Factors and Loadings Common to Two or More Datasets

Bill Menke, December 9, 2024

I didn't include this variant of factor analysis in Environmental Data Analysis with MATLAB® or Python. But it's relatively straightforward and I outline how it works here.

In ordinary factor analysis, an $N \times M$ sample matrix $\mathbf{S}$ is represented as the product of factors $\mathbf{F}$ and loadings $\mathbf{C}$ via the rule $\mathbf{S} = \mathbf{CF}$. Singular value decomposition can be used to write

$$\mathbf{S} = \mathbf{U\Sigma V}^T \text{ with } \mathbf{C} \equiv \mathbf{U\Sigma} \text{ and } \mathbf{F} \equiv \mathbf{V}^T$$

Here, $\mathbf{U}$ is an N $\times$ N unary matrix of left eigenvectors, $\mathbf{\Sigma}$ is an $N \times M$ diagonal matrix of singular values and $\mathbf{V}$ is an M $\times$ M unary matrix of right eigenvectors. Note that

$$\mathbf{S}^T\mathbf{S} = (\mathbf{V\Sigma V U}^T)(\mathbf{U\Sigma V}^T) = \mathbf{V\Lambda V}^T \text{ with } \mathbf{\Lambda} \equiv \mathbf{\Sigma}^2$$

Because $\mathbf{U}$ and $\mathbf{V}$ are unary, $\mathbf{U}^T = \mathbf{U}^{-1}$ and $\mathbf{V}^T = \mathbf{V}^{-1}$, so we can solve for the diagonal matrix $\mathbf{\Lambda}$

$$\mathbf{V}^T(\mathbf{S}^T\mathbf{S})\mathbf{V} = \mathbf{\Lambda}$$

One way of defining the matrix $\mathbf{V}$ is to say that it is the unary matrix that diagonalizes $\mathbf{S}^T\mathbf{S}$. If one has two (or more) sample matrices

$$\mathbf{S}^{(1)} = \mathbf{U}^{(1)}\mathbf{\Sigma}^{(1)}\mathbf{V}^T \text{ and } \mathbf{S}^{(2)} = \mathbf{U}^{(2)}\mathbf{\Sigma}^{(2)}\mathbf{V}^T$$

that share – or approximately share - the same factor matrix $\mathbf{V}^T$, one could define the matrix $\mathbf{V}$ as the unary matrix that simultaneously diagonalizes – or approximately diagonalizes - them.

The well-known Joint Approximate Diagonalization under Orthogonality Constraints (JADOC) algorithm can be used to approximately diagonalize any number of symmetric matrices. One gives it $\mathbf{S}^{(1)}$ and $\mathbf{S}^{(2)}$ and it returns the matrix $\mathbf{V}$ for which

$$\mathbf{D}^{(1)} \equiv \mathbf{V}^T\mathbf{S}^{(1)T}\mathbf{S}^{(1)}\mathbf{V} \text{ and } \mathbf{D}^{(2)} \equiv \mathbf{V}^T\mathbf{S}^{(2)T}\mathbf{S}^{(2)}\mathbf{V}$$

are both approximately diagonal.

Example. Part 1: Building a test dataset

We build two sample matrices $\mathbf{S}^{(1)}$ and $\mathbf{S}^{(2)}$where each row is a different length-M times series. A pair of exemplary $M = 32$ matrices $\mathbf{V}^{(1)}$ and $\mathbf{V}^{(2)}$are constructed with identical first columns equal to a sinusoidal function of the form (Fig 1):

$$V_{n1} = \sin\left(\frac{pn\pi}{M}\right) \quad \text{with} \quad 1 \le p \le M$$

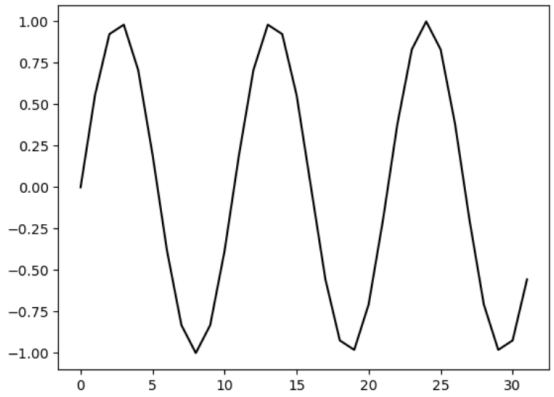

Fig. 1. Exemplary V_{n1} with $p = 5$

.

The rest of $\mathbf{V}^{(1)}$ and $\mathbf{V}^{(2)}$ are uncorrelated random noise with a standard deviation of unity. These factor matrices share one factor. We choose

$$\mathbf{\Sigma}^{(1)} = \mathbf{\Sigma}^{(2)} = \mathbf{\Sigma} \equiv \text{diag}[\sigma_1 \quad 1 \quad \cdots \quad 1] \quad \text{where} \quad \sigma_1 > 1$$

The matrices $\mathbf{U}^{(1)}$ and $\mathbf{U}^{(2)}$ are constructed from random noise where each vector has been separately bandpass filtered. The sample matrices (Fig 1) are then

$$\mathbf{S}^{(1)} = \mathbf{U}^{(1)}\mathbf{\Sigma}^{(1)}\mathbf{V}^{(1)T} + \mathbf{N} \quad \text{and} \quad \mathbf{S}^{(2)} = \mathbf{U}^{(2)}\mathbf{\Sigma}^{(2)}\mathbf{V}^{(2)T} + \mathbf{N}$$

where $\mathbf{N}$ is Normally-distributed random noise with standard deviation σ_S to all the elements. Although $\mathbf{V}^{(1)}$ and $\mathbf{V}^{(2)}$ are different, they share one common factor, and thus approximately are the desired form $\mathbf{S}^{(i)} \approx \mathbf{U}^{(i)}\mathbf{\Sigma}^{(i)}\mathbf{V}^{T}$.

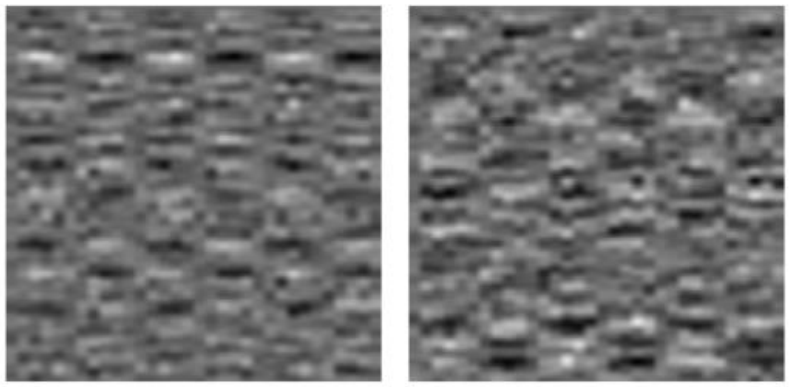

Fig. 2. The sample matrices $\mathbf{S}^{(1)}$ and $\mathbf{S}^{(2)}$, with $\sigma_1 = 10$ and $\sigma_S = 0.1$. The common factor is evident by the matching number of vertical stripes in the two images,

Part 2: Estimating the common factor.

We perform JADOC on the pair of matrices $\mathbf{S}^{(1)T}\mathbf{S}^{(1)}$ and $\mathbf{S}^{(2)T}\mathbf{S}^{(2)}$, to determine $\mathbf{V}^{(est)}$. We then compute the approximate diagonal matrices

$$\mathbf{D}^{(1)} = \mathbf{V}^{(est)T}\mathbf{S}^{(1)T}\mathbf{S}^{(1)}\mathbf{V}^{(est)} \quad \text{and} \quad \mathbf{D}^{(2)} = \mathbf{V}^{(est)T}\mathbf{S}^{(2)T}\mathbf{S}^{(2)}\mathbf{V}^{(est)}$$

The diagonals of these matrices are

$$\mathbf{d}^{(1)} \equiv \operatorname{diag}\mathbf{D}^{(1)} \quad \text{and} \quad \mathbf{d}^{(2)} \equiv \operatorname{diag}\mathbf{D}^{(2)}$$

We find the index, say m, corresponding to the largest diagonal element

$$m = \underset{i}{\operatorname{argmax}}\left[\left(d_i^{(1)}\right)^2 + \left(d_i^{(2)}\right)^2\right]$$

The estimate common factor (Fig. 3) is $V_{nm}^{(est)}$.

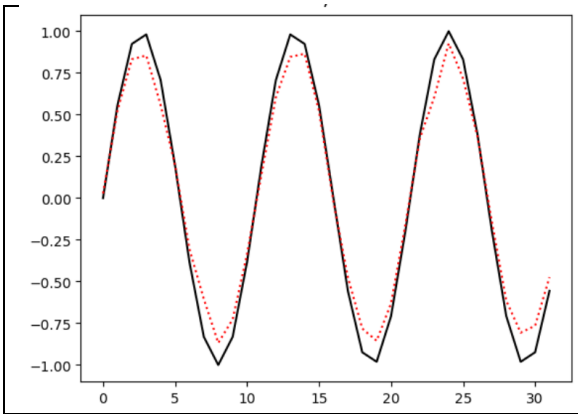

Fig. 3. True (black) and estimated (red) common factor

As $\mathbf{S} = \mathbf{C}\mathbf{V}^T$ the loadings are $\mathbf{C} = \mathbf{S}\mathbf{V}$ (Fig 4).

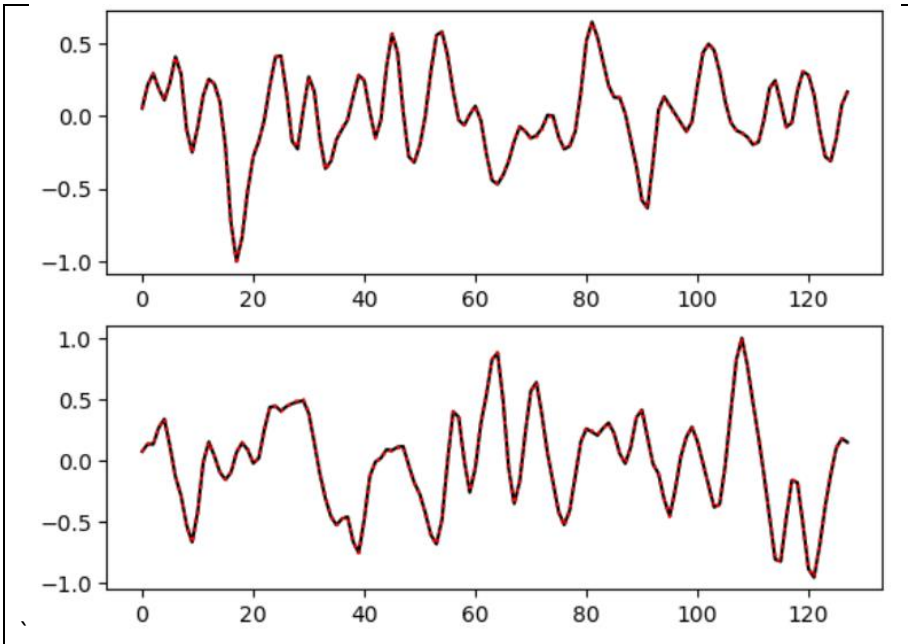

Fig. 4. True (black) and estimated (red) loadings of $\mathbf{S}^{(1)}$ (top) and $\mathbf{S}^{(2)}$ (bottom).

Addendum. In the above problem, we have assumed that there was one common factor and the loadings were all different. We could just have well assumed the converse, that the factors were all different and there was one common loading. That just amounts to transposing the sample matrices, so in effect, one is working with matrices of the form $\mathbf{S}^{(i)}\mathbf{S}^{(i)T}$ instead of $\mathbf{S}^{(i)T}\mathbf{S}^{(i)}$.

The JADOC algorithm applied to the pair of matrices $\mathbf{S}^{(1)}\mathbf{S}^{(1)T}$ and $\mathbf{S}^{(2)}\mathbf{S}^{(2)T}$ returns the matrix $\mathbf{U}^{(est)}$, which is the unnomalized loading and the unnomalized factors are $\mathbf{S}^{(1)T}\mathbf{U}^{(est)}$ and $\mathbf{S}^{(2)T}\mathbf{U}^{(est)}$. The normalizations are often not necessary, but are given by $\mathbf{C}^{(i)} = \mathbf{U}^{(est)}\mathbf{\Sigma}^{(i)}$ and $\mathbf{V}^{(i)} = \mathbf{S}^{(i)T}\mathbf{U}^{(est)}\left[\mathbf{\Sigma}^{(i)}\right]^{-1}$ where $\mathbf{\Sigma}^{(i)}$ is the square root of the diagonal part of $\mathbf{D}^{(i)}$.

In the following example, the factors $\mathbf{V}^{(1)}$ and $\mathbf{V}^{(2)}$ are composed of random vectors that have been band-passed filtered along the time axes. The singular values are the same as in the first example. The loadings are also random vectors that have been band-passed filtered along the time axes, except that vectors $U_{i1}^{(1)}$ and $U_{i1}^{(2)}$ are equal; that is, they are in common. As in the first example, noise with standard deviation σ_S is added to both sample matrices. The estimated unnormalized common loading (Fig.5) and the corresponding estimated unnormalized factors (Fig. 6) are close to the true ones.

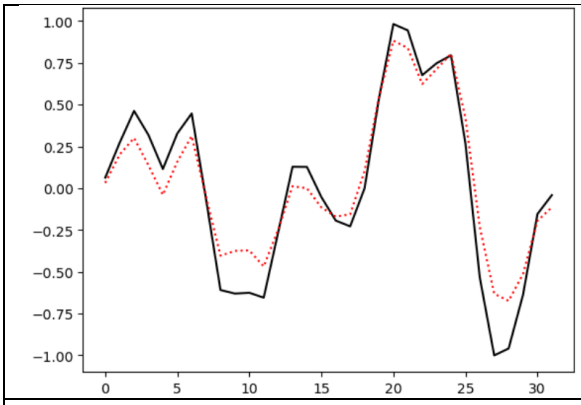

Fig. 5. True (black) and estimated (red) normalized common loading.

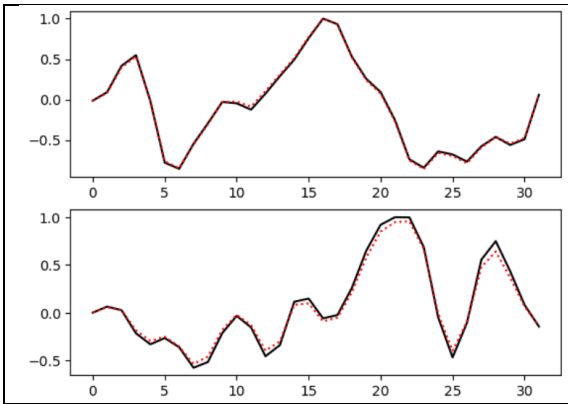

Fig 6. True (black) and estimated (red) normalized factors of the common loading.

Exemplary Python Code

```
%reset -f
from math import exp, pi, sin, cos, tan, sqrt, floor, ceil, log
import numpy as np
import scipy.linalg as la
import scipy.signal as sg
import matplotlib
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap

def eda_draw(*argv):
    bw = np.zeros((256,4));
    v = 0.9*(256 - np.linspace( 0, 255, 256 ))/255;
    bw[:,0] = v;
    bw[:,1] = v;
    bw[:,2] = v;
    bw[:,3] = np.ones(256);
    bwcmap = ListedColormap(bw);
    # size of plot
    W = 16;
    H = 4;
    fig1 = plt.figure(1);
    # figsize width and height in inches
    fig1.set_size_inches(W,H);
    ax1 = plt.subplot(1,1,1);
    plt.axis([0, W, -H/2, H/2]);
    plt.axis('off');
    LM = W/6;    # matrix width and heoght
    LV = W/40;   # vector width
    FS = 0.12;    # character width
    TO = 0.4;    # title vertical offset
    SP = 0.2;    # space between objects
    LS = 0.2;    # leading space
    p = LS; # starting x-position
    istitle=0; # flags presence of a title
    for a in argv:
        if isinstance(a,np.ndarray):
            sh = np.shape(a);
            if len(sh) == 1:  # conversion to nx1 array
                n = sh[0];
                m = 1;
                ap = a;
                a = np.zeros((n,1));
                a[:,0] = ap;
            else:
                n = sh[0];
                m = sh[1];
            if m==1:
                pold=p;
                left=p;
                right=p+LV;
                bottom=-LM/2;
                top=LM/2;
                plt.imshow( a, cmap=bwcmap, vmin=np.min(a), vmax=np.max(a),
extent=(left,right,bottom,top) );
                p = p+LV;
                pm = (p+pold)/2;
                if istitle:
                    plt.text(pm,-(LM/2)-TO,titlestr,horizontalalignment='center');
                    istitle=0;
                p = p+SP;
            else:
```

```
                pold=p;
                left=p;
                right=p+LM;
                bottom=-LM/2;
                top=LM/2;
                plt.imshow( a, cmap=bwcmap, vmin=np.min(a), vmax=np.max(a),
extent=(left,right,bottom,top) );
                p = p+LM;
                pm = (p+pold)/2;
                if istitle:
                    plt.text(pm,-(LM/2)-TO,titlestr,horizontalalignment='center');
                    istitle=0;
                p = p+SP;
        elif isinstance(a,str):
            ns = len(a);
            istitle=0;
            if( ns>=6 ):
                if 'title ' in a[0:6]:
                    istitle=1;
                    titlestr=a[6:];
            if( istitle != 1):
                plt.text(p,0,a);
                p = p + ns*FS + SP;
    plt.show();
    return 1;

# eda_draw function makes a "pictorial matrix equation"
# arguments are vectors, matrices and strings
# which are plotted in the order that the appear
# except that strings starting with 'title ' are plotted
# under the subseqeunt matrix or vector
# always returns a status of 1



# function to make a numpy N-by-1 column vector
# c=eda_cvec(v1, v2, ...) from a list of several
# array-like entities v1, v2, including a number
# a list of numbers, a tuple of numbers, an N-by-0 np array
# and a N-by-1 np array. The function
# also insures that, if v is an np array, that
# c is a copy, as contrasted to a view, of it
# It promotes integers to floats, and integers
# and floats to complex, by context.
# This version concatenates many argments,
# whereas c=eda_cvec1(v1) takes just one argiment.
# I recommend always using eda_cvec(v1, v2, ...)
def eda_cvec(*argv):
    t = int;
    Nt = 0;
    for a in argv:
        v = eda_cvec1(a);
        N,M = np.shape(v);
        Nt = Nt + N;
        if( N==0 ):
            continue; # skip vector of zero length
        if (t==int) and isinstance(v[0,0],float):
            t=float;
        elif isinstance(v[0,0],complex):
            t=complex;
    w = np.zeros((Nt,1),dtype=t);
    Nt = 0;
    for a in argv:
```

```
        v = eda_cvec1(a);
        N,M = np.shape(v);
        w[Nt:Nt+N,0] = v[0:N,0];
        Nt = Nt + N;
    return w;

# function to make a numpy N-by-1 column vector
# c=eda_cvec1(v) from entity v that is array-like,
# including a number, a list of numbers, a tuple
# of numbers, an N-by-0 np array and a N-by1 np array.
# It promotes integers to floats, and integers
# and floats to complex, by context. The function
# also insures that, if v is an np array, that
# c is a copy, as contrasted to a view, of it.
# This version takes just one input argment.
# whereas c=eda_cvec(v1,v2,...) concatenates
# many argiments.
def eda_cvec1(v):
    if isinstance(v, int) or isinstance(v, np.int32):
        w = np.zeros((1,1),dtype=int);
        w[0,0] = v;
        return w;
    elif isinstance(v, float):
        w = np.zeros((1,1),dtype=float);
        w[0,0] = v;
        return w;
    elif isinstance(v, complex):
        w = np.zeros((1,1),dtype=complex);
        w[0,0] = v;
        return w;
    elif isinstance(v, np.ndarray):
        s = np.shape(v);
        if len(s) == 1:
            return np.copy(np.reshape(v,(s[0],1)));
        else:
            [r,c]=s;
            if( c==1 ):
                return(np.copy(v));
            elif(r==1):
                return(np.copy(v.T));
            else:
                raise TypeError("eda_cvec: %d by %d ndarray not allowed" % (r, c));
    elif isinstance(v, list):
        r = len(v);
        t = int;
        for vi in v:
            if isinstance(vi,int) or isinstance(v, np.int32):
                pass;
            elif isinstance(vi,float):
                t=float;
            elif isinstance(vi,complex):
                t=complex;
                break;
            else:
                raise TypeError("eda_cvec: list contains unsupported type %s" %
type(vi));
        w = np.zeros((r,1),dtype=t);
        w[:,0] = v;
        return w;
    elif isinstance(v, tuple):
        r = len(v);
        t = int;
        for vi in v:
```

```
            if isinstance(vi,int) or isinstance(v, np.int32):
                pass;
            elif isinstance(vi,float):
                t=float;
            elif isinstance(vi,complex):
                t=complex;
                break;
            else:
                raise TypeError("eda_cvec: tuple contains unsupported type %s" %
type(vi));
        w = np.zeros((r,1),dtype=t);
        w[:,0] = v;
        return w;
    else:
        raise TypeError("eda_cvec: %s not supported" % type(v));



def chebyshevfilt(d, Dt, flow, fhigh):
    # chebyshevfilt
    # chebyshev IIR bandpass filter
    # d - input array of data
    # Dt - sampling interval
    # flow - low pass frequency, Hz
    # fhigh - high pass frequency, Hz
    # dout - output array of data
    # u - the numerator filter
    # v - the denominator filter
    # these filters can be used again using dout=filter(u,v,din);

    # make sure input timeseries is a column vector
    s = np.shape(d);
    N = s[0];
    if(N==1):
        dd = np.zeros((N,1));
        dd[:,0] = d;
    else:
        dd=d;

    # sampling rate
    rate=1/Dt;

    # ripple parameter, set to ten percent
    ripple=0.1;

    # normalise frequency
    fl=2.0*flow/rate;
    fh=2.0*fhigh/rate;

    # center frequency
    cf = 4 * tan( (fl*pi/2) ) * tan( (fh*pi/2) );

    # bandwidth
    bw = 2 * ( tan( (fh*pi/2) ) - tan( (fl*pi/2) ) );

    # ripple parameter factor
    rpf = sqrt((sqrt((1.0+1.0/(ripple*ripple))) + 1.0/ripple));
    a = 0.5*(rpf-1.0/rpf);
    b = 0.5*(rpf+1.0/rpf);

    u=np.zeros((5,1));
    v=np.zeros((5,1));
    theta = 3*pi/4;
```

```
    sr = a * cos(theta);
    si = b * sin(theta);
    es = sqrt(sr*sr+si*si);
    tmp= 16.0 - 16.0*bw*sr + 8.0*cf + 4.0*es*es*bw*bw - 4.0*bw*cf*sr + cf*cf;
    v[0,0] = 1.0;
    v[1,0] = 4.0*(-16.0 + 8.0*bw*sr - 2.0*bw*cf*sr + cf*cf)/tmp;
    v[2,0] = (96.0 - 16.0*cf - 8.0*es*es*bw*bw + 6.0*cf*cf)/tmp;
    v[3,0] = (-64.0 - 32.0*bw*sr + 8.0*bw*cf*sr + 4.0*cf*cf)/tmp;
    v[4,0] = (16.0 + 16.0*bw*sr + 8.0*cf + 4.0*es*es*bw*bw + 4.0*bw*cf*sr +
cf*cf)/tmp;
    tmp = 4.0*es*es*bw*bw/tmp;
    u[0,0] = tmp;
    u[1,0] = 0.0;
    u[2,0] = -2.0*tmp;
    u[3,0] = 0.0;
    u[4,0] = tmp;

    dout = eda_cvec(sg.lfilter(u.ravel(),v.ravel(),dd.ravel()));
    return (dout,u,v);


def eda_timedelay( uA, uB, Dt ):
# time delay between two similar timeseries, by
# cross-correlation followed by parabilic refinment.
# delay is positive when pulse on B is later than pulse on A
# input:
# uA, uB, two signals to be compared, as N-by-1 np.arrays
#         should have zero mean
#         if of unequal length, the shorter is zero padded
# Dt, sampling of both signals
# output
# tBmA_est, estimated time delay, positive when dB is delated with respect to uA
# Cmax_est, cross-correlation at estimated delay


    # cross correlate
    c = eda_cvec( np.correlate(uA.ravel(), uB.ravel(), mode='full') );
    Nc,i = np.shape(c);

    # rough estimate of lag
    NA, i = np.shape(uA);  # length of u
    NB, i = np.shape(uB);  # length of v
    cmax = np.max(c);
    icmax = np.argmax(c);
    tBmArough = -Dt * (icmax-NB+1);

    # parabolic refinement
    # (c-cmax) = m(1) + m(2) dt + m(3) dt^2
    # d(c-cmax)/ddt = 0 = m(2) + 2 m(3) dt
    # dt = -0.5*m(2)/m(3)
    # c = cmax + m(1) + m(2) dt + m(3) dt^2
    Dt2 = Dt**2;
    g1 =     eda_cvec([1.0, 1.0, 1.0]);
    g2 =  Dt*eda_cvec([-1.0, 0.0, 1.0]);
    g3 = Dt2*eda_cvec([1.0, 0.0, 1.0]);
    G = np.concatenate( (g1,g2,g3), axis=1);
    d = eda_cvec( [ c[icmax-1,0]-cmax, c[icmax,0]-cmax, c[icmax+1,0]-cmax ] );
    m = la.solve( np.matmul(G.transpose(),G), np.matmul(G.transpose(),d) );
    dt = -0.5*m[1,0]/m[2,0];
    Cmax_est = cmax + m[0,0] + m[1,0]*dt + m[2,0]*Dt2;
    tBmA_est = tBmArough - dt;

    return tBmA_est, Cmax_est;
```

```
import scipy.linalg
from scipy.sparse import linalg
import numpy as np
import time
from numba import njit,prange

def PerformJADOC(mC,mB0=None,iT=100,iTmin=10,dTol=1E-4,dTauH=1E-2,dAlpha=0.9,\
                 iS=None):
    """Joint Approximate Diagonalization under Orthogonality Constraints
    (JADOC)

    Authors: Ronald de Vlaming and Eric Slob
    Repository: https://www.github.com/devlaming/jadoc/

    Input
    ------
    mC : np.ndarray with shape (iK, iN, iN)
        iK Hermitian iN-by-iN matrices to jointly diagonalize

    mB0 : np.ndarray with shape (iN, iN), optional
        starting value for unitary transformation matrix such
        that mB@mC[i]@(mB.conj().T) is approximately diagonal for all i

    iT : int, optional
        maximum number of iterations; default=100

    iTmin : int, optional
        minimal number of iterations before convergence is tested; default=10

    dTol : float, optional
        stop if average magnitude elements gradient<dTol; default=1E-4

    dTauH : float, optional
        minimum value of second-order derivatives; default=1E-2

    dAlpha : float, optional
        regularization strength between zero and one; default=0.9

    iS : int, optional
        replace mC[i] by rank-iS approximation; default=None
        (set to ceil(iN/iK) under the default value)

    Output
    ------
    mB : np.ndarray with shape (iN, iN)
        unitary matrix such that mB@mC[i]@(mB.conj().T) is
        approximately diagonal for all i
    """
    print("Starting JADOC")
    (iK,iN,_)=mC.shape
    if iS is None:
        iS=(iN/iK)
        if (iS-int(iS))>0: iS=int(iS)+1
        else: iS=int(iS)
    if iS==iN: print("Computing decomposition of input matrices")
    elif iS>iN:
        raise ValueError("Desired rank (iS) exceeds dimensionality" \
                         +" of input matrices (iN)")
    else: print("Computing low-dimensional approximation of input matrices")
    if mB0 is None: mB=np.eye(iN)
    elif mB0.shape!=(iN,iN):
        raise ValueError("Starting value transformation matrix" \
                         +" has wrong shape")
```

```
    else: mB=mB0
    bComplex=np.iscomplexobj(mC)
    if bComplex: mA=np.empty((iK,iN,iS),dtype="complex128")
    else: mA=np.empty((iK,iN,iS))
    print("Regularization strength = "+str(dAlpha))
    vAlphaLambda=np.empty(iK)
    for i in range(iK):
        mD=mC[i]-ConjT(mC[i])
        if bComplex: dMSD=(np.real(mD)**2).mean()+(np.imag(mD)**2).mean()
        else: dMSD=(mD**2).mean()
        if dMSD>np.finfo(float).eps:
            if bComplex:
                raise ValueError("Input matrices are not Hermitian")
            else:
                raise ValueError("Input matrices are not real symmetric")
        if iS<iN:
            (vD,mP)=linalg.eigsh(mC[i],k=iS)
        else:
            (vD,mP)=np.linalg.eigh(mC[i])
        vD=abs(vD)
        vAlphaLambda[i]=dAlpha*((vD.sum())/iN)
        mA[i]=((1-dAlpha)**0.5)*mP*(np.sqrt(vD)[None,:])
        if mB0 is not None: mA[i]=np.dot(mB,mA[i])
    (mP,vD,mC)=(None,None,None)
    print("Starting quasi-Newton algorithm with line search (golden section)")
    bConverged=False
    for t in range(iT):
        (dLoss,mDiags,dRMSG,mU)=ComputeLoss(mA,vAlphaLambda,bComplex,dTauH)
        if dRMSG<dTol and t>=iTmin:
            bConverged=True
            break
        dStepSize=PerformGoldenSection(mA,mU,mB,vAlphaLambda,bComplex)
        print("ITER "+str(t)+": L="+str(round(dLoss,3))+", RMSD(g)=" \
              +str(round(dRMSG,6))+", step="+str(round(dStepSize,3)))
        (mB,mA)=UpdateEstimates(mA,mU,mB,dStepSize)
    if not(bConverged):
        print("WARNING: JADOC did not converge. Reconsider data or thresholds")
    print("Returning transformation matrix B")
    return mB

def ComputeLoss(mA,vAlphaLambda,bComplex,dTauH=None,bLossOnly=False):
    if bComplex:
        mDiags=((np.real(mA)**2).sum(axis=2))+((np.imag(mA)**2).sum(axis=2))\
            +vAlphaLambda[:,None]
    else:
        mDiags=((mA**2).sum(axis=2))+vAlphaLambda[:,None]
    (iK,iN,iS)=mA.shape
    dLoss=0.5*(np.log(mDiags).sum())/iK
    if bLossOnly:
        return dLoss
    else:
        if bComplex:
            mF=np.zeros((iN,iN),dtype="complex128")
            mF=ComputeFComplex(mF,mA,mDiags,iK,iN)
        else:
            mF=np.zeros((iN,iN))
            mF=ComputeFReal(mF,mA,mDiags,iK,iN)
        mG=(mF-ConjT(mF))
        if bComplex:
            dRMSG=np.sqrt((((np.real(mG)**2).sum())+((np.imag(mG)**2).sum()))\
                          /(iN*(iN-1)))
        else:
            dRMSG=np.sqrt(((mG**2).sum())/(iN*(iN-1)))
```

```
        mH=(mDiags[:,:,None]/mDiags[:,None,:]).mean(axis=0)
        mH=mH+mH.T-2.0
        mH[mH<dTauH]=dTauH
        mU=-mG/mH
        return dLoss,mDiags,dRMSG,mU

@njit
def ComputeFComplex(mF,mA,mDiags,iK,iN):
    for i in prange(iK):
        vDiags=(mDiags[i]).reshape((iN,1))
        mF+=np.dot(mA[i]/vDiags,mA[i].conj().T)
    mF=mF/iK
    return mF

@njit
def ComputeFReal(mF,mA,mDiags,iK,iN):
    for i in prange(iK):
        vDiags=(mDiags[i]).reshape((iN,1))
        mF+=np.dot(mA[i]/vDiags,mA[i].T)
    mF=mF/iK
    return mF

def PerformGoldenSection(mA,mU,mB,vAlphaLambda,bComplex):
    dTheta=2/(1+(5**0.5))
    iIter=0
    iMaxIter=15
    iGuesses=4
    (dStepLB,dStepUB)=(0,1)
    bLossOnlyGold=True
    (iK,iN,iS)=mA.shape
    mR=scipy.linalg.expm(mU)
    if bComplex: mAS=np.empty((iGuesses,iK,iN,iS),dtype="complex128")
    else: mAS=np.empty((iGuesses,iK,iN,iS))
    mAS[0]=mA.copy()
    mAS[1]=RotateData(mR,mA.copy())
    mAS[2]=(1-dTheta)*mAS[1]+dTheta*mAS[0]
    mAS[3]=(1-dTheta)*mAS[0]+dTheta*mAS[1]
    (mA,mR)=(None,None)
    dLoss2=ComputeLoss(mAS[2],vAlphaLambda,bComplex,bLossOnly=bLossOnlyGold)
    dLoss3=ComputeLoss(mAS[3],vAlphaLambda,bComplex,bLossOnly=bLossOnlyGold)
    while iIter<iMaxIter:
        if (dLoss2<dLoss3):
            mAS[1]=mAS[3]
            mAS[3]=mAS[2]
            dLoss3=dLoss2
            dStepUB=dStepLB+dTheta*(dStepUB-dStepLB)
            mAS[2]=mAS[1]-dTheta*(mAS[1]-mAS[0])
            dLoss2=ComputeLoss(mAS[2],vAlphaLambda,bComplex,\
                               bLossOnly=bLossOnlyGold)
        else:
            mAS[0]=mAS[2]
            mAS[2]=mAS[3]
            dLoss2=dLoss3
            dStepLB=dStepUB-dTheta*(dStepUB-dStepLB)
            mAS[3]=mAS[0]+dTheta*(mAS[1]-mAS[0])
            dLoss3=ComputeLoss(mAS[3],vAlphaLambda,bComplex,\
                               bLossOnly=bLossOnlyGold)
        iIter+=1
    return np.log(1+(dStepLB*(np.exp(1)-1)))

def UpdateEstimates(mA,mU,mB,dStepSize):
    mR=scipy.linalg.expm(dStepSize*mU)
    mB=np.dot(mR,mB)
```

```
    mA=RotateData(mR,mA)
    return mB,mA

@njit
def RotateData(mR,mData):
    iK=mData.shape[0]
    for i in prange(iK):
        mData[i]=np.dot(mR,mData[i])
    return mData

def ConjT(mA):
    if np.iscomplexobj(mA):
        return mA.conj().T
    else:
        return mA.T

def SimulateData(iK,iN,iR,dAlpha,bComplex=False,bPSD=True):
    if bComplex: sType1="Hermitian "
    else: sType1="real symmetric "
    if bPSD: sType2="positive (semi)-definite "
    else: sType2=""
    print("Simulating "+str(iK)+" distinct "+str(iN)+"-by-"+str(iN)+" " \
          +sType1+sType2+"matrices with alpha="+str(dAlpha) \
              +", for run "+str(iR))
    iMainSeed=15348091
    iRmax=10000
    if iR>=iRmax:
        return
    rngMain=np.random.default_rng(iMainSeed)
    vSeed=rngMain.integers(0,iMainSeed,iRmax)
    iSeed=vSeed[iR]
    rng=np.random.default_rng(iSeed)
    if bComplex:
        mX=rng.normal(size=(iN,iN))+1j*rng.normal(size=(iN,iN))
        mC=np.empty((iK,iN,iN),dtype="complex128")
    else:
        mX=rng.normal(size=(iN,iN))
        mC=np.empty((iK,iN,iN))
    for i in range(0,iK):
        if bComplex:
            mXk=rng.normal(size=(iN,iN))+1j*rng.normal(size=(iN,iN))
        else:
            mXk=rng.normal(size=(iN,iN))
        mXk=dAlpha*mX+(1-dAlpha)*mXk
        mR=scipy.linalg.expm(mXk-ConjT(mXk))
        vD=rng.normal(size=iN)
        if bPSD:
            vD=vD**2
        mC[i]=np.dot(mR*(vD[None,:]),ConjT(mR))
    return mC

def Test():
    iK=5
    iN=500
    iR=1
    dAlpha=0.9
    mC=SimulateData(iK,iN,iR,dAlpha)
    dTimeStart=time.time()
    mB=PerformJADOC(mC,dAlpha=.95,dTol=1E-5,iT=1000)
    dTime=time.time()-dTimeStart
    print("Runtime: "+str(round(dTime,3))+" seconds")
    mD=np.empty((iK,iN,iN))
    for i in range(iK):
```

```
        mD[i]=np.dot(np.dot(mB,mC[i]),mB.T)
    dSS_C=0
    dSS_D=0
    for i in range(iK):
        mOffPre=mC[i]-np.diag(np.diag(mC[i]))
        mOffPost=mD[i]-np.diag(np.diag(mD[i]))
        dSS_C+=(mOffPre**2).sum()
        dSS_D+=(mOffPost**2).sum()
    dRMS_C=np.sqrt(dSS_C/(iN*(iN-1)*iK))
    dRMS_D=np.sqrt(dSS_D/(iN*(iN-1)*iK))
    print("Root-mean-square deviation off-diagonals before transformation: " \
          +str(round(dRMS_C,6)))
    print("Root-mean-square deviation off-diagonals after transformation: " \
          +str(round(dRMS_D,6)))

N = 128;  # rows of S
M = 32;   # cols of S

print("There are two N by M sample matrices, S1 and S2");
print("with N: %d M: %d" % (N,M) );
print("each row of a sample matrix is a time series");
print("each column of the sample matrix is a different time series");
print("e.g. a years worth of time series each a day long");
print("  ");

print("The sample matrices have factorization");
print("S1 = U1 L1 V.T + noise");
print("S2 = U2 L2 V.T + noise");
print("that share the same factors V.T but different loadings U1 L1 and U2 L2");
print("  ");

print("Note that (S.T S) = V L1 U.T U L V.T = V L**2 V.T");
print("and that V.T (S.T S) V = L**2 is diaginal");
print("One way of defining the factors is to say that they diagonalize (S.T S)");
print("When one has two sample matrices, S1 and S2, one way of defining the common
factors");
print("is to say that they simultaneously (but apprximatelty) diagonalize (S1.T S1)
and (S2.T S2)" );
print("this code used the JADOC algorithm to perform the simultaneously (but
apprximatelty) diaginalization");

Dt = 1.0;
t = eda_cvec( Dt * np.linspace( 0, N-1, N ) )
print("The rows of the sample matrix are time series of length M");
print("  ");

P = 6;
print("Part 1: make synthetic sample matrices");
print("Factors: factor 0 is a sinusoid with jsine<%d half-wavelengths" % (P));
print("         the rest of the factors is random noise");
print("         we use two matrices V1 and V2 which the first vector");
print("         but not the rest");
print("  ");
# eigenvectors V with every column a vector
# most vectors are ranom noise but the first one is a sine
# with the number of oscilaltions randomly chosen
V1 = np.random.normal(loc=0.0, scale=1.0, size=(M,M));
V2 = np.random.normal(loc=0.0, scale=1.0, size=(M,M));

K = np.random.permutation( np.arange(P) ); # randonly choose one of the first P
jsine = K[0];
for i in range(M):
    V1[i,0] = sin( ((jsine+1)*pi*i)/M );
```

```
    V2[i,0] = sin( ((jsine+1)*pi*i)/M );

eda_draw(V1,V2);
print("Caption: V1, V2", np.shape(V1),  np.shape(V2));
print(" ");

plt.figure();
plt.subplot(1,1,1);
plt.plot(t[0:M,0:1], V1[0:M,0:1], 'k-' );
plt.show();
print("Caption sinusoidal eigenvector %d looks list this" % (jsine) );
print("  ");

sval = 10.0;
print("The first singular value is %.2f, the rest are all unity" % (sval) );
print("  ");
# singular values L1 and L2,  # all the same, except for the first
L1 = np.zeros((M,M));
L2 = np.zeros((M,M));
for i in range(M):
    if i==0:
        L1[i,i] = sval;
        L2[i,i] = sval;
    else:
        L1[i,i] = 1.0;
        L2[i,i] = 1.0;

print( np.diag(L1) );
print( np.diag(L2) );
print("Caption: singular values L1, L2");
print(" ");

print("Caption: The eigenvectors U1, U2 are random bandlimited times series");
print(" ");
# eigenvalues U, every column is a vector
# start them out as random
U1 = np.random.normal( loc=0.0, scale = 1.0, size=(N,M) );
U2 = np.random.normal( loc=0.0, scale = 1.0, size=(N,M) );
# but now filter them so that they have temporal correlation
fny = 1.0/(2.0*Dt);
flow  = 0.05*fny;
fhigh = 0.15*fny;
for i in range(M):
    u, uu, vv = chebyshevfilt( U1[0:N,i:i+1], Dt, flow, fhigh);
    U1[0:N,i:i+1] = u;
    u, uu, vv = chebyshevfilt( U2[0:N,i:i+1], Dt, flow, fhigh);
    U2[0:N,i:i+1] = u;

eda_draw(U1, U2);
print("Caption: U1, U2");
print("  ");

S1 = np.matmul( U1, np.matmul(L1, V1.T ) );
S2 = np.matmul( U2, np.matmul(L2, V2.T ) );

# add averall random noise
sigmaS = 0.1;
print("random noise of %e added to S1 and S2" % (sigmaS) );
S1 = S1 + np.random.normal(loc=0.0,scale=sigmaS,size=(N,M) );
S2 = S2 + np.random.normal(loc=0.0,scale=sigmaS,size=(N,M) );

eda_draw(S1, S2);
print("Caption: S1, S2", np.shape(S1), np.shape(S2) );
```

```
print("  ");

print("Caption: The algorithm is based in approximateley diaginalizing S1.T*S1 and
S2.T*S2", np.shape(S1), np.shape(S2) );
print("  ");

S1TS1 = np.matmul( S1.T, S1 );
S2TS2 = np.matmul( S2.T, S2 );
print("shape S1TS1", np.shape(S1TS1) );
print("shape S2TS2", np.shape(S2TS2) );
print("  ");

C = np.zeros( (2,M,M) );
for i in range(M):
    for j in range(M):
        C[0,i,j] = S1TS1[i,j];
        C[1,i,j] = S2TS2[i,j];

print("combined matrices: ", np.shape(C) );
print("  ");
print("Caption: The algorithm uses PerformJADOC to find the matrix Vest");
print("that simultaneously but approximately diagionalizes S1TS1 and S2TS2");
print("  ");

Vest = PerformJADOC( C ).T;

eda_draw(Vest);
print("Caption: rotation matrix Vest: ", np.shape(Vest) );
print("  ");

print("Caption: the matrix D = Vest.T * (S.T*S) * Vest is approxiimately diagoinal" );
print("  ");

D1 = np.matmul( Vest.T, np.matmul( S1TS1, Vest ) );
D2 = np.matmul( Vest.T, np.matmul( S2TS2, Vest ) );
eda_draw(D1,D2);
print("Caption: D1, D2", np.shape(D1), np.shape(D2) );

diag1 = np.diag(D1);
diag2 = np.diag(D2);
idiag = np.argmax( np.power(diag1,2) + np.power(diag2,2));
print("max combinaed diagional at position %d" % (idiag) );

plt.figure();
plt.subplot(1,1,1);
V1max = np.max(np.abs(V1[0:M,0:1]) );
Vestmax = np.max(np.abs(Vest[0:M,idiag:idiag+1]) );
x = V1[0:M,0:1]/V1max;
y = Vest[0:M,idiag:idiag+1]/Vestmax;
mysign = np.matmul(x.T,y)[0,0] / np.matmul(x.T,x)[0,0];
plt.plot(t[0:M,0:1], x, 'k-' );
plt.plot(t[0:M,0:1], mysign*y, 'r:' );
plt.title("%d" % (idiag) );
plt.show();
print("Caption: The factor common to the two sample matrices, True (black), Estimated
(red)" );

# S = C V.T  so   C = S*V

C1 = np.matmul(S1,V1);
C2 = np.matmul(S2,V2);

C1est = np.matmul(S1,Vest);
```

```
C2est = np.matmul(S2,Vest);

xx = C1[0:N,0:1];
xxmax = np.max(np.abs(xx));
xx = xx/xxmax;
yy = C1est[0:N,idiag:idiag+1];
yymax = np.max(np.abs(yy));
yy = yy/yymax;
mysign = np.matmul(xx.T,yy)[0,0]  / np.matmul(xx.T,xx)[0,0] ;
plt.figure();
plt.subplot(2,1,1);
plt.plot(t[0:N,0:1], xx, 'k-' );
plt.plot(t[0:N,0:1], mysign*yy, 'r:' );

xx = C2[0:N,0:1];
xxmax = np.max(np.abs(xx));
xx = xx/xxmax;
yy = C2est[0:N,idiag:idiag+1];
yymax = np.max(np.abs(yy));
yy = yy/yymax;
mysign = np.matmul(xx.T,yy)[0,0]  / np.matmul(xx.T,xx)[0,0] ;
plt.subplot(2,1,2);
plt.plot(t[0:N,0:1], xx, 'k-' );
plt.plot(t[0:N,0:1], mysign*yy, 'r:' );
plt.show();
print("Caption: True (black) and estinated (red) loadings of the common factors" );
print("  ");

# transposed case

N = 128;  # rows of S
M = 32;   # cols of S

Dt = 1.0;
t = eda_cvec( Dt * np.linspace( 0, N-1, N ) )

P = 6;

# factor are random, but bandpass filtered in time
V1 = np.random.normal(loc=0.0, scale=1.0, size=(M,M));
V2 = np.random.normal(loc=0.0, scale=1.0, size=(M,M));
fny = 1.0/(2.0*Dt);
flow  = 0.05*fny;
fhigh = 0.15*fny;
for i in range(M):
    u, uu, vv = chebyshevfilt( V1[0:M,i:i+1], Dt, flow, fhigh);
    V1[0:N,i:i+1] = u;
    u, uu, vv = chebyshevfilt( V2[0:M,i:i+1], Dt, flow, fhigh);
    V2[0:N,i:i+1] = u;
eda_draw(V1,V2);
print("Caption: V1, V2", np.shape(V1), np.shape(V2) );
print(" ");

sval = 10.0;
print("The first singular value is %.2f, the rest are all unity" % (sval) );
print("  ");
# singular values L1 and L2,  # all the same, except for the first
L1 = np.zeros((M,M));
L2 = np.zeros((M,M));
for i in range(M):
    if i==0:
        L1[i,i] = sval;
        L2[i,i] = sval;
```

```
    else:
        L1[i,i] = 1.0;
        L2[i,i] = 1.0;

print( np.diag(L1) );
print( np.diag(L2) );
print("Caption: singular values L1, L2");
print(" ");

print("Caption: The eigenvectors U1, U2 are random bandlimited times series");
print(" ");
# eigenvalues U, every column is a vector
# start them out as random
U1 = np.random.normal( loc=0.0, scale = 1.0, size=(N,M) );
U2 = np.random.normal( loc=0.0, scale = 1.0, size=(N,M) );
# but now filter them so that they have temporal correlation
fny = 1.0/(2.0*Dt);
flow  = 0.05*fny;
fhigh = 0.15*fny;
for i in range(M):
    u, uu, vv = chebyshevfilt( U1[0:N,i:i+1], Dt, flow, fhigh);
    U1[0:N,i:i+1] = u;
    u, uu, vv = chebyshevfilt( U2[0:N,i:i+1], Dt, flow, fhigh);
    U2[0:N,i:i+1] = u;
# now make one common vector
U2[0:N,0:1]=U1[0:N,0:1];

eda_draw(U1, U2);
print("Caption: U1, U2");
print("  ");

S1 = np.matmul( U1, np.matmul(L1, V1.T ) );
S2 = np.matmul( U2, np.matmul(L2, V2.T ) );

# add averall random noise
sigmaS = 0.1;
print("random noise of %e added to S1 and S2" % (sigmaS) );
S1 = S1 + np.random.normal(loc=0.0,scale=sigmaS,size=(N,M) );
S2 = S2 + np.random.normal(loc=0.0,scale=sigmaS,size=(N,M) );

eda_draw(S1, S2);
print("Caption: S1, S2", np.shape(S1), np.shape(S2) );
print("  ");

print("Caption: The algorithm is based in approximateley diaginalizing S1.T*S1 and
S2.T*S2", np.shape(S1), np.shape(S2) );
print("  ");

S1S1T = np.matmul( S1, S1.T );
S2S2T = np.matmul( S2, S2.T );
print("shape S1TS1", np.shape(S1S1T) );
print("shape S2TS2", np.shape(S2S2T) );
print("  ");

C = np.zeros( (2,N,N) );
for i in range(N):
    for j in range(N):
        C[0,i,j] = S1S1T[i,j];
        C[1,i,j] = S2S2T[i,j];

print("combined matrices: ", np.shape(C) );
print("  ");
print("Caption: The algorithm uses PerformJADOC to find the matrix Vest");
```

```
print("that simultaneously but approximately diagionalizes S1TS1 and S2TS2");
print("  ");

Uest = PerformJADOC( C ).T;

eda_draw(Vest);
print("Caption: rotation matrix Uest: ", np.shape(Uest) );
print("  ");

print("Caption: the matrix D = Uest.T * (S.T*S) * Uest is approxiimately diagoinal" );
print("  ");

D1 = np.matmul( Uest.T, np.matmul( S1S1T, Uest ) );
D2 = np.matmul( Uest.T, np.matmul( S2S2T, Uest ) );
eda_draw(D1,D2);
print("Caption: D1, D2", np.shape(D1), np.shape(D2) );

diag1 = np.diag(D1);
diag2 = np.diag(D2);
idiag = np.argmax( np.power(diag1,2) + np.power(diag2,2));
print("max combinad diagional at position %d" % (idiag) );

plt.figure();
plt.subplot(1,1,1);
U1max = np.max(np.abs(U1[0:M,0:1]) );
Uestmax = np.max(np.abs(Uest[0:M,idiag:idiag+1]) );
x = U1[0:M,0:1]/U1max;
y = Uest[0:M,idiag:idiag+1]/Uestmax;
mysign = np.matmul(x.T,y)[0,0] / np.matmul(x.T,x)[0,0];
plt.plot(t[0:M,0:1], x, 'k-' );
plt.plot(t[0:M,0:1], mysign*y, 'r:' );
plt.title("%d" % (idiag) );
plt.show();
print("Caption: The (unnormalized) loading common to the two sample matrices, True
(black), Estimated (red)" );

# unnormalized factors Vu = V sqrt(L)
# S = U sqrt(L) V.T
# S.T = V sqrt(L) U.T
# Vu = V sqrt(L) = S.T U

Vu1 = np.matmul(S1.T,U1);
Vu2 = np.matmul(S2.T,U2);

Vu1est = np.matmul(S1.T,Uest);
Vu2est = np.matmul(S2.T,Uest);

xx = Vu1[0:N,0:1];
xxmax = np.max(np.abs(xx));
xx = xx/xxmax;
yy = Vu1est[0:N,idiag:idiag+1];
yymax = np.max(np.abs(yy));
yy = yy/yymax;
mysign = np.matmul(xx.T,yy)[0,0]  / np.matmul(xx.T,xx)[0,0] ;
plt.figure();
plt.subplot(2,1,1);
plt.plot(t[0:M,0:1], xx, 'k-' );
plt.plot(t[0:M,0:1], mysign*yy, 'r:' );

xx = Vu2[0:N,0:1];
xxmax = np.max(np.abs(xx));
xx = xx/xxmax;
yy = Vu2est[0:N,idiag:idiag+1];
```

```
yymax = np.max(np.abs(yy));
yy = yy/yymax;
mysign = np.matmul(xx.T,yy)[0,0]  / np.matmul(xx.T,xx)[0,0] ;
plt.subplot(2,1,2);
plt.plot(t[0:M,0:1], xx, 'k-' );
plt.plot(t[0:M,0:1], mysign*yy, 'r:' );
plt.show();
print("Caption: True (black) and estimated (red) (unnormalized) factors of the common
loading" );
print("  ");
```