```
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
    """
    Y = X * 10 + 100
    return Y


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
    """
    Z = X + 10 * Y
    return Z


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
    """
    Z = X * Y - 10
    return Z


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?

    """
    Z = X @ Y
    return Z


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() !
    """
    Z = X.astype(dtype = np.int32)
    return Z


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])

    """
    Z = X.astype(dtype = np.float32) / Y.astype(dtype = np.float32)
    return Z


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 np.reshape(X, (-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.zeros((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 @ 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) @ 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
    """
    # inner product = dot product
    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
    """
    norm = np.sqrt(np.sum(v ** 2))
    return norm


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
    """
    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))
```

```
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!
    """
    eigen_value, eigen_vector = np.linalg.eig(X)
    return eigen_vector[:,np.argmin(eigen_value)]


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.maximum(X, 0)


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
```

```
    """
    return (R @ X.T).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
    """
    X = np.ones((N,N))
    X[:5, :], X[-5:, :], X[:, :5], X[:, -5:] = 0, 0, 0, 0
    return X


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
    """
    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:
```

```
    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
    """
    return (X - X.mean(axis = 1, keepdims = True))/X.std(axis = 1, keepdims = True)


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
    """

    norm = np.sum((q - k) ** 2, axis = 1, keepdims = True)
    return np.sum(np.exp(- norm) * 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.
    """

    C = np.array([X.mean(axis = 0) for X in Xs])
    c_norm = np.sum(C ** 2, axis = 1, keepdims = True)
    R = c_norm + c_norm.T - 2 * (C @ C.T)
    return np.sqrt(np.maximum(R, 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 = ||x||^2 + ||y||^2 - 2x^T y

    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.
        """

        x_norm = np.sum(X ** 2, axis = 1, keepdims = True)
        return np.sqrt(np.maximum((x_norm + x_norm.T - 2 * (X @ X.T)), 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_norm = np.sum(X ** 2, axis = 1, keepdims = True)
        y_norm = np.sum(Y ** 2, axis = 1, keepdims = True)
        return np.sqrt(np.maximum((x_norm + y_norm.T - 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
        """
        return np.argmax(q @ V.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
        """
        return np.linalg.lstsq(X, y, rcond = 1)[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
    """

    return X[:, :-1] / np.expand_dims(X[:, -1], axis = 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
    """
    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)
```

```
#!/usr/bin/python

import os
import numpy as np
import matplotlib.pyplot as plt
import cv2
import pdb


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
    """

    H, W = X.shape
    N = colors.shape[0]

    vmin = np.nanmin(X)
    vmax = np.nanmax(X)

    if not np.any(np.isfinite(X)):
        result_img = np.zeros((H, W, 3), dtype=np.uint8)
        return result_img

    if not np.isfinite(vmin) or not np.isfinite(vmax) or vmax == vmin:
        base = (np.clip(colors[0], 0, 1) * 255).astype(np.uint8)
        return np.tile(base, (H, W, 1))

    norm = (X - vmin) / (vmax - vmin)
    norm = np.where(np.isfinite(norm), norm, 0.0)
    norm = np.clip(norm, 0.0, 1.0)

    idx = np.minimum((norm * (N - 1)).astype(np.int64), N - 1)
    rgb = colors[idx]
    result_img = np.clip(rgb * 255.0, 0, 255).astype(np.uint8)
    return result_img


if __name__ == "__main__":
    colors = np.load("mysterydata/colors.npy")
    data = np.load("mysterydata/mysterydata.npy")


    mystery_data4 = np.load("mysterydata/mysterydata4.npy")
    # print(f"Shape mystery data 4 {mystery_data4.shape}")

    for i in range(9):
        mystery4_result_img = colormapArray(mystery_data4[:, :, i], colors)
        plt.imsave("3_4/vis_%d.png" % i, mystery4_result_img, cmap='plasma')

    # pdb.set_trace()
```

16-820: Advanced Computer Vision

HW-0

Akanksha Singal (asingal2)

September 3, 2025

Numpy Intro

```
(cv) akanksha21008@iras-hub:~/16820_advanced_computer_vision/hw0/numpy$ python run.py --allwarmups
Running w1
Running w2
Running w3
Running w4
Running w5
Running w6
Running w7
Running w8
Running w9
Running w10
Running w11
Running w12
Running w13
Running w14
Running w15
Running w16
Running w17
Running w18
Running w19
Running w20
Ran warmup tests
20/20 = 100.0
```

Figure 1: Warmups output

```
(cv) akanksha21008@iras-hub:~/16820_advanced_computer_vision/hw0/numpy$ python run.py --alltests
Running t1
Running t2
Running t3
Running t4
Running t5
Running t6
Running t7
Running t8
Running t9
Running t10
Running t11
Running t12
Running t13
Running t14
Running t15
Running t16
Running t17
Running t18
Running t19
Running t20
Ran all tests
20/20 = 100.0
```

Figure 2: Test output

Data Visualisation

Section 3.1

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> X = np.load("mysterydata/mysterydata2.npy")
>>> X.shape
(512, 512, 9)
>>> for i in range(9):
...     plt.imsave("output_3.1/vis_%d.png" % i, X[:, :, i])
...
```

Figure 3: Commands for 3.1

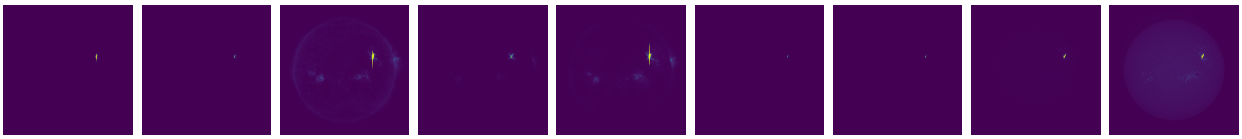

Figure 4: Output without correction

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> X = np.load("mysterydata/mysterydata2.npy")
>>> X.shape
(512, 512, 9)
>>> X = np.log1p(X)
>>> for i in range(9):
...     plt.imsave("corrected_output_3.1/vis_%d.png" % i, X[:, :, i])
...
```

Figure 5: Commands for 3.1 corrected images

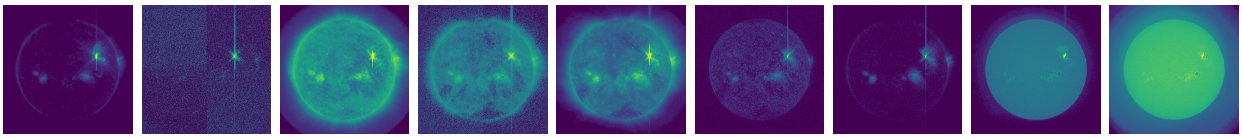

Figure 6: Output with correction

Section 3.2

```
>>> X = np.load("mysterydata/mysterydata3.npy")
>>> X = np.log1p(X)
>>> for i in range(9):
...     plt.imsave("output_3.2/vis_%d.png" % i, X[:, :, i], vmin = np.nanmin(X), vmax = np.nanmax(X))
...
>>>
```

Figure 7: Commands for 3.2

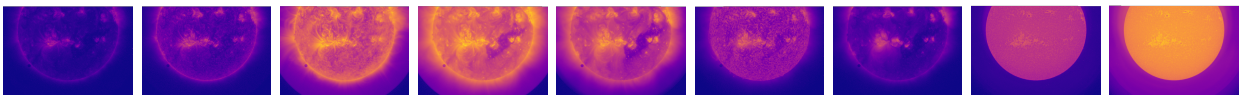

Figure 8: Result for section 3.2

Section 3.4

Figure 9: Result for section 3.4