Homework 1 Report

Lucas Nguyen nguye800@purdue.edu

0034489397

Signal Processor Code 2.1

Explanation

Parent class implementation correctly assigns input parameter data from which future child classes can inherit common characteristics (self.data)

```
class SignalProcessor ( object ):
    def __init__ (self, data):
        self.data = data
```

Sine Wave Code 2.2, 2.3, 2.4, 2.6

Explanation

__init__ method correctly calls parent class constructor and stores the 2 new parameters amplitude and frequency as variables.

__call__ method generates a sine wave from time steps 0 to duration-1 using the given formula for a sine wave y[n] = amplitude × sin(2π × frequency × n) and saves the values in self.data. Afterwards it prints the full wave values for all time steps as specified by instructions.

__iter__/__next__ work together to initialize an internal index used to navigate through self.data and enables iteration over signal values

__eq__ first checks if wave for comparison has an attribute data to compare against and additionally checks for equal length and will raise a ValueError with the appropriate message if different sizes as specified by instructions. Lastly, it will compare values of the wave per time step and count them as the same within a tolerance of 0.01

```python
from SignalProcessor import SignalProcessor
import math

class SineWaveFunction(SignalProcessor):
    def __init__(self, amplitude, frequency):
        # Your implementation here
        super().__init__([])
        self.amplitude = amplitude
        self.frequency = frequency

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

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

    def __iter__(self):
        self.idx = 0
        return self
```

```python
    def __next__(self):
        if self.idx >= len(self.data):
            raise StopIteration

        value = self.data[self.idx]
        self.idx += 1
        return value

    def __eq__(self, other):
        if not hasattr(other, "data"):
            return NotImplemented

        if len(other) != len(self.data):
            raise ValueError("Two signals are not equal in length!")

        count = 0
        for i in range(len(self.data)):
            if abs(self.data[i] - other.data[i]) < 0.01:
                count += 1
        return count
```

Square Wave Code 2.5, 2.6

Explanation

Square wave implements all magic methods specified for the sine wave. The only difference between the classes is that __call__ generates a square wave where the values of the wave can only be amplitude and -amplitude.

```python
from SignalProcessor import SignalProcessor
import math

class SquareWaveFunction(SignalProcessor):
    def __init__(self, amplitude, frequency):
        # Your implementation here
        super().__init__([])
        self.amplitude = amplitude
        self.frequency = frequency

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

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

    def __iter__(self):
        self.idx = 0
        return self

    def __next__(self):
        if self.idx >= len(self.data):
            raise StopIteration

        value = self.data[self.idx]
        self.idx += 1
        return value

    def __eq__(self, other):
        if not hasattr(other, "data"):
            return NotImplemented

        if len(other) != len(self.data):
            raise ValueError("Two signals are not equal in length!")

        count = 0
        for i in range(len(self.data)):
            if abs(self.data[i] - other.data[i]) < 0.01:
                count += 1
        return count
```

Testing Implementation Code 2.7

Explanation

Two durations were chosen for using in __call__ methods for waves in order to test different sizes of waves and verify functionality of __eq__ later in testing.

Multiple sine and square waves were created with different amplitudes and frequencies to show how parameter choices will affect wave values for both classes.

Tests were conducted by printing the name of the test which tested specific functionality of magic methods for both classes and were designed to verify correctness of edge cases in the class functionality.

```python
from SineWaveFunction import SineWaveFunction
from SquareWaveFunction import SquareWaveFunction

def main():
    duration_short = 10
    duration_mismatch = 6

    print("Initializing waves")
    # Sine
    sine1 = SineWaveFunction(amplitude=1.0, frequency=0.10)
    sine1_copy = SineWaveFunction(amplitude=1.0, frequency=0.10)
    sine2 = SineWaveFunction(amplitude=2.0, frequency=0.10)
    sine3 = SineWaveFunction(amplitude=1.0, frequency=0.20)

    # Square
    sq1 = SquareWaveFunction(amplitude=1.0, frequency=0.10)
    sq2 = SquareWaveFunction(amplitude=2.0, frequency=0.10)
    sq3 = SquareWaveFunction(amplitude=1.0, frequency=0.20)

    print("__call__ test")
    print("Sine Waves")
    sine1(duration_short)
    sine1_copy(duration_short)
    sine2(duration_mismatch)
    sine3(duration_short)

    print("Square Waves")
    sq1(duration_short)
    sq2(duration_mismatch)
    sq3(duration_short)

    print("__iter__/__next__ test")
    print("Sine test")
    for val in sine1:
        print(val)
```

```python
    print("Square test")
    for val in sq1:
        print(val)

    print("__len__ test")
    print("Sine test")
    print(len(sine1))
    print(len(sine2))
    print(len(sine3))

    print("Square test")
    print(len(sq1))
    print(len(sq2))
    print(len(sq3))

    print("__eq__ test")
    print("Sine1 vs Sq1: 0")
    print(sine1 == sq1)

    print("Sine1 vs Sine1_copy: 10")
    print(sine1 == sine1_copy)

    print("Sine1 vs Sine3: 2")
    print(sine1 == sine3)

    print("Sq1 vs Sq2: 5")
    print(sq1 == sq3)

    print("Compare 0.01 Tolerance")
    test1 = SineWaveFunction(amplitude=1.0, frequency=0.10)
    test2 = SineWaveFunction(amplitude=1.01, frequency=0.10)
    test1(2)
    test2(2)
    print(test1 == test2)


    print("Sine1 vs Sine2: duration mismatch")
    print(sine1 == sine2)

if __name__ == "__main__":
    main()
```

Testing Implementation Code Output 2.7

```
Initializing waves
__call__ test
Sine Waves
[0.0, 0.5877852522924731, 0.9510565162951535, 0.9510565162951536, 0.5877852522924732, 1.2246467991473532e-16, -0.587785252292473, -0.9510565162951535, -0.9510565162951536, -0.5877852522924734]
[0.0, 0.5877852522924731, 0.9510565162951535, 0.9510565162951536, 0.5877852522924732, 1.2246467991473532e-16, -0.587785252292473, -0.9510565162951535, -0.9510565162951536, -0.5877852522924734]
[0.0, 1.1755705045849463, 1.902113032590307, 1.9021130325903073, 1.1755705045849465, 2.4492935982947064e-16]
[0.0, 0.9510565162951535, 0.5877852522924732, -0.587785252292473, -0.9510565162951536, -2.4492935982947064e-16, 0.9510565162951535, 0.5877852522924734, -0.5877852522924728, -0.9510565162951538]
Square Waves
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, -1.0, -1.0]
[2.0, 2.0, 2.0, 2.0, 2.0, 2.0]
[1.0, 1.0, 1.0, -1.0, -1.0, -1.0, 1.0, 1.0, -1.0, -1.0]
__iter__/__next__ test
Sine test
0.0
0.5877852522924731
0.9510565162951535
0.9510565162951536
0.5877852522924732
1.2246467991473532e-16
-0.587785252292473
-0.9510565162951535
-0.9510565162951536
-0.5877852522924734
```

```
Square test
1.0
1.0
1.0
1.0
1.0
1.0
-1.0
-1.0
-1.0
-1.0
__len__ test
Sine test
10
6
10
Square test
10
6
10
__eq__ test
Sine1 vs Sq1: 0
0
Sine1 vs Sine1_copy: 10
10
Sine1 vs Sine3: 2
2
Sq1 vs Sq2: 5
5
Compare 0.01 Tolerance
[0.0, 0.5877852522924731]
[0.0, 0.5936631048153979]
2
Sine1 vs Sine2: duration mismatch
Traceback (most recent call last):
  File "c:\School\Y4\Spring\ECE60146\ECE60146-HW\HW1\comparison.py", line 77, in <module>
    main()
  File "c:\School\Y4\Spring\ECE60146\ECE60146-HW\HW1\comparison.py", line 74, in main
    print(sine1 == sine2)
          ^^^^^^^^^^^^^^
  File "c:\School\Y4\Spring\ECE60146\ECE60146-HW\HW1\SineWaveFunction.py", line 38, in __eq__
    raise ValueError("Two signals are not equal in length!")
ValueError: Two signals are not equal in length!
```

Analysis of Implementation 2.7:

Parameter Choices

Amplitudes of 1.0 and 2.0 were chosen to show vertical scaling of the wave and can be seen by comparing sine1 vs sine2 and sq1 vs sq2. Frequencies of 0.1 and 0.2 were chosen to show differences in oscillation speed and can be seen as faster sign/value changes for example sine3 achieves the same values of sine1 but temporally earlier. Amplitude of 1 and 1.01 at the bottom of the code were chosen to prove the 0.01 tolerance in the __eq__ magic method.

Observed Behavior

Sine1 produced peaks of roughly 0.95 due to the whole number time steps. Sine2 produced peaks exactly twice as large due to the amplitude changes for example 0.95->1.9. This matches the formula for the sine wave and proves that amplitude increases vertical height of the wave. For square waves they could only take two values (amplitude/-amplitude). Because of this, sq1 only outputs 1 and -1 and sine2 outputs 2 and -2. Having a lower frequency means that the wave doesn't oscillate as quickly, because of this effect we can see that sq1 stays positive/negative for longer before changing signs whereas sq3 switches signs quicker due to having double the frequency speed.

Magic Method Demonstrations

__init__: Correctly assigns amplitude and frequency arguments of values of the class and uses super class initialization for SignalProcessor with an empty array as data.

__call__: Generates a waveform for duration n amount of time and stores it in self.data and prints the resulting wave in a single line.

__iter__/__next__: Work together to print individual entries in self.data and move the index within self.data to the next element each step.

__len__: returns the length of self.data (either 10 or 6 depending on duration entered)

__eq__: Correctly demonstrates comparing wave values with a tolerance of 0.01. Sine1 vs Sine1_copy showed that all values were the same, Sine1 vs Sq1 showed that no values were the same, Sine1 vs Sine2 showed an error due to difference in length, Sine1 vs Sine3 showed that different frequency waves still overlapped at 2 points in the wave, Sq1 vs Sq3 showed an overlap of 5 times in the wave even though they are different frequencies. Comparing the eq tolerance used amplitude of 1 and 1.01 to show that waves with slightly different values at t=1 (0.5877 vs 0.5936) still were considered equal to each other.

Bonus Section 3.1

Composite Signal Code

Explanation

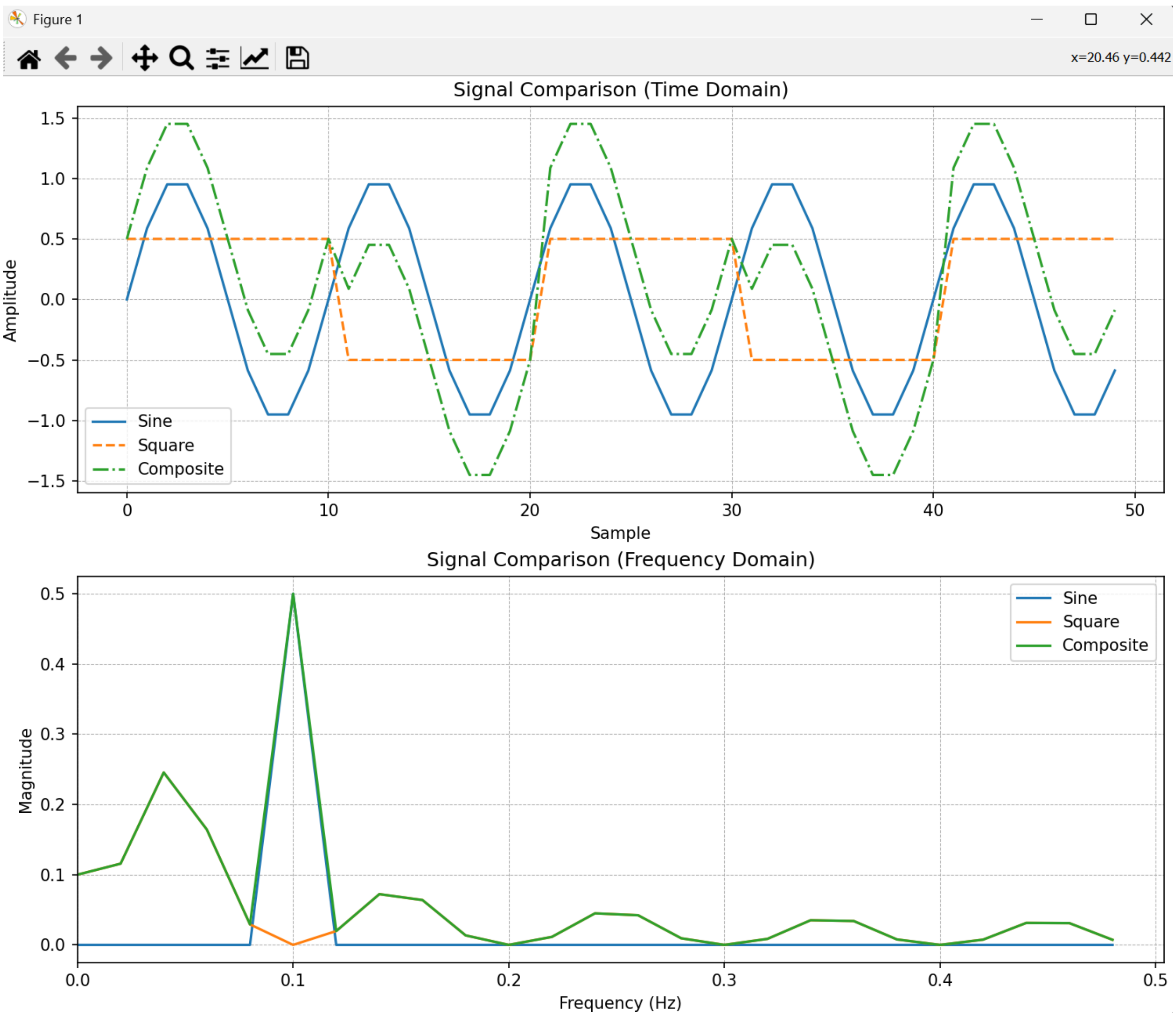
Organization of the code was almost identical to the previous code for sine and square waves. The only difference was in firstly __init__ by ensuring that the composite wave was initialized with at least one wave so that it wouldn't have a nonexistent wave, and secondly in __call__ by adding a secondary for loop to loop through the inputs that need to be composited into a single wave. This allowed for waves to be added together to form the new composite wave.

```python
from SignalProcessor import SignalProcessor

class CompositeSignalFunction(SignalProcessor):
    def __init__(self, inputs):
        super().__init__([])
        if not inputs:
            raise ValueError("CompositeSignalFunction requires at least one input signal.")
        self.inputs = list(inputs)

    def __call__(self, duration):
        if duration < 0:
            raise ValueError("duration must be non-negative")

        # Ensure each input signal has data for the requested duration
        for signal in self.inputs:
            if len(signal) != duration:
                signal(duration)

        self.data = []
        for idx in range(duration):
            sample_total = 0
            for signal in self.inputs:
                sample_total += signal.data[idx]
            self.data.append(sample_total)
        print(self.data)

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

    def __iter__(self):
        self.idx = 0
        return self

    def __next__(self):
        if self.idx >= len(self.data):
            raise StopIteration

        value = self.data[self.idx]
        self.idx += 1
        return value

    def __eq__(self, other):
        if not hasattr(other, "data"):
            return NotImplemented

        if len(other) != len(self.data):
            raise ValueError("Two signals are not equal in length!")

        count = 0
        for i in range(len(self.data)):
            if abs(self.data[i] - other.data[i]) < 0.01:
                count += 1
        return count
```

Signal Visualization 3.2

Explanation

As seen below the waves were created for a duration time of 50 as specified by instructions. The composite sign becomes larger than the sine wave when the square wave and sine wave are both the same sign validating the functionality of the composite wave code by adding together the two waves.




```
[0.0, 0.5877852522924731, 0.9510565162951535, 0.9510565162951536, 0.5877852522924732, 1.2246467991473532e-16, -0.587785252292473, -0.9510565162951535, -0.9510565162951536, -0.5877852522924734,
-2.4492935982947064e-16, 0.5877852522924729, 0.9510565162951535, 0.9510565162951536, 0.5877852522924734, 3.6739403974420594e-16, -0.5877852522924728, -0.9510565162951534, -0.9510565162951538, -
0.5877852522924735, -4.898587196589413e-16, 0.5877852522924727, 0.9510565162951534, 0.9510565162951538, 0.5877852522924736, 6.123233995736766e-16, -0.5877852522924726, -0.9510565162951534, -0.9
510565162951538, -0.5877852522924737, -7.347880794884119e-16, 0.5877852522924725, 0.9510565162951533, 0.9510565162951539, 0.5877852522924738, 8.572527594031472e-16, -0.5877852522924725, -0.9510
565162951533, -0.9510565162951539, -0.5877852522924739, -9.797174393178826e-16, 0.5877852522924724, 0.9510565162951533, 0.9510565162951539, 0.587785252292474, 1.102182119232618e-15, -0.58778525
22924722, -0.9510565162951532, -0.951056516295154, -0.5877852522924741]
[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, -0.5, -0.5, -0.5, -0.5, -0.
5, -0.5, -0.5, -0.5, -0.5, -0.5, 0.5, 0.5651629515364, -0.08778525229247336, 0.4999999999999998, 0.08778525229247292, 0.45105651629515353, 0.45105651629515364, 0.08778525229247336, -0.499999999
651629515364, -0.08778525229247336, 0.4999999999999998, 0.08778525229247292, 0.45105651629515353, 0.45105651629515364, 0.08778525229247336, -0.4999999999999996, -1.087785252292473, -1.451056516
2951534, -1.4510565162951536, -1.0877852522924734, -0.5000000000000004, 1.0877852522924727, 1.4510565162951534, 1.4510565162951536, 1.0877852522924736, 0.5000000000000007, -0.08778525229247258,
 -0.4510565162951534, -0.45105651629515375, -0.08778525229247369, 0.4999999999999993, 0.08778525229247247, 0.4510565162951533, 0.45105651629515386, 0.0877852522924738, -0.4999999999999917, -1.
0877852522924725, -1.4510565162951532, -1.4510565162951539, -1.0877852522924738, -0.5000000000000001, 1.0877852522924725, 1.4510565162951532, 1.4510565162951539, 1.087785252292474, 0.50000000000
00011, -0.08778525229247225, -0.4510565162951532, -0.451056516295154, -0.08778525229247414]
```

Signal Visualization Code

```python
import numpy as np
import matplotlib.pyplot as plt

from SineWaveFunction import SineWaveFunction
from SquareWaveFunction import SquareWaveFunction
from CompositeSignalFunction import CompositeSignalFunction


def generate_signals(duration):
    sine = SineWaveFunction(amplitude=1.0, frequency=0.1)
    square = SquareWaveFunction(amplitude=0.5, frequency=0.05)
    composite = CompositeSignalFunction(inputs=[sine, square])
    composite(duration)

    return {
        "Sine": np.array(sine.data, dtype=float),
        "Square": np.array(square.data, dtype=float),
        "Composite": np.array(composite.data, dtype=float),
    }


def compute_fft(signal, sample_spacing):
    n = len(signal)
    freqs = np.fft.fftfreq(n, d=sample_spacing)
    fft_vals = np.fft.fft(signal)
    mask = freqs >= 0
    return freqs[mask], np.abs(fft_vals)[mask] / n


def plot_time_domain(ax, time_axis, signals):
    for label, data in signals.items():
        if label == "Sine":
            ax.plot(time_axis, data, label=label, color="tab:blue", linestyle="-")
        elif label == "Square":
            ax.plot(time_axis, data, label=label, color="tab:orange", linestyle="--")
        elif label == "Composite":
            ax.plot(time_axis, data, label=label, color="tab:green", linestyle="-.")
        else:
            ax.plot(time_axis, data, label=label)

    ax.set_title("Signal Comparison (Time Domain)")
    ax.set_xlabel("Sample")
    ax.set_ylabel("Amplitude")
    ax.legend()
    ax.grid(True, which="both", linestyle="--", linewidth=0.5)

def plot_frequency_domain(ax, signals, sample_spacing):
    for label, data in signals.items():
        frequencies, magnitude = compute_fft(data, sample_spacing)
        ax.plot(frequencies, magnitude, label=label)
    ax.set_xlim(left=0)
    ax.set_title("Signal Comparison (Frequency Domain)")
    ax.set_xlabel("Frequency (Hz)")
    ax.set_ylabel("Magnitude")
    ax.legend()
    ax.grid(True, which="both", linestyle="--", linewidth=0.5)


def main():
    duration = 50
    sample_spacing = 1.0
    time_axis = np.arange(duration) * sample_spacing
    signals = generate_signals(duration)

    fig, (ax_time, ax_freq) = plt.subplots(2, 1, figsize=(10, 8), constrained_layout=True)
    plot_time_domain(ax_time, time_axis, signals)
    plot_frequency_domain(ax_freq, signals, sample_spacing)
    plt.show()


if __name__ == "__main__":
    main()
```