Overlapping Domain Decomposition 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/`.

Abstract

The report gives an introduction to the overlapping domain decomposition solvers of Schwarz type in `Diffpack`. It is meant as a tutorial for the use of iterative solvers, preconditioners and nonlinear solvers based on overlapping Schwarz methods for partial differential equations. Additive Schwarz methods serve as a standard method for solving equation systems on parallel computers. They are also useful for computations on complicated domains constructed from simple domains where efficient equations solvers are available. We provide an introduction to the implementation and use of such methods in `Diffpack`. The first steps are guided by a couple of examples and exercises. We also want to refer to an accompanied tutorial on multigrid methods in `Diffpack`, which methods and codes are quite related.

Contents

Overlapping Domain Decomposition 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 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. The solution of two problems of half the size is cheaper than solving one large problem. The bisection strategy can of course be applied recursively. The question now is how to

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

divide a problem into sub-problems and how to put the solutions of the sub-problems together to an approximate solution of the global problem. We are constructing iterative solvers or preconditioners for a global problem by splitting the global domain into smaller domains and using solvers for the smaller domains. There are several strategies to do that.

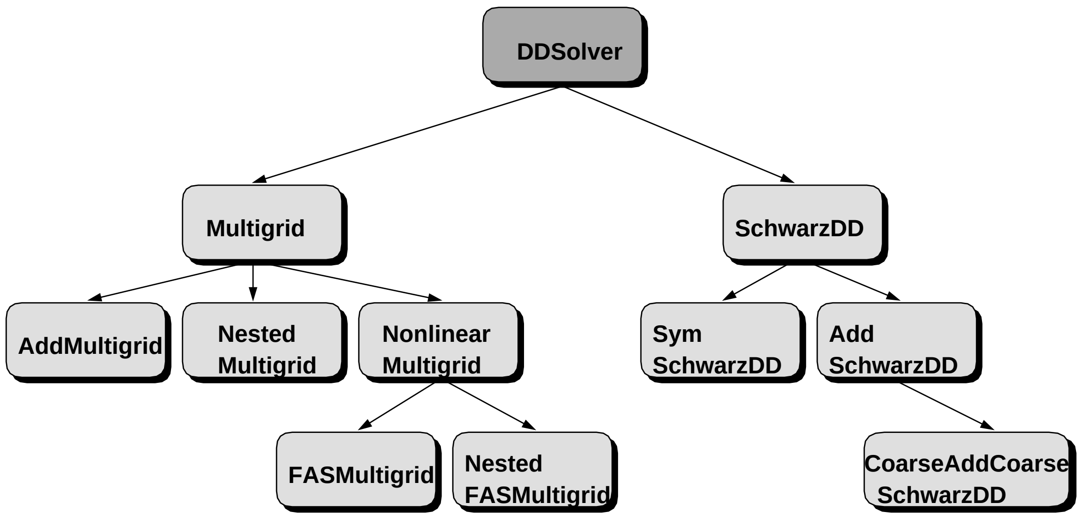

Figure 1: Hierarchy of multigrid and domain decomposition methods

It is just the purpose of this `Diffpack` tutorial to provide some guidance to the use of overlapping 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 parameters and features. 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.

exercises

Since the field of domain decomposition methods is a field of active research, there are lots of conference proceedings and thousands of research papers related. For further reading and for theory we will refer to some of the literature. We especially want to refer to [7] and as a starting point for further searches to the proceedings of the "Domain Decomposition" [3] conferences.

references

We assume familiarity with some of the basic concepts of `Diffpack` [1]. We will use and modify some examples presented in [4] and [8]. 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/`.

2 Overlapping Domain Decomposition

We fix some notation to define the overlapping domain decomposition methods. Given a second order differential operator $\mathcal{L}$ and a domain Ω, we look for the solution of

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

The idea is now to partition the global domain into overlapping sub-domains. Each domain is discretized the same way the global domain is discretized. This is a different way of splitting the finite element space into sub-spaces than the multigrid splitting.

We partition the global domain Ω into a set of sub-domains Ω_i'. We construct a set of overlapping sub-domains $\Omega_i \supset \Omega_i'$. We define the inner boundaries to be

$$\Gamma_i = \partial\Omega_i \setminus \partial\Omega$$

We assume that the finite element spaces on Ω, Ω_i and Ω_i' match. The inner boundaries Γ_i do not cut elements, but are part of some element boundaries. We also assume that the distance

$$c \geq \text{dist}(\Omega_i', \Gamma_i) \geq C$$

is bounded independent of mesh size.

We define the sub-problems for a given last iterate u_0 like this:

$$\begin{aligned}\mathcal{L}u &= f \quad \text{on } \Omega_i \\ u &= u_0 \quad \text{on } \Gamma_i \\ u &= g_1 \quad \text{on } \partial\Omega_i \cap \Gamma \\ \tfrac{\partial}{\partial n}u &= g_2 \quad \text{on } (\partial\Omega_i \cap \Gamma) \setminus \Gamma\end{aligned}$$

3 My first Domain Decomposition Preconditioner

We first consider the case of domain decomposition used as a preconditioner B. We define the additive Schwarz preconditioner as

$$B = \sum_j S_j Q_j$$

with an exact solver S_j for a sub problem on Ω_j and a projection Q_j from Ω to Ω_j. The evaluation of a preconditioner in a Krylov iteration can also be interpreted as one step of a iterative solution procedure with initial guess 0 applied to some right hand side. Hence the last iterate u_0 is zero. The boundary conditions at the inner boundaries of the sub-domains are therefore homogeneous Dirichlet conditions.

The method was originally proposed by Dryja and Widlund [2] for numerical computations.

3.1 Code

We start with the code `MultiGrid2` for the implementation of the additive Schwarz preconditioner. It is a simulator for the *Poisson* equation on a uniform grid on a unit square or unit (hyper-) cube implementing a multigrid preconditioner. The

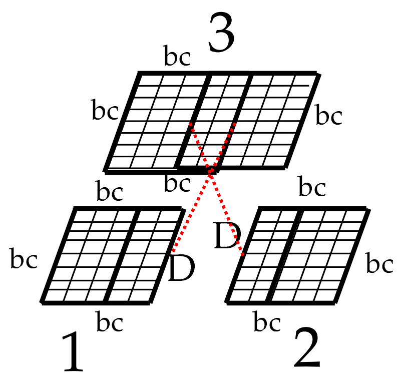

Figure 2: Boundary Conditions in a Overlapping Schwarz Iteration

differences in the header files are: We do not need `preSmooth` and `postSmooth`, since in the additive version each sub-domain is only visited once. We do not need the `residual` since we are constructing an additive preconditioner. We also change the term `smooth` to `sub_solve` more appropriate here.[1]

Overlap1.h

```
// prevent multiple inclusion of Overlap1.h
#ifndef Overlap1_h_IS_INCLUDED
#define Overlap1_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
#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 Overlap1 : 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

  // grid related data:
  int           no_of_grids;                        // number of domains
  prm(Precond)  precondPrm;                         // prm for DD preconditioner
  prm(DDSolver) ddsolver_prm;                       // parameters domain decomposition solver
  VecSimplest(Handle(LinEqSolver))       sub_solve;    // linear solution
  VecSimplest(Handle(prm(LinEqSolver)))  sub_solve_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
```

[1]you will find the code in `Overlap1/`

```
  VecSimplest(Handle(prm(Matrix(NUMT)))) mat_prm;   // Matrix parameters
  VecSimplest(Handle(Proj))              proj;      // projection operators
  Handle(DDSolver)                       ddsolver;  // domain decomposition 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 grids
  virtual void initProj();               // setup proj
  virtual void initMatrices();           // setup stiffness matrices on coarse grids

public:
  Overlap1 ();
 ~Overlap1 () {}

  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);
  BooLean transfer (const LinEqVector& fv, SpaceId fi,
                          LinEqVector& tv, SpaceId ti,
                    BooLean add_to_t= dpFALSE, DDTransferMode=TRANSFER);  // apply proj

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

We need one global grid discretizing Ω which has the number `no_of_grids`. This is the linear system we want to solve using a preconditioned conjugate gradient iteration. The additive Schwarz preconditioner uses the grids 1 to `no_of_grids-1`. The grid are arranged in a pattern given by `partition`. Each grid is structured as given by `subdomain`. The grids are aligned with a fixed `overlap`. The grid are constructed in the procedure `scanGrids`.

The grid transfer operation needed in the additive Schwarz iterations are projections from the global grid to a local grid and back (see figure 3). We set up `no_of_grids-1`

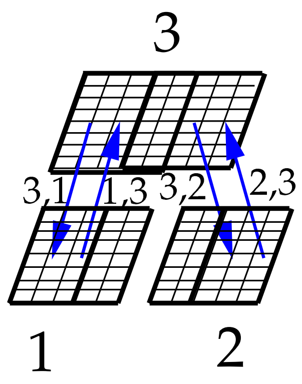

Figure 3: Additive Overlapping Schwarz Iteration

transfer operators which are mainly copy operators, copying the appropriate parts of the global vector to the local vector and vice versa. The operators are set up in `initProj` and are called in `transfer`.

The homogeneous Dirichlet boundary conditions on the inner sub-domain boundaries are implemented automatically since these conditions are imposed on all boundaries. In the case the conditions differ on $\partial\Omega$, the boundary indicators may be used to denote inner boundaries. This will be demonstrated for the multiplicative Schwarz iteration.

Overlap1.C

```
#include <Overlap1.h>
#include <PreproBox.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <PrecDD.h>
#include <createElmDef.h>        // for calling hierElmDef in Overlap1::define
#include <createMatrix_real.h>   // creating stiffness matrices
#include <createDDSolver.h>      // creating multigrid object
#include <createLinEqSolver.h>   // creating sub domain solver
#include <createRenumUnknowns.h> // renumbering grids
#include <RenumUnknowns.h>       // renumbering grids

Overlap1:: Overlap1 () {}

void Overlap1:: 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 Overlap1:: define (MenuSystem& menu, int level)
{
  // the domain is fixed: [0,1]^nsd
  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,
                "subdomain",    // menu command/name
                "subdomain",    // command line options: +partition
                "string like 2,4,2",
                "[4,4]",        // default answer: 4x4 division (5x5 nodes)
                "S");           // valid answer: string

  menu.addItem (level,
                "partition",   // menu command/name
                "partition",   // command line options: +refinement
                "string like [2,2,2] = 8 domains",
                "[2,2]",        // default answer: 2x2 domains
                "S");           // valid answer: string

  menu.addItem (level,
                "overlap",      // menu command/name
                "overlap",      // command line option: +overlap
                "",
                "1",            // default answer
                "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:
  LinEqAdm::        defineStatic (menu, level+1);// linear system parameters
  prm(DDSolver)::   defineStatic (menu, level+1);// DD parameters

  menu.setCommandPrefix("local");
  prm(LinEqSolver)::defineStatic (menu, level+1);// sub domain solver parameters
  menu.unsetCommandPrefix();

  menu.addItem (level,
                "renumber unknowns",              // menu item command/name
                "ren",                            // command line option (+ren here)
                "select a renumbering algorithm",
                hierRenumUnknowns()[0],           // default answer
                validationString(hierRenumUnknowns()) ); // list all classnames

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

void Overlap1:: scan (MenuSystem& menu)
{
  // load answers from the menu:
```

```
  scanGrids(menu); // scan and construct the 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

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

  menu.setCommandPrefix("local");
  for (i=1; i<no_of_grids; i++) {
    sub_solve_prm(i).rebind(new prm(LinEqSolver));
    sub_solve_prm(i)->scan (menu);
    sub_solve(i).rebind(createLinEqSolver (sub_solve_prm(i)()));
    system(i).rebind(new LinEqSystemStd (EXTERNAL_STORAGE));
  }
  menu.unsetCommandPrefix();

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

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

void Overlap1:: scanGrids (MenuSystem& menu) // construct hierarchy of grids
{
  int i;
  int nsd = menu.get ("no of space dimensions").getInt();
  int overlap = menu.get ("overlap").getInt();

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

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

```
Ptv(int) dom(nsd);
for (i = 1; i <= nsd; i++)
  dom(i) = subdom(i) * part(i) - overlap * (part(i)-1);

no_of_grids = 1;
for (i = 1; i<= nsd; i++)
  no_of_grids *= part(i);
no_of_grids += 1;   // compute no_of_grids

sub_solve.redim     (no_of_grids);
system.redim        (no_of_grids);
sub_solve_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);

String elm_tp = menu.get ("element type");

for (i=1; i<=no_of_grids; i++) {
  int j;
  // ---- 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 = "[";
  int k = i-1;
  for (j = 1; j <= nsd; j++) {
    real x0, x1;
    if (i<no_of_grids) {
      int ix = k % part(j); // split into row, column ...
      k = k / part(j);
      x0 = (ix * (subdom(j) - overlap)            ) / (real) dom(j);
      x1 = (ix * (subdom(j) - overlap) + subdom(j)) / (real) dom(j);
    } else
      { x0 = 0.; x1 = 1.;}
    geometry += aform("[%g,%g]", x0, x1);  grading  += "1"; // [.3,.7]x[0,1]
    if (j < nsd) {
      geometry += "x";  grading  += ",";
    }
  }
  grading += "]";

  String part = "[";   // partition string e.g. [4,4]
  for (j=1; j<=nsd; j++) {
    int n;
    if (i<no_of_grids) n = subdom(j);
    else               n = dom(j);
    part += aform("%d",n);
    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)());

     String reduce = menu.get ("renumber unknowns");
     RenumUnknowns* r = createRenumUnknowns(reduce);
       r->renumberNodes (grid(i)());
     delete r;
   }

   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";
   s_o << "\n sub domain:\tNo of nodes: " << grid(1)->getNoNodes()
       << ",\tno of elements: " << grid(1)->getNoElms();
   s_o << "\n total     :\tNo of nodes: " << grid(no_of_grids)->getNoNodes()
       << ",\tno of elements: " << grid(no_of_grids)->getNoElms();
   s_o << "\n\n";
 }

 void Overlap1:: 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 Overlap1:: 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 Overlap1:: initProj() // setup proj operators
{
  for (int i=1; i<no_of_grids; i++) {
    proj(i) = new ProjInterpSparse();
    proj(i)->rebindDOF(*dof(i), *dof(no_of_grids));
    proj(i)->init();
  }
}

void Overlap1:: initMatrices()   // setup stiffness matrices on sub-domains
{
  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 Overlap1:: solveProblem () // main routine of class Overlap1
{
  initMatrices();
  initProj();
  fillEssBC (no_of_grids);                  // set essential boundary conditions
  makeSystem (dof(no_of_grids)(), lineq()); // calculate linear system

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

  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());
  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 Overlap1:: 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() < 100)
    error->values().print("FILE=error.dat","Nodal values of the error field");
}

real Overlap1:: 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 Overlap1:: k (const Ptv(real)& /*x*/)
{ return 1; }

SpaceId Overlap1:: getNoOfSpaces() const
{ return no_of_grids; }

BooLean Overlap1:: solveSubSystem (
  LinEqVector& b, LinEqVector& x,
  SpaceId space, StartVectorMode /*start*/, DDSolverMode /*mode*/)
{
  sub_solve_prm (space)->startmode = ZERO_START;
  system (space)->attach (x, b);
  sub_solve (space)->solve  ( system (space)() );
  system(space)->allow_factorization = dpFALSE;

  Vec(NUMT) &sol = CAST_REF(x.vec(), Vec(NUMT));
  dof(space)->fillEssBC (sol);

  return dpTRUE; // solution has changed
}

BooLean Overlap1:: transfer (
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode)
{
  if (ti == no_of_grids)
    proj(fi)->apply(fv, tv, NOT_TRANSPOSED, add_to_t);
  else if (fi == no_of_grids)
    proj(ti)->apply(fv, tv, TRANSPOSED, add_to_t);
  else fatalerrorFP("Overlap1:: transfer","undefined");
  return dpTRUE;
}

int Overlap1:: getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork) const
{
  if (ti == no_of_grids)
    return proj(fi)->getWork();
  else if (fi == no_of_grids)
    return proj(ti)->getWork();
  return 0;
}

real Overlap1:: getStorageTransfer (SpaceId fi, SpaceId ti) const
{
  if (ti == no_of_grids)
    return proj(fi)->getStorage();
  return 0;
}

int Overlap1:: getWorkSolve (SpaceId space, const PrecondWork) const
{ return sub_solve (space)->getWork(); }

real Overlap1:: getStorageSolve (SpaceId space) const
{ return sub_solve (space)->getStorage(); }

String Overlap1:: comment ()
{ return "Overlap1 additive Schwarz preconditioner test"; }
```

3.2 Partition and overlap

The first exercises with overlapping additive Schwarz preconditioners deal with asymptotics. We try to figure out the dependency of the convergence rates on the size of the sub-domains, the number of sub-domains and the size of the overlap.[2]

Exercise 1 *Size of sub-domains.*

(table 1, `test1.i`)

menu item	**answer**
no of space dimensions	2
subdomain	{[2,2] & [4,4] & [8,8] & [16,16]}
partition	[2,2]
overlap	1
element type	ElmB4n2D
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	AddSchwarzDD
local basic method	GaussElim

Table 1: Size of sub-domains, `test1.i`

We set up a standard version of an additive Schwarz preconditioner. We use the 2 dimensional test problem. We fix the overlap 1, the partition of sub-domains as 2 by 2 and a tolerance for the conjugated gradient iteration. We want to have a look at the convergence rate or the number of iterations needed to solve the problem dependent on the number of unknowns in each sub-domains.

Increasing the number of unknowns for a local problem also increases the number of unknowns of the global problem. Hence we expect some degradation of the performance which should be compensated by the preconditioner. Additionally the relative size of the overlap decreases which also degrades performance. Figure out some numbers and guess some formulas for the dependence. Store the numbers for comparisons with subsequent exercises.

Exercise 2 *Size of overlap.*

(table 2, `test2.i`)

We now fix the number of unknowns on each sub-domain as well as the number of sub-domains. We vary the overlap. Look at the convergence rates for increasing overlap. How is the dependence? Can you guess a formula?

[2]you will find the input parameters in `Overlap1/Verify/`

menu item	**answer**
subdomain	[4,4]
partition	[8,8]
overlap	{1 & 2 & 3}
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	AddSchwarzDD

Table 2: Size of overlap, `test2.i`

We expect some speed up by increasing the overlap. This is obvious for complete overlap but also true for small increasing overlap.

Exercise 3 *Absolute overlap.*

(table 3, `test3.i`)

menu item	**answer**
subdomain	{[4,4] & [8,8] & [16,16]}
partition	[2,2]
overlap	{1 & 2 & 4}
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	AddSchwarzDD

Table 3: Absolute overlap, `test3.i`

We now combine the last two exercises. We fix the absolute size of the overlap according to the assumption at the beginning of the chapter. We increase the number of unknowns in each sub-domain and at the same time we increase the overlap. We again compare the convergence rates. How do they look like?

We can observe the pure preconditioning effect now, which should give a bounded number of iterations.

Exercise 4 *Number of sub-domains.*

(table 4, `test4.i`)

We now have a look at the number of sub-domains. We fix the overlap to 1, the number of unknowns in each sub-domain and the global solution tolerance. We increase the number of sub-domains. Observe the convergence rate of the iteration.

The number of unknowns on the global grid is increasing. Hence the performance will deteriorate. Information transport is only local since each sub-domains shares data only with its neighbors. Increasing the number of sub-domains means therefore increasing the number of cycles to transport information from one sub-domain to all

menu item	**answer**
subdomain	[5,5]
partition	{[2,2] & [3,3] & [4,4] & [8,8]}
overlap	1
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	AddSchwarzDD

Table 4: Number of sub-domains, `test4.i`

other domains. This further degrades performance. The last issue will lead us to Schwarz methods with a coarse grid acceleration.

3.3 Shape and dimension

Exercise 5 *The shape of sub-domains.*

(table 5, `test5.i`)

menu item	**answer**
subdomain	{[6,6] & [4,9] & [3,12] & [2,18]}
partition	{[4,4] & [2,8] & [1,16]}
overlap	1
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	AddSchwarzDD

Table 5: The shape of sub-domains, `test5.i`

Up to now we always have put square shaped sub-domains Ω_j together to a square shaped Ω. This means cutting the global domain Ω along x and y axis into pieces. Alternatively we can also cut Ω into rectangular shaped stripes. We compare the convergence rates of both methods. We fix the number of sub-domains, the number of unknowns in each sub-domain and the discretization of the global domain, but we vary the shape of the sub-domains. Which shape gives the best performance? Try to explain the result.

The number of cycles to transport information from one sub-domain to all other domains from neighbor to neighbor may serve as a model for explanation. Difficulties arising in large cycles here may be overcome by using an additional coarse grid.

While additive Scharz iteration usually is used on parallel computers other issues like the number of neighbors and the size of the inner boundaries play a role. Hence it may even be more favorable for large numbers of sub-domains to use squared shaped (isotropic) partitions of the domain Ω.

Exercise 6 *Three dimensions.*

(table 6, `test6.i`)

menu item	**answer**
no of space dimensions	3
subdomain	{[3,3,3] & [6,6,6]}
partition	[2,2,2]
overlap	{1 & 2}
element type	ElmB8n3D
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	AddSchwarzDD

Table 6: Three dimensions, `test6.i`

We now have a look at a three dimensional test example on the unit cube. The cube is partitioned into cube shaped sub-domains in a pattern like $2 \times 2 \times 2$. We can redo all experiments done for the two dimensional case. However the main question is about boundedness of the convergence rate for fixed absolute overlap (exercise 3). Redo at least this exercise and compare the results.

It is not at all obvious in general that convergence results derived for the two dimensional case also hold in three dimensions. We can also compare results in one dimension or on hyper-cubes in higher dimensional spaces.

Solving larger problems than we did up to now usually means both, solving larger sub-problems and increasing the number of sub-problems. We solved all sub-problems by a direct Gauss elimination. Solving larger sub-problems requires more efficient algorithms. Switching to an iterative sub-problem solver immediately imposes the question of termination criteria, solution tolerance and the influence of iteration errors on the preconditioning.

3.4 Inexact solver

Exercise 7 *Inexact sub-domain solver.*

(table 7, `test7.i`)

We have a look at iterative solvers for sub-domain problems. The size and number of sub-domains is fixed. We choose a preconditioned conjugate gradient iteration, a Jacobi iteration and a symmetric SSOR iteration as sub-domain solvers. We choose a relative termination criterion. Compare the deterioration of the convergence rate when we relax the tolerance for the sub-domain solver. How does inexact sub-domain solution influence the overall convergence? What is a good termination criterion without loosing too much efficiency in the outer iteration? What is the optimal criterion concerning the total number of operations?

The final goal is to use even more efficient sub-domain solver such as a multigrid or domain decomposition itself. The reason to use an additive Schwarz iteration on top

menu item	**answer**
local basic method	ConjGrad
local max iterations	{1 & 2 & 4 & 8 & 16 & 32}
subdomain	[10,10]
partition	[2,2]
overlap	1
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	AddSchwarzDD

Table 7: Inexact sub-domain solver, `test7.i`

of such efficient solvers often is parallel computing. The additive Schwarz requires only little communication during each step and has a fairly simple structure.

3.5 Averaging on the overlap

We now want to improve and extend the implementation given step by step. There is one small modification to improve the performance of the preconditioner for overlap larger than one. The sub-domains solution are just collected and added up on the global grid. On the overlap, that is the part of the domain shared by several sub-domains, there are several contributions added. This occurs if the size `overlap` is greater than one. One idea to improve the preconditioner is to use an average of the contributions on the overlap instead of just adding the contributions. A way of writing this is by the construction of a (continuous) partition of unity $\{\chi_j\}$.

partition of unity

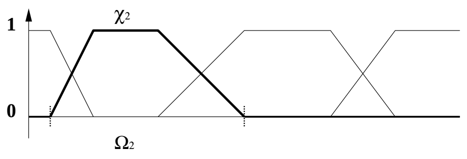

Figure 4: Partition of unity

$$\begin{array}{rcl} \chi_j(x) & \geq & 0 \\ \text{supp}\chi_j & \subset & \overline{\Omega_j} \\ \sum_j \chi_j & \equiv & 1 \qquad \text{on } \Omega \end{array}$$

The averages are computed taking the functions χ_j as weights. The preconditioner looks like

$$B = \sum_j \sqrt{\chi_j} \cdot S_j \cdot \sqrt{\chi_j} Q_j$$

which easily can be hidden in the projection Q_j and its adjoint.

We extend the `initProj` procedure to modify the transfer operator. The partition is computed as one over the number of sub-domains a point is in.[3]

[3]you will find the code in `Overlap2/`

```
#include <Overlap2.h>

Overlap2:: Overlap2 () {}

void Overlap2:: initProj() // setup proj operators
{
  Overlap1:: initProj();
  modifyProj(1, no_of_grids-1);
}

void Overlap2:: modifyProj(SpaceId i0, SpaceId i1) // modify proj
{
  int i;
  Vec(NUMT) &uu = u->values();
  uu.fill(0);
  for (i=i0; i<=i1; i++) {
    Vec(NUMT) loc (dof(i)->getTotalNoDof ());
    loc.fill(1);
    dof(i)->fillEssBC2zero ();
    dof(i)->fillEssBC (loc);
    dof(i)->unfillEssBC2zero (); // 1 inside and 0 on the boundary

    LinEqVector uv(uu);
    proj(i)->apply(loc, uv, NOT_TRANSPOSED, dpTRUE); // sum up
  }
  {
    Vec(NUMT) &uu = u->values();
    int s = uu.size();
    for (int k=1; k<=s; k++)
      uu(k) = 1 / sqrt(uu(k)); // sqrt(the local contribution)
  }
  for (i=i0; i<=i1; i++) {
    Vec(NUMT) loc (dof(i)->getTotalNoDof ());
    LinEqVector ll(loc);
    proj(i)->apply( uu, ll, TRANSPOSED);

    dof(i)->fillEssBC2zero ();
    dof(i)->fillEssBC (loc);
    dof(i)->unfillEssBC2zero (); // result inside and 0 on the boundary

    proj(i)->scale(loc, NOT_TRANSPOSED); // modify the transfer operator
  }
}
```

Exercise 8 *Size of overlap.*

(table 2, see `Verify/test2.i`)

We just repeat the exercise 2 for the new code. That is the variation of the size of the overlap. Compare the results with the original implementation `Overlap1`. Compare also the `error` and the solution plots for both performing just one iteration.

The procedure of averaging is also quite useful in the case of multiplicative Schwarz iteration, especially to construct initial guesses for iterative sub-domain solvers.

We have used a piecewise constant partition of unity. One way to improve the procedure further is to use continuous or even smooth partitions. This applies for overlap sizes greater than two elements.

3.6 Coarse grids acceleration

As we already saw, the Schwarz iteration is quite sensitive on the number of sub-domains involved. The problem can be explained by transporting information from one sub-domain to the others. If there is a long chain of neighbors involved, it takes several cycles to pass along the information. This means information of the right hand side cannot be faster. On the other hand we actually observe this slow down of the Schwarz iteration.

An idea to fix the problem is the introduction of a coarse grid. This enables some global information transport in just one iteration. We introduce one additional sub-problem number 0 which covers the whole domain Ω using a very cheap discretization compared to the original discretization to solve on. We choose the same kind of finite element discretization as used the the other sub-problems. Each element covers about the size of one sub-domain Ω'_j. However the coarse grid is not exactly aligned to sub-domain boundaries, but the coarse grid lines lay on the overlap. The coarse space is not a sub-space of the global finite element space.

$$\begin{aligned} \Omega_0 &= \Omega \\ B &= \sum_{j=0}^{n} S_j Q_j \end{aligned}$$

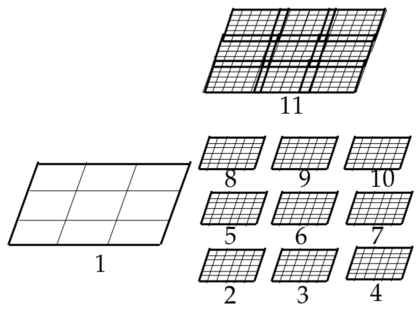

Figure 5: Overlapping Schwarz Iteration with a Coarse Grid

We extend the `scanGrids` procedure generating an additional coarse grid. The transfer from and to the coarse grid is not modified by the partition of unity.[4]

Overlap3.C

```
#include <Overlap3.h>
#include <PreproBox.h>
```

[4]you will find the code in `Overlap3/`

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

Overlap3:: Overlap3 () {}

void Overlap3:: initProj() // setup proj operators
{
  Overlap1:: initProj();
  modifyProj(2, no_of_grids-1);
}

void Overlap3:: scanGrids (MenuSystem& menu) // construct hierarchy of grids
{
  int i;
  int nsd = menu.get ("no of space dimensions").getInt();
  int overlap = menu.get ("overlap").getInt();

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

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

  Ptv(int) dom(nsd);
  for (i = 1; i <= nsd; i++)
    dom(i) = subdom(i) * part(i) - overlap * (part(i)-1);

  no_of_grids = 1;
  for (i = 1; i<= nsd; i++)
    no_of_grids *= part(i);
  no_of_grids += 2;   // compute no_of_grids, coarse grid + global grid

  sub_solve.redim     (no_of_grids);
  system.redim        (no_of_grids);
  sub_solve_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);

  String elm_tp = menu.get ("element type");

  for (i=1; i<=no_of_grids; i++) {
    int j;
    // ---- 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 = "[";
    int k = i-2;
    for (j = 1; j <= nsd; j++) {
      real x0, x1;
      if ((i<no_of_grids)&&(i>1)) {
int ix = k % part(j); // split into row, column ...
k = k / part(j);
x0 = (ix * (subdom(j) - overlap)            ) / (real) dom(j);
x1 = (ix * (subdom(j) - overlap) + subdom(j)) / (real) dom(j);
      } else
{ x0 = 0.; x1 = 1.;}
      geometry += aform("[%g,%g]", x0, x1);  grading  += "1"; // [.3,.7]x[0,1]
      if (j < nsd) {
geometry += "x";  grading  += ",";
      }
    }
    grading += "]";

    String part_s = "[";   // partition string e.g. [4,4]
    for (j=1; j<=nsd; j++) {
      int n;
      if (i==1)                n = part(j); // coarse grid
      else if (i<no_of_grids) n = subdom(j);
      else                     n = dom(j);  // global grid
      part_s += aform("%d",n);
      if (j<nsd)
part_s += ",";
    }
    part_s += "]";

    String partition = aform("d=%d elm_tp=%s div=%s grading=%s",
     nsd,elm_tp.chars(),part_s.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)());

    String reduce = menu.get ("renumber unknowns");
    RenumUnknowns* r = createRenumUnknowns(reduce);
      r->renumberNodes (grid(i)());
    delete r;
  }

  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";
  s_o << "\n coarse grid:\tNo of nodes: " << grid(1)->getNoNodes()
      << ",\tno of elements: " << grid(1)->getNoElms();
  s_o << "\n sub domain:\tNo of nodes: " << grid(2)->getNoNodes()
      << ",\tno of elements: " << grid(2)->getNoElms();
  s_o << "\n total     :\tNo of nodes: " << grid(no_of_grids)->getNoNodes()
      << ",\tno of elements: " << grid(no_of_grids)->getNoElms();
  s_o << "\n\n";
}
```

We can redo the previous exercises comparing the results with the overlapping Schwarz method with and without a coarse grid. The most interesting part probably is the dependence on the number of sub-domains.

Exercise 9 *The number of sub-domains.*

(table 4, `Verify/test4.i`)

We use an additive Schwarz preconditioner with fixed overlap and a fixed number of unknowns in each sub-domain. We increase the number of sub-domains and look at the convergence rate or the number of iterations. What kind of dependence can be observed? How does this compare to the results in exercise 4?

The computations introduce additional work per iteration. Compare the numbers according to the total number of operations (or computing time). Find a criterion when it pays off to use a coarse grid and when it does not.

Exercise 10 *Size of overlap.*

(table 2, `Verify/test2.i`)

It may be interesting to compare the three version of additive Schwarz method discussed up to now. We choose the exercise with varying overlap size again. Compare the convergence rates or number of iterations for all methods. Standard additive Schwarz, with averaging on the overlap and with additional coarse grid. It may also be instructive to compare this to the standard additive Schwarz method with coarse grid.

We have used a conforming finite element coarse grid discretization for reasons of simplicity. There are several other possible ways to construct an appropriate coarse grid discretization. One could construct a coarse grid discretziation which really matches the global grid.

One lower order method on the coarse grid is a piecewise constant approximation on the same coarse grid we have used. This means averaging approximately on each sub-domain and using this value as an unknown in the coarse grid system. The main purpose of the coarse grid system is long distance information transport in the presence of a large number of sub-domain. The approximation properties of the coarse grid are of less importance.

4 Overlapping Schwarz as an iterative method

While a preconditioner can be considered as one step of an iterative procedure applied to a zero initial guess, using an iterative procedures requires handling of non zero initial data. One way to do this (also called Richardson iteration) is to evaluate the residual, apply one iteration with zero initial data to the residual and treat the solution (or a multiple) as an update of last iterate.

$$u \rightarrow u - B(A\,u - f)$$

This approach may be appropriate if the whole solution has changed like in the correction step of a multigrid method. It is too much work, if only parts of the solution are changed like in a multiplicative Schwarz iteration or only parts of the solution are needed like in overlapping Schwarz iteration.

Hence in the following implementation we use a different approach. We use a copy of the right hand side of the global grid on each sub-domain. We use the given data on the boundary of each sub-domain as a Dirichlet boundary condition for computation on that sub-domain. In the notation above this is $u_0 \neq 0$. We avoid computing residuals.

In order to use the Schwarz iteration without preconditioning, we have to employ an averaging technique on the overlap similar to the averaging for the additive preconditioner with a partition of unity $\{\chi_i\}$. We now only apply the scaling to the transfer from a local sub-domain to the global domain since we have copied the residual without modifications from the global domain to sub-domain. Hence we redefine the transposed transfer operators.

$$\begin{array}{rcl}\bar{Q}_i & = & \chi_i \cdot Q_i \\ \bar{Q}_i^* & = & \chi_i \cdot Q_i^*\end{array}$$

We extend the additive Schwarz implementation `Overlap2` to a multiplicative Schwarz iteration and a preconditioner.[5]

We change the data of the transfer operators from a vector to a matrix of projections. We also introduce scaling data for the transfer from one sub-domain onto itself.

MOverlap1.h

```
MatSimplest(Handle(Proj))          proj;      // projection operators
VecSimplest(Handle(LinEqVector))   unity;     // partition of unity
```

The main changes in the code concern boundary conditions and data transfer. We introduce two boundary indicators, number one for the Dirichlet boundary Γ of $\partial\Omega$ and number two for the inner boundaries Γ_i. The original boundary indicators introduced by `PreproBox` [5] for a (hyper-) cube are mapped to the new ones.

MOverlap1.C

in function `MOverlap1::  scanGrids`

```
  p.generateMesh (grid(i)());

  String boInd_g = "nb=2 names= global inner 1=(";
  String boInd_i = "), 2=(";
```

[5]you will find the code in `MOverlap1/`

```
    k = i-1;
    for (j = 1; j <= nsd; j++) {
      int ix = k % part(j) + 1; // split into row, column ...
      k = k / part(j);

      String b1 = aform("%d ", j);
      if ((ix==part(j))||(i==no_of_grids))
        boInd_g += b1;
      else boInd_i += b1;

      String b0 = aform("%d ", j+nsd);
      if ((ix==1)||(i==no_of_grids))
        boInd_g += b0;
      else boInd_i += b0;
    }
    boInd_g += boInd_i;
    boInd_g += ")";
    grid(i)->redefineBoInds(boInd_g);
```

Based on this distinction of boundary conditions, we are able to implement the variable Dirichlet conditions on Γ_i. The `fillEssBC` function has an additional argument. It is a solution vector on the sub-domain containing the prescribed Dirichlet values. The `fillEssBC` function inserts the values into the `dof` object. In order to use a stiffness matrix assembled once and to insert the Dirichlet values later it is necessary to set some flags prior to assembly.

function `MOverlap1:: fillEssBC (LinEqVector& x, SpaceId space)`

```
void MOverlap1:: fillEssBC (LinEqVector& x, SpaceId space) //begin_fill
{
  Vec(NUMT) &xx = CAST_REF(x.vec(), Vec(NUMT));
  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?
      if (grid(space)->BoNode (i, 1))
        dof(space)->fillEssBC (i, 0.);    // u=0 at nodes on the boundary
      else if (grid(space)->BoNode (i, 2))
        dof(space)->fillEssBC (i, xx(i)); // inner boundary
}                      // end_fill
```

in function `MOverlap1:: initMatrices()`

```
    dof(i)->symmModDue2essBC(OFF);   // do not insert Dirichlet BCs now
    dof(i)->modifyVecDue2essBC(OFF); // do not change rhs due to BCs
```

Unfortunately this implies an unsymmetric modification of the stiffness matrix in the current version of `Diffpack`. We are forced to use unsymmetric sub-domain solver.

We continue the documentation of the code extending the grid transfer operators to include the multiplicative Schwarz iteration too.

4.1 Multiplicative Schwarz

The standard idea to improve the performance of an additive scheme is to transform it into a multiplicative one. Instead of running all sub-domain solvers independently on data of the last iteration step, we run the sub-domain solvers in a specific order and use the latest data available.

$$B \;=\; I \;-\; (I - S_1 Q_1)\cdot(I - S_2 Q_2)\cdots(I - S_n Q_n)$$

The method was originally proposed by Schwarz [6] in the context of analytical functions using overlapping domains.

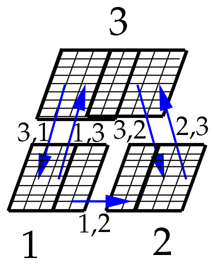

Figure 6: Multiplicative overlapping Schwarz Iteration

In the multigrid case we have used the `residual` function to update the right hand side according to the previous modifications of the solutions. This was necessary since the solution on the whole grid changed. In the overlapping Schwarz case modifications by a sub-domain correction affect only part of the global domain. This means there is a cheaper way than computing the residual on the whole domain. It is sufficient to update the data only on the new sub-domain to compute on.

In order to minimize data transfer from and to the global grid, we introduce transfer operators from one sub-domain to a neighboring sub-domain.

$$\begin{aligned}\bar{Q}_{i,j} &= \bar{Q}_j \bar{Q}_i^* \\ &= Q_j \chi_i \cdot Q_i^*\end{aligned}$$

The `transfer`, `initProj` and `modifyProj` functions are modified according to the matrix of projections `Proj` and scaling data `unity`. We set up the transfer operators such that $\bar{Q}_{i,j}$ is not the adjoint of $\bar{Q}_{j,i}$ and $\bar{Q}_i$ is not the adjoint $\bar{Q}_i^*$ as discussed above.[6]

function `MOverlap1:: transfer` MOverlap1.C

```
BooLean MOverlap1:: transfer (                  // begin_transfer
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode)
{
  if (fi!=ti)
```

[6]this code is also in `MOverlap1/`

```
    proj(fi, ti)->apply(fv, tv, NOT_TRANSPOSED, add_to_t);
  else {
    Vec(NUMT) &t = CAST_REF(tv.vec(), Vec(NUMT));
    Vec(NUMT) &u = CAST_REF(unity(ti)->vec(), Vec(NUMT));
    for (int i=1; i<=t.size(); i++)
      t(i) *= u(i);
  }
  return dpTRUE;
}                 // end_transfer
```

function MOverlap1:: initProj()

```
void MOverlap1:: initProj() // setup proj operators   begin_initp
{
  int i,j;
  for (i=1; i<=no_of_grids; i++)
    for (j=1; j<=no_of_grids; j++)
      if (i != j) {
      proj(i, j) = new ProjInterpSparse();
      proj(i, j)->rebindDOF(*dof(i), *dof(j));
      proj(i, j)->init();
    }
  modifyProj(1, no_of_grids-1);
}                          //   end_initp
```

function MOverlap1:: modifyProj()

```
void MOverlap1:: modifyProj(SpaceId i0, SpaceId i1) // begin_modify proj
{
  int i, j;
  Vec(NUMT) &uu = u->values();
  uu.fill(0);
  for (i=i0; i<=i1; i++) {
    Vec(NUMT) loc (dof(i)->getTotalNoDof ());
    loc.fill(1);
    dof(i)->fillEssBC2zero ();
    dof(i)->fillEssBC (loc);
    dof(i)->unfillEssBC2zero (); // 1 inside and 0 on the boundary

    LinEqVector uv(uu);
    proj(i, no_of_grids)->apply(loc, uv, NOT_TRANSPOSED, dpTRUE); // sum up

    for (j=i0; j<=i1; j++)
      if (i!=j)
        proj(i, j)->scale(loc, NOT_TRANSPOSED); // no data from boundary i
  }
  int s = uu.size();
  for (int k=1; k<=s; k++)
    uu(k) = 1 / uu(k); // the local contribution
  for (i=i0; i<=i1; i++) {
    Handle(Vec(NUMT)) loc = new Vec(NUMT) (dof(i)->getTotalNoDof ());
    LinEqVector ll(loc());
    proj(i, no_of_grids)->apply( uu, ll, TRANSPOSED);
    dof(i)->fillEssBC2zero ();
    dof(i)->fillEssBC (loc());
    dof(i)->unfillEssBC2zero (); // result inside and 0 on the boundary

    proj(i, no_of_grids)->scale(loc(), NOT_TRANSPOSED); // modify the transfer operator
    unity(i) = new LinEqVector(loc()); // scaling i -> i
    for (j=i0; j<=i1; j++)
      if (i != j)
        proj(i, j)->scale(loc(), NOT_TRANSPOSED); // modify the transfer operator
  }
}               // end_modify
```

In the case we are using direct sub-domain solver which do not depend on an initial guess, we can restrict the grid transfer further from one sub-domain to another sub-domain to transferring data on the boundary of the overlap.

4.2 Symmetric multiplicative Schwarz

The construction of a multiplicative Schwarz preconditioner for a conjugated gradient iteration requires symmetry. To achieve this we have to modify the Schwarz iteration procedure, cycling forwards and backwards in the order of sub-domains. We may perform the iteration on sub-domain n only once, but the other sub-domains are visited twice (except for the first sub-domain in later iteration steps).

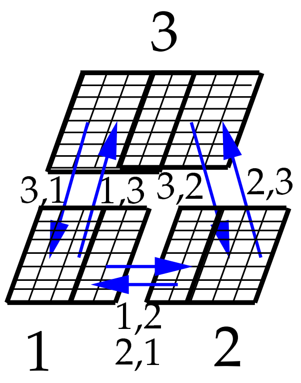

Figure 7: Symmetric Multiplicative Overlapping Schwarz Iteration

$$B_{\text{sym}} = I - (I - S_1 Q_1) \cdot (I - S_2 Q_2) \cdots (I - S_n Q_n) \cdots (I - S_2 Q_2) \cdot (I - S_1 Q_1)$$

4.3 Experiments

Exercise 11 *Additive preconditioner and iteration.*

(table 8 and 1, `test1.i` and `test1a.i`)

menu item	**answer**
subdomain	{[2,2] & [4,4] & [8,8] & [16,16]}
partition	[2,2]
overlap	1
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	AddSchwarzDD
local basic method	GaussElim

Table 8: Additive preconditioner and iteration, `test1a.i`

The first test is a comparison of the additive Schwarz iteration and the additive Schwarz preconditioner. We did the tests for the preconditioner already, so we add the test for the iteration. We can compare the number of iterations and the computing time. Does the surrounding conjugate gradient method for the preconditioner pay off? How is the dependency on the number of sub-domains? Compare the memory requirement of both methods.

Can you explain, why the conclusion whether to prefer the iteration or the preconditioner may look different than for the multigrid method? Do you have an idea, why we even can use the additive as a stand alone iterative solver?

Exercise 12 *Additive and multiplicative iteration.*

(table 9, `test2.i`)

menu item	**answer**
subdomain	[8,8]
partition	[2,2]
overlap	1
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	{AddSchwarzDD & SchwarzDD & SymSchwarzDD}
local basic method	GaussElim

Table 9: Additive and multiplicative iteration, `test2.i`

We now can compare the additive and multiplicative methods. We choose the stand alone iterative procedures. We use an exact sub-domain solver to avoid side-effects. Compare the number of iterations. Try to estimate the number of operations for each iteration for the three methods: Additive Schwarz, multiplicative Schwarz and symmetric multiplicative Schwarz iteration. Which one is the most efficient method?

Another aspect is parallel computing, of course. Running multiplicative methods in parallel usually requires some coloring strategies and some factor (the number of different colors) more sub-domains than processors. Hence additive methods are usually preferred for parallel implementations.

Exercise 13 *Additive and multiplicative preconditioner.*

(table 10, `test3.i`)

menu item	**answer**
subdomain	[8,8]
partition	[2,2]
overlap	1
basic method	ConjGrad
preconditioning type	PrecDD
domain decomposition method	{AddSchwarzDD & SymSchwarzDD}
local basic method	GaussElim

Table 10: Additive and multiplicative preconditioner, `test3.i`

We now compare additive and multiplicative methods used as preconditioners. Since we use a conjugated gradient method, we cannot use the (alternating) multiplicative Schwarz iteration, but we have to use the symmetric variant instead. Compare the number of iterations.

The previous remark on parallel computing and additive methods also applies here.

Exercise 14 *Size of overlap in the Schwarz iteration.*

(table 11, `test4.i`)

menu item	**answer**
subdomain	[4,4]
partition	[8,8]
overlap	{1 & 2 & 3}
basic method	DDIter
preconditioning type	PrecNone
domain decomposition method	SchwarzDD
#1: max error	1.0e-3

Table 11: Size of overlap in the Schwarz iteration, `test4.i`

We have a look at the question for the size of the overlap again. We now ask for the optimal overlap in the presence of an iterative Schwarz solver. The sub-domains communicate only via the overlap, there is no surrounding iteration facilitating any further data exchange. Compare the number of iterations for different overlap sizes. What kind of dependency do you observe?

How about the overall performance of the solution procedure? How does this overall performance look like for very cheap (only few iterations) sub-domain solvers? Why is the issue of the overlap more important for the iterative Schwarz method than for the Schwarz preconditioner?

Exercise 15 *Multiplicative preconditioner and iteration.*

(table 10 and 9)

Finally we can compare the symmetric multiplicative Schwarz method used as an iterative procedure and as a preconditioner. Since we did all the necessary tests already, we can compare previous results. Look at the number of iterations. Does the conjugated gradient method improve the efficiency? Does it pay off? Compare this to your findings for the additive Schwarz iteration and preconditioner.

5 Nonlinear Schwarz iteration

If we want to solve nonlinear problems by the overlapping Schwarz method, we have several possiblities:

Newton-Schwarz

We can use the overlapping Schwarz method discussed so far as a linear solver or a preconditioner for a linear solver inside some nonlinear solution procedure like Newton iteration or successive iteration. Procedures like this may be called Newton-Schwarz methods. Most of the work is done in assembling matrices and solving linear problems which both can be nicely done for example in parallel. The nonlinear outer iteration inherits the parallel performance from the inner Schwarz iteration. The open question is the termination criterion for the inner loop, the coupling of the control for both linear inner and nonlinear outer loop. We leave the actual implementation of such a method to the user, since it is mainly the combination of codes `NlElliptic` and `Overlap3` already presented [8].

Schwarz-Newton

Instead we present the opposite approach of a nonlinear Schwarz iteration. The idea is to use the mechanism of splitting the domain into overlapping sub-domains and the pattern of grid transfer between the sub-domains and apply this mechanism directly to the nonlinear problem. The sub-problems turn out to be nonlinear problems now, which are solved approximately by some nonlinear iterative solvers like Newton iteration or successive iteration. The term Schwarz-Newton may be used for this. The major difference to the overlapping Schwarz iteration discussed so far is that the linear sub-domain solvers are replaced by nonlinear sub-domain solvers. The Schwarz algorithm, the grid transfer and the boundary conditions remain the same.

5.1 Code

Based on the nonlinear template `NlElliptic` of [8] we create a simulator for the nonlinear overlapping Schwarz iteration. We combine features of the nonlinear multi-grid implementation `NlMultiGrid1` and the linear overlapping Schwarz iteration

MOverlap1 and mainly merged the code. It may be useful to consult the documentation of these codes given already in addition to the following remarks.[7]

We introduce three different test examples, a linear one, one problem with nonlinear right hand side and one problem with nonlinear operator.

The main structure of the code is taken from the nonlinear multigrid example NlMultiGrid1. The sub-domain solvers are built upon a class NlLevel. It is instantiated for each sub-domain and contains the grid, the nonlinear assembly procedure, the nonlinear solvers and linear solvers to be used in the interior loop of the nonlinear ones.

The grid transfer operators and the interface to the nonlinear overlapping Schwarz iteration as well as handles to the sub-domains are contained in the main class NlOverlap1. This splitting (borough from NlMultiGrid1) is necessary because we use different nonlinear solvers at the same time (on different levels) and the nonlinear interface in Diffpack are based on inheritance of NonLinEqSolverUDC.

NlOverlap1.h

```
#ifndef NlOverlap1_h_IS_INCLUDED
#define NlOverlap1_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(DegFreeFE) dof;   // mapping: nodal values <-> linear system unknowns
  Handle(FieldFE) u;       // finite element field, the primary unknown

  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 fillEssBC0();               // set zero boundary conditions
  virtual void fillEssBC (Vec(NUMT)& x);   // set given boundary conditions
  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
```

[7] you will find the code in NlOverlap1/

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

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

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

  virtual BooLean solveSubSystem (LinEqVector& b, LinEqVector& x,
  StartVectorMode start, DDSolverMode mode);
  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 NlOverlap1 : 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
  MatSimplest(Handle(Proj))        proj;       // projection operators
  VecSimplest(Handle(LinEqVector)) unity;      // partition of unity

  virtual void initProj();                     // set up projection matrices
  virtual void modifyProj(SpaceId i0, SpaceId i1); // modify them
public:
  NlOverlap1 ();
 ~NlOverlap1 () {}

  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);
  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 assembly is done in the standard way in `NlLevel`. We do not need the special options for introduction of Dirichlet values later to assembly, because we have to assemble the matrices each time the nonlinear sub-domain solver requires an update. At this time the Dirichlet values imposed by the Schwarz iteration are known already. The nonlinear solvers include a Newton iteration and a successive substitution procedure. The standard linear sub-problem solver are available.

The grid generation, boundary indicators in `NlLevel` and grid transfer operators in `NlMultiGrid1` are copied from our previous implementation of the Schwarz iteration `MOverlap1`. The additive, the multiplicative and the symmetric multiplicative version of the overlapping Schwarz iteration are available.

NlOverlap1.C

```
#include <NlOverlap1.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, String& boInds)
{
  grid.rebind (new GridFE());              // create empty grid object
  PreproBox p;
  p.geometryBox() .scan (geometry);
  p.partitionBox().scan (partition);
  p.generateMesh (grid());                 // fill grid
  grid->redefineBoInds(boInds);             // redefine boundary indicators

  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:: fillEssBC0 ()
{
  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 on any boundary.
}

void NlLevel:: fillEssBC (Vec(NUMT)& x)
{
  dof->initEssBC ();            // init for assignment below
  int nno = grid->getNoNodes(); // no of nodes
  for (int i = 1; i <= nno; i++)
    if (grid->BoNode (i))       // is node i subj. to any boundary indicator?
      if (grid->BoNode (i, 1))
        dof->fillEssBC (i, 0.);    // u=0 at nodes on the boundary
      else if (grid->BoNode (i, 2))
        dof->fillEssBC (i, x(i)); // inner boundary
}

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
  fillEssBC(nonlin_solution());

  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());

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

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

int NlLevel:: getWorkSolve () const
{//  return lineq->getLinEqSystem ().getWork();
  return 0;
}

real NlLevel:: getStorageSolve () const
{//  return lineq->getLinEqSystem ().getStorage();
  return 0;
}

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_);}
//______________________________________________________________________

NlOverlap1:: NlOverlap1 () {}

void NlOverlap1:: 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 NlOverlap1:: 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,
                "subdomain",    // menu command/name
                "subdomain",    // command line options: +partition
                "string like 2,4,2",
                "[4,4]",        // default answer: 4x4 division (5x5 nodes)
                "S");           // valid answer: string

  menu.addItem (level,
                "partition",   // menu command/name
                "partition",   // command line options: +refinement
                "string like [2,2,2] = 8 domains",
                "[2,2]",        // default answer: 2x2 domains
                "S");           // valid answer: string

  menu.addItem (level,
                "overlap",      // menu command/name
                "overlap",      // command line option: +overlap
                "",
                "1",            // default answer
                "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("local");
  NlLevel             ::defineStatic (menu, level);
  menu.unsetCommandPrefix();
}

void NlOverlap1:: scan (MenuSystem& menu)
{
  // load answers from the menu:
  int i;
  int nsd = menu.get ("no of space dimensions").getInt();
int overlap = menu.get ("overlap").getInt();

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

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

Ptv(int) dom(nsd);
for (i = 1; i <= nsd; i++)
  dom(i) = subdom(i) * part(i) - overlap * (part(i)-1);

no_of_grids = 1;
for (i = 1; i<= nsd; i++)
  no_of_grids *= part(i);
no_of_grids += 1;   // compute no_of_grids

proj.redim       (no_of_grids, no_of_grids);
unity.redim      (no_of_grids-1);
level.redim      (no_of_grids);

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

int p = menu.get ("problem").getInt();
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("NlOverlap1:: scan","illegal problem number");
  }

String elm_tp = menu.get ("element type");

for (i=1; i<=no_of_grids; i++) {
  int j;
// ---- 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 = "[";
    int k = i-1;
    for (j = 1; j <= nsd; j++) {
      real x0, x1;
      if (i<no_of_grids) {
int ix = k % part(j); // split into row, column ...
k = k / part(j);
x0 = (ix * (subdom(j) - overlap)            ) / (real) dom(j);
x1 = (ix * (subdom(j) - overlap) + subdom(j)) / (real) dom(j);
      } else
{ x0 = 0.; x1 = 1.;}
      geometry += aform("[%g,%g]", x0, x1);  grading  += "1"; // [.3,.7]x[0,1]
      if (j < nsd) {
geometry += "x";  grading  += ",";
      }
    }
    grading += "]";

    String part_s = "[";   // partition string e.g. [4,4]
    for (j=1; j<=nsd; j++) {
      int n;
      if (i<no_of_grids) n = subdom(j);
      else               n = dom(j);
      part_s += aform("%d",n);
      if (j<nsd)
part_s += ",";
    }
    part_s += "]";

    String partition = aform("d=%d elm_tp=%s div=%s grading=%s",
     nsd,elm_tp.chars(),part_s.chars(),
     grading.chars());

    String boInd_g = "nb=2 names= global inner 1=(";
    String boInd_i = "), 2=(";
    k = i-1;
    for (j = 1; j <= nsd; j++) {
      int ix = k % part(j) + 1; // split into row, column ...
      k = k / part(j);

      String b1 = aform("%d ", j);
      if ((ix==part(j))||(i==no_of_grids))
        boInd_g += b1;
      else boInd_i += b1;

      String b0 = aform("%d ", j+nsd);
      if ((ix==1)||(i==no_of_grids))
        boInd_g += b0;
      else boInd_i += b0;
    }
    boInd_g += boInd_i;
    boInd_g += ")";

    menu.setCommandPrefix("local");
    level(i)->scan (menu, geometry, partition, boInd_g);
    menu.unsetCommandPrefix();
    level(i)->attachSol (ddsolver(), i);
    //if (i==no_of_grids)
    level(i)->attachRhs (ddsolver(), i);
```

```
  }
  initProj();

  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 NlOverlap1:: initProj() // setup proj operators
{
  int i,j;
  for (i=1; i<=no_of_grids; i++)
    for (j=1; j<=no_of_grids; j++)
      if (i != j) {
      proj(i, j) = new ProjInterpSparse();
      proj(i, j)->rebindDOF(level(i)->getDof(), level(j)->getDof());
      proj(i, j)->init();
    }
  modifyProj(1, no_of_grids-1);
}

void NlOverlap1:: modifyProj(SpaceId i0, SpaceId i1)
{
  int i, j;
  Vec(NUMT) uu (level(no_of_grids)->getDof().getTotalNoDof ());
  uu.fill(0);
  for (i=i0; i<=i1; i++) {
    Vec(NUMT) loc (level(i)->getDof().getTotalNoDof ());
    loc.fill(1);
    level(i)->getDof().fillEssBC2zero ();
    level(i)->getDof().fillEssBC (loc);
    level(i)->getDof().unfillEssBC2zero (); // 1 inside and 0 on the boundary

    LinEqVector uv(uu);
    proj(i, no_of_grids)->apply(loc, uv, NOT_TRANSPOSED, dpTRUE); // sum up

    for (j=i0; j<=i1; j++)
      if (i!=j)
        proj(i, j)->scale(loc, NOT_TRANSPOSED); // no data from boundary i
  }
  int s = uu.size();
  for (int k=1; k<=s; k++)
    uu(k) = 1 / uu(k); // the local contribution
  for (i=i0; i<=i1; i++) {
    Handle(Vec(NUMT)) loc = new Vec(NUMT) (level(i)->getDof().getTotalNoDof ());
    LinEqVector ll(loc());
    proj(i, no_of_grids)->apply( uu, ll, TRANSPOSED);

    level(i)->getDof().fillEssBC2zero ();
    level(i)->getDof().fillEssBC (loc());
    level(i)->getDof().unfillEssBC2zero (); // result inside and 0 on the boundary

    proj(i, no_of_grids)->scale(loc(), NOT_TRANSPOSED); // modify the transfer operator
    unity(i) = new LinEqVector(loc()); // scaling i -> i
    for (j=i0; j<=i1; j++)
      if (i != j)
        proj(i, j)->scale(loc(), NOT_TRANSPOSED); // modify the transfer operator
  }
}

void NlOverlap1:: solveProblem () // main routine of class NlOverlap1
{
  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("NlOverlap1::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 NlOverlap1:: 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 NlOverlap1:: getNoOfSpaces() const
{ return no_of_grids; }

BooLean NlOverlap1:: solveSubSystem (
    LinEqVector& b, LinEqVector& x,
    SpaceId space, StartVectorMode start, DDSolverMode mode)
{
  BooLean res = level(space)->solveSubSystem(b, x, start, mode);
  return res;
}

BooLean NlOverlap1:: transfer (
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode)
{
  if (fi!=ti)
    proj(fi, ti)->apply(fv, tv, NOT_TRANSPOSED, add_to_t);
  else {
    Vec(NUMT) &t = CAST_REF(tv.vec(), Vec(NUMT));
    Vec(NUMT) &u = CAST_REF(unity(ti)->vec(), Vec(NUMT));
    for (int i=1; i<=t.size(); i++)
      t(i) *= u(i);
  }
  return dpTRUE;
}

int NlOverlap1:: getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork) const
{
  if (fi!=ti) return proj(fi, ti)->getWork();
  if (fi==no_of_grids) return 0;
  return unity(fi)->size();
}

real NlOverlap1:: getStorageTransfer (SpaceId fi, SpaceId ti) const
{
  if (fi!=ti) return proj(fi, ti)->getStorage();
  if (fi==no_of_grids) return 0;
  return unity(fi)->size();
}

int NlOverlap1:: getWorkSolve (SpaceId space, const PrecondWork) const
{ return level(space)->getWorkSolve(); }

real NlOverlap1:: getStorageSolve (SpaceId space) const
{ return level(space)->getStorageSolve(); }

String NlOverlap1:: comment ()
{ return "NlOverlap1 nonlinear Schwarz iteration test"; }
```

5.2 Experiments

In some sense, we can redo most of the exercises proposed for the linear version of the overlapping Schwarz method (exercises 4–7, 12). We can always verify the properties of the linear solvers running the linear test case. The idea then is to look for the differences for the nonlinear test cases. The second part of the exercises is concerned with the inexact solution of sub-domain problems which is natural for nonlinear problems.[8]

Exercise 16 *Size of overlap.*

(table 12, `test1.i`)

All exercises with nonlinear can be done with all test problems available. The question is, how the performance of a method deteriorates by the nonlinearity. So it is of interest to compare the results of the linear problem with both nonlinear problems. Of course a comparison of the performance for different types of nonlinearity is also of interest. This includes the two test cases of a nonlinear right hand side and a nonlinear coefficient of the differential operator. This can only serve as some example of nonlinear problems, but may give some hints for more general problems.

[8]you will find the input parameters in **`NlOverlap1/Verify/`**

menu item	**answer**
problem	{1 & 2 & 3}
subdomain	[4,4]
partition	[2,2]
overlap	{1 & 2 & 3}
no of space dimensions	2
element type	ElmB4n2D
nonlinear iteration method	NonLinDD
max estimated nonlinear error	1.0e-2
nonlinear iteration stopping criterion	3
domain decomposition method	SchwarzDD
local basic method	SOR
local max iterations	4
local preconditioning type	PrecNone
local nonlinear iteration method	SuccessiveSubst
local max nonlinear iterations	4
local nonlinear iteration stopping criterion	3
local convergence reports	0

Table 12: Size of overlap, `test1.i`

We can redo many of the exercises already performed for linear additive and multiplicative Schwarz methods. Of course there is a big similarity of the linear and nonlinear methods.

The first test is related to the size of the overlap. Compare the number of iterations for different overlap sizes. Relate the numbers to the linear and nonlinear test cases. Observe the very moderate tolerance we are using, since nonlinear methods (in this demonstration implementation) tend to be quite slow. Implementation specifically dedicated to a kind of nonlinearity or a certain solution algorithm are of course much more efficient (and numerically equivalent). So we only compare iteration numbers and number of operations rather than computing times.

Exercise 17 *Number of sub-domains.*

(table 13, `test2.i`)

We vary the number of sub-domains. We use a moderate precision nonlinear subdomain solver and compare the effect of different nonlinearities. Compare the number of iterations. What do you observe? Is there a difference between the linear and the nonlinear test cases? Do the conclusions correspond with previous observations on the number of sub-domains?

Exercise 18 *Additive and multiplicative iteration.*

(table 14, `test3.i`)

menu item	**answer**
problem	{1 & 2 & 3}
subdomain	[4,4]
partition	{[2,2] & [3,3] & [4,4]}
overlap	1
nonlinear iteration method	NonLinDD
domain decomposition method	SchwarzDD
local basic method	SOR
local max iterations	4
local nonlinear iteration method	SuccessiveSubst
local max nonlinear iterations	4

Table 13: Number of sub-domains, `test2.i`

menu item	**answer**
problem	{1 & 2 & 3}
subdomain	[5,5]
partition	[3,3]
overlap	1
nonlinear iteration method	NonLinDD
domain decomposition method	{AddSchwarzDD & SchwarzDD & SymSchwarzDD}
local basic method	SOR
local max iterations	4
local nonlinear iteration method	SuccessiveSubst
local max nonlinear iterations	4

Table 14: Additive and multiplicative iteration, `test3.i`

The next test is the comparison of the additive, the multiplicative and the symmetric multiplicative Schwarz iteration for nonlinear problems. We did this comparison for linear problems already. We use some moderate precision sub-domain solver. Compare the number of iterations and and estimate for the number of operations. How does the additive method compare to the multiplicative ones with respect to the operation count and to the domain of convergence/ robustness?

Exercise 19 *Inexact sub-domain solver.*

(table 15, `test4.i`)

The next exercises deal with the sub-domain solvers. The present implementation uses some steps of a nonlinear solver calling some steps of a linear solver as approximative sub-domain solvers. Of course it would be too expensive to solve nonlinear sub-problems exactly. The outer Schwarz method is able to cope with inexact sub-domain solvers, so we do not want to waste effort on the sub-domains. The question

menu item	**answer**
problem	{1 & 2 & 3}
subdomain	[10,10]
partition	[2,2]
overlap	1
nonlinear iteration method	NonLinDD
domain decomposition method	SchwarzDD
local basic method	SOR
local max iterations	1
local nonlinear iteration method	SuccessiveSubst
local max nonlinear iterations	{1 & 4 & 8}

Table 15: Number of sub-domains, `test4.i`

now is, how precise the sub-domain solvers have to be in order to achieve a good overall performance.

We are looking at this question in three steps: We vary the number of nonlinear solution steps on a sub-domain using a very poor linear solver. Compare the number of iterations and an estimate for the overall efficiency? How many nonlinear iterations seem to be optimal? Is there a difference between the different nonlinearities?

Exercise 20 *Inexact nonlinear sub-domain solver.*

(table 16, `test5.i`)

menu item	**answer**
problem	{1 & 2 & 3}
subdomain	[10,10]
partition	[2,2]
overlap	1
nonlinear iteration method	NonLinDD
domain decomposition method	SchwarzDD
local basic method	SOR
local max iterations	{1 & 10 }
local nonlinear iteration method	SuccessiveSubst
local max nonlinear iterations	1

Table 16: Inexact nonlinear sub-domain solver, `test5.i`

Now we fix the number of nonlinear iterations on a sub-domain and vary the number of linear iterations. This means that we use poor nonlinear sub-domain solvers and even vary the quality of the linear algebra inside. Compare the number of iterations. What is the optimal parameter? What is the difference between nonlinear and linear problems?

Since nonlinear iterations are usually more expensive than linear ones, one uses linear iterations rather than nonlinear ones. This may of course affect the quality of the overall nonlinear Schwarz iteration. For the linear case the distribution of linear and nonlinear iteration does not play a role, while the total number of iterations is important.

Exercise 21 *Different nonlinear sub-domain solvers.*

(table 17, `test6.i`)

menu item	**answer**
problem	{1 & 2 & 3}
subdomain	[10,10]
partition	[2,2]
overlap	1
nonlinear iteration method	NonLinDD
domain decomposition method	SchwarzDD
local basic method	SOR
local max iterations	4
local nonlinear iteration method	{SuccessiveSubst & NewtonRaphson}
local max nonlinear iterations	4

Table 17: Inexact nonlinear sub-domain solver, `test6.i`

The last exercise compares different nonlinear sub-domain solvers. We use the successive substitution (Picard iteration) and the Newton-Raphson iteration. The Newton iteration is considered to be faster in the vicinity of the solution, while the successive substitution is more robust and cheaper.

We use a moderate precision linear solver and a few steps of the nonlinear solution procedure. Compare the number of iterations and the number of operations. How do you compare both methods with respect to efficiency and robustness?

6 Conclusion

In this report we have demonstrated the use of overlapping Schwarz methods in `Diffpack`. This type of domain decomposition can be used as iterative linear equation solver, nonlinear equation solver and most efficiently as preconditioner for iterative equation solvers. The Schwarz method is based on equation solvers on the overlapping sub-domains. All linear and nonlinear equation solvers available in `Diffpack` can be utilized here, including multigrid and domain decomposition methods itself.

Overlapping domain decomposition can be used to solve problems defined on complicated domains which can be constructed from simple shaped domains with efficient sub-domain solvers. However, the main field of application of overlapping Schwarz methods is parallel computing, where the different sub-domains reside on different processors. The Schwarz algorithm manages the communication between the sub-domains. Both subjects, complicated geometry and parallel computing will be covered in subsequent reports.

We have particularly emphasized the flexibility of the overlapping Schwarz method and its similarity to the multigrid implementation in `Diffpack`. For practical usage it is essential to be able to make good choice of the avaliable parameters. This is not only true for general choices like additive and multiplicative methods and the use of a coarse grid, but especially for details like the parameters of sub-domain solvers and the partition of the global domain. The exercises could serve as some guidance even for more complex problems to solve than treated in this introductory report.

References

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

[2] M. Dryja and O. Widlund, *An Additive Variant of the Schwarz Alternating Method for the Case of Many Subregions*, Courant Institute, New York, 1987. Technical Report 339.

[3] D. E. Keyes and J. Xu, *Domain Decomposition Methods in Scientific and Engineering Computing: Proceedings of the Seventh International Conference on Domain Decomposition*, vol. 180 of Contemporary Mathematics, American Mathematical Society, Providence, Rhode Island, 1994.

[4] H. P. Langtangen, *Getting started with finite element programming in Diffpack*, Tech. Rep. STF33 A94050, SINTEF Informatics, Oslo, 1994.

[5] H. P. Langtangen, G. Pedersen, and W. Shen, *Finite element preprocessors in Diffpack*, Tech. Rep. STF33 A94051, SINTEF Informatics, Oslo, 1994.

[6] H. A. Schwarz, *Über einige Abbildungsaufgaben*, Ges. Math. Abh., 11 (1869), pp. 65–83.

[7] B. Smith, P. Bjørstad, and W. Gropp, *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, New York, 1996.

[8] G. W. Zumbusch, *Multigrid methods in Diffpack*, Tech. Rep. STF42 F96016, SINTEF Applied Mathematics, Oslo, 1996.