1. McDonalds recently negotiated a large purchasing deal for fish, chicken, and beef. They have agreed to purchase 40 million tons of fish, 40 million tons of chicken, and 100 million tons of beef. As such, they want to create an advertising campaign to ensure that consumers eat the correct portion of each meat product.

   After paying to have a commercial produced, McDonalds collects the following data: After watching the commercial once, a person who initially wanted fish now has a 10% chance of buying a fish product, a 60% chance of buying a beef product, and a 30% chance of buying a chicken product; after watching, a person who initially wanted to buy a beef product has a 20% chance of buying a fish product, a 60% chance of buying a beef product, and a 20% chance of buying a chicken product; after watching, a person who initially wanted to by a chicken product has a 40% chance of buying a fish product, a 50% chance of buying a beef product, and a 10% chance of buying a chicken product.

   (a) If the vector $\vec{e}_1$ represents a person who wants to buy a fish product, $\vec{e}_2$ represents a person who wants to buy a beef product, and $\vec{e}_3$ represents a person who wants to buy a chicken product, find a matrix M such that $M\vec{e}_i$ gives the probability of buying fish, beef, or chicken after watching the commercial once.

   (b) Compute the eigenvalues and eigenvectors of M.

   (c) Assume that a fish product takes 50 grams of fish, a beef product takes 50 grams of beef, and a chicken product takes 50 grams of chicken. Further, assume that each time a person watches the commercial it has the same impact (i.e., watching the commercial twice means the likelihood of buying a particular product is given by M^2). If McDonalds ensures that the average customer sees the commercial 3000 times, what are the relative proportions of fish, beef, and chicken McDonalds expects to sell?

   (d) Should McDonalds run the ad? Does the initial population's preferences for fish, beef, or chicken matter? *Explain your reasoning.*

2. Throughout this problem, let P be an $n \times n$ *stochastic matrix*.

   (a) Prove that if $\vec{q}$ is a *probability vector*, then $P\vec{q}$ is also a probability vector.

   (b) Prove that P^k is a stochastic matrix for all $k \geq 0$.

   (c) A *left eigenvector* for P is a non-zero row vector $\vec{w}$ so that $\vec{w}P = \lambda\vec{w}$.
   Does P have a left eigenvector with eigenvalue 1? Why or why not?

   (d) Show that the left and right eigenvectors of P may be different but the left and right eigenvalues of P must be the same. *Hint: you may use facts from linear algebra about determinants and transposes.*

   (e) The *unit* $(n-1)$*-symplex* is the set of all convex linear combinations of $\vec{e}_1, \ldots, \vec{e}_n$. Let S be the unit $(n-1)$ symplex, and let $\mathcal{P} : \mathbb{R}^n \to \mathbb{R}^n$ be the linear transformation defined by $\mathcal{P}(\vec{x}) = P\vec{x}$. Show that $\mathcal{P}(S) \subseteq S$. *Hint: start by showing that vectors in S are probability vectors.*

   (f) The Brouwer Fixed-point Theorem states that a continuous map from a simplex into itself has at least one fixed point. Use the Brouwer Fixed-point Theorem to show that P has at least one (right) eigenvector with eigenvalue 1 *which is also a probability vector.*

3. We're going to prove some linear algebra facts because, *just maybe* they'll be useful.

   Let P be an $n \times n$ stochastic matrix and let $\mathcal{P} : \mathbb{R}^n \to \mathbb{R}^n$ be the linear transformation induced by P (i.e., given by matrix multiplication). Let S be the unit $(n-1)$-symplex.

   (a) Draw S when $n = 1$, 2, and 3.

   (b) Prove that S is equal to the set of all probability vectors in $\mathbb{R}^n$.

   (c) Prove that if $\vec{p}, \vec{q} \in S$, then all convex linear combinations of $\vec{p}$ and $\vec{q}$ are in S.

(d) For the rest of this problem, assume $n \geq 2$.
The *boundary* of S, written ∂S, consists of all vectors in S where at least one coordinate is zero.
Let $\vec{a}, \vec{b} \in S$ be distinct points and let $\ell \subseteq \mathbb{R}^n$ be the line passing through $\vec{a}$ and $\vec{b}$. Prove that ℓ intersects the boundary of S.

(e) Prove that if $V \subseteq \mathbb{R}^n$ is a subspace of *dimension at least two*, then the following holds: if $V \cap S$ is nonempty, then $V \cap \partial S$ is non-empty.

(f) Prove that if $\vec{a}, \vec{b} \in S$ are distinct eigenvectors for P with eigenvalue 1, then there exists a $\vec{d} \in \partial S$ which is an eigenvector for P with eigenvalue 1.

4. Let $\mathcal{M} = (M_0, M_1, \ldots)$ be a stationary Markov chain on a graph $\mathcal{G}$ with n vertices, and let P be the (stochastic) transition matrix for $\mathcal{M}$. Further, suppose $\mathcal{M}$ is modeled by the dynamical system (T, Ω), where Ω is the space of probability distributions on the n vertices.

(a) Produce examples where $\lim_{k\to\infty} P^k$ exists and does not exist. Can you find conditions on $\mathcal{M}$ and $\mathcal{G}$ so that $\lim_{k\to\infty} P^k$ always exists?

(b) A *stationary distribution* for $\mathcal{M}$ is defined to be a fixed-point of (T, Ω). Produce examples where $\mathcal{M}$ has exactly 1, 2, and 3 stationary distributions.

(c) Prove that a convex linear combination of stationary distributions for $\mathcal{M}$ is a stationary distribution for $\mathcal{M}$.

(d) Prove that $\mathcal{M}$ always has *at least one* stationary distribution.

(e) A Markov chain is called *primitive* if there exists a $k \in \mathbb{N}$ such that the probability of transitioning from state i to state j in exactly k steps is positive *for every* i *and* j.
A distribution $\vec{d} \in \Omega$ is said to have *full support* if none of the entries in $\vec{d}$ are zero.
Show that if $\mathcal{M}$ is primitive, then every stationary distribution for $\mathcal{M}$ must have full support.

(f) Prove that if $\mathcal{M}$ is primitive, then $\mathcal{M}$ has a *unique* stationary distribution.

(g) Show that if $\mathcal{M}$ is primitive and $\lim_{k\to\infty} P^k = P'$ exists, then $P' = [\vec{s}|\vec{s}|\cdots|\vec{s}]$, where $\vec{s}$ is the unique stationary distribution for $\mathcal{M}$.

Programming Problems

For the programming problems, please use the Jupyter notebook available at

https://utoronto.syzygy.ca/jupyter/user-redirect/git-pull?repo=https://github.com/siefkenj/2020-MAT-335-webpage&subPath=homework/homework2-exercises.ipynb

Make sure to comment your code and use "Markdown" style cells to explain your answers.

1. `np.random.rand()` will generate a random number chosen uniformly from the interval $[0, 1]$. That means, if $x, y \in [0, 1]$ and $x < y$, the probability that the output of `np.random.rand()` lies in $[x, y]$ is $y - x$.

(a) Using `np.random.rand()`, create a function `pick_random` which inputs a list of the form `[(`⟨*probability*⟩`,` ⟨*state*⟩`),...]` and returns one of the states chosen at random (but with the appropriate probability).

(b) Create a function `pick_random_n` which inputs a distribution and a number n and outputs n *samples* from that distribution.
Using the code provided, plot the theoretical (exact) distribution and the empirical (simulated) distribution.

2. A Python *dictionary* is an like an array except that it can be indexed by things other than numbers[1]. It's written with curly braces, colons, and commas. For instance, writing

[1] In general, such an object is called an *associative array*.

`x = {"a": 4, "b": 100}` sets `x` to be a dictionary with keys `"a"` and `"b"`. Executing `x["a"]` will return `4` and `x["b"]` will return `100`.

Dictionaries are an ideal object for storing *graphs* and consequently Markov chains. We will store a Markov chain in ⟨*state*⟩`:` ⟨*distribution*⟩ pairs listed in a dictionary.

(a) Create a function `step(`⟨*starting state*⟩`,` ⟨*chain*⟩`)` that inputs a starting state and a Markov chain and outputs the results of going one step along the chain.

(b) Create a function `n_orbit` that takes a starting state, Markov chain, and a number of steps n, and returns the n-step realization of that Markov chain.

(c) The `Counter` function will input a list and output a dictionary with the count of each item in that list. Using `Counter`, create a `make_dist` function which inputs a sequence of states and outputs the empirical distribution coming from those states[2]. Make sure the resulting distribution is *sorted in alphabetical order by state*[3].

Using `make_dist`, plot the empirical and theoretical steady state distributions for CHAIN1.

3. (a) Make a plot showing the empirical distributions arising from realizations of CHAIN2 starting at states `"a"`, `"b"`, …, etc.. Does the starting state affect the empirical distribution? Is this what you expect?

The `ensure_consistent_dists()` function can be used to ensure that a list of distributions share the same states so that they can be graphed together.

(b) Compute the stationary distribution for CHAIN2[4]. Does this match what you see from the empirical distribution?

(c) Instead of computing an empirical distribution from a single realization. Let's use a bunch of realizations! Suppose r_{100} represents the state at step 100 of a realization of CHAIN2. Using at least 10000 different realizations, plot the empirical distribution of the different r_{100}'s. Do the same for the r_{101}'s. Is this the same or different from the stationary distribution for CHAIN2? Is that what you expected?

(d) Repeat (a)-(c) for CHAIN3. Explain your results.

4. **Shakespeare vs. Lawyers**

(a) Execute the code which downloads the complete works of Shakespeare and the Ontario Criminal Code.

(b) Create a function `make_chain` which takes in a realization of a Markov chain and outputs a dictionary of transition probabilities (in the same format that CHAIN1, CHAIN2, and CHAIN3 are stored).

(c) Create Markov chains using the complete works of Shakespeare and the Ontario Criminal Code as "realizations".

(d) Using your Shakespeare and Ontario Criminal Code Markov chains, create length-30 realizations with the starting words "the", "sunset", and "satisfied"[5].

(e) The simple Markov chain's we've made don't pay attention to punctuation or sentence length. Using your human judgement, properly format the realizations of your Markov chains (as best you can). The sentences might not make complete English sense, but that's okay!

5. **Iterated Function Systems**

(a) Read and understand the first three cells in the *Iterated Functions* section.

[2] The *empirical* distribution of a sequence of states is the relative proportions of those states in the sequence.

[3] You may want to look up the `sorted` command in Python.

[4] You can use `np.linalg.eig` to compute eigenvectors and eigenvalues in Python. Read the documentation to figure out exactly what the return values are!

[5] You can use the command `" ".join(`⟨*list*⟩`)` to create a string consisting of the words in a list.

(b) The sample code takes a single realization of the iterated function system, applies it to the point $(1, 1)$, and plots it.

Make a plot of the distribution of this iterated function system by plotting the image of many points under many different realizations. What shape do you see?

(c) Using brightness, we can visualize distributions more easily. If you set `additive=True` in the `render_points_to_array()` function, a value of 1 will be added for each point that lands in a particular pixel. This means the more points that land in a pixel, the "brighter" it will look.

Create a new chain, `F_CHAIN2` which is the same as `F_CHAIN` but with a 1/4 chance of transitioning to `f1` and `f2` and a 1/2 chance of transitioning to `f3` (regardless of state). Plot the distributions of both `F_CHAIN` and `F_CHAIN2`. Are the results what you expect?

Hint: You may find it easier to visualize if you graph log-intensities instead of direct intensities. You could do this by applying `np.log(`⟨*density array*⟩ `+ 1)` on your data before plotting.

(d) Numpy can perform matrix multiplication with the `@` symbol. For example, `np.array([[1,2],[3,4]]) @ np.array([1,1])` will multiply the matrix $\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$ by the vector $\begin{bmatrix}1\\ 1\end{bmatrix}$. Many other linear algebra functions can be found by typing `np.linalg.`⟨*tab*⟩.

Create a new iterated function system, `ROT_CHAIN`, with the same functions as `F_CHAIN`, except that f_1 rotates the resulting vector counter-clockwise by 30° before returning it, and f_2 rotates the resulting vector 30° clockwise before returning it.

Plot the resulting distribution. Is it what you expected?

(e) Read about the Barnsley Fern https://en.wikipedia.org/wiki/Barnsley_fern. Create an iterated function system `BARNSLEY_CHAIN` whose maximal invariant set is the Barnsley Fern. Graph the maximal invariant set (or the distribution, whichever you please).

6. **Flam3** A very popular algorithm for generating fractals is the *flam3* algorithm[6]. The *flam3* algorithm consists of two parts: (a) functions to be used as part of an iterated function system, and (b) a rendering algorithm that produces smoothed-out graphics with varying color (instead of the pixely-looking fractals that we are rendering).

Look over the functions used in the *flam3* algorithm here: https://github.com/AlexanderJenke/FractalFlame/blob/master/functions.py Each function accepts an (x, y) pair and a number of parameters which can be set by the user.

Download and run JWildfire http://www.jwildfire.org which is a program that will let you interactively (and in real time) adjust the parameters of a *flam3* fractal[7].

For this question: *make a pretty fractal!*

If you want to start with something familiar, download and save https://raw.githubusercontent.com/siefkenj/2020-MAT-335-webpage/master/homework/basic_sierpinksi.flame Then load `basic_sierpinksi.flame` into JWildfire. You will then see three "linear3D" functions in the *Transformations* tab. Below, in the *Affine* tab, you can see the matrices that correspond to each transformation. If you click one of the triangles in the middle section, you can drag them around to interactively change the defining matrix. You can also middle click or right click to adjust the parameters in different ways. You can also have lots of fun by playing with the *Nonlinear* tab.

[6] It is pronounced *flame*, like the thing that comes out of a candle.
[7] If you're picky about software, you can use a different *flam3* renderer.