ECE 60146 and BME 64600: Homework 1

Spring 2026
Due Date: Tuesday, Jan 20, 2026, 11:59 pm ET
TA: Somosmita Mitra (mitra26@purdue.edu)

Turn in typed solutions via Gradescope. Additional instructions can be found at the end. Late submissions will be accepted with penalty: -10 points per-late-day, up to 5 days.

1 Introduction

This assignment focuses on Python Object-Oriented Programming (OOP), which forms the foundation for working with PyTorch.

In future homeworks, you will work extensively with PyTorch classes, either using them directly or extending them through inheritance. Consider this assignment as establishing the conceptual framework upon which all subsequent work will build.

This homework is based on the most elementary concepts in signal processing. If you know what a sine function is, you should not have any conceptual problems understanding what this homework is asking you to do.

Note: Use Python 3.x (not Python 2.x) for this and all future assignments. For additional background on Python OOP, refer to Prof. Kak's Week 1 lecture slides [1].

2 Programming Tasks (100 points)

The following tasks are designed to be completed sequentially, with each building upon the previous implementation.

1. **Creating the Base Class (10 points)** Create a base class that serves as the foundation for signal functions:

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

The `data` parameter should be a list of numerical values (*e.g.*, `[0, 1, 2, 3, 4]`). This class serves as the parent class from which specialized signal functions will inherit common characteristics.

2. **Creating the Sine Wave Function (10 points)** Create a specialized signal function as a subclass of SignalProcessor:

```python
class SineWaveFunction(SignalProcessor):
    def __init__(self, amplitude, frequency):
        # Your implementation here
        pass
```

Implement inheritance by calling the parent class constructor and storing the two signal parameters. A sine wave is characterized by two properties: *amplitude* (signal magnitude) and *frequency* (oscillation rate).

3. **Making Objects Callable (10 points)** Implement the callable interface for the SineWaveFunction class. When invoked with a duration parameter, the object should generate a sine wave and store it in the `data` attribute. Use the following mathematical formula:

$$y[n] = \text{amplitude} \times \sin(2\pi \times \text{frequency} \times n)$$

where n ranges from 0 to `duration-1`. Expected behavior:

```python
import math
SG = SineWaveFunction(amplitude=2.0, frequency=0.1)
SG(duration=5) # Create and print the sine wave
print(len(SG)) # Output: 5
```

Implementation note: Implement the `__call__` and `__len__` methods. The generated sequence should be printed upon creation.

4. **Making Objects Iterable (10 points)** Implement the iterator protocol to enable iteration over signal values:

```python
SG = SineWaveFunction(amplitude=2.0, frequency=0.1)
SG(duration=5)
print([val for val in SG]) # Print all signal values
```

Implementation note: Implement `__iter__` and `__next__` methods. The `__iter__` method should return an iterator object (typically `self`), and `__next__` should return successive values, raising `StopIteration` when exhausted.

5. **Square Wave Function (20 points)** Implement a square wave function class. Square waves are binary signals that alternate between high and low values. Create a `SquareWaveFunction` class with functionality analogous to the sine wave. The square wave function logic: output positive amplitude when the corresponding sine value is non-negative, otherwise output negative amplitude:

$$y[n] = \begin{cases} \text{amplitude} & \text{if } \sin(2\pi \times \text{frequency} \times n) \geq 0 \\ -\text{amplitude} & \text{if } \sin(2\pi \times \text{frequency} \times n) < 0 \end{cases}$$

Expected behavior:

```
SW = SquareWaveFunction(amplitude=3.0, frequency=0.1)
SW(duration=5)
print(len(SW)) # Output: 5
print([val for val in SW]) # Output: square wave values
```

6. **Comparing Signals (20 points)** Implement signal comparison functionality. Due to floating-point precision limitations, implement element-wise comparison with a tolerance of 0.01. For signals of different lengths, raise a `ValueError` with the message `"Two signals are not equal in length!"`. Otherwise, return the count of elements that match within the specified tolerance.

```
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) # Output: number of matching elements

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

print(SG1 == SG3) # Raises ValueError
```

Ensure this functionality works for both sine and square wave functions. *Implementation note: Implement the `__eq__` method. Use `abs(a - b) < 0.01` for floating-point comparison.*

7. **Demonstrate Your Implementation (20 points)**

   Select appropriate parameter values to demonstrate your implementation. Create sine waves and square waves with varying amplitudes and frequencies. Include all outputs in your report and provide analysis of the observed behavior. Discuss how different parameters affect signal characteristics.

3 Bonus Section (20 points)

The following optional tasks provide additional challenges for advanced implementation:

1. **Composite Signal Function (10 points)** Practical signals often comprise multiple component signals. Implement a `CompositeSignalFunction` that accepts multiple signal functions and produces their element-wise sum:

```
SG = SineWaveFunction(amplitude=1.0, frequency=0.1)
SW = SquareWaveFunction(amplitude=0.5, frequency=0.05)
CSG = CompositeSignalFunction(inputs=[SG, SW])
CSG(duration=10) # Create sum of component signals
```

2. **Signal Visualization (10 points)** Create visualizations of your signal implementations:

   - Generate sine, square, and composite signals (minimum 50 samples each)
   - Plot all signals on a single figure using distinct colors and line styles
   - Create a frequency domain plot using FFT analysis
   - Include appropriate labels, legends, and titles
   - Add grid lines for improved readability

   This visualization will demonstrate the differences between signal types and their frequency domain characteristics.

4 Frequently Asked Questions

Q: Import errors occur when attempting to use mathematical functions. How should this be resolved?

A: Ensure the math module is imported at the beginning of your implementation: `import math`. Subsequently use `math.sin()`, `math.pi`, etc.

Q: Generated sine wave values do not match expected outputs exactly. Is this acceptable?

A: Minor variations due to floating-point precision are expected. The tolerance of 0.01 in the comparison function accommodates these differences. Values within this tolerance range are considered correct.

Q: Must all special methods (`__init__`, `__call__`, etc.) be implemented in every class?

A: No. Methods can be inherited from the parent class when appropriate. Consider which functionality each class requires and implement accordingly.

Q: Is NumPy permitted for this assignment?

A: While NumPy would simplify certain calculations, this assignment emphasizes understanding fundamental OOP concepts. Implement solutions using standard Python and the math module. NumPy may be used for the bonus visualization section.

Q: What level of detail is expected in the report?

A: Include well-commented source code, output from all example executions, results from your parameter selections, and explanations of your implementation approach. Describe your problem-solving process and any challenges encountered.

Q: The implementation of the iterator is challenging. Are there implementation suggestions?

A: An iterator must maintain its current position state. Consider adding an instance variable to track the current index in the data list. The `__next__` method should return the next item, advance the position, and raise `StopIteration`

when the sequence is exhausted.

Q: How does signal regeneration work on the same instance?

A: Each invocation of the function with a new duration parameter overwrites the previous data. This is the intended behavior for this implementation.

5 Submission Instructions

Adherence to the submission format is essential to prevent penalization during grading:

1. **PDF Report**: Create a report containing:
   - Complete, well-documented source code
   - Output from executing all provided example snippets
   - Results from your selected parameter values
   - Explanations of your implementation approach and observations

   File naming: hw1_<FirstName><LastName>.pdf
2. **Code Submission**: Submit Python source code as a .zip file named hw1_<FirstName><LastName>.zip. **Strict adherence to this naming convention is required, incorrect naming will result in a** 10 **point deduction.**
3. **Development Environment**: Jupyter notebooks (.ipynb) may be used for development and report generation, but final code submissions must be converted to .py files. **Direct submission of .ipynb files is not permitted.**
4. **Resubmissions**: Multiple submissions are permitted before the deadline, with each submission overwriting the previous version. **For late submissions, submit only once**. Multiple late submissions cannot be guaranteed to use the intended version for grading.
5. **Academic Integrity**: All code and report content must represent your individual work. Course materials and referenced tutorials may be consulted, but implementations must be original. Previous year solutions are for reference purposes only.

6. **Support Resources**: Questions should be posted on Piazza or addressed during office hours.

References

[1] Python OO for DL. URL https://engineering.purdue.edu/kak/pdf-kak/PythonOO.pdf.