Deep Learning Book Series · 2.8 Singular Value Decomposition

26-03-2018

This content is part of a series following the chapter 2 on linear algebra from the Deep Learning Book by Goodfellow, I., Bengio, Y., and Courville, A. (2016). It aims to provide intuitions/drawings/python code on mathematical theories and is constructed as my understanding of these concepts. You can check the syllabus in the introduction post.

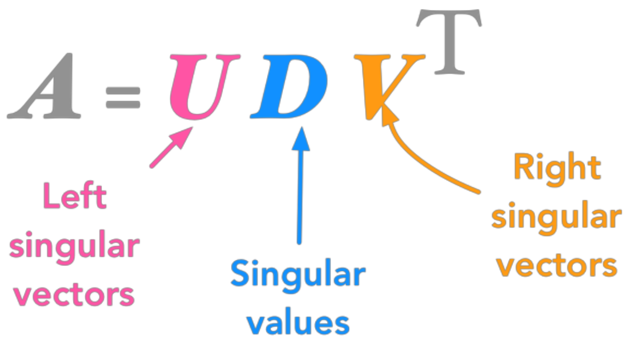
Introduction

We will see another way to decompose matrices: the Singular Value Decomposition or SVD. Since the beginning of this series, I emphasized the fact that you can see matrices as linear transformation in space. With the SVD, you decompose a matrix in three other matrices. You can see these new matrices as *sub-transformations* of the space. Instead of doing the transformation in one movement, we decompose it in three movements. As a bonus, we will apply the SVD to image processing. We will see the effect of SVD on an image of Lucy the goose (it is just a goose named Lucy…) so keep on reading!

2.8 Singular Value Decomposition

We saw in 2.7 that the eigendecomposition can be done only for square matrices. The way to go to decompose other types of matrices that can't be decomposed with eigendecomposition is to use **Singular Value Decomposition** (SVD).

We will decompose $\boldsymbol{A}$ into 3 matrices (instead of two with eigendecomposition):



The singular value decomposition

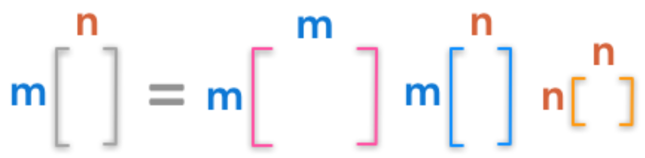
The matrices $\boldsymbol{U}$, $\boldsymbol{D}$, and $\boldsymbol{V}$ have the following properties:

- $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal matrices ($\boldsymbol{U}$T=$\boldsymbol{U}$−1 and $\boldsymbol{V}$T=$\boldsymbol{V}$−1; see 2.6 for more details about orthogonal matrices)

- $\boldsymbol{D}$ is a diagonal matrix (all 0 except the diagonal ; see 2.6). However $\boldsymbol{D}$ is not necessarily square.

The columns of $\boldsymbol{U}$ are called the left-singular vectors of $\boldsymbol{A}$ while the columns of $\boldsymbol{V}$ are the right-singular vectors of $\boldsymbol{A}$. The values along the diagonal of $\boldsymbol{D}$ are the singular values of $\boldsymbol{A}$.

Here are the dimensions of the factorization:



The dimensions of the singular value decomposition

The diagonal matrix of singular values is not square but have the shape of ***A***. Look at the example provided in the Numpy doc to see that they create a matrix of zeros with the same shape as ***A*** and fill it with the singular values:

```
smat = np.zeros((9, 6), dtype=complex)
smat[:6, :6] = np.diag(s)
```

Intuition

I think that the intuition behind the singular value decomposition needs some explanations about the idea of matrix transformation. For that reason, here are several examples showing how the space can be transformed by 2D square matrices. Hopefully, this will lead to a better understanding of this statement: ***A*** is a matrix that can be seen as a linear transformation. This transformation can be decomposed in three sub-transformations: 1. rotation, 2. re-scaling, 3. rotation. These three steps correspond to the three matrices ***U***, ***D***, and ***V***.

A is a matrix that can be seen as a linear transformation. This transformation can be decomposed in three sub-transformations: 1. rotation, 2. re-scaling, 3. rotation. These three steps correspond to the three matrices U, D, and V.

You can look at this animation from the Wikipedia article on the SVD. If you scroll down the page you will see each step.

Every matrix can be seen as a linear transformation

You can see a matrix as a specific linear transformation. When you *apply* this matrix to a vector or to another matrix you will apply this linear transformation to it.

Example 1.

We will modify the vector:

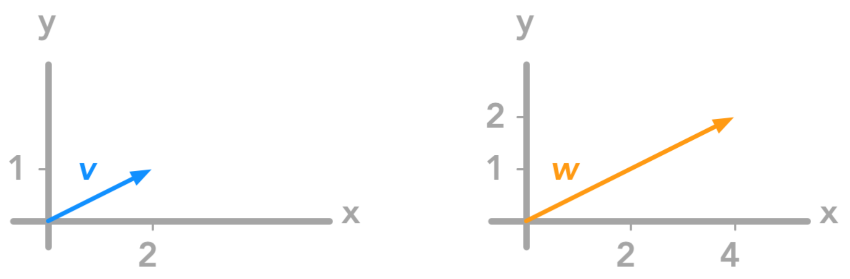
by applying the matrix:

We will have:

$[x'y']=[2002][xy]=[2x+0y0x+2y]=[2x2y]$

We see that applying the matrix:

just doubled each coordinate of our vector. Here are the graphical representation of $\boldsymbol{v}$ and its transformation $\boldsymbol{w}$:

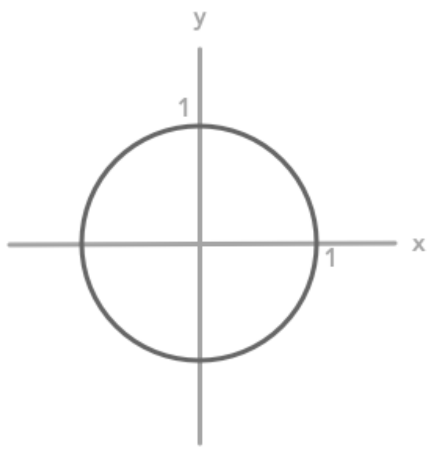


Applying the matrix on the vector multiplied each coordinate by two

You can look at other examples of simple transformations on vectors and unit circle in this video.

Example 2.

To represent the linear transformation associated with matrices we can also draw the unit circle and see how a matrix can transform it (see the BONUS in 2.7). The unit circle represents the coordinates of every unit vectors (vector of length 1, see 2.6).



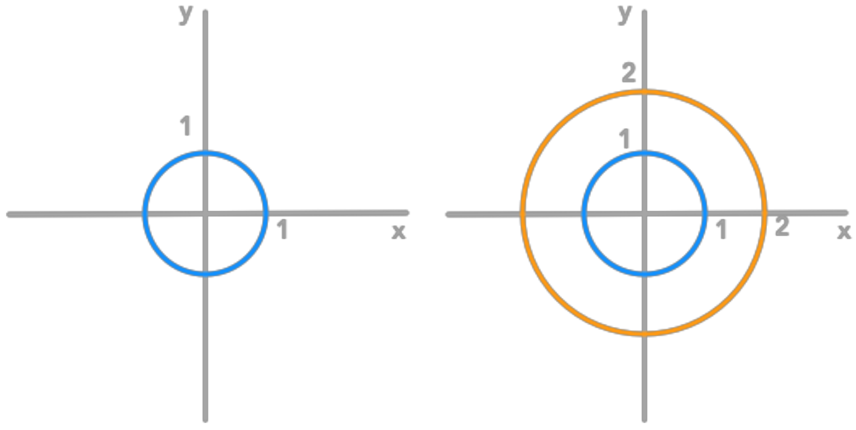
The unit circle

It is then possible to apply a matrix to all these unit vectors to see the kind of deformation it will produce.

Again, let’s apply the matrix:

to the unit circle:

$[x'y']=[2002][xy]=[2x2y]$



Another representation of the effect of the matrix: each coordinate of the unit circle was multiplied by two

We can see that the matrix doubled the size of the circle. But in some transformations, the change applied to the *x* coordinate is different from the change applied to the *y* coordinate. Let's see what it means graphically.

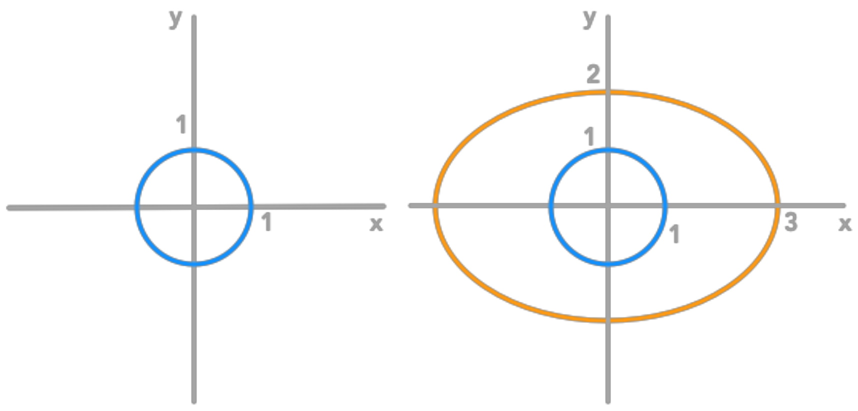
Example 3.

We will apply the matrix:

to the unit circle:

$[x'y']=[3002]\cdot[xy]=[3x2y]$

This gives the following new circle:



This time the matrix didn't rescale each coordinate with the same weight

We can check that with the equations associated with this matrix transformation. Let's say that the coordinates of the new circle (after transformation) are x' and y'. The relation between the old coordinates (x, y) and the new coordinates (x', y') is:

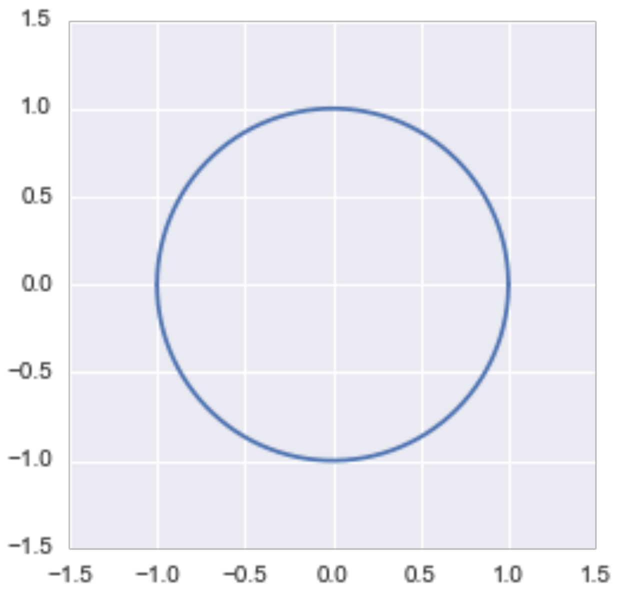
$[x'y']=[3x2y]\Leftrightarrow\{x=x'3y=y'2$

We also know that the equation of the unit circle is $x2+y2=1$ (the norm of the unit vectors is 1, see 2.5). By replacement we end up with:

$(x'3)2+(y'2)2=1(y'2)2=1-(x'3)2y'2=\surd 1-(x'3)2y'=2\surd 1-(x'3)2$

We can check that this equation corresponds to our transformed circle. Let's start by drawing the old circle. Its equation is:

```
x = np.linspace(-1, 1, 100000)
y = np.sqrt(1-(x**2))
plt.plot(x, y, sns.color_palette().as_hex()[0])
plt.plot(x, -y, sns.color_palette().as_hex()[0])
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.show()
```



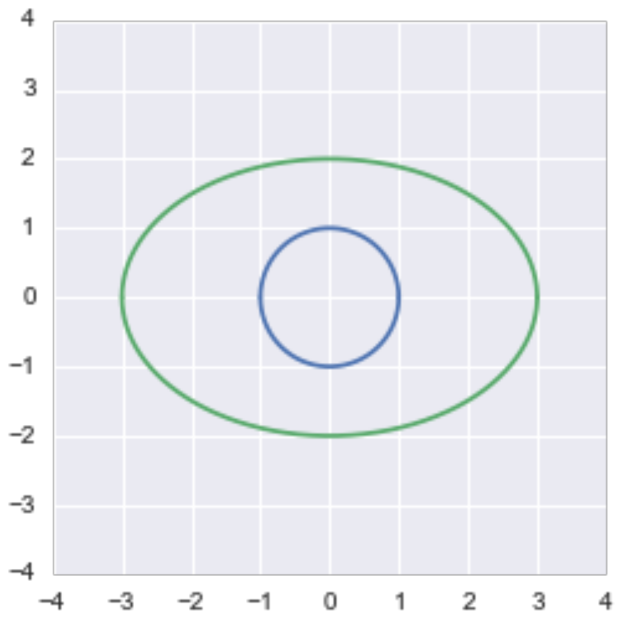
The unit circle plotted in Python

So far so good!

Coding tip: You can see the trick to plot a circle here: you create the x variable, then y is defined from x. This means that for each x, the corresponding y value is calculated (and thus y has the same shape as x). Since the result of the square root can be negative or positive (for instance, 4 can be the result of 22 but also of (−2)2) we need to plot both solutions (y and $-y$ in `plt.plot`). Note also that a lot of values are needed if we want the connection between the two demi-spheres. See also some discussion here.

Now let's add the circle obtained after matrix transformation. We saw that it is defined with

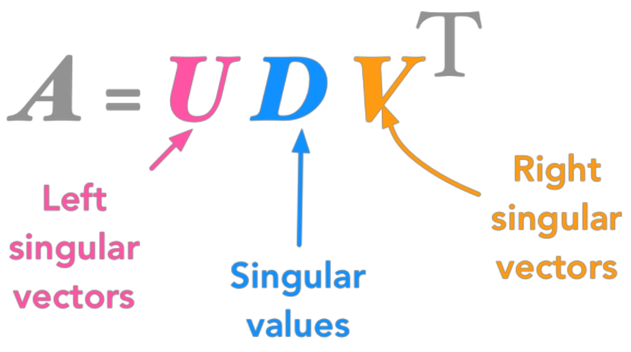
```
x1 = np.linspace(-3, 3, 100000)
y1 = 2*np.sqrt(1-((x1/3)**2))
plt.plot(x, y, sns.color_palette().as_hex()[0])
plt.plot(x, -y, sns.color_palette().as_hex()[0])
plt.plot(x1, y1, sns.color_palette().as_hex()[1])
plt.plot(x1, -y1, sns.color_palette().as_hex()[1])
plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.show()
```



The transformed circle from the equation

This shows that our transformation was correct.

Note that these examples used **diagonal matrices** (all zeros except the diagonal). The general rule is that the transformation associated with diagonal matrices imply only a rescaling of each coordinate **without rotation**. This is a first element to understand the SVD. Look again at the decomposition



The singular value decomposition

The transformation associated with diagonal matrices imply only a rescaling of each coordinate without rotation

We saw that the matrix $\boldsymbol{D}$ is a diagonal matrix. And we saw also that it corresponds to a rescaling without rotation.

Example 4. rotation matrix

Matrices that are not diagonal can produce a rotation (see more details here). Since it is easier to think about angles when we talk about rotation, we will use a matrix of the form

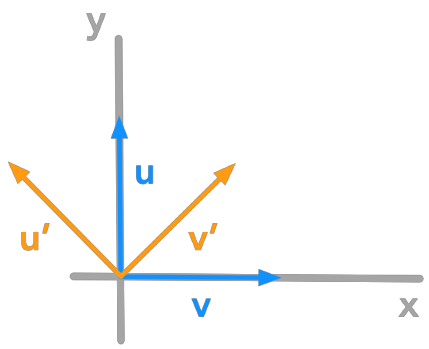
$R=[cos(\theta)-sin(\theta)sin(\theta)cos(\theta)]$

This matrix will rotate our vectors or matrices counterclockwise through an angle θ. Our new coordinates will be

$[x'y']=[cos(\theta)-sin(\theta)sin(\theta)cos(\theta)][xy]=[xcos(\theta)-ysin(\theta)xsin(\theta)+ycos(\theta)]$

Let's rotate some vectors through an angle of $\theta=45^{\circ}$.

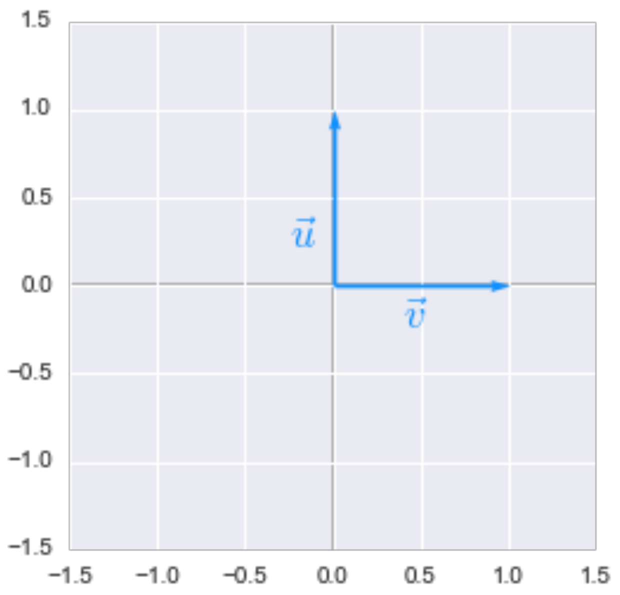
Let's start with the vector $\boldsymbol{u}$ of coordinates $x=0$ and $y=1$ and the vector $\boldsymbol{v}$ of coordinates $x=1$ and $y=0$. The vectors $\boldsymbol{u'}$ $\boldsymbol{v'}$ are the rotated vectors.



Counter clockwise rotation of the unit vectors with $\theta=45^{\circ}$

First, let's plot $\boldsymbol{u}$ and $\boldsymbol{v}$.

```
orange = '#FF9A13'
blue = '#1190FF'

u = [1,0]
v = [0,1]

plotVectors([u, v], cols=[blue, blue])

plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)

plt.text(-0.25, 0.2, r'$\vec{u}$', color=blue, size=18)
plt.text(0.4, -0.25, r'$\vec{v}$', color=blue, size=18)
plt.show()
```



Unit vectors plotted with Python

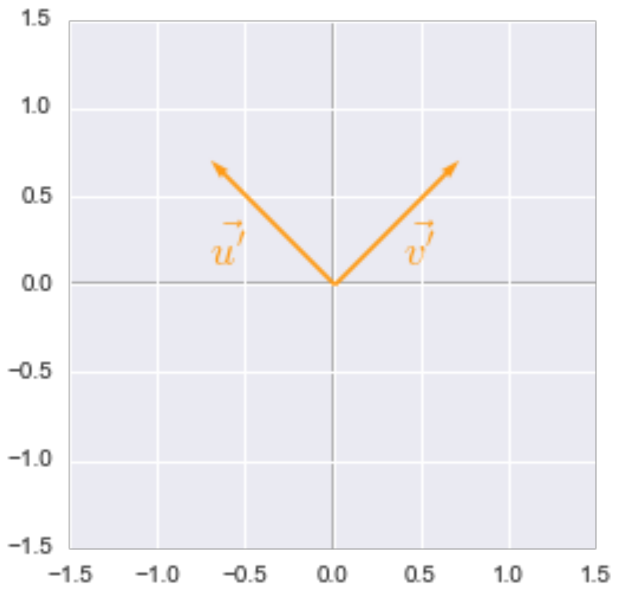
They are the basis vectors of our space. We will calculate the transformation of these vectors:

$$\begin{cases} u_x = 0 \cdot cos(45) - 1 \cdot sin(45) \\ u_y = 0 \cdot sin(45) + 1 \cdot cos(45) \end{cases} \Leftrightarrow \begin{cases} u_x = -sin(45) \\ u_y = cos(45) \end{cases}$$

$$\begin{cases} v_x = 1 \cdot cos(45) - 0 \cdot sin(45) \\ v_y = 1 \cdot sin(45) + 0 \cdot cos(45) \end{cases} \Leftrightarrow \begin{cases} v_x = cos(45) \\ v_y = sin(45) \end{cases}$$

We will now plot these new vectors to check that they are well our basis vectors rotated through an angle of 45°.

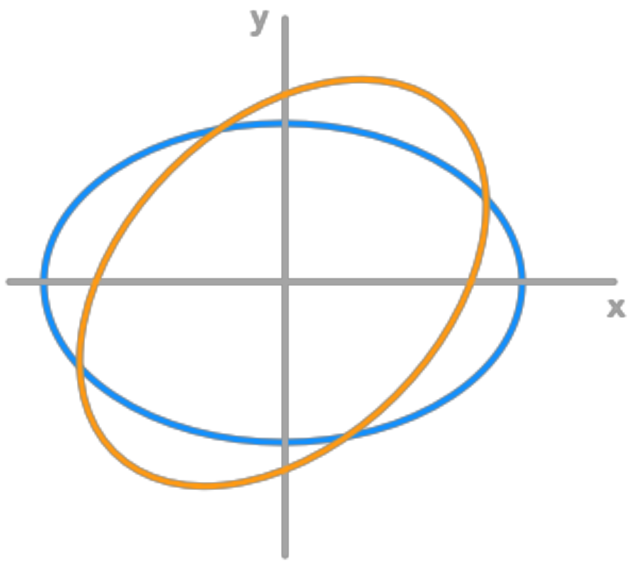
```
u1 = [-np.sin(np.radians(45)), np.cos(np.radians(45))]
v1 = [np.cos(np.radians(45)), np.sin(np.radians(45))]

plotVectors([u1, v1], cols=[orange, orange])
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)

plt.text(-0.7, 0.1, r"$\vec{u'}$", color=orange, size=18)
plt.text(0.4, 0.1, r"$\vec{v'}$", color=orange, size=18)
plt.show()
```



Unit vectors rotated plotted with Python

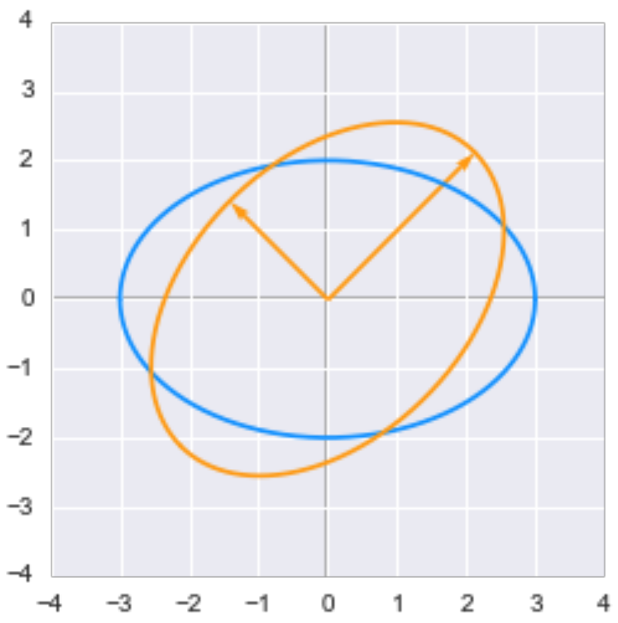
Coding tip: the numpy functions `sin` and `cos` take input in radians. We can convert our angle from degrees to radians with the function `np.radians()`.

We can also transform a circle. We will take a rescaled circle (the one from the example 3.) to be able to see the effect of the rotation.



The effect of a rotation matrix on a rescaled circle

```
x = np.linspace(-3, 3, 100000)
y = 2*np.sqrt(1-((x/3)**2))

x1 = x*np.cos(np.radians(45)) - y*np.sin(np.radians(45))
y1 = x*np.sin(np.radians(45)) + y*np.cos(np.radians(45))

x1_neg = x*np.cos(np.radians(45)) - -y*np.sin(np.radians(45))
y1_neg = x*np.sin(np.radians(45)) + -y*np.cos(np.radians(45))

u1 = [-2*np.sin(np.radians(45)), 2*np.cos(np.radians(45))]
v1 = [3*np.cos(np.radians(45)), 3*np.sin(np.radians(45))]

plotVectors([u1, v1], cols=['#FF9A13', '#FF9A13'])

plt.plot(x, y, '#1190FF')
plt.plot(x, -y, '#1190FF')

plt.plot(x1, y1, '#FF9A13')
plt.plot(x1_neg, y1_neg, '#FF9A13')

plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.show()
```



The effect of a rotation matrix on a rescaled circle plotted in Python

We can see that the circle has been rotated by an angle of 45∘. We have chosen the length of the vectors from the rescaling weight from example 3 (factor 3 and 2) to match the circle.

Summary

I hope that you got how vectors and matrices can be transformed by rotating or scaling matrices. The SVD can be seen as the decomposition of one complex transformation in 3 simpler transformations (a rotation, a scaling and another rotation).

Note that we took only square matrices. The SVD can be done even with non square matrices but it is harder to represent transformation associated with non square matrices. For instance, a 3 by 2 matrix will map a 2D space to a 3D space.

$$\begin{bmatrix}1&2\\3&4\\5&6\end{bmatrix}\overset{\overset{\text{2D}}{\uparrow}}{\begin{bmatrix}x\\y\end{bmatrix}}=\overset{\overset{\text{3D}}{\uparrow}}{\begin{bmatrix}1x+2y\\3x+4y\\5x+6y\end{bmatrix}}$$

A non square matrix change the number of dimensions of the input

The three transformations

Now that the link between matrices and linear transformation is clearer we can check that a transformation associated with a matrix can be decomposed with the help of the SVD.

But first let's create a function that takes a 2D matrix as an input and draw the unit circle transformation when we apply this matrix to it. It will be useful to visualize the transformations.

```
def matrixToPlot(matrix, vectorsCol=['#FF9A13', '#1190FF']):
    """
    Modify the unit circle and basis vector by applying a matrix.
    Visualize the effect of the matrix in 2D.

    Parameters
    ----------
    matrix : array-like
        2D matrix to apply to the unit circle.
    vectorsCol : HEX color code
        Color of the basis vectors

    Returns:

    fig : instance of matplotlib.figure.Figure
        The figure containing modified unit circle and basis vectors.
    """
    # Unit circle
    x = np.linspace(-1, 1, 100000)
    y = np.sqrt(1-(x**2))

    # Modified unit circle (separate negative and positive parts)
    x1 = matrix[0,0]*x + matrix[0,1]*y
    y1 = matrix[1,0]*x + matrix[1,1]*y
    x1_neg = matrix[0,0]*x - matrix[0,1]*y
    y1_neg = matrix[1,0]*x - matrix[1,1]*y

    # Vectors
    u1 = [matrix[0,0],matrix[1,0]]
    v1 = [matrix[0,1],matrix[1,1]]

    plotVectors([u1, v1], cols=[vectorsCol[0], vectorsCol[1]])

    plt.plot(x1, y1, 'g', alpha=0.5)
    plt.plot(x1_neg, y1_neg, 'g', alpha=0.5)
```

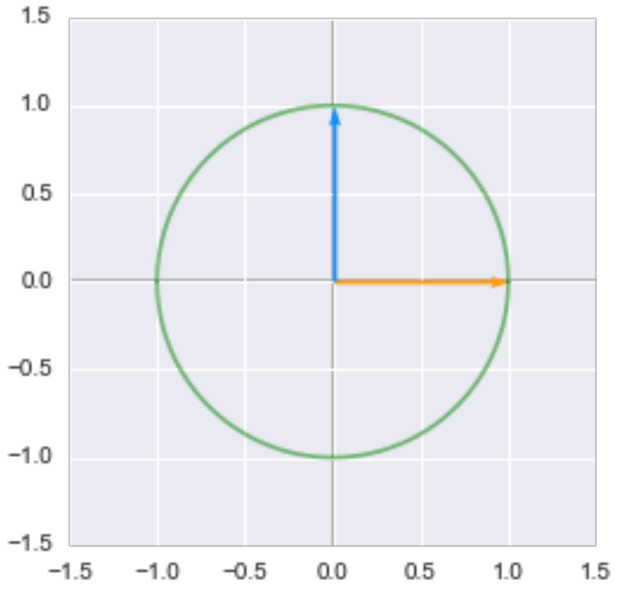
We can use it to check that the three transformations given by the SVD are equivalent to the transformation done with the original matrix. We will also draw each step of the SVD to see the independant effect of the first rotation, the scaling and the second rotation.

We will use the matrix:

and plot the unit circle and its transformation by $\boldsymbol{A}$:

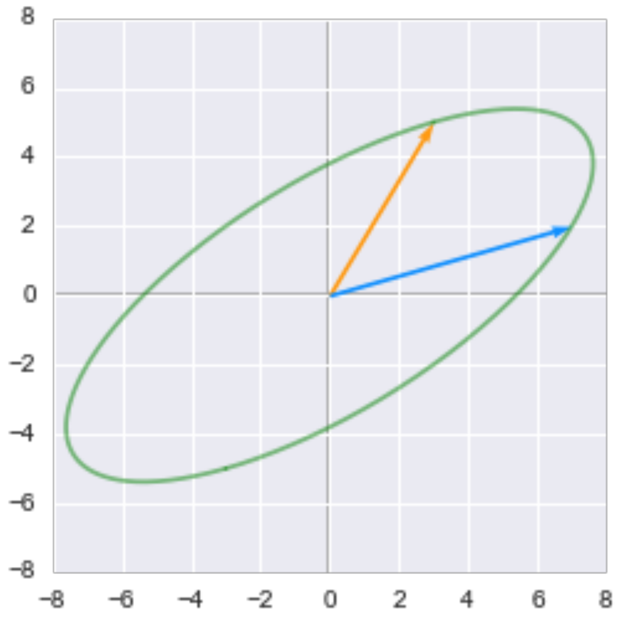
```
A = np.array([[3, 7], [5, 2]])

print 'Unit circle:'
matrixToPlot(np.array([[1, 0], [0, 1]]))
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.show()

print 'Unit circle transformed by A:'
matrixToPlot(A)
```

```
plt.xlim(-8, 8)
plt.ylim(-8, 8)
plt.show()
```



The unit circle and vectors plotted in Python

```
Unit circle transformed by A:
```



The unit circle transformed by the matrix A

This is what we get when we apply the matrix $\boldsymbol{A}$ to the unit circle and the basis vectors. We can see that the two base vectors are not necessarily rotated the same way. This is related to the sign of the determinant of the matrix (see 2.11).

Let's now compute the SVD of $\boldsymbol{A}$:

```
U, D, V = np.linalg.svd(A)
U
```

```
array([[-0.85065081, -0.52573111],
       [-0.52573111,  0.85065081]])
```

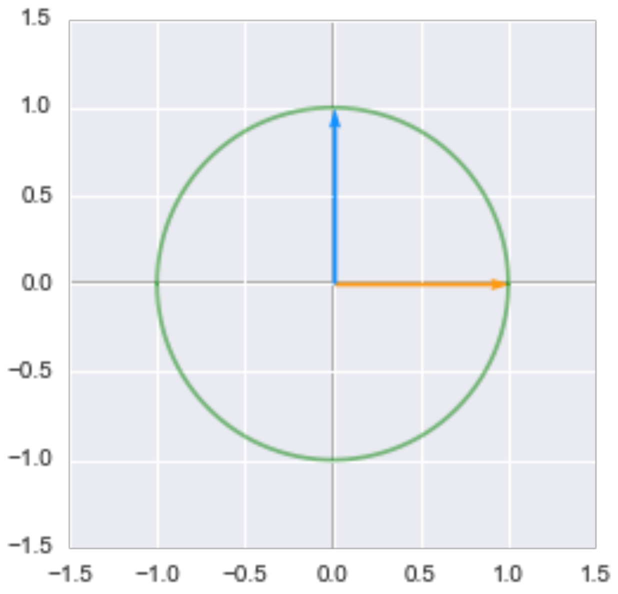
```
array([ 8.71337969,  3.32821489])
```

```
array([[-0.59455781, -0.80405286],
       [ 0.80405286, -0.59455781]])
```

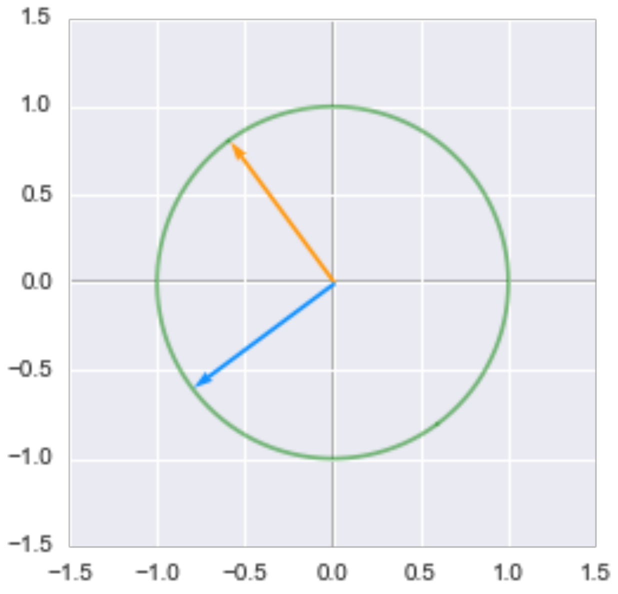
We can now look at the sub-transformations by looking at the effect of the matrices $\boldsymbol{U}$, $\boldsymbol{D}$ and $\boldsymbol{V}$ in the reverse order. Note that it returns the right singular vector **already transposed** (see the doc).

```
# Unit circle
print 'Unit circle:'
matrixToPlot(np.array([[1, 0], [0, 1]]))
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.show()

print 'First rotation:'
matrixToPlot(V)
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.show()

print 'Scaling:'
```

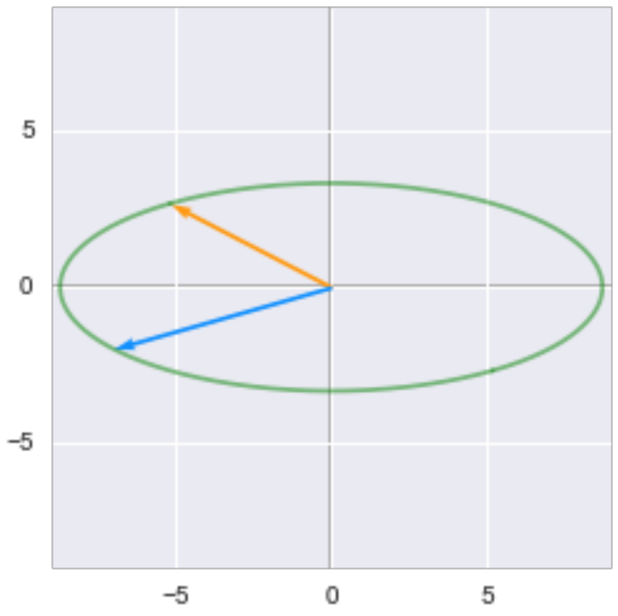
```
matrixToPlot(np.diag(D).dot(V))
plt.xlim(-9, 9)
plt.ylim(-9, 9)
plt.show()

print 'Second rotation:'
matrixToPlot(U.dot(np.diag(D)).dot(V))
plt.xlim(-8, 8)
plt.ylim(-8, 8)
plt.show()
```



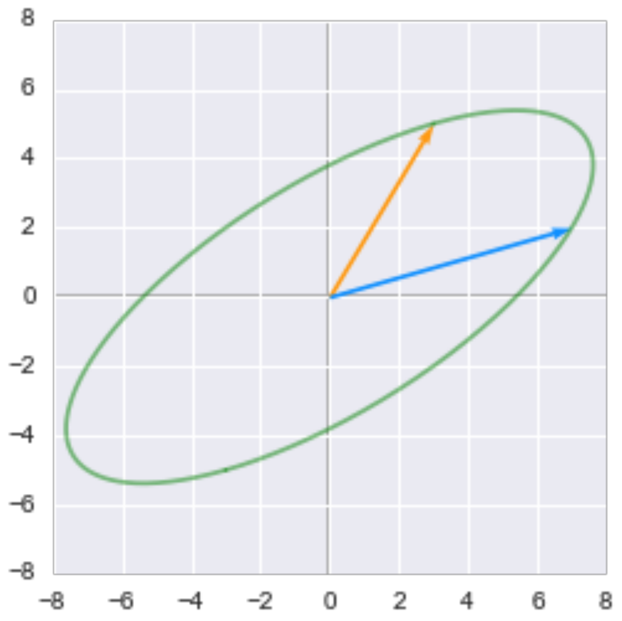
The unit circle and vectors plotted in Python



The unit circle rotated by the matrix V



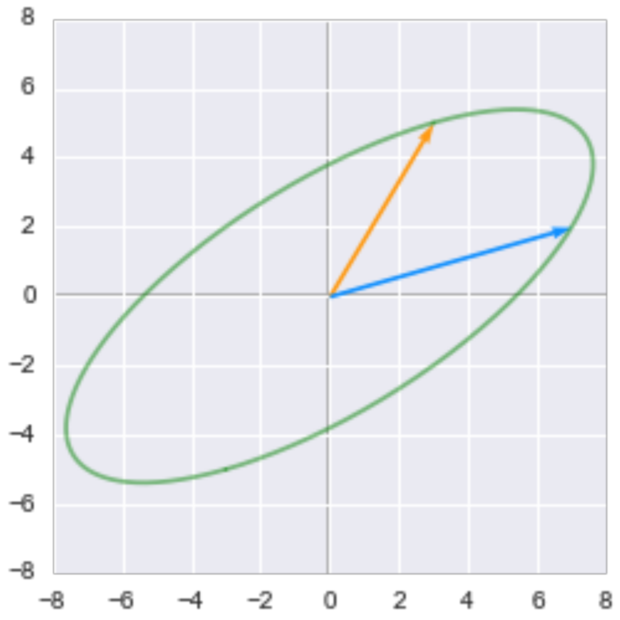
After the rotation, the unit circle is rescaled by D



Finally a third rotation is done with U

Just to be sure, you can compare this last step with the transformation by ***A***. Fortunately, you will see that the result is the same:

```
matrixToPlot(A)
plt.xlim(-8, 8)
plt.ylim(-8, 8)
plt.show()
```

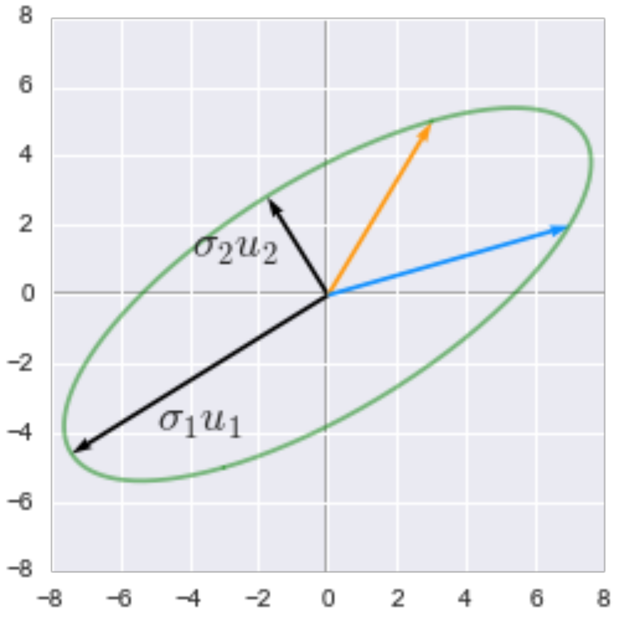


The unit circle transformed by the matrix A

Singular values interpretation

The singular values are ordered by descending order. They correspond to a new set of features (that are a linear combination of the original features) with the first feature explaining most of the variance. For instance from the last example we can visualize these new features. The major axis of the elipse will be the first left singular vector ($u1$) and its norm will be the first singular value ($\sigma1$).

```
u1 = [D[0]*U[0,0], D[0]*U[0,1]]
v1 = [D[1]*U[1,0], D[1]*U[1,1]]

plotVectors([u1, v1], cols=['black', 'black'])

matrixToPlot(A)

plt.text(-5, -4, r"$\sigma_1u_1$", size=18)
plt.text(-4, 1, r"$\sigma_2u_2$", size=18)

plt.xlim(-8, 8)
plt.ylim(-8, 8)
plt.show()
```



The singular values and vectors show the major and minor axes of the transformed circle

They are the major ($\sigma 1 u 1$) and minor ($\sigma 2 u 2$) axes of the elipse. We can see that the feature corresponding to this major axis is associated with more variance (the range of value on this axis is bigger than the other). See 2.12 for more details about the variance explained.

SVD and eigendecomposition

Now that we understand the kind of decomposition done with the SVD, we want to know how the sub-transformations are found.

The matrices $\boldsymbol{U}$, $\boldsymbol{D}$ and $\boldsymbol{V}$ can be found by transforming $\boldsymbol{A}$ in a square matrix and by computing the eigenvectors of this square matrix. The square matrix can be obtain by multiplying the matrix $\boldsymbol{A}$ by its transpose in one way or the other:

- $\boldsymbol{U}$ corresponds to the eigenvectors of $\boldsymbol{AA}^\text{T}$
- $\boldsymbol{V}$ corresponds to the eigenvectors of $\boldsymbol{A}^\text{T}\boldsymbol{A}$
- $\boldsymbol{D}$ corresponds to the eigenvalues $\boldsymbol{AA}^\text{T}$ or $\boldsymbol{A}^\text{T}\boldsymbol{A}$ which are the same.

Let's take an example of a non square matrix:

The singular value decomposition can be done with the `linalg.svd()` function from Numpy (note that `np.linalg.eig(A)` works only on square matrices and will give an error for `A`).

```
A = np.array([[7, 2], [3, 4], [5, 3]])
U, D, V = np.linalg.svd(A)
U
```

```
array([[-0.69366543,  0.59343205, -0.40824829],
       [-0.4427092 , -0.79833696, -0.40824829],
       [-0.56818732, -0.10245245,  0.81649658]])
```

```
array([ 10.25142677,   2.62835484])
```

```
array([[-0.88033817, -0.47434662],
       [ 0.47434662, -0.88033817]])
```

The left-singular values

The left-singular values of $\boldsymbol{A}$ correspond to the eigenvectors of $\boldsymbol{AA}^\text{T}$.

Example 5.

Note that the sign difference comes from the fact that eigenvectors are not unique. The `linalg` functions from Numpy return the normalized eigenvectors. Scaling by `-1` doesn't change their direction or the fact that they are unit vectors.

```
U, D, V = np.linalg.svd(A)
```

Left singular vectors of A:

```
array([[-0.69366543,  0.59343205, -0.40824829],
       [-0.4427092 , -0.79833696, -0.40824829],
       [-0.56818732, -0.10245245,  0.81649658]])
```

Eigenvectors of AA_transpose:

```
np.linalg.eig(A.dot(A.T))[1]
```

```
array([[-0.69366543, -0.59343205, -0.40824829],
       [-0.4427092 ,  0.79833696, -0.40824829],
       [-0.56818732,  0.10245245,  0.81649658]])
```

The right-singular values

The right-singular values of $\boldsymbol{A}$ correspond to the eigenvectors of $\boldsymbol{A}\text{T}\boldsymbol{A}$.

Example 6.

```
U, D, V = np.linalg.svd(A)
```

Right singular vectors of A:

```
array([[-0.88033817, -0.47434662],
       [ 0.47434662, -0.88033817]])
```

Eigenvectors of A_transposeA:

```
np.linalg.eig(A.T.dot(A))[1]
```

```
array([[ 0.88033817, -0.47434662],
       [ 0.47434662,  0.88033817]])
```

The nonzero singular values

The nonzero singular values of $\boldsymbol{A}$ are the square roots of the eigenvalues of $\boldsymbol{A}\text{T}\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{A}\text{T}$.

Example 7.

```
U, D, V = np.linalg.svd(A)
D
```

```
array([ 10.25142677,   2.62835484])
```

Eigenvalues of A_transposeA:

```
np.linalg.eig(A.T.dot(A))[0]
```

```
array([ 105.09175083,    6.90824917])
```

Eigenvalues of AA_transpose:

```
np.linalg.eig(A.dot(A.T))[0]
```

```
array([ 105.09175083,    6.90824917,   -0.        ])
```

Square root of the eigenvalues:

```
np.sqrt(np.linalg.eig(A.T.dot(A))[0])
```

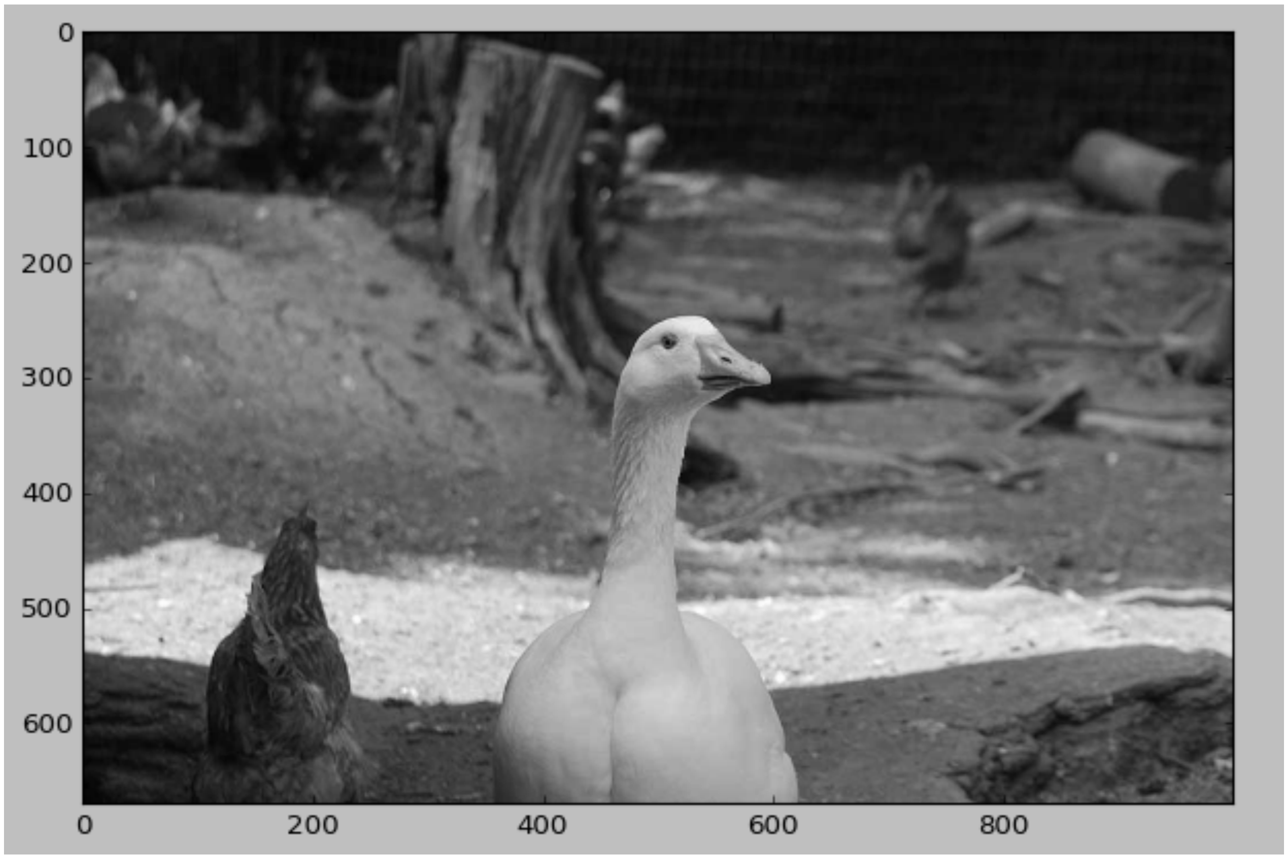
```
array([ 10.25142677,   2.62835484])
```

BONUS: Apply the SVD on images

In this example, we will use the SVD to extract the more important features from the image. It is nice to see the effect of the SVD on something very visual. The code is inspired/taken from this blog post.

Let’s start by loading an image in python and convert it to a Numpy array. We will convert it to grayscale to have one dimension per pixel. The shape of the matrix corresponds to the dimension of the image filled with intensity values: 1 cell per pixel.

```
from PIL import Image

plt.style.use('classic')
img = Image.open('test_svd.jpg')
# convert image to grayscale
imggray = img.convert('LA')
# convert to numpy array
imgmat = np.array(list(imggray.getdata(band=0)), float)
# Reshape according to orginal image dimensions
imgmat.shape = (imggray.size[1], imggray.size[0])

plt.figure(figsize=(9, 6))
plt.imshow(imgmat, cmap='gray')
plt.show()
```



A beautiful picture of Lucy the goose

We will see how to test the effect of SVD on **Lucy the goose**! Let's start to extract the left singular vectors, the singular values and the right singular vectors:

```
U, D, V = np.linalg.svd(imgmat)
```

Let's check the shapes of our matrices:

```
(669, 1000)
```

```
(669, 669)
```

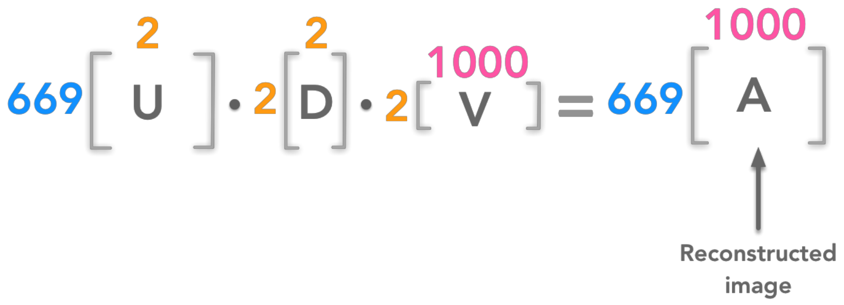
```
(669,)
```

```
(1000, 1000)
```

Remember that ***D*** are the singular values that need to be put into a diagonal matrix. Also, ***V*** doesn't need to be transposed (see above).

The singular vectors and singular values are ordered with the first ones corresponding to the more variance explained. For this reason, using just the first few singular vectors and singular values will provide the reconstruction of the principal elements of the image.

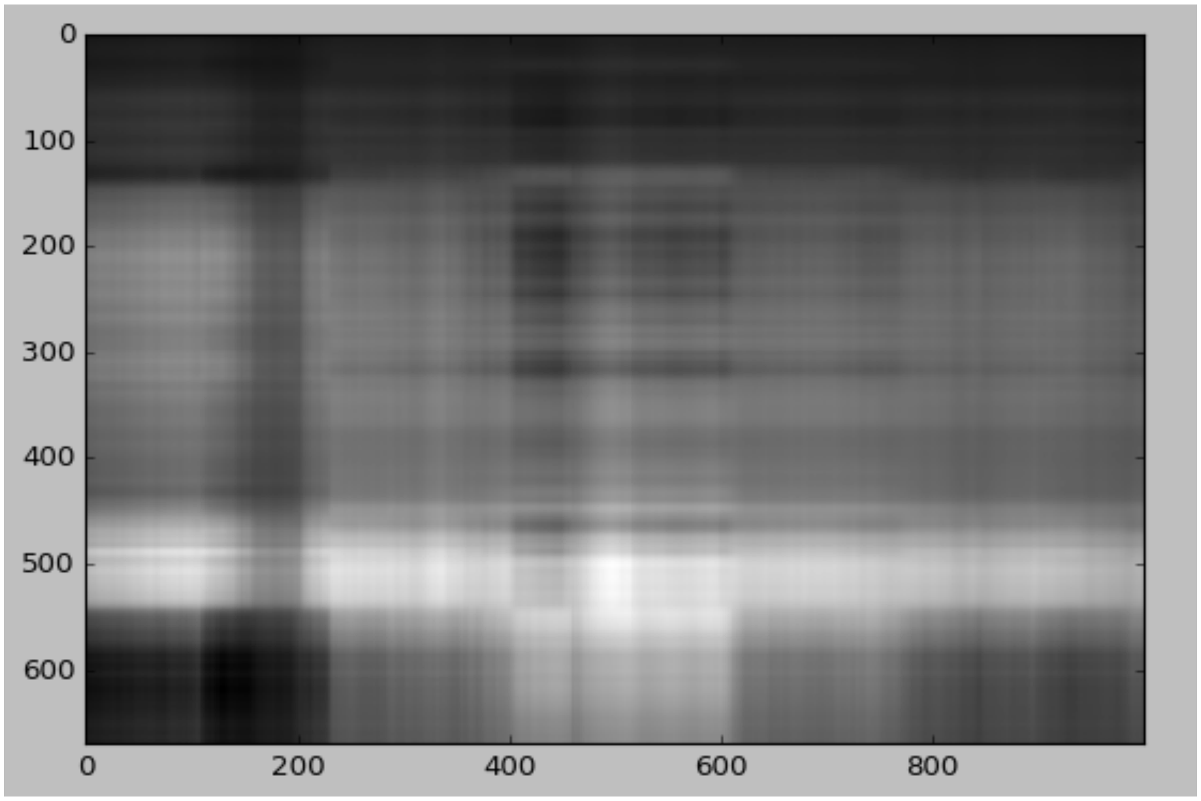
We can reconstruct an image from a certain number of singular values. For instance for 2 singular values we will have:



We can reconstruct the image from few components

In this example, we have reconstructed the 699px by 1000px image from two singular values.

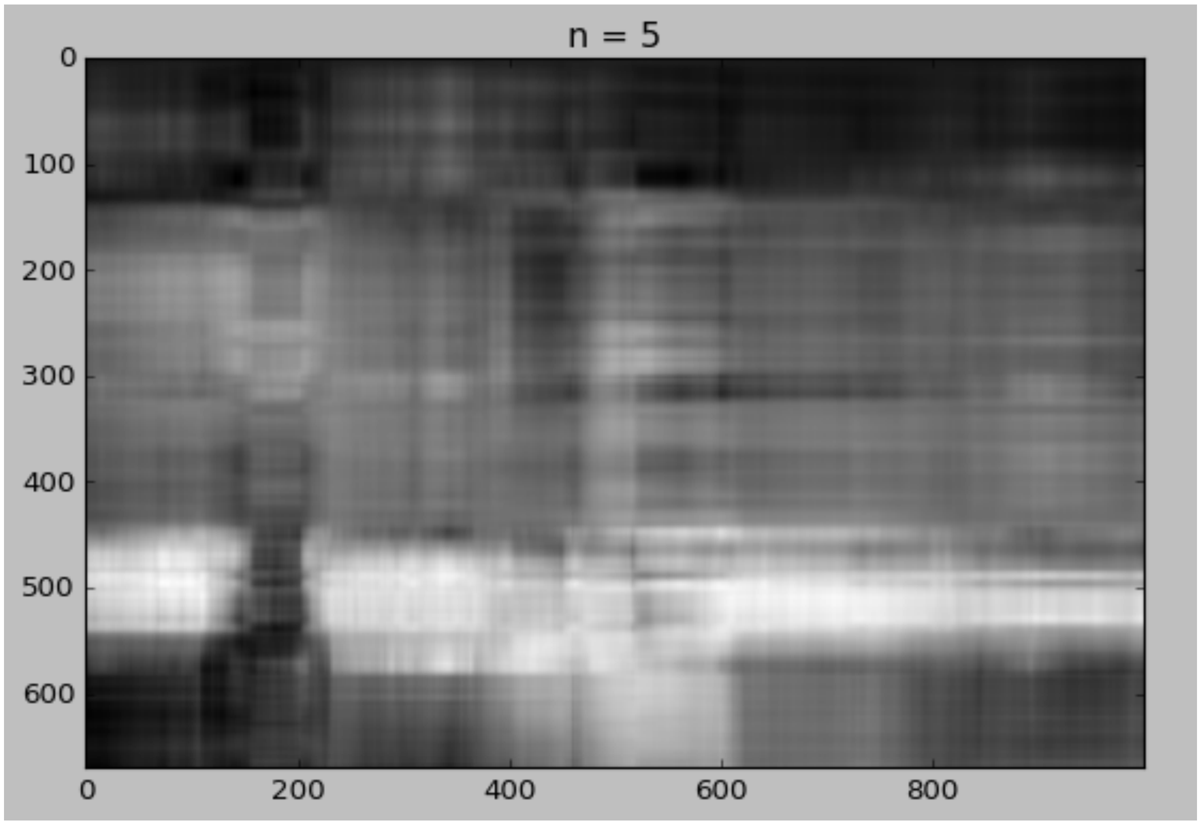
```
reconstimg = np.matrix(U[:, :2]) * np.diag(D[:2]) * np.matrix(V[:2, :])
plt.imshow(reconstimg, cmap='gray')
plt.show()
```

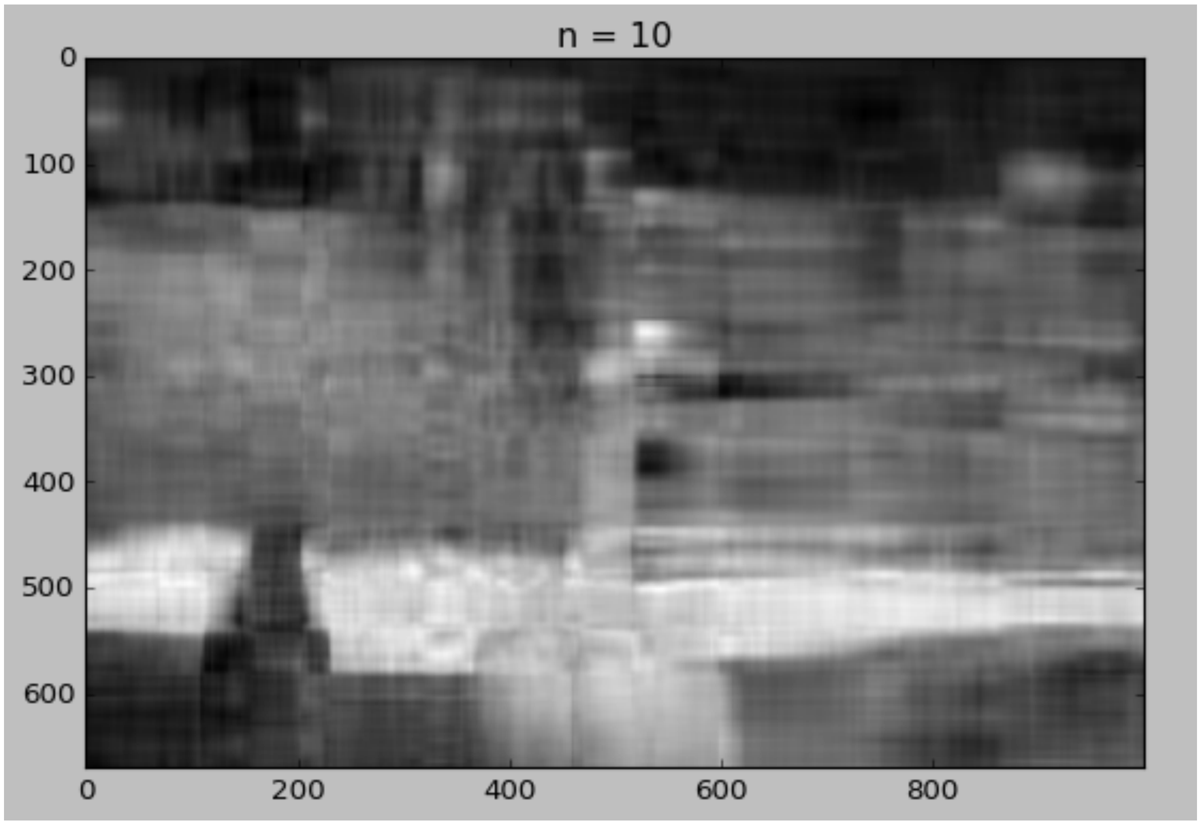


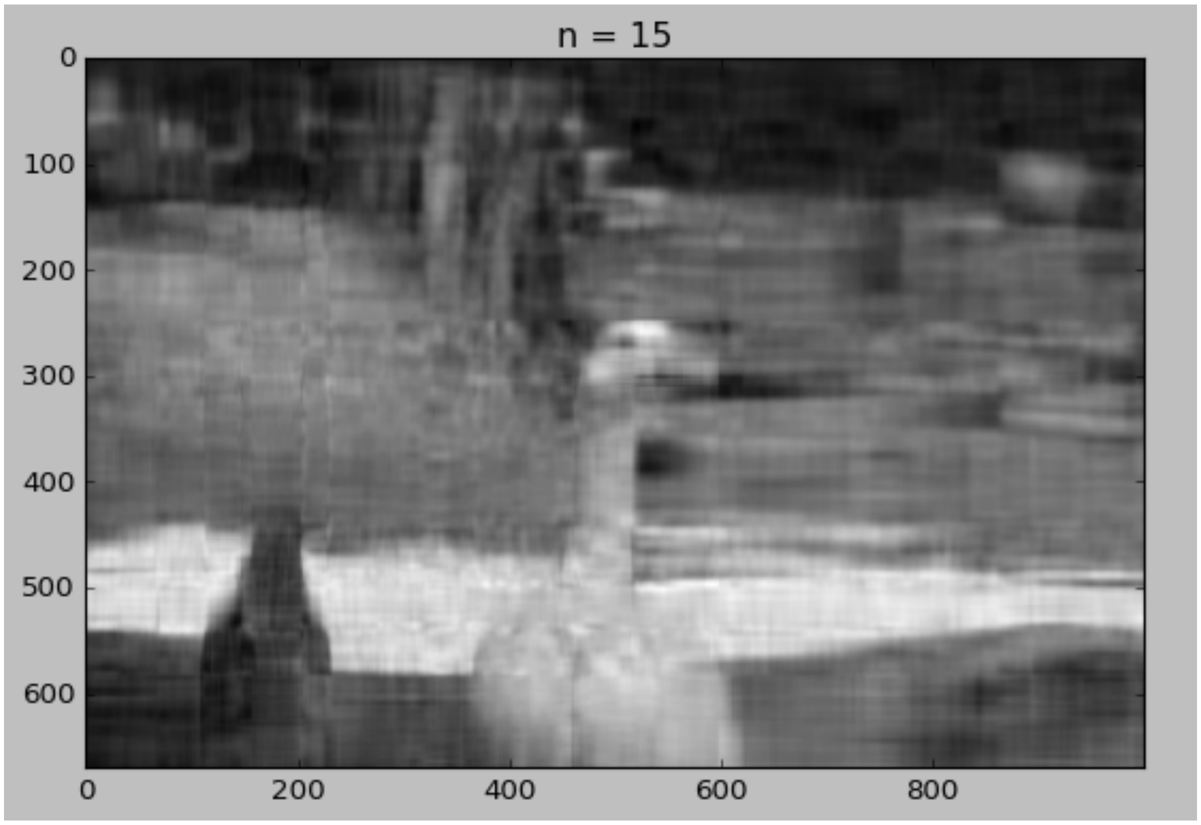
It is hard to see Lucy with only two singular values and singular vectors. But we already see something!

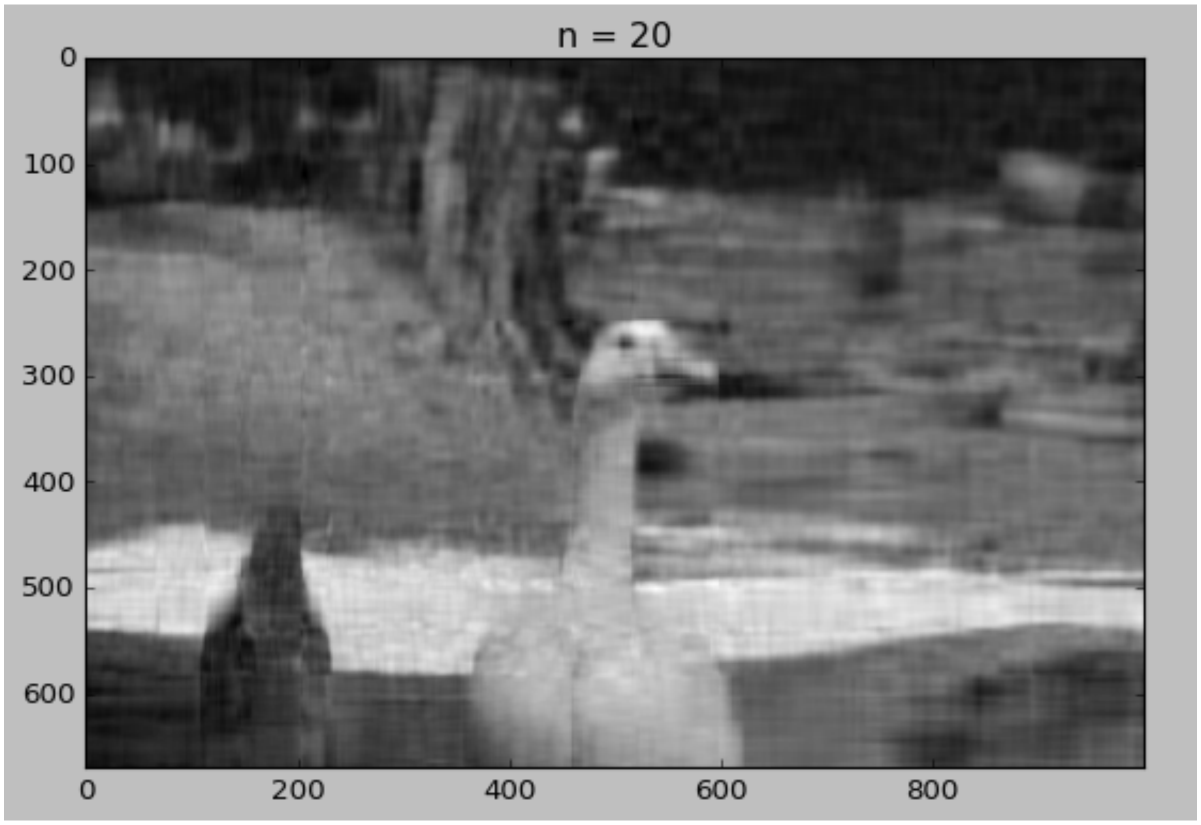
We will now draw the reconstruction using different number of singular values.

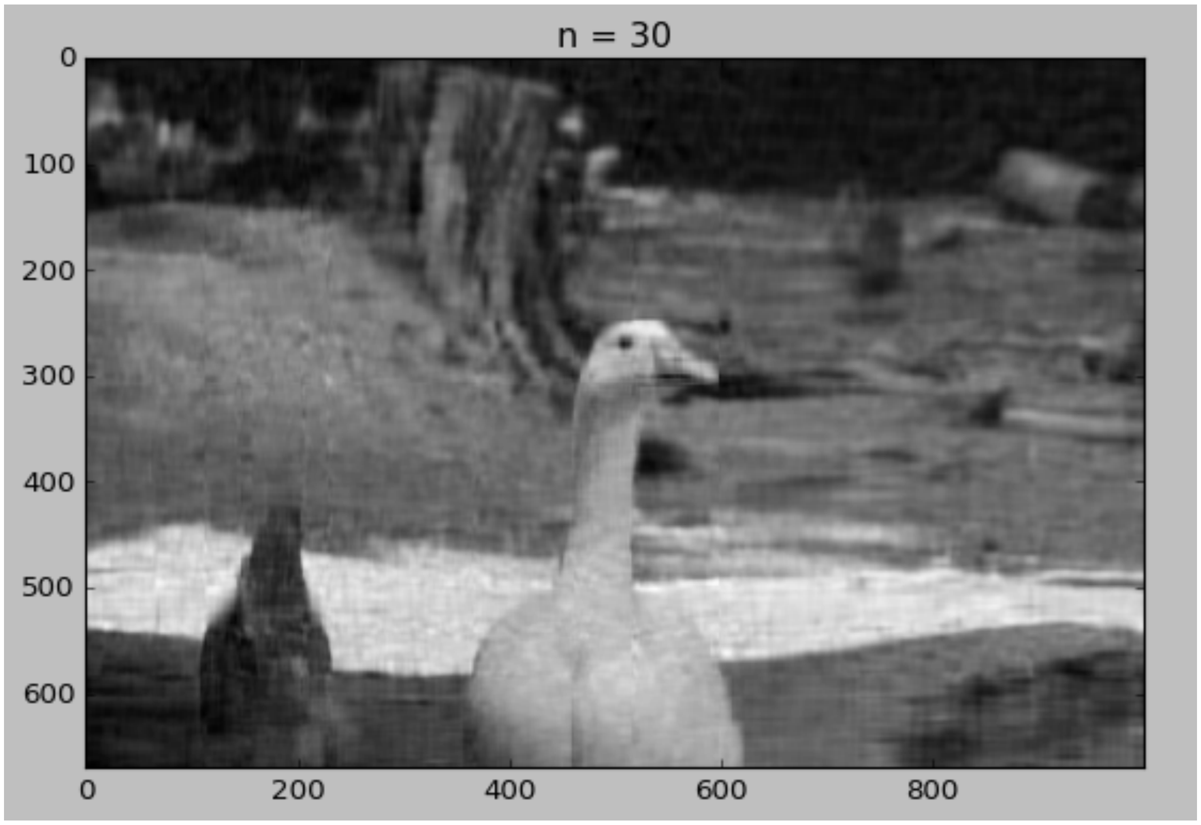
```
for i in [5, 10, 15, 20, 30, 50]:
    reconstimg = np.matrix(U[:, :i]) * np.diag(D[:i]) * np.matrix(V[:i, :])
    plt.imshow(reconstimg, cmap='gray')
    title = "n = %s" % i
    plt.title(title)
    plt.show()
```

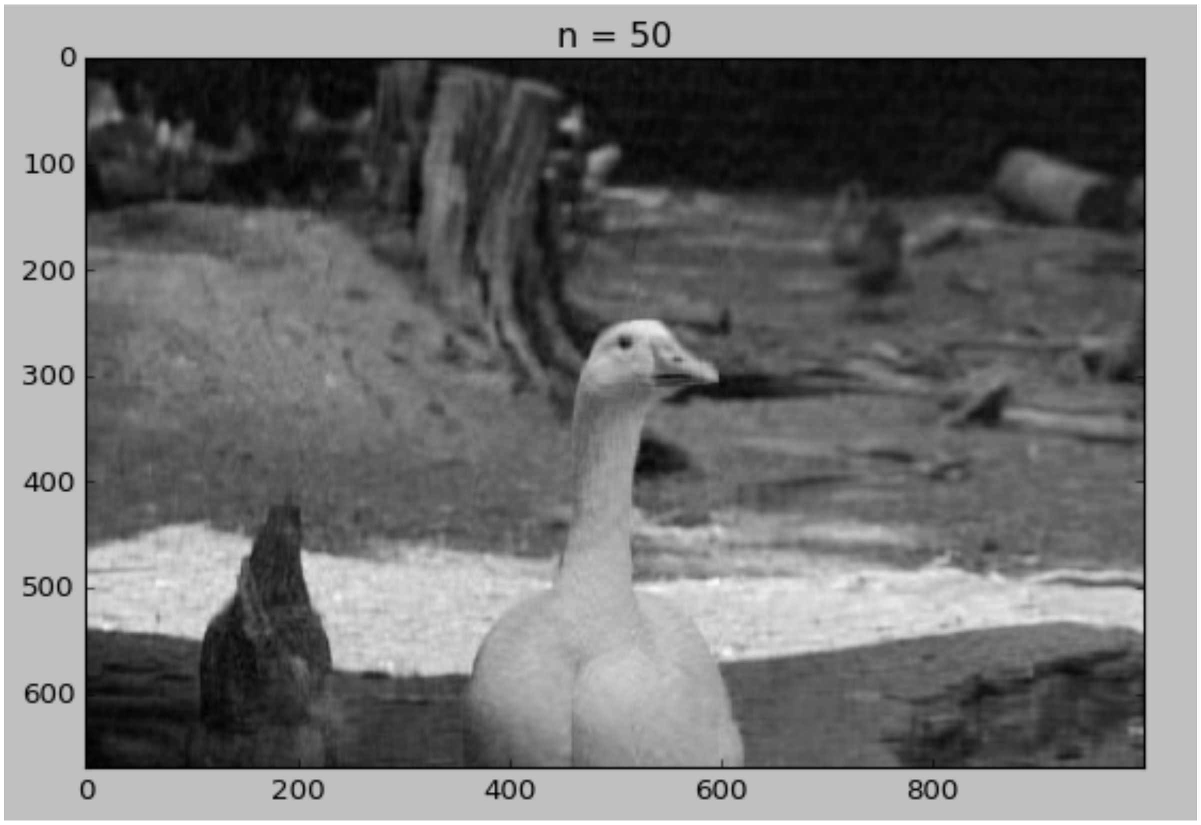

n = 5
0
100
200
300
400
500
600
0
200
400
600
800


n = 10
0
100
200
300
400
500
600
0
200
400
600
800


n = 15
0
100
200
300
400
500
600
0
200
400
600
800


n = 20
0
100
200
300
400
500
600
0
200
400
600
800





Whaou! Even with 50 components, the quality of the image is not bad!

Conclusion

I like this chapter on the SVD because it uses what we have learned so far in a concrete application. The next chapter on the pseudo-inverse is quite cool as well so keep on reading! We will see how to find a near-solution of a system of equation that minimizes the error and at the end we will see an example that uses the pseudo-inverse to find the best fit line of a set of data points.

References

Drawing a circle with Matplotlib

- SE - Plot equation showing a circle

Rotation matrix

- Wikipedia - Rotation matrix

Basis vectors

- Wikipedia - Basis

Linear transformation

- Aran Glancy - Linear transformation and matrices

SVD

- Singular Value Decomposition - Wikipedia

- Professor-svd