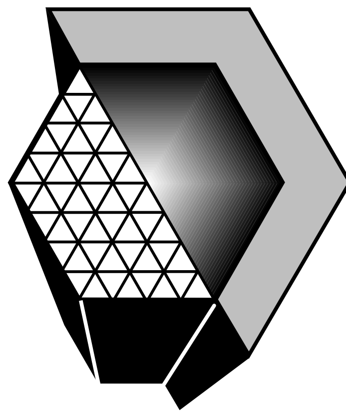
# Multigrid applied to different partial differential operators

Gerhard W. Zumbusch



Diffpack

The Diffpack Report Series

November 22, 1996

SINTEF

*This report is compatible with version 2.4 of the Diffpack software.*

The development of `Diffpack` is a cooperation between

- SINTEF Applied Mathematics,
- University of Oslo, Department of Informatics.
- University of Oslo, Department of Mathematics

The project is supported by the Research Council of Norway through the technology program: *Numerical Computations in Applied Mathematics* (110673/420).

For updated information on the Diffpack project, including current licensing conditions, see the web page at

`http://www.oslo.sintef.no/diffpack/`.

**Copyright © SINTEF, Oslo**
**November 22, 1996**

Permission is granted to make and distribute verbatim copies of this report provided the copyright notice and this permission notice is preserved on all copies.

**Abstract**

The report is a continuation of an introductory report on the multigrid iterative solvers in `Diffpack`. We consider the solution of systems of equations as arising in linear elasticity, non-symmetric equations as in convection-diffusion problems, anisotropic operators and bad conditioned equations as for jumping coefficients problems. In the introductory report only the Laplacian and smooth coefficients were treated. The first steps are guided by a couple of examples and exercises.

## Contents

1 Introduction 1

2 Systems of Equations, linear Elasticity 2

3 Convection-Diffusion equation 8

4 Anisotropic problems 14

5 Rough coefficient operators 19

6 Different models on different scales 26

7 Conclusion 27

References 28

# Multigrid applied to different partial differential operators

Gerhard W. Zumbusch *

November 22, 1996

## 1 Introduction

The solution of partial differential equations often leads to the solution of equation systems. For large problem sizes this solution tends to dominate the overall complexity of the whole simulation. Hence efficient equation solver like the multigrid method are needed. The idea is to construct an iterative solver based on several discretizations on different scales. The multigrid method reaches optimal linear complexity which is comparable to the assembly and input/output procedures in a finite element computation.

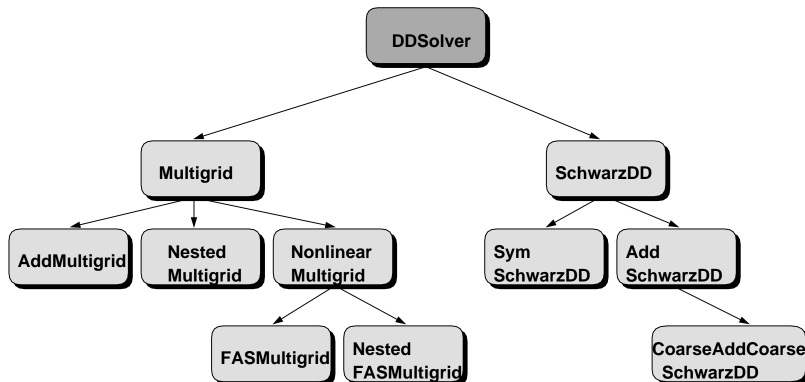



Figure 1: Hierarchy of multigrid and domain decomposition methods

Multigrid methods and domain decomposition methods are implemented in `Diffpack` in a common framework applicable to iterative solvers, preconditioners and nonlinear solvers. The user has to add approximative solvers on the different discretizations and grid transfer operators projecting and interpolating residuals and corrections from one discretization to another. These components are specified in the `DDSolverUDC` interface in `Diffpack`.

The multigrid algorithm itself applies the approximate solvers on the different discretizations and uses coarse (= cheap) discretization to correct solutions on finer (= expensive) discretization. The standard way to do this is called V-cycle.

*SINTEF Applied Mathematics. **Email:** `Gerhard.Zumbusch@math.sintef.no`.

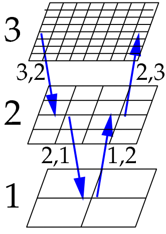



Figure 2: Multigrid V-Cycle

The algorithm may be written recursively like this

algorithm

$$\begin{aligned} x^1 &= \mathcal{S}^1(x,b) \\ x^2 &= x^1 + R_{j-1,j}\Phi_{j-1}(0, R_{j,j-1}(b-\mathcal{L}_j x^1)) \\ \Phi_j(x,b) &= \mathcal{S}^2(x^2,b) \end{aligned}$$

where $\mathcal{S}$denote the approximative solvers and $R_{j-1,j}$ and $R_{j-1,j}$ are the grid transfer operators. The evaluation of the residual is denoted by $b - \mathcal{L}x$. The algorithm on level one can be defined as

$$\Phi_1(x,b) = \mathcal{S}(x,b)$$

We assume familiarity with some of the basic concepts of `Diffpack` [BL96, Lan94b]. We will use and modify some examples presented in in the multigrid introduction [Zum96]. For a more detailed presentation of the multigrid method we refer to text books like [Hac85] and other references found in [Zum96]. It may be helpful to have access to the `Diffpack` manual pages `dpman` while reading this tutorial. The source codes and all the input files are available at `$DPR/src/app/pde/ddfem/src/`.

The report is organized as follows: We treat each partial differential equation in a separate chapter. The first one is dedicated to linear elasticity, the next one is the convection-diffusion equation, then anisotropic problems and rough coefficient problems follow. We conclude with some remarks on the combination of several models on different scales in a multigrid method.

## 2 Systems of Equations, linear Elasticity

We want to extend the code to systems of equations. Take for example the Lameé equations of linear elasticity with conforming linear finite elements, displacement approach with low Poisson ratio (high compressibility).

$$\begin{aligned} \epsilon_{ij} &= \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \\ \sigma &= 2\mu\epsilon + \lambda(\text{trace } \epsilon)I - (3\lambda + 2\mu)\alpha T(x) \\ \text{div } \sigma &= 0 \end{aligned}$$

with positive Lamé constants $\lambda$ and $\mu$. We have an elliptic selfadjoint operator similar to the Laplace operator for the primary unknowns $u_i$ (displacement). We have two unknowns $u_i$ in two dimensions and three unknowns in three dimensions. We can apply multigrid in a block-sense, treating the displacement vector at each node as

one block. The grid transfer can be done for each component using the standard procedure. The smoother may be some block-relaxation scheme. Using point-wise smoother instead of block-wise may be dangerous, so at least we have to care about the node numbering.

The equations may also be written with positive constants, Young's elasticity modulus $e$ and the Poisson ratio $\nu < 1/2$

$$\begin{aligned} \lambda &= \frac{e\nu}{(1+\nu)(1-2\nu)} \\ \mu &= \frac{e}{2+2\nu} \end{aligned}$$

We start with the `Diffpack` multigrid simulator `MultiGrid2` of [Zum96] and a prototype implementation of linear elasticity given in `Diffpack` [Lan96]. The following input parameters may be some guideline for your experiments[1].

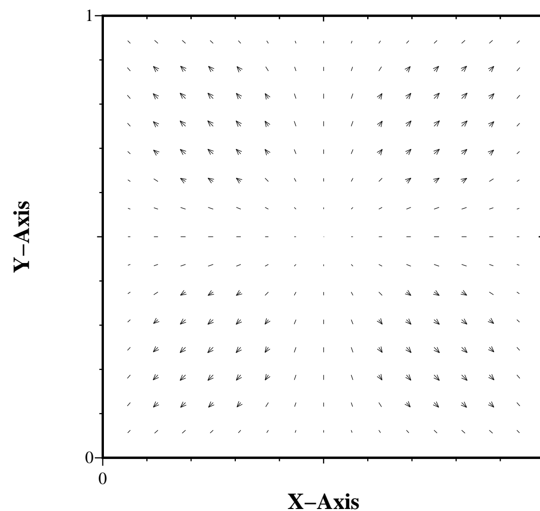



Figure 3: Displacement field, linear elasticity problem

MGOp4.h

```
// prevent multiple inclusion of MGOp4.h
#ifndef MGOp4_h_IS_INCLUDED
#define MGOp4_h_IS_INCLUDED

#include <MultiGrid2.h>

class MGOp4 : public MultiGrid2
{
protected:
  Handle(FieldsFE)  ud;              // displacement field
  real lambda, mu;                   // Lame's contants
  real E,nu;                         // Young's module and Poisson's ratio
```

[1] files are in MGOp4/Verify/

```
  real alpha;                       // temperature expansion coefficient
  virtual void integrands (ElmMatVec& elmat, FiniteElement& fe);
  void fillEssBC (SpaceId space);
public:
  MGOp4 ();
 ~MGOp4 () {}

  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu);
  virtual void solveProblem ();         // main driver routine
  virtual void resultReport ();         // write error norms to the screen
};
#endif
```

The new class is derived from `MultiGrid2`. It additionally contains the Lamé constant and the thermal expansion coefficient $\alpha$ and extends the menu accordingly. The displacement vector field $u_i$ is stored in a new data structure `Fields` instead of the `Field` data used previously. The `integrands` and `fillEssBC` are adapted to the new differential equation.

MGOp4.C

```
#include <MGOp4.h>
#include <DDIter.h>
#include <PrecDD.h>
#include <createLinEqSolver.h> // creating smoothers
#include <createDDSolver.h> // creating multigrid object
#include <FieldFE.h>

MGOp4:: MGOp4 () {}

void MGOp4:: define (MenuSystem& menu, int level)
{
  menu.addItem (level,
              "Young's module",
              "Emodule",
              "elasticity constant",
              "1.0",
              "R[0:1.0e+20]1");
  menu.addItem (level,
              "Poisson's ratio",
              "nu",
              "elasticity constant",
              "0.25",
              "R[0:1.0e+20]1");
  menu.addItem (level,
              "alpha",
              "alpha",
              "thermal expansion coefficient",
              "1.0",
              "R1");
  MultiGrid2:: define(menu, level);
}

void MGOp4:: scan (MenuSystem& menu)
{  // load answers from the menu:
```

```
alpha = menu.get ("alpha").getReal();
E = menu.get ("Young's module").getReal();
nu = menu.get ("Poisson's ratio").getReal();
lambda = (nu*E)/((1+nu)*(1-2*nu));
mu = 0.5*E/(1+nu);

no_of_grids = menu.get ("no of grid levels").getInt();
smooth.redim (no_of_grids);
system.redim (no_of_grids);
smooth_prm.redim (no_of_grids);
proj.redim (no_of_grids-1);
grid.redim (no_of_grids);
dof.redim (no_of_grids);
mat_prm.redim (no_of_grids-1);

scanGrids(menu); // scan and construct the hierarchy of grids

// allocate data structures in the class:
const nsd = grid(1)->getNoSpaceDim();
ud.rebind (new FieldsFE (grid(no_of_grids)(),"u")); // was FieldFE
int i;
for (i=1; i<=no_of_grids; i++)
  dof(i).rebind (new DegFreeFE (grid(i)(), nsd));  // nsd unknown per node
lineq.rebind (new LinEqAdm());
lineq->scan (menu);
linsol.redim (dof(no_of_grids)->getTotalNoDof());
lineq->attach (linsol);

precondPrm.scan(menu);
lineq->attach (precondPrm);

// read_sweeps
Is is(menu.get ("sweeps"));
is->ignore ('[');
is->get (preSmooth);
is->ignore (',');
is->get (postSmooth);

menu.setCommandPrefix("coarse grid");
for (i=1; i<=no_of_grids; i++) {
  smooth_prm(i).rebind(new prm(LinEqSolver));
  smooth_prm(i)->scan (menu);
  smooth(i).rebind(createLinEqSolver (smooth_prm(i)()));
  system(i).rebind(new LinEqSystemStd (EXTERNAL_STORAGE));
  menu.setCommandPrefix("smoother");
}
menu.unsetCommandPrefix();

for (i=1; i<no_of_grids; i++)
  proj(i).rebind(new ProjInterpSparse());

ddsolver_prm.scan(menu);
ddsolver = createDDSolver(ddsolver_prm);
ddsolver->attachUserCode(*this);

for (i=1; i<no_of_grids; i++) {
  mat_prm(i).rebind(new prm(Matrix(NUMT)) );
  mat_prm(i)->scan (menu);
  mat_prm(i)->sparse_adrs.rebind (new SparseDS);
```

```
  }
}

void MGOp4:: solveProblem () // main routine of class MGOp4
{
  initProj();
  initMatrices();

  fillEssBC (no_of_grids);                  // set essential boundary conditions
  makeSystem (dof(no_of_grids)(), lineq()); // calculate linear system

  system(no_of_grids)->attach(lineq->Al ());
  ddsolver->attachLinRhs(lineq->bl (), no_of_grids, dpFALSE);
  ddsolver->attachLinSol(lineq->xl (), no_of_grids);

  if (lineq->getSolver().description().contains("Domain Decomposition")) {
    BasicItSolver& sol = CAST_REF(lineq->getSolver(), BasicItSolver);
    DDIter& ddsol      = CAST_REF(sol, DDIter);
    ddsol.attach(*ddsolver);
  }

  Precond &prec =lineq->getPrec();
  if (prec.description().contains("Domain Decomposition")) {
    PrecDD& sol = CAST_REF(prec, PrecDD);
    sol.init(*ddsolver);
  }

  linsol.fill (0.0);           // set all entries to 0 in start vector
  dof(no_of_grids)->fillEssBC (linsol); // insert boundary values in start vector
  lineq->solve();              // solve linear system
  int niterations; BooLean c;  // for iterative solver statistics
  if (lineq->getStatistics(niterations,c))  // iterative solver?
    s_o << oform("\n\n *** solver%sconverged in %3d iterations ***\n\n",
                 c ? " " : " not ",niterations);

  // the solution is now in linsol, it must be copied to the ud fields:
  dof(no_of_grids)->vec2field (linsol, ud());
  Store4Plotting::dump (ud());                // dump for later visualization
}

void MGOp4:: resultReport () { }

void MGOp4:: fillEssBC (SpaceId space)
{
  dof(space)->initEssBC ();            // init for assignment below
  int nno = grid(space)->getNoNodes(); // no of nodes
  int nsd_ = grid(space)->getNoSpaceDim();
  int i,k;
  for (i = 1; i <= nno; i++)
    if (grid(space)->BoNode (i))    // is node i subj. to any boundary indicator?
      for (k = 1; k <= nsd_; k++)
        dof(space)->fillEssBC (dof(space)->fields2dof(i,k), 0.0); // u=0 at nodes on the boundary

  //dof(space)->printEssBC (s_o, 2); // for checking the essential boundary cond.
}

void MGOp4:: integrands (ElmMatVec& elmat, FiniteElement& fe)
{
  int i,j;    // basis function counters
```

```
  int k,r,s;  // 1,..,nsd (space dimension) counters
  int ig,jg;  // element dof, based on i,j,r,s

  real nabla2, shear_term, volume_term;
  const int nsd_ = fe.getNoSpaceDim();
  const int nbf = fe.getNoBasisFunc();
  const real detJxW = fe.detJxW();
  static Mat(real) matnod (nsd_,nsd_);

  // find the global coord. x of the current integration point:
  Ptv(real) x (nsd_);
  fe.getGlobalEvalPt (x);
  real f_value = f(x);
  for (i = 1; i <= nbf; i++) {
    for (j = 1; j <= nbf; j++) {
      nabla2 = 0;
      for (k = 1; k <= nsd_; k++)
        nabla2 += fe.dN(i,k)*fe.dN(j,k);

      for (r = 1; r <= nsd_; r++)
        for (s = 1; s <= nsd_; s++)
          matnod (r,s) = mu*fe.dN(i,s)*fe.dN(j,r);

      for (r = 1; r <= nsd_; r++)
        matnod (r,r) += mu*nabla2;

      for (r = 1; r <= nsd_; r++)
        for (s = 1; s <= nsd_; s++) {
          shear_term  = matnod(r,s);
          volume_term = lambda*fe.dN(i,r)*fe.dN(j,s);

          ig = nsd_*(i-1)+r;
          jg = nsd_*(j-1)+s;
          elmat.A(ig,jg) += (shear_term + volume_term)*detJxW;
        }
    }
    for (r = 1; r <= nsd_; r++) {
      shear_term  = 2*mu*    (alpha*f_value*fe.dN(i,r));
      volume_term = 3*lambda*(alpha*f_value*fe.dN(i,r));

      ig = nsd_*(i-1)+r;
      elmat.b(ig) += (shear_term + volume_term)*detJxW;
    }
  }
}
```

The Poisson ratio $\nu = 0$ leads to a Laplace type equation $\vec{\Delta} u = f$. We know how to solve such problems. The limit $\nu = 1/2$ means an incompressible material. The low order discretization of the primal unknowns, the displacement, is not suitable for this limit case. One rather prefers a mixed finite element formulation, including the strains $\epsilon_{ij}$ in the discretization. If we stick to the previous displacement approach, we have to keep away from this limit.

locking

Hence it is interesting to lock at the dependency on the Poisson ratio $\nu \to 1/2$. The phenomena of a bad primal discretization is often referred to as "locking". We can also observe trouble of an iterative solver for this limit, see table 1, input file

| **menu item** | **answer** |
|---|---|
| Young's module | 1.0 |
| Poisson's ratio | $\{0\ \&\ .2\ \&\ .3\ \&\ .4\ \&\ .45\}$ |
| alpha | 1.0 |
| no of grid levels | 4 |
| no of space dimensions | 2 |
| coarse partition | [2,2] |
| refinement | [2,2] |
| sweeps | [2,2] |
| basic method | DDIter |
| domain decomposition method | Multigrid |
| smoother basic method | SOR |
| smoother renumber unknowns | RenumNoUnknowns |

Table 1: Linear elasticity, different Poisson ratios, `test1.i`

`test1.i`.

| **menu item** | **answer** |
|---|---|
| Poisson's ratio | $\{0\ \&\ .2\ \&\ .3\ \&\ .4\ \&\ .45\}$ |
| no of grid levels | 3 |
| no of space dimensions | 3 |
| coarse partition | [2,2,2] |
| element type | ElmB8n3D |

Table 2: Linear elasticity, different Poisson ratios, 3D, `test2.i`

We can repeat the same test for dependence on the Poisson ratio $\nu \to 1/2$ on the three dimensional cube and compare the findings to the two dimensional case, see table 2, input file `test2.i`.

The construction of an efficient multigrid algorithm requires the tuning of its components. Since there is no block smoothing available right now (try to write one!), we have to be careful especially with the smoothing parameters. Compare the different pre- and post-smoothing variants, see table 3, input file `test3.i`.

Of course there are much more complicated cases of systems of equations. Usually some of the unknowns have to be treated differently than others.

# 3 Convection-Diffusion equation

Let us have a look at scalar convection-diffusion equations. We know how to deal with the diffusion term already. The convection term is of lower order and should not cause too much trouble on the finest grid. However, we have a non-symmetric equation system. We have to use a stable discretization which means some upwind scheme or artificial diffusion. Non-symmetric iterations on the finest grid work well,

| menu item | answer |
|---|---|
| Poisson's ratio | .3 |
| no of grid levels | 4 |
| no of space dimensions | 2 |
| coarse partition | [2,2] |
| refinement | [2,2] |
| sweeps | {[1,0] & [0,1] & [2,0] & [1,1] & [0,2]} |
| smoother basic method | SOR |

Table 3: Linear elasticity, different smoothing, `test3.i`

if this grid is fine enough (or we also apply stabilization).

$$-\Delta u \;+\; \vec{v}\cdot\nabla u \;-\; Hu \;=\; f$$

If we want to apply multigrid, we also have to use coarser discretizations. On coarser grids the lower order convection term $\vec{v}\cdot\nabla$ starts to dominate. This means more stabilization, e.g. artificial diffusion on coarser grids and this means less accurate and less useful discretizations on coarse grids. Hence very coarse grids do not contribute much to the multigrid performance.

The next issue is the smoother: To improve the performance of the smoothing we have to take the convection direction into account. The information and the errors are transported along the convection, the (isotropic) diffusion will be mall. Smoother transporting information upstream, against the convection direction will therefore perform best. We will do experiments with such line-oriented methods like zebra-line Gauss-Seidel (SOR) (figure 5). ILU with an upstream node ordering is also very successful.

In the case such an upstream node ordering is available, it is no problem to construct an efficient multigrid method. If not, one can use heuristics finding suitable orderings instead. Another approach is using smoothers for several directions at once, so the every possible convection direction is covered. This leads to line-smoothers treating whole lines exactly. Problems arise in three dimensions where smoothers may treat coordinate planes instead of lines. In this case smoothers become more expensive. Other approaches try to construct more robust smoothers as for example special versions of ILU.

We start with the `Diffpack` multigrid simulator `MultiGrid2` of [Zum96]. For a documentation of the discretization of the convection-diffusion equation and the use of upwind discretizations in `Diffpack` we refer to [Lan94a] and the manual pages of `UpwindFE`.

MGOp1.h

```
// prevent multiple inclusion of MGOp1.h
#ifndef MGOp1_h_IS_INCLUDED
#define MGOp1_h_IS_INCLUDED

#include <MultiGrid2.h>
```

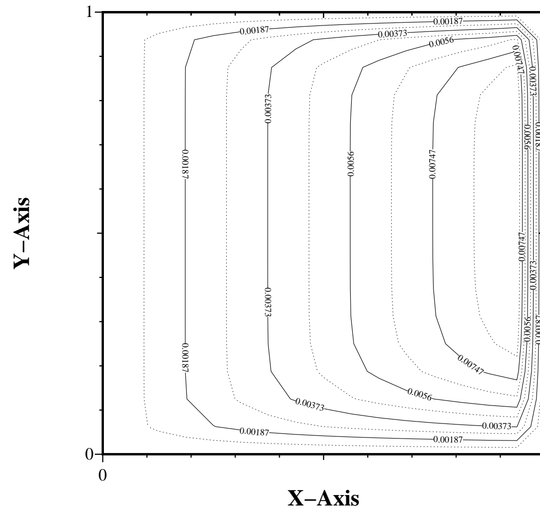



Figure 4: Isolines of an convection-diffusion solution.

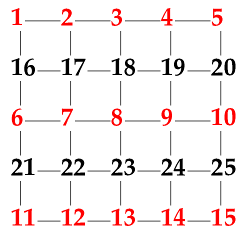



Figure 5: Zebra-line node ordering

```
#include <UpwindFE.h>

class MGOp1 : public MultiGrid2 // convection diffusion
{
protected:
  // general data:
  UpwindFE  PG;
  // coefficients in the equation:
  Ptv(real) velocity;
  Ptv(real) diffusion;
  Ptv(real) v_scale;  // scaling velocity

  virtual real f(const Ptv(real)& x);    // source term in the PDE
  virtual real k(const Ptv(real)& x);    // coefficient in the PDE
  virtual real h(const Ptv(real)& x);    // Helmholtz term in the PDE
  virtual void v(const Ptv(real)& x, Ptv(real)& v_); // velocity in the PDE

  virtual void integrands                // evaluate weak form in the FEM equations
    (ElmMatVec& elmat, FiniteElement& fe);
```

```
public:
  MGOp1 ();
 ~MGOp1 () {}

  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu);
  virtual void solveProblem ();          // main driver routine
  virtual void resultReport ();
};
#endif
```

The class is derived from MultiGrid2. It contains an UpwindFE object to enable the upwind discretization of the convection term. The additional coefficients are implemented as functions $v$ the velocity, $f$ the source term, $h$ and optional Helmholtz term and $k$ the coefficient for the second order operator. The procedures handling the input menu have been extended accordingly. The new integrands function implements the weak form of the differential operator including the upwind discretization of the convection.

MGOp1.C

```
#include <MGOp1.h>
#include <FiniteElement.h>
#include <Vec_real.h>
#include <DDIter.h>
#include <PrecDD.h>

MGOp1:: MGOp1 () {}

void MGOp1:: define (MenuSystem& menu, int level)
{
  UpwindFE:: defineStatic (menu, level+1);

  menu.addItem (level,
                "velocity", // menu command/name
                "velocity",        // command line option: +velocity
                "scale velocity",
                "[1.0,1.0]",        // default answer 2D
                "S");               // valid answer: String

  MultiGrid2::define(menu, level);
}

void MGOp1:: scan (MenuSystem& menu)
{
  // load answers from the menu:
  PG.scan (menu);

  MultiGrid2::scan(menu);

  const int nsd = grid(1)->getNoSpaceDim();
  velocity.redim(nsd);
  diffusion.redim(nsd);
  v_scale.redim(nsd);

  Is rIs(menu.get ("velocity"));
```

```
  rIs->ignore ('[');
  for (int i = 1; i <= nsd; i++) {
    rIs->get (v_scale(i));
    if (i < nsd)
      rIs->ignore (',');
  }
}

void MGOp1:: integrands (ElmMatVec& elmat, FiniteElement& fe)
{
  int i,j,q;
  const int nbf = fe.getNoBasisFunc(); // no of nodes (or basis functions)
  const real detJxW = fe.detJxW();     // det J times numerical itg.-weight
  const int nsd = fe.getNoSpaceDim();

  // find the global coord. x of the current integration point:
  Ptv(real) x (grid(1)->getNoSpaceDim());
  fe.getGlobalEvalPt (x);
  real f_value = f(x);
  real h_value = h(x);
  diffusion = k(x);
  v(x, velocity);
  PG.calcWeightingFunction(fe, velocity, diffusion, DUMMY, dpTRUE);

  real nabla_prod, conv;
  for (i = 1; i <= nbf; i++) {
    for (j = 1; j <= nbf; j++) {
      nabla_prod = 0;
      conv = 0;
      for (q = 1; q <= nsd; q++) {
        nabla_prod += diffusion(q) * PG.dW(i,q)*fe.dN(j,q);
        conv += velocity(q) * fe.dN(j,q);
      }
      conv *= PG.W(i);
      elmat.A(i,j) += (conv + nabla_prod - h_value*PG.W(i)*fe.N(j))
        *detJxW;
    }
    elmat.b(i) += PG.W(i)*f_value*detJxW;
  }
}

void MGOp1:: solveProblem () // main routine of class MGOp1
{
  initProj();
  initMatrices();

  fillEssBC (no_of_grids);                  // set essential boundary conditions
  makeSystem (dof(no_of_grids)(), lineq()); // calculate linear system

  system(no_of_grids)->attach(lineq->Al ());
  ddsolver->attachLinRhs(lineq->bl (), no_of_grids, dpFALSE);
  ddsolver->attachLinSol(lineq->xl (), no_of_grids);

  if (lineq->getSolver().description().contains("Domain Decomposition")) {
    BasicItSolver& sol = CAST_REF(lineq->getSolver(), BasicItSolver);
    DDIter& ddsol      = CAST_REF(sol, DDIter);
    ddsol.attach(*ddsolver);
  }
```

```
  Precond &prec =lineq->getPrec();
  if (prec.description().contains("Domain Decomposition")) {
    PrecDD& sol = CAST_REF(prec, PrecDD);
    sol.init(*ddsolver);
  }

  linsol.fill (0.0);           // set all entries to 0 in start vector
  dof(no_of_grids)->fillEssBC (linsol); // insert boundary values in start vector
  lineq->solve();              // solve linear system
  int niterations; BooLean c;  // for iterative solver statistics
  if (lineq->getStatistics(niterations,c))  // iterative solver?
    s_o << oform("\n\n *** solver%sconverged in %3d iterations ***\n\n",
                 c ? " " : " not ",niterations);

  // the solution is now in linsol, it must be copied to the u field:
  dof(no_of_grids)->vec2field (linsol, u());
  Store4Plotting::dump (u());                // dump for later visualization
   lineCurves(u());
}


void MGOp1:: resultReport ()
{
  if (grid(no_of_grids)->getNoNodes() < 300)
    u->values().print("FILE=u.dat","Nodal values of the solution field");
}

real MGOp1:: f (const Ptv(real)& /*x*/)
{
  return 1;
}

real MGOp1:: k (const Ptv(real)& /*x*/)
{
  return 1;
}

real MGOp1:: h (const Ptv(real)& /*x*/)
{
  return 0;
}

void MGOp1:: v(const Ptv(real)& /*x*/, Ptv(real)& v_)
{
  v_ = v_scale;
}
```

The following input parameters may be some guideline for your experiments[2].

We can apply different upwind schemes to the convection diffusion equation. The main parameter to play with is certainly the velocity $v$. Velocity zero means a standard symmetric Poisson equation. Velocities large compared to the grid size on the finest level are supposed to cause trouble. Remember that the upwind scheme is a first order discretization in contrast to second order discretization of the elliptic term.

[2] files are in MGOp1/Verify/

| **menu item** | **answer** |
|---|---|
| upwind weighting function method | {1 & 5} |
| velocity | [100,10] |
| no of space dimensions | 3 |
| no of grid levels | 4 |
| coarse partition | [2,2] |
| refinement | [2,2] |
| sweeps | [2,2] |
| matrix type | MatSparse |
| basic method | DDIter |
| domain decomposition method | NestedMultigrid |
| smoother basic method | SOR |
| smoother renumber unknowns | RenumNoUnknowns |

Table 4: Convection diffusion, `test1.i`

This means large velocities are discretized not that accurately and we encounter two problems: A low order discretization and trouble to solve the equations iteratively. As a starting point, see table 4, input file `test1.i`.

| **menu item** | **answer** |
|---|---|
| velocity | [100,10,10] |
| no of space dimensions | 3 |
| coarse partition | [2,2,2] |
| refinement | [2,2,2] |
| matrix type | MatSparse |

Table 5: Convection diffusion 3D, `test2.i`

Redo your experiments for the three dimensional case, starting with table 5, input file `test2.i`.

# 4 Anisotropic problems

We have already discussed semi-coarsening in the presence of anisotropic grids (chapter 4.3 in [Zum96]). This may also be necessary in the case of anisotropic operators.

We write an anisotropic second order operator in the form

$$\mathcal{L} \;=\; -\nabla K \nabla$$

with a positive symmetric coefficient tensor $A$. We can always diagonalize the tensor by rotating the coordinate system.

$$K = \mathrm{diag}(k_1, k_2, k_3)$$

The operator is anisotropic, if the eigenvalues differ. Problems arise, if the order of the eigenvalues differs. Standard estimates just use upper and lower bounds of the eigenvalues, which may be a large number in the case of a strong anisotropy.

On a isotropic mesh with small aspect ratio elements, which is for example elements of square or cube shape, the discretization error is anisotropic. In the limit case of a ratio $\max k_i / \min k_i$ tending to infinity we have a lower dimensional problem. We end up with a boundary value problem along one direction and independent values along a orthogonal direction. This also means that the error in an iterative procedure may behave anisotropic.

We can transform the differential equation to an isotropic one on a distorted domain. The grid on the domain is also distorted. The elements are distorted and have a large aspect ratio. It is immediately clear that we will also have trouble to solve the equation here, since standard estimates rely on the shape regularity of the elements.

The idea now is to change the discretization to a regular grid on the distorted grid with an isotropic operator. Constructing a hierarchy of grid suitable for multigrid on the distorted grid means to use semi-coarsening as long as the grid is distorted. Once an isotropic grid is reached, isotropic refinement can be used. This procedure corresponds to distorted grids on the original domain with an anisotropic operator until the anisotropic operator on the distorted grids appears as an isotropic operator.

Smoothers well suited for the distorted grids can be constructed to cope with the limit case. This means good solvers for the limit lower dimensional case. Such directional smoother can be constructed using SOR or ILU schemes with some line node ordering (see figure 5).

We start with the `Diffpack` multigrid simulator `MultiGrid2` of [Zum96].

MGOp3.h

```
// prevent multiple inclusion of MGOp3.h
#ifndef MGOp3_h_IS_INCLUDED
#define MGOp3_h_IS_INCLUDED

#include <MultiGrid2.h>

class MGOp3 : public MultiGrid2  // anisotropic operator
{
protected:
  MatSimple(real) k_tensor; // coefficient tensor
  virtual real f(const Ptv(real)& x);    // source term in the PDE
  virtual void k(const Ptv(real)& x, MatSimple(real)& k_);
    // coefficient tensor in the PDE
  virtual void integrands(ElmMatVec& elmat, FiniteElement& fe);
    // evaluate weak form in the FEM equations
public:
  MGOp3 ();
 ~MGOp3 () {}

  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu);
  virtual void solveProblem ();          // main driver routine
  virtual void resultReport ();          // write error norms to the screen
};
```

```
#endif
```

The class is derived from `MultiGrid2`. It contains an additional symmetric coefficient tensor `k_tensor` ($= K$) implemented as a two dimensional array. The menu handling procedures are extended accordingly. The function `integrands` implements the modified differential operator.

MGOp3.C

```
#include <MGOp3.h>
#include <DDIter.h>
#include <PrecDD.h>

MGOp3:: MGOp3 () {}

void MGOp3:: define (MenuSystem& menu, int level)
{
  menu.addItem (level,
                "k tensor",     // menu command/name
                "ktensor",      // command line option: +level
                "k tensor like (upper triangle) [1., 0., 1.]",
                "[1., 0., 1.]",  // default answer 2D
                "S");           // valid answer: string
  MultiGrid2:: define(menu, level);
}

void MGOp3:: scan (MenuSystem& menu)
{
  MultiGrid2:: scan(menu);

  const int nsd = grid(1)->getNoSpaceDim();
  k_tensor.redim(nsd, nsd);

  Is rIs(menu.get ("k tensor"));
  rIs->ignore ('[');
  for (int i = 1; i <= nsd; i++)
    for (int j = i; j <= nsd; j++) {
      rIs->get (k_tensor(i,j));
      k_tensor(j,i) = k_tensor(i,j);
      if ((i < nsd)||(j < nsd))
rIs->ignore (',');
    }
  s_o<<"k_tensor =\n"<<k_tensor<<endl<<endl;
}

void MGOp3:: solveProblem () // main routine of class MGOp3
{
  initProj();
  initMatrices();

  fillEssBC (no_of_grids);                   // set essential boundary conditions
  makeSystem (dof(no_of_grids)(), lineq()); // calculate linear system

  system(no_of_grids)->attach(lineq->Al ());
  ddsolver->attachLinRhs(lineq->bl (), no_of_grids, dpFALSE);
  ddsolver->attachLinSol(lineq->xl (), no_of_grids);
```

```
  if (lineq->getSolver().description().contains("Domain Decomposition")) {
    BasicItSolver& sol = CAST_REF(lineq->getSolver(), BasicItSolver);
    DDIter& ddsol      = CAST_REF(sol, DDIter);
    ddsol.attach(*ddsolver);
  }

  Precond &prec =lineq->getPrec();
  if (prec.description().contains("Domain Decomposition")) {
    PrecDD& sol = CAST_REF(prec, PrecDD);
    sol.init(*ddsolver);
  }

  linsol.fill (0.0);           // set all entries to 0 in start vector
  dof(no_of_grids)->fillEssBC (linsol); // insert boundary values in start vector
  lineq->solve();              // solve linear system
  int niterations; BooLean c;  // for iterative solver statistics
  if (lineq->getStatistics(niterations,c))  // iterative solver?
    s_o << oform("\n\n *** solver%sconverged in %3d iterations ***\n\n",
                 c ? " " : " not ",niterations);

  // the solution is now in linsol, it must be copied to the u field:
  dof(no_of_grids)->vec2field (linsol, u());
  Store4Plotting::dump (u());                // dump for later visualization
   lineCurves(u());
}


void MGOp3:: resultReport ()
{
  if (grid(no_of_grids)->getNoNodes() < 300)
    u->values().print("FILE=u.dat","Nodal values of the error field");
}

void MGOp3:: integrands (ElmMatVec& elmat, FiniteElement& fe)
{
  int i,j,q,r;
  const int nbf = fe.getNoBasisFunc(); // no of nodes (or basis functions)
  const real detJxW = fe.detJxW();     // det J times numerical itg.-weight
  const int nsd = fe.getNoSpaceDim();

  // find the global coord. x of the current integration point:
  Ptv(real) x (nsd);
  MatSimple(real) k_value(nsd,nsd);
  fe.getGlobalEvalPt (x);
  real f_value = f(x);
  k(x, k_value);

  real nabla_prod;
  for (i = 1; i <= nbf; i++) {
    for (j = 1; j <= nbf; j++) {
      nabla_prod = 0;
      for (q = 1; q <= nsd; q++)
for (r = 1; r <= nsd; r++)
  nabla_prod += k_value(q,r) * fe.dN(i,q) * fe.dN(j,r);

      elmat.A(i,j) += nabla_prod*detJxW;
    }
    elmat.b(i) += fe.N(i)*f_value*detJxW;
```

```
  }
}

real MGOp3:: f (const Ptv(real)& /*x*/)
{
  return 1.;
}

void MGOp3:: k (const Ptv(real)& x, MatSimple(real)& k_)
{
  k_ = k_tensor;
}
```

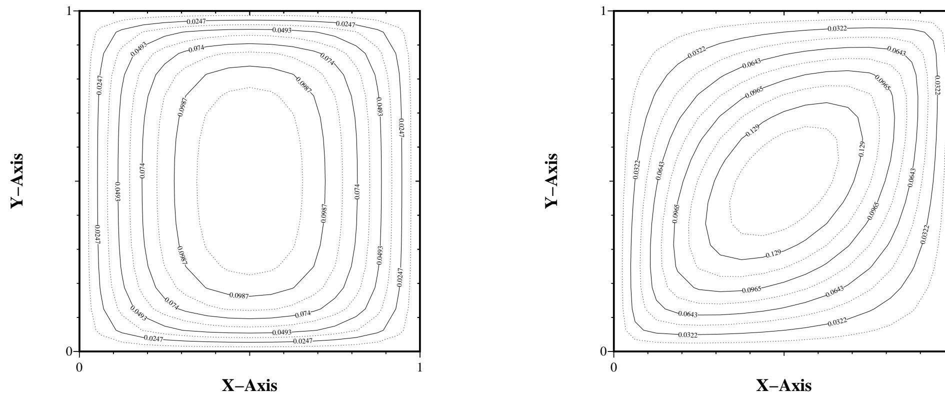



Figure 6: Isolines of anisotropic solutions.

The following input parameters may be some guideline for your experiments[3].

| menu item | answer |
|---|---|
| k tensor | {[1, 0, 1] & [10, 0, 1] & [1, 0, 10] & [1, 0, .1] & [.1, 0, 1] & [.01, 0, 1]} |
| no of grid levels | 4 |
| no of space dimensions | 2 |
| coarse partition | [2,2] |
| refinement | [2,2] |
| sweeps | [2,2] |
| matrix type | MatSparse |
| basic method | DDIter |
| domain decomposition method | Multigrid |
| smoother basic method | SOR |
| smoother renumber unknowns | RenumNoUnknowns |

Table 6: Anisotropic operator, `test1.i`

The play parameter here is of course the coefficient tensor $K$. It is specified in the

[3]files are in `MGOp3/Verify/`

format `[`$k_{11}$`, `$k_{12}$`, `$k_{22}$`]` in two dimensions. The first input file distorts either the $x_1$ or the $x_2$ axis by a factor of 10, see table 6, input file `test1.i`. We are using a SOR smoother which is not completely isotropic, but uses a specific iteration order. Hence distortion parallel to the $x_1$ or the $x_2$ axis may have a different effect. The question is, how this multigrid version behaves for these data. Further experiments could cover semi-coarsening and modification of the smoothers (renumbering!).

| **menu item** | **answer** |
|---|---:|
| k tensor | {[1, 0, .1] & [.55, .45, .55] & [.1, 0, 1] & [.55, -.45, .55]} |
| no of space dimensions | 2 |
| coarse partition | [2,2] |
| refinement | [2,2] |

Table 7: Anisotropic operator, rotated by $\pi/4$, `test2.i`

The next test includes distortion in different directions, rotating the main axes of the coefficient tensor, see table 7, input file `test2.i`. Observe how the smoothers handle distortion parallel to the diagonal compared to distortion parallel one coordinate axis.

| **menu item** | **answer** |
|---|---:|
| k tensor | {[1, 0, .01] & [1, 0, .001] & [1, 0, .0001]} |
| no of space dimensions | 2 |
| coarse partition | [2,16] |
| refinement | [2,1] |
| coarse grid basic method | SOR |

Table 8: Anisotropic operator, anisotropic refinement, `test3.i`

The next test covers different ratios of distortion, see table 8, input file `test3.i`. We employ semi-coarsening to guarantee convergence of the method. Observe both the difficulties in solving the problem as the large differences in the discretization parallel to the coordinate axes.

The last test covers the three dimensional case. Even if we only rotate the coefficient tensor by $\pi/4$, there are many different cases. We use a distortion by a factor of 10 along different directions, see table 9, input file `test4.i`.

# 5 Rough coefficient operators

We are solving a second order differential equation with a differential operator L. We write the operator in the form

$$\mathcal{L} \;=\; -\nabla K \nabla$$

with a (symmetric) coefficient tensor $K(x)$ which usually depends on space. Usually we require $L^{\inf}$ bounds for $K(x)$ and lower and upper bounds $k_1$, $k_2$ for the eigenvalues.

$$0 < k_1 \leq yK(x)y \leq k_2 \quad \forall x \in \Omega \text{ and } ||y||_2 = 1$$

| menu item | answer |
|---|---|
| k tensor | { [1, 0, 0, 1, 0, 1] &<br>[.1, 0, 0, 1, 0, 1] &<br>[.1, 0, 0, .1, 0, 1] &<br>[.55, .45, 0, .55, 0, 1] &<br>[.55, .45, 0, .55, 0, .1] &<br>[.55, .45, 0, .775, .225, .775]} |
| no of grid levels | 3 |
| no of space dimensions | 3 |
| coarse partition | [2,2,2] |
| refinement | [2,2,2] |
| element type = ElmB8n3D | |

Table 9: Anisotropic operator 3D, `test4.i`

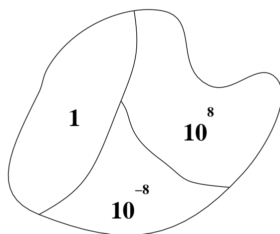



Figure 7: Domain with rough coefficients

Standard estimates just make use of an upper bound for the ratio $k_2/k_1$. In the case of large variation in $K(x)$ this at least leads to bad a priori estimates. Such a variation often is caused by the presence of different media joined together in the domain having different coefficients. The jumps in the coefficients may be several orders of magnitude. This definitely affects the quality of the discretization and the performance of iterative solvers.

resolved on coarse grid

In the case the different media zones are resolved on the coarse grid which means the interface between different media coincides with element interfaces on the coarse grid we usually do not have to care about the jumps. We assume to have smooth $A(x)$ with low variation inside each medium. Applying multigrid means that the coefficient jumps are visible on each level. Even the additive multigrid with an exact course grid solver and Jacobi smoother, which has to handle the rough coefficient problem, works fine. This means that the convergence does not depend on the size of the material jumps at the interface but on the local ratios $a_2/a_1$ in each medium.

Galerkin product

The problem becomes harder if the coarse grid does not cover the coefficient jumps. This may be the case of geometrically complicated shapes of the interfaces or too many domains and interfaces with rough coefficients. In the case we cannot resolve the interfaces correctly because of their shape, we can apply the following procedure. The assumption is that each separate domain of a medium on the finest grid is represented on the coarsest grid with at least one point. Hence we have a geometrically crude representation of the geometry on the coarsest grid. The idea is now to con-

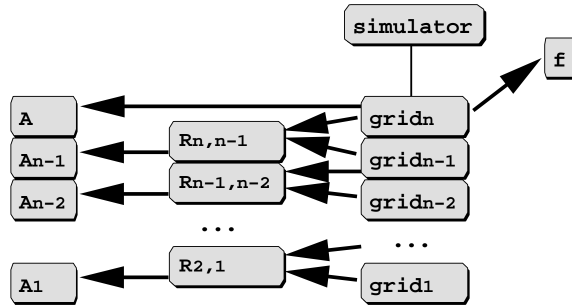



Figure 8: Galerkin products to construct stiffness matrices

struct specific operators approximating the coefficient jumps on coarser grids. We can do this using Galerkin products to construct stiffness matrices on coarser levels

$$A_{j-1} \;=\; R_{j,j-1}\, A_j\, R^*_{j,j-1}$$

This kind of averaging constructs quite good coarse grid problems compared to simple averaging or more advanced homogenization procedures.

Krylov iteration

In the case the coarsest grid is not even approximately suited for representing the coefficient jumps which means there are more different domains of media than degrees of freedom on the coarsest grid, we have to use different estimates. The goal is to have an iteration which is not sensitive to the size of coefficient jumps. We assume that the coefficient jumps can be resolved on the finest grid without averaging. We additionally assume some appropriate right hand side with jumps only on media interfaces. If we now employ some Krylov iteration preconditioned with a multigrid we will obtain a low dependence on the coefficients. This is true because of some multiple extreme eigenvalues immediately removed by the Krylov iteration. In the case of averaging the values on the finest grid the spectrum will smear out instead of containing multiple eigenvalues destroying the convergence rate.

In the case all restrictions and assumptions above are still too strict there are some properties influencing the convergence rate. Analyzing the error of iterative solvers applied to rough coefficient problems resembles in the question of information transport across the media interfaces. In the case of quasi-monotone distribution of coefficients iterative solvers usually perform better than for randomly distributed coefficients. This means there is always a path monotone in the coefficients across edges (faces) of the media interface connecting media with two different coefficients.

Another approach is to still use Galerkin products to construct the coarser grid stiffness matrices but to use algebraic multigrid heuristics to construct better transfer operators. In this case even a coarsest grid with one degree of freedom may make sense.

adaptivity

One comment on discretization error in the presence of large coefficient jumps: One question is the solution of such systems, but another question is of course about the discretization error of such a fine grid. Given a smooth right hand side one expects large errors in domains with small coefficients. This calls for refined grids and smaller elements in that area. Such an adaptive refinement both improves the global

discretization error and the performance of the iterative solver since the coefficient jumps visible in the stiffness matrix vanish. The situation changes with rough right hand side and low values inside domains with low coefficients (see above).

mixed methods

In the field of rough coefficients mixed finite element discretization can be applied successfully.

We start with the `Diffpack` multigrid simulator `MultiGrid2` of [Zum96].

MGOp2.h

```
// prevent multiple inclusion of MGOp2.h
#ifndef MGOp2_h_IS_INCLUDED
#define MGOp2_h_IS_INCLUDED

#include <MultiGrid2.h>

class MGOp2 : public MultiGrid2  // discontinous coefficients
{
protected:
  real    f_inner, k_inner; // coefficients inside [1/3,2/3]^d, outside 1
  virtual real f(const Ptv(real)& x);    // source term in the PDE
  virtual real k(const Ptv(real)& x);    // coefficient in the PDE
public:
  MGOp2 ();
 ~MGOp2 () {}

  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu);
  virtual void solveProblem ();          // main driver routine
  virtual void resultReport ();          // write error norms to the screen
};
#endif
```

The class is derived from `MultiGrid2`. It implements basically a new coefficient function `k` using the parameter `k_inner` in the domain $[1/3, 2/3]^2$ and the coefficient 1 elsewhere. The source term is implemented similarly with a parameter `f_inner` in the domain $[1/3, 2/3]^2$ and the value 1 elsewhere. The menu handling procedures are extended for the new parameters.

MGOp2.C

```
#include <MGOp2.h>
#include <DDIter.h>
#include <PrecDD.h>

MGOp2:: MGOp2 () {}

void MGOp2:: define (MenuSystem& menu, int level)
{
  menu.addItem (level,
              "inner rhs value", // menu command/name
              "irhs",        // command line option: +level
              "inner rhs value",
              "1.0",           // default answer
```

```
                "R1");          // valid answer: 1 real
  menu.addItem (level,
                "inner k value", // menu command/name
                "ik",        // command line option: +level
                "inner coefficient k",
                "1.0",           // default answer
                "R1");          // valid answer: 1 real
  MultiGrid2:: define(menu, level);
}

void MGOp2:: scan (MenuSystem& menu)
{
  f_inner = menu.get ("inner rhs value").getReal();
  k_inner = menu.get ("inner k value").getReal();
  MultiGrid2:: scan(menu);
}

void MGOp2:: solveProblem () // main routine of class MGOp2
{
  initProj();
  initMatrices();

  fillEssBC (no_of_grids);                  // set essential boundary conditions
  makeSystem (dof(no_of_grids)(), lineq()); // calculate linear system

  system(no_of_grids)->attach(lineq->Al ());
  ddsolver->attachLinRhs(lineq->bl (), no_of_grids, dpFALSE);
  ddsolver->attachLinSol(lineq->xl (), no_of_grids);

  if (lineq->getSolver().description().contains("Domain Decomposition")) {
    BasicItSolver& sol = CAST_REF(lineq->getSolver(), BasicItSolver);
    DDIter& ddsol      = CAST_REF(sol, DDIter);
    ddsol.attach(*ddsolver);
  }

  Precond &prec =lineq->getPrec();
  if (prec.description().contains("Domain Decomposition")) {
    PrecDD& sol = CAST_REF(prec, PrecDD);
    sol.init(*ddsolver);
  }

  linsol.fill (0.0);           // set all entries to 0 in start vector
  dof(no_of_grids)->fillEssBC (linsol); // insert boundary values in start vector
  lineq->solve();              // solve linear system
  int niterations; BooLean c;  // for iterative solver statistics
  if (lineq->getStatistics(niterations,c))  // iterative solver?
    s_o << oform("\n\n *** solver%sconverged in %3d iterations ***\n\n",
                 c ? " " : " not ",niterations);

  // the solution is now in linsol, it must be copied to the u field:
  dof(no_of_grids)->vec2field (linsol, u());
  Store4Plotting::dump (u());                // dump for later visualization
   lineCurves(u());
}


void MGOp2:: resultReport ()
{
  if (grid(no_of_grids)->getNoNodes() < 300)
```

```
    u->values().print("FILE=u.dat","Nodal values of the error field");
}

real MGOp2:: f (const Ptv(real)& x)
// coefficients inside [1/3,2/3]^d, outside 1
{
  const real a = 1./3.;
  const real b = 2./3.;
  const int nsd = grid(1)->getNoSpaceDim();
  for (int i = 1; i <= nsd; i++)
    if ((x(i)<a)||(x(i)>b)) return 1.;
  return f_inner;
}

real MGOp2:: k (const Ptv(real)& x)
{
  const real a = 1./3.;
  const real b = 2./3.;
  const int nsd = grid(1)->getNoSpaceDim();
  for (int i = 1; i <= nsd; i++)
    if ((x(i)<a)||(x(i)>b)) return 1.;
  return k_inner;
}
```

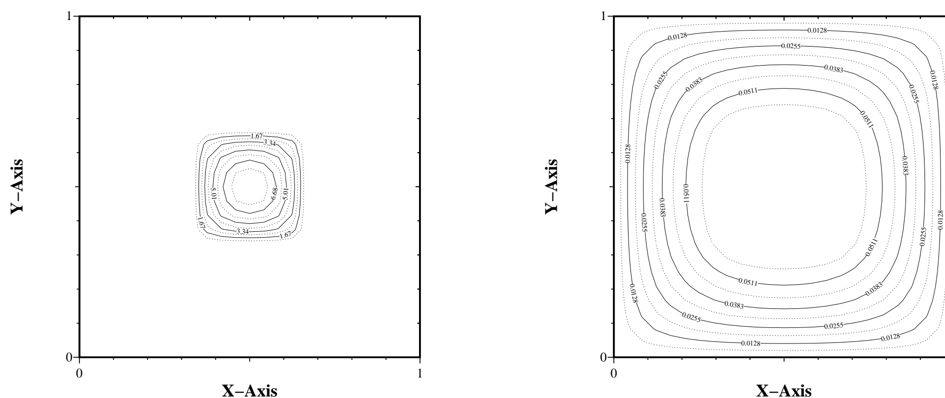



Figure 9: Isolines of solutions with low $k$ (left) and with high $k$ (right).

The following input parameters may be some guideline for your experiments[4].

First we look at the case where the coefficient jumps are resolved on the coarse grid using a $3 \times 3$ partition of the unit square. The play parameters are the values for the coefficient function and the source term inside the domain $[1/3, 2/3]^2$. The cases $f = k = 1$ is included. The question is how the number of iteration behaves for extreme values of $f$ and $k$, if the values of $f$ and $k$ are changed independently or if they are changed correspondingly. If the values are changed correspondingly, the equations inside $[1/3, 2/3]^2$ and outside $[1/3, 2/3]^2$ are equivalent. There remains the coefficient jumps at the inner boundary. If the values of $f$ and $k$ are changed independently,

[4] files are in MGOp2/Verify/

| menu item | answer |
|---|---:|
| inner rhs value | {0 & .001 & 1 & 1000} |
| inner k value | {.001 & 1 & 1000} |
| no of grid levels | 4 |
| no of space dimensions | 2 |
| coarse partition | [3,3] |
| refinement | [2,2] |
| sweeps | [2,2] |
| element type | ElmB4n2D |
| basic method | DDIter |
| domain decomposition method | Multigrid |
| smoother basic method | SOR |

Table 10: Discontinuous coefficients resolved on the coarsest grid, `test1.i`

either the solution inside or the solution outside of $[1/3, 2/3]^2$ dominates, while the solution is almost constant elsewhere, see table 10, input file `test1.i`.

| menu item | answer |
|---|---:|
| inner rhs value | {0. & 1. & 1000.} |
| inner k value | {0.001 & 1000.} |
| coarse partition | {[2,2] & [4,4]} |
| relative quadrature order | 1 |

Table 11: Discontinuous coefficients not aligned with / not resolved on the coarse grid, `test2.i`

We can redo the computations for the case of a non-matching coarsest grid. We study some effects of the multigrid solver. This can be either too coarse to resolve the coefficient jumps (partition $2\times 2$) or fine enough but simply not aligned (partition $4\times 4$). Compare these two case with the case of an aligned coarsest grid, see table 11, input file `test2.i`. There will be one case of divergence due to jumping coefficients.

| menu item | answer |
|---|---:|
| inner rhs value | {0. & 1. & 1000.} |
| inner k value | {0.001 & 1000.} |
| no of grid levels | 3 |
| no of space dimensions | 3 |
| coarse partition | [3,3,3] |
| refinement | [2,2,2] |
| element type | ElmB8n3D |

Table 12: Discontinuous coefficients aligned with the coarse grid, 3D, `test3.i`

We can redo the computations for the three dimensional case on the unit cube. The coefficients and source term functions are define inside and outside of $[1/3, 2/3]^3$. The

coarsest grid is chosen to resolve the jumps with a partition of $3 \times 3 \times 3$, see table 12, input file `test3.i`.

# 6 Different models on different scales

As we already saw the type of differential equation may change with the scale of the discretization. Looking at a convection-diffusion problem on a fine level reveals a diffusion equation with some unsymmetric perturbation caused by the convection term (section 3). Restricting this equation to a coarse grid means a dominant convection term with some small additional diffusion which is a transport equation. Numerical methods to discretize and solve both equations usually differ because the equations differ. Since a multigrid method makes use of both scales, we can think of combining two different numerical methods on two different scales into one multigrid method.

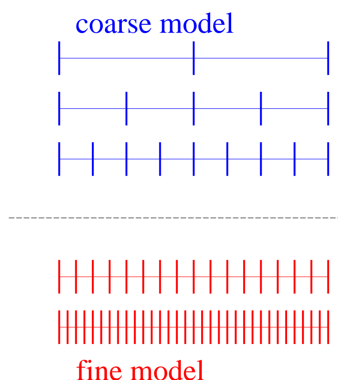



Figure 10: Different models on different scales

There are some other examples of physical models changing with scale:

| **fine scale** | **coarse scale** |
|---|---|
| convection-diffusion | transport |
| wave-optics | geometrical optics |
| wave-mechanics | geometrical optics |
| particle system | gas dynamics |
| quantum mechanics | particle system |

Building a multigrid method on top of discretizations on different scales requires several levels in between and transfer operators between the levels. Both can cause trouble: Using two completely distinct models means using one model on some levels and switching to another model on the rest. There is an abrupt change in the model between two levels. The models have to be compatible in some sense to construct meaningful grid transfer operators. Both constructing grid transfer operators and determining the model switching scale may be difficult.

Having some models continuously depending on some scale parameters leads to smoother transitions from one level to the next. Hence grid transfer operations can be constructed more naturally. The models on different levels are compatible.

# 7 Conclusion

In this report we have demonstrated the use of multigrid equation solvers for the finite element discretization of different partial differential equations. While the previous introductory report on multigrid covered scalar Poisson equation, smooth linear and nonlinear coefficient problems, this reports extends the variety of equations. We applied multigrid to systems of equations implementing linear thermo elasticity, to non-symmetric problems implementing the convection-diffusion equation with an upwind schemes, to jumping coefficient problems and to anisotropic problems.

The simulators were based on a standard Poisson equation simulator with multigrid developed in the introductory report. The extensions of the simulator to implement the different operators and input parameters were quite short. Some basic strategies for efficient multigrid were discussed, while no changes to the multigrid implementation of the simulators were applied. Although implementation of more advanced strategies to deal with different operators were discussed and are straightforward to implement in the framework, not all of them were actually implemented.

This discussion of different partial differential equations is necessarily incomplete and we have to refer to the literature for the treatment of other operators.

## References


[BL96] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations; Diffpack. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics.* Birkhäuser, 1996.

[Hac85] W. Hackbusch. *Multi–Grid Methods and Applications.* Springer, Berlin, 1985.

[Lan94a] H. P. Langtangen. A Diffpack module for scalar convection-diffusion equations. Technical Report $TIMR/doc/ps/doc/CD.ps, SINTEF Informatics, Oslo, 1994.

[Lan94b] H. P. Langtangen. Getting started with finite element programming in Diffpack. Technical Report STF33 A94050, SINTEF Informatics, Oslo, 1994.

[Lan96] H. P. Langtangen. A solver for the equations of linear thermo-elaticity. Technical Report $TIMR/doc/ps/doc/EL.ps, SINTEF Applied Mathematics, Oslo, 1996.

[Zum96] G. W. Zumbusch. Multigrid methods in Diffpack. Technical Report STF42 F96016, SINTEF Applied Mathematics, Oslo, 1996.