Section 2 warmups

A computer screen with colorful text

Description automatically generated

Section 2 tests

A screen shot of a computer

Description automatically generated

Section 3

3.1:

A light in the dark

Description automatically generatedA light in the dark

Description automatically generated

3.2

A black background with a circle in the middle

Description automatically generatedA black circle with white smoke

Description automatically generatedA black and white image of a planet

Description automatically generatedA black circle with white clouds

Description automatically generatedA black circle with white smoke

Description automatically generatedA black and white image of a planet

Description automatically generatedA black background with a circle in the middle

Description automatically generatedA black circle with white spots

Description automatically generatedA black and white image of a planet

Description automatically generated

3.3

A blue circle with purple light

Description automatically generatedA blue circle with purple lights

Description automatically generatedA purple circle with yellow lights

Description automatically generated with medium confidenceA blue circle with purple lights

Description automatically generated

**Code:**

Section 2

**Warmups.py:**

import numpy as np

def w1(X):

    """

    Input:

    - X: A numpy array

    Returns:

    - A matrix Y such that Y[i, j] = X[i, j] \* 10 + 100

    Hint: Trust that numpy will do the right thing

    """

    return X \* 10 + 100

def w2(X, Y):

    """

    Inputs:

    - X: A numpy array of shape (N, N)

    - Y: A numpy array of shape (N, N)

    Returns:

    A numpy array Z such that Z[i, j] = X[i, j] + 10 \* Y[i, j]

    Hint: Trust that numpy will do the right thing

    """

    return X + 10 \* Y

def w3(X, Y):

    """

    Inputs:

    - X: A numpy array of shape (N, N)

    - Y: A numpy array of shape (N, N)

    Returns:

    A numpy array Z such that Z[i, j] = X[i, j] \* Y[i, j] - 10

    Hint: By analogy to +, \* will do the same thing

    """

    return X \* Y - 10

def w4(X, Y):

    """

    Inputs:

    - X: Numpy array of shape (N, N)

    - Y: Numpy array of shape (N, N)

    Returns:

    A numpy array giving the matrix product X times Y

    Hint:

    1. Be careful! There are different variants of \*, @, dot

    2.  a = [[1,2],

             [1,2]]

        b = [[2,2],

             [3,3]]

        a \* b = [[2,4],

                 [3,6]]

    Is this matrix multiplication?

    """

    return X @ Y

def w5(X):

    """

    Inputs:

    - X: A numpy array of shape (N, N) of floating point numbers

    Returns:

    A numpy array with the same data as X, but cast to 32-bit integers

    Hint: Check .astype() !

    """

    return X.astype(np.int32)

def w6(X, Y):

    """

    Inputs:

    - X: A numpy array of shape (N,) of integers

    - Y: A numpy array of shape (N,) of integers

    Returns:

    A numpy array Z such that Z[i] = float(X[i]) / float(Y[i])

    """

    return X.astype(np.float64)/Y.astype(np.float64)

def w7(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    - A numpy array Y of shape (N \* M, 1) containing the entries of X in row

      order. That is, X[i, j] = Y[i \* M + j, 0]

    Hint:

    1) np.reshape

    2) You can specify an unknown dimension as -1

    """

    return X.reshape(-1,1)

def w8(N):

    """

    Inputs:

    - N: An integer

    Returns:

    A numpy array of shape (N, 2N)

    Hint: The error "data type not understood" means you probably called

    np.ones or np.zeros with two arguments, instead of a tuple for the shape

    """

    return np.ones((N,2\*N))

def w9(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M) where each entry is between 0 and 1

    Returns:

    A numpy array Y where Y[i, j] = True if X[i, j] > 0.5

    Hint: Try boolean array indexing

    """

    return X > 0.5

def w10(N):

    """

    Inputs:

    - N: An integer

    Returns:

    A numpy array X of shape (N,) such that X[i] = i

    Hint: np.arange

    """

    return np.arange(N)

def w11(A, v):

    """

    Inputs:

    - A: A numpy array of shape (N, F)

    - v: A numpy array of shape (F, 1)

    Returns:

    Numpy array of shape (N, 1) giving the matrix-vector product Av

    """

    return A.dot(v)

def w12(A, v):

    """

    Inputs:

    - A: A numpy array of shape (N, N), of full rank

    - v: A numpy array of shape (N, 1)

    Returns:

    Numpy array of shape (N, 1) giving the matrix-vector product of the inverse

    of A and v: A^-1 v

    """

    return np.linalg.inv(A).dot(v)

def w13(u, v):

    """

    Inputs:

    - u: A numpy array of shape (N, 1)

    - v: A numpy array of shape (N, 1)

    Returns:

    The inner product u^T v

    Hint: .T

    """

    return u.T @ (v)

def w14(v):

    """

    Inputs:

    - v: A numpy array of shape (N, 1)

    Returns:

    The L2 norm of v: norm = (sum\_i^N v[i]^2)^(1/2)

    You MAY NOT use np.linalg.norm

    """

    return np.sqrt(np.sum(v\*\*2))

def w15(X, i):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    - i: An integer in the range 0 <= i < N

    Returns:

    Numpy array of shape (M,) giving the ith row of X

    """

    return X[i]

def w16(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    The sum of all entries in X

    Hint: np.sum

    """

    return np.sum(X)

def w17(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    A numpy array S of shape (N,) where S[i] is the sum of row i of X

    Hint: np.sum has an optional "axis" argument

    """

    return np.sum(X, axis=1)

def w18(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    A numpy array S of shape (M,) where S[j] is the sum of column j of X

    Hint: Same as above

    """

    return np.sum(X, axis=0)

def w19(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    A numpy array S of shape (N, 1) where S[i, 0] is the sum of row i of X

    Hint: np.sum has an optional "keepdims" argument

    \*\*\* keepdims=True simply makes it (N,1) for a 2D answer rather than (N,)

    """

    return np.sum(X, axis=1, keepdims=True)

def w20(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    A numpy array S of shape (N, 1) where S[i] is the L2 norm of row i of X

    """

    return np.sqrt(np.sum(X\*\*2, axis=1, keepdims=True))

**Tests.py:**

import numpy as np

def t1(L):

    """

    Inputs:

    - L: A list of M numpy arrays, each of shape (1, N)

    Returns:

    A numpy array of shape (M, N) giving all inputs stacked together

    Par: 1 line

    Instructor: 1 line

    Hint: vstack/hstack/dstack, no for loop

    """

    return np.vstack(L)

def t2(X):

    """

    Inputs:

    - X: A numpy array of shape (N, N)

    Returns:

    Numpy array of shape (N,) giving the eigenvector corresponding to the

    smallest eigenvalue of X

    Par: 5 lines

    Instructor: 3 lines

    Hints:

    1) np.linalg.eig

    2) np.argmin

    3) Watch rows and columns!

    """

    eig\_vals, eig\_vecs = np.linalg.eig(X) # returns an Eig object of ([eigenvalues], [eigenvectors])

    eig\_val\_min = np.argmin(eig\_vals) # selects the INDEX of the smallest argument in the 1D array of Eigenvalues

    '''

    Eigenvectors is an (N,N) matrix where each column (,n) is an eigenvector rather than the rows

    eig\_vecs[eig\_val\_min] returns the row which is incorrect while eig\_vecs[: , eig\_val\_min] returns the column

    '''

    # Use the index of the minimum to return the corresponding eigenvector

    return eig\_vecs[: ,eig\_val\_min]

def t3(X):

    """

    Inputs:

    - A: A numpy array of any shape

    Returns:

    A copy of X, but with all negative entires set to 0

    Par: 3 lines

    Instructor: 1 line

    Hint:

    1) If S is a boolean array with the same shape as X, then X[S] gives an

       array containing all elements of X corresponding to true values of S

    2) X[S] = v assigns the value v to all entires of X corresponding to

       true values of S.

    """

    return np.where(X < 0, 0, X)

def t4(R, X):

    """

    Inputs:

    - R: A numpy array of shape (3, 3) giving a rotation matrix

    - X: A numpy array of shape (N, 3) giving a set of 3-dimensional vectors

    Returns:

    A numpy array Y of shape (N, 3) where Y[i] is X[i] rotated by R

    Par: 3 lines

    Instructor: 1 line

    Hint:

    1) If v is a vector, then the matrix-vector product Rv rotates the vector

       by the matrix R.

    2) .T gives the transpose of a matrix

    Why use R.T instead of R?

        Rotation matrices are typically orthogonal, meaning that to rotate vectors,

        you should multiply by the transpose of the rotation matrix

    This correctly applies the rotation to each row vector in X (vector-matrix multiplication)

    Rotation matrices typically assume column vectors when they are defined mathematically.

    However, in numerical programming, row vectors (each row being a vector) are often used,

    requiring a transpose operation for correct application.

    So in matrix multiplication, if you were to do nothing, you'd be applying stuff row-wise, but by transposing

    you apply the second matrix to the first matrix column wise, thus achieving the desired transformation

    """

    return X @ R.T

def t5(X):

    """

    Inputs:

    - X: A numpy array of shape (N, N)

    Returns:

    A numpy array of shape (4, 4) giving the upper left 4x4 submatrix of X

    minus the bottom right 4x4 submatrix of X.

    Par: 2 lines

    Instructor: 1 line

    Hint:

    1) X[y0:y1, x0:x1] gives the submatrix

       from rows y0 to (but not including!) y1

       from columns x0 (but not including!) x1

    """

    return X[:4, :4] - X[-4:,-4:]

def t6(N):

    """

    Inputs:

    - N: An integer

    Returns:

    A numpy array of shape (N, N) giving all 1s, except the first and last 5

    rows and columns are 0.

    Par: 6 lines

    Instructor: 3 lines

    """

    a = np.ones((N, N))  # Create an NxN matrix filled with 1s

    a[:5, :] = 0  # Set the first 5 rows to 0

    a[-5:, :] = 0  # Set the last 5 rows to 0

    a[:, :5] = 0  # Set the first 5 columns to 0

    a[:, -5:] = 0  # Set the last 5 columns to 0

    return a

def t7(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    A numpy array Y of the same shape as X, where Y[i] is a vector that points

    the same direction as X[i] but has unit norm.

    Par: 3 lines

    Instructor: 1 line

    Hints:

    1) The vector v / ||v||| is the unit vector pointing in the same direction

       as v (as long as v != 0)

    2) Divide each row of X by the magnitude of that row

    3) Elementwise operations between an array of shape (N, M) and an array of

       shape (N, 1) work -- try it! This is called "broadcasting"

    4) Elementwise operations between an array of shape (N, M) and an array of

       shape (N,) won't work -- try reshaping

    The row magnitudes would be the L2 norm. So executing the solution from the

    last of the warmups, storing them in an (N,1) array and then dividing X by

    that array would work, but np.linalg.norm already executes this.

    np.linalg.norm(axis =1, keepdims=True) executes row-wise and keeps as a 2D array

    """

    return X / np.linalg.norm(X, axis=1, keepdims=True)

def t8(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    A numpy array Y of shape (N, M) where Y[i] contains the same data as X[i],

    but normalized to have mean 0 and standard deviation 1.

    Par: 3 lines

    Instructor: 1 line

    Hints:

    1) To normalize X, subtract its mean and then divide by its standard deviation

    2) Normalize the rows individually

    3) You may have to reshape

    """

    row\_mean = X.mean(axis=1, keepdims=True)

    row\_std = X.std(axis=1, keepdims=True)

    return (X - row\_mean) / row\_std

def t9(q, k, v):

    """

    Inputs:

    - q: A numpy array of shape (1, K) (queries)

    - k: A numpy array of shape (N, K) (keys)

    - v: A numpy array of shape (N, 1) (values)

    Returns:

    sum\_i exp(-||q-k\_i||^2) \* v[i]

    Par: 3 lines

    Instructor: 1 ugly line

    Hints:

    1) You can perform elementwise operations on arrays of shape (N, K) and

       (1, K) with broadcasting

    2) Recall that np.sum has useful "axis" and "keepdims" options

    3) np.exp and friends apply elementwise to arrays

    \*\*\* First conduct the sums of the Euclidean distances followed by

    exponent of the negative of the distances sums, THEN the sum of that \* V

    """

    dist\_sum = np.sum((q-k)\*\*2, axis=1, keepdims=True)

    return np.sum(np.exp(-dist\_sum)\*v)

def t10(Xs):

    """

    Inputs:

    - Xs: A list of length L, containing numpy arrays of shape (N, M)

    Returns:

    A numpy array R of shape (L, L) where R[i, j] is the Euclidean distance

    between C[i] and C[j], where C[i] is an M-dimensional vector giving the

    centroid of Xs[i]

    Par: 12 lines

    Instructor: 3 lines (after some work!)

    Hints:

    1) You can try to do t11 and t12 first

    2) You can use a for loop over L

    3) Distances are symmetric

    4) Go one step at a time

    5) Our 3-line solution uses no loops, and uses the algebraic trick from the

       next problem.

    """

    # Compute the centroid aka mean of rows

    ''' Killer way to execute this is using what I'm calling an array comprehension

            Basically a list comprehension inside np.array instantion'''

    centroids\_of\_arrays = np.array([np.mean(X, axis=0) for X in Xs]) # row-wise centroids aka means

    # Use equations below

    dist\_of\_centroids = np.sum(centroids\_of\_arrays\*\*2, axis=1, keepdims=True)

    Distance\_squared = dist\_of\_centroids + dist\_of\_centroids.T - 2 \* centroids\_of\_arrays @ centroids\_of\_arrays.T

    return np.sqrt(np.maximum(Distance\_squared, 0))

def t11(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    A numpy array D of shape (N, N) where D[i, j] gives the Euclidean distance

    between X[i] and X[j], using the identity

    ||x - y||^2 = is the distance squared so square root that

    ||x||^2 + ||y||^2 - 2x^T y is what to focus on

    \*\*\* And y = X.T

    Par: 3 lines

    Instructor: 2 lines (you can do it in one but it's wasteful compute-wise)

    Hints:

    1) What happens when you add two arrays of shape (1, N) and (N, 1)?

    2) Think about the definition of matrix multiplication

    3) Transpose is your friend

    4) Note the square! Use a square root at the end

    5) On some machines, ||x||^2 + ||x||^2 - 2x^Tx may be slightly negative,

       causing the square root to crash. Just take max(0, value) before the

       square root. Seems to occur on Macs.

    """

    # Square X in preparation for full equation

    X\_squared = np.sum(X\*\*2, axis=1, keepdims=True)

    Distance\_squared = X\_squared + X\_squared.T - 2 \* X @ X.T

    return np.sqrt(np.maximum(Distance\_squared, 0))

def t12(X, Y):

    """

    Inputs:

    - X: A numpy array of shape (N, F)

    - Y: A numpy array of shape (M, F)

    Returns:

    A numpy array D of shape (N, M) where D[i, j] is the Euclidean distance

    between X[i] and Y[j].

    Par: 3 lines

    Instructor: 2 lines (you can do it in one, but it's more than 80 characters

                with good code formatting)

    Hints: Similar to previous problem

    """

    X\_squared = np.sum(X\*\*2, axis=1, keepdims=True)

    Y\_squared = np.sum(Y\*\*2, axis=1, keepdims=True).T

    return np.sqrt(np.maximum(X\_squared + Y\_squared - 2 \* X @ Y.T, 0))

def t13(q, V):

    """

    Inputs:

    - q: A numpy array of shape (1, M) (query)

    - V: A numpy array of shape (N, M) (values)

    Return:

    The index i that maximizes the dot product q . V[i]

    Par: 1 line

    Instructor: 1 line

    Hint: np.argmax

    \*\*\* Dimensions:

        V.T (transpose of V) will have shape (M, N).

        q @ V.T results in shape (1, N).

        And we need (N,1) so do it like below

        where (N,M) \* (M, 1)

    """

    return np.argmax(V @ q.T)

def t14(X, y):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    - y: A numpy array of shape (N, 1)

    Returns:

    A numpy array w of shape (M, 1) such that ||y - Xw||^2 is minimized

    Par: 2 lines

    Instructor: 1 line

    Hint: np.linalg.lstsq or np.linalg.solve

    \*\*\* Finding the optimal weight vecor that minimizes the squared error

        AKA a linear least squares problem and np.linalg.lstsq aka least squares

    """

    return np.linalg.lstsq(X,y, rcond=None)[0]

def t15(X, Y):

    """

    Inputs:

    - X: A numpy array of shape (N, 3)

    - Y: A numpy array of shape (N, 3)

    Returns:

    A numpy array C of shape (N, 3) such C[i] is the cross product between X[i]

    and Y[i]

    Par: 1 line

    Instructor: 1 line

    Hint: np.cross

    """

    return np.cross(X,Y)

def t16(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    A numpy array Y of shape (N, M - 1) such that

    Y[i, j] = X[i, j] / X[i, M - 1]

    for all 0 <= i < N and all 0 <= j < M - 1

    Par: 1 line

    Instructur: 1 line

    Hints:

    1) If it doesn't broadcast, reshape or np.expand\_dims

    2) X[:, -1] gives the last column of X

    \*\*\* expand\_dims solution:  X[:, :-1] / np.expand\_dims(X[:, -1], axis=1)

    Probably best solution: X[:, :-1] / X[:, -1, np.newaxis]

    """

    return X[:, :-1] / X[:, -1].reshape(-1, 1)

def t17(X):

    """

    Inputs:

    - X: A numpy array of shape (N, M)

    Returns:

    A numpy array Y of shape (N, M + 1) such that

        Y[i, :M] = X[i]

        Y[i, M] = 1

    Par: 1 line

    Instructor: 1 line

    Hint: np.hstack, np.ones

    \*\*\* hstack adds columns while vstack adds rows

    """

    # Basically just use np.ones of shape, X number of rows to create an array of size(1,M) with the value 1 and hstack it to X

    return np.hstack((X, np.ones((X.shape[0], 1))))

def t18(N, r, x, y):

    """

    Inputs:

    - N: An integer

    - r: A floating-point number

    - x: A floating-point number

    - y: A floating-point number

    Returns:

    A numpy array I of floating point numbers and shape (N, N) such that:

    I[i, j] = 1 if ||(j, i) - (x, y)|| < r

    I[i, j] = 0 otherwise

    Par: 3 lines

    Instructor: 2 lines

    Hints:

    1) np.meshgrid and np.arange give you X, Y. Play with them. You can also do

    it without them, but np.meshgrid and np.arange are easier to understand.

    2) Arrays have an astype method

    """

    X, Y = np.meshgrid(np.arange(N), np.arange(N))

    return (np.sqrt((X - x)\*\*2 + (Y - y)\*\*2) < r).astype(float)

def t19(N, s, x, y):

    """

    Inputs:

    - N: An integer

    - s: A floating-point number

    - x: A floating-point number

    - y: A floating-point number

    Returns:

    A numpy array I of shape (N, N) such that

    I[i, j] = exp(-||(j, i) - (x, y)||^2 / s^2)

    Par: 3 lines

    Instructor: 2 lines

    """

    X, Y = np.meshgrid(np.arange(N), np.arange(N))

    return np.exp(-((X - x)\*\*2 + (Y - y)\*\*2) / s\*\*2)

def t20(N, v):

    """

    Inputs:

    - N: An integer

    - v: A numpy array of shape (3,) giving coefficients v = [a, b, c]

    Returns:

    A numpy array of shape (N, N) such that M[i, j] is the distance between the

    point (j, i) and the line a\*j + b\*i + c = 0

    Par: 4 lines

    Instructor: 2 lines

    Hints:

    1) The distance between the point (x, y) and the line ax+by+c=0 is given by

       abs(ax + by + c) / sqrt(a^2 + b^2)

       (The sign of the numerator tells which side the point is on)

    2) np.abs

    \*\*\*

    """

    X, Y = np.meshgrid(np.arange(N), np.arange(N))

    return np.abs(v[0] \* X + v[1] \* Y + v[2]) / np.sqrt(v[0]\*\*2 + v[1]\*\*2)

Visualize:

import os

import numpy as np

import matplotlib.pyplot as plt

import cv2

def colormapArray(X, colors):

    """

    Basically plt.imsave but return a matrix instead

    Given:

        a HxW matrix X

        a Nx3 color map of colors in [0,1] [R,G,B]

    Outputs:

        a HxW uint8 image using the given colormap. See the Bewares

    """

    X\_normalized = (X - np.nanmin(X)) / (np.nanmax(X) - np.nanmin(X))

    X\_normalized = np.clip(X\_normalized, 0, 1)

    indices = (X\_normalized \* (len(colors) - 1)).astype(int)

    colormapped\_image = (colors[indices] \* 255).astype(np.uint8)

    return colormapped\_image

if \_\_name\_\_ == "\_\_main\_\_":

    # Solve 3.1: Nonlinear correction and visualization

    data2 = np.load("mysterydata/mysterydata2.npy")

    corrected\_sqrt = np.sqrt(data2)

    corrected\_log1p = np.log1p(data2)

    plt.imsave("mysterydata2\_sqrt.png", corrected\_sqrt[:, :, 0], cmap='gray')

    plt.imsave("mysterydata2\_log1p.png", corrected\_log1p[:, :, 0], cmap='gray')

    print("Saved corrected images for mysterydata2.npy")

    # Solve 3.2: Handling NaN and Inf values in mysterydata3.npy

    data3 = np.load("mysterydata/mysterydata3.npy")

    finite\_fraction = np.mean(np.isfinite(data3))

    print(f"Fraction of finite values: {finite\_fraction}")

    if finite\_fraction < 1:

        data3\_cleaned = np.nan\_to\_num(data3, nan=np.nanmin(data3), posinf=np.nanmax(data3), neginf=np.nanmin(data3))

    else:

        data3\_cleaned = data3

    for i in range(9):

        vmin, vmax = np.nanmin(data3\_cleaned[:, :, i]), np.nanmax(data3\_cleaned[:, :, i])

        plt.imsave(f"vis3\_{i}.png", data3\_cleaned[:, :, i], vmin=vmin, vmax=vmax, cmap='gray')

    print("Saved cleaned images for mysterydata3.npy")

    # Solve 3.3: Custom colormap visualization for mysterydata4.npy

    data4 = np.load("mysterydata/mysterydata4.npy")

    colors = np.load("mysterydata/colors.npy")

    for i in range(9):

        colormap\_image = colormapArray(data4[:, :, i], colors)

        cv2.imwrite(f"vis4\_{i}.png", cv2.cvtColor(colormap\_image, cv2.COLOR\_RGB2BGR))

    print("Saved colormap applied images for mysterydata4.npy")