Math 578 Final

Luke Wukmer

Fall 2016

1 Implementation

1.1 The 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
```

1.2 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 Results

```
In [5]: run mgcg.py
```

```
M= 11 L= 6
method: <function interpolation_matrix at 0x7fe7c1569d08>
65 iterations
time elapsed= 7.588343381881714
```

```
M= 11 L= 6
method: <function interpolation_matrix_2 at 0x7fe7c1569d90>
41 iterations
time elapsed= 4.9514479637146
```

```
M= 11 L= 6
method: None
512 iterations
time elapsed= 3.9358863830566406
```

```
M= 12 L= 7
method: <function interpolation_matrix at 0x7fe7c1569d08>
99 iterations
time elapsed= 45.84742569923401
```

```
M= 12 L= 7
method: <function interpolation_matrix_2 at 0x7fe7c1569d90>
61 iterations
time elapsed= 28.84040141105652
```

```
M= 12 L= 7
```

```
method: None
1024 iterations
time elapsed= 29.221490144729614
```

```
M= 13 L= 8
method: <function interpolation_matrix at 0x7fe7c1569d08>
58 iterations
time elapsed= 256.6284234523773
```

```
M= 13 L= 8
method: <function interpolation_matrix_2 at 0x7fe7c1569d90>
87 iterations
time elapsed= 154.2831733226776
```

```
M= 13 L= 8
method: None
2048 iterations
time elapsed= 230.1282079219818
```

3 The Bonus Question Answered

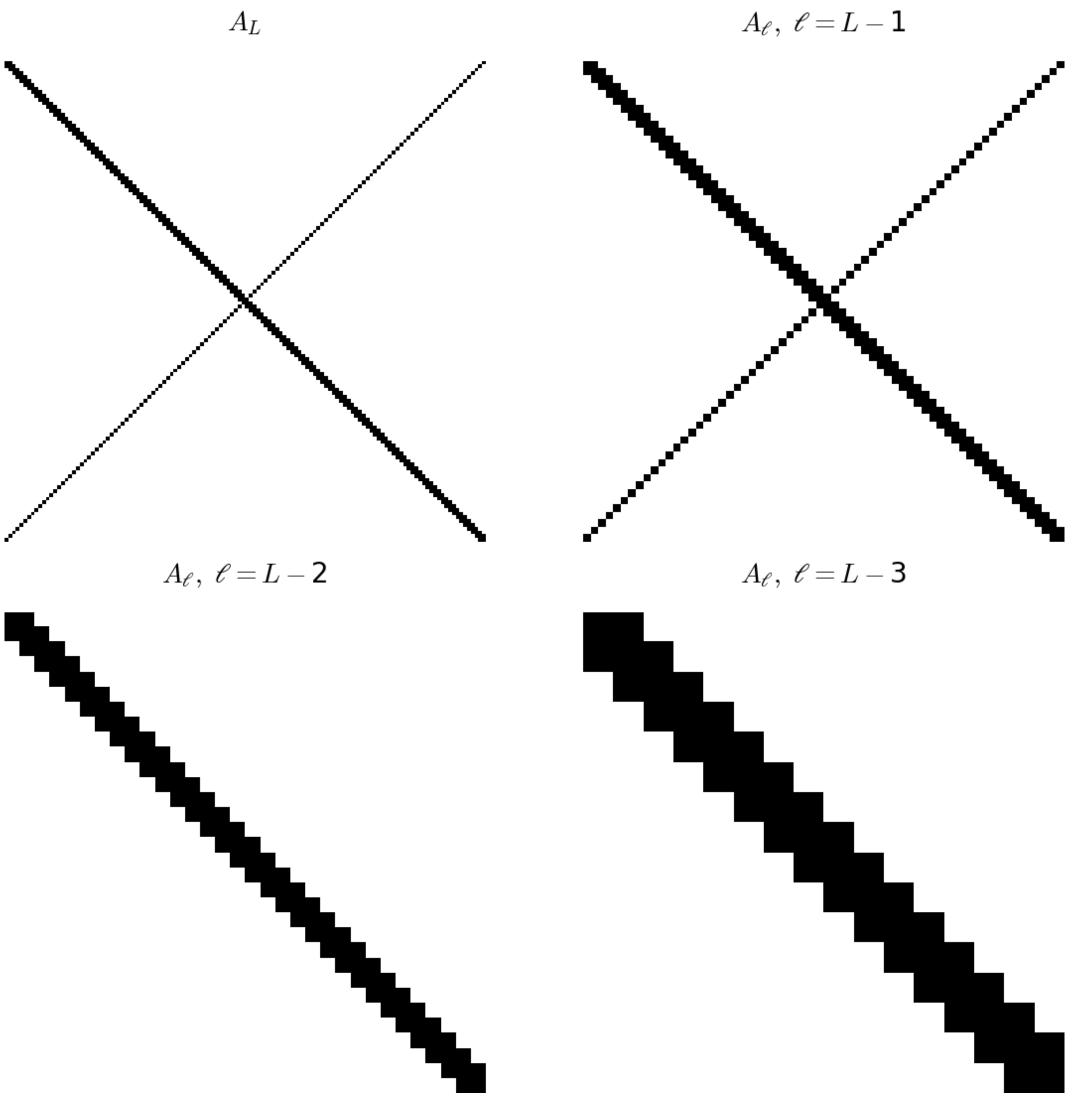
If MGCG is performed on the system-in-question A using the interpolation method known as ”scenario #2”, the optimal choice of level-depth L is **exactly 3** for any size $M \geq 3$. That is, the ”standard” interpolation method (2) is used to interpolate between $A_L \to A_{L-1}$ and $A_{L-2} \to A_{L-3} = A_1$, and the ”special” interpolation method introduced in scenario #2 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.

The following figure 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.

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

```python
#!/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):
```

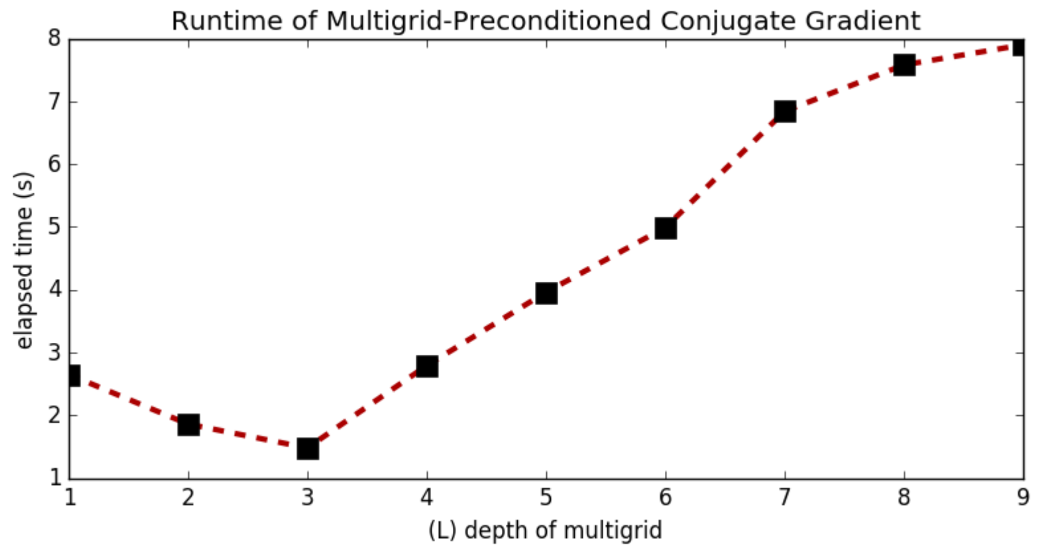

A_L
$A_\ell,\ \ell = L-1$
$A_\ell,\ \ell = L-2$
$A_\ell,\ \ell = L-3$

```
    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()
```

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 = np.ones((2**M, 1))/2**(11/2)
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 shown in the graph below:


Runtime of Multigrid-Preconditioned Conjugate Gradient
elapsed time (s)
(L) depth of multigrid
1
2
3
4
5
6
7
8
9