1. Let (T, X) be a dynamical system.

   (a) Prove that if $x \in X$ is a fixed point of T, then the *basin of attraction* of x, A_x, is non-empty.

   (b) Fix $y \in X$. Prove that if $R \subseteq A_y$, then $T^{-1}(R) \subseteq A_y$. (Recall $T^{-1}(R) = \{w \in X : T(w) \in R\}$ is the *inverse image of R under T*. T is **not necessarily** invertible.)

2. You will prove that under certain conditions, Newton's method converges.

   Let $f : \mathbb{R} \to \mathbb{R}$ be a twice differentiable function and suppose that $f(0) = 0$ and that f' and f'' are positive on the interval $(0, a]$. Let T_f be the function that applies one iteration of Newton's method to its input.

   (a) Find a function f satisfying the required properties and graph it. Does it look like Newton's method will converge on $(0, a]$?

   (b) Show that if $x \in (0, a)$, then $0 \leq T_f(x) < a$.

   (c) The *Monotone Convergence Theorem* states that if (a_n) is a sequence of real numbers that is bounded below *and* $a_{n+1} \leq a_n$, then $\lim_{n\to\infty} a_n$ exists and is a real number.

   Use the Monotone Convergence Theorem to prove that $\lim_{n\to\infty} T_f^n(x)$ converges for all $x \in (0, a]$.

   (d) A point y is called a *fixed point* of a function g if $g(y) = y$.

   Fix $x_0 \in (0, a]$, and define $\boldsymbol{x} = \lim_{n\to\infty} T_f^n(x_0)$. Show that $\boldsymbol{x}$ is a fixed point of T_f.

   Hint: You may use the fact that if g is a continuous function, then $g(\lim_{n\to\infty} a_n) = \lim_{n\to\infty} g(a_n)$ for any convergent sequence (a_n).

   (e) Prove that for any $x \in (0, a]$, $\lim_{n\to\infty} T_f^n(x) = 0$.

   (f) Combine your results and explain why Newton's method will always converge for f if you pick an initial guess in $(0, a]$.

3. Let's prove more about Newton's method! Suppose $f : \mathbb{R} \to \mathbb{R}$ is twice differentiable and $f(0) = 0$. Further, suppose f' and f'' are both positive on the interval $[-a, a]$ (for some $a > 0$).

   (a) Use a picture to argue that Newton's method might not converge for some $x \in [-a, a]$.

   (b) Show that there is some $b > 0$ so that for $x \in [-b, a]$, Newton's method always converges.

   (c) Let $g : \mathbb{R} \to \mathbb{R}$ and define $h : \mathbb{R} \to \mathbb{R}$ by $h(t) = g(-t)$. Prove that if Newton's method converges for g with a starting point of x_0, then Newton's method converges for h with a starting point of $-x_0$.

   (d) Prove that if $f : \mathbb{R} \to \mathbb{R}$ satisfies $f(0) = 0$, and $f' < 0$ and $f'' > 0$ on the interval $[-a, a]$, then there exists a $b > 0$ so that Newton's method converges for f on $[-a, b]$.

   (e) Prove that if $p : \mathbb{R} \to \mathbb{R}$ is a twice differentiable function satisfying $p(x_0) = 0$ and $p'(x_0), p''(x_0) \neq 0$, then there is an open interval about x_0 on which Newton's method will converge to x_0.

4. Let $f : \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = x^2 - 1$ and let T_f be the function that applies Newton's method to its inputs.

   (a) Find $T_f^{-1}(\{1, 2, 3\})$ (recall that $T_f^{-1}(\{1, 2, 3\})$ is the *inverse image* of the set $\{1, 2, 3\}$ under T_f. It is not the same as applying the *inverse* of T_f, since T_f might not be invertible).

   (b) Is $T_f(x)$ defined for all $x \in \mathbb{R}$?

   (c) Find the largest possible domain, $X \subseteq \mathbb{R}$, so that $T_f : X \to X$ is a *dynamical system.*

   (d) Prove that $\lim_{n\to\infty} T_f^n(4)$ exists. What value is it?

(e) Find all fixed points of T_f.

(f) For each fixed point of T_f, find its *basin of attraction*.

5. Let $f : \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = \frac{2}{1+x^2} - \frac{5}{4}$ and let T_f be the function that applies Newton's method to its inputs.

   (a) Is 0 in the domain of T_f?

   (b) Find $T_f^{-1}(\{0\})$ (you can use a computer algebra system).

   (c) Find $T_f^{-2}(\{0\})$ (you can use a computer algebra system).

   (d) Describe largest possible domain, $X \subseteq \mathbb{R}$, so that $T_f : X \to X$ is a *dynamical system*. How many connected components does this domain consist of? (Use computers to help you!)

6. Recall the *box-counting dimension* from the course notes. Throughout this problem, if we refer to the box-counting dimension of a set, you may assume that the box-counting dimension of that set exists (i.e., the limit in the definition converges to a finite number).

   (a) Prove that the line segment from $\vec{0}$ to $\sum \vec{e}_i$ in $\mathbb{R}^d$ has box-counting dimension 1.

   (b) In the definition of the box-counting dimension of $X \subseteq \mathbb{R}^m$, the set B is taken to be a minimal-dimensional, minimally-sized superset of X. Show that we can drop the both the assumption that B is minimally sized and that B is of a minimal dimension. (*Hint: prove these separately.*)

   (c) Prove that box-counting dimension is *translation invariant*. That is, if Y is a translation of $X \subseteq \mathbb{R}^m$, then X and Y have the same box-counting dimension.

   (d) Sets $X, Y \subseteq \mathbb{R}^m$ are called *box-disjoint* if there exists disjoint m-dimensional boxes A, B so that $X \subseteq A$ and $Y \subseteq B$. Prove that if $X, Y \subseteq \mathbb{R}^m$ are box-disjoint sets, then

   $$\dim(X \cup Y) = \max\{\dim(X), \dim(Y)\},$$

   where dim stands for the box-counting dimension.

   (e) Show that if $T : \mathbb{R}^m \to \mathbb{R}^m$ is a dilation given by $\vec{x} \mapsto k\vec{x}$ for some $k \in \mathbb{Z}^+$ and $X \subseteq \mathbb{R}^m$ is a set, then the box-counting dimension of X equals the box-counting dimension of $T(X)$.

   (f) Show that if $T : \mathbb{R}^m \to \mathbb{R}^m$ is a linear transformation, the box-counting dimension of $X \subseteq \mathbb{R}^m$ is bounded above by the box-counting dimension of $T(X)$.

   (g) Show that if $T : \mathbb{R}^m \to \mathbb{R}^m$ is an invertible linear transformation, then the box-counting dimension of $X \subseteq \mathbb{R}^m$ and $T(X)$ are equal.

   (h) (Optional) Show that if $\varphi : \mathbb{R}^m \to \mathbb{R}^m$ is a differentiable function with continuous derivative and $\det(D\varphi) \neq 0$, then the box-counting dimension of $X \subseteq \mathbb{R}^m$ and $\varphi(X)$ are equal.

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/homework1-exercises.ipynb

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

1. Implement Newton's method for the function $f(x) = x(x-2)(x-3)$.

   (a) Functions `f` and `f_prime` are already provided. Implement the functions `T_f` and `newt`. `T_f` inputs a guess and applies one iteration of Newton's method. `newt` should repeatedly apply Newton's method until a root is reached.

Since Newton's method may take an infinite number of iterations to converge, we won't require a point to actually reach a root. Instead, `newt` will input a *tolerance*. If $|f(x)| <$ *tolerance*, we will say that x is close enough to a root, and we won't apply Newton's method any more.

However, we also know Newton's method can totally fail to converge! To prevent this from causing an issue, `newt` also accepts a variable `max_iterations`. Your `newt` function should apply `T_f` at most `max_iterations` number of times. If it hasn't found a root by then, return `np.nan`[1].

(b) `v_newt` is a *vectorized* version of `newt`, which means that when given an array as input, it applies `newt` to each element of the array. Execute the next cell to plot the result of Newton's method and f.

(c) Plot a version of the above graph that is "zoomed-in" to the boundary between two basins of attraction.

(d) Make a plot showing just the basin of attraction of 3 and the graph of f.

2. Making Newton's method complex! Complex numbers can be thought of as two-dimensional analogs of real numbers—they can be added, multiplied, and divided. But, a complex number has two "coordinates" called the *real part* and the *imaginary part*. Oftentimes we visualize complex numbers by plotting them in $\mathbb{R}^2$ with their real part along the x-axis and imaginary part along the y-axis.

   Since Newton's method involves only basic operations that also work on complex numbers, we can apply it to complex values and see what various basins of attraction look like as subsets of the complex plane.

   (a) Since we are now dealing with 2d arrays which have *way* more points in them, it will be convenient to make a fast-and-cheap version of Newton's method. Numpy has many pre-vectorized built-in functions that work very fast. Among these ⟨*array*⟩`*`⟨*array*⟩, ⟨*array*⟩`/`⟨*array*⟩, ⟨*number*⟩$\pm$⟨*array*⟩ all perform very fast element-wise operations. This means the `T_f` you created before is already vectorized!

   Create a `newt2` function which inputs an array and a number of iterations and applies `T_f` to the input array the specified number of times[2].

   (b) The `newt2` function indiscriminately applies `T_f`, and after it executes, some points may have converged and some may not have. However, since we know the roots of f, we can nudge our output to ensure it's correct.

   Create a function `clamped_newt` which applies `newt2` to its input (with the specified number of iterations) and then returns values that have been "clamped" to 0, 2, or 3. That is, every output value in the resulting array should be 0, 2, or 3, depending on which number it is closest to.

   (c) Plot the basins of attractions of f in the complex plane in the rectangle with lower-left corner $-1/2 - i$ and upper-right corner $7/2 + i$.

   Notes:

   - Python uses `j` for imaginary numbers instead of i, but you can't use '`j`' alone. For example, to get the number i, you must write `1j` in Python[3].
   - If you get a `TypeError: Image data cannot be converted to float` error, you're probably asking `imshow` to graph a complex image. `imshow` only knows how to graph real images; you might try calling `np.real(...)` on your data first.

   (d) Make a new plot that is zoomed into an interesting region.

[1] `np.nan` stands for *Not a Number* and is a way to indicate there is no numeric solution in Python.

[2] Your original function would stop iterating when you were sufficiently close to the root. This one will not, but because of how fast Numpy functions are compared to Python functions, it will actually be faster.

[3] Since `i` is so commonly used as a loop variable in programming, most programming languages use `j` for complex numbers.

3. Investigate another Newton fractal of your choosing. Some interesting functions you might try: $f(x) = sin(x^2) - 1$, $f(x) = x^3 - 1$, $f(x) = (\frac{2}{1+x^2} - \frac{5}{4})(1 + x^{1.5})$. Produce two plots of your Newton fractal, one zoomed in and one zoomed out.

   Some things to keep in mind:

   - You can redefine `f` and `f_prime` and `newt2` will just work. However `clamped_newt` will no longer give you the information you want, since the roots have changed.
   - `imshow` can only plot reals, so use `np.real` and `np.imag` appropriately.
   - Wild functions tend to have wild values. If you see a solid color, there may still be a fractal buried in there, but it is being washed out by very large numbers in your array. Try eliminating large (and large negative) numbers.
   - If you want to use non-rational functions, use their Numpy versions. E.g., `np.sin` and `np.exp`. That way they will be automatically vectorized.
   - (Optional) If you want to learn another package, `sympy` can symbolically represent math in Python (and compute derivatives for you). `sympy.utilities.lambdify` can be used to convert a Sympy function into a Numpy function. Google for documentation to see some examples.

4. **The Cantor set**

   (a) The Jupyter notebook includes a `cantorize` function which takes in a list of line segments and removes the middle third from each. Review and execute this function and its associated cells (which will produce a picture of the Cantor set).

   (b) We can use Numpy to estimate the number of "boxes" a sequence of line segments intersects. The `render_segments_to_array` function inputs a list of line segments, an array, and an `extent`, and "draws" the line segments to the array by filling in 1s in every coordinate of the array the line segments touch.

   Using this function (and other tools at your disposal), numerically estimate the box-counting dimension of the Cantor set.

5. **The Koch Snowflake**

   (a) Implement a `kochize` function that inputs a list of segments and applies the Koch snowflake substitution to each of them. Graph one side of K_4 (the Koch snowflake after 4 substitutions).

   For reference, `kochize` applied to `[((0, 0), (1, 0))]` should produce `[((0, 0), (0.3333333333333333, 0.0)), ((0.3333333333333333, 0.0), (0.5, 0.28867513459481287)), ((0.5, 0.28867513459481287), (0.6666666666666666, 0.0)), ((0.6666666666666666, 0.0), (1, 0))]`.

   (b) Make a plot of the full Koch snowflake (all sides of it, not just the top side).

   (c) Estimate the box-counting dimension of the Koch snowflake. How does this compare with its similarity dimension?

6. **The Strange Koch Snowflake**

   (a) Implement a `strange_kochize` function that inputs a list of segments and applies the Koch snowflake substitution to each of them except that it randomly chooses (with 50/50 probability) to point the "spike" up or down along each segment. Graph the 6^{th} iteration of the strange Koch snowflake.

   Notes:

   - In Numpy, you can use `np.random.uniform()` to generate a uniform random number in the interval $[0, 1]$. So, to execute a statement 50% of the time, you could say, `if np.random.uniform() < .5: ....`

- Every time you re-run your Python code, new random numbers will be generated (and your picture will change). To "fix" the random numbers, run `np.random.seed(10)` before you run your code (10 can be replaced with any number of your choosing).

(b) Estimate the box-counting dimension of the strange Koch snowflake. If you changed the probability of a spike pointing "up" to 75/25, would this change the dimension of the resulting fractal? Support your claim with evidence.

7. (Optional) **Sierpinski's Triangle**. View the `matplotlib.collections` documentation and figure out how to draw polygons. Then make a `sierpinskinate` function which applies the Sierpinski triangle substitution. Estimate the box-counting dimension of the Sierpinski triangle.

8. **Our favorite f**. Recall $f(x) = x(x-2)(x-3)$.

   (a) The function `find_edges` will input a 2d array and output an array of 0s and 1s depending on whether the input array differs when shifting one index to the right, down, or diagonal. Use this function to graph the boundary of the Newton fractal given by $f(x) = x(x-2)(x-3)$.
   Notes:

   - `find_edges` returns 1 if there is not literal equality of values. For example, if one pixel of your image has a value of 3.00001 and the next has a value of 3.00002, `find_edges` will assume there is a boundary there. So: either modify `find_edges` to not be so strict, or make sure you have clamped the values in the array that stores your Newton fractal.

   (b) Estimate the fractal dimension of the above Newton fractal's boundary.

   (c) Graph a Newton fractal whose boundary has higher dimension than the one generated by $f(x) = x(x-2)(x-3)$. Back up your claim with evidence.