PHY408 Lab 0

Wednesday, January 17, 2024

Jace Alloway - 1006940802 - alloway1

This assignment was compiled in VS Code using the Python and LaTeX extensions. Matplotlib does not produce inline plots in terminal, so the `%matplotlib inline` command was commented out. **Collaborators for all questions: none.**

Problem 1

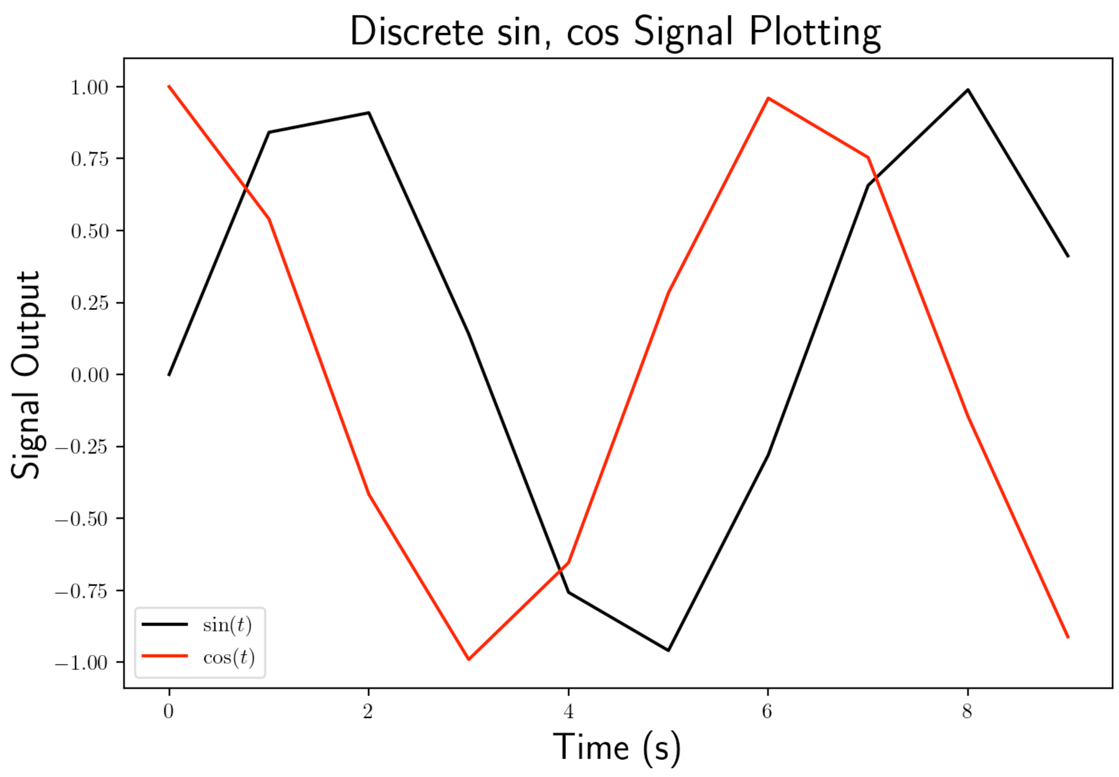
Some introductory warm-up exercises where we investigate operations in python:

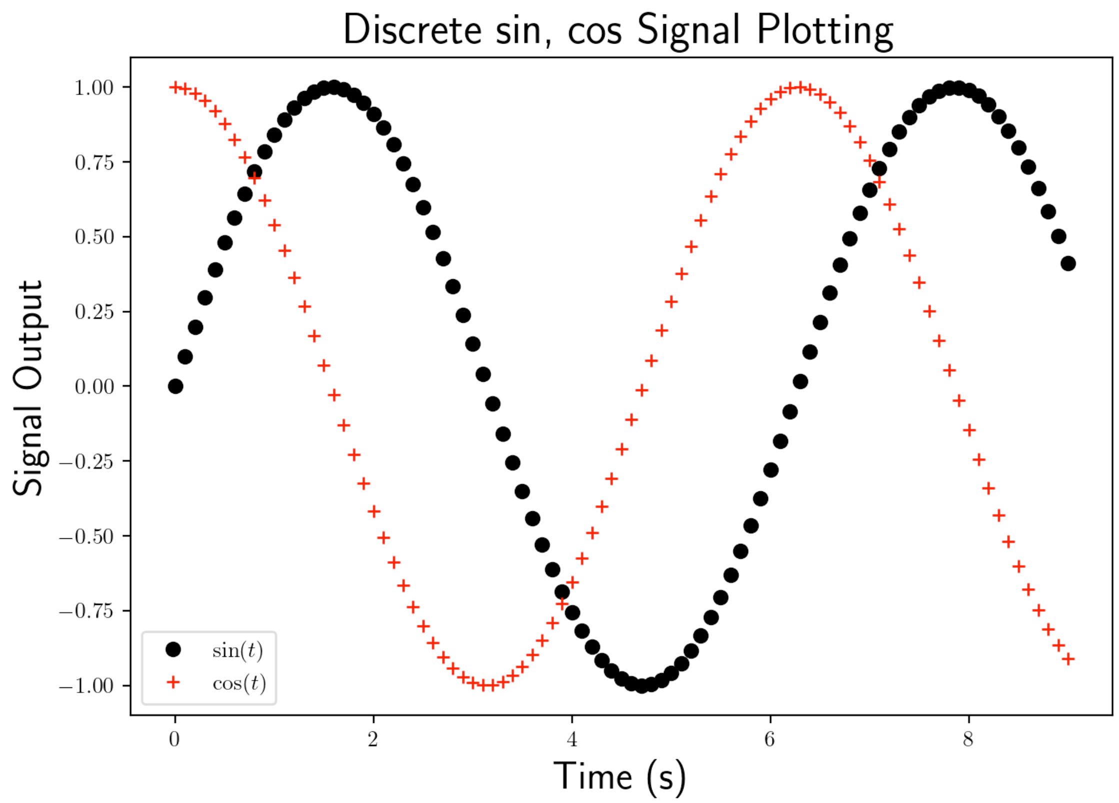
- `a/b` operates as regular division, $\frac{a}{b}$.
- `a//b` operates as floor division, rounding the fraction down to the nearest integer $\left\lfloor \frac{a}{b} \right\rfloor$
- `a-b` defines subtraction $a - b$.
- `a*b` operates as multiplication, $ab \equiv a \cdot b$ for scalar integers or floats.
- `a**b` defines exponentiation, raising scalars a to the power of b, a^b.

For $a = 6$, and $b = 2$, we find that `a/b = 3.0`, `a//b = 3`, `a-b = 4`, `a*b = 12`, and `a**b = 36` as expected. We may also define arrays as lists of values (or other arrays) up to $n \times m$ dimensions, $A = [[3, 1], [1, 3]]$ and $B = [[3], [5]]$. We find that operations on matrices are different than those of working with scalar floats:

- `A*B` multiplies the i-th value of each array through the other (matrix multiplication), which gives another array $[[9, 3], [5, 15]]$.
- The `np.dot(A, B)` function takes the dot product between vector valued-entries in the array, multiplication of respective components, then addition of respective components: `np.dot(A,B)`$=$ $[[9 + 5], [3 + 15]] = [[14], [18]]$ (a column array).
- For `np.dot(B.T, A)`, the `.T` function transposes B into a single array (row array goes to column, vice versa), then takes the dot product as before with A: `np.dot(B.T, A)`$= [14, 18]$ (a row array).
- We may also define the inverse of the matrix A by raising it to the power of -1.0, `C = A**(-1.0)`. Hence, since $A^{-1}A = \mathbf{1}$, the distributive matrix multiplication between A and C also give the unit matrix: `C*A = [[1, 1],[1,1]]`.

Plotting was done using matplotlib.pyplot. Since discrete data arrays are plotted on the x and y axes, we must define these arrays. A simple time array can be generated using `np.arange()`, generating evenly spaced values (by 1) between two endpoints. Thus
`t = np.arange(10) = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)`. Evaluating periodic functions as arrays such as `\np.sin(t), \np.cos(t)` produce the appropriate values as the function would. This was plotted in matplotlib:



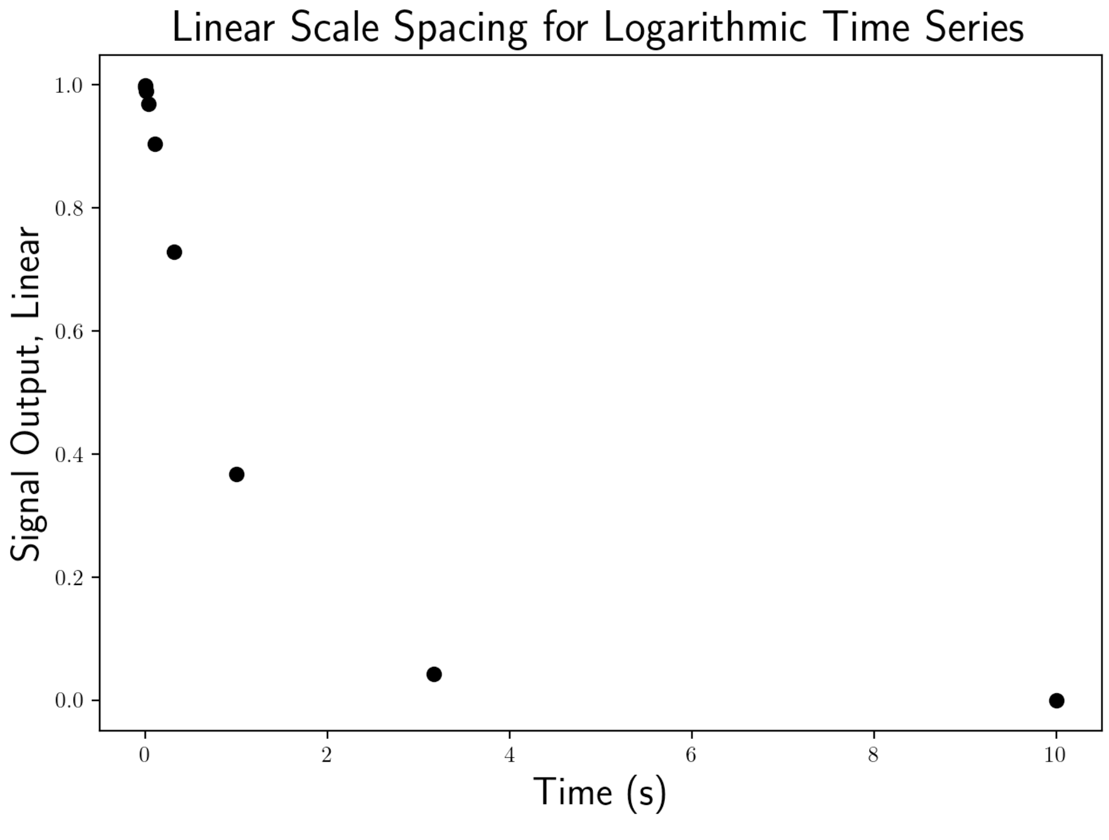


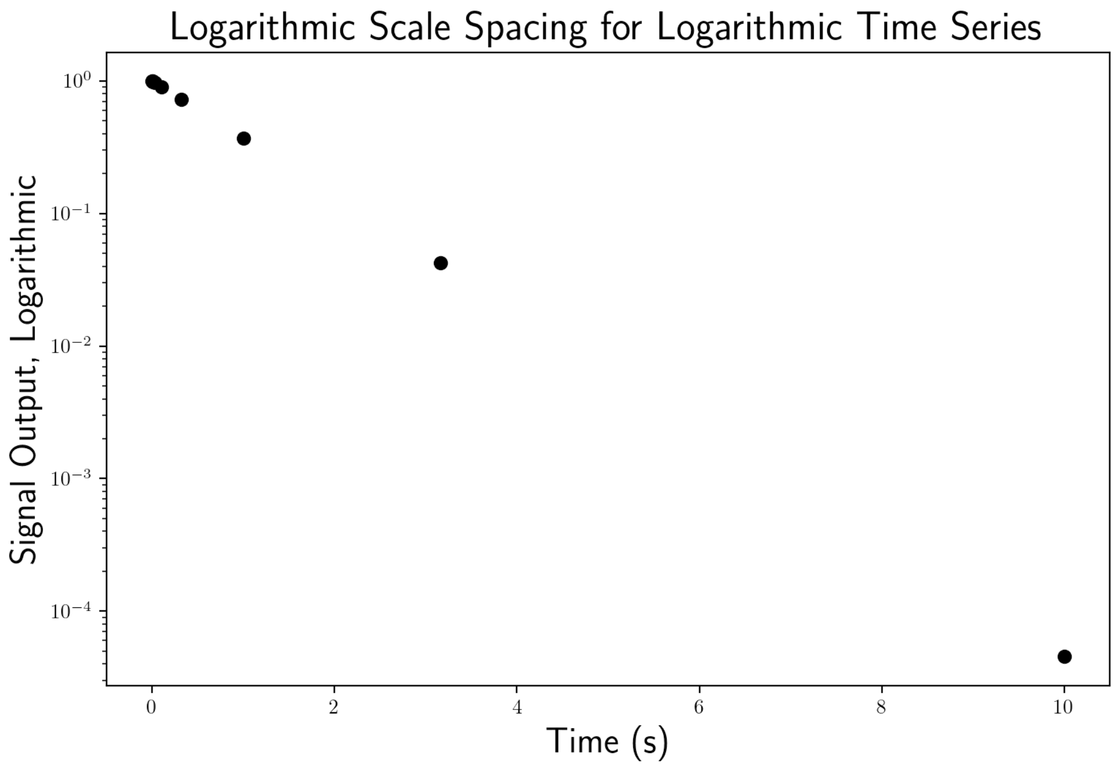
Here, the second figure is plotted as crosses and dots instead of a continuous line, this time using `np.arange(0, 9.1, 0.1)` to specify the input array; a list of 90 numbers 0 to 9 whose spacing is 0.1.

For the last exercise, we may also experiment with various times of spacings using functions other than `np.arange`. The `np.linspace()` function specifies endpoints but instead of a spacing argument, takes the number of points you want in between your endpoints. That is, `np.linsace(0, 10, 20)` prints 20 points evenly spaced between 0 and 10. The `np.logspace()` function, similarly, takes endpoints and the number of points between them, however spacing the points logarithmically with base e. The endpoint argument(s) take real values from $-\infty$ to $+\infty$, representing the codomain of the function $\log(t)$, then mapping them to the respective t values on the interval $(0, \infty)$. Thus we obtain

- `np.linspace(0, 10, 20) = [0, 0.5263, 1.0526, ..., 9.4736, 1]`
- `np.logspace(0.001, 10, 9) = [1.002, 17.818, 316.774, ..., 1e+10]`
- `np.logspace(-3, 1, 9) = [1e-3, 3.162e-3, ..., 3.162, 10].`

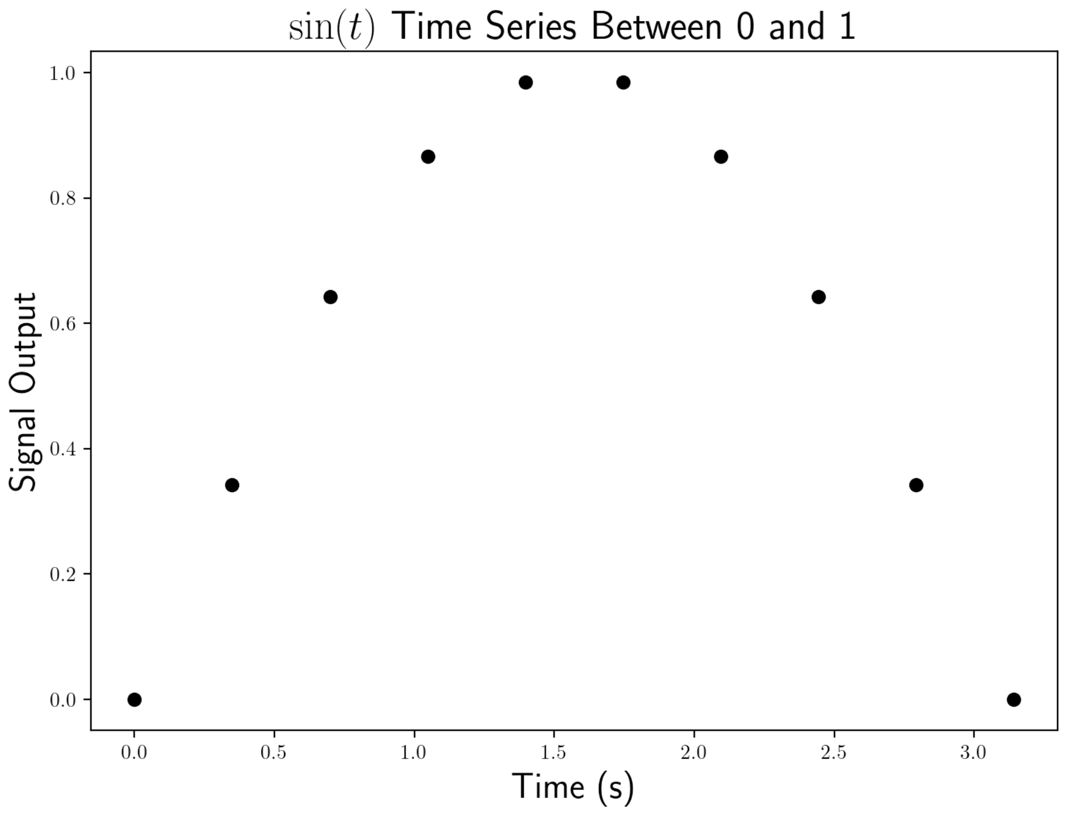
Then, for logarithmically spaced time values, we have `y=np.exp(-t)` raising the values by base e. Plotting this along a linear plot scale yields a decreasing exponential, but plotting along a logarithmic scale (along y) must yield a line with slope -1, as we find:




Logarithmic Scale Spacing for Logarithmic Time Series
Signal Output, Logarithmic
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
2
4
6
8
10
Time (s)

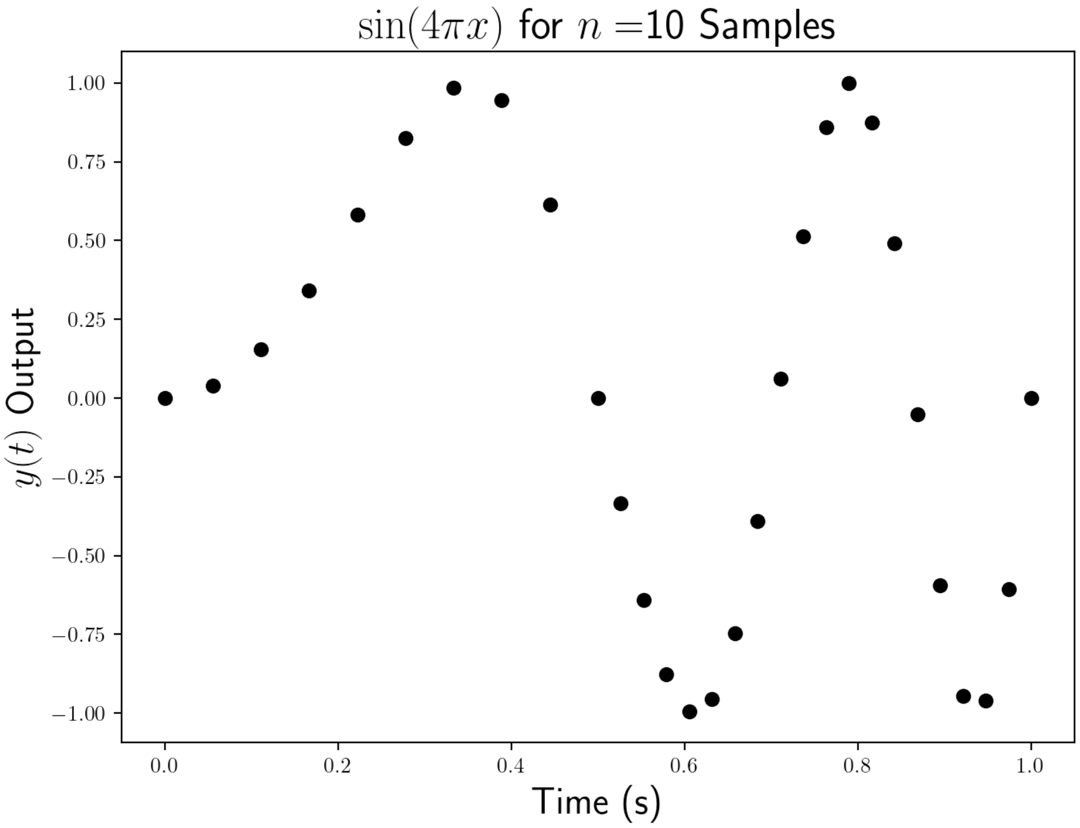
Problem 2

We now examing the integration of time-series outputs. Using the function as defined in the assignment paper, notice that the integration limits are not defined in-function but actually as the boundaries of the time-series array. For the integral $\int_0^\pi \sin(t)\,dt$, we using 10 sampling points at constant dt interval, approximating the integral to 1.979. In the $dt \to 0$ limit, the integral is 2, making this a relatively ok approximation for the low sampling value.



To improve the accuracy of the integral, we can increase the number of samples `nt` between 0 and π. This is because, as defined in the function, we take each integration 'rectangle' as the average of upper and lower sums multiplied by the width, as one would do for a Riemman sum. Increasing `nt` means decreasing dt, achieving closer rectangles to the actual curve.

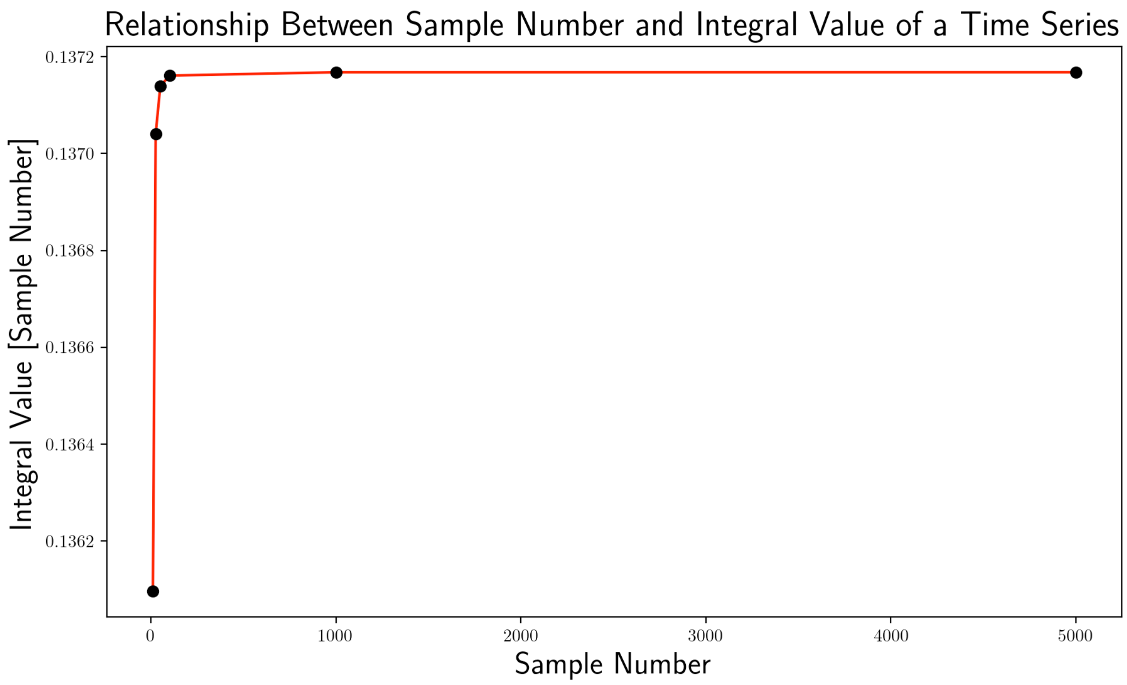
For a nonlinearly spaced time series (`np.linspace(0, 0.5, nt)` concatenated with `np.linspace(0.5, 1, 2*nt)`, and `nt` is the number of time samples), we evaluate the integral $\int_0^1 \sin(4\pi t^2)\,dt$. As before, start with `nt=10`, and use dt as an array to represent the spacing jump at the 10th interval. The integration function for 10 samples gives ≈ 0.136.



We should now consider the $nt \to \infty$ limit, observing the actual convergence of the integral.

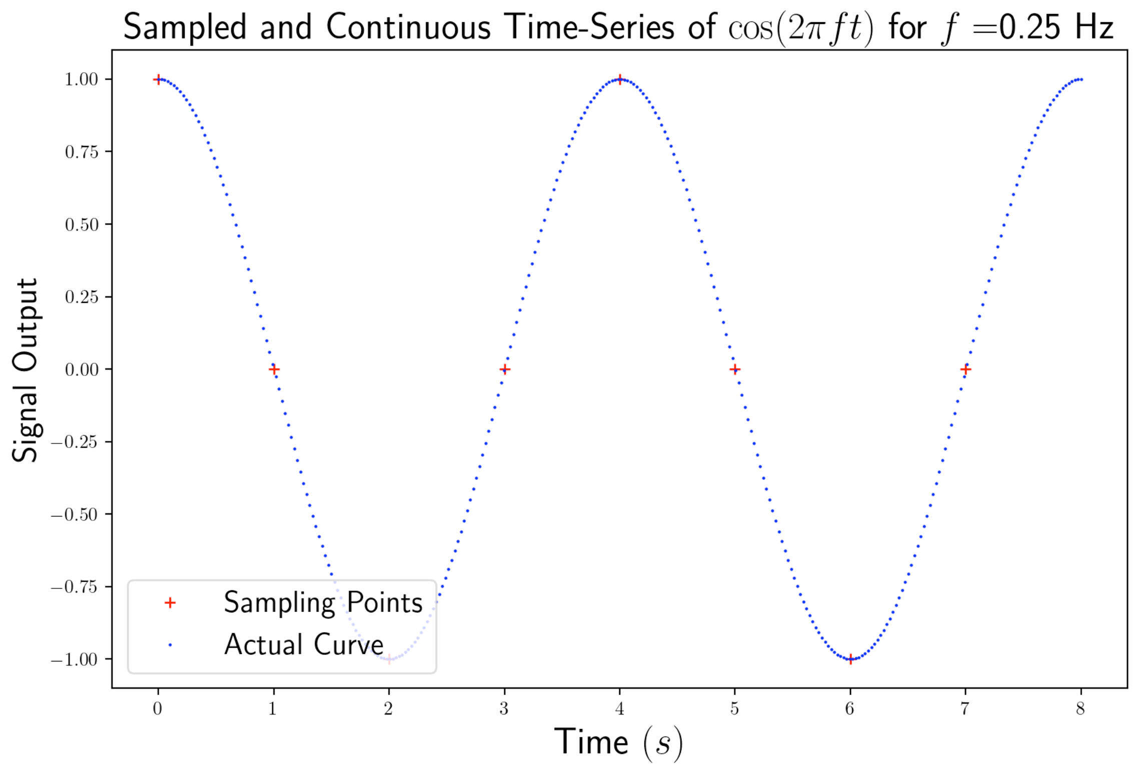
nt	I(nt)
10	0.13609684
25	0.13704010
50	0.13713883
100	0.13716087
1000	0.13716768
5000	0.13716775

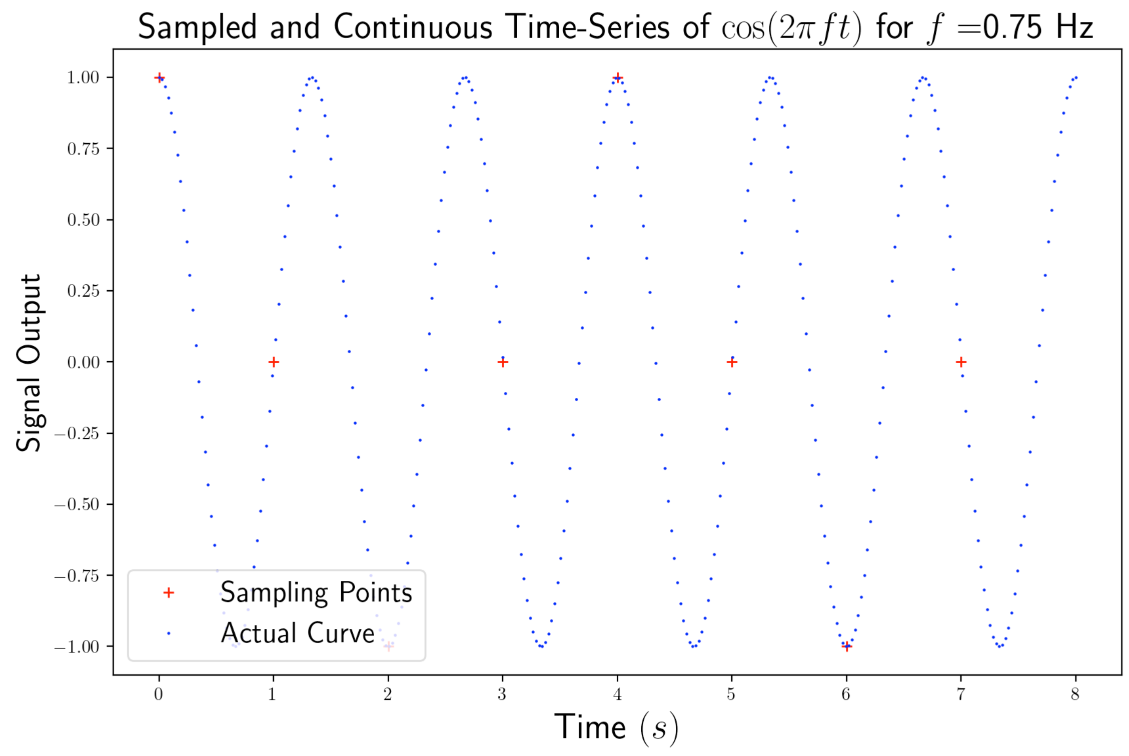
This was by looping over various `nt` values. Theoretically [by Integral Calculator], the actual value of the integral converges to
$0.1371677527918979 - 8.392975430303765 \times 10^{-67}i$, making this approximating of $nt \to \infty$ valid.


Relationship Between Sample Number and Integral Value of a Time Series
Integral Value [Sample Number]
Sample Number
0.1372
0.1370
0.1368
0.1366
0.1364
0.1362
0
1000
2000
3000
4000
5000

Problem 3

For the last problem, we examing sampling accuracy and the validity of taking smaller/larger sampling intervals dt. We sample the function $g(t) = \cos(2\pi ft)$ for frequency values $f = (0, 0.25, 0.5, 0.75, 1.0)$ Hz at a $dt = 1$ interval. Below are the plots for $f = 0.25, 0.75$ Hz using a finely spacing time interval to represent the actual $g(t)$ series.





In some cases, the $f = 0.0$ and $f = 0.25$ series, the sampled time series appproximately models the continuous series and may be viewed as a fair representation just by connecting points. However, for cases in which f increases ($f = 0.50, 0.75, 1.0$) it can be seen that the series oscillates in between sampled points, thus making the sampled series a poor representation of the true signal.
Inspecting each graph, we can determine the apparent frequency of the sampled series by counting the number of points it takes for the series to repeat itself. For $dt = 1$,

f (Hz)	N (Points Per Period)	Apparent Frequency (Hz)	Good Representation?
0.00	1	0 (Flat)	Yes
0.25	4	0.25	Yes
0.50	2	0.50	Yes
0.75	4	0.25	No
1.00	1	0 (Flat)	No

One finds that there exists a frequency f such that the sample no longer appears reasonables since the sampling interval is so large: $dt = 1$ here. For $f = 0.25$ Hz, it is easy to estimate the value(s) of the original (continuous) time series just by interpolation, assuming the curve follows a sinusoidal periodicity (no saw waves). Even for $f = 0.50$ Hz, one may interpolate the cosine curve just by taking the sampled points as the maxima and minima, however any frequency greater appears to poorly represent the time series, since the actual signal may oscillate more in between points, as one may observe with the $f = 0.75$ Hz case: the apparent frequency is not 0.75 Hz, but 0.25 Hz. Thus $f = 0.50$ Hz appears to be the maximum frequency of $g(t)$ such that the sampled time series fairly represents the actual signal.