Last updated: Monday 28th March, 2022 at 14:21.

Programming assignment #3: eigenvalues

Problem 1. Program the power method and inverse power method to compute the maximum and minimum eigenvalue/eigenvector pairs of the symmetric matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$.

To test your code, you can create a random eigenvalue decomposition with specified eigenvalues as follows:

```
Q = np.random.randn(n, n) # sample a matrix with iid N(0, 1) entries
Q = np.linalg.qr(Q)[0] # the Q factor in its QR decomposition is an
                       # orthogonal matrix sampled uniformly from
                       # the orthgonal group O(n)
Lam = np.diag(...) # set the diagonal matrix of eigenvalues manually
A = Q@Lam@Q.T # compute A from Q and Lam
```

This way you can fix the values of λ_1 and λ_2 (or $1/\lambda_n$ and $1/\lambda_{n-1}$) to control the convergence rate of the (inverse) power method.

Make a plot (or several plots) of the convergence of the (inverse) power method for several different eigenvalue ratios. Since you set the eigenvalues of your test matrix A yourself, you can easily compare with the true solution. Your plot(s) should show evidence that the convergence rate is largely controlled by, e.g., λ_2/λ_1.

(*Note*: in this problem, I would like you to look at *both* the power method and the inverse power method. For the inverse power method, use `np.linalg.lu` or `scipy.linalg.lu` instead of your own LU decomposition to compute the LU decomposition of A. You can then use the function `scipy.linalg.solve_triangular` to do the forward and backward solves involving your triangular factors.)

Problem 2. Incorporate shifts into your power method (i.e., apply the power method and inverse power method to the shifted matrix $\boldsymbol{A} - \lambda \boldsymbol{I}$). Come up with a strategy for choosing the shift λ to accelerate the convergence of the power method. Note: the shift can vary from iteration to iteration. In general, there can be many possible strategies—you must simply come up with some strategy that reduces the overall number of iterations. However, be careful with the inverse power method: if you change the shift, you must recompute the LU decomposition used to apply $(\boldsymbol{A} - \lambda \boldsymbol{I})^{-1}$ at each step.

Problem 3. Recall the matrix $\boldsymbol{A} \in \mathbb{R}^{(N-1)^2 \times (N-1)^2}$ from PHW #2. This matrix is *sparse*—that is, it has $O(N^2)$ rows and columns, but only $O(N^2)$ nonzero entries. Storing the zeros of $\boldsymbol{A}$ is extremely wasteful, since it requires us to store $O(N^4)$ entries instead of only $O(N^2)$.

Read about the *compressed sparse row* (CSR) sparse matrix format. SciPy provides a class implementing a sparse matrix using the CSR format: see `scipy.sparse.csr_matrix`.

Set up a version of $\boldsymbol{A}$ for $N = 64$ using `scipy.sparse.csr_matrix`. Next, read the documentation for the function `scipy.sparse.linalg.eigsh`. Use it to compute the $k = 16$

smallest *nonzero* eigenvalues and eigenvectors of $\boldsymbol{A}$ (*hint*: use the `sigma` keyword argument). Using `plt.subplot`, make a 4×4 grid of 2D plots of each of the eigenvectors, as well as a plot of the eigenvalues that you compute.

Problem 4a. The matrix $\boldsymbol{A}$ can be interpreted as a *graph*. Its rows and columns are naturally identified with the nodes in the original lattice, which we think of as the graph's *vertices*. There is an *edge* between a pair of vertices if the corresponding pair of nodes is adjacent in the lattice. Recall from lecture that $\boldsymbol{A}$ is a discretized Laplacian.

On the other hand, if we start with a graph $G = (V, E)$, where $N = |V|$ and $V = \{v_1, \ldots, v_N\}$ is the set of vertices and $E = \{(i, j) : v_i \text{ and } v_j \text{ are adjacent}\}$ is the set of edges, we can define the *graph Laplacian* $\boldsymbol{L} \in \mathbb{R}^{N \times N}$ by:

$$\boldsymbol{L}_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j, \\ -1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Note that the adjacent matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ for G is defined by:

$$\boldsymbol{A}_{ij} = \begin{bmatrix} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{bmatrix} \tag{2}$$

Download a graph from:

`https://sites.google.com/site/xiaomengsite/research/resources/graph-dataset`

or some other website online. Load it using Python and build a CSR version of $\boldsymbol{L}$ for your graph. Next, as in Problem 3, use `scipy.sparse.linalg.eigsh` (note that $\boldsymbol{L}$ should be symmetric!) to compute the first $k = 2$ smallest nonzero eigenvalue and eigenvector pairs of $\boldsymbol{L}$. Let $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ be these eigenvectors, and experiment with making plots of $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ to visualize the graph. Please be creative and feel free to experiment.

(*Hint*: the process of getting your graph into Python and building $\boldsymbol{L}$ may be annoying and require some data processing. Don't be a hero—use a relatively simple graph to make this easy. Between a few hundred vertices and a few thousand vertices is a good target.)

Problem 4b (bonus). Compute $\boldsymbol{u}_3$ and make a 3D visualization of your graph. I recommend starting with `mplot3d` for 3D plotting, but feel free to contact me for more recommendations, and feel free to experiment.

Problem 5 (bonus). Read about *bipartite graphs* online. An off-diagonal block of the adjaceny matrix (*not* the graph Laplacian matrix $\boldsymbol{L}$!) of your graph from Problem 4 is a bipartite graph. Pick such an off-diagonal block and denote it $\boldsymbol{A}_{I,J}$, where I and J are the row and column indices of the block ($I \cap J = \emptyset$).

Let $m = |I|$ and $n = |J|$ and $p = \min(m, n)$. Compute the singular value decomposition $\boldsymbol{A}_{I,J} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$, where $\boldsymbol{U} \in \mathbb{R}^{m \times p}$, $\boldsymbol{V} \in \mathbb{R}^{n \times p}$, and $\Sigma \in \mathbb{R}^{p \times p}$. Do this using `np.linalg.svd`

with `full_matrices=False`. Make a `semilogy` plot of the singular values $\sigma_1, \ldots, \sigma_p$ and observe how rapidly they decay.

Next, compute a *truncated* SVD by retaining only the first $k < p$ terms of the singular value decompositions, recalling that we can write:

$$\boldsymbol{A}_{I,J} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top = \sum_{i=1}^{p} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top. \tag{3}$$

Define:

$$\boldsymbol{A}_{I,J}^{(k)} = \sum_{i=1}^{k} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top. \tag{4}$$

Since $\boldsymbol{A}_{I,J}$ is sparse, the singular matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ should be sparse. Indeed, $\boldsymbol{A}^{(k)}$ should be sparse, as well, and can be interpreted as an approximation to the original graph represented by the bipartite adjacency matrix $\boldsymbol{A}_{I,J}$.

For several different choices of I and J, vary k and make visualizations of the bipartite graphs corresponding to $\boldsymbol{A}_{I,J}^{(k)}$. You do not have to use the visualization method you came up with for Problem 4, but you are free to do so. Write up your observations in a short paragraph or two and include it as a separate PDF file with your submission.