PROGRAMMING OF MULTIGRID METHODS

LONG CHEN

In this note, we explain the implementation details of multigrid methods. We adopt the approach based on space decomposition and the subspace correction framework; see *Subspace Correction Method and Auxiliary Space Method*. The matrix formulation arises naturally once the basis representations of the functions are introduced.

To distinguish functions from vectors, we use boldface letters for the matrix representation of an operator or the vector representation of a function.

1. Two Level Methods

We use a two-level method to illustrate how to realize operators in matrix form. The space decomposition is

$$\mathbb{V} = \mathbb{V}_1 + \mathbb{V}_2 \qquad \text{with} \qquad \mathbb{V}_1 \subset \mathbb{V}_2 = \mathbb{V}.$$

We call $\mathbb{V}$ the fine space and $\mathbb{V}_1$ the coarse space, since $\mathbb{V}_1$ is usually based on a coarse mesh.

Recall that the PSC method for this two-level decomposition, in operator form, is:

(1) $r = f - Au^k$;
(2) $e = I_1 R_1 I_1^\intercal\, r \;+\; I_2 R_2 I_2^\intercal\, r$;
(3) $u^{k+1} = u^k + e$.

The matrix forms of steps 1 and 3 are straightforward. We therefore focus on the realization of step 2.

Since $\mathbb{V}_2 = \mathbb{V}$, we have $I_2 = I_2^\intercal = I$. The smoother R_2 may be chosen as the weighted Jacobi iteration $\boldsymbol{R}_2 = \alpha \boldsymbol{D}^{-1}$ (with $\alpha = 0.5$ as the recommended default) or the symmetric Gauss–Seidel iteration $\overline{\boldsymbol{R}}_2$ with $\boldsymbol{R}_2 = (\boldsymbol{D} + \boldsymbol{L})^{-1}$.

The transformation to the coarse space $\mathbb{V}_1$ is less direct. Three operators must be realized in matrix form: I_1, R_1, and $I_1^\intercal$.

The prolongation matrix. We first discuss the operator $I_1 = I_1^2 : \mathbb{V}_1 \to \mathbb{V}_2$. By definition, it is the natural inclusion $\mathbb{V}_1 \hookrightarrow \mathbb{V}_2$, i.e., a function $u_1 \in \mathbb{V}_1$ is viewed as a function in $\mathbb{V}_2$ since $\mathbb{V}_1 \subset \mathbb{V}_2$. Thus, as an operator, I_1 is the identity. However, its matrix representation is *not* the identity matrix because $\mathbb{V}_1$ and $\mathbb{V}_2$ have different bases. We use a 1D two-level grid in Figure 1 to illustrate this point.

In this example, the matrix $\boldsymbol{I}_1^2 : \mathbb{R}^3 \to \mathbb{R}^5$ maps a shorter vector to a longer one and is called the *prolongation* matrix. We determine this matrix using the following facts:

(1) $\boldsymbol{u}_1$ and $\boldsymbol{u}_2 = \boldsymbol{I}_1^2 \boldsymbol{u}_1$ represent the same function in $\mathbb{V}_2$;
(2) a function in $\mathbb{V}_2$ is uniquely determined by its values at the grid points.

On nodes shared by both the fine and coarse grids,

$$\boldsymbol{u}_2(1) = \boldsymbol{u}_1(1), \qquad \boldsymbol{u}_2(3) = \boldsymbol{u}_1(2), \qquad \boldsymbol{u}_2(5) = \boldsymbol{u}_1(3).$$

Date: Latest update: November 17, 2025.

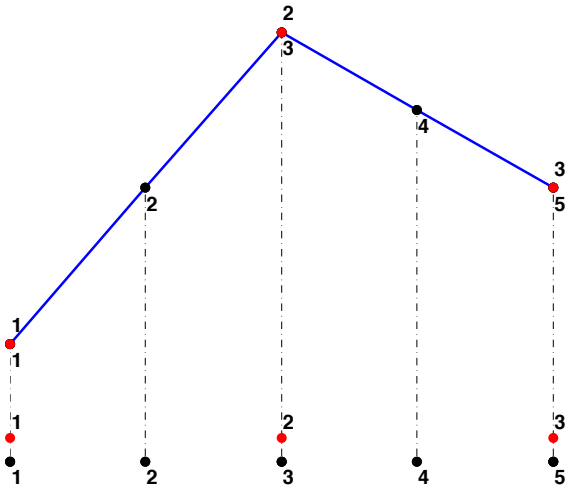

FIGURE 1. Prolongation

On nodes that exist only in the fine grid, the values are obtained by evaluating the coarse-grid function. Since we use linear elements,

$$\boldsymbol{u}_2(2) = \tfrac{1}{2}(\boldsymbol{u}_1(1) + \boldsymbol{u}_1(2)), \qquad \boldsymbol{u}_2(4) = \tfrac{1}{2}(\boldsymbol{u}_1(2) + \boldsymbol{u}_1(3)).$$

Thus the prolongation matrix $\boldsymbol{I}_1^2 \in \mathbb{R}^{5\times 3}$ is

$$\boldsymbol{I}_1^2 = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}.$$

One can also obtain the prolongation matrix by looking at the representation of the coarse grid basis function in the fine grid.

To define the prolongation matrix, we need to know the correspondences of the index of nodes between two grids. Different index mapping will give different prolongation matrix. A better hierarchical index of the fine grid nodes is [1 4 2 5 3], for which the prolongation matrix is

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1/2 & 1/2 & 0 \\ 0 & 1/2 & 1/2 \end{bmatrix}.$$

The identity rows can be omitted, which saves the computational cost of forming $\boldsymbol{I}_1^2\boldsymbol{x}$ when applying the prolongation operator.

The construction of the prolongation matrix can be easily generalized to higher dimensions for linear elements. The only information needed is the index mapping between coarse-grid points and fine-grid points. We classify the fine-grid nodes into two groups:

- $\mathcal{C}$: nodes that belong to both the fine and coarse grids;
- $\mathcal{F}$: nodes that belong only to the fine grid.

For the group $\mathcal{F}$, we may use the `HB` (hierarchical basis) matrix, where `HB(:,2:3)` contains the two parent nodes of the child node `HB(:,1)`. Here `HB(:,1)` stores fine-level indices, and `HB(:,2:3)` stores the corresponding coarse-level indices. The interpolation at nodes in $\mathcal{F}$ is then realized as

```
uf(HB(1:end,1)) = (uc(HB(1:end,2)) + uc(HB(1:end,3)))/2;
```

For the group $\mathcal{C}$, although these nodes lie in both coarse and fine grids, their indices may differ. Thus we need an index map `coarseNodeFineIdx` for nodes in $\mathcal{C}$. In the example of Fig. 1, `coarseNodeFineIdx = [1 3 5]`. The interpolation for these nodes is simply the identity:

```
uf(coarseNodeFineIdx) = uc;
```

The prolongation matrix can be constructed by the following self-explained code:

```
ii = [coarseNodeFineIdx; HB(:,1); HB(:,1)];
jj = [coarseNode;        HB(:,2); HB(:,3)];
ss = [ones(nCoarseNode,1); 0.5*ones(nFineNode,1); 0.5*ones(nFineNode,1)];
Pro = sparse(ii,jj,ss,nFineNode,nCoarseNode);
```

The restriction matrix. How do we compute $I_1^\intercal = Q_1$? To compute the L^2 projection, we would need to invert the mass matrix, which is not cheap. Fortunately, we are not actually computing the L^2 projection of a function. Instead, we are dealing with a functional. Recall the definition

$$(Q_1 r, u_1) = (r, u_1) = (r, I_1 u_1). \tag{1}$$

The quantity $Q_1 r$ is simply the restriction of the action of the dual $r \in \mathbb{V}_2'$ to elements in $\mathbb{V}_1$ only. It is better to write this operator as $I_2^1 : \mathbb{V}_2' \to \mathbb{V}_1'$. Note that $\mathbb{V}_1 \subset \mathbb{V}_2$ implies $\mathbb{V}_2' \subset \mathbb{V}_1'$. Thus the operator I_2^1 is also a natural inclusion on the dual spaces. Again, r and $I_2^1 r$ have different vector representations.

The matrix form of (1) is

$$(\boldsymbol{I}_2^1 \boldsymbol{r})^\intercal \boldsymbol{u}_1 = \boldsymbol{r}^\intercal \boldsymbol{I}_1^2 \boldsymbol{u}_1, \tag{2}$$

which implies

$$\boldsymbol{I}_2^1 = (\boldsymbol{I}_1^2)^\intercal.$$

If we have the prolongation matrix `Pro` formed explicitly, the restriction matrix is simply its transpose, i.e., `Res = Pro'`.

Exercise 1.1. Use `HB` and `coarseNodeFineIdx` to code the restriction without forming the matrix.

Remark 1.2. Interpolation and restriction matrices must be altered at boundary points or at neighbors of boundary points to impose the correct boundary condition.

Coarse grid matrix and smoothers. The last components we need are the smoother R_1 and the coarse-grid operator A_1. If we know a priori the PDE and its discretization, we can easily code a smoother in the coarse space. For example, for the 5-point stencil discretization of the Poisson equation, one step of Gauss–Seidel iteration can be implemented using `for` loops:

```
for i = 2:N-1
    for j = 2:N-1
        u(i,j) = (b(i,j) + (u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))) / 4;
```

```
    end
end
```

For a general operator A, to implement the Gauss–Seidel method, we need the matrix $\boldsymbol{A}_1$. We can assemble this matrix if the coarse grid is available, but there are several reasons not to take this approach. First, assembling the matrix is time-consuming; this is one of the common criticisms of finite element methods compared with finite difference methods. Second, assembling requires mesh and PDE information, making the algorithm problem dependent. Third—and most important—we have a better alternative.

Recall that A_1 is the restriction of A to the subspace $\mathbb{V}_1$. Namely,

$$(A_1u_1, v_1) := (Au_1, v_1) = (AI_1u_1, I_1v_1) = (I_1^\intercal AI_1u_1, v_1), \tag{3}$$

which implies $A_1 = I_1^\intercal AI_1$. In matrix form,

$$\boldsymbol{A}_1 = \boldsymbol{I}_2^1\boldsymbol{A}_2\boldsymbol{I}_1^2 = \texttt{Res} * A * \texttt{Pro}.$$

Thus we can form the coarse-grid matrix by a triple product of matrices. Because the derivation is based on the bilinear form (3), this construction is often referred to as the *Galerkin method* or the *variational method*.

2. SSC AND MULTIGRID METHOD

In this section, we discuss the implementation of the successive subspace correction (SSC) method when the subspaces are nested. Let $\mathbb{V} = \sum_{i=1}^J \mathbb{V}_i$ be a space decomposition into nested subspaces, i.e.,

$$\mathbb{V}_1 \subset \mathbb{V}_2 \subset \cdots \subset \mathbb{V}_J = \mathbb{V}.$$

Recall that the operator form of the SSC method is

```
function e = SSC(r)
% Solve the residual equation Ae = r by the SSC method
e = 0;  rnew = r;
for i = J:-1:1
    ri   = Ii' * rnew;   % restrict the residual to subspace
    ei   = Ri  * ri;     % solve the residual equation in the subspace
    e    = e + Ii * ei;  % prolongate the correction
    rnew = r - A * e;    % update the residual
end
```

We use the `for` loop from `J:-1:1` to reflect the ordering from fine to coarse. The operators $I_i^\intercal = Q_i : \mathbb{V} \to \mathbb{V}_i$ and $I_i : \mathbb{V}_i \to \mathbb{V}$ are defined with respect to the finest space. Since N_i decays geometrically, the number of levels satisfies $J = \mathcal{O}(\log N)$. At each level, the prolongation matrix has size $N \times N_i$, so the cost per level is $\mathcal{O}(N)$. Therefore, the total cost of this direct implementation of SSC is $\mathcal{O}(N \log N)$.

When the subspaces are nested, we do not need to return to the finest space at every step. Suppose $r_i = I_i^\intercal(r - Ae_{\text{old}})$ is known in the subspace $\mathbb{V}_i$, and the correction e_i is used to update $e_{\text{new}} = e_{\text{old}} + e_i$. We can compute r_{i-1} using

$$\begin{aligned} r_{i-1} &= Q_{i-1}(r - Ae_{\text{new}}) \\ &= Q_{i-1}Q_i(r - Ae_{\text{old}} - Ae_i) \\ &= Q_{i-1}\big(r_i - Q_iAQ_i^\intercal e_i\big) \\ &= Q_{i-1}(r_i - A_ie_i). \end{aligned}$$

In the second step, we use the nested property $\mathbb{V}_{i-1} \subset \mathbb{V}_i$, which gives $Q_{i-1} = Q_{i-1}Q_i$.

The correction step can also be performed accumulatively. Rewrite the correction as

$$e = e_J + I_{J-1}e_{J-1} + \cdots + I_1 e_1.$$

This can be computed by the loop

$$e_i = e_i + I_{i-1}^i e_{i-1}, \qquad i = 2 : J.$$

Therefore, only the prolongation and restriction operators between consecutive levels are needed. Since the work on level i is $\mathcal{O}(N_i)$ and the refinement satisfies $N_i = \gamma N_{i-1}$ for some $\gamma > 1$, the total computational cost is $\mathcal{O}(N)$.

From this point of view, SSC on a nested space decomposition naturally produces a V-cycle multigrid method. We summarize the algorithm below. We use the notation e_i and r_i to emphasize that, on each level, we are solving the residual equation $A_i e_i = r_i$. We assume that all transfer operators and discretization matrices have already been constructed using the methods described in the previous section.

```
function e = Vcycle(r,J)
ri = cell(J,1);  ei = cell(J,1);
ri{J} = r;
for i = J:-1:2
    ei{i} = R{i}*ri{i};                          % pre-smoothing
    ri{i-1} = Res{i-1}*(ri{i} - Ai{i}*ei{i});    % update + restrict
end
ei{1} = Ai{1}\ri{1};                              % exact coarse solve
for i = 2:J
    ei{i} = ei{i} + Pro{i}*ei{i-1};               % prolongate + correct
    ei{i} = ei{i} + R{i}'*(ri{i} - Ai{i}*ei{i});  % post-smoothing
end
e = ei{J};
```

In the second loop (the upward sweep), we add a post-smoothing step using the transpose R_i' of the pre-smoothing operator. For example, if $R_i = (D_i + L_i)^{-1}$ is one step of forward Gauss–Seidel, then $R_i' = (D_i + U_i)^{-1}$ is backward Gauss–Seidel. This choice makes the overall multigrid iterator B symmetric and therefore suitable as a preconditioner for the Preconditioned Conjugate Gradient (PCG) method.

The function call `e = Vcycle(r,J)` indicates that the V-cycle is used to solve the residual equation `A*e = r` and is applied as an iterator in the residual-correction scheme `u = u + Vcycle(f - A*u, J)`.

Because the V-cycle is linear, we may also formulate a direct-update version of the multigrid method as `u = Vcycle(u, f, J)`; see the next section.

3. Simplified Implementation for Finite Difference Methods

We discuss the implementation of the main components, focusing in particular on the prolongation and restriction operators for multigrid methods on uniform grids.

To evaluate the residual, we need to compute the matrix–vector product `A*u`, whose implementation is discussed in *Programming of Finite Difference Methods*. A typical Gauss–Seidel smoother is also treated in detail there. In what follows, we focus on the transfer operators: prolongation and restriction.

We consider the 1-D case first. A coarse grid is refined by adding the middle points. It is easy to figure out the index map from coarse to fine: $i \to 2i - 1$. We use the linear interpolation to construct the prolongation. The matrix-free implementation will be

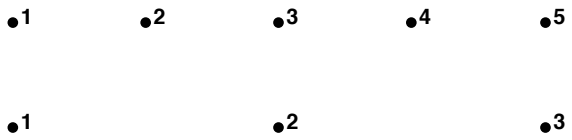

FIGURE 2. Indices of a coarse and fine grids

```
N = length(u);
uf = zeros(2*N-1,0);
j = 2:N-1; % interior nodes of the coarse grid
jf = 2*j-1; % index of coarse nodes in the fine grid
uf(jf) = u(j);
uf(jf-1) = uf(jf-1) + 0.5*u(j);
uf(jf+1) = uf(jf+1) + 0.5*u(j);
```

One can also construct a sparse matrix `Ix`

```
N = length(u); % number of points
Nf = 2*N - 1; % number of points in the fine grid
j = 2:N-1;     % interiori points
jf = 2*j-1; % index of coarse nodes in the fine grid
ii = [jf jf-1 jf+1];
jj = [j j j];
ss = [ones(1,N-2) 0.5*ones(1,N-2) 0.5*ones(1,N-2)];
Ix = sparse(ii,jj,ss,Nf,N);
```

Then `uf = Ix*u` will produce the desired result. One advantage of using the matrix form of the prolongation operator is that the restriction operator can be taken as the transpose of the prolongation matrix (with a possible scaling related to h, depending on the scaling used in the 5-point stencil).

When moving to two dimensions, we can still work with subscripts to implement a matrix-free version. Bilinear interpolation is used to evaluate function values at fine-grid points. In stencil notation, the prolongation for fine nodes on horizontal lines, vertical lines, and cell centers can be written as

$$(0.5, *, 0.5), \qquad \begin{pmatrix} 0.5 \\ * \\ 0.5 \end{pmatrix}, \qquad \begin{pmatrix} 0.25 & & 0.25 \\ & * & \\ 0.25 & & 0.25 \end{pmatrix}.$$

Here $*$ indicates the position of the fine-grid point.

A more elegant method uses the tensor-product structure:

```
uf = Ix*u*Ix';
```

Recall that the unknown vector `u(1:n,1:n)` is stored as a matrix. The left product `uftemp = Ix*u` prolongates the function values along the column direction, so `uftemp` has size $N_f \times N$. The right product `uftemp*Ix'` then prolongates along the row direction. One may choose different prolongations in each direction if the mesh sizes are non-uniform or anisotropic.

For a d-dimensional grid, a coarse-grid point has $3^d - 1$ neighboring fine points. Working directly with subscripts becomes tedious, while the tensor-product matrix formulation

remains efficient by using the permutation trick discussed in *Programming of Finite Difference Methods*.

Due to the matrix-free implementation, the direct-update form of the iteration is preferable for finite difference methods. For example, the Gauss–Seidel smoother is better coded as `u = GS(u,f)`. The direct-update form of the V-cycle is sketched below. The matrix–vector product and the exact solver on the coarsest level can be implemented either in a matrix-free manner or by using the tensor-product structure.

```
function u = MG(u,f,J,mu)
%% Direct update form of Multigrid Method
if J == 1  % coarsest level: exact solve
    u = A(J)\f;
end
% Presmoothing
for i = 1:mu
    u = R(u,f,J);
end
% Restriction
rc = Res(f-A(J)*u);
% Coarse grid correction
ec = MG(0,rc, J-1,mu);
if W-cycle
    ec = MG(ec,rc,J-1,mu);  % W-cycle
end
% Prolongation
u = u + Pro(ec);
% Postsmoothing
for i = 1:mu
    u = R'(u,f,J);
end
```