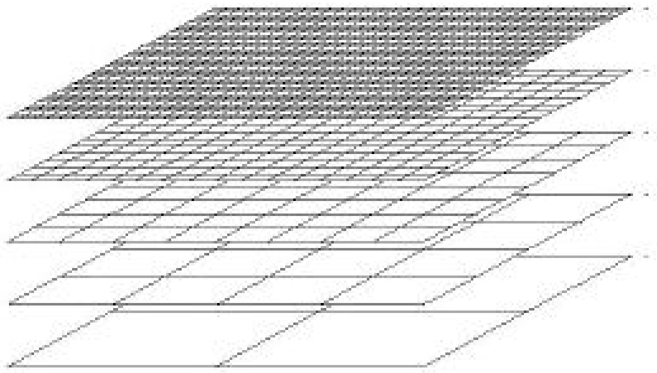
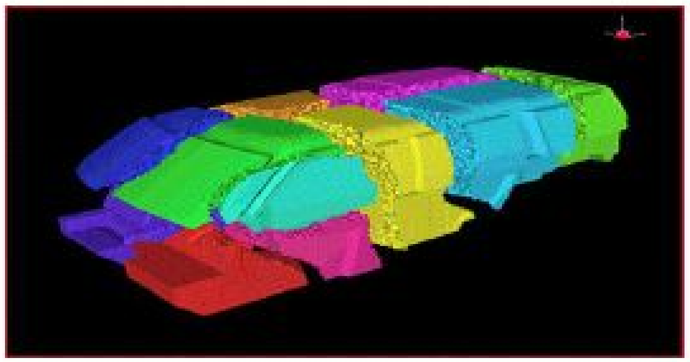




Multigrid and Domain Decomposition in PETSc

Barry Smith

PETSc Developer

Mathematics and Computer Science Division
Argonne National Laboratory

Order of Presentation

- Composition of preconditioners
- Overlapping Schwarz methods
- Multigrid methods
  - Background
  - Low level interface
  - Simple interface
  - Nonlinear methods (FAS)
  - algebraic methods
- Balancing Neumann-Neumann algorithm

Composition of Preconditioners

at least in some subspace

A **linear** operator that **improves** an approximate solution to a linear system.

$$x \Leftarrow x + B(b - Ax) = x + BA(x^* - x) = x + Be$$

Constructing a preconditioner from two preconditioners.

$$y \Leftarrow x + B_1(b - Ax)$$

$$x \Leftarrow y + B_2(b - Ay)$$

Multiplicative version

$$x \Leftarrow x + (B_1 + B_2 - B_2AB_1)(b - Ax)$$

Additive version

$$x \Leftarrow x + (B_1 + B_2)(b - Ax)$$

Generally accelerated with a Krylov method (e.g. GMRES or CG).

Composition of Preconditioners

```
#include "petscpc.h"
PCSetType(pc,PCCOMPOSITE);
PCCompositeSetType(pc,[PC_COMPOSITE_ADDITIVE,
                       PC_COMPOSITE_MULTIPLICATIVE]);
PCCompositeSetUseTrue(pc);

PCCompositeAddPC(pc,PCJACOBI);
PCCompositeAddPC(pc,PCILU);
```

```
-pc_type composite
-pc_composite_type [additive,multiplicative]
-pc_composite_true
-pc_composite_pcs jacobi,ilu
-sub_pc_ilu_levels 2
```

Preconditioners Defined by (near) Galerkin Process

Define restriction operators:

- R_i maps from a right hand side to a smaller, weighted right hand side.
- R_i^T interpolates from a subspace of the solution space to the solution space.

$$\begin{aligned} B_i &= R_i^T (R_i A R_i^T)^{-1} R_i \\ B_i &= R_i^T S_i R_i \end{aligned}$$

Special cases - R_i has a single 1 per row, $R_i A R_i^T$ is a submatrix of A

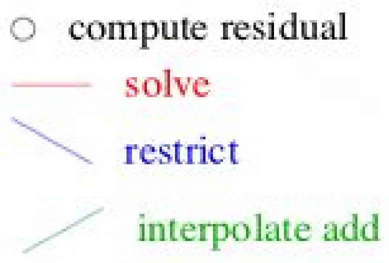
- overlapping Schwarz methods - R_i selects all unknowns in a local domains
- field split methods - R_i selects the ith component at each grid point

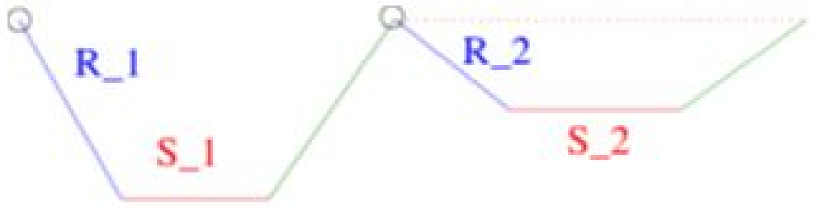
Composition of "Galerkin" Preconditioners

Multiplicative version

$$y \Leftarrow x + R_1^T S_1 R_1 (b - Ax)$$

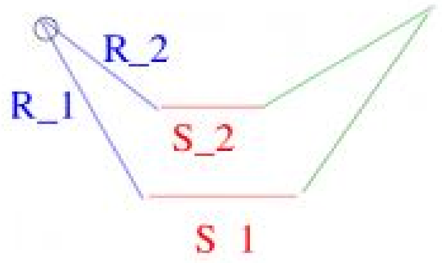
$$x \Leftarrow y + R_2^T S_2 R_2 (b - Ay)$$





Additive version

$$x \Leftarrow x + (R_1^T S_1 R_1 + R_2^T B_2 R_2)(b - Ax)$$



Additive Schwarz Methods

Symmetric

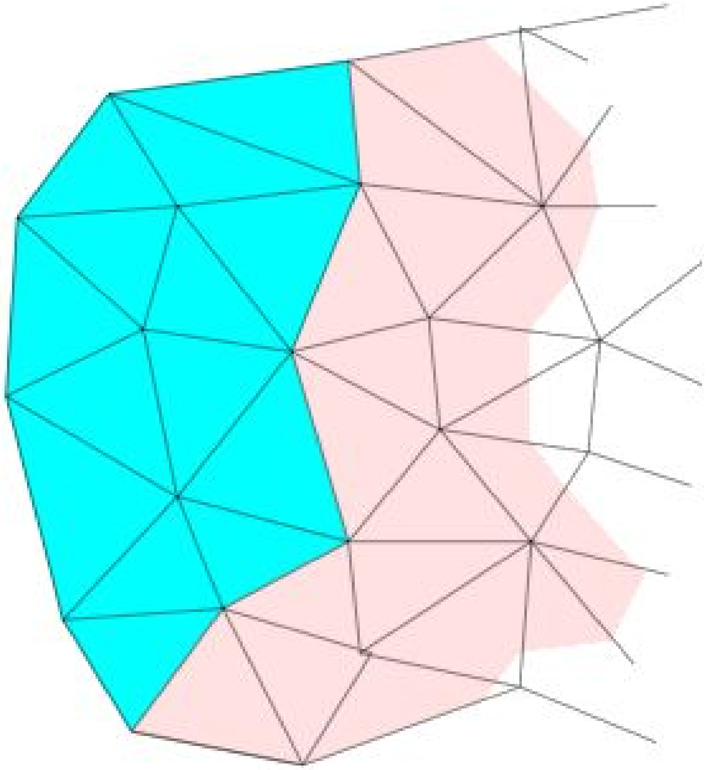
PCSetType(pc,PCASM);

PCASMSetType(pc,[PC_ASM_BASIC,PC_ASM_RESTRICT,PC_ASM_INTERPOLATE])

−pc_asm_type [basic,restrict,interpolate]

Default

$$B_i = \hat{R}_i^T (R_i A R_i^T)^{-1} \tilde{R}_i$$



Additive Schwarz Method Options

PCASMSetTotalSubdomains(pc,n)

PCASMSetOverlap(pc,o) ← Defaults to one grid point

PCASMSetUseInPlace(pc)

−pc_asm_subdomains n

−pc_asm_overlap o

−pc_asm_in_place

Domains need not overlap

PCASMSetLocalSubdomains(pc,l,is[])

User function that adjusts sub-matrices; usually changing boundary conditions of subdomains

PCASMGetLocalSubdomains(pc,**int** *l,*is[])

PCASMGetSubKSP(pc,**int** *l,**int** *lstart,*ksps[])

PCASMGetLocalSubmatrices(pc,**int** *l,*mat[])

PCSetModifySubMatrices(pc,(*f)(PC,**int** l,IS rows[],IS cols[],mats[],**void** *ctx),**void** *ctx)

Extending the Overlap

$$\begin{array}{c|ccccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
0 & x & x & & x & & x & & & & & x \\
1 & x & x & x & & & x & & & x & & \\
2 & & x & x & x & & & & & & x & \\
3 & & x & x & x & x & x & & & & & \\
4 & & & x & x & x & x & & & & & \\
5 & & & x & & x & x & x & & & & \\
6 & & & & x & & x & x & & & & x \\
7 & & x & & & & & x & x & & & \\
8 & & x & & & & & & x & x & x & \\
9 & & & & & & x & & & x & x & x \\
10 & & & x & x & & & & & x & x & x
\end{array}$$

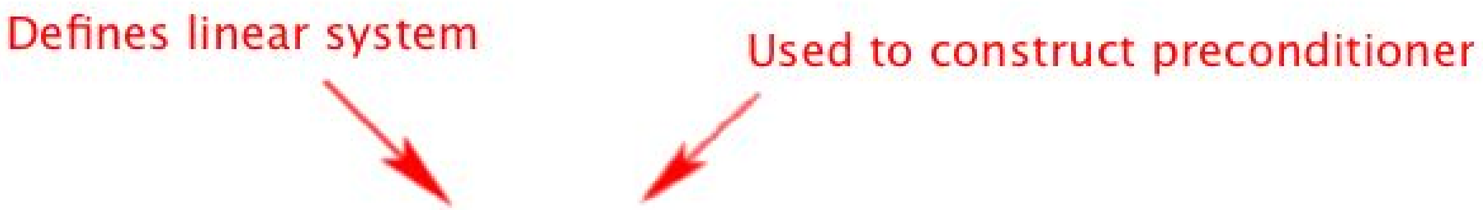
Extending the Overlap

$$\begin{array}{c|ccccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
0 & x & x & x & x & x & x & x & x & x & x & x \\
1 & x & x & x & & & x & & & x & & \\
2 & & x & x & x & & & & & & x & \\
3 & & x & x & x & x & x & & & & & \\
4 & & & x & x & x & x & & & & & \\
5 & & & x & & x & x & x & & & & \\
6 & & & & x & & x & x & & & & x \\
7 & & x & & & & & x & x & & & \\
8 & & x & & & & & & x & x & x & \\
9 & & & & & & x & & & x & x & x \\
10 & & & x & x & & & & & x & x & x
\end{array}$$

Extending the Overlap

$$\begin{array}{cc}
 & \begin{array}{ccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} &
\begin{pmatrix}
x & x & x & x & x & x & x & x & x & x & x \\
x & x & x &   &   & x &   &   & x &   &   \\
  & x & x & x &   &   &   &   &   & x &   \\
  & x & x & x & x & x &   &   &   &   &   \\
  &   & x & x & x & x &   &   &   &   &   \\
  &   & x &   & x & x & x &   &   &   &   \\
  &   &   & x &   & x & x &   &   &   & x \\
  & x &   &   &   &   & x & x &   &   &   \\
  & x &   &   &   &   &   & x & x & x &   \\
  &   &   &   &   & x &   &   & x & x & x \\
  &   & x & x &   &   &   &   & x & x & x
\end{pmatrix}
\end{array}$$

Matrix Free Versions of Additive Schwarz Method



KSPSetOperators(ksp,Mat A,Mat B,SAME_NONZERO_PATTERN)

- Use matrix-free A, but sparse representation matrix B or
- Use MATSHELL (MatCreateShell()) for B and
    - provide custom MatGetSubMatrices() that returns either

      - matrix-free or

      - sparse representation matrices

MatShellSetOperation(B,MATOP_GET_SUBMATRICES,MyGetSubMatrices)

Field Split Methods

PCSetType(pc,PCFIELDSPLIT)

PCFieldSplitSetType(pc,[PC_COMPOSITE_ADDITIVE,
PC_COMPOSITE_MULTIPLICATIVE])

PCFieldSplitSetFields(pc,nfields,**int** *fields)

PCFieldSplitGetSubKSP(pc,**int** *n,KSP *ksps[])

−pc_fieldsplit_type additive,**multiplicative**

−pc_fieldsplit_%d_fields f1,f2,...; e.g. −pc_fieldsplit_0_fields 0,1

−fieldsplit_%d_ksp_type typename; eg. −pc_fieldsplit_0_ksp_type gmres

−pc_fieldsplit_default ← Put unlisted fields into seperate solvers

PETSc Inconsistency

PCASM and PCFIELDSPLIT have Krylov methodes (KSPs) for "inner" solvers; PCCOMPOSITE has PCs.

Why? No good reason.

If you want them to use Krylov methods (be KSPs), use a PCType of PCKSP for the composite preconditioners.

Similarly, for ASM and field split "inner" solves, use a KSPType of KSPPRE-ONLY to skip the Krylov method.

Subspace Methods

Basic idea: Decompose the solution space into several subspaces; for each of which you have an efficient solver. Compose the resulting preconditioners to generate an efficient global solver.

Define the subspaces by interpolation from the **subspace representation** to the **solution (global) space representation**, R_i^T.

Subspace representation simply means the coefficients of the subspace vector.

- In ASM the subspaces (and hence interpolations) are defined by domains;
- for field split methods they are defined by components;
- for multigrid they are defined by coarser grids. More preciously, they are defined by the "rough" (high energy, as measured in the A norm) modes on each of the coarser grids. (The smooth modes are handled by the coarser grids).

Subspace Method Components

Subspace methods are completely defined algorithmically by

- R_i (more generally $\hat{R}_i$ and $\tilde{R}_i^T$)
- the operator on each subspace, A_i, e.g. $R_i A R_i^T$
- the solver, $S_i(A_i)$, on each subspace and
- the way the "inner" solvers are composed.

Special case - the R_i are obtained by interpolating between neighboring grids

- multigrid

Definitions for Multigrid

- n is the number of grids
- 0 is **always** the coarsest grid
- $n-1$ is **always** the finest grid
- A_{n-1} is the fine grid (true) operator
- r_{i+1} represents the restriction from level $i+1$ to level i

  (there is no r_0).

$$R_i = r_i r_{i+1} ... r_{n-1}$$

Dang: this notation is inconsistent with the generic way of letting R_i represent the restriction to the ith subspace but it is, :-), what was used in PETSc.

Subspace Methods with Two Levels

(1) solve on the fine grid

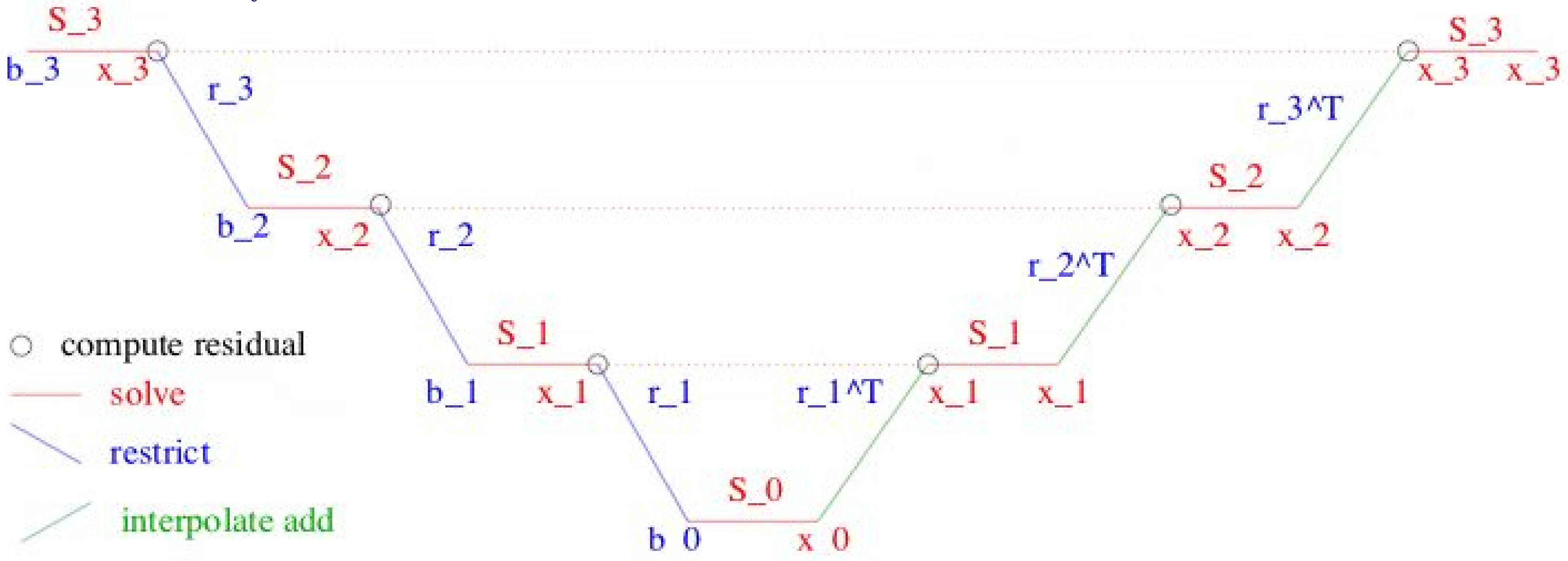
(2) solve on the coarse grid

(3) solve on the fine grid

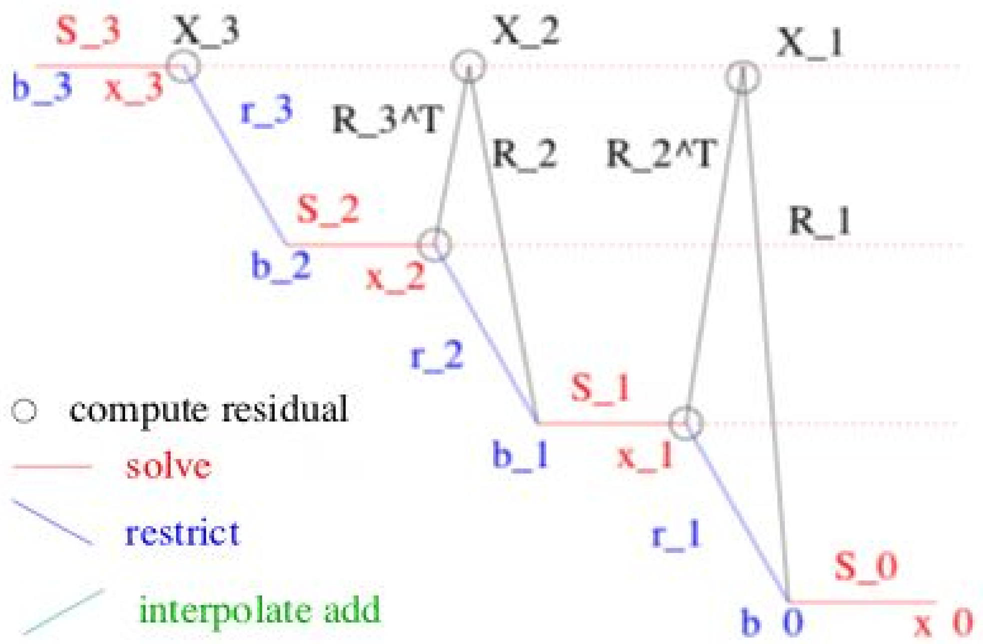
$$\begin{aligned}
x_1 &\Leftarrow S_1 b \\
x_1 &\Leftarrow x_1 + R_0^T S_0 R_0 (b - A_1 x_1) \\
x_1 &\Leftarrow x_1 + S_1 (b - A_1 x_1)
\end{aligned}$$

V-Cycle Multigrid Definition

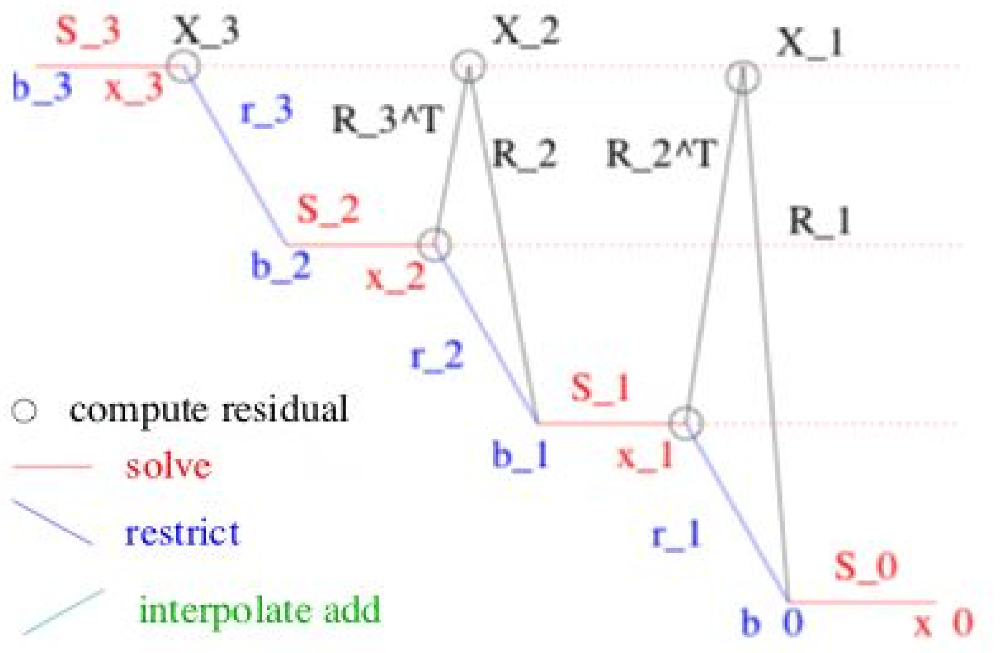
```
V-Cycle($b_i$)
$x_i \Leftarrow S_i b_i$
if not coarsest level
    $x_i + \Leftarrow r_{i+1}^T$ V-Cycle($r_{i+1}(b_{i+1} - A_{i+1} x_{i+1})$)
    $x_i + \Leftarrow S_i(b_i - A_i x_i)$
return $x_i$
```



V-Cycle Multigrid as a Subspace Method



V-Cycle Multigrid as a Subspace Method



$$
\begin{aligned}
X_{n-1} &\Leftarrow S_{n-1} b \\
X_{n-2} &\Leftarrow X_{n-1} + R^T_{n-1} S_{n-2} r_{n-1} (b - A x_{n-1}) \\
X_{n-3} &\Leftarrow X_{n-2} + R^T_{n-2} S_{n-3} r_{n-2} (b_{n-2} - A_{n-2} x_{n-2}) \\
&\ldots \\
X_1 &\Leftarrow X_2 + R^T_2 S_1 r_2 (b_2 - A_2 x_2) \\
X_0 &\Leftarrow X_1 + R^T_1 S_0 r_1 (b_1 - A_1 x_1)
\end{aligned}
$$

V-Cycle Multigrid as a Subspace Method

$$\begin{aligned}
X_{n-1} &\Leftarrow S_{n-1}b \\
X_{n-2} &\Leftarrow X_{n-1} + R_{n-1}^T S_{n-2} \boxed{r_{n-1}(b - A_{n-1}x_{n-1})} \\
X_{n-3} &\Leftarrow X_{n-2} + R_{n-2}^T S_{n-3} \boxed{r_{n-2}(b_{n-2} - A_{n-2}x_{n-2})} \\
&\dots \\
X_1 &\Leftarrow X_2 + R_2^T S_1 \boxed{r_2(b_2 - A_2x_2)} \\
X_0 &\Leftarrow X_1 + R_1^T S_0 r_1(b_1 - A_1x_1)
\end{aligned}$$

is actually identical to

$$\begin{aligned}
X_{n-1} &\Leftarrow S_{n-1}b \\
X_{n-2} &\Leftarrow X_{n-1} + R_{n-1}^T S_{n-2} \boxed{R_{n-1}(b - AX_{n-1})} \\
X_{n-3} &\Leftarrow X_{n-2} + R_{n-2}^T S_{n-3} \boxed{R_{n-2}(b - AX_{n-2})} \\
&\dots \\
X_1 &\Leftarrow X_2 + R_2^T S_1 \boxed{R_2(b - AX_2)} \\
X_0 &\Leftarrow X_1 + R_1^T S_0 R_1(b - AX_1)
\end{aligned}$$

$$r_{i+1}(b_{i+1} - A_{i+1}x_{i+1}) = R_{i+1}(b - AX_{i+1})$$

Proof by induction: Assume

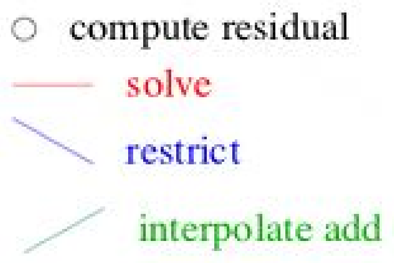
$$r_{i+2}(b_{i+2} - A_{i+2}x_{i+2}) = R_{i+2}(b - AX_{i+2})$$

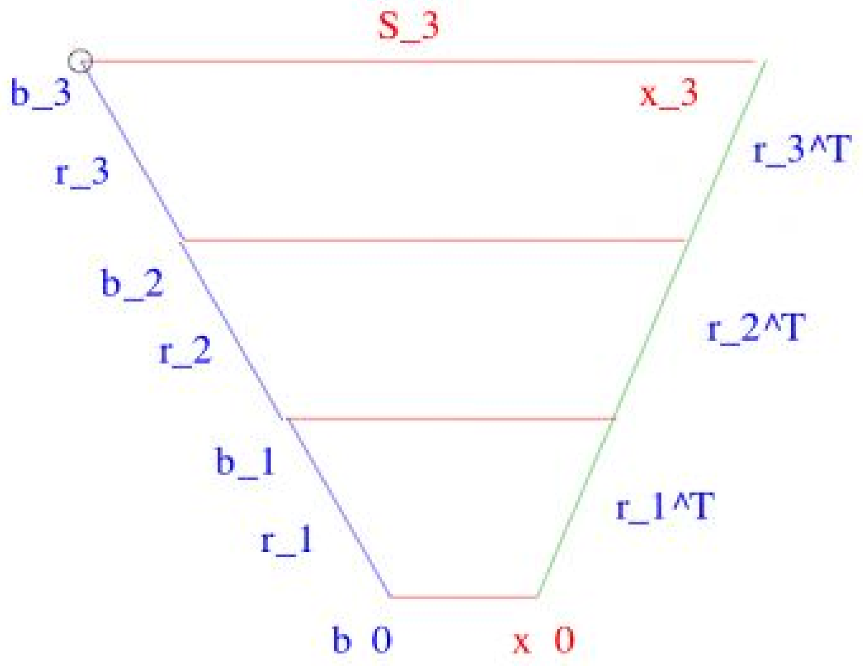
$$\begin{aligned}
r_{i+1}(b_{i+1} - A_{i+1}x_{i+1}) &= r_{i+1}(r_{i+2}(b_{i+2} - A_{i+2}x_{i+2}) - A_{i+1}x_{i+1}) \\
&= r_{i+1}(R_{i+2}(b - AX_{i+2}) - A_{i+1}x_{i+1}) \\
&= r_{i+1}(R_{i+2}(b - AX_{i+2}) - R_{i+2}AR_{i+2}^T x_{i+1}) \\
&= r_{i+1}R_{i+2}(b - AX_{i+2} - AR_{i+2}^T x_{i+1}) \\
&= R_{i+1}(b - A(X_{i+2} + R_{i+2}^T x_{i+1})) \\
&= R_{i+1}(b - AX_{i+1})
\end{aligned}$$

Note: Induction is going down from $n-1$ to 1.

Additive Multigrid - Multilevel Methods

$$
\begin{aligned}
B &\Leftarrow \sum_{i=0}^{n-1} R_i^T S_i R_i \\
&\Leftarrow r_{n-1}^T(S_{n-1} + r_{n-2}^T(S_{n-2} + r_{n-3}^T(....)...)r_{n-3})r_{n-2})r_{n-1}
\end{aligned}
$$

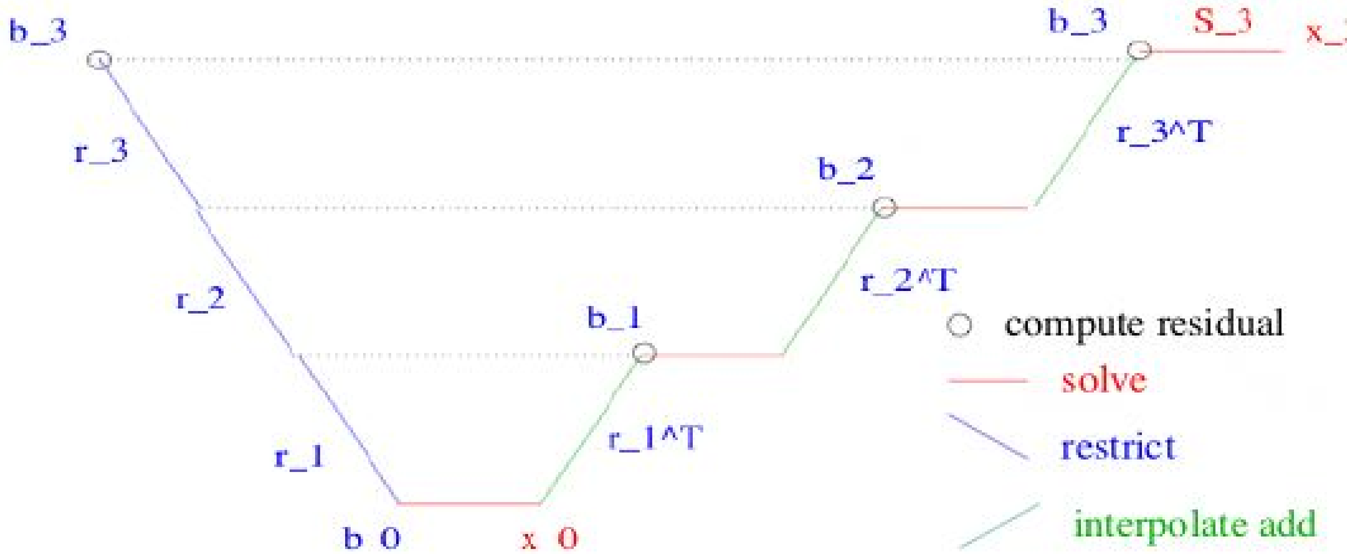




Preconditioner Cascadic - One Way Multigrid

$$b_0 \Leftarrow r_1 b_1 \Leftarrow r_1(r_2 b_2) \Leftarrow r_1(r_2(r_3 b_3))\ldots$$

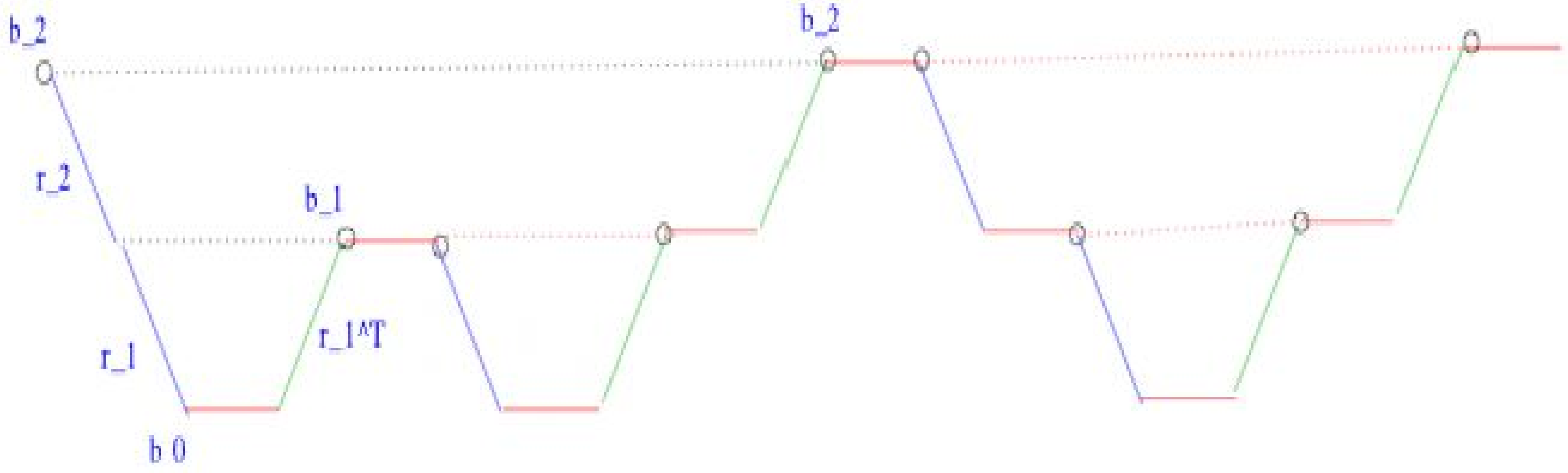
$$\begin{aligned} x_0 &\Leftarrow S_0 b_0 \\ x_1 &\Leftarrow r_1^T x_0 + S_1(b_1 - A_1 r_1^T x_0) \\ &\ldots \\ x_{n-1} &\Leftarrow r_{n-1}^T x_{n-2} + S_{n-1}(b - A r_{n-1}^T x_{n-2}) \end{aligned}$$



Full Multigrid Preconditioner

$$b_0 \Leftarrow r_1 b_1 \Leftarrow r_1(r_2 b_2) \Leftarrow r_1(r_2(r_3 b_3))...$$

$$\begin{aligned}
x_0 &\Leftarrow S_0 b_0 \\
x_1 &\Leftarrow r_1^T x_0 + \text{V-cycle}(b_1 - A_1 r_1^T x_0) \\
&\;... \\
x_{n-1} &\Leftarrow r_{n-1}^T x_{n-2} + \text{V-cycle}(b - A r_{n-1}^T x_{n-2})
\end{aligned}$$



PETSc's Multigrid: Algorithmic Options

#include "petscmg.h"

PCSetType(pc,PCMG);

MGSetType(pc,[**MGMULTIPLICATIVE**,MGADDITIVE,MGFULL,MGCASCADE]

−pc_mg_type **multiplicative**,additive,full,cascade

MGSetLevels(pc,**int** nlevels,MPI_Comm *);

MGGetLevels(pc,**int** *nlevels);

−pc_mg_nlevels nlevels

MGSetCycles(pc,[**MG_V_CYCLE**,MG_W_CYCLE]);

MGSetCyclesOnLevel(pc,**int** level,[**MG_V_CYCLE**,MG_W_CYCLE]);

−pc_mg_cycles [1,2]

MGSetNumberSmoothUp(pc,**int** s); MGSetNumberSmoothDown(pc,**int** s);

−pc_mg_smoothup s −pc_mg_smoothdown s

PETSc's Multigrid: Smoother Options

Each solver (smoothers and coarse grid solve) is represented by a KSP object.

- Use same pre and post smoother

  MGGetSmoother(pc,**int** level,KSP *ksp);

- Use different pre and post smoother

  MGGetSmootherDown(pc,**int** level,KSP *dksp);
  MGGetSmootherUp(pc,**int** level,KSP *uksp);

Set smoother options via the KSP objects.

MGGetCoarseSolve(pc,KPS *cksp) == MGGetSmoother(pc,0,KSP *cksp)

PETSc's Multigrid: Smoother Options

Command line options for smoothers

−mg_coarse_[ksp,pc]_xxx

−mg_levels_[ksp,pc]_xxx

−mg_levels_%d_[ksp,pc]_xxx

Cannot set different options for pre and post smoothers from the command line.

PETSc's Multigrid: Monitoring

−pc_mg_log – log information about time spent on each level of the solver

−pc_mg_monitor – call −ksp_monitor on all levels of smoothers

−pc_mg_dump_matlab – dump all the multigrid matrices to Matlab

−pc_mg_dump_binary – dump all the multigrid matrices to a binary file (coming soon)

You can also, of course, monitor individual smoothers with, for example,

−mg_levels_3_ksp_monitor

All the multigrid options in use (including all smoother options)

−ksp_view

PETSc's Multigrid: Defaults

- Traditional (multiplicative) multigrid
- V-cycle
- 1 pre and 1 post smooth
- Direct solver on coarse problem
  - + run redundantly on each process
  - + can use parallel direct solver
    - * MUMPS, Spooles, SuperLU_dist or
    - * the Tufo-Fischer scalable coarse solver (-mg_coarse_pc_type tfs)
- FGMRES on outer iteration (often overkill)
- Smoothers
  - + GMRES (often overkill)
  - + block Jacobi ilu(0)

PETSc's Multigrid as a Solver

By default, multigrid in PETSc is treated as a preconditioner; not a standalone solver. Use

−ksp_type richardson

or

KSPSetType(ksp,KSPRICHARDSON);

to treat it as a solver. **Not** -ksp_type preonly.

PETSc's Low-Level Multigrid Interface: Vectors

Must provide 3 vectors for each level

```
MGSetRhs(pc,int l,Vec b);
MGSetX(pc,int l,Vec x);
MGSetR(pc,int l,Vec r);
```

Used as work vectors for the multigrid process.

b and x are not needed on the finest grid

r is not needed on the coarsest grid

r is not needed for MGADDITIVE

All Optional (coming soon)

PETSc's Low-level Interface: Restriction/Interpolati

Must provide restriction and interpolation.

MGSetRestriction(pc,**int** level,Mat R);
MGSetInterpolate(pc,**int** level,Mat P);

$$0 < level < nlevels$$

If only one is provided, its transpose is used for the other. (coming soon)

Recall Mat may represent a matrix-free matrix so restriction/interpolation may be defined by a function and not explicitly represented as a sparse matrix.

PETSc's Low-level Interface: Operators

Must provide operator (as Mat) for each level

MGGetSmoother(pc,**int** level,KSP *ksp);

KSPSetOperators(ksp,A[level],B[level],SAME_NONZERO_PATTERN);

Will default to system matrices on finest level if not given (coming soon).

Can use different operators on pre and post smoothing

MGGetSmootherDown(pc,**int** level,KSP *dksp);

MGGetSmootherUp(pc,**int** level,KSP *uksp);

Or, provide only fine grid operator (coming soon)

MGSetGalerkinCoarse(pc);

−pc_mg_galerkin

PETSc's Low-level Interface: Residual Computation

$$r = b - Ax$$

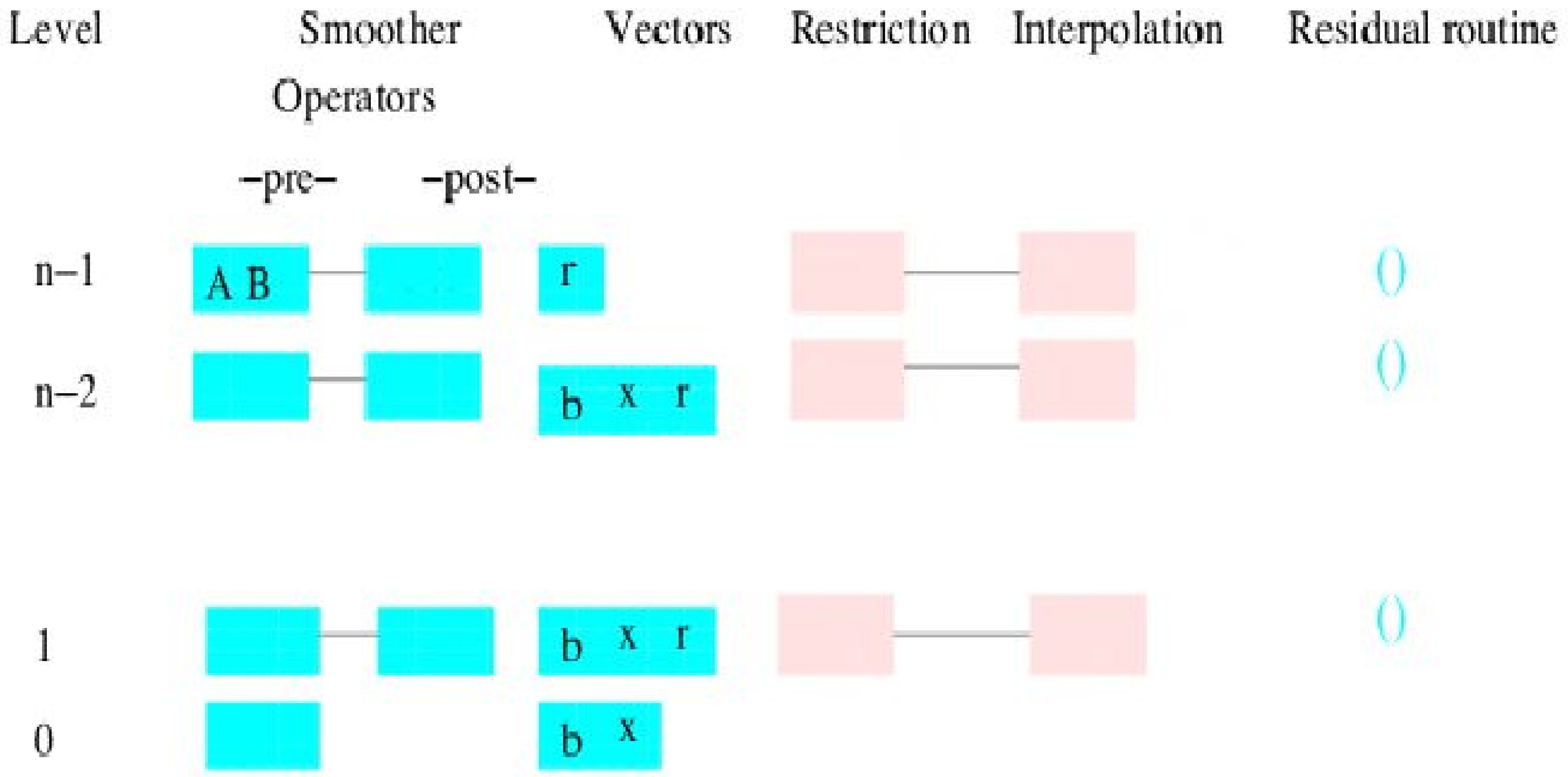
MGSetResidual(pc,level,PetscErrorCode (*residual)(Mat,Vec b,Vec x,Vec r),Mat A);

$$0 < level < nlevels$$

Defaults to computing it explicitly using the operator given on each level with

MGDefaultResidual(Mat,Vec b,Vec x,Vec r);

PETSc's Low-level Interface: Summary



PETSc's Low-level Interface: Sample Code

```
PCSetType(pc,PCMG);
MGSetLevels(pc,nlevels,PETSC_NULL);
for (i=1; i<nlevels; i++) {
  MatCreate(......,&R)
  .....
  MGSetRestriction(pc,i,R);
}
for (i=0; i<nlevels; i++) {
  MatCreate(......,&A)
  .....
  MGGetSmoother(pc,i,&sksp);
  KSPSetOperators(sksp,A,A,SAME_NONZERO_PATTERN);
}
KSPSolve(ksp,b,x);
```

PETSc's High-level Multigrid Interface

DMMG - manages construction of multigrid preconditioner/solver for

- linear problems (using KSP)
- nonlinear problems (using SNES)
- also supports grid sequencing (with/out multigrid solving).

User/Library - provides codes to generate

- right hand side (linear problems)
- Jacobian matrices
- interpolation/restriction operators
- function evaluations (nonlinear problems).

for a given level of discretization.

KSP (and SNES) vs DMMG

- KSP (and SNES) represent a classical (but object oriented) procedural programming style.

  * You construct the various objects needed in your code, put them together, and then call the solver.

  * You think you know what the code is actually doing when it runs.

- DMMG represents an object oriented "framework" programming style.

  * The "framework" DMMG creates virtually all the objects for you and "runs them".

  * You may have little idea what the "framework" is actually doing.

  * Essentially someone (me in this case) has taken a combination of pieces of "classical (but object oriented) procedural programming style" code and combined them to allow easy solution of a class of problems.

  + If the class fits, the framework is useful for you.

  + If the class does not fit, the framework is useless.

PETSc's High-level Multigrid Interface: Example 1

```
#include "petscdmmg.h"
extern FormRHS(DMMG,Vec);
extern FormMatrix(DMMG,Mat);

DMMGCreate(PETSC_COMM_WORLD,3,PETSC_NULL,&dmmg);
DACreate2d(PETSC_COMM_WORLD,DA_NONPERIODIC,DA_STENCIL_STAR,
           3,3,PETSC_DECIDE,PETSC_DECIDE,1,1,0,0,&da);
DMMGSetDM(dmmg,(DM)da);

DMMGSetKSP(dmmg,FormRHS,FormMatrix);
DMMGSolve(dmmg);
Vec x = DMMGGetx(dmmg);
```

PETSc's High-level Multigrid Interface: Example 2

```
#include "petscdmmg.h"
extern FormFunction(DMMG,Vec,Vec,void *usr);
extern FormJacobian(DMMG,Vec,Mat*,Mat*,MatStructure *str,void* usr);

DMMGCreate(PETSC_COMM_WORLD,3,PETSC_NULL,&dmmg);
DACreate2d(PETSC_COMM_WORLD,DA_NONPERIODIC,DA_STENCIL_STAR,
             3,3,PETSC_DECIDE,PETSC_DECIDE,1,1,0,0,&da);
DMMGSetDM(dmmg,(DM)da);

DMMGSetSNES(dmmg,FormFunction,FormJacobian);
DMMGSolve(dmmg);
Vec x = DMMGGetx(dmmg);
```

PETSc's High-level Multigrid Interface: Example 3

```
#include "petscdmmg.h"
extern FormFunctionLocal(DALocalInfo *info,Field **x,Field **f,void *usr);
extern FormJacobianLocal(DALocalInfo *info,Field** x,Mat A,void *usr);

DMMGCreate(PETSC_COMM_WORLD,3,PETSC_NULL,&dmmg);
DACreate2d(PETSC_COMM_WORLD,DA_NONPERIODIC,DA_STENCIL_STAR,
              3,3,PETSC_DECIDE,PETSC_DECIDE,1,1,0,0,&da);
DMMGSetDM(dmmg,(DM)da);

DMMGSetSNESLocal(dmmg,FormFunctionLocal,FormJacobianLocal,...
DMMGSolve(dmmg);
Vec x = DMMGGetx(dmmg);
```

DMMG - DM Multigrid

What does it do?

- Creates a KSP or SNES object
- Sets the PCType to PCMG
- For each level creates and fills up in the PCMG object
    - * the vectors
    - * the restriction/interpolation
    - * the matrices

How does it work?

- That comes later.

DMMG Optional Functions

Either provide an initial guess or code to compute it

DMMGInitialGuessCurrent(DMMG,Vec);

DMMGSetInitialGuess(DMMG*,PetscErrorCode (*)(DMMG,Vec));

DMMGView(DMMG*,PetscViewer);

DMMGSetUseMatrixFree(DMMG*);

DMMGSetUseGalerkinCoarse(DMMG*);

−dmmg_galerkin

DMMGSetNullSpace(DMMG*,PetscTruth const?,**int** nsize, (*generatenull)(DMMG,Vec[]));

* compare to KSPSetNullspace()

Access and Controlling DMMG

Acessing the right hand side and solution

Vec DMMGGetb(DMMG*)

Vec DMMGGetx(DMMG*)

The Jacobian and “approximate” Jacobian

Mat DMMGGetJ(DMMG*)

Mat DMMGGetB(DMMG*)

Acessing the solvers to set parameters

KSP DMMGGetKSP(DMMG*)

SNES DMMGGetSNES(DMMG*)

Access and Controlling DMMG

int DMMGGetLevels(DMMG*)

Accessing the user context (the data based to user functions)

void* DMMGGetUser(DMMG*,level)

DMMGSetUser(DMMG*,level,**void** *usr)

Access the objects that manage the "grids and discretizations"

DA DMMGGetDA(DMMG*)

VecPack DMMGGetVecPack(DMMG*)

Controlling DMMG

Setting the number of levels at runtime

−dmmg_nlevels

Use true grid sequencing

−dmmg_grid_sequence

For linear problems equivalent to standard full multigrid.

Can be used with any linear solver, does not require multigrid as the solver.

Monitoring/Viewing DMMG

−dmmg_view

−dmmg_vecmonitor

−dmmg_ksp_monitor

−dmmg_snes_monitor

Monitoring/Viewing DMMG: Example 1

```
ex29 -dmmg_view
```

```
DMMG Object with 3 levels
  X range of indices: 0 20, Y range of indices: 0 5
  X range of indices: 0 40, Y range of indices: 0 10
  X range of indices: 0 80, Y range of indices: 0 20
```

Monitoring/Viewing DMMG: Example 1

```
ex29 -dmmg_view
```

```
DMMG Object with 3 levels
  X range of indices: 0 20, Y range of indices: 0 5
  X range of indices: 0 40, Y range of indices: 0 10
  X range of indices: 0 80, Y range of indices: 0 20

FieldNames:   phi   psi   U   F
```

Monitoring/Viewing DMMG: Example 1

```
ex29 -dmmg_view


DMMG Object with 3 levels
  X range of indices: 0 20, Y range of indices: 0 5
  X range of indices: 0 40, Y range of indices: 0 10
  X range of indices: 0 80, Y range of indices: 0 20

FieldNames:   phi   psi   U   F

KSP Object on finest level:
  type: fgmres
    GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogo
    GMRES: happy breakdown tolerance 1e-30
  maximum iterations=10000, initial guess is zero
  tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
  right preconditioning
```

Monitoring/Viewing DMMG: Example 1

```
DMMG Object with 3 levels
  X range of indices: 0 20, Y range of indices: 0 5
  X range of indices: 0 40, Y range of indices: 0 10
  X range of indices: 0 80, Y range of indices: 0 20

FieldNames:   phi   psi   U   F

KSP Object on finest level:
  type: fgmres
    GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogo
    GMRES: happy breakdown tolerance 1e-30
  maximum iterations=10000, initial guess is zero
  tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
  right preconditioning
PC Object:
  type: mg
    MG: type is full, levels=3 cycles=1, pre-smooths=1, post-smooths=1
```

Monitoring/Viewing DMMG: Example 1

```
KSP Object on finest level:
  type: fgmres
    GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogo
    GMRES: happy breakdown tolerance 1e-30
  maximum iterations=10000, initial guess is zero
  tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
  right preconditioning

PC Object:
  type: mg
    MG: type is full, levels=3 cycles=1, pre-smooths=1, post-smooths=1
  Coarse gride solver -- level 0 -------------------------------
    KSP Object:(mg_coarse_)
      type: preonly
      maximum iterations=1, initial guess is zero
      tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
      left preconditioning
```

Monitoring/Viewing DMMG: Example 1

```
Coarse gride solver -- level 0 -------------------------------
  KSP Object:(mg_coarse_)
    type: preonly
    maximum iterations=1, initial guess is zero
    tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
    left preconditioning
  PC Object:(mg_coarse_)
    type: lu
      LU: out-of-place factorization
        matrix ordering: nd
      LU: tolerance for zero pivot 1e-12
      LU: using Manteuffel shift
        LU nonzeros 4044
    linear system matrix = precond matrix:
    Matrix Object:
      type=aij, rows=100, cols=100
      total: nonzeros=1225, allocated nonzeros=1225
```

Monitoring/Viewing DMMG: Example 1

```
      LU: tolerance for zero pivot 1e-12
      LU: using Manteuffel shift
        LU nonzeros 4044
    linear system matrix = precond matrix:
    Matrix Object:
      type=aij, rows=100, cols=100
      total: nonzeros=1225, allocated nonzeros=1225
        not using I-node routines

Down solver (pre-smoother) on level 1 -------------------------------
   KSP Object:(mg_levels_1_)
     type: gmres
       GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Ort
       GMRES: happy breakdown tolerance 1e-30
     maximum iterations=1
     tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
     left preconditioning
```

Monitoring/Viewing DMMG: Example 1

```
Down solver (pre-smoother) on level 1 ------------------------------
  KSP Object:(mg_levels_1_)
    type: gmres
      GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Ort
      GMRES: happy breakdown tolerance 1e-30
    maximum iterations=1
    tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
    left preconditioning

  PC Object:(mg_levels_1_)
    type: ilu
      ILU: 0 levels of fill
      ILU: max fill ratio allocated 1
      ILU: tolerance for zero pivot 1e-12
           out-of-place factorization
           matrix ordering: natural
           Factored matrix follows
```

Monitoring/Viewing DMMG: Example 2

```
ex19 -dmmg_snes_monitor -dmmg_grid_sequence

        0 SNES Function norm 2.223797239155e-01
        1 SNES Function norm 7.801891853093e-05
        2 SNES Function norm 9.491170165169e-12
      0 SNES Function norm 1.123712372636e-01
      1 SNES Function norm 5.489474836141e-06
      2 SNES Function norm 1.417436966859e-13
    0 SNES Function norm 6.585046383524e-02
    1 SNES Function norm 1.201084857206e-06
    2 SNES Function norm 1.024510825027e-12
```

Options for Computing Jacobians with DMMG

```
fd - finite differences with coloring
ad - automatic differentation with coloring
mf - matrix free
```

```
 -dmmg_jacobian_fd
 -dmmg_jacobian_ad
 -dmmg_jacobian_mf_fd_operator
 -dmmg_jacobian_mf_fd
 -dmmg_jacobian_mf_ad_operator
 -dmmg_jacobian_mf_ad
```

Lag the computation of the Jacobian

```
 -dmmg_jacobian_period <p>
```

Monitoring/Viewing DMMG: Example 3

```
ex19 -dmmg_ksp_monitor -pc_mg_type multiplicative
    0 KSP Residual norm 7.886953101160e-02
      0 KSP Residual norm 2.897287065900e+00
      1 KSP Residual norm 8.758599233278e-01
      2 KSP Residual norm 5.492865211952e-01
        0 KSP Residual norm 7.538992301817e-01
        1 KSP Residual norm 2.416478999283e-01
        2 KSP Residual norm 7.436304547810e-02
          0 KSP Residual norm 7.716244681100e-02
          1 KSP Residual norm 1.264375185297e-17
        0 KSP Residual norm 4.471398243037e-02
        1 KSP Residual norm 9.347059883866e-03
        2 KSP Residual norm 3.081460695647e-03
      0 KSP Residual norm 7.028027964002e-01
      1 KSP Residual norm 2.340423305354e-01
      2 KSP Residual norm 1.276226209783e-01
    1 KSP Residual norm 6.427408754349e-03
```

DMMG - DM Multigrid

What does it do?

- Creates a KSP or SNES object
- Sets the PCType to PCMG
- For each level creates and fills up in the PCMG object

  * the vectors

  * the restriction/interpolation

  * the matrices

How does it work?

DM - Objects that do Just Enough for Multigrid

- Refine themselves
- Create global vectors
- Create appropriate sparse matrices (may be matrix-free)
- Generate interpolation between levels
- Generate coloring of matrix (if using FD or AD Jacobians)
- Generate injection between levels (SNES only)
- Create local (ghosted) vectors (SNESLocal only)
- Communicate between local and global vectors (SNESLocal only)

You can think of them as “containing” a mesh, discretization and ways of generating algebraic objects from them.

Default DMs in PETSc

- DA - logically rectangular meshes in 1, 2 and 3 dimensions

  - automatic generation of sparsity structure of matrix

  - automatic generation of matrix coloring (for automatic Jacobian computation)

  - flexible support for finite difference schemes (you provide)

  - basic interpolation schemes in place (more can be added easily)

- VecPack - collections of rectangular meshes **plus** “extra variables” that are generally coupled to all mesh variables (e.g. design variables).

  - basic interpolation schemes in place (more can be added easily)

  - no automatic generation of matrix sparsity currently

DA Construction

```
DACreate2d(MPI_Comm
           DAPeriodicType DA_NONPERIODIC,DA_XPERIODIC,DA_YPERIODIC,DA_XY
           DAStencilType  DA_STENCIL_STAR,DA_STENCIL_BOX
           int Mx,My
           int Px,Py
           int degrees of freedom per node
           int stencil width
           ...
           &da);
```

DA Operations

Keeping global and local (ghosted) vectors

```
DACreateGlobalVector(da,Vec *g);
DACreateLocalVector(da,Vec *l);
```

Getting work vectors

```
DAGetGlobalVector(da,Vec *g);
DAGetLocalVector(da,Vec *g);
DARestoreGlobalVector(da,Vec *g);
DARestoreLocalVector(da,Vec *g);
```

Moving between global and local vectors

```
DAGlobalToLocalBegin(da,Vec g,ADD_VALUES or INSERT_VALUES,Vec l);
DAGlobalToLocalEnd(da,Vec g,ADD_VALUES or INSERT_VALUES,Vec l);

DALocalToGlobalBegin(da,Vec l,Vec g);
DALocalToGlobalEnd(da,Vec l,Vec g);

DALocalToGlobal(da,Vec l,ADD_VALUES or INSERT_VALUES, Vec g);
```

DA Operations: Example

```
PetscErrorCode FormFunctionMatlab(SNES snes,Vec X,Vec F,void *ptr)
{
  ...
  DAGetLocalVector(da,&localX);

  DAGlobalToLocalBegin(da,X,INSERT_VALUES,localX);
  DAGlobalToLocalEnd(da,X,INSERT_VALUES,localX);

  // compute values in F using ghosted values in localX

  DARestoreLocalVector(da,&localX);
}
```

src/snes/examples/tutorials/ex5.c

Truckloads of other DA operations

```
DAGetInfo(DA da,int *dim,int *Mx,int *My,int *Mz,
                    int *Px,int *Py,int *Pz,
                    int *dof,int *stencil width,
                    DAPeriodicType *,DAStencilType *)
```

Get information about which part of the mesh/vector this process owns

```
DAGetCorners(DA da,int *x,int *y,int *z,int *m,int *n,int *p)
DAGetGhostCorners(DA da,int *x,int *y,int *z,int *m,int *n,int *p)
...
```

Writing a FormFunctionLocal(): 1

Name your fields

```
typedef struct {
  PetscScalar u,v,omega,temp;
} Field;
```

```
 DASetFieldName(DMMGGetDA(dmmg),0,"x-velocity");
 DASetFieldName(DMMGGetDA(dmmg),1,"y-velocity");
 DASetFieldName(DMMGGetDA(dmmg),2,"Omega");
 DASetFieldName(DMMGGetDA(dmmg),3,"temperature");
```

WRITING A FORMFUNCTIONLOCAL(): 2

```
FormFunctionLocal(DALocalInfo *info,Field **x,Field **f,void *ptr)
 {
  AppCtx         *user = (AppCtx*)ptr;
  ...
  for (j=info->ys; j<info->ys+info->ym; j++) {
    for (i=info->xs; i<info->ys+info->xm; i++) {

/*   convective coefficients for upwinding */

vx = x[j][i].u; avx = PetscAbsScalar(vx);
        vxp = .5*(vx+avx); vxm = .5*(vx-avx);
vy = x[j][i].v; avy = PetscAbsScalar(vy);
        vyp = .5*(vy+avy); vym = .5*(vy-avy);
```

Writing a FormFunctionLocal(): 3

```
/* U velocity */
        u           = x[j][i].u;
        uxx         = (2.0*u - x[j][i-1].u - x[j][i+1].u)*hydhx;
        uyy         = (2.0*u - x[j-1][i].u - x[j+1][i].u)*hxdhy;
        f[j][i].u   = uxx + uyy - .5*(x[j+1][i].omega-x[j-1][i].omega)*hx

/* V velocity */
        u           = x[j][i].v;
        uxx         = (2.0*u - x[j][i-1].v - x[j][i+1].v)*hydhx;
        uyy         = (2.0*u - x[j-1][i].v - x[j+1][i].v)*hxdhy;
        f[j][i].v   = uxx + uyy + .5*(x[j][i+1].omega-x[j][i-1].omega)*hy

/* Omega */
        u           = x[j][i].omega;
        uxx         = (2.0*u - x[j][i-1].omega - x[j][i+1].omega)*hydhx;
        uyy         = (2.0*u - x[j-1][i].omega - x[j+1][i].omega)*hxdhy;
f[j][i].omega = uxx + uyy + (vxp*(u - x[j][i-1].omega) + ....
```

DALocalInfo

```
typedef struct {
  int            dim,dof,sw;
  DAPeriodicType pt;
  DAStencilType  st;
  int            mx,my,mz;    /* global number of grid points in each di
  int            xs,ys,zs;    /* starting pointd of this processor, excl
  int            xm,ym,zm;    /* number of grid points on this processor
  int            gxs,gys,gzs;    /* starting point of this processor inc
  int            gxm,gym,gzm;    /* number of grid points on this proces
  DA             da;
} DALocalInfo;
```

Full Approximation Scheme: Playing

Nonlinear multigrid

- smooth (nonlinear Gauss-Seidel) on each level
- restrict/interpolation
- vectors on each level

We already have the infrastructure; lack nonlinear smoother.

Nonlinear Smoother: Playing

Uses Newton to solve for one (or several) unknowns at a time.

FormFunctionLocal() won't cut it. Need to be able to evaluate a

- a single (or several) function coefficients
- a one (or several) dimensional Jacobian

```
FormFunctionLocali(DALocalInfo *info,MatStencil *st,
                   Field **x,PetscScalar *f,void *ptr)
```

```
typedef struct {
  PetscInt k,j,i,c;
} MatStencil;
```

Controling the FAS

```
-dmmg_fas

-dmmg_fas_view
-dmmg_fas_monitor
-dmmg_fas_monitor_all

-dmmg_fas_presmooth its
-dmmg_fas_postmooth its
-dmmg_fas_coarsesmooth its

-dmmg_fas_rtol rtol
-dmmg_fas_atol atol

-dmmg_fas_newton_its its
```

Algebraic Methods Available from PETSc

- ML (part of the Trilinos package our of SNL) (coming soon)

  Uses the PETSc multigrid infrastructure for iteration

  Trilinos is just used to generate the restriction operations and coarser grid matrices

- BoomerAMG (part of the hypre package out of LLNL)

  Currently uses its own algorithms/software for iteration (we hope to change this)

- Prometheus (developed by Mark Adams at Berkeley) (coming soon)

  Uses the PETSc multigrid infrastructure for iteration

ML in PETSc (coming soon)

ML 3.0 uses Smoothed Aggregation.

- Start with piecewise constant interpolation
- Smooth it a few times with the finer grid operator to generate the interpolant
- Generate coarse grid operator via Galerkin $A_{i-1} = RA_iR^T$

```
PCSetType(pc,PCML)  or -pc_type ml
-pc_ml_maxNlevels nmax
-pc_ml_maxCoarseSize Nmax
-pc_ml_CoarsenScheme Uncoupled,Coupled,MIS,METIS
-pc_ml_DampingFactor d
-pc_ml_Threshold rtol
```

BoomerAMG/hypre in PETSc

Preconditioner Generation Options

```
PCSetType(pc,PCHYPRE)  or -pc_type hypre
-pc_hypre_boomeramg_max_levels nmax
-pc_hypre_boomeramg_truncfactor
-pc_hypre_boomeramg_strong_threshold
-pc_hypre_boomeramg_max_row_sum
-pc_hypre_boomeramg_no_CF
-pc_hypre_boomeramg_coarsen_type CLJP,Ruge-Stueben,modifiedRuge-Stueben,
-pc_hypre_boomeramg_measure_type local,global
```

Does not scale well to large numbers of processes (setup time dominates).

Currently costs one extra fine grid matrix copy (we could fix this if it becomes a showstopper for anyone).

BoomerAMG/hypre in PETSc

Preconditioner Iteration Options

```
-pc_hypre_boomeramg_relax_type_all Jacobi,sequential-Gauss-Seidel,
   SOR/Jacobi,backward-SOR/Jacobi,symmetric-SOR/Jacobi,Gaussian-eliminat
-pc_hypre_boomeramg_relax_type_fine
-pc_hypre_boomeramg_relax_type_down
-pc_hypre_boomeramg_relax_type_up
-pc_hypre_boomeramg_relax_weight_all r
-pc_hypre_boomeramg_outer_relax_weight_all r

-pc_hypre_boomeramg_grid_sweeps_down n
-pc_hypre_boomeramg_grid_sweeps_up n
-pc_hypre_boomeramg_grid_sweeps_coarse n

-pc_hypre_boomeramg_tol tol
-pc_hypre_boomeramg_max_iter it
```

PETSc BoomerAMG/hypre Warning

To use BoomerAMG directly as a solver (rather than a preconditioner for a KSP) you must use -ksp_type richardson not -ksp_typre preonly.

Balancing Neumann-Neumann Preconditioner

Iterative substructuring domain decomposition algorithm.

$$A = \sum A^i$$

$$A^i = \begin{pmatrix} A^i_I & A^i_{IB} \\ A^i_{BI} & A^i_{BB} \end{pmatrix}$$

$$S^i = A^i_{BB} - A^i_{BI}(A^i_I)^{-1}A^i_{IB}$$

$$S = \sum S^i$$

Note that applying S requires solving a Dirchlet boundary value problem, A^i_I, on each subdomain.

Neumann-Neumann Preconditioner

Precondition $\sum S^i$ by solving $(S^i)^{-1}$ on each subdomain. Turns out solving

$$S^i x = b$$

is equivalent to solving

$$\begin{pmatrix} A^i_I & A^i_{IB} \\ A^i_{BI} & A^i_{BB} \end{pmatrix} \begin{pmatrix} \cdot \\ x \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix}$$

which is a Neumann problem for each domain.

Neumann-Neumann Preconditioner

Requires unassembled subdomain stiffness matrices.

Generate matrix elements using “local” process numbering not global numbering.

```
MatSetType(mat,MATIS);
MatSetLocalToGlobalMapping(mat,ISLocalToGlobalMapping mapping)
...
MatSetValuesLocal(mat,nrow,local row indices,ncol,local col indices,...
```

PETSc Options for Neumann-Neumann Preconditioner

```
-is_localD_ksp/pc_....
-is_localN_ksp/pc_....
```

Problematic: Interior subdomain Neumann problems are singular.

```
-is_localN_pc_lu/cholesky_shift_nonzero
-is_localN_pc_lu/cholesky_shift_positive_definite
```

Works very well for a small number of processes.

Balancing Neumann-Neumann Preconditioner

Coarse grid "problem" is defined by eliminating null space from "floating" subdomains.

Linear system has $\sum dim(Null(S_i))$.

```
-pc_nn_turn_off_first_balancing
     (this skips the first coarse grid solve in the preconditioner)
-pc_nn_turn_off_second_balancing
     (this skips the second coarse grid solve in the preconditioner)
-pc_is_damp_fixed <fact>
-pc_is_remove_nullspace_fixed
-pc_is_set_damping_factor_floating <fact>
-pc_is_not_damp_floating
-pc_is_not_remove_nullspace_floating
```

Currently, :-), only supports null space of the constant functions. Others have done prototypes for nontrivial problems (Olof's group).

SUMMARY

- Composition of preconditioners - PCCOMPOSITE
- Overlapping Schwarz methods - PCASM, PCFIELDSPLIT
- Multigrid methods
  - Background
  - Low level interface - PCMG
  - Simple interface - DMMG, DA
  - Nonlinear methods (FAS)
  - algebraic methods - PCHYPRE, PCML, PCPROMETHEUS
- Balancing Neumann-Neumann algorithm - PCNN

We appreciate your concrete feedback!

petsc-maintmcs.anl.gov \\