, but it may be a good idea to clean the initial mesh. This can be seen for example in the code for the solution of the poisson problem.

Listing 1: Excerpt from **`solvePoisson`**

```
dirichletBd = meshBd(mesh,1);
neumannBd = meshBd(mesh,2);
b = inner_L2_P1_L2(mesh,      f, [], quadDeg) ...
  + inner_L2_P1_L2(neumannBd, g, [], quadDeg) ...
  - A*discretizer(dirichletBd, uD);
```

`dirichletBd` and `neumannBd` will certainly not be “clean” considering the usage of all points in
`mesh.coordinates`, but this makes it more simple to add the three contributions of `b`, as their results will all be **`numCoordinates`**`(mesh)−by−1` vectors. Whereas cleaning the boundary meshes would yield vectors of the size **`numCoordinates`**`(mesh)−by−1`, **`numCoordinates`**`(neumannBd)−by−1` and **`numCoordinates`**`(dirichletBd)−by−1`, respectively.

`meshBd` – Mesh structure of the boundary

`bdj = meshBd(mesh, j)` is equivalent to `bdj = genMesh(mesh.bd(j).elements, mesh.coordinates)`. You can use this to call other functions expecting a mesh data structure as input. For example

`integrate(meshBd(mesh,1), f)`

can be used to integrate a function f on the Dirichlet boundary $\int_{\Gamma_D} f\,dx$.

`dimMesh` – Dimension of the mesh

`d = dimMesh(mesh)` returns the dimension of the mesh.

`dimSpace` – Dimension of the embedding space

`d = dimSpace(mesh)` returns the dimension of the embedding space.

`numElements` – Number of elements in the mesh

`nE = numElements(mesh)` returns the number of elements in the mesh.

`numCoordinates` – Number of nodes in the mesh

`nC = numCoordinates(mesh)` returns the number of nodes in the mesh.

`numBoundaries` – Number of boundaries

`nB = numBoundaries(mesh)` returns the number of boundaries, which are defined on the mesh.

`numBoundaryElements` – Number of elements in boundaries

`nBE = numBoundaryElements(mesh)` is an array containing the number of elements `nBE(j)` of `mesh.bd(j).elements`.

2.2 Geometrical computations

`getBarycenters` – Barycenters of elements

`centers = getBarycenters(mesh)` returns the barycenters of the elements of the mesh. You can use `bdCenters = getBarycenters(meshBd(mesh, j))` to get the barycenters of the boundary `mesh.bd(j)`.

`barycentricToCartesians` – Convert from barycentric to cartesian coordinates

`cartesians = barycentricToCartesians(mesh, barycentric)` yields an array `cartesians` of the size **`numElements`**-by-**`dimSpace`**, containing the points obtained by using the given barycentric coordinate as local coordinate on all the elements.

`getNormals` – Normal vectors of all elements

`normals = getNormals(mesh, 'outwards')` or `normals = getNormals(mesh, 'inwards')` returns the outwards/inwards facing normals of each elements hyperfaces.

`getElementVolumes` – **Volumes of all elements**

`volumes = getElementVolumes(mesh)` computes the volumes of all elements.

`getSignedElementVolumes` – **Signed volumes of all elements**

`volumes = getElementVolumes(mesh)` computes the signed volumes of all elements assuming a right handed coordinate system.

`getPatchVolumes` – **Volumes of all node patches**

`volumes = getPatchVolumes(mesh)` computes the volumes of all node patches.

`orientMesh` – **Orient a mesh**

`mesh = orientMesh(mesh, 'inwards')` or
`mesh = orientMesh(mesh, 'outwards')` will orient the elements of the mesh and their boundaries, so that its normal vectors will all be either facing inwards or outwards.

`orientMeshWithoutBoundary` – **Orient a mesh, leave boundaries untouched**

`mesh = orientMeshWithoutBoundary(mesh, 'inwards')` or
`mesh = orientMeshWithoutBoundary(mesh, 'outwards')` will orient the elements of the mesh, so that its normal vectors will all be either facing inwards or outwards. The boundary facets will not be changed.

2.3 Topological computations

`getBoundary` – **Topological boundary of the mesh**

`boundary = getBoundary(mesh)` returns the topological boundary of the mesh. Additionaly, the boundary can be oriented using
`boundary = getBoundary(mesh, 'outwards')` or
`boundary = getBoundary(mesh, 'inwards')`.

`getEdges` – **Edges of the mesh**

`[edge2nodes, element2edges, boundaryOne2Edges, ...] = getEdges(mesh)` returns the edges of the mesh. The rows `element2edges(i,:)` are indices into the rows of `edge2nodes` and will yield the `dimMesh*(dimMesh+1)/2` edges of the i-th element. The rows `boundaryOne2Edges(i,:)` contain the `dimMesh` edges of the boundary facet `mesh.bd.elements(1,:)`.

`getHyperfaces` – **Hyperfaces of the mesh**

`[face2nodes, element2faces, boundaryOne2Face, ... ] = getHyperfaces(mesh)` returns the hyperfaces of the mesh. The rows `element2faces(i,:)` are indices into the rows of `face2nodes` and will yield the `dimMesh+1` faces of the i-th element. The entry `boundaryOne2Face(i)` contains the index of the boundary hyperface `mesh.bd(1).elements(i,:)`.

`getSubsimplices` – Subsimplices of special type or ordering

`[subS2nodes, element2SubS, boundaryOne2SubS] = ...`
`getSubsimplices``(mesh, subsimplex, bdSubsimplex)` is the more flexible variant of **`getEdges`** and **`getHyperfaces`**. You can choose your own ordering of the array `element2SubS`.

```
[subS2nodes, element2SubS] = getSubsimplices(mesh, [2,3; 1,3; 1,2])
```

will result in `element2SubS(i,1)` being the edge `mesh.elements(i,[2,3])` Keep in mind though, that the sorting of the nodes in `subS2nodes` will always be ascending. You could use the functions **`simplexEdges`** and **`simplexHyperfaces`** as input for this function to get the functionality of **`getEdges`** and **`getHyperfaces`**.

`getNeighbors` – Elements sharing a common hyperface

`element2Neighbors = `**`getNeighbors`**`(mesh)` returns a **`numElements`**`–by–(`**`dimMesh`**`+1)` array containing the row indices of the neighbors of the elements. `element2Neighbors(i,j)` is the neighbor of the i-th element corresponding to its j-th face: `elements(i,(1:`**`dimMesh`**`+1)~=j)`.

`getElementsOfBoundary` – Elements attached to boundary facets

`[bdOne2element, bdTwo2element, ...] = `**`getElementsOfBoundary`**`(mesh)` returns arrays `bdX2element` containing indices describing to which element a certain boundary facet belongs. `mesh.elements(bdOne2element(i),:)` is the element which is attached to the boundary facet `mesh.bd(1).elements(i,:)`.

2.4 Refinement schemes

This package does offer refinement routines for 1D, 2D and 3D meshes. The 1D and 2D meshes will work for an arbitrary embedding space, whereas the 3D refinement scheme only works in $\mathbb{R}^3$. Especially for the three dimensional case it is very important that the mesh satisfies certain topological properties. To get a mesh fulfilling these properties the function **`genBisectionMesh`** can be used.

`genBisectionMesh` – Generate a mesh that can be used for bisection

`mesh = `**`genBisectionMesh`**`(mesh)` returns a mesh that can be used for adaptive refinement using bisection. In the 3D case the mesh will be finer than the original mesh. In the 1D and 2D case the mesh will remain the same. Coarsening of certain 2D configurations is not possible using an arbitrary mesh. To gain this ability you can force the generation of a finer 2D mesh, that has the property of being able to coarsen it to its initial state. For this you can use the option `mesh = `**`genBisectionMesh`**`(mesh, 'forceRefine')`.

`bisectionRefine` – Refine a mesh by bisection

`mesh = `**`bisectionRefine`**`(mesh, markedElements)` refines the elements `markedElements` via bisection, refining also other elements to retain the regularity of the mesh.

`bisectionCoarsen` – Coarsen a mesh refined by bisection

`mesh = `**`bisectionCoarsen`**`(mesh, markedElements)` tries to coarsen the elements markedElements Default behaviour is to only coarsen nodes, whose entire node patches have been marked

for coarsening. You can use
`mesh = bisectionCoarsen(mesh, markedElements, 'force')` to coarsen a node patch, even though only one of its elements has been marked. This however does only perform a single coarsening step, and especially only if the step is directly possible. If you want to mark all elements you can use
`mesh = bisectionCoarsen(mesh, 'all')`

`redRefine3D` – Uniform refinement of a 3D mesh

`mesh = redRefine3D(mesh)` refines a given mesh uniformly using red/regular refinement.

3 Differentiation

`getHatGrads` – Gradients of piecewise linear hat functions

`hatGrads = getHatGrads(mesh)` returns the gradients of the hat functions (nodal basis of the elementwise linear, globally continuous functions) on the simplicial mesh `mesh`. `hatGrads` is a `(dimMesh+1)`-cell array, so that `hatGrads{j}(i,:)` is the gradient of the hat function corresponding to the node `mesh.elements(i,j)` on the element `mesh.elements(i,:)`.

`gradP1` – Gradient of a piecewise linear function

`P0 = gradP1(mesh, P1)` returns the gradient of the function `P1` represented via the nodal basis as a `numCoordinates–by–dimP1` matrix, where `dimP1` is the dimension of the codomain of the function `P1`. `P0` is a `numElements–by–dimSpace*dimP1` array containing the gradients on all elements.

`hyperfaceJumpsP0` – Normal jumps of a piecewise constant function

`jumps = hyperFaceJumpsP0(mesh, P0)` returns an array `jumps` corresponding to the faces returned by `getHyperfaces`. `P0` is a `dimSpace`-vector-valued elementwise constant function. `jumps` is a function defined on the mesh's hyperfaces("skeleton", i.e. the array `face2nodes` returned by `getHyperfaces`) and is piecewise constant. For boundary faces it yields the outwards jump. `jumps` is NOT scaled by the areas of the hyperfaces. As the skeleton is also a simplicial complex you can use the integration/inner product routines to integrate the jumps, to get the the jumps scaled with their corresponding areas. This function can be used to compute the normal derivative of a function `P1` in the L^1 or L^2 norm.

```
Jhdudn = hyperfaceJumpsP0(mesh, gradP1(mesh,x));
skeleton = genMesh(face2nodes, mesh.coordinates);
normsL1 = integrateP0(skeleton, abs(Jhdudn), 'elementwise');
normsL2 = integrateP0(skeleton, Jhdudn.^2, 'elementwise');
normsL2 = inner_P0_P0_L2(skeleton, Jhdudn, Jhdudn, 'elementwise');
```

4 Integration

4.1 Pure integration

`simplexQuadratureRule` – Quadrature rule for single simplex

`[weights, points] = simplexQuadratureRule(dim, quadDeg)` returns quadrature weights and points in barycentric coordinates for a simplex of dimension `dim` using `quadDeg^dim` quadrature points. These quadrature rules are used for all the underlying numerical integrations of function handles done by the routines in this section.

`integrate` – $\int f\,dx$ – Integrate a function handle

`integral = integrate(mesh, f, quadDeg)` returns the integral over the given mesh of the function handle `f`. `quadDeg` is an integer representing the number of quadrature points used in the underlying tensor gaussian quadrature.
`integrals = integrate(mesh, f, quadDeg, 'elementwise')` returns a **`numElements`**`-by-dimf` array containing the elementwise integrals.
`integrals = integrate(mesh, f, quadDeg, 'patchwise')` returns a **`numCoordinates`**`-by-dimf` array containing the integrals of all node patches. The function f must be given in a manner, that n points can be evaluated simultaneously given as a `n-by-`**`dimSpace`** matrix. If you don't have a vectorized version of the function f it can be integrated using the (slower) option `'for'`:
`integrals = integrate(..., 'for')`.

`integrateP0` – $\int \chi\,dx$ – Integrate a piecewise constant function

`integral = integrateP0(mesh, P0)` returns the integral over the given mesh of the function `P0`, which is given by a **`numElements`**`-by-dimP0` matrix.
`integrals = integrateP0(mesh, P0, 'elementwise')` returns a **`numElements`**`-by-dimP0` array containing the elementwise integrals.
`integrals = integrateP0(mesh, P0, 'patchwise')` returns a **`numCoordinates`**`-by-dimP0` array containing the integrals of all node patches.

`integrateP1` – $\int \phi\,dx$ – Integrate a piecewise linear function

`integral = integrateP1(mesh, P1)` returns the integral of the function `P1` which is given by a **`numCoordinates`**`-by-dimP1` matrix over the given mesh.
`integrals = integrateP1(mesh, P1, 'elementwise')` returns a `nE-by-dimP1` array containing the elementwise integrals.
`integrals = integrateP1(mesh, P1, 'patchwise')` returns a **`numCoordinates`**`-by-dimP1` array containing the integrals of all node patches.

4.2 Inner products and testing with testfunctions

`inner_L2_L2_L2` – $\int f\cdot g\,dx$ – Inner product of two general L^2 functions

`S = inner_L2_L2_L2(mesh, f, g)` returns the L^2 scalar product of `f` and `g`. The functions must be given as function handles.
`S = inner_L2_L2_L2(mesh, f, g, quadDeg)` uses `quadDeg` as quadrature degree.
`Ss = inner_L2_L2_L2(mesh, f, g, quadDeg, 'elementwise')` yields the elementwise L^2 scalar product.

inner_L2_P0_L2 – $\int f \cdot \chi \, dx$ – Inner product of general L^2 and piecewise constant function

`V = inner_L2_P0_L2(mesh, f)` returns the vector `V` of the function `f` tested with the basis of the piecewise constant functions.

$$V_i = \int f \cdot \mathbb{1}_{T_i} \, dx,$$

where T_i is the element `mesh.elements(i,:)`.
`V = inner_L2_P0_L2(mesh, f, P0)` returns the inner product $(f, p_0)_{L^2}$. To specify the quadrature degree the commands
`S = inner_L2_P0_L2(mesh, f, P0, quadDeg)` and
`V = inner_L2_P0_L2(mesh, f, [], quadDeg)` can be used.

inner_L2_P1_L2 – $\int f \cdot \phi \, dx$ – Inner product of general L^2 and piecewise linear function

`V = inner_L2_P1_L2(mesh, f)` returns the vector `V` of `f` tested with all piecewise linear hat functions φ_i, which correspond to the node `mesh.coordinates(i,:)`.
`S = inner_L2_P1_L2(mesh, f, P1)` returns the inner product $(f, p_1)_{L^2}$ To specify the quadrature degree the commands
`S = inner_L2_P1_L2(mesh, f, P1, quadDeg)` and
`V = inner_L2_P1_L2(mesh, f, [], quadDeg)` can be used.

inner_P1_P1_L2 – $\int \phi_1 \cdot \phi_2 \, dx$ – Inner product of two piecewise linear functions

`S = inner_P1_P1_L2(mesh, phi_1, phi_2)` returns the inner product $(\phi_1, \phi_2)_{L^2}$.
`V = inner_P1_P1_L2(mesh, phi)` returns the vector `V` of `phi` tested with all hat functions.

$$V_i = \int \phi \cdot \varphi_i \, dx,$$

where φ_i is the hat function associated with the node `mesh.coordinates(i,:)`.
`M = inner_P1_P1_L2(mesh)` or `M = inner_P1_P1_L2(mesh, [], [], dimPhi)` returns the matrix `M`, whose entries are defined by $M_{ij} = \int \varphi_i \varphi_j \, dx$. This matrix is sometimes called "mass matrix".

inner_P0_P1_L2 – $\int \chi \cdot \phi \, dx$ – Inner product of piecewise constant and piecewise linear function

`S = inner_P0_P1_L2(mesh, P0, P1)` returns the inner product $(p_0, p_1)_{L^2}$.
`V = inner_P0_P1_L2(mesh, P0)` returns the vector `V` of the function `P0` tested with all hat functions.

$$V_i = \int p_0 \cdot \varphi_i \, dx,$$

where φ_i is the hat function associated with the node `mesh.coordinates(i,:)`.
`V = inner_P0_P1_L2(mesh, [], P1)` returns the vector `V` of `P1` tested with the basis of the piecewise constant functions.

$$V_i = \int p_1 \cdot \mathbb{1}_{T_i} \, dx,$$

where T_i is the element `mesh.elements(i,:)`.
`M = `**`inner_P0_P1_L2`**`(mesh)` or `M = `**`inner_P0_P1_L2`**`(mesh, [], [], dimPhi)` returns the matrix `M`, whose entries are defined by

$$M_{ij} = \int \mathbb{1}_{T_i} \cdot \varphi_j \, dx.$$

`inner_P0_P0_L2` – $\int \chi_1 \cdot \chi_2 \, dx$ – **Inner product of two piecewise constant functions**

`S = `**`inner_P0_P0_L2`**`(mesh, P0_1, P0_2)` returns the inner product $(p_{0,1}, p_{0,2})_{L^2}$.
`V = `**`inner_P0_P0_L2`**`(mesh, P0)` returns the vector `V` of the function `P0` tested with the basis of the piecewise constant functions.

$$V_i = \int p_0 \cdot \mathbb{1}_{T_i} \, dx,$$

where T_i is the element `mesh.elements(i,:)`.
`M = `**`inner_P0_P0_L2`**`(mesh)` or `M = `**`inner_P0_P0_L2`**`(mesh, [], [], dimP0)` returns the matrix `M`, whose entries are defined by

$$M_{ij} = \int \mathbb{1}_{T_i} \cdot \mathbb{1}_{T_j} \, dx.$$

`inner_gradP1_gradP1_L2` – $\int \nabla\phi_1 \cdot \nabla\phi_2 \, dx$ – **Inner product of gradients of piecewise linear functions**

`S = `**`inner_gradP1_gradP1_L2`**`(mesh, phi_1, phi_2)` returns the inner product $(\nabla\phi_1, \nabla\phi_2)_{L^2}$.
`V = `**`inner_gradP1_gradP1_L2`**`(mesh, phi)` returns the vector `V` of $\nabla\phi$ tested with all gradients of hat functions.

$$V_i = \int \nabla\phi \cdot \nabla\varphi_i \, dx$$

where φ_i is the hat function associated with the node `mesh.coordinates(i,:)`.
`M = `**`inner_gradP1_gradP1_L2`**`(mesh)` or `M = `**`inner_gradP1_gradP1_L2`**`(mesh, [], [], dimPhi)` returns the matrix, whose entries are defined by

$$M_{ij} = \int \nabla\varphi_i \cdot \nabla\varphi_j \, dx$$

`inner_GradHatI_P1_L2` – $\int \phi \cdot \nabla\varphi_i \, dx$ – **Inner products of gradients of hat functions and piecewise linear function**

`V = `**`inner_GradHatI_P1_L2`**`(mesh, P1)` returns the vector of `P1` tested by all gradients of hat functions.

$$V_i = \int p_1 \cdot \nabla\varphi_i \, dx.$$

`norm_L2_L2` – $\|f\|_{L^2}$ – **L^2 norm of L^2 function**

`N = `**`norm_L2_L2`**`(mesh, f)` returns the L^2 norm of the function handle `f`.

`norm_P1_L2` – $\|\phi\|_{L^2}$ – **L^2 norm of piecewise linear function**

`N = `**`norm_P1_L2`**`(mesh, P1)` returns the L^2 norm of the piecewise linear function `P1`.

`norm_P0_L2` – $\|\chi\|_{L^2}$ – **L^2 norm of piecewise constant function**

`N = `**`norm_P0_L2`**`(mesh, P0)` returns the L^2 norm of the piecewise constant function `P0`.

4.3 Reduced integration

`inner_P1_P1_h` – $\int \mathcal{I}_h(\phi_1 \cdot \phi_2)\,dx$ – **h inner product of piecewise linear functions**

`S = `**`inner_P1_P1_h`**`(mesh, phi_1, phi_2)` returns the reduced inner product

$$(\phi_1, \phi_2)_h := \int \mathcal{I}_h(\phi_1 \cdot \phi_2)\,dx,$$

with $\mathcal{I}_h$ being the nodal interpolant.
`V = `**`inner_P1_P1_h`**`(mesh, phi)` returns the vector `V` of `phi` tested with all hat functions.

$$V_i = \int \mathcal{I}_h(\phi \cdot \varphi_i)\,dx,$$

where φ_i is the hat function associated with the node `mesh.coordinates(i,:)`.
`M = `**`inner_P1_P1_h`**`(mesh)` or `M = `**`inner_P1_P1_h`**`(mesh, [], [], dimPhi)` returns the matrix `M`, whose entries are defined by $M_{ij} = \int \mathcal{I}_h(\varphi_i \cdot \varphi_j)\,dx$. This matrix is sometimes called "reduced mass matrix".

`norm_P1_h` – $\|\phi\|_h$ – **h norm of piecewise linear function**

`N = `**`norm_P1_h`**`(mesh, P1)` returns the norm induced by the reduced inner product $(\cdot,\cdot)_h$ of some function `P1`.

5 Operators

5.1 (Quasi-)Interpolation

`interpolateClement` – **Clement interpolate a general function**

`P1 = `**`interpolateClement`**`(mesh, f)` returns the elementwise linear, globally continuous function `P1`, whose nodal-values are the integral means of `f` on the corresponding node-patches.

`interpolateClementP0` – **Clement interpolate a piecewise constant function**

`P1 = `**`interpolateClementP0`**`(mesh, P0)` returns the elementwise linear, globally continuous function P1, whose nodal-values are the integral means of `P0` on the corresponding node-patches.

`interpolateNodal` – **Nodal interpolant for a general function**

`P1 = `**`interpolateNodal`**`(mesh, f)` returns the elementwise linear, globally continuous function `P1`, whose nodal-values are the evaluations of the function `f`.

5.2 L^2 projections

`L2ProjectL2ToP1` – **L^2 projection onto piecewise linear functions**

`P1 = `**`L2ProjectL2ToP1`**`(mesh, f)` returns the L^2-projection of `f` onto the space of elementwise linear, globally continuous functions.

`L2ProjectP1ToP0` – L^2 projection onto piecewise constant functions

`P0 =` **`L2ProjectP1ToP0`**`(mesh, P1)` projects a piecewise linear function `P1` onto the space of elementwise constant functions using the L^2 projection.

6 The Poisson problem

`solvePoisson` – Solve the poisson problem

`x =` **`solvePoisson`**`(mesh, f, g, uD, discretizer, quadDeg)` solves the poisson problem

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx + \int_{\Gamma_N} g v \, ds, \qquad \text{for all } v \in H^1_D(\Omega)$$

assuming the additional boundary condition $u = u_D$ on Γ_D.

- `x` is the piecewise linear solution,
- `mesh` is a mesh data structure generated by **`genMesh`**, describing the domain Ω and its boundaries Γ_D and Γ_N,
- `f`, `g` and `uD` are function handles representing f, g and u_D,
- `discretizer` is a function handle of the signature `@(dirichletBd, uD)` that discretizes the function u_D,
- `quadDeg` is the quadrature degree used for the underlying numerical integrations of f and g.

`adaptivePoisson` – Solve the poisson problem using adaptive mesh refinement

`[x, refinedMesh, indicators] =` **`adaptivePoisson`**`(mesh, f, g, uD, nEmax, ...`
`rho, discretizer, quadDeg, postProcessor)`

- `x` is the piecewise linear solution, corresponding to the output `refinedMesh`
- `indicators` are the last iterations error indicators, returned by **`estimatorPoissonResidual`**,
- `mesh` is a mesh data structure generated by **`genMesh`**, describing the domain Ω and its boundaries Γ_D and Γ_N,
- `f`, `g` and `uD` are function handles representing f, g and u_D,
- `discretizer` is a function handle of the signature `@(dirichletBd, uD)` that discretizes the function u_D,
- `quadDeg` is the quadrature degree used for the underlying numerical integrations of f and g.
- `postProcessor` is an optional function handle with the signature `@(mesh, x, indicators)` which can be used for plotting or saving the solution, e.g. `postProcessor = @surfSolution` or `postProcessor = @(mesh, x, ind_)` **`saveSolution`**`('example2D', mesh, x)`

Here is an example, that solves the poisson problem on a square.

Listing 2: `poisson.m`

```
%% Generate a mesh
mesh = genMesh([1,2,3;3,4,1], [0,0;1,0;1,1;0,1], [1,2], [2,3;3,4;4,1]);
mesh = genBisectionMesh(mesh);
%% Define poisson problem data
f = @(X) -sum(abs(X),2).^(1/2);
g = @(X) 1*X(:,2).^3;
uD = @(X) 1/2*X(:,1);
%% Define parameters for adaptive algorithm
nEmax = 1e5;
rho = 0.1;
discretizer = @(dirichletBd, uD) L2ProjectL2ToP1(dirichletBd, uD, 2);
quadDeg = 2;
%% Start computation
[x, refinedMesh, indicators] = adaptivePoisson(mesh, f, g, uD, nEmax, rho, ...
                                               discretizer, quadDeg, @surfSolution);
```

`estimatorPoissonResidual` – Error estimator for the poisson problem

`etaR = estimatorPoissonResidual(mesh, x, f, g, quadDeg)` computes the error estimator given on an element T by the equation

$$\eta_T^2 := h_T^2 \|f\|_{L^2(T)}^2 + h_T \|J_h(\partial_n U)\|_{L^2(\partial T \cap \Omega)}^2 + h_T \|g - \partial_n U\|_{L^2(\partial T \cap \Gamma_N)}^2.$$

- `etaR` is a vector containing the values η_T^2,
- `mesh` is a mesh data structure generated by **genMesh**, describing the domain Ω and its boundaries Γ_D and Γ_N,
- `f`, `g` are function handles representing f and g,
- `quadDeg` is the quadrature degree used for the underlying numerical integrations of f and g.

7 Visualization and Postprocessing

`saveSolution` – Save the solutions during simulation

Use this function as the `postProcessor` argument in the adaptivePoisson function.
`postProcessor = @(mesh, x, ind_) saveSolution('example2D', mesh, x)`
It will save the solution of each adaptive step.

`surfSolution` – Plot the solutions during simulation

Use this function as the `postProcessor` argument in the adaptivePoisson function.
`postProcessor = @surfSolution` It will plot the solution of each adaptive step.

`surfMesh` – Plot a mesh structure

`surfMesh(mesh)` will plot a one, two or three dimensional mesh.
`surfMesh(mesh, P1)` will plot a piecewise linear function `P1` corresponding to the nodes of the mesh.