Multigrid for different finite differences equations

Gerhard W. Zumbusch

The `Diffpack` Report Series

November 22, 1996

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/`.

Abstract

The report is a continuation of an introductory report on the multigrid iterative solvers for finite differences in `Diffpack`. We consider the solution of partial differential equations discretized by finite differences. We consider varying coefficient and anisotropic operators and a variety of strategies for the convection-diffusion equation and the biharmonic equation. In the introductory report only the Laplacian was treated. We also discuss different multigrid restriction and prolongation operators arising in some special multigrid versions. The first steps are guided by a couple of examples.

Contents

Multigrid for different finite differences equations

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.

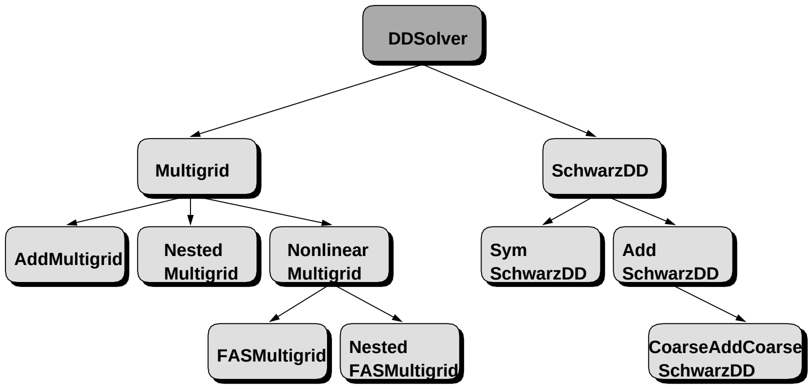

Figure 1: Hierarchy of multigrid and domain decomposition methods

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`. For details we refer to the introductory tutorial on the implementation of multigrid methods for finite differences [Zum96b].

algorithm

The V-cycle algorithm (figure 2) may be written recursively like this

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

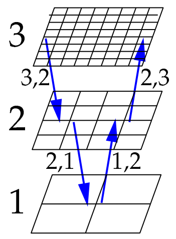

Figure 2: Multigrid V-Cycle

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

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]. We will extend the code presented in in the multigrid introduction [Zum96b]. For a more detailed presentation of the multigrid method (in the context of finite differences) we refer to text books like [Hac85, Wes92] and other references found in [Zum96c]. 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: First we will discuss the case of variable coefficient Laplace type equations. This covers the preconditioning with constant coefficients and with the operator itself. Next we have a look at anisotropic operators, which is equivalent with a Laplace operator discretized on a distorted grid. In the following chapter we experiment with different restriction and prolongation stars commonly used in multigrid methods for finite differences, especially some higher order stars. Next we treat the convection-diffusion equation in detail, with an artificial viscosity discretization, an upwind discretization and a discretization with Galerkin products and operator dependent restriction and prolongation in an algebraic multigrid fashion. In the last chapter we discuss a multigrid scheme for the finite difference discretization of the biharmonic equation. Some conclusions follow.

2 Variable coefficients

In the introductory tutorial on multigrid methods for finite difference discretization we only looked at the discretized Laplacian.

$$\Delta \approx \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$$

This five-point difference stencil is applied to every interior node and forms the stiffness matrix of the linear equation system to be solved by the multigrid method. However, the stiffness matrix itself is never formed. Only the difference stencil needs to be stored, which is five real numbers, independently of the number of degrees of freedom and the size of the stiffness matrix. It can be considered as a very memory efficient storage scheme. Of course this holds only for the constant coefficient case.

In the case that the difference stencils varies in the domain, we have to modify this storage scheme. In order to maintain a minimal storage representation, we choose a procedural representation of the difference stencils. This is accomplished using the `PtOpWf` point operator function in `Diffpack`. We implement the general symmetric, elliptic, varying coefficient problem

$$\begin{aligned} \nabla(k\nabla u) &= f && \text{on } \Omega \\ u &= g_1 && \text{on } \Gamma \subset \partial\Omega \\ \frac{\partial}{\partial n} u &= g_2 && \text{on } \partial\Omega \setminus \Gamma \end{aligned}$$

with positive variable coefficients $k(x)$ in Ω.

2.1 Preconditioning with constant coefficients

We derive two different multigrid codes. The first one is only suitable as a preconditioner in a Krylov iteration method, preferably a conjugated gradient method. The idea is to use the standard multigrid method as it has been implemented for the Laplacian already.

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$$

The variable coefficient function/ tensor $k(x)$ has to fulfill certain conditions. It has to be bounded and bounded away from zero

$$0 < k_0 \leq k(x) \leq k_1$$

or in the tensor case

$$0 < k_0 ||y||_2 \leq y^t k(x) y \leq k_1 ||y||_2 \text{ for all } y \neq 0$$

Given this condition, the operator $\nabla k \nabla$ is norm-equivalent to the Laplacian ∇^2 with an equivalence constant k_1/k_0. This justifies the use of a Laplacian based preconditioner for a more complicated operator problem.

The implementation in `Diffpack` is based on the sample multigrid implementation for finite differences `MGfdm2` described in the introductory tutorial [Zum96b]. The discretization of the linear equation system to be solved has to be modified. Hence the procedure `makeMatrix()` is overloaded. The other parts of the code remain the same. In order to implement the variable coefficients, we derive a point operator form the base class `PtOpWf`. It is constructed with two parameters: The grid parameter `h` and a scaling factor `k`. The basic functionality of this class `MGfdm4Wf` is implemented in the procedure `userfunc`. The header file follows:[1].

MGfdm4.h

```
#ifndef MGfdm4_h_IS_INCLUDED
#define MGfdm4_h_IS_INCLUDED

#include <MGfdm2.h>

class MGfdm4Wf: public PtOpWf(real)
{
protected:
  real h;              // step size
  real k;              // factor
public:
  MGfdm4Wf(real h_, real k_);
  virtual real userfunc(const Ptv(int)& index, const Ptv(int)& offset);
};

class MGfdm4: public MGfdm2
{
protected:
  virtual void makeMatrix();             // set up lineq Matrix
};
#endif
```

The main change in the procedure `makeMatrix()` is the use of the instances of the class `MGfdm4Wf` instead of simple `real` values. We have one instance for the central point 4 in the stencil and one instance for the other points -1. We also supply the grid-size.

The class `MGfdm4Wf` is used to define the `userfunc`. For a given central point `index` and the `offset` to the central point, the entry of the difference stencil has to be returned. We implement the variable coefficient function

$$k(x) \;=\; 1 \;+\; x_1 x_2 \;+\; (x_1 x_2)^2$$

The scaling factor `k` has been set to 4 or -1 to implement the difference stencil itself.

MGfdm4.C

```
#include <MGfdm4.h>

void MGfdm4:: makeMatrix()
{
  int n = gridSize(no_of_grids);

  Handle(MatPtOp(real)) A = new MatPtOp(real);

  Handle(IndexSet)       ind;       // 'interior' index set
  Handle(IndexSet)       bou;       // 'boundary' index set
```

[1]you will find the code in **MGfdm4/**

```
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),1);         // with an offset of one from
// the central element A(0,0).
  real h = 1.0 / real(n);

  MGfdm4Wf* af = new MGfdm4Wf(h, 4.);
  MGfdm4Wf* bf = new MGfdm4Wf(h, -1.);

                    (*A)( 0,-1) = *bf;
  (*A)(-1, 0) = *bf;  (*A)( 0, 0) = *af;  (*A)( 1, 0) = *bf;
                    (*A)( 0, 1) = *bf;

  FieldFD* b = new FieldFD(grid(no_of_grids)(),"b"); // !!
  b->values() = 0;

  u.rebind (new FieldFD(grid(no_of_grids)(),"u"));
  u->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    b->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions
  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    u->valueIndex(is(1),is(2)) = u0(ps);
  }

  lineq->attach(*A, *u, *b);
}

//----------------------------------------------------------------------

MGfdm4Wf:: MGfdm4Wf(real h_, real k_) : h(h_), k(k_)
{}

real MGfdm4Wf:: userfunc(const Ptv(int)& index, const Ptv(int)& offset)
{
  real s = (index(1) + offset(1)) * h * (index(2) + offset(2)) * h; // x * y
  return k * (1 + s * (1 + s));                          // 1 + xy + (xy)^2
}
```

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

[2]files are in `MGfdm4/Verify/`

have implemented a smooth varying coefficient problem. We can compare the performance of the multigrid method for this problem with the performance for the Laplace problem.

The first test compares the conjugated gradient method without any preconditioning, see table 1, input file `test1.i`. Here a comparison of the number of iterations or the convergence rate depending on the number of levels (the grid size) may be of interest. How do the number change due to the varying coefficients?

menu item	**answer**
no of grid levels	3
coarse lattice	2
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecNone

Table 1: Conjugated gradients for a variable coefficient problem, `test1.i`

The next test deals with the multigrid preconditioned conjugate gradient method, see table 2, input file `test3.i`. The number of iterations/ the convergence rate as a function of the number of levels may be interesting. Compare these numbers with the numbers for the pure Laplacian. How do the varying coefficients affect the convergence? Is it possible to compensate the effect by a higher number of smoothing sweeps, a higher cycle parameter `gamma` or a different smoothing or coarse grid algorithm? How do the additive multigrid and the nested multigrid algorithms perform for this problem?

menu item	**answer**
no of grid levels	3
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 2: Variable coefficient problem, conjugated gradients with Multigrid preconditioner, `test3.i`

2.2 Preconditioning with the true operator

The quality of the previous approach to use multigrid as a preconditioner strongly depends on the variable coefficients and the upper bound k_1/k_0. For a small ratio, the cheaper preconditioner may be an advantage, while for large variations we have to construct a better preconditioner. The idea is to incorporate the variable coefficients into the multigrid method. We use the variable coefficient operator on every level. So we can at least in part get rid of the k_1/k_0 dependency of the convergence rate. The success of this approach depends on the ability to resolve the variations of the coefficients on the coarser levels.

The implementation is based on the previous multigrid code `MGfdm4`. The discretization procedure `makeMatrix(int)` for the multigrid discretizations is overloaded. The code looks like this:[3].

MGfdm5.h

```
#ifndef MGfdm5_h_IS_INCLUDED
#define MGfdm5_h_IS_INCLUDED

#include <MGfdm4.h>

class MGfdm5: public MGfdm4
{
protected:
  virtual void makeMatrix(int i);        // set up smooth matrix
};
#endif
```

The implementation of the member function `makeMatrix(int)` is basically unchanged. The difference stencil is defined by the point operator functions defined in the previous section, in the same way, as it was done with the difference stencil for the equation system in `MGfdm4`.

MGfdm5.C

```
#include <MGfdm5.h>

void MGfdm5:: makeMatrix(int i)
{
  int n = gridSize(i);

  Handle(MatPtOp(real)) A;
  A.rebind( new MatPtOp(real) );

  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
```

[3]you will find the code in `MGfdm5/`

```
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

  real h = 1.0 / real(n);

// point operator, two dimensional
  A->redim(ind(),1);         // with an offset of one from
// the central element A(0,0).
  MGfdm4Wf* af = new MGfdm4Wf(h, 4.);
  MGfdm4Wf* bf = new MGfdm4Wf(h, -1.);

                      (*A)( 0,-1) = *bf;
  (*A)(-1, 0) = *bf;  (*A)( 0, 0) = *af;  (*A)( 1, 0) = *bf;
                      (*A)( 0, 1) = *bf;

  FieldFD* rhs = new FieldFD(grid(i)(),"rhs"); // !!
  rhs->values() = 0;

  FieldFD* sol = new FieldFD(grid(i)(),"sol"); // !!
  sol->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    rhs->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions

  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    sol->valueIndex(is(1),is(2)) = u0(ps);
  }

  smooth(i)->attach(*A, *sol, *rhs);
  LinEqAdm &s = smooth(i)->linAdm();
  ddsolver->attachLinRhs(s.bl(), i, dpTRUE);
  ddsolver->attachLinSol(s.xl(), i);
}
```

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

The first test is the application of the multigrid method as an iterative method. This is possible, since the multigrid now operates on the true differential equation. The questions here are concerned with the number of iterations and the convergence rate, see table 3, input file `test2.i`. The convergence rate depends on the smoother, the number of smoothing steps, the coarse grid solver and the size of the coarse grid.

[4]files are in `MGfdm5/Verify/`

However, the main question is always, whether the convergence rate is independent of the number of levels, that is the grid size.

menu item	**answer**
no of grid levels	3
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 3: Variable coefficient multigrid, `test2.i`

The second test can serve as a comparison, both with the multigrid method base on the Laplacian in the previous section and with the multigrid method as an iterative method, see table 4, input file `test3.i`. Since the multigrid implementation with the true operator, as it is done now, might be more expensive and slower, the question arises, whether this is compensated by an improved convergence of the Krylov method. You may want to check this for several multigrid parameters and grid sizes. However, it might also be interesting to modify the variable coefficient function to study this effect.

The second question is the comparison of preconditioner and iterative solver. The Krylov method around the multigrid method introduces some overhead, both in storage as in operations. Certainly the Krylov delivers some robustness and might increase the convergence for certain distributions of the eigenvalues of the differential operator. The question of course is, whether that pays of. You might also want to test this for different variable coefficient functions.

3 Anisotropic operator

In this chapter, we are studying anisotropic operators. We also refer to a related chapter for finite element computations in [Zum96a]. Given the model problem

$$\begin{aligned} \nabla(k\nabla u) &= f \quad \text{on } \Omega \\ u &= g_1 \quad \text{on } \Gamma \subset \partial\Omega \\ \frac{\partial}{\partial n}u &= g_2 \quad \text{on } \partial\Omega \setminus \Gamma \end{aligned}$$

with a symmetric positive definite tensor k, we study the influence of k on the equation solver. The tensor k can be diagonalized, rotating the coordinate system. We then

menu item	answer
no of grid levels	3
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 4: Conjugated gradients with variable coefficient multigrid preconditioner, `test3.i`

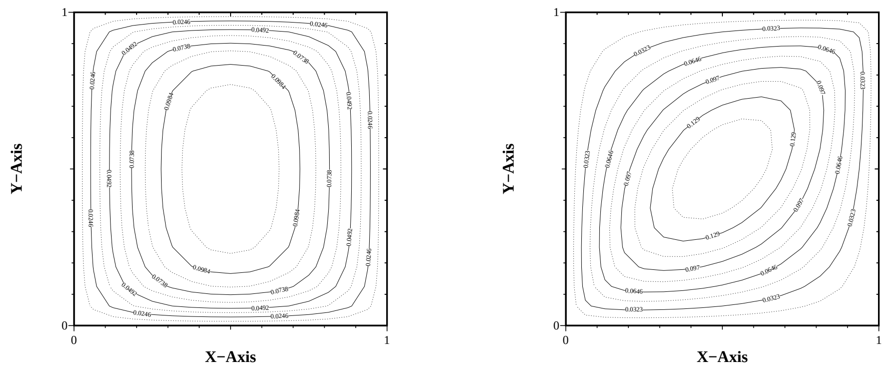

Figure 3: Solutions of an anisotropic problem. Left: anisotropy along y-axis, right: diagonal

have a tensor of the form

$$k = \begin{pmatrix} k_1 & 0 \\ 0 & k_2 \end{pmatrix}$$

Problem arise from the fact, that the element size of the discretization is not correlated to the behavior of the solution. While the elements are isotropic, that is square shaped in our example, the operator has a preference direction. The ratio k_2/k_1 measures the strength of the anisotropy. The source for the anisotropy may be physical material properties. Another possible source can of course be an anisotropic grid/ domain with an isotropic Laplace type operator.

In general the anisotropy is not aligned with a coordinate axis. This means that simple grid refinement does not remove the problem.

$$k = \begin{pmatrix} k_{11} & k_{12} \\ k_{12} & k_{22} \end{pmatrix}$$

The anisotropic operator is discretized as

$$k_1 \begin{bmatrix} & 0 & \\ -1 & 2 & -1 \\ & 0 & \end{bmatrix} + k_2 \begin{bmatrix} & -1 & \\ 0 & 2 & 0 \\ & -1 & \end{bmatrix} + k_3 \begin{bmatrix} & & -1 \\ & 2 & \\ -1 & & \end{bmatrix} + k_4 \begin{bmatrix} -1 & & \\ & 2 & \\ & & -1 \end{bmatrix}$$

with suitable coefficients k_1, k_2, k_3, and k_4, which are not uniquely determined by the differential operator. These parameters are read by the code. For anisotropy along coordinate axis, we use the equivalent of the five-point stencil, that is coefficients k_1 and k_2.

The code is derived from the standard multigrid code for finite differences[5].

MGfdm6.h

```
#ifndef MGfdm6_h_IS_INCLUDED
#define MGfdm6_h_IS_INCLUDED

#include <MGfdm2.h>

class MGfdm6: public MGfdm2 // see also MGOp3
{
protected:
  VecSimple(real) k_coeff;              // coefficient vector
  virtual void makeMatrix();            // set up lineq Matrix
  virtual void makeMatrix(int i);       // set up smooth matrix
public:
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu);
};
#endif
```

The coefficients of the operator are stored in `k_coeff`. The generation of the matrices in `makeMatrix()` and `makeMatrix(int)` needs to be modified. We also extend the menu handling `define` and `scan`.

MGfdm6.C

```
#include <MGfdm6.h>
#include <MatPtROp_real.h>

void MGfdm6:: define (MenuSystem& menu, int level)
{
  menu.addItem (level,
               "k coeff",    // menu command/name
               "kcoeff",     // command line option: +level
               "star [-b -c -d][-a +x -a][-d -c -b]",
```

[5]you will find the code in **MGfdm6/**

```
                "[1., 0., 1., 0.]",// default answer
                "S");          // valid answer: string
  MGfdm2:: define(menu, level);
}

void MGfdm6:: scan(MenuSystem& menu)
{
  MGfdm2:: scan(menu);

  k_coeff.redim(4);

  Is rIs(menu.get ("k coeff"));
  rIs->ignore ('[');
  for (int i = 1; i <= 4; i++) {
      rIs->get (k_coeff(i));
      if (i < 4)
        rIs->ignore (',');
    }
}

void MGfdm6:: makeMatrix()
{
  int n = gridSize(no_of_grids);

  Handle(MatPtOp(real)) A = new MatPtOp(real);

  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),1);         // with an offset of one from
// the central element A(0,0).
  (*A)(-1, 0) = -k_coeff(1);
  (*A)( 1, 0) = -k_coeff(1);
  (*A)( 1,-1) = -k_coeff(2);
  (*A)(-1, 1) = -k_coeff(2);
  (*A)( 0,-1) = -k_coeff(3);
  (*A)( 0, 1) = -k_coeff(3);
  (*A)(-1,-1) = -k_coeff(4);
  (*A)( 1, 1) = -k_coeff(4);
  (*A)( 0, 0) = 2 * (k_coeff(1) +k_coeff(2) + k_coeff(3) + k_coeff(4));

  FieldFD* b = new FieldFD(grid(no_of_grids)(),"b"); // !!
  b->values() = 0;

  u.rebind (new FieldFD(grid(no_of_grids)(),"u"));
  u->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  real h = 1.0 / real(n);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    b->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions
  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    u->valueIndex(is(1),is(2)) = u0(ps);
  }

  lineq->attach(*A, *u, *b);
}

void MGfdm6:: makeMatrix(int i)
{
  int n = gridSize(i);

  Handle(MatPtOp(real)) A;
  A.rebind( new MatPtOp(real) );

  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),1);         // with an offset of one from
// the central element A(0,0).
  (*A)(-1, 0) = -k_coeff(1);
  (*A)( 1, 0) = -k_coeff(1);
  (*A)( 1,-1) = -k_coeff(2);
  (*A)(-1, 1) = -k_coeff(2);
  (*A)( 0,-1) = -k_coeff(3);
  (*A)( 0, 1) = -k_coeff(3);
  (*A)(-1,-1) = -k_coeff(4);
  (*A)( 1, 1) = -k_coeff(4);
  (*A)( 0, 0) = 2 * (k_coeff(1) +k_coeff(2) + k_coeff(3) + k_coeff(4));

  FieldFD* rhs = new FieldFD(grid(i)(),"rhs"); // !!
  rhs->values() = 0;

  FieldFD* sol = new FieldFD(grid(i)(),"sol"); // !!
  sol->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  real h = 1.0 / real(n);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    rhs->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions

  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    sol->valueIndex(is(1),is(2)) = u0(ps);
  }

  smooth(i)->attach(*A, *sol, *rhs);
  LinEqAdm &s = smooth(i)->linAdm();
  ddsolver->attachLinRhs(s.bl(), i, dpTRUE);
  ddsolver->attachLinSol(s.xl(), i);
}
```

The code reads the four parameters k_1, k_2, k_3, and k_4 and defines the new differences stencils.

The following input parameters may be some guideline for your experiments[6].

We first have a look at anisotropy along coordinate axis, see table 5, input file `test1.i`. The first experiment is about the convergence rate of the conjugate gradient method in the presence of anisotropy. Compare the number of iterations for the different test cases. Is there a difference between the second and the third test case, both an anisotropy of 1 : 10?

menu item	**answer**
k coeff	{[1, 0, 1, 0] & [1, 0, 10, 0] & [1, 0, .1, 0]}
no of grid levels	4
coarse lattice	2
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecNone

Table 5: Conjugated gradients, anisotropy along coordinate axis, `test1.i`

The next two test contain multigrid for the anisotropy along the axis. The first case is the reference computation for the isotropic Laplacian, see table 6, input file `test2.i`.

How is the multigrid performance affected by the presence of the anisotropy? Compare the number of iterations/ the convergence rate. Try some modifications of the smoother. The node ordering in an SOR smoother plays a role. So you can perform experiments changing the direction of the anisotropy and observe the difference in the convergence rate. You can also have a look at large coarse grids, with a low quality coarse grid solver. How is the multigrid iteration affected now?

We can do the same tests with multigrid as a preconditioner, see table 7, input file `test3.i`. Does the additional conjugated gradient algorithm increase the perfor-

[6] files are in `MGfdm6/Verify/`

menu item	**answer**
k coeff	{[1, 0, 1, 0] & [1, 0, 10, 0] & [1, 0, .1, 0]}
no of grid levels	4
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 6: Multigrid, anisotropy along coordinate axis, `test2.i`

mance of the iteration? Is it preferable to use the Krylov iteration in the presence of anisotropy?

We next set of test is concerned with directions of anisotropy, which is not aligned with the coordinate axis, see table 8, input file `test4.i`. We compare one of the previous examples with case witch are rotated by $\pi/4$. In fact we are able to compare two different discretizations for the rotated case.

$$\begin{bmatrix} -.45 & -.1 & \\ -.1 & 1.3 & -.1 \\ & -.1 & -.45 \end{bmatrix}$$

and

$$\begin{bmatrix} & -.9 & .45 \\ -.9 & 2.7 & -.9 \\ .45 & -.9 & \end{bmatrix}$$

There is a substantial difference in the performance of the multigrid method. Both solutions look quite similar. However, if it comes to iterative solvers, one discretization is better suited. Why? (Look at the signs of the entries in the stencil, compare the M-matrix property)

We can redo the test with the rotated anisotropic operator for the conjugated gradient method preconditioned by the multigrid method, see table 9, input file `test5.i`. You can also look at the additive multigrid preconditioner.

4 Different prolongation and restriction stencils

Up to now we have considered a nine-point stencil for prolongation and an adjoint operator for restriction. This is equivalent to the interpolation scheme for bi-linear

menu item	**answer**
k coeff	{[1, 0, 1, 0] & [1, 0, 10, 0] & [1, 0, .1, 0]}
no of grid levels	4
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 7: Conjugated gradients with Multigrid preconditioner, anisotropy along coordinate axis, `test3.i`

finite elements on square shaped domains. However, in finite differences there is no such natural interpolation scheme connected with the discretization. We are free to use different schemes for restriction and prolongation. So we will test some cheaper stencils in the next section, while we have a look at higher order interpolation afterwards.

4.1 Low order prolongation and restriction stencils

We will have a look at several cheap restriction and prolongation stencils from the multigrid literature for finite differences. The standard nine-point stencil reads

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ 1 \\ \frac{1}{2} \end{bmatrix} * \begin{bmatrix} \frac{1}{2} & 1 & \frac{1}{2} \end{bmatrix}$$

We study the seven-point stencil

$$\begin{bmatrix} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & \end{bmatrix}$$

arising in the interpolation of linear triangular shaped elements (in a type 1 triangulation), the five point stencil

$$\begin{bmatrix} & \frac{1}{4} & \\ \frac{1}{4} & 1 & \frac{1}{4} \\ & \frac{1}{4} & \end{bmatrix}$$

menu item	**answer**
k coeff	{[1, 0, .1, 0] & [.1, .45, .1, 0] & [.9, 0, .9, -.45] & [.1, 0, 1, 0] & [.9, -.45, .9, 0] & [.1, 0, .1, .45]}
no of grid levels	4
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 8: Multigrid, anisotropy in different directions, `test4.i`

which is sometimes referred to as "half-weighting" and the trivial injection

$$\begin{bmatrix} & . & \\ . & 1 & . \\ & . & \end{bmatrix}$$

which is only applicable as a restriction.

We derive the test simulator from the standard finite difference multigrid simulator `MGfdm7`[7].

MGfdm7.h

```
#ifndef MGfdm7_h_IS_INCLUDED
#define MGfdm7_h_IS_INCLUDED

#include <MGfdm2.h>

class MGfdm7: public MGfdm2
{
protected:
  virtual void initProj();              // setup proj
public:
  virtual void define (MenuSystem& menu, int level = MAIN);
};
#endif
```

[7]you will find the code in `MGfdm7/`

menu item	answer
k coeff	{[1, 0, .1, 0] & [.1, .45, .1, 0] & [.9, 0, .9, -.45] & [.1, 0, 1, 0] & [.9, -.45, .9, 0] & [.1, 0, .1, .45]}
no of grid levels	4
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 9: Conjugated gradients with Multigrid preconditioner, anisotropy in different directions, `test5.i`

The menu handling procedure `define` is extended to read the restriction and the prolongation stencil. The function `initProj` reads these descriptions of the stencils and defines the projectors for restriction `proj_r` and for prolongation `proj_p` accordingly. The transfer in the nested iteration `proj_nest` is always set to the bi-linear interpolation scheme.

MGfdm7.C

```
#include <MGfdm7.h>
#include <MatPtROp_real.h>

void MGfdm7:: define (MenuSystem& menu, int level)
{
  menu.addItem (level,
              "prolongation",    // menu command/name
              "prolongation",     // command line option: +level
              "star [a b c][d e f][g h i]",
              "[.25, .5, .25][.5, 1, .5][.25, .5, .25]",// default answer
              "S");          // valid answer: string
  menu.addItem (level,
              "restriction",    // menu command/name
              "restriction",     // command line option: +level
              "star [a b c][d e f][g h i]",
              "[.25, .5, .25][.5, 1, .5][.25, .5, .25]",// default answer
              "S");          // valid answer: string
  MGfdm2:: define(menu, level);
}

void MGfdm7:: initProj() // setup proj operators
{
  int i, j, k;
  MatSimple(real) pro(3,3);
  Is pIs(menu_system->get ("prolongation"));
  for (i = 1; i <= 3; i++) {
    pIs->ignore ('[');
    for (j = 1; j <= 3; j++) {
      pIs->get (pro(i,j));
      if (j<3)
        pIs->ignore (',');
    }
  }

  MatSimple(real) res(3,3);
  Is rIs(menu_system->get ("restriction"));
  for (i = 1; i <= 3; i++) {
    rIs->ignore ('[');
    for (j = 1; j <= 3; j++) {
      rIs->get (res(i,j));
      if (j<3)
        rIs->ignore (',');
    }
  }

  for (i=1; i<no_of_grids; i++) {
    int n = gridSize(i);
    int m = smooth(i+1)->linAdm().getLinEqSystem (). A().mat().getNoRows();
    Handle(IndexSet)       ind1;        // 'interior' index set fine
    defineIndexSetI (ind1, 2 * n, 2);

    Handle(IndexSet)       ind2;        // 'interior' index set coarse
    defineIndexSetI (ind2, n, 1);

    Handle(MatPtROp(real)) M;

    // restriction

    M.rebind(new MatPtROp(real));
    M->redim(ind1(), ind2(), 1);
    M->setNoRows(m);

    for (j=1; j<=3; j++)
      for (k=1; k<=3; k++)
        (*M)(j-2, k-2) = res(j,k);

    M->optimize();
    Handle(LinEqMatrix) MM;
    MM.rebind(new LinEqMatrix(*M));

    proj_r(i).rebind(new ProjMatrix());
    proj_r(i)->rebindMatrix(*MM);
    proj_r(i)->init();

    // prolongation
    M.rebind(new MatPtROp(real));
    M->redim(ind1(), ind2(), 1);
    M->setNoRows(m);

    for (j=1; j<=3; j++)
      for (k=1; k<=3; k++)
        (*M)(j-2, k-2) = pro(j,k);

    M->optimize();
    MM.rebind(new LinEqMatrix(*M));

    proj_p(i).rebind(new ProjMatrix());
    proj_p(i)->rebindMatrix(*MM);
    proj_p(i)->init();

    // nested

    M.rebind(new MatPtROp(real));
    M->redim(ind1(), ind2(), 1);
    M->setNoRows(m);

    (*M)(-1,-1) = .25;  (*M)(-1, 0) = .5;   (*M)(-1, 1) = .25;
    (*M)( 0,-1) = .5;   (*M)( 0, 0) = 1.;   (*M)( 0, 1) = .5;
    (*M)( 1,-1) = .25;  (*M)( 1, 0) = .5;   (*M)( 1, 1) = .25;

    M->optimize();
    MM.rebind(new LinEqMatrix(*M));

    proj_nest(i).rebind(new ProjMatrix());
    proj_nest(i)->rebindMatrix(*MM);
    proj_nest(i)->init();
  }
}
```

The following input parameters may be some guideline for your experiments[8].

For reference we include the standard input file for the un-preconditioned conjugated gradient method, see table 10, input file `test1.i`. Changes in the restriction and prolongation do not affect this procedure.

menu item	**answer**
no of grid levels	4
coarse lattice	2
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecNone
#1: convergence monitor name	CMAbsTrueResidual
#1: max error	1.0e-7

Table 10: Conjugated gradients on a finite difference discretization, `test1.i`

We test the multigrid method with the given restriction and prolongation stencils, see

[8]files are in **`MGfdm7/Verify/`**

table 11, input file `test2.i`. The code executes all combinations of the parameters. This means that there are several cases, where the restriction and the prolongation are not adjoint. Hence we have an unsymmetric iterative solver.

We can compare the performance of the multigrid method for the different stencils. It is easy to compare the number of iterations and the convergence rates. It is more difficult to actually observe a difference in the execution time for one multigrid cycle. Although we use optimized stencils, that is the removal of zeros in the stencil, and the application of a five point stencil requires less (floating point) operations than a seven point stencil in `Diffpack`, this effect is small compared to the overall execution time. However, the effect of the convergence rate is definitely visible.

The "half-weighting" stencil and the trivial injection stencil have been introduced in specifically tuned versions of the multigrid method. This is usually tied to a certain choice for the smoothing algorithm. To be able to repeat these methods exactly, specific choice have to made. This e.g. requires a red-black Gauss-Seidel iteration, which can be implemented with some modifications of the iterators `defineIndexSetI()`. See also the red-black Gauss-Seidel in the finite element multigrid tutorial [Zum96c].

menu item	**answer**
prolongation	{[.25, .5, .25][.5, 1, .5][.25, .5, .25] & [0, .5, .5][.5, 1, .5][.5, .5, 0] & [0, .25, 0][.25, 1, .25][0, .25, 0]}
restriction	{[.25, .5, .25][.5, 1, .5][.25, .5, .25] & [0, .5, .5][.5, 1, .5][.5, .5, 0] & [0, .25, 0][.25, 1, .25][0, .25, 0] & [0, 0, 0][0, 1, 0][0, 0, 0]}
no of grid levels	4
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 11: Multigrid with different restriction and prolongation stencils, `test2.i`

We can redo the same test for the multigrid preconditioner, see table 12, input file `test3.i`. Since some of the multigrid versions, in the case the restriction and prolongation are not adjoint, are in fact unsymmetric operations, you might want to use an unsymmetric Krylov iteration like `BiCGstab` or `CGS` instead of the conjugated gradient iteration. Does the surrounding Krylov iteration improve the multigrid performance, especially for very cheap restriction or prolongation operators?

menu item	**answer**
prolongation	{[.25, .5, .25][.5, 1, .5][.25, .5, .25] & [0, .5, .5][.5, 1, .5][.5, .5, 0] & [0, .25, 0][.25, 1, .25][0, .25, 0]}
restriction	{[.25, .5, .25][.5, 1, .5][.25, .5, .25] & [0, .5, .5][.5, 1, .5][.5, .5, 0] & [0, .25, 0][.25, 1, .25][0, .25, 0] & [0, 0, 0][0, 1, 0][0, 0, 0]}
no of grid levels	4
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 12: Conjugated gradients with Multigrid preconditioner using different restriction and prolongation stencils, `test3.i`

4.2 High order prolongation

The motivation for low order restriction and prolongation operators was the cost. Less entries in the operator stencil mean less number of operations. This is especially true, if values computed by the prolongation are overwritten immediately by the smoothing algorithm, like in some red-black Gauss-Seidel iterations.

So the application of higher order transfer operators here is rather un-usual. There is another occasion, where an interpolation scheme is needed. In the nested iteration the solution on one grid has to be interpolated to the next finer grid. This interpolant serves as an initial guess for the following multigrid iteration.

In order to use the all the information, which is contained in this vector, a higher oder interpolation scheme may be appropriate. Since this interpolation is only performed once per grid in the complete multigrid solution procedure, one can afford a more expensive scheme here. This procedure of course will only pay off in regions, where the solution to be interpolated is regular enough.

We now look at the implementation of a bi-cubic interpolation on the square.

$$\begin{bmatrix} -\frac{1}{16} \\ 0 \\ \frac{9}{16} \\ 1 \\ \frac{9}{16} \\ 0 \\ -\frac{1}{16} \end{bmatrix} * \begin{bmatrix} -\frac{1}{16} & 0 & \frac{9}{16} & 1 & \frac{9}{16} & 0 & -\frac{1}{16} \end{bmatrix}$$

Since this requires a larger support of the stencil, which is not available near the boundary, we have to modify the scheme there. Near the boundary we use an unsymmetric quadratic interpolation scheme instead. Near to the left boundary we use

$$\begin{bmatrix} 0 & 0 & \frac{3}{8} & 1 & \frac{3}{4} & 0 & -\frac{1}{8} \end{bmatrix}$$

and near to the right boundary

$$\begin{bmatrix} -\frac{1}{8} & 0 & \frac{3}{4} & 1 & \frac{3}{8} & 0 & 0 \end{bmatrix}$$

If even this scheme is not applicable, we use our standard linear interpolation scheme.

$$\begin{bmatrix} \frac{1}{2} & 1 & \frac{1}{2} \end{bmatrix}$$

In general we use the highest order possible and use a tensor product approach to construct the formula.[9].

MGfdm11.h

```
#ifndef MGfdm11_h_IS_INCLUDED
#define MGfdm11_h_IS_INCLUDED

#include <MGfdm2.h>

class MGfdm11: public MGfdm2
{
protected:
  virtual void initProj();              // setup proj
  virtual void makeProj(int i);         // create proj i,i+1
  virtual void makeProj_nest(int i);    // create nested proj i,i+1
  virtual int noPtop(int n);            // number of higher order stencils on a grid
  virtual BooLean defineIndexSetIc(Handle(IndexSet)& indi,
   int n, int x, int y, int step); // higher order
  virtual BooLean defineIndexSetIl(Handle(IndexSet)& indi,
   int n, int step); // linear
public:
};
#endif
```

[9]you will find the code in `MGfdm11/`

We derive our simulator from the standard multigrid code `MGfdm2`. In order to implement these interpolation schemes, we have to modify the procedures, which handle the transfer operators: `makeProj(int)` is split into the part for standard restriction and prolongation operators and `makeProj_nest(int)` defines the new nested interpolation schemes. `initProj()` now calls both initialization procedures. We define new iterators for all nodes, which have to be interpolated linearly `defineIndexSetl` and nodes that are interpolated cubic or quadratic `defineIndexSetc` in x and in y direction, depending on the flags `x` and `y`.

MGfdm11.C

```
#include <MGfdm11.h>
#include <MatPtROp_real.h>

void MGfdm11:: makeProj(int i)
{
  int n = gridSize(i);
  int m = smooth(i+1)->linAdm().getLinEqSystem (). A().mat().getNoRows();
  Handle(IndexSet)       ind1;        // 'interior' index set fine
  defineIndexSetI (ind1, 2 * n, 2);

  Handle(IndexSet)       ind2;        // 'interior' index set coarse
  defineIndexSetI (ind2, n, 1);

  Handle(MatPtROp(real)) M;

  M.rebind(new MatPtROp(real));
  M->redim(ind1(), ind2(), 1);
  M->setNoRows(m);

  (*M)(-1,-1) = .25;  (*M)(-1, 0) = .5;   (*M)(-1, 1) = .25;
  (*M)( 0,-1) = .5;   (*M)( 0, 0) = 1.;   (*M)( 0, 1) = .5;
  (*M)( 1,-1) = .25;  (*M)( 1, 0) = .5;   (*M)( 1, 1) = .25;

  M->optimize();
  Handle(LinEqMatrix) MM;
  MM.rebind(new LinEqMatrix(*M));

  // restriction

  proj_r(i).rebind(new ProjMatrix());
  proj_r(i)->rebindMatrix(*MM);
  proj_r(i)->init();

  // prolongation

  proj_p(i).rebind(new ProjMatrix());
  proj_p(i)->rebindMatrix(*MM);
  proj_p(i)->init();
}

int MGfdm11:: noPtop(int n)
{
  if (n < 4) return 1;
  return 9;
}
```

```
BooLean MGfdm11:: defineIndexSetIc
(Handle(IndexSet)& indi, int n, int x, int y, int step)
{  // higher order interpolation points (formula x,y)
  if (n < 4 * step) return dpFALSE;

  BoxIndices* interior = new BoxIndices;
  indi.rebind (interior);

  Ptv(int) steps(2);
  steps = step;

  int x0 = 2*step;
  int x1 = n-2*step;
  if (x<0) x1 = x0 = step;   // left layer
  if (x>0) x0 = x1 = n-step; // right layer

  int y0 = 2*step;
  int y1 = n-2*step;
  if (y<0) y1 = y0 = step;
  if (y>0) y0 = y1 = n-step;

  interior->scan(aform("2(%d,%d)  (%d,%d)", x0, y0, x1, y1));
  interior->setSteps(steps);
  return dpTRUE;
}

BooLean MGfdm11:: defineIndexSetIl
(Handle(IndexSet)& indi, int n, int step)
{  // linear interpolation points
  if (n >= 4 * step) return dpFALSE;

  BoxIndices* interior = new BoxIndices;
  indi.rebind (interior);

  Ptv(int) steps(2);
  steps = step;

  interior->scan(aform("2(%d,%d)  (%d,%d)", step, step, n-step, n-step));
  interior->setSteps(steps);
  return dpTRUE;
}

void MGfdm11:: makeProj_nest(int i)
{
  const int offset = 3;
  static const real inter[][1+2*offset] = {
    {0    ,  0, .375,  1, .75,   0, -.125}, // quadratic (left)
    {-.0625, 0, .5625, 1, .5625, 0, -.0625},// cubic
    {-.125,  0, .75,   1, .375,  0, 0}      // quadratic (right)
    // {0,      0, .5,    1, .5,    0, 0},     // linear
  };
  int n = gridSize(i);
  int m = smooth(i+1)->linAdm().getLinEqSystem (). A().mat().getNoRows();

  Handle(MatPtROp(real)) M;
  M.rebind(new MatPtROp(real));
  M->init(noPtop(n)); // number different stencils

  int ptop = 1;
  // cubic (quadratic) interpolation along an axis, as high as possible
  for (int x=-1; x<=1; x++)
    for (int y=-1; y<=1; y++) {
      Handle(IndexSet)      ind1;       // 'interior' index set fine
      if (defineIndexSetIc (ind1, 2 * n, x, y, 2)) {

Handle(IndexSet)      ind2;       // 'interior' index set coarse
defineIndexSetIc (ind2, n, x, y, 1);

Handle(PtROpDS(real)) P;
P.rebind(new PtROpDS(real));
P->redim(ind1(), ind2(), offset);

for (int l=-offset; l<=offset; l++)
  for (int m=-offset; m<=offset; m++)
    (*P)(l, m) = inter[1+x][l+offset] * inter[1+y][m+offset];

M->attach(*P, ptop++);
      }
    }

  // (bi-)linear interpolation, where necessary
  Handle(IndexSet)      ind1;       // 'interior' index set fine
  if (defineIndexSetIl (ind1, 2 * n, 2)) {

    Handle(IndexSet)      ind2;       // 'interior' index set coarse
    defineIndexSetIl (ind2, n, 1);

    Handle(PtROpDS(real)) P;
    P.rebind(new PtROpDS(real));
    P->redim(ind1(), ind2(), 1);

    (*P)(-1,-1) = .25;  (*P)(-1, 0) = .5;   (*P)(-1, 1) = .25;
    (*P)( 0,-1) = .5;   (*P)( 0, 0) = 1.;   (*P)( 0, 1) = .5;
    (*P)( 1,-1) = .25;  (*P)( 1, 0) = .5;   (*P)( 1, 1) = .25;

    M->attach(*P, ptop++);
  }
  if (noPtop(n)!= ptop-1) warningFP("MGfdm11:: makeProj_nest",
    "noPtop not ok");

  M->optimize();
  M->setNoRows(m);
  Handle(LinEqMatrix) MM;
  MM.rebind(new LinEqMatrix(*M));

  s_o<<"nested interpolation, level "<<i<<endl; M->print(s_o);

  proj_nest(i).rebind(new ProjMatrix());
  proj_nest(i)->rebindMatrix(*MM);
  proj_nest(i)->init();
}

void MGfdm11:: initProj() // setup proj operators
{
  for (int i=1; i<no_of_grids; i++) {
    makeProj(i);
    makeProj_nest(i);
  }
```

```
}
```

The procedure `defineIndexSetIc` defines an iterator for all interior domain nodes applicable for quadratic or cubic interpolation. The parameters `x` and `y` indicate the exact interpolation method used, along the x and the y axis. The values -1 and 1 are used for quadratic interpolation on the left and the right layer, while a value 0) indicates cubic interpolation. So we define nine different types of nodes. The tenth type contains the remaining nodes, for which bi-linear interpolation is used (procedure `defineIndexSetIl`). The number of non-empty sets of node types is computed by `noPtop(int)`. Each `defineIndexSet` returns false in the case of an empty node iterator, since this case has to be treated differently form the non-empty iterators.

The following input parameters may be some guideline for your experiments[10].

We run a nested multigrid iteration, see table 13, input file `test2.i`. The idea is to compare the number of iterations with the nested iteration multigrid with bi-linear nested interpolation of `MGfdm2`. Since the difference is small, you may want to test different smoothers and number of smoothing steps to study the quality of the higher order interpolation.

menu item	**answer**
no of grid levels	4
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	NestedMultigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 13: Multigrid on a finite difference discretization, `test2.i`

5 Convection-diffusion equation

We are now studying the scalar linear convection-diffusion equation as a prototype for an unsymmetric equation and a model for transport equations.

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

[10]files are in `MGfdm11/Verify/`

Since we already know how to treat the diffusion, the convection term $\vec{v} \cdot \nabla u$ introduces some difficulties. In the case of large convection, a straightforward second order discretization with central differences for the convection[11].

$$\frac{v_1}{2h}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} + \frac{v_2}{2h}\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

is unstable. So we have to modify the discretization, either add some stabilization term, which we do in the next section, or use a different (lower order) difference stencil for convection term, which we do in a later section. This choice of the discretization of course affects the multigrid solution algorithm.

5.1 Artificial viscosity

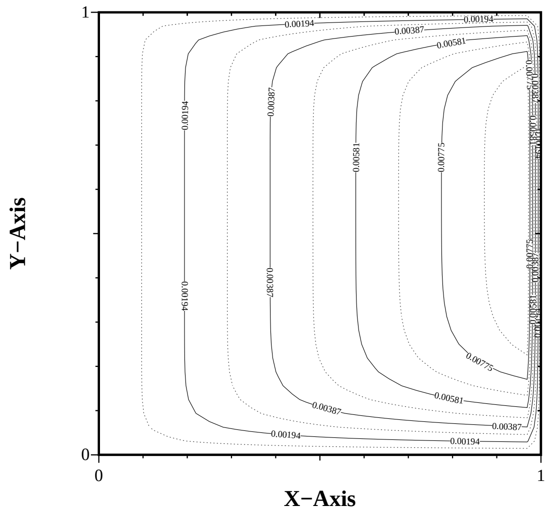

Figure 4: Solution of a convection diffusion problem with artificial viscosity.

The instability of the central difference discretization can be observed, if one of the off-central entries of the difference stencil changes sign and becomes positive

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} + \frac{v_1}{2h}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} + \frac{v_2}{2h}\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

This is caused, if the first order convection term starts to dominate over the second order diffusion term. One way to enforce stability in such a case is to increase the

[11] In this chapter it is important to denote the $1/h$ and $1/h^2$ terms in the first and second order differential operators. In the implementation however, all terms are multiplied by h^2 like in the previous chapters.

diffusion. This is needed, if

$$|v| > \frac{2}{h}$$

which is very likely for coarse grid discretizations. In such a case we introduce a diffusion of

$$k = h|v|$$

instead of the original diffusion of 1. There are some refinement for this scheme: It is enough to enforce stability along each coordinate axis for this difference stencil. So we can introduce an anisotropic diffusion operator with just enough diffusion to stabilize the equation in all directions. Another refinement for the application of multigrid may be to introduce a C^1 dependence of the artificial diffusion term form the mesh size h. In the standard approach, the artificial diffusion is just switched on, when the mesh-size reaches the threshold of $\frac{2}{|v|}$.

Our sample simulator is derived from the standard `Diffpack` multigrid simulator for finite differences. We introduce a new parameter vector `velocity`, which is initialized by the menu handling procedures `define` and `scan`. The procedures `makeMatrix()` and `makeMatrix(int)` implement an anisotropic artificial viscosity discretization[12].

MGfdm8.h

```
#ifndef MGfdm8_h_IS_INCLUDED
#define MGfdm8_h_IS_INCLUDED

#include <MGfdm2.h>

class MGfdm8: public MGfdm2 // see also MGOp1, artificial viscosity
{
protected:
  Ptv(real) velocity;
  virtual void makeMatrix();             // set up lineq Matrix
  virtual void makeMatrix(int i);        // set up smooth matrix
public:
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu);
};
#endif
```

The main change in the `makeMatrix` procedures is the introduction of the convection term with an artificial viscosity stabilization.

MGfdm8.C

```
#include <MGfdm8.h>
```

[12]you will find the code in `MGfdm8/`

```
#include <MatPtROp_real.h>

void MGfdm8:: define (MenuSystem& menu, int level)
{
  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
  MGfdm2:: define(menu, level);
}

void MGfdm8:: scan (MenuSystem& menu)
{
  MGfdm2::scan(menu);

  velocity.redim(2);
  Is rIs(menu.get ("velocity"));
  rIs->ignore ('[');
  for (int i = 1; i <= 2; i++) {
    rIs->get (velocity(i));
    if (i < 2)
      rIs->ignore (',');
  }
}

void MGfdm8:: makeMatrix()
{
  int n = gridSize(no_of_grids);

  Handle(MatPtOp(real)) A = new MatPtOp(real);

  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),1);         // with an offset of one from
// the central element A(0,0).
  real h = 1.0 / real(n);

  real cx = h * .5 * fabs(velocity(1));   // artificial viscosity x
  if (fabs(cx) < 1) cx = 1;

  real cy = h * .5 * fabs(velocity(2));   // artificial viscosity y
  if (fabs(cy) < 1) cy = 1;

            (*A)( 0,-1) = -cy - h * .5 * velocity(2);
  (*A)(-1, 0) = -cx - h * .5 * velocity(1);
            (*A)( 0, 0) =  2 * (cx + cy);
                        (*A)( 1, 0) = -cx + h * .5 * velocity(1);
            (*A)( 0, 1) = -cy + h * .5 * velocity(2);

  FieldFD* b = new FieldFD(grid(no_of_grids)(),"b");
  b->values() = 0;

  u.rebind (new FieldFD(grid(no_of_grids)(),"u"));
  u->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    b->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions
  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    u->valueIndex(is(1),is(2)) = u0(ps);
  }

  lineq->attach(*A, *u, *b);
}

void MGfdm8:: makeMatrix(int i)
{
  int n = gridSize(i);

  Handle(MatPtOp(real)) A;
  A.rebind( new MatPtOp(real) );

  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),1);         // with an offset of one from
// the central element A(0,0).
  real h = 1.0 / real(n);

  real cx = h * .5 * fabs(velocity(1));   // artificial viscosity x
  if (fabs(cx) < 1) cx = 1;

  real cy = h * .5 * fabs(velocity(2));   // artificial viscosity y
  if (fabs(cy) < 1) cy = 1;

              (*A)( 0,-1) = -cy - h * .5 * velocity(2);
  (*A)(-1, 0) = -cx - h * .5 * velocity(1);
              (*A)( 0, 0) =  2 * (cx + cy);
                          (*A)( 1, 0) = -cx + h * .5 * velocity(1);
              (*A)( 0, 1) = -cy + h * .5 * velocity(2);

  FieldFD* rhs = new FieldFD(grid(i)(),"rhs");
  rhs->values() = 0;

  FieldFD* sol = new FieldFD(grid(i)(),"sol");
  sol->values() = 0;
  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    rhs->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions

  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    sol->valueIndex(is(1),is(2)) = u0(ps);
  }

  smooth(i)->attach(*A, *sol, *rhs);
  LinEqAdm &s = smooth(i)->linAdm();
  ddsolver->attachLinRhs(s.bl(), i, dpTRUE);
  ddsolver->attachLinSol(s.xl(), i);
}
```

The following input parameters may be some guideline for your experiments[13].

The first test is the solution of the convection-diffusion problem with a Krylov iteration, see table 14, input file `test1.i`. We use the `BiCGStab` method, because the equation system is unsymmetric. One possible study is the dependence of the convergence rate on the velocity v. A velocity zero means a symmetric Laplace type problem. A large velocity probably introduces trouble for the iterative solver.

menu item	**answer**
velocity	[100,10]
no of grid levels	4
coarse lattice	2
matrix type	MatPtOp
basic method	BiCGStab
preconditioning type	PrecNone

Table 14: Conjugated gradients for an artificial viscosity discretization, `test1.i`

The next test is the multigrid method applied to the same problem, see table 15, input file `test2.i`. We can compare the number of iterations and the convergence rate for different values of the velocity. It might also be interesting to study effects of the smoothing algorithm and the number of smoothing steps on the convergence.

[13] files are in `MGfdm8/Verify/`

For large coarse grids the influence of the (inexact) coarse grid solver can be studied. Since the quality of the discretization of the coarse grid problem for convection-diffusion problems is often very poor, the overall performance now depends on the quality of the smoothers. So studies of smoother, which are exact for pure transport equations as some upstream Gauss-Seidel or ILU schemes, are of special interest.

menu item	**answer**
velocity	[100,10]
no of grid levels	4
coarse lattice	2
sweeps	[2,2]
matrix type	MatPtOp
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 15: Multigrid for an artificial viscosity discretization, `test2.i`

The last test for the artificial viscosity discretization is the preconditioned `BiCGStab` iteration. Now also the non-symmetric Krylov iteration improves the transport of the solution- and error-components, see table 16, input file `test3.i`. In a comparison of the number of iterations with the ordinary multigrid method, you have to take into account, that the `BiCGStab` iteration requires more work per iteration than just a standard conjugated gradient iteration.

5.2 Upwind discretization

The alternative to the artificial viscosity discretization (adding diffusion) is the modification of the difference stencil for the first order convection term. The idea is to switch to a first order one-sided difference stencil.

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} + \frac{v_1}{h}\begin{bmatrix} -1 & 1 & 0 \end{bmatrix} + \frac{v_2}{h}\begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}$$

for positive v_1, v_2 and similar one-sided (shifted) stencils for negative v components, e.g. for $v_1 < 0$

$$-\frac{v_1}{h}\begin{bmatrix} 0 & 1 & -1 \end{bmatrix}$$

The stability is maintained and no operator has to be changed. However, this discretization is only first order in contrast to the second order central difference scheme above. Some refinement of the upwind scheme is to modify the discretization in a

menu item	**answer**
velocity	[100,10]
no of grid levels	4
coarse lattice	2
sweeps	[2,2]
matrix type	MatPtOp
basic method	BiCGStab
preconditioning type	PrecDD
domain decomposition method	Multigrid
cycle type gamma	1
smoother matrix type	MatPtOp
smoother basic method	SOR
coarse grid matrix type	MatPtOp
coarse grid basic method	SOR
coarse grid max iterations	1

Table 16: Conjugated gradients with Multigrid preconditioner for an artificial viscosity discretization, `test3.i`

$C^1(h)$ way to change into the second order central difference scheme for small h in order to increase precision.

We implement the upwind scheme in a simulator derived from the previous convection-diffusion simulator[14].

MGfdm9.h

```
#ifndef MGfdm9_h_IS_INCLUDED
#define MGfdm9_h_IS_INCLUDED

#include <MGfdm8.h>

class MGfdm9: public MGfdm8 // see also MGOp1, upwind
{
protected:
  virtual void makeMatrix();             // set up lineq Matrix
  virtual void makeMatrix(int i);        // set up smooth matrix
};
#endif
```

We have to modify the implementation of the discretization of the convection term in the procedures `makeMatrix`.

MGfdm9.C

```
#include <MGfdm9.h>
```

[14]you will find the code in `MGfdm9/`

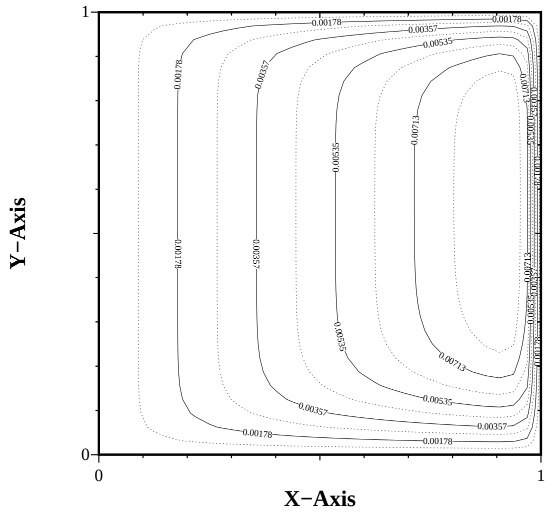

Figure 5: Solution of a convection diffusion problem with an upwind scheme.

```
#include <MatPtROp_real.h>
void MGfdm9:: makeMatrix()
{
  int n = gridSize(no_of_grids);

  Handle(MatPtOp(real)) A = new MatPtOp(real);

  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),1);         // with an offset of one from
// the central element A(0,0).
  real h = 1.0 / real(n);

  real c = fabs(velocity(1)) + fabs(velocity(2)); // upwind

                    (*A)( 0,-1) = -1;
  (*A)(-1, 0) = -1;  (*A)( 0, 0) =  4 + c * h;  (*A)( 1, 0) = -1;
                    (*A)( 0, 1) = -1;

  if (velocity(1)>0)
    (*A)(-1, 0) = -1 - h * velocity(1);
  else
    (*A)( 1, 0) = -1 + h * velocity(1);

  if (velocity(2)>0)
    (*A)( 0,-1) = -1 - h * velocity(2);
  else
    (*A)( 0, 1) = -1 + h * velocity(2);
  FieldFD* b = new FieldFD(grid(no_of_grids)(),"b");
  b->values() = 0;

  u.rebind (new FieldFD(grid(no_of_grids)(),"u"));
  u->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    b->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions
  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    u->valueIndex(is(1),is(2)) = u0(ps);
  }

  lineq->attach(*A, *u, *b);
}

void MGfdm9:: makeMatrix(int i)
{
  int n = gridSize(i);

  Handle(MatPtOp(real)) A;
  A.rebind( new MatPtOp(real) );

  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),1);         // with an offset of one from
// the central element A(0,0).
  real h = 1.0 / real(n);

  real c = fabs(velocity(1)) + fabs(velocity(2)); // upwind
                   (*A)( 0,-1) = -1;
  (*A)(-1, 0) = -1;  (*A)( 0, 0) =  4 + c * h;  (*A)( 1, 0) = -1;
                   (*A)( 0, 1) = -1;

  if (velocity(1)>0)
    (*A)(-1, 0) = -1 - h * velocity(1);
  else
    (*A)( 1, 0) = -1 + h * velocity(1);

  if (velocity(2)>0)
    (*A)( 0,-1) = -1 - h * velocity(2);
  else
```

```
    (*A)( 0, 1) = -1 + h * velocity(2);

  FieldFD* rhs = new FieldFD(grid(i)(),"rhs");
  rhs->values() = 0;

  FieldFD* sol = new FieldFD(grid(i)(),"sol");
  sol->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    rhs->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions

  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    sol->valueIndex(is(1),is(2)) = u0(ps);
  }

  smooth(i)->attach(*A, *sol, *rhs);
  LinEqAdm &s = smooth(i)->linAdm();
  ddsolver->attachLinRhs(s.bl(), i, dpTRUE);
  ddsolver->attachLinSol(s.xl(), i);
}
```

We can redo the experiments of the last section on artificial viscosity and compare the performance of the iterative solvers for the different types of discretization[15].

The first test is the un-preconditioned Krylov iteration for the solution of the unsymmetric equation system, see table 14, input file `test1.i`. The next test is the application of a multigrid method, see table 15, input file `test2.i`. Here some studies of the performance of different smoother are interesting. Last is the multigrid preconditioned Krylov iteration, see table 16, input file `test3.i`, with a comparison to the ordinary multigrid iteration.

5.3 Operator dependent transfers and Galerkin products

We discuss a third strategy to cope with the stability problem in convection-diffusion equations. However, this more algebraic strategy has a wider scope than just the convection-diffusion equation. Assume we have a stable discretization of the equation on the finest grid. It is likely that similar discretizations on coarser levels will become

[15] files are in **`MGfdm9/Verify/`**

unstable. So we generate these discretization in a different, an algebraic way. The idea is to use the property of a *Galerkin* discretization

Galerkin

$$A_j = r\, A_{j+1}\, p$$

with restriction r, prolongation p, a fine grid stiffness matrix A_{j+1} and a coarse grid matrix A_j. Given a prolongation operator as the standard interpolation and defining the restriction as the adjoint $r = p^*$, we are able to compute A_j without the need for a discretization procedure. If we start with a discretization A_{j+1} that is equivalent to a finite element discretization of an equation, and we use the associated interpolation scheme p, the resulting coarse grid discretization A_j is equivalent to the associated coarse grid finite element discretization. However, we are free to modify both discretization and interpolation scheme.

algebraic multigrid

We apply this scheme step by step to construct the sequence of matrices, starting with the given fine grid stiffness matrix. This procedure is sometimes referred to as algebraic multigrid, although a pure algebraic multigrid procedure additionally requires an heuristic procedure to construct a suitable prolongation p for a given matrix (-graph) A_{j+1}.

However, there is one drawback of this approach so far: We are starting with a stable fine grid discretization for the convection-diffusion equation, for example with an upwind discretization A_l

$$\frac{v_1}{h}\begin{bmatrix} -1 & 1 & 0 \end{bmatrix}$$

and we iterate the coarsening scheme $A_j = p^* A_{j+1} p$ several times to compute A_1. If we are looking at the limit case $A_\infty \rightarrow A_1$, we observe that the discretization on the coarsest level A_1 now is a central difference scheme

$$\frac{v_1}{2h}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

which is unstable and which we wanted to avoid. The idea now is to modify the restriction/ prolongation in order to maintain stable discretizations. We construct a prolongation scheme that depends on the differential operator or, more precise, on a given discretization. In one dimension, the choice is fairly straightforward: If we denote the difference stencil by

operator dependent transfer

$$A_{j+1} = \begin{bmatrix} a_{-1} & a_0 & a_1 \end{bmatrix}$$

then we can define the prolongation stencil by

$$p = r^* = \begin{bmatrix} -\frac{a_1}{a_0} & 1 & -\frac{a_{-1}}{a_0} \end{bmatrix}$$

The idea is to preserve both a one-sided difference as well as a central difference operator and to look at the expression $A_{j+1}p$.

The generalization of this operator dependent prolongation scheme into two and three dimensions is not clear or unique. We follow one possible line of generalization. We denote the difference stencil in two dimensions by

$$A_{j+1} = \begin{bmatrix} a_{-1,1} & a_{0,1} & a_{1,1} \\ a_{-1,0} & a_{0,0} & a_{1,0} \\ a_{-1,-1} & a_{0,-1} & a_{1,-1} \end{bmatrix}$$

and define the prolongation by

$$p \;=\; r^* \;=\; \begin{bmatrix} p_{-1,1} & p_{0,1} & p_{1,1} \\ p_{-1,0} & 1 & a_{1,0} \\ p_{-1,-1} & p_{0,-1} & p_{1,-1} \end{bmatrix}$$

In analogy to the one dimensional case we define average difference stencils along a coordinate axis and use the one-dimensional formulas. We define the entries

$$\begin{aligned} p_{\pm 1,0} &= -\textstyle\sum_{j=-1,0,1} a_{\mp 1,j} / \sum_{j=-1,0,1} a_{0,j} \\ p_{0,\pm 1} &= -\textstyle\sum_{j=-1,0,1} a_{j,\mp 1} / \sum_{j=-1,0,1} a_{j,0} \end{aligned}$$

The entries $p_{\pm 1,\pm 1}$ are still missing. We choose

$$p_{\pm 1,\pm 1} \;=\; p_{\pm 1,0}\, p_{0,\pm 1}$$

to mimic the bi-linear nine-point interpolation stencil. For alternative choices we refer to the literature [Hac85, Wes92].

The implementation of the operator dependent transfers and the Galerkin product discretization is based on the finite differences multigrid code for the convection-diffusion equation with an upwind discretization. Of course it is also possible to start with the artificial viscosity discretization from the last section. However, we have to start with a stable discretization of the convection-diffusion equation.[16].

We have to modify the initialization procedure used in the multigrid codes. We split it into a management procedure `initProjMatrices`, which calls the Galerkin products `makeMatrixG(int)` and transfer operator initialization `makeProj(int)` in the right order. It is not possible to initialize differential operators and transfer operators separately.

MGfdm10.h

```
#ifndef MGfdm10_h_IS_INCLUDED
#define MGfdm10_h_IS_INCLUDED

#include <MGfdm8.h>

class MGfdm10: public MGfdm8
{
protected:
  virtual void initProjMatrices();       // setup proj and matrices
  virtual void makeProj(int i);          // set up projection i+1,i
  virtual void makeMatrixG(int i);       // set up smooth matrix i
public:
  virtual void solveProblem ();
};
#endif
```

[16]you will find the code in `MGfdm10/`

The procedure `makeProj(int i)` implements the operator dependent transfer defined earlier. Based on the operator `i+1` grabbed from `smooth(i+1)` the transfer operator on level `i` is constructed. All projection operators are initialized that way.

The procedure `makeMatrixG(int i)` implements the Galerkin products $A_i = p^* A_{i+1} p$ based on the operator in `smooth(i+1)` and the prolongation in `proj_p(i)`.

The management procedure `initProjMatrices` calls the standard discretization procedure `makeMatrix` on the finest grid and subsequently the operator dependent transfers and Galerkin product procedures until the coarsest level is reached.

MGfdm10.C

```
#include <MGfdm10.h>
#include <MatPtROp_real.h>
#include <DDIter.h>
#include <PrecDD.h>

void MGfdm10:: makeProj(int i) // operator dependent transfer
{
  int n = gridSize(i);
  Handle(IndexSet)      ind1;       // 'interior' index set fine
  defineIndexSetI (ind1, 2 * n, 2);

  Handle(IndexSet)      ind2;       // 'interior' index set coarse
  defineIndexSetI (ind2, n, 1);

  MatPtOp(real)& A = CAST_REF(smooth(i+1)->linAdm().getLinEqSystem ().
      A().mat(), MatPtOp(real));
  Handle(MatPtROp(real)) M;
  M.rebind(new MatPtROp(real));

  M->redim(ind1(), ind2(), 1);
  M->setNoRows(A.getNoRows());

  Ptv(int) ci(1,1); // dim 2
  int j, k;
  real amj = 0, a0j = 0, apj = 0;
  for (k=-1; k<=1; k++) {
    amj += A(-1, k).eval(ci);
    a0j += A( 0, k).eval(ci);
    apj += A( 1, k).eval(ci);
  }

  real ajm = 0, aj0 = 0, ajp = 0;
  for (k=-1; k<=1; k++) {
    ajm += A(k, -1).eval(ci);
    aj0 += A(k,  0).eval(ci);
    ajp += A(k,  1).eval(ci);
  }

                        (*M)(-1, 0) = -apj/a0j;
  (*M)( 0,-1) = -ajp/aj0; (*M)( 0, 0) = 1.;       (*M)( 0, 1) = -ajm/aj0;
                        (*M)( 1, 0) = -amj/a0j;

  for (j=-1; j<=1; j += 2)
    for (k=-1; k<=1; k += 2)
      (*M)(j, k) = (*M)( j, 0).eval(ci) * (*M)( 0, k).eval(ci);
```

```
//      (*M)(j, k) = -A(-j, -k).eval(ci) / A(0, 0).eval(ci);

  M->optimize();
  s_o<<"proj, level "<<i<<endl; M->print(s_o);

  Handle(LinEqMatrix) MM;
  MM.rebind(new LinEqMatrix(*M));

  // restriction
  proj_r(i).rebind(new ProjMatrix());
  proj_r(i)->rebindMatrix(*MM);
  proj_r(i)->init();

  // prolongation
  proj_p(i).rebind(new ProjMatrix());
  proj_p(i)->rebindMatrix(*MM);
  proj_p(i)->init();

  // nested
  proj_nest(i).rebind(new ProjMatrix());
  proj_nest(i)->rebindMatrix(*MM);
  proj_nest(i)->init();
}

void MGfdm10:: makeMatrixG(int i) // Galerkin products
{
  int n = gridSize(i);

  Handle(MatPtOp(real)) A;
  A.rebind( new MatPtOp(real) );

  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),1);         // with an offset of one from
// the central element A(0,0).
  MatPtOp(real)& AF = CAST_REF(smooth(i+1)->linAdm().getLinEqSystem ().
       A().mat(), MatPtOp(real));   // fine matrix
  MatPtOp(real)& M = CAST_REF(proj_p(i)->A().mat(), MatPtOp(real));
                                          // transfer operator
  Ptv(int) ci(1,1); // dim 2

  int j0, j1, k0, k1, l0, l1;
  for (j0=-1; j0<=1; j0++)
    for (j1=-1; j1<=1; j1++) {
      real t = 0;
      for (k0=-1+2*j0; k0<= 1+2*j0; k0++)
        for (k1=-1+2*j1; k1<= 1+2*j1; k1++) {
            real a = M(k0-2*j0, k1-2*j1).eval(ci);
            for (l0=-1+k0; l0<=1+k0; l0++)
              if ((-1 <=l0)&&(l0<= 1))
                for (l1=-1+k1; l1<=1+k1; l1++)
                  if ((-1 <=l1)&&(l1<= 1))
                    t += a * AF(k0-l0, k1-l1).eval(ci) * M(l0, l1).eval(ci);
        }
      (*A)(j0, j1) = t;
    }

  A->optimize();
  s_o<<"matrix, level "<<i<<endl; A->print(s_o);

  real h = 1.0 / real(n);

  FieldFD* rhs = new FieldFD(grid(i)(),"rhs");
  rhs->values() = 0;

  FieldFD* sol = new FieldFD(grid(i)(),"sol");
  sol->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    rhs->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions

  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*is(1);
    ps(2) = h*is(2);
    sol->valueIndex(is(1),is(2)) = u0(ps);
  }

  smooth(i)->attach(*A, *sol, *rhs);
  LinEqAdm &s = smooth(i)->linAdm();
  ddsolver->attachLinRhs(s.bl(), i, dpTRUE);
  ddsolver->attachLinSol(s.xl(), i);
}

void MGfdm10:: initProjMatrices()
{
  makeMatrix(no_of_grids);          // original discretiztion
  s_o<<"matrix, finest level\n";
  smooth(no_of_grids)->linAdm().getLinEqSystem().A().mat().print(s_o);

  for (int i=no_of_grids-1; i>=1; i--) {
    makeProj(i);                    // operator dependent transfer
    makeMatrixG(i);                 // Galerkin product
    smooth(i)->makeSystem();
  }
  makeMatrix();
  lineq->makeSystem();
}

void MGfdm10::solveProblem()
{
  initProjMatrices();
  if (lineq->linAdm().getSolver().description().contains("Domain Decomposition")) {
    BasicItSolver& sol = CAST_REF(lineq->linAdm().getSolver(), BasicItSolver);
    DDIter& ddsol      = CAST_REF(sol, DDIter);
    ddsol.attach(*ddsolver);
  }

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

  lineq->solve();                // solve eqution system
  int niterations; BooLean c;    // for iterative solver statistics
  if (lineq->linAdm().getStatistics(niterations,c))  // iterative solver?
    s_o << oform("\n\n *** solver%sconverged in %3d iterations ***\n\n",
                 c ? " " : " not ",niterations);

  Store4Plotting::dump(u());                // dump for later visualization
  lineCurves(u());
}
```

The following input parameters may be some guideline for your experiments[17]. The test in this section remains the same. We can compare the performance of the multigrid method, see table 15, input file `test2.i` and the multigrid preconditioned Krylov iteration, see table 16, input file `test3.i` with the performance for the equation solvers for the original upwind discretization. Another comparison could be with the artificial viscosity. Studies of the performance of different smoothers and number of smoothing steps in the presence of the modified restriction and prolongation operators might also be of interest.

However, this operator dependent transfer scheme is also applicable to jumping coefficient problems. So further experiments could be to study the performance of this multigrid version in the presence of rough or jumping coefficients, where the modified transfer operators also proof to be useful.

jumping coefficient

6 Biharmonic equation

In this chapter we are looking at a different scalar symmetric elliptic differential equation. The fourth order biharmonic equation. One physical motivation, among others, of the biharmonic equation is the Kirchhoff plate bending model.

$$\begin{aligned} \Delta^2 u &= f \quad \text{on } \Omega \\ u &= g_1 \quad \text{on } \partial\Omega \\ \frac{\partial}{\partial n} u &= g_2 \quad \text{on } \partial\Omega \end{aligned}$$

Standard finite element procedures to solve this problem are higher order conforming, that is C^1 finite elements. They introduce some problems in a multigrid scheme because the finite element spaces are either non-nested and the elements are quite

[17]files are in **`MGfdm10/Verify/`**

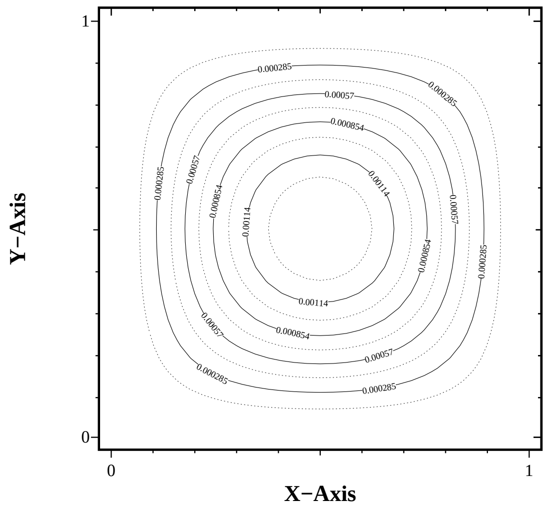

Figure 6: Solution of biharmonic equation.

complicated and expensive. The alternative in finite elements are non-conforming approaches, solving a coupled system of two Poisson equations. Techniques for non-conforming finite elements apply.

However, the finite difference approach for the biharmonic equation is completely different. Using a standard 13-point difference stencil in two dimensions

$$\frac{1}{h^4}\begin{bmatrix} & & 1 & & \\ & 2 & -8 & 2 & \\ 1 & -8 & 20 & -8 & 1 \\ & 2 & -8 & 2 & \\ & & 1 & & \end{bmatrix}$$

We have a rather simple conforming approach, where all interior nodes can be treated in the same way. The boundary conditions require some special treatment of two layers of boundary nodes, since the support of the difference stencil is larger than the one layer of the 5- 7- or 9- point Laplacian. The Dirichlet conditions given above (note that we impose two conditions per node) can be implemented by defining the values of all nodes on both boundary layers of nodes.

The main theoretical difficulty for iterative solvers for this discretization is, that the stiffness matrix is not longer an M-matrix. For the application of multigrid theory also requires the use of higher order restriction and prolongation schemes (at least quadratic). This reflects the fact that the finite element discretization also requires higher order elements.

We derive the finite difference multigrid code for the biharmonic equation in `Diffpack` from the standard finite difference multigrid code `MGfdm2`[18].

[18]you will find the code in `MGfdm12/`

MGfdm12.h

```
#ifndef MGfdm12_h_IS_INCLUDED
#define MGfdm12_h_IS_INCLUDED

#include <MGfdm2.h>

class MGfdm12: public MGfdm2
{
protected:
  virtual void makeMatrix();            // set up lineq Matrix
  virtual void makeMatrix(int i);       // set up smooth matrix

  virtual void initProj();              // setup proj

  virtual int gridSize(SpaceId i);      // grid size n, lattice n*n
  virtual void scanGrid (MenuSystem& menu);

  virtual void defineIndexSetB (Handle(IndexSet)& boundary, int n);
  virtual void defineIndexSetI (Handle(IndexSet)& interior, int n, int s=1);
public:
};
#endif
```

We have to redefine the definition of the differential operators in the `makeMatrix` procedures. We also have to modify the grid handling procedures and the procedures defining the node iterators in order to deal with the two boundary layers of nodes and the additional boundary conditions.

We use the 13-point stencil discretization of the biharmonic operator and bi-linear restriction and prolongation. The additional boundary layer extends the side length of the global grid by $2h$.

MGfdm12.C

```
#include <MGfdm12.h>
#include <MatPtROp_real.h>

int MGfdm12:: gridSize(SpaceId i)
{
  return 2 + ((coarse_grid / 2) << i);
}

void MGfdm12:: makeMatrix()
{
  int n = gridSize(no_of_grids);

  Handle(MatPtOp(real)) A = new MatPtOp(real);
  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),2);         // with an offset of two from
// the central element A(0,0).

  real h = 1.0 / real(n-2);
  real s = h*h;

                         (*A)( 0,-2) =  1/s;
      (*A)(-1,-1) = 2/s; (*A)( 0,-1) = -8/s; (*A)( 1,-1) = 2/s;
  (*A)(-2, 0) = 1/s;
      (*A)(-1, 0) = -8/s; (*A)( 0, 0) = 20/s; (*A)( 1, 0) = -8/s;
                                                 (*A)( 2, 0) = 1/s;
      (*A)(-1, 1) = 2/s; (*A)( 0, 1) = -8/s; (*A)( 1, 1) = 2/s;
                         (*A)( 0, 2) =  1/s;

  FieldFD* b = new FieldFD(grid(no_of_grids)(),"b"); // !!
  b->values() = 0;

  u.rebind (new FieldFD(grid(no_of_grids)(),"u"));
  u->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*(is(1)-1);
    ps(2) = h*(is(2)-1);
    b->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions
  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*(is(1)-1);
    ps(2) = h*(is(2)-1);
    u->valueIndex(is(1),is(2)) = u0(ps);
  }

  lineq->attach(*A, *u, *b);
}

void MGfdm12:: makeMatrix(int i)
{
  int n = gridSize(i);

  Handle(MatPtOp(real)) A;
  A.rebind( new MatPtOp(real) );

  Handle(IndexSet)       ind;        // 'interior' index set
  Handle(IndexSet)       bou;        // 'boundary' index set
  defineIndexSetI(ind, n);
  defineIndexSetB(bou, n);

// point operator, two dimensional
  A->redim(ind(),2);         // with an offset of two from
// the central element A(0,0).
  real h = 1.0 / real(n-2);
  real s = h*h;

                           (*A)( 0,-2) =  1/s;
        (*A)(-1,-1) = 2/s; (*A)( 0,-1) = -8/s; (*A)( 1,-1) = 2/s;
  (*A)(-2, 0) = 1/s;
        (*A)(-1, 0) = -8/s; (*A)( 0, 0) = 20/s; (*A)( 1, 0) = -8/s;
                                                 (*A)( 2, 0) = 1/s;
        (*A)(-1, 1) = 2/s; (*A)( 0, 1) = -8/s; (*A)( 1, 1) = 2/s;
                           (*A)( 0, 2) =  1/s;

  FieldFD* rhs = new FieldFD(grid(i)(),"rhs"); // !!
  rhs->values() = 0;

  FieldFD* sol = new FieldFD(grid(i)(),"sol"); // !!
  sol->values() = 0;

  // initialization of the right hand side

  Ptv(int)  is(2);
  Ptv(real) ps(2);

  ind().startIterator(is);
  while (ind().iterate()) {
    ps(1) = h*(is(1)-1);
    ps(2) = h*(is(2)-1);
    rhs->valueIndex(is(1),is(2)) = h*h*f(ps);
  }

  // initialization of the boundary conditions

  bou().startIterator(is);
  while (bou().iterate()) {
    ps(1) = h*(is(1)-1);
    ps(2) = h*(is(2)-1);
    sol->valueIndex(is(1),is(2)) = u0(ps);
  }

  smooth(i)->attach(*A, *sol, *rhs);
  LinEqAdm &st = smooth(i)->linAdm();
  ddsolver->attachLinRhs(st.bl(), i, dpTRUE);
  ddsolver->attachLinSol(st.xl(), i);
}

void MGfdm12:: defineIndexSetI (Handle(IndexSet)& indi, int n, int step)
{
  BoxIndices* interior = new BoxIndices;
  indi.rebind (interior);

  Ptv(int) steps(2);
  steps = step;

  interior->scan(aform("2(%d,%d)  (%d,%d)", 1+step, 1+step, n-step-1, n-step-1));
  interior->setSteps(steps);
```

```
}

void MGfdm12:: defineIndexSetB (Handle(IndexSet)& indb, int n)
{
  IndexSetIndices& boundary =  *new IndexSetIndices(4);
  indb.rebind (boundary);

  BoxIndices boxx1;  // two lines of boundary

  boxx1.scan(aform("2(0,0)  (%d,1)",n));
  boundary.add(boxx1);
  boxx1.scan(aform("2(0,%d)  (%d,%d)",n-1,n,n));
  boundary.add(boxx1);
  boxx1.scan(aform("2(0,2)  (1,%d)",n-2));
  boundary.add(boxx1);
  boxx1.scan(aform("2(%d,2)  (%d,%d)",n-1,n,n-2));
  boundary.add(boxx1);
}

void MGfdm12:: scanGrid(MenuSystem& menu)
{
  for (int i=1; i<=no_of_grids; i++) {
    int n = gridSize(i);
    real h = 1. / n;

    grid(i).rebind (new GridLattice);
    grid(i)->scan(aform("d=2 [%g,%g]x[%g,%g] index:[0:%d]x[0:%d]",
                    -h, 1+h, -h, 1+h, n, n));
    if (i>1)
      menu.setCommandPrefix("smoother");
    else
      menu.setCommandPrefix("coarse grid");
    smooth(i).rebind (new FdmLinAdm);
    smooth(i)->scan(menu);
    menu.unsetCommandPrefix();
  }
}

void MGfdm12:: initProj()
{
  for (int i=1; i<no_of_grids; i++) {
    int n = gridSize(i);
    int m = smooth(i+1)->linAdm().getLinEqSystem (). A().mat().getNoRows();

    Handle(MatPtROp(real)) M;
    M.rebind(new MatPtROp(real));

    Handle(IndexSet)       ind1;        // 'interior' index set fine
    defineIndexSetI (ind1, gridSize(i+1), 2);

    Handle(IndexSet)       ind2;        // 'interior' index set coarse
    defineIndexSetI (ind2, n, 1);

    M->redim(ind1(), ind2(), 1);

    (*M)(-1,-1) = .25;  (*M)(-1, 0) = .5;   (*M)(-1, 1) = .25;
    (*M)( 0,-1) = .5;   (*M)( 0, 0) = 1.;   (*M)( 0, 1) = .5;
    (*M)( 1,-1) = .25;  (*M)( 1, 0) = .5;   (*M)( 1, 1) = .25;
    M->optimize();
    M->setNoRows(m);
    Handle(LinEqMatrix) MM;
    MM.rebind(new LinEqMatrix(*M));

    // restriction
    proj_r(i).rebind(new ProjMatrix());
    proj_r(i)->rebindMatrix(*MM);
    proj_r(i)->init();

    // prolongation
    proj_p(i).rebind(new ProjMatrix());
    proj_p(i)->rebindMatrix(*MM);
    proj_p(i)->init();

    // nested
    proj_nest(i).rebind(new ProjMatrix());
    proj_nest(i)->rebindMatrix(*MM);
    proj_nest(i)->init();
  }
}
```

The following input parameters may be some guideline for your experiments[19].

Since we have a fourth order differential operator, which means a condition number of $\mathcal{O}(h^{-4})$ instead of $\mathcal{O}(h^{-2})$ for the Laplacian, and a discretization, which violates the M-matrix property, we expect the iterative equation solvers to be less efficient than for the Laplacian. First we try the conjugated gradient method, see table 17, input file `test1.i`. Observe the number of iterations dependent on the grid size/ number of levels. You should be able to observe the effect of the $\mathcal{O}(h^{-4})$ increase of the condition number.

menu item	**answer**
no of grid levels	4
coarse lattice	2
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecNone

Table 17: Conjugated gradients for the biharmonic equation, `test1.i`

Due to the discretization, we also expect the multigrid method to be sensitive to changes in the smoother and its parameters and the number of smoothing steps. So we suggest to start with a more robust multigrid W-cycle instead of the standard V-cycle, see table 18, input file `test2.i`. The main point is of course the dependence of the number of iterations/ the convergence rate on the number of discretization levels. You should do some experiments with different smoothers such as `SOR`, `Jacobi` and `ConjGrad` instead, modify relaxation parameters and number of steps. You can also try to find good parameters for the V-cycle.

[19]files are in `MGfdm6/Verify/`

menu item	**answer**
no of grid levels	4
coarse lattice	2
sweeps	[2,2]
matrix type	MatPtOp
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	Multigrid
cycle type gamma	2
smoother matrix type	MatPtOp
smoother basic method	SSOR
smoother relaxation parameter	1.4
coarse grid matrix type	MatPtOp
coarse grid basic method	SSOR
coarse grid max iterations	1

Table 18: Multigrid for the biharmonic equation, `test2.i`

The comparison with the multigrid preconditioned conjugated gradient method in this case is interesting, see table 19, input file `test3.i`. So a comparison of the convergence rate and the work per iteration for both alternatives is interesting. Try to explain, why the conjugated gradient method is so efficient for this discretization of the biharmonic equation.

7 Conclusion

In this report we have demonstrated the use of iterative multigrid equation solvers for the finite difference discretization of different partial differential equations. While the previous introductory report on multigrid covered the Poisson equation, this reports extends the variety of equations. We applied multigrid to non-symmetric problems implementing the convection-diffusion equation with artificial viscosity and with upwind schemes, to varying coefficient problems and to anisotropic problems.

The simulators were based on a standard finite difference Poisson equation simulator with multigrid developed in the introductory report [Zum96b]. 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 needed.

We could use the multigrid method both as a stand-alone iterative solver and as a preconditioner for a conjugated gradient method (Krylov method in general). Both the discretization and the transfer operators were implemented in a finite difference stencil fashion, which requires only a constant amount of memory regardless the number of unknowns and matrix size.

The discussion of the code and some numerical properties was accompanied with numerical experiments and exercise to be done using the codes. This discussion of

menu item	**answer**
no of grid levels	4
coarse lattice	2
sweeps	[2,2]
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	Multigrid
cycle type gamma	2
smoother matrix type	MatPtOp
smoother basic method	SSOR
smoother relaxation parameter	1.4
coarse grid matrix type	MatPtOp
coarse grid basic method	SSOR
coarse grid max iterations	1

Table 19: Conjugated gradients with multigrid preconditioner for the biharmonic equation, `test3.i`

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.

[Wes92] P. Wesseling. *An Introduction to Multigrid Methods.* John Wiley & Sons, Chichester, 1992.

[Zum96a] G. W. Zumbusch. Multigrid applied to different partial differential operators. Technical report, SINTEF Applied Mathematics, Oslo, 1996.

[Zum96b] G. W. Zumbusch. Multigrid for finite differences. Technical report, SINTEF Applied Mathematics, Oslo, 1996.

[Zum96c] G. W. Zumbusch. Multigrid methods in Diffpack. Technical Report STF42 F96016, SINTEF Applied Mathematics, Oslo, 1996.