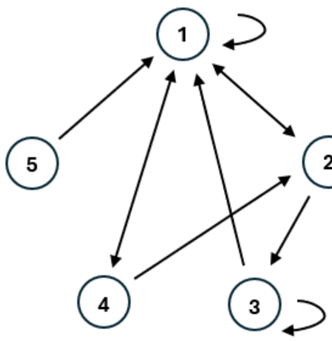
Homework 4: Coding portion
AMATH 301, Winter 2025
University of Washington
Due Friday, February 7, 2025, 11:59PM in Gradescope
20 points

1. In this problem we will find the largest (real) eigenvalue of a matrix, and its associated eigenvector, approximately using Power iteration. Create the following matrix as a `np.array` object called `Apower`.

$$A = \begin{bmatrix} 1/3 & 1/2 & 1/2 & 1/2 & 1 \\ 1/3 & 0 & 0 & 1/2 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Make sure not to round fractions when creating `Apower`.

This matrix could be a representation of the web (directed graph) below in Google PageRank, in which case the eigenvector associated with the largest eigenvalue is proportional to the fraction of the time a random surfer is located at each webpage (node).



Start at a guess of

$$x_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

and use Power iteration to repeatedly update x_n; call this new vector x_{n+1}. The step distance from x_n to x_{n+1} can be quantified by the 2-norm:

$$\text{step} = ||x_{n+1} - x_n||_2.$$

You can use `np.linalg.norm(y,2)` to calculate the 2-norm, where y is the vector whose norm you desire.

Stop the Power iteration as soon as the value of the step is below 10^{-5}, or after 100 iterations, whichever comes first. This x vector is the (approximate) eigenvector associated with the largest real eigenvalue in absolute value.

(a) (2 points) Record the number of iterations taken, as the variable `iterspower`. Start counting iterations at 1.

(b) (2 points) Save your eigenvector as an `np.array` object named `eigvecpower`. Make sure that the eigenvector has length $||x||_2 = 1$ and the first component of x is positive.

(c) (2 points) Find the associated eigenvalue using the formula:

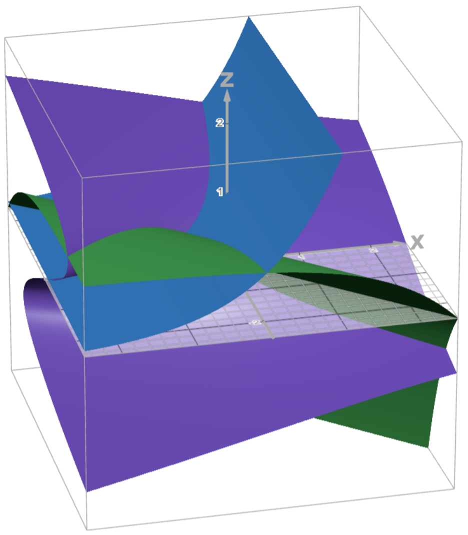
$$\lambda = \frac{x^T A x}{x^T x}.$$

Save that number as the variable `eigvalpower`.

2. Given the equations:

$$\begin{cases} f_1(x,y,z) &= z - e^x &= 0 \\ f_2(x,y,z) &= z - 1 + e^{x+y} &= 0 \\ f_3(x,y,z) &= y - xz - z^3 &= 0 \end{cases}$$

The value $[\ x \quad y \quad z\ ]^T$ where all three equations are satisfied is the point of intersection of three surfaces, which is shown below. You can find an interactive version here.



Define a Python function `f(v)` which takes as an argument a vector $v = [\ x \quad y \quad z\ ]^T$ as a `np.array` object, and outputs the vector

$$f(v) = \begin{bmatrix} f_1(x,y,z) \\ f_2(x,y,z) \\ f_3(x,y,z) \end{bmatrix}.$$

Then define a Python function `J(v)` which takes as an argument a vector $v = [\ x \quad y \quad z\ ]^T$ as a `np.array` object, and outputs the Jacobian matrix of $f(v)$:

$$J(v) = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial z} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial z} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial z} \end{bmatrix}.$$

Start with an initial guess $v_0 = [\ 0 \quad 0 \quad 0\ ]^T$.

(a) (2 points) Plug v_0 into $f(v)$ and save the resulting $f(v_0)$ in a `np.array` object called `fv0`. Plug v_0 into $J(v)$ and save the resulting $J(v_0)$ in a `np.array` object called `Jv0`.

(b) (4 points) Perform the Newton-Raphson method:

$$v_{n+1} = v_n + \Delta v_n$$

where

$$J(v_n)\Delta v_n = -f(v_n).$$

Use `np.linalg.solve` to solve the linear system for Δv_n at each iteration. Terminate when $||f(v)||_2 < 10^{-5}$. Record the final approximate solution v as a `np.array` object named `vnonlin`, and record the number of iterations as the number `numiternonlin`. Start counting the number of iterations at 0.

3. One important application of SVD is in image compression. We will work with the picture file `seattle.jpg` which looks like this:

It is 300 pixels in width and 200 pixels in height. The image was converted to grayscale and then to a `np.array` which is located in the file `imggray.npy`. Download that file to the same folder as your Python file. Then, import the matrix using:

```
imggray = np.load('imggray.npy')
```

`imggray` is a 200×300 matrix of values between 0 and 1, where 0 represents a white pixel and 1 represents a black pixel. Thus to store the full, uncompressed grayscale image, we need $200 \cdot 300 = 60000$ values between 0 and 1. Next we will investigate how to compress the image.

Use `np.linalg.svd` to find the singular value decomposition of `imggray`:

$$A = U\Sigma V^T$$

where A represents `imggray`; the columns of U are the left-singular vectors; the columns of V are the right-singular vectors; and Σ has diagonal entries which are the singular values $\sigma_1, \sigma_2, \cdots$, all of which are nonnegative and in decreasing order, and off-diagonals are zero. The sizes of the matrices are:

- A: 200×300
- U: 200×200
- Σ: 200×300 (note: `np.linalg.svd` returns a 1D, not 2D array– just the diagonal elements. `np.diag` can transform this into a matrix if needed.)
- V: 300×300 (note: `np.linalg.svd` returns V^T, if you wish to recover V, you need to transpose again)

The largest singular value, that is, σ_1 (counting starting at 1), and its associated left- and right-singular vectors contain the most "information" about the image. The second-largest value in σ_2 contains the second-most "information," and so on. Thus, a compression tactic would be to only keep some of the largest singular values, and ignore the rest.

(a) (2 points) Let u be the first column of U and let v be the first row of V^T (counting starting at 1). Construct as a `np.array` object the 200×300 matrix $A = u\sigma_1 v$. The function `np.outer` may be useful. Call this matrix `Arank1`. This will produce an image only using the largest singular value.

(b) (2 points) Let U_{10} be a matrix containing only the first 10 columns of U, so that U_{10} is size 200×10. Let $(V^T)_{10}$ be a matrix containing only the first 10 rows of V^T, so that $(V^T)_{10}$ is 10×300. Let Σ_{10} be a 10×10 diagonal matrix whose entries are the first 10 singular values. Construct as a `np.array` object the 200×300 matrix $A = U_{10}\Sigma_{10}(V^T)_{10}$. Call this matrix `Arank10`. This will produce an image only using the 10 largest singular values.

(c) (3 points) Rather than using a pre-specified number of singular values, like 10 as we did in part (b), maybe we want to include enough so that we have encompassed a certain amount of the total. Let r be the smallest positive integer such that

$$\sum_{k=1}^{r} \sigma_k \geq 0.8 \sum_{k=1}^{200} \sigma_k.$$

In other words, we wish to capture 80% of the total sum of all the σ_ks by only including the first r of them. Let $r_{svd} = r - 1$. Call the value of r_{svd} as `rsvd`.

Let U_{rsvd} be a matrix containing only the first r_{svd} columns of U, so that U_{rsvd} is size $200 \times r_{svd}$. Let $(V^T)_{rsvd}$ be a matrix containing only the first r_{svd} rows of V^T, so that $(V^T)_{rsvd}$ is $r_{svd} \times 300$. Let Σ_{rsvd} be a $r_{svd} \times r_{svd}$ diagonal matrix whose entries are the first r_{svd} singular values. Construct as a `np.array` object the 200×300 matrix $A = U_{rsvd}\Sigma_{rsvd}(V^T)_{rsvd}$. Call this matrix `Arankr`. This will produce an image only using the r_{svd} largest singular values.

(d) (1 point) Consider how much compression has happened in part (c). The original uncompressed grayscale image contained 60000 pixels, which was stored as 60000 values between 0 and 1. The number of numerical values that must be stored to create the `Arankr` matrix is the number of values in U_{rsvd}, $(V^T)_{rsvd}$, and the singular values $\sigma_1, \sigma_2, \cdots \sigma_{rsvd}$. Find that number, and divide by 60000 to get the fraction of the total values that need to be stored to generate `Arankr`. Call this value `compressionfraction`. For example, if `compressionfraction` were 0.9, that would mean that the file size is 90% as large as the uncompressed image file size.

Finally, include the lines below at the bottom of your code, which will display the `Arank1`, `Arank10`, and `Arankr` images, as well as the original image. You should see that the `Arankr` image is not perfect but decently clear, even though `compressionfraction` is significantly below 1.

```
fig, ax = plt.subplots(2,2)
ax[0,0].imshow(Arank1, cmap='gray')
ax[0,1].imshow(Arank10, cmap='gray')
ax[1,0].imshow(Arankr, cmap='gray')
ax[1,1].imshow(imggray, cmap='gray')
```

Optional note: if you want to try this out for fun with your own `jpg` image instead of the Seattle one, you can use:

```
from skimage.color import rgb2gray
from skimage import io
from skimage.transform import resize
imgcolor = io.imread('filename.jpg')
imggray = rgb2gray(imgcolor)
```

However, the Autograder is not able to use the `skimage` library, so make sure to remove any reference to that library before submitting your code to the Autograder.