HOMEWORK 6

Due Thursday, March 2, at 11pm

Please enter your answers into a Jupyter notebook and submit by the deadline via canvas.

Fake Goldbach. More accurately: Goldbach's wrong conjecture. It was proposed by Goldbach that every odd number can be written as the sum of a prime and twice a square. For example:

$$\begin{aligned} 7 &= 7 + 0^2 \\ 9 &= 7 + 2 \times 1^2 \\ 11 &= 11 + 0^2 \\ 15 &= 7 + 2 \times 2^2 \\ 21 &= 3 + 2 \times 3^2 \\ 25 &= 7 + 2 \times 3^2 \\ 27 &= 19 + 2 \times 2^2 \\ 33 &= 31 + 2 \times 1^2 \end{aligned}$$

Prove Golbach wrong by finding the first odd number not to be a prime plus twice a square.

Random walk. A drunk bear called Randi is standing on the origin in $\mathbb{R}$. At each time step, he goes 1 unit to the left with probability $p = 0.5$ and 1 unit to the right with probability $1 - p = 0.5$. Say each random walk is of length $M = 30$ (at which point Randi collapses to the ground). An example simulation of Randi's walk would be `[1,0,1,2,1,0,-1,-2,...,-3]`.

- Make a numpy array of shape (1000, 30) that stores the result of 1000 simulated random walks.
- Compute the mean and standard deviation of the ending point of Randi's walk using `np.mean` and `np.std`.
- Make a histogram of where we will find Randi at the end of his walk.
- Let r_M be the ratio of walks where, *at any point during the walk*, Randi returned to the origin. For $M = 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots, 100$, compute r_M and make a graph of r_M as a function of M. (idea: you don't need to make a new numpy array every

time, make one numpy array of shape (1000,100) and take subarrays to do your computation)

Polynomials class v2.0. In the previous homework, we designed a class for Polynomials (please check last week's solutions if you are not sure about this). Our class supported initialization by list (i.e. `__init__(self, xs)`), printing (i.e. `__repr__(self)`), addition (`__add__(self, other)`), and evaluation (`eval(self, x)`).

Fill in the deleted parts of the code below to add the following functionality to Polynomial. For `p = Polynomial([1.0, 2.0, 0.0])` corresponding to the polynomial $1 + 2x^2 + 0x^3 = 1 + 2x2$.

- `p.cleanup()`, should modify `p` by removing the unnecessary 0-coefficients at the end of `p` (assume a float is zero if it is less than `epsilon = 0.00000000001`). For example, `p = Polynomial([1.0,1.0,0.0000000000000000000000000001])`, `p.cleanup()`, `print(p)` should print `1.0 + 1.0x^1`.
- `p.degree()` should return the degree of p. For example, `Polynomial([1.0, 0.0, 0.5]).degree()` should be 2. Be careful: `Polynomial([1.0, 0.0, 0.0, 0.00000000000000000001]).degree()` should be zero.
- `power_of_x(n)`, should return x^n as a `Polynomial`. (this function should be outside the class) For example `print(power_of_x(4)) should print 1.0x^4` (or similar depending on how you implemented print before)
- `p == q`, should return `True` if all the coefficients of `p` and `q` are within epsilon of each other, `False` otherwise. You do this by implementing a method called `__eq__(self, other)` in the Polynomial class.
- `p * q`, should return the product of two polynomials. You do this by implementing `__mul__(self, other)` within the class.
- `p.derivative()`, should return the derivative of p(x).
- `p.integral(a, b)`, should return the integral of p(x) from `a` to `b`.
- (optional) `p.compose(q)` should return the composition of p and q. i.e. the resulting polynomial should be $p(q(x))$.

  Make sure you test each method with a couple of examples.

```
class Polynomial():
    def __init__(self, xs):
      self.coeffs = xs
    def __repr__(self):
      # last homework

    def __add__(self, other):
      # last homework

    def eval(self, x):
```

```
      # last homework

    # removes 0 coefficients in high degrees
    # e.g. p = Polynomial([1., 0., 2., 0., 0.])
    #      p.cleanup()
    #      print(p)
    # should give: 1.0x^0 + 0.0x^1 + 2.0x^2
    def cleanup(self):
      pass     # pass prevents python from error
               # because function def is empty

    # returns degree of poynomial (be careful of extra 0's in high degrees)
    def degree(self):
      pass


    # checks if self and other have all coefficients within 10**(-11) of each other
    def __eq__(self, other):
       pass

    # scalar multiplies polynomial by number (modifies polynomial)
    def scalar_mult(self, alpha):
       pass

    # returns the product polynomial of self and other
    def __mul__(self, other):
       pass

    # returns the derivative of the polynomial with respect to x
    def derivative(self):
      pass

    # returns the integral of the polynomial from a to b
    def integral(self, a, b):
      pass

    # optional. returns the composition p(q(x))
    def compose(self, other):
      pass


# returns the nth power of x as a polynomial
def power_of_x(n):
  pass
```