Section 6: Numerical Linear Algebra Operations Using Singular Value Decomp with NumPy and SciPy

In the previous section, we introduced numerical linear algebra routines in NumPy and SciPy. We discussed basic linear algebra operations -- vector norms, determinants, vector & matrix multplications, etc -- and how to solve systems of linear equations by writing the equations in the form $\mathbf{Ax} = \mathbf{b}$ and inverting $\mathbf{A}$ to solve for $\mathbf{x}$.

Now, we will build upon this foundation by discussing NumPy/SciPy's implementations of singular value decomposition here. In the next section, we will discuss eigenvector/eigenvalue decomposition and how these concepts are connected.

Implementation of Singular Value Decomposition (SVD) in Python

Singular value decomposition is another ultra-powerful linear algebra operation. For my research, it becomes especially powerful in cases where the data are noisy, such that a full matrix inversion tends to amplify noise (which, you know, is bad). Because we live in the real world, *all* data are noisy to some level, sometimes at a level that matters. Hence SVD.

SVD decomposes a matrix of m rows and n columns into the product of three matrices: the left singular matrix U, a diagonal matrix containing the singular values Σ, and a right singular matrix V^T ("the transpose of V"):

$$\mathbf{A} = \mathbf{U}\ \Sigma\ \mathbf{V}^T$$

Note that the dimensions of each of these matrices is different: the left singular matrix U has dimensions of mxm (i.e. a square matrix equal to the number of rows), the diagonal matrix containing singular values Σ have mxn dimensions, and the right singular matrix V^T has nxn dimensions. Note that, being a diagonal matrix, $\Sigma = 0$ for off-diagonal terms (i.e. $\Sigma_{ij} = 0$ for $i \neq j$); along the diagonal, Σ has entries σ, where $\sigma_i \geq 0$ and $\sigma_1 \geq \sigma_2$ and so on.

The columns m of U and columns n of V are "left singular vectors" and "right singular vectors", respectively.

In Numpy, the function to do SVD is `np.linalg.svd` , which returns a 3-element tuple of NumPy arrays. E.g.

```
aaa=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16],[17,18,19,20
]])

aaa.shape
#(5,4)

U,s,Vt=np.linalg.svd(aaa)

S=np.zeros(np.shape(aaa)) ##THIS IS IMPORTANT!
np.fill_diagonal(S,s)


#note, another way of doing this is ...

#S=np.zeros((aaa.shape[0],aaa.shape[1]))
#S[:aaa.shape[1],:aaa.shape[1]]=np.diag(s)


#Array values

#U
#array([[-0.09654784, -0.76855612,  0.60361934,  0.11849959,  0.14697464]
,
       [-0.24551564, -0.4896142 , -0.58187858, -0.58473646,  0.13964448],
       [-0.39448345, -0.21067228, -0.45158207,  0.7625265 , -0.12094203],
       [-0.54345125,  0.06826963,  0.23432255, -0.24484197, -0.76494794],
       [-0.69241905,  0.34721155,  0.19551877, -0.05144765,  0.59927085]]
)

#S
#array([[3.86226568e+01, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 2.07132307e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 7.60977226e-16, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 3.86063773e-16]])

#Vt
#array([[-0.44301884, -0.47987252, -0.51672621, -0.55357989],
       [ 0.70974242,  0.26404992, -0.18164258, -0.62733508],
       [-0.03307776, -0.36339946,  0.82603221, -0.42955499],
       [ 0.54672284, -0.75361849, -0.13293153,  0.33982718]])
```

```
#dimensionality

U.shape
#(5,5)

S.shape
#(5,4)

Vt.shape
#(4,4)

reconstructed_mat=np.dot(U,S).dot(Vt) #multiply each of these matrices to
gether

#confirm that this is equal to the original matrix aaa within machine pre
cision
residual=aaa-reconstructed_mat

residual
#array([[-1.77635684e-15,  4.44089210e-16, -2.66453526e-15,
         8.43769499e-15],
       [-8.88178420e-16, -3.55271368e-15, -2.66453526e-15,
         1.77635684e-15],
       [-1.77635684e-15, -3.55271368e-15, -3.55271368e-15,
         5.32907052e-15],
       [-1.77635684e-15, -5.32907052e-15, -5.32907052e-15,
         1.77635684e-15],
       [ 0.00000000e+00, -7.10542736e-15, -7.10542736e-15,
         7.10542736e-15]])
```

In the above example, note the cautions about the singular value matrix, s. What np.linalg.svd actually returns for s is a 1-D vector of singular values (my guess is that this is for speed/memory issues).

So, caution: Σ is a diagonal matrix with these values, but the matrix itself is 2-D (the off-diagonal elements are zero). To get the full middle matrix you have to create another array equal in dimensionality to the original matrix, set all elements originally to zero, and then fill the diagonals with s.

Before proceeding, there are a few other useful things to note.

First, 1) the matrices U and V are *orthogonal* matrices, so the column vectors making up U and V comprise an orthonormal (perpendicular) set of vectors, so they have a norm of 1.

We can confirm that the norm of each column vector is 1

```
np.linalg.norm(U[:,0])
#1.0
```

Second, 2) for orthogonal matrices like U and V, $UU^T = U^T U = I$ and $VV^T = V^T V = I$.

We can see this is the case as follows: e.g.

$U^T \; x \; U = (u_1 \; ... \; u_n \;)^T (u_1 \; ... \; u_n \;)$

$U^T \; x \; U = \begin{pmatrix} u_1 \\ ... \\ u_n \end{pmatrix} (u_1 \; ... \; u_n \;)$, which yields

$$\begin{pmatrix} u_1 . u_1 & ... & u_1 . u_n \\ ... & ... & ... \\ u_n . u_1 & ... & u_n . u_n \end{pmatrix}$$

Since the vectors making up U are pairwise perpendicular (i.e. orthogonal), then

$u_i . u_j = 1$ if $i = j$

else

$u_i . u_j = 0$ if $i \neq j$

We can confirm this using NumPy's dot product calculation

```
np.dot(U,U.T)
#array([[ 1.00000000e+00, -1.94004804e-16, -1.48891722e-16,
        -7.54556766e-18, -9.45675836e-18],
       [-1.94004804e-16,  1.00000000e+00,  1.56683683e-16,
         5.37058712e-17, -2.52897914e-16],
       [-1.48891722e-16,  1.56683683e-16,  1.00000000e+00,
        -8.97092254e-17, -3.41546853e-17],
       [-7.54556766e-18,  5.37058712e-17, -8.97092254e-17,
         1.00000000e+00,  9.49280878e-17],
       [-9.45675836e-18, -2.52897914e-16, -3.41546853e-17,
         9.49280878e-17,  1.00000000e+00]])
```

Third, 3) a further consequence of this property is that the inverse of U or V is equal to its transpose. E.g. $U^T = U^{-1}$.

We can see this as follows:

```
np.linalg.inv(U)
#array([[-0.09654784, -0.24551564, -0.39448345, -0.54345125, -0.69241905]
,
       [-0.76855612, -0.4896142 , -0.21067228,  0.06826963,  0.34721155],
       [ 0.60361934, -0.58187858, -0.45158207,  0.23432255,  0.19551877],
       [ 0.11849959, -0.58473646,  0.7625265 , -0.24484197, -0.05144765],
       [ 0.14697464,  0.13964448, -0.12094203, -0.76494794,  0.59927085]]
)


U.T
array([[-0.09654784, -0.24551564, -0.39448345, -0.54345125, -0.69241905],
       [-0.76855612, -0.4896142 , -0.21067228,  0.06826963,  0.34721155],
       [ 0.60361934, -0.58187858, -0.45158207,  0.23432255,  0.19551877],
       [ 0.11849959, -0.58473646,  0.7625265 , -0.24484197, -0.05144765],
       [ 0.14697464,  0.13964448, -0.12094203, -0.76494794,  0.59927085]]
)

uut=np.dot(U,U.T)
#array([[ 1.00000000e+00, -1.94004804e-16, -1.48891722e-16,
        -7.54556766e-18, -9.45675836e-18],
       [-1.94004804e-16,  1.00000000e+00,  1.56683683e-16,
         5.37058712e-17, -2.52897914e-16],
       [-1.48891722e-16,  1.56683683e-16,  1.00000000e+00,
        -8.97092254e-17, -3.41546853e-17],
       [-7.54556766e-18,  5.37058712e-17, -8.97092254e-17,
         1.00000000e+00,  9.49280878e-17],
       [-9.45675836e-18, -2.52897914e-16, -3.41546853e-17,
         9.49280878e-17,  1.00000000e+00]])
```

These points will end up being useful when we try to (pseudo-)invert matrices using SVD (see below).

(Pseudo-)Inverting Matrices Using SVD

SVD has particular power when it comes to matrix inversions. First, let's try to directly invert the matrix `aaa` and see what happens:

```
aaa.shape
#(5,4)

np.linalg.inv(aaa)

#Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in inv
  File "/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/site-pack
ages/numpy/linalg/linalg.py", line 540, in inv
    _assert_stacked_square(a)
  File "/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/site-pack
ages/numpy/linalg/linalg.py", line 203, in _assert_stacked_square
    raise LinAlgError('Last 2 dimensions of the array must be square')
numpy.linalg.LinAlgError: Last 2 dimensions of the array must be square
```

We get a Traceback error because (not surprisingly) the matrix `aaa` is not a square matrix. SVD allows us to work around that.

The SVD of a matrix leaves:

$\mathbf{A} = \mathbf{U}\ \Sigma\ \mathbf{V}^T$

What if we try to do the inversion in "SVD space"? I.e.

$(\mathbf{U}\ \Sigma\ \mathbf{V}^T)^{-1}$

Well, first remember matrix rules. Specifically ...

$(\mathbf{U}\ \Sigma\ \mathbf{V}^T)^{-1}$ --> $(\mathbf{V}^T)^{-1}\,\Sigma^{-1}\,\mathbf{U}^{-1}$

second, from item 3) above, that the inverse and transpose of an orthogonal matrix is the same thing and both U and V are orthogonal matrices. So $(\mathbf{V}^T)^{-1}$ --> $\mathbf{V}$. Thus, $\mathbf{V}\Sigma^{-1}\,\mathbf{U}^{-1}$.

Hence,

the "inverse", or more accurately the Moore-Penrose *pseudoinverse* (yes, *that* Penrose), of the SVD'd version of $\mathbf{A}$ is

$A^{\dagger} = \mathbf{V}\Sigma^{-1}\,\mathbf{U}^T$.

The function to compute the pseudoinverse in NumPy is `np.linalg.pinv` . For SciPy, it is `scipy.linalg.pinv` .

Now, hold on ... we started this section talking about how we cannot directly invert a non-square matrix. Yet we are still effectively tasked with that because Σ has dimensions mxn, while U is mxm and V^T is nxn. If m isn't the same as n we don't have a square matrix. So not invertible, right?

Not exactly ... again, we are dealing with a *pseudo-inversion* but the consequence is very much what we want.

The easiest way to figure out how Python works around this the square matrix problem by looking at the `np.linalg.pinv` source code itself (annotated a bit by me)

```
def pinv(a, rcond=1e-15, hermitian=False):
    a, wrap = _makearray(a)
    rcond = np.asarray(rcond)
    if _is_empty_2d(a):
        m, n = a.shape[-2:]
        res = np.empty(a.shape[:-2] + (n, m), dtype=a.dtype)
        return wrap(res)
    a = a.conjugate()
    u, s, vt = svd(a, full_matrices=False)

    # discard small singular values
    cutoff = rcond[..., np.newaxis] * np.amax(s, axis=-1, keepdims=True)
    large = s > cutoff
    s = np.divide(1, s, where=large, out=s)
    s[~large] = 0

    res = np.matmul(np.transpose(vt), np.multiply(s[..., np.newaxis], np.
transpose(u)))
    return wrap(res)


def _makearray(a):
    new = np.asarray(a)
    wrap = getattr(a, "__array_prepare__", new.__array_wrap__)
    return new, wrap
```

Basically, here is what we do ... we compute the SVD of the original matrix, do some Python trickery to turn single numbers into arrays, take 1 over each value for Σ (which is what the inverse actually does) and then *truncate* the matrix Σ: setting all values to 0 that are below our *rcond* number _relative to the largest singular value.

The `res` part is the trickiest, arguably, and it is easiest to go right to left. First, we do

`s[...,np.newaxis]`. The ellipsis means something like "for all dimensions except ___". E.g. in a 2x2x2x2 matrix called "largematrix", `largematrix[...,0]` will retain all first row elements in the 4th dimension. It really doesn't do much here in practical terms, though, as the syntax is equivalent to `s = s[:,np.newaxis]`, which gives us a (4,1) array:

```
s_version2=s[...,np.newaxis]
s_version2.shape
#(4,1)
```

Then we multiply this vector by the transpose of u (which has shape (4,5)), yielding a matrix with dimensions (4,5).

```
right_matrix=np.multiply(s[...,np.newaxis],np.transpose(u))
```

Then multiply the transpose of V^T (which is just V) by this (4,5) matrix.

```
pseudo_inv=np.matmul(np.transpose(vt),right_matrix)
pseudo_inv.shape
#(4,5)
```

This yields a (4,5) matrix. As our original matrix is (5,4), dimensionally this works out as a matrix that has been (pseudo-)inverted.

The full code comparing the canned pseudo-inverse from NumPy with our "manual" version is `def ex1` in the script `svd_pinv_demo.py` (remember, to run do `from svd_pinv_demo import ex1; ex1()`).

We repeat it here for completeness:

```
def ex1():


 aaa=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16],[17,18,19,2
0]])


#with np.linalg.svd

 aaa_inv=np.linalg.pinv(aaa)

 print("the NumPy-produced inverted matrix is \n")
 print(aaa_inv)
```

```
 #[[-2.30000000e-01 -1.45000000e-01 -6.00000000e-02  2.50000000e-02
   1.10000000e-01]
 [-8.50000000e-02 -5.25000000e-02 -2.00000000e-02  1.25000000e-02
   4.50000000e-02]
 [ 6.00000000e-02  4.00000000e-02  2.00000000e-02 -3.61867637e-17
  -2.00000000e-02]
 [ 2.05000000e-01  1.32500000e-01  6.00000000e-02 -1.25000000e-02
  -8.50000000e-02]]

#manual version

 new = np.asarray(aaa) #not really needed but we are following the NumPy
syntax

# wrap = getattr(aaa, "__array_prepare__", new.__array_wrap__) ### def no
t needed to demonstrate the point

 rcond=1e-15 #default value for pseudo-inverse
 rcond=np.asarray(rcond)
 new = new.conjugate()

#taking the SVD
 u,s,vt=np.linalg.svd(new,full_matrices=False)

 #print(u.shape,s.shape,vt.shape)
 cutoff = rcond[..., np.newaxis] * np.amax(s, axis=-1, keepdims=True)

 large = s > cutoff
 s = np.divide(1, s, where=large, out=s)
 s[~large] = 0

 s_version2=s[...,np.newaxis]

 right_matrix=np.multiply(s_version2,np.transpose(u))

 aaa_inv2=np.matmul(vt.T,right_matrix)

 print("the manually-produced inverted matrix is \n")
 print(aaa_inv2)

 #[[-2.30000000e-01 -1.45000000e-01 -6.00000000e-02  2.50000000e-02
   1.10000000e-01]
 [-8.50000000e-02 -5.25000000e-02 -2.00000000e-02  1.25000000e-02
   4.50000000e-02]
```

```
 [ 6.00000000e-02  4.00000000e-02  2.00000000e-02 -3.61867637e-17
  -2.00000000e-02]
 [ 2.05000000e-01  1.32500000e-01  6.00000000e-02 -1.25000000e-02
  -8.50000000e-02]]

 print("the residuals are ...")
 print(aaa_inv2-aaa_inv)

 #[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Truncated SVD in Python; Matrix Condition Number and Matrix Rank/Null Space

To do a head-to-head comparison with a direct inverse, let's make a 4 by 4 matrix and compare a direct inversion with an SVD. This is described in `def ex2():`, and we repeat the most important lines here.

```
aaa=np.array([[1.1,2.2,3.3,4.4],[5.5,6.6,7.7,8.7],[9.98,10.1,11.11,12.12]
,[13,14,15,16],[17,18,19,20]])
aaa=aaa[:4,:] #making a 4x4 matrix

print(aaa)
#[[ 1.1   2.2   3.3   4.4 ]
 [ 5.5   6.6   7.7   8.7 ]
 [ 9.98 10.1  11.11 12.12]
 [13.   14.   15.   16.  ]]
```

Then compute the difference between NumPy's regular inverse, its pseudo-inverse, and a manual pseudo-inverse.

```
print("the straightforward inverse yields")
print(aaa_invreg)

print("the NumPy pseudo-inverse yields")
print(aaa_inv)

print("the manual pseudo-inverse yields")
print(aaa_inv2)

print("the difference between the straightforward inverse and NumPy pseud
o-inverse are")
print(aaa_invreg-aaa_inv)
```

For the direct inverse we get

```
[[-3.43888321e-01 -4.37257068e-15  1.12359551e+00 -7.56554307e-01]
 [ 6.21807967e+00 -1.00000000e+01 -2.24719101e+00  5.42977528e+00]
 [-1.26166156e+01  2.00000000e+01  1.12359551e+00 -8.25655431e+00]
 [ 6.66666667e+00 -1.00000000e+01  9.97953281e-15  3.66666667e+00]]
```

For the NumPy pseudo-inverse we get

```
[[-3.43888321e-01 -2.09825694e-14  1.12359551e+00 -7.56554307e-01]
 [ 6.21807967e+00 -1.00000000e+01 -2.24719101e+00  5.42977528e+00]
 [-1.26166156e+01  2.00000000e+01  1.12359551e+00 -8.25655431e+00]
 [ 6.66666667e+00 -1.00000000e+01 -3.02435502e-14  3.66666667e+00]]
```

And the differences are

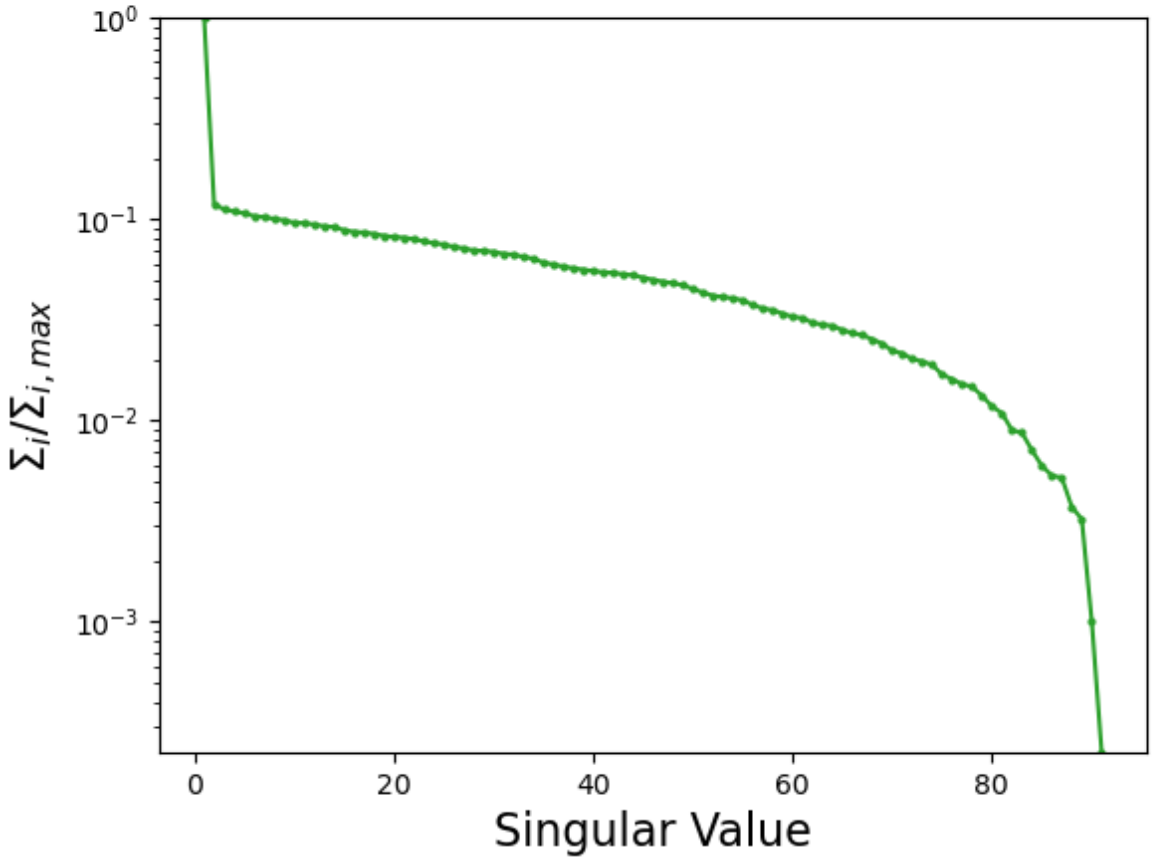
```
[[-1.01585407e-14  1.66099987e-14  2.22044605e-16 -5.99520433e-15]
 [-2.77999845e-13  4.20996571e-13  3.37507799e-14 -1.79412041e-13]
 [ 6.51922960e-13 -1.00541797e-12 -7.30526750e-14  4.24549285e-13]
 [-3.64153152e-13  5.63105118e-13  4.02230830e-14 -2.37143638e-13]]
```

This is tiny, comparable to the machine precision.

But now let's do a slightly different matrix: a 91x91 element matrix of random numbers. `aaa=np.random.rand(91,91)`. The matrix (and thus the inverse) will be different each time, but let's report the minimum and maximum singular value from the Σ diagonal matrix:

```
#maximum
[45.80273108]
#minimum
[0.01025289]
```

And now we can plot the *relative* value of the singular values (basically, dividing each s by the maximum s).



If we repeat this matrix generation a bunch of times we will get qualitatively similar answers. The point: we see one very large singular value, a bunch (60 or so?) about 0.02-0.1 times as large, and then a rapid tail at very small singular values.

A useful (for now, at least: see later) interpretation is that the first singular value is the dominant one in explaining the variance in the data, while very low-value singular values explain little of the variance and can be discarded without much loss of accuracy***.

***(An analogy, start with an ellipse with coordinates x and y and then transform to 3 dimensions (i.e. add a z direction). If the magnitude along the z direction is tiny compared to x and y, then this spheroid is essentially a 2-D ellipse).

Another thing becomes important when we try to *invert* matrices. Matrices with singular values of 0 are bad news (can't invert them). You can determine if a matrix is a singular matrix if the determinant of the matrix is 0. those with values *close* to 0 are also dangerous because it may not be possible to accurately compute them. In these cases, matrices are *ill-conditioned* because dividing by the singular values $1/s_i$ -- i.e. what you do when you invert Σ (Σ^{-1}) -- for s_i that are really close to zero will result in numerical errors.

The degree to which ill-conditioning prevents a matrix from being inverted accurately depends on the ratio of its largest to smallest singular value, a quantity known as the **condition number**:

$c.n. = s_i/s_n$.

In the example above, the condition number of Σ is nothing to be worried about from a numerical error standpoint (about 10^{-4}). But remember all those 10^{-15} numbers we saw when comparing the direct inverse to the pseudo-inverse? That's a sign that we are getting close to machine precision (which is $\sim 10^{-15}$--10^{-16} for floating-point operations in Python). In other words, if the condition number for your matrix is near the machine precision, then that's a sign you are risking propagating errors in your matrix inversions. Then *truncating* the Σ matrix at some threshold helps guard against numerical errors.

Another context where this useful is when your matrix is "noisy" (e.g. working with images). In such a case, a full matrix inversion will end up amplifying this noise. Truncated SVD therefore guards against this problem.

The *rank* of a matrix is equal to: (i) its number of linearly independent columns; (ii) its number of linearly independent rows (these end up being the same thing). You can think of the matrix rank as the dimensionality of the vector space spanned by its rows or its columns.

SVD gives you a particularly straightforward to identify the rank of the matrix: the rank is equal to the number of non-zero singular values of a matrix. We can identify the matrix rank in NumPy from `np.linalg.matrix_rank([name of matrix])`. The main keyword here is `tol` (e.g. for a matrix M, np.linalg.matrix_rank(M,tol=1e-15) gives its rank). Otherwise, what this does is that it calculates the rank of the matrix from the number of *non-zero* singular values. Conversely, the singular values equal to zero refer to the *null space* of the matrix.

For example, let's go back to the matrix aaa:

```
aaa=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16],[17,18,19,20
]])

rank=np.linalg.matrix_rank(aaa)

print(rank)
#2
```

We get the same answer if we chop off one of the rows of aaa (i.e. aaa=aaa[:4,:]). We can confirm by returning the matrix Σ

```
U,s,Vt=np.linalg.svd(aaa)
S=np.zeros(np.shape(aaa))
np.fill_diagonal(S,s)

S
#array([[5.35202225e+01, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 2.36342639e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 4.11380109e-15, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 6.65983308e-16],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00]])
```

There, there are two singular vlaues ($\sim$ 53.5 and 2.36) and then two that are $\sim$0 within machine precision. Hence, the rank of the matrix is 2.

So even though the matrix has 5 rows and 4 columns, its rank is two.

Now, try the same with the "edited" version of aaa which we used in this section to demonstrate truncated SVD

```
aaa=np.array([[1.1,2.2,3.3,4.4],[5.5,6.6,7.7,8.7],[9.98,10.1,11.11,12.12]
,[13,14,15,16]])

print(np.linalg.matrix_rank(aaa))
#4

#a check
U,s,Vt=np.linalg.svd(aaa)
S=np.zeros(np.shape(aaa))
np.fill_diagonal(S,s)

print(S)
#[[3.94670012e+01 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 2.28462303e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 4.02701645e-01 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 3.23542887e-02]]
```

The dimensions of this matrix about the same as the previous one but here we have four diagonal elements that are very much larger than $\sim 10^{-14}$--10^{-16}. Hence, the rank of this matrix is 4.

So then what happens to the matrix rank when we truncate Σ? It is *reduced*. E.g. if we truncate the previous matrix aaa (i.e. the one we just determined is a rank-4 matrix) at tol=0.05, then the rank is reduced to 3

```
np.linalg.matrix_rank(aaa,tol=0.05)
#3
```

This helps us interpret exactly what we are doing when we use truncated SVD for matrix inversions. I.e. truncating Σ produces a *reduced-rank* (or "lower-rank") approximation to the full matrix A. Since we can think of each one of the column vectors U or V^T (which have corresponding singular values) as dimensions, truncated SVD can be thought of as "dimensionality reduction".

---

tl;dr ...

So, we can perform *truncated* singular value decomposition by setting to zero all of the singular values below some threshold. In `np.linalg.pinv` this threshold is `rcond`. The number is normalized *relative* to the largest singular value. E.g. so if the largest singular value is 6 and you set `rcond=0.01`, then all singular values (i.e. all diagonal elements of Σ) less than 0.06 times the maximum value will bet set to zero. The rank of the matrix returns the number of non-

zero singular values; the null space corresponds to singular values = 0. Truncated SVD produces a lower-rank approximation to the original matrix A and can be thought of as a dimensionality reduction.

Practical Examples of SVD for Scientific Python

Exoplanet Direct Imaging (Modifying LOCI)

Recall the LOCI equation from the previous section, which is basically a least-squares solution to a system of linear equations. In direct imaging instead of directly solving **Ax**=**b**, we can decompose the **A** matrix as $\mathbf{U}\ \Sigma\ \mathbf{V}^T$.

Then to invert **A** (i.e. $\mathbf{A}^{-1}$) we compute $\mathbf{V}{\Sigma_t}^{-1}\ \mathbf{U}^T$, where the subscript t stands for *truncate*. I.e. we truncate Σ below some threshold (relative) singular value. There are multiple reasons why this may be useful, including guarding against the impact of detector noise, circumstellar disk emission (long story: basically, it causes lst-sq algorithms to converge to weird results).

Image Compression and Pattern Matching

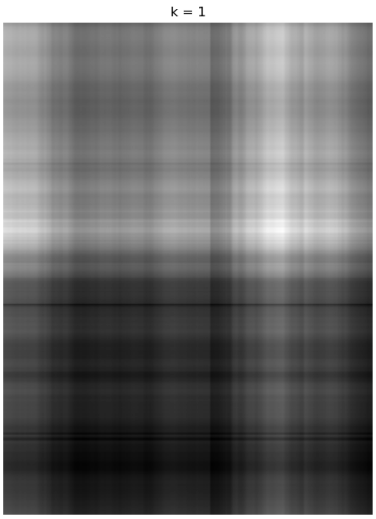
Related practical applications of SVD deals with the broad topics of image compression and feature matching. Note that this is very similar thing to the exoplanet topic but in a different research focus (least-squares solutions with imaging like LOCI depend on matching patterns -- i.e. speckles -- across an image set).

When we want to compress a file, we're always looking for the most efficient approach with the lowest amount of unnecessary data. The smaller the image, the less the cost of storage and transmission. The SVD Algorithm will help us by decomposing a given matrix (an image is a matrix with different values representing colors) into three matrices to finally represent the image with a smaller set of values.

That way, image compression will be achieved while preserving the important features that make the original picture.

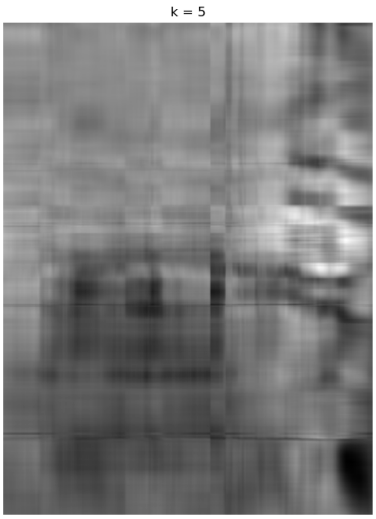
Here's an example of a picture of a celebration of one of my best friends. Look what happens to the image as we go from retaining only the largest singular value, to the first 5, and so on.

First singular value retained, only:



Hard to see what this is but we do know there are regions of the image that are bright and regions taht are dark.

First five:



Still unclear but we are starting to see the shape of something. There is more structure to the light and dark regions.

Twenty:

k = 20

Now we are getting warmer. This looks like a dog?

One hundred

k = 100

Two-fifty

k = 250

500

k = 500

We can see that truncating at the first singular value gets just the very rough levels of bright and dark right, and the image of my dog starts to show up slightly by k=5 and noticeably by k=20. By k=100, the differences with the original image (in the same folder) are minor.