Multigrid Methods in Diffpack

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 gives an introduction to the multigrid iterative solvers in `Diffpack`. It is meant as a tutorial for the use of iterative solvers, preconditioners and nonlinear solvers based on multigrid methods. The first steps towards this efficient equation solvers are guided by a couple of examples and exercises. Since multigrid is a recipe to construct solution algorithms rather than black-box algorithms itself, there is lots of freedom for the user to tailor the actual solver. Reflecting this fact there are lots of possibilities to use the appropriate classes in `Diffpack`. Hence there is much advice needed not to get started, but also to use the methods efficiently. The exercises are meant to give some experience needed for applications and questions not covered in this introductory report.

Contents

1 Introduction **1**

2 Interface **4**

3 My first multigrid solver **5**

3.1 Code 9

3.2 Number of grids and iterations 17

3.3 Smoother 18

3.4 W-cycle and nested iteration 20

4 Increasing the flexibility **22**

4.1 Pre- and post-smoother 27

4.2 Coarse grid solver 29

4.3 Semi-coarsening and non-standard refinement 31

4.4 Multigrid as a preconditioner 32

4.5 Additive Preconditioner 35

5 Nonlinear problems **36**

5.1 `Diffpack` nonlinear interface 36

5.2 Inexact solver 42

5.3 Nonlinear multigrid 44

5.4 Experiments 56

6 Summary **58**

References **60**

Multigrid Methods in Diffpack

Gerhard W. Zumbusch *

November 22, 1996

1 Introduction

The increase of computer power enables larger and larger numerical simulations to be performed. In the field of partial differential equations, especially in finite elements, one can easily reach the limits of any given computer. Unfortunately the size of the simulations cannot grow the same way as the computer memory and performance grows using standard methods. The bottleneck usually is the solution of the system of equations. While most operations in finite elements have linear complexity and are well suited for parallel computing with local communication patterns (like matrix assembly), standard linear algebra has a higher complexity and more expensive communication patterns. Hence the complexity of linear algebra tends to dominate any large scale simulation.

This observation leads to the development of several more efficient equation solvers especially suited for finite element computations. Starting with dense matrix and banded matrix Gaussian elimination, node ordering schemes for more efficient sparse matrix Gaussian elimination were developed. The next line of development covers the use of standard iterative solvers like Gauss-Seidel iteration and conjugated gradients with some suitable algebraic preconditioning. The equations are no longer solved exactly, but up to a precision small compared to other errors introduced in the computation. This also means that there is some responsibility left to the user to employ a suitable termination criterion for the iterative solver.

This is typical for the path of development: We are leaving simple-to-use black-box solvers like Gaussian elimination and introduce more flexibility. This also means more user responsibility for the efficiency of the method. The potential danger is twofold: The method may be inefficient due to a poor choice made by the user, and even worse the method may give wrong results due to a too early termination of the solver.

divide and conquer

Since we are still not satisfied with the performance of standard iterative solvers for large scale simulations, we introduce a divide and conquer strategy: The complexity of a standard iterative solver is still larger than *linear* complexity. Even two times the solution of a problem half the size is cheaper than solving one large problem. We consider the large problem on different scales and combine the solutions to an

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

approximation of the large problems solution. The question now is how to create coarse scale problems and how to put the solutions of the sub-problems together. We are constructing iterative solvers or preconditioners for a global problem by using solvers for smaller problems. There are several strategies to do that.

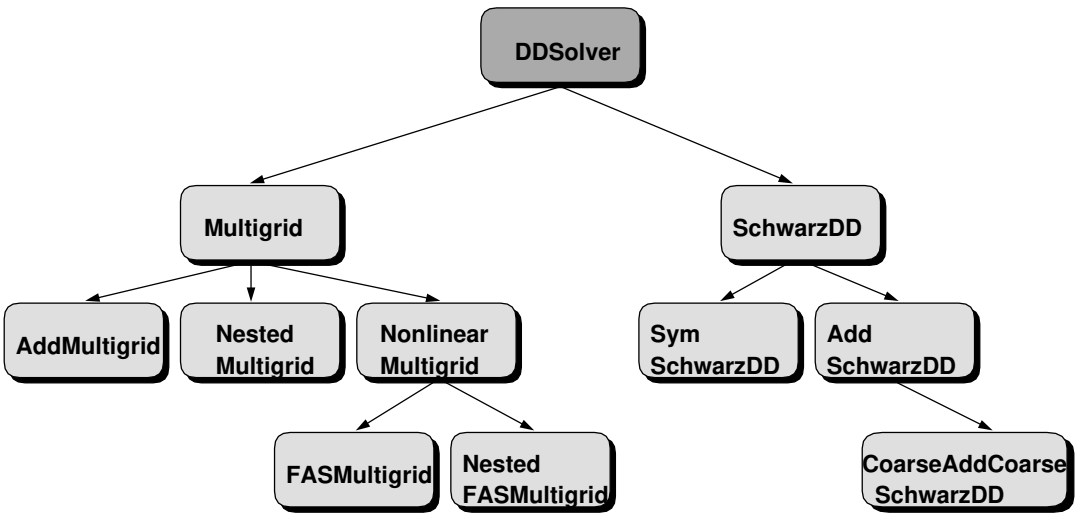

Figure 1: Hierarchy of multigrid and domain decomposition methods

The different versions of multigrid methods mainly differ in the way treating the sub-problems and putting the solutions together. This is a decision left to the user. It will turn out to be problem dependent. The multigrid method in general is only a recipe to construct a solution algorithm. Only for some model problems the most efficient components for these algorithms are known. In full generality there are only guidelines and hints for a choice of components.

recipe

It is just the purpose of this `Diffpack` tutorial to give some guidance to the use of multigrid and domain decomposition methods. Of course we will have to explain how to use the methods in `Diffpack` first. But beyond getting your own code up and running, we will discuss several applications. Different types of differential operators, grids and discretizations each lead to a specific choice of an algorithm and specific layout of its components. Users writing simulators not covered in these introductory examples may nevertheless find the discussion and the several exercises useful. The exercises cover questions, which are more general and not restricted to the specific model. They may be helpful for more advanced simulators. We also refer to forthcoming related [Zum96a] and more advanced reports [Zum96b].

exercises

Since the field of multigrid methods is a field of active research, there are lots of books, conference proceedings and thousands of research papers related. For further reading we suggest [Bri87, Joh87, Wes92] and for theory we refer to some of the literature [Hac85, Bra93]. We also refer to the proceedings of e.g. the "Copper Mountain" [MMM93] conferences and to some web pages related `ftp://na.cs.yale.edu/pub/mgnet/www/mgnet.html` and references therein.

references

We assume familiarity with some of the basic concepts of `Diffpack` [BL96]. We will use and modify some examples presented in [Lan94]. 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/`.

For the presentation of the multigrid method we will stick to the notation of Hackbusch [Hac85]. We will also use some of the examples discussed there. The multigrid method was originally proposed by Fedorenko [Fed64] and made popular in the seventies by Brandt [Bra73].

history

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

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

We discretize the problem using finite elements. We denote the finite element space V_j using elements of the size h_j (e.g. diameter). We use a family of nested finite element spaces

$$V_1 \subset V_2 \subset V_3 \ldots \subset V_j \subset \mathrm{H}^1(\Omega)$$

We actually want to compute the solution in the space V_j and we solve auxiliary problems on the coarser spaces to speed up the computation. The multigrid iteration $\Phi_j(x,b)$ on level j with start vector (initial guess) x and right hand side b reads like this

algorithm

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

coarse grid

On level 1 we use a direct coarse grid solver

$$\Phi_1(x,b) = \mathcal{L}^{-1}b$$

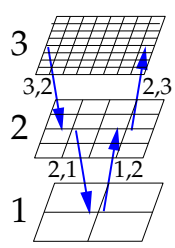

Figure 2: Multigrid V-Cycle

smoother

The definition of two operators is still missing: The smoothing operators

$$\mathcal{S} : V_j \mapsto V_j$$

as approximate solvers on the grid and the restriction $R_{j,j-1}$ and prolongation $R_{j-1,j}$ operators

grid transfer

$$\begin{aligned} R_{j,j-1} &: V_j \mapsto V_{j-1} \\ R_{j-1,j} &: V_{j-1} \mapsto V_j \end{aligned}$$

as transfer between different grids. The total scheme is also called a V-cycle (just look at figure 2).

v-cycle

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 separatly. For example multiplying with a matrix by a zero vector is cheaper than multiplying it by any non-zero vector. Zero start vectors are quite common, especially if the solver is used as a preconditioner. `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 (or projection) $j \to j-1$. We also refer to the appropriate manual pages.

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 My first multigrid solver

We start with the `Elliptic1` example simulator described in [Lan94]. We want to extend it to be able to use multigrid. It is a simulator for the *Poisson* equation on a uniform grid on a unit square or unit (hyper-) cube.

The header looks like this:[1]

MultiGrid1

```
// prevent multiple inclusion of MultiGrid1.h
#ifndef MultiGrid1_h_IS_INCLUDED
#define MultiGrid1_h_IS_INCLUDED

#include <FEM.h>           // FEM algorithms, FieldFE, GridFE etc
#include <DegFreeFE.h>     // mapping: nodal values -> unknowns in linear sys.
#include <LinEqAdm.h>      // linear systems, storage and solution
#include <MenuUDC.h>       // menu system utilities
```

[1] you will find the code in `MultiGrid1/MultiGrid1.h`

```
#include <Store4Plotting.h>// storage tool for later visualization
#include <VecSimplest_Handle.h> // VecSimplest's needed
#include <DDSolver.h>      // DDSolver
#include <DDSolverUDC.h>   // interfacing to DDSolver
#include <DDSolver_prm.h>  // DDSolver parameters

class MultiGrid1 : public FEM, public MenuUDC, public Store4Plotting, public DDSolverUDC
{
protected:
  // general data:
  Handle(FieldFE)   u;                // finite element field, the primary unknown
  Vec(real)         linsol;           // solution of linear system

  // multigrid related data:
  int no_of_grids;                                    // multigrid levels
  prm(DDSolver)                         ddsolver_prm;// parameters multigrid solver
  VecSimplest(Handle( LinEqSolver ))       smooth;    // linear solution
  VecSimplest(Handle( prm(LinEqSolver) ))  smooth_prm;// linear solution parameter
  VecSimplest(Handle( LinEqSystemStd ))    system;    // linear system, storage

  VecSimplest(Handle( GridFE ))            grid;      // finite element grid
  VecSimplest(Handle( DegFreeFE ))         dof;       // trivial mapping here: nodal values
  VecSimplest(Handle( prm(Matrix(NUMT)) )) mat_prm;   // Matrix parameters
  VecSimplest(Handle( Proj ))              proj;      // projection operators
  Handle( DDSolver )                       ddsolver;  // multigrid solver

  // general data:
  Handle( LinEqAdm )  lineq;            // linear system, storage and solution
  Handle( FieldFE )   error;            // the error field (analytical - numerical sol.)
  real L1_error, L2_error, Linf_error;  // various norms of the error

  virtual real f(const Ptv(real)& x);   // source term in the PDE
  virtual real k(const Ptv(real)& x);   // coefficient in the PDE

  virtual void fillEssBC (SpaceId space);// set boundary conditions
  virtual void integrands               // evaluate weak form in the FEM equations
    (ElmMatVec& elmat, FiniteElement& fe);
  virtual void scanGrids(MenuSystem& menu);// construct hierarchy of grids
  virtual void initProj();              // setup proj
  virtual void initMatrices();          // setup stiffness matrices on coarse grids

public:
  MultiGrid1 ();
 ~MultiGrid1 () {}

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

  // DDSolverUDC
  SpaceId getNoOfSpaces() const;        // no_of_grids
  BooLean solveSubSystem (LinEqVector& b, LinEqVector& x,
  SpaceId space, StartVectorMode start,
  DDSolverMode mode=SUBSPACE);
  // apply smoother
  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
```

The simulator class `MultiGrid1` is additionally derived from the multigrid interface `DDSolverUDC`. The appropriate function for smoothers and transfer operators have been overloaded. There is also some additional data like the number of grids `no_of_grids`, parameters and a `Handle` for the actual multigrid method `DDSolver`. Some data like `Handle(GridFE)` and `Handle(DegFreeFE)` has been turned into vectors. We now have one `GridFE` and one `DegFreeFE` object per grid with indices from 1 to `no_of_grids`.

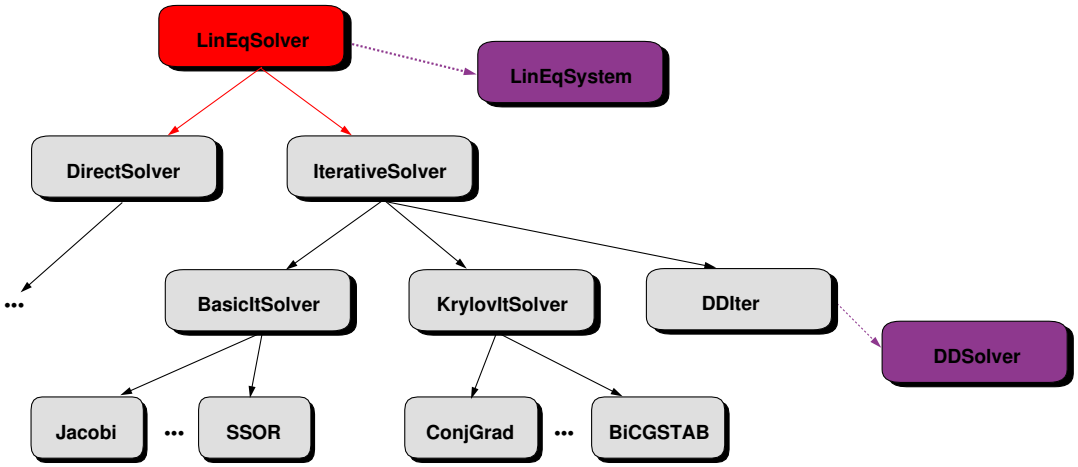

Figure 3: Linear solvers available as smoother

The transfer operators are build upon the class `Proj`. The smoothers are implemented using parameters and a `Handle` for a `LinEqSolver` (see figure 3), a `Handle` for a `LinEqSystemStd` and a parameter for `Matrix(NUMT)`.

The functions overloading the `DDSolverUDC` interface look like this:

DDSolverUDC

```
SpaceId MultiGrid1:: getNoOfSpaces() const
{ return no_of_grids; }

BooLean MultiGrid1:: solveSubSystem (
    LinEqVector& b, LinEqVector& x,
    SpaceId space, StartVectorMode start, DDSolverMode m)
{
```

```
  smooth_prm (space)->startmode = start;
  system     (space)->attach (x, b);
  smooth     (space)->solve  ( system (space)() );

  return dpTRUE; // solution has changed
}

void MultiGrid1:: residual (
    LinEqVector& b, LinEqVector& x, LinEqVector& r, SpaceId space)
{
  system (space)->attach   (x, b);
  system (space)->residual (r);
}

BooLean MultiGrid1:: transfer (
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode)
{
  if (fi == ti-1) // prolongation
      proj (fi)->apply (fv, tv, NOT_TRANSPOSED, add_to_t);
  else if (fi == ti+1) // restriction
      proj (ti)->apply (fv, tv, TRANSPOSED, add_to_t);
  else fatalerrorFP("MultiGrid1:: transfer","from %d to %d", fi, ti);
  return dpTRUE;
}

int MultiGrid1:: getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork) const
{
  if (fi == ti-1)
    return proj (fi)->getWork();
  if (fi == ti+1)
    return proj (ti)->getWork();
  return 0;
}

real MultiGrid1:: getStorageTransfer (SpaceId fi, SpaceId ti) const
{
  if (fi == ti-1)
    return proj (fi)->getStorage();
  return 0;
}

int MultiGrid1:: getWorkSolve (SpaceId space, const PrecondWork) const
{ return smooth (space)->getWork(); }

real MultiGrid1:: getStorageSolve (SpaceId space) const
{ return smooth(space)->getStorage(); }

String MultiGrid1:: comment ()
{ return "MultiGrid1 multigrid test"; }
```

The function `getNoOfSpaces` (j) simply returns the (hopefully) initialized number of grids variable.

The function `solveSubSystem` (smoother $\mathcal{S}$) is based on the `solve` function of `LinEqSolver`. The linear equation system is assumed to be initialized, the matrices have been assembled and attached. The `StartVectorMode` is passed via the `prm(LinEqSolver)`

class.

The function `residual` ($b-\mathcal{L}_j x$) is based on the `residual` function of `LinEqSystemStd`.

The function `transfer` ($R_{j-1,j}$ and $R_{j,j-1}$) uses the interpolation function `Proj`, which has been initialized previously. We choose pure interpolation as prolongation $R_{j-1,j}$ and the adjoint (`TRANSPOSED`) operation as restriction

$$R_{j,j-1} = R^*_{j-1,j}$$

3.1 Code

The code, including the initialization and the menu definitions, looks like this:[2]

MultiGrid1.C

```
#include <MultiGrid1.h>
#include <PreproBox.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <DDIter.h>
#include <createElmDef.h>      // for calling hierElmDef in MultiGrid1::define
#include <createMatrix_real.h> // creating stiffness matrices
#include <createDDSolver.h>    // creating multigrid object
#include <createLinEqSolver.h> // creating smoothers

MultiGrid1:: MultiGrid1 () {}

void MultiGrid1:: 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 MultiGrid1:: define (MenuSystem& menu, int level)
{
  // the domain is fixed: [0,1]^nsd
  menu.addItem (level,          // menu level (level+1 indicates a submenu)
                "no of grid levels", // menu command/name
                "level",        // command line option: +level
                "no of uniform refinements",
                "4",            // default answer (2D problem)
                "I1");          // valid answer: 1 integer

  menu.addItem (level,          // menu level (level+1 indicates a submenu)
                "no of space dimensions", // menu command/name
                "nsd",          // command line option: +nsd
                "",
                "2",            // default answer (2D problem)
                "I1");          // valid answer: 1 integer
```

[2] you will find the code in MultiGrid1/MultiGrid1.C

```
  menu.addItem (level,          // menu level (level+1 indicates a submenu)
                "element type", // menu item command/name
                "elm_tp",       // command line option (+elm_tp here)
                "classname in ElmDef hierarchy",
                "ElmB4n2D",     // default answer
                // valid answers are the classnames in the ElmDef hierarchy
                // where all the elements in Diffpack are defined:
                validationString(hierElmDef())); // list all the classnames

  // submenus:
  LinEqAdm::        defineStatic (menu, level+1);// linear system parameters
  prm(DDSolver)::   defineStatic (menu, level+1);// multigrid parameters
  menu. setCommandPrefix ("smoother");
  prm(LinEqSolver)::defineStatic (menu, level+1);// smoother parameters
  menu. unsetCommandPrefix ();
  FEM::             defineStatic (menu, level+1);// numerical integration rule
  Store4Plotting::  defineStatic (menu, level+1);// dumping of fields and curves
}

void MultiGrid1:: scan (MenuSystem& menu)
{
  // load answers from the menu:
  no_of_grids = menu.get ("no of grid levels").getInt();
  smooth.redim (no_of_grids);
  system.redim (no_of_grids);
  smooth_prm.redim (no_of_grids);
  proj.redim (no_of_grids-1);
  grid.redim (no_of_grids);
  dof.redim (no_of_grids);
  mat_prm.redim (no_of_grids-1);

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

  // allocate data structures in the class:
  u.rebind (new FieldFE (grid(no_of_grids)(),"u"));     // allocate, with field name "u"
  error.rebind (new FieldFE (grid(no_of_grids)(), "error"));
  int i;
  for (i=1; i<=no_of_grids; i++)
    dof(i).rebind (new DegFreeFE (grid(i)(), 1)); // 1 for 1 unknown per node
  lineq.rebind (new LinEqAdm());           // make linear system and solvers
  lineq->scan (menu);                      // determine storage and solver type
  linsol.redim (dof(no_of_grids)->getTotalNoDof()); // init length of lin.sys. solution
  lineq->attach (linsol);                  // use linsol as sol.vec. in lineq

  menu.setCommandPrefix("smoother");
  for (i=1; i<=no_of_grids; i++) { // all grids read the same
    smooth_prm(i).rebind(new prm(LinEqSolver));
    smooth_prm(i)->scan (menu);
    smooth(i).rebind(createLinEqSolver (smooth_prm(i)()));
    system(i).rebind(new LinEqSystemStd (EXTERNAL_STORAGE));
  }
  menu.unsetCommandPrefix();

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

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

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

void MultiGrid1:: scanGrids (MenuSystem& menu) // construct hierarchy of grids
{
  String elm_tp = menu.get ("element type");
  int nsd = menu.get ("no of space dimensions").getInt();

  // ---- make grid using a box preprocessor and the menu information: ----
  // construct the right syntax for the box preprocessor:
  // d=2 [0,1]x[0,1]
  // d=2 elm_tp=ElmB4n2D [2,2] [1,1]
  // this must valid for any nsd so we must make some string manipulations:
  String geometry = aform("d=%d ",nsd);  // e.g. "d=2"
  String grading = "[";                  // unit (hyper-) cube
  int i;
  for (i = 1; i <= nsd; i++) {
    if (i < nsd) {
      geometry += "[0,1]x";  grading  += "1,";
    } else {
      geometry += "[0,1]";   grading  += "1";
    }
  }
  grading += "]";

  int d = 1;
  for (i=1; i<=no_of_grids; i++) {
    d *= 2;
    int j;
    String part = "[";       // make partition strings e.g.
    for (j=1; j<=nsd; j++) { // [2,2], [4,4], [8,8], [16,16] ...
      part += aform("%d",d);
      if (j<nsd)
        part += ",";
    }
    part += "]";
    String partition = aform("d=%d elm_tp=%s div=%s grading=%s",
     nsd,elm_tp.chars(),part.chars(),
     grading.chars());
    PreproBox p;
    p.geometryBox() .scan (geometry);
    p.partitionBox().scan (partition);
    grid(i).rebind (new GridFE());  // make an empty grid
    p.generateMesh (grid(i)());
  }

  FEM::scan (menu);  // load type and order of the numerical integration rule
  Store4Plotting::scan (menu, grid(no_of_grids)->getNoSpaceDim());

  s_o << "\n **** Finite element grids: ****\n";
  s_o << " element type: " << elm_tp << "\n";
  for (i=1; i<=no_of_grids; i++)
    s_o << "\n Grid " << i << ":\tNo of nodes: " << grid(i)->getNoNodes()
      << ",\tno of elements: " << grid(i)->getNoElms();
```

```
  s_o << "\n\n";
}

void MultiGrid1:: fillEssBC (SpaceId space)
{
  dof(space)->initEssBC ();            // init for assignment below
  int nno = grid(space)->getNoNodes(); // no of nodes
  for (int i = 1; i <= nno; i++)
    if (grid(space)->BoNode (i))       // is node i subj. to any boundary indicator?
    dof(space)->fillEssBC (i, 0.0);  // u=0 at nodes on the boundary

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

void MultiGrid1:: integrands (ElmMatVec& elmat, FiniteElement& fe)
{
  int i,j,q;
  const int nbf = fe.getNoBasisFunc(); // no of nodes (or basis functions)
  const real detJxW = fe.detJxW();     // det J times numerical itg.-weight
  const int nsd = fe.getNoSpaceDim();

  // find the global coord. x of the current integration point:
  Ptv(real) x (grid(1)->getNoSpaceDim());
  fe.getGlobalEvalPt (x);
  real f_value = f(x);
  real k_value = k(x);

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

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

real analyticalSolution (const Ptv(real)& x, real /*t*/)
{
  const int nsd = x.size();
  real p = 1;
  for (int i = 1; i <= nsd; i++)
    p *= x(i) * (x(i) - 1);
  return p;
}

void MultiGrid1:: initProj() // setup proj operators
{
  for(int i=1; i<no_of_grids; i++) {
    proj(i)->rebindDOF(dof(i)(), dof(i+1)());
    proj(i)->init();
  }
}

void MultiGrid1:: initMatrices() // setup stiffness matrices on coarse grids
{
```

```
  for(int i=1; i<no_of_grids; i++) {
    fillEssBC (i);                // set essential boundary conditions
    Handle(Vec(NUMT)) u;
    Handle(Vec(NUMT)) rhs;
    u = new Vec(NUMT)  (dof(i)->getTotalNoDof ());
    rhs = new Vec(NUMT)(dof(i)->getTotalNoEqs ());

    mat_prm(i)->nrows = dof(i)->getTotalNoEqs ();
    mat_prm(i)->ncolumns = dof(i)->getTotalNoDof ();
    mat_prm(i)->nsd = dof(i)->grid().getNoSpaceDim();

    if (mat_prm(i)->storage == "MatStructSparse")
      makeSparsityPattern (mat_prm(i)->offset,
   mat_prm(i)->ndiagonals,  dof(i)());
    else if (mat_prm(i)->storage.contains("Sparse"))
      makeSparsityPattern (mat_prm(i)->sparse_adrs(), dof(i)());
    else if (mat_prm(i)->storage == "MatBand")
      mat_prm(i)->bandwidth = dof(i)->getHalfBandwidth();

    Handle(Matrix(NUMT)) A;
    A = createMatrix(NUMT) (mat_prm(i)());

    dof(i)->initAssemble();
    makeSystem (dof(i)(), A(), rhs());

    system(i)->attach(A());
    ddsolver->attachLinRhs(rhs(), i, dpTRUE);
    ddsolver->attachLinSol(u(), i);
  }
}

void MultiGrid1:: solveProblem () // main routine of class MultiGrid1
{
  initProj();
  initMatrices();

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

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

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

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

  // the solution is now in linsol, it must be copied to the u field:
  dof(no_of_grids)->vec2field (linsol, u());
```

```
  Store4Plotting::dump (u());                // dump for later visualization
  lineCurves(u());
  ErrorEstimator::errorField (analyticalSolution, u(), DUMMY, error());
  Store4Plotting::dump (error());
  ErrorEstimator::Lnorm (analyticalSolution, // supplied function (see above)
                         u(),                // numerical solution
                         DUMMY,              // point of time
                         L1_error, L2_error, Linf_error, // error norms
                         GAUSS_POINTS);      // point type for numerical integ.
}


void MultiGrid1:: resultReport ()
{
  s_o << oform("\nL1-error=%12.5e,  L2-error=%12.5e,  max-error=%12.5e\n\n",
               L1_error, L2_error, Linf_error);
  // in small problems (less than 100 nodes), print the nodal error
  // values on the file "errors.dat"
  if (grid(no_of_grids)->getNoNodes() < 300)
    error->values().print("FILE=error.dat","Nodal values of the error field");
}

real MultiGrid1:: f (const Ptv(real)& x)
{
  const int nsd = grid(1)->getNoSpaceDim();
  // could check nsd == x.size() for consistency
  int i,j; real s,p;
  s = 0;
  for (i = 1; i <= nsd; i++) {
    p = 1;
    for (j = 1; j <= nsd; j++)
      if (i != j)
        p *= x(j) * (x(j) - 1);
    s += 2*p;
  }
  return -s;
}

real MultiGrid1:: k (const Ptv(real)& /*x*/)
{ return 1; }

SpaceId MultiGrid1:: getNoOfSpaces() const
{ return no_of_grids; }

BooLean MultiGrid1:: solveSubSystem (
    LinEqVector& b, LinEqVector& x,
    SpaceId space, StartVectorMode start, DDSolverMode m)
{
  smooth_prm (space)->startmode = start;
  system     (space)->attach (x, b);
  smooth     (space)->solve  ( system (space)() );

  return dpTRUE; // solution has changed
}

void MultiGrid1:: residual (
    LinEqVector& b, LinEqVector& x, LinEqVector& r, SpaceId space)
{
  system (space)->attach   (x, b);
```

```
  system (space)->residual (r);
}

BooLean MultiGrid1:: transfer (
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode)
{
  if (fi == ti-1) // prolongation
      proj (fi)->apply (fv, tv, NOT_TRANSPOSED, add_to_t);
  else if (fi == ti+1) // restriction
      proj (ti)->apply (fv, tv, TRANSPOSED, add_to_t);
  else fatalerrorFP("MultiGrid1:: transfer","from %d to %d", fi, ti);
  return dpTRUE;
}

int MultiGrid1:: getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork) const
{
  if (fi == ti-1)
    return proj (fi)->getWork();
  if (fi == ti+1)
    return proj (ti)->getWork();
  return 0;
}

real MultiGrid1:: getStorageTransfer (SpaceId fi, SpaceId ti) const
{
  if (fi == ti-1)
    return proj (fi)->getStorage();
  return 0;
}

int MultiGrid1:: getWorkSolve (SpaceId space, const PrecondWork) const
{ return smooth (space)->getWork(); }

real MultiGrid1:: getStorageSolve (SpaceId space) const
{ return smooth(space)->getStorage(); }

String MultiGrid1:: comment ()
{ return "MultiGrid1 multigrid test"; }
```

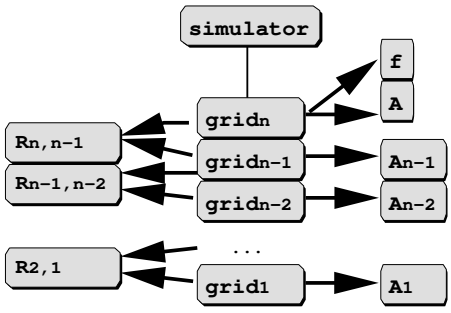

Figure 4: Initializing the multigrid components

The interesting part here is the initialization of the different components. The `GridFE`s are constructed in `scanGrids` passing different strings to `PreproBox`. There

are more advanced ways of constructing the family of grids, which will be covered elsewhere [Zum96b].

The projection operators are initialized in `initProj` using `rebindDOF`. This is a general purpose interpolation procedure. The order of the arguments is essential here. We evaluate the coarse grid functions on a finer grid, not reverse. We will see more efficient and more specialized versions of projection operators [Zum96b]. There are also applications, where the projection operators depend on the stiffness matrices on the grids or coarse grid matrices depend on the projection operators.

The operators on the coarser grids are initialized in `initMatrices()`. The fine grid matrix is assembled the standard way in `solveProblem` supported by `LinEqAdm`. The matrix is reused for the smoother on the finest grid.

```
system(no_of_grids)->attach(lineq->Al ());
```

Unfortunately it is not that easy to use the `LinEqAdm` functionality for the coarser grids, so we have to construct the matrices on our own. The `makeSystem` call actually assembles the matrices and the right hand sides. The right hand sides are only useful for the nested iteration multigrid, which we will use in some of the tests. Otherwise one can skip the assembly of `rhs`. There are also cases where we use the projection operators and the fine grid matrix instead of assembling matrices on the coarser levels, which will be covered elsewhere [Zum96b].

For a matter of completeness, we also provide an input file (table 1)[3] and the main program:

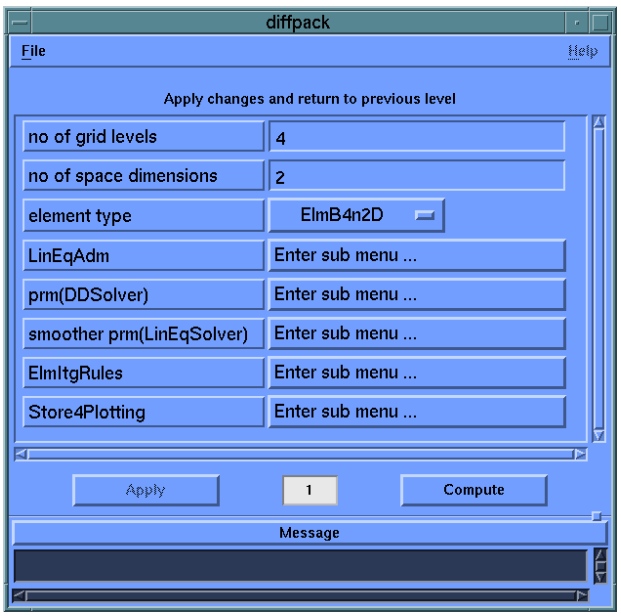

Figure 5: Main application menu

[3] complete input files are in `MultiGrid1/Verify/*.i`

menu item	answer
no of grid levels	4
no of space dimensions	2
basic method	DDIter
no of additional convergence monitors	1
preconditioning type	PrecNone
#1: convergence monitor name	CMAbsTrueResidual
#1: residual type	ORIGINAL_RES
element type	ElmB4n2D
domain decomposition method	Multigrid
cycle type gamma	1
smoother basic method	SOR
smoother max iterations	1

Table 1: Input file

MultiGrid1/
main

```
#include <MultiGrid1.h>
int main (int nargs, const char** args)
{
  initDIFFPACK (nargs, args);
  global_menu.init ("Flexible Poisson equation simulator","MultiGrid1");
  MultiGrid1 problem;                // make a simulator object, called problem
  global_menu.multipleLoop (problem);// solve one or several problems
  DBP("leaving main");
  return 0;
}
```

3.2 Number of grids and iterations

Even with this simple simulator we can do several experiments with multigrid. Just to get some feeling for different components of the algorithm, we encourage you to do some tests on your own. Playing around with parameters will be useful, if you want to apply this techniques to more advanced problems. The following exercises may be some guideline for your experiments[4].

Exercise 1 *The number of iterations.*

(table 2, `test1.i`)

Take a multigrid V-cycle with an exact coarse grid solver, one pre- and one post-smoothing step. Use an absolute residual termination criterion for some arbitrarily small tolerance. The coarse grid may consist of 2×2 elements. Now the question: What is the dependence of the number of iterations on the number of grids or the

[4] files are in `MultiGrid1/Verify/`

menu item	answer
no of grid levels	{2 & 3 & 4}

Table 2: The number of iterations, `test1.i`

number of unknowns? Do tests for a 4×4 grid (two grids), a 8×8 grid (three grids) and increase the number of unknowns further. Is the number of iterations (really) bounded? The number of iteration is displayed by the convergence monitor. What do you observe?

A remark on this exercise: A bounded number of iterations (for a fixed tolerance) means a bounded number of operations per unknown: The operations per iteration sum up to some constant times the number of unknowns. This means that we are solving the equation system for n unknowns in $\mathcal{O}(n)$ operations, which is optimal. Hence it is the ultimate goal to construct a multigrid algorithm which only needs a bounded number of iterations. For many applications it can be proven that such algorithms exist. However there are applications where an optimal algorithm is not known.

Exercise 2 *The unit cube.*

(table 3, `test2.i`)

menu item	answer
no of grid levels	{2 & 3 & 4}
no of space dimensions	3
element type	ElmB8n3D

Table 3: The unit cube, `test2.i`

Redo the computations in three dimensions. Compare the number of iterations to the two dimensional case. Are the number of iterations still bounded? Also have a look at the total execution time and the the number of unknowns. The execution times for long runs are available in file `SIMULATION.dp`. Is there a qualitative different behavior in three dimensions or do only the constants differ?

The multigrid method in general is an $\mathcal{O}(n)$ operation algorithm independent of the dimensions. However, the performance of some variants like hierarchical basis methods, deteriorate at least logarithmically in the number of unknowns n (linear in the number of multigrid levels j). The three dimensional case usually is a harder test case than the two dimensional one.

3.3 Smoother

Exercise 3 *Different smoothers.*

(table 4, `test3.i`)

menu item	answer
smoother basic method	{SOR & SSOR & Jacobi & ConjGrad}

Table 4: Different smoothers `test3.i`

Use different iterative solvers as smoothers. Try the classic methods like Gauss-Seidel (SOR) and symmetric Gauss-Seidel (SSOR) iteration, Jacobi iteration and conjugated gradients (without preconditioning). Compare the number of iterations. How does the comparison look like, if you compare execution time or number of multigrid iterations instead?

menu item	answer
(S)SOR relaxation parameter	1.0

Table 5: Relaxation parameter, `test3.i`

Caveat: The over-relaxation parameter for SOR and SSOR is set to one (table 5). The optimal parameter differs from the optimal parameter as stand-alone iterative solvers. Rather under- than over-relaxation is appropriate. You may also perform some tests on this parameter.

The performance of these iterative solvers deteriorates with increasing grid size used as stand-alone solvers. In connection with multigrid, this is not longer true. But there are of course significant differences between the smoothing procedures. We will cover this subject elsewhere [Zum96b].

In the context of preconditioning we will see that symmetric smoothers can be necessary.

Exercise 4 *The number of smoothing sweeps.*

(table 6, `test4.i`)

menu item	answer
smoother max iterations	{1 & 2 & 3 & 4}

Table 6: The number of smoothing sweeps, `test4.i`

Use a direct solver for the coarse grid, a multigrid V-cycle, SOR or Jacobi smoother, symmetric pre- and post-smoothing. Now the question: How many smoothing steps are optimal? Compare the total number of operations for the solution for different numbers of smoothing steps. Start with a $V_{1,1}$, $V_{2,2}$ and increase the number. The number of multigrid iterations will drop, but the work per iteration increases. You will soon find an optimal value. What is it?

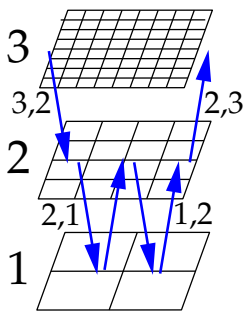

Figure 6: Multigrid W-cycle

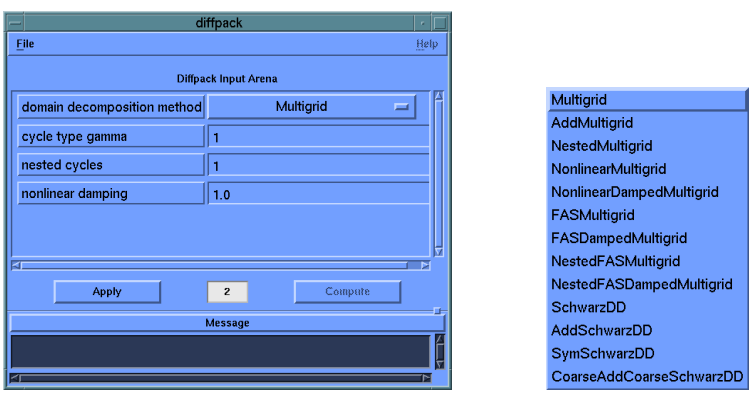

Figure 7: `DDSolver` parameter menu

3.4 W-cycle and nested iteration

We define the multigrid W-cycle and other cycles introducing a `cycle` parameter. Instead of defining multigrid Φ_j on level j by a single recursion (V-cycle) using Φ_{j-1} once, we use a multiple recursion. We apply the multigrid Φ_{j-1} on grid $j-1$ `cycle`-times. The coarser grids are visited more often. The W-cycle is `cycle` equals two. If you want to know why it is called "W" just take a look at figure 6.

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

Exercise 5 *Different multigrid cycles.*

(table 7, `test5.i`)

menu item	**answer**
cycle type gamma	{1 & 2 & 3 & 4}

Table 7: Different multigrid cycles, `test5.i`

Vary the `cycle` parameter. This means multiple recursive call of the multigrid code at each level. This number increases the complexity of the algorithm. It is usually used for more complicated grids or equations. If you really encounter convergence problems, you can try a W-cycle multigrid or parameters gamma even higher. Look at the performance in our case. How does the number of iterations behave? Compare this to the total execution time.

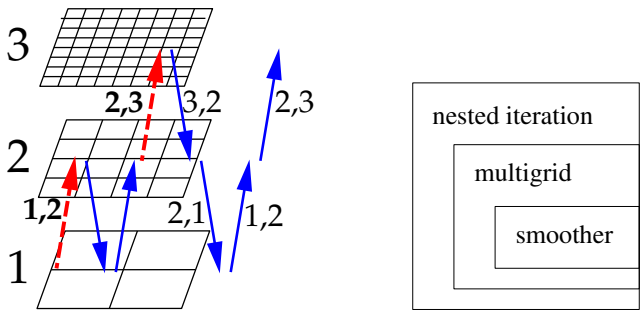

Figure 8: Nested iteration, multigrid V-cycle

Nested iteration, or full multigrid or cascadic iteration like it is called sometimes, is based on the idea that a coarse grid solution may serve as a good start guess for a fine grid iteration. The solver on the fine grid is a multigrid cycle. The coarse grid solution has been itself obtained by a multigrid iteration. This solution does not have to be that accurate. Discretization error is enough. It is also not obvious that the transport of the start solution is the same as a multigrid prolongation between the grids. In fact the procedures can differ, what we will use elsewhere (higher order interpolation) [Zum96b].

Exercise 6 *Nested iteration.*

(table 8, `test6.i`)

menu item	**answer**
domain decomposition method	NestedMultigrid
nested cycles	{1 & 2 & 3 & 4}

Table 8: Nested iteration, `test6.i`

Our first test of the nested iteration is just calling it. There is a parameter responsible for the coarse grid solution. `nested cycles` controls the number of multigrid cycles before the solution is passed to the next finer grid as a start solution. Observe the number of total iterations and the computing times. Compare these numbers to the ordinary multigrid starting with a zero vector. Can you explain why the first iteration in the convergence plot is so much better than the later ones?

Further experiments: How does the convergence plot look for one `nested cycle` and different numbers of levels? How is this related to the discretization error on each level? This resembles in the question about a suitable termination criterion of the inner loop. How could such a criterion look like and be implemented?

4 Increasing the flexibility

We want to extend the flexibility of the `MultiGrid1` example simulator. We will include options for using multigrid as a preconditioner for a Krylov iteration instead of an iterative solver, including different pre- and post-smoothing and coarse-grid operators and generating differently refined grid hierarchies.

The header declaration is extended by a few lines[5]:

MultiGrid2

```
int           preSmooth;                    // no of iterations
int           postSmooth;                   // no of iterations
prm(Precond)  precondPrm;                   // prm for DD preconditioner
```

This are parameters for constructing a preconditioner and controlling pre- and post-smoothing.

The pre- and post-smoothing control is implemented like this:

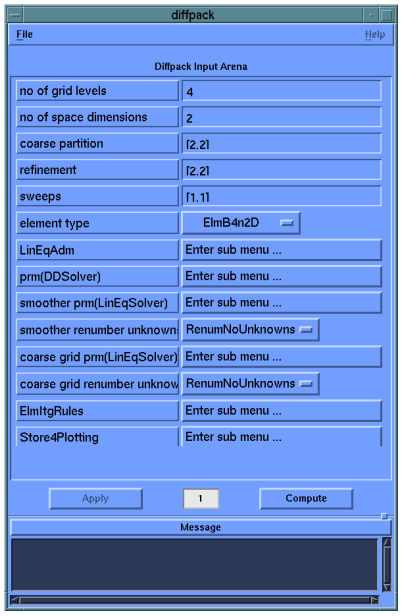

Figure 9: Main application menu

in function `MultiGrid2::  define (MenuSystem& menu, int level)`

```
//define_sweeps
menu.addItem (level,
              "sweeps",       // menu command/name
              "sweeps",       // command line options: +sweeps
              "string like [2,2] = pre & post smoothing sweeps",
```

[5] you will find the complete code in `MultiGrid2/MultiGrid2.h` and the new input files in `MultiGrid2/Verify/*.i`

```
"[1,1]",        // default answer: V1,1 cycle
"S");           // valid answer: string
```

in function `MultiGrid2:: scan (MenuSystem& menu)`

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

function `MultiGrid2:: solveSubSystem`

```
BooLean MultiGrid2:: solveSubSystem (
    LinEqVector& b, LinEqVector& x,
    SpaceId space, StartVectorMode start, DDSolverMode mode)
{
  if (space>1) {
    if ((mode==SUBSPACE_FWD)||(mode==SUBSPACE))
      smooth_prm   (space)->max_iterations = preSmooth;
    else if (mode==SUBSPACE_BACK)
      smooth_prm   (space)->max_iterations = postSmooth;
    else fatalerrorFP("MultiGrid2:: solveSubSystem","mode ", mode);
    if (smooth_prm (space)->max_iterations == 0)
      return dpFALSE; // solution has not changed
  }
  smooth_prm (space)->startmode = start;
  system (space)->attach (x, b);
  smooth (space)->solve  ( system (space)() );
  return dpTRUE; // solution has changed
}
//smoother
```

The number of iterations is read from the menu. It is passed to the smoother via the parameter `max_iterations` of `prm(LinEqSolver)`. The `DDSolverMode` is used to determine whether the pre-smoother `SUBSPACE_FWD` or the post-smoother `SUBSPACE_BACK` is required. On the coarsest grid and in the case of an additive multigrid (see section 4.5)there is only one smoother. The value `SUBSPACE` indicates this. We treat it in our implementation as a pre-smoother. In the case we do not perform pre-smoothing or post-smoothing (we have to do at least one on each level), which means zero iterations, we return `dpFALSE` to indicate that the solution has not changed. For reasons of efficiency we do not set the solution to zero, even if `StartVectorMode` indicates this, as long as we flag the result vector as untouched returning `dpFALSE`.

We also want to treat the coarsest grid solver different than the other grid solvers, using a special parameter block for `prm(LinEqSolver)`. In addition we introduce node renumbering. This is used to optimize the direct solver performance on the coarsest grid and the smoother performance:

```
#include <createRenumUnknowns.h> // renumbering grids
#include <RenumUnknowns.h> // renumbering grids
```

in function `MultiGrid2:: define (MenuSystem& menu, int level)`

```
//define_renumber
menu.setCommandPrefix("smoother");
prm(LinEqSolver)::defineStatic (menu, level+1);// smoother parameters
menu.addItem (level,
              "renumber unknowns",    // menu item command/name
              "",
              "select a renumbering algorithm",
              hierRenumUnknowns()[0], // default answer
              validationString(hierRenumUnknowns()) ); // list all classnames
menu.unsetCommandPrefix();

menu.setCommandPrefix("coarse grid");
prm(LinEqSolver)::defineStatic (menu, level+1);// coarse grid solver
menu.addItem (level,
              "renumber unknowns",    // menu item command/name
              "",
              "select a renumbering algorithm",
              *hierRenumUnknowns(),   // default answer
              validationString(hierRenumUnknowns()) ); // list all classnames

menu.unsetCommandPrefix();
//end_define_renumber
```

in function `MultiGrid2:: scanGrids (MenuSystem& menu)`

```
  //generate grids
  PreproBox p;
  p.geometryBox() .scan (geometry);
  p.partitionBox().scan (partition);
  grid(i).rebind (new GridFE());  // make an empty grid
  p.generateMesh (grid(i)());

  if (i==1)
    menu.setCommandPrefix("coarse grid");
  else
    menu.setCommandPrefix("smoother");
  String reduce = menu.get ("renumber unknowns");
  RenumUnknowns* r = createRenumUnknowns(reduce);
    r->renumberNodes (grid(i)());
  delete r;
  menu.unsetCommandPrefix();
```

The code using multigrid as a preconditioner is short:

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

in function `MultiGrid2:: scan (MenuSystem& menu)`

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


Precond &prec =lineq->getPrec();
```

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

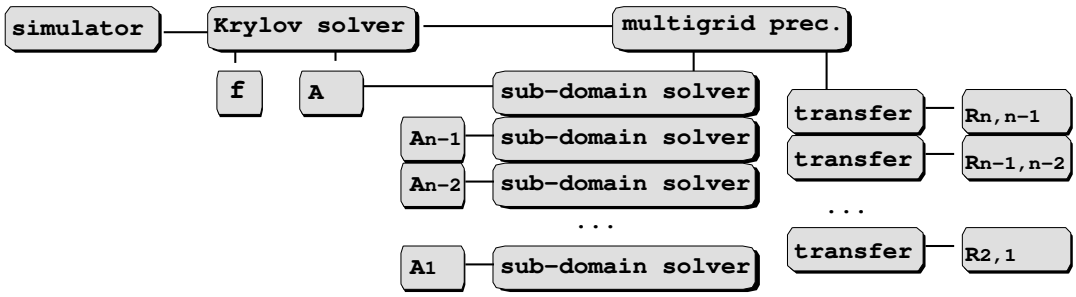

Figure 10: Components of multigrid as a preconditioner

We pass the `DDSolver` object to the `PrecDD` preconditioner. The parameters are used to have access to the preconditioner. Otherwise it would be initialized by `LinEqAdm` inside `solve`, which is too late for our purpose. The decision, whether to use a multigrid solver or a preconditioner is made via the menu, choosing `DDIter` and `PrecNone` or `ConjGrad` and `PrecDD` (see table 9).

menu item	iterative solver	preconditioner
basic method	DDIter	ConjGrad
preconditioning type	PrecNone	PrecDD

Table 9: Solver and preconditioner

Finally we want to increase flexibility in the generation of the grids. The coarse grid partition can be specified. We can have finer coarse grids than the previous $[2 \times 2]$ grid. We also allow specification of the refinement, which was just bisection along each coordinate axis, division by a factor two. We allow different factors now. Additionally a grid can be refined along one axis differently than in along another axis. The parameters are choosen like $[2, 3, 4]$ analog to the Compare the `refineIfBox(const Ptv(int)&)` function of `GridFE`.

in function `MultiGrid2:: define (MenuSystem& menu, int level)`

```
//define_partition
menu.addItem (level,
              "coarse partition",    // menu command/name
              "partition",    // command line options: +partition
              "string like 2,4,2",
              "[2,2]",        // default answer: 2x2 division (3x3 nodes)
              "S");           // valid answer: string

menu.addItem (level,
              "refinement",   // menu command/name
              "refinement",   // command line options: +refinement
```

```
                "string like [2,2,2] = bisect",
                "[2,2]",        // default answer: isotropic bisection 2x2
                "S");           // valid answer: string
  //end_define_partition
```

in function `MultiGrid2::  scanGrids (MenuSystem& menu)`

```
  //scan_partition
  Ptv(int) d(nsd);
  Is dIs(menu.get ("coarse partition"));
  dIs->ignore ('[');
  for (i = 1; i <= nsd; i++) {
    dIs->get (d(i));
    if (i < nsd)
      dIs->ignore (',');
  }

  Ptv(int) ref(nsd);
  Is rIs(menu.get ("refinement"));
  rIs->ignore ('[');
  for (i = 1; i <= nsd; i++) {
    rIs->get (ref(i));
    if (i < nsd)
      rIs->ignore (',');
  }

  for (i=1; i<=no_of_grids; i++) {
    int j;
    String part = "[";   // partition string e.g. [2,2]
    for (j=1; j<=nsd; j++) {
      part += aform("%d",d(j));
      d(j) *= ref(j);
      if (j<nsd)
part += ",";
    }
    part += "]";
    String partition = aform("d=%d elm_tp=%s div=%s grading=%s",
     nsd,elm_tp.chars(),part.chars(),
     grading.chars());
    //generate grids
    PreproBox p;
    p.geometryBox() .scan (geometry);
    p.partitionBox().scan (partition);
    grid(i).rebind (new GridFE());  // make an empty grid
    p.generateMesh (grid(i)());
```

4.1 Pre- and post-smoother

Exercise 7 *Pre- and Postsmoothing.*

(table 10, `test1.i`, `test1b.sh` in `MultiGrid2/Verify/`)

Use a direct solver for the coarse grid and a sufficiently large number of levels. Keep the sum of pre- and post-smoothing steps fixed on each level. For the Laplace operator on the unit square with structured grids take a small number of smoothing

menu item	**answer**
no of space dimensions	2
coarse partition	[2,2]
refinement	[2,2]
sweeps	{[4,0] & [3,1] & [2,2] & [1,3] & [0,4]}

Table 10: Pre- and Postsmoothing, `test1.i`, `test1b.sh`

steps ν with SOR iteration (relaxation factor 1) or Jacobi iteration. Use a multigrid V-cycle. What are the differences between different distribution of pre- and post-smoothing steps? The symbol $V_{i,j}$ denotes a V-cycle, the first number is the number of pre-smoothings and the second denotes the number of post-smoothings. Compare the number of iterations needed for a pure pre-smoothing $V_{\nu,0}$ cycle, a symmetric smoothing $V_{\nu/2,\nu/2}$ cycle, a pure post-smoothing $\ldots V_{0,\nu}$ cycle and some cycles in between.

Another question is about the influence of pre- and post-smoothing on the termination criterion. Is there a difference if you take a termination criterion that compares the solution instead of the residual?

Also have a look at the iteration error (after one iteration, see `test1b.sh`). Is there a qualitative difference of the solutions? What kind of differences do you observe?

There is another reason to choose a specific number of pre- and post-smoothings. If you have a self-adjoint operator and want to construct a symmetric preconditioner (for a conjugated gradient solver), you will have to use a $V_{\nu,\nu}$ cycle because of its symmetry. We will cover this topic later.

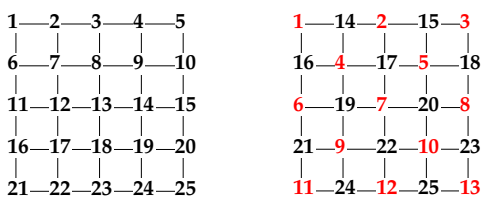

Figure 11: Lexicographic and red-black node ordering

Exercise 8 *Red-black vs. lexicographic Gauss-Seidel*

(table 11, `test5.i`)

We now want to continue our study of smoothing operators. We choose the `SOR` method, a standard multigrid V-cycle and want to look at the node ordering in conjunction with smoothers. Some smoothers like `Jacobi` or ConjGrad are independent of the node ordering. The performance of other smoothers like `SOR` and `SSOR` depends on the node ordering. During one `SOR` iteration one update of a variable depends on all previously updated variables. Reordering nodes changes these dependencies. Compare the number of iterations for the different node numberings. How large is the difference between them?

menu item	answer
element type	ElmT3n2D
smoother basic method	SOR
smoother renumber unknowns	{RenumNoUnknowns & RedBlack}

Table 11: Red-black vs. lexicographic Gauss-Seidel, `test5.i`

For structured grids and isotropic operators discretized by a 5-point stencil (= element type `ElmT3n2D`) the red-black ordering (see figure 11) is known to optimal for (S)SOR iterations. You can think of two virtually independent grids. First one iterates on the red sub-grid and afterwards on the black sub-grid. This means nearly two times the performance than a single iteration on one grid (half the number of iterations). The trick now is, that one can think of extending the red and the black grid to the global grid (= the same reduction rate), but computing only the values needed in the next step (= half the number of operations).

Caveat: The term red-black is only useful for structured grids. In the case of a 9-point stencil (= element type `ElmB4n2D`) one can use a 4-coloring of the grid instead. Nodes on unstructured grids can also be colored in a more expensive (and heuristic) approach using slightly more colors.

Exercise 9 *Red-black and prolongation.*

One can optimize the multigrid method further in the case of red-black `SOR` smoothing. Look at the prolongation and restriction operators. Usually they operate on all nodes of a grid. We already saw that now there is a difference between red and black nodes. They are updated at different times. How does this affect prolongation? Can you skip one half of the operations there? Which color?

4.2 Coarse grid solver

Exercise 10 *Coarse grid solution.*

(table 12, `test2.i`)

menu item	answer
coarse partition	[8,8]
sweeps	[1,1]
coarse grid basic method	{SOR & ConjGrad & GaussElim}
coarse grid max iterations	{1 & 10 }

Table 12: Coarse grid solution, `test2.i`

Take a coarse grid with several unknowns. Fix some V-cycle multigrid algorithm. Now use an iterative coarse grid solver. How many iterations are optimal for the

coarse grid solution? Increasing the number of iterations will improve global convergence and reduces the number of global iterations, while the cost per iteration also increases. There will be some optimal value. Do experiments with a relative and an absolute termination criterion for the coarse grid iteration. Compare the total number of iterations. What is your conclusion?

Exercise 11 *Direct coarse grid solution.*

(table 13, `test3.i`)

menu item	**answer**
matrix type	MatSparse
coarse partition	[16,16]
coarse grid basic method	GaussElim
coarse grid renumber unknowns	{RenumNoUnknowns & AMDhat & AMDbar}

Table 13: Direct coarse grid solution, `test3.i`

Take a coarse grid with several unknowns and use a direct Gaussian elimination solver for sparse matrices. The performance of the solver depends on the order of the nodes on the coarse grid. The total multigrid performance should not be affected except for the total execution time. Compare the number of operations for the different node orderings and compare the total execution time. How do the node orderings affect the execution time? Do they pay off?

The reordering of the nodes on the coarse grid itself can be a time consuming task. However the coarse grid is visited often, so even a more expensive reordering method may be used here. The question is of course of major importance for large coarse grids, which may be needed due to the geometry of the domain or the structure of the coefficients. In the unit square case the coarse grid is usually not that important, as one can choose very coarse grids and the optimal node ordering is known a priori (some nested dissection ordering scheme).

Exercise 12 *The number of levels.*

(table 14, `test4.i`)

menu item	**answer**
no of grid levels	{2 & 3 & 4 & 5}
coarse partition	{[2,2] & [4,4] & [8,8] & [16,16]}

Table 14: Direct coarse grid solution, `test4.i`

We now look at a $V_{1,1}$ multigrid cycle with a direct coarse grid solver. Take some band matrix or sparse matrix data structure and apply a direct solver. Take care

about a good node ordering suited for your direct solver. Fix the number of unknowns on the finest grid. The question is, what is the optimal size of the coarse grid? A large coarse grid improves convergence and reduces the number of iteration, but is expensive to solve. Compare the total number of operations to solve the system up to an arbitrary small precision. How large is optimal coarse grid? Can you guess a simple formula for its size?

This formula depends on the performance of the coarse grid solver and therefore will be different for three dimensional problems.

4.3 Semi-coarsening and non-standard refinement

Exercise 13 *Semi-coarsening*

(table 15, `test6.i`)

menu item	**answer**
coarse partition	{[16,2] & [2,16]}
refinement	{[1,2] & [2,1]}

Table 15: Semi-coarsening, `test6.i`

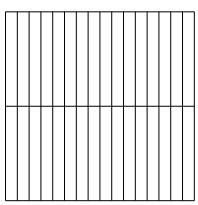

Figure 12: Anisotropic grid, derived by semi-coarsening

We now have a look at different ways of grid refining. One way is anisotropic refinement. This means refinement along different directions may differ. In two dimensions this can be bisection along the x-axis and no refinement along the y-axis or vice versa. We use a standard multigrid $V_{1,1}$ cycle and a coarse grid solver. We fix the finest grid which is an isotropic grid (just standard). This means that the coarser grids become anisotropic (figure 12). Compare the performance (the number of iterations) for different anisotropic refinements. Is there a difference to standard isotropic $[2, 2]$ refinement?

The direct coarse grid solver is essential. What happens if the coarse grid solution is just one SOR-cycle? How do the iterates deterioate?

Anisotropic refinement is not necessary in this example, but it may be useful for advection-diffusion problems. The advection term dominates on the coarse grids, so anisotropic coarse grids stretched along the direction of advection stabilize (improve) discretization. Sometimes there are also constraints by geometry or coefficients, where anisotropic grids are the only way to geometrically construct coarse grids. Think e.g. of the flow in a long channel. This will be covered elsewhere [Zum96b].

Exercise 14 *Non-bisecting refinement*

(table 16, `test7.i`)

menu item	**answer**
no of grid levels	{2 & 3}
coarse grid basic method	GaussElim
refinement	{[2,2] & [4,4]}

Table 16: Non-bisecting refinement, `test7.i`

Up to now we only have considered refinement via bisection which means a factor of two. The grids and spaces were nested, the coarser space a subset of the finer space. This means an integer refinement factor. Using a factor greater than 2 decreases the number of intermediate levels. We choose a standard $V_{1,1}$ multigrid cycle and fix the coarsest and the finest grid. We vary the refinement factor and the number of intermediate levels. Observe the convergence rates and the number of multigrid iterations. What is the effect of higher refinement factors?

If the multigrid performance degrades too much, we loose the optimal complexity of the multigrid algorithm. Hence improvement of the smoother (increasing the number of iterations) may be appropriate. Try a larger number of SOR smoothing steps.

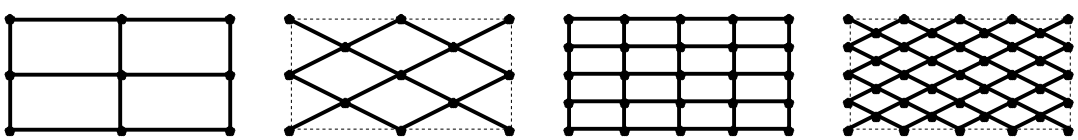

Figure 13: rotated grids with $\sqrt{2}$ progression

Usually higher refinement factors are not used although some two-level domain decomposition methods can be interpreted as such. There are also some lower refinement factors like $\sqrt{2}$ via rotated grids (figure 13). We will cover non-nesting refinement elsewhere [Zum96b], where the refinement factors can be arbitrary.

4.4 Multigrid as a preconditioner

We now want to use multigrid as a preconditioner instead of a stand-alone iterative solver. We choose the conjugated gradient algorithm to solve the Laplace equation. The number of iterations needed depends on the condition number of the preconditioned operator.

$$\kappa := \frac{\lambda_{\max}(B\ \mathcal{L})}{\lambda_{\min}(B\ \mathcal{L})}$$

with eigenvalues λ, stiffness matrix $\mathcal{L}$ and a preconditioner B. The convergence rate is bounded by

$$\rho = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}$$

The condition number for a second order problem grows with $\mathcal{O}(h^{-2})$ with a characteristic element size (mesh size) h. This means that the number of iterations grows with a decrease of h and an increase of the number of unknowns. We already saw that a multigrid iteration can reach optimal complexity i.e. there is an upper limit for the number of iterations. The convergence rate of a single multigrid cycle is bounded below 1. In the case of preconditioning this means that $\kappa \leq c$ is limited independent of the mesh size h.

The conjugated gradient algorithm requires a $\mathcal{L}$-symmetric preconditioner. This means

$$\mathcal{L} \cdot B = B \cdot \mathcal{L}$$

For a multigrid cycle this is equivalent to: The pre- ($\mathcal{S}^1$) and the post-smoothing ($\mathcal{S}^2$) is adjoint

$$\mathcal{S}^1 = \mathcal{S}^{2^*}$$

and the restriction and prolongation are also adjoint $R_{j,j-1} = R^*_{j-1,j}$. The number of pre- and post-smoothing has to equal. For example take a $V_{1,1}$ cycle or a $V_{2,2}$ cycle.

- One way to match the condition is to take a self-adjoint smoother: $\mathcal{S} = \mathcal{S}^*$ like Jacobi iteration or symmetric Gauss-Seidel iteration (SSOR).
- Another way to do it, is to use an unsymmetric smoother as a pre-smoother and its adjoint as a post-smoother: Take a Gauss-Seidel iteration (SOR) or a (R)ILU iteration with a node ordering $1, 2, \ldots, n$ as pre-smoother and the same method with a node ordering $n, n-1, \ldots, 1$ as post-smoother.
- One alternative is also to use an additive multigrid (see section 4.5) with a self-adjoint smoother: $\mathcal{S} = \mathcal{S}^*$ like Jacobi iteration or symmetric Gauss-Seidel iteration (SSOR).

Exercise 15 *The smoother.*

(table 17, `prec1.i`)

menu item	**answer**
basic method	ConjGrad
preconditioning type	PrecDD
smoother basic method	{SOR & SSOR & Jacobi & ConjGrad}

Table 17: The smoother, `prec1.i`

We now compare the performance of different smoothers. We use a standard $V_{1,1}$ multigrid cycle as a preconditioner for a conjugated gradient solver. We apply Gauss-Seidel (S)SOR with over-relaxation parameter one and Jacobi smoothers. We also apply a conjugated gradient method without further precondition as a smoother. Compare the number of multigrid iterations/ reduction rates and the computing times. Especially compare the symmetric ones with the non-symmetric SOR smoother. How does the non-symmetry influence the global convergence? How does the (outer) conjugate gradient algorithm improve the multigrid convergence?

Exercise 16 *Iterative solver vs. preconditioner.*

Using the same data we are now also able to compare the performance of multigrid as a preconditioner (exercise 15) and multigrid as an iterative solver (exercise 3). Compare the number of iterations and the global computing time. How does the number of iterations change? Is the influence of the additional vector operations of the conjugated gradients algorithm visible in the computing time? Do you know how much memory the conjugated gradients algorithm additionally needs? Compare this to the global amount of memory used.

The findings of this exercise are problem specific. Krylov methods like conjugated gradients are specifically well suited for problems having only few eigenvalues which can be eliminated in a few iterations in addition to the standard (narrow) spectrum of eigenvalues. This may be the case for some jumping coefficient problems or specific geometries. Often this is related to the issue of multiple eigenvalues. The conjugated gradient method eliminates extreme eigenvalues independent of their multiplicity. Hence problems with identical extreme eigenvalues can be solved quickly. However in the case the eigenvalues do not coincide (numerically) but differ slightly, conjugate gradients will need several iterations more and slow down.

Exercise 17 *Non-symmetric preconditioner.*

(table 18, `prec2.i`)

menu item	**answer**
sweeps	[2,0]
smoother basic method	SOR
basic method	{BiCGStab & ConjGrad & CGS}
preconditioning type	PrecDD

Table 18: Non-symmetric preconditioner, `prec2.i`

We now turn to non-symmetric Krylov methods. Solvers like `BiCGStab`, `CGS` or `GMRes` are designed for the solution of non-symmetric problems. A preconditioner is usually not symmetric and does not have to be in this context. Although we only have a symmetric problem at hand, we will compare solvers with a non-symmetric preconditioner. We choose a non-symmetric $V_{2,0}$ multigrid preconditioner and run the different solvers.

Observe the divergence of the conjugated gradient method. This is rather a negative example. Look at the convergence plot. Whenever you see such a pattern remember to be careful about such issues like symmetry.

Compare the number of iterations and the computing time needed by the other solvers. Try to find out how many matrix multiplications and calls of the preconditioner are performed each iterations. Compare the methods based on some rough work estimates.

When we will turn to advection-diffusion problems we will have to discuss this issue again. We will cover this elsewhere [Zum96b].

4.5 Additive Preconditioner

We will now have a short look at a variant of the multigrid V cycle preconditioner. Some additive versions with Jacobi smoothers originally proposed by [BPX90] called BPX or "multilevel diagonal scaling" (MDS). It played an important role for the proof of optimal complexity of multigrid. The interpretation as additive multigrid was found later.

algorithm

We start with an initial guess $x = 0$. Hence the residual $b - \mathcal{L}x$ equals b.

$$\Phi_j(b) \quad = \quad \mathcal{S}(b) + R_{j-1,j}\Phi_{j-1}(R_{j,j-1}b)$$

coarse grid

On level 1 we use a direct coarse grid solver

$$\Phi_1(b) = \mathcal{L}^{-1}b$$

The idea is to run the corrections on the different grid levels independently. The corrections on each level are independent for each unknown using a Jacobi smoother $\mathcal{S}$. The independence of lots of operations may serve as a source of parallelism although the grid transfer operations, the restrictions and prolongations still are serial operations. They can be broken up into independent operations splitting the computational domain. The additive method usually will converge slower as the multiplicative method but each step is better suited for parallel compution.

Exercise 18 *Additive vs. multiplicative preconditioner.*

(table 19, `prec3.i`)

menu item	**answer**
sweeps	[1,1]
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	{Multigrid & AddMultigrid & NestedMultigrid}

Table 19: Additive vs. multiplicative preconditioner, `prec3.i`

We now want to compare additive and multiplicative multigrid. We do not consider the parallel aspects of both algorithms so the result does not take into account the independence of some operations. We choose a conjugated gradient outer iteration and precondition by a multigrid $V_{1,1}$ respectively a V_1 cycle. We use standard multiplicative multigrid, additive multigrid and nested iteration multiplicative multigrid. Compare the number of operations and the computing time needed.

Compare an additive V_2 cycle instead using the same number of smoothing steps on each level as the multiplicative version. How do the convergence rates/ number of iterations now compare?

Comparing the nested iteration for multiplicative multigrid, we have to mention that there is also a nested iteration for the conjugated gradient method. The concept of nested iteration is in fact not tied to any specific iterative solver at all. We will treat this elsewhere [Zum96b]. This nested iteration can utilize any kind of preconditioner like additive or multiplicative multigrid, but it needs more administration in the outer loop and some extensions of the code. We also refer to section 5.2.

5 Nonlinear problems

We first recall the usage of nonlinear solvers in `Diffpack`. Based on the class `NonLinEqSolver` there are several algorithms available, such as successive substitution (or Picard iteration), Newton's method (or Newton-Raphson) iteration and nonlinear conjugated gradients.

5.1 `Diffpack` nonlinear interface

The problem is, that the user interface of the nonlinear methods is different than the linear solvers. In the linear case we used the `LinEqAdm` class to manage nearly all memory allocation and initialization tasks. In the nonlinear case we have to keep track of the data ourselves. This is due to the fact that we have to update the stiffness matrix or the right hand side frequently, data shared between application code and nonlinear solver. We also have to provide a linear equation solver ourselves. This is accomplished filling in a `NonLinEqSolverUDC` interface. Since this is only documented for time dependent problems up to now (chapter 5 in [Lan94]), we include here the code for the stationary case. We will use it as a starting point for the development of nonlinear multigrid.[6]

NlElliptic

```
#ifndef NlElliptic_h_IS_INCLUDED
#define NlElliptic_h_IS_INCLUDED

#include <FEM.h>                // FEM algorithms, FieldFE, GridFE etc
#include <DegFreeFE.h>          // mapping: nodal values -> linear system vec
#include <LinEqAdm.h>           // linear systems, storage and solution
#include <NonLinEqSolverUDC.h>  // user's class interface to nonlinear solvers
#include <NonLinEqSolver_prm.h> // parameters for nonlinear solvers
#include <NonLinEqSolver.h>     // interface to nonlinear solvers
#include <Store4Plotting.h>
class NlElliptic : public FEM, public NonLinEqSolverUDC,
                public MenuUDC, public Store4Plotting
{
protected:
  // general data:
  Handle(GridFE) grid;     // finite element grid
  Handle(DegFreeFE) dof;   // mapping: nodal values <-> linear system unknowns
  Handle(FieldFE) u;       // finite element field, the primary unknown

  Vec(real)              nonlin_solution;      // nonlinear solution
```

[6] you will find the complete code in `NlElliptic/`

```
  Vec(real)              linear_solution;      // solution of linear subsystem
  prm(NonLinEqSolver)    nlsolver_prm;         // parameters for solver
  Handle(NonLinEqSolver) nlsolver;             // nonlinear solver

  Handle(LinEqAdm) lineq;  // linear system, storage and solution

  virtual void fillEssBC();
  virtual void integrands (ElmMatVec& elmat, FiniteElement& fe);
  virtual void makeAndSolveLinearSystem ();
  virtual real f (const Ptv(real)& x, real u); // nonlinear source term
  virtual real k (const Ptv(real)& x, real u); // nonlinear coefficient
                          // needed in Newton Raphson iterations
  virtual real df(const Ptv(real)& x, real u); // d/du of nonlinear source term
  virtual real dk(const Ptv(real)& x, real u); // d/du of nonlinear coefficient
public:
  NlElliptic ();
 ~NlElliptic () {}

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

class NlElliptic2 : public NlElliptic
{
protected: // nonlinear rhs
  virtual real f (const Ptv(real)& x, real u); // nonlinear source term
  virtual real df(const Ptv(real)& x, real u); // d/du of nonlinear source term
public:
  NlElliptic2 () {}
 ~NlElliptic2 () {}
};

class NlElliptic3 : public NlElliptic
{
protected: // nonlinear operator
  virtual real k (const Ptv(real)& x, real u); // nonlinear coefficient
  virtual real dk(const Ptv(real)& x, real u); // d/du of nonlinear coefficient
public:
  NlElliptic3 () {}
 ~NlElliptic3 () {}
};
#endif
```

We define three different test cases: (1) Problem `NlElliptic` is a linear, (2) problem `NlMultiGrid2` has a nonlinear right hand side and (3) problem `NlMultiGrid3` has a nonlinar operator. We both implement successive substitution and Newton's method. For Newton's method we provide the derivatives `df` and `dk`.

$$
\begin{array}{rrcll}
1) & -\nabla\cdot\nabla u & = & 1 & \text{on } \Omega \\
2) & -\nabla\cdot\nabla u & = & e^u & \text{on } \Omega \\
3) & -\nabla\cdot e^u\nabla u & = & 1 & \text{on } \Omega \\
 & u & = & 0 & \text{on } \partial\Omega
\end{array}
\tag{1}
$$

NlElliptic/
main

```
#include <NlElliptic.h>
int main (int nargs, const char** args)
{
  initDIFFPACK (nargs, args);
  global_menu.init ("Nonlinear test case", "NlElliptic");
  int p = 1;
  initFromCommandLineArg("-case", p, p /* default is linear, p=1 */);
  NlElliptic *problem;
  switch (p) {
  case 1: problem = new NlElliptic();
    break;
  case 2: problem = new NlElliptic2();
    break;
  case 3: problem = new NlElliptic3();
    break;
  }
  global_menu.multipleLoop (*problem);
  delete problem;
  return 0;
}
```

In the main program the user is able to choose between the three different problems. The nonlinear elliptic scalar problem is solved on the unit square. The `integrand` is both suited for succesive substitution and Newton's method.

NlElliptic.C

```
#include <NlElliptic.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <readOrMakeGrid.h>
#include <createNonLinEqSolver.h>
#include <ErrorEstimator.h>
NlElliptic:: NlElliptic () {}

void NlElliptic:: adm (MenuSystem& 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 NlElliptic:: define (MenuSystem& menu, int level)
{
  menu.addItem (level,
                "gridfile",
                "gridfile",
                "filename or PREPROCESSOR=PreproSupElSet/geometry/partition",
                "PREPROCESSOR=PreproBox/d=2 [0,1]x[0,1]/d=2 elm_tp=ElmB4n2D\
                 div=[5,5], grading=[1,1]",
                "S");
  // submenus:
  LinEqAdm::          defineStatic (menu, level+1);
```

```
  prm(NonLinEqSolver)::defineStatic (menu, level+1);
  Store4Plotting::     defineStatic (menu, level+1);
  FEM::                defineStatic (menu, level+1);
}

void NlElliptic:: scan (MenuSystem& menu)
{
  String gridfile = menu.get ("gridfile");
  grid.rebind (new GridFE());              // create empty grid object
  readOrMakeGrid (grid(), gridfile);       // fill grid
  Store4Plotting::scan (menu, grid->getNoSpaceDim());
  FEM::scan (menu);

  lineq.rebind (new LinEqAdm());
  lineq->scan (menu);
  u.rebind (new FieldFE (grid(),"u"));            // allocate space for u
  dof.rebind (new DegFreeFE (grid(), 1));         // 1 unknown per node

  nlsolver_prm.scan (menu);
  nonlin_solution.redim (dof->getTotalNoDof()); // size = total no of unknowns
  linear_solution.redim (dof->getTotalNoDof());
  lineq->attach (linear_solution);
  nlsolver.rebind (createNonLinEqSolver (nlsolver_prm));
  nlsolver->attachUserCode (*this);
  nlsolver->attachNonLinSol (nonlin_solution);
  nlsolver->attachLinSol (linear_solution);
}

void NlElliptic:: fillEssBC ()
{
  dof->initEssBC ();           // init for assignment below
  const int nno = grid->getNoNodes();

  for (int i = 1; i <= nno; i++)
    if (grid->BoNode (i))      // is node i subjected any boundary indicator?
      dof->fillEssBC (i, 0.0); // homogeneous Dirichlet.
}

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

  const real u_pt = u->valueFEM (fe);          // U (at present itg. point)

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

  real nabla1,nabla2,h;

  if (nlsolver->getCurrentState().method == NEWTON_RAPHSON)
    {
      Ptv(real) Du_pt (nsd);                       // grad U
      u->derivativeFEM (Du_pt, fe);                // interpolate Du_pt
```

```
      real df_value = df(x, u_pt);
      real dk_value = dk(x, u_pt);

      for (i = 1; i <= nbf; i++) {
        nabla1 = 0;
        for (s = 1; s <= nsd; s++) {
          nabla1 += fe.dN(i,s)*Du_pt(s);
        }
        for (j = 1; j <= nbf; j++) {
          nabla2 = 0;
          for (s = 1; s <= nsd; s++)
            nabla2 += fe.dN(i,s)*fe.dN(j,s);
          h = k_value*nabla2 + dk_value*fe.N(j)*nabla1 -
              df_value*fe.N(i)*fe.N(j);
          elmat.A(i,j) += h*detJxW;
        }
        h = k_value*nabla1 - f_value*fe.N(i);
        elmat.b(i) -= h*detJxW;
      }
    }
  else if (nlsolver->getCurrentState().method == SUCCESSIVE_SUBST)
    {
      for (i = 1; i <= nbf; i++) {
        for (j = 1; j <= nbf; j++) {
          nabla2 = 0;
          for (s = 1; s <= nsd; s++)
            nabla2 += fe.dN(i,s)*fe.dN(j,s);
          elmat.A(i,j) += k_value*nabla2*detJxW;
        }
        elmat.b(i) += fe.N(i)*f_value*detJxW;
      }
    }
  else
    errorFP("NlElliptic::integrands",
            "Linear subsystem for the nonlinear method %s is not implemented",
            getEnumValue(nlsolver->getCurrentState().method).chars());
    // getEnumValue: returns a string of the enum, .chars() transforms the
    // string to a const char* that can be fed into the printf-like errorFP
}

void NlElliptic:: makeAndSolveLinearSystem ()
{
  dof->vec2field (nonlin_solution, u());  // copy most recent guess to u

  if (nlsolver->getCurrentState().method == NEWTON_RAPHSON)
    dof->fillEssBC2zero();  // ensure no correction of known values!
  else
    dof->unfillEssBC2zero();// (set back to) normal treatment of ess. b.c.

  makeSystem (dof(), lineq());

  // init startvector (linear_solution) for iterative solver:
  if (nlsolver->getCurrentState().method == NEWTON_RAPHSON)
    // start for a correction vector (should -> 0)
    linear_solution.fill (0.0);
  else
    //  use most recent nonlinear solution
    linear_solution = nonlin_solution;
```

```
  lineq->solve();  // invoke a linear system solver
}

void NlElliptic:: solveProblem () // main routine of class NlElliptic
{
  fillEssBC ();                       // set essential boundary condition
  nonlin_solution.fill (1.0);         // set all entries to 1 in start vector
  dof->fillEssBC (nonlin_solution);

  // call nonlinear solver:
  if (!nlsolver->solve ())
    errorFP("NlElliptic::solve");
  // load nonlinear solution found by the solver into the u field:
  s_o<<"maximum = "<<nonlin_solution.norm(Linf)<<endl;

  dof->vec2field (nonlin_solution, u());
  Store4Plotting::dump (u());                // dump for later visualization
  lineCurves(u());
}

void NlElliptic:: resultReport ()
{
  // in small problems (less than 100 nodes), print the nodal error
  // values on the file "errors.dat"
  if (grid->getNoNodes() < 100)
    u->values().print("FILE=u.dat","Nodal values of the solution field");
}

real NlElliptic:: f (const Ptv(real)&, real) { return 1.;}
real NlElliptic:: df(const Ptv(real)&, real) { return 0.;}

real NlElliptic:: k (const Ptv(real)&, real) { return 1.;}
real NlElliptic:: dk(const Ptv(real)&, real) { return 0.;}

real NlElliptic2:: f (const Ptv(real)&, real u_) { return exp(u_);}
real NlElliptic2:: df(const Ptv(real)&, real u_) { return exp(u_);}

real NlElliptic3:: k (const Ptv(real)&, real u_) { return exp(u_);}
real NlElliptic3:: dk(const Ptv(real)&, real u_) { return exp(u_);}
```

Exercise 19 *Newton and successive substitution.*[7]

(table 20, `test1.i`, `test1.sh`)

We do a first exercise with nonlinear solvers comparing Newton's method and succesive substitution. This is comparing an asymptotic quadratic convergent iteration with a linear convergent one. Applied to the linear problem, the result should be the same: Just one iteration. We now look at the two nonlinear problems: Compare the number of iterations and the computing time. Look at the convergence rates of the Newton iteration. When do we obtain quadratic convergence? How do the first steps look like? Do the comparisons for different tolerances. When is the succesive substitution faster and at what tolerances is Newton iteration faster?

[7] files are in `NlElliptic/Verify/`

menu item	answer
gridfile	PREPROCESSOR=PreproBox/ d=2 [0,1]x[0,1]/ d=2 e=ElmB4n2D div=[8,8] g=[1,1]
nonlinear iteration method	{NewtonRaphson & SuccessiveSubst}
max estimated nonlinear error	1.0e-10
nonlinear iteration stopping criterion	1
basic method	GaussElim

Table 20: Newton and successive substitution, `test1.i`

In the case Newton's iteration does not converge, a damping strategy can be employed at the first steps. Convergence is enforced via smaller corrections as long as an attractor is reached. Damping strategies usually use some additional assumptions and heuristics.

damped Newton

Especially for large equations sytems the computation of the Jacobian matrix may be quite expensive. A general modification often used is an approximative Jacobian matrix. Such approximations may be computed via numerical differentiation or some updates of older Jacobians.

inexact Newton

5.2 Inexact solver

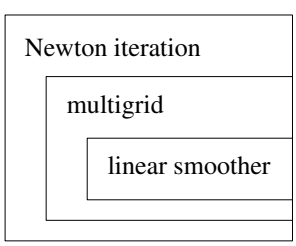

Figure 14: Newton-multilevel

One way of combining multigrid and nonlinear solvers is to use linear multigrid inside a nonlinear solver. We use successive substitution or Newton's method and solve the linear systems by multigrid either exactly or only approximatly using one cycle. This method is called Newton-Multilevel. The difference between the Newton iteration and successive substitution is roughly, that we expect the Newton method to converge quadratically in the vicinity of a solution, while the substitution method only converges linearly. This resembles in the question to find appropriate starting vectors close enough to the solution and a desired precision small enough to really achieve quadratic convergence.

Newton-multilevel

Each iterate of the Newton method with exact Jacobian is also more expensive to compute than succesive substitution. The substitution method neglects the derivatives instead.

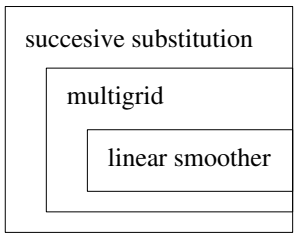

Figure 15: Successive substitution-multilevel

The question whether to use only one multigrid cycle or more to solve up to a certain tolerance is a trade off between the cost of a nonlinear matrix update and the cost of a linear multigrid cycle. This of course strongly depends on the problem we want to solve. If one multigrid cycle is not precise enough, we certainly will loose the quadratic convergence property of the Newton method, obtaining only linear convergence. This may be circumvented using a thorough termination criterion for the inner multigrid iteration being precise enough.

Exercise 20 *Inexact linear solvers.*[8]

(table 21, `test2.i`, `test2.sh`)

menu item	**answer**
gridfile	PREPROCESSOR=PreproBox/ d=2 [0,1]x[0,1]/ d=2 e=ElmB4n2D div=[8,8] g=[1,1]
nonlinear iteration method	{NewtonRaphson & SuccessiveSubst}
basic method	ConjGrad
#1: convergence monitor name	CMRelResidual
#1: max error	1.0e-2

Table 21: Inexact linear solvers, `test2.i`

A second exercise is some kind of preparation for the use of multigrid as a linear solver inside a nonlinear solver. The nonlinear solver is designed to operate with exact solutions of the given linear problems. Solving the linear problems iteratively of course means disturbing the outer nonlinear loop. We start experimenting with some convergence criteria and have a look whether convergence is still given. We compare the total ammount of work or the computing times. Choose a relative and an absolute residual based criterion for an inner conjugated gradient linear solver. The outer loop may be Newton's iteration and successive substitution. Fix the tolerance for the outer iteration. Compare different tolerances for the inner iteration. When do we have a convergent method and what is the optimal tolerance, always measuring computational work? What criterion and what nonlinear iteration is preferable? What is the relation between inner loop and outer loop tolerance?

[8] files are in `Ml`Elliptic/Verify/

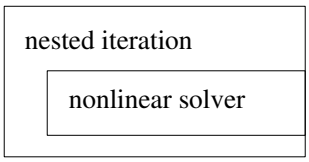

Figure 16: Nested iteration

The question of a good start vector for nonlinear solvers is crucial. It is a question if the algorithm will converge and which will be the result if there are several solutions of the nonlinear equation. In the context of multigrid methods there is the idea of nested iteration constructing start vectors (or cascadic iteration). Like in chapter 3.4 we use the solution computed on one grid as a start solution for the iteration on the next finer grid. We repeat that there are benefits of higher order interpolation here. In the case of a Newton iteration (or any other nonlinear solver), we start the iteration on one level with an interpolation of a solution on the next coarser grid. This idea is independent on the use of multigrid inside this nonlinear solver. However the grid hierarchy can be reused efficiently. This is called multilevel-Newton.

multilevel-Newton

Combining both methods, we end up with a multilevel-Newton-multilevel type algorithm. The decision, which of several solutions the iteration is converging to, may be taken on the coarsest grid. If that solution is close enough to the continuous solution, and the chain of nested spaces are "close" enough, the solution is then determined and convergence is given. This condition is not precise at all. In practice it would be difficult to modify the hierarchy of grids just to ensure convergence of the nonlinear method. We also want to mention, that the number of solutions on the coarse grids may be misleading. Not all of them have to be present (or separate) on the coarsest grid. Solutions may develop (or fork off) at a certain discretization level.

exercise

We leave the implementation of the (multilevel-)Newton-multilevel to the reader. Combine the examples `MultiGrid1` and `NlElliptic`. Substitute the use of `LinEqAdm` in `NlElliptic::MakeAndSolveSystem` by a call of a multigrid solver.

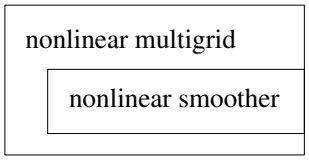

Figure 17: Nonlinear multigrid

5.3 Nonlinear multigrid

nonlinear multigrid

Instead we turn to another way of using multigrid methods for nonlinear problems: Nonlinear multigrid. The idea is to use the multigrid cycle as the outer loop and dealing with the nonlinearity inside the multigrid method. Since restriction and prolongation operators are not affected by the nonlinearity of the problem, we only have to deal with smoothers for nonlinear problems. The computation of a residual

is just the evaluation of the (nonlinear) operator. As smoothers we can use standard nonlinear solvers, like nonlinear versions of SOR, Jacobi, conjugated gradients and so on.

The problem now is that we have to decide where to evaluate the operator for all these operations. In the linear case this did not matter. On the finest grid this is clear: We have to use the latest version of the nonlinear solution. However on coarser grids, it is not longer clear, what the latest nonlinear solution really is. That is why there are several nonlinear multigrid versions around. They differ exactly in that detail. One idea is to use the restriction of the latest version of the fine grid (FAS, full approximation scheme) and another is to use the best solution available during a nested iteration setup. A third version uses linear combinations of both.[9]

NlMultiGrid1

```
#ifndef NlMultiGrid1_h_IS_INCLUDED
#define NlMultiGrid1_h_IS_INCLUDED

#include <FEM.h>                // FEM algorithms, FieldFE, GridFE etc
#include <DegFreeFE.h>          // mapping: nodal values -> linear system vec
#include <LinEqAdm.h>           // linear systems, storage and solution
#include <NonLinEqSolverUDC.h>  // user's class interface to nonlinear solvers
#include <NonLinEqSolver_prm.h> // parameters for nonlinear solvers
#include <NonLinEqSolver.h>     // interface to nonlinear solvers
#include <Store4Plotting.h>
#include <DDSolver.h>           // DDSolver
#include <DDSolverUDC.h>        // interfacing to DDSolver
#include <DDSolver_prm.h>       // DDSolver parameters
#include <VecSimplest_Handle.h>
class NlLevel : public FEM, public NonLinEqSolverUDC, public virtual HandleId
{
protected:
  // general data:
  Handle(GridFE) grid;     // finite element grid
  Handle(FieldFE) u;       // finite element field, the primary unknown
  Handle(Proj) proj;       // projection operators
  Handle(DegFreeFE) dof, dof2; // mapping: nodal values <-> linear system unknowns

  Handle(Vec(NUMT))      nonlin_solution;      // nonlinear solution
  Vec(NUMT)              linear_solution;      // solution of linear subsystem
  Handle(LinEqVector)    linear_rhs;           // rhs of linear subsystem
  prm(NonLinEqSolver)    nlsolver_prm;         // parameters for solver
  Handle(NonLinEqSolver) nlsolver;             // nonlinear solver

  Handle(LinEqAdm) lineq;  // linear system, storage and solution

  virtual void fillEssBC();
  virtual void integrands (ElmMatVec& elmat, FiniteElement& fe);
  virtual void makeAndSolveLinearSystem ();
  virtual real f (const Ptv(real)& x, real u); // nonlinear source term
  virtual real k (const Ptv(real)& x, real u); // nonlinear coefficient
                           // needed in Newton Raphson iterations
  virtual real df(const Ptv(real)& x, real u); // d/du of nonlinear source term
  virtual real dk(const Ptv(real)& x, real u); // d/du of nonlinear coefficient
public:
```

[9] you will find the complete code in `NlMultiGrid1/`

```
  NlLevel ();
 ~NlLevel () {}

  static  void defineStatic (MenuSystem& menu, int level = MAIN);
  virtual void scan (MenuSystem& menu, String& geometry, String& partition);

  virtual void attachSol(DDSolver& ddsolver, SpaceId i);
  virtual void attachRhs(DDSolver& ddsolver, SpaceId i);
  virtual DegFreeFE& getDof();
  virtual void initProj(DegFreeFE& dofTo);
  virtual Vec(NUMT)& getNonLinSolution();

  virtual BooLean solveSubSystem (LinEqVector& b, LinEqVector& x,
  StartVectorMode start, DDSolverMode mode);
  virtual void residual (LinEqVector& b, LinEqVector& x, LinEqVector& r);
  virtual void matVec (const LinEqVector& b, LinEqVector& x);
  virtual BooLean transfer (const LinEqVector& fv, LinEqVector& tv,
    BooLean add_to_t, DDTransferMode, TransposeMode trans);

  virtual int  getWorkTransfer () const;
  virtual real getStorageTransfer () const;
  virtual int  getWorkSolve () const;
  virtual real getStorageSolve () const;

  CLASS_INFO
};

#define ClassType NlLevel
#include <Handle.h>
#undef  ClassType

#define Type Handle(NlLevel)
#include <VecSimplest.h>
#undef Type

class NlLevelf : public NlLevel
{
protected: // nonlinear rhs
  virtual real f (const Ptv(real)& x, real u); // nonlinear source term
  virtual real df(const Ptv(real)& x, real u); // d/du of nonlinear source term
public:
  NlLevelf () {}
 ~NlLevelf () {}
};

class NlLevelk : public NlLevel
{
protected: // nonlinear operator
  virtual real k (const Ptv(real)& x, real u); // nonlinear coefficient
  virtual real dk(const Ptv(real)& x, real u); // d/du of nonlinear coefficient
public:
  NlLevelk () {}
 ~NlLevelk () {}
};

class NlMultiGrid1 : public MenuUDC, public Store4Plotting,
    public NonLinEqSolverUDC, public DDSolverUDC
{
protected:
```

```
  // general data:
  VecSimplest(Handle(NlLevel)) level;  // refinement levels
  Handle(DegFreeFE) dof;   // mapping: nodal values <-> linear system unknowns
  Handle(FieldFE)   u;     // finite element field, the primary unknown

  Handle(Vec(NUMT))      nonlin_solution;      // nonlinear solution
  Handle(Vec(NUMT))      linear_solution;      // solution of linear subsystem
  prm(NonLinEqSolver)    nlsolver_prm;         // parameters for solver
  Handle(NonLinEqSolver) nlsolver;             // nonlinear solver

  int                    no_of_grids;          // multigrid levels
  prm(DDSolver)          ddsolver_prm;         // parameters multigrid solver
  Handle(DDSolver)       ddsolver;             // multigrid solver

public:
  NlMultiGrid1 ();
 ~NlMultiGrid1 () {}

  virtual void adm    (MenuSystem& menu);
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu); // read and intialize data
  virtual void solveProblem ();  // main driver routine
  virtual void resultReport ();  // write error norms to the screen

  // DDSolverUDC
  SpaceId getNoOfSpaces() const; // no_of_grids
  BooLean solveSubSystem (LinEqVector& b, LinEqVector& x,
    SpaceId space, StartVectorMode start, DDSolverMode mode=SUBSPACE);
  // apply smoother
  void residual (LinEqVector& b, LinEqVector& x, LinEqVector& r, SpaceId space);
  void matVec (const LinEqVector& b, LinEqVector& x, SpaceId space); // apply operator
  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 use the nonlinear solver interface twice. The class `NlMultiGrid1` calls a nonlinear multigrid. Inside the multigrid we make use of nonlinear solves as smoothers on each grid. Therefore we use a second class `NlLevel` derived from `NonLinEqSolverUDC` that implements the smoothers. At this level we also have to specify the differential equation respectively the intergrands and the coefficients. The smoother itself looks similar to the nonlinear solvers discussed previously. We use a vector of `NlLevel` classes, for each grid level one instance of `NlLevel`.

NlMultiGrid1/
main

```
#include <NlMultiGrid1.h>
int main (int nargs, const char** args)
{
```

```
  initDIFFPACK (nargs, args);
  global_menu.init ("Nonlinear test case", "NlMultiGrid1");
  NlMultiGrid1 problem;
  global_menu.multipleLoop (problem);
  return 0;
}
```

The code implements first the nonlinear smoothers `NlLevel` and second the nonlinear multigrid around `NlMultiGrid1`. There are different problems to select, a linear one, one with a nonlinear right hand side and one with a nonlinear operator as in the nonlinear example code `NlElliptic` (equation 1). The implementation for linear right hand side or linear operator is not optimal since the assembly is repeated several times. In an implementation dedicated to a specific problem this should be optimized.

NlMultiGrid1.C

```
#include <NlMultiGrid1.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <createNonLinEqSolver.h>
#include <ErrorEstimator.h>
#include <PreproBox.h>
#include <createElmDef.h>
#include <NonLinDD.h>
#include <createDDSolver.h>

#define Type Handle(NlLevel)
#include <VecSimplest.C>
#undef Type

NlLevel:: NlLevel () {}

INIT_CLASS_INFO(NlLevel)

void NlLevel:: defineStatic  (MenuSystem& menu, int level)
{
  LinEqAdm::           defineStatic (menu, level+1);
  prm(NonLinEqSolver)::defineStatic (menu, level+1);
  FEM::                defineStatic (menu, level+1);
}

void NlLevel:: scan (MenuSystem& menu, String& geometry, String& partition)
{
  grid.rebind (new GridFE());              // create empty grid object
  PreproBox p;
  p.geometryBox() .scan (geometry);
  p.partitionBox().scan (partition);
  p.generateMesh (grid());                 // fill grid

  u.rebind (new FieldFE (grid(),"u"));     // allocate, with field name "u"
  FEM::scan (menu);

  lineq.rebind (new LinEqAdm());
  lineq->scan (menu);
  dof.rebind (new DegFreeFE (grid(), 1));  // 1 unknown per node
```

```
  nlsolver_prm.scan (menu);
  linear_solution.redim  (dof->getTotalNoDof());
  nonlin_solution.rebind (new Vec(NUMT));
  nonlin_solution->redim (dof->getTotalNoDof());
  lineq->attach (linear_solution);
  nlsolver.rebind (createNonLinEqSolver (nlsolver_prm));
  nlsolver->attachUserCode (*this);
  nlsolver->attachLinSol (linear_solution);
}

void NlLevel:: attachSol(DDSolver& ddsolver, SpaceId i)
{
  nonlin_solution->fill (0.0);
  ddsolver.attachLinSol(nonlin_solution(), i);
}

void NlLevel:: attachRhs(DDSolver& ddsolver, SpaceId i)
{
  Handle(Vec(NUMT)) z;
  z.rebind(new Vec(NUMT));
  z->redim(dof->getTotalNoDof());
  Handle(LinEqVector) zero;
  zero.rebind(new LinEqVector(z()));
  zero() = 0.;
  ddsolver.attachLinRhs(zero(), i, dpTRUE);
}

DegFreeFE& NlLevel:: getDof()
{ return dof(); }

Vec(NUMT)& NlLevel:: getNonLinSolution()
{ return nonlin_solution(); }

void NlLevel:: initProj(DegFreeFE& dofTo) // next finer grid
{
  dof2.rebind(&dofTo);
  fillEssBC();
  proj.rebind(new ProjInterpSparse());
  proj->rebindDOF(dof(), dofTo);
  proj->init();
}

void NlLevel:: fillEssBC ()
{
  dof->initEssBC ();           // init for assignment below
  const int nno = grid->getNoNodes();

  for (int i = 1; i <= nno; i++)
    if (grid->BoNode (i))      // any boundary indicator?
      dof->fillEssBC (i, 0.0); // homogeneous Dirichlet.
}

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

```
  const real u_pt = u->valueFEM (fe);  // U (at present itg. point)

  // find the global coord. x of the current integration point:
  Ptv(real) x (nsd);
  fe.getGlobalEvalPt (x);

  const real f_value = f(x, u_pt);
  const real k_value = k(x, u_pt);

  real nabla1,nabla2,h;

  if (nlsolver->getCurrentState().method == NEWTON_RAPHSON)
    {
      Ptv(real) Du_pt (nsd);                        // grad U
      u->derivativeFEM (Du_pt, fe);                 // interpolate Du_pt
      const real df_value = df(x, u_pt);
      const real dk_value = dk(x, u_pt);

      for (i = 1; i <= nbf; i++) {
        nabla1 = 0;
        for (s = 1; s <= nsd; s++) {
          nabla1 += fe.dN(i,s)*Du_pt(s);
        }
        for (j = 1; j <= nbf; j++) {
          nabla2 = 0;
          for (s = 1; s <= nsd; s++)
            nabla2 += fe.dN(i,s)*fe.dN(j,s);
          h = k_value*nabla2 + dk_value*fe.N(j)*nabla1 -
              df_value*fe.N(i)*fe.N(j);
          elmat.A(i,j) += h*detJxW;
        }
        h = k_value*nabla1 - f_value*fe.N(i);
        elmat.b(i) -= h*detJxW;
      }
    }
  else if (nlsolver->getCurrentState().method == SUCCESSIVE_SUBST)
    {
      for (i = 1; i <= nbf; i++) {
        for (j = 1; j <= nbf; j++) {
          nabla2 = 0;
          for (s = 1; s <= nsd; s++)
            nabla2 += fe.dN(i,s)*fe.dN(j,s);
          elmat.A(i,j) += k_value*nabla2*detJxW;
        }
        elmat.b(i) += fe.N(i)*f_value*detJxW;
      }
    }
  else
    errorFP("NlLevel::integrands",
            "Linear subsystem for the nonlinear method %s is not implemented",
            getEnumValue(nlsolver->getCurrentState().method).chars());
    // getEnumValue: returns a string of the enum, .chars() transforms the
    // string to a const char* that can be fed into the printf-like errorFP
}

void NlLevel:: makeAndSolveLinearSystem ()
{
  dof->vec2field (nonlin_solution(), u());  // copy most recent guess to u
```

```
  if (nlsolver->getCurrentState().method == NEWTON_RAPHSON)
    dof->fillEssBC2zero();  // ensure no correction of known values!
  else
    dof->unfillEssBC2zero();// (set back to) normal treatment of ess. b.c.


  makeSystem (dof(), lineq());
  lineq->attach (linear_solution);
  lineq->bl().add(lineq->bl(), linear_rhs());

  // init startvector (linear_solution) for iterative solver:
  if (nlsolver->getCurrentState().method == NEWTON_RAPHSON)
    // start for a correction vector (should -> 0)
    linear_solution.fill (0.0);
  else
    //  use most recent nonlinear solution
    linear_solution = nonlin_solution();

  lineq->solve();  // invoke a linear system solver
}

BooLean NlLevel:: solveSubSystem (
    LinEqVector& b, LinEqVector& x, StartVectorMode start, DDSolverMode)
{
  nonlin_solution.rebind(CAST_REF(x.vec(), Vec(NUMT)));
  nlsolver->attachNonLinSol (nonlin_solution());

  linear_rhs.rebind(b);
  if (start==ZERO_START)
    nonlin_solution->fill (0.0);   // set all entries to 0 in start vector
  fillEssBC ();                    // set essential boundary condition
  dof->fillEssBC (nonlin_solution());

  // call nonlinear solver:
  nlsolver->solve ();
  return dpTRUE;
}

void NlLevel:: residual (LinEqVector& b, LinEqVector& x, LinEqVector& r)
{
  if (notBooLean(lineq->ok(dpTRUE)))
    errorFP("NlLevel::residual");
  dof->vec2field (CAST_REF(x.vec(), Vec(NUMT)), u());
  makeSystem (dof(), lineq());

  lineq->getLinEqSystem ().attach (x);
  lineq->getLinEqSystem ().residual(r);
  r.add(r, b);
}

void NlLevel:: matVec (const LinEqVector& x, LinEqVector& f)
{
  dof->vec2field (CAST_REF(x.vec(), Vec(NUMT)), u());
  makeSystem (dof(), lineq());

  lineq->getLinEqSystem ().attach ((LinEqVector&)x);
  lineq->getLinEqSystem ().residual(f);
  Vec(NUMT) &ff = CAST_REF(f.vec(), Vec(NUMT));
  ff.mult(-1.);
```

```
}

BooLean NlLevel:: transfer (
  const LinEqVector& fv, LinEqVector& tv,
  BooLean add_to_t, DDTransferMode mode, TransposeMode trans)
{
  proj->apply(fv, tv, trans, add_to_t);
  Vec(NUMT)& t = CAST_REF(tv.vec(), Vec(NUMT));
  if ((trans==TRANSPOSED)&&(mode==TRANSFER_NESTED)) { // FAS
    Vec(NUMT)& t = CAST_REF(tv.vec(), Vec(NUMT));
    dof-> fillEssBC(t);
  }
  return dpTRUE;
}

int NlLevel:: getWorkTransfer () const
{ return proj->getWork(); }

real NlLevel:: getStorageTransfer () const
{ return proj->getStorage(); }

int NlLevel:: getWorkSolve () const
{ return ((LinEqAdm&)lineq()).getLinEqSystem ().getWork(); }

real NlLevel:: getStorageSolve () const
{ return ((LinEqAdm&)lineq()).getLinEqSystem ().getStorage(); }

real NlLevel:: f (const Ptv(real)&, real) { return 1.;}
real NlLevel:: df(const Ptv(real)&, real) { return 0.;}

real NlLevel:: k (const Ptv(real)&, real) { return 1.;}
real NlLevel:: dk(const Ptv(real)&, real) { return 0.;}


real NlLevelf:: f (const Ptv(real)&, real u_) { return exp(u_);}
real NlLevelf:: df(const Ptv(real)&, real u_) { return exp(u_);}


real NlLevelk:: k (const Ptv(real)&, real u_) { return exp(u_);}
real NlLevelk:: dk(const Ptv(real)&, real u_) { return exp(u_);}
//_____________________________________________________________________

NlMultiGrid1:: NlMultiGrid1 () {}

void NlMultiGrid1:: adm (MenuSystem& 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 NlMultiGrid1:: define (MenuSystem& menu, int level)
{
  menu.addItem (level,
                "problem",      // menu command/name
                "problem",      // command line option: +nsd
                "1 linear, 2 rhs, 3 coeff",
                "2",            // default answer
```

```
                "I[1:3]1");     // valid answer: 1 integer

  // the domain is fixed: [0,1]^nsd
  menu.addItem (level,
                "no of grid levels", // menu command/name
                "level",        // command line option: +level
                "no of uniform refinements",
                "4",            // default answer (4 levels)
                "I1");          // valid answer: 1 integer

  menu.addItem (level,
                "no of space dimensions", // menu command/name
                "nsd",          // command line option: +nsd
                "",
                "2",            // default answer (2D problem)
                "I1");          // valid answer: 1 integer

  menu.addItem (level,
                "element type", // menu item command/name
                "elm_tp",       // command line option (+elm_tp here)
                "classname in ElmDef hierarchy",
                "ElmB4n2D",     // default answer
                // valid answers are the classnames in the ElmDef hierarchy
                // where all the elements in Diffpack are defined:
                validationString(hierElmDef())); // list all the classnames

  // submenus:
  prm(NonLinEqSolver) ::defineStatic (menu, level+1);
  prm(DDSolver)       ::defineStatic (menu, level+1);
  Store4Plotting      ::defineStatic (menu, level+1);
  menu.setCommandPrefix("smoother");
  NlLevel             ::defineStatic (menu, level);
  menu.unsetCommandPrefix();
}

void NlMultiGrid1:: scan (MenuSystem& menu)
{
  // load answers from the menu:
  no_of_grids = menu.get ("no of grid levels").getInt();
  level.redim (no_of_grids);

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

  int nsd = menu.get ("no of space dimensions").getInt();
  Store4Plotting::scan (menu, nsd);

  int p = menu.get ("problem").getInt();
  int i;
  for (i=1; i<=no_of_grids; i++)
    switch (p) {
    case 1: level(i).rebind (new NlLevel());
      break;
    case 2: level(i).rebind (new NlLevelf());
      break;
    case 3: level(i).rebind (new NlLevelk());
      break;
    default: fatalerrorFP("NlMultiGrid1:: scan","illegal problem number");
```

```
  }

// ---- make grid using a box preprocessor and the menu information: ----
// construct the right syntax for the box preprocessor:
// d=2 [0,1]x[0,1]
// d=2 elm_tp=ElmB4n2D [2,2] [1,1]
// this must valid for any nsd so we must make some string manipulations:
String geometry = aform("d=%d ",nsd);  // e.g. "d=2"
String grading = "[";
for (i = 1; i <= nsd; i++) {
  if (i < nsd) {
    geometry += "[0,1]x";  grading  += "1,";
  } else {
    geometry += "[0,1]";   grading  += "1";
  }
}
grading += "]";

String elm_tp = menu.get ("element type");
menu.setCommandPrefix("smoother");
int d = 1;
for (i=1; i<=no_of_grids; i++) {
  d *= 2;
  int j;
  String part = "[";   // partition string e.g. [2,2]
  for (j=1; j<=nsd; j++) {
    part += aform("%d",d);
    if (j<nsd)
      part += ",";
  }
  part += "]";
  String partition = aform("d=%d elm_tp=%s div=%s grading=%s",
   nsd,elm_tp.chars(),part.chars(),
   grading.chars());

  level(i)->scan (menu, geometry, partition);
  level(i)->attachSol (ddsolver(), i);
  level(i)->attachRhs (ddsolver(), i);
}
menu.unsetCommandPrefix();

for (i=1; i<no_of_grids; i++)
  level(i)->initProj (level(i+1)->getDof());

dof.rebind (level(no_of_grids)->getDof());

u.rebind (new FieldFE (level(no_of_grids)->getDof().grid(),"u"));
// allocate, with field name "u"

nlsolver_prm.scan (menu);
linear_solution.rebind (level(no_of_grids) ->getNonLinSolution() );
nonlin_solution.rebind (new Vec(NUMT));
nonlin_solution->redim(level(no_of_grids) ->getDof().getTotalNoDof() );

nlsolver.rebind (createNonLinEqSolver (nlsolver_prm));
nlsolver->attachLinSol    (linear_solution());
nlsolver->attachNonLinSol (nonlin_solution());
nlsolver->attachUserCode  (*this);
```

```
  NonLinDD& sol = CAST_REF(nlsolver(), NonLinDD);
  sol.attach (ddsolver());
}

void NlMultiGrid1:: solveProblem () // main routine of class NlMultiGrid1
{
  nonlin_solution->fill (1.0);     // set all entries to 1 in start vector
  level(1)->getNonLinSolution().fill (1.0);

  // call nonlinear solver:
  if (!nlsolver->solve ())
    errorFP("NlMultiGrid1::solve","failed");
  // load nonlinear solution found by the solver into the u field:
  s_o<<"maximum = "<<nonlin_solution->norm(Linf)<<endl;

  dof->vec2field (nonlin_solution(), u());
  Store4Plotting::dump (u());      // dump for later visualization
  lineCurves(u());
}

void NlMultiGrid1:: resultReport ()
{
  // in small problems (less than 100 nodes), print the nodal error
  // values on the file "errors.dat"
  if (dof->getTotalNoDof() < 100)
    u->values().print("FILE=u.dat","Nodal values of the solution field");
}

SpaceId NlMultiGrid1:: getNoOfSpaces() const
{ return no_of_grids; }

BooLean NlMultiGrid1:: solveSubSystem (
    LinEqVector& b, LinEqVector& x,
    SpaceId space, StartVectorMode start, DDSolverMode mode)
{ return level(space)->solveSubSystem(b, x, start, mode); }

void NlMultiGrid1:: residual (
  LinEqVector& b, LinEqVector& x, LinEqVector& r, SpaceId space)
{ level(space)->residual(b, x, r); }

void NlMultiGrid1:: matVec (const LinEqVector& b, LinEqVector& x, SpaceId space)
{ level(space)->matVec(b, x); }

BooLean NlMultiGrid1:: transfer (
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode mode)
{
  if (fi == ti-1)      // prolongation
      level(fi)->transfer(fv, tv, add_to_t, mode, NOT_TRANSPOSED);
  else if (fi == ti+1) // restriction
      level(ti)->transfer(fv, tv, add_to_t, mode, TRANSPOSED);
  else fatalerrorFP("NlMultiGrid1:: transfer","from %d to %d", fi, ti);
  return dpTRUE;
}

int NlMultiGrid1:: getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork) const
{
  if (fi == ti-1)
    return level(fi)->getWorkTransfer();
```

```
  if (fi == ti+1)
    return level(ti)->getWorkTransfer();
  return 0;
}

real NlMultiGrid1:: getStorageTransfer (SpaceId fi, SpaceId ti) const
{
  if (fi == ti-1)
    return level(fi)->getStorageTransfer();
  return 0;
}

int NlMultiGrid1:: getWorkSolve (SpaceId space, const PrecondWork) const
{ return level(space)->getWorkSolve(); }

real NlMultiGrid1:: getStorageSolve (SpaceId space) const
{ return level(space)->getStorageSolve(); }

String NlMultiGrid1:: comment ()
{ return "NlMultiGrid1 nonlinear multigrid test"; }
```

5.4 Experiments

Exercise 21 *Nonlinear multigrid methods.*

(table 22, `test1.i`)

menu item	**answer**
problem	{1 & 2 & 3}
no of grid levels	4
no of space dimensions	2
element type	ElmB4n2D
nonlinear iteration method	NonLinDD
max estimated nonlinear error	1.0e-10
nonlinear iteration stopping criterion	3
domain decomposition method	{NonlinearMultigrid & NestedFASMultigrid & FASMultigrid}
cycle type gamma	1
nested cycles	3
nonlinear damping	0.1
smoother basic method	SOR
smoother max iterations	2
smoother nonlinear iteration method	SuccessiveSubst
smoother max nonlinear iterations	1
smoother convergence reports	0

Table 22: Nonlinear multigrid methods, `test1.i`

The first exercise[10] with the nonlinear multigrid method is a comparison of the different nonlinear multigrid versions. We use a standard $V_{1,1}$ cycle with a succesive substitution smoother using one iteration SOR. We compare the nested FAS scheme (with damping), the nonlinear multigrid proposed by Hackbusch [Hac85] and the FAS scheme started with zero on the finest grid. Compare the number of iterations and the computing time. Which method performs best (or fails at a test)? Compare nonlinear multigrid methods with and without damping. How does the damping affect the stability of the method? Wht are good damping values? Which of the three problems is the hardest problem concerning stability?

In the case the initial guess is critical for convergence or the selection of a specific solution, a damping strategy may be necessary. Take a look at the implementation of damping (`nonlinear damping`) and how the damping procedure `getSigma()` in `...DampedMultigrid` may be extended.

damping

Exercise 22 *Nonlinear nested multigrid.*

(table 23, `test2.i`)

menu item	**answer**
problem	{2 & 3}
domain decomposition method	NonlinearMultigrid
nested cycles	{1 & 2 & 3 & 4 & 5}

Table 23: Nonlinear nested multigrid, `test2.i`

There are lots of parameters and details to tune. We change the number of cycles in the nested iteration comparing nested FAS and nonlinear multigrid. We use the $V_{1,1}$ cycle and one nonlinear SOR smoothing step. Find an optimal cycle parameter. What happens if there are not enough cycles on a level? How robust are the methods?

A way of increasing robustness in nonlinear multigrid is applying damping. Now have a look at the damping parameter in the nonlinear multigrid. The factor is relative to the vetor norm of the residual. What are valid damping values? What happens with very large and with very small values? Do you observe some stabilization or increased robustness adjusting the damping parameter?

Exercise 23 *Nonlinear smoothers.*

(table 24, `test3.i`)

We now look at the inner smoothing iteration. We have two parameters at hand: The number of nonlinear smoothing iterations and the number of linear steps inside. We use successive substitution and a linear SOR iteration inside. We compare the number of multigrid iterations needed and we also compare the computation times. First we fix the number of linear SOR iterations to one. What is the optimal number of smoothing steps?

[10] files are in `NlMultiGrid1/Verify/`

menu item	answer
problem	{2 & 3}
smoother basic method	SOR
smoother max iterations	{1 & 2 & 4}
smoother max nonlinear iterations	{1 & 2 & 4}

Table 24: Nonlinear smoothers, `test3.i`

What happens if we use only half the number of smoothing steps but twice the number of linear SOR cycles? The computing time for each step will be lower, but the convergence of the nonlinear multigrid probably suffers. Find a good balance of both linear and nonlinear iteration.

Think of using a relative termination criterion for the nonlinear smoothing iteration. Can you figure out some appropriate tolerance values?

6 Summary

In this report we have presented `Diffpack` simulators that use multigrid equation solvers. It is meant as an introduction and a tutorial for the multigrid algorithms available in `Diffpack`. We did not explain details or syntax of the programs assuming familiarity with `Diffpack` and refer to [Lan94]. For mathematical details of the multigrid algorithms we have referred to the literature[Bri87, Joh87, Wes92] and [Hac85, Bra93]. For the discussion of the related domain decomposition methods available in this framework we refer to [Zum96a]. We also want to refer to [Zum96b] for more advanced topics concerning different grids, different differential operators and different discretizations.

The main solution algorithms discussed in this report were:

- (multiplicative) multigrid
- nested multigrid
- multigrid as preconditioner
- additive multigrid as preconditioner
- Newton's method
- successive substitution (Picard iteration)
- nonlinear multigrid
- nonlinear nested multigrid

The domain decomposition and multigrid interface `DDSolverUDC` is based on

- `solveSubSystem` implements the smoother $\mathcal{S}$

- `transfer` implements the restriction $R_{j,j-1}$ and prolongation $R_{j-1,j}$
- optional `residual` implements the evaluation of the residual $b - \mathcal{L}_j x$

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.

[BPX90] J. H. Bramble, J. E. Pasciak, and J. Xu. Parallel multilevel preconditioners. *Math. Comp.*, 55:1–22, 1990.

[Bra73] A. Brandt. Multi–level adaptive technique (MLAT) for fast numerical solution to boundary value problems. In H. Cabannes and R. Teman, editors, *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*, volume 18 of *Lecture Notes in Physics*, pages 82–89, Berlin, 1973. Springer–Verlag.

[Bra93] J. H. Bramble. *Multigrid Methods*, volume 294 of *Pitman Research Notes in Mathematical Sciences*. Longman Scientific & Technical, Essex, England, 1993.

[Bri87] W. L. Briggs. *A Multigrid Tutorial*. SIAM Books, Philadelphia, 1987.

[Fed64] R. P. Fedorenko. The speed of convergence of one iteration process. *Z. Vycisl. Mat. i. Mat. Fiz.*, 4:559–563, 1964. Also in U.S.S.R. Comput. Math. and Math. Phys., 4 (1964), pp. 227–2356.

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

[Joh87] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Cambridge University Press, Cambridge, 1987.

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

[MMM93] N. D. Melson, T. A. Manteuffel, and S. F. McCormick, editors. *Sixth Copper Mountain Conference on Multigrid Methods*, volume CP 3224, Hampton, VA, 1993. NASA.

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

[Zum96a] G. W. Zumbusch. Domain decomposition methods in Diffpack. Technical report, SINTEF Applied Mathematics, Oslo, 1996.

[Zum96b] G. W. Zumbusch. Multigrid methods in Diffpack, advanced features. Technical report, SINTEF Applied Mathematics, Oslo, 1996. in preparation.