Homework #5

Sam Fleischer

December 9, 2016

Problem 1

Write a program to solve the discrete Poisson equation on the unit square using preconditioned conjugate gradient. Set up a test problem and compare the number of iterations and efficiency of using (i) no preconditioning, (ii) SSOR preconditioning, (iii) multigrid preconditioning. Run your tests for different grid sizes. How does the number of iterations scale with the number of unknowns as the grid is refined? For multigrid preconditioning, compare the efficiency with that of multigrid as a standalone solver.

I used all of the same multigrid code from my last homework submission. However I wrote a Gauss-Seidel Red-Black-Black-Red (GSRBBR) smoother and a symmetric SOR (SSOR) smoother since PCG requires a symmetric preconditioner. Here is the GSRBBR code:

```
def GSRBBR(u,rhs,N,h):
    # u:   the grid to smooth
    # rhs: the right hand side function to use
    # N:   the size of the grid
    # h:   the grid spacing

    # Get a checkerboard of 1s and 0s
    checkerboard = get_checkerboard_of_size(u.shape[0])
    # Gather the indices of the "red" and "black" points
    red_indices = np.where(checkerboard == 1)
    black_indices = np.where(checkerboard == 0)
    red_indices = zip(red_indices[0],red_indices[1])
    black_indices = zip(black_indices[0],black_indices[1])
    # Iterate over the red points
    v = GS_iteration(red_indices,u,rhs,N-3,h)
    # Iterate over the black points
    z = GS_iteration(black_indices,v,rhs,N-3,h)
    # Iterate over the black points
    q = GS_iteration(black_indices,z,rhs,N-3,h)
    # Iterate over the red points
    w = GS_iteration(red_indices,q,rhs,N-3,h)
    #return the smoothed grid
    return w
```

where the functions `get_checkerboard_of_size` and `GS_iteration` are given in the last homework assignment. Here is the SSOR code:

```
def SSOR(r, N, h):
    # r: the righthand side of the Laplacian
    # N: The number of grid points
    # h: the grid spacing

    # Calculate the optimal omega
    omega = 2*(1 - np.pi*h)
    # pad the matrix with 0s since my code doesn't take
    # the 0 boundary into account.  It makes the loops easier.
    r = np.pad(r,((1,1),(1,1)),mode="constant")
    # initialize the new matrix (initial guess = 0)
    u = np.zeros(np.shape(r))
    # iterate going forward
    for i in xrange(1, N-1):
        for j in xrange(1,N-1):
            # update the matrix
            UDLR = u[i-1][j] + u[i][j-1] + u[i+1][j] + u[i][j+1]
```

```
            update = UDLR - (h**2)*r[i][j]
            u[i][j] = (omega/4)*update + (1 - omega)*u[i][j]
    # iterate going backwards
    for i in xrange(N-2,0,-1):
        for j in xrange(N-2,0,-1):
            # update the matrix
            UDLR = u[i-1][j] + u[i][j-1] + u[i+1][j] + u[i][j+1]
            update = UDLR - (h**2)*r[i][j]
            u[i][j] = (omega/4)*update + (1 - omega)*u[i][j]
    # return the matrix without the padding
    return u[1:-1, 1:-1]
```

Finally, here is the PCG code:

```
def PCG(u,p,z,r,A,ARGS,X,Y,N,h,Ls,normf):
    # iterate for "max_iterations" or until the condition is met.
    t = tqdm(xrange(ARGS.max_iterations))
    for i in t:
        # copy the old variables
        u_old = deepcopy(u)
        p_old = deepcopy(p)
        z_old = deepcopy(z)
        r_old = deepcopy(r)
        # get w = Ap
        w = A.dot(p_old.flatten()).reshape(N-2,N-2)
        # get alpha = z.r/p.w
        alpha_num = np.dot(z_old.flatten(),r_old.flatten())
        alpha_denom = np.dot(p_old.flatten(),w.flatten())
        alpha = alpha_num/alpha_denom
        # update u = u + alpha.p
        u = u_old + alpha*p_old
        # update r = r - alpha.w
        r = r_old - alpha*w
        # calculate ||r||
        normr = np.amax(np.abs(r))
        # check the exit condition
        if normr <= normf*ARGS.tol:
            # if the condition is met, break out of the loop
            break
        # Set z as the approximate solution of Mz = r, where
        # M is the discrete Laplacian.
        # z is either one iteration of multigrid, one
        # iteration of SSOR, or simply r
        if ARGS.shampoo == "MG":
            z = V_cycle(ARGS.power,np.zeros(z_old.shape),r,(1,1),X,Y,N,h,Ls)
        if ARGS.shampoo == "SSOR":
            z = SSOR(r,N,h)
        elif ARGS.shampoo == "none":
            z = deepcopy(r)
        # Set beta = z_k.r_k/z_{k-1}.r_{k-1}
        beta_num = np.dot(z.flatten(),r.flatten())
        beta_denom = np.dot(z_old.flatten(),r_old.flatten())
        beta = beta_num/beta_denom
        # update p = z + beta.p
        p = z + beta*p_old
```

```
42          # update python output display with the norm of the residual
43          t.set_description("||res||=%.10f" % normr)
44
45      # return the solution and number of iterations
46      return u, i+1
```

Here are some results. The following tables show how many iterations it takes to solve $\nabla^2 u = -\exp\left[-(x-0.25)^2-(y-0.6)^2\right]$ with initial guess $u \equiv 0$.

Method	Grid Spacing	Iterations
SOR	2^{-6}	262
	2^{-7}	508
	2^{-8}	1024
	2^{-9}	2048
MG-GSRB $(\nu_1, \nu_2) = (1,1)$	2^{-6}	9
	2^{-7}	9
	2^{-8}	9
	2^{-9}	9
CG	2^{-6}	49
	2^{-7}	109
	2^{-8}	172
	2^{-9}	349
PCG (SSOR)	2^{-6}	35
	2^{-7}	53
	2^{-8}	81
	2^{-9}	121
PCG (MG-GSRBBR)	2^{-6}	7
	2^{-7}	7
	2^{-8}	7
	2^{-9}	7

All iterations were ran with tolerance= 10^{-7} and break condition $\|r\| \le \text{tol} \cdot \|f\|$

We see that the number of SOR iterations approximately doubles for each halving of the grid spacing. Multigrid is grid-independent. The number of conjugate gradient iterations approximately doubles for each halving of the grid spacing. The number of PCG with SSOR iterations approximately increases by 50% for each halving of the grid spacing. And PCG with MG-GSRBBR is grid-independent. Since PCG with MG-GSRBBR is essentially twice the work per iteration as MG-GSRB, I think the best method is simply MG-GSRB.

Problem 2

I provided the code to give a matrix and right hand side for a discretized Poisson equation on a domain which is the intersection of the interior of the unit square and exterior of a circle centered at $(0.3, 0.4)$ with radius 0.15. The boundary conditions are zero on the square and 1 on the circle.
Use your preconditioned conjugate gradient code to solve this problem. Explore the performance of no preconditioning and multigrid preconditioning for different grid sizes. Comment on your results. Note that the MG code is based on an MG solver for a different domain, and so it cannot be used as a solver for this problem. Is it an effective preconditioner?

SSOR preconditioning Symmetric SOR (SSOR) consists of one forward sweep of SOR followed by one backward sweep of SOR. For the discrete Poisson equation, one step of SSOR is

$$u_{i,j}^{k+1/2} = \frac{\omega}{4}\left(u_{i-1,j}^{k+1/2} + u_{i,j-1}^{k+1/2} + u_{i+1,j}^{k} + u_{i,j+1}^{k} - h^2 f_{i,j}\right) + (1-\omega)u_{i,j}^{k}$$
$$u_{i,j}^{k+1} = \frac{\omega}{4}\left(u_{i-1,j}^{k+1/2} + u_{i,j-1}^{k+1/2} + u_{i+1,j}^{k+1} + u_{i,j+1}^{k+1} - h^2 f_{i,j}\right) + (1-\omega)u_{i,j}^{k+1/2}$$

It can be shown that one step of SSOR in matrix form is equivalent to

$$\frac{1}{\omega(2-\omega)}(D-\omega L)D^{-1}(D-\omega U)(u^{k+1}-u^{k}) = f,$$

where $A = D - L - U$. For the constant coefficient problem, this suggests the preconditioner

$$M = (D-\omega L)(D-\omega U).$$

Multigrid preconditioning To use MG as a preconditioner, the product $M^{-1}r$ is computed by applying one V-cycle with zero initial guess with right hand side r. If the smoother is symmetric and the number of pre and post smoothing steps are the same, this preconditioner is symmetric and definite and may be used with CG. Note that GSRB is not symmeric.

I wrote the given MATLAB code in Python since the rest of my code is in Pyton. Here is the result.

```
def make_matrix_rhs_circleproblem(N):
    # N: grid size

    # Get grid spacing
    dx = 1/(N+1)
    # create a meshgrid
    x = [i*dx for i in xrange(1,N+1)]
    X,Y = np.meshgrid(x,x)
    # Create the 2D discrete Laplacian
    A = (dx**2)*get_laplacian(N)
    # initialize the right hand side
    f = np.zeros(N**2)
    # Define the boundary condition on the circle
    Ub = 1
    # Define the center and radius of the circle
    xc = 0.3
    yc = 0.4
    rad = 0.15
    # precompute the distances of all points on the meshgrid
    # to the center of the circle
    phi = np.sqrt((X - xc)**2 + (Y - yc)**2) - rad
    # Predefine common index additions (up,down,left,right)
    IJ = [[-1,0],[1,0],[0,-1],[0,1]]
    # Loop through the grid
```

```
    for j in xrange(1,N-1):
        for i in xrange(1,N-1):
            # Don't do anything inside the circle
            if phi[i][j] < 0:
                continue
            # loop through common indicies (up,down,left,right)
            for k in xrange(4):
                # only update matrix for points on the boundary
                if phi[i + IJ[k][0]][j + IJ[k][1]] < 0:
                    # Get the appropriate distance to the boundary
                    alpha_num = phi[i][j]
                    alpha_den = phi[i][j]-phi[i+IJ[k][0]][j+IJ[k][1]]
                    alpha = phi[i][j]/alpha_den
                    # Get the distance to the boundary
                    kr = sub2ind([i,j],(N,N))
                    kc = sub2ind([i+IJ[k][0],j+IJ[k][1]],(N,N))
                    # adjust the right hand side
                    f[kr] = f[kr] - Ub/alpha
                    # adjust the diagonal entry
                    A[kr,kr] = A[kr,kr] + 1 - 1/alpha
                    # adjust the off-diagonal entries
                    A[kr,kc] = 0
                    A[kc,kr] = 0
    # return the matrix A and the right hand side f
    return (A, f)
```

where `sub2ind` is Python's `numpy.ravel_multi_index` and `get_laplacian` is given in the previous homework assignment. Here are some results. The following tables show how many iterations it takes to solve $\nabla^2 u = 0$ on Ω where

$$\Omega = \left\{(x,y) \in [0,1]\times[0,1] \text{ such that } (x-0.3)^2 + (y-0.4)^2 \geq 0.15^2\right\},$$

i.e. all points in the unit square, but outside the circle of radius 0.15 at center $(0.3, 0.4)$. The boundary condition is

$$u(x,y) = \begin{cases} 0 & \text{if } x = 0 \text{ or } x = 1 \text{ or } y = 0 \text{ or } y = 1 \\ 1 & \text{if } (x-0.3)^2 + (y-0.4)^2 = 0.15^2 \end{cases},$$

i.e. all points on the unit square are equal to 0 and all points on the circle are equal to 1.

Method	Grid Spacing	Iterations
CG	2^{-6}	181
	2^{-7}	471
	2^{-8}	1241
	2^{-9}	3533
PCG (SSOR)	2^{-6}	45
	2^{-7}	75
	2^{-8}	126
	2^{-9}	232
PCG (MG-GSRBBR)	2^{-6}	31
	2^{-7}	48
	2^{-8}	74
	2^{-9}	120

All iterations were ran with tolerance= 10^{-7} and break condition $\|r\| \leq \text{tol} \cdot \|f\|$

For the un-preconditioned conjugate gradient method, we see iteration increases from 2.6 to 2.8 fold for each halving of the grid spacing. For the SSOR-conditioned conjugate gradient method, we see iteration increases from 1.6 to 1.8 fold for each halving of the grid spacing. For the multigrid-preconditioned conjugate gradient method, we see increases from 1.5 to 1.6 for each halving of the grid spacing. Clearly PCG with MG-GSRBBR is no longer grid independent. I think this means it is no longer worth the work per iteration if PCG with SSOR does slightly worse but is much less expensive per iteration. This means the best option for this problem is PCG with SSOR.