In [1]:
```
# Lab 0 - Python and Jupyter notebook introduction
# (Alex) Xi Zeng, 1007099373
```

In [136…
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

Warm-up Exercises

Try the following commands on your jupyter notebook or python editor and see what output they produce.

In [137…
```
a = 1 + 5
b = 2
c = a + b
print(a / b)
print(a // b)
print(a - b)
print(a * b)
print(a**b)
```

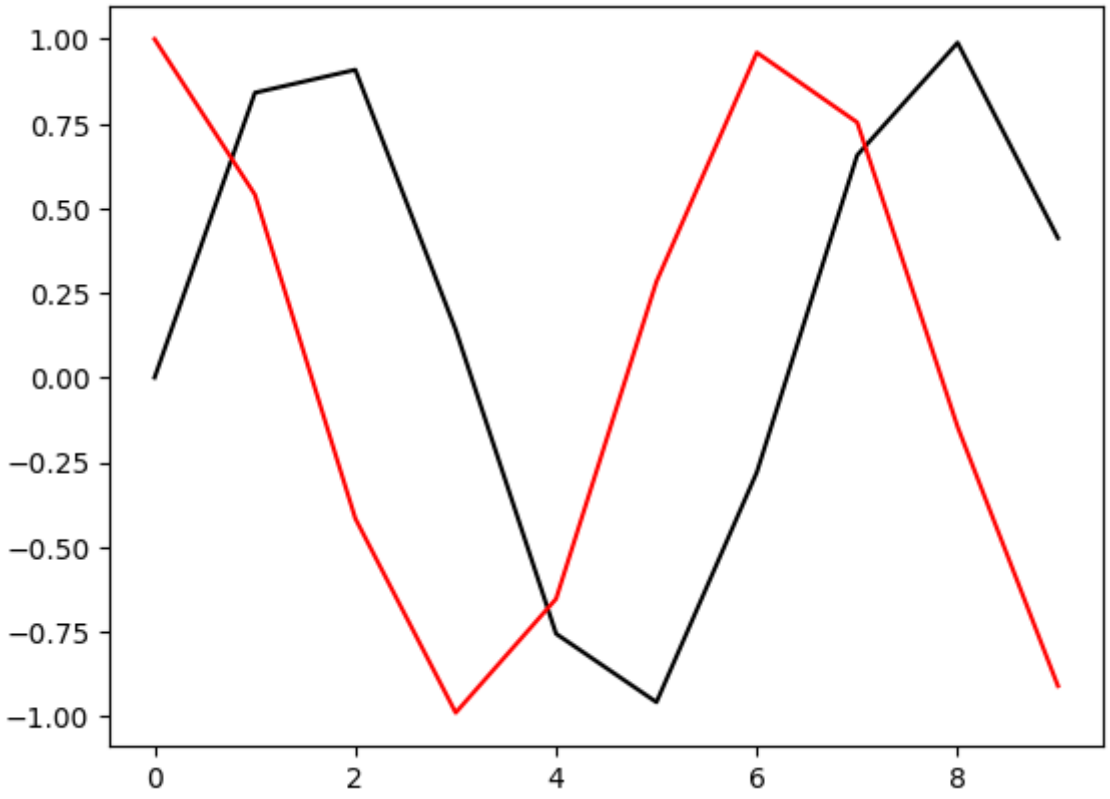
```
3.0
3
4
12
36
```

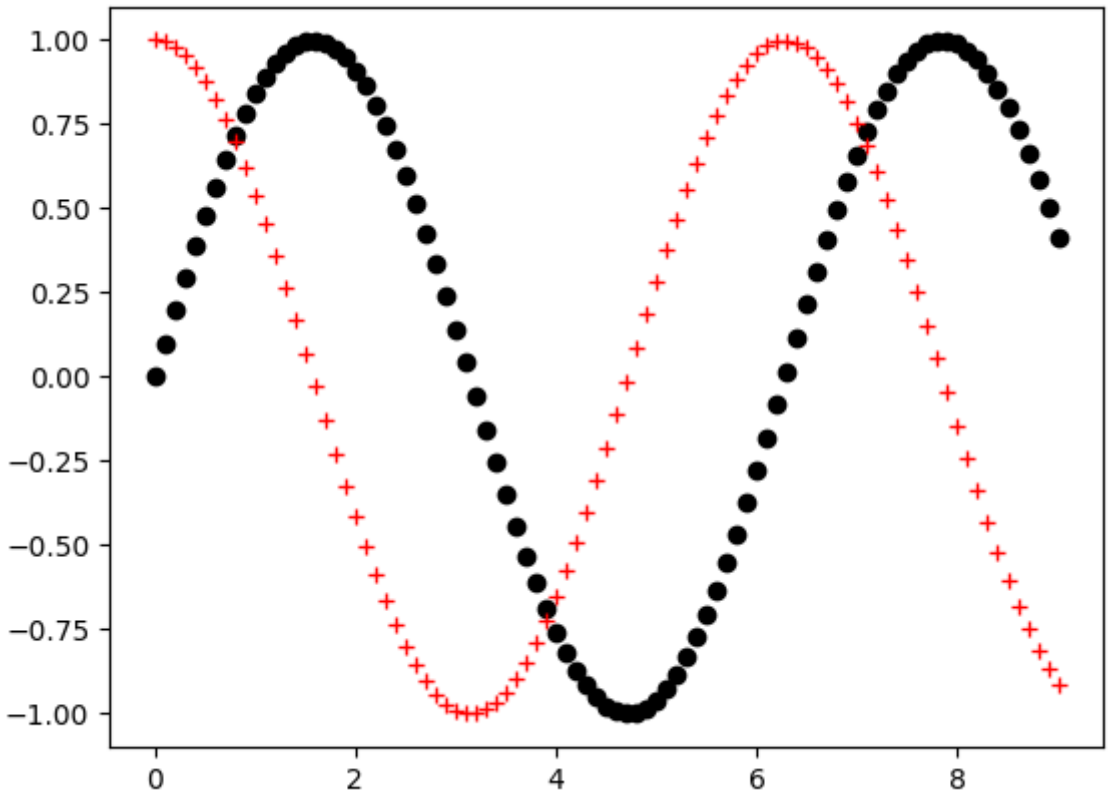
In [138…
```
a = np.array([[3, 1],
              [1, 3]])
b = np.array([[3],
              [5]])
print(a * b)

# physical meaning of dot product: how much do they overlap
print(np.dot(a, b))
print(np.dot(b.T, a))

c = a**(-1.0)
print(c * a)
```

```
[[ 9  3]
 [ 5 15]]
[[14]
 [18]]
[[14 18]]
[[1. 1.]
 [1. 1.]]
```

In [139…
```
t = np.arange(10) # generate set of numbers

g = np.sin(t)
h = np.cos(t)
plt.figure()

plt.plot(t, g, 'k', t, h, 'r');
```

```
t = np.arange(0, 9.1, 0.1)
g = np.sin(t)
h = np.cos(t)
plt.figure()
plt.plot(t, g, 'ok', t, h, '+r');
```





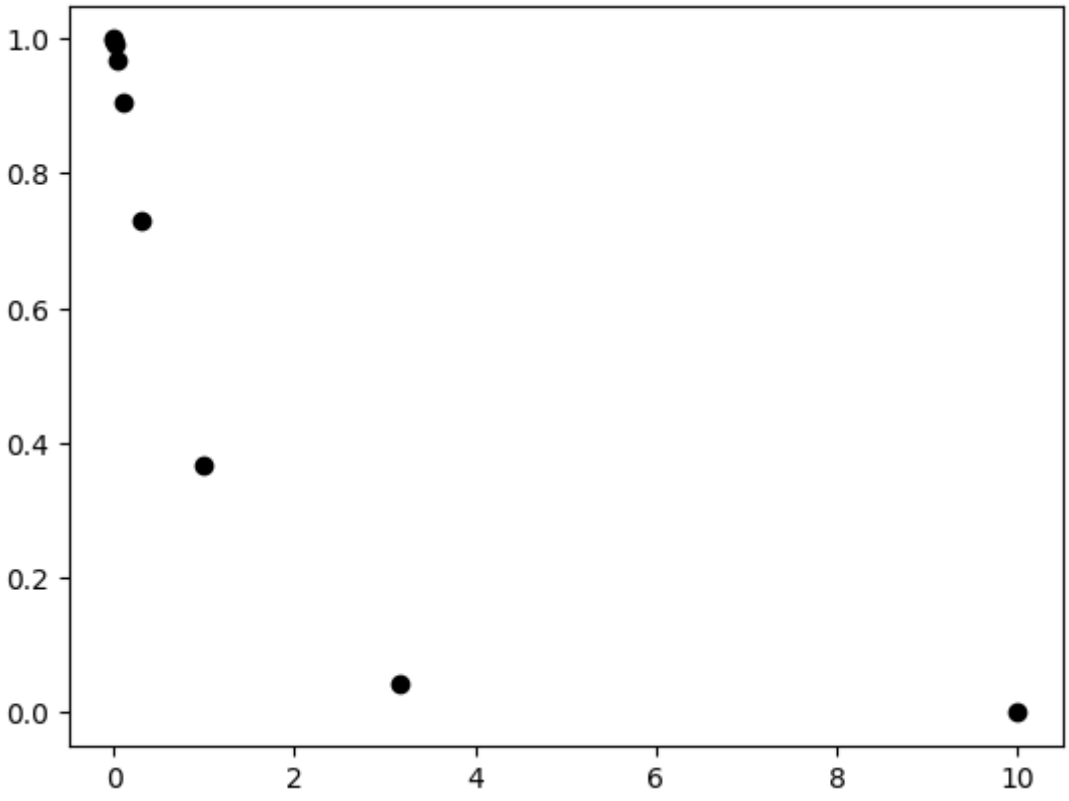
In [140…

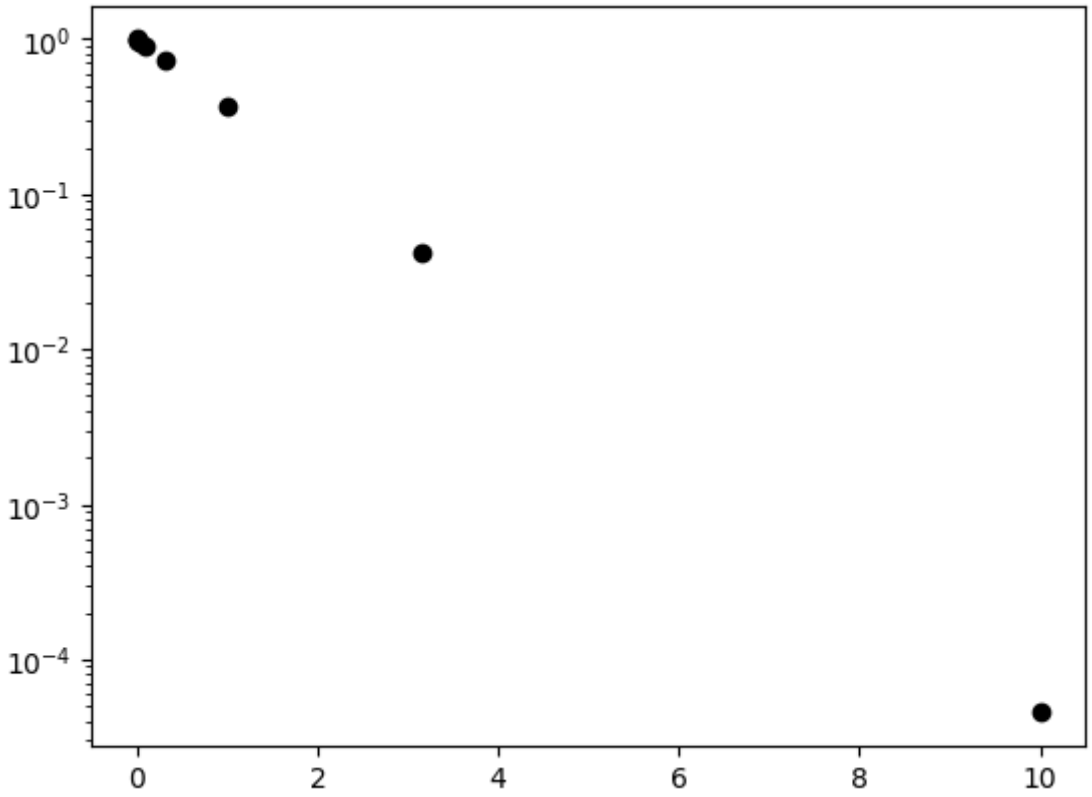
```
#linspace: Return evenly spaced numbers over a specified interval
t = np.linspace(0, 10, 20)
print(t)
t = np.logspace(0.001, 10, 9)
print(t)
t = np.logspace(-3, 1, 9)
print(t)
y = np.exp(-t)

plt.figure()
plt.plot(t, y, 'ok')
plt.figure()
plt.semilogy(t, y, 'ok') # semilogy: Make a plot with log scaling
```

```
[ 0.          0.52631579  1.05263158  1.57894737  2.10526316  2.63157895
  3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.78947368
  6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.94736842
  9.47368421 10.        ]
[1.00230524e+00 1.78186583e+01 3.16774344e+02 5.63151182e+03
 1.00115196e+05 1.77981556e+06 3.16409854e+07 5.62503203e+08
 1.00000000e+10]
[1.00000000e-03 3.16227766e-03 1.00000000e-02 3.16227766e-02
 1.00000000e-01 3.16227766e-01 1.00000000e+00 3.16227766e+00
 1.00000000e+01]
```

Out[140]:

```
[<matplotlib.lines.Line2D at 0x10fea8810>]
```





Integration Function

Here is a more complicated function that computes the integral $y(x)$ with interval dx:

$$c = \int y(x)dx \sim \sum_{i=1}^{N} y_i dx_i.$$

It can deal with both cases of even and uneven sampling.

In [141…

```
def integral(y, dx):
    # function c = integral(y, dx)
    # To numerically calculate integral of vector y with interval dx:
    # c = integral[ y(x) dx]

    # ------ This is a demonstration program ------
    n = len(y) # Get the length of vector y
    nx = len(dx) if np.iterable(dx) else 1 # Check whether or not an object car
    c = 0 # initialize c because we are going to use it
    # dx is a scalar <=> x is equally spaced

    if nx == 1: # '==', equal to, as a condition
        for k in range(1, n):
            c = c + (y[k] + y[k-1]) * dx / 2

    # x is not equally spaced, then length of dx has to be n-1
    elif nx == n-1:
        for k in range(1, n):
            c = c + (y[k] + y[k-1]) * dx[k-1] / 2
    # If nx is not 1 or n-1, display an error messege and terminate program
```

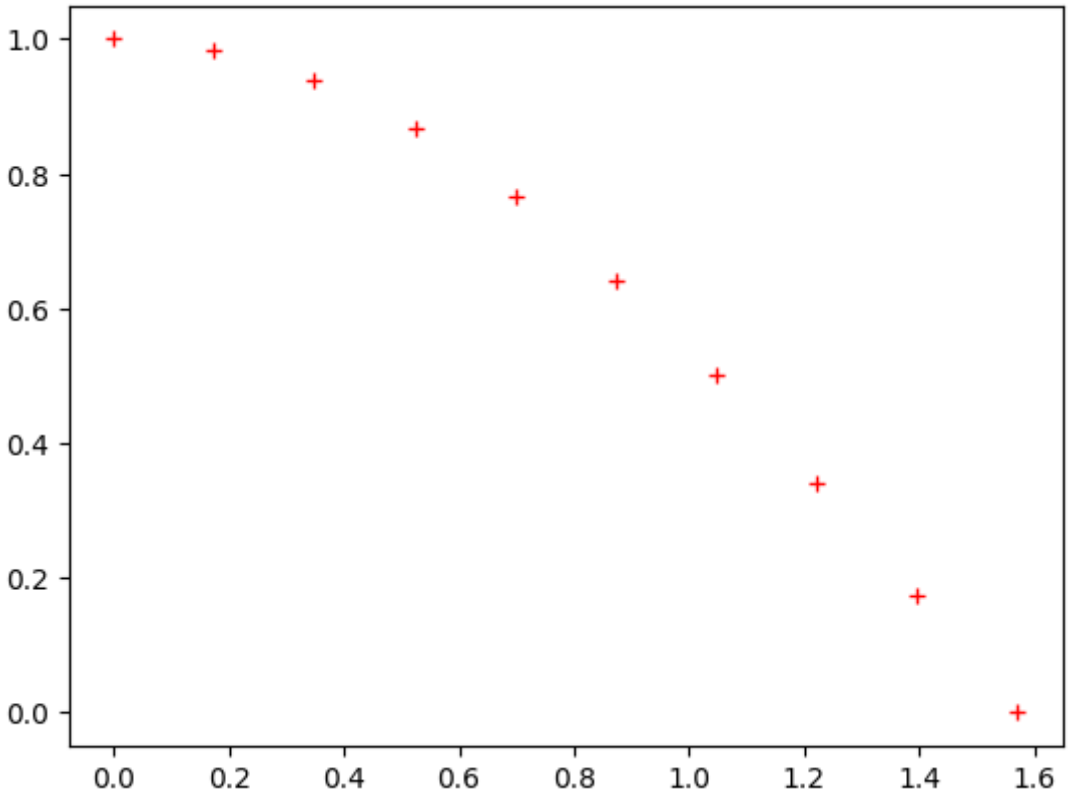
```
    else:
        print('Lengths of y and dx do not match!')
    return c
```

Use this function (`integral`) to compute $\int_0^{\pi/2} \cos(t)dt$ with an evenly sampled time series.

In [142…

```
# number of samples
nt = 10
# generate time vector
t = np.linspace(0, np.pi/2, nt)
print("t:  ", t)

# compute sample interval (evenly sampled, only one number)
dt = t[1] - t[0]
y = np.cos(t)
plt.plot(t, y, 'r+')
c = integral(y, dt)

# print("dt: ", dt)
print("c when nt = 10: ", c)
```

```
t:   [0.         0.17453293 0.34906585 0.52359878 0.6981317  0.87266463
 1.04719755 1.22173048 1.3962634  1.57079633]
c when nt = 10:  0.9974602317917262
```



Part 1 Question

First plot $y(t)$. Is the output c value what you are expecting for $\int_0^{\pi/2} \cos(t)dt$? How can you improve the accuracy of your computation?

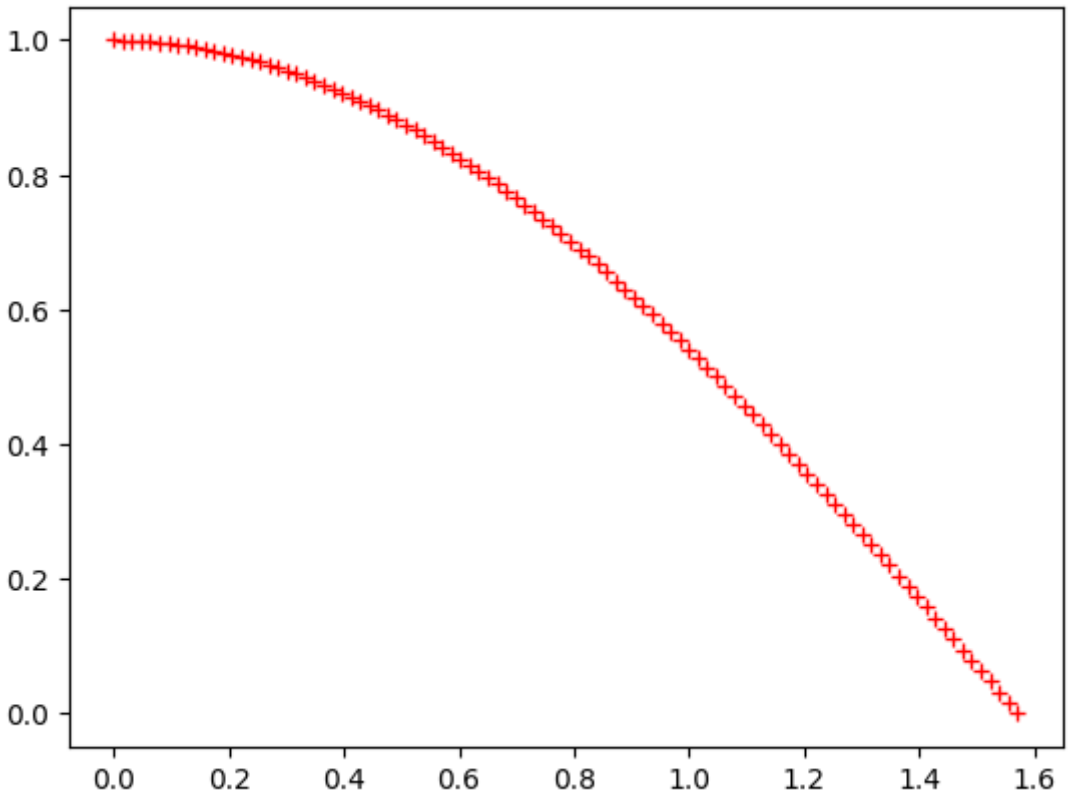
Part 1 Solution

The plot for $\int_0^{\pi/2} \cos(t)dt$ with $nt = 10$ has been plotted above. The expected solution for this integral for this integral is 1, and the above algorithemn outputs 0.9974602317917262, which is 0.2% off from the expected solution. To improve the accuracy, we can try to increase the number of samples. For example, if we change the number of samples, nt, to 100, as shown below, the calculated solution for this integral becomes 0.00002097924. Which is only 0.002% off from the actual solution, now we have a calculated solution that is much closer to the expected solution.

In [143…

```
# number of samples
nt = 100
# generate time vector
t = np.linspace(0, np.pi/2, nt)

# compute sample interval (evenly sampled, only one number)
dt = t[1] - t[0]
y = np.cos(t)
plt.plot(t, y, 'r+')
c = integral(y, dt)
print("c when nt = 100: ", c)
```

```
c when nt = 100:  0.999979020750832
```



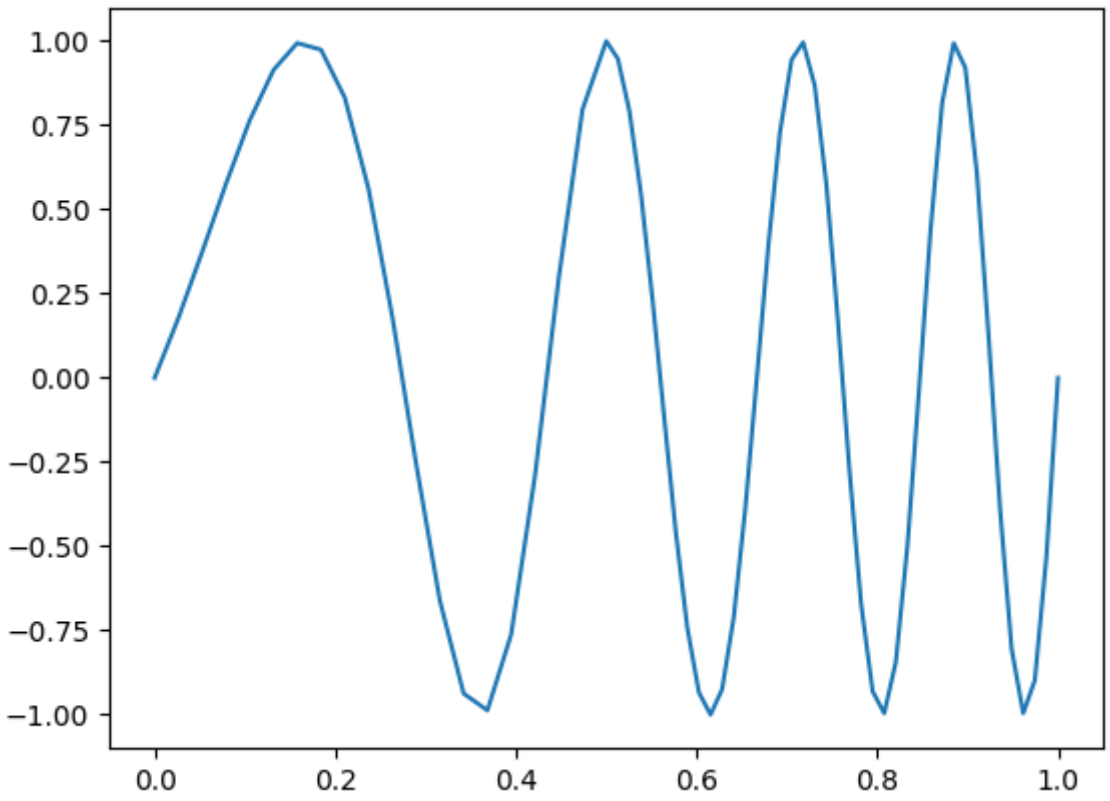
Part 2 Question

For an unevenly spaced time series that depicts $\sin[(2\pi(t+3t^2)]$ (so-called chirp function), compute $\int_0^1 \sin[(2\pi(t+3t^2)]dt$.

In [144…

```
nt = 20
# sampling between [0,0.5]
t1 = np.linspace(0, 0.5, nt)
# double sampling between [0.5,1]
t2 = np.linspace(0.5, 1, 2*nt)
# concatenate time vector
t = np.concatenate((t1[:-1], t2))
# compute y values (f=2t)
y = np.sin(2 * np.pi * (t + 3*t**2))
plt.plot(t, y)
# compute sampling interval vector
dt = t[1:] - t[:-1]
c = integral(y, dt)
print("c when nt = 20: ", c)
```

```
c when nt = 20:  0.08673999688870114
```



Show your plot of $y(t)$ for $nt = 50$. Try different nt values and see how the integral results change. Write a `for` loop around the statements above to try a series of nt values (e.g, 50, 100, 500, 1000, 5000) and generate a plot of $c(nt)$. What value does c converge to after using larger and larger nt? (Please include your modified Python code.)

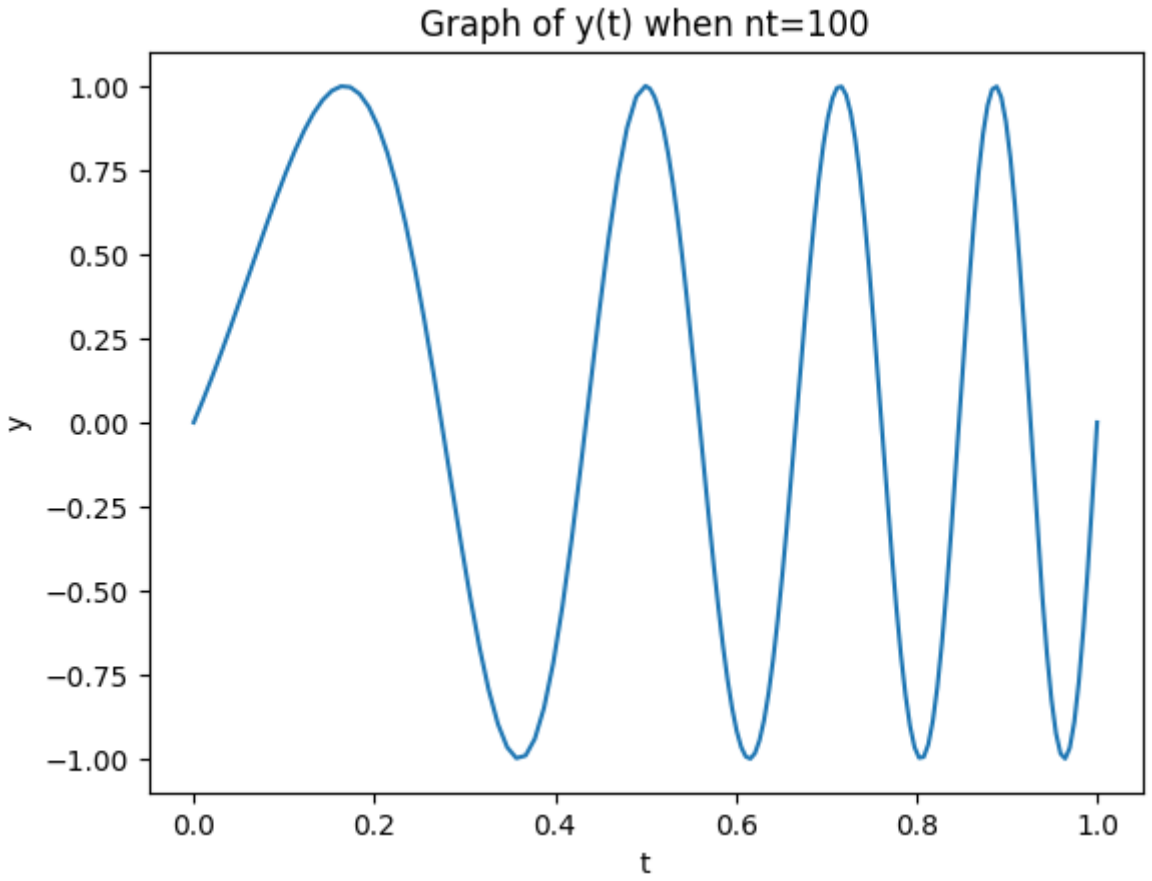
Part 2 Solution:

In [153…

```
nt = 50
# sampling between [0,0.5]
t1 = np.linspace(0, 0.5, nt)
```

```
# double sampling between [0.5,1]
t2 = np.linspace(0.5, 1, 2*nt)
# concatenate time vector
t = np.concatenate((t1[:-1], t2))
# compute y values (f=2t)
y = np.sin(2 * np.pi * (t + 3*t**2))
plt.plot(t, y)
plt.title("Graph of y(t) when nt=100")
plt.xlabel("t")
plt.ylabel("y")
# compute sampling interval vector
dt = t[1:] - t[:-1]
c = integral(y, dt)
print("c when nt = 50: ", c)
```

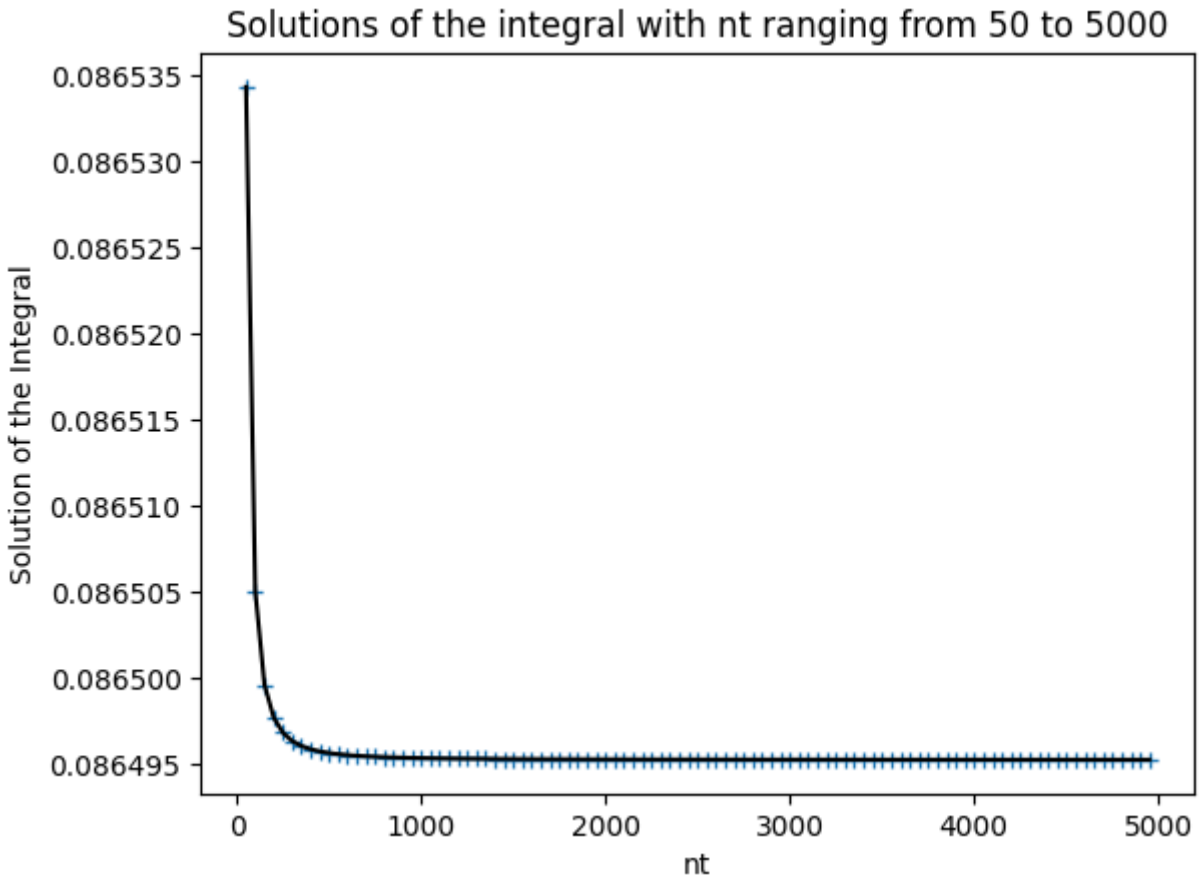
```
c when nt = 50:  0.08653434800479506
```



In [146…

```
all_nt = np.arange(50, 5000, 50)
c = []
x = []

for nt in all_nt:
    # sampling between [0,0.5]
    t1 = np.linspace(0, 0.5, int(nt))
    # double sampling between [0.5,1]
    t2 = np.linspace(0.5, 1, 2*int(nt))
    # concatenate time vector
    t = np.concatenate((t1[:-1], t2))
    # compute y values (f=2t)
    y = np.sin(2 * np.pi * (t + 3*t**2))
    # plt.plot(t, y)
    # compute sampling interval vector
```

```
    dt = t[1:] - t[:-1]
    c.append(integral(y, dt))
    x.append(nt)
    # print("nt = : ",nt, "   c = ", c)

plt.plot(x, c, "+")
plt.plot(x, c, "k")
plt.title("Solutions of the integral with nt ranging from 50 to 5000")
plt.xlabel("nt")
plt.ylabel("Solution of the Integral")
plt.show()
# print(c)
```



By inspecting the data and graph shown above, we can conclude that c is converging toward 0.086496 as we increase nt.

Accuracy of Sampling

Let us sample the function $g(t) = \cos(2\pi ft)$ at sampling interval $dt = 1$, for frequency values of $f = 0, 0.25, 0.5, 0.75, 1.0$ hertz.

In each case, plot on the screen the points of the resulting time series (as isolated red crosses) to see how well it approximates $g(t)$ (plotted as a blue-dotted line, try a very small dt fine sampling). Submit only plots for frequencies of 0.25 and 0.75 Hertz, use xlabel, ylabel, title commands to annotate each plot.

For each frequency that you investigated, do you think the sampling time series is a fair representation of the original time series $g(t)$?

What is the apparent frequency for the sampling time series? (Figure out after how many points (N) the series repeats itself, then the apparent frequency = $1/(N * dt)$. You can do this either mathematically or by inspection. A flat time series has apparent frequency = 0.)

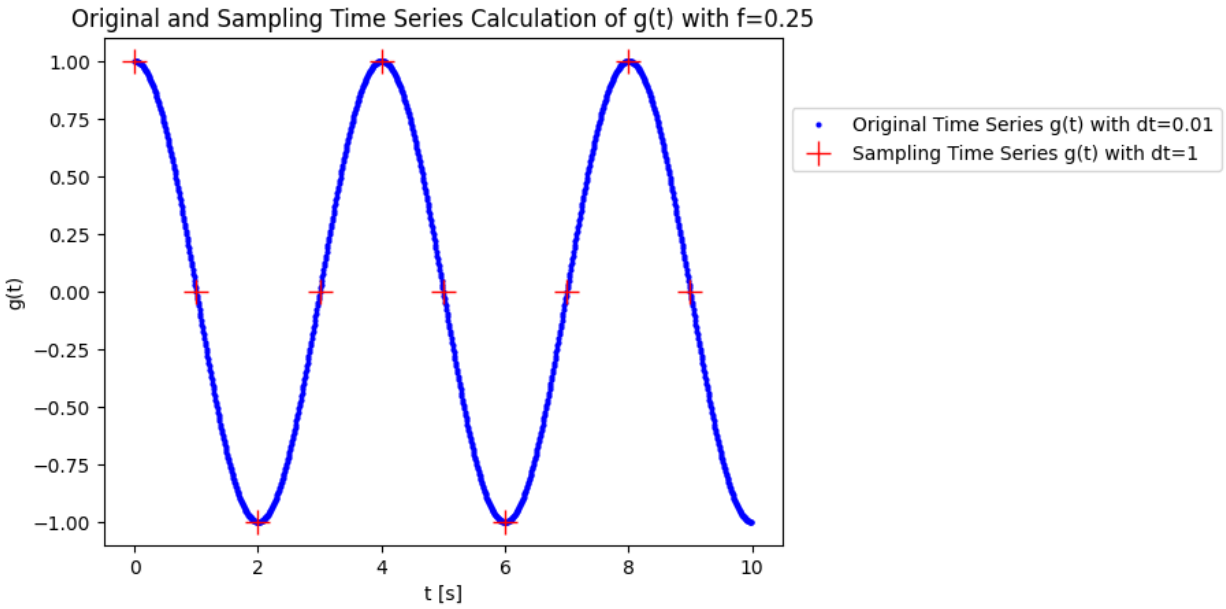
Can you guess with a sampling interval of $dt = 1$, what is the maximum frequency f of $g(t)$ such that it can be fairly represented by the discrete time series? (Please attach your Python code.)

All code, calculation and graphs for the Accuracy of Sampling Section:

In [147…
```
# to show all graphs (personal calculation only)
f_all = [0, 0.25, 0.5, 0.75, 1.0]
```

In [154…
```
# to only show graphs with f = 0.25 or 0.75 (as required from this lab)
f_all = [0.25, 0.75]
```

In [149…
```
dt = 1
t = np.arange(0, 10, dt)
t_fine = np.arange(0, 10, 0.01)

for f in f_all:
    g = np.cos(2*np.pi*f*t)
    g_fine = np.cos(2*np.pi*f*t_fine)

    # plotting a line plot after changing it's width and height
    plt.plot(t_fine, g_fine, "ob", markersize=2,
             label="Original Time Series g(t) with dt=0.01")
    plt.plot(t, g, "+r", markersize=13,
             label="Sampling Time Series g(t) with dt=1")
    plt.title("Original and Sampling Time Series \
              Calculation of g(t) with f="+ str(f))
    plt.xlabel("t [s]")
    plt.ylabel("g(t)")
    plt.legend(loc='center left', bbox_to_anchor=(1, 0.8))
    plt.show()
```



Apparent Frequency Calculation:

f	N	Aparent Frequency
0	∞	0
0.25	4	0.25
0.50	2	0.50
0.75	4	0.25
1.00	∞	0

Using the equation given to calculate the aparent frequency, the aparent frequency we calculated for the f=0.25 and f=0.5 graph is pretty accurate. For the graphs with f=0 and f=1, the Sampling Time Series are not providing any useful information since their Apparent Frequencies are both 0. In both cases, the Sampling Time Sereis are not representing the

Original Time Series at all, meaning if we draw a line of best fit using only the information from the Sampling Time Series, the line of best fit will not match the Original Time Series at all. The Sampling Time Series for f=0.75 is 0.25, which is also very off from the expected value.

Based on all the graphs generated with different f (as required by the question: 0, 0.25, 0.5, 0.75, 1.0), the max frequency f of g(t) such that it can be fairly represented by the discrete time series is when f=0.5 since that's the last graph when you see the elevation of the Original Time Series matches the Sampling Time Series that gives us a good representation of what the expected trend would look like.