Section6

Elliott Ashby

November 21, 2022

1 q1

Using the code as suggested, but with a key difference, only printing the values when t is 1 or t is 2 allows us to easily obtain the required date at the specified times.

```
def f(y, t):
    return -y + 1.0


def odestep(f, y, t, dt):
    return y + dt * f(y, t)


if __name__ == '__main__':
    for dt in [0.2, 0.1, 0.02, 0.01]:
        t = 0
        y = 0
        tf = 2.0
        nsteps = int(tf / dt)
        print('when dt is', dt)
        for i in range(nsteps):
            y = odestep(f, y, t, dt)
            t = (i + 1) * dt
            if t == 1.0 or t == 2.0:
                print(t, y)
```

Since this code will be used for q2 aswell, different values of dt are considered, to find the wanted values at dt is 0.2 simply look at the output below "when dt is 0.2".

when dt is 0.2 and when
RC=1.0, V=0.67232
RC=2.0, V=0.8926258176

2 q2

The above code also prints the values when dt is 0.1, 0.02 and 0.01.

when dt is 0.1 and when
RC=1.0, V=0.6513215599000001
RC=2.0, V=0.8784233454094308

when dt is 0.02 and when
RC=1.0, V=0.6358303199128829
RC=2.0, V=0.867380444105247

when dt is 0.01 and when
RC=1.0, V=0.6339676587267709
RC=2.0, V=0.8660203251420382

3 q3

```
def f(y, t):
    return -y + 1.0


def odestep(f, y, t, dt):
    dA = f(y, t)
    Bpr = y + dt * f(y, t)
    Bpry = f(Bpr, t)
    return y + 0.5 * (dA + Bpry) * dt
```

Here we modify odestep into a form that takes into account the possible change in gradient of the slope without having dt very small. We can do this by taking the average of two gradients like:

$$y(t+\delta t) = y(t) + \frac{1}{2}\left[\left(\frac{dy}{dt}\right)_A + \left(\frac{dy}{dt}\right)_{B'}\right]\delta t$$

4 q4

Using the new system yields:

when dt is 0.2 and when
RC=1.0, V=0.6292601568
RC=2.0, V=0.8625519686640395

5 q5

In order to get more accurate values we can decrease dt until only extremely small changes occur in V. In order to implement this, we can simply make a requirement that the previous value of V has to be less than a certain magnitude, in this case I chose 1.0e-6 arbritrarily.

```
if __name__ == '__main__':
    dt = 0.2
    a = [0.6292601568, 0.6314590151664482]
    b = [0.8625519686640395, 0.8641775424979157]
    t = 0
```

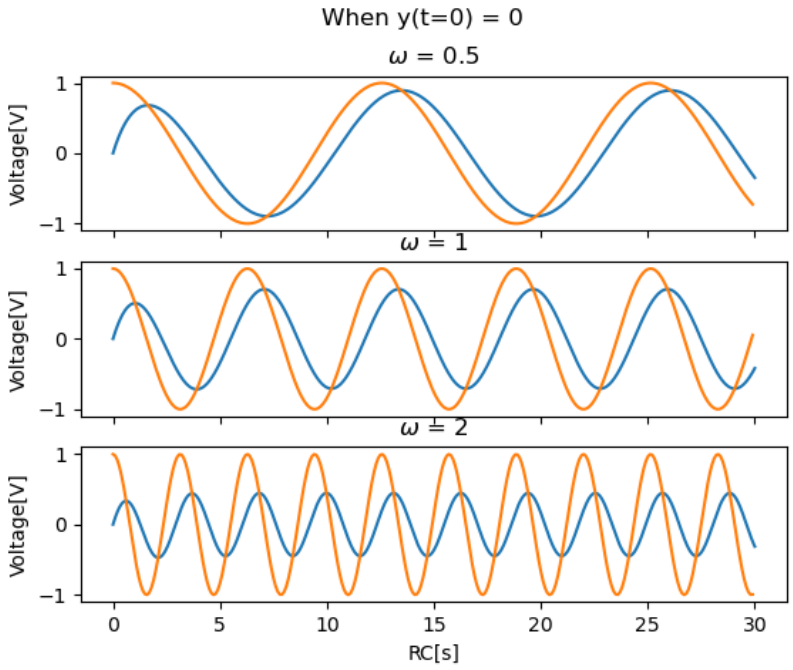
```
tf = 2.0
while abs(a[-1] - a[-2]) > 1.0e-6 and abs(b[-1] - b
    ↪ [-2]) > 1.0e-6:
    y = 0
    nsteps = int(tf / dt)
    print('when dt is', dt)
    for i in range(nsteps):
        y = odestep(f, y, t, dt)
        t = (i + 1) * dt
        if t == 1.0:
            a.append(y)
            print(t, a[-1])
        elif t == 2.0:
            b.append(y)
            print(t, b[-1])
    dt = dt / 2
```

Here, if both of the V values at t = 1 and t = 2 meet the stopping value, no more values are calculated.

6 q6

Here we simply alter the code to include g(t) = cos(t) and up the value of tf, then we can plot $\omega = 0.5$, 1 and 2 along with the input signal.

```
from math import cos
from math import pi
import copy
from numpy import arange
import matplotlib.pyplot as plt
import scipy


def odestep(f, y, t, dt, w):
    dA = f(y, t, w)
    Bpr = y + dt * f(y, t, w)
    Bpry = f(Bpr, t, w)
    return y + 0.5 * (dA + Bpry) * dt


def g(w, t):
    return cos(w * t)
```



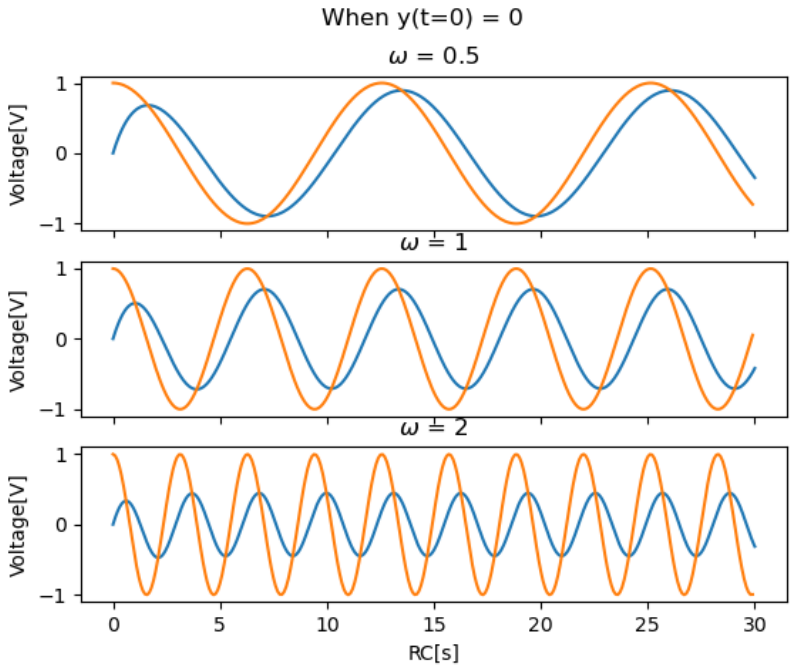
7 q7

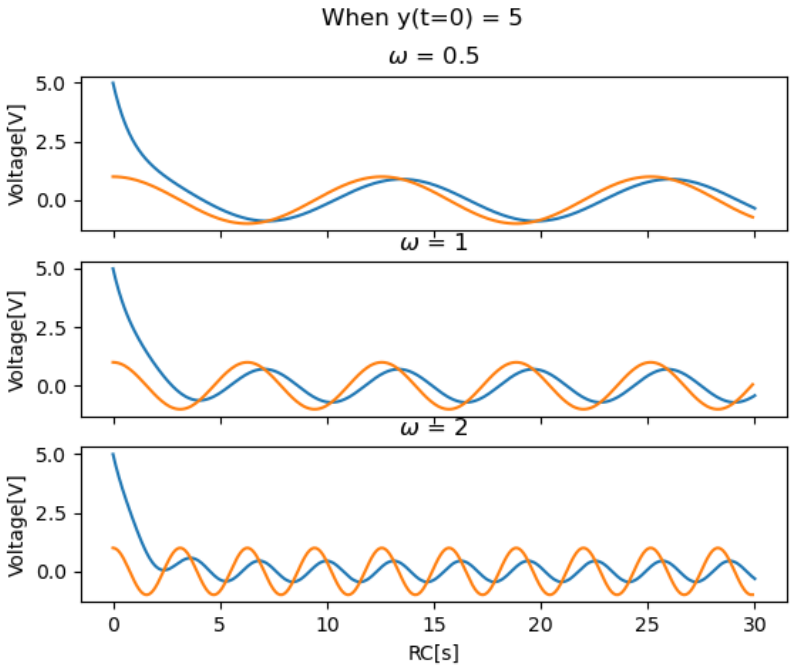
We can make plots for increasing values of y in order to inspect their differences.

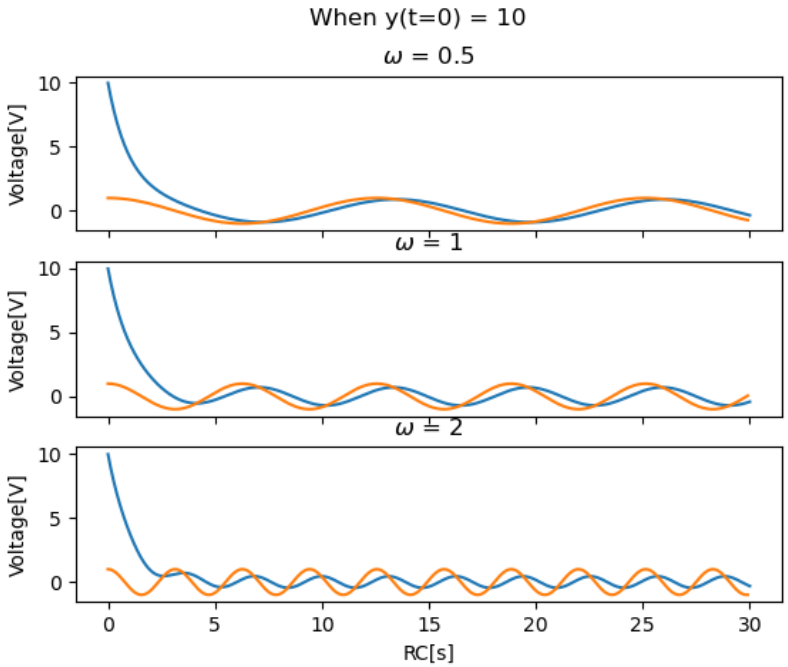
```
def counter():
    n = 0
    while True:
        yield n
        n += 1


if __name__ == '__main__':
    for i in counter():
        if i >= 11:
            break
        data: list = []
        for w in [0.5, 1, 2]:
            t = 0
            y = i
            dt = 0.0015
            startt = copy.copy(i)
            tf = 30.0
            nsteps = int(tf / dt)
            temp1 = []
            temp2 = []
```

```
        for x in range(nsteps):
            y = odestep(f, y, t, dt, w)
            t = (x + 1) * dt
            temp1.append(y)
            temp2.append(t)
        data.append([temp2, temp1])
    fig, (axs0, axs1, axs2) = plt.subplots(3, 1,
        ↪ sharex=True, sharey=True)
    axs0.plot(data[0][0], data[0][1])
    axs0.set_title('$\omega$ = 0.5')
    axs0.plot(arange(0, int(tf), 0.1), [f(0, t, 0.5)
        ↪ for t in arange(0, int(tf), 0.1)])
    axs1.plot(data[1][0], data[1][1])
    axs1.set_title('$\omega$ = 1')
    axs1.plot(arange(0, int(tf), 0.1), [f(0, t, 1)
        ↪ for t in arange(0, int(tf), 0.1)])
    axs2.plot(data[2][0], data[2][1])
    axs2.set_title('$\omega$ = 2')
    axs2.plot(arange(0, int(tf), 0.1), [f(0, t, 2)
        ↪ for t in arange(0, int(tf), 0.1)])
    axs = (axs0, axs1, axs2)
    for ax in axs:
        ax.set(ylabel='Voltage[V]', xlabel='RC[s]')

    fig.suptitle(f'When y(t=0) = {startt}')

    for ax in axs:
        ax.label_outer()
    plt.savefig(f'./q6_6_t{startt}.png')
    plt.clf()
    print(f'when y(t=o) is {startt}')
    print(max(data[0][1]))
    print(max(data[1][1]))
    print(max(data[2][1]))
    print(scipy.signal.correlate(data[0][1], [f(0, t,
        ↪  0.5) for t in arange(0, int(tf), 0.1)])[
        ↪ data[0][1].index(max(data[0][1]))] % 2 * pi
        ↪ )
    print(scipy.signal.correlate(data[1][1], [f(0, t,
        ↪  0.5) for t in arange(0, int(tf), 0.1)])[
        ↪ data[1][1].index(max(data[1][1]))] % 2 * pi
        ↪ )
    print(scipy.signal.correlate(data[2][1], [f(0, t,
        ↪  0.5) for t in arange(0, int(tf), 0.1)])[
        ↪ data[2][1].index(max(data[2][1]))] % 2 * pi
        ↪ )
```

Here we simply count i upwards to 10 (the only reason I used a yield function to play around with generators) Then we generate a new plot of all three ωs while substituting the y value for the value of i. Here are where y = 0, 5 and 10.


When y(t=0) = 0
ω = 0.5
ω = 1
ω = 2
Voltage[V]
RC[s]


When y(t=0) = 5
ω = 0.5
ω = 1
ω = 2
Voltage[V]
RC[s]



We can see here that the y(t=0) value determines the starting value of the curve, meaning, the further away from the average value of the curve is is the longer the curve takes to "settle" into it's predicted cosine pattern.

8 q8

In order to find the amplitude of each value of ω we can simply print the maximum value of the data set. And to find the phase we can cross correlate the input signal and the output signal.

When $\omega = 0.5$
Amplitude = 0.8944271397260981
Phase difference = 1.096944476599115 rads

When $\omega = 1$
Amplitude = 0.7071066610319001
Phase difference = 5.045320129800867 rads

When $\omega = 2$
Amplitude = 0.44721361538056764
Phase difference = 5.42234390436165 rads

This was done like:

```
print(scipy.signal.correlate(data[0][1], [f(0, t,
    ↪  0.5) for t in arange(0, int(tf), 0.1)])[
    ↪ data[0][1].index(max(data[0][1]))] % 2 * pi
    ↪ )
print(scipy.signal.correlate(data[1][1], [f(0, t,
    ↪  0.5) for t in arange(0, int(tf), 0.1)])[
```

```
    ↪ data[1][1].index(max(data[1][1]))] % 2 * pi
    ↪ )
print(scipy.signal.correlate(data[2][1], [f(0, t,
    ↪  0.5) for t in arange(0, int(tf), 0.1)])[
    ↪ data[2][1].index(max(data[2][1]))] % 2 * pi
    ↪ )
```