ECON526: Quantitative Economics with Data Science Applications

Applications of Linear Algebra

Jesse Perla

Table of contents i

Overview

Motivation and Materials

- In this lecture, we will cover some applications of the tools we developed in the previous lecture
- The goal is to build some useful tools and to sharpen your intuition on linear algebra and eigenvalues/eigenvectors
- Some additional material and references
    - QuantEcon Python
    - QuantEcon DataScience
    - A First Course in Quantitative Economics with Python

Packages

This section uses the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
from numpy.linalg import cond, matrix_rank, norm
from scipy.linalg import inv, solve, det, eig, lu, eigvals
from scipy.linalg import solve_triangular, eigvalsh, cholesky
from sklearn.decomposition import PCA
```

Difference Equations

Linear Difference Equations as Iterative Maps

- Consider $A : \mathbb{R}^N \to \mathbb{R}^N$ as the linear map for the state $x_t \in \mathbb{R}^N$
- An example of a linear difference equation is

$$x_{t+1} = Ax_t$$

where

$$A \equiv \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.8 \end{bmatrix}$$

```
A = np.array([[0.9, 0.1], [0.5, 0.8]])
x_0 = np.array([1, 1])
x_1 = A @ x_0
print(f"x_1 = {x_1}, x_2 = {A @ x_1}")
```

```
x_1 = [1.  1.3], x_2 = [1.03 1.54]
```

Iterating with rho(A) > 1

Iterate $x_{t+1} = Ax_t$ from x_0 for $t = 100$

```
x_0 = np.array([1, 1])
t = 200
print(f"rho(A) = {np.max(np.abs( eigvals(A)))}")
print(f"x_{t} = {np.linalg.matrix_power(A, t) @ x_0}")
```

```
rho(A) = 1.079128784747792
x_200 = [3406689.32410673 6102361.18640516]
```

- Diverges to $x_\infty = \begin{bmatrix} \infty \\ \infty \end{bmatrix}$
- $\rho = 1 + 0.079$ says in worst case, expands by 7.9% on each iteration

Iterating with rho(A) < 1

```
A = np.array([[0.6, 0.1], [0.5, 0.8]])
print(f"rho(A) = {np.max(np.abs( eigvals(A)))}")
print(f"x_{t} = {np.linalg.matrix_power(A, t) @ x_0}")
```

```
rho(A) = 0.9449489742783178
x_200 = [6.03450418e-06 2.08159603e-05]
```

- Converges to $x_\infty = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Iterating with rho(A) = 1

- To make a matrix that has $\rho(A) = 1$ reverse eigenvalue decomposition!
- Leave previous eigenvectors, just change eigenvalues to rig $\rho(A)$

```
Q = np.array([[-0.85065081, -0.52573111], [0.52573111, -0.85065081]])
print(f"orthogonal if dot(x_1, x_2) approx 0?: {np.dot(Q[:,0], Q[:,1])}")
Lambda = [1.0, 0.8]  # choosing eigenvalue so max_n|lambda_n| = 1
A = Q @ np.diag(Lambda) @ inv(Q)  # not Q.T unless symmetric
print(f"rho(A) = {np.max(np.abs( eigvals(A)))}")
print(f"x_{t} = {np.linalg.matrix_power(A, t) @ x_0}")
```

```
orthogonal if dot(x_1, x_2) approx 0?: 0.0
rho(A) = 1.0
x_200 = [ 0.27639321 -0.17082039]
```

Unemployment Dynamics

Dynamics of Employment without Population Growth

- Consider an economy where in a given year $\alpha = 5\%$ of employed workers lose job and $\phi = 10\%$ of unemployed workers find a job
- We start with $E_0 = 900,000$ employed workers, $U_0 = 100,000$ unemployed workers, and no birth or death. Dynamics for the year:

$$E_{t+1} = (1-\alpha)E_t + \phi U_t$$
$$U_{t+1} = \alpha E_t + (1-\phi)U_t$$

- Can write this as a matrix equation

$$\underbrace{\begin{bmatrix} E_{t+1} \\ U_{t+1} \end{bmatrix}}_{X_{t+1}} = \underbrace{\begin{bmatrix} 1-\alpha & \phi \\ \alpha & 1-\phi \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} E_t \\ U_t \end{bmatrix}}_{X_t}$$

Simulating

Simulate by iterating $X_{t+1} = AX_t$ from X_0 until $T = 100$

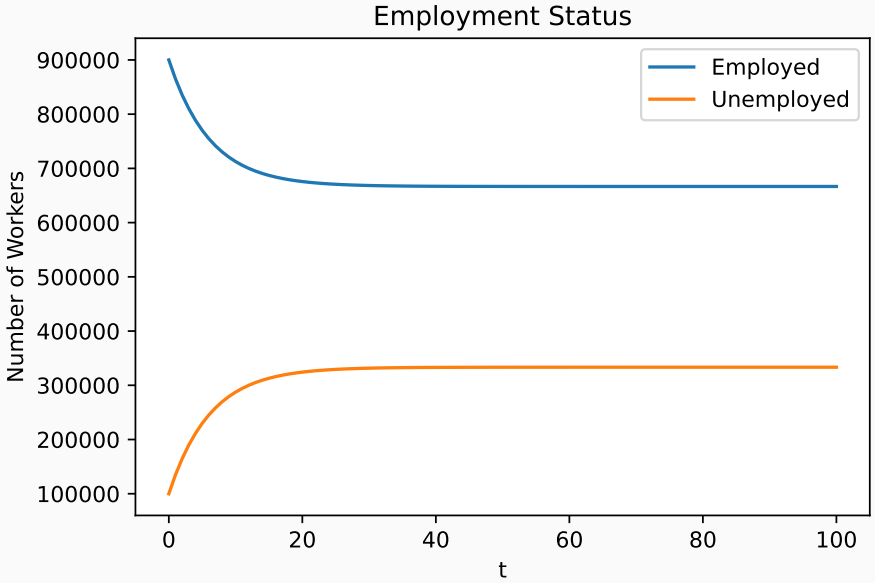
```
def simulate(A, X_0, T):
    X = np.zeros((T+1, 2))
    X[0,:] = X_0
    for t in range(T):
        X[t+1,:] = A @ X[t,:]
    return X
X_0 = np.array([900000, 100000])
A = np.array([[0.95, 0.1], [0.05, 0.9]])
T = 100
X = simulate(A, X_0, T)
print(f"X_{T} = {X[T,:]}")
```

```
X_100 = [666666.6870779  333333.31292209]
```

Plotting Code

```
fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(range(T+1), X, label=["Employed", "Unemployed"])
ax.set(xlabel="t", ylabel="Number of Workers", title="Employment Status")
ax.legend()
plt.show()
```

Dynamics of Unemployment



Convergence to a Longrun Distribution

- Find X_∞ by iterating $X_{t+1} = AX_t$ many times from a X_0?
  - Check if it has converged with $X_\infty \approx AX_\infty$
  - Is X_∞ the same from any X_0? Will discuss "ergodicity" later
- Alternatively, note that this expression is the same as

$$1 \times \bar{X} = A\bar{X}$$

  - i.e, a $\lambda = 1$ with $\bar{X}$ is the corresponding eigenvector of A!
  - Is $\lambda = 1$ always an eigenvalue? (yes if all $\sum_{n=1}^{N} A_{ni} = 1$ for all i)
  - Does $\bar{X} = X_\infty$? Maybe?
  - Multiple eigenvalues with $\lambda = 1 \implies$ multiple $\bar{X}$

Using the First Eigenvector for the Steady State

```
Lambda, Q = eig(A)
print(f"real eigenvalues = {np.real(Lambda)}")
print(f"eigenvectors are column-by-column in Q =\n{Q}")
print(f"first eigenvalue = 1? {np.isclose(Lambda[0], 1.0)}")
X_bar = Q[:,0] / np.sum(Q[:,0]) * np.sum(X_0)
print(f"X_bar = {X_bar}\nX_{T} = {X[T,:]}")
```

```
real eigenvalues = [1.   0.85]
eigenvectors are column-by-column in Q =
[[ 0.89442719 -0.70710678]
 [ 0.4472136   0.70710678]]
first eigenvalue = 1? True
X_bar = [666666.66666667 333333.33333333]
X_100 = [666666.6870779  333333.31292209]
```

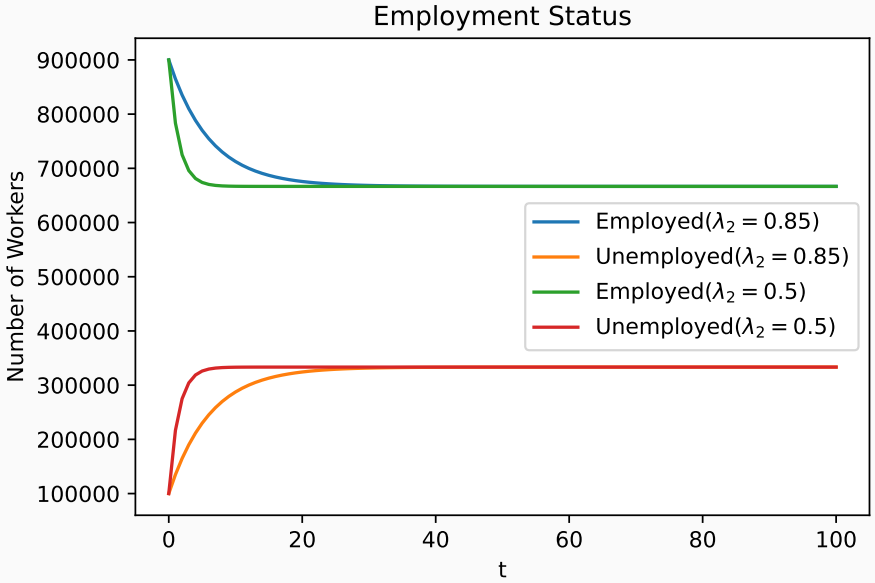
Using the Second Eigenvalue for the Convergence Speed

- The second largest ($\lambda_2 < 1$) provides information on the speed of convergence
  - Closer to 0 means faster convergence
  - Closer to 1 means slower convergence
- Create a new matrix with the same steady state, different speed

```
Lambda_fast = np.array([1.0, 0.5])
A_fast = Q @ np.diag(Lambda_fast) @ inv(Q) # same eigenvectors
X_fast = simulate(A_fast, X_0, T)
print(f"X_{T} = {X_fast[T,:]}")
```

```
X_100 = [666666.66666667 333333.33333333]
```

Dynamics of Unemployment For Difference Convergence Speeds



Understanding Latent Variables

Principle Components and Factor Analysis

- Another application of eigenvalues and eigenvectors is dimension reduction
- Principle Components Analysis (PCA) is a technique related to Singular Value Decomposition (SVD)
- Given a matrix $X \in \mathbb{R}^{N \times M}$, we can find a lower-dimensional representation $Z \in \mathbb{R}^{N \times L}$ for $L < M$ that captures the most variation in X
- The columns of Z are called the principle components of X
- The goal is to invert the X data to find the Z—and provide a mapping to reduce the dimensionality for future data
- This is a linear dimension reduction technique. Many ML algorithms have similar ideas but are non-linear
- See QuantEcon Singular Value Decomposition (SVD) Notes for more, and references to more applications

Decomposing the Data

PCA typically uses SVD in practice, but with our eigenvector decomposition

$$XX^T = Q\Lambda Q^T = Q\sigma\sigma^T Q^T, \quad \text{defining the singular values } \Lambda \equiv \sigma\sigma^T$$

Hence, denoting the nth column of Q as Q_n, we have

$$X = Q\sigma = Q_1\sigma_1 + Q_2\sigma_2 + ... + Q_M\sigma_M$$

Dimension Reduction

- Assume we sorted so $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_M$. Frequently $\sigma_1 \gg \sigma_M$
- For many problems, the σ_m decay quickly, so we can approximate X with fewer terms by truncating the sum at $L < M$. Roughly,

$$X \approx Q_1\sigma_1 + Q_2\sigma_2 + ... + Q_L\sigma_L$$

- The eigenvector decomposition can find the orthogonal directions of the data that capture the most variation
  - Can prove it is the solution to the optimization problem to explain the most variation in the data with the lowest dimensionality
- This is useful even if it is not necessary to reduce the dimensionality of the data
  - Many high-dimensional data sources are low-dimensional in the suitable space.
  - This is especially true when models allow for nonlinear transformations (e.g., neural networks, autoencoders, etc.)

Creating a Dataset with Latent Factors

Create a dataset with 2 latent factors, the first dominating

```
N = 50 # number of observations
L, M = 2, 3 # number of latent and observed factors
Z = np.random.randn(N, L) # latent factors
F = np.array([[1.0, 0.05], # X_1 = Z_1 + 0.05 Z_2
              [2.0, 0.0], # X_2 = 2 Z_1
              [3.0, 0.1]]) # X_3 = 3 Z_1 + 0.1 Z_2
X = Z @ F.T + 0.1 * np.random.randn(N, M) # added noise
```

PCA without any Dimension Reduction

- See QuantEcon SVD for coding yourself. We will use the sklearn package
- The explained variance is the fraction of the variance explained by each factor

```
pca = PCA(n_components=3)
pca.fit(X)
with np.printoptions(precision=4, suppress=True, threshold=5):
  print(f"Singular Values (sqrt eigenvalues):\n{pca.singular_values_}")
  print(f"Explained Variance (ordered):\n{pca.explained_variance_ratio_}")
```

```
Singular Values (sqrt eigenvalues):
[29.942   0.8313  0.5776]
Explained Variance (ordered):
[0.9989 0.0008 0.0004]
```

```
pca = PCA(n_components=2) # one less, and correctly specified
Z_hat = pca.fit_transform(X) # transformed by dropping last factor
# Scale and sign may not match since indeterminate, but check correlation
print(f"Correlation of Z_1 to Z_hat_1 = {np.corrcoef(Z.T, Z_hat.T)[0,2]}")
print(f"Correlation of Z_2 to Z_hat_2 = {np.corrcoef(Z.T, Z_hat.T)[1,3]}")
```

```
Correlation of Z_1 to Z_hat_1 = -0.9996092479665427
Correlation of Z_2 to Z_hat_2 = 0.45066371339252087
```

Interpreting the Results

- The first factor in the decomposition is nearly perfectly (positive or negatively) correlated with the more important latent factor
  - The sign could have gone either way. The key is the shared information
  - How could you have known the sign is indeterminate?
- The 2nd factor has a good but not great correlation with the 2nd latent. Why?
- The variance decomposition that gave a 3rd factor with non-zero variance
  - In our process, there are only 2 latent variables. Why didn't it figure it out?
- How could you have changed the DGP to make this **less** successful?

Present Discounted Values

PDV

Matrix Conditioning and Stability

Matrix Conditioning

- Poorly-conditioned matrices can lead to inaccurate or wrong solutions
- Tends to happen when matrices are close to singular, or when they have very different scales - so there will be times when you need to rescale your problems

```
eps = 1e-7
A = np.array([[1, 1], [1 + eps, 1]])
print(f"A =\n{A}")
print(f"A^{-1} =\n{inv(A)}")
```

```
A =
[[1.        1.       ]
 [1.0000001 1.       ]]
A^-1 =
[[-9999999.99336215  9999999.99336215]
 [10000000.99336215 -9999999.99336215]]
```

Condition Numbers of Matrices

- $\det(A) \approx 0$ may say it is "almost" singular, but it is not scale-invariant
- $cond(A) \equiv ||A|| \cdot ||A^{-1}||$ where $|| \cdot ||$ is the matrix norm - expensive to calculate in practice. Connected to eigenvalues $cond(A) = |\frac{\lambda_{max}}{\lambda_{min}}|$
- Gives scaleless measure of problems that are prone to numerical issues
- Intuition: if $cond(A) = K$, then $b \to b + \nabla b$ change in b amplifies to a $K\nabla b$ error when solving $Ax = b$.
- See Matlab Docs on inv for example where `inv` is a bad idea due to poor conditioning

```
print(f"condition(I) = {cond(np.eye(2))}")
print(f"condition(A) = {cond(A)}, condition(A^(-1)) = {cond(inv(A))}")
```

```
condition(I) = 1.0
condition(A) = 40000001.962777555, condition(A^(-1)) = 40000002.02779216
```

Example with Interpolation

- Consider fitting data $x \in \mathbb{R}^{N+1}$ and $y \in \mathbb{R}^{N+1}$ with an N-degree polynomial
- That is, find $c \in \mathbb{R}^{N+1}$ such that

$$\begin{aligned} c_0 + c_1 x_1 + c_2 x_1^2 + ... + c_N x_1^N &= y_1 \\ ... &= ... \\ c_0 + c_1 x_N + c_2 x_N^2 + ... + c_N x_N^N &= y_N \end{aligned}$$

- Which we can then use as $P(x) = \sum_{n=0}^{N} c_n x^n$ to interpolate between the points

Writing as a Linear System

- Define a matrix of all of the powers of the x values

$$A \equiv \begin{bmatrix} 1 & x_0 & x_0^2 & ... & x_0^N \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 & ... & x_N^N \end{bmatrix}$$

- Then solve for c as the solution to,

$$Ac = y$$

- Which we can solve using our tools. As long as x_n are unique, it is A is invertible
- Lets look at the numerical error here from the interpolation using the inf-norm, i.e. $||x||_\infty = \max_n |x_n|$

Solving an Example

```
N = 5
x = np.linspace(0.0, 10.0, N + 1)
y = np.exp(x)  # example function to interpolate
A = np.array([[x_i**n for n in range(N + 1)] for x_i in x])  # or np.vander
c = solve(A, y)
c_inv = inv(A) @ y
print(f"error = {norm(A @ c - y, np.inf)}, \
error using inv(A) = {norm(A @ c_inv - y, np.inf)}")
print(f"cond(A) = {cond(A)}")
```

```
error = 1.574562702444382e-11, error using inv(A) = 1.1932570487260818e-09
cond(A) = 564652.3214000753
```

Things Getting Poorly Conditioned Quickly

```
N = 10
x = np.linspace(0.0, 10.0, N + 1)
y = np.exp(x)  # example function to interpolate
A = np.array([[x_i**n for n in range(N + 1)] for x_i in x])  # or np.vander
c = solve(A, y)
c_inv = inv(A) @ y # Solving with inv(A) instead of solve(A, y)
print(f"error = {norm(A @ c - y, np.inf)}, \
error using inv(A) = {norm(A @ c_inv - y, np.inf)}")
print(f"cond(A) = {cond(A)}")
```

```
error = 5.334186425898224e-10, error using inv(A) = 6.22717197984457e-06
cond(A) = 4462824600234.486
```

Matrix Inverses Fail Completely for $N = 20$

```
N = 20
x = np.linspace(0.0, 10.0, N + 1)
y = np.exp(x)  # example function to interpolate
A = np.array([[x_i**n for n in range(N + 1)] for x_i in x])  # or np.vander
c = solve(A, y)
c_inv = inv(A) @ y # Solving with inv(A) instead of solve(A, y)
print(f"error = {norm(A @ c - y, np.inf)}, \
error using inv(A) = {norm(A @ c_inv - y, np.inf)}")
print(f"cond(A) = {cond(A):.4g}")
```

```
error = 6.784830475226045e-10, error using inv(A) = 31732.823760853855
cond(A) = 1.697e+24
```

Moral of this Story

- Use `solve`, which is faster and can often solve ill-conditioned problems. Rarely `inv`
- Check conditioning of matrices when doing numerical work, as it is a good indicator of potential problems and collinearity
- For approximation, never use a monomial basis for polynomials
  - Chebyshev polynomials which are designed to be as orthogonal as possible

```
N = 40
x = np.linspace(-1, 1, N+1)  # Or any other range of x values
A = np.array([[np.polynomial.Chebyshev.basis(n)(x_i) for n in range(N+1)] for
A_monomial = np.array([[x_i**n for n in range(N + 1)] for x_i in x])  # or np
print(f"cond(A) = {cond(A):.4g}, cond(A_monimial) = {cond(A_monomial):.4g}")
```

```
cond(A) = 3.64e+09, cond(A_monimial) = 2.926e+18
```