420-SN1-RE Programming in Science - Lab Exercise 19

November 5, 2024

Introduction

In this lab you'll learn more about using `numpy` and vectorized operations. Goals for this lab:

- Perform simple `numpy` operations.
- Create a graph using normal lists.
- Create the same graph using vectorized operations.

Files distributed with this lab:

- `lab19.pdf` - These lab instructions.

Exercise 0

For this part, you do *not* have to submit anything. You can do all of the work in the IDLE shell.

1. Import the `numpy` package and create a simple linear space between -1 and +1.

   ```
   import numpy as np
   x = np.linspace(-1, 1)
   ```

2. Check the type of `x`. It should be `numpy.ndarray`.

3. A `linspace` is just an evenly-spaced series of values including both the start and stop. By default the number of values is 50. You can see the number of dimensions and the *shape* of the `ndarray` by typing:

   ```
   print(x.ndim)
   print(x.shape)
   ```

4. You can check the *element* type of an `ndarray` by printing the `dtype` attribute:

   ```
   print(x.dtype)
   ```

5. In many ways, a `numpy.ndarray` is similar to a Python list. You should be able to use the following functions, for example:

   ```
   len(x)
   min(x)
   max(x)
   sum(x)
   ```

6. You can create a `ndarray` from a nested list:

```
y = np.array([[10, 11], [12, 13], [14, 15]])
print(y.ndim)
print(y.shape)
print(y.size)
print(y.dtype)
```

7. You can access items using indexing with square brackets, exactly like a regular `list`:

   ```
   print(y[1][1])
   ```

   Unlike a regular `list`, a `ndarray` also allows the row and column to be listed in a single pair of square brackets:

   ```
   print(y[1, 1])
   ```

8. Unlike a regular nested `list`, you can take the sum of a `ndarray`. However, the result may not be what you expect:

   ```
   print(sum(y))
   ```

9. There is a `sum()` method that adds all of the elements of the `ndarray`:

   ```
   print(y.sum())
   ```

10. The `ndarray` type is mutable:

    ```
    y[0,0] = 9
    ```

    However, the `ndarray` cannot change size once it is created.

11. You can use slices to extract sections of a `ndarray`:

    ```
    print(y[1:, :])
    print(y[:, 1:])
    ```

12. As with `pandas DataFrame`, you can select elements in an `ndarray` with a Boolean expression:

    ```
    print(y[y % 3 == 0]) # elements divisible by 3
    ```

13. Boolean operators can be combined with `&` for `and` and `|` for `or`.

    ```
    print(y[(y > 10) & (y % 3 == 0)])
    ```

14. Arithmetic operators will work with `ndarray` objects, as long as they have the same shape:

    ```
    z = np.array([[1, 2], [3, 4], [5, 6]])
    print(y + z)
    print(y - z)
    print(y * z)
    print(y / z)
    ```

15. Arithmetic operators will also work with `ndarray` objects and a scalar:

    ```
    print(y - 2)
    print(y * 3)
    print(100 // y)
    ```

Exercise 1

A *Taylor polynomial of degree n centred at* $x = a$ for a function $f(x)$ that can be differentiated n times can be written as follows:

$$T_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots + \frac{f^{(n)}(a)}{n!}(x-a)^n$$

The sine function, in particular, can be approximated as a simplified Taylor polynomial as follows:

$$T_{2m+1}(x) = \sum_{k=0}^{m} \frac{(-1)^k}{(2k+1)!}x^{2k+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{(-1)^m}{(2m+1)!}x^{2m+1}$$

The subscript to T indicates the highest degree of the Taylor polynomial, so if $m = 3$ we could evaluate the first four $(m + 1)$ terms:

$$T_7(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

In this exercise, you will use standard Python lists to create a graph illustrating how the approximation of the sine function improves as the value of m increases.

Create a new file named `lab19ex1.py` that does the following:

1. Create a Python function `T(x, m)` to evaluate the series at a point x by summing the first $m + 1$ terms in the series. Remember that the upper limit m in "big-sigma" notation is an *inclusive* limit. Also note that there is a `factorial()` function available in the standard `math` module.

2. Create a list to represent the x-axis of your graph, covering the range from -2π to $+2\pi$ (inclusive), evenly spaced over 100 elements.

3. Create a matplotlib plot showing the "true" sine function over this range, as well as the Taylor series approximations for $m = 0$, 1, 2 and 3. Because you are using standard Python lists, you will *have* to use a loop to create the values for the y-axis:

   ```
   y = []
   for v in x:
     y.append(sin(v))
   ```

4. In your graph, be sure to limit the y-axis to the range -2,+2. This can be done with the `matplotlib` function `plt.ylim()`:

   ```
   plt.ylim([-2, 2])
   ```

5. Add an appropriate legend and title to your graph.

6. Your figure should look like Figure 1.

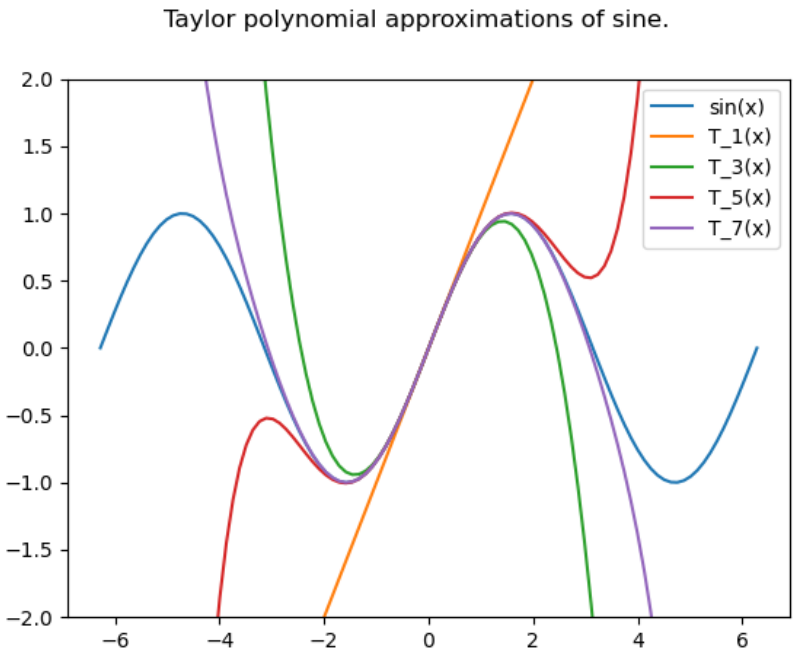

Figure 1: The figure for exercise 1

Exercise 2

Now you will recreate the same graph using `numpy` vectorized operations. Using vectorized operations will make your code shorter by avoiding the need for loops.

In a new file named `lab19ex2.py`:

1. Use the `linspace` function to create the values for your x axis.
2. There are vectorized versions of many standard functions available in `numpy`. To create the plot of the sine function, you can write the following:

   ```
   y = np.sin(x) # x should be a numpy.ndarray
   ```

3. If you have written your `T(x, m)` function correctly, it should *automatically* work when `x` is a `ndarray` supporting vectorized operations. You should be able to write a single line like:

   ```
   y = T(x, 0)
   ```

   to create the approximation for $m = 0$, where both `x` and `y` will be `ndarray` objects.

4. You should create another graph identical to Figure 1.

Exercise 3 (OPTIONAL)

The exponential function $y = e^x$ also has a Taylor polynomial equivalent:

$$T_n(x) = \sum_{k=0}^{k} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

Using `numpy`, write a Python program `lab19ex3.py` that creates a graph similar to Figure 2 for this series.

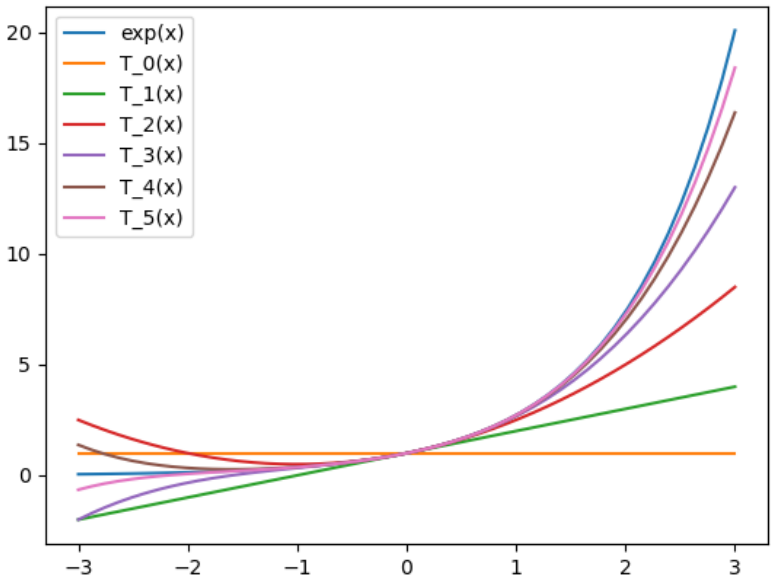

Figure 2: The figure for exercise 3

Submission requirements

Combine your Python files into a single zip file and upload them to Omnivox. Be sure to submit a *single* file containing all of your work.