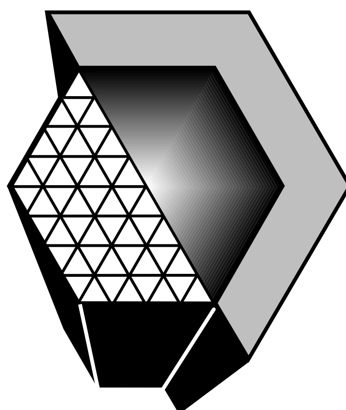
Schur Complement Domain Decomposition Methods in Diffpack

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/`.

Copyright © SINTEF, Oslo
November 22, 1996

Permission is granted to make and distribute verbatim copies of this report provided the copyright notice and this permission notice is preserved on all copies.

Abstract

The report gives an introduction to the Schur complement domain decomposition solvers in `Diffpack`. It is meant as a tutorial for the use of iterative solution methods of equation systems arising in the discretization of partial differential equations. Schur complement iterative solvers are discussed, without and with preconditioners. They are also referred to as iterative sub-structuring methods or non-overlapping domain decomposition methods. Domain decomposition methods are well suited and efficient equation solvers on parallel computers. Schur complement methods are also advantageous if there are abrupt changes in the coefficients of the differential operator due to abrupt changes in material properties. We provide an introduction to the implementation and use of such methods in `Diffpack`. We cover the basic Schur complement method along with preconditioners of eigen-decomposition, BPS, wire-basket and Neumann-Neumann type (with coarse grid). The first steps are guided by a couple of examples and exercises. We also want to refer to the related tutorials on overlapping domain decomposition [Zum96b] and on multigrid [Zum96a] methods in `Diffpack`.

Contents

1 Introduction 1

2 Schur complement iteration 3

2.1 Definition 3

2.2 Code 6

3 Neumann-Neumann preconditioner for the Schur complement 19

3.1 Definition 20

3.2 Code 21

3.3 Stabilizing pure Neumann problems 27

3.4 Coarse grid acceleration 33

4 Eigen-decomposition preconditioner for the Schur complement 39

5 BPS and Wire-basket preconditioner for the Schur complement 43

5.1 Definition 43

5.2 Code 44

6 Inexact Schur complement regarded as a preconditioner 60

7 Conclusion 62

References 64

Schur Complement 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 system of equations. While most operations in finite elements have linear complexity and are well suited for parallel computing with local communication patterns (like matrix assembly), standard linear algebra has a higher complexity and more expensive communication patterns. Hence the complexity of linear algebra tends to dominate any large scale simulation.

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

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

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

divide and conquer

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

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 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.

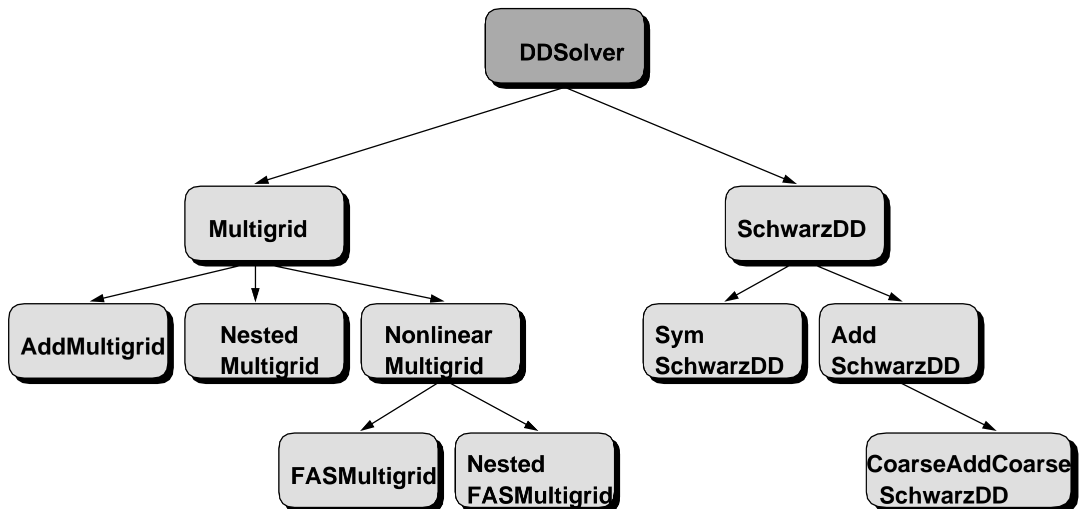


Figure 1: Hierarchy of multigrid and domain decomposition methods

We will discuss non-overlapping domain decomposition methods. They differ in the way of splitting the problem into sub-problems. The different instances of the methods mainly differ in the way putting the solutions together. This is a decision left to the user. It will turn out to be problem dependent. The domain decomposition concept in general is only a recipe to construct a solution algorithm. In full generality there are only guidelines and hints for a choice of components.

recipe

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

exercises

For further reading and for theory we will refer to some of the literature, especially the introduction [SBG96] and the overview [CM94] and references therein. We also refer to the proceedings of the "Domain Decomposition" [KX95, GPSW96].

references

We assume familiarity with some of the basic concepts of `Diffpack` [BL96]. We will use and modify some examples presented in [Lan94]. Some more detailed description of the programming interface of the domain decomposition and multigrid software

in `Diffpack` can be found in [Zum96a]. The related overlapping Schwarz domain decomposition methods are described in the tutorial [Zum96b]. It may also 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 Schur complement iteration

2.1 Definition

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

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

The idea is now to partition the global domain into non-overlapping sub-domains and to compute the solution on the interface between the sub-domains. Each domain is discretized the same way the global domain is discretized.

We partition the global domain Ω into a pairwise disjoint set of sub-domains Ω_i.

$$\overline{\bigcup_j \Omega_j} = \overline{\Omega}$$

We define the interior boundaries to be

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

The interface $\mathcal{I}$ is defined as the sum of all interior boundaries

$$\mathcal{I} = \bigcup_j \Gamma_j$$

Non-overlapping means in some sense that we have an overlap of the sub-domains of the size zero.

We discretize the equations on the domain Ω by finite elements or finite differences. The observation now is that the interface decouples the discretizations on the different sub-domains. In the discretized equation system some of the unknowns in a sub-domain are coupled to unknowns on the interface, but none of the unknowns in one sub-domain are coupled to unknowns of another sub-domain. To guarantee this decoupling, we restrict ourselves to matching discretzations. The discretization has to be aligned with the boundaries Γ_i and a finite element is not allowed to intersect the interface $\mathcal{I}$. An element may share a complete face, an edge or a node with the interface, but it is not allowed to share only part of a face or an edge. The discretizations on different sub-domains match.

$$A = \begin{pmatrix} A_{11} & A_{1\mathcal{I}} & 0 \\ A_{\mathcal{I}1} & A_{\mathcal{I}\mathcal{I}} & A_{\mathcal{I}2} \\ 0 & A_{2\mathcal{I}} & A_{22} \end{pmatrix}$$

To fulfill the decoupling condition, the interface for higher order approximations or higher order operators consequently may consist of a strip of unknowns rather than a thin line Γ_i, depending on the discretzation.

The idea is to eliminate the unknowns in each sub-domain and to derive an equation for the unknowns on the interface. This system is also called Schur complement. It may be solved directly called direct sub-structuring method. This means numerically just a special ordering of the unknowns for Gaussian elimination. In practice it turns out to be a strategy to perform a factorization of a matrix, which is too large to be hold in the computer memory. The computer must be able to store a local matrix block, not the global matrix. The alternative to direct sub-structuring is to solve the Schur complement iteratively with some (preconditioned) Krylov method. We will discuss some preconditioners for this interface problem later.

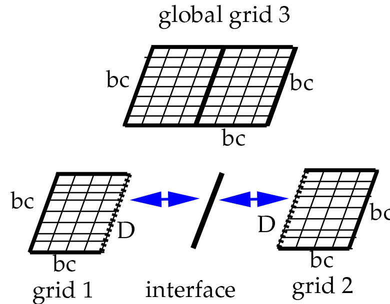


Figure 2: Non-Overlapping Domain Decomposition.

We reorder the unknowns. First come the interior nodes of a sub-domain, domain by domain (domain 1 and 2 here) and last are the nodes of the interface ($\mathcal{I}$). We split the global stiffness matrix in the same way:

$$A = \begin{pmatrix} A_{11} & 0 & A_{1\mathcal{I}} \\ 0 & A_{22} & A_{2\mathcal{I}} \\ A_{\mathcal{I}1} & A_{\mathcal{I}2} & A_{\mathcal{I}\mathcal{I}} \end{pmatrix}$$

The essential observation here is that the sub-domain problems are independent, c.f. the zero blocks in the matrix representation. This is due to the requirement that discretization and interface match.

$$A \begin{pmatrix} x_1 \\ x_2 \\ x_{\mathcal{I}} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_{\mathcal{I}} \end{pmatrix}$$

Eliminating the interior unknowns in each domain leads to the Schur complement system for the unknowns on the interface $x_{\mathcal{I}}$

$$S\, x_{\mathcal{I}} = b_{\mathcal{I}} - \sum_{k=1}^{2} A_{\mathcal{I}k}\, A_{kk}^{-1}\, b_k$$

with the Schur complement matrix S defined as

$$S = A_{\mathcal{I}\mathcal{I}} - \sum_{k=1}^{2} A_{\mathcal{I}k}\, A_{kk}^{-1}\, A_{k\mathcal{I}}$$

The matrix S is symmetric and positive definite as is the matrix A.

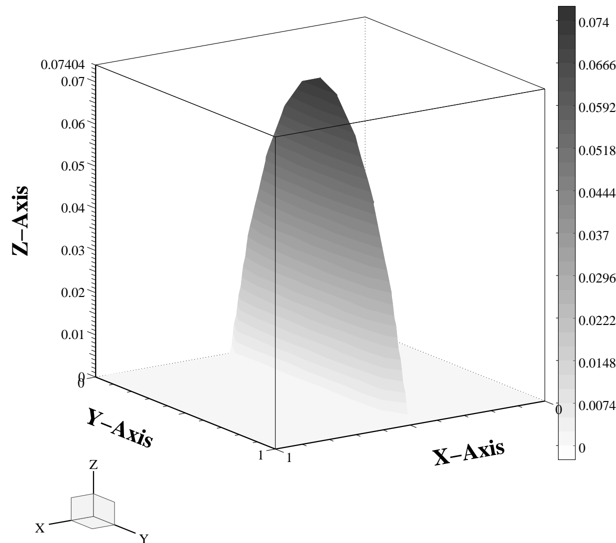


Figure 3: Solution on the interface.

Instead of solving the global system with matrix A, we seek for a solution of the smaller system with matrix S. Once we got a solution $x_{\mathcal{I}}$, we can easily compute the global solution by back substitution

$$x_k = A_{kk}^{-1}(b_k - A_{k\mathcal{I}}x_{\mathcal{I}})$$

This of course is also true for splitting the global domain into more than two sub-domains.

Applying a Krylov iteration to compute $x_{\mathcal{I}}$ requires multiplications by S. This means solving local subproblems on each sub-domain to evaluate $A_{kk}^{-1}y$. We can use direct or iterative solvers to solve the problems independently for each sub-domain. First we have to compute the right hand side for the interface system, solving $A_{kk}^{-1}b_k$. When we have computed a solution $x_{\mathcal{I}}$ on the interface we have to solve sub-domain problems a last time to compute values on the whole domain x. Using a Krylov iteration with j matrix multiplications, we have to solve a system $A_{kk}^{-1}y$ on each sub-domain $(j+2)$-times.

The solution of sub-problems with matrix A_{kk} can be interpreted as Dirichlet problems (for$\Gamma = \partial\Omega$). Any solution procedure for symmetric positive definite matrices stemming from the discretization with finite elements may be used here. In parallel computing typically these computations for different sub-domains are distributed to different processors.

We can state some differences to the overlapping domain decomposition methods now: The primary unknowns are only on the interface which is much less than the number of all unknowns on the domain. The Krylov iteration only operates on the interface. Each matrix multiplication requires the solution of a local problem on each

sub-domain. The local solutions have to be precise enough. The Schur complement matrix usually is not assembled, so matrix parts like diagonal entries are not available. The final result on the interface needs to be extended onto the whole domain.

2.2 Code

We start with the `Elliptic1` example simulator described in [Lan94]. We want to modify it to be able to operate on the Schur complement. We apply the conjugated gradient iteration to the Schur complement system.[1]

We use a special type of `LinEqMatrix` called `MatSchur`. This matrix has a very restricted number of methods available. There are no matrix decompositions available. The main method is matrix multiplication `prod`. This is delegated to a user function which is attached to `MatSchur` via the `MatSchurUDC` interface. This means we have to implement the matrix vector product `prod` now as a part of the `MatSchurUDC` interface and attach it to the matrix.

We do not assemble the global matrix and split it, but we assemble local matrices on each sub-domain. We use a homogeneous Neumann boundary condition on interior boundaries. Adding the local matrices similar to a matrix assembly procedure in finite elements we could create the global stiffness matrix. This process is called sub-assembly since each sub-matrix contains only part of the data. However the part of a local matrix connected with the interior nodes is in fact complete. In `initMatrices()` we extract this part A_{kk} `mat_inner` from the local matrices needed for the computation. Also the matrices coupling the interior of the domain with the interface is complete. We can extract $A_{\mathcal{I}k}$ and $A_{k\mathcal{I}}$ which are transposed. We do not store both of them `mat_trans`. The interface matrix $A_{\mathcal{I}\mathcal{I}}$ `mat_interf` is the sum of the appropriate parts of the local matrices.

The translation tables of the numbering schemes from the assembled local matrices to the sub-domain matrices and the interface are stored in `num_inner` and `num_interf`. The translation from interface to the sub-domain matrices is done via `num_trans`. The tables are set up in `initInterface()`.

We create linear equation solvers for each Dirichlet problem `sub_solve` using the matrix `mat_inner` attached to `system`. The global linear equation system is managed by `lineq` now using the `MatSchur` matrix. There are only linear solvers available which solely rely on matrix multiplication. Accordingly the solution and right hand side vectors `linsol` and `linrhs` are of dimension of the interface `interface_dim`. The right hand side is initialized in `initRhs()` from the right hand side components `rhs_inner` and `rhs_interf` which were constructed similarly to the matrix components. The final global solution is computed in `projSol()` based on the solution on the interface. It is using projections `proj` from the sub-domains to a global grid to provide a global solution ready for output.

Schur1.h

```
// prevent multiple inclusion of Schur1.h
#ifndef Schur1_h_IS_INCLUDED
```

[1] this code is also in `Schur1/`

```
#define Schur1_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 <MatSchur_real.h>
#include <VecSimplest_VecSimple_int.h>
#include <Proj.h>

typedef int SpaceId;

class Schur1 : public FEM, public MenuUDC, public Store4Plotting, public MatSchurUDC(NUMT)
{
protected:
  // general data:
  Handle(FieldFE)   u;      // finite element field, the primary unknown
  Vec(real)         linsol; // solution of Schur system
  Vec(real)         linrhs; // rhs of Schur system, updated rhs_interf

  // sub grid related data:
  int no_of_grids;                        // number of domains
  int global_grid;                        //
  int local_grid;                         //
  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
  VecSimplest(Handle(Proj))              proj;      // projection operators
  VecSimplest(Handle(prm(Matrix(NUMT)))) mat_prm;   // Matrix parameters

  // sub problem related data:
  int                                    interface_dim;// dim of Schur system
  VecSimple(int)                         inner_dim; // dim of local system
  VecSimplest(Handle(Matrix(NUMT)))      mat_inner; // Matrix a_11
  VecSimplest(Handle(Matrix(NUMT)))      mat_trans1i; // Matrix a_1i
  VecSimplest(Handle(Matrix(NUMT)))      mat_transi1; // Matrix a_i1
  Handle(Matrix(NUMT))                   mat_interf;// Matrix a_ii
  VecSimplest(Handle(LinEqVector))       rhs_inner; // b_1
  Handle(LinEqVector)                    rhs_interf;// b_i
  VecSimplest(VecSimple(int))            num_inner; // projection operators
  VecSimplest(VecSimple(int))            num_trans; // projection operators
  VecSimple(int)                         num_interf; // projection operators

  // 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 initInterface();       // setup interface numbering
  virtual Handle(Matrix(NUMT)) initMatrix(int i);// setup stiffness matrix i
  virtual void initMatrices();        // setup stiffness matrices on sub domains grids
  virtual void initRhs();             // compute linrhs by local rhs
  virtual void projSol();             // compute u() by local solutions

public:
  Schur1 ();
 ~Schur1 () {}

  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

  // MatSchurUDC:
  virtual void prod
    (const Vector(NUMT)& xb,
           Vector(NUMT)& yb,
           TransposeMode tpmode = NOT_TRANSPOSED,
           BooLean add_to_yb = dpFALSE);

  String  comment ();
};
#endif
```

The code operates on the unit (hyper-) cube in any dimension as usual. In the case of extending it to more complicated domains, one has to be careful with the matching condition of partition and global grid. In contrast to the previous methods, the sub-domain grids have to be compatible.

All sub-domains are of the same size, because all are initialized with the same `subdomain` string read from the menu. The number of sub-domains and the pattern they are arranged is determined by the `partition` string. The properties of the local Dirichlet solvers can be chosen via the `local LinEqAdm` menu, while the iteration on the Schur system is `LinEqAdm`.

Creating the matrices and grids we use one (1) boundary indicator for the Dirichlet boundary Γ and another one $((2) \setminus (1))$ for the interface $\mathcal{I}$. We base the translation tables on these boundary indicators. The matrices are assembled in a standard fashion. However we have to do some initialization on our own. The matrices are split using the `splitMatrix` mechanism providing the translation tables.

The local Dirichlet problems are solved in `initRhs()`, `projSol()` and `prod` using `sub_solve` and `system`.

Schur1.C

```
#include <Schur1.h>
#include <PreproBox.h>
```

```
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <createElmDef.h>        // for calling hierElmDef in Schur1::define
#include <createMatrix_real.h>   // creating stiffness matrices
#include <createLinEqSolver.h>   // creating sub domain solver
#include <splitMatrix_real.h>    // splitting stiffness matrices

Schur1:: Schur1 () {}

void Schur1:: 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 Schur1:: 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,
              "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

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

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

void Schur1:: 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(global_grid)(),"u")); // allocate, with field name "u"
  error.rebind (new FieldFE (grid(global_grid)(), "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(EXTERNAL_STORAGE));    // make linear system and solvers
  lineq->scan (menu);                         // determine storage and solver type
  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();

  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 Schur1:: scanGrids (MenuSystem& menu) // construct grids
{
  int i;
  int nsd = menu.get ("no of space dimensions").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);
```

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

sub_solve.redim  (no_of_grids-1);
system.redim     (no_of_grids-1);
sub_solve_prm.redim (no_of_grids-1);
proj.redim       (no_of_grids-1);
grid.redim       (no_of_grids);
dof.redim        (no_of_grids);
mat_prm.redim    (no_of_grids-1);
mat_inner.redim  (no_of_grids-1);
mat_trans1i.redim  (no_of_grids-1);
mat_transi1.redim  (no_of_grids-1);
rhs_inner.redim  (no_of_grids-1);
num_inner.redim  (no_of_grids-1);
num_trans.redim  (no_of_grids-1);
inner_dim.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<=local_grid) {
      int ix = k % part(j); // split into row, column ...
      k = k / part(j);
      x0 = (ix     * subdom(j)) / (real) dom(j);
      x1 = ((ix+1) * 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<=local_grid) 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());
    //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 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==global_grid))
        boInd_g += b1;
      else boInd_i += b1;

      String b0 = aform("%d ", j+nsd);
      if ((ix==1)||(i==global_grid))
        boInd_g += b0;
      else boInd_i += b0;
    }
    boInd_g += boInd_i;
    boInd_g += ")";
    grid(i)->redefineBoInds(boInd_g);
  }
  for (i = 1; i <= nsd; i++) // mark interface on grid(global_grids)
    for (int ix=1; ix < part(i); ix++) {
      String boInd = "n=1 b2=";
      for (int j = 1; j <= nsd; j++) {
        if (i==j) {
          real x = (ix * subdom(i)) / (real) dom(i);   // e.g. [.5,.5]
          boInd += "[" + aform("%g", x) + "," + aform("%g", x) + "]";
        } else
          boInd += "[0,1]";
        if (j<nsd) boInd += "x";
      }
     grid(global_grid)->addBoIndNodes(boInd);
    }

  FEM::scan (menu);  // load type and order of the numerical integration rule
  Store4Plotting::scan (menu, grid(global_grid)->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(global_grid)->getNoNodes()
      << ",\tno of elements: " << grid(global_grid)->getNoElms();
}
```

```
void Schur1:: fillEssBC (SpaceId space)
{
  int nno = grid(space)->getNoNodes(); // no of nodes
  dof(space)->initEssBC ();            // init for assignment below
  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
// inner boundaries are hom. Neumann
}

void Schur1:: 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 Schur1:: initProj() // setup proj operators
{
  for (int i=1; i<=local_grid; i++) {
    proj(i) = new ProjInterpSparse();
    proj(i)->rebindDOF(*dof(i), *dof(global_grid));
    proj(i)->init();
  }
}

void Schur1:: initInterface() //  set up num_inner, num_trans, num_interf
{
  int i, j;
  interface_dim = 0;
```

```
  int nno = grid(global_grid)->getNoNodes(); // no of nodes
  num_interf.redim(nno);
  num_interf = 0;
  for (j = 1; j <= nno; j++)
    if (grid(global_grid)->BoNode (j)) {  // is node j subj. to any boundary indicator?
      if (notBooLean(grid(global_grid)->BoNode (j, 1))) {
        interface_dim++;
        num_interf(j) = interface_dim;
      }
    }

  Vec(NUMT) interf_r(nno);
  for (j = 1; j <= nno; j++)
    interf_r(j) = num_interf(j);

  for(i=1; i<=local_grid; i++) {
    int nno = grid(i)->getNoNodes(); // no of nodes
    int inner = 0;
    VecSimple(int) ni(nno);
    for (j = 1; j <= nno; j++)
      if (grid(i)->BoNode (j)) {  // is node j subj. to any boundary indicator?
        if (grid(i)->BoNode (j, 1)) {
          inner++;
          ni(inner) = j;
        }
      } else {
        inner++;
        ni(inner) = j;
      }
    num_inner(i).redim(inner);
    for (j = 1; j <= inner; j++)
      num_inner(i)(j) = ni(j);
    inner_dim(i) = inner;

    Vec(NUMT) inner_r(nno);
    LinEqVector inner_vec(inner_r);
    proj(i)->apply(interf_r, inner_vec, TRANSPOSED, dpFALSE);

    num_trans(i).redim(interface_dim);
    num_trans(i) = 0;
    for (j = 1; j <= nno; j++)
      if (inner_r(j)>.5)
        num_trans(i)((int)inner_r(j)) = j;
  }
  s_o << "\n interface :\tNo of nodes: " << interface_dim<< "\n\n";
}

Handle(Matrix(NUMT)) Schur1:: initMatrix(int i)   // setup stiffness matrix on sub-domain i
{
  int j;
  fillEssBC (i);     // set essential boundary conditions
  Handle(Vec(NUMT)) rhs_l;
  rhs_l = 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_l());

  //split rhs_l() into rhs_inner(i), rest is added to rhs_interf
  Handle(Vec(NUMT)) v = new Vec(NUMT) (inner_dim(i));
  for (j=1; j<=inner_dim(i); j++)
    (*v) (j) = rhs_l() (num_inner(i)(j));
  rhs_inner(i) = new LinEqVector(*v);
  // update rhs_interf
  Vec(NUMT)& v_interf = CAST_REF(rhs_interf().vec(), Vec(NUMT));
  for (j=1; j<=interface_dim; j++)
    if (num_trans(i)(j)>0)
      v_interf (j) += rhs_l() (num_trans(i)(j));

  //split A() into mat_inner(i) and mat_trans1i(i), mat_transi1(i), rest is added to mat_interf
  mat_inner(i) = splitMatrix(NUMT) (A(), num_inner(i), num_inner(i),
    mat_prm(i)());
  mat_trans1i(i) = splitMatrix(NUMT) (A(), num_inner(i), num_trans(i));
  mat_transi1(i) = splitMatrix(NUMT) (A(), num_trans(i), num_inner(i));
  splitMatrix(NUMT) (A(), mat_interf(), num_trans(i), num_trans(i), dpTRUE);

  Handle(Vec(NUMT)) rhs_i = new Vec(NUMT) (inner_dim(i));
  Handle(Vec(NUMT)) u_i   = new Vec(NUMT) (inner_dim(i));
  system(i)->attach(mat_inner(i)(), u_i(), rhs_i());

  return A;
}

void Schur1:: initMatrices()   // setup stiffness matrices on sub-domains
{
  linsol.redim (interface_dim);            // init length of lin.sys. solution
  linrhs.redim (interface_dim);            // init length of lin.sys. rhs
  int i;
  Handle(Vec(NUMT)) v_interf = new Vec(NUMT) (interface_dim);
  v_interf->fill(0.);
  rhs_interf = new LinEqVector(*v_interf);
  Handle(Mat(NUMT)) m_interf = new Mat(NUMT) (interface_dim, interface_dim);
  m_interf->fill(0.);
  mat_interf = m_interf();

  for(i=1; i<=local_grid; i++)
    initMatrix(i);

  dof(global_grid)->initEssBC ();
  Handle(MatSchur(NUMT)) A = new MatSchur(NUMT) (interface_dim);
  A->attachUserCode(*this);
  dof(global_grid)->initAssemble();
  lineq->attach (A(), linsol, linrhs);      // use linsol as sol.vec. in lineq
}
```

```
void Schur1:: initRhs() // compute linrhs by local rhs
{
  linrhs = CAST_REF(rhs_interf().vec(), Vec(NUMT));
  for (int i=1; i<=local_grid; i++) {
    Vec(NUMT) &b = CAST_REF(system (i)->b().vec(), Vec(NUMT));
    system (i)->b() = rhs_inner(i)();
    b.mult(-1.);

    sub_solve(i)->solve ( system(i)() );
    system(i)->allow_factorization = dpFALSE;

    Vec(NUMT) &x = CAST_REF(system (i)->x().vec(), Vec(NUMT));
    mat_transi1(i)->prod (x, linrhs, NOT_TRANSPOSED, dpTRUE);
  }
}

void Schur1:: projSol() // compute u() by local solutions
{
  u->values().fill(0.0);
  for (int i=1; i<=local_grid; i++) {
    Vec(NUMT) &b = CAST_REF(system (i)->b().vec(), Vec(NUMT));
    mat_trans1i(i)->prod (linsol, b, NOT_TRANSPOSED, dpFALSE);
    Vec(NUMT) &r = CAST_REF(rhs_inner(i)->vec(), Vec(NUMT));
    b.add(r, '-', b);

    sub_solve(i)->solve ( system(i)() );
    system(i)->allow_factorization = dpFALSE;

    Vec(NUMT) &x = CAST_REF(system (i)->x().vec(), Vec(NUMT));
    Vec(NUMT) grid_sol( grid(i)->getNoNodes() );
    for (int j=1; j<=x.getNoEntries(); j++)
      grid_sol (num_inner(i)(j)) = x(j);

    LinEqVector uu(u->values());
    proj(i)->apply(grid_sol, uu, NOT_TRANSPOSED, dpTRUE);
  }
  Vec(NUMT) &x = CAST_REF(u->values(), Vec(NUMT));
  for(int j=1; j<=x.getNoEntries(); j++)
    if (num_interf(j)>0)
      x(j) = linsol(num_interf(j));
}

void Schur1:: solveProblem ()  // main routine of class Schur1
{
  initProj();                  // set up transfer global <> local
  initInterface();             // set up splitting Schur <> inner systems
  initMatrices();              // local matrices, split into inner & Schur part
  initRhs();                   // compute rhs for Schur system
  linsol.fill (0.0);           // set all entries to 0 in start vector Schur

  lineq->solve();              // solve Schur 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);

  projSol();                   // compute global solution from Schur solution
```

```
  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 Schur1:: 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(global_grid)->getNoNodes() < 100)
    error->values().print("FILE=error.dat","Nodal values of the error field");
}

real Schur1:: 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 Schur1:: k (const Ptv(real)& /*x*/)
{ return 1.; }

void Schur1:: prod
    (const Vector(NUMT)& xb,
           Vector(NUMT)& yb,
           TransposeMode tpmode,
           BooLean add_to_yb)
{
  mat_interf->prod(xb, yb, tpmode, add_to_yb);

  if (tpmode == NOT_TRANSPOSED) {
    for (int i=1; i<=local_grid; i++) {
      Vec(NUMT) &b = CAST_REF(system (i)->b().vec(), Vec(NUMT));
      mat_trans1i(i)->prod (xb, b, tpmode, dpFALSE);
      b.mult(-1.);

      sub_solve(i)->solve ( system(i)() );
      system(i)->allow_factorization = dpFALSE;

      Vec(NUMT) &x = CAST_REF(system (i)->x().vec(), Vec(NUMT));
```

```
      mat_transi1(i)->prod (x, yb, tpmode, dpTRUE);
    }
  } else errorFP("Schur1:: prod", "transposed not implemented");
}

String Schur1:: comment ()
{ return "Schur complement iteration"; }
```

main.C

```
#include <Schur1.h>
int main (int nargs, const char** args)
{
  initDIFFPACK (nargs, args);
  global_menu.init ("Flexible Poisson equation simulator","Schur1");
  Schur1 problem;                    // make a simulator object, called problem
  global_menu.multipleLoop (problem);// solve one or several problems
  DBP("leaving main");
  return 0;
}
```

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

We study the performance of the conjugated gradient method applied to the Schur complement matrix, see table 1, input file `prec1.i`. The first test is about the partition into stripes. Try to figure out the dependency of the number of iterations on the grid-size, that is the size of a sub-domain. Is there a difference between two and three sub-domains? Can you explain the effect?

You can compare the iteration counts with a conjugated gradient method applied to the global system `Elliptic1` in [Lan94]. Do you observe the better condition number of the Schur complement compared to the original global matrix? Another comparison is about the work required to solve both the Schur complement system and the global system. Of course this also depends on the work required to solve the local Dirichlet problems.

The next test is about a partition with cross-points, that are interior points where several interior edges meet, see table 2, input file `prec2.i`.

We repeat the test with a three dimensional example, see table 3, input file `prec3.i`.

The last experiment is about inexact Dirichlet solvers in the Schur complement, see table 4, input file `prec4.i`. Here the main criterion is the precision of the overall method. Only when the tolerance is met, the number of iterations is of interest. The problem here is that an initial error is usually not corrected by the iteration later, since the whole iteration is inexact.

[2] files are in `Schur1/Verify/`

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	{[2,1] & [3,1]}
element type	ElmB4n2D
matrix type	MatSparse
basic method	ConjGrad
preconditioning type	PrecNone
#1: convergence monitor name	CMAbsResidual
local basic method	GaussElim

Table 1: Partition into stripes, `prec1.i`

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	{[2,2] & [3,3]}
basic method	ConjGrad
preconditioning type	PrecNone
local basic method	GaussElim

Table 2: Partition with cross-points, `prec2.i`

3 Neumann-Neumann preconditioner for the Schur complement

The condition number of the Schur complement equation system is lower than the condition number of the original global problem. This means that the number of Krylov iterations required to solve the Schur complement system is lower than the number of iterations for the global system (if the termination criteria are comparable). If we want to reduce the number of iterations further, we can apply a preconditioner in the Krylov iteration. However, there is a problem in applying standard preconditioner like diagonal scaling (Jacobi) or incomplete factorization (ILU), since the

menu item	answer
no of space dimensions	3
subdomain	[4,4,4]
partition	[2,2,2]
element type	ElmB8n3D
basic method	ConjGrad
preconditioning type	PrecNone
local basic method	GaussElim

Table 3: Partition in three dimensions, `prec3.i`

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	[2,2]
basic method	ConjGrad
preconditioning type	PrecNone
local basic method	SSOR
local startvector mode	ZERO_START
local max iterations	{1 & 5 & 20}

Table 4: Inexact Schur complement, `prec4.i`

Schur complement matrix S is not explicitly known. We have to use other types of preconditioners for the interface problem instead.

3.1 Definition

One popular interface preconditioner is the Neumann-Neumann preconditioner [BGLV89, RL91]. It is a preconditioner for the Schur complement without using the matrix S and operates on the interface. We use the notation of the last chapter on the Schur complement iteration.

The basic idea is to solve one auxiliary problem of Neumann type on each sub-domain. The right hand side is a partitioned residual on the interface and the restrictions of the solutions are averaged on the interface. This is an expensive procedure, but the amount of work is comparable to the solution of a Dirichlet problem required for the multiplication with the Schur complement matrix S.

We define the sub-problems i like this:

$$\begin{aligned} \mathcal{L} u_i &= 0 && \text{on } \Omega_i \\ \frac{\partial}{\partial n} u_i &= r_i && \text{on } \Gamma_i \\ u_i &= g_1 && \text{on } \partial\Omega_i \cap \Gamma \\ \frac{\partial}{\partial n} u_i &= g_2 && \text{on } (\partial\Omega_i \cap \Gamma) \setminus \Gamma \end{aligned}$$

where r_i is a partition of the residual r given on the interface extended to the interior nodes by zero.

$$\sum_i r_i = r$$

Introducing a partition χ_i of unity on the interface

$$\begin{aligned} \chi_i(x) &\geq 0 \\ \text{supp}\chi_i &\subset \overline{\Gamma_i} \\ \sum_i \chi_i &\equiv 1 \quad \text{on } \mathcal{I} \end{aligned}$$

we define the action of the preconditioner as

$$\sum_i \chi_i u_i$$

restricted to the interface nodes for the right hand side r extended to zero and

$$r_i = \chi_i r$$

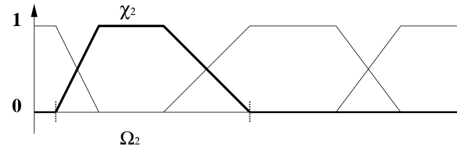


Figure 4: Partition of unity

The Neumann-Neumann preconditioner usually is implemented using the sub-assembly technique which we already applied for the assembly of the Schur complement system. The idea is to assemble a local matrix for each sub-domain with homogeneous Neumann boundary conditions at the interface nodes. Adding up the matrices leads to the global stiffness matrix. In this case we reuse the local matrices for preconditioning.

The local problems which participate with the Dirichlet boundary Γ are of course not pure Neumann problems but contain also some Dirichlet data. Interior domains without Dirichlet boundary lead to pure Neumann boundary conditions. The solution is determined only up to a constant which may cause some trouble. We will handle this ambiguity later. Very Small numbers of sub-domains or special partition patterns may give no such interior sub-domains.

The Neumann-Neumann method is quite general and can cope with all kinds of partitions of the domain. This is especially true for the presence of cross-points, which are points in the domain, where more than two sub-domains meet. In some methods to be discussed later, the cross-points play a prominent role. If there are no cross-points, we do not need the partition of unity χ_i and we can use the identity instead.

$$B = \sum_j Q_j^* S_j Q_j$$

We can write the algorithm as an additive Schwarz preconditioner applied to the Schur complement system. Several corrections are computed independently and summed up afterwards. Hence we will use the framework of Schwarz solvers for implementation.

3.2 Code

We extend the `Schur1` class and introduce a Neumann-Neumann preconditioner. It is implemented as an additive Schwarz iteration like in `Overlap1` documented in the tutorial [Zum96b]. We merge some elements of this code into the new one.[3]

The new simulator is derived from the Schur complement iteration `Schur1` discussed earlier and the interface for domain decomposition methods `DDSolverUDC`. The local

[3] this code is also in `Neumann1/`

Neumann problem solvers are `neu_solve` and the Neumann problems `neu_system`. The partition of unity χ_i is stored in `unity` and the additive Schwarz method will be constructed by the `ddsolver` object.

We have to implement the standard members of the `DDSolverUDC` interface. This is mainly the solution procedure `solveSubSystem` and the `transfer` between the interface and the local problems.

Neumann1.h

```
// prevent multiple inclusion of Neumann1.h
#ifndef Neumann1_h_IS_INCLUDED
#define Neumann1_h_IS_INCLUDED

#include <Schur1.h>
#include <DDSolver.h>           // DDSolver
#include <DDSolverUDC.h>        // interfacing to DDSolver
#include <DDSolver_prm.h>       // DDSolver parameters

class Neumann1 : public Schur1, public DDSolverUDC
{
protected:
  // preconditioner related data:
  VecSimplest(Handle(LinEqSolver))      neu_solve;    // linear solution
  VecSimplest(Handle(prm(LinEqSolver))) neu_solve_prm;// linear solution parameter
  VecSimplest(Handle(LinEqSystemStd))   neu_system;   // linear system, storage
  prm(DDSolver) ddsolver_prm;                         // parameters domain decomposition solver
  Vec(real)                             unity;        // partition of unity on interface
  Handle(DDSolver)                      ddsolver;     // domain decomposition preconditioner

  virtual void scanGrids(MenuSystem& menu);// construct grids
  virtual Handle(Matrix(NUMT)) initMatrix(int i);// setup stiffness matrix i
  virtual void initMatrices();         // setup stiffness matrices on sub domains grids

public:
  Neumann1 ();
 ~Neumann1 () {}

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

  // 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);

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

An additional menu entry contains the parameters for the Neumann problem solvers `Neumann LinEqAdm`. The stiffness matrices for the Neumann problems are produced by the sub-assembly procedure implemented in `Schur1`.

The discrete partition of unity is computed as one over the number of contributing domains at an interface node.

The transfer of data between the interface and the local problems is implemented directly. Using a translation table `num_trans` for the different node numbering it consists of copying node data from interface nodes to nodes on the interior boundary of the sub-domain or vice versa. The data has to be extended by zeros where necessary.

Neumann1.C

```
#include <Neumann1.h>
#include <PreproBox.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <createElmDef.h>        // for calling hierElmDef in Neumann1::define
#include <createMatrix_real.h>   // creating stiffness matrices
#include <createLinEqSolver.h>   // creating sub domain solver
#include <splitMatrix_real.h>    // splitting stiffness matrices
#include <PrecDD.h>
#include <createDDSolver.h>      // creating multigrid object

Neumann1:: Neumann1 () {}

void Neumann1:: define (MenuSystem& menu, int level)
{
  Schur1:: define (menu, level);
  prm(DDSolver)::   defineStatic (menu, level+1);// DD parameters
  menu.setCommandPrefix("Neumann");
  prm(LinEqSolver)::defineStatic (menu, level+1);// sub domain solver parameters
  menu.unsetCommandPrefix();
}

void Neumann1:: scan (MenuSystem& menu)
{
  Schur1:: scan (menu);

  // prm for DD preconditioner
  Handle(prm(Precond)) precondPrm = new prm(Precond);
  precondPrm->scan(menu);
  lineq->attach (precondPrm());

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

  menu.setCommandPrefix("Neumann");
  for (int i=1; i<=local_grid; i++) {
    neu_solve_prm(i).rebind(new prm(LinEqSolver));
```

```
    neu_solve_prm(i)->scan (menu);
    neu_solve(i).rebind(createLinEqSolver (neu_solve_prm(i)()));
    neu_system(i).rebind(new LinEqSystemStd (EXTERNAL_STORAGE));
  }
  menu.unsetCommandPrefix();
}

void Neumann1:: scanGrids (MenuSystem& menu) // construct grids
{
  Schur1:: scanGrids(menu);
  neu_solve.redim     (local_grid);
  neu_system.redim    (local_grid);
  neu_solve_prm.redim (local_grid);
}

Handle(Matrix(NUMT)) Neumann1:: initMatrix(int i)
{   // setup stiffness matrix on sub-domain i
  Handle(Matrix(NUMT)) A = Schur1:: initMatrix(i);
  neu_system(i)->attach(A());

  Handle(Vec(NUMT)) rhs_n = new Vec(NUMT) (dof(i)->getTotalNoEqs ());
  Handle(Vec(NUMT)) u_n   = new Vec(NUMT) (dof(i)->getTotalNoEqs ());
  ddsolver->attachLinRhs(rhs_n(), i, dpFALSE);
  ddsolver->attachLinSol(u_n(), i);
  return A;
}

void Neumann1:: initMatrices()   // setup stiffness matrices on sub-domains
{
  Schur1:: initMatrices();
  unity.redim (interface_dim);            // init length of partition of unity
  for (int j=1; j<=interface_dim; j++) {
    int u = 0;
    for (int i=1; i<=local_grid; i++)
      if (num_trans(i)(j)>0)
u++;
    unity (j) = 1.0 / u;
  }
}

extern real analyticalSolution (const Ptv(real)& x, real t);

void Neumann1:: solveProblem () // main routine of class Neumann1
{
  initProj();                  // set up transfer global <> local
  initInterface();             // set up splitting Schur <> inner systems
  initMatrices();              // local matrices, split into inner & Schur part
  initRhs();                   // compute rhs for Schur system
  linsol.fill (0.0);           // set all entries to 0 in start vector Schur

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

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

```
  lineq->solve();              // solve Schur 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);

  projSol();                   // compute global solution from Schur solution
  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.
}

SpaceId Neumann1:: getNoOfSpaces() const
{ return no_of_grids; }

BooLean Neumann1:: solveSubSystem (
  LinEqVector& b, LinEqVector& x,
  SpaceId space, StartVectorMode /*start*/, DDSolverMode /*mode*/)
{
  neu_solve_prm (space)->startmode = ZERO_START;
  neu_system (space)->attach (x, b);
  neu_solve (space)->solve  ( neu_system (space)() );
  neu_system(space)->allow_factorization = dpFALSE;

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

  return dpTRUE; // solution has changed
}

BooLean Neumann1:: transfer (
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode)
{
  const Vec(NUMT) &fvv = CAST_REF(fv.vec(), Vec(NUMT));
  Vec(NUMT) &tvv = CAST_REF(tv.vec(), Vec(NUMT));
  if (ti == no_of_grids) {
    if (add_to_t) {
      for (int j=1; j<=interface_dim; j++)
if (num_trans(fi)(j)>0)
  tvv (j) += unity(j) * fvv (num_trans(fi)(j));
    } else { // not used
      tvv = 0.;
      for (int j=1; j<=interface_dim; j++)
if (num_trans(fi)(j)>0)
  tvv (j) = unity(j) * fvv (num_trans(fi)(j));
    }
  }
  else if (fi == no_of_grids) {
    if (add_to_t) { // not used
      for (int j=1; j<=interface_dim; j++)
if (num_trans(ti)(j)>0)
```

```
  tvv (num_trans(ti)(j)) += unity(j) * fvv (j);
    } else {
      tvv = 0.;
      for (int j=1; j<=interface_dim; j++)
if (num_trans(ti)(j)>0)
  tvv (num_trans(ti)(j)) = unity(j) * fvv (j);
    }
  }
  else fatalerrorFP("Neumann1:: transfer","undefined");
  return dpTRUE;
}

int Neumann1:: getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork) const
{
  if (ti == no_of_grids)
    return interface_dim;
  else if (fi == no_of_grids)
    return interface_dim;
  return 0;
}

real Neumann1:: getStorageTransfer (SpaceId /*fi*/, SpaceId ti) const
{
  if (ti == no_of_grids)
    return interface_dim;
  return 0;
}

int Neumann1:: getWorkSolve (SpaceId space, const PrecondWork) const
{ return neu_solve (space)->getWork(); }

real Neumann1:: getStorageSolve (SpaceId space) const
{ return neu_solve (space)->getStorage(); }

String Neumann1:: comment ()
{ return "Neumann-Neumann preconditioner"; }
```

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

We perform a couple of experiments with the Neumann-Neumann preconditioner. We re-use some tests form the Schur complement example. We can study the effect of preconditioning by a comparison of the number of iterations/ the convergence rate of the conjugated gradient method with and without a preconditioner. So the first test is about a domain decomposed into stripes, see table 5, input file `prec1.i`.

The second test is about a domain partitioned with cross-points, see table 6, input file `prec2.i`. Is there a difference to the partition into stripes? Try to explain it.

This test is about a three dimensional problem, see table 7, input file `prec3.i`. Does the convergence rate degrade from 2D to 3D (as it does for some iterative methods)?

The next test is about inexact Neumann solvers, see table 8, input file `prec4.i`. In each iteration we have to solve a Neumann problem in each sub-domain. If we

[4] files are in `Neumann1/Verify/`

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	{[2,1] & [3,1]}
element type	ElmB4n2D
matrix type	MatSparse
basic method	ConjGrad
preconditioning type	PrecDD
left preconditioning	ON
#1: convergence monitor name	CMAbsResidual
local basic method	GaussElim
Neumann basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 5: Partition into stripes, `prec1.i`

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	{[2,2] & [3,3]}
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Neumann basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 6: Partition with cross-points, `prec2.i`

use an approximate solver here, the quality of the preconditioner decreases. So the convergence rate will be worse. However, the iteration will converge to the exact solution. Can you quantify this effect?

The last test is about inexact Dirichlet solvers in the forming of the Schur complement and additionally inexact Neumann solvers in the preconditioner, see table 9, input file `prec5.i`. Here we have to be careful, whether the solution is correct, since errors in the Dirichlet solve are not corrected, while errors in the Neumann solver are. If we have verified, that a solution is ok, we can compare the iteration counts to find the most efficient solution parameters.

3.3 Stabilizing pure Neumann problems

One drawback of the present implementation, as we have seen in the experiments, are the pure Neumann problems. As we have mentioned already, sub-domains without any Dirichlet boundary conditions (without Γ) define solutions only up to a constant. Although it might be enough to just use one possible solution and to ignore warnings by some solution procedure (or modify the Gaussian elimination procedure [RL91]),

menu item	answer
no of space dimensions	3
subdomain	[4,4,4]
partition	[2,2,2]
element type	ElmB8n3D
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Neumann basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 7: Partition in three dimensions, `prec3.i`

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	[2,2]
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Neumann basic method	SSOR
Neumann startvector mode	ZERO_START
Neumann max iterations	{1 & 2 & 4 & 8}
domain decomposition method	AddSchwarzDD

Table 8: Inexact Neumann solvers, `prec4.i`

we want to determine such a solution uniquely.

One suggestion to construct local Neumann problems with a unique solution is to modify the differential operator slightly [DW91]. Adding some lower order mass term (sometimes called Helmholtz term, be careful about the sign!) does not destroy the overall convergence properties, but provides uniqueness of the sub-problems.

For example for the local Laplace problem defined as

$$\begin{aligned} -\Delta u_i &= 0 \quad \text{on } \Omega_i \\ \frac{\partial}{\partial n} u_i &= r_i \quad \text{on } \Gamma_i \end{aligned}$$

we can use the sub-problem

$$\begin{aligned} (-\Delta + \epsilon I) u_i &= 0 \quad \text{on } \Omega_i \\ \frac{\partial}{\partial n} u_i &= r_i \quad \text{on } \Gamma_i \end{aligned}$$

instead for preconditioning, with a positive regularization parameter $\epsilon > 0$. This leads also to a positive symmetric stiffness matrix, which now is strictly positive and does not contain an eigen-value zero any longer.

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	[2,2]
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	SSOR
local startvector mode	ZERO_START
local max iterations	{1 & 5 & 20}
Neumann basic method	SSOR
Neumann startvector mode	ZERO_START
Neumann max iterations	{1 & 2 & 4}
domain decomposition method	AddSchwarzDD

Table 9: Inexact Schur complement and inexact Neumann solvers, `prec5.i`

We implement this method as an extension of the original class `Neumann1`. We modify the construction of the Neumann matrix assembly.[5]

A new class `Neumann2mass` contains a modified weak form `integrands` of the differential operator including the mass term. The regularization parameter ϵ is passed via `setValue`. The simulator class `Neumann2` contains one instance of the `Neumann2mass` class and implements a new `initMatrix` method. Besides reading the regularization parameter from the menu, there are no further changes compared to the original Neumann-Neumann preconditioner `Neumann1`.

Neumann2.h

```
// prevent multiple inclusion of Neumann2.h
#ifndef Neumann2_h_IS_INCLUDED
#define Neumann2_h_IS_INCLUDED

#include <Neumann1.h>

class Neumann2mass : public FEM
{
protected:
  real m_value;
  virtual real k(const Ptv(real)& x);    // coefficient in the PDE
  virtual void integrands                // evaluate weak form in the FEM equations
    (ElmMatVec& elmat, FiniteElement& fe);
public:
  Neumann2mass ();
 ~Neumann2mass () {}
  void setValue (real m);                // mass matrix coefficient
};

class Neumann2 : public Neumann1
{
protected:
```

[5]this code is also in `Neumann2/`

```
  virtual Handle(Matrix(NUMT)) initMatrix(int i);// setup stiffness matrix i
  Neumann2mass mass_system;    // system for Neumann problems only
public:
  Neumann2 ();
 ~Neumann2 () {}

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

  String  comment ();
};
#endif
```

The new class `Neumann2mass` has a modified version of the method `integrands`. It makes use of the regularization parameter `m_value`, which is read from the menu and set in `scan`. The `initMatrix` function does not use the original local matrix for initialization of the local Neumann systems `neu_system`, but it assembles a different system. The `makeSystem` function of class `Neumann2mass` assembles the modified system.

Neumann2.C

```
#include <Neumann2.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <createElmDef.h>        // for calling hierElmDef in Neumann2::define
#include <createMatrix_real.h>   // creating stiffness matrices

Neumann2mass:: Neumann2mass () :
  m_value(0.)
{}

void Neumann2mass:: setValue (real m)
{
  m_value = max(0., m);
}

real Neumann2mass:: k (const Ptv(real)& /*x*/)
{ return 1.; }

void Neumann2mass:: 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 (nsd);
  fe.getGlobalEvalPt (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 +
m_value * fe.N(i) * fe.N(j) ) * detJxW;
    }
  }
}

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

Neumann2:: Neumann2 () {}

void Neumann2:: define (MenuSystem& menu, int level)
{
  Neumann1:: define (menu, level);
  menu.addItem (level,
               "mass term",    // menu command/name
               "mass",         // command line option: +nsd
               "positive real",
               "1.0",          // default answer (2D problem)
               "R1");          // valid answer: 1 real
}

void Neumann2:: scan (MenuSystem& menu)
{
  Neumann1:: scan (menu);
  mass_system.setValue(menu.get ("mass term").getReal());
}

Handle(Matrix(NUMT)) Neumann2:: initMatrix(int i)   // setup stiffness matrix on sub-domain i
{
  Neumann1:: initMatrix(i);
  Handle(Matrix(NUMT)) M;
  M = createMatrix(NUMT) (mat_prm(i)());

  //dof(i)->initAssemble();
  mass_system.makeSystem (dof(i)(), M());
  neu_system(i)->attach(M());
  return M;
}

String Neumann2:: comment ()
{ return "Neumann-Neumann preconditioner with mass matrix term"; }
```

look at example 3×3 sub-domains

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

We redo the experiments for the modified version of the Neumann-Neumann preconditioner. We have to use partitions with at least one interior domain, to observe an effect. We start with a two dimensional decomposition, see table 10, input file

[6] files are in **Neumann2/Verify/**

`prec1.i`. This test is a parameter study for the regularization parameter. The value zero is equivalent to the original method. Compare the number of iterations. Which one is the optimal parameter?

menu item	**answer**
no of space dimensions	2
subdomain	[10,10]
partition	[3,3]
element type	ElmB4n2D
mass term	{0 & 1e-7 & 1e-3 & 1 & 1e3}
matrix type	MatSparse
basic method	ConjGrad
preconditioning type	PrecDD
left preconditioning	ON
#1: convergence monitor name	CMAbsResidual
local basic method	GaussElim
Neumann basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 10: Varying the mass term, `prec1.i`

This test is about the dependence of the convergence rate on the partition size, see table 11, input file `prec2.i`. What kind of dependence do you observe?

menu item	**answer**
no of space dimensions	2
subdomain	[10,10]
partition	{[3,3] & [4,4]}
mass term	1.0
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Neumann basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 11: Varying the partition, `prec2.i`

We redo the experiment for the three dimensional problem, see table 12, input file `prec3.i`.

This test is about inexact Neumann solvers, see table 13, input file `prec4.i`. Compare the results to the original Neumann-Neumann preconditioner.

The last test is about inexact Dirichlet and inexact Neumann solvers, see table 14, input file `prec5.i`. Compare the convergence rates and be check for the precision of the final result.

menu item	answer
no of space dimensions	3
subdomain	[4,4,4]
partition	[3,3,3]
element type	ElmB8n3D
mass term	1.0
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Neumann basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 12: Partition in three dimensions, `prec3.i`

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	[3,3]
mass term	1.0
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Neumann basic method	SSOR
Neumann startvector mode	ZERO_START
Neumann max iterations	{1 & 2 & 4 & 8}
domain decomposition method	AddSchwarzDD

Table 13: Inexact Neumann solvers, `prec4.i`

3.4 Coarse grid acceleration

The Neumann-Neumann preconditioner shows a good performance for increasing number of unknowns at a fixed number of sub-domains. There is some weak dependence on the grid size parameter h for the limit $h \to \infty$. However if we fix the number of unknowns per sub-domain and increase the number of sub-domains, we do not obtain such a good convergence behavior. This is due to the lack of global communication in the Neumann-Neumann preconditioner. There is a practical limit in the number of sub-domains, if the preconditioner has to be efficient.

The remedy is to introduce a coarse grid like it is commonly used for the overlapping Schwarz iteration. A coarse grid is also the base for multi-level methods, which employ a hierarchy of nested grids. Since we are preconditioning the interface problem rather than the whole domain, we have to use slightly different coarse grid mechanisms. We present a coarse grid construction proposed by Mandel [Man93] called balancing domain decomposition.

The idea of the coarse grid operator stems form the stabilization of the local pure

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	[3,3]
mass term	1.0
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	SSOR
local startvector mode	ZERO_START
local max iterations	{1 & 5 & 20}
Neumann basic method	SSOR
Neumann startvector mode	ZERO_START
Neumann max iterations	{1 & 2 & 4}
domain decomposition method	AddSchwarzDD

Table 14: Inexact Schur complement and inexact Neumann solvers, `prec5.i`

Neumann problems. The solution of a pure Neumann problem is determined only up to a constant.

$$u_i \in \bar{u} + \mathrm{Z}_i$$

with a null-space Z_i of the local Neumann differential operator containing all constant functions on the domain. However the resulting preconditioner has to be unique. Hence we project the solution u_i onto the orthogonal complement of the null space of the local Neumann problems.

$$\langle u_i, v \rangle = 0 \text{ for all } v \in \mathrm{Z}_i$$

This is equivalent with

$$\langle \chi_i u, \mathrm{Z}_i \rangle = 0$$

which leads to the equation system

$$\left(\begin{array}{c} \chi_1 z_1 \chi_2 z_2 \end{array} \right)^* S \left(\begin{array}{c} \chi_1 z_1 \chi_2 z_2 \end{array} \right) u = 0$$

with

$$\mathrm{Z}_i = \langle z_i \rangle$$

This system may serve as a coarse grid equation. For a one dimensional space Z_k of constant functions, the coarse grid equation may be interpreted as a discretization with piecewise constant function in each sub-domain. This means one degree of freedom per sub-domain in the coarse grid equation. The sub-domains which contribute to the Dirichlet boundary Γ are not included in this coarse grid.

In order to use the coarse grid solver as a projection, we have to combine it in a multiplicative way with the standard additive Neumann-Neumann preconditioner. For symmetric preconditioning purposes we suggest the order

$$B \;=\; B_{\text{coarse}}(\sum_k Q_k^* S_k^{-1} Q_k) B_{\text{coarse}}$$

We implement the Neumann-Neumann preconditioner with balancing coarse grid on top of the simulator class empNeumann1.[7]

We introduce coarse grid related data such as the projection from the interface to the coarse space `proj_coarse`, the index of the coarse grid in the vector of grids `coarse_grid` and the dimension of the coarse grid system. We have to extend the `transfer` function to include the data transfer from an to the coarse grid using `proj_coarse`. In order to implement the multiplicative combination of coarse grid and standard preconditioner we also provide the method `residual` for the `DDSolverUDC` interface.

Neumann3.h

```
// prevent multiple inclusion of Neumann3.h
#ifndef Neumann3_h_IS_INCLUDED
#define Neumann3_h_IS_INCLUDED

#include <Neumann1.h>

class Neumann3 : public Neumann1
{
protected:
  Mat(NUMT) proj_coarse;
  int coarse_grid;
  int coarse_dim;
  VecSimple(int) coarse_num; // projection operators

  virtual void scanGrids(MenuSystem& menu);// construct grids
  virtual void initMatrices();             // setup stiffness matrices on sub domains grids
public:
  Neumann3 ();
 ~Neumann3 () {}

  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu);
  // DDSolverUDC
  BooLean transfer (const LinEqVector& fv, SpaceId fi,
          LinEqVector& tv, SpaceId ti,
    BooLean add_to_t= dpFALSE, DDTransferMode=TRANSFER);
  void residual (LinEqVector& b, LinEqVector& x, LinEqVector& r, SpaceId space);

  String  comment ();
};
#endif
```

We read a new set of parameters for the coarse grid solver from the menu `coarse LinEqAdm`

Neumann3.C

```
#include <Neumann3.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
```

[7] this code is also in `Neumann3/`

```
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <createLinEqSolver.h>   // creating sub domain solver
#include <createElmDef.h>        // for calling hierElmDef in Neumann3::define
#include <createMatrix_real.h>   // creating stiffness matrices

Neumann3:: Neumann3 () {}

void Neumann3:: define (MenuSystem& menu, int level)
{
  Neumann1:: define (menu, level);
  menu.setCommandPrefix("coarse");
  prm(LinEqSolver)::defineStatic (menu, level+1);// coarse system solver parameters
  menu.unsetCommandPrefix();
}

void Neumann3:: scan (MenuSystem& menu)
{
  Neumann1:: scan (menu);

  menu.setCommandPrefix("coarse");
  neu_solve_prm(coarse_grid).rebind(new prm(LinEqSolver));
  neu_solve_prm(coarse_grid)->scan (menu);
  neu_solve(coarse_grid).rebind(createLinEqSolver (neu_solve_prm(coarse_grid)()));
  neu_system(coarse_grid).rebind(new LinEqSystemStd (EXTERNAL_STORAGE));
  menu.unsetCommandPrefix();
}

void Neumann3:: scanGrids (MenuSystem& menu) // construct grids
{
  Schur1:: scanGrids(menu);
  no_of_grids++;
  global_grid++;
  coarse_grid = no_of_grids-1;

  sub_solve.redim  (no_of_grids-1);
  system.redim     (no_of_grids-1);
  sub_solve_prm.redim (no_of_grids-1);
  dof.redim        (no_of_grids);
  mat_prm.redim    (no_of_grids-1);

  int i;
  VecSimplest(Handle(GridFE))  g (no_of_grids-1);
  for (i=1; i<no_of_grids; i++)
    g(i).rebind(grid(i)());
  grid.redim (no_of_grids);
  for (i=1; i<=local_grid; i++)
    grid(i).rebind(g(i)());
  grid(coarse_grid).rebind(g(no_of_grids-1)()); // dummy
  grid(no_of_grids).rebind(g(no_of_grids-1)());

  neu_solve.redim     (no_of_grids-1);
  neu_system.redim    (no_of_grids-1);
  neu_solve_prm.redim (no_of_grids-1);

  // determine sub domains with Dirichlet boundary
  coarse_num.redim(local_grid);
  coarse_dim = 0;
  int nsd = menu.get ("no of space dimensions").getInt();
```

```
  Ptv(int) part(nsd);
  Is dIs(menu.get ("partition"));
  dIs->ignore ('[');
  for (i = 1; i <= nsd; i++) {
    dIs->get (part(i));
    if (part(i)<3) warningFP("Neumann3:: scanGrids", "no pure Neumann problems");
    if (i < nsd)
      dIs->ignore (',');
  }
  for (i=1; i<=local_grid; i++) {
    BooLean boundary = dpFALSE;
    int j;
    int k = i-1;
    for (j = 1; j <= nsd; j++) {
      int ix = k % part(j) + 1; // split into row, column ...
      k = k / part(j);
      if ((ix==1)||(ix==part(j))) {
boundary = dpTRUE;
break;
      }
    }
    if (boundary) coarse_num(i) = 0;
    else {
      coarse_num(i) = 1;
      coarse_dim += coarse_num(i);
    }
  }
}

void Neumann3:: initMatrices()   // setup stiffness matrices on sub-domains
{
  Neumann1:: initMatrices();

  Handle(Mat(NUMT)) A = new Mat(NUMT) (coarse_dim, coarse_dim);
  neu_system(coarse_grid)->attach(A());

  Handle(Vec(NUMT)) rhs_n = new Vec(NUMT) (coarse_dim);
  Handle(Vec(NUMT)) u_n   = new Vec(NUMT) (coarse_dim);
  ddsolver->attachLinRhs(rhs_n(), coarse_grid, dpFALSE);
  ddsolver->attachLinSol(u_n(), coarse_grid);

  proj_coarse.redim(coarse_dim, interface_dim);

  VecSimplest(Handle(LinEqVector)) SpI(coarse_dim);
  VecSimplest(Handle(LinEqVector)) pI(coarse_dim);
  int i, j, k;
  i=1;
  for (k=1; k<= local_grid; k++)
    if (coarse_num(k)>0) {
      Vec(NUMT) *p = new Vec(NUMT)(interface_dim);
      pI(i) = *new LinEqVector(*p);
      for (j=1; j<=interface_dim; j++) {
if (num_trans(k)(j)>0)
  (*p)(j) = unity(j);
else
  (*p)(j) = 0.0;
proj_coarse(i, j) = (*p)(j);
      }
      Vec(NUMT) *s = new Vec(NUMT)(interface_dim);
```

```
      SpI(i) = *new LinEqVector(*s);
      prod(*p, SpI(i)->vec(), NOT_TRANSPOSED);    // Schur complement matrix
      i++;
    }
  for (i=1; i<= coarse_dim; i++)
    for (j=1; j<= coarse_dim; j++)
      A()(i, j) = SpI(i)->inner (pI(j)());
}

void Neumann3:: residual (
    LinEqVector& b, LinEqVector& x, LinEqVector& r, SpaceId space)
{
  if (space!=no_of_grids) fatalerrorFP("Neumann3:: residual","not implemented");
  lineq->getLinEqSystem ().attach   (x, b);
  lineq->getLinEqSystem ().residual (r);
  lineq->getLinEqSystem ().attach   (linsol, linrhs); // restore
}

BooLean Neumann3:: transfer (
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode m)
{
  if ((fi != coarse_grid)&&(ti != coarse_grid))
    return Neumann1:: transfer(fv, fi, tv, ti, add_to_t, m);

  const Vec(NUMT) &fvv = CAST_REF(fv.vec(), Vec(NUMT));
  Vec(NUMT) &tvv = CAST_REF(tv.vec(), Vec(NUMT));
  if (ti == no_of_grids)
    proj_coarse.prod(fvv, tvv, TRANSPOSED, add_to_t);
  else if (fi == no_of_grids)
    proj_coarse.prod(fvv, tvv, NOT_TRANSPOSED, add_to_t);
  else fatalerrorFP("Neumann3:: transfer","undefined");
  return dpTRUE;
}

String Neumann3:: comment ()
{ return "Neumann-Neumann preconditioner with balancing coarse grid"; }
```

The implementation of an alternative coarse grid operator proposed by Dryja and Widlund [DW91] is left to the reader.

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

We redo part of the experiments for the balancing domain decomposition method or Neumann-Neumann preconditioner with coarse grid. The first test is about different partitions in two dimensions, see table 15, input file `prec2.i`. Compare the number of iterations/ the convergence rate of the original Neumann-Neumann preconditioner, the stabilized version and this version with coarse grid. You can also try to compare the amount of work needed for all three methods.

This test is about the three dimensional problem, see table 16, input file `prec3.i`.

This test is about the balancing domain decomposition method with both inexact Neumann solvers and inexact coarse grid solver, see table 17, input file `prec4.i`.

[8] files are in `Neumann3/Verify/`

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	{[3,3] & [4,4]}
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Neumann basic method	GaussElim
coarse basic method	GaussElim
domain decomposition method	CoarseAddCoarseSchwarzDD

Table 15: Varying the partition, `prec2.i`

menu item	answer
no of space dimensions	3
subdomain	[4,4,4]
partition	[3,3,3]
element type	ElmB8n3D
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Neumann basic method	GaussElim
coarse basic method	GaussElim
domain decomposition method	CoarseAddCoarseSchwarzDD

Table 16: Partition in three dimensions, `prec3.i`

Both inexact solvers deteriorate the quality of the preconditioner and reduce the work per iteration. Try to find parameters for an efficient overall iteration.

The last test is about additional inexact Dirichlet solvers, see table 18, input file `prec5.i`. Remember to check the precision of the solution before comparing convergence rates.

4 Eigen-decomposition preconditioner for the Schur complement

The Neumann-Neumann preconditioner is quite expensive. It requires the solution of one local Neumann problem per sub-domain and iteration. We will now have a look at a very cheap preconditioner, which is more restricted in the application range. We could have included the eigen decomposition preconditioner proposed by [Dry81] for historical reasons only. However, we will use the preconditioner as a building block for larger preconditioners of BPS and wire-basket type. We will present the preconditioner. An implementation follows in the next chapter, where it can be accessed as a special case of domain partitioning.

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	[3,3]
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Neumann basic method	SSOR
Neumann startvector mode	ZERO_START
Neumann max iterations	{1 & 2 & 4}
coarse basic method	SSOR
coarse startvector mode	ZERO_START
coarse max iterations	{1 & 2 & 4}
domain decomposition method	CoarseAddCoarseSchwarzDD

Table 17: Inexact Neumann solvers, `prec4.i`

Since the entries of the Schur complement matrix are not available directly in the preconditioner, either the differential operator itself or the discretization of auxiliary problems on sub-domains provide the necessary data for a preconditioner. We construct a preconditioner, which does not use any auxiliary sub-problems.

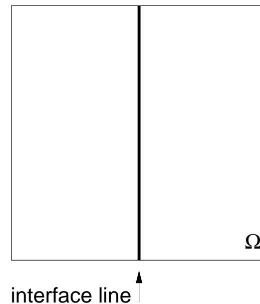


Figure 5: Domain decomposition by one line.

If a two dimensional domain is cut into stripes the interface consists of lines. In the simplest case the interface problem is a one dimensional problem on a line (figure 5). It is quite cheap to solve one dimensional problems arising in the discretization of differential operators, if the operator is known. The continuous equivalent to the Schur complement is the interface operator. For the Laplace equation on the global domain it is in a sense the square root of the Laplace operator. Such a system can be solved efficiently by FFT (the discrete sine transform). The idea now is to use such a solver as a preconditioner for the Schur problem.

menu item	answer
no of space dimensions	2
subdomain	[10,10]
partition	[3,3]
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	SSOR
local startvector mode	ZERO_START
local max iterations	{1 & 5 & 20}
Neumann basic method	SSOR
Neumann startvector mode	ZERO_START
Neumann max iterations	{1 & 4}
coarse basic method	SSOR
coarse startvector mode	ZERO_START
coarse max iterations	{1 & 4}
domain decomposition method	CoarseAddCoarseSchwarzDD

Table 18: Inexact Schur complement and inexact Neumann solvers, `prec5.i`

For a with n equidistant unknowns this preconditioner for looks like

$$B = \mathcal{F}^* \begin{pmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \sigma_3 & \\ & & & \ddots \end{pmatrix} \mathcal{F}$$

with the discrete sine transform $\mathcal{F} = \mathcal{F}^*$ and

$$\sigma_j = \sqrt{4 \sin^2 \left(\frac{j\pi}{2(n+1)} \right)}$$

in the version of Dryja [Dry81] and

$$\sigma_j = \sqrt{4 \sin^2 \left(\frac{j\pi}{2(n+1)} \right) + 4 \sin^4 \left(\frac{j\pi}{2(n+1)} \right)}$$

in the version of Golub and Meyers [GM84]. The generalization to interfaces for three dimensional problems, that is a preconditioner for rectangular shaped faces with $n \times m$ equidistant spaced unknowns, is as follows (figure 6 left):

$$B = \mathcal{F}^* \begin{pmatrix} \sigma_{1,1} & & & & & & \\ & \sigma_{1,2} & & & & & \\ & & \ddots & & & & \\ & & & \sigma_{1,m} & & & \\ & & & & \sigma_{2,1} & & \\ & & & & & \sigma_{2,2} & \\ & & & & & & \ddots \end{pmatrix} \mathcal{F}$$

with

$$\sigma_{j,k} = \sqrt{4\sin^2\left(\frac{j\pi}{2(n+1)}\right) + 4\sin^2\left(\frac{k\pi}{2(m+1)}\right)}$$

In the case the unknowns are not equidistant spaced, one can project them onto such a grid a perform the preconditioning step.

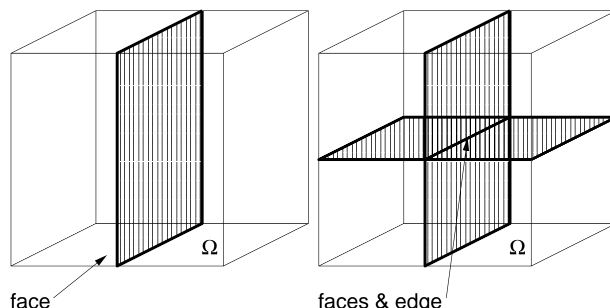


Figure 6: Domain decomposition by one face or by four faces and one edge.

The limitation of the method clearly is the restriction to geometrically simple interfaces like a line in 2D or a square in 3D. Of course it is possible to use several preconditioners at a time to precondition the interface of a domain cut into several stripes (figure 7). However, it is not possible to deal with cross-points, points where several lines of the interface meet. We will construct more complicated preconditioners which handle this case in the next section.

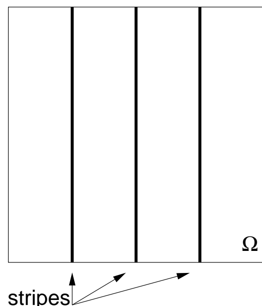


Figure 7: Domain decomposition by several stripes.

We present an implementation of the eigen-decomposition preconditioner taken from the preconditioner to be constructed in the next section. The preconditioner does not require a grid or a discretization. The number of unknowns is enough and the assertion, that the data represents an equidistant discretization is enough.

from Wire1.C

You may want to have a look at the numerical experiments with eigen-decomposition preconditioners in the next chapter: Two dimensional tests with the edge precondi-

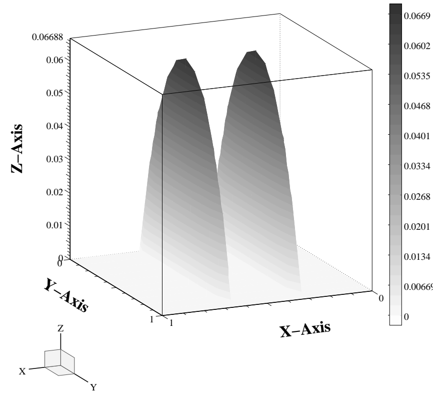


Figure 8: Solution on the interface consisting of two stripes.

tioner in table 19, three dimensional tests with face preconditioners in table 21 and the combination of edge and face preconditioner in table 22.

5 BPS and Wire-basket preconditioner for the Schur complement

5.1 Definition

There are lot of methods proposed in a series of papers starting with [BPS86] all referred to as *BPS* which might be a little confusing. The idea is to use sub-domain problems for preconditioning the Schur system.

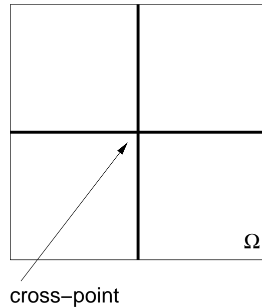


Figure 9: Domain decomposition with a cross-point.

The preconditioners based on the eigen-decomposition in the last chapter were not able to handle the case of cross-points (see figure 9 and 10). The straightforward approach would be, to partition and resort the unknowns on the interface into nodes related to an edge E_j and to a Vertex V_j. We then construct a block-preconditioner, which is one of the previous preconditioners for each edge and a diagonal scaling

(Jacobi) for each vertex. Unfortunately this approach does not create a good preconditioner.

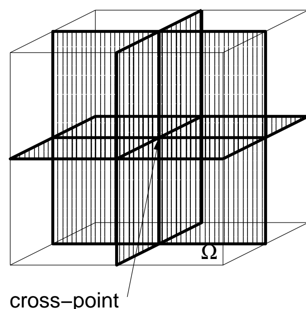


Figure 10: Three dimensional domain decomposition with a cross-point.

The idea of the BPS preconditioner now is to interpret the set of vertices V_j (see figure 11) as one coarse grid problem. We can discretize and solve this coarse grid problem with standard finite elements and direct or iterative equation solvers S_0. The trick now is to put this into an additive Schwarz framework and use standard interpolation Q_0 from and to the coarse grid.

$$B \; = \; \sum_j B_{E_j} \; + \; Q_0^* S_0 Q_0$$

A generalization to the three dimensional (see [Smi91] and figure 10)case reads as

$$B \; = \; \sum_j B_{E_j} \; + \; \sum_j B_{F_j} \; + \; Q_0^* S_0 Q_0$$

with faces F_j. Here it not necessary to use an expensive edge solver B_{E_j}. Diagonal scaling (Jacobi) is sufficient.

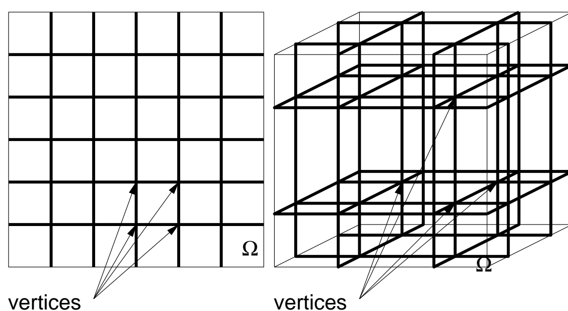


Figure 11: Vertices of the coarse grid.

5.2 Code

The implementation is based on the sample code for the Schur complement iteration `Schur1`[9]. The preconditioner is defined in the setting of additive Schwarz method

[9]this code is also in `Wire1/`

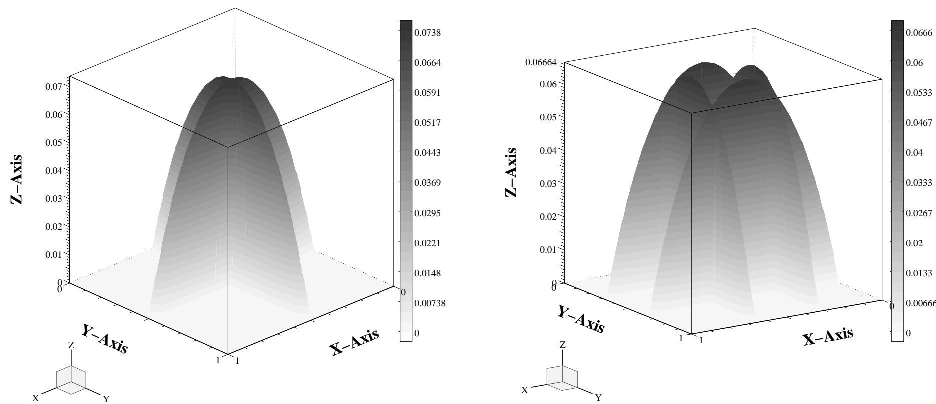


Figure 12: Solution on the interface. One cross-point/ four cross-points.

with the `DDSolver` programming interface. One coarse grid, called `vertex_system` with an associated interpolation `vertex_proj` is constructed. The edge and face preconditioner are implemented in a grid free manner via the tables `edge_trans`, `edge_dim` and `face_dim`. Both types of preconditioners share the same resources.

It is important to note the numbering scheme: Numbers 1 to `no_of_edges` enumerate the different edges, numbers `no_of_edges+1` to `no_of_edges+no_of_faces` enumerate the different faces and number `no_of_edges+no_of_faces+1` denotes the coarse grid, if there is any (see `no_vertex_space`=1?).

Wire1.h

```
// prevent multiple inclusion of Wire1.h
#ifndef Wire1_h_IS_INCLUDED
#define Wire1_h_IS_INCLUDED

#include <Schur1.h>
#include <DDSolver.h>           // DDSolver
#include <DDSolverUDC.h>        // interfacing to DDSolver
#include <DDSolver_prm.h>       // DDSolver parameters

class Wire1 : public Schur1, public DDSolverUDC
{
protected:
  int coarse_grid;
  // preconditioner related data:
  int no_vertex_space;                   // vertex space?
  int no_of_edges;                       // number of inner edges
  int no_of_faces;                       // number of inner faces (in 3D)

  VecSimplest(VecSimple(int))          edge_trans; // projection onto edge
  VecSimple(int)                       edge_dim; // dim of edge system
  MatSimple(int)                       face_dim; // dim of face system

  prm(DDSolver) ddsolver_prm;                         // parameters domain decomposition solver
```

```
  Handle(DDSolver)                      ddsolver;    // domain decomposition preconditioner

  // vertex space related data:
  Handle(LinEqSolver)        vert_solve;    // linear solution vertex space
  Handle(prm(LinEqSolver))   vert_solve_prm;// linear solution parameter
  Handle(LinEqSystemStd)     vert_system;   // linear system, storage
  Handle(prm(Matrix(NUMT)))  vert_mat_prm;  // Matrix parameters
  Handle(Proj)               vert_proj;     // projection operator

  virtual void scanGrids(MenuSystem& menu);// construct grids
  virtual void mark2D (Ptv(int) &part, Ptv(int) &subdom, Ptv(int) &dom);
  virtual void mark3D (Ptv(int) &part, Ptv(int) &subdom, Ptv(int) &dom);
  virtual void initInterface();        // setup interface numbering
  virtual void initMatrices();         // setup stiffness matrices on sub domains grids
  virtual void initVertex();           // setup vertex space matrix
  virtual void initVertexProj();       // setup projection to vertex space
  virtual BooLean solveEdge (LinEqVector& b, LinEqVector& x,
     SpaceId, StartVectorMode, DDSolverMode);
  virtual BooLean solveFace (LinEqVector& b, LinEqVector& x,
     SpaceId, StartVectorMode, DDSolverMode);
  virtual BooLean solveVertex (LinEqVector& b, LinEqVector& x,
       SpaceId, StartVectorMode, DDSolverMode);

public:
  Wire1 ();
 ~Wire1 () {}

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

  // DDSolverUDC
  SpaceId getNoOfSpaces() const;           // no_vertex_space+no_of_edges+no_of_faces
  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);

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

The codes for edge and face preconditioners has been presented previously. The main part is in some sense the `scanGrids` procedure, which together with `mark2D` and `mark3D` implements the numbering scheme. Using this scheme, the procedure `initInterface` defines the basic translation table `edge_trans` that is a primitive version of a projection operator used in `transfer`. It also counts the dimensions of the different sub-spaces needed in `solveEdge` and `solveFace`.

Wire1.C

```
#include <Wire1.h>
#include <PreproBox.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <ErrorEstimator.h>
#include <Vec_real.h>
#include <createElmDef.h>       // for calling hierElmDef in Wire1::define
#include <createMatrix_real.h>  // creating stiffness matrices
#include <createLinEqSolver.h>  // creating sub domain solver
#include <splitMatrix_real.h>   // splitting stiffness matrices
#include <PrecDD.h>
#include <createDDSolver.h>     // creating multigrid object
#include <FFT.h>

Wire1:: Wire1 () {}

void Wire1:: define (MenuSystem& menu, int level)
{
  Schur1:: define (menu, level);
  prm(DDSolver)::  defineStatic (menu, level+1);// DD parameters
  menu.setCommandPrefix("Vertex");
  prm(LinEqSolver)::defineStatic (menu, level+1);// sub domain solver parameters
  menu.unsetCommandPrefix();
}

void Wire1:: mark2D (Ptv(int) &part, Ptv(int) &subdom, Ptv(int) &dom)
{
  const int nsd = 2;
  int counter = 2; // number 1 is outer Dirichlet BC condition, start with 2
  int i;
  Ptv(int) pt(nsd);
  // mark edges on grid(global_grids)
  for (i = 1; i <= nsd; i++) // mark edges 2D
    for (pt(1)= (i==1)?0 : 1; pt(1)<part(1); pt(1)++)
      for (pt(2)= (i==2)?0 : 1; pt(2)<part(2); pt(2)++) {
String boInd = aform("n=1 b%d =", counter);
for (int j = 1; j <= nsd; j++) {
  if (i!=j) {
    real x = (pt(j) * subdom(j)) / (real) dom(j);   // e.g. [.5,.5]
    boInd += aform("[%g,%g]", x, x);
  } else {
    real x1 = (pt(j) * subdom(j)) / (real) dom(j);  // e.g. [.2,.4]
    real x2 = ((pt(j)+1) * subdom(j)) / (real) dom(j);
    boInd += aform("[%g,%g]", x1, x2);
  }
}
grid(global_grid)->addBoIndNodes(boInd);
counter++;
      }
}

void Wire1:: mark3D (Ptv(int) &part, Ptv(int) &subdom, Ptv(int) &dom)
{
  const int nsd = 3;
  int counter = 2; // number 1 is outer Dirichlet BC condition, start with 2
  int i;
  Ptv(int) pt(nsd);
  // mark edges on grid(global_grids)
  for (i = 1; i <= nsd; i++) // mark edges 3D
```

```
    for (pt(1)= (i==1)?0 : 1; pt(1)<part(1); pt(1)++)
      for (pt(2)= (i==2)?0 : 1; pt(2)<part(2); pt(2)++)
for (pt(3)= (i==3)?0 : 1; pt(3)<part(3); pt(3)++) {
  String boInd = aform("n=1 b%d =", counter);
  for (int j = 1; j <= nsd; j++) {
    if (i!=j) {
      real x = (pt(j) * subdom(j)) / (real) dom(j);   // e.g. [.5,.5]
      boInd += aform("[%g,%g]", x, x);
    } else {
      real x1 = (pt(j) * subdom(j)) / (real) dom(j);  // e.g. [.2,.4]
      real x2 = ((pt(j)+1) * subdom(j)) / (real) dom(j);
      boInd += aform("[%g,%g]", x1, x2);
    }
  }
  grid(global_grid)->addBoIndNodes(boInd);
  counter++;
}

  for (int i1 = 1; i1 < nsd; i1++) // mark faces 3D
    for (int i2 = i1+1; i2 <= nsd; i2++)
      for (pt(1)= (i1==1 || i2==1)?0 : 1; pt(1)<part(1); pt(1)++)
for (pt(2)= (i1==2 || i2==2)?0 : 1; pt(2)<part(2); pt(2)++)
  for (pt(3)= (i1==3 || i2==3)?0 : 1; pt(3)<part(3); pt(3)++) {
    String boInd = aform("n=1 b%d =", counter);
    for (int j = 1; j <= nsd; j++) {
      if (i1!=j && i2!=j) {
real x = (pt(j) * subdom(j)) / (real) dom(j);   // e.g. [.5,.5]
boInd += aform("[%g,%g]", x, x);
      } else {
real x1 = (pt(j) * subdom(j)) / (real) dom(j);  // e.g. [.2,.4]
real x2 = ((pt(j)+1) * subdom(j)) / (real) dom(j);
boInd += aform("[%g,%g]", x1, x2);
      }
    }
    grid(global_grid)->addBoIndNodes(boInd);
    face_dim(counter-1, 1) = subdom(i1)-1;
    face_dim(counter-1, 2) = subdom(i2)-1; // store geometric info
    counter++;
  }
}

void Wire1:: scan (MenuSystem& menu)
{
  Schur1:: scan (menu);

  // prm for DD preconditioner
  Handle(prm(Precond)) precondPrm = new prm(Precond);
  precondPrm->scan(menu);
  lineq->attach (precondPrm());

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

  menu.setCommandPrefix("Vertex");
    vert_solve_prm.rebind(new prm(LinEqSolver));
    vert_solve_prm->scan (menu);
    vert_solve.rebind(createLinEqSolver (vert_solve_prm()));
    vert_system.rebind(new LinEqSystemStd (EXTERNAL_STORAGE));
```

```
  menu.unsetCommandPrefix();
  vert_mat_prm.rebind(new prm(Matrix(NUMT)) );
  vert_mat_prm->scan (menu);
  vert_mat_prm->sparse_adrs.rebind (new SparseDS);
}

void Wire1:: scanGrids (MenuSystem& menu) // construct grids
{
  int i;
  int nsd = menu.get ("no of space dimensions").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 (',');
  }

  for (i = 1; i <= nsd; i++) {
    int n = subdom(i);
    while (n>1) {
      if (n%2 == 1) {
errorFP("Wire1:: scanGrids, scan subdomain",
"subdomain(%d)=%d must be a power of two for FFT",
i, subdom(i));
break;
      }
      n = n / 2;
    }
  }

  no_vertex_space = 1;
  for (i = 1; i <= nsd; i++)
    if (part(i)<2) {
      no_vertex_space = 0;
      break;
    }

  no_of_edges = 0;
  for (i = 1; i <= nsd; i++) {
    int n = 1;
    for (int j=1; j<= nsd; j++)
      if (i==j)
n *= part(j);
      else
n *= part(j) -1;
    no_of_edges += n;
  }
```

```
  no_of_faces = 0;
  if (nsd>2)
    for (i = 1; i < nsd; i++)
      for (int j=i+1; j<= nsd; j++) {
int n = 1;
for (int k=1; k<= nsd; k++)
  if (i==k || j==k)
    n *= part(k);
  else
    n *= part(k) -1;
no_of_faces += n;
      }

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

  no_of_grids = 1;
  for (i = 1; i<= nsd; i++)
    no_of_grids *= part(i);
  local_grid = no_of_grids;
  no_of_grids += 1;   // global grid
  global_grid = no_of_grids;
  no_of_grids += no_vertex_space;
  coarse_grid = no_of_grids;   // coarse grid

  edge_trans.redim(no_of_edges + no_of_faces);
  edge_dim.redim(no_of_edges + no_of_faces);
  if (no_of_faces>0)
    face_dim.redim(no_of_edges + no_of_faces, 2);

  sub_solve.redim  (no_of_grids-1);
  system.redim     (no_of_grids-1);
  sub_solve_prm.redim (no_of_grids-1);
  proj.redim       (no_of_grids-1);
  grid.redim       (no_of_grids);
  dof.redim        (no_of_grids);
  mat_prm.redim    (no_of_grids-1);
  mat_inner.redim  (no_of_grids-1);
  mat_trans1i.redim  (no_of_grids-1);
  mat_transi1.redim  (no_of_grids-1);
  rhs_inner.redim  (no_of_grids-1);
  num_inner.redim  (no_of_grids-1);
  num_trans.redim  (no_of_grids-1);
  inner_dim.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<=local_grid) {
    int ix = k % part(j); // split into row, column ...
    k = k / part(j);
    x0 = (ix     * subdom(j)) / (real) dom(j);
    x1 = ((ix+1) * subdom(j)) / (real) dom(j);
  } else // global or coarse grid
    { 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<=local_grid) n = subdom(j);
  else if (i==global_grid) n = dom(j);
  else n = part(j);  // coarse 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 boInd_g = "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==global_grid)||(i==coarse_grid))
    boInd_g += b1;
  else boInd_i += b1;

  String b0 = aform("%d ", j+nsd);
  if ((ix==1)||(i==global_grid)||(i==coarse_grid))
    boInd_g += b0;
  else boInd_i += b0;
}
boInd_g += boInd_i;
boInd_g += ")";
if (i!=global_grid)
  boInd_g = "nb=2 names= global 2in " + boInd_g;
```

```
    else {
      String boInd_n = aform("nb=%d names= global inner ",
     1 + no_of_edges + no_of_faces);
      for (j=3; j<= 1 + no_of_edges + no_of_faces; j++) {
boInd_n += aform(" %din ", j);
boInd_g += aform(", %d=()", j);
      }
      boInd_g = boInd_n + boInd_g;
    }
    grid(i)->redefineBoInds(boInd_g);
  }

  if (nsd==2) mark2D (part, subdom, dom);
  else if (nsd==3) mark3D (part, subdom, dom);

  FEM::scan (menu);  // load type and order of the numerical integration rule
  Store4Plotting::scan (menu, grid(global_grid)->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(global_grid)->getNoNodes()
      << ",\tno of elements: " << grid(global_grid)->getNoElms();
  s_o << "\n No of edges: "<<no_of_edges<<"\tNo of faces: "<<no_of_faces
      <<"  \tVertex space active: "<<no_vertex_space<<endl;
}

void Wire1:: initInterface() //  set up edge_trans, edge_dim
{
  Schur1:: initInterface();

  int nno = grid(global_grid)->getNoNodes(); // no of nodes
  VecSimple(int) mark(nno);
  mark = 0;
  int j, k;

  edge_dim = 0;
  for (j = 1; j <= nno; j++)                    // count dimension
    if (grid(global_grid)->BoNode (j))
      if (notBooLean(grid(global_grid)->BoNode (j, 1))) { // interface
int ed = 0, e;
for (k=1; k<= no_of_edges; k++)
  if (grid(global_grid)->BoNode (j, k+1)) {
    ed++;
    e = k;
  }
if (ed==1) {
  edge_dim(e) ++; // edge, if ed>1 vertex
  mark(j) = e;
}
else {
  int vd = 0, v;
  for (k=no_of_edges+1; k<= no_of_edges + no_of_faces; k++)
    if (grid(global_grid)->BoNode (j, k+1)) {
      vd++;
      v = k;
    }
  if (vd==1) {
```

```
    edge_dim(v) ++; // face, if vd>1 error
    mark(j) = v;
  }
  else ;//s_o<<"=unknown"; // vertex
}
      }

  for (k=1; k<= no_of_edges + no_of_faces; k++) // redim
    edge_trans(k).redim(edge_dim(k));

  edge_dim = 0;
  for (j = 1; j <= nno; j++) {
    int m = mark(j);
    if (m>0) {                  // assign node numbers
      edge_dim(m) ++;
      edge_trans(m)(edge_dim(m)) = num_interf(j); // != 0
    }
  }
}

void Wire1:: initVertexProj() // setup proj operators
{
  ProjInterpSparse p;
  p.rebindDOF(*dof(coarse_grid), *dof(global_grid));
  p.init();

  int i, n = grid(coarse_grid)->getNoNodes();
  VecSimple(int) vert(n);
  for (i=1; i<=n; i++) vert(i) = i;

  VecSimple(int) inter(interface_dim);
  int m = dof(global_grid)->getTotalNoDof ();
  for (i=1; i<= m; i++)
    if (num_interf(i)>0)
      inter(num_interf(i)) = i;

  Handle(Matrix(real)) A;
  A = splitMatrix(real) (p.A().mat(), inter, vert);

  Handle(LinEqMatrix) a;
  a = new LinEqMatrix(A());
  vert_proj = new ProjMatrix();
  vert_proj->rebindMatrix(a());
  vert_proj->init();
}

void Wire1:: initVertex()   // setup vertex space matrix
{
  fillEssBC (coarse_grid);     // set essential boundary conditions
  Handle(Vec(NUMT)) u;
  Handle(Vec(NUMT)) rhs;
  u = new Vec(NUMT)  (dof(coarse_grid)->getTotalNoDof ());
  rhs = new Vec(NUMT) (dof(coarse_grid)->getTotalNoEqs ());

  vert_mat_prm->nrows = dof(coarse_grid)->getTotalNoEqs ();
  vert_mat_prm->ncolumns = dof(coarse_grid)->getTotalNoDof ();
  vert_mat_prm->nsd = dof(coarse_grid)->grid().getNoSpaceDim();

  if (vert_mat_prm->storage == "MatStructSparse")
```

```
    makeSparsityPattern (vert_mat_prm->offset,
                         vert_mat_prm->ndiagonals,  dof(coarse_grid)());
  else if (vert_mat_prm->storage.contains("Sparse"))
    makeSparsityPattern (vert_mat_prm->sparse_adrs(), dof(coarse_grid)());
  else if (vert_mat_prm->storage == "MatBand")
    vert_mat_prm->bandwidth = dof(coarse_grid)->getHalfBandwidth();

  Handle(Matrix(NUMT)) A;
  A = createMatrix(NUMT) (vert_mat_prm());

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

  vert_system->attach(A());
  ddsolver->attachLinRhs(rhs(), getNoOfSpaces()-1, dpTRUE);
  ddsolver->attachLinSol(u(), getNoOfSpaces()-1);

  initVertexProj();
}

void Wire1:: initMatrices()   // setup stiffness matrices on sub-domains
{
  Schur1:: initMatrices();

  for (int i=1; i<=no_of_edges + no_of_faces; i++) {
    Handle(Vec(NUMT)) rhs_n = new Vec(NUMT) (edge_dim(i));
    Handle(Vec(NUMT)) u_n   = new Vec(NUMT) (edge_dim(i));
    ddsolver->attachLinRhs(rhs_n(), i, dpFALSE);
    ddsolver->attachLinSol(u_n(), i);
  }

  if (no_vertex_space>0)
    initVertex();
}

extern real analyticalSolution (const Ptv(real)& x, real t);

void Wire1:: solveProblem () // main routine of class Wire1
{
  initProj();                  // set up transfer global <> local
  initInterface();             // set up splitting Schur <> inner systems
  initMatrices();              // local matrices, split into inner & Schur part
  initRhs();                   // compute rhs for Schur system
  linsol.fill (0.0);           // set all entries to 0 in start vector Schur

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

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

  lineq->solve();              // solve Schur 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);

  projSol();                    // compute global solution from Schur solution
  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.
}

SpaceId Wire1:: getNoOfSpaces() const
{
  return no_of_edges + no_of_faces + no_vertex_space + 1;
}

BooLean Wire1:: solveEdge (
  LinEqVector& b, LinEqVector& x,
  SpaceId space, StartVectorMode /*start*/, DDSolverMode /*mode*/)
{
  int i;
  int n = edge_dim(space);
  if (n==0) return dpFALSE;
  Vec(NUMT) &rhs = CAST_REF(b.vec(), Vec(NUMT));
  Vec(NUMT) &sol = CAST_REF(x.vec(), Vec(NUMT));

  Vec(NUMT) f(n+1); // must be a power of two for FFT
  f(1) = 0;
  for (i=1; i<=n; i++)
    f(i+1) = rhs(i);

  FFT::sinft (f, n+1, 0); // forward

  real a = M_PI / (2*(n+1)); // scale
  real s;
  for (i=1; i<=n; i++) {
    s = sin(i * a);
    // f(i) = f(i) / (2 * s);  // Dryja
    f(i) = f(i) / (2 * s * sqrt(1 + s*s));  // Golub & Meyers
  }

  FFT::sinft (f, n+1, 1); // backward

  for (i=1; i<=n; i++)
    sol(i) = f(i+1);
  return dpTRUE;
}

BooLean Wire1:: solveFace (
  LinEqVector& b, LinEqVector& x,
  SpaceId space, StartVectorMode /*start*/, DDSolverMode /*mode*/)
{
  if (edge_dim(space)==0) return dpFALSE;
  int i, j;
  Vec(NUMT) &rhs = CAST_REF(b.vec(), Vec(NUMT));
  Vec(NUMT) &sol = CAST_REF(x.vec(), Vec(NUMT));
```

```
  int n1 = face_dim(space, 1);
  int n2 = face_dim(space, 2); // n1 * n2 == edge_dim(space);
  Vec(NUMT) f1(n1+1); // must be a power of two for FFT
  Vec(NUMT) f2(n2+1); // must also be a power of two for FFT
  f1(1) = f2(1) = 0;

  for (j=1; j<=n2; j++) { // forward
    for (i=1; i<=n1; i++)
      f1(i+1) = rhs(i + (j-1)*n1);
    FFT::sinft (f1, n1+1, 0);
    for (i=1; i<=n1; i++)
      sol(i + (j-1)*n1) = f1(i+1);
  }
  for (i=1; i<=n1; i++) {
    for (j=1; j<=n2; j++)
      f2(j+1) = sol(i + (j-1)*n1);
    FFT::sinft (f2, n2+1, 0);
    for (j=1; j<=n2; j++)
      sol(i + (j-1)*n1) = f2(j+1);
  }

  real a1 = M_PI / (2*(n1+1));
  for (i=1; i<=n1; i++) {
    real s = sin(i * a1);
    f1(i+1) = 4 * s * s;
  }
  real a2 = M_PI / (2*(n2+1));
  for (j=1; j<=n2; j++) {
    real s = sin(j * a2);
    f2(j+1) = 4 * s * s;
  }
  for (i=1; i<=n1; i++)
    for (j=1; j<=n2; j++)
      sol(i + (j-1)*n1) /= sqrt(f1(i+1) + f2(j+1));  // scale

  for (i=1; i<=n1; i++) { // backward
    for (j=1; j<=n2; j++)
      f2(j+1) = sol(i + (j-1)*n1);
    FFT::sinft (f2, n2+1, 1);
    for (j=1; j<=n2; j++)
      sol(i + (j-1)*n1) = f2(j+1);
  }
  for (j=1; j<=n2; j++) {
    for (i=1; i<=n1; i++)
      f1(i+1) = sol(i + (j-1)*n1);
    FFT::sinft (f1, n1+1, 1);
    for (i=1; i<=n1; i++)
      sol(i + (j-1)*n1) = f1(i+1);
  }
  return dpTRUE;
}

BooLean Wire1:: solveVertex (
  LinEqVector& b, LinEqVector& x,
  SpaceId /*space*/, StartVectorMode /*start*/, DDSolverMode /*mode*/)
{
  Vec(NUMT) &rhs = CAST_REF(b.vec(), Vec(NUMT));
  dof(coarse_grid)->fillEssBC (rhs);
```

```
  vert_solve_prm->startmode = ZERO_START;
  vert_system ->attach (x, b);
  vert_solve ->solve  ( vert_system() );
  vert_system->allow_factorization = dpFALSE;

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

  return dpTRUE;
}

BooLean Wire1:: solveSubSystem (
  LinEqVector& b, LinEqVector& x,
  SpaceId space, StartVectorMode start, DDSolverMode mode)
{
  if (space <= no_of_edges)
    return solveEdge(b, x, space, start, mode);
  else if (space-no_of_edges <= no_of_faces)
    return solveFace(b, x, space, start, mode);
  return solveVertex(b, x, space, start, mode);
}

BooLean Wire1:: transfer (
   const LinEqVector& fv, SpaceId fi, LinEqVector& tv, SpaceId ti,
   BooLean add_to_t, DDTransferMode)
{
  const Vec(NUMT) &fvv = CAST_REF(fv.vec(), Vec(NUMT));
  Vec(NUMT) &tvv = CAST_REF(tv.vec(), Vec(NUMT));

  if (ti == getNoOfSpaces()) {
    if (fi <= no_of_edges + no_of_faces) {
      if (add_to_t) {
for (int j=1; j<=edge_dim(fi); j++)
  tvv (edge_trans(fi)(j)) += fvv (j);
      } else { // not used
tvv = 0.;
for (int j=1; j<=edge_dim(fi); j++)
  tvv (edge_trans(fi)(j)) = fvv (j);
      }
    } else {
      vert_proj->apply(fv, tv, NOT_TRANSPOSED, add_to_t);
    }
  } else if (fi == getNoOfSpaces()) {
    if (ti <= no_of_edges + no_of_faces) {
      if (add_to_t) { // not used
for (int j=1; j<=edge_dim(ti); j++)
  tvv (j) += fvv (edge_trans(ti)(j));
      } else {
for (int j=1; j<=edge_dim(ti); j++)
  tvv (j) = fvv (edge_trans(ti)(j));
      }
    } else {
      vert_proj->apply(fv, tv, TRANSPOSED, add_to_t);
    }
  }
  else fatalerrorFP("Wire1:: transfer","undefined");
  return dpTRUE;
}
```

```
int Wire1:: getWorkTransfer (SpaceId fi, SpaceId ti, const PrecondWork) const
{
  int i;
  if (ti == getNoOfSpaces()) i = fi;
  else if (fi == getNoOfSpaces()) i = ti;
  else return 0;
  if (i <= no_of_edges + no_of_faces)
    return edge_dim(i);
  return vert_proj->getWork();
}

real Wire1:: getStorageTransfer (SpaceId fi, SpaceId ti) const
{
  if (ti == getNoOfSpaces()) return 0;
  if (fi != getNoOfSpaces()) return 0;
  if (ti <= no_of_edges + no_of_faces)
    return edge_dim(ti);
  return vert_proj->getStorage();
}

int Wire1:: getWorkSolve (SpaceId space, const PrecondWork) const
{
  if (space <= no_of_edges + no_of_faces)
    return edge_dim(space);
  return vert_solve->getWork();
}

real Wire1:: getStorageSolve (SpaceId space) const
{
  if (space <= no_of_edges + no_of_faces)
    return edge_dim(space);
  return vert_solve->getStorage();
}

String Wire1:: comment ()
{ return "wire-basket preconditioner"; }
```

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

We start our experiments with the generic implementation of BPS and wire-basket methods with tests of the components. The first test is about the eigen-decomposition based preconditioners for a line and for stripes, see table 19, input file `prec1.i` and figure 7. Compare the number of iterations with the un-preconditioned version of the conjugated gradient method (in `Schur1`) and with the Neumann-Neumann preconditioner. Note that we had to choose powers of two as sub-domain sizes because of our implementation of the discrete sine transform. You can also compare the two versions of the edge preconditioner (Dryja/ Golub & Mayers). Compare the number of operations for the different preconditioners and iteration types.

This two dimensional test requires the coarse grid solver and the BPS preconditioner, see table 20, input file `prec2.i` and figure 9.

This test is about the face preconditioner in three dimensions, see table 21, input file

[10] files are in `Wire1/Verify/`

menu item	answer
no of space dimensions	2
subdomain	[8,8]
partition	{[2,1] & [3,1]}
element type	ElmB4n2D
matrix type	MatSparse
basic method	ConjGrad
preconditioning type	PrecDD
left preconditioning	ON
#1: convergence monitor name	CMAbsResidual
local basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 19: Eigen-decomposition type stripe preconditioner, `prec1.i`

menu item	answer
no of space dimensions	2
subdomain	[4,4]
partition	{[2,2] & [3,3]}
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Vertex basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 20: BPS preconditioner, eigen-decomposition on faces and direct coarse grid solver, `prec2.i`

`prec3.i` and figure 6 left.

This test is about the edge and the face preconditioner in three dimensions, see table 22, input file `prec3b.i` and figure 6 right.

This test is about the wire-basket preconditioner in three dimensions combining edges preconditioners, face preconditioners and a coarse grid solver, see table 23, input file `prec3.i`.

This test is about the effect of inexact coarse grid solvers in a BPS preconditioner, see table 24, input file `prec5.i`.

The last experiment covers inexact Dirichlet solvers in the Schur complement and an inexact coarse grid solver in the BPS preconditioner, see table 25, input file `prec4.i` and figure 10 right.

menu item	answer
no of space dimensions	3
subdomain	[4,4,4]
partition	{[2,1,1] & [3,1,1]}
element type	ElmB8n3D
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 21: Eigen-decomposition type face preconditioner in 3D, `prec3.i`

menu item	answer
no of space dimensions	3
subdomain	[4,4,4]
partition	{[2,2,1] & [3,2,1]}
element type	ElmB8n3D
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 22: Eigen-decomposition type edge and face preconditioner in 3D, `prec3b.i`

6 Inexact Schur complement regarded as a preconditioner

The major limitation of the Schur complement algorithm so far is the need of precise Dirichlet solvers. The evaluation of

$$S\,x \;=\; (A_{\mathcal{I}\mathcal{I}} - \sum_k A_{\mathcal{I}k}\, A_{kk}^{-1}\, A_{k\mathcal{I}})x$$

requires the solution of a local Dirichlet type problem A_{kk}^{-1} on each sub-domain. In the case we do not solve these problems precise enough, we introduce errors to the right hand side and the residual evaluations performed by an iterative solver. This systematic error will not be reduced by additional iterations or other efforts to improve the outer loop. This is in contrast to preconditioners for example, where a perturbation usually can be compensated by an increased number of iterations. However the only way to improve the final solution for Schur complement method is to start with a better solver for A_{kk}^{-1} right from the beginning.

Of course it would be convenient to be able to use approximate iterative solvers for the Dirichlet problems, since the sub-problems may be of considerable size itself. The idea is now to consider the Schur complement method itself as a preconditioner acting on the global system. In this sense the Schur complement method may be inexact using itself inexact Dirichlet solvers.

menu item	answer
no of space dimensions	3
subdomain	[4,4,4]
partition	{[2,2,2] & [3,3,3]}
element type	ElmB8n3D
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Vertex basic method	GaussElim
domain decomposition method	AddSchwarzDD

Table 23: Wire basket preconditioner in 3D with direct coarse grid solver, `prec3c.i`

menu item	answer
no of space dimensions	2
subdomain	[4,4]
partition	[4,4]
element type	ElmB4n2D
matrix type	Mat
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	GaussElim
Vertex basic method	SSOR
Vertex startvector mode	ZERO_START
Vertex max iterations	{1 & 2 }
domain decomposition method	AddSchwarzDD

Table 24: BPS preconditioner with inexact coarse grid solver, `prec5.i`

We write the Schur complement as a preconditioner. We start with an exact decomposition of the stiffness matrix.

$$\begin{aligned} A &= \begin{pmatrix} A_{11} & 0 & A_{1\mathcal{I}} \\ 0 & A_{22} & A_{2\mathcal{I}} \\ A_{\mathcal{I}1} & A_{\mathcal{I}2} & A_{\mathcal{I}\mathcal{I}} \end{pmatrix} \\ &= \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ A_{\mathcal{I}1}A_{11}^{-1} & A_{\mathcal{I}2}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & 0 & 0 \\ 0 & A_{22} & 0 \\ 0 & 0 & S \end{pmatrix} \begin{pmatrix} I & 0 & A_{11}^{-1}A_{1\mathcal{I}} \\ 0 & I & A_{22}^{-1}A_{2\mathcal{I}} \\ 0 & 0 & I \end{pmatrix} \end{aligned}$$

Inverting this, we get a preconditioner

$$B = \begin{pmatrix} I & 0 & -A_{11}^{-1}A_{1\mathcal{I}} \\ 0 & I & -A_{22}^{-1}A_{2\mathcal{I}} \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} A_{11}^{-1} & 0 & 0 \\ 0 & A_{22}^{-1} & 0 \\ 0 & 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ -A_{\mathcal{I}1}A_{11}^{-1} & -A_{\mathcal{I}2}A_{22}^{-1} & I \end{pmatrix}$$

Of course we cannot evaluate S^{-1}, but we substitute this term by one of the preconditioners for the Schur complement system discussed earlier. We can approximate

menu item	answer
no of space dimensions	2
subdomain	[8,8]
partition	[2,2]
basic method	ConjGrad
preconditioning type	PrecDD
local basic method	SSOR
local startvector mode	ZERO_START
local max iterations	5
Vertex basic method	SSOR
Vertex startvector mode	ZERO_START
Vertex max iterations	1
domain decomposition method	AddSchwarzDD

Table 25: Inexact Dirichlet and coarse grid solvers, BPS preconditioner, `prec4.i`

S^{-1} for example by the Neumann-Neumann preconditioner or the wire-basket preconditioner.

In this framework we can make further approximations substituting the Dirichlet problems A_{kk}^{-1} by some iterative solvers like Krylov methods, multigrid iterations or other domain decomposition methods. Even one step of such an iterative procedure is a legal substitution.

Our main algorithm now is a Krylov iteration on the global domain

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_{\mathcal{I}} \end{pmatrix}$$

using the preconditioner B and the system matrix A.

The implementation of the scheme, starting with `Schur1` and constructing a preconditioner for the global system, analogue to the overlapping Schwarz preconditioner in `Overlap1`, is left to the reader. One has to be careful to separate the preconditioner for the global system (the Schur complement iteration) and the preconditioner for the Schur complement itself.

7 Conclusion

In this report we have demonstrated the use of non-overlapping Schur complement domain decomposition methods in `Diffpack`. We have introduced the concept of the Schur complement. Instead of solving a global equation system, one computes the solution of the problem only on the interface between the sub-domains. It is easy to extend this solution to the solution of the global system. The problem on the interface is smaller and better conditioned than the original system.

However, it is too expensive to compute the Schur complement matrix explicitly. So we solve the equations with a Krylov (conjugated gradient) method that requires only

matrix multiplications. Each multiplication with the Schur complement requires the solution of Dirichlet type problems on each sub-domain.

In order to accelerate the iterative solution on the interface, we construct several preconditioners. One family of general purpose preconditioner are the Neumann-Neumann preconditioner, which require the solution of a Neumann type problem on each sub-domain. We also discuss modifications of the preconditioner and the application of a coarse grid (balancing domain decomposition).

Next we introduce preconditioners, which employ the decomposition into eigen-modes via a discrete sine transform. Although restricted to geometrically simple interfaces, they serve as a building block for BPS and wire-basket type preconditioners. These preconditioners combine preconditioners for edges (and faces in three dimensions) with a coarse grid represented by the vertices (cross-points) in an additive Schwarz manner.

Iterative sub-structuring methods, or non-overlapping domain decomposition methods, how they are called sometimes, do not suffer from discontinuous coefficients in a problem, as long as the jumps are resolved by the decomposition. This is a clear advantage compared to overlapping Schwarz methods. Another nice feature is the possibility to combine domains with different materials, properties or even models, as long as one is able to set up a global equation system. However, the main application is probably parallel computing, where each sub-domain is mapped to a single processor. Here it is important to have a solution procedure with a small number and volume of communication of communication compared to the local work on each processor, which is clearly the case for the iterative sub-structuring method.

There are several other promising preconditioner for the Schur complement, which we did not mention in this report. Among these preconditioners are vertex space methods, that introduce some overlapping domain around each vertex, algebraic preconditioners like "probing" and methods like Dirichlet-Neumann. Of course it is possible to implement them also in the given framework in `Diffpack`, but we leave this as future work.

References

[BGLV89] J. F. Bourgat, R. Glowinski, P. LeTallec, and M. Vidrascu. Variational formulation and algorithm for trace operator in domain decomposition calculations. In T. F. Chan, R. Glowinski, J. Périaux, and O. B. Widlund, editors, *Proc. Second Int. Conf. on Domain Decomposition Meths.*, pages 3–16. SIAM, Philadelphia, 1989.

[BL96] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations; Diffpack. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics.* Birkhäuser, 1996.

[BPS86] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. An iterative method for elliptic problems on regions partitioned into substructures. *Math. Comp.*, 46:361–369, 1986.

[CM94] T. F. Chan and T. P. Mathew. Domain decomposition algorithms. In *Acta Numerica*, pages 61–143. Cambridge Univ. Press, 1994.

[Dry81] M. Dryja. An algorithm with a capacitance matrix for a variational–difference scheme. In G. I. Marchuk, editor, *Variational–Differnece Methods in Mathematical Physics*, pages 63–73. USSR Academy of Sciences, Novosibirsk, 1981.

[DW91] M. Dryja and O. Widlund. *Additive Schwarz Methods for Elliptic Finite Element Problems in Three Dimensions.* Courant Institute, New York, 1991. Technical Report 570.

[GM84] G. H. Golub and D. Mayers. The use of preconditioning over irregular regions. In R. Glowinski and J.-L. Lions, editors, *Computing Methods in Applied Sciences and Engineering VI*, pages 3–14. North–Holland, Amsterdam, 1984.

[GPSW96] R. Glowinski, J. Périaux, Z.-C. Shi, and O. B. Widlund, editors. *Proc. Eighth Int. Conf. on Domain Decomposition Meths.* Wiley and Sons, Chichester, 1996.

[KX95] D. E. Keyes and J. Xu, editors. *Proc. Seventh Int. Conf. on Domain Decomposition Meths.* Number 180 in Contemporary Mathematics. AMS, Providence, 1995.

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

[Man93] J. Mandel. Balancing domain decomposition. *Comm. Numer. Meth. Engrg.*, 9:233–241, 1993.

[RL91] Y-H. de Roeck and P. LeTallec. Analysis and test of a local domian decomposition. In R. Glowinski, Yu. A. Kuznetsov, G. A. Meurant,

J. Périaux, and O. B. Widlund, editors, *Proc. Fourth Int. Conf. on Domain Decomposition Meths.*, pages 112–128, Philadelphia, 1991. SIAM.

[SBG96] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, 1996.

[Smi91] B. F. Smith. A domain decomposition algorithm for elliptic problems in three dimensions. *Numer. Math.*, 60(2):210–234, 1991.

[Zum96a] G. W. Zumbusch. Multigrid methods in Diffpack. Technical Report STF42 F96016, SINTEF Applied Mathematics, Oslo, 1996.

[Zum96b] G. W. Zumbusch. Overlapping domain decompsoition methods in Diffpack. Technical report, SINTEF Applied Mathematics, Oslo, 1996.