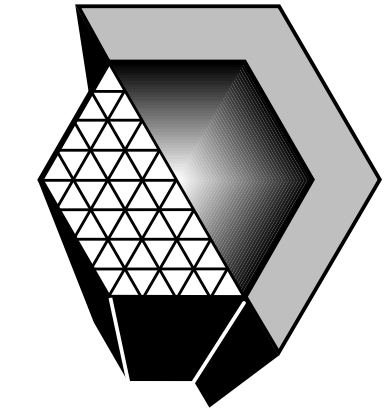
Multigrid for finite differences

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/`.

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 serves as an alternative introductory report on the multigrid iterative solvers in `Diffpack` using finite differences instead of finite elements covered previously. We consider the solution of elliptic partial differential equations on different domains. We solve the resulting linear equation systems with a multigrid iteration or a Krylov iteration with a multigrid preconditioner. The multigrid restriction and prolongation are also implemented using finite "difference" type stencils. The first steps are guided by a couple of examples and exercises.

Contents

Multigrid for finite differences

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 at a time. The multigrid method reaches optimal linear complexity which is comparable to input/output procedures and other data handling in a computation.

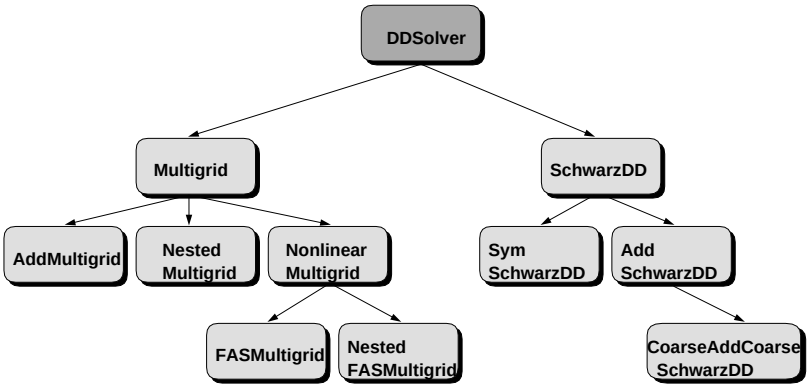


Figure 1: Hierarchy of multigrid and domain decomposition methods

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) discretizations to correct solutions on finer (= expensive) discretizations. The standard way to do this is called V-cycle.

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

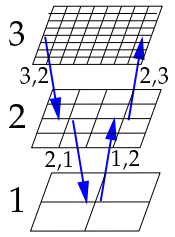


Figure 2: Multigrid V-Cycle

Given a second order differential operator $\mathcal{L}$ and a domain Ω, we look for the solution of

$$\begin{aligned} \mathcal{L}u &= f \quad \text{on } \Omega \\ u &= g_1 \quad \text{on } \Gamma \subset \partial\Omega \\ \tfrac{\partial}{\partial n}u &= g_2 \quad \text{on } \partial\Omega \setminus \Gamma \end{aligned}$$

algorithm

on a bounded domain Ω. The multigrid algorithm for an initial guess x and a right hand side b to solve the equation system may be written recursively as

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

where $\mathcal{S}$ denotes 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]. For a more detailed presentation of the multigrid method we refer to text books like [Hac85],[Hac94, chapter 10]. An alternative introduction to multigrid methods in `Diffpack`, based on finite element discretizations can be found in [Zum96] together with additional references. It may be helpful to have access to the `Diffpack` manual pages `dpman` while reading this tutorial. Some of the documentation of finite differences in `Diffpack` can only be found in the online manual, see e.g. `dpman MatPtOp` and `dpman PtOpDS`. The source codes and all the input files are available at `$DPR/src/app/pde/ddfem/src/`.

The report is organized as follows: We introduce the multigrid and domain decomposition solver interface in `Diffpack`. Next we develop a finite difference Poisson equation simulator based on the `Diffpack` class `MatPtOp` with the standard choice of direct and iterative solvers offered by `LinEqAdm`. We extend this simulator for a multigrid iterative solver and a multigrid preconditioner for Krylov iterations. In the last chapter we show how to run the simulator on an L-shaped domain instead of the unit square. All the codes along with detailed explanations and sample input files are given. Several experiments with the code are proposed to explore the properties of the methods under consideration.

2 Interface

The `Diffpack` implementation of the multigrid method is based on the `DDSolverUDC` interface. We will explain how to implement the necessary functions.

DDSolverUDC

```
/*<DDSolverUDC:*/
class DDSolverUDC : public HandleId
{
public:
  DDSolverUDC () {}
  virtual ~DDSolverUDC ();

  virtual SpaceId getNoOfSpaces() const = 0;

  virtual void setStart (LinEqVector& x, SpaceId space, StartVectorMode start);

  virtual BooLean solveSubSystem (
    LinEqVector& b, LinEqVector& x, SpaceId space,
    StartVectorMode start, DDSolverMode mode=SUBSPACE) = 0;
    // return value indicates changes of the solution vector

  virtual void residual (
    LinEqVector& b, LinEqVector& x, LinEqVector& r, SpaceId space);

  virtual void matVec (
    const LinEqVector& b, LinEqVector& x, SpaceId space);

  virtual BooLean transfer (
    const LinEqVector& fv, SpaceId fi,
          LinEqVector& tv, SpaceId ti,
          BooLean add_to_t= dpFALSE, DDTransferMode=TRANSFER) = 0;
          // indicates changes of the solution vector

  virtual int  getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork work_tp)
    const = 0;
  virtual real getStorageTransfer (SpaceId fi, SpaceId ti) const = 0;
  virtual int  getWorkSolve (SpaceId space, const PrecondWork work_tp)
    const = 0;
  virtual real getStorageSolve (SpaceId space) const = 0;

  virtual String comment ();
};
/*>DDSolverUDC:*/
```

The function `getNoOfSpaces` returns the number of grids j. The function `solveSubSystem` implements the smoother $\mathcal{S}(x,b)$ on the grid number `space`. The `start` argument may indicate a zero start vector x. A zero start vector simplifies parts of the computation and is therefore treated differently. For example multiplying with a matrix by zero is easy. `mode` is used to make a distinction between the pre- $\mathcal{S}^1$ and the post-smoothing $\mathcal{S}^2$ if necessary. One should be able to compute the residual $b - \mathcal{L}_j x$ by the function `residual`. The grid transfer is done via the `transfer` function. It implements a transfer

$$\text{transfer}(f,t) : V_f \mapsto V_t$$

In the multigrid case it is only used for the prolongation $j-1 \to j$ and the restriction $j \to j-1$.

We need some enumeration flags to indicate different modes, for the smoothers `DDSolverMode` and the common `StartVectorMode` and `DDTransferMode` for some specific transfer operators. For the multigrid method we only need some of the values. We also use a special type enumerating the grids.

enum

```
/*<SpaceId:*/
typedef int SpaceId;
/*>SpaceId:*/
/*<DDSolverMode:*/
enum DDSolverMode
{
  SUBSPACE = 1,       // only one solver: coarse solver, symmetric solver ...
  SUBSPACE_FWD = 2,   // first solver, presmoothing, ...
  SUBSPACE_BACK = 3   // second solver, postsmoothing, ...
};

enum DDTransferMode
{
  TRANSFER = 1,       // standard transfer
  TRANSFER_NESTED = 2 // higher order transfer for nested iteration
};
/*>DDSolverMode:*/
```

3 The Poisson equation

We want to compute an approximation of

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega = [0,1]^2 \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

First we introduce a simulator for the linear scalar Poisson equation based on a finite difference scheme. We use a five point stencil to represent the Laplacian

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

on a uniform lattice grid on the unit square $[0,1]^2$. The grid spacing is $h = 1/n$ with $(n-1) \times (n-1)$ interior points. The finite difference operator is represented by a `MatPtOp` point operator matrix in `Diffpack`.

The `Diffpack` code is in the files `MGfdm1.h` and `MGfdm1.C`[1]. It is similar to the standard `Diffpack` simulators for finite elements [Lan94] and it is based in part on the sample codes given in the online manual `dpman MatPtOp`.

MGfdm1.h

[1] you will find the code in `MGfdm1/`

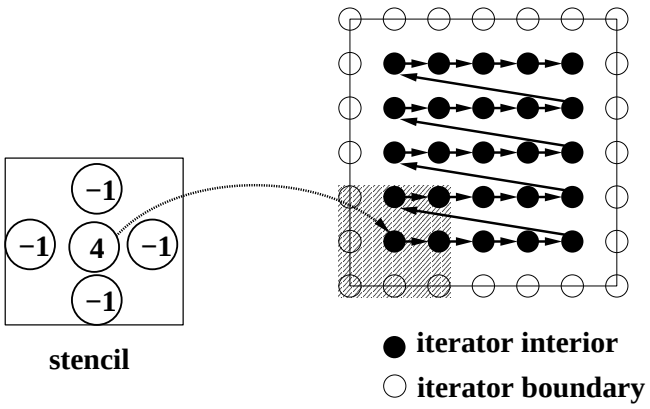


Figure 3: The index sets and the finite difference stencil cycling over all interior points.

```
#ifndef MGfdm1_h_IS_INCLUDED
#define MGfdm1_h_IS_INCLUDED

#include <FdmLinAdm.h>
#include <MenuUDC.h>
#include <Store4Plotting.h>
#include <DrawFD.h>
#include <UnixUtil.h>

class MGfdm1: public MenuUDC, public Store4Plotting
{
protected:
  // general data:
  real                        h;        // mesh size parameter
  int                         n;        // partition constant
  Handle(IndexSet)          ind;        // 'interior' index set
  Handle(IndexSet)          bou;        // 'boundary' index set
  Handle(MatPtOp(real))       A;        // point operator
  Handle(GridLattice)      grid;        // the grid
  Handle(FieldFD)             u;        // the difference approximation
  Handle(FieldFD)             b;        // the right hand side

  Handle(FdmLinAdm)        lineq;  // interface to equation solvers

  virtual real  f(Ptv(real)& ) const;
  virtual real u0(Ptv(real)& ) const;

  void scanGrid (MenuSystem& menu);
  virtual void defineIndexSet
   (Handle(IndexSet)& interior, Handle(IndexSet)& boundary);

public:
   MGfdm1();
  ~MGfdm1() {}

  virtual void adm    (MenuSystem& menu);
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu);
  virtual void solveProblem ();
```

```
  virtual void resultReport ();
};
#endif
```

The index set `ind` contains all interior points of the lattice, where the finite difference stencil is to be applied. The boundary nodes with Dirichlet boundary conditions are contained in the index set `bou`. Both index sets are initialized in `define IndexSet`. The complete lattice grid is represented by `grid`, which is initialized in `scanGrid`. The right hand side of the linear equation system is assembled in b and the solution is in `u`. The right hand side of the differential equation is implemented as function `f` and the initial solution used by an iterative solver is given by `u0`. The linear equation system and the equation solver are managed by `lineq` which is derived from `LinEqAdm`.

MGfdm1.C

```
#include <MGfdm1.h>

MGfdm1:: MGfdm1 ()
{}

void MGfdm1:: adm (MenuSystem& menu)  // administer the menu
{
  MenuUDC::attach (menu); // enables later access to menu arg. as menu_system->
  define (menu);          // define/build the menu
  menu.prompt();          // prompt user, read menu answers into memory
  scan (menu);            // read menu answers into class variables and init
}

void MGfdm1:: define (MenuSystem& menu, int level)
{
  menu.addItem(level, "n", "n",
       "No of intervals in [0,1] ",
       "16","I");
  FdmLinAdm::defineStatic(menu,level+1);
  Store4Plotting::defineStatic(menu,level+1);
}

void MGfdm1:: scan(MenuSystem& menu)
{
  A.rebind (new MatPtOp(real));
  grid.rebind (new GridLattice);
  u.rebind (new FieldFD);
  b.rebind (new FieldFD);
  lineq.rebind (new FdmLinAdm);

  n = menu.get("n").getInt();
  h = 1.0 / real(n);

  defineIndexSet(ind,bou);

  scanGrid(menu);
}

void MGfdm1:: scanGrid(MenuSystem& menu)
```

```
{
  lineq->scan(menu);
  Store4Plotting::scan(menu, 2); // dim

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

  grid().scan(aform("d=2 [0,1]x[0,1] index:[0:%d]x[0:%d]", n,n));

  u().redim(grid(),"u");
  b().redim(grid(),"b");

  u().values() = 0;
  b().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 MGfdm1:: defineIndexSet (Handle(IndexSet)& indi, Handle(IndexSet)& indb)
{
  BoxIndices& interior = *new BoxIndices;
  indi.rebind (interior);

  interior.scan(aform("2(%d,%d)  (%d,%d)",1,1,n-1,n-1));

  IndexSetIndices& boundary =  *new IndexSetIndices(6);
  indb.rebind (boundary);
  BoxIndices boxx1;

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

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

void MGfdm1::solveProblem()
{
  lineq->makeSystem();           // make the equation system
  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());
}

void MGfdm1:: resultReport()
{}

real MGfdm1:: f(Ptv(real)& ) const
{
  return 1.;
}

real MGfdm1:: u0(Ptv(real)& ) const
{
  return 0.;
}
```

The procedures `define` and `scan` handle the input menu. The procedure `resultReport` is a template for output of the solution and statistics. The five point Difference stencil is set up in the procedure `scanGrid`. The right hand side is scaled by h^2. The alternative would be to scale the difference stencil instead, which is also quite common.

$$Ax \;=\; b \text{ with } b_i = h^2 f(x_i)$$

with lattice points x_i and vectors x and b. The `defineIndexSet` procedure defines the iterators on the nodes, `indi` cycling over all interior nodes $[1, n-1]^2$ and `indb` cycling over all Dirichlet boundary nodes $[0, n]^2 \setminus [1, n-1]^2$. This also defines the order of the iteration, which is lexicographic. Some administration is hidden in the class `FdmLinAdm` which extends some of the capabilities of `LinEqAdm`. One feature is the conversion of the finite difference type operator `MatPtOp` into a banded matrix in the case you employ a Gaussian elimination equation solver or you request another matrix storage type.

The main program is only given for this example, initializing `Diffpack`, constructing the simulator class and running it. The multiple loop environment enables several different runs using all combinations of alternatives of input data, see [LP94].

main.C

```
#include <MGfdm1.h>
```

```
int main(int nargs, const char** args)
{
  initDIFFPACK(nargs,args);
  global_menu.init("finite difference Poisson equation simulator","MGfdm1");
  MGfdm1 problem;                    // make a simulator object, called problem
  global_menu.multipleLoop (problem);// solve one or several problems
  DBP("leaving main");
  return 0;
}
```

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

We give the input parameters for a run of the simulator for a solution with a conjugated gradient iteration and an absolute termination criterion, see table 1, input file `test1.i`.

menu item	**answer**
n	16
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecNone
#1: convergence monitor name	CMAbsTrueResidual
#1: max error	1.0e-7

Table 1: Conjugated gradients on a finite difference discretization, `test1.i`

Possible experiments include:

- the number of iterations depending on the grid size n
- the number of iterations depending on the iterative solution procedure like SOR and Jacobi
- a comparison of direct and iterative solvers
- a study of different termination criteria and tolerances
- a study of preconditioners for the conjugated gradient method

4 Multigrid for finite differences

We will now implement a multigrid preconditioner for the simulator `MGfdm1`. We use finite differences to represent the differential operator on each grid and we construct restriction and prolongation operators implemented in a similar way with "difference" stencils. We create a new simulator `MGfdm2` modifying and extending the original code `MGfdm1`. First we need to include the `DDSolver` related files. The class is inherited

[2]files are in `MGfdm1/Verify/`

from `DDSolverUDC` to implement the domain decomposition interface. The multigrid algorithms available in the library use this interface.

The code of the simulator follows[3]:

MGfdm2.h

```
#ifndef MGfdm2_h_IS_INCLUDED
#define MGfdm2_h_IS_INCLUDED

#include <FdmLinAdm.h>
#include <MenuUDC.h>
#include <Store4Plotting.h>
#include <DrawFD.h>
#include <UnixUtil.h>
#include <VecSimplest_Handle.h> // VecSimplest's needed
#include <DDSolver.h>           // DDSolver
#include <DDSolverUDC.h>        // interfacing to DDSolver
#include <DDSolver_prm.h>       // DDSolver parameters

class MGfdm2: public MenuUDC, public Store4Plotting, public DDSolverUDC
{
protected:
  // general data:
  VecSimplest(Handle(GridLattice))   grid;  // the grids
  Handle(FieldFD)                       u;  // the difference approximation

  prm(Precond)                 precondPrm;  // prm for DD preconditioner
  Handle(FdmLinAdm)                lineq;  // interface to equation solvers
  VecSimplest(Handle(FdmLinAdm))   smooth;  // equation solvers

  virtual real  f(Ptv(real)& ) const;
  virtual real u0(Ptv(real)& ) const;

  virtual void scanGrid (MenuSystem& menu);
  virtual void defineIndexSetB (Handle(IndexSet)& boundary, int n);
  virtual void defineIndexSetI (Handle(IndexSet)& interior, int n, int s=1);

  // multigrid related data:
  int           no_of_grids;                      // multigrid levels
  int           coarse_grid;                      // size coarse grid
  int           preSmooth;                        // no of iterations
  int           postSmooth;                       // no of iterations
  prm(DDSolver) ddsolver_prm;                     // parameters multigrid solver
  VecSimplest(Handle(Proj))             proj_p;   // projection operators
  VecSimplest(Handle(Proj))             proj_r;   // projection operators
  VecSimplest(Handle(Proj))             proj_nest; // nested transfer
  Handle(DDSolver)                      ddsolver;  // multigrid solver

  virtual void initProj();              // setup proj
  virtual void initMatrices();          // init matrices A(i)
  virtual void makeMatrix();            // set up lineq Matrix
  virtual void makeMatrix(int i);       // set up smooth matrix
  virtual int gridSize(SpaceId i);      // grid size n, lattice n*n

public:
   MGfdm2();
```

[3] you will find the code in `MGfdm2/`

```
  ~MGfdm2() {}

  virtual void adm    (MenuSystem& menu);
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu);
  virtual void solveProblem ();
  virtual void resultReport ();

  // DDSolverUDC
  SpaceId getNoOfSpaces() const;         // no_of_grids
  BooLean solveSubSystem (LinEqVector& b, LinEqVector& x,
                          SpaceId space, StartVectorMode start,
                          DDSolverMode mode=SUBSPACE);
  void residual (LinEqVector& b, LinEqVector& x, LinEqVector& r, SpaceId space);
  BooLean transfer (const LinEqVector& fv, SpaceId fi,
                          LinEqVector& tv, SpaceId ti,
                    BooLean add_to_t= dpFALSE, DDTransferMode=TRANSFER);  // apply proj

  virtual int  getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork work_tp) const;
  virtual real getStorageTransfer (SpaceId fi, SpaceId ti) const;
  virtual int  getWorkSolve (SpaceId space, const PrecondWork work_tp) const;
  virtual real getStorageSolve (SpaceId space) const;
  String  comment ();
};
#endif
```

We need a hierarchy of lattice grids `grid` in contrary to the one grid in the previous case. Although we are only interested in the solution `u` on the finest grid, we will perform computations on all grids. The grids are numbered from 1 (coarse) to `no_of_grids` (fine). We have one handle to `FdmLinAdm` `lineq` to manage the equation system we want to solve. Additionally we have one `FdmLinAdm` `smooth` for each grid to manage the smoothing on the grid. A `FdmLinAdm` basically stores the representation of the matrix representing the differential operator, some vectors and a equation solver.

The equation solver of `lineq` uses a equation solver `DDIter`, which is a template for the multigrid algorithms, or a preconditioned Krylov iteration (e.g. `ConjGrad`) with a multigrid preconditioner `PrecDD`. The smoothers are implemented by the equation solvers in `smooth`. This usually is a fixed number of iterations of some iterative method like `SOR` or `Jacobi` and for larger coarse grids some direct solver in `smooth(1)`. The finite difference stencil is stored for each `FdmLinAdm` object along with an iterator over all interior nodes. This means that usually the same difference stencil is present several times, which does not require much memory. There are cases, where the stencils differ. Even the finite difference operator in `smooth(no_of_grids)` and `lineq` do not need to coincide if we use multigrid as a preconditioner.

The right hand side `f` and the initial guess `u0` remain unchanged. The `scanGrid` procedure is now responsible to create the whole hierarchy of grids. We double the

grid size n from one grid to the next grid starting with a grid size of $n =$`coarse_grid`, which means $(n-1)^2$ interior nodes. This is coded in the function `gridSize`, which returns the number n for each level. The grids are nested.

We split the creation of iterators into one procedure for interior nodes used for `lineq`, `smooth` and the restriction and prolongation operators and another procedure for boundary nodes used in `lineq`. The parameter s denotes the step size for the iterator. The default is 1, an iterator over all nodes. The step size 2 in each direction is used in restriction and prolongation operators, applying a stencil for each node, which has a counterpart on the next coarser grid.

The parameters `preSmooth` and `postSmooth` are used to override the number of iterations for all the equation solvers in `smooth` except for the coarse grid solver `smooth(1)`. The parameter `ddsolver` store the type of multigrid algorithm and its parameters. It is read from the menu.

We introduce some new procedures to structure the initialization process. The procedure `initProj` initializes the prolongation `proj_p`, restriction `proj_r` and interpolation `proj_nest` (for nested multigrid) operators. The `initMatrices` procedure calls `makeMatrix()` to set up the equations to solve in `lineq` and `makeMatrix(int)` to set up the equations visible for the smoothers on each level. The finite difference stencil presented earlier is implemented here.

The procedures `adm`, `define`, `scan`, `solveProblem` and `resultReport` are the standard building blocks of `Diffpack` simulators and are adapted accordingly.

The `DDSolverUDC` interface is implemented through the following procedures: `getNoOfSpaces` returns the number of grid levels stored in `no_of_grids`. The procedure `solveSubSystem` implements the smoother, which is the approximate solution on a given grid `space` with right hand side `b` and initial guess `x`. The solution is returned in `x`. The flag `start` indicates whether the initial guess is valid or the iteration should be started with zero. The flag `mode` enables us to separate pre-smoothing and post-smoothing and to use the appropriate number of iterations. The return value indicates whether the vector `x` has been touched. It does not have to be touched for zero iterations, either zero pre- or zero post-smoothing steps.

The procedure `transfer` implements the projection and restriction operators. The vector `fv` on grid `fi` has to be transferred to grid `ti`. The result is stored in vector `tv`. This can both be prolongation $i \to i+1$ as a restriction $i+1 \to i$. Either the operator `proj_p` or `proj_r` is called. The flag `add_to_t` indicates whether the projected vector should be overwritten onto tv or it should be added. This flag is passed to the projection operators, which themselves pass this flag to the matrix multiply usually employed. The flag `mode` can be used to separate the case of standard prolongation of corrections and some other interpolation of a solution in a nested iteration cycle. In this case we can use the same operator. This would not be true if we had used another scaling of the finite difference stencil. In such a case both operations would always differ.

The last procedures `getWorkTransfer`, `getStorageTransfer`, `getWorkSolve` and `getStorageSolve` provide some statistics which is passed to the linear solver or preconditioner statistics and describes estimates for the smoothers and transfer op-

erators, both in memory requirement and number of operations. An identifier string `comment` should also be implemented.

MGfdm2.C

```
#include <MGfdm2.h>
#include <createDDSolver.h> // creating multigrid object
#include <DDIter.h>
#include <PrecDD.h>
#include <MatPtROp_real.h>

MGfdm2:: MGfdm2 ()
{}

void MGfdm2:: adm (MenuSystem& menu)  // administer the menu
{
  MenuUDC::attach (menu); // enables later access to menu arg. as menu_system->
  define (menu);          // define/build the menu
  menu.prompt();          // prompt user, read menu answers into memory
  scan (menu);            // read menu answers into class variables and init
}

void MGfdm2:: define (MenuSystem& menu, int level)
{
  menu.addItem (level,
                "no of grid levels", // menu command/name
                "level",        // command line option: +level
                "no of uniform refinements",
                "4",            // default answer
                "I1");          // valid answer: 1 integer

  menu.addItem (level,
                "coarse lattice", // menu command/name
                "lattice",        // command line option: +level
                "lattice is n*n",
                "2",            // default answer
                "I1");          // valid answer: 1 integer

  menu.addItem (level,
                "sweeps",       // menu command/name
                "sweeps",       // command line options: +sweeps
                "string like [2,2] = pre & post smoothing sweeps",
                "[1,1]",        // default answer: V1,1 cycle
                "S");           // valid answer: string

  FdmLinAdm:: defineStatic(menu,level+1);
  prm(DDSolver):: defineStatic (menu, level+1);// multigrid parameters

  menu.setCommandPrefix("smoother");
  FdmLinAdm:: defineStatic(menu,level+1);
  menu.setCommandPrefix("coarse grid");
  FdmLinAdm:: defineStatic(menu,level+1);
  menu.unsetCommandPrefix();

  Store4Plotting:: defineStatic(menu,level+1);
}

void MGfdm2:: scan(MenuSystem& menu)
{
```

```
  // load answers from the menu:
  no_of_grids = menu.get ("no of grid levels").getInt();
  coarse_grid = menu.get ("coarse lattice").getInt();

  grid.redim (no_of_grids);
  smooth.redim (no_of_grids);
  proj_p.redim (no_of_grids-1);
  proj_r.redim (no_of_grids-1);
  proj_nest.redim (no_of_grids-1);

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

  lineq.rebind (new FdmLinAdm);
  lineq->scan(menu);
  Store4Plotting::scan(menu, 2); // dim

  scanGrid(menu);

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

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

int MGfdm2:: gridSize(SpaceId i)
{
  return (coarse_grid / 2) << i;
}

void MGfdm2:: scanGrid(MenuSystem& menu)
{
  for (int i=1; i<=no_of_grids; i++) {
    int n = gridSize(i);

    grid(i).rebind (new GridLattice);
    grid(i)->scan(aform("d=2 [0,1]x[0,1] index:[0:%d]x[0:%d]", 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 MGfdm2:: 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)( 0,-1) = -1;
  (*A)(-1, 0) = -1;  (*A)( 0, 0) =  4;  (*A)( 1, 0) = -1;
                     (*A)( 0, 1) = -1;

  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 MGfdm2:: 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)( 0,-1) = -1;
  (*A)(-1, 0) = -1;  (*A)( 0, 0) =  4;  (*A)( 1, 0) = -1;
```

```
                    (*A)( 0, 1) = -1;

  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);
}

void MGfdm2:: 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)",step,step,n-step,n-step));
  interior->setSteps(steps);
}

void MGfdm2:: defineIndexSetB
(Handle(IndexSet)& indb, int n)
{
  IndexSetIndices& boundary =  *new IndexSetIndices(4);
  indb.rebind (boundary);

  BoxIndices boxx1;

  boxx1.scan(aform("2(0,0)  (%d,0)",n));
```

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

void MGfdm2:: initProj() // setup proj operators
{
  for (int i=1; i<no_of_grids; i++) {
    int n = gridSize(i);
    Handle(IndexSet)       ind1;        // 'interior' index set fine
    defineIndexSetI (ind1, gridSize(i+1), 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(smooth(i+1)->linAdm().getLinEqSystem ().
 A().mat().getNoRows());

    (*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();

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

void MGfdm2:: initMatrices()
{
  for (int i=1; i<=no_of_grids; i++) {
    makeMatrix(i);
    smooth(i)->makeSystem();
  }
  makeMatrix();
```

```
  lineq->makeSystem();
}

void MGfdm2::solveProblem()
{
  initMatrices();
  initProj();

  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());
}

void MGfdm2:: resultReport()
{}

real MGfdm2:: f(Ptv(real)& ) const
{
  return 1.;
}

real MGfdm2:: u0(Ptv(real)& ) const
{
  return 0.;
}

SpaceId MGfdm2:: getNoOfSpaces() const
{
  return no_of_grids;
}

BooLean MGfdm2:: solveSubSystem (
    LinEqVector& b, LinEqVector& x,
SpaceId space, StartVectorMode start, DDSolverMode mode)
{
  prm(LinEqSolver) &smooth_prm = (prm(LinEqSolver)&)
    smooth(space)->linAdm().getSolverPrm ();
  if (space>1) {
    if ((mode==SUBSPACE_FWD)||(mode==SUBSPACE))
      smooth_prm.max_iterations = preSmooth;
    else if (mode==SUBSPACE_BACK)
      smooth_prm.max_iterations = postSmooth;
```

```
    else fatalerrorFP("MGfdm2:: solveSubSystem","mode ", mode);
    if (smooth_prm.max_iterations == 0)
      return dpFALSE; // solution has not changed
  }
  smooth_prm.startmode = start;

  LinEqSystemPrec &lin = smooth (space)->linAdm().getLinEqSystem ();
  LinEqSolver     &sol = smooth (space)->linAdm().getSolver();
  lin.attach (x, b);

  sol.solve  (lin);

  return dpTRUE; // solution has changed
}

void MGfdm2:: residual (
    LinEqVector& b, LinEqVector& x, LinEqVector& r, SpaceId space)
{
  LinEqSystemPrec &lin = smooth (space)->linAdm().getLinEqSystem ();
  lin.attach   (x, b);

  lin.residual (r);
}

BooLean MGfdm2:: transfer (
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode mode)
{
  if (mode == TRANSFER) {
    if (fi == ti-1)                          // prolongation
      proj_p (fi)->apply (fv, tv, NOT_TRANSPOSED, add_to_t);
    else if (fi == ti+1)                     // restriction
      proj_r (ti)->apply (fv, tv, TRANSPOSED, add_to_t);
    else fatalerrorFP("MGfdm2:: transfer","from %d to %d", fi, ti);
  } else if (mode == TRANSFER_NESTED) {
    if (fi == ti-1)                          // nested interpolation
      proj_nest (fi)->apply (fv, tv, NOT_TRANSPOSED, add_to_t);
    else fatalerrorFP("MGfdm2:: transfer nested","from %d to %d", fi, ti);
  }
  else fatalerrorFP("MGfdm2:: transfer mode?","mode %d", mode);

  return dpTRUE;
}

int MGfdm2:: getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork) const
{
  if (fi == ti-1)
    return proj_p (fi)->getWork();
  if (fi == ti+1)
    return proj_p (ti)->getWork();
  return 0;
}

real MGfdm2:: getStorageTransfer (SpaceId fi, SpaceId ti) const
{
  if (fi == ti-1)
    return proj_p (fi)->getStorage();
  return 0;
}
```

```
int MGfdm2:: getWorkSolve (SpaceId space, const PrecondWork) const
{ return CAST_CONST_AWAY(MGfdm2)->smooth (space)->
    linAdm().getSolver().getWork(); }

real MGfdm2:: getStorageSolve (SpaceId space) const
{ return CAST_CONST_AWAY(MGfdm2)->smooth (space)->
    linAdm().getSolver().getStorage(); }

String MGfdm2:: comment ()
{ return "MGfdm2 multigrid test"; }
```

The finite difference stencil in `makeMatrix()` and `makeMatrix(int)` representing the Laplacian is again chosen to be a five point star.

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

The prolongation and adjoint restriction operators are defined in procedure `initProj`. We use the interpolation scheme analogue to bilinear finite elements.

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

In fact the setting of nested finite element spaces and a natural prolongation, if $x \in V_1 \subset V_2$ then also $x \in V_2$ does not apply in the case of finite differences, since functions are only known at certain nodes on a grid. However, we can use the operators derived from multigrid for finite elements. We can also use different operators, which do not have a continuous counterpart.

We generate a rectangular shaped matrix `M` of size

$$\mathtt{M} \in \mathrm{R}^{\dim V_2 \times \dim V_1}$$

which defines the prolongation/ interpolation. The transposed is used for restriction. We use a projection of type `ProjMatrix` which is initialized by the matrix. In the finite element context we use some projection classes with built-in interpolation algorithms instead.

We construct a matrix of type `MatPtROp` which is a rectangular counterpart of the matrix type `MatPtOp` for $n \times n$ matrices used for the finite difference stencil. The matrix `MatPtROp` is initialized with two index sets instead of one: We use one iterator for the rows V_2 and one iterator for the columns V_1 (see figure 4). Both iterators are used at the same time and have to iterate over exactly the same number of nodes. To achieve this, we iterate over all nodes on the coarse grid V_1. The corresponding iterator on the fine grid V_2 iterates over every second node in each direction. The matrix multiply is done in a way that the stencil is applied to every node pointed to by the iterator. The 3×3 stencil with given center point is then applied on the V_2

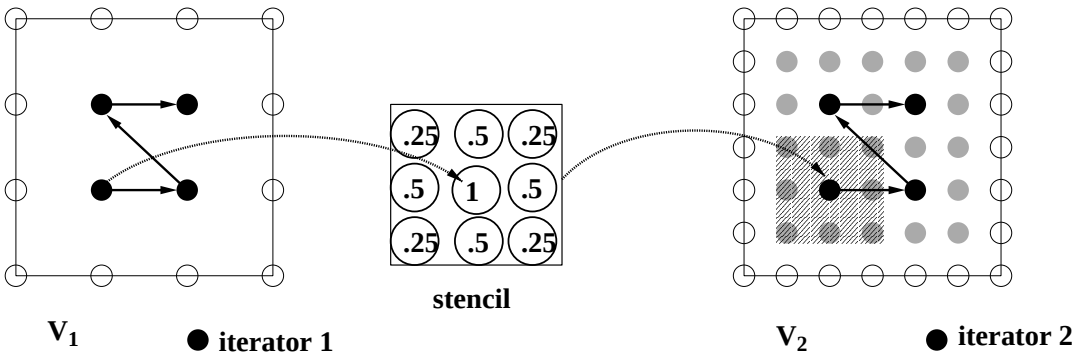


Figure 4: Projection and restriction with stencils using `MatPtROp`.

side which corresponds to the node pointed to on the V_1 side. Which side is read and which is written depends on whether we apply the operator or its transposed.

The procedure `solveProblem` performs two `DDSolver` specific tasks. Since a multigrid solver form the menu is only a template `DDIter` and a multigrid preconditioner is a template `PrecDD`, we have to pass a reference to the actual algorithm contained in a `DDSolver` object. This is done with an `attach` respective an `init` call.

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

We give the input parameters for the new simulator to run the same examples as in the previous chapter, see table 2, input file `test1.i`. This standard iterative solvers can be used for comparisons with the following multigrid solvers

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

The input parameters for multigrid as an iterative solver is given in table 3, input file `test2.i`. We use a multigrid V-cycle, indicated by `cycle type gamma` equals 1. A number two means a multigrid W-cycle instead. We use one pre- and one post-smoothing step specified by `sweeps`. This is also called a $V_{1,1}$-cycle. We use a `SOR` smoother and one iteration `SOR` as a coarse grid solver.

The coarse grid is a 2×2 grid specified in `coarse lattice`. This means one interior node and one degree of freedom on the coarsest grid. Hence the coarse grid problem is solved exactly be one step of the un-damped SOR iteration. The final solution

[4]files are in `MGfdm2/Verify/`

is obtained on grid number 4 (`no of grid levels`), which means 3 refinements by bisection and a grid size of 16. We have to specify the `MatPtOp` matrix type for each grid, the global grid `matrix type`, the coarsest grid `coarse grid matrix type` and the finer grids the smoother operates on `smoother matrix type`.

menu item	**answer**
no of grid levels	4
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	DDIter
preconditioning type	PrecNone
#1: convergence monitor name	CMAbsTrueResidual
#1: max error	1.0e-7
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: Multigrid on a finite difference discretization, `test2.i`

The input file for the conjugated gradient method with a multigrid preconditioner is given in table 4, input file `test3.i`. The difference to the previous table are the lines with `basic method` and `preconditioning type` determining the role of the multigrid algorithm to play.

Possible further experiments include:

- a comparison of multigrid as a preconditioner and multigrid as an iterative solver
- a comparison of multigrid as a preconditioner and other preconditioners
- a comparison of multigrid as a iterative solver and other iterative solvers
- the number of multigrid iterations depending on the grid size/ the number of levels
- the number of multigrid iterations depending on the smoother, both the type of algorithm and the total number of smoothing steps
- the number of multigrid iterations depending on the number of smoothing steps, the difference between pre- and post-smoothing (with respect to the error norm)
- the performance of non-symmetric multigrid as a preconditioner for conjugated gradients and other non-symmetric Krylov methods with non-symmetry due to non-symmetric smoothers (e.g. SOR) or different numbers of pre- and post-smoothing

menu item	answer
no of grid levels	4
coarse lattice	2
sweeps	[1,1]
matrix type	MatPtOp
basic method	ConjGrad
preconditioning type	PrecDD
#1: convergence monitor name	CMAbsTrueResidual
#1: max error	1.0e-7
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 Multigrid preconditioner, `test3.i`

- the difference between additive and multiplicative multigrid as preconditioner
- the difference between multiplicative and nested multigrid as iterative solver
- the number of multigrid iterations depending on the coarse grid, the size and the precision of the coarse grid solution
- the computing time depending on the coarse grid, different direct and iterative solvers on a large coarse grid

5 The L-shaped domain

One severe restriction for the application of finite differences is the geometry of the domain. We have developed a code for a structured, uniform lattice grid for the unit square. We will now have a look at a domain derived as a subset of this square shaped lattice grid. Of course there are also mechanisms to cope with not aligned and even curved boundaries, modifying difference stencils locally, but this will be covered elsewhere.

The L-shaped domain is defined as the initial unit square $[0,1]^2$ minus one quarter of the square: $[0,1]^2 \setminus [1/2,1] \times [0,1/2]$, see figure 5.

We now compute an approximation of

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega = [0,1]^2 \setminus [\tfrac{1}{2},1] \times [0,\tfrac{1}{2}] \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

The code for the new class `MGfdm3` follows[5].

[5] you will find the code in `MGfdm3/`

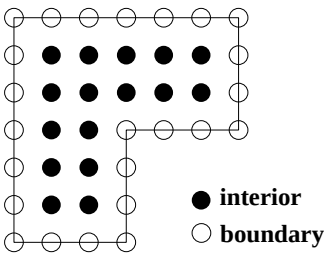


Figure 5: Index sets on the L-shaped domain.

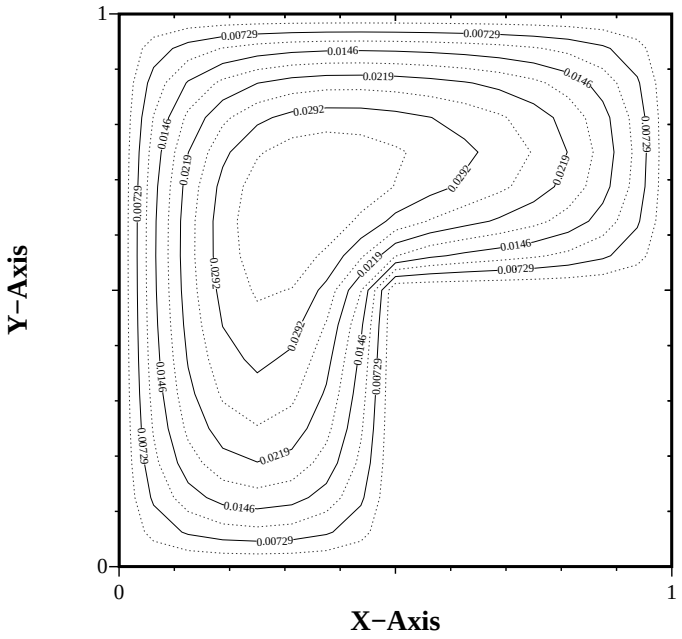


Figure 6: Solution on an L-shaped domain

MGfdm3.h

```
#ifndef MGfdm3_h_IS_INCLUDED
#define MGfdm3_h_IS_INCLUDED

#include <MGfdm2.h>

class MGfdm3: public MGfdm2 // L-shape
{
  virtual void defineIndexSetB (Handle(IndexSet)& boundary, int n);
  virtual void defineIndexSetI (Handle(IndexSet)& interior, int n, int s=1);
};
#endif
```

The class `MGfdm3` is derived from the multigrid for finite differences class `MGfdm2`. It implements the index sets differently. Consequently the initialization procedures `defineIndexSetB` for the boundary and `defineIndexSetI` for the interior nodes are overloaded. There are no further changes necessary.

MGfdm3.C

```
#include <MGfdm3.h>

void MGfdm3:: defineIndexSetI
(Handle(IndexSet)& indi, int n, int step)
{
  IndexSetIndices& interior = *new IndexSetIndices(2);

  indi.rebind (interior);

  int n2 = n/2;
  if (!n2) errorFP("MGfdm2::defineIndexSet","n>1 !");

  BoxIndices boxx1;
  Ptv(int) steps(2);
  steps = step;

  boxx1.scan(aform("2(%d,%d)  (%d,%d)",step,step,n2-step,n2));
  boxx1.setSteps(steps);
  interior.add( boxx1 );

  boxx1.scan(aform("2(%d,%d)  (%d,%d)",step,n2+step,n-step,n-step));
  boxx1.setSteps(steps);
  interior.add( boxx1 );
}

void MGfdm3:: defineIndexSetB
(Handle(IndexSet)& indb, int n)
{
  IndexSetIndices& boundary =  *new IndexSetIndices(6);

  indb.rebind (boundary);

  int n2 = n/2;
  if (!n2) errorFP("MGfdm2::defineIndexSetB","n>1 !");

  BoxIndices boxx1;

  boxx1.scan(aform("2(0,0)  (%d,0)",n2));
  boundary.add(boxx1);
  boxx1.scan(aform("2(%d,1)  (%d,%d)", n2,n2,n2));
  boundary.add(boxx1);
  boxx1.scan(aform("2(%d,%d)  (%d,%d)", n2+1,n2,n,n2));
  boundary.add(boxx1);
  boxx1.scan(aform("2(%d,%d)  (%d,%d)", n,n2,n,n));
  boundary.add(boxx1);
  boxx1.scan(aform("2(%0,%d)  (%d,%d)", n,n-1,n));
```

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

The new index sets are created as compounds of several box iterators. We cannot use the definition of the domain Ω as a boolean minus, but we can add several disjoint components. The interior is implemented as the disjoint sum of one rectangle and one square and the boundary is put together from 6 lines.

The input parameters for this example of multigrid on the L-shaped domain are similar to the parameters used on the unit square[6]. The difference is the parameter `coarse lattice`, since we have to use a larger coarse grid to have some degrees of freedom on the coarsest grid. The old 2×2 grid minus one quarter would had left only boundary nodes.

menu item	answer
coarse lattice	4

The rest of the input files remains unchanged and the new versions are also called `test1.i` (table 2) for the conjugated gradient method, `test2.i` (table 3) for the multigrid iteration and `test3.i` (table 4) for the multigrid preconditioner.

Possible further experiments include:

- a comparison of the solvers for the unit square and the L-shaped domain.
- the number of multigrid iterations depending on the grid size/ the number of levels

and of course the other experiments of the list for the unit square example. The point here is not only the different geometry of the domain, but the less regular solution and its influence on the convergence.

6 Conclusion

In this report we have demonstrated the use of iterative multigrid equation solvers for the finite difference discretization of partial differential equations. We introduced a `Diffpack` simulator for the solution of a Poisson problem based on finite differences.

We extended the simulator to execute a multigrid solver. This meant to manage and store a hierarchy of discretizations and to introduce transfer operators from one grid to the next finer and the next coarser grid. We introduced several types of equation solvers, for the coarsest grid, for a grid in the hierarchy and for the global equation system. Finally we were able to use the `DDSolver` family of multigrid (and domain decomposition) methods available in `Diffpack`. We could use the multigrid

[6] files are in `MGfdm3/Verify/`

method both as a stand-alone iterative solver and as a preconditioner for a conjugated gradient method (Krylov methods 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. This in connection with a structured array-like storage scheme for the unknowns can be considered as a great advantage compared to ordinary unstructured finite elements. We showed that this approach is also applicable to some non-unit-square shaped domains.

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

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.

[Hac94] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations.* Springer, Berlin, 1994.

[Lan94] H. P. Langtangen. Getting started with finite element programming in Diffpack. Technical Report STF33 A94050, SINTEF Informatics, Oslo, 1994.

[LP94] H. P. Langtangen and G. Pedersen. Simple and flexible input data handling in C++ programs. Technical Report STF33 A94053, SINTEF Informatics, Oslo, 1994.

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