Ogmg: A Multigrid Solver for Overture
User Guide, Version 1.00

William D. Henshaw Centre for Applied Scientific Computing

Lawrence Livermore National Laboratory
Livermore, CA, 94551
henshaw@llnl.gov
http://www.llnl.gov/casc/people/henshaw
http://www.llnl.gov/casc/Overture June 10, 2002 UCRL-MA-134446

Abstract: We describe a C++ class, Ogmg, that can be used to solve elliptic boundary value problems on composite overlapping grids with the multigrid algorithm. Ogmg solves problems in two and three space dimensions on overlapping grids. Currently only second order accurate scalar elliptic boundary value problems can be solved.

Contents

1 Introduction **4**

2 The multigrid algorithm for overlapping grids **4**

3 Using Ogmg **4**
3.1 Basic Steps . . . 4
3.2 Convergence criteria . . . 5
3.3 Test routine ogmgt . . . 5

4 Generating an overlapping grid with multigrid levels **6**

5 Black Box Multigrid **8**
5.1 Operator averaging . . . 8
5.1.1 Operator averaging at boundaries . . . 8

6 Numerical Results **9**
6.1 Square . . . 10
6.2 Circle in a channel . . . 11
6.3 Box . . . 13
6.4 Sphere in a box . . . 14

7 Singular problems **14**

8 Ogmg Function Descriptions **16**
8.1 constructor . . . 16
8.2 constructor . . . 16
8.3 setPlotStuff . . . 16
8.4 updateToMatchGrid . . . 16
8.5 getMaximumResidual . . . 16
8.6 getNumberOfIterations . . . 16
8.7 sizeOf . . . 17
8.8 setOgmgParameters . . . 17
8.9 solve . . . 17
8.10 cycle . . . 17
8.11 printStatistics . . . 17
8.12 setCoefficientArray . . . 17
8.13 setOrderOfAccuracy . . . 18
8.14 interpolate . . . 18
8.15 update . . . 18
8.16 update . . . 18
8.17 smooth . . . 18
8.18 smoothJacobi . . . 18
8.19 smoothGaussSeidel . . . 18
8.20 smoothRedBlack . . . 19
8.21 smoothLine . . . 19
8.22 alternatingLineSmooth . . . 19
8.23 fineToCoarse(level) . . . 19
8.24 fineToCoarse(level,grid) . . . 20
8.25 coarseToFine(level) . . . 20
8.26 coarseToFine(level,grid) . . . 20
8.27 defect(level) . . . 20
8.28 defect(level,grid) . . . 20
8.29 getDefect . . . 21
8.30 initializeBoundaryConditions . . . 21
8.31 applyBoundaryConditions(level,...) . . . 21
8.32 applyBoundaryConditions(level,grid,...) . . . 21

8.33 buildExtraLevels 22

1 Introduction

Ogmg is a multigrid solver for use with Overture [1],[2]. Ogmg can solve scalar elliptic problems on overlapping grids. It has a variety of smoothers including Red-Black and line smoothers. A sparse direct or sparse iterative solver such as GMRES can be used to solve the coarse grid equations – the Overture solver `Oges` is used for this purpose.

The system of equations that Ogmg solves are specified as a "coefficient-array" grid function. The coefficient array can be created using the Overture operator classes. The user of Ogmg is responsible for creating these equations at all multigrid levels.

Ogmg does not yet handle the case of singular problems such as a Poisson equation with all Neumann boundary conditions.

2 The multigrid algorithm for overlapping grids

Ogmg uses the standard defect correction algorithm. The implementation on overlapping grids is relatively straight-forward. See the paper [4] for further discussion.

- Typically Jacobi, Red-Black or line (zebra) smoothers are used.
- The fine to coarse *Restriction* operator is the *full weighting* operator except at boundaries.
- The coarse to fine Prolongation operator is second or fourth order interpolation; second-order by default.
- The **cycle** chosen is either adaptive or can be fixed to a desired one.

while *not converged* **do**
  smooth ν_1 times or until the smoothing rate $> \eta$
    $v_1 \leftarrow S^{\nu_1} v_1$
  form the defect and transfer to the coarser grid
    $f_2 \leftarrow R^{1\to 2}(f - Av_1)$
  "solve" the defect equation (at least to an "accuracy" of δ)
    $A_2 v_2 \approx f_2$
  correct the fine grid solution from the coarse grid solution
    $v_1 \leftarrow v_1 + P^{2\to 1} v_2$
  smooth ν_2 times or until the smoothing rate $> \eta$
    $v_1 \leftarrow S^{\nu_2} v_1$
end while

The **smoothing step** represented by the operator S is a *composite-smooth* where each grid in turn is smoothed:

for *each grid g in a CompositeGrid* **do**
  smooth grid g ν_g times
  interpolate
end for

The smoother and the number of smooths may vary from component grid to component grid. We try to choose n_g, the number of smooths on each component grid, so that the residual stays about the same size on each component grid. The approximate rule we use is that

$$n_g \approx \frac{\| \text{ residual on grid } g\|}{\min_g \|\text{residual on grid } g\|}$$

The grid with the smallest residual will have $n_g = 1$.

3 Using Ogmg

3.1 Basic Steps

To use Ogmg to solve a problem the user should take the follwoing steps:

1. Generate an overlapping grid with the grid generator `ogen` that has more than one multigrid level.
2. Use the operator classes to build a coefficient matrix defining the discretization of an elliptic boundary value problem. This must be done at each multigrid level (although with the operator classes this is quite easy).

3. Create an Ogmg object and assign parameters such as the type of smoother.

4. Define the right hand side function for the PDE, including boundary conditions. This need only be done at the finest level.

5. solve the problem.

3.2 Convergence criteria

Ogmg uses two possible convergence criteria. One can either measure the convergence through the maximum norm of the residual, or by an estimate of the error. **Both** criteria must be met for convergence.

The residual convergence criteria is

$$\|\text{residual}\|_\infty < \texttt{residualTolerance} \times \text{number of grid points}$$

where the '`residualTolerance`' is the user specified tolerance. We scale by the number of grid points as an attempt to make this tolerance somewhat indepedent from the number of grid points. Note that for a typical elliptic problem the very best value that can be expected for the residual is proportional to the round-off error, ϵ/h^2, where ϵ is the machine epsilon and h is the grid spacing (so that $1/h^2$ is approximately the number of grid points).

The second converge criteria is based on an error estimate,

$$E^n < \texttt{errorTolerance}$$

where '`errorTolerance`' is a user specified value. The estimate E^n for the error at iteration n is computed as

$$\delta^n = \frac{\|u^{n+1} - u^n\|}{\|u^n - u^{n-1}\|}$$
$$E^n = \frac{\delta}{1-\delta}\|u^n - u^{n-1}\|$$

where we use the maximum norm. This estimate follows assuming the solution is converging at a rate δ,

$$u^n - u^\infty \approx \delta^n v, \qquad (E^n \approx \|u^n - u^\infty\|_\infty)$$

3.3 Test routine ogmgt

The test routine `Overture/Ogmg/ogmgt.C` shows how to call Ogmg to solve Poisson's equation with either Dirichlet or Neumann boundary conditions:

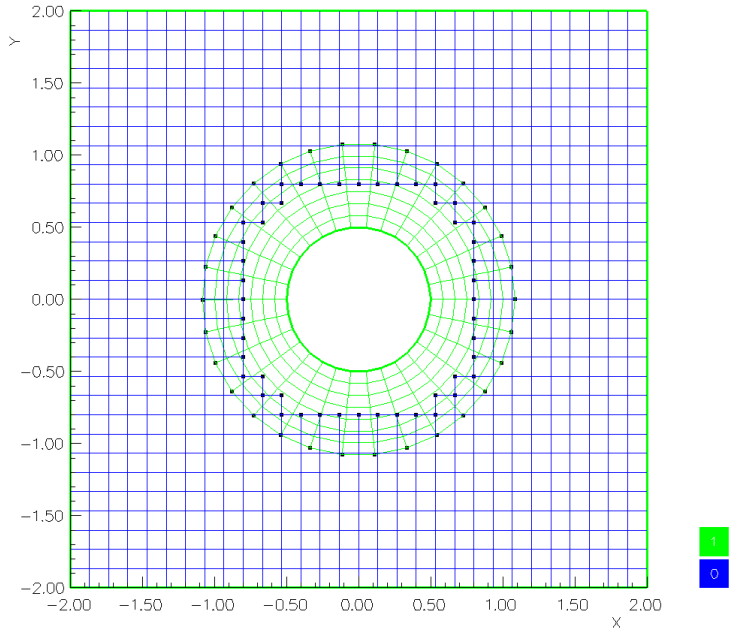
$$\Delta u = f \qquad \text{for } \mathbf{x} \in \Omega$$
$$u = g, \text{ or } u_n = g \qquad \text{for } \mathbf{x} \in \partial\Omega.$$

The forcing functions f and g are chosen so that the true solution is known. We use the Twilight-Zone functions defined in the `OGPolyFunction` and `OGTrigFunction` classes to define the true solution and its derivatives. See the **OtherStuff** documentation [3] for further details on these classes.

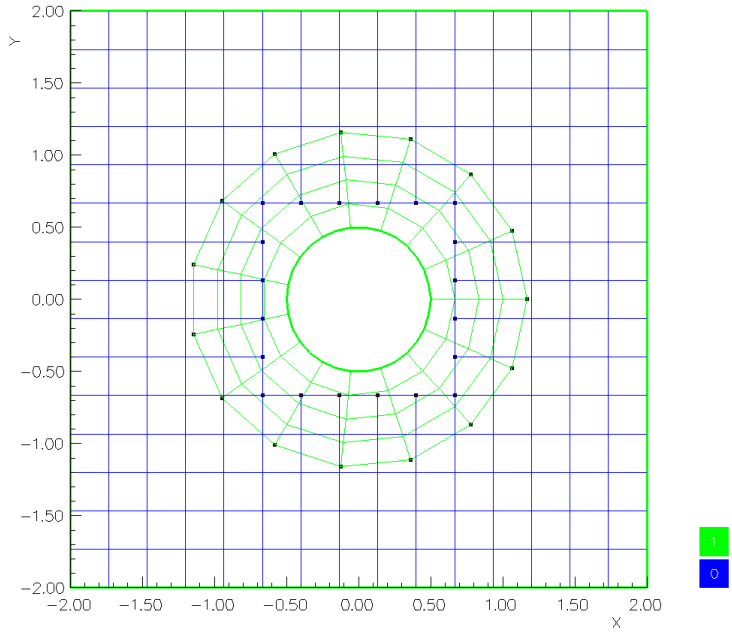
4 Generating an overlapping grid with multigrid levels

The grid generator **Ogen** can be used to build an overlapping grid with some number of multigrid levels.

Here is an example command file for ogen for creating the 'circle in a channel' grid, `cicmg.hdf`. In this example the overlapping grid algorithm fails if we were to ask for any more than 2 multigrid levels.



An overlapping grid for a cylinder in a channel, multigrid level 0.



An overlapping grid for a cylinder in a channel, multigrid level 1.

In figure (1) we show two multigrid levels for a two-dimensional valve.

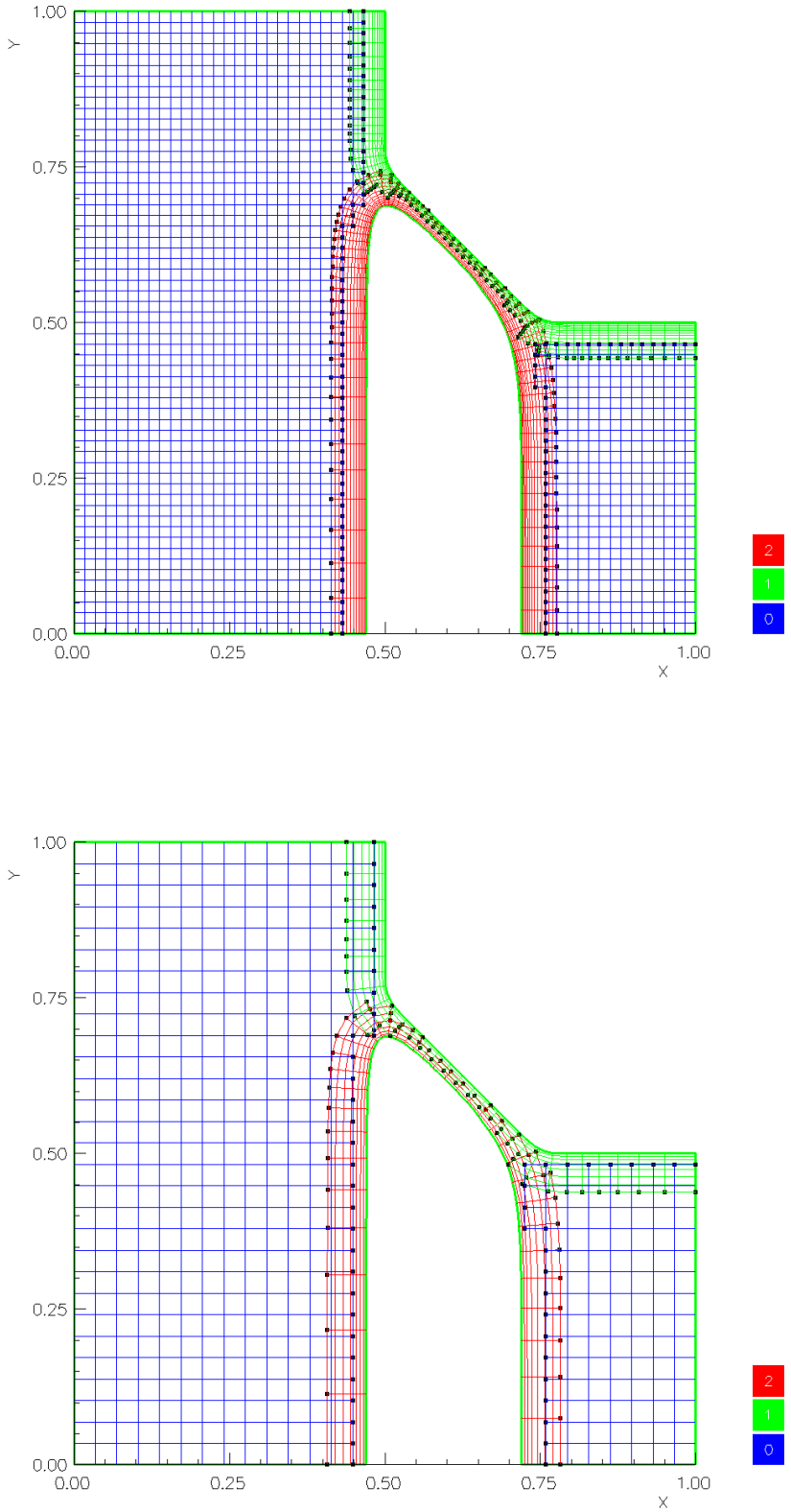


Figure 1: An overlapping grid for a valve, 2 multigrid levels.

5 Black Box Multigrid

A new feature of Ogmg is the ability to take a problem defined on a CompositeGrid with only one level and to automatically generate the information for coarser levels. There are two key ingredients to making this work. The first is that we need to generate the coarse grid coefficient matrices automatically. The second is to handle interpolation points on the coarse grid which may or may not sit on interpolation points on the fine grid.

5.1 Operator averaging

To generate a coarse grid operator from a fine grid operator we can average the operator on the fine grid and then restrict the result to the coarse grid.

Consider a 3 point stencil operator in one dimension. If we look at the stencil for rows $i-1, i, i+1$ arranged in a matrix then we get

$$\begin{array}{ccccc} a_{i-1}u_{i-2} & b_{i-1}u_{i-1} & c_{i-1}u_i & 0 & 0 \\ 0 & a_iu_{i-1} & b_iu_i & c_iu_{i+1} & 0 \\ 0 & 0 & a_{i+1}u_i & b_{i+1}u_{i+1} & c_{i+1}u_{i+2} \end{array}$$

If we replace row i by the weighted average of rows $i-1, i, i+1$ with weights α, β, α then we get the wide stencil

$$\alpha a_{i-1}u_{i-2} \quad (\alpha b_{i-1} + \beta a_i)u_{i-1} \quad (\alpha(c_{i-1} + a_{i+1}) + \beta b_i)u_i \quad (\alpha b_{i+1} + \beta c_i)u_{i-1} \quad \alpha c_{i+1}u_{i+2}$$

If we distribute the values at point $i-1$ using $u_{i-1} = \frac{1}{2}(u_{i-2} + u_i)$ and at point $i+1$ using $u_{i+1} = \frac{1}{2}(u_{i+2} + u_i)$ then we have a wide stencil only defined at points $i-2, i, i+2$.

$$(\alpha(a_{i-1} + \tfrac{1}{2}b_{i-1}) + \tfrac{1}{2}\beta a_i)u_{i-2} \qquad (\alpha(\tfrac{1}{2}b_{i-1} + \tfrac{1}{2}b_{i+1} + c_{i-1} + a_{i+1}) + \beta(b_i + \tfrac{1}{2}a_i + \tfrac{1}{2}c_i))u_i \qquad (\alpha(c_{i+1}\tfrac{1}{2}b_{i+1}) + \tfrac{1}{2}\beta c_i)u_{i+2}$$

Typically we take $\alpha = \gamma = 1/4$, and $\beta = 1/2$. In more than one space dimension we can apply the above averaging procedure sequentially in each direction.

5.1.1 Operator averaging at boundaries

At a boundary we will typically have a boundary condition such as a dirichlet, neumann or mixed boundary condition. We need to decide how to average near the boundary and on the boundary or ghost line.

Ogmg is aware of two types of boundary conditions. These 'boundary conditions' actually just indicate how the **ghost line** should be updated. The condition `extrapolation` indicates the ghost line is extrapolated and usually means that a dirichlet boundary condition is applied on the boundary. The `equation` boundary condition indicates that some equation is applied on the ghost line; this is usually associated with a neumann or mixed boundary condition.

dirichlet : In this case we just impose a dirichlet BC in the coarse grid operator.

neumann : (or neumann like condition) coarse grid operator. The coarse grid operator is obtained by distributing the fine grid ghost line equation to the coarser grid but not averaged in the tangential direction.

$$(\alpha(a_i + \tfrac{1}{2}b_i) + \tfrac{1}{2}\beta a_i)u_{i-2} \qquad (\alpha(b_i + c_i + a_i) + \beta(b_i + \tfrac{1}{2}a_i + \tfrac{1}{2}c_i))u_i \qquad (\alpha(c_i + \tfrac{1}{2}b_i) + \tfrac{1}{2}\beta c_i)u_{i+2}$$

extrapolation : coarse grid operator is also extrapolation.

equation : ghost line has some equation on it. The coarse grid operator is obtained by averaging the ghostline equations (i.e. averaging is only done in the tangential directions).

Remark: The coarse grid matrix A^2 can also be defined from the fine grid matrix A^1 using the prologation and restriction operators

$$A^2 = RA^1P$$

The first step above where the rows were combined corresponds to premultiplying A^1 by R. The second step where the values at points $i+1$ and $i+1$ were removed corresponds to the post-multiplication by P.

6 Numerical Results

In this section we present some numerical results for some two and three dimensional problems.
Notation:

$$\begin{aligned}
\text{WU(i)} &= \text{number of work units for iteration i} \\
\text{res(i)} &= \text{residual for iteration i} \\
\text{rate(i)} &= \text{convergence rate, res(i)/res(i-1)} \\
\text{ECR(i)} &= \text{effective convergence rate} \\
&= \left(\frac{\text{res}(i+1)}{\text{res}(i)}\right)^{1/WU(i)} \\
\text{err(i)} &= \text{maximum error in the solution for iteration i} \\
n_s &= \text{number of smooths per level}
\end{aligned}$$

A work unit is defined to be the amount of work (number of multiplications) required for a single Jacobi iteration. The work units reported here are only reasonable approximations.

The effective convergence rate (ECR) is a normalized convergence rate that takes into account the amount of work required for each multigrid iteration. The ECR is the convergence rate that a Jacobi iteration would have to achieve per iteration in order to be as good as the multigrid algorithm. Recall that the Jacobi iteration has a convergence rate for standard elliptic problems that quickly approaches 1, $ECR = 1 - O(h^2)$ as the mesh size h goes to zero. Optimal SOR is better with $ECR = 1 - O(h)$. Multigrid effective convergence rates are generally in the range $.5$ to $.8$, independent of the mesh size. (The convergence rate for a full cycle is usually about $.1$). Since the convergence rate does not depend on h the method has optimal complexity – the work required to compute the solution to a given accuracy requires a fixed number of iterations, independent of h, and is thus proportional to the number of unknowns.

6.1 Square

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$6.7e-01$	0.039	5.1	0.53
2	$2.7e-02$	0.041	5.1	0.54
3	$1.2e-03$	0.043	5.1	0.54
4	$5.2e-05$	0.044	5.1	0.54
5	$2.4e-06$	0.046	5.1	0.55
6	$1.1e-07$	0.047	5.1	0.55
7	$5.4e-09$	0.048	5.1	0.55
8	$2.6e-10$	0.048	5.1	0.55
9	$1.3e-11$	0.049	5.1	0.56

Red-Black, square16mg
2 levels, $n_s = 3$, Dirichlet

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$1.3e+01$	$4.1e-02$	5.3	0.55
2	$5.6e-01$	$4.2e-02$	5.3	0.55
3	$2.4e-02$	$4.3e-02$	5.3	0.55
4	$1.1e-03$	$4.4e-02$	5.3	0.56
5	$4.9e-05$	$4.5e-02$	5.3	0.56
6	$2.2e-06$	$4.6e-02$	5.3	0.56
7	$1.0e-07$	$4.7e-02$	5.3	0.56
8	$4.9e-09$	$4.7e-02$	5.3	0.56
9	$2.3e-10$	$4.7e-02$	5.3	0.56

Red-black, square64mg.
4 levels, $n_s = 3$, Dirichlet

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$3.8e+01$	$7.0e-02$	5.3	0.61
2	$2.9e+00$	$7.5e-02$	5.3	0.61
3	$2.2e-01$	$7.7e-02$	5.3	0.62
4	$1.7e-02$	$7.7e-02$	5.3	0.62
5	$1.3e-03$	$7.8e-02$	5.3	0.62
6	$1.1e-04$	$7.9e-02$	5.3	0.62
7	$8.5e-06$	$8.0e-02$	5.3	0.62
8	$6.8e-07$	$8.0e-02$	5.3	0.62
9	$5.5e-08$	$8.0e-02$	5.3	0.62

Alternating-Zebra, square64mg.
4 levels, $n_s = 3$, Dirichlet

Table 1: Multigrid convergence rates for a square.

6.2 Circle in a channel

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$3.7e-01$	$4.0e-02$	5.1	0.53
2	$3.1e-02$	$8.3e-02$	5.1	0.61
3	$5.4e-03$	$1.7e-01$	5.1	0.71
4	$1.0e-03$	$1.9e-01$	5.1	0.72
5	$2.1e-04$	$2.0e-01$	5.1	0.73

Red-Black, cicmg.
2 levels, $n_s = 3$, Dirichlet

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$1.7e+00$	$1.1e-01$	5.1	0.65
2	$4.9e-01$	$2.8e-01$	5.1	0.78
3	$1.3e-01$	$2.6e-01$	5.1	0.77
4	$3.6e-02$	$2.8e-01$	5.1	0.78
5	$1.0e-02$	$2.8e-01$	5.1	0.78
6	$2.5e-03$	$2.5e-01$	5.1	0.76
7	$6.6e-04$	$2.6e-01$	5.1	0.77

Alternating-Zebra, cicmg.
2 levels, $n_s = 1$, Dirichlet

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$3.7e+00$	0.018	9.6	0.66
2	$9.0e-02$	0.024	8.3	0.64
3	$9.9e-03$	0.111	7.4	0.74
4	$1.5e-03$	0.148	7.8	0.78
5	$1.7e-04$	0.113	7.9	0.76
6	$1.7e-05$	0.104	7.7	0.75
7	$2.1e-06$	0.124	7.4	0.75

Red-Black, cicmgFine. 112,850 grid points.
4 levels, $n_s = 4$, Dirichlet.

Table 2: Multigrid convergence rates for a circle in a channel, cicmg.

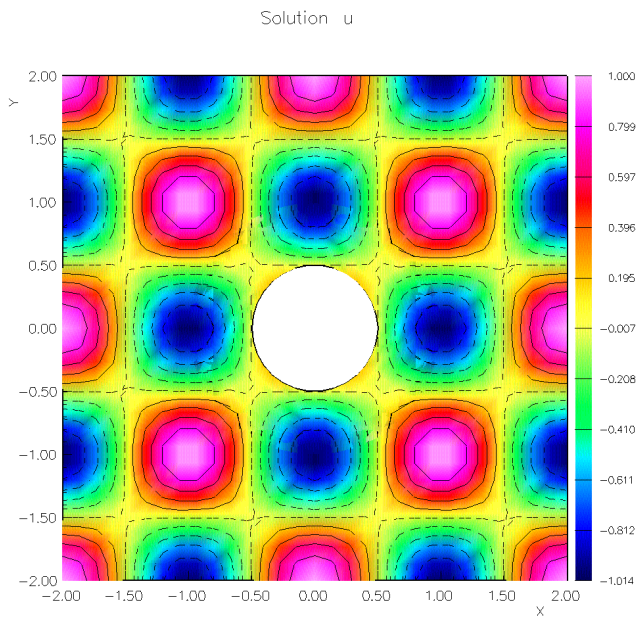


Figure 2: Computed solution on the `cicmg` grid

6.3 Box

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$1.8e+00$	$9.2e-02$	4.8	0.61
2	$1.7e-01$	$9.1e-02$	4.8	0.60
3	$1.5e-02$	$9.1e-02$	4.8	0.60
4	$1.5e-03$	$1.0e-01$	4.8	0.62

Red-Black, box8mg
2 levels, $n_s = 3$, Dirichlet

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$8.8e+00$	0.080	4.8	0.59
2	$7.7e-01$	0.087	4.8	0.60
3	$6.7e-02$	0.088	4.8	0.60
4	$6.0e-03$	0.088	4.8	0.60
5	$5.3e-04$	0.089	4.8	0.60
6	$4.7e-05$	0.089	4.8	0.60
7	$4.3e-06$	0.090	4.8	0.60
8	$3.9e-07$	0.091	4.8	0.60
9	$3.5e-08$	0.091	4.8	0.61
10	$3.2e-09$	0.092	4.8	0.61
11	$3.0e-10$	0.092	4.8	0.61

Red-Black, box16bmg
4 levels, $n_s = 3$, Dirichlet

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$1.5e+02$	0.080	4.7	0.58
2	$1.3e+01$	0.087	4.7	0.59
3	$1.2e+00$	0.087	4.7	0.59
4	$1.0e-01$	0.088	4.7	0.60
5	$9.1e-03$	0.088	4.7	0.60
6	$8.1e-04$	0.089	4.7	0.60
7	$7.2e-05$	0.089	4.7	0.60
8	$6.5e-06$	0.090	4.7	0.60
9	$5.8e-07$	0.090	4.7	0.60
10	$5.3e-08$	0.091	4.7	0.60

Red-Black, box64bmg
5 levels, $n_s = 3$, Dirichlet

Table 3: Multigrid convergence rates for a 3D box.

6.4 Sphere in a box

i	res(i)	rate(i)	WU(i)	ECR(i)
1	$5.7e+00$	0.088	6.1	0.67
2	$4.5e-01$	0.079	6.1	0.66
3	$3.6e-02$	0.081	6.1	0.66
4	$3.9e-03$	0.108	6.1	0.69
5	$7.7e-04$	0.197	5.6	0.75
6	$1.5e-04$	0.199	5.4	0.74
7	$2.7e-05$	0.176	5.1	0.71
8	$5.4e-06$	0.201	5.5	0.75
9	$9.5e-07$	0.175	5.1	0.71
10	$1.8e-07$	0.192	5.6	0.74

Red-Black, sibmg
2 levels, Dirichlet

Table 4: Multigrid convergence rates for a 3D sphere in a box.

7 Singular problems

When the problem being solved is singular, such as the Neumann problem

$$\Delta u = f$$
$$u_n = g$$

there are some difficulties using multigrid and estimating the errors and convergence rates. The problem stems from the facts that the solution is only determined up to a constant and that there is a compatability condition on f and g that may not be exactly satisfied in the discrete equations. If the compatability condition is not satisfied then the residual cannot be driven to zero but rather the solution can only be expected to converge in a generalized sense.

$$AU = F$$

If $l^T A = 0$ then the compatability condition is $l^T F = 0$.

When Oges is used to solve a singular system such as the one given it solves the related non-singular problem

$$\begin{bmatrix} A & \mathbf{r} \\ \mathbf{r}^T & 0 \end{bmatrix} \begin{bmatrix} U \\ \alpha \end{bmatrix} = \begin{bmatrix} F \\ 1 \end{bmatrix} \tag{1}$$

where $\mathbf{r} = [1, 1, \ldots, 1]^T$ is the right null vector. Although the matrix A is singular the augmented matrix is nonsingular and has a unique solution:

$$AU = F - \alpha \mathbf{r}$$
$$\mathbf{r}^T U = 0$$
$$\alpha = \frac{\mathbf{l}^T F}{\mathbf{l}^t \mathbf{r}}$$

We could try to solve the above non-singular system with multigrid. The question arises how to compute α or how to compute iterates α^n that converge. I don't know how to do this. If we knew the left null-vector then we could compute α directly. The left null vector is expensive to compute and thus we may wish to avoid computing it (especially in a moving grid application where the left null vector would be changing as the grid changes.)

Another approach is to use multigrid to iterate as if the problem were non-singular. If we further set the mean value of the solution to be zero at every iteration we would then expect the solution to converge. The only problem is that the residual will not go to zero if the compatibility condition is not satisfied. The multigrid solver relies on knowing an estimate for the norm of the residual in order to cycle properly. We therefore would like an estimated residual that does go to zero.

To get this estimate we can define the solution to our singular problem to be the one that satisfies

$$\mathbf{r}^n = \mathbf{f} - A\mathbf{u}^n$$
$$\tilde{\mathbf{r}}^n = \mathbf{r}^n - \alpha \mathbf{z}^*$$

where α minimizes the expression

$$\min_{\alpha} \|A\mathbf{u} - \mathbf{f} - \alpha \mathbf{w}\|$$

and $\mathbf{w}$ satisfies

$$A\mathbf{w} = \mathbf{r}$$
$$\mathbf{r}^T \mathbf{w} = 1$$

8 Ogmg Function Descriptions

8.1 constructor

Ogmg()

Description: Default constructor.

8.2 constructor

Ogmg(CompositeGrid & mg,
GenericGraphicsInterface *ps_ = 0)

Description: Build a multigrid solver.

mg (input) : grid to use

ps_ (input) : supply an optional GenericGraphicsInterface object for plotting.

Notes: Here are some notes. See the Ogmg User Guide for further details.

Boundary Conditions : Ogmg looks at the coefficient matrix to determine if the ghost line values are extrapolated or not. If not it assumes there is some sort of neumann or mixed boundary conditions and does a few things differently.

Interpolants The multigrid solver needs to interpolate at the different levels. If no Interpolant is found to be associated with a CompositeGrid at a given level the `updateToMatchGrid` routine will build an Interpolant the first time (and update it on subsequent calls). Thus if you have built an Interpolant for any level then you are responsible to update it if the grid changes.

8.3 setPlotStuff

void
set(GenericGraphicsInterface *ps_)

Description: Supply a GenericGraphicsInterface object to use for plotting.

ps_ (input) : pointer to a GenericGraphicsInterface object.

8.4 updateToMatchGrid

void
updateToMatchGrid(CompositeGrid & mg_)

Description: Update the solver to match this grid.

mg (input) : grid to use

8.5 getMaximumResidual

real
getMaximumResidual() const

Description: Return the maximum residual from the last solve

8.6 getNumberOfIterations

int
getNumberOfIterations() const

Description: Return the number of multigrid iterations (cycles).

Return value: the number of iterations.

8.7 sizeOf

real
sizeOf(FILE *file =NULL) const

Description: Return number of bytes allocated by Ogmg; Optionally print detailed info to a file

An estimate of space requirements is $20N$ (2D) or $40N$ (3D) where N is the number of grid points on the finest grid (assumes maximum number of levels so that $.5 + .25 + .125 + ... = 1$. For line smoothers there is addition space required.

file (input) : optionally supply a file to write detailed info to. Choose file=stdout to write to standard output.

Return value: the number of bytes.

8.8 setOgmgParameters

int
setOgmgParameters(OgmgParameters & parameters_)

Description: set parameters equal to another parameter object.

8.9 solve

int
solve(realCompositeGridFunction & u, realCompositeGridFunction & f)

Description: Solve Au=f with multigrid

u (input/output) : initial guess on input, answer on output. It is NOT necessary that u be defined on all multigrid levels. u may only live on finest level. It is best if u satisfies the boundary conditions on input although this is not required.

f (input) : right hand side for the problem. f should be defined for the finest level. As with u, f can only be defined on the finest level if desired.

8.10 cycle

int
cycle(const int & level, const int & iteration, real & maximumDefect)

Description: Perform a multigrid cycle. This routine is called recursively.

8.11 printStatistics

void
printStatistics(FILE *file_ =stdout) const

Description: Print performance statistics such as the cpu time required by various routines.

8.12 setCoefficientArray

int
setCoefficientArray(realCompositeGridFunction & coeff)

Description: Supply the coefficient matrix. matrix (input) : a coefficient matrix defined on all levels.

Ogmg::updateToMatchGrid should have already been called at this point so that the the multigrid-composite grid with extra levels has already been built.

8.13 setOrderOfAccuracy

int
setOrderOfAccuracy(const int & orderOfAccuracy_)

Description: Set the order of accuracy (2 or 4).

orderOfAccuracy_ (input) :

8.14 interpolate

int
interpolate(realCompositeGridFunction & u, const int & grid =-1 */, int level /* =-1)

Description: Interpolate here so we can keep track of the cpu time used.

grid (input) : interpolate this grid only (if possible)

8.15 update

int
update(GenericGraphicsInterface & gi)

Description: Update parameters interactively.

8.16 update

int
update(GenericGraphicsInterface & gi, CompositeGrid & cg)

Description: Update parameters interactively. Use this update if you have not already given a CompositeGrid to Ogmg (through a constructor or with the updateToMatchGrid function).

8.17 smooth

void
smooth(const int & level, int numberOfSmoothingSteps)

Description: This is the ”composite” smooth routine. Smooth on all grids using a possibly different smoother for each grid.

8.18 smoothJacobi

void
smoothJacobi(const int & level, const int & grid)

Description: Jacobi Smoother.

8.19 smoothGaussSeidel

void
smoothGaussSeidel(const int & level, const int & grid)

Description: Gauss Seidel Smoother. NOT implemented yet.

8.20 smoothRedBlack

void
smoothRedBlack(const int & level, const int & grid)

Description: Red-Black Smoother

First smooth ”red” points, then black points

```
Two-dimensions:
      shift1  shift2
        0       0
        1       1
        1       0
        0       1
Three-dimensions:
      shift1  shift2  shift3
        0       0      0
        1       1      0
        1       0      1
        0       1      1
        1       0      0
        0       1      0
        0       0      1
        1       1      1
```

8.21 smoothLine

void
smoothLine(const int & level, const int & grid, const int & direction)

Description: Line Smoother. Zebra line smoothing.

direction (input) : smooth on lines in this direction, 0,1,2

8.22 alternatingLineSmooth

void
alternatingLineSmooth(const int & level, const int & grid)

Description: Here we do alternating line smooths; one line smooth in each direction.

8.23 fineToCoarse(level)

void
fineToCoarse(const int & level)

Description: Transfer the defect from the fine grid at ’level’ to the coarse grid at ’level+1’

level (input): *** these next comments are probably wrong ****

```
             Fine to Coarse (Restriction) Transfer
             -------------------------------------
                f2 <- Restriction( f1 )

      Notes:

       (1) Full Weighting at all Interior Nodes with mask() > 0
       (2) Boundary nodes(*) are full weighted in the boundary
```

```
    (i.e. 1 dimension less) if mask > 0
(3) The first line of fictitious points(*) are full weighted
    amongst fictitious points of the first line,
    (i.e. 1 dimension less) if mask(boundary) > 0
(*) except BC corners or edges where the defect is injected

Philosophy:
  Don't average together defects that come from different types
  of equations. Boundary nodes are assumed to be distinct from
  interior nodes and the first line of fictitious points are
  also assumed to be of a different type. Corners in 2D
  or BC edges in 3D are also assumed to be distinct.
```

8.24 fineToCoarse(level,grid)

void
fineToCoarse(const int & level, const int & grid)

Description: Transfer the defect from the fine grid at 'level' to the coarse grid at 'level+1'

level,grid (input):

Notes: Full weighting on interior and boundary points where the equation is applied. If the BC for ghost points is extrapolation (e.g. if there is a dirichlet BC) then the boundary defects are averaged along the boundary.

8.25 coarseToFine(level)

void
coarseToFine(const int & level)

Description: Correction: coarse to fine transfer.

$$u[level] += \textbf{Prolongation}[u[level+1]]$$

8.26 coarseToFine(level,grid)

void
coarseToFine(const int & level, const int & grid)

Description: Correct a Component Grid

$$u(i,j) = u(i,j) + P[u2(i,j)] \qquad \text{(P : Prolongation)}$$

cp21,cp22,cp23 : coeffcients for prolongation, 2nd order cp41,cp41,cp43 : coeffcients for prolongation, 4th order

8.27 defect(level)

void
defect(const int & level)

Description: Defect computation

Fill in defectMG[level] with f-Lu

8.28 defect(level,grid)

void
defect(const int & level, const int & grid)

Description: Defect computation on a component grid

Compute defectMG.multigridLevel[level][grid]

8.29 getDefect

void
getDefect(const int & level,
const int & grid,
realArray & f,
realArray & u,
const Index & I1,
const Index & I2,
const Index & I3,
realArray & defect,
const int lineSmoothOption)

Description: Defect computation on a component grid

Determine the defect = f - C*u

This routine knows how to efficiently compute the defect for rectangular and non-rectangular grids. It also knows how to compute the defect for line smoothers Input - level,grid f,u I1,I2,I3 lineSmoothOption = -1 : = 0,1,2 : compute defect for line solve in direction 0,1,2 Output - defect

8.30 initializeBoundaryConditions

int
initializeBoundaryConditions(realCompositeGridFunction & coeff)

Description: Determine the type of boundary condition that is imposed on the **GHOSTLINE** for each side of each grid. There are 3 possibilities:

1. extrapolation : ghost point is extrapolated. This requires a special formula for the defect.
2. equation : an equation such as a neumann or mixed boundary condition. This uses basically the same formula for the defect, but shifted to be centred on the boundary
3. combination : a combination of the above two appears on the boundary.

Notes: We check the classify array to determine the type of boundary condition.

8.31 applyBoundaryConditions(level,...)

int
applyBoundaryConditions(const int & level, RealCompositeGridFunction & u, RealCompositeGridFunction & f)

Description: Assign boundary conditions on the **GHOSTLINE** for each side of each grid.

8.32 applyBoundaryConditions(level,grid,...)

int
applyBoundaryConditions(const int & level,
const int & grid,
RealMappedGridFunction & u,
RealMappedGridFunction & f)

Description: Assign boundary conditions on the **GHOSTLINE** for each side of a grid. Values on the actual boundary are assumed to be done in the smoothing step since the coefficient maxtrix should hold the proper equation there (a Dirichlet BC for example).

level,grid (input) :

u (input/output) : apply BC's to this grid function.

f (input): rhs to the equation (needed to compute the defect for non-extrapolation BC's)

There are 3 possibilities:

1. extrapolation : ghost point is extrapolated. This requires a special formula for the defect.
2. equation : an equation such as a neumann or mixed boundary condition. This uses basically the same formula for the defect, but shifted to be centred on the boundary
3. combination : a combination of the above two appears on the boundary.

8.33 buildExtraLevels

int
buildExtraLevels(CompositeGrid & mg)

Description: Build extra multigrid levels. This routine will create coarser levels automatically. The tricky part is to determine how to interpolate on the new coarser levels. After a grid is coarsened it may no longer have enough interpolation points. We add new interpolation points to fill in the gaps. The width of the interpolation stencil is reduced, on a point by point basis, if necessary.

mg (input/output):

References

[1] W. HENSHAW, *Overture: An object-oriented framework for solving PDEs in moving geometries on overlapping grids using C++*, in Proceedings of the Third Symposium on Overset Composite Grid and Solution Technology, 1996.

[2] ——, *Oges user guide, a solver for steady state boundary value problems on overlapping grids*, Research Report UCRL-MA-132234, Lawrence Livermore National Laboratory, 1998.

[3] ——, *Other stuff for overture, user guide, version* 1.0, Research Report UCRL-MA-134292, Lawrence Livermore National Laboratory, 1999.

[4] W. HENSHAW AND G. CHESSHIRE, *Multigrid on composite meshes*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 914–923.

Index

basic steps, 4

convergence criteria, 5

multigrid, 4
    overlapping grid algorithm, 4