EE2703 - Week 5

Santhosh S P ee21b119

March 8, 2023

1 Importing and installing the requirements

[ ]:
```
%matplotlib inline
from IPython.display import HTML
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.animation import FuncAnimation
```

- `%matplotlib inline` is used to render plots in the notebook.
- `matplotlib.pyplot` and `numpy` are used for plotting and numerical computations.
- `FuncAnimation` from `matplotlib.animation` is used to create the animation.
- `HTML` module from `Ipython.display` and `ffmpeg` is used to convert the animation into a video and make it playable in the notebook.
- `ipython` can be installed using `pip` by `pip3 install ipython` and `ffmpeg` can be installed inside the python environment using `sudo apt install ffmpeg`.

2 Setting the plot, init() and update() functions

2.1 The init() and morph() function

- First we create an empty plot and initialise the variables.
- `init()` function initialises the graph by setting its x-axis and y-axis limits.
- `morph(x1, y1, x2, y2, alpha)` function can return any value `xm` in the range from `x2` to `x1` and any value `ym` from `y2` to `y1` based on different values of `alpha`. So on increasing the value of `alpha` from 0 to 1, the values change smoothly from `x2` and `y2` to `x1` and `y1`.

2.2 The update() function

- Inside the `update(frame)` function, to make 10 transitions, we create 10 conditional statments after dividing the value of frame into 10 ranges. The value of `frame` goes from 0 to 1 (as defined using the `FuncAnimation() function later`), therefore we define the 10 ranges as 0 to 0.1, 0.1 to 0.2 and so on till 0.9 to 1 for each transition.
- For each transition, we use the `morph()` function with results of `n_gon(t,n)` function as arguments.
- `n_gon(t,n)` function (defined in the next cell), returns a x-coordinate array and a y-coordinate array for a general n-sided regular polygon with vertices on the unit circle. Here the argument, `t` refers to an array of uniformly spaced numbers in 0 to 2π, and `n` refers to the number of sides of the regular polygon.

- The argument `alpha` for `morph()` function in each transition, is modified in each case to make sure that it goes from 0 to 1, with each passing frame.
- The outputs of the `morph()` function are stored into the global variables `xdata` and `ydata` and added to the plot.

[ ]:
```
##creating an empty plot
fig, ax = plt.subplots()
xdata, ydata = [], []
ln, = ax.plot([], [], 'r')



#setting the axis limits inside the init() function
def init():
    ax.set_xlim(-1.2, 1.2)
    ax.set_ylim(-1.2, 1.2)
    return ln,

#defining a linear morph function that goes from x2 to x1 and y2 to y1 as alpha␣
 ↪goes from
#0 to 1
def morph(x1, y1, x2, y2, alpha):
    xm = alpha * x1 + (1-alpha)*x2
    ym = alpha * y1 + (1-alpha)*y2
    return xm, ym

def update(frame):

        #to make sure that we're accessing the global varibles xdata and ydata
        global xdata,ydata

        #to make 10 transistion, we divide the values of frame into 10 regions

        #transition from triangle to square
        if frame>0 and frame<=0.1:
            xdata, ydata = morph(n_gon(t,4)[0], n_gon(t,4)[1], n_gon(t,3)[0],␣
 ↪n_gon(t,3)[1], 10*(frame))

        #transition from square to pentagon
        if frame>0.1 and frame<=0.2:
            xdata, ydata = morph(n_gon(t,5)[0], n_gon(t,5)[1], n_gon(t,4)[0],␣
 ↪n_gon(t,4)[1], 10*(frame-0.1))

        #transition from pentagon to hexagon
        if frame>0.2 and frame<=0.3:
            xdata, ydata = morph(n_gon(t,6)[0], n_gon(t,6)[1], n_gon(t,5)[0],␣
 ↪n_gon(t,5)[1], 10*(frame-0.2))
```

```
        #transition from hexagon to heptagon
        if frame>0.3 and frame<=0.4:
            xdata, ydata = morph(n_gon(t,7)[0], n_gon(t,7)[1], n_gon(t,6)[0],␣
↪n_gon(t,6)[1], 10*(frame-0.3))

        #transition from heptagon to octagon
        if frame>0.4 and frame<=0.5:
            xdata, ydata = morph(n_gon(t,8)[0], n_gon(t,8)[1], n_gon(t,7)[0],␣
↪n_gon(t,7)[1], 10*(frame-0.4))


        ##reverse transitions, going from an octagon back to triagle

        #transition from octagon to heptagon
        if frame>0.5 and frame<=0.6:
            xdata, ydata = morph(n_gon(t,7)[0], n_gon(t,7)[1], n_gon(t,8)[0],␣
↪n_gon(t,8)[1], 10*(frame-0.5))

        #transition from heptagon to hexagon
        if frame>0.6 and frame<=0.7:
            xdata, ydata = morph(n_gon(t,6)[0], n_gon(t,6)[1], n_gon(t,7)[0],␣
↪n_gon(t,7)[1], 10*(frame-0.6))

        #transition from hexagon to pentagon
        if frame>0.7 and frame<=0.8:
            xdata, ydata = morph(n_gon(t,5)[0], n_gon(t,5)[1], n_gon(t,6)[0],␣
↪n_gon(t,6)[1], 10*(frame-0.7))

        #transition from pentagon to square
        if frame>0.8 and frame<=0.9:
            xdata, ydata = morph(n_gon(t,4)[0], n_gon(t,4)[1], n_gon(t,5)[0],␣
↪n_gon(t,5)[1], 10*(frame-0.8))

        #transition from square back to triangle
        if frame>0.9 and frame<=1:
            xdata, ydata = morph(n_gon(t,3)[0], n_gon(t,3)[1], n_gon(t,4)[0],␣
↪n_gon(t,4)[1], 10*(frame-0.9))

        ln.set_data(xdata, ydata)
        return ln,
```

3 Mathematics behind the animation

- The function `r(theta, a, b, c)` converts a straight line $ax + by + c = 0$ to polar form $r(\theta)$ according to

$$r(\theta) = \left| \frac{c}{\sqrt{a^2 + b^2}} \sec\left(\theta - \arctan\frac{a}{b}\right) \right|$$

- The function `linefinder(p1,p2)` takes two inputs `p1`, `p2` (two tuples representing coordinates of two points) and returns a, b and c for the line $ax + by + c = 0$ that contains the two given points. We use the fact that points (a_1, b_1) and (a_2, b_2) lie on the line $(b_1 - b_2)x + (a_2 - a_1)y + a_1(b_2 - b_1) - b_1(a_2 - a_1) = 0$
- The function `sidemaker(point_list)` generates arrays of x-coordinates and y-coordinates after inputting the vertices of the polygon as a list of tuples. It chooses all consecutive entries in the `point_list` and finds the parameters a, b and c of the line joining them.
- If the regular polygon has n sides, the array `t` (containing uniformly spaced values from 0 to 2π) is sliced into n parts. For the slice corresponding to a particular side, the function `r(theta, a, b, c)` is evaluated with `theta` as the corresponding slice of `t` and `a`, `b`, `c` as the values found from the `linefinder(p1,p2)` function with consecutive points.
- The array $r(\theta).\cos\theta$ is added to the array with x-coordinates, and $r(\theta).\sin\theta$ is added to the array with y-coordinates. This is done for each side with the corresponding slice of `t` as θ and the coordinate arrays are returned.
- `n_gon(t,n)` function, returns a x-coordinate array and a y-coordinate array for a general n-sided regular polygon with vertices on the unit circle. Here the argument, `t` refers to an array of uniformly spaced numbers in 0 to 2π, and `n` refers to the number of sides of the regular polygon. A array of points is made and stored in `points`, which is fed as the input to the function `sidemaker()` and the coordinate array is returned.

[ ]:
```
#converts ax+by+c=0 to polar form
def r(theta,a,b,c):
    return np.abs(c/(np.sqrt(a**2+b**2)*np.cos(theta-np.arctan(b/a))))

#returns a, b and c for a line ax+by+c=0 containing two points (a1,b1), (a2,b2)
#takes two tuples as input
#a=b1-b2
#b=a2-a1
#c=a1*(b2-b1)-b1*(a2-a1)
def linefinder(p1,p2):
    a1,b1=p1
    a2,b2=p2
    a=b1-b2
    b=a2-a1
    c=a1*(b2-b1)-b1*(a2-a1)
    return a, b, c

#generates arrays of x-coordinates and y-coordinates after inputting the␣
↪vertices of the polygon
```

```
#as a list of tuples
def sidemaker(point_list):

    #creating the arrays
    xp=np.array([])
    yp=np.array([])
    index=0

    #choosing all consecutive entries (except the line joining first and the␣
 ↪last one)
    for p1, p2 in zip(point_list[:-1], point_list[1:]):
        #p1 and p2 are tuples, referring to two points

        #finding out the line joining the point
        a,b,c=linefinder(p1=p1, p2=p2)

        #for each side we add r(theta).cos(theta) for all theta corresponding␣
 ↪to that side in x-coord list
        xp=np.concatenate([xp, r(t[int(index*len(t)/len(point_list)):
 ↪int((index+1)*len(t)/len(point_list))],a,b,c)*np.cos(t[int(index*len(t)/
 ↪len(point_list)):int((index+1)*len(t)/len(point_list))])])

        #for each side we add r(theta).sin(theta) for all theta corresponding␣
 ↪to that side in y-coord list
        yp=np.concatenate([yp, r(t[int(index*len(t)/len(point_list)):
 ↪int((index+1)*len(t)/len(point_list))],a,b,c)*np.sin(t[int(index*len(t)/
 ↪len(point_list)):int((index+1)*len(t)/len(point_list))])])
        index+=1

    #for the last pair (line joining the first and the last point)
    a,b,c=linefinder(p1=point_list[-1], p2=point_list[0])

    #similarly another set of x-coords and y-coords are added
    xp=np.concatenate([xp, r(t[int(index*len(t)/len(point_list)):
 ↪int((index+1)*len(t)/len(point_list))],a,b,c)*np.cos(t[int(index*len(t)/
 ↪len(point_list)):int((index+1)*len(t)/len(point_list))])])
    yp=np.concatenate([yp, r(t[int(index*len(t)/len(point_list)):
 ↪int((index+1)*len(t)/len(point_list))],a,b,c)*np.sin(t[int(index*len(t)/
 ↪len(point_list)):int((index+1)*len(t)/len(point_list))])])
    return xp, yp

#returns a x-coordinate array and a y-coordinate array for a general n-sided␣
 ↪regular polygon
#with vertices on the unit circle and plots it
def n_gon(t, n):
    points=[]
```

```
    for i in range(n):
        points.append((np.cos(2*np.pi*i/n),np.sin(2*np.pi*i/n)))

    xt,yt=sidemaker(point_list=points)
    return xt,yt
```

4 Creating the animation and playing it as a video

- An array `t` is defined to contain 20160 values in the range 0 to 2π. It must be made sure that number of elements in `t` should be a multiple of $LCM(3, 4, 5, 6, 7, 8)$ to make sure that slices of `t` for each side of the polygon are of the same length.
- The animations is created by defining `frames` as an array of 528 values in the range 0 to 1, with the interval betweeen frames being $20ms$.
- The line `HTML(ani.to_html5_video())` converts the FuncAnimation object to a video and makes it playable in the python notebook.
- The animation generated is slightly different from the one given because of a different kind of morphing function leading to a slightly different transition.

[ ]:
```
t=np.linspace(0,2*np.pi, 20160)
ani = FuncAnimation(fig, update, frames=np.linspace(0,1,528),init_func=init,
 ↪blit=True, interval=20, repeat=True)
HTML(ani.to_html5_video())
```

[ ]: <IPython.core.display.HTML object>