Mathematisches Praktikum (MB08)
Linear Algebra und Data Science - Programmierpraktikum

UNIVERSITÄT BONN

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

“Linear algebra is also used in most sciences and fields of engineering, because it allows modeling many natural phenomena, and computing efficiently with such models. ”
- Wikipedia

“What is data science, if not linear algebra persevering?”
- Vision (maybe)

“Math is the language of engineering, but coding is believing [...] it.”
- Chad Jenkins

Koordinaten.

- Dozent: Philipp Hieronymi (`hieronymi@math.uni-bonn.de`)
- Zeit: Freitag 10.15-12, PC-Pool IAM, Endenicher Allee 60, Nebengebäude
- Erstes Treffen: Freitag 31.10.

Anmeldung. **Wichtig!** Es gibt 15 Plätze in diesem Praktikum. Falls Sie Interesse haben, schicken Sie bitte bis zum 12.10. eine E-Mail an `hieronymi@math.uni-bonn.de`. Teilen Sie mir kurz mit, ob Sie Vorkenntnisse über Python haben und ob Sie einen eigenen Laptop/Tablet zum Praktikum mitbringen können.

Inhalt. Immer wieder stellen Schülerinnen und Schüler, aber auch Studierende die Frage: wozu lerne ich diese abstrakte, vermeintlich anwendungsfreie (reine) Mathematik? Das ist eigentlich erstaunlich, da viele unsere moderne Technologien fundamental und direkt auf Resultaten aus der (reinen) Mathematik basieren. In diesem (Programmier-)Praktikum sollen Studierende lernen, wie apriori abstrakte Theoreme der Linear Algebra insbesondere in der Informationstechnologie und der Data Science unmittelbare Anwendung finden. Beispiele sind

- Googles PageRank Algorithmus,
- Bild- und Tonkompressionsmethoden wie JPEG and MP3,
- Automatische Gesichts- und Schrifterkennung,
- Data Science und Machine Learning Anwendungen.

Wir werden außerdem die Verteilung von Memes in soziale Netzwerke analysieren, die Hauptfigur in *Game of Thrones* bestimmen, eine Karaoke-Version von Queens *Bohemian Rhapsody* erstellen und vieles mehr.

Vorkenntnisse. Teilnehmer müssen die Vorlesung *Linear Algebra (MB05)* oder eine entsprechende andere Vorlesung zur linearen Algebra besucht haben. Es werden allerdings auch Videos zur Verfügung gestellt, um everntuell fehlenden Stoff nachzuholen.

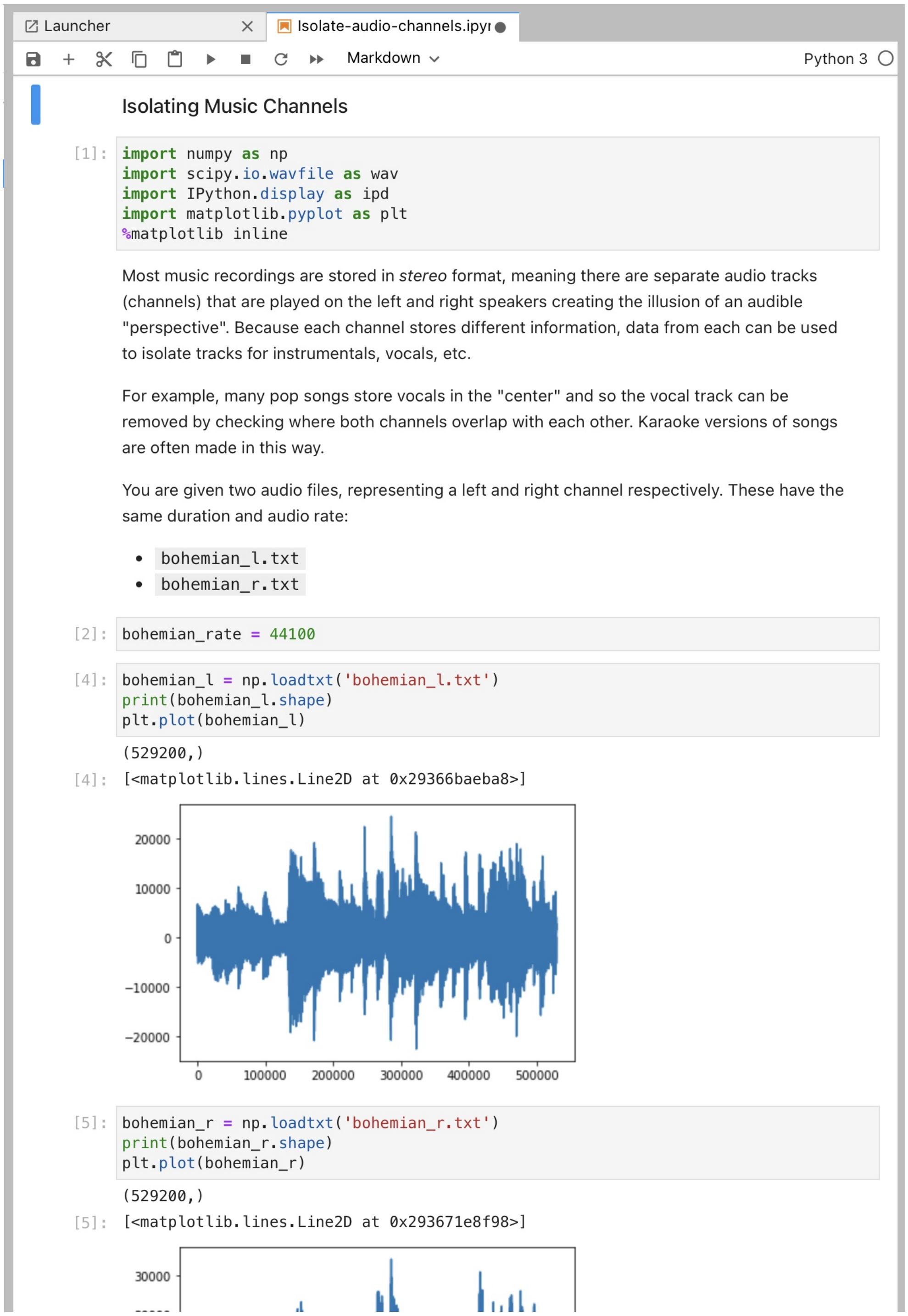
Wir benutzen die Programmiersprache *Python* über die Online-Platform *PrairieLearn*. Vorkenntnisse in Python sind sicherlich hilfreich, aber nicht notwendig, wenn die Bereitschaft vorhanden ist, sich etwas einzuarbeiten. Auch wenn die Veranstaltung *Programmierpraktikum* heißt, werden wir Python (und PrairieLearn) eher als einen hochwertigen Ersatz für einen Taschenrechner und nicht als eine eigenständige Programmiersprache ansehen. Hier wird es um Mathematik gehen und nicht um Informatik!

Ablauf. Sie werden an 11 Terminen als kleine Gruppen zusammen an Python Worksheets in PrairieLearn arbeiten. Insbesondere für Studierende, die noch nie Python benutzt haben, werden wir ein Python Tutorial in der ersten Woche habe. In jeder Woche erhalten Sie in PrairieLearn auch eine kleine Programmier-Hausaufgabe.

Termin	Lab	Thema
31.10.	1	Python tutorial, Working with vectors
7.11.	2	Matrix operations
14.11.	3	Solving systems of linear equations
21.11.	4	Graphs and Algebraic Graph Theory
28.11.	5	Data compression
5.12.	6	Markov Chains
12.12.	-	Kein Praktikum
19.12.	7	Dynamical Systems
9.1.	8	Linear Regression
16.1	9	SVD and applications
23.1.	10	Principal Component Analysis
30.1.	11	Vorträge
6.2.	12	Vorträge (falls notwendig)

Technische Anforderungen. Wir benutzen das Onlinesystem *PrairieLearn*. Dies funktioniert ohne Installation von weiterer Software auf im Prinzip allen Betriebsystemen und modernen Browsern. Daher können Sie einfach Ihren eigenen Laptop oder Ihr eigenes Tablet benutzen (oder als Gruppe einen Laptop benutzen). Falls Sie keinen Laptop oder Tablet besitzen, dann ist dies auch kein Problem und Sie können einen Institutsrechner benutzen.

Eindrücke. Das Material wurde an der University of Illinois entwickelt und ist daher auf Englisch. Die Unterrichtssprache wird allerdings Deutsch sein. Auf den folgenden Seiten finden Sie einige Screenshots, damit Sie einen Eindruck davon bekommen, was Sie in diesem Praktikum erwartet.



Isolating Music Channels

```
[1]: import numpy as np
     import scipy.io.wavfile as wav
     import IPython.display as ipd
     import matplotlib.pyplot as plt
     %matplotlib inline
```

Most music recordings are stored in *stereo* format, meaning there are separate audio tracks (channels) that are played on the left and right speakers creating the illusion of an audible "perspective". Because each channel stores different information, data from each can be used to isolate tracks for instrumentals, vocals, etc.

For example, many pop songs store vocals in the "center" and so the vocal track can be removed by checking where both channels overlap with each other. Karaoke versions of songs are often made in this way.

You are given two audio files, representing a left and right channel respectively. These have the same duration and audio rate:

- `bohemian_l.txt`
- `bohemian_r.txt`

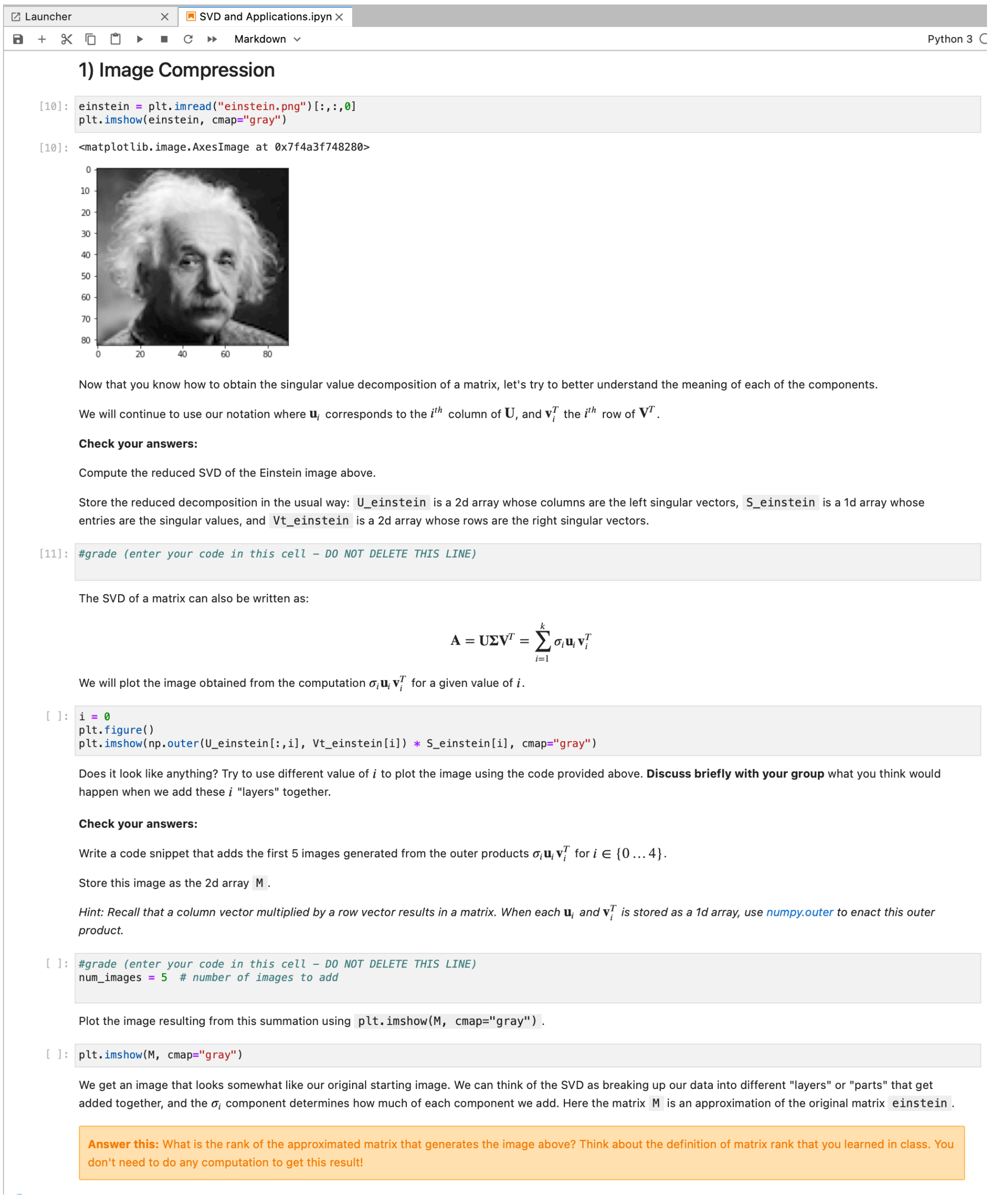
```
[2]: bohemian_rate = 44100
```

```
[4]: bohemian_l = np.loadtxt('bohemian_l.txt')
     print(bohemian_l.shape)
     plt.plot(bohemian_l)
```

```
     (529200,)
[4]: [<matplotlib.lines.Line2D at 0x29366baeba8>]
```

```
[5]: bohemian_r = np.loadtxt('bohemian_r.txt')
     print(bohemian_r.shape)
     plt.plot(bohemian_r)
```

```
     (529200,)
[5]: [<matplotlib.lines.Line2D at 0x293671e8f98>]
```



1) Image Compression

```
[10]: einstein = plt.imread("einstein.png")[:,:,0]
      plt.imshow(einstein, cmap="gray")
```

```
[10]: <matplotlib.image.AxesImage at 0x7f4a3f748280>
```

Now that you know how to obtain the singular value decomposition of a matrix, let's try to better understand the meaning of each of the components.

We will continue to use our notation where $\mathbf{u}_i$ corresponds to the i^{th} column of $\mathbf{U}$, and $\mathbf{v}_i^T$ the i^{th} row of $\mathbf{V}^T$.

Check your answers:

Compute the reduced SVD of the Einstein image above.

Store the reduced decomposition in the usual way: `U_einstein` is a 2d array whose columns are the left singular vectors, `S_einstein` is a 1d array whose entries are the singular values, and `Vt_einstein` is a 2d array whose rows are the right singular vectors.

```
[11]: #grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

The SVD of a matrix can also be written as:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

We will plot the image obtained from the computation $\sigma_i \mathbf{u}_i \mathbf{v}_i^T$ for a given value of i.

```
[ ]: i = 0
     plt.figure()
     plt.imshow(np.outer(U_einstein[:,i], Vt_einstein[i]) * S_einstein[i], cmap="gray")
```

Does it look like anything? Try to use different value of i to plot the image using the code provided above. **Discuss briefly with your group** what you think would happen when we add these i "layers" together.

Check your answers:

Write a code snippet that adds the first 5 images generated from the outer products $\sigma_i \mathbf{u}_i \mathbf{v}_i^T$ for $i \in \{0 \dots 4\}$.

Store this image as the 2d array `M`.

Hint: Recall that a column vector multiplied by a row vector results in a matrix. When each $\mathbf{u}_i$ and $\mathbf{v}_i^T$ is stored as a 1d array, use numpy.outer to enact this outer product.

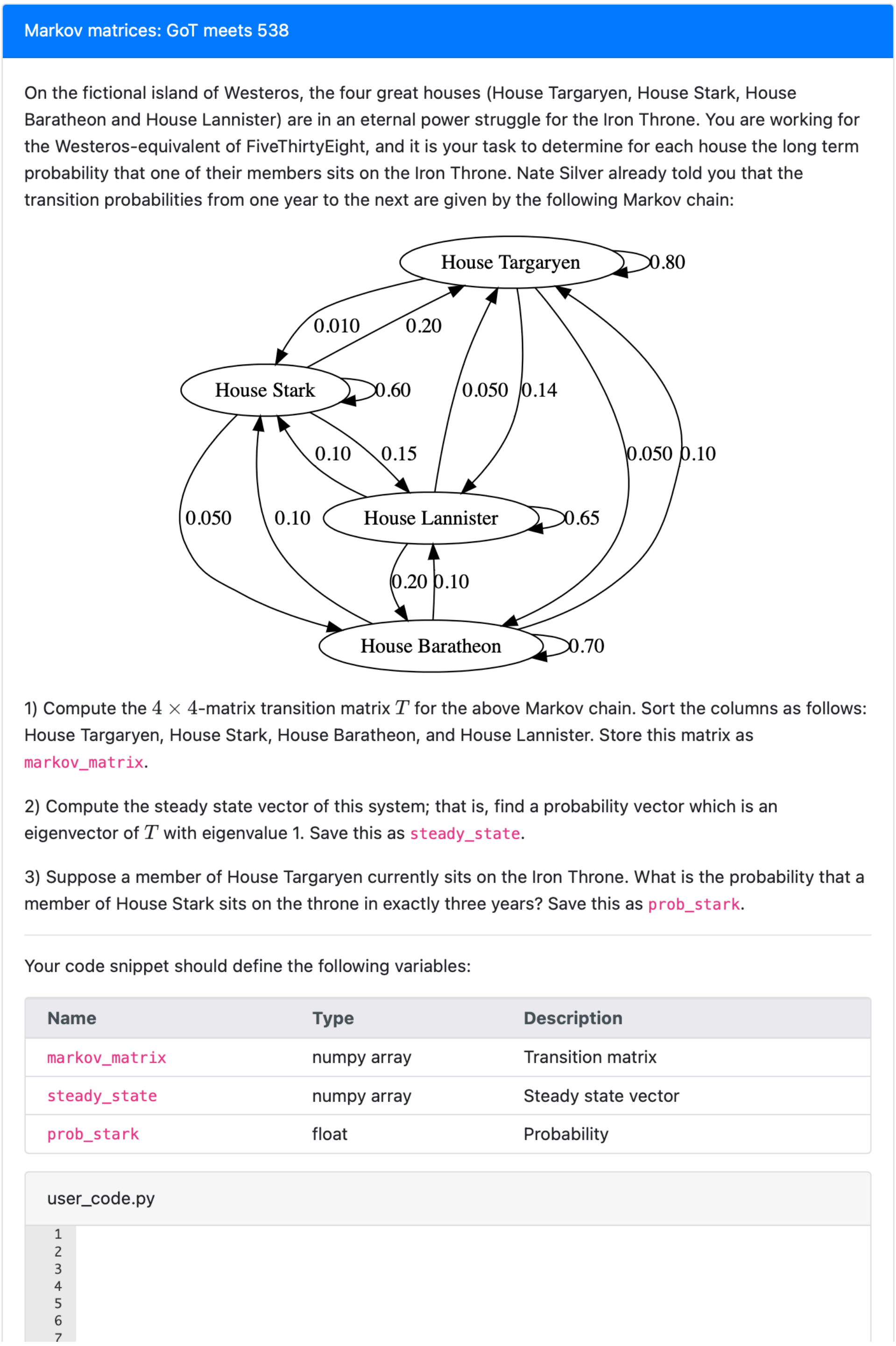
```
[ ]: #grade (enter your code in this cell - DO NOT DELETE THIS LINE)
     num_images = 5  # number of images to add
```

Plot the image resulting from this summation using `plt.imshow(M, cmap="gray")`.

```
[ ]: plt.imshow(M, cmap="gray")
```

We get an image that looks somewhat like our original starting image. We can think of the SVD as breaking up our data into different "layers" or "parts" that get added together, and the σ_i component determines how much of each component we add. Here the matrix `M` is an approximation of the original matrix `einstein`.

Answer this: What is the rank of the approximated matrix that generates the image above? Think about the definition of matrix rank that you learned in class. You don't need to do any computation to get this result!



Markov matrices: GoT meets 538

On the fictional island of Westeros, the four great houses (House Targaryen, House Stark, House Baratheon and House Lannister) are in an eternal power struggle for the Iron Throne. You are working for the Westeros-equivalent of FiveThirtyEight, and it is your task to determine for each house the long term probability that one of their members sits on the Iron Throne. Nate Silver already told you that the transition probabilities from one year to the next are given by the following Markov chain:

1) Compute the 4×4-matrix transition matrix T for the above Markov chain. Sort the columns as follows: House Targaryen, House Stark, House Baratheon, and House Lannister. Store this matrix as `markov_matrix`.

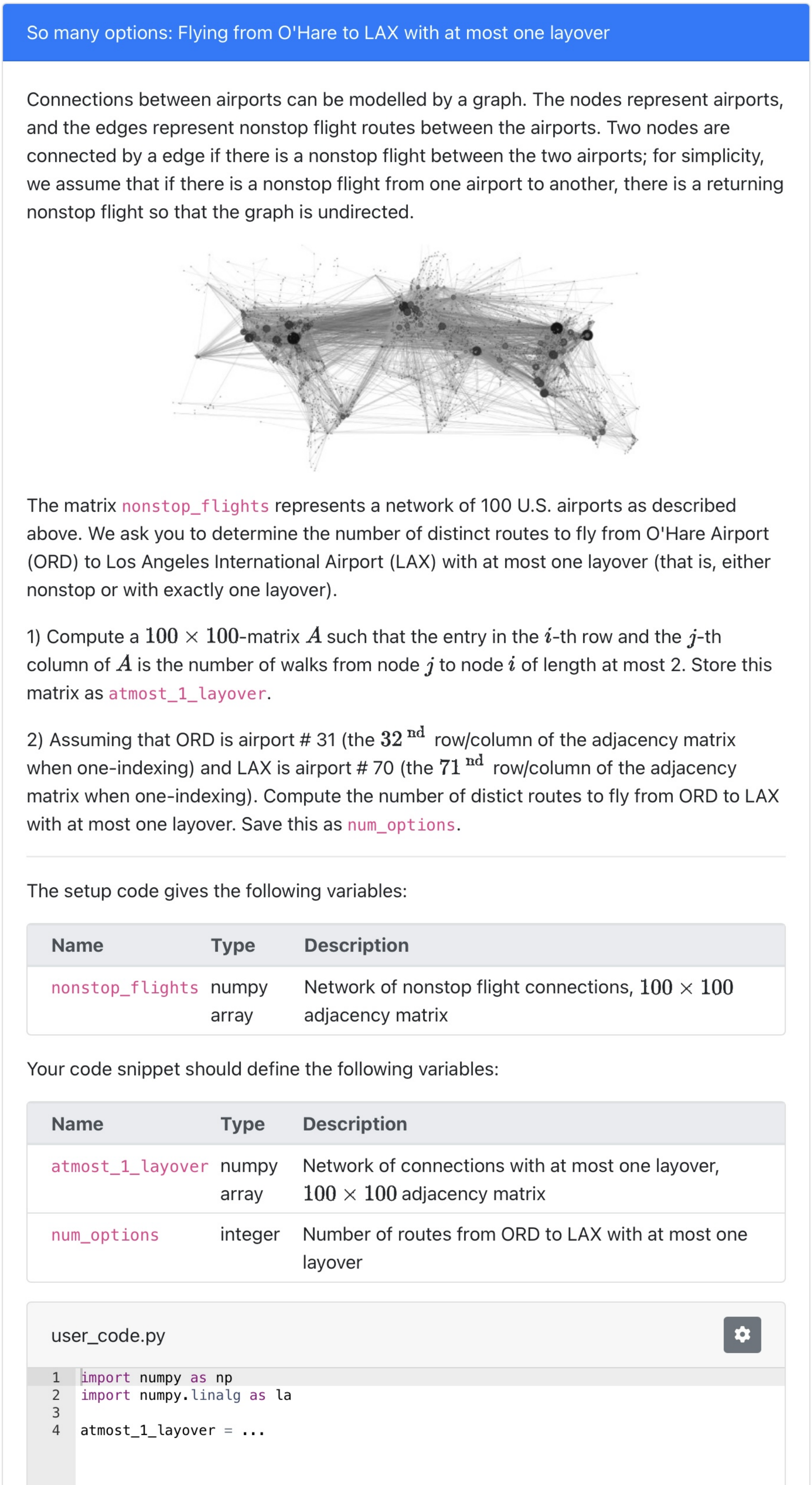
2) Compute the steady state vector of this system; that is, find a probability vector which is an eigenvector of T with eigenvalue 1. Save this as `steady_state`.

3) Suppose a member of House Targaryen currently sits on the Iron Throne. What is the probability that a member of House Stark sits on the throne in exactly three years? Save this as `prob_stark`.

Your code snippet should define the following variables:

Name	Type	Description
`markov_matrix`	numpy array	Transition matrix
`steady_state`	numpy array	Steady state vector
`prob_stark`	float	Probability

user_code.py



So many options: Flying from O'Hare to LAX with at most one layover

Connections between airports can be modelled by a graph. The nodes represent airports, and the edges represent nonstop flight routes between the airports. Two nodes are connected by a edge if there is a nonstop flight between the two airports; for simplicity, we assume that if there is a nonstop flight from one airport to another, there is a returning nonstop flight so that the graph is undirected.

The matrix `nonstop_flights` represents a network of 100 U.S. airports as described above. We ask you to determine the number of distinct routes to fly from O'Hare Airport (ORD) to Los Angeles International Airport (LAX) with at most one layover (that is, either nonstop or with exactly one layover).

1) Compute a 100×100-matrix A such that the entry in the i-th row and the j-th column of A is the number of walks from node j to node i of length at most 2. Store this matrix as `atmost_1_layover`.

2) Assuming that ORD is airport # 31 (the 32^{nd} row/column of the adjacency matrix when one-indexing) and LAX is airport # 70 (the 71^{nd} row/column of the adjacency matrix when one-indexing). Compute the number of distict routes to fly from ORD to LAX with at most one layover. Save this as `num_options`.

The setup code gives the following variables:

Name	Type	Description
`nonstop_flights`	numpy array	Network of nonstop flight connections, 100×100 adjacency matrix

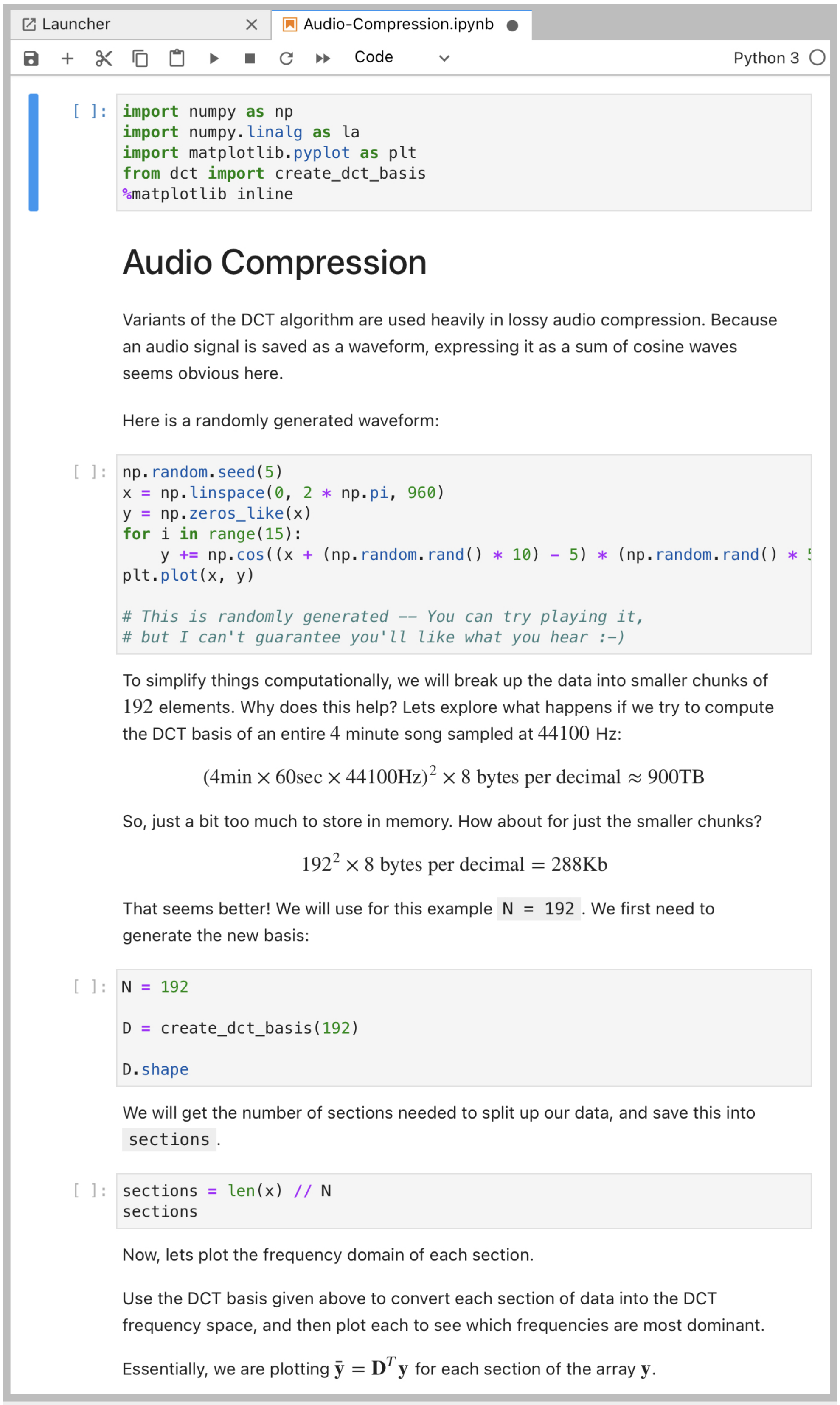
Your code snippet should define the following variables:

Name	Type	Description
`atmost_1_layover`	numpy array	Network of connections with at most one layover, 100×100 adjacency matrix
`num_options`	integer	Number of routes from ORD to LAX with at most one layover

user_code.py

```python
import numpy as np
import numpy.linalg as la

atmost_1_layover = ...
```



```
[ ]: import numpy as np
     import numpy.linalg as la
     import matplotlib.pyplot as plt
     from dct import create_dct_basis
     %matplotlib inline
```

Audio Compression

Variants of the DCT algorithm are used heavily in lossy audio compression. Because an audio signal is saved as a waveform, expressing it as a sum of cosine waves seems obvious here.

Here is a randomly generated waveform:

```
[ ]: np.random.seed(5)
     x = np.linspace(0, 2 * np.pi, 960)
     y = np.zeros_like(x)
     for i in range(15):
         y += np.cos((x + (np.random.rand() * 10) - 5) * (np.random.rand() * 5
     plt.plot(x, y)

     # This is randomly generated -- You can try playing it,
     # but I can't guarantee you'll like what you hear :-)
```

To simplify things computationally, we will break up the data into smaller chunks of 192 elements. Why does this help? Lets explore what happens if we try to compute the DCT basis of an entire 4 minute song sampled at 44100 Hz:

$$(4\text{min} \times 60\text{sec} \times 44100\text{Hz})^2 \times 8 \text{ bytes per decimal} \approx 900\text{TB}$$

So, just a bit too much to store in memory. How about for just the smaller chunks?

$$192^2 \times 8 \text{ bytes per decimal} = 288\text{Kb}$$

That seems better! We will use for this example `N = 192`. We first need to generate the new basis:

```
[ ]: N = 192

     D = create_dct_basis(192)

     D.shape
```

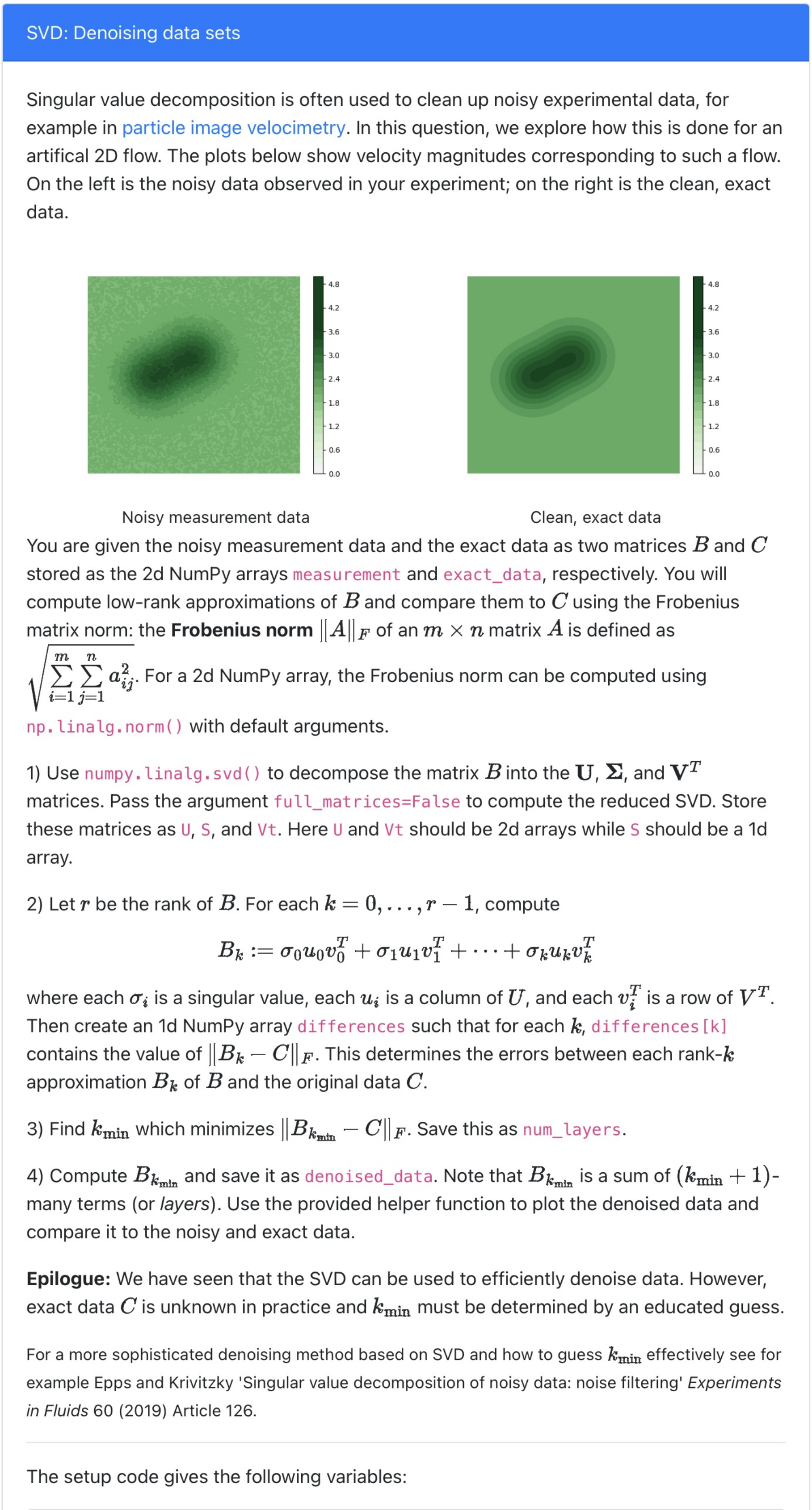
We will get the number of sections needed to split up our data, and save this into `sections`.

```
[ ]: sections = len(x) // N
     sections
```

Now, lets plot the frequency domain of each section.

Use the DCT basis given above to convert each section of data into the DCT frequency space, and then plot each to see which frequencies are most dominant.

Essentially, we are plotting $\bar{\mathbf{y}} = \mathbf{D}^T\mathbf{y}$ for each section of the array $\mathbf{y}$.



SVD: Denoising data sets

Singular value decomposition is often used to clean up noisy experimental data, for example in particle image velocimetry. In this question, we explore how this is done for an artifical 2D flow. The plots below show velocity magnitudes corresponding to such a flow. On the left is the noisy data observed in your experiment; on the right is the clean, exact data.

Noisy measurement data

Clean, exact data

You are given the noisy measurement data and the exact data as two matrices B and C stored as the 2d NumPy arrays `measurement` and `exact_data`, respectively. You will compute low-rank approximations of B and compare them to C using the Frobenius matrix norm: the **Frobenius norm** $\|A\|_F$ of an $m \times n$ matrix A is defined as $\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}$. For a 2d NumPy array, the Frobenius norm can be computed using `np.linalg.norm()` with default arguments.

1) Use `numpy.linalg.svd()` to decompose the matrix B into the $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}^T$ matrices. Pass the argument `full_matrices=False` to compute the reduced SVD. Store these matrices as `U`, `S`, and `Vt`. Here `U` and `Vt` should be 2d arrays while `S` should be a 1d array.

2) Let r be the rank of B. For each $k = 0, \dots, r-1$, compute

$$B_k := \sigma_0 u_0 v_0^T + \sigma_1 u_1 v_1^T + \cdots + \sigma_k u_k v_k^T$$

where each σ_i is a singular value, each u_i is a column of U, and each v_i^T is a row of V^T. Then create an 1d NumPy array `differences` such that for each k, `differences[k]` contains the value of $\|B_k - C\|_F$. This determines the errors between each rank-k approximation B_k of B and the original data C.

3) Find k_{min} which minimizes $\|B_{k_{\text{min}}} - C\|_F$. Save this as `num_layers`.

4) Compute $B_{k_{\text{min}}}$ and save it as `denoised_data`. Note that $B_{k_{\text{min}}}$ is a sum of $(k_{\text{min}} + 1)$-many terms (or *layers*). Use the provided helper function to plot the denoised data and compare it to the noisy and exact data.

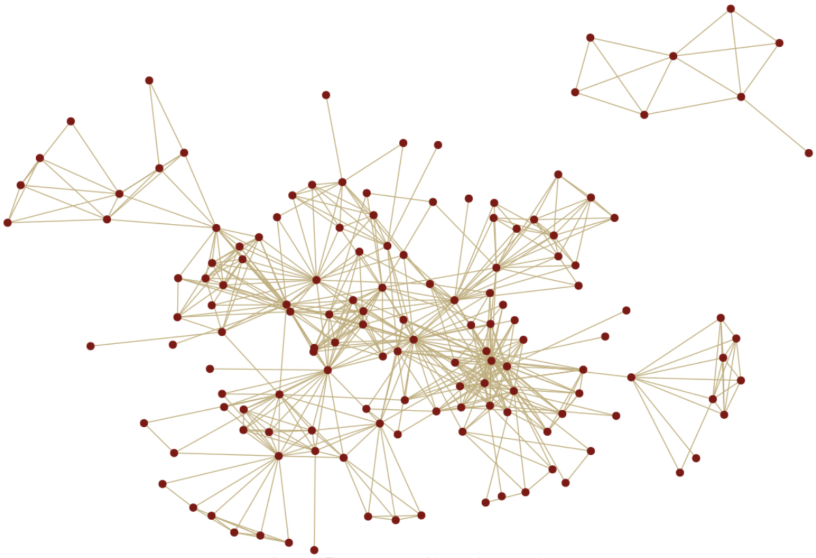
Epilogue: We have seen that the SVD can be used to efficiently denoise data. However, exact data C is unknown in practice and k_{min} must be determined by an educated guess.

For a more sophisticated denoising method based on SVD and how to guess k_{min} effectively see for example Epps and Krivitzky 'Singular value decomposition of noisy data: noise filtering' *Experiments in Fluids* 60 (2019) Article 126.

The setup code gives the following variables:

PageRank is coming: Determining the protagonist of Game of Thrones

Game of Thrones (GoT) is an HBO television series popularizing George R. R. Martin's *A Song of Ice and Fire* novel series. The TV series features one the largest-ever ensemble casts, and it is hard to determine the most important character. In this exercise, we will use the PageRank algorithm to gauge the relative importance of each character in Season 6 of *Game of Thrones*.



Game of Thrones season 6 interaction network

You are given an undirected, weighted adjacency matrix `A` with the character interaction data for Season 6 of the show; two characters interact whenever they speak consecutively, speak about each other, are spoken about together, or appear in a scene together. The weights of `A` demark how many interactions each pair of characters has. The weightedness of this matrix will be handled in the next step, which normalizes our data. First, **convert `A` to a Markov matrix `M`**. Recall that Markov matrices are normalized so that the sum of each column is 1. If a character has no interactions, we can model the character as randomly interaction with each other character with a probability of $1/N$ we can model it as randomly going to any other page with a probability of $1/N$ where N is the total number of characters included in `A`.

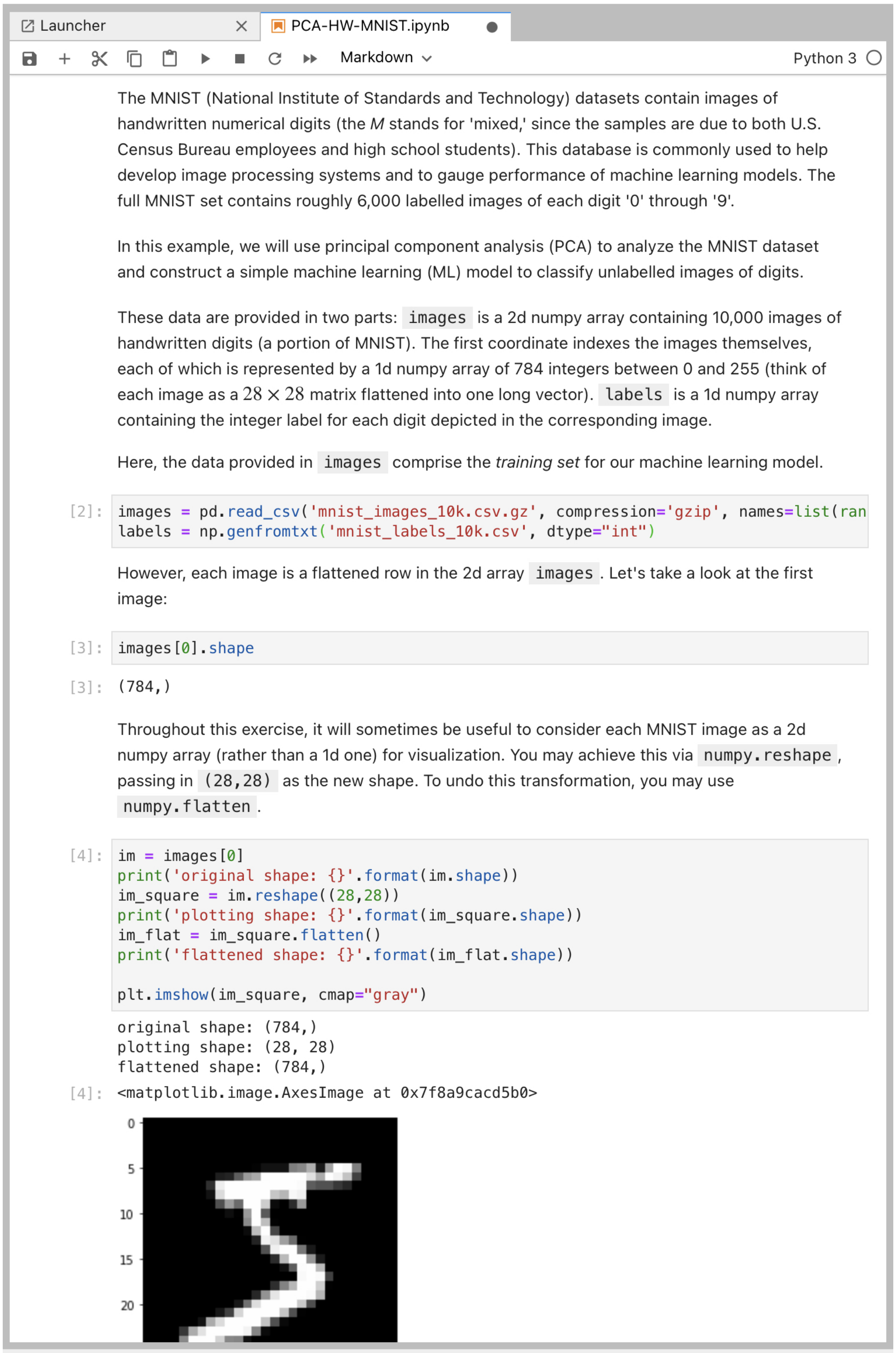
For a set value α between 0 and 1, the PageRank model for Markov matrix M is a matrix

$$G = \alpha M + (1-\alpha)\frac{1}{N}\mathbb{1},$$

where $\mathbb{1}$ is a matrix with the same shape as M containing 1 in every position. The number α is called the *damping factor*. You can consider this feature as a model of probable character interactions in some eventual reboot of the series.

Now, **construct the matrix `G` representing the PageRank model for *Game of Thrones***. Use $\alpha = 0.85$.

Then, **use the provided function `power_iteration` to get the steady-state eigenvector `x` of your *Game of Thrones* matrix `G`**. The steady state is the vector with the PageRank score for each character. You do not need to implement your own power iteration method; this is the same function that you created during the computational lab and has the following signature:



The MNIST (National Institute of Standards and Technology) datasets contain images of handwritten numerical digits (the *M* stands for 'mixed,' since the samples are due to both U.S. Census Bureau employees and high school students). This database is commonly used to help develop image processing systems and to gauge performance of machine learning models. The full MNIST set contains roughly 6,000 labelled images of each digit '0' through '9'.

In this example, we will use principal component analysis (PCA) to analyze the MNIST dataset and construct a simple machine learning (ML) model to classify unlabelled images of digits.

These data are provided in two parts: `images` is a 2d numpy array containing 10,000 images of handwritten digits (a portion of MNIST). The first coordinate indexes the images themselves, each of which is represented by a 1d numpy array of 784 integers between 0 and 255 (think of each image as a 28×28 matrix flattened into one long vector). `labels` is a 1d numpy array containing the integer label for each digit depicted in the corresponding image.

Here, the data provided in `images` comprise the *training set* for our machine learning model.

[2]:
```
images = pd.read_csv('mnist_images_10k.csv.gz', compression='gzip', names=list(ran
labels = np.genfromtxt('mnist_labels_10k.csv', dtype="int")
```

However, each image is a flattened row in the 2d array `images`. Let's take a look at the first image:

[3]:
```
images[0].shape
```

[3]:
```
(784,)
```

Throughout this exercise, it will sometimes be useful to consider each MNIST image as a 2d numpy array (rather than a 1d one) for visualization. You may achieve this via `numpy.reshape`, passing in `(28,28)` as the new shape. To undo this transformation, you may use `numpy.flatten`.

[4]:
```
im = images[0]
print('original shape: {}'.format(im.shape))
im_square = im.reshape((28,28))
print('plotting shape: {}'.format(im_square.shape))
im_flat = im_square.flatten()
print('flattened shape: {}'.format(im_flat.shape))

plt.imshow(im_square, cmap="gray")
```

```
original shape: (784,)
plotting shape: (28, 28)
flattened shape: (784,)
```

[4]:
```
<matplotlib.image.AxesImage at 0x7f8a9cacd5b0>
```