Simulation and High-Performance Computing

Part 9: Multigrid Methods

Steffen Börm

Christian-Albrechts-Universität zu Kiel

October 2nd, 2020

Richardson iteration

Goal: Efficient solvers for very large systems $Ax = b$ resulting, e.g., from finite difference discretizations.

Idea: If A is positive definite, we have

$$0 \leq \langle x - x_m, A(x - x_m) \rangle = \langle x - x_m, b - Ax_m \rangle,$$

i.e., the residual $r_m = b - Ax_m$ "points" roughly in the same direction as the error $x - x_m$.

Richardson iteration

Goal: Efficient solvers for very large systems $Ax = b$ resulting, e.g., from finite difference discretizations.

Idea: If A is positive definite, we have

$$0 \leq \langle x - x_m, A(x - x_m) \rangle = \langle x - x_m, b - Ax_m \rangle,$$

i.e., the residual $r_m = b - Ax_m$ "points" roughly in the same direction as the error $x - x_m$.

Richardson iteration: Add scaled residual to current approximation.

$$x_{m+1} := x_m + \theta(b - Ax_m), \qquad \theta \in \mathbb{R}_{>0}.$$

Similar to gradient method, but far less computational work per step.

Relaxation methods

Idea: Compute subsets of variables to satisfy subsets of equations.
Simplest case: Choose $\tilde{x}_i$ such that the i-th equation holds

$$a_{ii}\tilde{x}_i + \sum_{\substack{j=1\\j\neq i}}^{n} a_{ij}x_j = b_i \qquad \iff \qquad \tilde{x}_i = \frac{1}{a_{ii}}\Bigg(b_i - \sum_{\substack{j=1\\j\neq i}}^{n} a_{ij}x_j\Bigg).$$

Relaxation methods

Idea: Compute subsets of variables to satisfy subsets of equations.
Simplest case: Choose $\tilde{x}_i$ such that the i-th equation holds

$$a_{ii}\tilde{x}_i + \sum_{\substack{j=1\\j\neq i}}^{n} a_{ij}x_j = b_i \qquad \iff \qquad \tilde{x}_i = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1\\j\neq i}}^{n} a_{ij}x_j\right).$$

Jacobi method: Compute all $\tilde{x}_i$ based on the original approximation.

- Parallelizable.
- Requires auxiliary memory to store original approximation.

Relaxation methods

Idea: Compute subsets of variables to satisfy subsets of equations.
Simplest case: Choose $\tilde{x}_i$ such that the i-th equation holds

$$a_{ii}\tilde{x}_i + \sum_{\substack{j=1\\j\neq i}}^{n} a_{ij}x_j = b_i \qquad \iff \qquad \tilde{x}_i = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1\\j\neq i}}^{n} a_{ij}x_j\right).$$

Jacobi method: Compute all $\tilde{x}_i$ based on the original approximation.

- Parallelizable.
- Requires auxiliary memory to store original approximation.

Gauß-Seidel method: Compute $\tilde{x}_i$ based on previous updates.

- Faster convergence.
- No auxiliary memory required.

Implementation: Grid functions

Idea: Treat general boundary conditions by including boundary points.

```
typedef struct {
  /* Number of intervals in x and y direction */
  int nx;
  int ny;

  /* Stepsize */
  real h;

  /* Point values */
  real *v;
} gridfunc;
```

The grid is given by

$$\Omega_h := \{(ih, jh) \ : \ i \in [0 : n_x], \ j \in [0 : n_y]\},$$

the value $u_h(ih, jh)$ can be found in `u->v[i+j*ld]` with `ld`$= n_x + 1$.

Implementation: Discrete Laplacian for grid functions

Idea: Treat $-\Delta_h$ implicitly instead of storing its coefficients.

$$-\Delta_h u_h(x) = \frac{1}{h^2}\Big(4u_h(x) - u_h(x_1 - h, x_2) - u_h(x_1 + h, x_2) \\ - u_h(x_1, x_2 - h) - u(x_1, x_2 + h)\Big)$$

```
diag = 4.0 / h / h;
off = -1.0 / h / h;

for(j=1; j<ny; j++)
  for(i=1; i<nx; i++)
    yv[i+j*ld] += alpha * (diag * xv[i+j*ld]
                          + off * xv[(i-1)+j*ld]
                          + off * xv[(i+1)+j*ld]
                          + off * xv[i+(j-1)*ld]
                          + off * xv[i+(j+1)*ld]);
```

Advantages: Very efficient, no special treatment for boundary values.

Implementation: Richardson for grid functions

Idea: Treat $-\Delta_h$ implicitly instead of storing its coefficients.

```
diag = 4.0 / h / h;
off = -1.0 / h / h;

for(j=1; j<ny; j++)
  for(i=1; i<nx; i++)
    dv[i+j*ld] = bv[i+j*ld] - diag * xv[i+j*ld]
                            - off * xv[(i-1)+j*ld]
                            - off * xv[(i+1)+j*ld]
                            - off * xv[i+(j-1)*ld]
                            - off * xv[i+(j+1)*ld];

for(j=1; j<ny; j++)
  for(i=1; i<nx; i++)
    xv[i+j*ld] += theta * dv[i+j*ld];
```

Advantages: Very efficient, even easily parallelizable.

Implementation: Jacobi for grid functions

Idea: Treat $-\Delta_h$ implicitly instead of storing its coefficients.

```
diag = 4.0 / h / h;
off = -1.0 / h / h;

for(j=1; j<ny; j++)
  for(i=1; i<nx; i++)
    xn[i+j*ld] = (bv[i+j*ld] - off * xv[(i-1)+j*ld]
                             - off * xv[(i+1)+j*ld]
                             - off * xv[i+(j-1)*ld]
                             - off * xv[i+(j+1)*ld]) / diag;

for(j=1; j<ny; j++)
  for(i=1; i<nx; i++)
    xv[i+j*ld] += (1.0-theta) * xv[i+j*ld]
                        + theta * xn[i+j*ld];
```

Advantages: Very efficient, even easily parallelizable.

Implementation: Gauß-Seidel for grid functions

Idea: Treat $-\Delta_h$ implicitly instead of storing its coefficients.

```
diag = 4.0 / h / h;
off = -1.0 / h / h;

for(j=1; j<ny; j++)
  for(i=1; i<nx; i++)
    xv[i+j*ld] = (bv[i+j*ld] - off * xv[(i-1)+j*ld]
                             - off * xv[(i+1)+j*ld]
                             - off * xv[i+(j-1)*ld]
                             - off * xv[i+(j+1)*ld]) / diag;
```

Advantages: Very efficient, no auxiliary memory required.

Implementation: Gauß-Seidel for grid functions

Idea: Treat $-\Delta_h$ implicitly instead of storing its coefficients.

```
diag = 4.0 / h / h;
off = -1.0 / h / h;

for(j=1; j<ny; j++)
  for(i=1; i<nx; i++)
    xv[i+j*ld] = (bv[i+j*ld] - off * xv[(i-1)+j*ld]
                             - off * xv[(i+1)+j*ld]
                             - off * xv[i+(j-1)*ld]
                             - off * xv[i+(j+1)*ld]) / diag;
```

Advantages: Very efficient, no auxiliary memory required.

Checkerboard Gauß-Seidel: First process all points where $i + j$ is even, then all points where $i + j$ is odd. $\rightarrow$ Allows parallelization of both phases.

Experiment: Checkerboard Gauß-Seidel

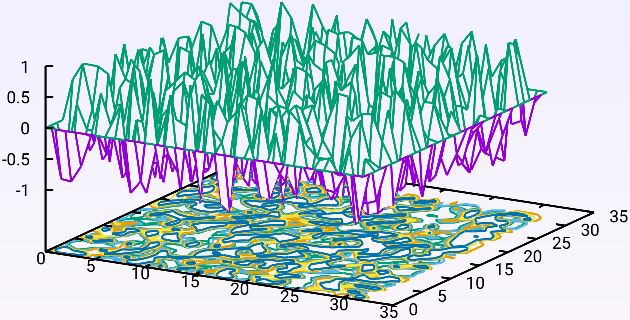
Goal: Approximate solution u_h of $-\Delta_h u_h = f_h$.

	$h = 1/16$		$h = 1/32$		$h = 1/64$	
m	error	ratio	error	ratio	error	ratio
0	1.50_{+1}		3.10_{+1}		6.30_{+1}	
1	1.33_{+1}	1.13	2.93_{+1}	1.06	6.13_{+1}	1.02
2	1.25_{+1}	1.07	2.85_{+1}	1.03	6.05_{+1}	1.01
3	1.18_{+1}	1.06	2.78_{+1}	1.02	5.98_{+1}	1.01
4	1.12_{+1}	$1,05$	2.72_{+1}	1.02	5.92_{+1}	1.01
5	1.07_{+1}	1.05	2.67_{+1}	1.02	5.87_{+1}	1.01
10	8.69_{+0}	1.04	2.47_{+1}	1.01	5.67_{+1}	1.01
20	5.87_{+0}	1.04	2.18_{+1}	1.01	5.38_{+1}	1.005

Observation: Very slow convergence, getting slower as h grows smaller.

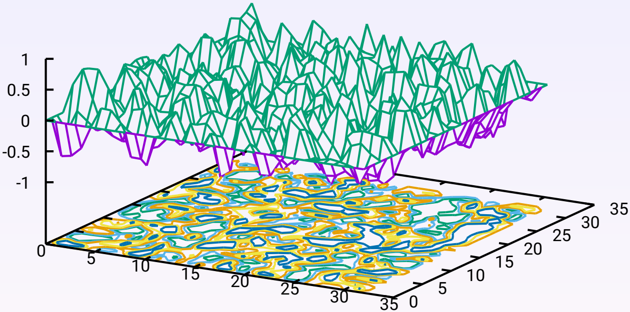
Smoothing

Approach: Take a closer look at the errors obtained by the Jacobi iteration.



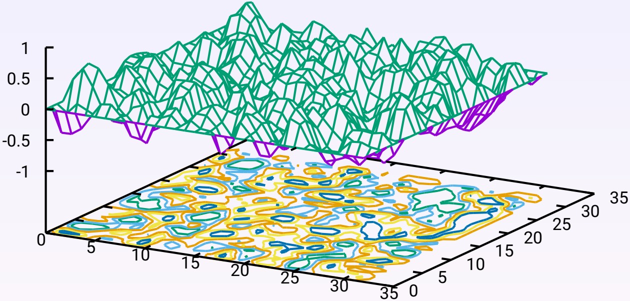
Smoothing

Approach: Take a closer look at the errors obtained by the Jacobi iteration.



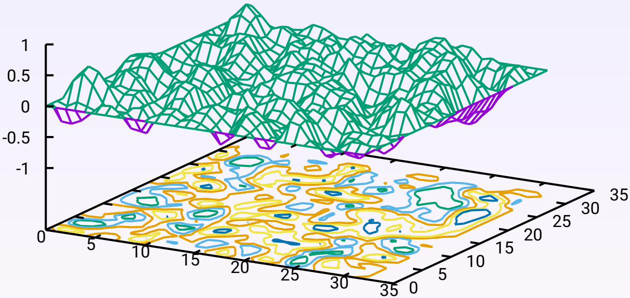
Smoothing

Approach: Take a closer look at the errors obtained by the Jacobi iteration.



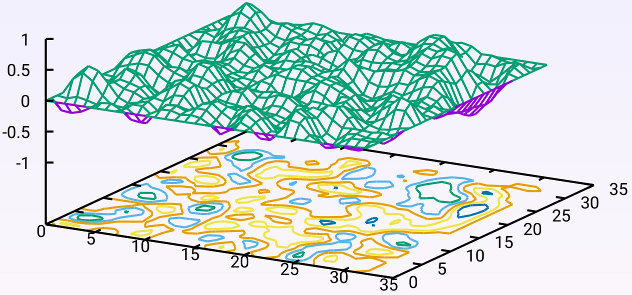
Smoothing

Approach: Take a closer look at the errors obtained by the Jacobi iteration.



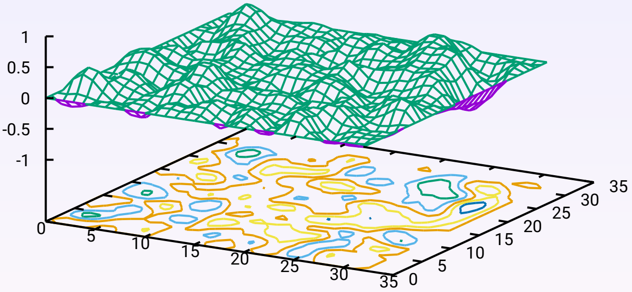
Smoothing

Approach: Take a closer look at the errors obtained by the Jacobi iteration.



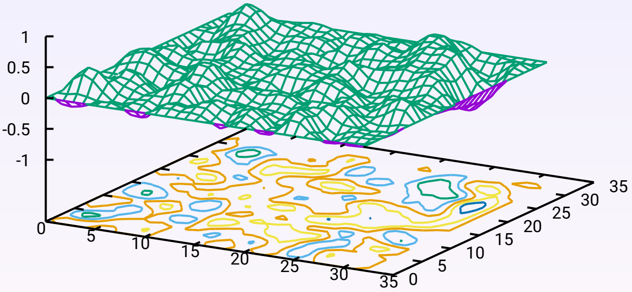
Smoothing

Approach: Take a closer look at the errors obtained by the Jacobi iteration.



Smoothing

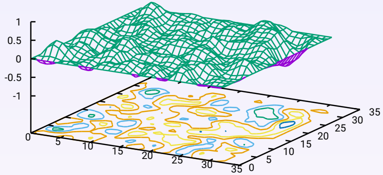
Approach: Take a closer look at the errors obtained by the Jacobi iteration.



Observation: The error decreases slowly, but it becomes smooth.

Coarse grid approximation

Idea: If the error is smooth, we can approximate it on a coarser grid.



Questions: How do we compute this approximation efficiently?
And how to we subtract it from the current approximation on the fine grid?

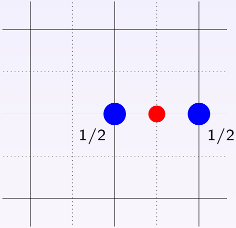
Grid transfer

Goal: Map functions on a coarse grid Ω_H to a fine grid Ω_h.

Grid transfer

Goal: Map functions on a coarse grid Ω_H to a fine grid Ω_h.

Simple approach: Ensure $H = 2h$, use linear interpolation to map from coarse to fine.

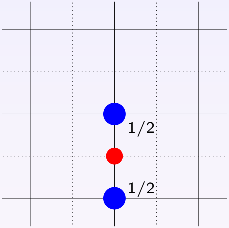


- Interpolate in x direction,

Grid transfer

Goal: Map functions on a coarse grid Ω_H to a fine grid Ω_h.

Simple approach: Ensure $H = 2h$, use linear interpolation to map from coarse to fine.

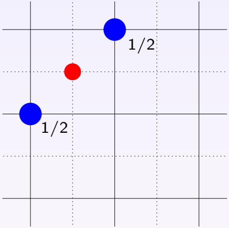


- Interpolate in x direction,
- interpolate in y direction,

Grid transfer

Goal: Map functions on a coarse grid Ω_H to a fine grid Ω_h.

Simple approach: Ensure $H = 2h$, use linear interpolation to map from coarse to fine.

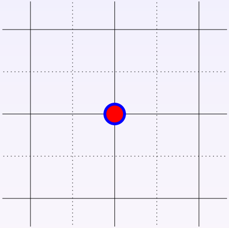


- Interpolate in x direction,
- interpolate in y direction,
- interpolate diagonally, and

Grid transfer

Goal: Map functions on a coarse grid Ω_H to a fine grid Ω_h.

Simple approach: Ensure $H = 2h$, use linear interpolation to map from coarse to fine.

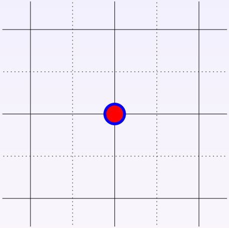


- Interpolate in x direction,
- interpolate in y direction,
- interpolate diagonally, and
- copy coarse grid values.

Grid transfer

Goal: Map functions on a coarse grid Ω_H to a fine grid Ω_h.

Simple approach: Ensure $H = 2h$, use linear interpolation to map from coarse to fine.



- Interpolate in x direction,
- interpolate in y direction,
- interpolate diagonally, and
- copy coarse grid values.

Prolongation: $p \in \mathbb{R}^{\Omega_h \times \Omega_H}$ maps coarse to fine grid functions.

Restriction: $r := \frac{1}{4} p^T$ maps fine to coarse grid functions.

Coarse grid correction

Goal: Approximate the smooth error $x - x_m$ by a coarse-grid function c.

$$x - x_m \approx \quad c,$$

Coarse grid correction

Goal: Approximate the smooth error $x - x_m$ by a coarse-grid function c.

$$x - x_m \approx p\,c,$$

Coarse grid correction

Goal: Approximate the smooth error $x - x_m$ by a coarse-grid function c.

$$x - x_m \approx p\,c,$$
$$-\Delta_h(x - x_m) \approx -\Delta_h(p\,c),$$

Since we do not know x, we apply $-\Delta_h$.

Coarse grid correction

Goal: Approximate the smooth error $x - x_m$ by a coarse-grid function c.

$$\begin{aligned} x - x_m &\approx p\,c, \\ -\Delta_h(x - x_m) &\approx -\Delta_h(p\,c), \\ f_h + \Delta_h x_m &\approx -\Delta_h\, p\,c, \end{aligned}$$

Since we do not know x, we apply $-\Delta_h$.

Coarse grid correction

Goal: Approximate the smooth error $x - x_m$ by a coarse-grid function c.

$$\begin{aligned} x - x_m &\approx p\,c, \\ -\Delta_h(x - x_m) &\approx -\Delta_h(p\,c), \\ f_h + \Delta_h\,x_m &\approx -\Delta_h\,p\,c, \\ r(f_h + \Delta_h\,x_m) &= -r\,\Delta_h\,p\,c \end{aligned}$$

Since we do not know x, we apply $-\Delta_h$. In order to obtain a square matrix, we apply the restriction.

Coarse grid correction

Goal: Approximate the smooth error $x - x_m$ by a coarse-grid function c.

$$\begin{aligned} x - x_m &\approx p\,c, \\ -\Delta_h(x - x_m) &\approx -\Delta_h(p\,c), \\ f_h + \Delta_h\,x_m &\approx -\Delta_h\,p\,c, \\ r(f_h + \Delta_h\,x_m) &= -r\,\Delta_h\,p\,c = -\Delta_H\,c. \end{aligned}$$

Since we do not know x, we apply $-\Delta_h$. In order to obtain a square matrix, we apply the restriction. Galerkin identity $\Delta_H = r\Delta_h p$.

Coarse grid correction

Goal: Approximate the smooth error $x - x_m$ by a coarse-grid function c.

$$\begin{aligned} x - x_m &\approx p\,c, \\ -\Delta_h(x - x_m) &\approx -\Delta_h(p\,c), \\ f_h + \Delta_h\,x_m &\approx -\Delta_h\,p\,c, \\ r(f_h + \Delta_h\,x_m) &= -r\,\Delta_h\,p\,c = -\Delta_H\,c. \end{aligned}$$

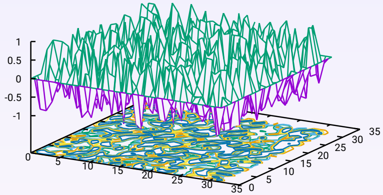
Since we do not know x, we apply $-\Delta_h$. In order to obtain a square matrix, we apply the restriction. Galerkin identity $\Delta_H = r\Delta_h p$.

Algorithm:

1. Compute the residual $d_h \leftarrow f_h + \Delta_h x_m$.
2. Restrict it to the coarse grid $f_H \leftarrow r\,d_h$.
3. Solve the coarse-grid equation $-\Delta_H c \leftarrow f_H$.
4. Update the approximation $x_m \leftarrow x_m + p\,c$.

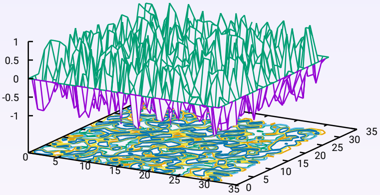
Two-grid iteration

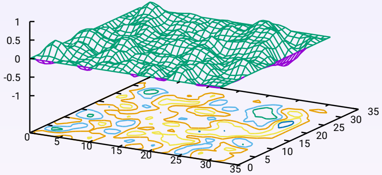
1. Start with an initial guess.



Two-grid iteration

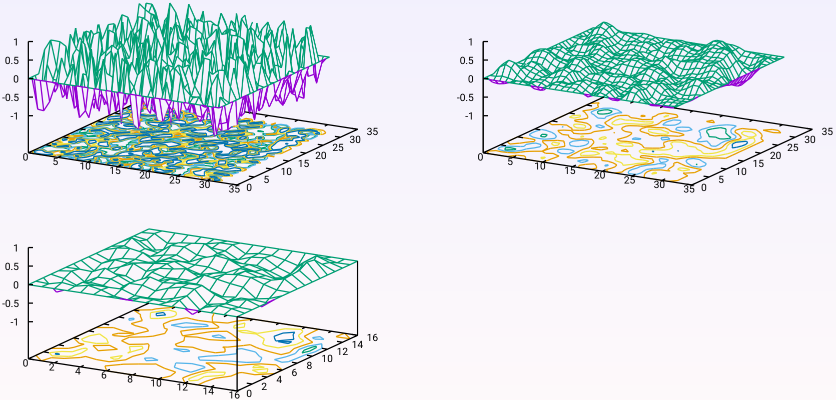
1. Start with an initial guess.
2. Perform smoothing steps with a simple iterative method.





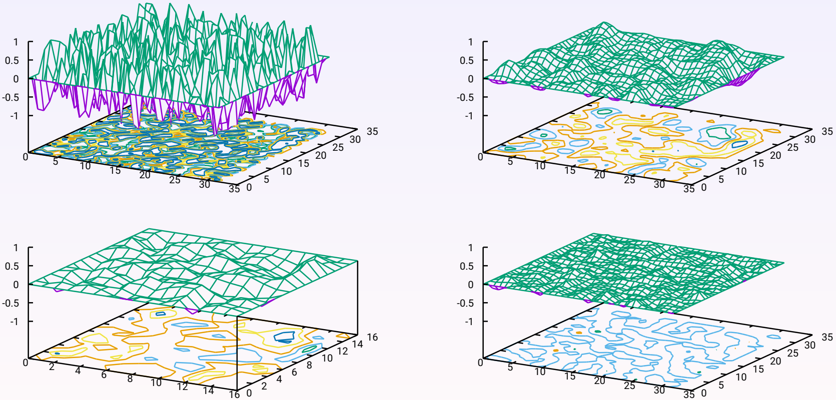
Two-grid iteration

1. Start with an initial guess.
2. Perform smoothing steps with a simple iterative method.
3. Approximate remaining error on a coarse grid.



Two-grid iteration

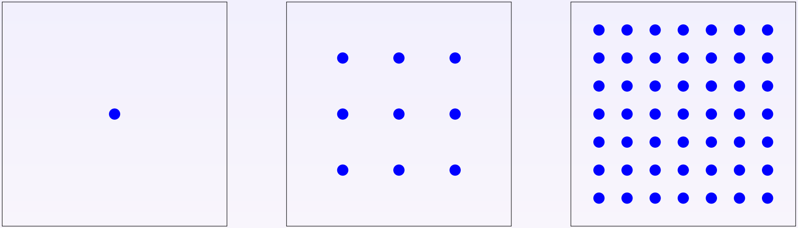
1. Start with an initial guess.
2. Perform smoothing steps with a simple iterative method.
3. Approximate remaining error on a coarse grid.
4. Add coarse-grid correction.



Multigrid iteration

Problem: The coarse grid Ω_H may still be too fine.

Idea: Use an entire hierarchy of grids $\Omega_0 \subseteq \Omega_1 \subseteq \Omega_2 \subseteq \ldots \subseteq \Omega_L = \Omega_h$.



Instead of solving the coarse-grid system exactly, apply smoothing and coarse-grid corrections recursively to approximate the error.

Implementation: Prolongation

Approach: Add the function values of all coarse-grid points to adjacent fine-grid points.

```
for(j=1; j<ny; j++)
  for(i=1; i<nx; i++) {
    c = xv[i+j*ld];
    yv[(2*i  )+(2*j  )*ld] += c;
    yv[(2*i-1)+(2*j  )*ld] += 0.5 * c;
    yv[(2*i+1)+(2*j  )*ld] += 0.5 * c;
    yv[(2*i  )+(2*j-1)*ld] += 0.5 * c;
    yv[(2*i  )+(2*j+1)*ld] += 0.5 * c;
    yv[(2*i-1)+(2*j-1)*ld] += 0.5 * c;
    yv[(2*i+1)+(2*j+1)*ld] += 0.5 * c;
  }
```

Implementation: Restriction

Approach: Accumulate the function values of all fine-grid points to adjacent coarse-grid points.

```
for(j=1; j<ny; j++)
  for(i=1; i<nx; i++) {
    xv[i+j*ld] = 0.25 * (yv[(2*i  )+(2*j  )*ld]
                         + 0.5 * yv[(2*i-1)+(2*j  )*ld]
                         + 0.5 * yv[(2*i+1)+(2*j  )*ld]
                         + 0.5 * yv[(2*i  )+(2*j-1)*ld]
                         + 0.5 * yv[(2*i  )+(2*j+1)*ld]
                         + 0.5 * yv[(2*i-1)+(2*j-1)*ld]
                         + 0.5 * yv[(2*i+1)+(2*j+1)*ld]);
  }
```

Implementation: Multigrid iteration

Assumption: Grid functions in arrays x, b, d.

```
for(l=L; l>0; l--) {
  smoother(b[l], x[l], d[l]);

  copy(b[l], d[l]);
  addlaplace(-1.0, x[l], d[l]);
  restriction(d[l], b[l-1]);

  zero(x[l-1]);
}
solve(b[0], x[0]);
for(l=1; l<=L; l++) {
  prolongation(x[l-1], x[l]);

  smoother(b[l], x[l], d[l]);
}
```

Experiment: Multigrid

Approach: Multigrid with checkerboard Gauß-Seidel smoothing.

	$h = 1/16$		$h = 1/32$		$h = 1/64$		$h = 1/8192$	
m	error	ratio	error	ratio	error	ratio	error	ratio
0	1.50_{+1}		3.10_{+1}		6.30_{+1}		8.19_{+3}	
1	2.52_{+0}	5.94	5.39_{+0}	5.75	1.11_{+1}	5.68	1.46_{+3}	5.62
2	4.49_{-1}	5.62	9.68_{-1}	5.57	1.99_{+0}	5.56	2.62_{+2}	5.57
3	8.09_{-2}	5.55	1.76_{-1}	5.51	3.62_{-1}	5.51	4.64_{+1}	5.52
4	1.47_{-2}	5.52	3.21_{-2}	5.47	6.62_{-2}	5.47	8.65_{+0}	5.48
5	2.67_{-3}	5.49	5.89_{-3}	5.45	1.22_{-2}	5.44	1.59_{+0}	5.45
10	5.46_{-7}	5.46	1.27_{-6}	5.40	2.64_{-6}	5.39	3.44_{-4}	5.38
20	2.35_{-14}	5.45	6.17_{-14}	5.38	1.32_{-13}	5.37	1.73_{-11}	5.36

Observation: Very stable rate of convergence.

Summary

Smoothers like Richardson, Jacobi, or Gauß-Seidel converge very slowly for discretizations with fine grids, but the smoothe the remaining error.

Coarse-grid correction: Approximate the smoothed error using a coarser grid, improve the current approximation.

Multigrid method: Since exact coarse-grid corrections would take too long, replace with approximate corrections obtained by recursively applying smoothing and coarse-grid corrections.

Result: $\sim n$ operations per step, stable rate of convergence for all grids.