pa4

March 18, 2023

1 CECS 229 Programming Assignment #4

Due Date: Sunday, 3/19 @ 11:59 PM

Submission Instructions: To receive credit for this assignment, you must submit to CodePost this file converted to a Python script named `pa4.py`

Objectives:

1. Apply vector operations to translate, scale, and rotate a set of points representing an image.
2. Perform various operations with or on vectors: addition, subtraction, dot product, norm.

1.0.1 Helpful Websites

- norm and implementation in python
- more norms and app to python
- more vector norms
- complex numbers in python
- book of complex vectors, UMD
- general vector spaces over a field, USC
- rotating a complex number, stack exchange
- rotating complex numbers, khan academy

Needed Import Statements

[72]:
```
# MAKE SURE TO RUN THIS CELL BEFORE RUNNING ANY TESTER CELLS
"""
In order for the import statements below to work, you must download and save
the plotting.py and image.py files to the same folder where this file is
 ↪located.
"""
import math
from plotting import plot
import image
```

Problem 1: Create a function `translate(S, z0)` that translates the points in the input set S by $z_0 = a_0 + b_0 i$. The function should satisfy the following:

1. INPUT:
    - S - set S
    - z0 - complex number
2. OUTPUT:
    - a set consisting of points in S translated by z_0

[73]:
```
def translate(S, z0):
    """
    translates the complex numbers of set S by z0
    INPUT:
        * S - set of complex numbers
        * z0 - complex number
    OUT:
        * a set consisting of points in S translated by z0
    """
    # NOTE: accessing real and imag parts of z0, z0.real and z0.imag
    # create a set of new translated points
    translated = set()
    # add z0 to each complex number in the set
    for num in S:
        new = num + z0 # the new translated number
        translated.add(new)
    return translated
```

[74]:
```
# """
# TESTER CELL #1 FOR translate()

# WORKS
# """

# S = {2 + 2j, 3 + 2j , 1.75 + 1j, 2 + 1j, 2.25 + 1j, 2.5 + 1j, 2.75 + 1j, 3 +␣
↪1j, 3.25 + 1j}


# print("ORGINAL VALUES:", S)

# plot([(S, 'black')], 10, title = "Original Values") # original values will be␣
↪plotted in black

# T1 = translate(S, -3-2j)  # values translated by -3-2i will be plotted in red
# T2 = translate(S, 3-2j)   # values translated by 3-2i will be plotted in green
# T3 = translate(S, -3+2j)  # values translated by -3+2i will be plotted in␣
↪orange
# T4 = translate(S, 3+2j)   # values translated by 3+2i will be plotted in blue

# print("SHIFT LEFT 3, DOWN 2:", T1)
```

```
# print("Expected: {0j, (-0.25-1j), (-1+0j), -1j, (-1-1j), (-1.25-1j), (-0.
↪75-1j), (0.25-1j), (-0.5-1j)}\n")
# print("\nSHIFT RIGHT 3, DOWN 2:", T2)
# print("Expected: {(6-1j), (6.25-1j), (5+0j), (6+0j), (5-1j), (5.75-1j), (5.
↪5-1j), (4.75-1j), (5.25-1j)}\n")
# print("\nSHIFT LEFT 3, UP 2:", T3)
# print("Expected: {(-1+3j), (-1.25+3j), (-0.75+3j), (-1+4j), (-0.5+3j), 4j,␣
↪(-0.25+3j), 3j, (0.25+3j)}\n")
# print("\nSHIFT RIGHT 3, UP 2:", T4)
# print("Expected: {(4.75+3j), (5.5+3j), (5.25+3j), (5+3j), (5+4j), (6+4j), (5.
↪75+3j), (6+3j), (6.25+3j)}\n")

# # plotting original values against translated values
# plot([(S, 'black'), (T1, 'red'), (T2, 'green'), (T3, 'orange'), (T4,␣
↪'blue')], 10, title = "Original + Shifted Values")
```

[75]:
```
# """
# TESTER CELL #2 FOR translate()
# WORKS
# """
# img = image.file2image('img01.png')
# gray_img = image.color2gray(img)
# complex_img = image.gray2complex(gray_img)
# translated_img = translate(complex_img, -200 + 0j)
# plot([(complex_img, 'black')], 200, title = "Original Image")
# plot([(translated_img, 'black')], 200, title = "Translated Image")
```

Problem 2: Create a function `scale(S, k)` that scales the points in the input set S by a factor of k:

1. INPUT:
   - `S` - set S
   - `k` - positive float, raises ValueError if k $\leq$ 0.
2. OUTPUT:
   - a set consisting of points in S scaled by k.

[99]:
```
def scale(S, k):
    """
    scales the complex numbers of set S by k.
    INPUT:
        * S - set of complex numbers
        * k - positive float, raises ValueError if k <= 0
    OUT:
        * T - set consisting of points in S scaled by k

```

```
    """
    # check precondition
    if k <= 0:
        raise ValueError("K must be positive")
    # create a set of new scaled points
    T = set()
    # scale each complex number in the set by k
    for num in S:
        new = k*num # the new scaled complex number
        T.add(new)
    return T
```

[100]:
```
# """
# TESTER CELL #1 FOR scale()
# WORKS
# """
# sets = [S, T1, T2, T3, T4]
# scaled_sets = [scale(A, 2) for A in sets]
# for i in range(len(scaled_sets)):
#     print("Original Set:", sets[i])
#     print("After Scaling by 2:", scaled_sets[i], "\n")

# plot([(S, 'black')], 10, title = "Original Values")
# plot_data = list(zip(scaled_sets, ['black', 'red', 'green', 'orange',␣
 ↪'blue']))
# plot(plot_data, 10, title = "Scaled by 2") #second parameter affects window␣
 ↪size
```

[101]:
```
# """
# TESTER CELL #2 FOR scale()
# WORKS
# """
# img = image.file2image('img01.png')
# gray_img = image.color2gray(img)
# complex_img = image.gray2complex(gray_img)
# scaled_img = scale(complex_img, 1.5)
# plot([(complex_img, 'black')], 200, title = "Original Image")
# plot([(scaled_img, 'black')], 200, title = "Image Scaled 1.5x")
```

Problem 3: Create a function `rotate(S, theta)` that rotates the points in the input set S by θ radians:

1. INPUT:
    - `S` - set S
    - `theta` - float. If negative, the rotation is clockwise. If positive the rotation is counter-clockwise. If zero, no rotation.

2. OUT:
   - a set consisting of points in S rotated by θ

[116]:
```
def rotate(S, theta):
    """
    rotates the complex numbers of set S by theta radians.
    INPUT:
        * S - set of complex numbers
        * theta - float. If negative, the rotation is clockwise. If positive␣
↪the rotation is counterclockwise. If zero, no rotation.
    OUT:
        * a set consisting of points in S rotated by theta radians
    """
    # create a set of new rotated points
    rotated = set()
    # rotate each complex num in the set
    for num in S:
        x = num.real * math.cos(theta) - num.imag * math.sin(theta) # real part␣
↪for the rotation
        y = num.real * math.sin(theta) + num.imag * math.cos(theta) # imag part␣
↪for the rotation
        new = complex(x, y) # new rotated points
        rotated.add(new)
    return rotated
```

[118]:
```
# """
# TESTER CELL #1 FOR rotate()

# WORKS
# """
# rotated_sets = [rotate(A, math.pi/2) for A in sets]
# for i in range(len(rotated_sets)):
#     print("Original Set:", sets[i])
#     print("After Scaling by 2:", rotated_sets[i], "\n")

# plot_data_rot = list(zip(rotated_sets, ['black', 'red', 'green', 'orange',␣
↪'blue']))
# plot(plot_data_rot, 10, title = "Rotated by 90 degrees") #second parameter␣
↪affects window size

# rotated_sets_2 = [rotate(A, -1*math.pi/2) for A in sets]
# for i in range(len(rotated_sets_2)):
#     print("Original Set:", sets[i])
#     print("After Scaling by 2:", rotated_sets_2[i], "\n")
```

```
# plot([(S, 'black'), (T1, 'red'), (T2, 'green'), (T3, 'orange'), (T4,␣
↪'blue')], 10, title = "Original Values")

# plot_data_rot_2 = list(zip(rotated_sets_2, ['black', 'red', 'green',␣
↪'orange', 'blue']))
# plot(plot_data_rot_2, 10, title = "Rotated by -90 degrees") #second parameter␣
↪affects window size
```

[119]:
```
# """
# TESTER CELL #2 FOR rotate()

# WORKS
# """
# img = image.file2image('img01.png')
# gray_img = image.color2gray(img)
# complex_img = image.gray2complex(gray_img)
# rotated_img = rotate(complex_img, -1*math.pi/2)
# plot([(complex_img, 'black')], 200, title = "Original Image")
# plot([(rotated_img, 'black')], 200, title = "Image Rotated by -90 degrees")
```

[120]:
```
# """
# Full image transformation = rotation + scaling + translation

# WORKS
# """
# img = image.file2image('img01.png')
# gray_img = image.color2gray(img)
# complex_img = image.gray2complex(gray_img)

# rotated_img = rotate(complex_img, -1*math.pi/2)
# scaled_img = scale(rotated_img, 1.5)
# translated_img = translate(scaled_img, -125 + 150j)

# plot([(complex_img, 'black')], 200, title = "Original Image")
# plot([(translated_img, 'black')], 200, title = "Transformed Image")
```

Problem 4: Finish the implementation of class `Vec` which instantiates row-vector objects with defined operations of addition, subtraction, scalar multiplication, and dot product. In addition, `Vec` class overloads the Python built-in function `abs()` so that when it is called on a `Vec` object, it returns the Euclidean norm of the vector.

[156]:
```
class Vec:
    def __init__(self, contents = []):
        """
        Constructor defaults to empty vector
```

```
        INPUT: list of elements to initialize a vector object, defaults to␣
↪empty list
        """
        self.elements = contents
        return


    def __abs__(self):
        """
        Overloads the built-in function abs(v)
        returns the Euclidean norm of vector v
        """
        sum = 0
        # iterate over the elements of contents
        for element in self.elements:
            sum += element ** 2
        sum = sum ** 0.5 # square root of sum
        return sum


    def __add__(self, other):
        """Overloads the + operator to support Vec + Vec
         raises ValueError if vectors are not same length
        """
        sum = []
        # check the precondition
        if len(self.elements) != len(other.elements):
            raise ValueError("Vectors are not same length")
        # add the elements of each vector
        for sNum, oNum in zip(self.elements, other.elements):
            sum.append(sNum + oNum)
        sum = Vec(sum) # convert array to Vec object?
        return sum


    def __sub__(self, other):
        """
        Overloads the - operator to support Vec - Vec
        Raises a ValueError if the lengths of both Vec objects are not the same
        """
        difference = []
        # check the precondition
        if len(self.elements) != len(other.elements):
            raise ValueError("Vectors are not same length")
        # subtract the elements of each vector
        for sNum, oNum in zip(self.elements, other.elements):
            difference.append(sNum - oNum)
```

```
        difference = Vec(difference) # convert array to Vec object?
        return difference


    def __mul__(self, other):
        """Overloads the * operator to support
            - Vec * Vec (dot product) raises ValueError if vectors are not same␣
↪length in the case of dot product
            - Vec * float (component-wise product)
            - Vec * int (component-wise product)

        """
        if type(other) == Vec: #define dot product
            # check the precondition
            if len(self.elements) != len(other.elements):
                raise ValueError("Vectors are not same length")
            dot = 0
            # iterate over both vectors
            for sNum, oNum in zip(self.elements, other.elements):
                new = sNum * oNum # mult. both
                dot += new # add them to the total
            return dot
        elif type(other) == float or type(other) == int: #scalar-vector␣
↪multiplication
            product = []
            # mult. each component of vec by other (float or int)
            for element in self.elements:
                product.append(other * element) # mult. other and element, then␣
↪add to list
            product = Vec(product) # convert array to Vec object?
            return product


    def __rmul__(self, other):
        """Overloads the * operation to support
            - float * Vec
            - int * Vec
        """
        product = []
        # mult. each component of vec by other (float or int)
        for element in self.elements:
            product.append(other * element) # mult. other and element, then add␣
↪to list
        product = Vec(product) # convert array to Vec object?
        return product
```

```python
    def __str__(self):
        """returns string representation of this Vec object"""
        return str(self.elements) # does NOT need further implementation
```

[169]:

```python
# """----------------------TESTER CELL----------------------"""

# u = Vec([1, 2, 3])
# w = Vec([0, 1, -1])
# v = Vec([0, -3])
# print("u = ", u)
# print("w = ", w)
# print("\nEuclidean norm of u:", abs(u))
# print("Expected:", math.sqrt(sum([ui**2 for ui in u.elements])))
# print("\nEuclidean norm of v:", abs(v))
# print("Expected: 3.0")
# print("\nu left scalar multiplication by 2:", 2*u)
# print("Expected: [2, 4, 6]")
# print("\nw right scalar multiplication by -1:", w * -1)
# print("Expected: [0, -1, 1]")
# print("\nVector addition:", u + w)
# print("Expected: [1, 3, 2]")
# print("\nVector addition:", u - w)
# print("Expected: [1, 1, 4]")
# print("\nDot product:", w*u)
# print("Expected: -1")


# try:
#     print("\nDot product:", v*u)
#     print("If this line prints, you forgot to raise a ValueError for taking␣
 ↪the dot product of vectors of different lengths")
# except:
#     print("If this line prints, an error was correctly raised.")
```