Math 578 HW$^{\#}$5

Luke Wukmer

Fall 2016

The entirety of this code is contained in the included single file `hw5.py`. The main executing code that solves problems #1-3 is contained within the (`if __name__ == "__main__":`) clause. See the **Appendix** for code for all three problems.

Problem 1. **(a)** The system for problem 1 is generated by `p1_system()`. PCG is implemented in pcg() (which is identical to the code used on the final project, with the exception of an additional optional parameter `max_iterations`). Refer to the **Appendix**.

(b) Iteration counts and size of $\|Ax - b\|$ for systems of the size N for CG and PCG (with Jacobi preconditioning) is shown in the table below:

N	method	iterations	$\|Ax - b\|_2$
10	**CG**	10	1.605e-15
50		62	1.217e-12
100		200	0.001
200		400	362.102
500		1000	3126.407
10	**PCG (Jacobi)**	10	5.768e-16
50		18	6.405e-12
100		19	8.602e-13
200		19	8.842e-13
500		19	9.519e-13

Table 1: Problem 1(b) output

For each of these runs, pcg was run with `max_iterations=2*N` . Of course, *theoretically* CG should converge successfully within N iterations. Notice that for larger systems CG simply does not converge–in fact, the residual norm increases. It's pretty easy to say in this case that CG is not a useful iterative method for this particular ill-conditioned system. This is compared to PCG, which begins to converge in a constant number of iterations, regardless of change to problem size N.

Problem 2. The system for problem 2 is constructed by `p2_system()`.

(a) Check inverse. Let B be the matrix given by

$$B = \frac{1}{2}\left(I + \sum_{k=1}^{\infty}\left[\frac{\alpha^k}{2^k}W^k\right]\right)$$

Then

$$\begin{aligned}
AB &= (2I - \alpha W)\, B \\
&= \frac{1}{2}(2I - \alpha W)\left(I + \sum_{k=1}^{\infty}\left[\frac{\alpha^k}{2^k}W^k\right]\right) \\
&= \frac{1}{2}\left(2I - \alpha W + 2\sum_{k=1}^{\infty}\left[\frac{\alpha^k}{2^k}W^k\right] - \alpha W\sum_{k=1}^{\infty}\left[\frac{\alpha^k}{2^k}W^k\right]\right) \\
&= I - \frac{1}{2}\alpha W + \sum_{k=1}^{\infty}\left[\frac{\alpha^k}{2^k}W^k\right] - \sum_{k=1}^{\infty}\left[\frac{\alpha^{k+1}}{2^{k+1}}W^{k+1}\right] = I
\end{aligned}$$

as desired, so the infinite series B is in fact the inverse of A.

(b) Using the first three terms of this representation of the inverse, PCG was run again. Output is given in Table #2.

N	method	iterations	$\|Ax-b\|_2$
50	**CG**	26	2.662e-15
100		51	3.908e-15
200		101	1.072e-14
500		198	8.682e-13
50	**PCG**	13	4.402e-15
100		26	4.148e-15
200		51	6.312e-15
500		99	8.682e-13

Table 2: Iteration counts for the problem $Ax = b$ in Problem 2(b).

(c) In the (P)CG algorithm there is exactly 1 multiplication by A and 1 multiplication by M^{-1} per iteration. In the case of no preconditioning (conventional CG), the M^{-1} multiplication is clearly omitted. In terms of this problems system and choice of M^{-1}, each of these counts as one $A-$multiplication. Thus the number of A multiplications required for CG is $N + 1$, while PCG is $2(N + 1)$, including the initialization step. Fr the table above see see that PCG converges in almost *exactly half* as many iterations as CG, while requiring twice as much ”work” for the problem in question. I guess they’re equivalent for solving this particular $Ax = b$.

Problem 3. (GMRES)

(a) Code for `gmres()`, as well as `apply_givens()` and such is contained in the **Appendix**.

(b) I couldn’t get the fucking thing to work.

A Relevant Code

hw5.py

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

import numpy as np
from numpy.linalg import norm
from itertools import count
from functools import partial

from scipy import sparse

def backsolve(R,b):
    """
    solves a system Rx=b via back substituation where R is an
    upper-triangular matrix
    """
    n = R.shape[0]

    x = np.zeros(b.shape)

    x[-1] = b[-1] / R[-1,-1] # last entry to initialize
    for i in range(n-2,-1,-1):
        x[i] = (b[i] - R[i,i+1:]@x[i+1:]) / R[i,i]

    return x

def givens(x):
    """
    returns the components c,s of the rotation matrix

    G = ( c   -s )
        ( s    c )

    such that the 2x1 input x = [a b]ˆT would be rotated to align with e_1:

    G x = [ ˜, 0]ˆT for some number ˜
    """

    a, b = x

    # this equals 2**(-52) exactly when x is float64
    eps = np.finfo(x.dtype).eps

    if np.isclose(a,0):
        # np.sign output is 1 or -1
        c, s = np.sign(a), 0

    elif np.isclose(b,0)
        c, s = 0, -np.sign(b)

    elif np.abs(a) > np.abs(b):
        t = b/a
        u = np.sign(a)*np.abs(np.sqrt(1+t*t))
        c = 1/u
        s = -c*t

    else:
        t = a/b
        u = np.sign(b)*np.abs(np.sqrt(1+t*t))
        s = -1/u
        c = -s*t

    return c,s

def apply_givens(v,h):
    """
    apply the givens transformation to a vector
    """
    print('starting apply_givens with v.shape ==',v.shape, 'and',
            'h.shape==',h.shape)

    if v.size == 0:
        pass  # apply no rotations; v is empty
    else:
        for j in range(v.shape[0]-1):
            c,s = v[j,:] # jth givens coefficients

            G = np.array([[c, -s],
                          [s,  c]])

            h[j:j+2] = G @ h[j:j+2]

    return h

def pcg(A,b, Minv, tol=1e-8, x_init=None, return_iterations=False,
        return_error=False, verbose=False, max_iterations=None):
    """
    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

    if max_iterations is None:
        # start from one and count to infinity
        counter = count(1)
    else:
        counter = range(1,max_iterations+1)
    for iterations in counter:

        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

def pcgmres(A, b, tol, M_inv):
    """
    preconditioned GMRES
    inputs
        A   input system A nxn
        b   inital conditions
        tol
        M_inv inverse of preconditioner M

    outputs
        x   solution to Ax=b
        it  iteration count

    """
    if b.ndim == 1:
        b = np.expand_dims(b,-1)

    b = M_inv@b
       = norm(b,2)

    Q = b /       # will store Arnoldi vectors basis (expands in cols)

    R = np.empty((0,0)) # expands in size
    V = np.empty((0,2)) # expands in size

    r =    # always a scalar
    t = np.array([[  ]]) # expanding vector

    n = A.shape[0]

    # make sure to check & fix iteration count
    for it in range(1,n+1):
        print('iteration ', it)
        print('\tr=',r)
        print('\ttol*  =',tol*  )
        if (r <= tol*  ):
            break

        z = M_inv @ (A @ Q[:,-1:]) # Aq_k (i.e. latest Q vector)
        h = (Q.T @ z)
        h_tilde = np.sqrt(np.abs(np.vdot(z,z) - np.vdot(h,h)))

        #h_tilde = norm(z-Q@h,2)
        # will be empty on first pass, hope that's okay
        h_hat = apply_givens(V,h)

        c, s = givens(np.array([h_hat[-1,-1] , h_tilde]))
        if np.isnan(c) or np.isnan(s):
            raise

        #h_hat[-1,-1] = c*h_hat[-1,-1] - s*h_tilde
        h_hat[-1,-1] = c*h_hat[-1,-1] - s*h_tilde

        h_new = np.vstack((h_hat,h_tilde))
        # apply and store givens rotations, etc.
        V = np.vstack((V, np.array([c,s]))) # k x 2

        # form upper triangular matrix R
        R = np.hstack((np.vstack((R,np.zeros((1,R.shape[1])))),h_hat))
        print(R)

        # t grows in size by one. can't figure out how to do this cleanly
        t = np.resize(t, (t.size+1,1))
        # just to be safe--this isn't actually used until it's overwritten
        t[-1,-1] = 0

        # apply c & s to second to last t to get last t
        t[-2:] = V[-1:].T @ t[-2:-1]
        print(t)
        r = np.abs(t[-1,-1])


        # if there will be another iteration
        if r >= tol*   and it < n:
            # compute next Arnoldi vector
            q = z - Q@h
            q = q / norm(q,2) # or w / h_tilde (same number)

            # add on additional basis vector
            Q = np.hstack((Q,q))

    y = backsolve(R,t[:-1])
    #y = np.linalg.solve(R,t[:-1])
    # form approximation x
    x = Q@y

    return x, it, locals()


def p1_system(n):
    """
    A_{i,j} = { 2 + (1.1)^i     if j = i
              { -1              if j = i+1 , i-1
              { 0               otherwise
    """
    A = -1*np.eye(n,k=-1) - np.eye(n,k=1)
    A += np.diag(np.fromfunction(lambda i: 2+1.1**(i+1),(n,)))

    return A

def p2_system(n,alpha):

    W = -1*np.eye(n,k=-1) - np.eye(n,k=1)
    W[0,-1] = -1
    W[-1,0] = -1

    return  2*np.eye(n) - alpha*W

def p2_preconditioner(n,alpha):
    """ as a literal matrix"""
    W = -1*np.eye(n,k=-1) - np.eye(n,k=1)
    W[0,-1] = -1
    W[-1,0] = -1

    return .5*np.eye(n) + 0.25*alpha*W

if __name__ == "__main__":


    identity = lambda x: x
    jacobi_preconditioner = lambda x: (x / np.fromfunction(lambda i,j:
            2 + 1.1**(i+1),(x.size,1)))

    Ns = (10,50,100,200,500)
    preconditioners = {"CG":identity,
                       "PCG (Jacobi)":jacobi_preconditioner
                       }
    print("---------------Problem 1(b) output:")
    for label, preconditioner in preconditioners.items():
```

```python
        for N in Ns:
            A = p1_system(N)
            b = np.ones((N,1))
            x, its = pcg(A,b, Minv=preconditioner,
                    tol=1e-12, return_iterations=True,
                    max_iterations=2*N)
            err = norm(A@x-b,2)
            print(N, label, its, err)

    # PROBLEM 2(B)

    Ns = (50,100,200,500)

    preconditioners = {"CG":identity,
                       "PCG": None
                       }
    print("-----------------Problem 2(b) output:")
    for label, preconditioner in preconditioners.items():
        for N in Ns:

            if label == "PCG":
                Minv = p2_preconditioner(N,.99)
                preconditioner = lambda x: Minv @ x

            A = p2_system(N, .99)
            b = np.zeros((N,1))
            b[N//2 - 1] = 1 # the N/2th element is 1 only
            x, its = pcg(A,b, Minv=preconditioner,
                    tol=1e-12, return_iterations=True,
                    max_iterations=2*N)
            err = norm(A@x-b,2)
            print(N, label, its, err)



    #----------------Problem 1(b) output:
    #10 CG 10 1.60503080768e-15
    #50 CG 62 1.21717759909e-12
    #100 CG 200 0.00122692975884
    #200 CG 400 362.102526103
    #500 CG 1000 3126.40771968
    #10 PCG (Jacobi) 10 5.76888805915e-16
    #50 PCG (Jacobi) 18 6.40516273459e-12
    #100 PCG (Jacobi) 19 8.60237158945e-13
    #200 PCG (Jacobi) 19 8.84213756097e-13
    #500 PCG (Jacobi) 19 9.51937987479e-13
    #-----------------Problem 2(b) output:
    #50 CG 26 2.66251064586e-15
    #100 CG 51 3.90885062235e-15
    #200 CG 101 1.07271654085e-14
    #500 CG 198 8.6827800646e-13
    #50 PCG 13 4.40221279272e-15
    #100 PCG 26 4.14852699702e-15
    #200 PCG 51 6.31278960557e-15
    #500 PCG 99 8.68282914175e-13
```