PROJECT 2: EIGENVALUES OF TRIDIAGONAL MATRICES

AYODEJI ODETOLA, JAMES MILLER DEVON REED, JIAWANG LIU

Date: April, 13 2020.

Part 1: Tridiagonal Systems

Solving tridiagonal systems of linear equations. In this part of the project we have written a procedure in `Python` to solve the $n \times n$ system $\mathbf{Ax} = \mathbf{b}$ where the sytem is in the following form:

$$\begin{bmatrix} d_1 & c_1 & & & & & \\ a_1 & d_2 & c_2 & & & \text{\Large 0} & \\ & a_2 & d_3 & c_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & \text{\Large 0} & & & a_{n-2} & d_{n-1} & c_{n-1} \\ & & & & & a_{n-1} & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_i \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

To solve such a system, naive Gaussian elimination is used. Our first step begins by subtracting a_1/d_1 times row 1 from row 2, thus creating a 0 in the a_1 position. Then, this process is repeated using the new row 2 as the pivot row. Thus, here is how the b_i's and d_i's are updated:

$$\begin{cases} d_2 \leftarrow d_2 - \left(\frac{a_1}{d_1}\right) c_1 \\ b_2 \leftarrow b_2 - \left(\frac{a_1}{d_1}\right) b_1 \end{cases}$$

In general, for all rows we obtain

$$\begin{cases} d_i \leftarrow d_i - \left(\frac{a_{i-1}}{d_{i-1}}\right) c_{i-1} \\ b_i \leftarrow b_i - \left(\frac{a_{i-1}}{d_{i-1}}\right) b_{i-1} \qquad (2 \le i \le n) \end{cases}$$

At the end of this phase, this is what the form of the system looks like:

$$\begin{bmatrix} d_1 & c_1 & & & & & \\ & d_2 & c_2 & & & & \\ & & d_3 & c_3 & & & \\ & & & \ddots & \ddots & & \\ & & & & \ddots & \ddots & \\ & & & & & d_{n-1} & c_{n-1} \\ & & & & & & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_i \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

Next, we perform the back substitution phase which solves for x_n, x_{n-1}, $\ldots$, x_1 which takes the form

$$x_i \leftarrow \frac{1}{d_i}(b_i - c_i x_{i+1}) \qquad (i = n-1, n-1, \ldots, 1)$$

Lastly, we programmed this algorithm in `Python` and our solution can be seen below[1]:

```
def triDyGuy(a, b, c, d):
  '''Funtion that takes a tridiagonal matrix A with diaganoals a, b, and c
  where a is the sub diagonal, b is the true diagonal, and c is the super
  diagonal. Vector d is the constant vector. These are used to compute and
  return the solution vector x in from the form Ax = b'''
  n = len(d) # number of equations or rank of matrix
  a, b = map(np.array, (a.astype(float), b.astype(float))) # prepare arrays
  c, d = map(np.array, (c.astype(float), d.astype(float))) # prepare arrays
  for i in range(1, n):
    xi = a[i-1] / b[i-1]
    b[i] = b[i] - xi*c[i-1]
    d[i] = d[i] - xi*d[i-1]
  x = b
  x[-1] = d[-1] / b[-1]
  for i in range(n-2, -1, -1):
    x[i] = (d[i] - c[i]*x[i+1])/ b[i]
  return x
```

Deriving a recursive formula for the characteristic polynomial. In order to find the eigenvalues of a matrix, we look at the roots of the characteristic polynomial which can be obtained recursively. Let $\varphi_n(\mu) = \det(J - \mu I)$ with I being the identity matrix and J being a Hermitian matrix of the form

$$J = \begin{bmatrix} \delta_1 & \bar{\gamma}_2 & & & & \\ \gamma_2 & \delta_2 & \bar{\gamma}_3 & & 0 & \\ & \gamma_3 & \delta_3 & \bar{\gamma}_4 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \gamma_{n-1} & \delta_{n-1} & \bar{\gamma}_n \\ & & & & \gamma_n & \delta_n \end{bmatrix}, \quad \delta_j = \bar{\delta}_j.$$

Thus we have,

$$\varphi_n(\mu) = \det\left(\begin{bmatrix} (\delta_1 - \mu) & \bar{\gamma}_2 & & & & \\ \gamma_2 & (\delta_2 - \mu) & \bar{\gamma}_3 & & & \\ & \gamma_3 & (\delta_3 - \mu) & \bar{\gamma}_4 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \gamma_{n-1} & (\delta_{n-1} - \mu) & \bar{\gamma}_n \\ & & & & \gamma_n & (\delta_n - \mu) \end{bmatrix}\right)$$

[1]The b and d matrices were swapped in our Python code thus, creating a slight difference from our steps above

Hence, for n =1 that $\varphi_1(\mu) = (\delta_1 - \mu)$ trivially [2]. For $n = 2$ we have

$$\varphi_2(\mu) = \det\left(\begin{bmatrix}(\delta_1 - \mu) & \bar{\gamma}_2 \\ \gamma_2 & (\delta_2 - \mu)\end{bmatrix}\right) = (\delta_2 - \mu)(\delta_1 - \mu) - |\gamma_2^2|$$

We then proceed by using expansion by minors along the last row. We let

$$\varphi_{n-1}(\mu) = \det\left(\begin{bmatrix}(\delta_1 - \mu) & \bar{\gamma}_2 & & & \\ \gamma_2 & (\delta_2 - \mu) & \bar{\gamma}_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \gamma_{n-2} & (\delta_{n-2} - \mu) & \bar{\gamma}_{n-1} \\ & & & \gamma_{n-1} & (\delta_{n-1} - \mu)\end{bmatrix}\right)$$

and

$$\varphi_{n-2}(\mu) = \det\left(\begin{bmatrix}(\delta_1 - \mu) & \bar{\gamma}_2 & & & \\ \gamma_2 & (\delta_2 - \mu) & \bar{\gamma}_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \gamma_{n-1} & (\delta_{n-1} - \mu) & \bar{\gamma}_{n-2} \\ & & & \gamma_{n-2} & (\delta_{n-2} - \mu)\end{bmatrix}\right)$$

It then follows that

$$\varphi_n(\mu) = \begin{cases}(\delta_1 - \mu),\ n = 1 \\ (\delta_2 - \mu)(\delta_1 - \mu) - |\gamma_2^2|,\ n = 2 \\ (\delta_n - \mu)\cdot\varphi_{n-1}(\mu) - |\gamma_n^2|\cdot\varphi_{n-2}(\mu),\ n > 2\end{cases}$$

Properties of Eigenvalues. We know that in order for λ to be an eigenvalue of matrix J, it must satisfy the following property,

$$\lambda\vec{x} = J\vec{x}$$

Thus, using the properties of norms we then get that

$$||\lambda\vec{x}||_\infty = |\lambda|\cdot||\vec{x}||_\infty$$

Hence, $|\lambda|\cdot||\vec{x}||_\infty = ||J\vec{x}||_\infty \leq ||J||_\infty\cdot||\vec{x}||_\infty \implies \lambda \leq |\lambda| \leq ||J||_\infty$. Therefore we get that

$$\lambda \leq \max_{1\leq k\leq n}\left\{\sum_{l=1}^{n}|J_{kl}|\right\}$$

We then observe that $\{\sum_{l=1}^{n}|J_{kl}|\} = \{|\gamma_k| + |\delta_k| + |\bar{\gamma}_{k+1}|\}$ since J is a tridiagonal matrix. Therefore we get the final inequality

$$\lambda \leq \max_{1\leq k\leq n}\{|\gamma_k| + |\delta_k| + |\bar{\gamma}_{k+1}|\}$$

[2]the proof is left as an excersice to Dr. Juan Gil

Part 2: Earthquake induced vibrations

Finding the Eigenvalues of A. For the study of earthquake induced vibrations on multistory buildings, let us assume that the free transverse oscillations satisfy a system of second order differential equations of the form

$$m\mathbf{X}'' = kA\mathbf{X}, \tag{*}$$

where A is an $n \times n$-matrix (for n floors), m is the mass of each floor, k denotes the stiffness of the columns supporting the floor, and $\mathbf{X}$ is the vector $(x_1(t), \ldots, x_n(t))$ describing the deformation of the columns.

On the i-th floor ($i \neq 1, n$) the acting forces are

$$k(x_i - x_{i-1}) - k(x_{i+1} - x_i) = k(-x_{i-1} + 2x_i - x_{i+1})$$

giving rise to the differential equation

$$mx_i'' = k(x_{i-1} - 2x_i + x_{i+1}).$$

In the first and last floors the equations are $mx_1'' = k(-2x_1 + x_2)$ and $mx_n'' = k(x_n - x_{n-1})$, respectively. For a five-story building the matrix A from (*) becomes

$$A = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix}.$$

To find the Eigenvalues of A we employ the recursive formula from part 1 using `Python`[3] to get the following characteristic polynomial:

$$-\left(x^5 + 9x^4 + 28x^3 + 35x^2 + 15x + 1\right)$$

Then, using our newton's method program[4] and the previously obtained characteristic polynomial, the 5 Eigenvalues λ_1, λ_2, $\ldots$, λ_5 were approximated to be the following real numbers:

$$\lambda_i \approx \begin{bmatrix} -3.682507065662369 \\ -2.8308300260037758 \\ -1.715370323453437 \\ -0.6902785321094295 \\ -0.08101405277100507 \end{bmatrix}$$

[3] See the appendix for our `getCharPoly` function's `Python` code which was used to efficiently obtain the characteristic polynomial

[4] See the appendix for our `superNewton` function's `Python` code which was used to find the roots of the characteristic polynomial to a precision of 1×10^{-10}

Finding the Eigenvectors of A. In order to find the Eigenvectors of the matrix A, we employed an existing Numpy method[5] in `Python`. The Eigenvectors $\vec{\mathbf{V}}$ were then approximated as follows, where the Eigenvalue λ_i corresponds to the Eigenvector $\vec{\mathbf{V}}_i$

$$\vec{\mathbf{V}}_i = \left[\begin{bmatrix} -0.32601868 \\ 0.54852873 \\ -0.59688479 \\ 0.45573414 \\ -0.16989112 \end{bmatrix} \begin{bmatrix} -0.54852873 \\ 0.45573414 \\ 0.16989112 \\ -0.59688479 \\ 0.32601868 \end{bmatrix} \begin{bmatrix} 0.59688479 \\ 0.16989112 \\ -0.54852873 \\ -0.32601868 \\ 0.45573414 \end{bmatrix} \begin{bmatrix} -0.45573414 \\ -0.59688479 \\ -0.32601868 \\ 0.16989112 \\ 0.54852873 \end{bmatrix} \begin{bmatrix} 0.16989112 \\ 0.32601868 \\ 0.45573414 \\ 0.54852873 \\ 0.59688479 \end{bmatrix}\right]$$

Verification of solution to (*). Let $\vec{v}$ be an Eigenvector of A with Eigenvalue λ, with $\omega \in \mathbb{C}$ and $\omega = -\lambda \cdot \frac{k}{m}$. (i.e.) we have

$$X(t) = \begin{bmatrix} x_1(t) \\ x_2(2) \\ \vdots \\ x_n(t) \end{bmatrix} = \alpha \cos(\omega t)\vec{v}$$

Thus we get

$$\begin{cases} X(t) = \alpha \cos(\omega t)\vec{v} \\ X'(t) = -\alpha\omega \sin(\omega t)\vec{v} \\ X"(t) = -\alpha\omega^2 \cos(\omega t)\vec{v} \end{cases}$$

$\implies X"(t) = \alpha\frac{\lambda \cdot k}{m} \cos(\omega t)\vec{v}$. Thus from (*) we get

$$\begin{aligned} mX"(t) &= \lambda\alpha k \cdot \cos(\omega t)\vec{v} \\ &= k\lambda\vec{v}\alpha \cdot \cos(\omega t) \end{aligned}$$

Since $\vec{v}$ is an eigen vector, it then follows tha $mX"(t) = k \cdot A\alpha \cos(\omega t)\vec{v} \implies mX"(t) = kAX$ (since $X(t) = \alpha \cos(\omega t)\vec{v}$). Therefore verifying that $X(t)$ is a solution to (*).

Solving for the Solution Vector. Let $m = 1250$, $k = 10000$, and $\alpha = 0.075$. To calculate the solution matrix, first we had to find the value of ω. To do this, we used the following equation:

$$\omega^2 = -\lambda(\frac{k}{m})$$

Plugging in the smallest Eigenvalue for λ, which we found to be -0.0810140, we determined ω to be equal to 0.8050542839. Then using the corresponding Eigenvector as v in the equation for X(t), which we know to be $X(t) =$

[5]The numpy function is `numpy.linalg.eig()` which returns both the Eigenvalues and vectors of a matrix

$\alpha cos(\omega t)v$, we calculated the solution vector X(t). The solution vector that we found was:

$$\begin{bmatrix} 0.012741834\cos(\omega t) \\ 0.024451401\cos(\omega t) \\ 0.0341800605\cos(\omega t) \\ 0.0411396548\cos(\omega t) \\ 0.0447663593\cos(\omega t) \end{bmatrix}$$

Plotting Parametric curves. Here we plot the curves $x_k(t) + k$ (for $k = 1, 2, ..5$). The curves describe the deformation of the support columns during the earthquake over time.

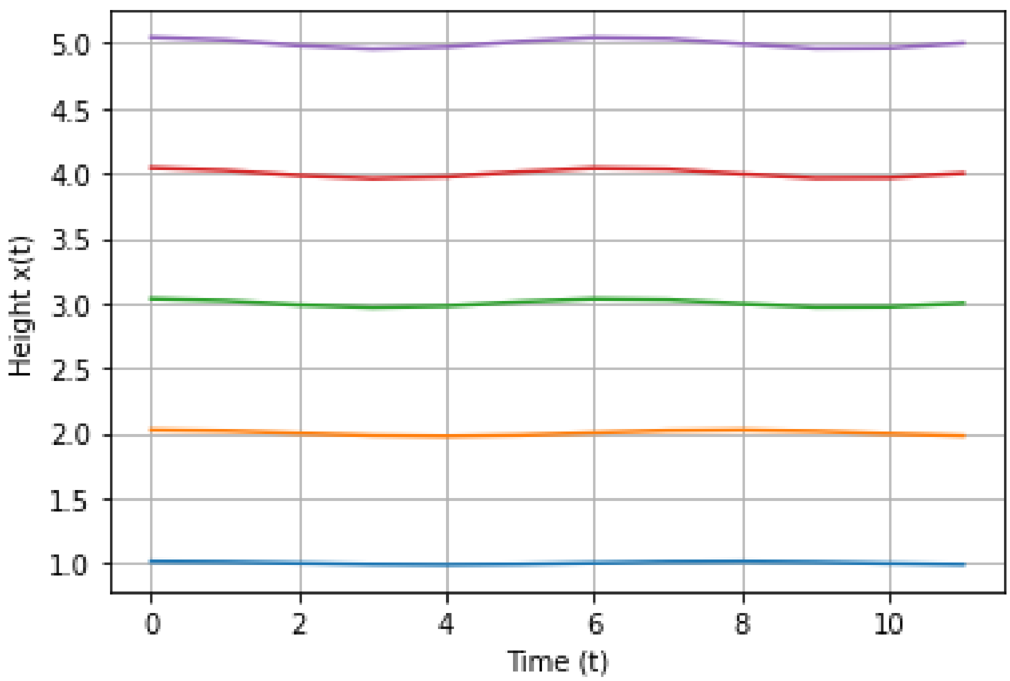

FIGURE 1. The curves describe the vibrations of the floors

Appendix

Tridiagonal Systems of Equations.

```
def triDyGuy(a, b, c, d):
  '''Funtion that takes a tridiagonal matrix A with diaganoals a, b, and c
  where a is the sub diagonal, b is the true diagonal, and c is the super
  diagonal. Vector d is the constant vector. These are used to compute and
  return the solution vector x in from the form Ax = b'''
  n = len(d) # number of equations or rank of matrix
  a, b = map(np.array, a.astype(float), b.astype(float)) # prepare arrays
  c, d = map(np.array, c.astype(float), d.astype(float)) # prepare arrays
  for i in range(1, n):
    xi = a[i-1] / b[i-1]
    b[i] = b[i] - xi*c[i-1]
    d[i] = d[i] - xi*d[i-1]
  x = b
  x[-1] = d[-1] / b[-1]
  for i in range(n-2, -1, -1):
    x[i] = (d[i] - c[i]*x[i+1])/ b[i]
  return x
```

Newton's Method.

```
def superNewton(f, x0, e=1e-10):
  iterations = [x0]
  x = symbols('x')
  f_prime = lambdify(x, diff(f(x), x))
  while True:
    if f_prime(x0) == 0:
      raise ValueError('You messed up.')
    x1 = float(x0 - (f(x0) / f_prime(x0)))
    x0 = x1
    iterations.append(x0)
    if abs(f(x0)) < e:
      break
  return x0, iterations
```

Recursive Characteristic Polynomial.

```
def getCharPoly(subd, d, n):
  '''This function takes the  subdiagonal and diagonal of the hermetian
  matrix and computes the characteristic polynomial implementing the
  recursive formula from part 1. n is the rank of the hermetian matrix.
  This function returns a factored sympy polynomial expression'''
  x = symbols('x')
  subd, d = subd.astype(float), d.astype(float)
  if n == 1:
    p = (d[0] - x)
    return p
  elif n == 2:
    p = (d[1] - x)*(d[0] - x) - abs(subd[0]**2)
    return p
  else:
    a = (d[n-1] - x)*getCharPoly(subd, d, n-1)
    p = a - abs(subd[n-2]**2)*subd[n-3]*getCharPoly(subd, d, n-2)
    return factor(p)
```

```
x = symbols('x')
subd = np.array([1,1,1,1])
d = np.array([-2,-2,-2,-2,-1])

f = getCharPoly(subd, d, 5)
print("the characteristic polynomial is:", f)
print("latex", latex(f))
# these two lines of code use newton's method to find the roots of the
# charicteristic polynomial and add the roots to teh foundEVS list
#f = sp.lambdify(x, f, 'numpy')
#a, b = superNewton(f, -2.5)
foundEVS = [-0.08101405277100507, -0.6902785321094295,
-3.682507065662369, -1.715370323453437, -2.8308300260037758]
foundEVS.sort()
print("Devon's homemade Eigenvalues are:\n\t", foundEVS)
print("Numpy's calculated Eigenvalues:\n\t", ev)
```

Graphing parametric curve.

```
import matplotlib.pyplot as plt
import numpy as np
""" Plots parametric curve x_k(t) + k for k = (1, 2, 3, 4, 5)"""
w = 0.8050542839 # calulated value of omega
def x_1(t): #k = 1
 return 0.12741834 * np.cos(w*t)+ 1

def x_2(t): #k = 2
 return 0.024457401*np.cos(w*t) + 2

def x_3(t):#k = 3
 return 0.0341800605*np.cos(t)+3

def x_4(t):#k = 4
 return 0.0411396548*np.cos(t) + 4

def x_5(t):#k = 5
 return 0.0447663593*np.cos(t)+5

t_range = (np.arange(0, 5, 0.05))
a = x_1(t_range)
b = x_2(t_range)
c = x_3(t_range)
d = x_4(t_range)
e = x_5(t_range)
plt.plot(a)
plt.plot(b)
plt.plot(c)
plt.plot(d)
plt.plot(e)
plt.show()
```