```
                              d=2 e=ElmB4n2D { [2,2] } [1,1]
 ok
 sub MLSolver_prm
  set multilevel method = Multigrid
  set cycle type gamma = 1   ! V cycle
 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 the number of sweeps
 ok
 set coarse grid solver = false ! => SSOR on the coarse grid
ok
```

Referring to the notation in [10, App. C.4], the `sweeps` item equals $[\nu_q, \mu_q]$, where q is the grid level, `cycle type gamma` is γ_q, and the `no of grid levels` is $K = 4$. You can find the complete input file for this example in `Verify/test1.i`.

If `coarse grid solver` is set to false, the `smoother` menu items is used to initialize the coarse grid solver. In this example it will therefore be SSOR. Switching to an alternative coarse-grid solver is easy, just set 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:

```
 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 true and nothing else is specified, 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.

3.5 Playing Around with Multigrid

Even with this simple `Poisson1MG` 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 multigrid in more advanced problems.

The following sections will explore various aspects 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 a preconditioner for Conjugate Gradient-like (Krylov) methods.

3.5.1 Number of Grids and Number of Iterations

The number of iterations. Take a multigrid V-cycle with an exact coarse grid solver, one pre- and one post-smoothing step, use a relative residual termination criterion for some arbitrarily prescribed tolerance, and let the coarse grid consist of 2×2 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 & 8 & 9}
```

or directly on the command line[5]

```
--no_of_grid_levels '{ 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 }'
```

The number of iterations in multigrid is written by the `Poisson1MG` code to the screen. Here is a sample from an experiment on a Unix machine:

```
unix> ./app --iscl --Default Verify/test1.i \
      --no_of_grid_levels '{ 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 }' > tmp.1
unix> grep 'solver converged' tmp.1
  solver converged in   5 iterations
  solver converged in   6 iterations
  solver converged in   6 iterations
  solver converged in   6 iterations
  solver converged in   6 iterations
  solver converged in   6 iterations
  solver converged in   6 iterations
  solver converged in   6 iterations
```

The total number of unknowns in these eight experiments are 25, 81, 289, 1098, 4225, 16641, 66049, and 263169 and it appears that the number of iterations is constant. The V-cycle has a convergence rate, γ, independent of h. Hence, the number of iterations, k, to achieve convergence can be determined from the relation, $\frac{\epsilon_k}{\epsilon_0} \leq \gamma^k$, where $\epsilon_k = \|\mathbf{e}_k\| = \|\mathbf{x} - \mathbf{x}_k\|$ and $\|\cdot\|$ is, e.g., the l_2-norm, $\mathbf{x}$ is the exact numerical solution and $\mathbf{x}_k$ is the approximated solution after k iterations. Since, $\frac{\epsilon_k}{\epsilon_{k-1}}$ is fixed, the number of iterations $k = \frac{\log \frac{\epsilon_k}{\epsilon_0}}{\log \gamma}$ should be fixed. However, we do not measure the error in this experiment. The relative residual $\frac{\|\mathbf{r}_k\|}{\|\mathbf{r}_0\|}$ is used as a convergence criterion. Therefore we cannot conclude that multigrid has a convergence rate independent of h, but it is still a good indication. A more rigorous test is done in Section 3.6.2, where `MGtools` is extended with some debugging functionality. Nevertheless, we continue our discussion about the number or iterations.

A bounded number of iterations (for a fixed tolerance), independent of the number of unknowns n, means a bounded number of operations per unknown:

[5] A list of all command line options is generated by the command `DpMenu__HTML`

The operations per iteration sum up to some constant times the number of unknowns. This means that we are solving the equation system for n unknowns in $\mathcal{O}(n)$ operations, which is (order) optimal.

This should not be confused with the optimality of the Full Multigrid method, considered in Section 3.5.4. Full Multigrid reaches the discretization error in $\mathcal{O}(n)$ iterations. The discretization error, E_h, for elliptic problems is on the form $E_h \leq Kh^2$, $h = \frac{1}{n^{1/d}}$, where K is a constant independent of h, and d is the number of space dimensions. Hence, if the numerical error should be comparable with the discretization error we can not use a fixed tolerance, but need to choose $\epsilon_k \leq K\frac{1}{n^{2/d}}$ and determine k. A straightforward calculation shows that,

$$k \geq \frac{\log\left(\frac{K}{\epsilon_0}\frac{1}{n^{2/d}}\right)}{\log\gamma} = \frac{\log n^{2/d}}{\log 1/\gamma} - \frac{\log\frac{\epsilon_0}{K}}{\log 1/\gamma}\,.$$

Therefore, with a stopping criterion suitable for practical purposes the multigrid method requires the work of $\mathcal{O}(n\log n)$ operations.

3.5.2 Convergence Monitors

In the previous experiment we saw that the number of iterations was constant even though we changed the number of unknowns (grid levels). The reason was that multigrid reduces the error by a constant rate, independent of the grid size. So how can we choose a convergence criterion that guarantees that the numerical solution is approximated with the same accuracy as the discretization error? Several criteria can be used. In the previous experiment we simply used the $\|\mathbf{r}_k\|/\|\mathbf{r}_0\|$. This is implemented as `CMRelResidual`. We can estimate the error in the energy norm by measuring the residual. The residual equation reads,

$$\mathbf{A}\mathbf{e}_k = \mathbf{r}_k\,. \tag{3.1}$$

From this equation we derive,

$$(\mathbf{A}\mathbf{e}_k, \mathbf{e}_k) = (\mathbf{r}_k, \mathbf{e}_k) = (\mathbf{r}_k, \mathbf{A}^{-1}\mathbf{r}_k) \leq \|\mathbf{A}^{-1}\|\|\mathbf{r}_k\|^2\,.$$

The multigrid method is a fix point iteration, hence $\|\mathbf{x}_k - \mathbf{x}_{k-1}\|$ can be used. We have,

$$\|\mathbf{x}_k - \mathbf{x}_{k-1}\| \leq \|\mathbf{x}_k - \mathbf{x}\| + \|\mathbf{x} - \mathbf{x}_{k-1}\| \leq \gamma^{k-1}(\gamma+1)\|\mathbf{x} - \mathbf{x}_0\|\,. \tag{3.2}$$

This criterion is implemented as `CMAbsSeqSolution`. One can also measure the convergence factor, ρ, in the energy norm,

$$(\mathbf{A}\mathbf{e}_k, \mathbf{e}_k) \leq \rho^k(\mathbf{A}\mathbf{e}_0, \mathbf{e}_0), \tag{3.3}$$

where $\mathbf{A}$ is the matrix to be solved in `Poisson1`. $\mathbf{A}$ is a symmetric positive definite (SPD) matrix. However, the error $\mathbf{e} = \mathbf{x} - \mathbf{x}_k$ is obviously unavailable in general and therefore unusable as convergence criterion. Instead we can

use the residual equation (3.1), to derive a suitable criterion for the residual. First of all we consider some general properties. Multigrid is a linear operator, meaning that it can be represented as a matrix. However, it is more efficiently implemented as an algorithm, which produces an output vector $\mathbf{v}_k$, given an input vector $\mathbf{r}_k$. We therefore introduce the algorithm as an operator $\mathbf{B}$, and $\mathbf{v}_k = \mathbf{B}\mathbf{r}_k$ makes sense. It is known that $\mathbf{B}$ is spectrally equivalent with $\mathbf{A}^{-1}$, and spectral equivalence of $\mathbf{A}^{-1}$ and $\mathbf{B}$ is defined by,

$$c_0(\mathbf{A}^{-1}\mathbf{x}, \mathbf{x}) \leq (\mathbf{B}\mathbf{x}, \mathbf{x}) \leq c_1(\mathbf{A}^{-1}\mathbf{x}, \mathbf{x}), \quad \forall \mathbf{x}, \tag{3.4}$$

or

$$c_0(\mathbf{A}\mathbf{x}, \mathbf{x}) \leq (\mathbf{A}\mathbf{B}\mathbf{A}\mathbf{x}, \mathbf{x}) \leq c_1(\mathbf{A}\mathbf{x}, \mathbf{x}), \quad \forall \mathbf{x}, \tag{3.5}$$

From (3.1), (3.3) and (3.5) we derive,

$$(\mathbf{B}\mathbf{r}_k, \mathbf{r}_k) = (\mathbf{B}\mathbf{A}\mathbf{e}_k, \mathbf{A}\mathbf{e}_k) = (\mathbf{A}\mathbf{B}\mathbf{A}\mathbf{e}_k, \mathbf{e}_k) \leq c_1(\mathbf{A}\mathbf{e}_k, \mathbf{e}_k)\,. \tag{3.6}$$

The term $(\mathbf{B}\mathbf{r}_k, \mathbf{r}_k)$ is already computed by the preconditioned Conjugate-Gradient method and is available at no cost. It is implemented in the convergence monitor `CMRelMixResidual`. The term `Rel` or `Abs` refers to the fact that the criterion is relative or absolute, respectively. These convergence monitors and several others are implemented in Diffpack, look up the man page (`dpman ConvMonitor`). We usually use a relative convergence criterion when we test the efficiency of multigrid, since the criterion will then be independent of the grid size.

Experiments in 3D. The code in the `Poisson1MG` works for 3D problems as well (cf. [10]). 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, 4, and 5 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 $\mathcal{O}(n \log n)$ operation algorithm in any number of space dimensions.

3.5.3 Smoother

Effect of Different Smoothers. In the previous examples (input files `test1.i` and `test2.i`) we used the SSOR method as smoother. What is the effect of

other choices, like SOR and Jacobi iterations? This is 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` file to extract the relevant information:

```
unix> egrep 'solver converged|solver_classname' tmp.1
```

The performance of Jacobi, Gauss-Seidel, SOR, and SSOR iterations 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 no longer true, but there are of course significant differences between the efficiency of various smoothing procedures in a multigrid context and in particular the relaxation parameter is very important. Choosing the "wrong" relaxation parameter may lead to poor performance, as we will see below.

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. For 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. In this case under-relaxation rather than over-relaxation is appropriate. It is trivial to test this too:

```
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` files 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.

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

```
set sweeps = { [1,1] & [2,2] & [3,3] & [4,4] }
```

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):

```
./app --iscl --Default Verify/test1.i --verbose 1 \
      --no_of_grid_levels 7 \
      --sweeps '{ [1,1] & [2,2] & [3,3] & [4,4] }' > tmp.1
```

The number of iterations decreases 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`

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

or use the command-line option:

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

With seven grid levels, two post-smoothings or one pre- and post-smoothing turned out to be the best choices. This is a little bit strange. The transfer operators (standard L_2 projection) are not perfect, high frequency errors are restricted to low frequency errors and may therefore pollute the coarse grid correction. It is therefore very important that the pre-smoother removes all high frequency error before transferring. These projection effects are usually called *aliasing*. A funny example of aliasing is apparent in old western movies, where the wheels seem to go backwards. This is simply a result of too coarse sampling rate of a high frequency phenomena. In general, because of aliasing, we need pre- smoothers (at least in theory). However, the above numerical experiment indicated that multigrid might very well work with only post-smoothings.

If you have a self-adjoint operator and want to construct a *symmetric* multigrid *preconditioner* (for a Conjugate-Gradient solver), you will have to use an equal number of pre- and post-smoothings and the pre-smoother and post-smoother should be adjoint, to obtain a symmetric preconditioner.

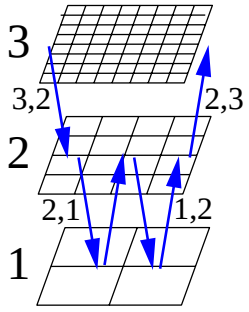

Fig. 3.1. Multigrid W-cycle

3.5.4 W Cycle and Nested Iteration

Different Multigrid Cycles. We specify the multigrid W-cycle and other cycles by introducing a *cycle* parameter, often denoted by γ [10]. The value $\gamma = 1$ gives a V-cycle, whereas $\gamma = 2$ gives a W-cycle (see Figure 3.1 for an example on a W-cycle). The menu item `cycle type gamma` is used to set γ. Use `test1.i` as input and override γ on the command line[7]:

```
unix> ./app --iscl --Default Verify/test1.i --verbose 1\
            --no_of_grid_levels 7 --gamma '{ 1 & 2 & 3 }' > tmp.1

unix> grep 'solver conv\|LinEqAdm' tmp.1
  LinEqAdm::solve: solve a linear system,  CPU time= 0.24
  solver converged in   6 iterations
  LinEqAdm::solve: solve a linear system,  CPU time= 0.31
  solver converged in   6 iterations
  LinEqAdm::solve: solve a linear system,  CPU time= 0.45
  solver converged in   6 iterations
```

The numerical experiments are done on an AMD Athlon 1800 MHz with 1 GB RAM.

The γ 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 γ values are usually used for more complicated grids or equations. For the current Poisson equation, a straightforward V-cycle is optimal.

Nested Iteration. Nested iteration, or full multigrid, or cascadic iteration[8] is based on the idea that a coarse grid solution may serve as a good start guess

[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.

[8] This is a special case of the nested iteration, where no restriction is used and coarser grids are never revisited.

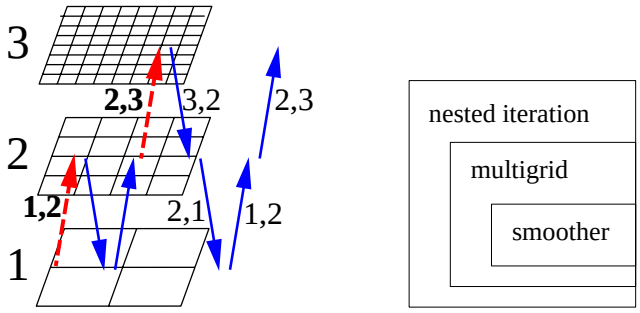

Fig. 3.2. Nested iteration, multigrid V-cycle

for a fine grid iteration. Hence, before the standard V- or W-cycle, a start vector is computed. This process starts by first computing an approximate solution on the coarsest grid. This solution is prolongated onto the next coarsest grid and a smoother is applied on this grid level. This process is repeated until the finest grid is reached, where the standard multigrid process starts. The right hand side on the coarsest grid is in Diffpack implemented as a restriction of the right hand side on the finest grid. It is cheaper to generate the right hand side on the coarsest grid, but this can not be done when the right hand side comes from a residual generated by Krylov solver. We have therefore chosen this implementation such that it is easy to use the nested iteration as a preconditioner for a Krylov solver.

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. In fact, nested multigrid is optimal, it reaches discretization error within $\mathcal{O}(n)$ iterations (see also page 109). 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 residuals evolution during the multigrid algorithm is shown briefly on the screen, but the plots are also available in some standard Diffpack curveplot files[9]:

```
tmp.LinEqConvMonitorData.SIMULATION_m01.map
tmp.LinEqConvMonitorData.SIMULATION_m02.map
```

Each of the mapfiles contains only one curve, but 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
```

Figure 3.3 shows the resulting plot, where we clearly see that nested multigrid results in faster decrease in the absolute norm of the residual. (When preparing the plot in Figure 3.3, we edited the Gnuplot command file, which has the name `.gnuplot.commands` when produced by `curveplot` [10, App. B.5.1], to improve the labels, and then we loaded this file into Gnuplot to produce a new plot.)

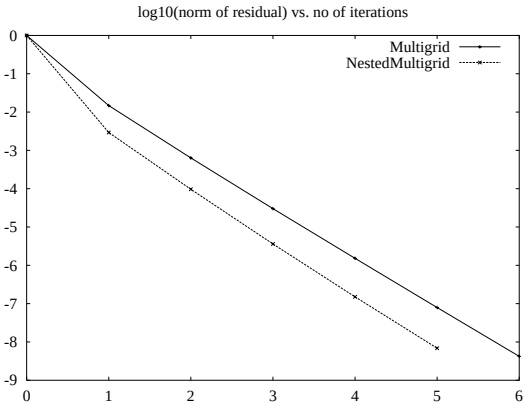

Fig. 3.3. Nested iteration vs. standard multigrid.

[9] Note that the filenames contain the string `"tmp"`, which means that the files will be automatically removed by the `Clean` script [10].

3.5.5 Coarse-Grid Solver

The Accuracy of the Coarse-Grid Solver. On page 107 we outlined how easy it is to switch from using Gaussian elimination as coarse-grid solver to using 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:

```
...
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
 ...
 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. This is why we have specified 8×8 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 also is a general result from the 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 result in approximately the same overall CPU time when the optimal (a large) number of coarse-grid iterations is specified. It is important to notice that the Conjugate-Gradient method is not a linear iteration and does not fit into most theoretical analysis of multilevel methods. However, the numerical experiments done here indicate that it may work anyway.

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. 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):

levels	coarse-grid partition
6	[4,4]
5	[8,8]

```
4               [16,16]
3               [32,32]
```

These modifications have been incorporated in `test6a.i` to `test6d.i` (which are essentially minor edits of `test5.i` – a more efficient and convenient method of handling the input file is to embed one input file in a Perl script as explained in [10, Ch. 3.12.9]). As the coarse-grid solver we use SSOR with unit relaxation parameter, i.e., symmetric Gauss-Seidel iteration, and a maximum number of iterations of 40. The number of iterations is constant at 7, but the CPU-time increases dramatically as we go from a 16×16 to a 32×32 coarse grid. Thus, a very coarse coarse grid with many levels seems to be an optimal combination.

3.5.6 Multigrid as a Preconditioner

We now want to use multigrid as a preconditioner [10, App. C] instead of as a stand-alone iterative solver. We choose the Conjugate-Gradient algorithm to solve the Poisson equation. This algorithm requires a symmetric preconditioner and a symmetric matrix, 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 satisfy 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 non-symmetric 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.
- Another alternative is to use an additive multigrid (see section 3.5.7) 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 steps must be equal.

Using multigrid as a preconditioner is pretty much the same as using it as a stand-alone solver, 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 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 3.3.1.

Specifying Multigrid as Preconditioner. In the input file we basically change

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

Table 3.2. Comparison of Multigrid as a solver and as a preconditioner

	MG		CG/MG	
h	#it.	CPU time	#it.	CPU time
2^{-2}	5	0.01	3	0.01
2^{-3}	5	0.01	4	0.01
2^{-4}	6	0.01	5	0.01
2^{-5}	6	0.01	5	0.02
2^{-6}	6	0.06	5	0.07
2^{-7}	6	0.25	5	0.29
2^{-8}	7	1.22	5	1.19
2^{-9}	7	4.85	5	4.75
2^{-10}	7	23.08	6	31.26

Multigrid as a Preconditioner vs. a Basic Iteration Method. In Table 3.2 we compare multigrid as a solver with multigrid as a preconditioner for the Conjugate Gradient method. The results are striking, there seems to be no point in using Krylov subspace acceleration for a highly efficient multigrid method as we have here (see also [13], Chapter 7.8). In general, multigrid preconditioners are more robust and in real-life applications it is often difficult to choose optimal multigrid parameters. While multigrid may reduce most parts of the error, other parts remain essentially unchanged. These are picked up by the Krylov method (see the Sections 3.6.4 and 3.7).

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 the

preconditioner, and the following parameters for the smoother and the number of smoothing sweeps:

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

The results are striking: using `[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 non-symmetric, but applies twice the CPU time and twice the number of Conjugate-Gradient iterations compared with the symmetric smoothers (Jacobi and SSOR). The typical behavior can be seen in Figure 3.4. Initially, we get nice convergence, but later we get stagnation or even divergence.

Non symmetric Smoothers and Conjugate Gradient-like Algorithms. If we apply a Conjugate Gradient-like method for non-symmetric 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 non-symmetric smoother: two Gauss-Seidel sweeps as pre-smoothing and no post-smoothing.

```
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
```

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). As expected, the symmetric Conjugate-Gradient solver stagnates in combination with the non-symmetric smoother, but the other algorithms behave well.

3.5.7 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 [14], and now it is often referred to as the BPX preconditioner (BPX) or "multilevel diagonal scaling" (MDS). The method played an important

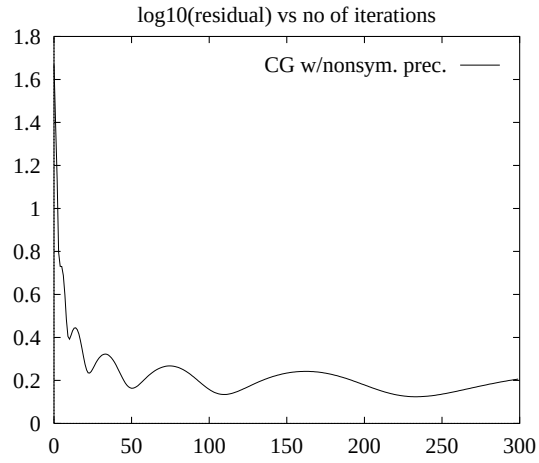

Fig. 3.4. The stagnation of a Conjugate-Gradient method with a multigrid V-cycle, combined with a non-symmetric smoother (two Gauss-Seidel pre-smoothing sweeps).

role for the proof of optimal complexity of multigrid, and the interpretation as additive multigrid was found later. Additive multigrid may diverge even for elliptic problems, but as a preconditioner, it is fairly efficient. 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:

```
sub LinEqSolver_prm
 set basic method = ConjGrad
 ...
sub Precond_prm
 set preconditioning type = PrecML
 ...
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
```