3038741162_Workshop10

April 27, 2023

1 Workshop 10: Linear Algebra in Python

Submit this notebook to bCourses to receive a grade for this Workshop.

Please complete workshop activities in code cells in this iPython notebook. The activities titled **Practice** are purely for you to explore Python, and no particular output is expected. Some of them have some code written, and you should try to modify it in different ways to understand how it works. Although no particular output is expected at submission time, it is *highly* recommended that you read and work through the practice activities before or alongside the exercises. However, the activities titled **Exercise** have specific tasks and specific outputs expected. Include comments in your code when necessary. Enter your name in the cell at the top of the notebook.

The workshop should be submitted on bCourses under the Assignments tab (both the .ipynb and .pdf files).

In lecture, you learned a bit about how to perform matrix operations using `numpy` in Python. Now, we will put those methods to work for us. If you are new to linear algebra, don't worry–we will build some basic intuition for the concepts below without resorting to any explicit proofs. This is perhaps the most important type of math you should learn in all sciences, especially physics, so I encourage you to study it.

Notation: In the equations in the text below, we will adopt the convention that matrices are denoted by capital letters and vectors are denoted by lower case letters.

1.1 Constructing matrices in Python, matrix algebra, and finding determinants

The following code demonstrates the syntax for writing a matrix, adding or multiplying matrices, and calculating determinants using the built-in function `np.linalg.det`

[2]:
```
import numpy as np
A = np.matrix(
    [[0, 4],
     [2, 0]]
)
B = np.matrix(
    [[-1, 2],
     [1, -2]]
)

# Access a single element
```

```
print("first row, first column element of A: %i\n" % A[1,1])

# Add matrices
C = A + B
print("C = A + B = \n",C, "\n")

D = A * B
print("D = A * B = \n",D, "\n")

determinant = np.linalg.det(C)
print("det(C) = %.3f" % determinant)

# Column vector
v = np.transpose(np.matrix([-1, 1]))
print("v = \n", v , "\n")

# Multiply matrix by vector
print("C * v = \n", C * v)

# How to check the dimensions of something, very useful for debugging
print("Shape of v is ", v.shape)
```

```
first row, first column element of A: 0

C = A + B =
 [[-1  6]
 [ 3 -2]]

D = A * B =
 [[ 4 -8]
 [-2  4]]

det(C) = -16.000
v =
 [[-1]
 [ 1]]

C * v =
 [[ 7]
 [-5]]
Shape of v is  (2, 1)
```

1.1.1 Exercise 1

1. Store a 3×3 matrix A which is with entries

$$A = \begin{pmatrix} 2 & 5 & 3 \\ 4 & 10 & 6 \\ 3 & 7 & 3 \end{pmatrix}$$

Calculate its determinant using the built-in `np.linalg.det` function.

2. Write your own determinant function `determinant(M)`, which takes a 3x3 matrix `M` as an argument and returns the determinant of that matrix. Consult the linear algebra refresher notebook in this directory (or Wikipedia) for help.

3. Test your determinant function on the matrix A above. Does your function agree with the built-in function? Is A singular (i.e. non-invertible)?

[55]:
```
import numpy as np
A = np.matrix(
    [[2, 5, 3],
     [4,10, 6],
     [3, 7, 3]]
)

# Code for exercise 1
determinant = np.linalg.det(A)
print("det(A) = %.3f" % determinant)

def det(matrix):
    m11, m12, m13 = matrix[0, 0], matrix[0, 1], matrix[0, 2]
    m21, m22, m23 = matrix[1, 0], matrix[1, 1], matrix[1, 2]
    m31, m32, m33 = matrix[2, 0], matrix[2, 1], matrix[2, 2]
    result = (m11*m22*m33) + (m21*m32*m13) + (m31*m12*m23) - (m11*m32*m23) -␣
 ↪(m31*m22*m13) - (m21*m12*m33)
    return result

print("my_det(A) = %.3f" % det(A))
```

```
det(A) = 0.000
my_det(A) = 0.000
```

1.2 Inverting matrices

Linear systems of equations are of the form

$$Av = b$$

where A is a square matrix, v is a vector of variables, and b is a vector of values. This equation is solved by

$$v = A^{-1}b$$

where A^{-1} is the inverse of A, defined such that $AA^{-1} = A^{-1}A = I$, the identity matrix. Typically inverting a matrix is not the optimal numerical method for solving such problems, but there may be other occasions when inverting a matrix is necessary; the code below gives an example of how to compute the inverse of a matrix using the built-in function `np.linalg.inv`.

[20]:
```
M = np.matrix(
[[10,-7,0],
[-3,2.099,6],
[5,-1,5]]
)
print('We have matrix M = \n')
print(M)
Minv = np.linalg.inv(M)
print('Its inverse M^-1 = \n')
print(Minv)
```

```
We have matrix M =

[[10.    -7.     0.   ]
 [-3.     2.099  6.   ]
 [ 5.    -1.     5.   ]]
Its inverse M^-1 =

[[-1.09930023e-01 -2.33255581e-01  2.79906698e-01]
 [-2.99900033e-01 -3.33222259e-01  3.99866711e-01]
 [ 4.99500167e-02  1.66611130e-01  6.66444518e-05]]
```

1.2.1 Exercise 2

1. Invert the matrices C and D in the code cell below. Then multiply C and D on the left by your answer for their respective inverses (calculate $C^{-1}C$ and $D^{-1}D$). Is the product what you expect?
2. Let
$$b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$
Use the matrices given to solve for v in the linear equations $Cv = b$ and $Dv = b$ by inverting C and D.

[21]:
```
# Matrices for Exercise 2
# Do not delete this cell
C = np.matrix([[5, 3],[-6, 7]])
D = np.matrix([[0, 1],[1, 1]])
```

[22]:
```
# Code for Exercise 2
Cinv = np.linalg.inv(C)
Dinv = np.linalg.inv(D)

CC = C * Cinv
DD = D * Dinv

print('Cinv * C = ', CC)
print('Dinv * D = ', DD)
```

```
b = np.matrix([[1], [2]])
v = Cinv * b
print('v = Cinv * b : ', v)
v = Dinv * b
print('v = Dinv * b : ', v)
```

```
Cinv * C =  [[ 1.0000000e+00 -6.9388939e-17]
 [-1.2490009e-16  1.0000000e+00]]
Dinv * D =  [[1. 0.]
 [0. 1.]]
v = Cinv * b :  [[0.01886792]
 [0.30188679]]
v = Dinv * b :  [[1.]
 [1.]]
```

1.3 Solving systems of linear equations

A typically more efficient way of solving the equation

$$Av = b$$

is using Gaussian elimination, which is implemented in `np.linalg.solve` as below.

[35]:
```
A = np.matrix(
[
    [25, 5, 1],
    [64, 8, 1],
    [144, 12, 1]
])

b = np.transpose(np.matrix([106.8, 177.2, 279.2]))

v = np.linalg.solve(A, b)
print("Solution: v=\n", v, "\n")
print("A * v = \n", A * v)
```

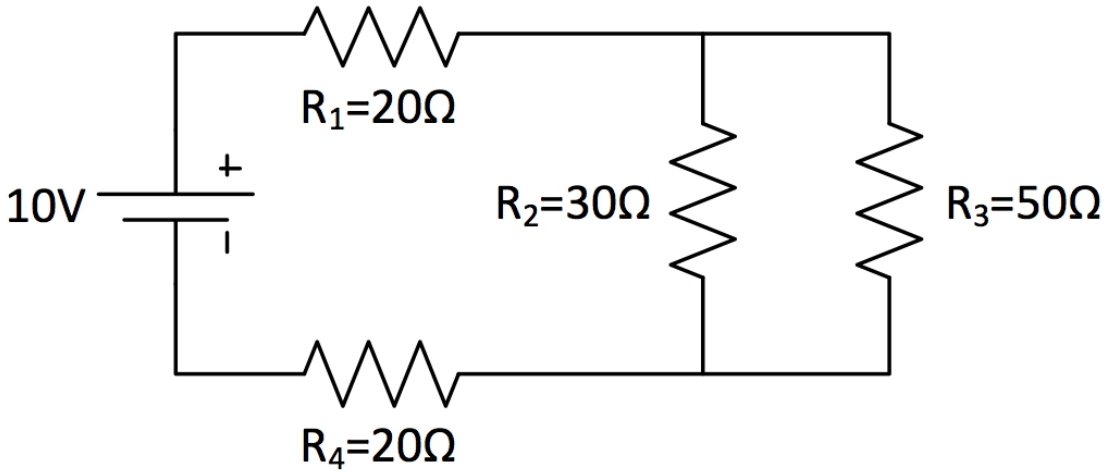
```
Solution: v=
 [[ 0.29047619]
 [19.69047619]
 [ 1.08571429]]

A * v =
 [[106.8]
 [177.2]
 [279.2]]
```

1.3.1 Exercise 3 (optional)

a. Solve the systems $Cv = b$ and $Dv = b$, with C, D, and b from Exercise 2 using Gaussian elimination. Do the answers agree?

b. Solving an electromagnetism problem - consider the circuit below:



(Picture of circuit from http://aplusphysics.com)

This circuit has 3 currents I_1, I_2, I_3 for which we need to solve using Kirchoff's loop and junction rules. By applying these rules, we can get the following system of equations:

$$V - I_1R_1 - I_2R_2 - I_1R_4 = 0$$
$$I_1 = I_2 + I_3$$
$$I_2R_2 - I_3R_3 = 0$$

Re-arrange these equations to take the form

$$Av = b$$

where v is a vector of your three unknown currents $(I_1, I_2, I_3)^T$ and A is a 3×3 matrix of coefficients and b is a vector of 3 constants. Then, use any numerical method you would like to solve for the three constants.

[40]:
```
# Code for Exercise 3a
b = np.transpose(np.matrix([1, 2]))

v = np.linalg.solve(C, b)
print("Solution: v=\n", v, "\n")
print("C * v = \n", C * v, "\n")

v = np.linalg.solve(D, b)
print("Solution: v=\n", v, "\n")
print("D * v = \n", D * v)
```

```
Solution: v=
 [[0.01886792]
 [0.30188679]]

C * v =
 [[1.]
 [2.]]

Solution: v=
 [[1.]
 [1.]]

D * v =
 [[1.]
 [2.]]
```

[47]:
```
# Code for Exercise 3b
R1 = 20
R2 = 30
R4 = 20
R3 = 50

A = np.matrix([[R1 + R4, -R2, 0], [1, -1, -1], [0, R2, -R3]])
b = np.transpose(np.matrix([10, 0, 0]))

I = np.linalg.solve(A, b)
print("Unknown currents: ", I)
```

```
Unknown currents:  [[0.47058824]
 [0.29411765]
 [0.17647059]]
```

1.4 Eigenvalues and Eigenvectors

Probably the most famous problem in all of science is the eigenvalue problem. It is ubiquitous across fields and is the defining problem of quantum mechanics. In this problem, we change the problem slightly from before. For a given A, we try to look for scalar values λ and vectors v that satisfy the matrix equation

$$Av = \lambda v$$

Notice the geometrical interpretation of this problem: the matrix A acting on the vector v does not change its direction; it only rescales v (changes its length). In this case, v is called an eigenvector, and λ is called its corresponding eigenvalue. An example of how to solve an eigenvalue problem using the built-in `np.linalg.eig` function is shown below.

[42]:
```
A = np.matrix(
[
    [25,5,1],
    [64,8,1],
    [144,12,1]
])
eigenvals, eigenvecs = np.linalg.eig(A)
print('Eigenvalues of A=', eigenvals)
print('Eigenvectors: \n', eigenvecs)
```

```
Eigenvalues of A= [40.01957922 -6.35011985  0.33054063]
Eigenvectors:
 [[-0.20222943 -0.06964965  0.01954073]
 [-0.43166576  0.24321109 -0.28790386]
 [-0.8790722   0.96746953  0.95745994]]
```

Each eigenvector is a **column** in the matrix `eigenvecs` above and its corresponding eigenvalue is in `eigenvals`. For example, the first eigenvalue is `40.01957922` and its corresponding eigenvector is the first column of `eigenvecs`:

[43]:
```
# Check that A * v * (1/lambda) = v
A * eigenvecs[:,0] / eigenvals[0]
```

[43]:
```
matrix([[-0.20222943],
        [-0.43166576],
        [-0.8790722 ]])
```

1.4.1 Example: Particle in a Box with Electric Field (adapted from Newman Ch.6)

Solving a quantum mechanics problem - if you haven't taken quantum mechanics, don't worry: I've done the physics portion for you, so the remaining part is just computational. If you have... notice how this method allows you to find approximate energy levels / wavefunctions for problems that may be analytically intractable!

Suppose a particle is in a square well potential V with infinite barriers at $x \leq 0$ and $x \geq L$. The basic problem, which is exactly solvable, is called the "particle in a box" and has $V = 0$ for $0 < x < L$:

Now, we want to go beyond this simple description and try to solve the case where $V(x) = (a/L)x$ for $0 < x < L$ (but still $V = \infty$ for $x \leq 0$ and $x \geq L$). This would be the case if, for example, the particle in the box were charged and a finite electric field were applied.

Because $V = \infty$ outside the box, the wavefunction $\psi(x)$ must be equal to zero for $x \leq 0$ and $x \geq L$ (remember that if $\psi(x) \neq 0$, there is a nonzero probability of finding the particle at x). To solve this, we can express $\psi(x)$ for $0 < x < L$ as a linear combination of functions which are equal to zero at $x = 0, L$. That way, they respect this condition. Those functions are of the form

$$\psi(x) = \sum_{n=1}^{\infty} \psi_n \sin\left(\frac{n\pi x}{L}\right)$$

The coefficients ψ_n are the unknowns for which we need to solve. Let us think of these unknowns as forming a vector $v = (\psi_1, \psi_2, \dots)^T$. The Schrodinger equation is a matrix equation that looks like

$$Hv = Ev$$

where H is a matrix (called the Hamiltonian) and E is a scalar (the energies of H). Therefore, this is an eigenvalue problem. What actually is the matrix H? For the potential described above, it can be shown that the entries of H take on the form

$$H_{mn} = \begin{cases} 0 & m \neq n \text{ and both are even or odd} \\ -\frac{8amn}{\pi^2(m^2-n^2)^2} & m \neq n \text{ and one is even and one is odd} \\ \frac{a}{2} + \frac{\hbar^2\pi^2n^2}{2ML^2} & m = n \end{cases}$$

Here M is the mass of the particle, $\hbar$ is the reduced Planck's constant (values given below). So now we have a way to calculate all of the elements of H, so we have enough to set up the problem. But there is one little thing we need to deal with. The expression for $\psi(x)$ involve summing from $n = 1$ to $n = \infty$! That means v has ∞ elements and H and is an $\infty \times \infty$ matrix! So we need to make some simplifications. One case we can consider is that if we are only interested in the lowest energies (lowest eigenvalues) of H, then we likely do not need to consider the high frequency (high n) functions in the expansion of $\psi(x)$. So let us choose to only consider $n = 1 \dots 10$, so v has 10 elements and H is a \$10×10 \$ matrix.

A physically realistic setup might have that our particle is an electron sitting in optical trap of width L. So here are some plausible values to use:

$$M = 9.109 \times 10^{-31} \text{ kg}$$
$$a = 1.60218 \times 10^{-18} \text{ J}$$
$$L = 5 \times 10^{-10} \text{ m}$$
$$\hbar = 1.0545 \times 10^{-34} \text{ J} \cdot \text{s}$$

Find the energies (eigenvalues) of the 10×10 matrix H. When you are populating the elements of H, beware that in the equations above, $n = 1 \dots 10$, but your matrix dimensions start at 0.

[41]:
```
import numpy as np
import scipy.constants as p

M = p.electron_mass   # kg
hbar = p.hbar         # J*s
a = 10 * p.e          # J, 10 electron volts
L = 5e-10             # m

n_max = 10

H = np.zeros((n_max,n_max))
```

```
for m_idx in range(H.shape[0]):
    for n_idx in range(H.shape[1]):
        m = m_idx + 1
        n = n_idx + 1
        if m == n:
            H[m_idx, n_idx] = a/2 + (hbar * np.pi * n)**2/(2 * M * L**2)
        elif (m%2 == n%2):
            H[m_idx, n_idx] = 0
        else:
            H[m_idx, n_idx] = (-8*a*m*n)/(np.pi**2 * (m**2 - n**2)**2)


energies, _ = np.linalg.eig(H)

print('Energies: (J)')
for energy in energies:
    print('\t{:8.2g}'.format(energy))
```

```
Energies: (J)
        2.5e-17
          2e-17
        1.6e-17
        9.4e-19
        1.8e-18
          3e-18
        4.7e-18
        6.8e-18
        9.5e-18
        1.3e-17
```

[ ]: