Math 578 Final

Luke Wukmer

Fall 2016

1 The Problem

We use Multigrid-Preconditioned Conjugate Gradient Method (henceforth referred to as **MGCG**) to solve a particular problem:

$$Ax = b$$

where

$$A \in \mathbb{R}^{2^M \times 2^M}, \quad A_{ij} = \begin{cases} 3 & \text{if} \quad j = i, \\ -1 & \text{if} \quad j = i \pm 1 \\ -1 & \text{if} \quad j = N - i + 1, i \neq \frac{N}{2}, \frac{N}{2} + 1 \end{cases}, \tag{1}$$

and b is a $N \times 1$ normalized one vector.

We will compare two methods of preconditioning via multigrid, defined here:

1.1 Scenario #1

In "scenario #1" (referred to henceforth as **SC1**), we precondition via multigrid using the following interpolation matrix (between levels l and $l-1$) (refer to equation #2 in the project spec):

```
def interpolation_matrix(M,L,el):
    """
    The interpolation matrix I_el between levels el and (el-1)
    INPUT:
    M -     refers to size of system 2^M x 2^M
    L -     number of levels/grids
    el-     interpolate between el and el-1

    OUTPUT:
    I_el    a 2^{ M + el - L } by 2^{M + el - (L + 1) }
            interpolation matrix
    """

    n = 2**(M + el - (L+1))

    # identity matrix but repeat each row twice
    # (equivalent to given system)
    return np.repeat(np.eye(n), 2, axis=0)
```

1.2 Scenario #2

Scenario #2, (**SC2**) is the same as the above, but we change precisely one level of interpolation: between the second and third finest levels. Refer to spec and the code below.

```
def interpolation_matrix_2(M,L,el):
    """
    The interpolation matrix I_el between levels el and (el-1)
    for example if M = 2, this function would return
         array([[ 1.,  0.,  0.,  0.],
                [ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  0.,  0.,  1.],
                [ 0.,  0.,  0.,  1.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  1.,  0.,  0.],
                [ 1.,  0.,  0.,  0.]])

    for the given question, this is only to be used for the
    (L-1) to (L-2)th level
    """
    if el != L-1:
        return interpolation_matrix(M,L,el)
    else:
        n = 2**(M-2)
        return np.concatenate((np.eye(n),np.flipud(np.eye(n))))
```

2 Implementation

2.1 The V-cycle

The following is a basic implementation of the multigrid ”v-cycle.”

```
def vcycle(l,b,e0, A, I, Lchol):

    omega = 2/3
    nu1 = 1
    # base case
    if l == 1:
        e_base = linalg.solve(Lchol.T,linalg.solve(Lchol,b))
        return e_base
    else:
        a = A[-(l-1)]
        i = I[-(l-1)]
        e = smooth(a, omega, nu1, b, e0)
        # compute and restrict error
        res = i.T @ (b - a@e)
        # correct error
        e = e + i @ vcycle(l-1,res, np.zeros_like(res), A,I,Lchol)
        # smooth nu1 times on a x = b with initial guess e
        e = smooth(a,omega,nu1,b,e)

    return e
```

2.2 Smoothing function

Here is the smoothing function. As described in the docstring, this function was first tested for accuracy by applying it to a 4×4 diagonally dominant system. This system was suggested by the Wikipedia page for Jacobi iteration.

```
def smooth(A, omega, nu, b, x0, tol=None):
    """
    smoothing function via   -weighted Jacobi iteration
    this is also a standard iterative method on a
    diagonally dominant system

    AA = np.array([[10., -1., 2., 0.],
                   [-1., 11., -1., 3.],
                   [2., -1., 10., -1.],
                   [0.0, 3., -1., 8.]])

    bb = np.array([6., 25., -11., 15.])
    x00 = np.zeros_like(bb)
    ans = smooth(AA,1., 50, bb, x00)

    converges after 24 iterations

    if omega is changed to 2/3 in the above, converges in 35 iterations
    """
    if x0.ndim == 1:
        x0 = np.expand_dims(x0,-1)
    if b.ndim == 1:
        b = np.expand_dims(b,-1)

    x = x0.copy()
    D = np.diag(A) # diagonal of system (as a Nx1 vector)
    # must be same shape as b or will broadcast to a matrix under division
    D = D.reshape(x.shape)

    W = np.tril(A, k=-1) + np.triu(A,k=1) #deleted diagonal

    for i in range(nu):
        x = (1-omega)*x + ((omega*(b- (W@x))) / D)
        if tol is not None and np.allclose(b,A@x, 1e-12):
            break
    else:
        if tol is not None:
            print("Warning, did not converge within tolerance", tol)

    return x
```

2.3 MGCG (Full Method)

```
def mgcg(A, b, n_levels=0,interpolation_method=None, verbose=False):
    """
    Multigrid preconditioned Conjugate Gradient Method
    This solves the system Ax=b for a particular system A.

    if n_levels is 0 do unpreconditioned CG
    """
    M = int(np.log2(A.shape[0]))
    # check that A is actually a power of 2 (no strategy otherwise)
    N = A.shape[0]

    assert 2**M == N

    preconditioned =  (n_levels != 0 and interpolation_method is not None)

    if preconditioned:

        # build *all* interpolation matrices and store
        I = tuple((interpolation_method(M,n_levels,l)
                for l in range(n_levels,1,-1)))
        a = A
        A_levels = list() # store all systems
        for interp in I:
            A_levels.append(a) # append last system matrix

            # yer done if interp is 0x0 (only an issue if L is larger than M)
            if not interp.size:
                break

            a = interp.T @ (a @ interp)

        # now base_case
        A_1 = a
        A_levels = tuple(A_levels) # make static

        # you may check np.allclose(Lchol, linalg.cholesky(A_1,lower=True))
        Lchol, it_chol = cholesky(A_1)

        Minv = lambda b: vcycle(n_levels,b,np.zeros_like(b),A_levels,I,Lchol)


    else:
        # if n_levels == 0 then do unpreconditioned CG
        # i.e. preconditioner is the identity
        Minv = lambda b: b

    pcg_sol, its = pcg(A,b, Minv, return_iterations=True, verbose=verbose)

    if preconditioned:
        return pcg_sol, its, A_levels, I
    else:
        return pcg_sol, its, A, None
```

3 Results

When the program `mgcg.py` is run by itself, the following output is generated. Here, `interpolation_matrix` and `interpolation_matrix_2` refer to PCG in scenario #1 and #2, respectively, while `None` refers to unpreconditioned CG.

```
_____________________________________________________________________
M= 11 L= 6
method: <function interpolation_matrix at 0x7f52b04bbc80>
 65 iterations
 time elapsed= 8.153933763504028
_____________________________________________________________________
_____________________________________________________________________
M= 11 L= 6
method: <function interpolation_matrix_2 at 0x7f52b04bbd08>
 41 iterations
 time elapsed= 4.9130401611328125
_____________________________________________________________________
_____________________________________________________________________
M= 11 L= 6
method: None
 512 iterations
 time elapsed= 3.9258689880371094
_____________________________________________________________________
_____________________________________________________________________
M= 12 L= 7
method: <function interpolation_matrix at 0x7f52b04bbc80>
 99 iterations
 time elapsed= 46.1121826171875
_____________________________________________________________________
_____________________________________________________________________
M= 12 L= 7
method: <function interpolation_matrix_2 at 0x7f52b04bbd08>
 61 iterations
 time elapsed= 28.609034299850464
_____________________________________________________________________
_____________________________________________________________________
M= 12 L= 7
method: None
 1024 iterations
 time elapsed= 29.122495651245117
_____________________________________________________________________
_____________________________________________________________________
M= 13 L= 8
method: <function interpolation_matrix at 0x7f52b04bbc80>
 158 iterations
 time elapsed= 258.28484988212585
_____________________________________________________________________
_____________________________________________________________________
M= 13 L= 8
method: <function interpolation_matrix_2 at 0x7f52b04bbd08>
 87 iterations
 time elapsed= 147.13387250900269
_____________________________________________________________________
_____________________________________________________________________
M= 13 L= 8
method: None
 2048 iterations
 time elapsed= 237.42299604415894
_____________________________________________________________________
```

For clarity, these results are summarized in a table below.

3.1 Calculation of A-multiplies in CG/PCG

A fairly straightforward implementation of preconditioned conjugate gradient method is given below.

```
        def pcg(A,b, Minv, tol=1e-8, x_init=None, return_iterations=False,
                return_error=False, verbose=False):
            """
            preconditioned conjugate gradient method
            solves Ax = b by preconditioning

            INPUT:

            A       - an NxN nd.array describing the system
            b       - initial conditions (can be 1D-array or 2D row vector)
            Minv    - preconditioner, which should be a function handle
            tol     - stopping tolerance (returns sol if ||A*sol - b}||_2 < tol )
                      (optional) default is 1e-8
            x_init  - initial guess (default is None, in which case the zero vector
                        is used)

            return_iterations   (optional) return iteration count (default False)
            return_error        (optional) return error (will be below tolerance
                                if converged)

            OUTPUT:
            x           - solution (an Nx1 nd.array)
            iterations  -(if return_iterations=True above) iterations to run
            err         -(||A*sol-b|| of solution calculated error of residual
            """

            # make sure initial guess is a column vector ala matlab
            if b.ndim == 1:
                b = np.expand_dims(b,-1)

            if x_init is None:
                x = np.zeros_like(b) # default to zero vector as initial guess
            else:
                x = x_init

            tol *= norm(b)  # for stopping check (save some divisions)

            r = b - A@x     # initial residual
            z = Minv(r)     # residual of preconditioned system
            p = z.copy()    # initial search direction
            d = A@p         # initial A-projected search direction


            for iterations in count(1):

                alpha = np.vdot(r,z) / np.vdot(p,d)

                x += alpha*p
```

```
            r_new = r - alpha*d

            # equivalent to norm(b - A@x) / norm(b)
            err = norm(r_new)

            if verbose:
                print(iterations, err, sep='\t| ')

            if err <= tol:
                break

            z_new = Minv(r_new)
            beta = np.vdot(z_new,r_new) / np.vdot(r,z)

            p = z_new + beta*p

            d = A@p

            r = r_new
            z = z_new


            #if err <= tol:
            #    break

        # return statement boogaloo
        if return_iterations:

            if return_error:
                return x, iterations, err / norm(b)
            else:
                return x, iterations

        elif return_error:
            return x, err / norm(b)

        else:
            return x
```

Careful counting of the above shows there is exactly one A-multiplication and one M^{-1}-multiplication in initialization, and one of each during each iteration. By hypothesis, each M^{-1} (which is a v-cycle) corresponds to 5 A-multiplies. Thus PCG requires $1+k+5*(1+k) = \boxed{6+6k}$ A-multiplies for k iterations. By this logic, we can see that CG will only require $\boxed{1+k}$ A-multiplies, (since CG is just PCG where $M^{-}1$ is identity).

3.2 Tables (Iteration Counts, Speedup Ratio, Work Ratio)

So solving our system (1) with unpreconditioned CG converges in the worst case, the exact size of the system. Either preconditioner is a massive improvement. Preconditioner #2 converges in fewer iterations than preconditioner #1.

Our tables suggest that implementation #2 is much more efficient than implementation #1, especially considering the number of A-multiplications per v-cycle can be drastically reduced (see section below).

	M = 11	M = 12	M = 13
CG	512	1024	2048
PCG #1	65	99	158
PCG #2	41	61	87

Table 1: Iteration counts of CG & PCG on our system

	M = 11	M = 12	M = 13
CG vs. PCG #1	7.877	10.343	12.962
CG vs. PCG #2	12.488	16.787	23.540

Table 2: Speedup ratios of CG & PCG on our system (approximate)

	M = 11	M = 12	M = 13
CG vs. PCG #1	1.295	1.708	2.148
CG vs. PCG #2	2.036	2.755	3.88

Table 3: Work ratios of CG & PCG on the system (approximate)

4 The Bonus Question Answered

4.1 The Answer

If MGCG is performed on the system A (1) using the interpolation method **SC2**, the optimal choice of level-depth L is **exactly 3** for any size $M \geq 3$. That is, the "standard" interpolation step in (SC1) is used to interpolate between $A_L \to A_{L-1}$ and $A_{L-2} \to A_{L-3} = A_1$, and the "special" interpolation method unique to **SC2** is performed once in between. There are two reasons why, and they are clearly seen from a visual depiction of the multigrid systems of A using scenario #2.

Refer to **Figure 1**, which was generated with the code `bonus_demo.py`. Here we see the four finest systems in the multigrid when $M = 7, L = 6$, although this result is obviously true for any choice of M.

4.2 A loose justification

The result is clear: applying the modified interpolation step between levels A_{L-1} and A_{L-2} converts the system from its original structure (with an antidiagonal) to a tridiagonal matrix, and all further interpolations preserve tridiagonality. Recalling that our goal in the V-cycle is to restrict the size of the system until we arrive at a system that is reasonable to solve directly, it makes sense that we should immediately solve such a system. There will be no additional benefit from successive restriction of the system once it is tridiagonal.

bonus_demo.py

```
#!/usr/bin/env python3

from mgcg import *
import matplotlib.pyplot as plt
import numpy as np

M = 7
levels = 6
A_L = make_system(M)
b = np.ones((2**M,1))/2**(M/2)


sol, its, As, Is = mgcg(A_L,b,n_levels=levels,
        interpolation_method=interpolation_matrix_2)

fig, ax = plt.subplots(2,2)

ax = ax.ravel()

for i, a in enumerate(ax):

    ax[i].spy(As[i])
    if i > 0:
        ax[i].set_title(r'$A_{\mathscr{l}},\;\mathscr{l}=L - $'+str(i),
                fontsize=20)
    else:
        ax[i].set_title(r'$A_L$', fontsize=20)
    ax[i].axis('off')

fig.tight_layout()
plt.show()
```

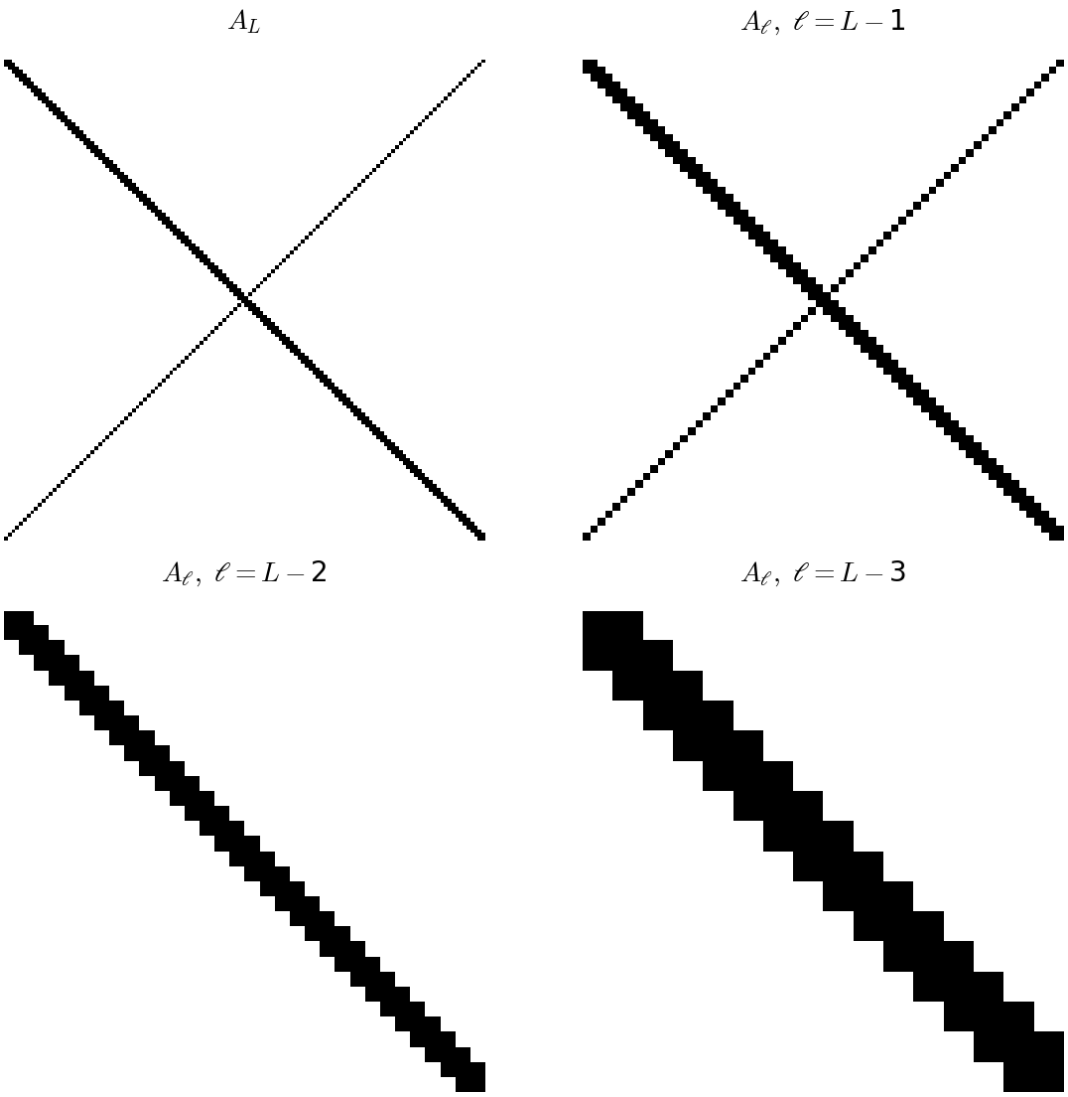

Figure 1: A look at the four finest levels of multigrid under **SC2**. After the second interpolation step, all subsequent levels are tridiagonal (and thus easily solved by Cholesky factorization)

4.3 Empirical Support via Run times

Of course, since performing the downscaling of the system A isn't computationally "free", we can directly see the superiority of choosing *exactly* 3 levels. The following shows a session in `ipython`, where we perform the entire MGCG procedure on our system when $M = 11$, iterating on the number of levels $L = 1, 2, \ldots, 9$. The average runtime over 3 runs is displayed for each L value.

```
In [1]: from mgcg import *
In [2]: M = 11
In [3]: A = make_system(M)
In [4]: b = make_initial_conditions(M)
In [5]: for L in range(1,10):
  ....:    %timeit mgcg(A,b,L,interpolation_matrix_2)
  ....:
1 loop, best of 3: 2.65 s per loop
1 loop, best of 3: 1.86 s per loop
1 loop, best of 3: 1.48 s per loop <-------------------------------
1 loop, best of 3: 2.79 s per loop
1 loop, best of 3: 3.96 s per loop
1 loop, best of 3: 4.99 s per loop
1 loop, best of 3: 6.84 s per loop
1 loop, best of 3: 7.58 s per loop
1 loop, best of 3: 7.9 s per loop
```

These times are depicted in the graph below:

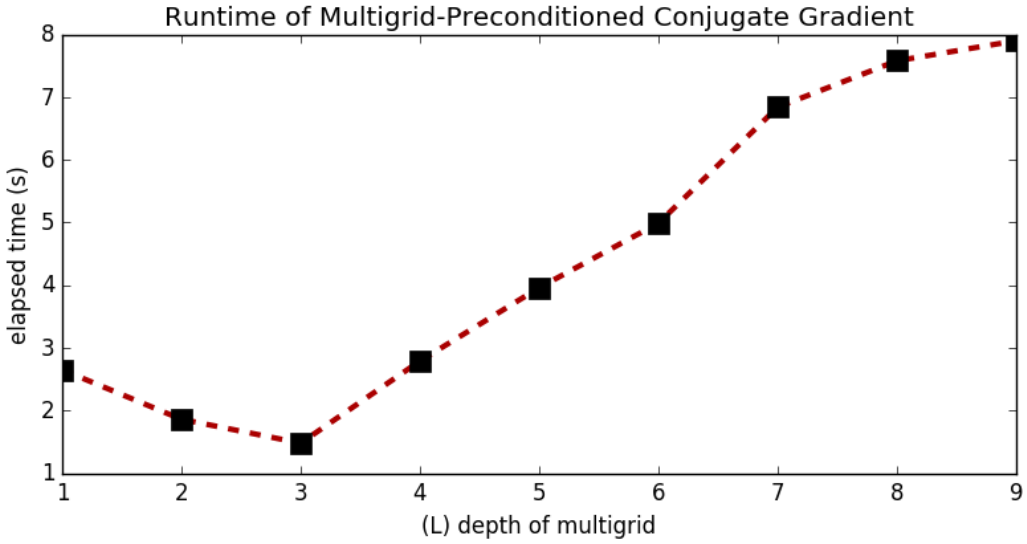

Figure 2: Measuring run time of solving $Ax = b$ with MGCG when $M = 11$, and varying L. Ran on a laptop using unoptimized code.

4.4 An alternative justification

Figure 3 shows the effect of the preconditioners using both **SC1** and **SC2** on the initial condtions, as compared to the actual found solution of $Ax = b$ with system (1). Note the progress made by **SC2** after 3 levels is both more than *ever* achieved by **SC1** when we compare how the initial conditions are transformed.

This picture also gives a strong (yet loose and possibly fallacious) rationalization for the particular nature of the novel step in **SC2**: After a single round of restriction, the L-1 level restriction/interpolation step *forces* the remaining solutions to be symmetric, with a crease in the middle (refer to implementation of $SC2$ in the spec and also this report). It just so happens that the particular solution to our $Ax = b$ is a very symmetrical solution, so (whatever the specifics), it is clear that the new step in **SC2** is exactly designed to exploit the *expected* symmetry of the solution to $Ax = b$, whereas (SC1) does not take advantage of the symmetry.

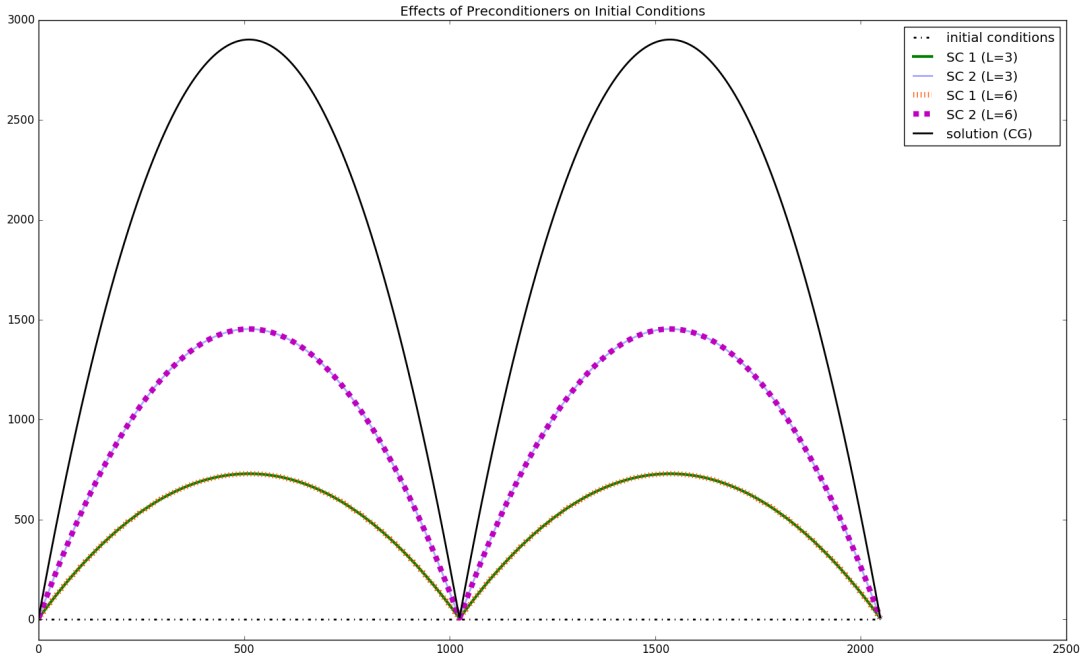

Figure 3: Measuring the proximity of a sample vector (the initial conditions and initial residual of PCG) to the eventual solution when $M = 11$.