PROGRAMMING OF MULTIGRID METHODS

LONG CHEN

In this note, we explain the implementation detail of multigrid methods. We will use the approach by space decomposition and subspace correction method; see *Chapter: Subspace Correction Method and Auxiliary Space Method*. The matrix formulation will be obtained naturally, when the functions' basis representation is inserted. We also include a simplified implementation of multigrid methods for finite difference methods. To distinguish functions and vectors, we use boldface letters for a matrix representation of an operator or a vector representation of a function.

1. Two Level Methods and Transfer Operators

We use a two-level method to illustrate how to realize operators by matrices. The space decomposition is

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

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

Recall that the PSC 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 form of step 1 and 3 is trivial. We only discuss the realization of step 2.

Since $\mathbb{V}_2 = \mathbb{V}$, $I_2 = I_2^\intercal = I$. The solver R_2 can be chosen as the weighted Jacobi iteration $\boldsymbol{R}_2 = \omega \boldsymbol{D}^{-1}$ ($\omega = 0.5$ is recommended as the default choice) or the symmetric Gauss-Seidel iteration $\overline{\boldsymbol{R}}_2$ with $\boldsymbol{R}_2 = (\boldsymbol{D} + \boldsymbol{L})^{-1}$.

The transformation to the coarse $\mathbb{V}_1$ is not easy. There are three operators to realize: I_1, R_1, and $I_1^\intercal$.

The prolongation matrix. Let us first discuss the operator $I_1 = I_1^2 : \mathbb{V}_1 \to \mathbb{V}_2$. By the definition, it is the natural inclusion $\mathbb{V}_1 \hookrightarrow \mathbb{V}$ i.e. treat a function $u_1 \in \mathbb{V}_1$ as a function in $\mathbb{V}_2$ since $\mathbb{V}_1 \subset \mathbb{V}_2$. So it is the identity operator. But the matrix representation is not the identity matrix since different bases of $\mathbb{V}_1$ and $\mathbb{V}_2$ are used! We use a 1-D two level grids in Figure 1 to illustrate the difference.

In this example $\boldsymbol{I}_1^2 : \mathbb{R}^3 \to \mathbb{R}^5$ will map a shorter vector to a longer one and thus called the *prolongation* matrix. We determine this matrix by the following two 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 the values at the grid points.

For nodes in both fine grids and coarse grids,

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

Date: Latest update: November 22, 2017.

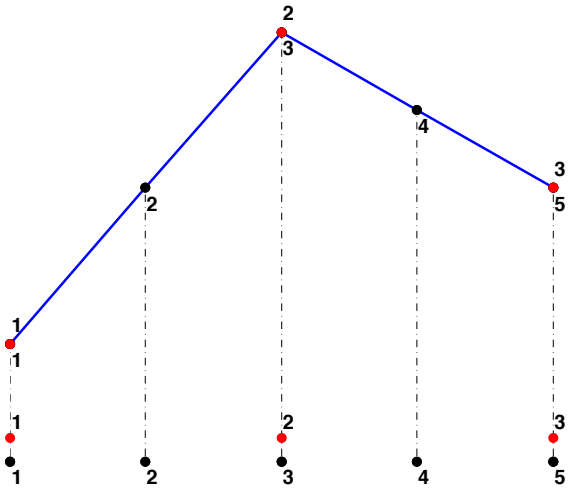

FIGURE 1. Prolongation

For the nodes only existing in the fine grids, by (1), values at these nodes can be evaluated in the coarse grids. Since we are using the linear element, we get

$$\boldsymbol{u}_2(2) = (\boldsymbol{u}_1(1) + \boldsymbol{u}_1(2))/2, \quad \boldsymbol{u}_2(4) = (\boldsymbol{u}_1(3) + \boldsymbol{u}_1(5))/2.$$

In matrix form, $\boldsymbol{I}_1^2 \in \mathbb{R}^{5\times 3}$ can be written as

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

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 presentness of the identity matrix can save the computational time of $\boldsymbol{I}_1^2\boldsymbol{x}$.

The construction of the prolongation matrix can be easily generalized to high dimensions for the linear element. The information needed is the index map between coarse grid points and fine grids points. We classify the grid points in the fine grid into two groups:

- $\mathcal{C}$: the points in both fine and coarse grids
- $\mathcal{F}$: the points in the fine grid only.

For group $\mathcal{F}$, we can use `HB` (hierarchical basis) matrix with `HB(:,2:3)` being two parent nodes of the node `HB(:,1)`. Note that `HB(:,1)` is the index in the fine level while

`HB(:,2:3)` are in the coarse level. Then the interpolation at the grids points in $\mathcal{F}$ can be realized

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

For group $\mathcal{C}$, although those grid points are in both coarse and fine grids, their indices could be different. For example, in Fig 1, the 3-rd point in the fine grid is the 2-nd one in the coarse grid. Therefore we need an index mapping, say `coarseNodeFineIdx`, for points in group $\mathcal{C}$. In the example in Fig 1, `coarseNodeFineIdx = [1 3 5]`. The interpolation for this group of points is simply the identity

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

Using `HB` and `coarseNodeFineIdx`, the prolongation matrix do not need to be formed explicitly. On the other hand, if needed, the prolongation matrix can be easily formed 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 to compute $I_1^{\intercal} = Q_1$? To compute the L^2 projection, we need to invert the mass matrix which is not cheap. Fortunately, we are not really computing the L^2 projection of a function. Instead we are dealing with a functional! Recall the definition

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

$Q_1 r$ is simply to restrict the action of the dual $r \in \mathbb{V}_2'$ to the elements in $\mathbb{V}_1$ only. It is better to write as $I_2^1 : \mathbb{V}_2' \to \mathbb{V}_1'$ and call it restriction. Note that $\mathbb{V}_1 \subset \mathbb{V}_2$ implies that $\mathbb{V}_2' \subset \mathbb{V}_1'$. So the operator I_2^1 is also a natural inclusion of the functional. Again r and $I_2^1 r$ will 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 will be 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 neighbors of boundary points to imposing the correct boundary condition.

The problem matrix and Smoothers in the coarse space. The last component is the smother R_1 and A_1. If we know a priori the information of the PDE and the discretization, we can easily code one in the coarse space. For example, for the 5-point stencil discretization of 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, if we want to choose more accurate local subspace solver, say Gauss-Seidel method, we need to know the matrix $\boldsymbol{A}_1$. Of course we can assemble one if we have the coarse grid. But there are several reasons to abandon this approach. First, the assembling is time consuming. Indeed this is one of the criticism of finite element methods comparing with finite difference methods. Second it requires the information of the mesh and the PDE. Then it will be problem dependent. Third, we have a better way to do it.

Recall that the operator A_1 is just the restriction of A to the space $\mathbb{V}_1$. 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 $A_1 = I_1^\intercal AI_1$ and in the matrix form

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

So we can apply a triple product to form the matrix on the coarse grid. Due to the bilinear form (3) used in the derivation, this approach is often referred as *the Galerkin method* or *the variational method*.

2. SSC AND MULTIGRID METHOD

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

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

Denoted by $N_i = \dim \mathbb{V}_i$ and in practice $N_i = \gamma N_{i-1}$ for a factor $\gamma > 1$. For example, for spaces based on a sequence of nested meshes in $\mathbb{R}^d$, the factor $\gamma \approx 2^d$.

Recall that the operator formation of SSC method is

```
function e = SSC(r)
% Solve the residual equation Ae = r by 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 subspace
    e = e + Ii*ei;   % prolongate the correction to the big space
    rnew = r - A*e;  % update residual
end
```

Here we use the `for` loop from `J:-1:1` to reflect to 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 related to the finest space. As N_i is geometrically decay, the number of level $J = \mathcal{O}(\log N)$. At each level, the prolongation matrix is of size $N \times N_i$ and thus the operation cost at each level is $\mathcal{O}(N)$. The total cost of the direct implementation of SSC is thus $\mathcal{O}(N \log N)$.

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

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

Here in the second step, we make use of the nested property $\mathbb{V}_{i-1} \subset \mathbb{V}_i$ to write $Q_{i-1} = Q_{i-1}Q_i$. Similarly the correction step can be also done accumulatively. Let us rewrite the correction as

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

The correction can be computed by the loop

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

Therefore only the prolongation and restriction operators between consecutive levels are needed. The cost at each level is reduced to $\mathcal{O}(N_i)$ and the total cost is $\mathcal{O}(N)$.

From this point of view, SSC on a nested space decomposition will result in a V-cycle multigrid method. We summarize the algorithm below. We use notation e_i, r_i to emphasize that in each level we are solving the residual equation $A_i e_i = r_i$ and assume the transfer operators and discretization matrices have already been computed using the method discussed 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 and restrict residual
end
ei{1} = Ai{1}\ri{1}; % exact solver in the coarsest level
for i = 2:J
    ei{i} = ei{i} + Pro{i}*ei{i-1};  % prolongate and correct
    ei{i} = ei{i} + R{i}'*(ri{i}-Ai{i}*ei{i}); % post-smoothing
end
e = ei{J};
```

In the second loop (/) part, we add a post-smoothing step and choose R_i' as the smoother which is the transpose of the pre-smoothing operator. For example, if $R_i = (D_i + L_i)^{-1}$ is the forward Gauss-Seidel method, then the post-smoothing is backward Gauss-Seidel $(D_i + U_i)^{-1}$. This choice will make the corresponding iterator B symmetric and thus can be used as a preconditioner in Preconditioned Conjugate Gradient (PCG) methods.

The function `e = Vcycle(r, ...)` suggests that the mg V-cycle is used to solve the residual equation `Ae = r` and will be used as an iterator in the residual-correction method `u = u + Vcycle(f-A*u,J)`.

Due to the linearity of the iteration, we can also formulate a direct update form of multigrid method `u = Vcycle(u,f,J)`; see the next section.

3. Simplified Implementation for Finite Difference Methods

We discuss implementation of main components, especially the prolongation and restriction operator, for multigrid methods on uniform grids.

In order to evaluate the residual, we need to compute the matrix-vector product `Au` which has been discussed in *Chapter: Programming of Finite Difference Methods*. A typical Gauss-Seidel smoother is also discussed in detail there. We now discuss 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

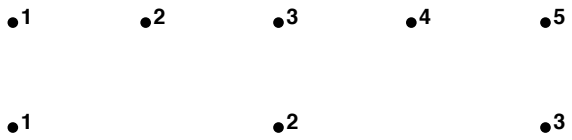

FIGURE 2. Indices of a coarse and fine grids

```
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 simply 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 move to 2-D, we can still use subscript to implement a matrix-free version. We will use the bilinear interpolation to get function values at fine grids. In stencil notation, the prolongation for fine nodes on horizontal lines, on vertical lines, and in the center of coarse cells can be summarized as

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

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

A more elegant way using the tensor product structure is

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

Here recall that the unknown vector `u(1:n,1:n)` is stored as a matrix. The left product `uftemp = Ix*u` will prolongate the function value along the column direction and thus `uftemp` is of size $N_f \times N$. The right product `uftemp*Ix'` will prolongate the function value along the row direction. One can chose different prolongation for different directions if the mesh size is non-uniform in each direction.

For a d-dimensional grid, a coarse grid point will have $3^d - 1$ neighboring points in the fine grid. Working on the subscript system is more tedious while the tensor product matrix version still works efficiently by using the permutation trick we mentioned in *Chapter: Programming of Finite Difference Methods*.

Due to the matrix-free implementation, for finite difference methods, the direct update form of iteration is preferable. For example, the G-S smoother is better coded as `u = GS(u,f)`. The direct update form of Vcycle is sketched below. The matrix-vector product and the exact solver in the coarsest level can be implemented in matrix-free or tensor product way.

```
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
```

4. Algebraic Multigrid Methods

The multigrid methods discussed in the previous sections depends heavily on the geometry of the underlying meshes and therefore called geometric multigrid methods. In most applications, the grid could be totally unstructured and a hierarchical structure is not available. In some cases, only the matrix is given without any grid information. It would be desirable to still solve the algebraic equation using the idea of multigrid.

Looking the procedure carefully, the hierarchical structure of grids is used to construct the transfer operators. After that, the matrix equation in the coarse grid can be computed by triple product and the smoother can be algebraically taking as G-S `tril(A)` or weighted Jacobi `omega*diag(A)`. Two essential ingredients are needed to construct the prolongation operator from a coarse grid to a fine grid

(1) Index map from coarse nodes to fine nodes.
(2) Interpolation of the fine variables.

Let us revisit these two ingredients in an algebraic way. Suppose A is an $N \times N$ matrix. The fine nodes are the index set $\mathcal{V} = \{1, 2, \cdots, N\}$. From the given matrix A, we could construct a weighted graph $\mathcal{G} = \mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$. The edge $[i, j]$ exists if $a_{ij} \neq 0$. As the matrix is symmetric, the graph $\mathcal{G}$ is undirected.

Coarsening. Recall that the node in the fine level can be classified into $\mathcal{C}$ and $\mathcal{F}$. Now if only matrix is given, a node will be understood as an abstract point. No coordinate is associated to it. The fine nodes are the index set $\mathcal{V} = \{1, 2, \cdots, N\}$. A subset $\mathcal{C}$ of $\mathcal{N}$ will

be identified as the nodes of a 'coarse grid' and the rest is $\mathcal{F}$, i.e. $\mathcal{N} = \mathcal{C} \cup \mathcal{F}$. In addition, for any $i \in \mathcal{F}$, the neighboring 'coarse nodes' $\mathcal{J}(i) \subset \mathcal{C}$ should be found. In hierarchical meshes case, $\mathcal{J}(i)$ is simply `HB` array which only contains two coarse nodes. In summary we need to pick up $\mathcal{C}$ and construct $\mathcal{J}(i)$ for all $i \in \mathcal{F}$.