CECS 229 Programming Assignment #5

Due Date:

Sunday, 4/9 @ 11:59 PM

Extended until 4/11 @ 11:59 PM

Submission Instructions:

To receive credit for this assignment you must submit the to CodePost a file named `pa5.py` by the due date:

Objectives:

1. Define a matrix data structure with relevant matrix operations.
2. Understand the role of matrices in simple image processing applications.

Problem 1.

Implement a class `Matrix` that creates matrix objects with attributes

1. `colsp` -column space of the `Matrix` object, as a list of columns (also lists)
2. `rowsp` -row space of the `Matrix` object, as a list of rows (also lists)

The constructor takes a list of rows as an argument, and constructs the column space from this rowspace. If a list is not provided, the parameter defaults to an empty list.

You must implement the following methods in the `Matrix` class:

Setters

- `set_col(self, j, u)` - changes the j-th column to be the list `u`. If `u` is not the same length as the existing columns, then the method raises a `ValueError` with the message `Incompatible column length.`
- `set_row(self,i, v)` - changes the i-th row to be the list `v`. If `v` is not the same length as the existing rows, then method raises a `ValueError` with the message `Incompatible row length.`
- `set_entry(self,i, j, x)` - changes the existing a_{ij} entry in the matrix to `x`.

Getters

- `get_col(self, j)` - returns the j-th column as a list.
- `get_row(self, i)` - returns the i-th row as a list `v`.
- `get_entry(self, i, j)` - returns the existing a_{ij} entry in the matrix.

- `col_space(self)` - returns the *list* of vectors that make up the column space of the matrix object
- `row_space(self)` - returns the *list* of vectors that make up the row space of the matrix object
- `get_diag(self, k)` - returns the k-th diagonal of a matrix where $k = 0$ returns the main diagonal, $k > 0$ returns the diagonal beginning at $a_{1(k+1)}$, and $k < 0$ returns the diagonal beginning at $a_{(-k+1)1}$. e.g. `get_diag(1)` for an $n \times n$ matrix returns [$a_{12}, a_{23}, a_{34}, \ldots, a_{(n-1)n}$]
- `__str__(self)` - returns a formatted string representing the matrix entries as

$$\begin{matrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{matrix}$$

Overloaded operators

In addition to the methods above, the `Matrix` class must also overload the `+`, `-`, and `*` operators to support:

1. `Matrix + Matrix` addition
2. `Matrix - Matrix` subtraction
3. `Matrix * scalar` multiplication
4. `Matrix * Matrix` multiplication
5. `Matrix * Vec` multiplication
6. `scalar * Matrix` multiplication

In [1]:
```
from Vec import Vec

class Matrix:

    def __init__(self, rowsp):
        self.rowsp = rowsp
        self.colsp = self._construct_cols(rowsp)

    def _construct_cols(self, rowsp):
        colsp = []
        for i in range(len(rowsp[0])):
            col = []
            for j in range(len(rowsp)):
                col.append(rowsp[j][i])
            colsp.append(col)
        return colsp

    def __add__(self, other):
        # check if matrix dimensions are valid
        if len(self.rowsp) != len(other.rowsp) or len(self.colsp) != len(other.
            raise ValueError("ERROR: Product is undefined.")
        else:
            newRowSp = []
            # iterate through the rows of the matrices
            for i in range(len(self.rowsp)):
                newRow = []
                sum = 0
                for j in range(len(self.rowsp[0])):
                    # add the components of self and other together
                    sum = self.rowsp[i][j] + other.rowsp[i][j]
                    newRow.append(sum)
                newRowSp.append(newRow)
        return Matrix(newRowSp)

    def __sub__(self, other):
        # check if matrix dimensions are valid
        if len(self.rowsp) != len(other.rowsp) or len(self.colsp) != len(other.
            raise ValueError("ERROR: Product is undefined.")
        else:
            newRowSp = []
            # iterate through the rows of the matrices
            for i in range(len(self.rowsp)):
                newRow = []
                sum = 0
                for j in range(len(self.rowsp[0])):
                    # subtract the components of self and other together
                    sum = self.rowsp[i][j] - other.rowsp[i][j]
                    newRow.append(sum)
                newRowSp.append(newRow)
        return Matrix(newRowSp)

    def __mul__(self, other):
        """
        Overrides * operation
        Implementation of MATRIX-SCALAR, MATRIX-MATRIX, and MATRIX-VECTOR Multi
        """
        if type(other) == float or type(other) == int:
            newRowSp = []
            for i in range(len(self.rowsp)):
                newRow = []
                product = 0
                for j in range(len(self.rowsp[0])):
                    # multiply each component by the scalar
                    product = self.rowsp[i][j] * other
                    # add that new product to new row
                    newRow.append(product)
                # add that new row to new row space
                newRowSp.append(newRow)
            return Matrix(newRowSp)
        elif type(other) == Matrix:
            # we know that mxn and nxp is valid.... so we should check their si
            if len(self.colsp) != len(other.rowsp):
                raise ValueError("ERROR: Product is undefined.")
            else:
                newRowSp = []
                for i in range(len(self.rowsp)):
                    newRow = []
                    for j in range(len(other.colsp)):
                        # we can use the rows of A and the columns of B - dot p
                        product = Vec(self.rowsp[i]) * Vec(other.colsp[j])
                        newRow.append(product)
                    # after we get each component for a row, we add it to the r
                    newRowSp.append(newRow)
                return Matrix(newRowSp)
        elif type(other) == Vec:
            if len(self.colsp) != len(other.elements):
                raise ValueError("ERROR: Incompatible dimensions.")
            else:
                newVec = []
                for i in range(len(self.rowsp)):
                    # dot product other and i-th row of self
                    product = Vec(self.rowsp[i]) * other
                    # add the new row to the new row space
                    newVec.append(product)
            return Vec(newVec)
        else:
            print("ERROR: Unsupported Type.")
        return

    def __rmul__(self, other):
        """
        Implementation of scalar=matrix multiplication
        """
        if type(other) == float or type(other) == int:
            newRowSp = []
            for i in range(len(self.rowsp)):
                newRow = []
                product = 0
                for j in range(len(self.rowsp[0])):
                    # multiply each component by the scalar
                    product = self.rowsp[i][j] * other
                    # add that new product to new row
                    newRow.append(product)
                # add that new row to new row space
                newRowSp.append(newRow)
            return Matrix(newRowSp)
        else:
            print("ERROR: Unsupported Type.")
        return


    # SETTERS

    def set_col(self, j, u):
        """changes column j to the u list"""
        # checks if column is the correct length
        if len(u) != len(self.colsp[0]):
            raise ValueError("ERROR: Incompatible column length.")
        else:
            self.colsp[j-1] = u  # Updating the column
            # update the row space
            newRowSp = []
            for i in range(len(self.colsp[0])):
                newRow = []
                for j in range(len(self.colsp)):
                    newRow.append(self.colsp[j][i])
                newRowSp.append(newRow)
            self.rowsp = newRowSp

    def set_row(self, i, v):
        """changes row i to the v list"""
        # checks if row is the correct length
        if len(v) != len(self.rowsp[0]):
            raise ValueError("ERROR: Incompatible row length.")
        else:
            self.rowsp[i-1] = v  # Updating the row
            # update column space
            self.colsp = self._construct_cols(self.rowsp)

    def set_entry(self, i, j, x):
        """changes ij-th entry in the matrix to x"""
        self.rowsp[i-1][j-1] = x # changes the entry
        self.colsp = self._construct_cols(self.rowsp) # updates the column spac


    # GETTERS

    def get_col(self, j):
        """returns the j-th column as a list."""
        return self.col_space()[j-1]

    def get_row(self, i):
        """returns the i-th row as a list v"""
        return self.row_space()[i-1]

    def get_entry(self, i, j):
        """returns the existing ij-th entry in the matrix"""
        return self.row_space()[i-1][j-1]

    def col_space(self):
        """returns the list of vectors that make up the column space of the mat
        return self.colsp

    def row_space(self):
        """returns the list of vectors that make up the row space of the matrix
        return self.rowsp

    def get_diag(self, k):
        """returns diagonal of a matrix. If k = 0, then the original diagonal i
        If k > 0, then returns diagonal starting at A 1(k+1)
        If k < 0, then returns diagonal starting at A (-k+1)1"""
        diagonal = []
        if k >= 0:
            for i in range(min(len(self.rowsp), len(self.rowsp[0]) - k)):
                diagonal.append(self.rowsp[i][i + k])
        else:
            for i in range(min(len(self.rowsp)+k, len(self.rowsp[0]))):
                diagonal.append(self.rowsp[i + abs(k)][i])
        return diagonal

    def __str__(self):
        """prints the rows and columns in matrix form """
        mat_str = ""
        for row in self.rowsp:
            mat_str += str(row) + "\n"
        return mat_str

    def __eq__(self, other):
        """overloads the == operator to return True if
        two Matrix objects have the same row space and column space"""
        return self.row_space() == other.row_space() and self.col_space() == ot

    def __req__(self, other):
        """overloads the == operator to return True if
        two Matrix objects have the same row space and column space"""
        return self.row_space() == other.row_space() and self.col_space() == ot
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/var/folders/sq/ffpvrx7d22n_5fsxxhkr9bh40000gn/T/ipykernel_90132/2268446372.py
in <module>
----> 1 from Vec import Vec
      2 
      3 class Matrix:
      4 
      5     def __init__(self, rowsp):

ModuleNotFoundError: No module named 'Vec'
```

In [3]:
```
# B = Matrix([ [1, 2, 3, 4], [0, 1, 2, 3], [-1, 0, 1, 2], [-2, -1, 2, 3]])
# A = Matrix([ [2, 0], [0, 2], [0, 0], [0, 0]])

# print("Matrix A:")
# print(A)
# print()

# print("Matrix B:")
# print(B)
```

Tester Cell for `get_diag()`

In [ ]:
```
# B = Matrix([ [1, 2, 3, 4], [0, 1, 2, 3], [-1, 0, 1, 2], [-2, -1, 2, 3]])
# print("Matrix:")
# print(B)

# print("Main diagonal:",B.get_diag(0))
# print("Expected: [1, 1, 1, 3]")
# print()
# print("Diagonal at k = -1:", B.get_diag(-1))
# print("Expected: [0, 0, 2]")
# print()
# print("Diagonal at k = -2:", B.get_diag(-2))
# print("Expected: [-1, -1]")
# print()
# print("Diagonal at k = -3:", B.get_diag(-3))
# print("Expected: [-2]")
# print()
# print("Diagonal at k = 1:", B.get_diag(1))
# print("Expected: [2, 2, 2]")
# print()
# print("Diagonal at k = 2:", B.get_diag(2))
# print("Expected: [3, 3]")
# print()
# print("Diagonal at k = 3:", B.get_diag(3))
# print("Expected: [4]")
```

Tester Cell for

- `row_space()`
- `col_space()`
- `get_row()`
- `get_col()`
- `set_row()`
- `set_col()`

In [ ]:
```
# A = Matrix([[1, 2, 3], [4, 5, 6]])
# print("Original Row Space:", A.row_space())
# print("Original Column Space:", A.col_space())
# print("Original Matrix:")
# print(A)
# print()


# A.set_row(1, [10, 20, 30])
# print("Modification #1")
# print("Row Space after modification:", A.row_space())
# print("Column Space after modification:", A.col_space())
# print("Modified Matrix:")
# print(A)
# print()

# A.set_col(2, [20, 50])
# print("Modification #2")
# print("Row Space after modification:", A.row_space())
# print("Column Space after modification:", A.col_space())
# print("Modified Matrix:")
# print(A)
# print()

# A.set_row(2, [40, 50, 6])
# print("Modification #3")
# print("Row Space after modification:", A.row_space())
# print("Column Space after modification:", A.col_space())
# print("Modified Matrix:")
# print(A)
# print()

# A.set_entry(2, 3, 60)
# print("Modification #4")
# print("Row Space after modification:", A.row_space())
# print("Column Space after modification:", A.col_space())
# print("Modified Matrix:")
# print(A)
# print()


# print("The 2nd row is:", A.get_row(2))
# print("The 3rd column is:", A.get_col(3))
# print()


# print("Modification #5")
# A.set_row(2, [40, 50])
# A.set_col(2, [30, 4, 1])
# print(A)
```

Expected Output

```
Original Row Space: [[1, 2, 3], [4, 5, 6]]
Original Column Space: [[1, 4], [2, 5], [3, 6]]
Original Matrix:
[1, 2, 3]
[4, 5, 6]
```

```
Modification #1
Row Space after modification: [[10, 20, 30], [4, 5, 6]]
Column Space after modification: [[10, 4], [20, 5], [30, 6]]
Modified Matrix: Set row 1 to [10, 20, 30]
[10, 20, 30]
[4, 5, 6]


Modification #2
Row Space after modification: [[10, 20, 30], [4, 50, 6]]
Column Space after modification: [[10, 4], [20, 50], [30, 6]]
Modified Matrix: Set col 2 to [20, 50]
[10, 20, 30]
[4, 50, 6]


Modification #3
Row Space after modification: [[10, 20, 30], [40, 50, 6]]
Column Space after modification: [[10, 40], [20, 50], [30, 6]]
Modified Matrix: Set row 2 to [40, 50, 6]
[10, 20, 30]
[40, 50, 6]


Modification #4
Row Space after modification: [[10, 20, 30], [40, 50, 60]]
Column Space after modification: [[10, 40], [20, 50], [30, 60]]
Modified Matrix: Set row 2 col 3 to 60
[10, 20, 30]
[40, 50, 60]


The 2nd row is: [40, 50, 60]
The 3rd column is: [30, 60]

Modification #5
---------------------------------------------------------------
------------
ValueError                                Traceback (most
recent call last)
~\AppData\Local\Temp/ipykernel_9756/1966277524.py in <module>
     48 #--------- MODIFICATION 5 ---------------#
     49 print("Modification #5")
---> 50 A.setRow(2, [40, 50])
     51 A.setCol(2, [30, 4, 1])
     52 print("Modified Matrix: Set row 2 to [40, 50] and col 2
to [30, 4, 1]")

~\AppData\Local\Temp/ipykernel_9756/2205165582.py in
setRow(self, i, v)
    159         """Sets the i-th row to be the list v"""
    160         if len(v) != len(self.Rowsp[0]):
--> 161             raise ValueError("ERROR: Incompatible row
length.")
    162         else:
    163             self.Rowsp[i-1] = v  # Updating the row

ValueError: ERROR: Incompatible row length.`
```

Tester cell for +, -, *

In [ ]:

```
# """-----------------------------------TESTER CELL-----------------------------
# "TESTING OPERATOR + "

# A = Matrix([[1, 2],[3, 4],[5, 6]])
# B = Matrix([[1, 2],[1, 2]])
# C = Matrix([[10, 20],[30, 40],[50, 60]])

# # P = A + B # dimension mismatch
# Q = A + C

# print("Matrix A")
# print(A)
# print()

# print("Matrix C")
# print(C)
# print()

# print("Matrix Q = A + C")
# print(Q)
# print()

# "TESTING OPERATOR * "
# # TESTING SCALAR-MATRIX MULTIPLICATION
# T = -0.5 * B
# print("Matrix B")
# print(B)
# print()

# print("Matrix T = -0.5 * B")
# print(T)
# print()


# # TESTING MATRIX-MATRIX MULTIPLICATION
# U = A * B
# print("Matrix U = A * B")
# print(U)
# print()


# # TESTING MATRIX-VECTOR MULTIPLICATION
# x = Vec([0, 1])  # Vec object
# b = A * x   # b is a Vec data type
# print("Vector b = A * x")
# print(b)
```

$$\begin{bmatrix}1 & 2\\3 & 4\\5 & 6\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\cdot 0 + 2\cdot 1\\3\cdot 0 + 4\cdot 1\\5\cdot 0 + 6\cdot 1\end{bmatrix} = \begin{bmatrix}2\\4\\6\end{bmatrix}$$

Expected Output:

```
Matrix A
[1, 2]
[3, 4]
[5, 6]


Matrix C
[10, 20]
[30, 40]
[50, 60]


Matrix Q = A + C
[11, 22]
[33, 44]
[55, 66]


Matrix R = A - C
[-9, -18]
[-27, -36]
[-45, -54]


Matrix B
[1, 2]
[1, 2]


Matrix T = -0.5 * B
[-0.5, -1.0]
[-0.5, -1.0]


Matrix A:
[1, 2]
[3, 4]
[5, 6]


Row-space of A:
[[1, 2], [3, 4], [5, 6]]

Column-space of A:
```

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

Matrix B:
[1, 2]
[1, 2]


Row-space of B:
[[1, 2], [1, 2]]

Column-space of B:
[[1, 1], [2, 2]]

Matrix U = A * B
[3, 6]
[7, 14]
[11, 22]


Vector b = A * x
[0, -2, -4]
```

Extra-Credit

- **Worth:** 5% extra-credit applied to midterm
- **To Receive Credit You Must:**
  1. Submit your work on this Jupyter NB to the appropriate Dropbox folder by Sunday, 4/9 @11:59 PM
  2. Submit your completed video to the appropriate Dropbox folder by Sunday, 4/9 @11:59 PM
  3. Demo your work to me during OH in order to receive the extra-credit. The last day you may demo is Thursday, 4/20.
- **No partial credit. This is an all-or-nothing EC opportunity**

Background:

One of my favorite bands is "Alt-J". Take a look at the music video for their song, "Matilda" at https://www.youtube.com/watch?v=Q06wFUi5OM8. The faces you see morphing into one another are the faces of the four members who were in the band at the time. In this exercise you will explore how a simplified version of this "morphing effect" can be achieved. In our simplified morphing effect, we will fade one face into another.

First, keep in mind that a video is just a collection of several still images displayed with a speed fast enough to make the change from one image to another imperceptible to the human eye.

To make the discussion simpler, suppose the images are grayscale pictures. We can represent a grayscale picture with $m \times n$ pixels as a matrix $P_{m\times n}$ where each entry $p_{ij} \in \{0, 1, \ldots, 255\}$ is the intensity value of the pixel at location (i, j), [*The intensity values range from 0 (black) to 255 (white)*]. We are able to prove that the set of matrices $\mathbb{S} = \{P_{m\times n} | p_{ij} \in \mathbb{Z}_{256}\}$ is a vector space, under addition and scalar multiplication defined as below:

Let $P, Q \in \mathbb{S}$, and $\alpha \in \mathbb{R}, 0 \le \alpha \le 1$

- Addition: $P + Q = [a_{ij}] \text{ where } a_{ij} = \begin{cases} p_{ij} + q_{ij} & \text{if the sum is 255 or less} \\ 255 & \text{otherwise} \end{cases}$
- Scalar Multiplication: $\alpha P = [a_{ij}] \text{ where } a_{ij} = \begin{cases} \alpha p_{ij} & \text{if the product is 255 or less} \\ 255 & \text{otherwise} \end{cases}$

Hence, given two "image-matrices" $P_1, P_2 \in \mathbb{S}$, we can form convex combinations of these two elements with the confidence that the resulting matrices will be in $\mathbb{S}$, and thus, still represent images; i.e., if $\alpha_1, \alpha_2 \in \mathbb{R}$ such that $\alpha_1 + \alpha_2 = 1$, then

$\alpha_1 P_1 + \alpha_2 P_2 \in \mathbb{S}$ and represents a new image.

Think: what would the image corresponding to matrix P look like if $P = 0.5P_1 + 0.5P_2$? Since the images P_1 and P_2 make an equal contribution to the intensity of each pixel in P, we can expect the image to look like an equal mix of the two images. e.g. if the two images contain faces in more-or-less the same position, the resulting image should display a face that more-or-less looks like both faces.

What if $P = 0.85P_1 + 0.15P_2$? Then, since most of the intensity in each pixel of P is being contributed by P_1, we can expect the resulting image P to display something that looks more like the first image, P_1, vs the second image, P_2.

Task 1:

1. Download the `png` and `image` modules. The `image` module contains the methods
    - `file2image()` - Reads an image into a list of lists of pixel values (triples with values representing the three intensities in the RGB color channels). e.g. `[[(1, 2, 3), (1, 2, 3), (1, 2, 3)],[(1, 2, 3), (1, 2, 3), (1, 2, 3)], [(1, 2, 3), (1, 2, 3), (1, 2, 3)]]` would be representing an image with 3×3 pixels.
    - `image2file()` - Writes an image in list of lists format to a file.
2. Use the functions listed above to implement:
    - `img2matrix(filename)` - creates and returns a `Matrix` object with the image data returned by `file2image()` from the module `image`. The parameter

`filename` is a string data type specifying the location of the image you wish to use. If the image is not grayscale, you must convert it to grayscale prior to creating the `Matrix` object. You can do so using the functions `isgray()` and `color2gray()`, also found in the `image` module.

- `matrix2img(image_matrix, path)` - creates a png file out of a `Matrix` object. You may want to use the function `image2file()` from the `image` module.

In [ ]:
```
# import image

# def image2matrix(filename):
#     """
#     takes a png file and returns a Matrix object of the pixels
#     INPUT: filename - the path and filename of the png file
#     OUTPUT: a Matrix object with dimensions m x n, assuming the png file has
#     """
#     #FIXME: a single line of code should go here
#     if #FIXME: the image is not gray:
#         image_data = #FIXME: make the image grayscale
#     return #FIXME


# def matrix2image(img_matrix, path):
#     """
#     returns a png file created using the Matrix object, img_matrix
#     INPUT:
#         * img_matrix - a Matrix object where img_matrix[i][j] is the intensit
#         * path - the location and name under which to save the created png fi
#     OUTPUT:
#         * a png file
#     """
#     pass
```

In [ ]:
```
# """------------------TESTER FOR FUNCTIONS png2graymatrix() AND graymatrix2png
# M = image2matrix("img11.png")  # matrix for img11.png
# F = image2matrix("img02.png")  # matrix for img02.png
# C = 0.40 * M + 0.60*F   # convex combo: first image contributes 40% of its in

# matrix2image(C, "mixedfaces.png")  # converting the matrix to png named mixed
```

Task 2

1. Download and extract the zip folder `faces.zip`. In it, you will find the images of 20 faces.
2. Use the functions you implemented in Task 1 to implement a function called `mix(img1, img2)` that generates a set of 101 images. These images must be the result of taking convex combinations of two given images. In particular, you should begin by combining the two images so that the 1st/101 picture looks completely like `img1`. Then, modify the scalars of the combination so that the mixed image is the sum of a percentage of the intensities for each image. For example, the 2nd/101 picture would be a mixture of 99% of the first image's intensity mixed with 1% of the second

image's intensity. The 51th/100 picture will look like both images equally mixed together (50/50). The 76th picture will looks like 25% of the first image's intensity mixed with 75% of the second image's intensity, and the 101st/101 picture looks like `img2` only.

3. Call your `mix()` function on two images of your choice found in the `faces.zip` folder. The resulting images should give the illusion that one face is morphing into the other.

In [ ]:
```
# def mix(img1, img2):
#     """
#     generates a set of 101 images that results from the convex mixing the giv
#     INPUT:
#         - img1: string of path + name of first image
#         - img2: string of path + name of second image

#     OUTPUT: None (images are saved to the path given to matrix2image())
#     """
#     # todo
#     pass
```

Task 3

Use the function `make_video()` below to create a video out of the 101 images you generated in Task 2. You will need to have installed the package `opencv` in order for the function to work. I recommend that you complete this entire task in a separate IDE such as PyCharm, where it is easier to install packages.

In [ ]:
```
# import cv2
# import os

# def make_video(images, outvid=None, fps=5, size=None, is_color=True, format='
#     """
#     Create a video from a list of images.

#     @param      outvid      output video
#     @param      images      list of images to use in the video
#     @param      fps         frame per second
#     @param      size        size of each frame
#     @param      is_color    color
#     @param      format      see http://www.fourcc.org/codecs.php
#     @return                 see http://opencv-python-tutroals.readthedocs.org

#     By default, the video will have the size of the first image.
#     It will resize every image to this size before adding them to the video.
#     """

#     fourcc = cv2.VideoWriter_fourcc(*format)
#     vid = None
#     for image in images:
#         if not os.path.exists(image):
#             raise FileNotFoundError(image)
#         img = cv2.imread(image)
#         if vid is None:
#             if size is None:
```

```
#                 size = img.shape[1], img.shape[0]
#             vid = cv2.VideoWriter(outvid, fourcc, float(fps), size, is_color)
#         if size[0] != img.shape[1] and size[1] != img.shape[0]:
#             img = cv2.resize(img, size)
#         vid.write(img)
#     vid.release()
#     return vid
```

Sample Usage:

```
img_path = "C:\\Users\\kapiv\\Documents\\CECS 229\\CA
#5\\faces\\"
vid_path = "C:\\Users\\kapiv\\Documents\\CECS 229\\CA
#5\\male_faces.avi"

images = []  # Initializing empty list of image paths
for i in range(15):  # adding images male00.png - male14.png to
the list
    if i < 10:
        file = f"{img_path}male0{i}.png"
    else:
        file = f"{img_path}male{i}.png"
    print("Adding:", file)
    images.append(file)


make_video(images, vid_path, format = "mp4v")  # creating video
```