Softwarepraktikum (Wissenschaftliches Rechnen) für Fortgeschrittene

Pavel Hron, Arsalan Farooq
Ruprecht-Karls-Universität Heidelberg
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
email: paja.hron@email.cz

February 12, 2010

1 Problem Formulation

Consider the steady-state groundwater flow equation (General Diffusion Equation) with corresponding Dirichlet and Neumann boundary conditions

$$-\nabla \cdot \{K(x)\nabla p\} = f \qquad \text{in } \Omega \subseteq \mathbb{R}^n,\ n = 2, 3, \tag{1a}$$

$$p = g \qquad \text{on } \Gamma_D \subseteq \partial\Omega, \tag{1b}$$

$$-\left(\nabla \cdot K(x)\nabla p\right) \cdot n(x) = j \qquad \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D \neq \partial\Omega, \tag{1c}$$

where K is the absolute permeability.

$K : \mathbb{R}^n \longrightarrow \mathbb{R}^{n\times n}$ is a projection, which relates to each point $x \in \Omega$ a $n \times n$ matrix $K(x)$. We demand also (for all $x \in \ \Omega$) that $K(x)$

1. $K(x) = K^T(x)$ and $\xi^T K(x)\xi > 0\ \forall \xi \in \mathbb{R}^n, \xi \neq 0$ (symmetric positive definite),
2. $C(x) := \min\{\xi^T K(x)\xi |\ \|\xi\| = 1\} \geq C_0 > 0$ (uniform ellipticity).

In this problem we assume that permeability K is strongly varying in the domain Ω.

Denote the volumetric flux function through a porous medium $\vec{J}$

$$\vec{J}(x) = -K(x)\nabla p. \tag{2}$$

We want to solve the equation (1) in $\Omega = [0, 1]^d$, without sink or source therms ($f = 0$) and appropriate boundary conditions. On the left hand side and right hand side of the domain Ω is Dirichlet boundary condition with $p\ = 1$ and $p = 0$ respectively. The upper and the lower boundaries of Ω are impermeable, which means Neumann boundary condition is zero.

Above mentioned problem description we can write in the mathematical form

$$\nabla \cdot \vec{J}(x) = 0 \qquad \text{in } \Omega = [0, 1]^d,\ d = 2, 3, \tag{3a}$$

$$p = 1 \qquad \text{on } \Gamma_{D_1} = \{x = (x_1, \ldots, x_d) | x_1 = 0, x_2, \ldots, x_d \in [0, 1]\} \subseteq \partial\Omega, \tag{3b}$$

$$p = 0 \qquad \text{on } \Gamma_{D_2} = \{x = (x_1, \ldots, x_d) | x_1 = 1, x_2, \ldots, x_d \in [0, 1]\} \subseteq \partial\Omega, \tag{3c}$$

$$-\nabla p \cdot n(x) = 0 \qquad \text{on } \Gamma_N = \partial\Omega \setminus (\Gamma_{D_1} \cup \Gamma_{D_2}) \neq \partial\Omega. \tag{3d}$$

The main goal of this project is to get the overall flux F over the right face Γ_{D_2} of the domain Ω

$$F = \int_{\Gamma_{D_2}} \vec{J}(x) \cdot n(x) \mathrm{d}l. \tag{4}$$

At first we have to solve system (3). For this purpose we use following numerical methods:

- Cell Centered Finite Volumes (FV)
- Mixed Finite Elements (MFEM)
- Conforming Finite Elements (FEM)
- Odeon-Baumann-Babuska Discontinuous Galerkin (DG)

2 Flux implementation in DUNE

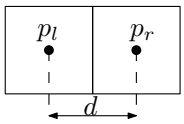
The DUNE (Distributed and Unified Numerics Environment, www.dune-project.org) consists of many modules. In the module dune-pdelab-howto above mentioned methods for problem (3) are already implemented. The objective of our work was the implementation of a function, which calculates the integral (4).

2.1 Cell Centered Finite Volumes

The result of (3) for CCFV method is function, which is constant on each grid cell. If we have conforming, structured rectangular (2D) or cuboid (3D) grid, the term $\nabla p \cdot n(x)$ on the face can be approximated as

$$\nabla p \cdot n(x) = \frac{p_l - p_r}{d}, \tag{5}$$

where p_l,p_r are the values of the pressure p in cells and d is the distance between centroids of this two cells. Then we can evaluate integral (4) using mid-point rule.



File `fluxfv.hh`

```
1  template<typename GV, typename DGF, typename KType>
2  void flux(const GV& gv, const DGF& dgf,const KType& k)
3  {
4    // first we extract the dimensions of the grid
5    const int dim = GV::dimension;
6    const int dimworld = GV::dimensionworld;
7    const double epsilon=1e-10;
8
9    // discrete grid function range type
10   typename DGF::Traits::RangeType RT;
11
12   // k inside the element
13   typename KType::Traits::RangeType k_inside;
14
15   // type used for coordinates in the grid
16   typedef typename GV::ctype ct;
17
18   // element iterators
19   typedef typename GV::template Codim<0>::Iterator ElementLeafIterator;
20   typedef typename GV::IntersectionIterator IntersectionIterator;
21
22   // flux overall
23   double flux_sum=0.0;
24   // loop over elements
25   for (ElementLeafIterator it = gv.template begin<0>();
26        it!=gv.template end<0>(); ++it)
27     {
28       // cell geometry type
29       Dune::GeometryType gt = it->type();
30
31       // cell center in reference element
32       const Dune::FieldVector<ct,dim>& local = Dune::ReferenceElements<ct,dim>::
33     general(gt).position(0,0);
34
35       // cell center in global coordinates
36       Dune::FieldVector<ct,dimworld>
```

```
37      global = it->geometry().global(local);
38
39        // cell volume, assume linear map here
40        double volume = it->geometry().integrationElement(local)
41      *Dune::ReferenceElements<ct,dim>::general(gt).volume();
42
43        // run through all intersections with neighbors and boundary
44        IntersectionIterator isend = gv.iend(*it);
45        for (IntersectionIterator is = gv.ibegin(*it); is!=isend; ++is)
46      {
47        // get geometry type of face
48        Dune::GeometryType gtf = is->type();
49
50        // center in face's reference element
51        const Dune::FieldVector<ct,dim-1>&
52          facelocal = Dune::ReferenceElements<ct,dim-1>::general(gtf).position(0,0);
53
54            // cell center in global coordinates
55            Dune::FieldVector<ct,dimworld>
56              faceglobal = is->geometry().global(facelocal);
57
58        // face volume
59        ct face_volume=is->intersectionGlobal().volume();
60
61        // distance:element center to face center
62        float d=0;
63            for (size_t i=0; i<dim; i++)
64              {
65                d+=(faceglobal[i]-global[i])*(faceglobal[i]-global[i]);
66              }
67            d=sqrt(d);
68
69        // flux evaluation
70        if (fabs(faceglobal[0]-1.0)<epsilon)
71              {
72            dgf.evaluate(*(it),local,RT);
73                k.evaluate(*(it),local,k_inside);
74                flux_sum+=face_volume*k_inside*RT/d;
75          }
76          }
77      }
78    std::cout << "flux␣overall␣" << flux_sum << std::endl;
79
80  }
```

2.2 Mixed Finite Elements

If we use MFEM method, we gain not only pressure p, but also the term $\vec{J}(x)$. That is why the calculation of integral (4) is easy.

```
1  template<typename GV, typename RT0DGF>
2  void flux(const GV& gv, const RT0DGF& rt0dgf)
3  {
4    double flux_sum=0.0;
5    double epsilon=1e-10;
6
7    // first we extract the dimensions of the grid
8    const int dim = GV::dimension;
9    const int dimworld = GV::dimensionworld;
10
11   typename RT0DGF::Traits::RangeType RT;
12   typedef typename RT0DGF::Traits::DomainFieldType DF;
13   typedef typename RT0DGF::Traits::RangeFieldType RF;
14
15
16   // type used for coordinates in the grid
17   typedef typename GV::ctype ct;
```

```
18   typedef typename GV::template Codim<0>::Iterator ElementLeafIterator;
19   typedef typename GV::IntersectionIterator IntersectionIterator;
20
21   // loop over elements
22   for (ElementLeafIterator it = gv.template begin<0>();
23        it!=gv.template end<0>(); ++it)
24     {
25       // cell geometry type
26       Dune::GeometryType gt = it->type();
27
28       // cell center in reference element
29       const Dune::FieldVector<ct,dim>&
30         local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
31
32       // run through all intersections with neighbors and boundary
33       IntersectionIterator isend = gv.iend(*it);
34       for (IntersectionIterator is = gv.ibegin(*it); is!=isend; ++is)
35     {
36           // get geometry type of face
37       Dune::GeometryType gtf = is->type();
38
39       // center in face's reference element
40       const Dune::FieldVector<ct,dim-1>&
41         facelocal = Dune::ReferenceElements<ct,dim-1>::general(gtf).position(0,0);
42
43       // center of face in global coordinates
44       Dune::FieldVector<ct,dimworld> faceglobal = is->geometry().global(facelocal);
45
46
47       // is the x-coordinates of facecenter = 1
48           if (fabs(faceglobal[0]-1.0)<epsilon)
49             {
50           // get normal vector scaled with volume
51           Dune::FieldVector<ct,dimworld> integrationOuterNormal =
52         is->integrationOuterNormal(facelocal);
53           integrationOuterNormal *=
54         Dune::ReferenceElements<ct,dim-1>::general(gtf).volume();
55
56           // loop over quadraturerules, order=1
57           const Dune::QuadratureRule<DF,dim-1>& rule =
58         Dune::QuadratureRules<DF,dim-1>::rule(gtf,1);
59                for (typename Dune::QuadratureRule<DF,dim-1>::const_iterator cit=
60               rule.begin(); cit!=rule.end(); ++cit)
61                  {
62                    // position of quadrature point in local coordinates of element
63                    Dune::FieldVector<DF,dim> local =
64           is->geometryInInside().global(cit->position());
65           RF factor = cit->weight();
66
67           // k*velocity evaluation
68           rt0dgf.evaluate(*it,local,RT);
69
70           flux_sum+= integrationOuterNormal*RT*factor;
71
72                 } //cit
73
74             } // border
75
76         } // is
77     } // it
78   std::cout << "flux overall " << flux_sum << std::endl;
79 }
```

2.3 Conforming Finite Elements, Discontinuous Galerkin

The solution $p(x)$ of problem (3) can be expressed as

$$p(x) = \sum_{e_i} \sum_{j=1}^{n_{e_i}} c_{e_i,j} \cdot \psi_{e_i,j}(x). \tag{6}$$

For each element e_i there is a set of base functions $\psi_{e_i,j}(x)$, which are multiplicated with coefficients $c_{e_i,j}$. Therefore ∇p we get as

$$\nabla p(x) = \sum_{e_i} \sum_{j=1}^{n_{e_i}} c_{e_i,j} \cdot \nabla \psi_{e_i,j}(x). \tag{7}$$

For evaluation of integral (4) we can use again numerical integration.

```
1  template<typename GV, typename X, typename GFS, typename KType>
2  void flux(const GV& gv, const X& x,const GFS& gfs,const KType& k,int intorder=1)
3  {
4
5    double flux_sum=0.0;
6    double epsilon=1e-10;
7
8    // first we extract the dimensions of the grid
9    const int dim = GV::dimension;
10   const int dimw= GV::dimensionworld;
11
12   typedef typename GFS::LocalFunctionSpace::Traits::LocalFiniteElementType::
13     Traits::LocalBasisType::Traits::DomainFieldType DF;
14   typedef typename GFS::LocalFunctionSpace::Traits::LocalFiniteElementType::
15     Traits::LocalBasisType::Traits::RangeFieldType RF;
16   typedef typename GFS::LocalFunctionSpace::Traits::LocalFiniteElementType::
17     Traits::LocalBasisType::Traits::JacobianType JacobianType;
18
19   typedef typename GFS::LocalFunctionSpace LFS;
20   LFS lfs(gfs); // local function spacel
21
22   std::vector<RF> xl; // local container
23   typename KType::Traits::RangeType k_inside; // K tensor
24
25
26   // type used for coordinates in the grid
27   typedef typename GV::ctype ct;
28
29   typedef typename GV::template Codim<0>::Iterator ElementLeafIterator;
30   typedef typename GV::IntersectionIterator IntersectionIterator;
31
32   for (ElementLeafIterator it = gv.template begin<0>();
33        it!=gv.template end<0>(); ++it)
34     {
35       // cell geometry type
36       Dune::GeometryType gt = it->type();
37
38       // cell center in reference element
39       const Dune::FieldVector<ct,dim>&
40         local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
41
42       // evaluate k tensor
43       k.evaluate(*(it),local,k_inside);
44
45       // run through all intersections with neighbors and boundary
46       IntersectionIterator isend = gv.iend(*it);
47       for (IntersectionIterator is = gv.ibegin(*it); is!=isend; ++is)
48         {
```

```
49            // get geometry type of face
50            Dune::GeometryType gtf = is->type();
51
52            // center in face's reference element
53            const Dune::FieldVector<ct,dim-1>&
54              facelocal = Dune::ReferenceElements<ct,dim-1>::general(gtf).position(0,0);
55
56            // get normal vector scaled with volume
57            Dune::FieldVector<ct,dimw> integrationOuterNormal
58          = is->integrationOuterNormal(facelocal);
59        integrationOuterNormal
60          *= Dune::ReferenceElements<ct,dim-1>::general(gtf).volume();
61
62            // center of face in global coordinates
63            Dune::FieldVector<ct,dimw> faceglobal = is->geometry().global(facelocal);
64
65            if (fabs(faceglobal[0]-1.0)<epsilon)
66              {
67                // loop over quadrature points
68                const Dune::QuadratureRule<DF,dim-1>& rule =
69          Dune::QuadratureRules<DF,dim-1>::rule(gtf,intorder);
70                for (typename Dune::QuadratureRule<DF,dim-1>::const_iterator
71               cit=rule.begin(); cit!=rule.end(); ++cit)
72                  {
73
74                    // position of quadrature point in local coordinates of element
75                    Dune::FieldVector<DF,dim> local =
76              is->geometryInInside().global(cit->position());
77
78                    RF factor = cit->weight();
79
80                    lfs.bind(*it);
81                    lfs.vread(x,xl); // write local container into xl
82
83                    // evaluate jacobian
84                    std::vector<JacobianType> js(lfs.size());
85                    lfs.localFiniteElement().localBasis().evaluateJacobian(local,js);
86
87                    // transform gradient to real element
88                    const Dune::FieldMatrix<DF,dimw,dim> jac =
89              it->geometry().jacobianInverseTransposed(local);
90                    std::vector<Dune::FieldVector<RF,dim> > gradphi(lfs.size());
91                    for (size_t i=0; i<lfs.size(); i++)
92                      {
93                        gradphi[i] = 0.0;
94                        jac.umv(js[i][0],gradphi[i]);
95                      }
96
97                    // compute gradient of u
98                    Dune::FieldVector<RF,dim> gradu(0.0);
99                    for (size_t i=0; i<lfs.size(); i++)
100                     gradu.axpy(xl[i],gradphi[i]);
101
102                   // compute K * gradient of u
103                   Dune::FieldVector<RF,dim> Kgradu(0.0);
104                   k_inside.umv(gradu,Kgradu);
105                   flux_sum+=integrationOuterNormal*Kgradu*factor;
106
107                 } // cit
108
109             } // border
110
111         } // is
112     }  //it
113
114   std::cout << "flux overall " << -1*flux_sum << std::endl;
115
116 }
```

3 Numerical results

In this chapter we will discuss some numerical results for different permeability fields. This examples shows, that the cell-centered Finite Volume Method, Discontinuous Galerkin Method and Mixed Finite Elements Method tend to underestimate the overall flux, the Finite Element Method tend to overestimate the overall flux. Finite Element Method converges only for continuous permeability fields.

3.1 Durlofsky permeability field

Durlofsky has defined the permeabillity field on a regular 20×20 mesh (see Fig. 1). The permeability in dark areas is $K = 10^{-6}$, in light areas is $K = 1$.

The unit square is discretized with 20×20 rectangular elements for cube grid or with $20 \times 20 \times 2$ triangular elements for simplex grid. Finer grids are obtained through regular refinement.

The exact value of F for Durlofsky permeability field[1] has been given in as 0.5205. In Table 1 and in Fig. 2 we show results for the unknown overall flux through the system F with rectangle grid discretization. We compare above mentioned numerical schemes. The results clearly show the unsuitability of the Conforming Finite Element Method for this type of problem, because this method has a bad approximation of average permeability.

For rectangular grid we get good results with the lowest order Mixed Finite Element Method. Discontinuous Galerkin Method gives the best results, this method is locally mass conservative and in comparison with Finite Volume Method uses more degrees of freedom. The result for triangular grid discretization are shown in Fig. 3 and in Table 2. In this case Mixed Finite Element Method is worse than rectangle grid discretization.

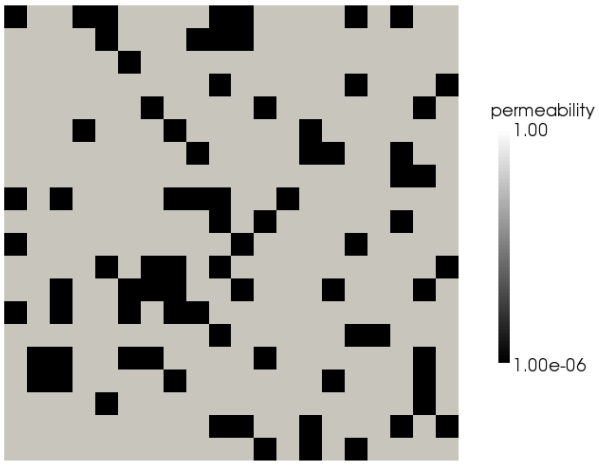


Figure 1: Durlofsky permeability field.

[1] L. J. Durlofsky, *Accuracy of mixed and control volume finite element approximations to Darcy velocity and related quantities*, Water Resources Reserach **30** (1994), no. 4, 965-973

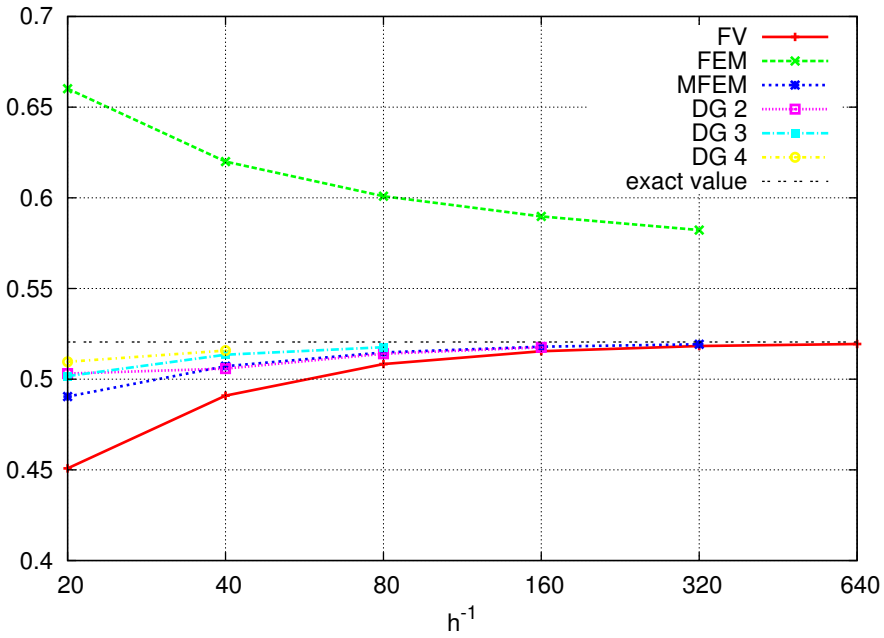


Figure 2: The overall flux F for the Durlovsky problem calculated with diferent discretization schemes and successive grid refinement, rectangle grid discretization.

h^{-1}	FV	MFEM	DG, r=2	DG, r=3	DG, r=4	FEM,Q1
20	0.450829	0.490408	0.503091	0.501675	0.509603	0.660146
40	0.49096	0.507119	0.505838	0.513495	0.51577	0.619998
80	0.508343	0.514745	0.513997	0.517623		0.600907
160	0.515426	0.517945	0.517588			0.589748
320	0.518264	0.515426				0.582236
640	0.519393					

Table 1: The overall flux F for the Durlovsky problem, rectangles.

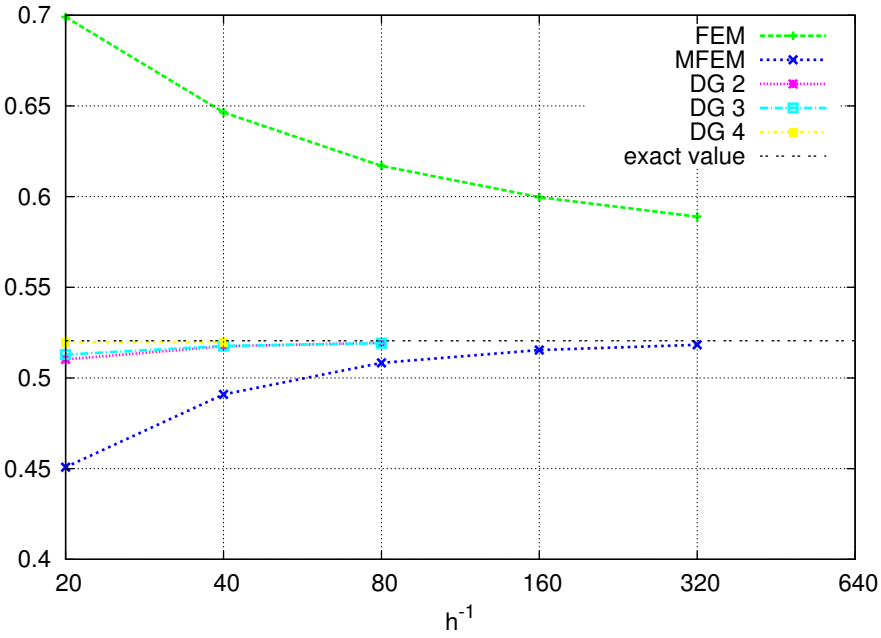


Figure 3: The overall flux F for the Durlovsky problem calculated with diferent discretization schemes and successive grid refinement, triangle grid discretization.

h^{-1}	MFEM	FEM, P1	DG, r=2	DG, r=3	DG, r=4
20	0.450829	0.699048	0.510242	0.512905	0.519586
40	0.49096	0.646453	0.517507	0.517643	0.519838
80	0.508343	0.616949	0.519378	0.519183	
160	0.515426	0.59974			
320	0.518264	0.588928			
640					

Table 2: The overall flux F for the Durlovsky problem, triangles.

3.2 Discontinuous permeability field

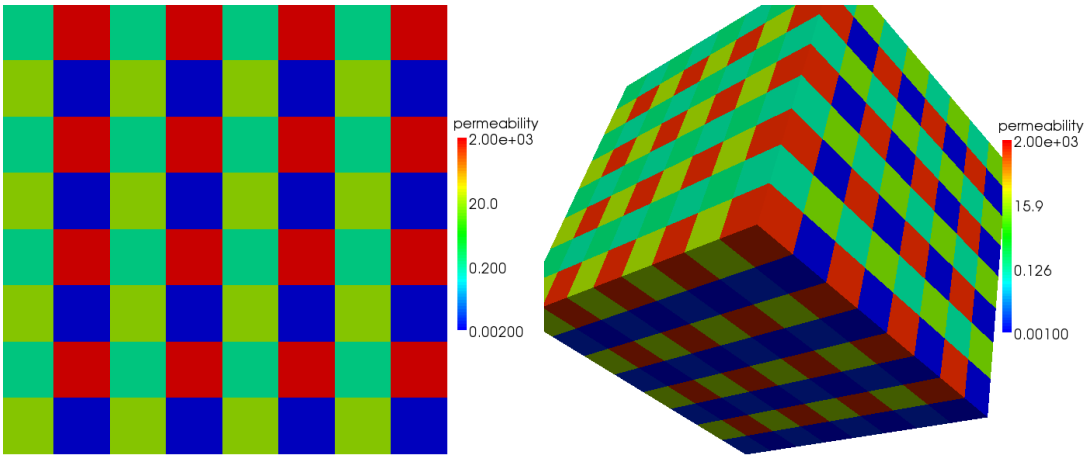
Durlofsky permeability field is discontinuous, it consists of only 2 values. The distribution is irregular. In this example we describe discontinuous permeability field with periodic distribution.

This discontinuous permeability field can be for all $x = (x_1, x_2, x_3) \in \Omega$ defined by the formula

$$K(x) = \begin{cases} 20.0 & \lfloor \frac{x_1}{h} \rfloor \text{ even}, \lfloor \frac{x_2}{h} \rfloor \text{ even}, \lfloor \frac{x_3}{h} \rfloor \text{ even} \\ 0.002 & \lfloor \frac{x_1}{h} \rfloor \text{ odd}, \lfloor \frac{x_2}{h} \rfloor \text{ even}, \lfloor \frac{x_3}{h} \rfloor \text{ even} \\ 0.2 & \lfloor \frac{x_1}{h} \rfloor \text{ even}, \lfloor \frac{x_2}{h} \rfloor \text{ odd}, \lfloor \frac{x_3}{h} \rfloor \text{ even} \\ 2000.0 & \lfloor \frac{x_1}{h} \rfloor \text{ odd}, \lfloor \frac{x_2}{h} \rfloor \text{ odd}, \lfloor \frac{x_3}{h} \rfloor \text{ even} \\ 1000.0 & \lfloor \frac{x_1}{h} \rfloor \text{ even}, \lfloor \frac{x_2}{h} \rfloor \text{ even}, \lfloor \frac{x_3}{h} \rfloor \text{ odd} \\ 0.001 & \lfloor \frac{x_1}{h} \rfloor \text{ odd}, \lfloor \frac{x_2}{h} \rfloor \text{ even}, \lfloor \frac{x_3}{h} \rfloor \text{ odd} \\ 0.1 & \lfloor \frac{x_1}{h} \rfloor \text{ even}, \lfloor \frac{x_2}{h} \rfloor \text{ odd}, \lfloor \frac{x_3}{h} \rfloor \text{ odd} \\ 10.0 & \lfloor \frac{x_1}{h} \rfloor \text{ odd}, \lfloor \frac{x_2}{h} \rfloor \text{ odd}, \lfloor \frac{x_3}{h} \rfloor \text{ odd} \end{cases}. \tag{8}$$

This permeability field in 2D and in 3D is shown in Figure 4.

The mesh size in this problem is taken as 16×16 to prevent the problem from complexity. Table 3.2 and 3.2 show the results of the flux for different methods with successive refinements. As in the case of Finite Element Method, we are not getting good results because this method has a bad approximation for permeability field and not locally mass conservative and secondly it is discrete and not smooth enough. Therefore, we are not including the fluxes for FEM in the graph.



(a) (b)

Figure 4: Discontinuous permeability field.

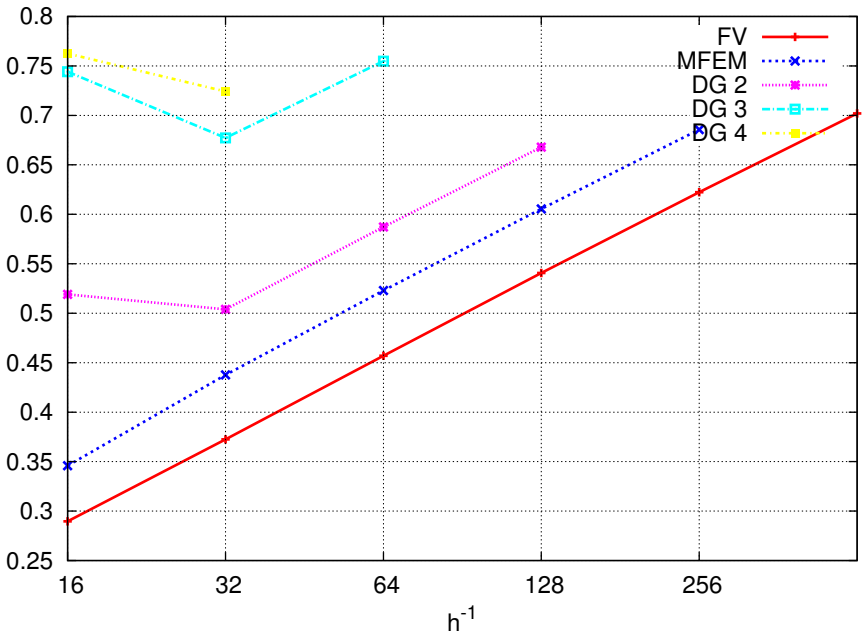


Figure 5: The overall flux F for the discontinuous permeability field (8) in 2D at successive refinements, rectangular grid discritization.

h^{-1}	FV	MFEM	DG, r=2	DG, r=3	DG, r=4	FEM,Q1
16	0.289657	0.345877	0.519077	0.74418	0.762491	11.6776
32	0.372477	0.437726	0.503865	0.677174	0.724126	9.22717
64	0.45705	0.522987	0.58715	0.754825		7.72284
128	0.540632	0.605493	0.667953			6.6705
256	0.622409	0.685629				5.89056
512	0.701939					5.29135

Table 3: The overall flux F for the discontinuous permeability field (8) in 2D, rectangles.

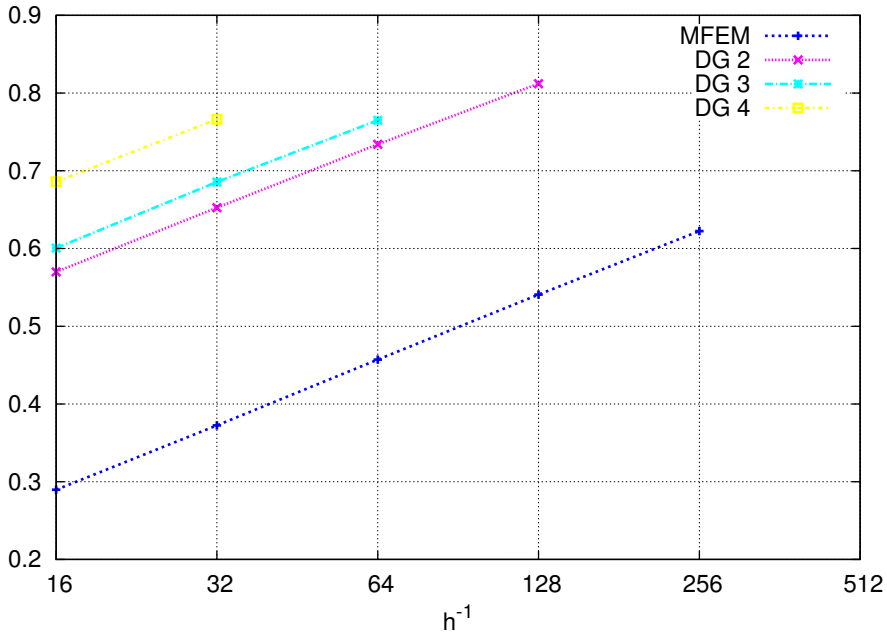


Figure 6: The overall flux F for the discontinuous permeability field (8) in 2D at successive refinements, triangular grid discritization.

h^{-1}	MFEM	DG, r=2	DG, r=3	DG, r=4	FEM,P1
16	0.289657	0.569913	0.600637	0.6859	14.2598
32	0.372476	0.652479	0.685519	0.766271	10.9272
64	0.457048	0.734066	0.764934		8.86621
128	0.540628	0.812042			7.48263
256	0.622395				6.49433
512					5.82135

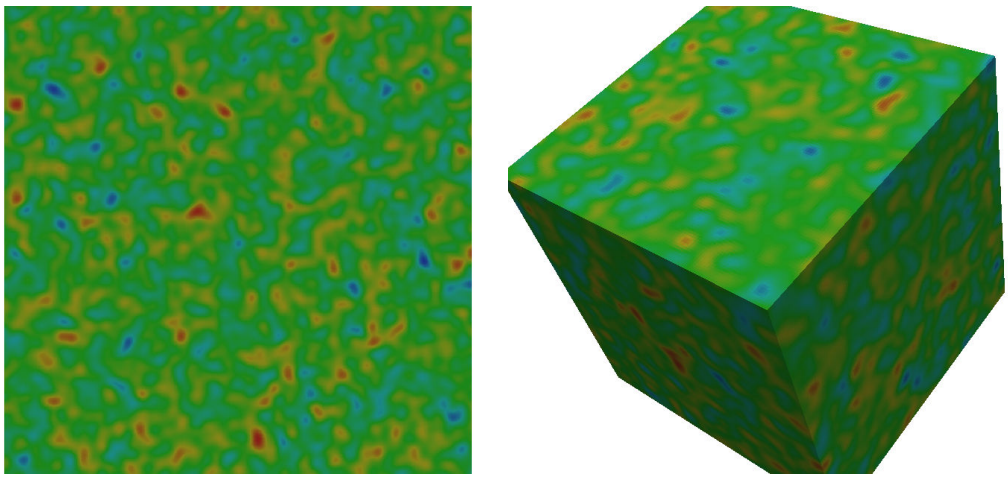
Table 4: The overall flux F for the discontinuous permeability field (8) in 2D, triangles.

h^{-1}	FV	MFEM	DG, r=2	FEM,Q1
16	0.163812	0.175001	0.296727	251.52
32	0.217076	0.259619	0.296727	94.7228
64	0.279882			48.9817
128	0.34436			27.0621

Table 5: The overall flux F for the discontinuous permeability field (8) in 3D, cuboids.

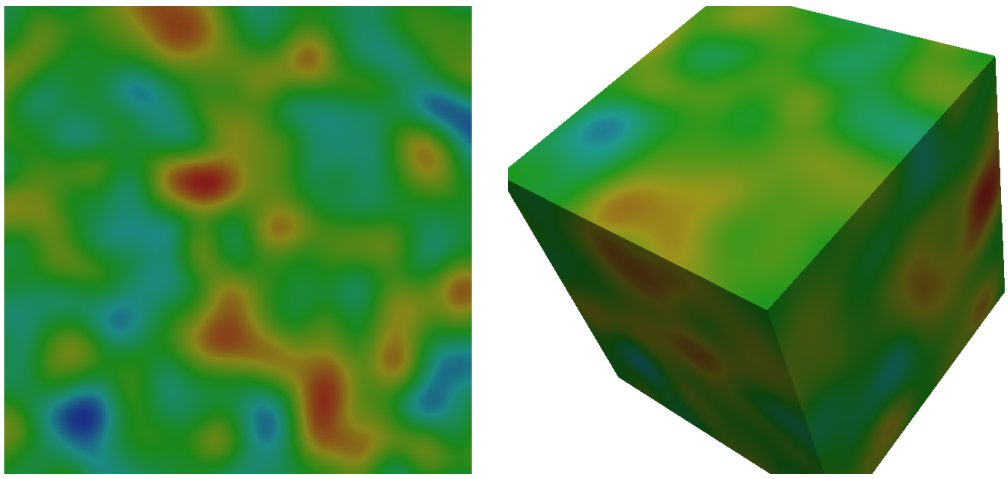
3.3 Random permeability field

The function $K(x)$ is log-normal distributed with a given mean of 0, a variance of 3 (i.e. the permeabilities alternating between 10^{-3} and 10^{3}) and a correlation length of 1/64 or 1/16 in 2D and 1/32 or 1/8 in 3D. Examples of these random permeability fields are shown in Fig. 7 and 8.



(a) (b)

Figure 7: Log-normal distributed permeability fields in 2D and 3D, correlation length of 1/64 in 2D and 1/32 in 3D.



(a) (b)

Figure 8: Log-normal distributed permeability fields in 2D and 3D, correlation length of 1/16 in 2D and 1/8 in 3D.

Numerical results for random permeability field with smaller correlation length (Fig. 9, 10 and Tab. 6, 7 and 8) shows, that all methods converge. The rate of convergence is smaller than the rate of convergence for random permeability field with greater correlation length (see Fig. 11, 12 and Tab. 9, 10 and 11). The reason is that random permeability field with greater correlation length is smoother. In this case FEM also converges, in contrast to discontinuous

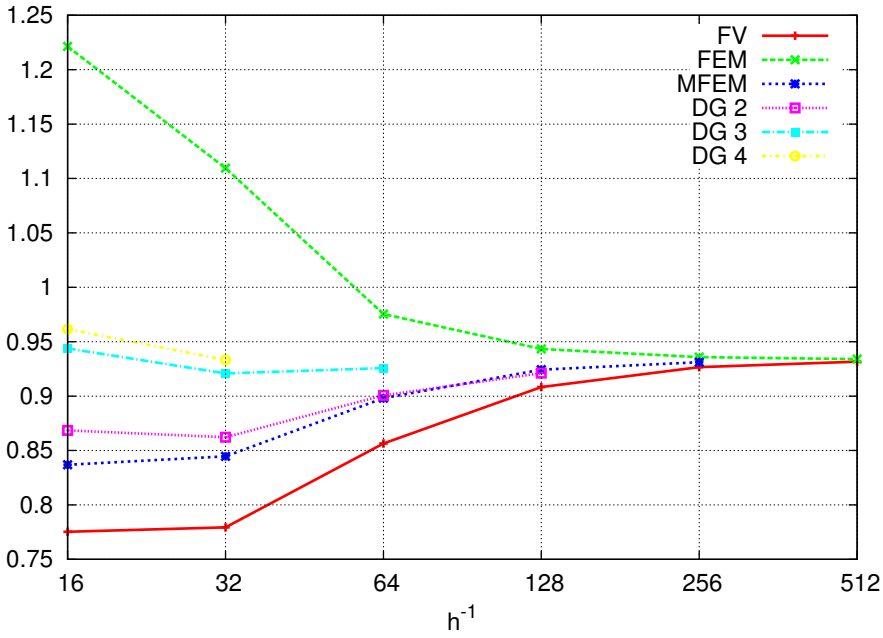


Figure 9: The overall flux F for the random permeability field 2D problem with correlation length of 1/64 calculated with diferent discretization schemes and successive grid refinement, rectangle grid discretization.

permeability field.

h^{-1}	FV	MFEM	DG, r=2	DG, r=3	DG, r=4	FEM,Q1
16	0.77534	0.836949	0.868496	0.943831	0.961535	1.22132
32	0.779322	0.84461	0.8621	0.920732	0.93326	1.1095
64	0.85643	0.898071	0.90065	0.9257423		0.975436
128	0.908352	0.924298	0.921081001			0.943341
256	0.926655	0.931132				0.935863
512	0.931683					0.934049

Table 6: The overall flux F for the random permeability 2D field with correlation length of 1/64, rectangles.

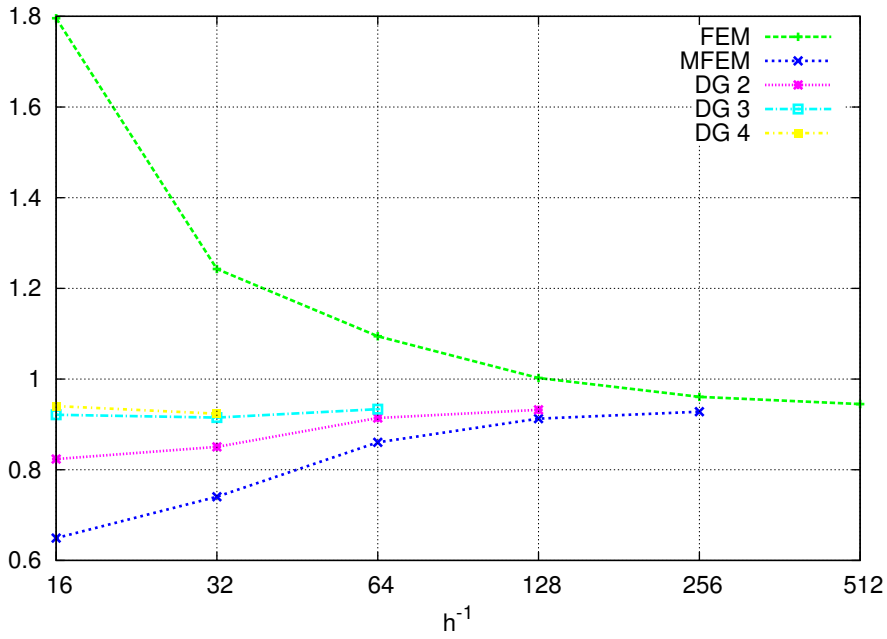


Figure 10: The overall flux F for the random permeability field 2D problem with correlation length of 1/64 calculated with diferent discretization schemes and successive grid refinement, triangle grid discretization.

h^{-1}	MFEM	FEM, P1	DG, r=2	DG, r=3	DG, r=4
16	0.649137	1.79596	0.823581	0.921255	0.940517
32	0.740271	1.24269	0.85048	0.9149971	0.923693
64	0.860335	1.09437	0.91421	0.933646	
128	0.912719	1.00245	0.932027		
256	0.928019	0.960686			
512		0.944735			

Table 7: The overall flux F for the random permeability 2D field with correlation length of 1/64, triangles.

h^{-1}	FV	MFEM	DG, r=2	FEM,Q1
8	1.1449	1.29826	1.35603	2.31804
16	1.01599	1.14758		1.83746
32	1.31719			1.66314
64	1.48614			1.591
128	1.54393			

Table 8: The overall flux F for the random permeability 3D field with correlation length of 1/32, cuboids.

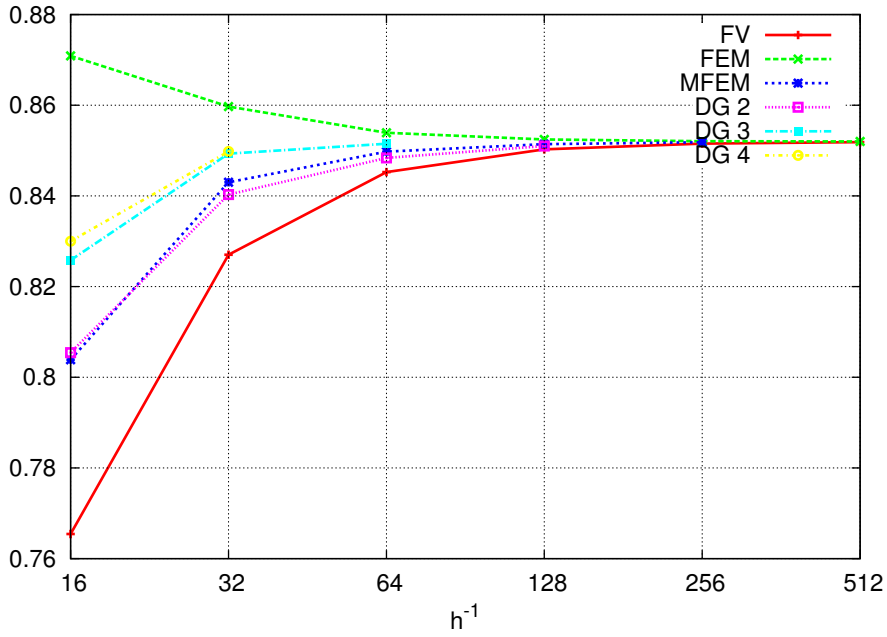


Figure 11: The overall flux F for the random permeability field 2D problem with correlation length of 1/16 calculated with diferent discretization schemes and successive grid refinement, rectangle grid discretization.

h^{-1}	FV	MFEM	DG, r=2	DG, r=3	DG, r=4	FEM,Q1
16	0.765417	0.803858	0.805419	0.825815	0.82999	0.870887
32	0.827035	0.843016	0.840281	0.849335	0.849788	0.859714
64	0.845258	0.849771	0.848381	0.851502		0.853922
128	0.85026	0.851424	0.851004			0.852462
256	0.851545	0.851836				0.852078
512	0.851872					0.851995

Table 9: The overall flux F for the random permeability 2D field with correlation length of 1/16, rectangles.

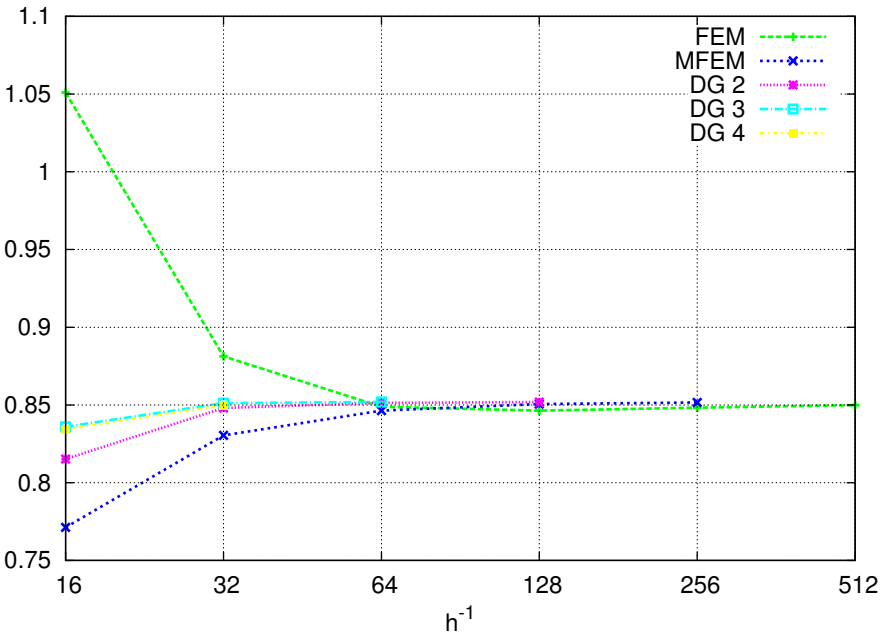


Figure 12: The overall flux F for the random permeability field 2D problem with correlation length of 1/16 calculated with diferent discretization schemes and successive grid refinement, triangle grid discretization.

h^{-1}	MFEM	FEM, P1	DG, r=2	DG, r=3	DG, r=4
16	0.771213	1.05094	0.815247	0.835966	0.834409
32	0.830419	0.881289	0.848087	0.851209	0.849858
64	0.846369	0.848858	0.85135	0.851946	
128	0.850556	0.846326	0.851841		
256	0.851618	0.848221			
512		0.849876			

Table 10: The overall flux F for the random permeability 2D field with correlation length of 1/16, triangles.

h^{-1}	FV	MFEM	DG, r=2	FEM,Q1
8	0.832339	0.879213	0.882525	0.984478
16	0.910224	0.932414		0.958893
32	0.938126			0.951348
64	0.946078			0.949462
128	0.94814			0.948991

Table 11: The overall flux F for the random permeability 3D field with correlation length of 1/8, cuboids.