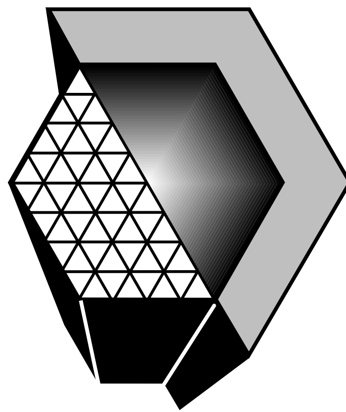
Multigrid on different grids

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 is a continuation of an introductory report on the multigrid iterative solvers in `Diffpack`. We consider the solution of equation systems stemming from the finite element discretization of partial differential equations on different grids. In the introductory report only uniform partitions of the unit square and unit cube were treated. Now we consider also multigrid for mapped elements, grids generated by the meshing of super elements and unstructured (and non nested) grids. The first steps are guided by a couple of examples and exercises.

Contents

1 Introduction 1

2 Mapped elements 2

3 Super element grids 8

4 Unstructured grids 13

5 Rough domain boundary 22

6 Conclusion 23

References 25

Multigrid on different grids

Gerhard W. Zumbusch *

November 22, 1996

1 Introduction

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

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

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

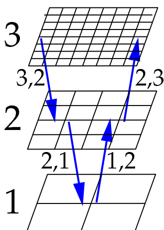


Figure 1: Multigrid V-Cycle

algorithm

The algorithm may be written recursively like this

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

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

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

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

A standard assumption is that the finite element spaces of different scales are nested

$$\mathcal{V}_1 \subset \mathcal{V}_2 \subset \mathcal{V}_3 \subset \dots$$

and that the projection

$$R_{j,j-1} : \mathcal{V}_j \to \mathcal{V}_{j-1}$$

and prolongation

$$R_{j-1,j} : \mathcal{V}_{j-1} \to \mathcal{V}_j$$

are cheap operations. This is of course true for structured grids on the unit square, but the nesting might be violated for more complicated unstructured grids. The efficiency of the transfer operators for unstructured grids also depends on the grids and the additional geometrical knowledge given, for example properties of the grid generation process.

Up to now we only have used a regular grid on a unit-square or unit-cube with standard assembly procedures available in `Diffpack`. We now show how to extend this to more general geometries. We assume familiarity with some of the basic concepts of `Diffpack` [BL96, Lan94]. We will use and modify some examples presented in in the multigrid introduction [Zum96]. For a more detailed presentation of the multigrid method we refer to text books like [Hac85] and other references found in [Zum96]. It may be helpful to have access to the `Diffpack` manual pages `dpman` while reading this tutorial. The source codes and all the input files are available at `$DPR/src/app/pde/ddfem/src/`.

The report is organized as follows: We discuss the extension of the multigrid simulator developed in the introductory report to mapped grids. Then we use grids created by the super element preprocessor and in the following section unstructured grids created with the `Diffpack` GeomPack interface. Finally we comment on some problem related to small geometrical features compared to the coarse grid size.

2 Mapped elements

Mapped elements can be used to approximate curved domain boundaries. The finite element discretization is equivalent to discretization of a regular grid on regular elements with varying coefficients. Hence a standard multigrid method suited for varying coefficients is applicable. However the given implementation should be extended for this purpose:

The general set-up for the grid transfer by `ProjInterpSparse` relies on interpolation using coordinate information. Computing the mapping and the inverse mapping for mapped elements can be expensive. Since the result is just the grid transfer of a regular grid, we can improve the performance in this case substituting the interpolation

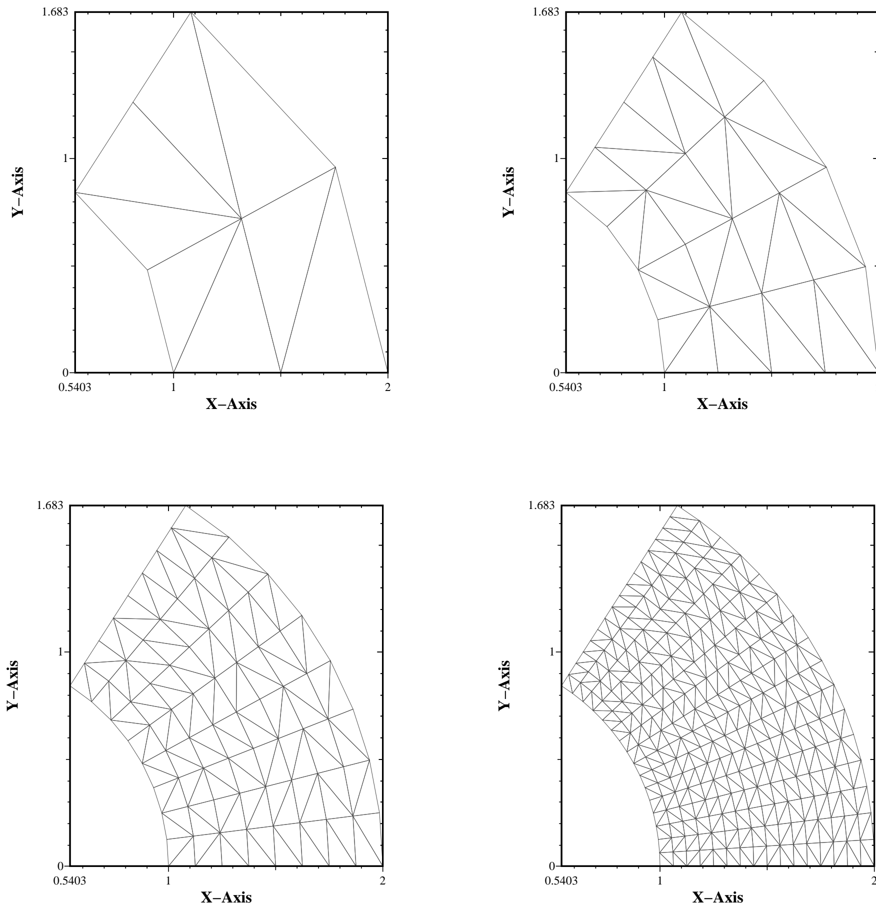


Figure 2: A Hierarchy of grids with mapped elements, angle $\theta = 1$

procedure.

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto \Big(x_1 a + (1 - x_1) b\Big) \cdot \begin{pmatrix} \cos(x_2\theta) \\ \sin(x_2\theta) \end{pmatrix}$$

We start with the `MultiGrid2` example simulator described in [Zum96]. We want to extend it to mapped finite elements using transfinite mappings. We use the approach described in chapter 2.7 [Lan94] and chapter 4 in [LPS94]. Combining the transfinite mapping with the standard multigrid simulator looks like this[1]. We also refer to the documentation in [Lan94].

MGGeo1.h

```
// prevent multiple inclusion of MGGeo1.h
#ifndef MGGeo1_h_IS_INCLUDED
#define MGGeo1_h_IS_INCLUDED
#include <MultiGrid2.h>
#include <TransfMap.h>

class Box2Disk : public GridMapUDC
{
  real theta_0, a, b;  // tranformation parameters
public:
```

[1] you will find the code in MGGeo1/

```
  Box2Disk (real a = 1.0, real b = 2.0, real theta_0 = M_PI_4);
  virtual void sideFunctions
    (Ptv(real)& deformed, const Ptv(real)& refbox, int side, real t = DUMMY);
};

class MGGeo1 : public MultiGrid2 // grids by transfinite mapping
{
  friend class DiskSolution;
protected:
  // general data:
  real a,b; // inner and outer radius

  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

public:
  MGGeo1 ();
 ~MGGeo1 () {}

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

class DiskSolution : public FieldFunc
{
  MGGeo1& data;
public:
  DiskSolution (MGGeo1& simulator) : data(simulator) {}
 ~DiskSolution () {}
  virtual real valuePt (const Ptv(real)& x, real t = DUMMY);
  virtual real operator() (const Ptv(real)& x, real t = DUMMY) const;
};
#endif
```

The simulator class `MGGeo1` is derived from the `MultiGrid2` class. The `Box2Disk` class is used to move the nodes of the grids from their position on the unit square to the annular sector. This is done in the `scan` procedure. The class `DiskSolution` implements the analytical solution to compare with. It is allowed to read the parameters of the geometry via the `friend` statement in class `MGGeo1`. The menu handling procedures of `MultiGrid2` are extended for the new geometric parameters.

MGGeo1.C

```
#include <MGGeo1.h>
#include <PreproBox.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <DDIter.h>
#include <PrecDD.h>
#include <createElmDef.h>  // for calling hierElmDef in MGGeo1::define
#include <createMatrix_real.h> // creating stiffness matrices
```

```
#include <createDDSolver.h> // creating multigrid object
#include <createLinEqSolver.h> // creating smoothers
#include <createRenumUnknowns.h> // renumbering grids
#include <RenumUnknowns.h> // renumbering grids

Box2Disk:: Box2Disk (real a_, real b_, real theta_0_)
  : a(a_), b(b_), theta_0(theta_0_)
{
  if (eq(theta_0,M_PI_2) || eq(theta_0,3*M_PI_2))
    fatalerrorFP("Box2Disk constr.",
                 "theta=%g, illegal value (tan(%g) can give NAN",

                 theta_0,theta_0);
}
// transformation of a rectangle to a part of a disk a<r<b:
void Box2Disk:: sideFunctions
  (Ptv(real)& deformed, const Ptv(real)& refbox, int side, real /*t*/)
{
  if (side == 1)
    {
      deformed(1) = b*cos(refbox(2)*theta_0);
      deformed(2) = b*sin(refbox(2)*theta_0);
    }
  else if (side == 2)
    {
      deformed(1) = (refbox(1)*(b-a)+a)*cos(theta_0);
      deformed(2) = tan(theta_0)*deformed(1);
    }
  else if (side == 3)
    {
      deformed(1) = a*cos(refbox(2)*theta_0);
      deformed(2) = a*sin(refbox(2)*theta_0);
    }
  else if (side == 4)
    {
      deformed(1) = refbox(1)*(b-a)+a;
      deformed(2) = 0;
    }
}

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

real DiskSolution:: valuePt (const Ptv(real)& x, real /*t*/)
{
  // could check consistency of x.size as we did in analyticalSolution
  const real r = sqrt (sqr(x(1)) + sqr(x(2)));
  const real ua = log(r/data.a)/log(data.b/data.a);
  return ua;
}

real DiskSolution:: operator() (const Ptv(real)& x, real t) const
{
  return CAST_CONST_AWAY(DiskSolution)->valuePt (x, t);
}

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

MGGeo1:: MGGeo1 () {}
```

```
void MGGeo1:: define (MenuSystem& menu, int level)
{
  MultiGrid2:: define (menu, level);
  menu.addItem (level, "theta", "theta", "angle of grid disk", "0.3", "R1");

  menu.addItem(level,
               "inner radius",
               "irad",
               "inner radius",
               "1.0",
               "I[0.05:10]");
  menu.addItem(level,
               "outer radius",
               "orad",
               "outer radius",
               "2.0",
               "I[0.07:20]");
}

void MGGeo1:: scan (MenuSystem& menu)
{
  MultiGrid2:: scan (menu);
  initProj(); // init projections before moving the grids!

  real theta_0 = menu.get("theta").getReal();
  a=menu.get("inner radius").getReal();
  b=menu.get("outer radius").getReal();
  // apply mesh transformation:
  Box2Disk mapping_formula (a, b, theta_0);
  TransfMap mapping (mapping_formula);
  for (int i=1; i<=no_of_grids; i++)
    grid(i)->move (mapping);
}

void MGGeo1:: 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,1))
      dof(space)->fillEssBC (i, 1.0);  // u=1 at r=b
    else if (grid(space)->BoNode (i,3))
      dof(space)->fillEssBC (i, 0.0);  // u=0 at r=a
  }
#ifdef DP_DEBUG
  dof(space)->printEssBC (s_o, 2);     // for checking the essential boundary cond.
#endif
}

void MGGeo1:: solveProblem ()
{
  initMatrices();

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

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

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

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

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

  // the solution is now in linsol, it must be copied to the u field:
  dof(no_of_grids)->vec2field (linsol, u());
  Store4Plotting::dump (u());                // dump for later visualization
   lineCurves(u());
  DiskSolution uanal(*this);
  ErrorEstimator::errorField (uanal, u(), DUMMY, error());
  Store4Plotting::dump (error());

  ErrorEstimator::Lnorm (uanal, // 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.
}

real MGGeo1:: f (const Ptv(real)& /*x*/)
{ return 0; }

real MGGeo1:: k (const Ptv(real)& /*x*/)
{ return 1; }
```

The example code only works for two dimensional problems. This is due to the two dimensional transformation from the unit square onto an annular ring. However, the extension to three dimensions with another transformation is straightforward.

As you see in figure 2, the grids do not seem to nested because the square shaped elements are subdivided into triangles in different ways. However the computations are done with bilinear square shaped elements, which means nested grids. The triangulation is an artifact of the printing software (`plotmtv`).

Notice that we have moved the initialization of the projection operators `initProj`. The interpolation is set up after the regular grids are created and before the grids are mapped to the curvilinear domain. The result would have been the same, if we did the interpolation for the mapped grids, but doing it on the unit square is computationally cheaper.

The following input file may be starting point for your experiments[2].

Exercise 1 *Mapped elements.*

(table 1, `test1.i`)

menu item	**answer**
theta	{0.3 & 1.0 & 1.57 & 3.14 }
inner radius	1.0
outer radius	2.0
no of grid levels	4
no of space dimensions	2
coarse partition	[2,2]
refinement	[2,2]
sweeps	[2,2]
basic method	DDIter
domain decomposition method	NestedMultigrid

Table 1: Mapped elements, `test1.i`

Interesting play parameters certainly are the parameters defining the geometry of the domain. In table 1 a parameter study for the angle θ is given. The ratio r_2/r_1 may also be of interest. An alternative view of the computations is the transformation back onto the unit square, now with an anisotropic smooth variable coefficient operator. Hence the numerical methods and results may be comparable to the results for such kinds of operators on the unit square.

3 Super element grids

Following chapter 5 in [LPS94] we introduce super elements and generate finite element grids with the super element preprocessor in `Diffpack`. The idea is to create a macro grid of huge "super" elements. The final grid is generated by subdividing (or meshing) each of the super elements. The advantage is that the specification of the grid does not contain all elements, but just geometric objects of the size necessary to describe the geometry. The meshing procedure itself can create structured grids, which is simple and efficient and does not require further parameters or heuristics.

For the multigrid method based on super elements we start with a very coarse meshing on the super element grid. Finer grids are obtained using finer meshing of the super element grid. The grids are nested if the partition of each super element is nested.

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

MGGeo2.h

```
// prevent multiple inclusion of MGGeo2.h
```

[2] files are in MGGeo1/Verify/

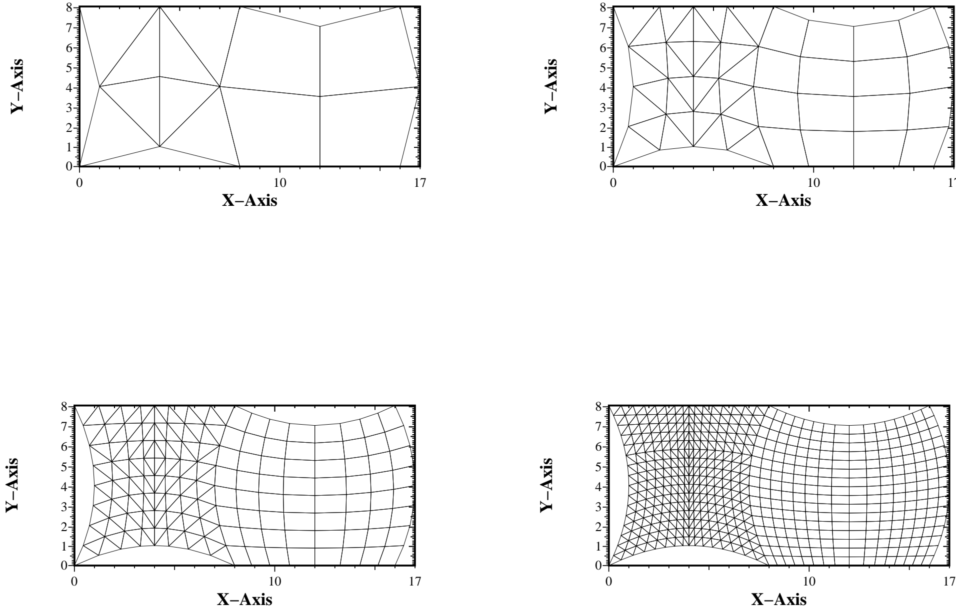


Figure 3: A Hierarchy of grids with super elements

```
#ifndef MGGeo2_h_IS_INCLUDED
#define MGGeo2_h_IS_INCLUDED

#include <MultiGrid2.h>

class MGGeo2 : public MultiGrid2 // grids by super elements
{
protected:
  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 scanGrids(MenuSystem& menu);// construct hierarchy of grids

public:
  MGGeo2 ();
 ~MGGeo2 () {}

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

The class `MGGeo` is derived from class `MultiGrid2`. The grid generation procedure `scanGrids` is changed to use the super element preprocessor `PreproSupElSet`. The menu handling is changed accordingly.

MGGeo2.C

```
#include <MGGeo2.h>
```

```
#include <PreproSupElSet.h>
#include <PartitionSupElSet.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <DDIter.h>
#include <PrecDD.h>
#include <createElmDef.h>  // for calling hierElmDef in MGGeo2::define
#include <createMatrix_real.h> // creating stiffness matrices
#include <createDDSolver.h> // creating multigrid object
#include <createLinEqSolver.h> // creating smoothers
#include <createRenumUnknowns.h> // renumbering grids
#include <RenumUnknowns.h> // renumbering grids

MGGeo2:: MGGeo2 () {}

void MGGeo2:: define (MenuSystem& menu, int level)
{
  menu.addItem (level, "geometry", "geometry", "superelement geometry file",
                "FILE=geo/two_supels.geom", "S");

  menu.addItem (level, "partition", "partition", "superelement coarse partition file",
                "FILE=geo/two_supels.part", "S");

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

  FEM::             defineStatic (menu, level+1);// numerical integration rule
  Store4Plotting::  defineStatic (menu, level+1);// dumping of fields and curves
}

void MGGeo2:: scanGrids (MenuSystem& menu) // construct hierarchy of grids
{
  String geometry = menu.get ("geometry");
  GeometrySupElSet geom;
  geom.scan (geometry);

  String partition = menu.get ("partition");
  PartitionSupElSet part;
  part.scan (partition);
  int nsd = part.getNsd();

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

  for (int i=1; i<=no_of_grids; i++) {
    PreproSupElSet p(geom, part);
    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();

    int nose = part.getNose();
    for (int e=1; e<= nose; e++) {
      for (int d=1; d<= nsd; d++) // refine partition for next grid
part.setDiv(e, d, ref(d) * part.getDiv(e, d));
      part.checkData(e);
      part.calcPartition(e);
    }
  }

  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";
  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 MGGeo2:: solveProblem () // main routine of class MGGeo2
{
  initProj();
  initMatrices();

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

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

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

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

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

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

void MGGeo2:: resultReport ()
{
  if (grid(no_of_grids)->getNoNodes() < 100)
    u->values().print("FILE=u.dat","Nodal values of the solution field");
}

real MGGeo2:: f (const Ptv(real)& /*x*/)
{ return 1; }

real MGGeo2:: k (const Ptv(real)& /*x*/)
{ return 1; }
```

The projection and interpolation procedures are not changed at all. This means the standard interpolation procedure `ProjInterpSparse` is used. Since it is known in advance that the grids are nested and set up by bisection, a more efficient set up of the transfer matrix could be implemented.

The following input files may be some guideline for your experiments[3]. The geometry files are taken from $TIMR/doc/prepro/ex and are explained in [LPS94]. The division in the partition files is set to $[2, 2]$ and $[2, 2, 2]$ respectively for the creation of a coarse grid. The finer grids are created dividing the grid into more elements.

menu item	**answer**
geometry	FILE=two_supels.geom
partition	FILE=two_supels.part
no of grid levels	4
refinement	[2,2]
sweeps	[2,2]
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	NestedMultigrid
smoother basic method	SOR

Table 2: Super elements, `test1.i`

We just give some input files for the code. Table 2, file `test1.i` runs multigrid on the region depicted in figure 3. Some standard efficiency test concerning smoother and number of refinement levels can be applied.

menu item	**answer**
geometry	FILE=test3D.geom
or	FILE=test3Db.geom
partition	FILE=test3D.part
or	FILE=test3Db.part
refinement	[2,2,2]
no of grid levels	3

Table 3: Super elements, 3D, `test2.i`, `test3.i`

Input files for the three dimensional case are in table 3, files `test2.i` and `test3.i` runs multigrid on a three dimensional region, the second one depicted in figure 4. Some standard efficiency test concerning smoother and number of refinement levels may be applied. Compare the results to results the unit square.

[3]files are in MGGeo2/Verify/

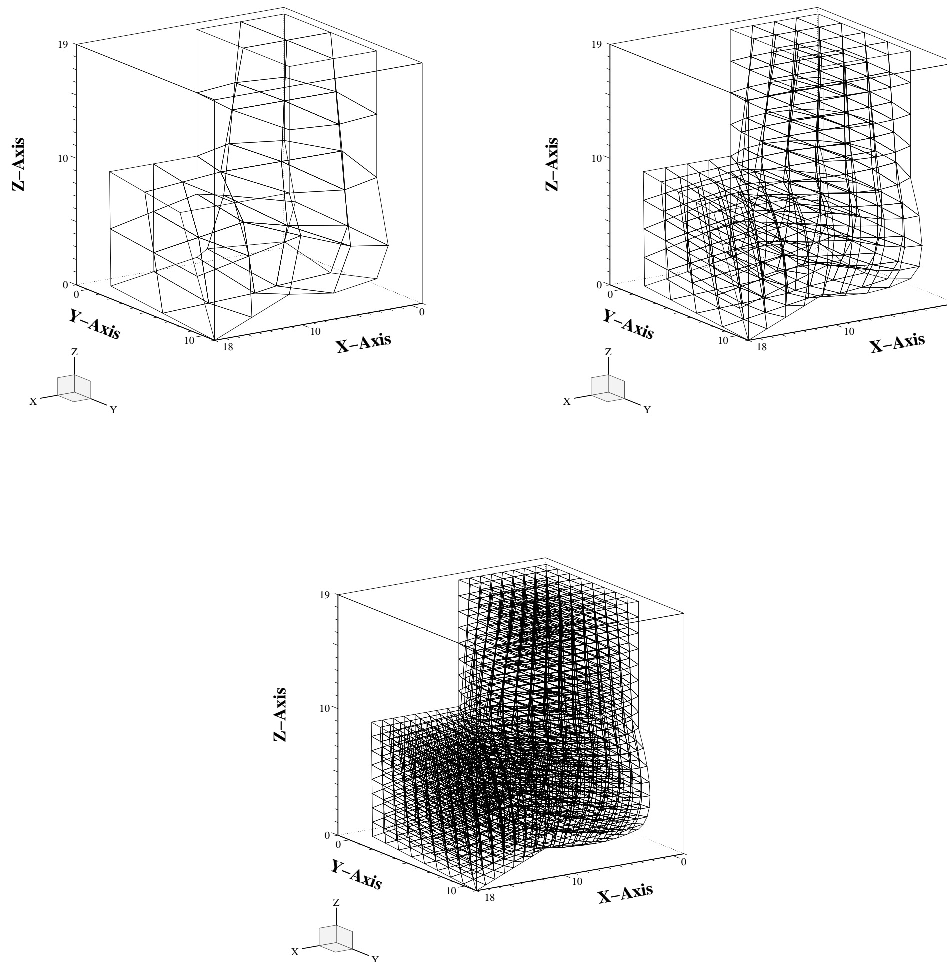


Figure 4: A Hierarchy of grids with super elements

4 Unstructured grids

Unstructured grids may be useful for geometrically complex domains. Multigrid can be applied to problems on unstructured grids. However the main problem usually is a (nested) hierarchy of grids. Starting with a grid resolving the given geometry it is easy to construct finer grids refining each element. The other way round starting with a unstructured fine grid is not possible in general. Coarsening usually will violate the nested-ness of the grids. This topic will be covered in section on non-nested grids.

Suppose a hierarchy of unstructured grids is given: We can feed these into our multigrid code. A time consuming part of the computation will be the interpolation since there is no connecting information available. The interpolation will do lots of local searches to find the right parent-child relations of both grids.

Although the multigrid will run, it will be wiser to supply this refinement information which we will do in the context of adaptive refinement and will be described elsewhere.

Using non-nested spaces sometimes may be necessary although usually ones tries to

avoid it. Technically this means that the coarser finite element space is not a subspace of the finer finite element space any longer. Hence there are two problems:

The issue of convergence: The coarse space has to be near enough to the finer space to have a convergent method. It may be necessary to improve the smoother (or use something like W cycles).

The computational cost: The interpolation procedure will be more expensive than in the nested case. There are more arithmetical operations necessary to compute the projection.

Our multigrid implementation also covers non-nested spaces. However improvement of the interpolation procedure may be necessary.

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

MGGeo3.h

```
// prevent multiple inclusion of MGGeo3.h
#ifndef MGGeo3_h_IS_INCLUDED
#define MGGeo3_h_IS_INCLUDED

#include <MultiGrid2.h>

class MGGeo3 : public MultiGrid2  // grids by geompack
{
protected:
  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 scanGrids(MenuSystem& menu);// construct hierarchy of grids

public:
  MGGeo3 ();
 ~MGGeo3 () {}

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

The class `MGGeo3` is derived from `MultiGrid2`. The grid generation in procedure `scanGrids` is changed to use the `GeomPack` interface in `Diffpack`. The menu handling is changed accordingly.

MGGeo3.C

```
#include <MGGeo3.h>
#include <PreproGeomPack.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <DDIter.h>
```

```
#include <PrecDD.h>
#include <createElmDef.h>  // for calling hierElmDef in MGGeo3::define
#include <createMatrix_real.h> // creating stiffness matrices
#include <createDDSolver.h> // creating multigrid object
#include <createLinEqSolver.h> // creating smoothers
#include <createRenumUnknowns.h> // renumbering grids
#include <RenumUnknowns.h> // renumbering grids

MGGeo3:: MGGeo3 () {}

void MGGeo3:: define (MenuSystem& menu, int level)
{
  menu.addItem (level, "geometry", "geometry", "geompack geometry file",
               "FILE=geo/annulus.geom", "S");

  menu.addItem (level, "partition", "partition", "geompack partition file",
               "FILE=geo/annulus.part", "S");

  menu.addItem (level,
               "refinement factor", // menu command/name
               "refinement",        // command line option: +level
               "progression of no of elements",
               "4.0",               // default answer (2D problem)
               "R1");               // valid answer: 1 real

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

  // submenus:
  LinEqAdm::        defineStatic (menu, level+1);// linear system parameters
  prm(DDSolver)::   defineStatic (menu, level+1);// multigrid parameters

  //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();

  FEM::             defineStatic (menu, level+1);// numerical integration rule
  Store4Plotting::  defineStatic (menu, level+1);// dumping of fields and curves
}

void MGGeo3:: scanGrids (MenuSystem& menu) // construct hierarchy of grids
{
  String geometry = menu.get ("geometry");
  String partition = menu.get ("partition");
  real refinement= menu.get ("refinement factor").getReal();

  PreproGeomPack p;
  p.geometryGeomPack() .scan (geometry);
  p.partitionGeomPack().scan (partition);
  real no_elem = p.partitionGeomPack().getNTriD();

  for (int i=1; i<=no_of_grids; i++) {
    p.partitionGeomPack().setNTriD((int)no_elem);
    grid(i).rebind (new GridFE());  // make an empty grid
    p.generateMesh (grid(i)());
    no_elem *= refinement;     // refine next grid

    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();
  }

  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";
  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 MGGeo3:: solveProblem () // main routine of class MGGeo3
{
  initProj();
  initMatrices();

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

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

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

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

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

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

void MGGeo3:: resultReport ()
{
  if (grid(no_of_grids)->getNoNodes() < 100)
    u->values().print("FILE=u.dat","Nodal values of the solution field");
}

real MGGeo3:: f (const Ptv(real)& /*x*/)
{ return 1; }

real MGGeo3:: k (const Ptv(real)& /*x*/)
{ return 1; }
```

The geometry files are taken from $TIMR/doc/prepro/gptest and are explained in [LPS94]. The requested number of triangles/ tetrahedra specified in the partition files `*.part` is changed to a low number for the initial coarse grid. The finer grids are created increasing this number by a refinement factor given in the input menu.

The following input files may be some guideline for your experiments[4]. The geometry files are taken from $TIMR/doc/prepro/gptest/Data2D and `Data3D` and are explained in [LPS94] in chapter 6.

Table 4 and file `test1.i` contain the input parameters for the computation on the annulus region in figure 5. A refinement factor of 4 is chosen to mimic some bisection strategy in two dimensions. Standard multigrid parameters are used and may be varied for some experiments with smoother parameters. The coarse grid solution is done with a direct solver because of the large number of unknowns. A renumbering

[4] files are in `MGGeo3`/Verify/

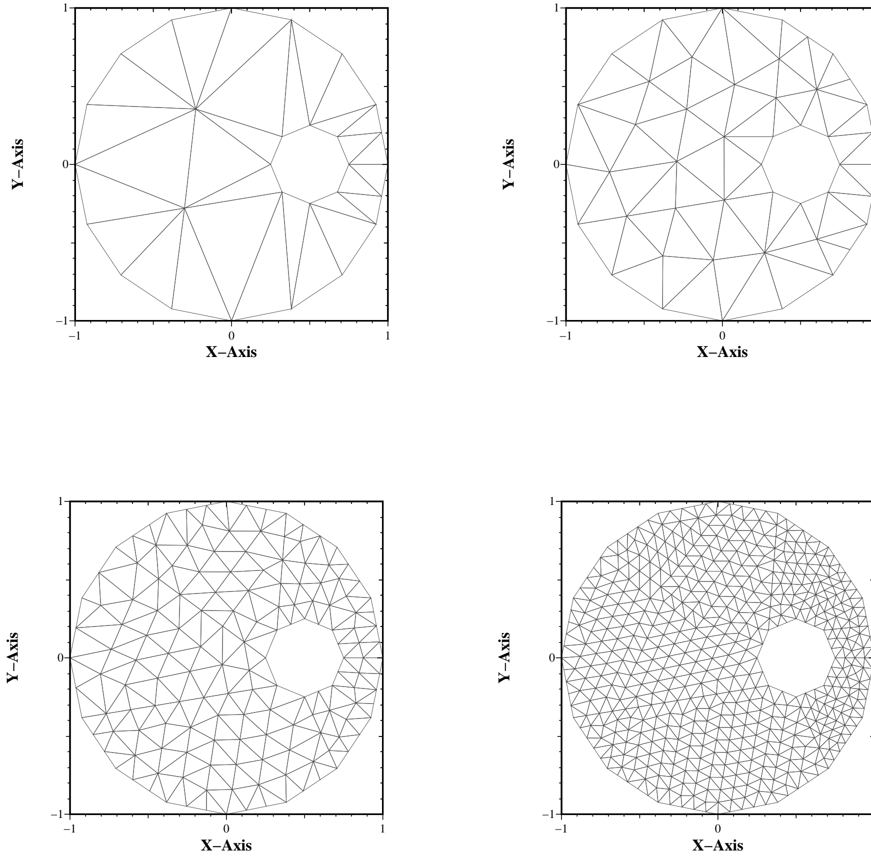
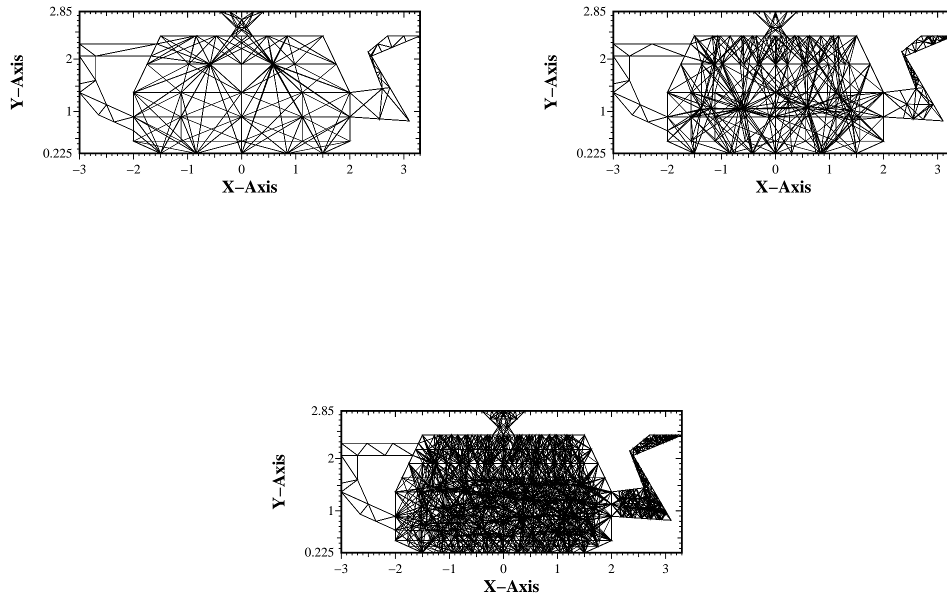


Figure 5: A Hierarchy of unstructured grids "annulus"

scheme suitable for a sparse matrix Gaussian elimination is chosen to restrict the fill in.

An example for a computation in three dimensions is given in table 5 and file `test2.i`. The grid are depicted in figure 6 while a cross section of the solution is in figure 7. The refinement factor for bisection in three dimension would be originally 8, but we have chosen a slightly lower value. The exact number of elements cannot be controlled and the given element number are just request for the GeomPack grid generator.



Figure

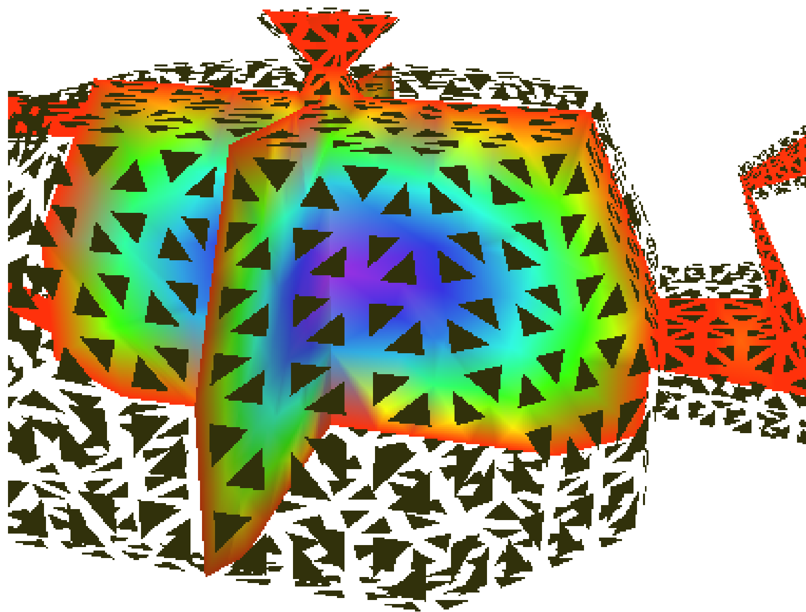


Figure 7: Solution on the 3D unstructured grid “teapot”

menu item	**answer**
geometry	FILE=annulus.geom
partition	FILE=annulus.part
no of grid levels	4
refinement factor	4.0
sweeps	[2,2]
matrix type	MatSparse
basic method	DDIter
domain decomposition method	NestedMultigrid
smoother basic method	SOR
smoother renumber unknowns	RenumNoUnknowns
coarse grid basic method	GaussElim
coarse grid renumber unknowns	AMDhat

Table 4: Unstructured grids, 2D, `test1.i`

menu item	**answer**
geometry	FILE=teapot.geom
partition	FILE=teapot.part
no of grid levels	3
refinement factor	6.0 *(standard is 8)*

Table 5: Unstructured grids, 3D, `test2.i`

5 Rough domain boundary

Rough boundaries of a domain lead to fine and complicated finite element grids. A hierarchy of grids for such a domain starts with a fine grid resolving the boundary and refines this grid further. It is difficult to apply multigrid on such grids since even the coarsest grid is quite fine and sometimes further refinement is not necessary.

boundary approximation

Often a coarse grid ignoring some of the details of the boundary is sufficient for a good multigrid method. So it is possible to construct a hierarchy of grids where only the finest grid represents the boundary exactly while the coarser grids only approximate the boundary and can be quite coarse (figure 8). Even all grids can be just approximations of the domain, see [KY94].

The question now is how to construct such coarser grids: We can construct unstructured grids with a geometrically less accurate representation of the boundary using some tolerance in the grid generation process. Having constructed such grids, we can run the standard multigrid code. In the vicinity of the boundary the grids will not be nested. The grids can also be constructed adaptively, see [DR96].

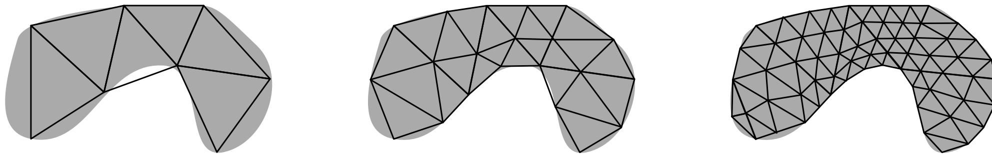


Figure 8: A hierarchy of topologically nested grids successively approximating the domain

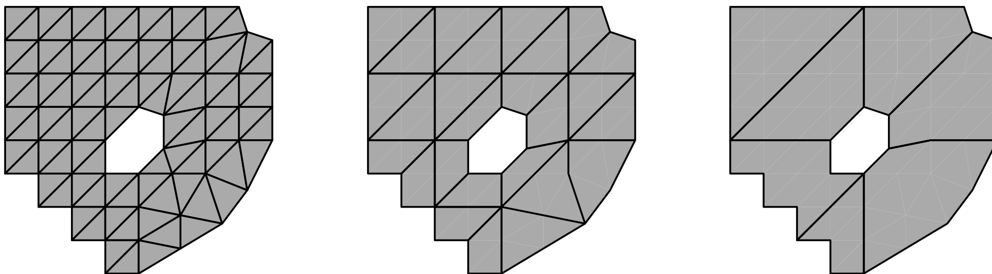


Figure 9: A hierarchy of composite finite element grids obtained by discrete "homogenization" with Galerkin products and standard interpolation

discrete homogenization

Under some additional conditions on the finest grid we can just apply dyadic coarsening and use the Galerkin products to construct the coarse grid matrices. We first construct the finest grid. We start with a regular coarse grid that covers the domain, but may overlap and may be a bad approximation. We apply regular refinement until the mesh-size is small enough to resolve geometrical details. We modify this fine grid by removing elements and moving nodes, but without changing the grid topology and without introducing too much distortion to the elements. This fine grid, which resolves all geometrical details, may serve as a fine grid (see figure 9 left). The sequence of coarse grids is constructed de-refining the grid in the reverse way as the grid

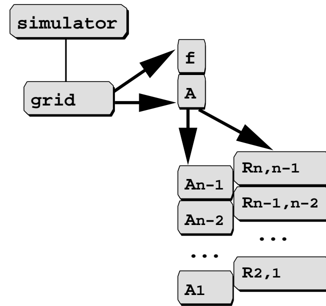


Figure 10: Set up algebraic multigrid

was produced by refinement. This leads to "composite" finite elements which resolve the boundary perfectly, but have a different shape than standard finite elements (see figure 9 middle and right). We use the standard interpolation procedures defined by the element shape functions. The coarse grid matrices are defined by Galerkin products using the interpolation and its adjoint. The geometrical interpretation of the coarse spaces are distorted shape functions on the coarse grid, see [HS95].

algebraic multigrid

The alternative is starting with a fine grid resolving all geometrical details. We can construct coarser grids via some techniques used in algebraic multigrid (see figure 10 and [RS87, BX94]) . We expect in the interior of the domain to have the standard nested grids and refinement. Near the boundary we get different operators resulting from a smoothing/ discrete homogenization of the boundary. The exact results depends on the algebraic multigrid heuristic we choose.

In the case we cannot avoid large coarse grids, we have to discuss the solution of the coarse grid system again. Assume we have n unknowns on the finest grid and N unknowns on the coarsest grid. If we use a direct solver with e.g. a quadratic complexity means $c \cdot N^2$ operation on the coarsest grid times the number of visits on the coarse grid ($\log n - \log N$). The rest of the multigrid method means $c \cdot n$ operations. If the total complexity should not be dominated by the coarse grid solver, we have to have $N < \sqrt{n}$.

We can improve the complexity of the coarse grid solver, using some iterative procedure to relax the condition for N. We have to solve the coarse grid precise enough not to deteriorate the overall multigrid performance.

In the case the coarse grid N is too large, we have to construct coarse grid equations another way: We can use algebraic multigrid applied to the coarsest grid. This means a standard geometric refinement and an algebraic coarsening starting from the coarse grid.

6 Conclusion

In this report we have demonstrated the use of multigrid equation solvers on different types of grids. The multigrid method operates on a sequence of grids of different mesh size covering the computation domain.While the previous introductory report

on multigrid methods covered the computation on uniform grids on the unit square, we have extended both the computation domain and the variety of grids.

We used mapped elements as a modification of structured grids. We used the super element preprocessor to generate a sequence of nested grids based on a coarse geometric description of the domain by super elements. Finally we introduced the GeomPack preprocessor interface to generate a sequence of non-nested unstructured grids of triangles or tetrahedra and run multigrid on this grids.

We made some comments on further modifications due to geometrical details of the domain which included strategies like algebraic multigrid and rough geometry approximations on coarse grids.

The efficiency of the multigrid method on unstructured grids relies on an efficient projection and interpolation procedure. In the case of the GeomPack interface, no auxiliary information to facilitate this is available. However for adaptive grids to be covered in a later report this information is available due to the refinement history.

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.

[BX94] R. E. Bank and J. Xu. The hierarchical basis multigrid method and incomplete LU decompostion. In *Proc. Seventh Int. Conf. on Domain Decomposition Meths.*, volume 180 of *Contemporary Mathematics*, pages 163–173, Providence, 1994. AMS.

[DR96] W. Dörfler and M. Rumpf. An adaptive strategy for elliptic problems including a posteriori controlled boundary approximation. Technical Report 1-96, Math. Fak., Univ. Freiburg, 1996.

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

[HS95] W. Hackbusch and S. A. Sauter. A new finite element approach for problems containing small geometric details. Technical Report 95-6, Mat. Sem., Univ. Kiel, 1995.

[KY94] R. Kornhuber and H. Yserentant. Multilevel methods for elliptic problems on domains not resolved by the coarse grid. In *Proc. Seventh Int. Conf. on Domain Decomposition Meths.*, volume 180 of *Contemporary Mathematics*, pages 49–60, Providence, 1994. AMS.

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

[LPS94] H. P. Langtangen, G. Pedersen, and W. Shen. Finite element preprocessors in Diffpack. Technical Report STF33 A94051, SINTEF Informatics, Oslo, 1994.

[RS87] J. W. Ruge and K. Stüben. Algebraic multigrid (AMG). In S. F. McCormick, editor, *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, Philadelphia, PA, 1987.

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