ECE60146 Homework 1

Suhas Dara: daras@purdue.edu

January 20, 2026

1 Implementation

1.1 Creating the Base Class

The base class `SignalProcessor` automatically inherits from the root class of `object` in Python. Its constructor `__init__` takes a singular list of `float`s and uses it as the data container.

```
class SignalProcessor:
    def __init__(self, data: list[float]):
        self.data = data
```

1.2 Creating the Sine Wave Function

The class `SineWaveFunction` inherits from the previously defined base class of `SignalProcessor`. However, it uses its own characteristic parameters of `amplitude` and `frequency`. It will not actually be instantiated with the data until later, when the class is made callable. So, for initialization, the `super` class's constructor is called with just an empty list as a placeholder.

```
class SineWaveFunction(SignalProcessor):
    def __init__(self, amplitude: float, frequency: float):
        super().__init__(data=[])
        self.amplitude = amplitude
        self.frequency = frequency
```

1.3 Making Objects Callable

An implementation of a `__call__` function is added to the `SineWaveFunction` class to make it callable. This handles the "heavy-lifting" of generating the data when an instance of the class is "called" like a function. The number of data samples generated depends on the integer `duration` parameter. Simple list comprehension is used to achieve this before printing to stdout.

```
class SineWaveFunction(SignalProcessor):
    ...
    def __call__(self, duration: int):
        self.data = [
            self.amplitude * math.sin(2 * math.pi * self.frequency * n)
            for n in range(0, duration)
        ]
        print(self.data)
```

As for the length of the data, this is implemented in the base class itself as its implementation will not change across the subclasses.

```
class SignalProcessor:
    ...
    def __len__(self):
        return len(self.data)
```

The rounded output (to 4 decimals) of the example usage in the following snippet of code

```
SG = SineWaveFunction(amplitude=2.0, frequency=0.1)
SG(duration=5)
print(len(SG))
```

is as follows.

```
[0.0, 1.1756, 1.9021, 1.9021, 1.1756]
5
```

1.4 Making Objects Iterable

All objects of type `SignalProcessor` or its subclasses will be iterable in the same manner, so this can be implemented in the base class itself once again. To make the class iterable, the `__iter__` and `__next__` functions are implemented. The `__iter__` function initializes the iterator index to `-1`; this needs to be done here as opposed to the constructor because each instance should be iterable more than once. The `__next__` function advances the iterator index and returns the item at the index as long as it is within bounds.

```
class SignalProcessor:
    ...
    def __iter__(self):
        self.iter_index = -1
        return self

    def __next__(self):
        self.iter_index += 1
        if self.iter_index < len(self):
            return self.data[self.iter_index]
        else:
            raise StopIteration
```

The rounded output (to 4 decimals) of the example usage in the following snippet of code

```
SG = SineWaveFunction(amplitude=2.0, frequency=0.1)
SG(duration=5)
print([val for val in SG])
```

is as follows (where the first print is from `__call__` and the second is from the iterator).

```
[0.0, 1.1756, 1.9021, 1.9021, 1.1756]
[0.0, 1.1756, 1.9021, 1.9021, 1.1756]
```

1.5 Square Wave Function

The class `SquareWaveFunction` also inherits from the previously defined base class of `SignalProcessor`. Similar to the `SineWaveFunction`, it also uses the same parameters of `amplitude` and `frequency`. This may make it seem like it should perhaps inherit from `SineWaveFunction` but, logically, these are unrelated classes beyond the base functionalities. The implementation of `__call__` differs from `SineWaveFunction` and is again implemented using list comprehension.

```
class SquareWaveFunction(SignalProcessor):
    def __init__(self, amplitude: float, frequency: float):
        super().__init__(data=[])
        self.amplitude = amplitude
        self.frequency = frequency

    def __call__(self, duration: int):
        self.data = [
            self.amplitude * (
                1 if math.sin(2 * math.pi * self.frequency * n) >= 0 else -1
            )
            for n in range(0, duration)
        ]
        print(self.data)
```

The output of the example usage in the following snippet of code

```
SW = SquareWaveFunction(amplitude=3.0, frequency=0.1)
SW(duration=5)
print(len(SW))
print([val for val in SW])
```

is as follows (where the first print is from `__call__` and the second is from the iterator). It is important to note that this particular wave is always positive for a short duration of 5 samples because of a low frequency.

```
[3.0, 3.0, 3.0, 3.0, 3.0]
5
[3.0, 3.0, 3.0, 3.0, 3.0]
```

1.6 Comparing Signals

The `__eq__` method needs to be implemented in the base class `SignalProcessor` to support using the `==` operator on any instances of the subclasses. A little unconventionally, it returns the number of elements that match in the two instances if they are of equal length, otherwise it throws a `ValueError`. This is accomplished by using the `sum` function in conjunction with list comprehension to check the threshold difference.

```
class SignalProcessor:
    ...
    def __eq__(self, other: 'SignalProcessor'):
        if len(self) != len(other):
            raise ValueError('Two signals are not equal in length!')
        return sum([
            abs(self.data[i] - other.data[i]) < 0.01 for i in range(len(self))
        ])
```

The rounded output (to 4 decimals) of the example usage in the following snippet of code

```
SG1 = SineWaveFunction(amplitude=2.0, frequency=0.1)
SG1(duration=5)
SG2 = SineWaveFunction(amplitude=2.0, frequency=0.15)
SG2(duration=5)
print(SG1 == SG2)

SG3 = SineWaveFunction(amplitude=2.0, frequency=0.1)
SG3(duration=3)
print(SG1 == SG3)
```

is as follows. When comparing `SG1` and `SG2`, it can be seen that there are two indices (0 and 2) where the values match. However, when attempting to compare `SG1` with `SG3`, it can be seen that the length mismatch leads to an error being raised.

```
[0.0, 1.1756, 1.9021, 1.9021, 1.1756]
[0.0, 1.6180, 1.9021, 0.6180, -1.1756]
2
[0.0, 1.1756, 1.9021]
Traceback (most recent call last):
  File "/home/suhas/data/s26-ece60146/hw1/hw1.py", line 75, in <module>
    print(SG1 == SG3)
          ^^^^^^^^^^
  File "/home/suhas/data/s26-ece60146/hw1/hw1.py", line 23, in __eq__
    raise ValueError('Two signals are not equal in length!')
ValueError: Two signals are not equal in length!
```

1.7 Demonstrate Your Implementation

Here, I will pick specific `frequency` and `amplitude` parameters to demonstrate my implementations of `SineWaveFunction` and `SquareWaveFunction`.

```python
print('Testing SineWaveFunction')
SN = SineWaveFunction(amplitude=math.pi, frequency=0.125)
SN(duration=10)
print(len(SN))
print([val for val in SN], '\n')

print('Testing SquareWaveFunction')
SQ = SquareWaveFunction(amplitude=math.pi, frequency=0.1)
SQ(duration=10)
print(len(SQ))
print([val for val in SQ], '\n')

print('Testing SignalProcessor.__eq__')
print(SN == SQ)
SQ(duration=8)  # can re-generate with different duration
print(len(SQ))
print(SN == SQ)  # should raise ValueError
```

The above parameter choices will print the following output (rounded to 4 decimals):

```
Testing SineWaveFunction
[0.0, 2.2214, 3.1416, 2.2214, 0.0, -2.2214, -3.1416, -2.2214, 0.0, 2.2214]
10
[0.0, 2.2214, 3.1416, 2.2214, 0.0, -2.2214, -3.1416, -2.2214, -0.0, 2.2214]

Testing SquareWaveFunction
[3.1416, 3.1416, 3.1416, 3.1416, 3.1416, 3.1416, -3.1416, -3.1416, -3.1416, -3.1416]
10
[3.1416, 3.1416, 3.1416, 3.1416, 3.1416, 3.1416, -3.1416, -3.1416, -3.1416, -3.1416]

Testing SignalProcessor.__eq__
2
[3.1416, 3.1416, 3.1416, 3.1416, 3.1416, 3.1416, -3.1416, -3.1416]
8
```

```
Traceback (most recent call last):
  File "/home/suhas/data/s26-ece60146/hw1/hw1.py", line 84, in <module>
    print(SN == SQ)  # should raise ValueError
  File "/home/suhas/data/s26-ece60146/hw1/hw1.py", line 23, in __eq__
    raise ValueError('Two signals are not equal in length!')
ValueError: Two signals are not equal in length!
```

In the testing of `SineWaveFunction`, it can be seen that by choosing a frequency that is a negative power of 2, we are able to sample at exactly the peak and trough of the sine wave where the amplitudes are appropriately π and $-\pi$ respectively. The duration of the signal of 10 is reflected in the output of a `len` call.

For the `SquareWaveFunction`, it can be seen that it switches from positive to negative amplitude at some point. Important to note that rounding errors allow for 0.0 to map to negative amplitude at index 9 because the value being compared is actually a very small negative number. The duration of the signal of 10 is reflected in the output of a `len` call.

Since the sine wave `SN` and square wave `SQ` initially have the same length, the `SignalProcessor.__eq__` operation does not error. They match only at the peak and the trough though, giving an answer of 2. Later, the square wave `SQ` is reused with a new duration of 8. The sine wave `SN` and the square wave `SQ` no longer have the same length (duration), so the `__eq__` operator appropriately raises a `ValueError`.

2 Bonus

2.1 Composite Signal Function

It is fairly straightforward to extend the base class of `SignalProcessor` to `CompositeSignalFunction` such that it takes in a list of `SignalProcessors` that are summed together to form a single signal. Once again, it is mainly the `__call__` function that differs. Each of the input signals is "called" before they are summed index-wise. The `duration` is common across all the input signals, so there should be no index errors.

```python
class CompositeSignalFunction(SignalProcessor):
    def __init__(self, inputs: list[SignalProcessor]):
        super().__init__(data=[])
        self.inputs = inputs

    def __call__(self, duration: int):
        for input_sig in self.inputs:
            input_sig(duration)
        self.data = [sum(n) for n in zip(*self.inputs)]
        print(self.data)
```

The rounded output (to 4 decimals) of the example usage in the following snippet of code

```
SG = SineWaveFunction(amplitude=1.0, frequency=0.1)
SW = SquareWaveFunction(amplitude=0.5, frequency=0.05)
CSG = CompositeSignalFunction(inputs=[SG, SW])
CSG(duration=10)
```

is as follows. "Calling" each signal first implies its data will get printed first. `SG`'s data is printed on the first line, and `SW`'s data is printed on the second line (it acts as a DC offset due to the low frequency and short duration). The combined signal is on the third line.

```
[0.0, 0.5878, 0.9511, 0.9511, 0.5878, 0.0, -0.5878, -0.9511, -0.9511, -0.5878]
[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]
[0.5, 1.0878, 1.4511, 1.4511, 1.0878, 0.5, -0.0878, -0.4511, -0.4511, -0.0878]
```

2.2 Signal Visualization

The following three signals (one of each type) were chosen to be visualized using `matplotlib` and `numpy` for a total duration of 100 samples / seconds at an implied sampling rate of 1Hz with the `frequency` parameter acting as a scaling factor.

```
sine_wave = SineWaveFunction(amplitude=1.4, frequency=0.05)
square_wave = SquareWaveFunction(amplitude=1.1, frequency=0.15)

# generate a smoothed stepped signal
sw = SineWaveFunction(amplitude=0.5, frequency=0.025)
sqw1 = SquareWaveFunction(amplitude=0.5, frequency=0.025)
sqw2 = SquareWaveFunction(amplitude=0.5, frequency=0.05)
sqw3 = SquareWaveFunction(amplitude=0.5, frequency=0.1)
composite_signal = CompositeSignalFunction(inputs=[sw, sqw1, sqw2, sqw3])
```

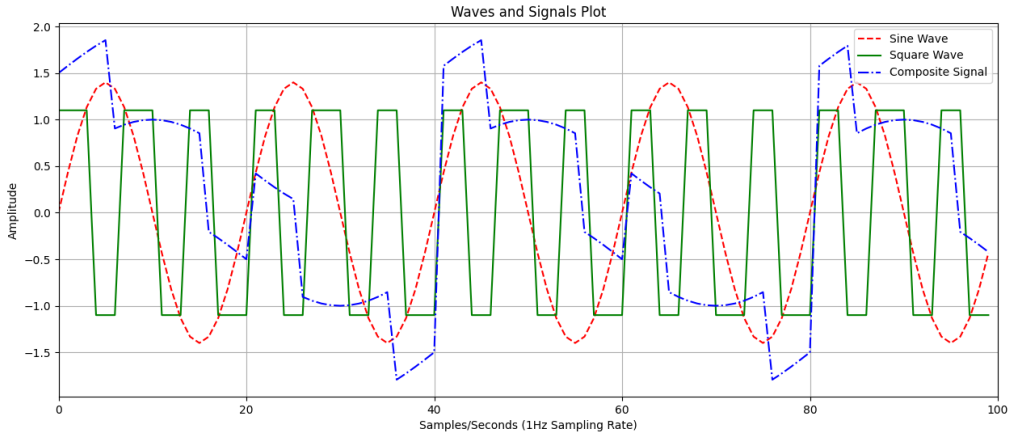

Figure 1: A plot showing a sine wave, a square wave, and a composite signal

An FFT analysis was then performed on the same three signals. Again, an implied sampling rate of 1Hz was used for this analysis.

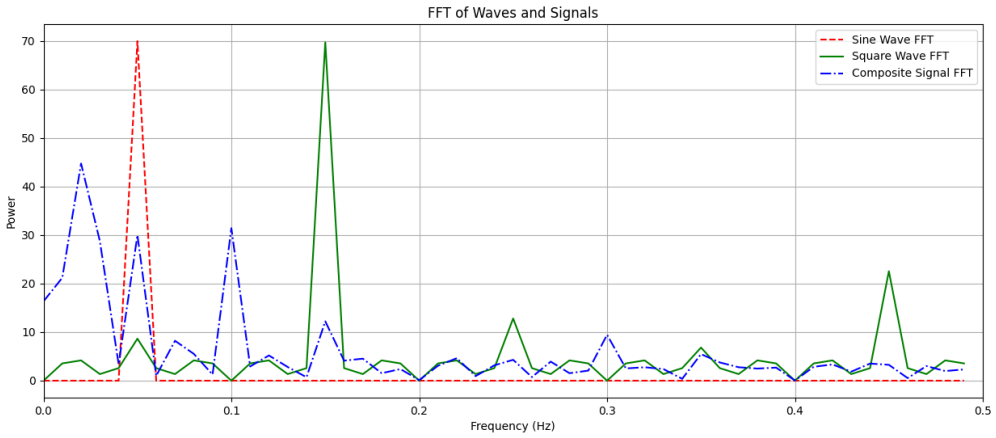

Figure 2: A plot showing the FFT analysis of a sine wave, a square wave, and a composite signal

In the FFT plot above, the following can be clearly seen:

- The sine wave appropriately has a peak at 0.05Hz.
- The square wave has peaks at the odd harmonics of 0.15Hz. This can be seen as 0.15Hz, 0.45Hz, 0.75Hz (aliased to 0.25Hz) etc.
- The composite signal is more complex but has peaks at 0.025Hz, representing the sine wave component and one square wave component, and at 0.05Hz and 0.1Hz for the other two square wave components (along with smaller odd harmonic peaks).

3 Code

3.1 Primary Code

`hw1.py`

```python
import math

class SignalProcessor:
    """Base Class for Signal Processing Functions."""
    def __init__(self, data: list[float]):
        self.data = data

    def __len__(self):
        return len(self.data)

    def __iter__(self):
        self.iter_index = -1
        return self

    def __next__(self):
        self.iter_index += 1
        if self.iter_index < len(self):
            return self.data[self.iter_index]
        else:
            raise StopIteration

    def __eq__(self, other: 'SignalProcessor'):
        if len(self) != len(other):
            raise ValueError('Two signals are not equal in length!')
        # Automatically coerce booleans into 0 or 1 to sum
        return sum([abs(self.data[i] - other.data[i]) < 0.01 for i in range(len(self))])


class SineWaveFunction(SignalProcessor):
    """A basic Sine Wave class."""
    def __init__(self, amplitude: float, frequency: float):
        super().__init__(data=[])
        self.amplitude = amplitude
        self.frequency = frequency

    def __call__(self, duration: int):
        self.data = [
            self.amplitude * math.sin(2 * math.pi * self.frequency * n)
            for n in range(0, duration)
        ]
        print(self.data)


class SquareWaveFunction(SignalProcessor):
    """A basic Square Wave class."""
    def __init__(self, amplitude: float, frequency: float):
        super().__init__(data=[])
        self.amplitude = amplitude
        self.frequency = frequency

    def __call__(self, duration: int):
        self.data = [
            self.amplitude * (1 if math.sin(2 * math.pi * self.frequency * n) >= 0 else -1)
            for n in range(0, duration)
        ]
        print(self.data)
```

```
class CompositeSignalFunction(SignalProcessor):
    """A class to combine multiple signals into one composite signal."""
    def __init__(self, inputs: list[SignalProcessor]):
        super().__init__(data=[])
        self.inputs = inputs

    def __call__(self, duration: int):
        for input_sig in self.inputs:
            input_sig(duration)
        # Sum the signals sample-wise
        self.data = [sum(n) for n in zip(*self.inputs)]
        print(self.data)


if __name__ == "__main__":
    print('Testing SineWaveFunction')
    SN = SineWaveFunction(amplitude=math.pi, frequency=0.1)
    SN(duration=10)
    print(len(SN))
    print([val for val in SN], '\n')

    print('Testing SquareWaveFunction')
    SQ = SquareWaveFunction(amplitude=math.pi, frequency=0.1)
    SQ(duration=10)
    print(len(SQ))
    print([val for val in SQ], '\n')

    print('Testing SignalProcessor.__eq__')
    print(SN == SQ)
    SQ(duration=8)  # can re-generate with different duration
    print(len(SQ))
    print(SN == SQ)  # should raise ValueError
```

3.2 Visualization Code

hw1_viz.py

```
import matplotlib.pyplot as plt
import numpy as np
from hw1 import SineWaveFunction, SquareWaveFunction, CompositeSignalFunction


def plot_signals(sine_wave: SineWaveFunction,
                 square_wave: SquareWaveFunction,
                 composite_signal: CompositeSignalFunction):
    """Plots the given signals on a single graph and saves the figure."""
    plt.figure(figsize=(15, 6))
    plt.plot(sine_wave.data, label='Sine Wave', linestyle='—', color='red')
    plt.plot(square_wave.data, label='Square Wave', linestyle='–', color='green')
    plt.plot(composite_signal.data, label='Composite Signal', linestyle='–.', color='blue')
    plt.title('Waves and Signals Plot')
    plt.xlim(0, len(sine_wave))
    plt.xlabel('Samples/Seconds (1Hz Sampling Rate)')
    plt.ylabel('Amplitude')
    plt.grid(True)
    plt.legend()
    plt.savefig('waves_and_signals_plot.png')


def plot_fft(sine_wave: SineWaveFunction,
             square_wave: SquareWaveFunction,
             composite_signal: CompositeSignalFunction):
    """Plots the FFT of the given signals on a single graph and saves the figure."""
    N = len(sine_wave)
    plt.figure(figsize=(15, 6))
    sine_wave_fft = abs(np.fft.fft(sine_wave.data))
    square_wave_fft = abs(np.fft.fft(square_wave.data))
    composite_signal_fft = abs(np.fft.fft(composite_signal.data))

    # Assuming an implied sampling rate of 1Hz. The nyquist would be 0.5Hz.
    # Take only positive frequencies for plotting.
    freq = np.fft.fftfreq(N, d=1.0)[:N//2]
    plt.xlim(0, 0.5)

    # Take only positive frequencies for plotting.
    plt.plot(freq, sine_wave_fft[:N//2], label='Sine Wave FFT', linestyle='—', color='red')
    plt.plot(freq, square_wave_fft[:N//2], label='Square Wave FFT', linestyle='–', color='green')
    plt.plot(freq, composite_signal_fft[:N//2], label='Composite Signal FFT', linestyle='–.', color='blue')
```

```
    plt.title('FFT of Waves and Signals')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Power')
    plt.grid(True)
    plt.legend()
    plt.savefig('waves_and_signals_fft.png')


if __name__ == "__main__":
    duration = 100
    sine_wave = SineWaveFunction(amplitude=1.4, frequency=0.05)
    square_wave = SquareWaveFunction(amplitude=1.1, frequency=0.15)

    # generate a stepped wave by combining square waves
    sw = SineWaveFunction(amplitude=0.5, frequency=0.025)
    sqw1 = SquareWaveFunction(amplitude=0.5, frequency=0.025)
    sqw2 = SquareWaveFunction(amplitude=0.5, frequency=0.05)
    sqw3 = SquareWaveFunction(amplitude=0.5, frequency=0.1)
    composite_signal = CompositeSignalFunction(inputs=[sw, sqw1, sqw2, sqw3])

    sine_wave(duration)
    square_wave(duration)
    composite_signal(duration)

    plot_signals(sine_wave, square_wave, composite_signal)
    plot_fft(sine_wave, square_wave, composite_signal)
```