Multigrid Methods in Diffpack

Kent-Andre Mardal *  Gerhard W. Zumbusch †
Hans Petter Langtangen ‡

December 19, 2000

Contents

1 Introduction 2

2 Sketch of How Multilevel Methods are Implemented in Diffpack 4

3 Extending an Existing Application with Multigrid Methods 6

4 Setting up an Input File 10
  4.1 Running a Plain Gaussian Elimination .......................... 10
  4.2 Filling Out MGtools Menu Items ................................. 11

5 Playing Around with Multigrid 13
  5.1 Number of Grids and Iterations ................................. 13
  5.2 Smoother .................................................................. 15
  5.3 W Cycle and Nested Iteration ..................................... 17

---

* Numerical Objects A.S. Email: kent-and@ifi.uio.no
† Institute for Applied Mathematics, University of Bonn, Germany. Email: zumbusch@iam.uni-bonn.de
‡ Department of Informatics, University of Oslo. Email: hpl@ifi.uio.no.
5.4 Coarse-Grid Solver ................................................. 20
5.5 Multigrid as a preconditioner ................................. 22
5.6 Additive Preconditioner .......................................... 24

6 Equipping the Poisson2 Solver with Multigrid ......... 25

Abstract

This report provides a tutorial for the multi-level solver module in Diffpack. The current version of the report is restricted to multigrid methods. With the \texttt{Multigrid} class, a standard Diffpack finite element application code can be equipped with a flexible multigrid solver by adding about 10 lines of extra code. The applications covered here are the Poisson equation, the equation of linear elasticity, and a nonlinear Poisson-equation problem.

1 Introduction

When discretizing partial differential equations by the finite element, difference, or volume methods, one often ends up with a system of linear equations (hereafter called \textbf{linear system}). In time-dependent problems such linear systems must be solved at each time level. Quite frequently one experiences that the solution of the linear system is the bottleneck of the simulation, especially on large grids. The reason for this is easy to explain. Linear systems solvers (often just called \textbf{linear solvers}) require work of the order of \( n^\alpha \), where \( n \) is the number of unknowns in the system and \( \alpha \geq 1 \). If \( \alpha > 1 \), which is the common case for Gaussian elimination, classical iterations, and conjugate-gradient-like methods [2], the solution of linear systems will dominate over the work required to set up the system (which is proportional to \( n \)).

In 70s and 80s, researchers constructed so-called \textbf{multi-level methods} having the nice property that \( \alpha = 1 \). These methods are characterized by employing a hierarchy of grids, or in other words, multi-level discretizations. Special cases of multi-level strategies are the multigrid and the domain decomposition methods. The current report is restricted to multigrid methods.

Looking at the whole family of linear solvers, one observes that direct methods, like Gaussian elimination and its variants, require no user-chosen parameters and the amount of work is known a priori, while iterative methods
mostly involve user-chosen, problem-dependent parameters and the amount of work is not known. It is also uncertain whether iterative methods will converge or not. The family of multigrid methods is the extreme part of this picture; it contains a wide range of user-chosen, problem-dependent parameters: the number and the type of grids, pre- and post-smoothing solvers, the number of pre- and post-smoothing iterations, the type of coarse-grid solver, the type of multigrid strategy (e.g., V-cycles versus W-cycles), and so on. For most applications, appropriate values of these user-chosen parameters are not known, or at best, some rough guidelines from numerical analysis and previous experience exist. A few simple problems have been analysed in detail, and the most effective multigrid strategies are then known.

To obtain an effective and working multigrid method the user must normally carry out lots of experiments to tune the various parameters. This requires great flexibility in the implementation of the multigrid solver, which is exactly the advantage of the Diffpack multigrid toolbox. The purpose of this report is to show how easy it is to implement the (complicated) multigrid algorithm in an existing Diffpack finite element solver and how flexible this implementation is.

The required background for reading the report is knowledge of typical finite element solvers in Diffpack, like the Poisson, Poisson2, and Elasticity1 solvers from [2]. Furthermore, the reader must know the basics of multigrid algorithms, e.g., from [2, app. C.4.2]. In particular, we will use mathematical symbols from [2, app. C.4.2].

In any implementation of the multigrid method one needs the following operations and data structures:

- a hierarchy of grids that can be nested\(^1\) or not,
- linear solvers for pre-smoothing,
- linear solvers for post-smoothing,
- projections (restriction and prolongation operators, transferring vectors from one grid to the next coarser or finer grid),
- a linear system at the coarsest grid level and an associated (sufficiently accurate) solver.

\(^1\)Element boundaries coincide with element boundaries in finer grids.
Diffpack has basic building blocks for all these ingredients so implementing multigrid is, in principle, a matter of putting these building blocks together.

The typical way of realizing the mentioned data structures and operations is to introduce a vector of grids, a vector of degrees of freedom handlers, a vector of coefficient matrices, a vector of solutions and right-hand sides, a vector projections, a vector of smoothers (linear solvers) and so on. This is accomplished by data structures like VecSimplest(Handle(LinEqSolver)). As you will need the same data structures from problem to problem, it is an idea to collect them in a multigrid toolbox, which in Diffpack is called MGtools. The MGtools class also contains functions for initializing and operating on these data structures. As an application programmer, all you have to do is to include an MGtools object in your class and add about 10 lines with calls to MGtools functions! With the Diffpack menu system you have full flexibility in choosing the type of smoothers, the number of pre- and post-smoothing iterations, the type of multigrid algorithm, the type of cycling, the coarse-grid solver and so on. Through examples and exercises in this report we shall demonstrate this great flexibility and how you can use it to learn about the behavior of multigrid in the application at hand.

2 Sketch of How Multilevel Methods are Implemented in Diffpack

Multigrid methods in Diffpack are implemented as subclasses of a multi-level base class MLSolver, see Figure 1. Various multigrid strategies are then realized as subclasses of Multigrid. A family of domain decomposition methods constitute the other branch of the class diagram for multi-level methods in Diffpack.

Multilevel solvers can act as both basic iterative solvers and as preconditioners. This means that the MLSolver hierarchy should be accessible from Diffpack’s linear solver hierarchy (LinEqSolver) and from the preconditioner hierarchy (Precond) as well. The glue between MLSolver and Diffpack’s linear solvers and preconditioners are the classes MLIter and PrecML. MLIter is a subclass of BasicIterSolver, the base class for all iterative solvers (and a subclass of LinEqSolver), and represent a general basic iterative solver based on multilevel techniques. PrecML is a subclass of Precond and represents a general preconditioner based on multilevel strategies. Both MLIter and PrecML have an MLSolver base-class pointer for accessing various run-
time chosen multigrid methods in the MLSolver hierarchy.

Since multilevel solvers need access to problem-dependent information in the user’s application code, the MLSolver class needs to access some virtual functions that the application code implements. The strategy for implementing such a feature is the standard one in Diffpack: the application code is derived from a base class with user-dependent code, here MLSolverUDC. In class MLSolver we then have a pointer to an MLSolverUDC object through which we can call appropriate information provided by the user. There are three central virtual functions in MLSolverUDC that the application code must implement:

- **solveSubSystem** for implementing a smoother (used for pre- and post-smoothing on an arbitrary grid level)
- **transfer** for implementing projections (i.e. restrictions and prolongations)
- **residual** for implementing the evaluation of the residual on an arbitrary grid level

Since these three functions are approximately the same for a wide range of applications, we have provided suitable default versions of the functions in class MGtools. This means that an application code does not need to be
derived from MLSolverUDC or to provide solveSubSystem, transfer, and residual if one applies the simplified multigrid interface in MGtools. This is what we aim at in the present report.

3 Extending an Existing Application with Multi-grid Methods

Let us choose one of the simplest finite element solvers from [2], class Poisson1, and equip it with a multigrid solver. The multigrid version of Poisson1 is called Poisson1MG. The following steps are required.

1. Make a new directory Poisson1MG with this command:

   ```bash
   mkdir -md Poisson1MG
   ```

2. Move to the Poisson1MG directory, take a copy of the Poisson1 files and change the filenames to Poisson1MG:

   ```bash
   mvname Poisson1 Poisson1MG Poisson1.*
   ```

   Also substitute Poisson1 by Poisson1MG in all files:

   ```bash
   perl -pi.old"' -e 's/Poisson1/Poisson1MG/g;' *.h *.cpp
   ```

3. Include the MGtools.h file in the header file Poisson1MG.h:

   ```c
   #include <MGtools.h>
   ```

4. At the beginning of the declaration of the Poisson1MG class, the following (partly new) data structures must be included as shown here:

   ```c
   Vec(real) linsol; // solution of linear system
   int no_of_grids;
   Handle(MGtools) mgtools;
   ```

   The linsol data structure was present in the original Poisson1 solver so it is a matter of moving the declaration to the top (to avoid problems when Poisson1MG cleans up all the data structures; linsol is attached to other structures and must exist until these are deleted from memory).
5. The fillEssBC function must now take a parameter reflecting the current grid level:

    virtual void fillEssBC (SpaceId space); // set boundary conditions

6. The Poisson1.cpp file needs to include

    #include <PrecoM.h>
    #include <MLIter.h>

7. In Poisson1MG::define you add the MGtools menu to the complete menu for your application, e.g., at the end of the function:

    MGtools::defineStatic(menu, level+1);

8. The MGtools utility has menu items for reading the grids (and specification of the hierarchy of grids) such that the gridfile item on the original Poisson1 menu can (and should be) removed. Recall to remove it in Poisson1MG::scan too, together with the associated readOrMakeGrid call. Instead, a tool in MGtools will compute all the grids we need.

9. In Poisson1MG::scan, after having declared a reference menu to the MenySystem object, you need to perform the following steps.

   (a) Start with binding the lineq handle to a new LinEqAdmFE object and then scan its menu items:

       lineq->scan (menu);          // this call is also in Poisson1

   (b) Create the MGtools object:

       mgtools.rebind(new MGtools(*this));
       mgtools->attach(lineq);
       mgtools->scan(menu);

   (c) If the user (on the menu) has chosen a basic solver to be of multilevel type, we need to insert the multilevel solver in mgtools as solver in the LinEqAdmFE object:

       if (lineq->getSolver().description().contains("multilevel"))
         {
           MLIter& ml = CAST_REF(lineq->getSolver(),MLIter);
           ml.attach(mgtools->getMLSolver());
         }  

---

2 Make sure that you move the lineq statements to the top of scan; a frequent error is to get two lineq initializations in scan, where the latter one destroys the multigrid initializations.
and we apply the same strategy to the preconditioner as well; if this is chosen to be of multilevel type, we insert \texttt{mgtools}' multi-grid solver as preconditioner in the \texttt{LinEqAdmFE} object:

```cpp
if (lineq->getPrec().getDescription().contains("multilevel")) {
    PrecML & ml = CAST_REF(lineq->getPrec(),PrecML);
    ml.attach(mgtools->getMLSolver());
}
```

(d) Make the grid hierarchy:
```
mgtools->initStructure();
```

(e) Connect the \texttt{GridFE} and \texttt{DegFreeFE} handles to the corresponding data structures in \texttt{MGtools} at the finest grid level:
```
no_of_grids = mgtools->getNoOfSpaces();
grid.rebind (mgtools->getGrid (no_of_grids));
dof.rebind (mgtools->getDof (no_of_grids));
```

(f) Perform grid manipulations on all grids (such manipulations are not present in the \texttt{Poisson1} code, but they appear in class \texttt{Poisson2}, the \texttt{Elasticity1} solver and other Diffpack simulators), see class \texttt{Poisson2\textunderscore MG::scan} for an example.

(g) The rest of \texttt{Poisson1\textunderscore MG::scan} proceeds as \texttt{Poisson1::scan}. If desired, one can print an overview of the generated grids, e.g., at the end of \texttt{scan},
```
mgtools->print (a_o, 1 /* level 1 */);
```

(Higher numbers for the level prints also the linear systems etc. at each grid level.)

10. The \texttt{Poisson1\textunderscore MG::fillEssBC} now takes a parameter reflecting the current grid level and the function is therefore slightly different from \texttt{Poisson1::fillEssBC}. The main difference is that we use the grid level parameter to extract the grid and the degree of freedom handler at the current grid level. Thereafter we proceed as in \texttt{Poisson1::fillEssBC} using these grid and degree of freedom handler objects. The grid level in multigrid is represented by the argument \texttt{SpaceId space}.

The reader might wonder why the argument is not \texttt{int grid\_level}. The reason for using the term \texttt{space} instead of grid level is that we then achieve a unified treatment of multigrid and domain decomposition methods. Here is the \texttt{fillEssBC} function:

\footnotetext[4]{See the section on preparing menu system input for how you can control the creation of the various grids in the multigrid method.}
\footnotetext[5]{\texttt{SpaceId} is actually an \texttt{int}.}
\footnotetext[6]{If desired, the reader can translate \texttt{SpaceId space} into \texttt{int grid\_level} when reading about Diffpack multigrid, at least mentally.}
void Poisson1MG::fillEssBC (SpaceId space) 
{
    // extract references to the grid at the current level and to
    // the degree of freedom handler at the current level:
    GridFE& current_grid = mgtools->getGrid(space);
    DegFreeFE& current_dof = mgtools->getDof(space);

    // the rest of the functions is similar to Poisson1::fillEssBC,
    // except that we now use current_grid and current_dof instead
    // of grid and dof
    current_dof.initEssBC ();
    const int nno = current_grid.getNoNodes();
    Prv(real) x;
    for (int i = 1; i <= nno; i++) {
        // is node i subjected to any boundary indicator?
        if (current_grid.boxNode (i)) {
            x = current_grid.getCoord (i); // extract coord. of node i
            current_dof.fillEssBC (i, g(x)); // u=g on the boundary
        }
    }
}

11. In Poisson1MG::solveProblem we must replace the makeSystem call in Poisson1::solveProblem by a call to our MGtools object for making the systems at all grid levels:

    mgtools->makeSystemM1();

The mgtools->makeSystemM1() function applies the FEM* pointer in class MGtools for calling up the functions integrands and fillEssBC (and integrands4side if it exists) in the application code when running the element-by-element assembly loop on all grid levels.

The rest of the solveProblem is not affected by multigrid methods. In particular, the call lineq->solve() use multigrid or any other Diffpack solver according to the user’s choice on the menu!

We are now done with all the modifications of the original Poisson1 code. You will see that very few additional complications occur when we move on to more advanced applications, so the amount of work involved in equipping any Diffpack simulator with multigrid is well described in the previously listed points.

The main.cpp file does not need to be modified as long as the previously run substitution command has replaced Poisson1 by Poisson1MG.

One modification is of fundamental importance: Diffpack must know that the new Poisson1MG code applies an add-on module (the original Poisson1
code applied core Diffpack libraries only). The safest way of ensuring correct linking of a Diffpack application that applies add-on modules like multilevel solvers or adaptivity is to create a new application directory by

    Mkdir -md Poisson1MG

You must then move your Poisson1MG.h, Poisson1MG.cpp, and main.cpp files to the Poisson1MG directory. The -md option to Mkdir specifies compilation and linkage against add-on modules.

The FAQ [3] (see index “add-on modules”) explains how the Makefile detects that an application is to be linked to an add-on module.

4 Setting up an Input File

4.1 Running a Plain Gaussian Elimination

The gridfile item on the original Poisson1 menu is now replaced by grid-handling items on the MGtools menu. Therefore, it may be instructive to first run a plain Gaussian elimination to see how the new multigrid framework affects the input file. An input file test0.i in Poisson1MG/Verify is meant to be a counterpart to Poisson1/Verify/test1.i and just generates 5 × 5 bilinear elements on the unit square:

    set A parameters = 2 1
    ! default LinEqAdmsFE is GaussElim on MatBand
    sub MGtools
        sub GridCollector
            set gridfile = F=PreproBox | d=2 [0,1]x[0,1] |
            d=2 e=ElmB4n20 [5,5] [1,1]
            ok
            ok
        end
    end

The results of this execution should of course be similar to running Poisson1 with its Verify/test1.i input.

Notice that the grid specification, either running a preprocessor or loading a gridfile, is always done on the GridCollector submenu of MGtools when your solver is extended with multigrid.

10
4.2 Filling Out MGtools Menu Items

Let us now create a simple input file for solving the test problem in Poisson1MG by multigrid as linear solver (without preconditioning).

Using multigrid as a basic solver is enabled by choosing basic method as MLIter on the LinEqSolver.prm menu and setting the preconditioner to PrecNone (no preconditioning):

```bash
sub LinEqAdmFE
  sub Matrixprm
    set matrix type = MatSparse
    ok
  sub LinEqSolver_prm
    set basic method = MLIter
    set max iterations = 300
    ok
  'plain multigrid, no preconditioning:
    sub Precond_prm
      set preconditioning type = PrecNone
      ok
    ok
```

The setting of multigrid parameters is done on the MGtools menu, which has various submenus:

- **GridCollector** for parameters related to the hierarchy of grids
- **MLSolver.prm** for choosing the multigrid type (classname in Multigrid hierarchy and V vs. W cycle)
- **smoother LinEqSolver.prm** for setting the linear solver that acts as pre- and post-smoother in the multigrid algorithm
- **coarse grid LinEqSolver.prm** for choosing a coarse-grid solver
- **Proj.prm** for choosing the type of projection (interpolation, restriction)
- **FieldCollector** as a companion to GridCollector for mixed finite element methods

Let us now specify a simple multigrid algorithm:

- V-cycle with three SSOR iterations as smoother,
- SSOR as coarse-grid solver, and
four grid levels with $2 \times 2$ bilinear elements as the coarse grid and a $2 \times 2$ subdivision of elements from one level to the next.

The corresponding `MGtools` menu might look like this:

```plaintext
sub MGtools
  sub GridCollector
    set no of grid levels = 4
    set refinement = [2,2]  ! subdivide each elm into 2x2
    set gridfile = P=PreproBox | d=2 [0,1]x[0,1] |
                         d=2 e=ElmB4n2D { [2,2] } [1,1]
  ok
sub MLSolver prm
  set multilevel method = Multigrid
  set cycle type gamma = 1  ! V cycle
  set nested cycles = 1
  ok
set sweeps = [1,1]  ! 1 pre- and 1 post-smoothing sweep (V1,1 cycle)
sub smoother LinEqSolver prm
  set smoother basic method = SSOR
  ! no of iterations governed by
  ok
set coarse grid solver = false  ! => GaussElim on the coarse grid
ok
```

Refering to the notation in [2, app. C.4], the `sweeps` item contains $[\nu_q, \mu_q]$, `cycle type gamma` is $\gamma_q$, and $K = 4$.

You can find the complete input file for this example in `Verify/test1.i`.

Switching to an iterative coarse-grid solver is easy, just turn the `coarse grid solver` menu item on and fill out a submenu on the `MGtools` menu that we did not cover in the previous example:

```plaintext
set coarse grid solver = true  ! iterative solver
sub coarse grid LinEqSolver prm
  set coarse grid basic method = SSOR
  set coarse grid max iterations = 20
  ok
```

(Notice that when `coarse grid solver` is false, the `default` Gaussian elimination solver is chosen. This means the `factLU` function in the chosen matrix format class. If Gaussian elimination with pivoting is required, one should set `coarse grid solver` to true and fill in the exact specification of the `GaussElim` procedure on the `coarse grid LinEqSolver prm` menu.)
5 Playing Around with Multigrid

The following sections will discuss various sides of multigrid by suggesting a number of exercises/examples that the reader can play around with. First we use standard multigrid as a basic iterative method. Then we investigate other variants of multigrid and use multigrid as preconditioner for Krylov methods.

5.1 Number of Grids and Iterations

Even with this simple PoissonMG simulator we can do several interesting experiments with multigrid. To get some feeling for different components of the algorithm, we encourage you to do some tests on your own. Playing around with parameters will be useful, especially if you want to apply this techniques to more advanced problems.

The number of iterations. Take a multigrid V-cycle with an exact coarse grid solver, one pre- and one post-smoothing step, use an absolute residual termination criterion for some arbitrarily small tolerance, and let the coarse grid consist of $2 \times 2$ bilinear elements on the unit square. That is, use the Verify/test1.i input file as starting point. A central question is how the number of iterations depends on the number of grids or the number of unknowns? The sensitivity to the number of grids can easily be investigated by changing the no of grid levels item to a multiple answer, either in the file (do it in a copy of Verify/test1.i)

```
set no of grid levels = { 2 & 3 & 4 & 5 & 6 & 7 }
```
or directly on the command line

```
--no_of_grid_levels '{ 2 & 3 & 4 & 5 & 6 & 7 }'
```

The number of iterations in multigrid is written by the PoissonMG code to the screen. Here is a sample from an experiment on Unix:

```
unix> ./app --iscl --Default Verify/test1.i \
   --no_of_grid_levels '{ 2 & 3 & 4 & 5 & 6 & 7 }' > tmp.1 
unix> grep 'solver converged' tmp.1  
  solver converged in 4 iterations  
  solver converged in 6 iterations  
  solver converged in 7 iterations  
  solver converged in 7 iterations  
  solver converged in 7 iterations
```

It appears that the number of iterations is fairly constant. The total number of unknowns in these six experiments are 16, 64, 256, 1024, 4096, and 16384. Trying grid level 8 and 9 as well shows that the number of iterations is up to 8 there.

The sensitivity of the number of iterations to the number of unknowns is best investigated by keeping the number of grid levels constant and changing the coarse grid:

```
unix> ./app --iscl --Default Verify/test1.i \
   --no_of_grid_levels 5 --gridfile \
   'P=PreproBox | d=2 [0,1]x[0,1] |  
   d=2 e=ElmB4n2D {[2,2] & [4,4] & [8,8]} [1,1]' > tmp.1 
unix> grep 'solver converged' tmp.1  
  solver converged in 7 iterations  
  solver converged in 7 iterations  
  solver converged in 7 iterations
```

The number of iterations appear to be independent of the number of unknowns (which is 1024, 4096, and 16384 in these experiments).

A bounded number of iterations (for a fixed tolerance), independent of the number of unknowns \( n \), means a bounded number of operations per unknown: The operations per iteration sum up to some constant times the number of unknowns. This means that we are solving the equation system for \( n \) unknowns in \( O(n) \) operations, which is optimal. For a general application it is always the ultimate goal to construct a multigrid algorithm which only needs a bounded number of iterations. For many applications it can be proven that such algorithms exist. However there are applications where an optimal algorithm is not known.
Experiments in 3D. The code in the Poisson1MG works for 3D problems as well (cf. [2, ch. 3.2]). We can redo the previous experiments to see if the number of iterations $n$ is bounded, i.e. independent of the number of unknowns, also in 3D. Since $n$ grows faster (with respect to the number of levels) in 3D than in 2D, we only try 2, 3, and 4 refinements on the unit cube. The relevant lines in Verify/test1.i that needs to be updated are three items on the GridCollector submenu:

```
set no of grid levels =  { 2 & 3 & 4 & 5 }
set refinement =  [2,2,2]  ! subdivide each elm into 2x2x2
set gridfile = P=PreproBox | d=3 [0,1]x[0,1]x[0,1] |
              d=3 e=ElmB8n3D [2,2,2] [1,1,1]
```

These modifications are incorporated in the Verify/test2.i file. Running the Poisson1MG with the test2.i input shows that the number of iterations seems to be no worse than constant (but higher than in the 2D experiment). The multigrid method is in general an $O(\log n)$ operation algorithm independent of the number of dimensions.

5.2 Smoother

Effect of Different Smoothers. In the previous examples (test1.i and test2.i) we used the SSOR method as smoother. What is the effect of other choices, like SOR and Jacobi? This is quickly investigated by editing Verify/test1.i a bit.\(^6\)

```
set no of grid levels = 6
set smoother basic method = { SSOR & SOR & Jacobi }
```

(The resulting file is Verify/test3.i.) The critical result parameters to be investigated are the number of iterations and the CPU time of the solver. Especially the latter is a relevant measure of the relative performance of the smoothers. The CPU time of the linear solver is written to the screen if you run the simulator with the command-line option --verbose 1:

```
unix> app --verbose 1 < Verify/test3.i > tmp.1
```

Again you will need to grep on the tmp.1 to extract the relevant information:

\(^6\)It would be convenient to just take test1.i as input and give the smoother method on the command-line. However, the command-line option --itscheme is ambiguous. We are therefore forced to use file modifications.
The performance of Jacobi, Gauss-Seidel, SOR, and SSOR iteration deteriorates with increasing number of unknowns in the linear system when these methods are used as stand-alone solvers. In connection with multigrid, this is not longer true. But there are of course significant differences between the efficiency of various smoothing procedures in a multigrid context.

**Influence of the Relaxation Parameter.** Our choice of SOR in the previous test actually means the Gauss-Seidel method, because the relaxation parameter in SOR is 1 by default. Of the same reason we used SSOR with a unit relaxation parameter. One should notice that the optimal relaxation parameter for SOR and SSOR as smoothers differs from the optimal value when using SOR and SSOR as stand-alone iterative solvers. Rather under- than over-relaxation is appropriate. It is trivial to test this too:

```plaintext
set smoother basic method = { SSOR & SOR }
set smoother relaxation parameter =
    { 0.8 & 1.0 & 1.2 & 1.4 & 1.6 & 1.8 }
```

(*test4.i contains these modifications.*) Now two menu items are varied. To see the menu combination in run number 5, just look at SIMULATION.m5.ml. From the *.ml we realize that the relaxation parameter is fixed while changing between SSOR and SOR (or in other words, the smoother has the fastest variation). A relaxation parameter around unity seems appropriate.

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

**The Number of Smoothing Steps.** The number of smoothing steps is another interesting parameter to investigate:  

```plaintext
```

How many smoothing steps are optimal? We can simply run the application with *test1.i* as input and use command-line arguments for the number of sweeps (and for increasing the CPU time by setting the number of grid levels to 7):

```bash
./app --iscl --Default Verify/test1.i --verbose 1 
--no_of_grid_levels 7 
```
The number of iterations decrease slightly with the number of sweeps, but recall that the work in each iteration increases with the number of sweeps. We have included the --verbose 1 option such that we can see the CPU time of the total multigrid solver. The CPU times point out that one sweep is optimal in this case.

Another open question is whether the number of pre- and post-smooth operations should be equal. Let us experiment with pre-smoothing only, post-smoothing only, and combined smoothing. We can either modify a copy of the test1.i

```plaintext
set sweeps = { [1,0] & [2,0] & [0,1] & [0,2] & [1,1] & [2,2] }
```

or use the command-line option:

```plaintext
--sweeps '{ [1,0] & [2,0] & [0,1] & [0,2] & [1,1] & [2,2] }'
```

With seven grid levels, two post-smoothenings or one pre- and post-smoothing turned out to be the best choices.

If you have a self-adjoint operator and want to construct a symmetric multigrid preconditioner (for a Conjugated-Gradient solver), you will have to use an equal number of pre- and post-smoothenings to obtain a symmetric preconditioner.

### 5.3 W Cycle and Nested Iteration

![Multigrid W-cycle](image)

**Figure 3:** Multigrid W-cycle

**Different Multigrid Cycles.** We define the multigrid W-cycle and other cycles introducing a cycle parameter, often denoted by $\gamma$ [2]. The value $\gamma = 1$
gives a V-cycle, whereas $\gamma = 2$ gives a W-cycle (see Figure 3 for an example
on a W-cycle). The menu item cycle type gamma is used to set $\gamma$. Use
test1.i as input and override $\gamma$ on the command line$^7$:

```
\renewcommand\{\baselineskip}\{0.85\}\footnotesize
\begin{verbatim}
unix> ./app --iscl --Default Verify/test1.i --verbose 1
    --no_of_grid_levels 7 --gamma \{ 1 \& 2 \& 3 \} > tmp.1
```

The $\gamma$ parameter increases the complexity of the algorithm (the recursive
calls of the multigrid routine). If you encounter convergence problems in an
application, you can try a W-cycle multigrid or even $\gamma > 2$. Higher $\gamma$
values are usually used for more complicated grids or equations. For the current
Poisson equation, a straightforward V-cycle is optimal.

```
\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig4.png}
\caption{Nested iteration, multigrid V-cycle}
\end{figure}
```

**Nested Iteration.** Nested iteration, or full multigrid, or cascading iteration is based on the idea that a coarse grid solution may serve as a good
start guess for a fine grid iteration. The solver on the fine grid is a multigrid

$^7$This is only possible as long as there are no --gamma command-line option from the
simulator’s own menu (or the Diffpack libraries for that sake). Adjusting the menu item
in a file is always safe.
cycle. The coarse grid solution has been itself obtained by a multigrid iteration. However, this solution does not have to be that accurate. It is not obvious that the transport of the start solution is the same as a multigrid prolongation between the grids.

We now want to run `NestedMultigrid`, which can be specified as the answer to the `multilevel method` menu item or the corresponding `--ml_method` command-line option. There is a parameter nested cycles (command-line option `--nestedcycles`) that controls the number of multigrid cycles before the solution is passed to the next finer grid as a start solution. We can include some values of this parameter:

```
./app --iscl --Default Verify/test1.i --verbose 1 \
   --nestedcycles '{ 1 & 2 & 3 }' --no_of_grid_levels 7 \ 
   --ml_method NestedMultigrid
```

A slight increase of the CPU time is seen as the `nested cycle` parameter increases.

It appears that the efficiency of nested multigrid is better than standard multigrid (with the zero vector as startvector) in this test case. Let us compare the two approaches directly and include a run-time plot of how the residual in the linear system decreases with the iteration number (the run time plot item on the `Define ConvMonitor #1` menu or the `--runtime_plot` command-line option):

```
./app --iscl --Default Verify/test1.i --verbose 1 \
   --no_of_grid_levels 7 --runtime_plot ON \ 
   --ml_method '{ Multigrid & NestedMultigrid }'
```

The run-time plot of the residual's evolution during the multigrid algorithm is shown briefly on the screen, but the plots are also available on some standard Diffpack curveplot files:\n
```
tmp.LinEqConvMonitorData.SIMULATION_m01.map
tmp.LinEqConvMonitorData.SIMULATION_m02.map
```

Each of the mapfiles contain only one curve so we can plot both curves by, e.g.,
```
curveplot gnuplot \ 
   -f tmp.LinEqConvMonitorData.SIMULATION_m01.map \ 
   -f tmp.LinEqConvMonitorData.SIMULATION_m02.map \ 
   -r '*,*,*,*' -ps r_nested_vs_std.ps
```

\*Note that the filenames contain the string "tmp", which means that the files will be automatically removed by the `Clean` script [2].
Figure 5 shows the resulting plot, where we clearly see that nested multigrid results in faster decrease in the norm of the absolute residual. (When preparing the plot in Figure 5, we edited the Gnuplot command file `gnuplot.commands` (produced by `curveplot` [2, app. B.4.1]) to improve the labels, and then we loaded this file into Gnuplot to produce a new plot.)

![Log10(norm of residual) vs. no of iterations](image)

Figure 5: Nested iteration vs. standard multigrid.

### 5.4 Coarse-Grid Solver

**The Accuracy of the Coarse-Grid Solver.** On page 12 we outlined how easy it is to switch from Gaussian elimination as coarse-grid solver to an iterative method. Let us investigate SOR, SSOR, and Conjugate-Gradients as coarse-grid solvers and how many iterations that are optimal (i.e., how accurate the solution need to be on the coarse grid). Input file `test5.i` has the relevant new menu items:

```plaintext
sub MGtools
... sub GridCollector ...
set no of grid levels = 5
set gridfile = P=PreproBox | d=2 [0,1]x[0,1] | d=2 e=ElmB4n2D [8,8] [1,1]
ok ...
```

20
set coarse grid solver = true
sub coarse grid LinEqSolver_prm
  set coarse grid basic method = { SOR & SSOR & ConjGrad }
  ! default relaxation parameter is 1.0
  set coarse grid max iterations = { 1 & 5 & 10 & 40 }
ok
...

Since we are using an iterative solver, we need a coarse grid with some
unknowns, at least more than 9. That is why we have specified $8 \times 8$ bilinear
elements for the coarsest grid. From the CPU-time values it appears that 10-
40 iterations have the best computational efficiency. That is, the coarse-grid
solver needs to be accurate, which is a general result also from analysis of
multigrid methods. These experiments show that the Conjugate-Gradient
method is more efficient than SSOR, which is more efficient than SOR,
when then maximum number of coarse-grid iterations is small. The different
solution methods results in approximately the same overall CPU time when
the optimal (a large) number of coarse-grid iterations is specified.

The Optimal Coarse Grid and Number of Levels. We now face
the important question of how to choose the coarse-grid partition and the
number of grid levels. A large coarse grid improves convergence and reduces
the number of iterations, but is expensive to solve. Let us try the following
combinations of grid levels and coarse-grid partitions, designed such that
the number of unknowns on the finest grid is constant (16384):

<table>
<thead>
<tr>
<th>levels</th>
<th>coarse-grid partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>[4, 4]</td>
</tr>
<tr>
<td>5</td>
<td>[8, 8]</td>
</tr>
<tr>
<td>4</td>
<td>[16, 16]</td>
</tr>
<tr>
<td>3</td>
<td>[32, 32]</td>
</tr>
</tbody>
</table>

These modifications have been incorporated in test6a.i to test6d.i (which
are essentially minor edits of test5.i – a more efficient and convenient of
handling the input file is to use to embed one input file in a Perl script as
explained in [2, ch. 3.10.5]). As coarse-grid solver we use SSOR with unit re-
laxation parameter, i.e., symmetric Gauss-Seidel iteration, and a maximum
number of iterations of 400. The number of iterations is constant at 7, but
the CPU-time increases dramatically as we go from a $16 \times 16$ to a $32 \times 32$
coarse grid. A coarse coarse grid with many levels seems to be an optimal
combination.
5.5 Multigrid as a preconditioner

We now want to use multigrid as a preconditioner [2, app. C] instead of a stand-alone iterative solver. We choose the Conjugate-Gradient algorithm to solve the Poisson equation. This algorithm requires a symmetric preconditioner, which in a multigrid context means that the pre- and post-smoothing operators as well as the restriction and prolongation operators must be adjoint. Several methods are available:

- One way to match the condition is to take a self-adjoint smoother like Jacobi iteration or symmetric Gauss-Seidel iteration (i.e. SSOR with unit relaxation parameter).
- Alternatively, use an unsymmetric smoother as a pre-smoother and its adjoint as a post-smoother. For example, take a Gauss-Seidel iteration (SOR) or a (R)ILU iteration with a node ordering \(1, 2, \ldots, n\) as pre-smoother and the same method with a reversed node ordering \(n, n - 1, \ldots, 1\) as post-smoother.
- One alternative is also to use an additive multigrid (see section 5.6) with a self-adjoint smoother like Jacobi iteration or symmetric Gauss-Seidel iteration.

A necessary requirement is that the number of pre- and post-smoothing must be equal.

Using multigrid as a preconditioner is pretty much the same as using it as method, except that applying the preconditioner means a single multigrid iteration (e.g. one V-cycle). Technically in Diffpack, the preconditioner PrecML must have a MLSolver, or in the case of multigrid a Multigrid, attached. PrecML just passes the the vectors \(x\) and \(b\) to Multigrid which does the computation. These steps were taken care of in the example of the simulator's scan function as we explained in Section 3.

**Specifying Multigrid as Preconditioner.** On the input file we basically change

```plaintext
sub LinEqAdmFE

sub LinEqSolver_prm
set basic method = ConjGrad
set max iterations = 300
```

22
Figure 6: Components of multigrid as a preconditioner

```plaintext
ok
sub Precond_prm
  set preconditioning type = PrecML
ok
...
```

The input file `test7.i` runs a test with both multigrid and Conjugate-Gradients as basic solver and multigrid and the identity matrix as preconditioners. The parameters in the multigrid method (whether used as a basic iteration scheme or a preconditioner) are set on the `MGtools` menu.

**Investigating the Effect of Different Smoothers.** Let us test the performance of different smoothers when multigrid is used as preconditioner. The `test8.i` file specifies Conjugate-Gradients as basic solver, multigrid V-cycle as preconditioner, and the following parameters for the smoother and the number of smoothing sweeps:

```plaintext
set sweeps = { [1,1] & [0,1] }
sub smoother LinEqSolver_prm
  set smoother basic method = { Jacobi & SSOR & SOR }
```

The results are striking: [0,1] as sweeps leads to divergence of the solver. This is in accordance with the requirement of an equal number of pre- and post-smoothing steps when multigrid is used as preconditioner, as we have already mentioned. The SOR smoother with one pre- and post-smoothing sweep converges well, despite being nonsymmetric, but applies twice the CPU time and twice the number of Conjugate-Gradient iterations compared with the symmetric smoothers (Jacobi and SSOR).
Nonsymmetric Smoothers and Conjugate Gradient-like Algorithms.

If we apply a Conjugate Gradient-like method for nonsymmetric linear systems, there is no requirement of a symmetric preconditioner, and we can play around with a wider range of smoothing strategies. The input file `test9.i` launches experiments with four basic solvers, BiCGStab, CGS, GMRES, and Conjugate-Gradients, combined with a highly nonsymmetric smoother: two Gauss-Seidel sweeps as pre-smoothing and no post-smoothing.

```plaintext
sub LinEqSolver_prm
  set basic method = { BiCGStab & GMRES & CGS & ConjGrad }
...
sub ConvMonitorList_prm
  sub Define ConvMonitor #1
  ...
    set #1: run time plot = ON
  ...  
  set sweeps = [2,0]
sub smoother LinEqSolver_prm
  set smoother basic method = SOR
```

As expected, the symmetric Conjugate-Gradient solver diverges in combination with the nonsymmetric smoother, but the other algorithms behave well. Both BiCGStab and CGS with two pre-smoothing Gauss-Seidel sweeps appear to be as efficient as Conjugate Gradients with a symmetric multigrid preconditioner (cf. the `test8.i` test). The divergence of Conjugate Gradients with a nonsymmetric preconditioner is visualized in the convergence plot in Figure 7.

## 5.6 Additive Preconditioner

Additive multigrid refers to a strategy where the corrections on the different grid levels are run independently. Originally, this method was proposed in [1], and now it is often referred to as the BPX preconditioner or “multilevel diagonal scaling” (MDS). The method played an important role for the proof of optimal complexity of multigrid, and the interpretation as additive multigrid was found later. The advantage of additive multigrid is that independent operations may serve as a source of parallelism, although the grid transfer operations, the restrictions and prolongations still are serial operations. These steps can be broken up into independent operations by splitting the computational domain into subdomains.

**Standard vs. Nested vs. Additive Multigrid.** We can start testing the additive multigrid method by running some experiments:
Figure 7: The divergence of a Conjugate-Gradient method with a multigrid V-cycle, combined with a nonsymmetric smoother (two Gauss-Seidel pre-smoothing sweeps).

```
sub LinEqSolver_prm
  set basic method = ConjGrad
  ...
sub Precond_prm
  set preconditioning type = PrecNL
  ...
sub MLSolver_prm
  set multilevel method = { AddMultigrid & Multigrid & NestedMultigrid }
  set cycle type gamma = 1    ! V cycle
  set nested cycles = 1
  ok
  set sweeps = [1,1]
sub smoother LinEqSolver_prm
  set smoother basic method = SSOR
```

Notice that additive multigrid is achieved by selecting the `AddMultigrid` multilevel method.

6 Equipping the Poisson2 Solver with Multigrid

The code in the `Poisson2` solver [2, ch. 3.5] is much larger and more complicated than the one in class `Poisson1`, but the modifications needed to incorporate multigrid are the same. The only difference is that the `Poisson2` solver performs some grid manipulations (changing boundary indicators for
instance), which now needs to be implemented for each grid in the grid hierarchy.

We copy the Poisson2 files and rename Poisson2 to Poisson2MG. The various multigrid modifications explained for the Poisson1 code are performed as explained previously. The Poisson2::scan function performs some grid manipulations that must be repeated for each grid:

```cpp
for (int i =1; i<= no_of_grids; i++)
{
    grid.rebind(mgtools->getGrid(i));
    String redef = menu.get ("redefine boundary indicators");
    if (!redef.contains("NONE")) grid->redefineBoInds (redef);
    String addbn = menu.get ("add boundary nodes");
    if (!addbn.contains("NONE")) grid->addBoIndNodes (addbn, ON);
    addbn = menu.get ("remove boundary nodes");
    if (!addbn.contains("NONE")) grid->removeBoIndNodes (addbn, OFF);
    String addmat = menu.get ("add material");
    if (!addmat.contains("NONE")) grid->addMaterial (addmat);
}
```

Notice that grid is a handle to the finest grid, as it should be, at the end of this loop.

Repeating the Poisson1MG Tests. We can run the same test case as for the Poisson1MG code, i.e., we choose to use the Poisnesum subclass which implements the same boundary value problem, with analytical solution, as in class PoissonMG. However, simply running

```
./app --class Poisnesum --iscl --Default Verify/test1.i
```

with Poisson1MG’s test1.i file gives results that differ from those generated by the Poisson1MG solver. The reason is due to the handling of the boundary conditions. In the Poisson2 solver, there are five different boundary indicators that influence the setting of boundary conditions, and we simply apply the default set of four indicators, one for each side. This is incorrect, as all sides should be subject to indicator no. 3 (a function specifying Dirichlet conditions). We must therefore add the following instruction on the test1.i file:

```plaintext
set redefine boundary indicators = n=3
    names= dummy1 dummy2 boundary i=(1) 2=() 3=(1 2 3 4)
```

Now Poisson2MG::scan gets the correct boundary indicator information and propagates it to all intermediate grids.
Looking at \texttt{Poi2sinesum::scan} we see that there is a comment block explaining how to carry out manipulations of the boundary indicators directly in the code. This manipulation adjusts the finest grid only, in a multigrid context. We must ensure that mapping of boundary indicators is applied to all grids, cf. \texttt{Poisson2MG::scan}. Some of the subclass solvers in \texttt{Poisson2} and \texttt{Poisson2MG} manipulate boundary indicators by forcing special menu answers. This will in general result in correct extraction of grid information in \texttt{Poisson2MG::scan} and correct propagation to the hierarchy of grids.

The nice thing about \texttt{Poisson2MG} is the automatic generation of reports. We can tailor this report to empirical investigation of the multigrid method.

\section*{References}


27