```python
#!/usr/bin/python
from common import *
import pdb
def p1(L):
    #Given:
    #   L, a list of 1xN vectors with len(L) = M
    #Return:
    #   MxN matrix of all the vectors stacked together
    #Par: 1 line
    #Instructor: 1 line
    #Hint: vstack/hstack/dstack, don't use a for loop
    P = np.vstack(L)

    return P

def p2(M):
    #Given:
    #   M, a n x n matrix
    #Return:
    #   v the eigenvector corresponding to the smallest eigenvalue
    #Par: 5 lines
    #Instructor: 3 lines
    #Hints:
    #   1) np.linalg.eig
    #   2) np.argmin
    #   3) Watch rows and columns!
    wm,vm = np.linalg.eig(M)
    sm = np.argmin(wm)
    P=vm[:,sm]

    return P

def p3(M):
    #Given:
    #   a matrix M
    #Return:
    #   M, but with all the negative elements set to 0
    #Par: 3 lines
    #Instructor: 2 lines (there's a fairly obvious one line solution -- hint
    #                     np.minimum/maximum/np.clip)
    #Hint:
    #   1) if S is the same size as M and is True/False, you can refer to all
    #   true entries via M[S]
    #   2) if M[S] is the set of all entries, you can assign to them all with
    #   M[S] = v for some value v
    P = np.clip(M,0,np.amax(M))

    return P


def p4(t):
    #Given:
    #   a tuple of 3x3 rotation matrix R
    #   Nx3 matrix M
    #Return:
    #   a Nx3 matrix of the rotated vectors
    #Par: 3 lines
    #Instructor: 1 line
    #Hint:
    #   1) Applying a rotation to a vector is right-multiplying the rotation
    #      matrix with the vector
    #   2) .T transposes; this may make your life easier
    #   3) np.dot matrix-multiplies
    R, M = t #unpack
    P=np.dot(M,R.T)

```

```
    return P

def p5(M):
    #Given:
    #   a NxN matrix M
    #Return:
    #   the upper left 4x4 corner - bottom right 4x4 corner
    #Par: 2 lines
    #Instructor: 1 line
    #Hint:
    #   M[ystart:yend,xstart:xend] pulls out the sub-matrix
    #       from rows ystart to (but not including!) yend
    #       from columns xstart to (but not including!) xend
    P = M[0:4,0:4]-M[-4:,-4:]

    return P

def p6(n):
    #Given:
    #   n -- an integer
    #Return:
    #   a nxn matrix of 1s, except the first and last 5 columns and rows are 0
    #Par: 5 lines
    #Instructor: 5 lines (can you make it shorter with np.zeros)
    #Hints:
    #   np.ones/np.zeros, it's ok to double-write

    M = np.zeros((n,n))
    M[5:n-5,5:n-5] = 1

    return M


def p7(M):
    #Given:
    #   a NxF matrix M
    #return:
    #   S -- the same matrix but where each row is scaled to have unit norm
    #  (i.e., S[i,:] is unit norm, and S[i,j] = M[i,j]*a[i] for some a[i])
    #Par: 3 lines
    #Instructor: 1 line
    #Hints:
    #   1) The vector v / ||v|| is unit norm for v != 0
    #   2) Compute the normalization factor as a Nx1 vector by N[i] = \sum_j M[i,j]^2
    #   3) Elementwise divide a NxF matrix by a Nx1 vector -- see what happens!
    #      (broadcasting)
    #   4) If it won't go together -- np.expand_dims or keepdims in np.sum or doing
    #      X[None,:] to add dimensions (try X[None,:] and X[None,:]) on a vector

    # N=np.square(N)
    # N=np.sum(M, axis=1)
    # N = np.sqrt(N)
    # P=M/N

    norm=np.linalg.norm(M,axis=1)
    norm = norm[None,:].T
    P = M/norm


    return P


def p8(M):
    #Given:
    #   a matrix M
    #Return:
```

```
    #   the same matrix but where each row is normalized to have mean 0 and std 1
    #   (i.e. mean(S[i,:]) = 0, std(S[i,:]) = 1 and S[i,j] = M[i,j]*a[i]+b[i] for some a[i],b[i])
    #Par: 3 lines
    #Instructor: 2 lines (but you can make it one)
    #Hints:
    #   1) If it won't broadcast, do np.expand_dims, keepdims
    M-=np.mean(M,axis=1,keepdims=True)
    M/=np.std(M,axis=1,keepdims=True)

    return M

def p9(t):
    #Given a:
    #   query q -- (1xK)
    #   keys k -- (NxK)
    #   values v -- (Nx1)
    #Return
    #   sum_i exp(-||q-k_i||^2) * v[i]
    #Par: 3 lines
    #Instructor: 1 incomprehensible line, not written in one go
    #Hints:
    #   1) Again A NxK matrix and a 1xK vector go together the way you think
    #       (broadcasting)
    #   2) np.sum has an axis and keepdims arguments
    #   3) np.exp, - and friends apply to matrices too
    Q, K, V = t #unpack

    J=np.exp(-np.linalg.norm((Q-K),axis=1)**2)
    J = J[:,None]
    P=np.sum(J*V)

    return P

def p10(L):
    #given:
    #   a list NxF matrices of length M
    #return:
    #   a MxM matrix R where
    #   R_ij = distance between the F-dimensional centroid of each matrix
    #Par: 12 lines
    #Instructor: 7 lines (there's a 9 line solution that avoids double work
    #             and apparently a 4 line solution too)
    #Hints:
    #   1) For loop over M
    #   2) Distances are symmetric, so don't double compute that
    #   3) Go one step at a time
    Cen = []*len(L)
    R = np.zeros((len(L),len(L)))
    for i in range(len(L)):
        Cen.append(np.sum(L[i],axis=0)/np.size(L[i],0))

    for i in range(len(L)):
        for j in range(len(L)):
            if j>=i:
                R[i,j] = np.linalg.norm(Cen[i]-Cen[j])
                R[j,i] = R[i,j]

    return R


def p11(M):
    #given:
    #   a NxF matrix M
    #compute the NxN matrix D
    #   D[i,j] = distance between M[i,:] and M[j,:]
    #   using ||x-y||^2 = ||x||^2 + ||y||^2 - 2x^T y
```

```
    #Par: 3 lines
    #Instructor: 2 lines (you can do this in one but it's wasteful compute-wise)
    #Hints:
    #   1) If I add a Nx1 vector and a 1xN vector, what do I get?
    #   2) Look at the definition of matrix multiplication for the second bit
    #   3) transpose is your friend
    #   4) Note the square! -- square root it at the end
    #   5) On some computers, you may have issues with ||x||^2 + ||x||^2 - 2x^Tx
    #      coming out as ever so slightly negative. Just make max(0,value) --
    #      note that the distance between x and itself should be exactly 0.
    #      Seems to occur on macs

       XNorm = np.sum(M**2,axis=1,keepdims=True)
       D_s = XNorm+XNorm.T - 2*np.dot(M,M.T)
       D = np.maximum(0,D_s)**0.5

       return D


def p12(t):
    #Given:
    #   a NxF matrix A
    #   a MxF matrix B
    #compute the NxM matrix D
    #   D[i,j] = distance between A[i,:] and B[j,:]
    #Par: 3 lines
    #Instructor: 1 line
    #Hints: same same but different; draw some boxes on a piece of paper
    A,B = t #unpack
    XNorm = np.sum(A**2,axis=1,keepdims=True)
    YNorm = np.sum(B**2,axis=1,keepdims=True)
    D = np.maximum(0,XNorm+YNorm.T - 2*np.dot(A,B.T))**0.5

    return D


def p13(t):
    #Given:
    #   a 1xF query vector q
    #   NxF matrix M
    #Return:
    #   the index i of the row with highest dot-product with q
    #Par: 1 line
    #Instructor: 1 line
    #Hint: np.argmax
    q, M = t #unpack

    ind=np.argmax(np.dot(M,q.T))

    return ind

def p14(t):
    #given a tuple of:
    #   X NxF matrix
    #   y Nx1 vector
    #Return the w Fx1 vector such that
    #   ||y-Xw||^2_2 is minimized
    #Par: 2 lines
    #Instructor: 1 line
    #Hint: np.linalg.lstsq or do it the old fashioned way (X^T X)^-1 X^T y
    X , y = t #unpack

    W,*_=np.linalg.lstsq(X,y,rcond=-1)
```

```
    return W


def p15(t):
    #Given a tuple of:
    #   X: Nx3 matrix
    #   Y: Nx3 matrix
    #Return a matrix:
    #   C such that C[i,:] = the cross product between X[i,:] and Y[i,:]
    #Par: 1 line
    #Instructor: 1 line
    #Hint: np.cross and read the documentation or just try it
    X, Y = t #unpack
    C= np.cross(X,Y)

    return C

def p16(X):
    #Given:
    #   a NxF matrix X
    #Return a Nx(F-1) matrix Y such that
    #   Y[i,j] = X[i,j] / X[i,-1]
    #   for all i and j
    #Par: 1 line
    #Instructor: 1 line
    #Hint: if it doesn't broadcast, np.expand_dims

    Y=X/np.expand_dims(X[:,-1],axis=1)

    return Y[:,:2]

def p17(X):
    #Given:
    #   a NxF matrix X
    #Return a Nx(F+1) matrix Y such that
    #   Y[i,:F] = X[i,:] and
    #   Y[i,F] = 1
    #Par: 1 line
    #Instructor: 1 lines
    #Hint: np.hstack, np.ones

    Y = np.hstack((X,np.ones((np.size(X,0),1))))

    return Y


def p18(t):
    #Given:
    #   an integer n
    #   a radius r
    #   an x coordinate x
    #   a y coordinate y
    #Return:
    #   An nxn image I such that
    #   I[i,j] = 1 if ||[j,i] - [x,y]|| < r
    #   I[i,j] = 0 otherwise
    #Par: 3 lines
    #Instructor: 2 lines
    #Hint:
    #   1) np.meshgrid and np.arange give you X,Y
    #   2) watch the <
    #   3) arrays have an astype method
    n,r,x,y = t #unpack

    X,Y = np.meshgrid(np.arange(n),np.arange(n),sparse=True)
    I= ((X-x)**2 +(Y-y)**2<r**2).astype(float)
```

```
    return I


def p19(t):
    #Given:
    #   an integer n
    #   a float s
    #   an x coordinate x
    #   a y coordinate y
    #Return:
    #   a nxn image such that
    #   I[i,j] = exp(-||[j,i]-[x,y]||^2  / s^2)
    #Par: 3 lines
    #Instructor: 2 lines
    #Hint: watch the types -- float and ints aren't the same!
    n, s, x, y = t #unpack

    X,Y = np.meshgrid(np.arange(n),np.arange(n),sparse=True)
    I = np.exp(-((X-x)**2+(Y-y)**2)/s**2)

    return I


def p20(t):
    #Given:
    #   an integer n
    #   a vector v = [a,b,c]
    #Return:
    #   a matrix M such that M[i,j] is the distance from the line a*j+b*i+c=0
    #
    #   Given a point (x,y) and line ax+by+c=0, the distance from x,y to the line
    #   is given by abs((ax+by+c) / sqrt(a^2 + b^2)) (the sign tells you which side)
    #Par: 4 lines
    #Instructor: 2 lines
    #Hints:
    #   np.abs works on matrices too
    n, v = t #unpack
    X,Y = np.meshgrid(np.arange(n),np.arange(n),sparse=True)

    M=np.abs((v[0]*X+v[1]*Y+v[2])/np.sqrt(v[0]**2 + v[1]**2))

    return M
```

```
iunghuiui-MacBook-Pro:mastery_assignment ungheelee$ python3 run.py --alltests
Running p1
Running p2
Running p3
Running p4
Running p5
Running p6
Running p7
Running p8
Running p9
Running p10
Running p11
Running p12
Running p13
Running p14
Running p15
Running p16
Running p17
Running p18
Running p19
Running p20
Ran all tests
20/20 = 100.0
```

```
from common import *


def b1(M):
    #Given:
    #   a matrix M
    #Return:
    #   the matrix S such that S[i,j] = M[i,j]*10+100
    #Hint: Trust that numpy will do the right thing
    S = M*10 + 100

    return S

def b2(t):
    #Given:
    #   a nxn matrix M1
    #   a nxn matrix M2
    #Return:
    #   the matrix P such that P[i,j] = M1[i,j]+M2[i,j]*10
    #Hint: Trust that numpy will do the right thing
    M1, M2 = t #unpack
    P = M1 + M2*10

    return P

def b3(t):
    #Given:
    #   a nxn matrix M1
    #   a nxn matrix M2
    #Return:
    #   the matrix P such that P[i,j] = M1[i,j]*M2[i,j]-10
    #Hint: By analogy to + , * will do the same thing
    M1, M2 = t #unpack
    P = M1*M2-10

    return P

def b4(t):
    #Given:
    #   a nxn matrix M1
    #   a nxn matrix M2
    #Return:
    #   the matrix product M1 M2
    #Hint: Not the same as * !
    M1, M2 = t #unpack
    P = M1.dot(M2)

    return P

def b5(M):
    #Given:
    #   a nxn matrix M of floats
    #Return:
    #   a nxn matrix M of integers
    #Hint: astype
    # M.astype(int)
    M=np.int64(M)

    return M

def b6(t):
    #Given:
    #   a nx1 vector M of integers
    #   a nx1 vector D of integers
```

```python
    #Return:
    #   the ratio (M/D), treating them as floats (i.e., 1/5 => 0.2)
    #Hint: dividing one integer by another is not the same as dividing two floats
    M, D = t #unpack
    P=M/D

    return P

def b7(M):
    #Given:
    #   a nxm matrix M
    #Return:
    #   a vector v of size (nxm)x1 containing the entries of M, listed in row order
    #Hint:
    #   1) np.reshape
    #   2) you can specify an unknown dimension as -1
    P= np.reshape(M,(-1,1))

    return P

def b8(n):
    #Given:
    #   an integer n
    #Return:
    #   a nx(2n) matrix of ones
    #Hint:
    #   data type not understood with calling np.zeros/np.ones is guaranteed
    #   to be an issue where you passed in two arguments, not a tuple
    P = np.ones((n,2*n), dtype=float)

    return P

def b9(M):
    #Given:
    #   a matrix M where each entry is between 0 and 1
    #Return:
    #   a matrix S where S[i,j] = True if M[i,j] > 0.5
    #Hint: Trust python to do the right thing
    S = M > 0.5

    return S

def b10(n):
    #Given:
    #   an integer n
    #Return:
    #   the n-entry vector of 0, ..., n-1
    #Hint: range+np.array/np.arange
    result=np.arange(n)

    return result

def b11(t):
    #Given:
    #   a NxF matrix A
    #   a Fx1 vector v
    #Return:
    #   the matrix-vector product Av
    A, v = t
    P = A.dot(v)

    return P

def b12(t):
    #Given:
```

```
    #   a NxN matrix A, full rank
    #   a Nx1 vector v
    #Return:
    #   the inverse of A times v: A^-1 v
    A, v = t
    P = np.linalg.inv(A).dot(v)

    return P


def b13(t):
    #Given:
    #   a Nx1 vector u
    #   a Nx1 vector v
    #Return:
    #   the innner product u^T v
    #Hint:
    #   .T
    u, v = t
    P=np.transpose(u).dot(v)

    return P

def b14(v):
    #Given:
    #   a Nx1 vector v
    #Return:
    #   the L2-norm without calling np.linalg.norm
    #norm = (\sum_i=1^N v[i]**2)**0.5
    P = np.linalg.norm(v)

    return P

def b15(t):
    #Given:
    #   a NxF matrix M
    #   an integer i
    #Return:
    #   the ith row of M
    M, i = t
    P = M[i,:]

    return P

def b16(M):
    #Given:
    #   a NxF matrix M
    #Return:
    #   the sum of all the entrices of the matrix
    #Hint:
    #   np.sum
    P = np.sum(M)

    return P

def b17(M):
    #Given:
    #   a NxF matrix M
    #Return:
    #   a N-entry vector S where S[i] is the sum along row i of M
    #Hint:
    #   np.sum has an axis optional arg; note keepdims if you already know this
    P = np.sum(M, axis=1)

    return P
```

```python

def b18(M):
    #Given:
    #   a NxF matrix M
    #Return:
    #   a F-entry vector S where S[j] is the sum along column j of M
    #Hint: same as above
    P = np.sum(M, axis=0)

    return P

def b19(M):
    #Given:
    #   a NxF matrix M
    #Return:
    #   a Nx1 matrix S where S[i,1] is the sum along row i of M
    #Hint:
    #   Watch axis, keepdims
    P = np.sum(M, axis=1)
    P = P.reshape(-1,1)

    return P


def b20(M):
    #Given:
    #   a NxF matrix M
    #Return:
    #   a Nx1 matrix S where S[i] is the L2-norm of row i of M
    #Hint:
    #   Put it together
    P =np.linalg.norm(M,axis=1)
    P = P.reshape(-1,1)

    return P
```

```
[iunghuiui-MacBook-Pro:mastery_assignment ungheelee$ python3 run.py --allwarmups
Running b1
Running b2
Running b3
Running b4
Running b5
Running b6
Running b7
Running b8
Running b9
Running b10
Running b11
Running b12
Running b13
Running b14
Running b15
Running b16
Running b17
Running b18
Running b19
Running b20
Ran warmup tests
20/20 = 100.0
```