Summary

September 22, 2022

---

1 Numpy

1.1 Array Addition

[ ]:
```
import numpy as np

e = np.array([1, 2, 3])
f = np.array([4, 5, 6])
e + f
```

[ ]:
```
array([5, 7, 9])
```

1.2 Array Multiplication

[ ]:
```
import numpy as np

e = np.array([1, 2, 3])
e*3
```

[ ]:
```
array([3, 6, 9])
```

1.3 Creating Numpy Array

- pip install numpy

[ ]:
```
import numpy as np

e = [1, 2, 3]
f = [4, 5, 6]

v1 = np.array(e)
v2 = np.array(f)

print(v1)
print(v2)
```

```
[1 2 3]
[4 5 6]
```

1.4 Linear Combination

[ ]:
```python
import numpy as np

e = [1, 2, 3]

v1 = np.array(e)
v2 = np.array([4, 5, 6])

print(v1 + v2)
print(3 * v1)
print((0.5 * v1) + (0.5 * v2))
```

```
[5 7 9]
[3 6 9]
[2.5 3.5 4.5]
```

Because array must also have the same shape (same array size), we can define a function to check the shape before adding.

[ ]:
```python
import numpy as np


def add_vec(v1, v2):
    if len(v1) == len(v2):
        return v1+v2
    else:
        return "error: vectors must be in the same size"


e = [1, 2, 3]
f = [4, 5, 6]

v1 = np.array(e)
v2 = np.array(f)

add_vec(v1, v2)
```

[ ]:
```
array([5, 7, 9])
```

2 Vectors

2.1 Dot Product

[ ]:
```
import numpy as np

v1 = np.array([1, 3])
v2 = np.array([4, 2])

print(np.dot(v1, v2))
print(v1 @ v2)
print(np.matmul(v1, v2))
```

```
10
10
10
```

2.2 Length of Vector

[ ]:
```
import numpy as np

v1 = np.array([1, 3])

print(np.sqrt(v1 @ v1))
print((v1 @ v1) ** 0.5)
print(np.linalg.norm(v1))
```

```
3.1622776601683795
3.1622776601683795
3.1622776601683795
```

2.3 Unit Vector

[ ]:
```
import numpy as np

v1 = np.array([1, 3])

print(v1 / np.linalg.norm(v1))
```

```
[0.31622777 0.9486833 ]
```

2.4 Angle Between Vectors

[ ]:
```
import numpy as np

v1 = np.array([1, 3])
v2 = np.array([4, 2])

u1 = v1/np.linalg.norm(v1)
```

```
u2 = v2/np.linalg.norm(v2)

angleDeg = np.arccos(u1 @ u2) * 180 / np.pi
angleRad = np.arccos(u1 @ u2)
print(angleDeg)
print(angleRad)
```

```
45.00000000000001
0.7853981633974484
```

---

3 Matrices

3.1 Matrix Combination

[ ]:
```
import numpy as np

v = np.array([1, 0])
A = np.array([[1, 2], [3, 4]])
B = np.array([[4, 5], [6, 7]])

print(np.add(A, B))
print(A + B, "\n")
print(np.subtract(A, B))
print(A - B, "\n")
```

```
[[ 5  7]
 [ 9 11]]
[[ 5  7]
 [ 9 11]]

[[-3 -3]
 [-3 -3]]
[[-3 -3]
 [-3 -3]]
```

[ ]:
```
import numpy as np

v = np.array([1, 0])
A = np.array([[1, 2],
              [3, 4]])
B = np.array([[4, 5],
              [6, 7]])

print(np.matmul(A, B))
print(A @ B)
```

```python
print(np.dot(A, B))
```

```
[[16 19]
 [36 43]]
[[16 19]
 [36 43]]
[[16 19]
 [36 43]]
```

3.2 Matrix Inverse

[ ]:
```python
import numpy as np

A = np.array([[1, 2],
              [3, 4]])

print(np.linalg.inv(A))
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

3.3 Solving Linear Equations

System of Equation Ax=b

The matrix form of a linear system is Ax=b - A is the coefficient - x is the vector of unknown - b is the vector of right hand side

So, x = (A^-1)b

3.3.1 Numpy solve() Function

[ ]:
```python
import numpy as np

A = np.array([[2, 4, -2],
              [4, 9, -3],
              [-2, -3, 7]])
b = np.array([[2],
              [8],
              [10]])

print(np.linalg.solve(A, b))
```

```
[[-1.]
 [ 2.]
 [ 2.]]
```

3.3.2 Inverse Method

[ ]:
```python
import numpy as np

A = np.array([[2, 4, -2],
              [4, 9, -3],
              [-2, -3, 7]])
b = np.array([[2],
              [8],
              [10]])

print(np.linalg.inv(A).dot(b))
```

```
[[-1.]
 [ 2.]
 [ 2.]]
```

3.3.3 Cramer's Rule / Determinant Method

[ ]:
```python
import numpy as np

A = np.array([[1, -2, 3],
              [-1, 3, 0],
              [2, -5, 5]])
b = np.array([[9],
              [-4],
              [17]])

detA = round(np.linalg.det(A), 3)
print(f"{detA = }\n")
if (detA == 0):
    print("The matrix is singular")
else:
    A1, A2, A3 = A.copy(), A.copy(), A.copy()

    for i in range(3):
        A1[i][0] = b[i]
        A2[i][1] = b[i]
        A3[i][2] = b[i]
    print(f"A1 =\n{A1}\n")
    print(f"A2 =\n{A2}\n")
    print(f"A3 =\n{A3}\n")

    detA1 = round(np.linalg.det(A1), 3)
    detA2 = round(np.linalg.det(A2), 3)
    detA3 = round(np.linalg.det(A3), 3)

    print(f"{detA1 = }")
```

```python
    print(f"{detA2 = }")
    print(f"{detA3 = }\n")

    x = detA1 / detA
    y = detA2 / detA
    z = detA3 / detA
    print(f"{x = }")
    print(f"{y = }")
    print(f"{z = }\n")

    sol = np.array([x, y, z])

    print(f"sol =\n {sol}")
```

```
detA = 2.0

A1 =
[[ 9 -2  3]
 [-4  3  0]
 [17 -5  5]]

A2 =
[[ 1  9  3]
 [-1 -4  0]
 [ 2 17  5]]

A3 =
[[ 1 -2  9]
 [-1  3 -4]
 [ 2 -5 17]]

detA1 = 2.0
detA2 = -2.0
detA3 = 4.0

x = 1.0
y = -1.0
z = 2.0

sol =
 [ 1. -1.  2.]
```

3.3.4 Gaussian Elimination

[ ]:
```python
import numpy as np

A = np.array([[1, -2, 3],
                          [-1, 3, 0],
```

```
                              [2, -5, 5]])
b = np.array([[9],
                              [-4],
                              [17]])

print(f"A =\n{A}\n")
print(f"b =\n{b}\n")

print("Forward Elimination")
A[1] = A[1] - (A[1][0] / A[0][0]) * A[0] # row 2 = row 2 - (row 2 / row 1) *␣
 ↪row 1
A[2] = A[2] - (A[2][0] / A[0][0]) * A[0] # row 3 = row 3 - (row 3 / row 1) *␣
 ↪row 1
print(f"A =\n{A}\n")

A[2] = A[2] - (A[2][1] / A[1][1]) * A[1] # row 3 = row 3 - (row 3 / row 2) *␣
 ↪row 2
print(f"A =\n{A}\n")

print("Back Substitution")
x = np.zeros(3)
x[2] = b[2] / A[2][2] # x3 = b3 / a33
x[1] = (b[1] - A[1][2] * x[2]) / A[1][1] # x2 = (b2 - a32 * x3) / a22
x[0] = ((b[0] - (A[0][1] * x[1])) - (A[0][2] * x[2])) / A[0][0] # x1 = ((b1 -␣
 ↪(a21 * x2)) - (a31 * x3)) / a11
print(f"x =\n{x}\n")
```

```
A =
[[ 1 -2  3]
 [-1  3  0]
 [ 2 -5  5]]

b =
[[ 9]
 [-4]
 [17]]

Forward Elimination
A =
[[ 1 -2  3]
 [ 0  1  3]
 [ 0 -1 -1]]

A =
[[ 1 -2  3]
 [ 0  1  3]
 [ 0  0  2]]
```

```
Back Substitution
x =
[-75.5 -29.5   8.5]
```

3.4 Decimal Places Precision

[ ]:
```
import numpy as np

np.set_printoptions(precision=4)

A = np.array([[1, 2, 3],
              [2, 5, 2],
              [6, -3, 1]])

print(A / 7)
```

```
[[ 0.1429  0.2857  0.4286]
 [ 0.2857  0.7143  0.2857]
 [ 0.8571 -0.4286  0.1429]]
```

3.5 Identity Matrix

[ ]:
```
import numpy as np

I = np.eye(3)
print(I)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

3.6 Matrix Size / Shape

[ ]:
```
import numpy as np

I = np.eye(3)
print(I.size)
```

```
9
```

3.7 Matrix Dimensions

[ ]:
```
import numpy as np

I = np.eye(3)
print(I.shape)
```

```
(3, 3)
```

3.8 Slicing Matrix

[ ]:
```python
import numpy as np

A = np.array([[1, 2, 3],
              [2, 5, 2],
              [6, -3, 1]])

print(A[:2])                        # first two rows
print(A[:2, 1:])                # first two rows, second column
print(A[1:3, :2])                # second and third rows, first two columns
print(A[::2, ::2])                # every second row, every second column
print(A[:, 2])                        # all rows, third column
print(A[::-1])                        # reverse the order of the rows
print(A[::-1, ::-1])        # reverse the order of the rows, reverse the order␣
 ↪of the columns
```

```
[[1 2 3]
 [2 5 2]]
[[2 3]
 [5 2]]
[[ 2  5]
 [ 6 -3]]
[[1 3]
 [6 1]]
[3 2 1]
[[ 6 -3  1]
 [ 2  5  2]
 [ 1  2  3]]
[[ 1 -3  6]
 [ 2  5  2]
 [ 3  2  1]]
```

3.9 Elimination Matrix

[ ]:
```python
import numpy as np


def mat_elim(A):
    E1 = np.eye(A.shape[0])
    E1[1, 0] = -A[1, 0] / A[0, 0]
    A1 = E1 @ A

    E2 = np.eye(A.shape[0])
    E2[2, 0] = -A1[2, 0] / A1[0, 0]
    A2 = E2 @ A1
```

```python
    E3 = np.eye(A.shape[0])
    E3[2, 1] = -A2[2, 1] / A2[1, 1]
    A3 = E3 @ A2

    return E3 @ (E2 @ E1)


A = np.array([[1, 9, 5],
              [2, 12, 7],
              [3, 5, 4]])
print("A = \n", A)
E = mat_elim(A)
print("\nE = \n", E)
print("\nEA = \n", np.around(E @ A, 3))
```

```
A =
 [[ 1  9  5]
 [ 2 12  7]
 [ 3  5  4]]

E =
 [[ 1.      0.      0.    ]
 [-2.      1.      0.    ]
 [ 4.3333 -3.6667  1.    ]]

EA =
 [[ 1.  9.  5.]
 [ 0. -6. -3.]
 [ 0.  0.  0.]]
```

3.10 Permutation Matrix

- If row 1 and row 2 are swapped, then the determinant and determinant is negative

[ ]:
```python
import numpy as np

I = np.eye(3)
P = I.copy()  # don't use P = I

print("Exchange row 2 and row 3")
print(f"Before:\n{P}")

P[1], P[2] = I[2], I[1]

print(f"After:\n{P}")
```

```
Exchange row 2 and row 3
Before:
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
After:
[[1. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]]
```

3.11 Transpose Matrix

[ ]:
```python
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6]])
print(f"A = \n{A}")
print(f"\nA.T = \n{A.T}")
```

```
A =
[[1 2 3]
 [4 5 6]]

A.T =
[[1 4]
 [2 5]
 [3 6]]
```

3.12 Matrix Power

- A^n is **not** raising the power each element of matrix by n times, but rather the dot product of that matrix n times!
- Must be a square matrix

[ ]:
```python
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
print(f"A^3 = \n{A @ A @ A}")
print(f"\nA^3 = \n{np.linalg.matrix_power(A, 3)}")
```

```
A^3 =
[[ 468  576  684]
 [1062 1305 1548]
 [1656 2034 2412]]

A^3 =
[[ 468  576  684]
 [1062 1305 1548]
 [1656 2034 2412]]
```

3.13 Adjoint Matrix

[ ]:
```
import numpy as np

A = np.array([[1, 1, 8],
              [5, 1, 3],
              [7, 6, 6]])

print(np.linalg.inv(A).T)
```

```
[[-0.0736 -0.0552  0.1411]
 [ 0.2577 -0.3067  0.0061]
 [-0.0307  0.227  -0.0245]]
```

3.14 Determinant

[ ]:
```
import numpy as np

A = np.array([[1, 2, 3],
              [2, 5, 2],
              [6, -3, 1]])

print(np.linalg.det(A))
```

```
-77.00000000000001
```

3.15 Matrix Rank

- The rank of a matrix A is the dimension of the vector space generated by its columns.

[ ]:
```
import numpy as np

# matrix rank
A = np.array([[1, 2, 3],
              [2, 5, 2],
              [6, -3, 1]])
print(f"rank(A) = {np.linalg.matrix_rank(A)}")
# shape return the number of rows and columns, so we take the first element␣
 ↪(since both are equal)
print(f"rank(A) = {np.linalg.matrix_power(A, 3).shape[0]}")
```

```
rank(A) = 3
rank(A) = 3
```

3.16 LU Factorization

- L @ U will give the initial matrix

[ ]:
```python
import numpy as np

A = np.array([[1, 2, 3],
              [2, 5, 2],
              [6, -3, 1]])

print(f"A = \n{A}\n")
L = np.eye(A.shape[0]) # (L)ower triangular matrix
U = A.copy() # (U)pper triangular matrix
L[1, 0] = U[1, 0] / U[0, 0] # row 2 = row 2 - (row 2 / row 1) * row 1
U[1] = U[1] - (L[1, 0] * U[0]) # row 2 = row 2 - (row 2 / row 1) * row 1

L[2, 0] = U[2, 0] / U[0, 0] # row 3 = row 3 - (row 3 / row 1) * row 1
U[2] = U[2] - (L[2, 0] * U[0]) # row 3 = row 3 - (row 3 / row 1) * row 1

L[2, 1] = U[2, 1] / U[1, 1] # row 3 = row 3 - (row 3 / row 2) * row 2
U[2] = U[2] - (L[2, 1] * U[1]) # row 3 = row 3 - (row 3 / row 2) * row 2

print(f"L = \n{L}\n")
print(f"U = \n{U}")
```

```
A =
[[ 1  2  3]
 [ 2  5  2]
 [ 6 -3  1]]

L =
[[  1.   0.   0.]
 [  2.   1.   0.]
 [  6. -15.   1.]]

U =
[[  1   2   3]
 [  0   1  -4]
 [  0   0 -77]]
```

3.17 Finding Pivots

Pivots are the first non-zero element in each row of this eliminated matrix.

[ ]:
```python
import numpy as np



def mat_elim(A):
        E1 = np.eye(A.shape[0])
        E1[1, 0] = -A[1, 0] / A[0, 0]
        A1 = E1 @ A
```

```
        E2 = np.eye(A.shape[0])
        E2[2, 0] = -A1[2, 0] / A1[0, 0]
        A2 = E2 @ A1

        E3 = np.eye(A.shape[0])
        E3[2, 1] = -A2[2, 1] / A2[1, 1]
        A3 = E3 @ A2

        return E3 @ (E2 @ E1)


A = np.array([[8, -6, 2],
                          [-6, 7, -4],
                          [2, -4, 3]])
print("A = \n", A)
E = mat_elim(A)
eliminated_mat = np.around(E @ A, 3)
print("\nEliminated = \n", eliminated_mat)

# The first non-zero element in each row of eliminated_mat
pivots = []
for i in range(eliminated_mat.shape[0]):
        for j in range(eliminated_mat.shape[0]):
                if eliminated_mat[i, j] != 0:
                        pivots.append(eliminated_mat[i, j])
                        break
print("\nPivots =", pivots)
```

```
A =
 [[ 8 -6  2]
 [-6  7 -4]
 [ 2 -4  3]]

Eliminated =
 [[ 8.  -6.   2. ]
 [ 0.   2.5 -2.5]
 [ 0.   0.   0. ]]

Pivots = [8.0, 2.5]
```

3.18 Column Space

The column space is a space spanned by the columns of the `initial matrix` that correspond to the pivot columns of the reduced matrix.

[ ]:

```
# Find pivots
import numpy as np
```

```
def mat_elim(A):
        E1 = np.eye(A.shape[0])
        E1[1, 0] = -A[1, 0] / A[0, 0]
        A1 = E1 @ A

        E2 = np.eye(A.shape[0])
        E2[2, 0] = -A1[2, 0] / A1[0, 0]
        A2 = E2 @ A1

        E3 = np.eye(A.shape[0])
        E3[2, 1] = -A2[2, 1] / A2[1, 1]
        A3 = E3 @ A2

        return E3 @ (E2 @ E1)


A = np.array([[1, 9, 5],
                          [2, 12, 7],
                          [3, 5, 4]])
print("A = \n", A)
E = mat_elim(A)
eliminated_mat = np.around(E @ A, 3)
print("\nEliminated = \n", eliminated_mat)

# The first non-zero element in each row of eliminated_mat
pivots = []
pos = ()
for i in range(eliminated_mat.shape[0]):
        for j in range(eliminated_mat.shape[0]):
                if eliminated_mat[i, j] != 0:
                        pivots.append(eliminated_mat[i, j])
                        pos += ((i, j),)
                        break
print("\nPivots =", pivots)

# Column position of each pivots
col_pos = [pos[i][1] for i in range(len(pos))]
print("Column position of each pivots =", col_pos)

# Print the column of the INITIAL MATRIX based on column position of each pivots
print("\nColumn space of the matrix is")
for i in range(len(col_pos)):
        print(A[:, i])
```

```
A =
 [[ 1  9  5]
 [ 2 12  7]
```

```
 [ 3  5  4]]

Eliminated =
 [[ 1.   9.   5.]
 [ 0.  -6.  -3.]
 [ 0.   0.   0.]]

Pivots = [1.0, -6.0]
Column position of each pivots = [0, 1]

Column space of the matrix is
[1 2 3]
[ 9 12  5]
```

3.19 Row Space

The row space is a space spanned by the nonzero rows of the `reduced matrix`.

[ ]:
```
# Find pivots
import numpy as np


def mat_elim(A):
        E1 = np.eye(A.shape[0])
        E1[1, 0] = -A[1, 0] / A[0, 0]
        A1 = E1 @ A

        E2 = np.eye(A.shape[0])
        E2[2, 0] = -A1[2, 0] / A1[0, 0]
        A2 = E2 @ A1

        E3 = np.eye(A.shape[0])
        E3[2, 1] = -A2[2, 1] / A2[1, 1]
        A3 = E3 @ A2

        return E3 @ (E2 @ E1)


A = np.array([[1, 9, 5],
                          [2, 12, 7],
                          [3, 5, 4]])
print("A = \n", A)
E = mat_elim(A)
eliminated_mat = np.around(E @ A, 3)
print("\nEliminated = \n", eliminated_mat)

# The first non-zero element in each row of eliminated_mat
pivots = []
```

```
pos = ()
for i in range(eliminated_mat.shape[0]):
        for j in range(eliminated_mat.shape[0]):
                if eliminated_mat[i, j] != 0:
                        pivots.append(eliminated_mat[i, j])
                        pos += ((i, j),)
                        break
print("\nPivots =", pivots)

# Row position of each pivots
row_pos = [pos[i][0] for i in range(len(pos))]

# Print the row of the REDUCED MATRIX based on row position of each pivots
print("\nRow space of the matrix is")
for i in range(len(row_pos)):
        print(eliminated_mat[i, :])
```

```
A =
 [[ 1  9  5]
 [ 2 12  7]
 [ 3  5  4]]

Eliminated =
 [[ 1.  9.  5.]
 [ 0. -6. -3.]
 [ 0.  0.  0.]]

Pivots = [1.0, -6.0]

Row space of the matrix is
[1. 9. 5.]
[ 0. -6. -3.]
```

3.20 Basis

The basis is a set of linearly independent vectors that spans the given vector space.

- There are many ways to find a basis. One of the ways is to find the row space of the matrix whose rows are the given vectors.
- Another way to find a basis is to find the column space of the matrix whose columns are the given vectors.
- If two different bases were found, they are both the correct answers

For example: A = np.array([[1, 9, 5], [2, 12, 7], [3, 5, 4]])

Column space of the matrix is [1 2 3] [9 12 5]

Row space of the matrix is [1. 9. 5.] [0. -6. -3.]

If two different bases were found, they are both the correct answers: we can choose any of them, for example, the first one. [1 2 3] [9 12 5]

3.21 Nullspace

The nullity of a matrix is the dimension of the basis for the null space.

[ ]:
```
import numpy as np
import sympy

A = np.array([[1, 9, 5],
                           [2, 12, 7],
                           [3, 5, 4]])

# Find Reduced Row Echelon Form
rref = sympy.Matrix(A).rref()
rref = np.array(rref[0], dtype=float)
print("Reduced Row Echelon Form:\n", rref)

"""
Solve Matrix Equation
[1.  0.  0.5][x]=[0]
[0.  1.  0.5][y]=[0]
[0.  0.  0. ][z]=[0]

x+0.5z=0
y+0.5z=0

Add equation for each free variable
x+0.5z=0
y+0.5z=0
z=z

Solve for each variable in terms of the free variables
x=-0.5z
y=-0.5z
z=z

Convert this into vectors
[-0.5, -0.5, 1]
"""



# Find null space
null_space = sympy.Matrix(A).nullspace()
# change to float
null_space = np.array(null_space).astype(float)
print("\nNull space of A is")
```

```
print(null_space)
```

```
Reduced Row Echelon Form:
 [[1.  0.  0.5]
 [0.  1.  0.5]
 [0.  0.  0. ]]

Null space of A is
[[[-0.5]
  [-0.5]
  [ 1. ]]]
```

---

4 Matplotlib

4.1 Colors

[ ]:
```
import matplotlib.colors as mcolors

red = mcolors.to_rgb([1, 0, 0])
green = mcolors.to_rgb([0, 1, 0])
blue = mcolors.to_rgb([0, 0, 1])
```

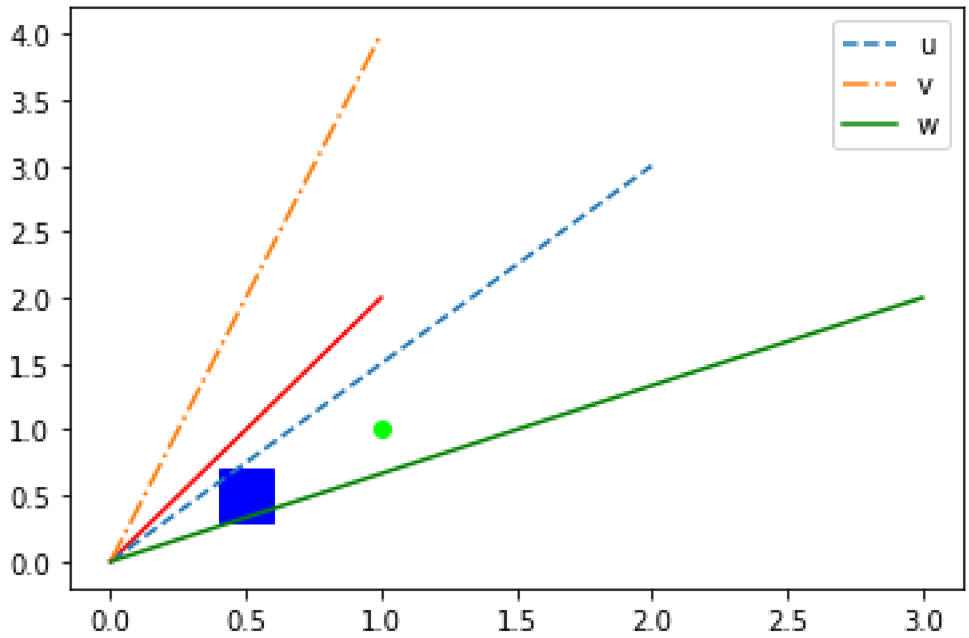
4.2 Plotting

[ ]:
```
import matplotlib.pyplot as plt

red = mcolors.to_rgb([1, 0, 0])
green = mcolors.to_rgb([0, 1, 0])
blue = mcolors.to_rgb([0, 0, 1])

plt.plot([0, 1], [0, 2], color=red)
plt.plot(1, 1, 'o', color=green)
plt.plot(0.5, 0.5, 's', color=blue, markersize=20)
plt.plot([0, 2], [0, 3], '--', label='u')
plt.plot([0, 1], [0, 4], '-.', label='v')
plt.plot([0, 3], [0, 2], 'g-', label='w')
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x28af1b83b80>
```

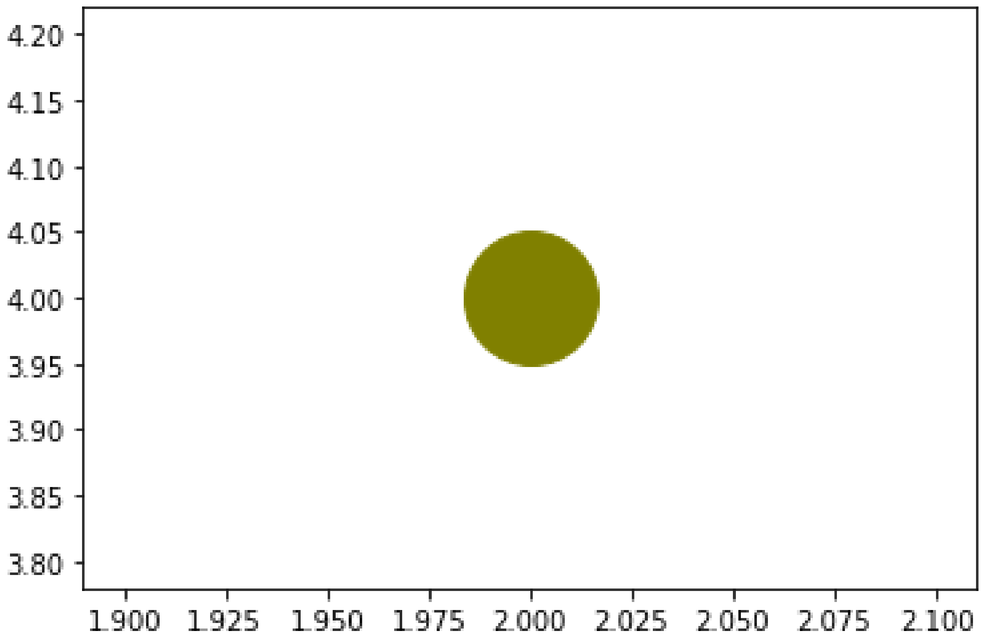


4.3 Mixing Colors

[ ]:
```
def color_mix(c1, c2):
    return (np.array(c1) * 0.5) + (np.array(c2) * 0.5)


red = mcolors.to_rgb([1, 0, 0])
green = mcolors.to_rgb([0, 1, 0])
blue = mcolors.to_rgb([0, 0, 1])
mycolor = color_mix(red, green)
plt.plot(2, 4, 'o', color=mycolor, markersize=50)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x28af1ca6740>]
```


4.20
4.15
4.10
4.05
4.00
3.95
3.90
3.85
3.80
1.900
1.925
1.950
1.975
2.000
2.025
2.050
2.075
2.100