NGsolve::Take the rough with the smooth

Multilevel methods

Jay Gopalakrishnan

Portland State University

Winter 2015

Download code (works only for version 6.0) for these notes from here.

Contents

Multigrid idea

A Multigrid iteration is an iteration that reduces error using a hierarchy of successively refined multilevel grids:

- The error has rough components and smooth components.
- Rough error components must be damped on fine grids.
  - Need smoothers that reduce the high frequencies of the error.
- Smooth error components may be corrected on coarser grids.
  - Coarser grids must be sent projection of errors.

We typically do not know the error. But to understand the ideas, we now consider a case where the exact solution $u = 0$, so that its approximating iterates u^n coincide with the error $u^n - 0$.

Classical point Jacobi iteration

If A is a symmetric positive definite matrix, and $D = diag(A)$, then

$$u^{n+1} = u^n + \omega D^{-1}(f - Au^n), \qquad n = 1, 2, \ldots$$

is the classical scaled Jacobi iteration.

- For what scaling factor ω does it converge to $A^{-1}f$?
  (See e.g., Theorem 50 of MG diary from a previous course.)
- We are not interested in convergence of Jacobi iterations, but rather in its smoothing properties.
- Take an look at the implementation in `smoothproject.cpp` with $f = 0$ (so the exact solution $u = A^{-1}f = 0$).

A simple implementation

```
class NumProcSmoothProject : public NumProc   {
  // :
  // :
  double Jacobi(const BaseSparseMatrix & A,
                const BaseSparseMatrix & B,
                BaseVector & u, const BaseVector & f) {

    auto r = u.CreateVector();
    r = A * u;
    double anormu2=InnerProduct(u,r);// compute || u ||_A^2
    r -= f ;                          // r =  A*u - f
    u -= B * r;                       // u = u + B*(f - A*u)
    return anormu2;
  }
```

- The assembled matrix A is got from a Laplace bilinear form.
- The matrix $B = \omega D^{-1}$ is made as private data in `NumProcSmoothProject::SetInitLevels()`.

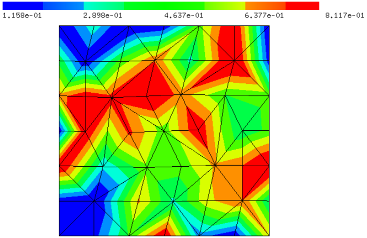
Run the pde file

Compile using `make`. Load `smoothproject.pde`.

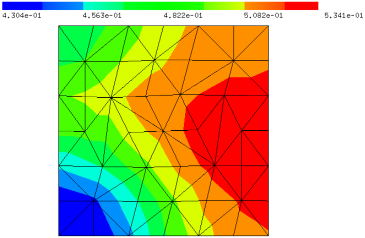
```
#  :                        FILE: smoothproject.pde
#  :

shared = libmg

fespace v -type=nodal     # lowest order space

bilinearform a -fespace=v -symmetric
laplace  (1.0)
mass     (1.0)

gridfunction u -fespace=v -nested

numproc smoothproject nps -gridfunction=u -bilinearform=a -fespace=v
                          -numiters=100 -omega=0.1 -random -demo=1
```

Make sure the `-demo=1` flag is set and press Solve twice.

Smoothing effect of Jacobi iterations



Random initial iterate u^1



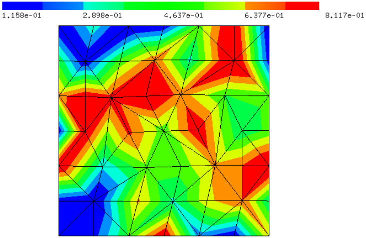
Jacobi iterate u^{100}

- In addition to observing that $\|u^n\|_A \to 0$, we also observe that

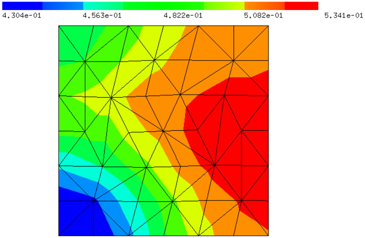
$$\|(I - P_0)Ku^n\|_A \to 0$$

where $K = I - \omega D^{-1}A$ and P_0 is the coarse "elliptic projection" (the projection in A-inner product) implemented in `NumProcSmoothProject::EllipticProjection`.

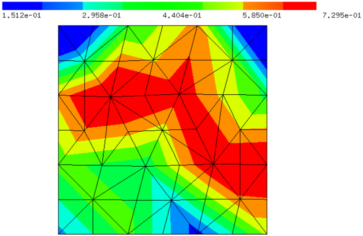
Smoothing effect of Jacobi iterations



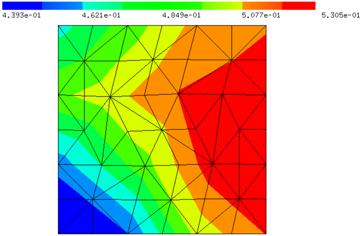
Random initial iterate u^1



Jacobi iterate u^{100}

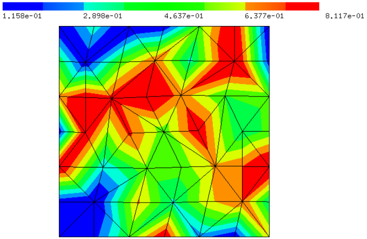


Coarse projection of u^1

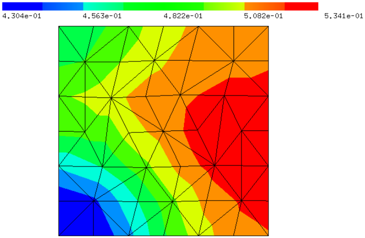


Coarse projection of u^{100}

Smoothing effect of Jacobi iterations



Random initial iterate u^1

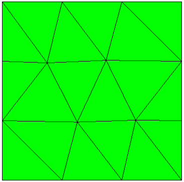


Jacobi iterate u^{100}

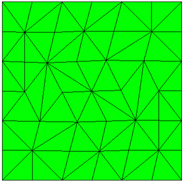
Conclusions from this demo:

- Jacobi iterations smooth the error.
- The smoothed iterates are well-representable on the coarser grid.
- Why not project to the next coarser grid and iterate there? $\rightarrow$ *MG*!

Prolongation and restriction



Lagrange space V_0
Basis $\{\phi_j^0\}$



Lagrange space V_1
Basis $\{\phi_j^1\}$

Since $V_0 \hookrightarrow V_1$, any function $v_0 \in V_0$ can be expressed in both basis:

$$v_0 = \sum_j c_j^0 \phi_j^0 = \sum_l c_l^1 \phi_l^1$$

- The **prolongation** matrix C_{lj} satisfies $c_l^1 = \sum_j C_{lj} c_j^0$.
- The **restriction** matrix is its transpose C^t.
- A object of `class Prolongation` can be obtained from the Lagrange finite element space in `NGSolve`.

A Multigrid Vcycle

```
class NumProcSmoothProject : public NumProc    {
  // :
  void MG(int level, BaseVector & u, const BaseVector & f) {
    if (level==0) { u = (*A0inv) * f;   }
    else {
      // :
      // get matrices A(k) and D(k) at level k, etc

      Jacobi( A, D, u, f );                    // u = u + D*(f - A*u)
      r = f - A * u;
      prl->RestrictInline(level, r);   // r0 = Q (f - A*u)

     MG( level-1, w0, r0 );                   // recurse: w0=MG(0,r0)

      prl->ProlongateInline(level,w); // w = w0
      u += w;                                  // u = u + MG(0,r0)

      Jacobi( A, D, u, f );                    // smooth again
    }
  }
```

Solve by multigrid cycles

- In the pde file, change `-demo=1` to `-demo=2` to run the multigrid V-cycle.
- You should see $\|u^n\|_A \to 0$ much faster.

Built-in facilties

- Use multigrid as a preconditioner in Conjugate Gradients:

```
preconditioner c -type=multigrid -bilinearform=a -smoother=block

numproc bvp np1  -preconditioner=c -bilinearform=a -linearform=f
                 -solver=cg -innerproduct=hermitian -gridfunction=u
```

- See examples in `pde_tutorial`: `d1_square.pde`, `d2_chip.pde`, etc.

- Higher order `FESpaces` use their lowest order subspaces for multigrid.

- An alternate technique to code the Jacobi smoother as a preconditioner is in `my_little_ngsolve/myPreconditioner.cpp`:

```
// Get matrix "mat" from bilinear form.  Then:
jacobi = mat.CreateJacobiPrecond (freedofs);
```

- For more general block smoothers, use `CreateBlockJacobiPrecond`.