Introduction to Numerical Data Science, RUB

Report on Matrix Compression

A Comparison between standard and randomized Singular Value Decomposition methods

by Peter Moeller

December 5, 2024

Introduction

The following report will shed light on the advantages of the Randomized Single Value Decomposition (RSVD), when compared against the standard Single Value Decomposition method. We will do this through the comparison of the complexity of the matrices to be decomposed (e.g. the complexity of the functions used to generate their entries or even images, which are far more complex), their dimensions, as well as the rank of the singular matrices, and evaluate these against the compression's precision (determined using the Frobenius norm), computational time, and memory usage.

The standard method for calculating the SVD separates a given matrix M into the product of three different matrices, U, V* (the conjugate transpose of V) and S, with U and V* being unitary matrices (and orthogonal matrices in the cases where the entries of the matrices are real and not complex), and S being a diagonal matrix containing all of the singular values (which are the square roots of the eigenvalues of MM* and M*M, which matrices are at least positive semi-definite) in descending order of magnitude with the number of singular values in S corresponding to the rank of the matrix M. We can thus write:

$$M = USV^*$$

If we wish only an approximation of M (to save memory), we can choose to truncate S (i.e. using only the singular values above a certain size and replacing the values smaller than these with zero). This method, however, relies on first computing the entire SVD and, only afterward, determining an approximation, which, in the case of large data sets, can become problematic. This is where the RSVD shows its strength.

The method for computing the RSVD does not require us to first calculate the entire SVD, but instead first projects the matrix M unto a lower-dimensional subspace based on a randomized matrix, which entries relies on a Gauss-distribution and a given threshold, carries through the computation, and then projects an approximation (of a precision of our choosing) back up to the full-dimensional space again. As we will see, this gives great advantages, especially when dealing with matrices of large dimensions.

In Exercise 1 and 2, we will illustrate the advantages of the RSVD over the standard SVD. After that, we will then apply the developed algorithms in the concrete case of a gray scale image of above 1000 x 1000 pixels (approximately 1800 x 1400) and illustrate the difference between the two methods in a graph comparing the tolerance (the detail of the image) to CPU time taken to generate it.

We will make use of Python's NumPy and Matplotlib libraries (e.g. using the built-in SVD function, as well as the built-in feature to generate Gaussian matrices), so as not having to implement every single step from scratch, saving not only the author considerable time, but also that of the reader.

The Results

We begin with the task of calculating the standard SVD given matrices of dimensions (10 x 10) and (100 x 100), which are formed using the following function on the domain [0.1, 14.5] $\times$ [6, 6] on an equidistant grid:

$$T_1(x) = \exp\left(-0.4 \cdot \tanh\left(\frac{x-7.7}{8}\right)\right)$$

$$f_1(x,y) = \frac{1}{\sqrt{2\pi T_1(x)}} \cdot \exp\left(-\frac{y^2}{2T_1(x)}\right)$$

We create the desired matrices by defining a Python-function, which takes as inputs the function to be used, the dimensions, and the domain boundaries. We use the NumPy linespace functionality to generate the equidistant grid, and calculate each entry in the matrices by using two for-loops, one for rows and one for the columns, and return the matrix.

The next task consists in computing the SVD, i.e. $F = USV^*$. This is done in the following manner:

- First, calculate A^TA and AA^T, and find their eigenvalues (which will be the same for both).

- Second, use the eigenvalues to find the eigenvectors for A^TA and AA^T respectively.
- U $(m \times m)$ is formed by the eigenvectors of AA^T and V $(n \times n)$ by A^TA.
- The singular matrix S is a diagonal matrix with entries made up of the square roots on the eigenvalues.
- Thus, the final expression will be: $F = USV^*$ with U and V^* being unitary matrices and S a diagonal matrix $(m \times n)$.

Here, we will however use the built-in SVD computation from NumPy, simply ask for S to be returned to use, since this is all we need currently. We then plot the values of our two matrices in separate diagrams, setting the y-axis to be in a logarithmic scale (which will be useful for us in a moment). The following graphs shows the values for S for our two respective matrices (see Figure 1 and 2).

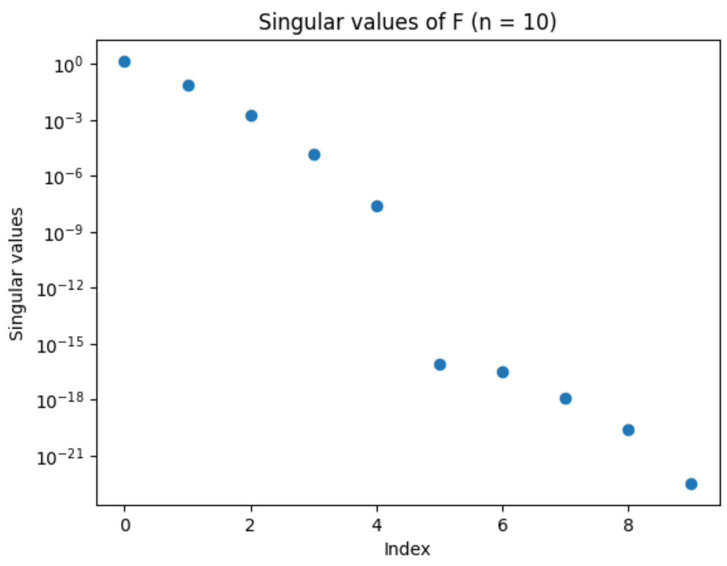

Figure 1: Singular values of F (n = 10)

We see that for our F (10×10), the first five singular values follows roughly a straight line, which, since the y-axis is defined logarithmically, means an exponential shape (like a radioactive decay-rate). Then there is a jump, followed by the next values. For the next evaluation of F (100×100) this exponential pattern is even clearer with a near perfect straight line in a logarithmic coordinate system (y-axis logarithmic, x-axis linear).

Next, we wish to decide the minimal rank r for our truncated SVD with $n = 100$, necessary to obtain an error less than 10^{-3} and 10^{-6} respectively. We

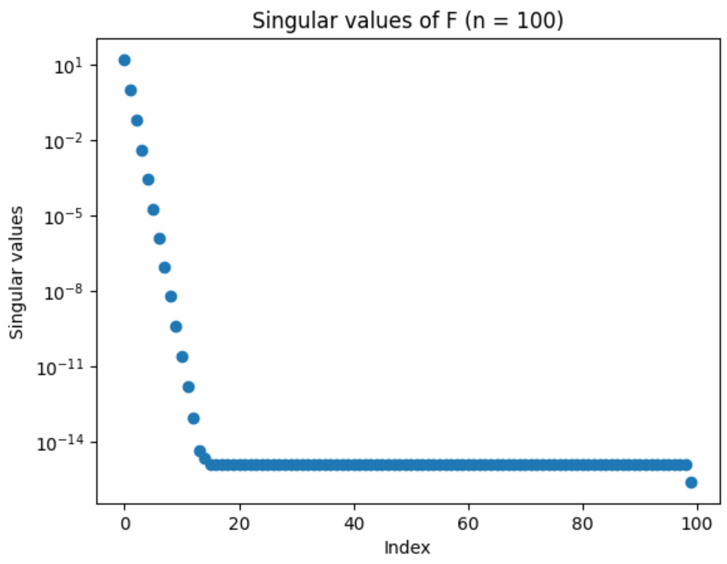

Figure 2: Singular values of F (n = 100)

do this be defining a function, which first computes the SVD for $n = 100$. We first compute the Frobenius norm for the entire singular matrix, as we will need this to compare with. We then proceed, step by step, to calculate the truncation error different ranks, until we arrive at the minimal rank, which gives us an error less than the set threshold, returning the rank r, the threshold (error), and the truncated error. In the loop for this procedure, we were careful to chose our range (starting from 1 instead of 0). We then get the following output:

Matrix F based on the given function (n = m = 100):
For an error below 0.001 a singular matrix of rank 3 is sufficient.
For an error below 1e-06 a singular matrix of rank 6 is sufficient.

For comparison's sake for later, we repeat the computation for a matrix (1500 x 1500). Here is the output:

Matrix F based on the given function (n = m = 1500):
For an error below 0.001 a singular matrix of rank 3 is sufficient.
For an error below 1e-06 a singular matrix of rank 6 is sufficient.

Now, we wish to try the same steps, but using a different function. We choose the following one:

$$f_2(x, y) = -(x^2 + y^2) + 4$$

setting the domain to be $[-2,2]\times[-2,2]$. After repeating the procedure, we get the following figures and output (see Figure 3 and 4):

```
Matrix F based on first own function (n = m = 100):
For an error below 0.001 a singular matrix of rank 2 is sufficient.
For an error below 1e-06 a singular matrix of rank 2 is sufficient.
```

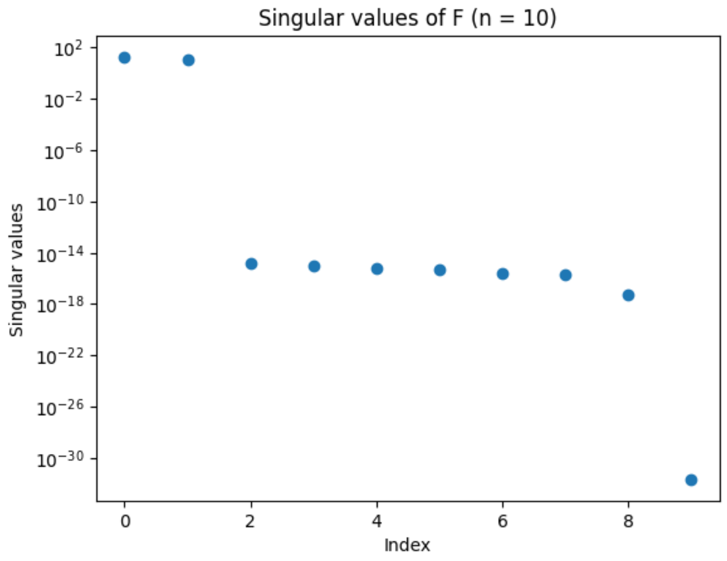

Figure 3: Singular values of F (n = 10)

Here, in Figure 3 as well as in Figure 4, we clearly see only two dominant singular values, while all the ones following are extremely small (negligible). This is exactly what we see in the text output as well. This is a consequence of function 2 being much simpler than the first function. In fact, we see easily that the function values based on the x-variable and the y-variable are independent of each other from which the necessity for only two dominant singular values follows.

We now continue with the second task (Exercise 2), in which we wish to implement the randomized range-finder algorithm, and then, based on this, the Randomized Singular Value Decomposition (RSVD). As mentioned in the introduction, the idea behind this is to approximate our initial matrix F, using a projection unto a subspace of lower dimension, which is randomly generated, in our case unto a matrix based on a Gauss-distribution. We can then proceed for a given relatively small rank to generate an approximation for F, project the approximation up unto the larger original space again, and calculate the error based on the Frobenius norm, which is the only aspect we need to calculate directly from the original matrix F. If the error is too great, we can simply

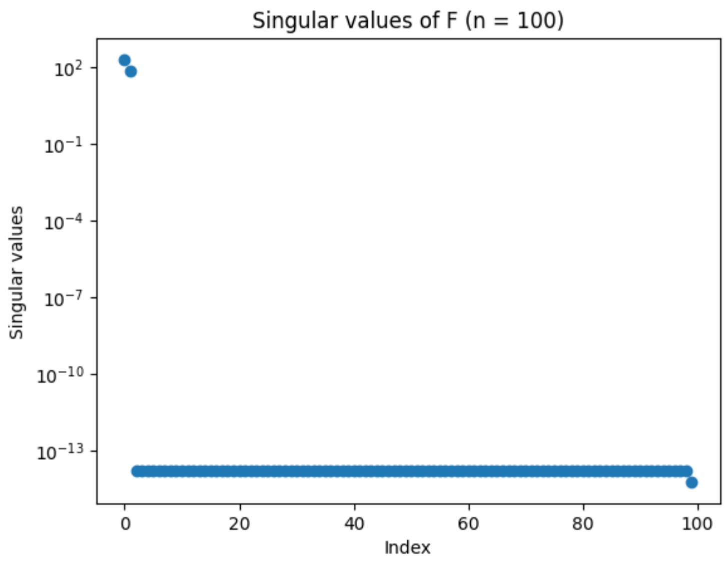

Figure 4: Singular values of F (n = 100)

increase the dimension of our subspace, and repeat the steps, until we attain an error below the desired threshold. In this way, we are never forced to compute the full SVD, like above.

The concrete steps are as follows:

- A matrix F with dimensions $m \times n$ is given.
- Compute the Frobenius norm of the full matrix F.
- We generate $\Omega(\text{n} \times k)$ with entries based on a Gaussian distribution $\mathcal{N}(0,1)$, and k being variable dimensionality, which we will increase until reaching an error less than desired threshold.
- $Y = A\Omega$.
- Next, we carry through a QR-decomposition of Y with Q being orthogonal and capturing the important directions in the column space of A.
- Once we have our Q, we can use it to compute a better approximation of A by projecting it back onto it: $A_k = Q(Q^T A)$
- We then compute the error: $\frac{||A-A_k||_F}{||A||_F}$ and check if it is less than our threshold. If not, we repeat the procedure with a larger k, etc., until an error below the threshold is reached.

This essentially describes our first function in the code, which then, if a threshold is reached, returns the rank of the desired matrix (the k in our randomized ($n \times k$-matrix)), plus the error. In our second function, we use this procedure to compute an approximation of F given a specific threshold. The final function simply returns an approximation given for any given k. For this function, we set $k = 5$ as default, based on the calculations in the first task, which computed a rank of 3 and a rank of 6 for a threshold 0.001 and e^{-6} respectively. Genereally, the initial size of k and the step, with which it should be increased, can be set according to experience, and various factors like the size of the matrix, its complexity, the threshold, etc.

We now proceed to redo the experiments from our first task, and compare the accuracy of the results of our randomized SVD to the standard SVD, as well as the truncated SVD (based on the standard SVD). Since we are mainly concerned with showing the advantages of the RSVD over the standard SVD, we will use only f_1, as this is the more complex of the two. Furthermore, we have included a matrix of dimensions (5000×5000) to demonstrate the advantages of RSVD as clearly as possible, and will compare both of the methods, not simply to their precision, but to the runtime as well as to computer memory used for each. The computations will be made based on the ranks found in the first task, which, as just mentioned, were 3 and 6. This is our generated output:

```
Computing singular values of matrix (100 x 100) of ranks 3 and 6:
Takes 0.00300002 seconds and uses 8192 bytes for the standard SVD
Takes 0.00100923 seconds and uses 104 bytes for the RSVD

Computing singular values of matrix (5000 x 5000) of ranks 3 and 6:
Takes 55.52715492 seconds and uses 200003584 bytes for the standard SVD
Takes 0.19060254 seconds and uses 3477504 bytes for the RSVD

Accuracy of truncated and randomized SVD's:
Accuracy of truncated S with rank 3: 99.99999680200061 %
Accuracy of S from RSVD with rank 3: 99.99998081003181 %

Accuracy of truncated and randomized SVD's:
Accuracy of truncated S with rank 6: 99.99999999999969 %
Accuracy of S from RSVD with rank 6: 99.99999999998381 %

For a matrix with dimensions 5000 x 5000, RSVD is approximately
- 291 times faster
- uses 1/58th of the memory
```

So, as expected, we clearly see the advantages of the RSVD over the standard SVD. Although the accuracy is essentially the same (with the truncated (standard) SVD slightly more accurate with one or two decimal points), the difference in the runtime and in memory use is quite significant (in orders of magnitude!). With even greater dimensionalities and more complex data (than one single mathematical function), the advantages of RSVD over the standard SVD is clear. We will see this in the last task in a moment, where we apply the algorithms to an image, and draw a graph of the runtime vs. accuracy.

(We realize that the code could have been written more efficiently, but time, unfortunately, did not permit for a cleaner implementation; luckily, the result does not suffer from it.)

In repeating the plots for the graphs of $n = 10$ and $n = 100$, we will see that a new plot is generated for each k, since we have to generate a new approximation of F with every increase in k; hence, the differently colored graphs (see Figure 5,6,7 and 8). We notice the same general trend for these plots with the exponential decline (linear in a logarithmically scaled coordinate system) and the two distinct singular values for f_2 both for $n = 10$ and $n = 100$ as for the previous plots.

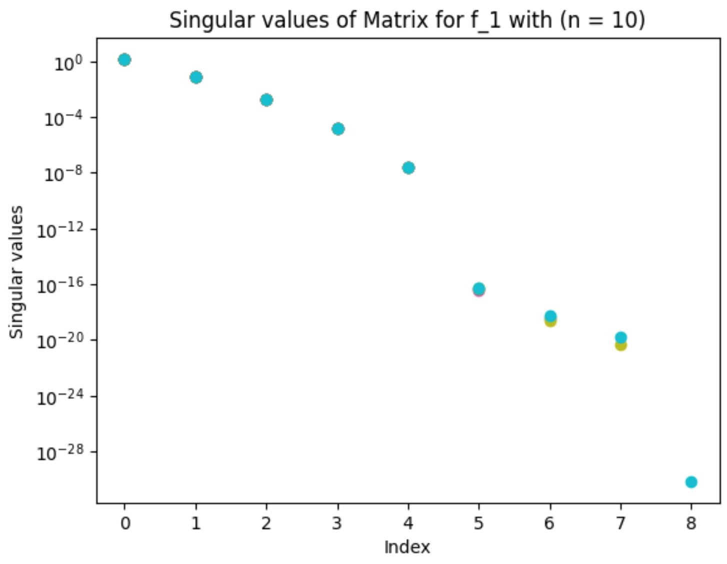

Figure 5: RSVD Singular values of F (n = 10)

In the last task (Exercise 3), we will apply the truncated (standard) SVD and the RSVD to a random gray scale image larger than 1000×1000 pixels. We begin by ensuring that the gray image chosen is a gray scale, as we had some issues with supposed gray scale images actually being RBG pictures. We are choosing tolerances between 0.10 and 0.01, as these, as we will see, are in the range which the greatest changes in the rank of our matrix is needed. The first Python function then computes the corresponding k-values based on the tolerances, which we will use in computing the series of approximations (both truncated (standard) SVD and RSVD), using the `rand_SVD_with_est`-function

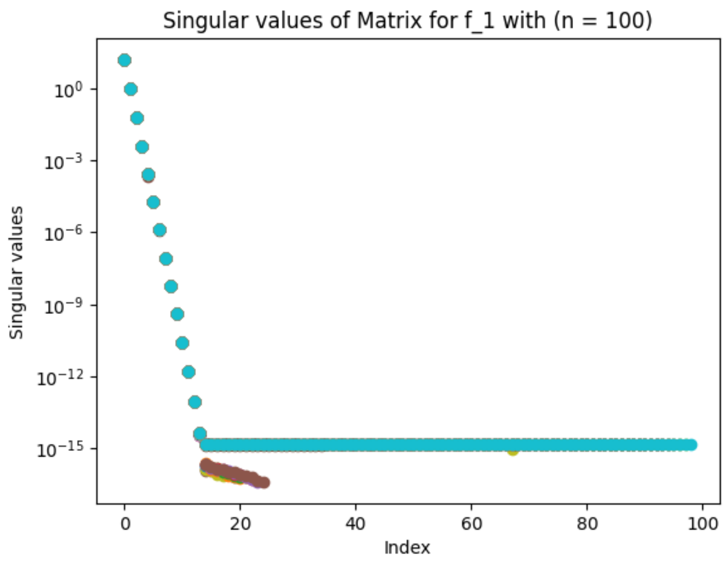

Figure 6: RSVD Singular values of F (n = 100)

from the previous tasks. The next too functions then computes the approximations, each functions saving the times for the truncated (standard) SVD and RSVD, respectively, in a list, which we will use to make plot. We also compute the overall time, by summing up each of the lists, will print the results. Here is the output and the graph, showing the relationship (see Figure 9):

> Total time for the standard SVD is 14.647 seconds, while the total runtime of RSVD is 0.612 seconds.

For comparison with the k values from the matrices generated by the mathematical functions, we also print the list for the values of k, computed based on the tolerances. Where the k value for a tolerance of 10^{-6}, was only 6:

> The values for k are:
> [3, 5, 8, 11, 20, 34, 60, 107, 193, 396]

Conclusion

One key result from Exercise 1 and 2, was, that when comparing the RSVD to the standard SVD in the case of a matrix with dimensions 5000×5000 (with entries computed from the given function f_1) the runtime was persistently around 290 times faster, using only about 1/50th of the memory, but with an accuracy

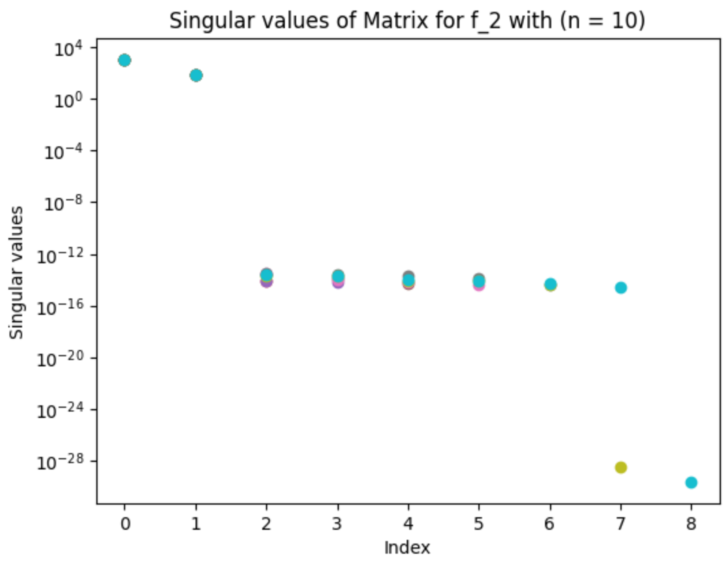

Figure 7: RSVD Singular values of F (n = 10)

within the same range. The author ran the algorithsm several times giving the following results:

For a matrix with dimensions 5000 x 5000, RSVD is approximately
- 240 times faster
- uses 1/19th of the memory

For a matrix with dimensions 5000 x 5000, RSVD is approximately
- 293 times faster
- uses 1/18th of the memory

For a matrix with dimensions 5000 x 5000, RSVD is approximately
- 291 times faster
- uses 1/44th of the memory

For a matrix with dimensions 5000 x 5000, RSVD is approximately
- 294 times faster
- uses 1/52th of the memory

For a matrix with dimensions 5000 x 5000, RSVD is approximately
- 291 times faster
- uses 1/58th of the memory

It is therefore clear from the preceding that the RSVD, although one or two

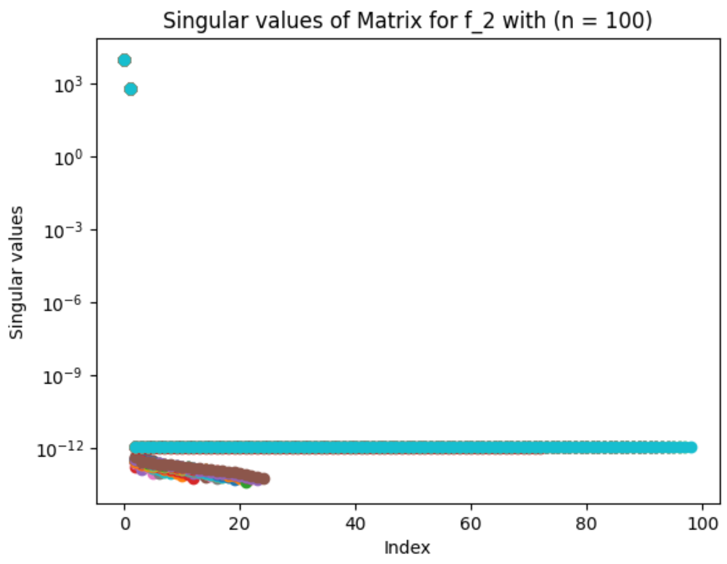

Figure 8: RSVD Singular values of F (n = 10)

decimals less precise, is significantly advantageous both in runtime and memory-use, and the greater the amount of data (e.g. a matrix of 100×100 compared to 5000×5000), as well as the data's complexity, the more this applies. Regarding the question of complexity, this was clear when comparing the matrix of dimensions 1500 x 1500, which needed only a value of $k = 6$ to account for a tolerance of 10^{-6}, where k values for the image, which was 1881 x 1411 pixels increased significantly as the range went from 0.1 to 0.01 - values much larger than 10^{-6}.

In O-notation, the Standard SVD has a runtime or computational complexity of either $O(m, n^2)$ if $m > n$, $O(m^2, n)$ if $m < n$, or $O(n^3)$ if $n = m$, compared to approximately $O(mnk)$ for small k's for the RSVD. If k is significantly smaller, then the runtime too, is significantly smaller, as we as have, at least in a limited, yet quite definite fashion, illustrated in this small report.

The Python code

```python

# 1. Compression by singular value decompositon:

import numpy as np
import matplotlib.pyplot as plt

matrix_size = [10,100,1500]
```

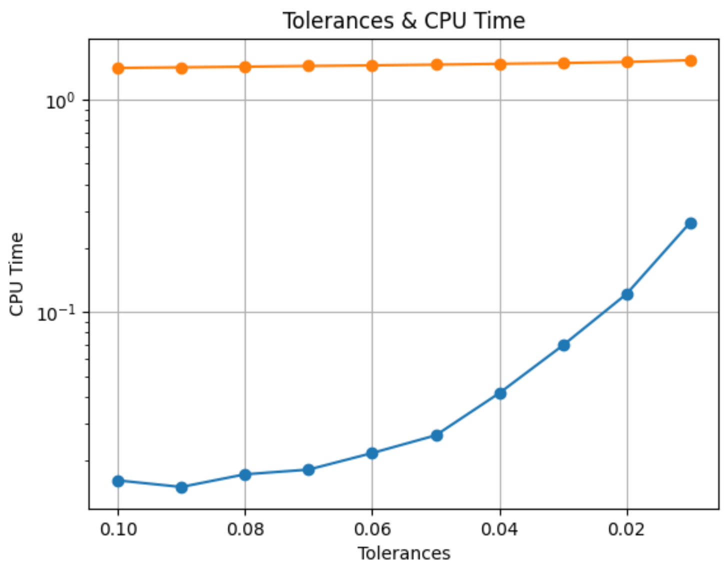

Figure 9: Tolerances vs. CPU runtime for truncated (standard) SVD and RSVD respectively

```python
threshold = [1e-3,1e-6]

# Define the function T(x):
def T_1(x):
    T_function = np.exp(-0.4*np.tanh((x-7.7)/8))
    return T_function

# Define the function f(x,y):
def f_1(x,y):
    return (1/(np.sqrt(2*np.pi*T_1(x))))*np.exp(-(y**2)/2*
        T_1(x))
    return

# Second function:
def f_2(x,y):
    return -(x**2 + y**2) + 4

# Define the matrix F given an input size and a chosen
    function:
def matrix_F(f, m, n, x_1=0.1, x_2=14.5, y_1=-6, y_2=6):
    x = np.linspace(x_1,x_2,m)
    y = np.linspace(y_1,y_2,n)
    m_ = len(x)
    n_ = len(y)
    F = np.zeros([m_,n_])

    # Creating each element of the matrix using for-loops:
    for i in range(len(x)):
        for j in range(len(y)):
            F[i][j] = f(x[i],y[j])

    # Return F
    return F

# Calculate S via built-in SVD-function and plot the singular values,
# using a log-scale for the y-axis:
def SVD_plot_singular(f, m, n, x_1=0.1, x_2=14.5, y_1=-6, y_2=6):
    _,S,_ = np.linalg.svd(matrix_F(f,m,n,x_1,x_2,y_1,y_2))
    # Plot singular values:
    plt.figure()
    plt.plot(S, 'o')
    plt.xlabel('Index')
    plt.ylabel('Singular values')
    plt.yscale('log')
    plt.title(f"Singular values of F (n = {m})")

# Determining the rank r of truncated standard SVD for a given threshold:
def frobenius_singular_truncated(f,m,n,error,x_1=0.1,x_2=14.5,y_1=-6,y_2=6):
    _,S,_ = np.linalg.svd(matrix_F(f,m,n,x_1,x_2,y_1,y_2))
    frob_norm = np.sqrt(np.sum(S**2))
    for r in range(1,min(m,n)+1):
        trunc_error = np.sqrt(np.sum(S[r:]**2)) / frob_norm
        if trunc_error < error:
            return r, error, trunc_error
    return min(m,n), error, trunc_error

# Plot singular values for n = 10 and n = 100 of f(x,y):
for i in matrix_size:
    SVD_F = SVD_plot_singular(f_1,i,i)

# Find sufficient rank for to meet threshold criteria:
ranks_f1 = []
for p in [1,2]:
    print(f"\nMatrix F based on the given function (n = m = {matrix_size[p]}):")
    for i in range(len(threshold)):
        frob_trunc_r,frob_trunc_error,difference = frobenius_singular_truncated(f_1,matrix_size[p],matrix_size[p],threshold[i])
        ranks_f1.append(frob_trunc_r)
        print(f"For an error below {frob_trunc_error} a singular matrix of rank {frob_trunc_r} is sufficient.")

# Repeating the exercise for the second function:

# Use the already defined function for computing the singular values and plot them for n = 10 and n = 100:
for i in matrix_size:
    SVD_F = SVD_plot_singular(f_2,i,i,-2,2,-2,2)

# Find the sufficient rank needed for given threshold:
ranks_f2 = []
print(f"Matrix F based on first own function (n = m = {matrix_size[p]}):")
for n in range(len(threshold)):
    frob_trunc_r,frob_trunc_error,_ = frobenius_singular_truncated(f_2,matrix_size[p],matrix_size[p],threshold[n],-2,2,-2,2)
    ranks_f2.append(frob_trunc_r)
    print(f"For an error below {frob_trunc_error} a singular matrix of rank {frob_trunc_r} is sufficient.")

# 2. Randomized singular value decomposition
# 2.1: Implement the randomized range-finder algorithm and randomized SVD:

# Using Frobenius norm to estimate the necessary k for a given threshold:
def frob_norm_est(A,threshold=1e-3,k=1,step=1):
    m,n = A.shape
    frob_A = np.linalg.norm(A, ord='fro')
    for i in range(k,min(m,n),step):
        omega = np.random.normal(loc=0,scale=1,size=(n,i))
        Y = A @ omega
        Q,_ = np.linalg.qr(Y)
        A_k = Q @ (Q.T @ A)
        frob_difference = np.linalg.norm(A - A_k, ord='fro')
        error = frob_difference / frob_A
        if error < threshold:
            return i,error
    return np.min([m,n]), error

# Randomized SVD using the frob_norm_est function:
def rand_SVD_with_est(A, threshold=1e-3, k=1, step=1):
    k_new, error = frob_norm_est(A,threshold,k,step)
    m,n = A.shape
    omega = np.random.normal(loc=0, scale=1, size=(n,k_new))
    Y = A @ omega
    Q,_ = np.linalg.qr(Y)
    B = Q.T @ A
    U_wave,S_list,Vt = np.linalg.svd(B, full_matrices=False)
    U = Q @ U_wave
    return U,S_list,Vt,k_new,error

# Randomized SVD without using frob_norm_est function,
    instead simply computing for a given k:
def rand_SVD_without_est(A, k=5):
    m,n = A.shape
    omega = np.random.normal(loc=0, scale=1, size=(n,k))
    Y = A @ omega
    Q,_ = np.linalg.qr(Y)
    B = Q.T @ A
    U_wave,S_list,Vt = np.linalg.svd(B,full_matrices=False)
    U = Q @ U_wave
    return U,S_list,Vt

# 2.2 Redo experiments of Exercise 1
# Compare the truncated standard SVD to the RSVD

import time
import psutil
import os

k_begin = 1 # define an initial k value
functions = f_1,f_2
M_100 = matrix_F(f_1,100,100) # compute F for n = 100
M_5000 = matrix_F(f_1,5000,5000) # compute F for n = 5000

# Create lists for the respective singular values:
S_trunc = []
S_RSVD = []

# Plotting singular values now for RSVD for n = 10 and n =
    100:
for f in functions:
    for n in matrix_size:
        plt.figure()
        plt.xlabel('Index')
        plt.ylabel('Singular values')
        plt.yscale('log')
        plt.title(f"Singular values of Matrix for {f.
            __name__} with (n = {n})")
        A = matrix_F(f,n,n)
        for k in range(n):
            _,S,_ = rand_SVD_without_est(A,k)
            plt.plot(S, 'o')

# Computing standard SVD and truncating for already computed
     ranks for matrix (100 x 100):
```

```
time_start_1 = time.time()
process_1 = psutil.Process(os.getpid())
memory_before_1 = process_1.memory_info().rss
_,S,_ = np.linalg.svd(M_100)
for i in ranks_f1:
    S_trunc.append(S[:(i)])
memory_after_1 = process_1.memory_info().rss
time_end_1 = time.time() - time_start_1
print(f"Computing singular values of matrix (100 x 100) of
    ranks 3 and 6:")
print(f"Takes {time_end_1:.8f} seconds and uses {
    memory_after_1 - memory_before_1} bytes for the standard
    SVD")

# Computing RSVD for already computed ranks for matrix (100
    x 100):
time_start_2 = time.time()
process_2 = psutil.Process(os.getpid())
memory_before_2 = process_2.memory_info().rss
for i in ranks_f1:
    S_RSVD.append(rand_SVD_without_est(M_100,k=i)[1])
memory_after_2 = process_2.memory_info().rss
time_end_2 = time.time() - time_start_2
print(f"Takes {time_end_2:.8f} seconds and uses {
    memory_after_2 - memory_before_2} bytes for the RSVD\n")

# Computing standard SVD and truncating for already computed
     ranks for matrix (5000 x 5000):
time_start_3 = time.time()
process_3 = psutil.Process(os.getpid())
memory_before_3 = process_3.memory_info().rss
_,S,_ = np.linalg.svd(M_5000)
for i in ranks_f1:
    S_trunc.append(S[:(i)])
memory_after_3 = process_3.memory_info().rss
memory_used_3 = memory_after_3 - memory_before_3
time_end_3 = time.time() - time_start_3

print(f"Computing singular values of matrix (5000 x 5000) of
     ranks 3 and 6:")
print(f"Takes {time_end_3:.8f} seconds and uses {
    memory_used_3} bytes for the standard SVD")

# Computing RSVD for already computed ranks for matrix (100
    x 100):
S_RSVD = []
time_start_4 = time.time()
process_4 = psutil.Process(os.getpid())
memory_before_4 = process_4.memory_info().rss
for i in ranks_f1:
```

```python
    S_RSVD.append(rand_SVD_without_est(M_5000,k=i)[1])
memory_after_4 = process_4.memory_info().rss
memory_used_4 = memory_after_4 - memory_before_4
time_end_4 = time.time() - time_start_4

print(f"Takes {time_end_4:.8f} seconds and uses {memory_used_4} bytes for the RSVD")

# We compare the accuracy of the results using the Frobenius norm:
for i in [0,1]:
    frob_S = np.linalg.norm(S)
    frob_S_trunc = np.linalg.norm(S[:ranks_f1[i]]) / frob_S
    frob_S_RSVD = np.linalg.norm(S_RSVD[i]) / frob_S
    print()

    print(f"Accuracy of truncated and randomized SVD's:")
    print(f"Accuracy of truncated S with rank {ranks_f1[i]}: {frob_S_trunc*100} %")
    print(f"Accuracy of S from RSVD with rank {ranks_f1[i]}: {frob_S_RSVD*100} %")

# If-statement, in the case a memory-use of 0 bytes is measured:
if memory_used_4 == 0:
    print(f"\nFor a matrix with dimensions 5000 x 5000, RSVD is approximately\n- {(time_end_3 / time_end_4):.0f} times faster\n- Uses no memory as compared to {memory_used_3} bytes for the standard SVD")
else:
    print(f"\nFor a matrix with dimensions 5000 x 5000, RSVD is approximately\n- {(time_end_3 / time_end_4):.0f} times faster\n- uses 1/{(memory_used_3 / memory_used_4):.0f}th of the memory")

# 3. Image compression:

import time
import matplotlib.pyplot as plt
from skimage.color import rgb2gray

# Load image to be used:
image = plt.imread('swimming_with_turtles_dudok_de_witt.jpg')

# Convert image to gray scale, if RGB:
if image.ndim == 3:  # If the image has 3 channels (RGB)
    image_gray = rgb2gray(image)
else:
    image_gray = image  # If the image is already grayscale
```

```

B = image_gray # B is a matrix and reference for the gray scale image
tolerances = np.round(np.arange(0.1, 0.0099, -0.01),2) # Set up range of tolerances to be used
k = 1 # Initial k value

# Create lists to store results to be used for the plots:
runtime_RSVD = []
runtime_standard = []
k_list = []

# Find the k's for the given tolerances from above and save in list:
for n in range(len(tolerances)):
    U,S_list,Vt,k_new, _ = rand_SVD_with_est(B,tolerances[n],k)
    S = np.diag(S_list)
    B_k = U @ (S @ Vt)
    k_list.append(k_new)

# Stanard SVD computation with already computed k values, and creating list for the runtimes:
time_start_1 = time.time()
U,S_list,Vt = np.linalg.svd(B,full_matrices=False)
for n in k_list:
    size = np.arange(0,n)
    B_k = U[:,size] @ np.diag(S_list[size]) @ Vt[size,:]
    time_finish_1 = time.time() - time_start_1
    runtime_standard.append(time_finish_1)

total_time_standard = np.sum(runtime_standard)

# RSVD computation with already computed k values, and creating list for the runtimes times:
for n in k_list:
    time_start_0 = time.time()
    U,S_list,Vt = rand_SVD_without_est(B,n)
    S = np.diag(S_list)
    B_k = U @ (S @ Vt)
    time_end_0 = time.time() - time_start_0
    runtime_RSVD.append(time_end_0)

total_time_RSVD = np.sum(runtime_RSVD)

print(f"\nTotal time for the standard SVD is {total_time_standard:.3f} seconds, while the total runtime of RSVD is {total_time_RSVD:.3f} seconds.")

print(f"\nThe values for k are:\n{k_list}")
```

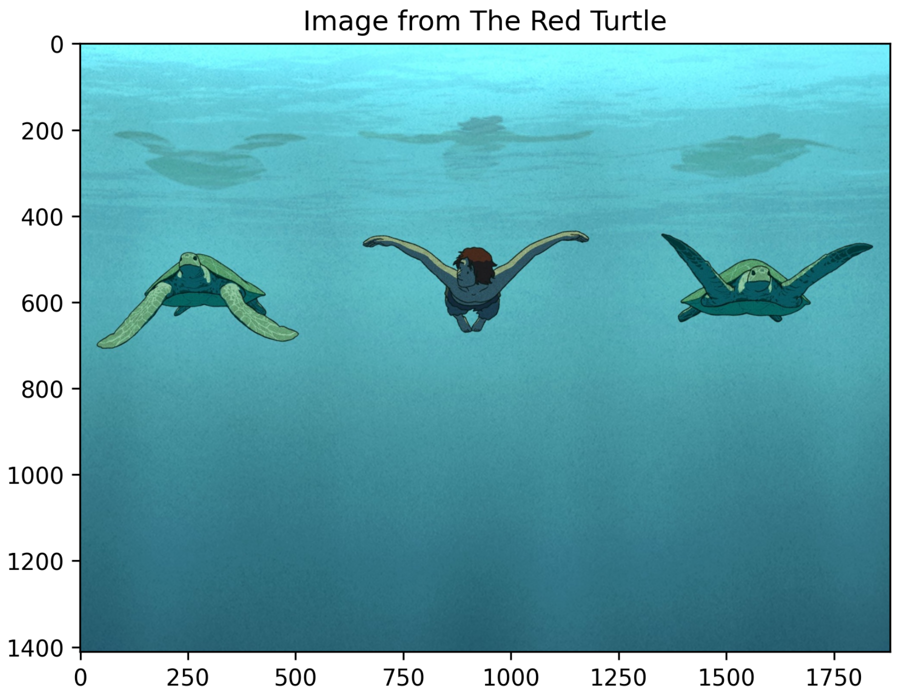

Figure 10: Color image used before conversion

```

# Plot the tolerances vs. CPU time for the two methods
plt.figure()
plt.plot(tolerances, runtime_RSVD, 'o-')
plt.plot(tolerances, runtime_standard, 'o-')
plt.title('Tolerances & CPU Time')
plt.xlabel('Tolerances')
plt.ylabel('CPU Time')
plt.gca().invert_xaxis()
plt.yscale('log')
plt.grid()
plt.show()

# Showing the image
plt.figure(dpi=300)
plt.title("Image from The Red Turtle")
plt.imshow(image);
```