MATH 157: Mathematics in the world
Homework 5 (Due March 20th, 2017 at 4:00PM)

PLEASE SOLVE COMPLETELY THE FOLLOWING 5 PROBLEMS

Problem 1

Amy has $n+1$ fair coins, while her brother Brad only has n. If both flip all of their coins, what is the probability that Amy will end up with more heads than Brad?

Problem 2

100 passengers are waiting in line to board an airplane, everybody with a ticket and a seat assigned. For simplicity, let's say the i-th person in line is assigned the seat number i. The first passenger in line lost the ticket though, and when boarding the plane just picks a seat on the plane randomly. All the following passenger will then go to their proper seat, unless it is already occupied; in such case, they will randomly pick a seat among all the ones left. What is the probability that the last person in line will seat in the seat number 100?

Problem 3

Suppose we have a coin which comes up heads with probability $p \in (0,1]$ consistently. We start flipping the coin until the first time we get heads. We will use X to denote the total number of flips we performed. Such a random variable is called *geometrically distributed*.

1. Given an integer $k \geq 1$, what is the probability $\mathbb{P}(X = k)$?
2. Express the expectation of X as an infinite sum, and deduce that $\mathbb{E}(X) = 1/p$. [1]
3. Show that [2]
$$\sum_{k\geq 1} k^2 x^k = \frac{x(x+1)}{(1-x)^3}.$$
4. Use the series in the previous part to prove that $\mathbb{E}(X^2) = (2-p)/p^2$.

[1] Hint: Use the derivative of $1/(1-x)$. You may also find the notes on generating functions helpful.
[2] Hint: Use the first two derivatives of $1/(1-x)$.

5. The *variance* of a random variable Y is defined as

$$\mathrm{Var}(Y) = \mathbb{E}\left[(Y - \mathbb{E}(Y))^2\right].$$

   Simplify this expression to arrive at [3]

$$\mathrm{Var}(Y) = \mathbb{E}(Y^2) - \mathbb{E}(Y)^2,$$

   and deduce that

$$\mathrm{Var}(X) = \frac{1-p}{p^2},$$

   where X is as above.

Problem 4

Imagine we have a coin which comes up heads with probability $p \in [0,1]$ consistently. We flip this coin $n \geq 1$ times, and let X be the number of heads we obtain. We say that X is a *binomially distributed* random variable.

1. Given an integer $0 \leq k \leq n$, what is the probability $\mathbb{P}(X = k)$?
2. Show that [4]

$$\sum_{k=0}^{n} k \binom{n}{k} x^k y^{n-k} = nx(x+y)^{n-1},$$

   and deduce that $\mathbb{E}(X) = np$.
3. An alternative, and arguably much easier, way to compute the expectation of X uses the additivity of expectation. Let X_k be 1 if the k-th flip came up heads, and 0 otherwise. Use $X = \sum_{k=1}^{n} X_k$ to deduce that $\mathbb{E}(X) = np$.
4. Show that $\mathbb{E}(X_k^2) = p$ for all $1 \leq k \leq n$, and $\mathbb{E}(X_k X_\ell) = p^2$ for all $1 \leq k < \ell \leq n$.
5. Conclude that [5]

$$\mathbb{E}(X^2) = np(1 - p + np), \qquad \mathrm{Var}(X) = np(1-p).$$

[3] Note that expectation is linear, that is, $\mathbb{E}(c_1X_1 + c_2X_2) = c_1\mathbb{E}(X_1) + c_2\mathbb{E}(X_2)$ for any random variables X_1, X_2 and constants c_1, c_2. Furthermore, the expectation of a constant c is also c: $\mathbb{E}(c) = c$.

[4] Hint: Differentiate $(x+y)^n$ with respect to x and multiply the result by x.

[5] Alternatively, we can use the fact that variance is additive for *uncorrelated* random variables. This is called the Bienaymé formula.

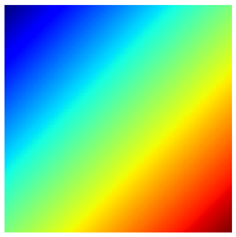
In fact, the X_k are *independent*, which is a stronger notion.

Problem 5

In this problem, we will learn how to draw the Julia set using Python. First, we demonstrate drawing a simple gradient image.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

## This function controls the image value at position x, y
f = lambda x, y: x + y

## Form a matrix of the values of f
a = np.fromfunction(np.vectorize(f), (100, 100))

## Save the result to an image
mpimg.imsave("gradient.png", a)

## Show the image
plt.imshow(a)
plt.show()
```

The result is the following image saved in `gradient.png`. [6]



Second, we should explain how to use complex numbers in Python. Fortunately, this is very easy and requires no additional setup. For example, the number $1 + 2i$ is written as `1 + 2j`. All basic arithmetic functions carry over to complex numbers automatically. If you are interested in using some of the more advanced mathematical functions which typically reside in the `math` module (e.g., `log`, `exp`), you should instead use the `cmath` module.

```
import cmath
```

[6] The gradient matrix we constructed has values between 0 and 198. To choose actual RGB colors, matplotlib uses what is called a colormap. In this case, the lowest values are blue, and the highest values are red. You can experiment with different colormaps by adding an argument `cmap` to `mpimg.imsave` or `plt.imshow`. For example, try `plt.imshow(a, cmap='hot')`. See `http://matplotlib.org/examples/color/colormaps_reference.html` for a complete list of the built in colormaps.

```
z = 1 + 1j

## 2j
print(z**2)

## (1.46869393992+2.28735528718j)
print(cmath.exp(z))
```

It is important to understand that functions are first rate objects in Python. They can be passed to other functions as arguments or returned as output. Consider the following example.

```
# This function returns the function "addition by x"
def addition_by(x):
    return lambda y: x + y

## Given a function f and an input value x, this function prints f(x)
def print_value_of_function(f, x):
    print(f(x))

## Prints 11
print_value_of_function(addition_by(10), 1)
```

We are now ready to describe the fractal images we plan to construct. [7] Given a complex number $c \in \mathbb{C}$, consider the function $f_c \colon \mathbb{C} \to \mathbb{C}$ given by $f_c(z) = z^2 + c$. [8] The image we would like to generate shades the point corresponding to a complex number z according the behavior of the recursive sequence

$$z_0 = z, \qquad z_{n+1} = f_c(z).$$

Depending on the starting value z, the sequence may converge to 0, diverge, or even stabilize to a fixed value.

Since infinite sequences are hard to study computationally, we need to simplify our problem. First, we choose a threshold $T > 0$ (a real constant). Given an starting point z, we will construct the sequence z_n looking for elements satisfying $|z_n| > T$ (imagine this corresponds to divergence). We will color z according to the smallest value n such that $|z_n| > T$. To avoid a potentially infinite loop when z_n converges to 0, we will use a value $n_{\max}$ for starting points z such that $|z_n| \leq T$ for all $n < n_{\max}$.

1. Our fist task is to translate between the image coordinates and complex numbers. Imagine we would like to use a canvas of size $x_{\max}$ pixels by $y_{\max}$ pixels which should

[7] If you are interested in learning more about this topic, see `http://en.wikipedia.org/wiki/Julia_set` and `http://en.wikipedia.org/wiki/Mandelbrot_set`.

[8] In Python, that is `f_c = lambda c: (lambda z: z**2 + c)`.

be mapped to the complex rectangle

$$\{a + bi \mid a_{\min} \leq a \leq a_{\max}, b_{\min} \leq b \leq b_{\max}\}.$$

Write a function `complexify(x, y, xmax, ymax, amin, amax, bmin, bmax)` returning the complex number $a + ib$. [9]

2. Write a function `compute_value(z, f, T, nmax)` which given a function f (imagine $f = f_c$ for some $c \in \mathbb{C}$), computes the value n associated with z, the threshold T, and $n_{\max}$. See above for a detailed description of this process. [10]

3. Write a function

   `draw_fractal(f, filename, xmax, ymax, amin, amax, bmin, bmax, T, nmax)`

   which draws the fractal corresponding to the function f (and the rest of the auxiliary arguments), and saves it in the file given by `filename`.

4. Use `draw_fractal` to construct four images given by the following parameters. [11]

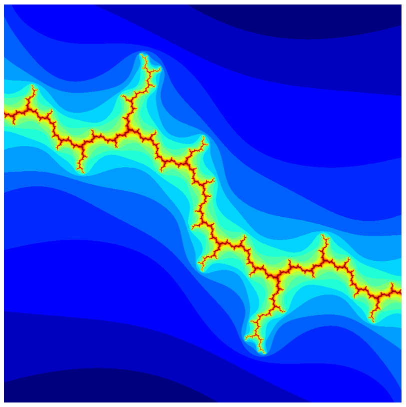
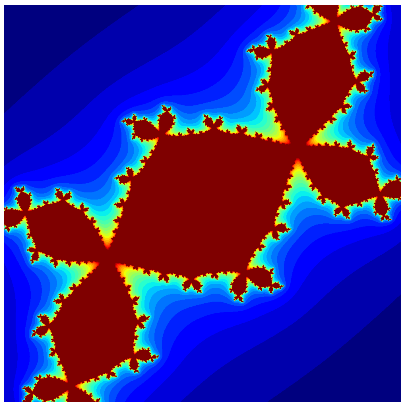
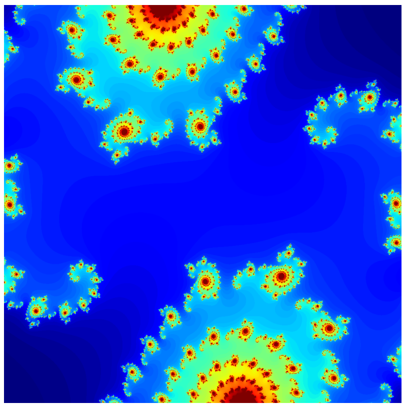
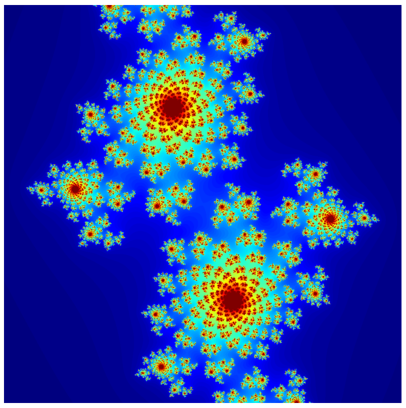
f	$x_{\max}$	$y_{\max}$	$a_{\min}$	$a_{\max}$	$b_{\min}$	$b_{\max}$	T	$n_{\max}$
$f_c,\ c = 1i$	500	500	-0.1	0.1	-0.1	0.1	2	25
$f_c,\ c = -0.123 + 0.745i$	500	500	-1	1	-1	1	100	30
$f_c,\ c = -0.75 - 0.2i$	500	500	-0.5	0.5	-0.5	0.5	10	100
$f_c,\ c = -0.75 - 0.3i$	500	500	-1	1	-1	1	10	130

[9] This may sound confusing due to the large number of variables involved, but it is quite simple actually. First, you can break the problem in two parts: determine a from x, $x_{\max}$, $a_{\min}$, $a_{\max}$, and likewise for b. If we focus on computing a, the task at hand is mapping the possible values $x = 0, \ldots, x_{\max} - 1$ equidistantly within the interval $[a_{\min}, a_{\max}]$.

[10] While the functions f_c produce many fascinating images, we could also use higher degree polynomials as well as other trigonometric or transcendental functions. We are implementing `compute_value` in a generic manner, so we can potentially use it to plot other types of fractals.

[11] On my computer, it takes about 5s to construct a 500×500 image. If you are trying to debug your code, try using a lower resolution first (e.g., 100×100) until you get everything working.

It is worth noting that our implementation is easy to understand but rather inefficient. The simplest way to make it faster is by modifying the functions `compute_value` and `complexify` to operate on matrices rather than single values. This operation is called vectorization and is one of the main tricks of modern high-performance computing. We were able to cheat numpy that our functions are vectorized using `np.vectorize`.

5. (Extra credit) Draw up to 5 other fractals. Try using higher degree polynomials, trigonometric functions, or other color maps. **Be creative!**