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.)

a) Suppose $x \in X$ is a fixed point of T, i.e, $T(x) = x$
WTS $A_x = \{y \in X : \lim_{n\to\infty} T^n y = x\} \neq \phi$

Claim: $x \in A_x$
Since $T(x) = x$, then $\lim_{n\to\infty} T^n x = \lim_{n\to\infty} T^{n-1} T(x) = \lim_{n\to\infty} T^{n-1} x = x$.

Thus, $x \in A_x$ so $A_x \neq \phi$.

b) Assume $R \subseteq A_y$, WTS $T^{-1}(R) \subseteq A_y$, i.e. if $x \in T^{-1}(R)$, then $x \in A_y$
Fix $x \in T^{-1}(R)$, know $T(x) \in R$
since $R \subseteq A_y$, then $x \in A_y$, and x is arbitrary
thus, $\forall x \in T^{-1}(R)$, $x \in A_y$, as needed.

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

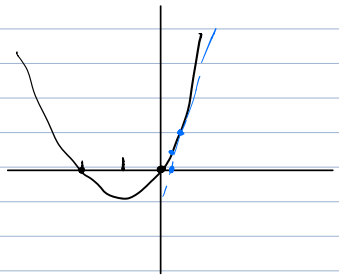
Let $f : \mathbb{R} \to \mathbb{R}$ be a twice differentiable function and suppose that $f(0) = 0$ and that f' and f'' are positive on the interval $(0, a]$. Let T_f be the function that applies one iteration of Newton's method to its input.

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

Pick $f(x) = (x+2)x = x^2 + 2x$

$f'(x) = 2x + 2$

$f''(x) = 2$



Yes, it converges to 0.

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

know $T_f(x) = \dfrac{-f(x)}{f'(x)} + x$ for all $x \in \mathbb{R}$. (from class)

Suppose $x \in (0, a)$. WTS $0 \leq \dfrac{-f(x)}{f'(x)} + x < a$ ↓ by f' is positive, i.e. $f'(x) > 0$

$\Leftrightarrow \quad 0 \leq -f(x) + x f'(x) < a f'(x)$

$\Leftrightarrow \quad f(x) \underset{①}{\leq} x f'(x) \underset{②}{<} a f'(x) + f(x)$

know $0 \leq x f'(x) < a f'(x)$ since $x \in (0, a]$ and $f'(x)$ is positive

Since f' is positive on $(0, a]$, then f is increasing on $(0, a]$ (1)

and f'' is positive on $(0, a]$, then f is concave up on $(0, a]$ (2)

Thus, $f(x) \geq 0$ since $f(0) = 0$ and $x \in (0, a]$, by (1) & (2) → ↑.

Then, know $x f'(x) < a f'(x) + f(x)$, ② is True

left to show ① $\Leftrightarrow x f'(x) - f(x) \geq 0$, compute $g'(x)$ where $g(x) = x f'(x) - f(x)$

$g'(x) = x f''(x) + f'(x) - f'(x) = x f''(x) \geq 0$ since $f''(x) > 0$, $x \in (0, a]$.

So, g is an increasing fn where $g(0) = 0$, so $g(x) \geq 0$, as needed.

(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 $x \in (0, a]$.

0 is a fixed point of f

Edge case a

Fix $x \in (0, a]$. WTS $\lim_{n\to\infty} T_f^n(x)$ converges.

from b), we know $0 \leq T_f(x_0) < a$ if $x_0 \in (0, a)$, so $(T_f^n(x))_{n\in\mathbb{N}}$ is bounded below, by 0.

WTS $T_f^{n+1}(x) \leq T_f^n(x)$ $\forall n \in \mathbb{N}$. Fix $n \in \mathbb{N}$

$\Leftrightarrow T_f(T_f^n(x)) \overset{?}{\leq} T_f^n(x)$

$\Leftrightarrow \dfrac{-f(T_f^n(x))}{f'(T_f^n(x))} + T_f^n(x) \overset{?}{\leq} T_f^n(x)$ $\Leftrightarrow -f(T_f^n(x)) \overset{✓}{\leq} 0$ since f is positive on $(0, a]$. (f' is positive)

Then, by Monotone Convergence Theorem, $\lim_{n\to\infty} T_f^n(x)$ converges $\forall x \in (0, a]$

(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).

WTS x is a fixed point of T_f, i.e., $T_f(x) = x$.
$T_f(x) = T_f(\lim_{n\to\infty} T_f^n(x_0)) = \lim_{n\to\infty} T_f(T_f^n(x_0)) = \lim_{n\to\infty} T_f^{n+1}(x_0) \overset{m=n+1}{=} \lim_{m\to\infty} T_f^m(x_0) = x$.
Since T_f is a cts fn on $(0, a]$, as T_f is a composition of cts fn.
(f, f' are cts fn as their derivatives exist on $(0, a]$)

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

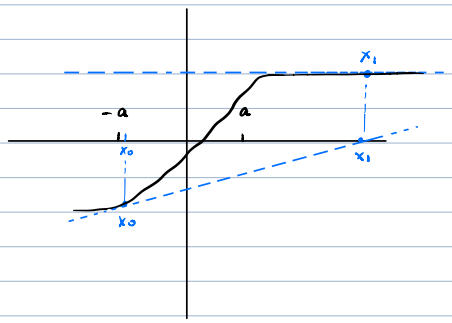
Fix $x \in (0, a]$, from c) we know $\lim_{n\to\infty} T_f^n(x)$ converges
Suppose $x' = \lim_{n\to\infty} T_f^n(x)$, WTS $x' = 0$, or equivalently, $x \in A_0$, basin of attraction of 0.
from d), we know x' is a fixed point of T_f. i.e., $T_f(x') = x'$.
Use def of limit, $\exists N \in \mathbb{N}$ s.t. $\forall n \geq N$, $T_f^n(x) = x'$
Then, $x \in T_f^{-N}(x')$ by def of pre-image.
By in class practice and the conditions of f ($f(0) = 0$, $f', f'' > 0$), we know $x \in A_{x'}$.
We also know $x' = 0$ since by b), $0 \leq T_f(x) < a$ as f is increasing, concave up on $(0, a]$.
Thus, $x \in A_0$.

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

from e), we know $\forall x \in (0, a]$, $\lim_{n\to\infty} T_f^n(x) = 0$, i.e. converges to 0.

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 $x \in [-a, a]$.



undefined slope, does not converge.
Intial guess x_0
$Tf(x_0) = x_1 \notin [-a, a]$

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

WTS $\exists b > 0$ s.t. $\forall x \in [-b, a]$, $\lim_{n\to\infty} Tf^n(x)$ exists.
Inspirated by Q2, we just need to show that $\forall x \in [-b, a]$, $0 \le Tf(x) \le a$, and with similiar proofs of 2.c), d), we can conclude limit exists

Since f is diff and cts on $[-a, 0] \subseteq [-a, a]$, by MVT, $\exists -b \in (-a, 0)$ s.t. $(b > 0)$

$$f'(-b) = \frac{f(0) - f(-a)}{0 - (-a)} = \frac{-f(-a)}{a}$$

WTS $\forall x \in [-b, a]$, $Tf(x) \in [0, a]$

First show $0 \le Tf(-b) \le a$, since $-b \in [-a, a]$, $f'(-b) > 0$, then

$$Tf(-b) = \frac{-f(-b)}{f'(-b)} + -b = \underset{\uparrow \text{by MVT}}{-f(-b) \cdot \frac{a}{-f(-a)}} + -b = \frac{a f(-b)}{f(-a)} - b$$

Since f is increasing on $[-a, a]$ $(f' > 0)$, know $f(-a) < f(-b) < \underbrace{f(0)}_{=0}$ ↙ $f(-a) < 0$, flip sign (since $f(0) = 0$ and f is increasing on $[-a, a]$ and concave up)

$$\Leftrightarrow \quad 1 > \frac{f(-b)}{f(-a)} > 0$$

So $Tf(-b) = \frac{a f(-b)}{f(-a)} - b < \underset{\uparrow -b < 0}{\frac{f(-b)}{f(-a)}} a < \underset{\uparrow \text{since } \frac{f(-b)}{f(-a)} \in (0,1)}{a}$

Thus, $Tf(-b) \le a$

Next, $Tf(-b) \overset{?}{\ge} 0 \quad \Leftrightarrow \quad \frac{-f(-b)}{f'(-b)} - b \overset{?}{\ge} 0 \quad \Leftrightarrow \quad -b f'(-b) - f(-b) \overset{?}{\ge} 0$

Let $g(y) = y f'(y) - f(y)$, $g'(y) = f'(y) - f'(y) + y f''(y)$
$= y f''(y) < 0$ on $y \in [-b, 0)$
$\Rightarrow g(y)$ is decreasing on $[-b, 0]$, so $g(-b) > g(0) = 0 \cdot f'(0) - f(0) = 0$
Then $g(-b) = -b f'(-b) - f(-b) > 0 \Rightarrow -b f'(-b) - f(-b) \ge 0$, as needed.
so $Tf(-b) \ge 0$.

Next, WTS $Tf(x) \in [0, a] \quad \forall x \in [-b, a]$
if $x \in [0, a]$, like Q2, done.
if $x \in [-b, 0)$, know $g(x) \le g(-b)$ since g is decreasing on $[-b, 0)$ proven previously.

$$\Leftrightarrow x f'(x) - f(x) \le b f'(b) - f(b)$$
$$\Leftrightarrow \underset{f'(b)}{x \frac{f'(x)}{f'(b)}} - \underset{f'(b)}{\frac{f(x)}{f'(b)}} \le b - \frac{f(b)}{f'(b)} \qquad (f' > 0.)$$

also know $f'(x) \geq f'(b) > 0$ since $f'' > 0$ on $[-a, a]$, concave up

$$T_f(x) = \frac{-f(x)}{f'(x)} + x = \frac{-f(x) + x f'(x)}{f'(x)} \leq \frac{-f(x) + x f'(x)}{f'(b)} \quad \text{by } f'(x) \geq f'(b)$$

$$= x\frac{f'(x)}{f'(b)} - \frac{f(x)}{f'(b)} \leq b - \frac{f(b)}{f'(b)} = T_f(b)$$

so $T_f(x) \leq T_f(b) \leq a$

Then, left to show $T_f(x) \geq 0 \Leftrightarrow g(x) > 0 = g(0)$

with similar argument to prove $T_f(b) \geq 0$, $x f''(x) = g'(x) < 0$ for $x \in [-b, 0)$,

so g is decreasing on $[b, 0]$, so $g(x) > g(0) = 0 \Rightarrow g(x) \geq 0$

Thus, $T_f(x) \geq 0$.

Then, we show $\forall x \in [-b, a]$, $T_f(x) \in [0, a]$, as needed.

(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 $-x_0$.

$h(-t) = g(t)$

Assume $\lim_{n\to\infty} T_g^n(x_0)$ exists and $= L_g \in \mathbb{R}$, WTS $\lim_{n\to\infty} T_h^n(-x_0)$ exists.

know $h'(t) = -g'(-t)$ $\forall t \in \mathbb{R}$

$$T_h(-x_0) = \frac{-h(-x_0)}{h'(-x_0)} - x_0 \overset{\uparrow}{=} \frac{-g(x_0)}{-g'(x_0)} - x_0 = \frac{g(x_0)}{g'(x_0)} - x_0 = -T_g(x_0)$$

Then, $\lim_{n\to\infty} T_h^n(-x_0) = \lim_{n\to\infty} T_h^{n-1}(T_h(-x_0)) = \lim_{n\to\infty} T_h^{n-1}(-T_g(x_0)) \overset{\text{by } T_h(-t) = -T_g(t)}{=} \lim_{n\to\infty} T_h^{n-2}(-T_g(T_g(x_0)))$

$$= \lim_{n\to\infty} -T_g^n(x_0) = -\lim_{n\to\infty} T_g^n(x_0) = -L_g$$

$\Rightarrow \lim_{n\to\infty} T_h^n(-x_0) = -L_g \in \mathbb{R}$

Thus, Newton's method converges for h w/ a starting point $-x_0$.

(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 $[-a, b]$.

Define $g : \mathbb{R} \to \mathbb{R}$ s.t. $g(t) = f(-t)$, then $g(0) = 0$, $g'(t) = -f'(t) \Rightarrow g' > 0$ and $g''(t) = --f''(t) = f''(t) > 0$.

from b), we know $\exists\, c > 0$ s.t. Newton's method converges for g on $[-c, a]$

from c), we know Newton's method converges for f on $[-a, c]$.

Pick $b = c > 0$. then $\exists\, b > 0$ s.t. Newton's method converges for f on $[-a, b]$, as needed.

(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.

we know that p is cts, and p' is cts by p is double differentiable. (1)

We also know from piazza that p'' is cts. (2)

First define $g(x) = p(x + x_0)$, change of variable does not affect (1) & (2), so g has same properties as p, with assumptions

Case 1, $g' > 0$, $g'' > 0$

by g' being cts, know $\exists\, \delta_1 > 0$ s.t. $g'(x - \delta_1, x + \delta_1) > 0$

" g'' " " $\exists\, \delta_2 > 0$ s.t. $g''(x - \delta_2, x + \delta_2) > 0$

Let $\delta = \min\{\delta_1, \delta_2\} > 0$

Pick interval of convergence to be $(-\delta, \delta)$, $g(x)$ converges on it under Newton's method by (3 b)

Then, p converges on $(x_0 - \delta, x_0 + \delta)$ by change of variable

Case 2 $g' < 0$, $g'' > 0$ (newton's method)

with similiar argument as case 1, and Q3 d), we know p converges on $(x_0 - \delta, x_0 + \delta)$ where δ is the same as previous question, but $\delta_1 > 0$ is for $g'(x - \delta_1, x + \delta_1) < 0$.

Case 3 $g' < 0$, $g'' < 0$

Consider $h(x) = -g(x) \Rightarrow h'(x) = -g'(x) > 0$, $h''(x) = -g''(x) > 0$

then, h is like case 1, with $\delta' = \min\{\delta_1', \delta_2'\}$ ($h'(-\delta_1', \delta_1') > 0$

and $(-\delta', \delta')$ is the convergence interval for h $h''(-\delta_2', \delta_2') > 0$)

Then, $(-\delta', \delta)$ is the convergence interval for g by Q3 c) and d) $\Rightarrow (x_0 - \delta', x_0 + \delta')$ for p

Case 4 $g' > 0$, $g'' < 0$

Consider $k(x) = -g(x) \Rightarrow k'(x) = -g'(x) < 0$, $k''(x) = -g''(x) > 0$

Then k is case 2, has an convergence interval $\Rightarrow$ g also has an convergence interval

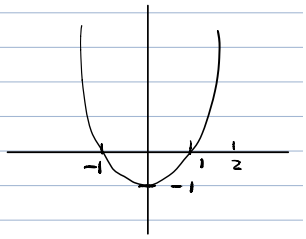
$\Rightarrow$ translation to get interval for p

Thus, $\exists$ an open interval about x_0 on which Newton's method will converge to x_0 for p.

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. $f'(x) = 2x$ $\quad f(x) = (x-1)(x+1)$

(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).

$T_f^{-1}(\{1,2,3\}) = \{1,\ 2\pm\sqrt{3},\ 3\pm\sqrt{10}\}$



$-\frac{f(x)}{f'(x)} + x = 1 \iff \frac{-x^2+1}{2x} + x = 1 \iff -x^2+1 = (1-x)\,2x$

$\iff x^2 - 2x + 1 = 0$

$\iff (x-1)(x-1) = 0$

$\Rightarrow x = 1$

$-\frac{f(x)}{f'(x)} + x = 2 \iff -x^2+1 = (2-x)\,2x \iff x^2 - 4x + 1 = 0$

$x = \frac{4 \pm \sqrt{16-4}}{2} = \frac{4 \pm 2\sqrt{3}}{2} = 2 \pm \sqrt{3}$

$-\frac{f(x)}{f'(x)} + x = 3 \iff -x^2+1 = (3-x)\,2x \iff x^2 - 6x + 1 = 0$

$x = \frac{6 \pm \sqrt{36+4}}{2} = 3 \pm \frac{2\sqrt{10}}{2} = 3 \pm \sqrt{10}$

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

no, when $x = 0 \in \mathbb{R}$, $T_f(x)$ is undefined since $f'(0) = 0$.

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

$X = \mathbb{R} \setminus \{0\}$ based on b).

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

from 3e), know $\exists\, \delta > 0$ s.t. $\lim_{n\to\infty} T_f^n(x) = 1$ for all $x \in (1-\delta, 1+\delta)$ since $f'(1), f''(1) \neq 0$ and $f(1) = 0$

Also, $\exists N$ s.t. $\forall n \geq N$, $T_f^n(4) \in (1-\delta, 1+\delta)$ by the graph

($T_f(4) = 2.125$, $T_f^2(4) \approx 1.298$, $T_f^3(4) \approx 1.0341\ldots$, $T_f^5(4) \approx 1.0006\ldots$)

Thus, $\lim_{n\to\infty} T_f^n(4) = 1$

(e) Find all fixed points of T_f.

roots of f are fixed point : $1, -1$

since $T_f(1) = -\left(\frac{1^2-1}{2\cdot 1}\right) + 1 = 1$ and, $T_f(-1) = -\left(\frac{(-1)^2-1}{2\cdot -1}\right) - 1 = -1$

Check if there are any other fixed pts x s.t. $T_f(x) = x = \frac{-f(x)}{f'(x)} + x = \frac{-(x^2-1)}{2x} + x$

$\iff 2x^2 = -x^2 + 1 + 2x^2$

$\iff x^2 - 1 = 0 = (x-1)(x+1)$

$\Rightarrow x = \pm 1$, the roots only.

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

fixed points by e) $\begin{cases} x_1 = 1 & A_1 = (0, \infty) \\ x_2 = -1 & A_{-1} = (-\infty, 0) \end{cases}$ based on the graph "

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?

$f'(x) = \frac{d}{dx}\left(2(1+x^2)^{-1} - \frac{5}{4}\right) = (-2(1-x^2)^{-2}) \cdot (-2x) = \frac{4x}{(1-x^2)^2}$

Then $T_f(0) = \frac{-f(0)}{f'(0)} + 0 = \frac{-(2 - \frac{5}{4})}{0} + 0 =$ undefined, as $f'(0) = 0$.

Thus $0 \notin$ domain(T_f).

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

based on the computer algebra system, $T_f^{-1}(\{0\}) = \{-\sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}\}$

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

gives complex #, excludes

based on the computer algebra system, $T_f^{-2}(\{0\}) = T_f^{-1}(\{-\sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}\}) \approx \{1.790797, -1.790797\}$

(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!)

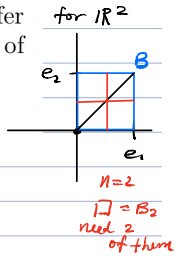
For each iteration $i \in \mathbb{N}$ of $T_f^{-i}(\{0\})$, domain gets two new disconnecting points p_{i1}, p_{i2} s.t. $T_f^{i+1}(T_f^{-i}(\{p_{i1}, p_{i2}\})) = T_f(\{0\}) = DNE$

This tells us there are infinite disconnecting points which newton's method for f does not converge.

Thus, $X = \Phi$.

Computer program (Python) also supports this as no matter how small the domain that I input is, it still gives me error after some iterations

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.

by def, B is minimal-dimensional, minimally sized superset of X with side length d, and n of Bn will intersect X

dimension $\longrightarrow \lim_{n\to\infty} \frac{\log n}{\log n} = 1$

(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.*)

B is minimized.

Suppose X's box-counting dimension of X is $\lim_{n\to\infty} \frac{\log C^n}{\log n}$, C^n = # of subboxes B_n intersecting X
let B′ be the minimal-dimensional superset of X
It's clear to see that C^n of B_n' intersects X since for $y \in B' \setminus B$, $y \notin X$ since $B \supseteq X$.
Thus, dimension of X with B′ as the box is still $\lim_{n\to\infty} \frac{\log C^n}{\log n}$

Similiarly, suppose B″ is the minimal-sized superset of X, and $B'' \subseteq \mathbb{R}^{m+k}$, $k \in \mathbb{N}$
It's also clear to see that C^n of B_n'' intersects X since dimension of B does not affect the number of subboxes intersecting X.
Thus, dimension of X with B″ as the box is the same as with B.

(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.

Assume Y is a translation of $X \subseteq \mathbb{R}^m$, where X is translated by vector $\vec{v} \in \mathbb{R}^m$ (i.e. $X + \{\vec{v}\} = Y$)
Suppose d = box-counting dimension of X, WTS dimension of Y is also d
Let B be the box for X, $B \subseteq \mathbb{R}^m$, then $B + \{\vec{v}\}$ is a box for Y, translated by $\vec{v}$.
Clearly translation does not affect the number of subboxes B_n intersecting X
Thus, $B + \{\vec{v}\}$ also has the same number of subboxes intersecting Y.
Take limit goes infinity, Y's dimension (box-counting) is also d.

(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.

Assume $X, Y \subseteq \mathbb{R}^m$ are box-disjoint sets, i.e. $\exists A, B$ disjoint boxes s.t. $X \subseteq A$, $Y \subseteq B$

Let $A = \prod_{i=1}^{m} [a_i, a_i']$ $\quad B = \prod_{i=1}^{m} [b_i, b_i']$

from b) we know that the box for $X \cup Y$ does not need to be minimized (in dimension & size)

so Consider $C = \prod_{i=1}^{m} [\min\{a_i, b_i\}, \max\{a_i', b_i'\}]$, and clearly $C \supseteq X \cup Y$

Suppose q^n of A_n subboxes intersect X, p^n of B_n' subboxes intersect Y, then

$X \to d_A = \lim_{n\to\infty} \frac{\log q^n}{\log n}$ as dim., $\quad Y \to d_B = \lim_{n\to\infty} \frac{\log p^n}{\log n}$ as dimension

Then C cut along axis with n equally-slices and has q^n subboxes of C_n intersect X, with p^n subboxes of C_n intersect Y.

WLOG, suppose $p > q$, $p, q > 0$, so $\max\{d_A, d_B\} = d_A$

thus, $\lim_{n\to\infty} \frac{\log(p^n + q^n)}{\log n} = \lim_{n\to\infty} \frac{\log(p^n(1 + (\frac{q}{p})^n))}{\log n} = \lim_{n\to\infty} \left(\frac{\log p^n}{\log n} + \frac{\log(1 + (\frac{q}{p})^n)}{\log n} \right)$

$\uparrow$ $\dim(X \cup Y)$ with C

$= \lim_{n\to\infty} \frac{\log p^n}{\log n}$ $\quad$ $\underbrace{\phantom{\frac{\log(1 + (\frac{q}{p})^n)}{\log n}}}_{= 0 \text{ as } n\to\infty \text{ since } \frac{q}{p} < 1}$

$= d_A = \max\{\dim(X), \dim(Y)\}$, as needed.

(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 $T(X)$.

Suppose B is the box of X s.t. $X \subseteq B$, with p of B_n intersect X

$T(X)$ will need kB as the box of $T(X)$ s.t. $kB \supseteq T(X)$

B's side length increases by k

Then, kp of B_n intersects $T(X)$

Thus, $d_{T(X)} = \lim_{n\to\infty} \frac{\log kp}{\log n} \overset{\text{log laws}}{=} \lim_{n\to\infty} \frac{\log k + \log p}{\log n} = \lim_{n\to\infty} \left(\frac{\log k}{\log n} + \frac{\log p}{\log n} \right)$ ($\frac{\log k}{\log n} \to 0$)

$= \lim_{n\to\infty} \frac{\log p}{\log n} = d_X$ by def.

So box-counting dimension of $T(X)$ = box-counting dimension of 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~~ below by the box-counting dimension of $T(X)$.

Let A be the matrix representation of T.

know $\exists E_1, \ldots, E_k, E_{k+1}, \ldots, E_{k+n}$ elementary matrices s.t. $A = E_1 E_2 \cdots E_k R E_{k+1} \cdots E_{k+n}$ (R: projection matrix)

we know elementary matrices geometrically either scale, shear, or change orientation. (shear: row additions; change orientation: switch rows)

- scaling, from e), has no affects to the box-counting dimension.
- flip orientation also has no affects, since the original box can be fliped to contain X and maintain same box-counting dimension (# of subboxes is still the same)
- Shear matrices have determinant 1, which implies it does not change the area of X, so maintains same box-counting dimension

Then, R as a projection might reduce the dimension of X

Thus, $\dim(T(X)) = \dim(A(X)) \leq \dim(X)$.

(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.

Let T be an invertible transformation. Know T^{-1} exists and is a linear transformation.
By d), know $\dim(X) \geq \dim(T(X))$
and, $\dim(T(X)) \geq \dim(T^{-1}(T(X))) = \dim(X)$ since T^{-1} is also a LT.
and $T(X) \subseteq \mathbb{R}^m$

Thus, $\dim(X) = \dim(T(X))$, as needed.
↳ box-counting dimension

(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.

I promise that I will try this after I complete the essay... :(

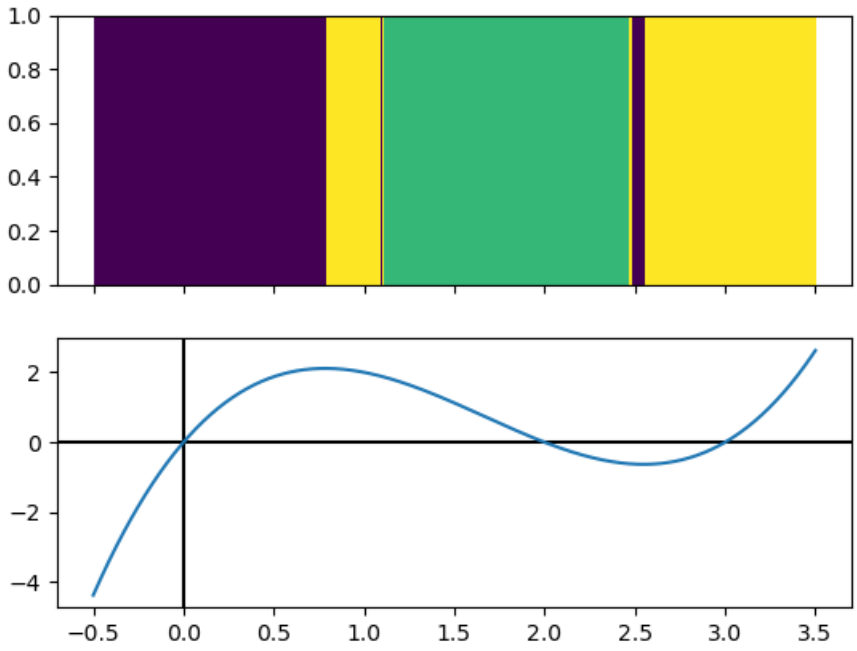
Homework 1

Do the programing part of Homework 1 in this notebook. Predefined are function *stubs*. That is, the name of the function and a basic body is predefined. You need to modify the code to fulfil the requirements of the homework.

In [91]:
```
# import numpy and matplotlib
import numpy as np
import matplotlib.pylab as plt
# We give the matplotlib instruction twice, because firefox sometimes gets upset if we don't.
# note these `%`-commands are not actually Python commands. They are Jupyter-notebook-specific
 commands.
%matplotlib notebook
%matplotlib notebook
```

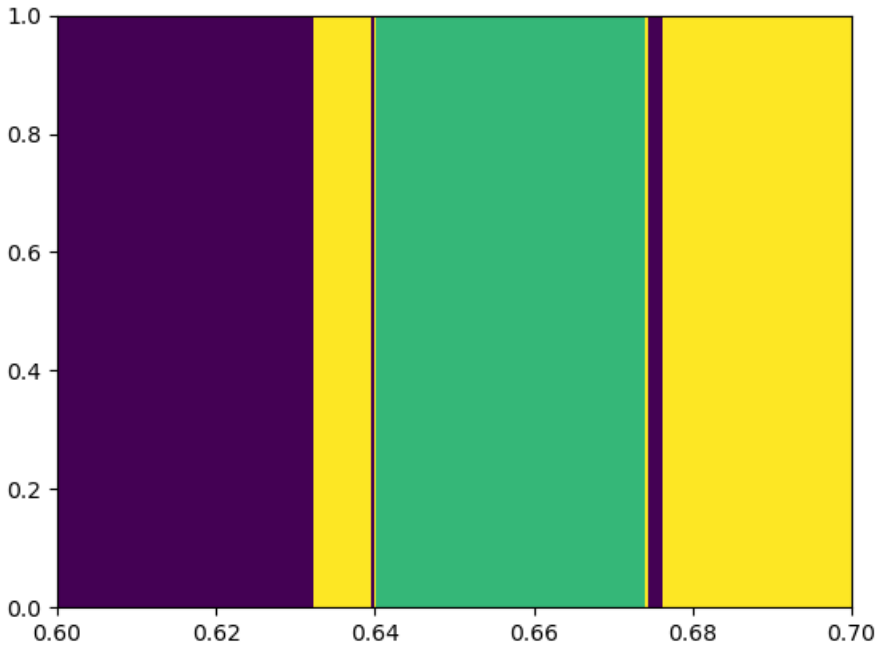
In [92]:
```
def f(x):
    return x*(x-2)*(x-3)

def f_prime(x):
    return 3*x*x-10*x+6

def T_f(guess):
    # import $T_f$ function, as its formualt is $T_f(x) = -f(x)/f'(x) + x$
    guess = -f(guess)/f_prime(guess) + guess
    return guess

def newt(guess, max_iterates = 20, tolerance=0.0001):
    i = 1
    while (not (abs(f(guess)) <= tolerance)) and (i != max_iterates):
        guess = T_f(guess)
        i += 1

    if  (abs(f(guess)) <= tolerance):
        return guess
    else:
        return np.nan

v_newt = np.vectorize(newt)
```

In [93]:
```
# N is how many points we will sample
N = 500
xs = np.linspace(-.5, 3.5, N)
ys = v_newt(xs)

# Create a figure with two plots stacked vertically. One will be for
# the basins of attraction and one will be for graphing f.
fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
# The `imshow` command assumes every pixel takes up one unit of space. By
# defining the `extent` we can tell imshow that we want the units to be
# something else. An extent is [x_min, x_max, y_min, y_max]
extent = [xs.min(), xs.max(), 0, 1]
# The `imshow` command expects a 2d array, but `ys` is a 1d array. We can
# make it a 2d array with the command `np.array([ys])`
ax1.imshow(np.array([ys]), extent=extent, aspect="auto")
# Draw some axis lines on the second plot
ax2.axhline(y=0, color='k')
ax2.axvline(x=0, color='k')
# Plot the function
ax2.plot(xs, f(xs))
```



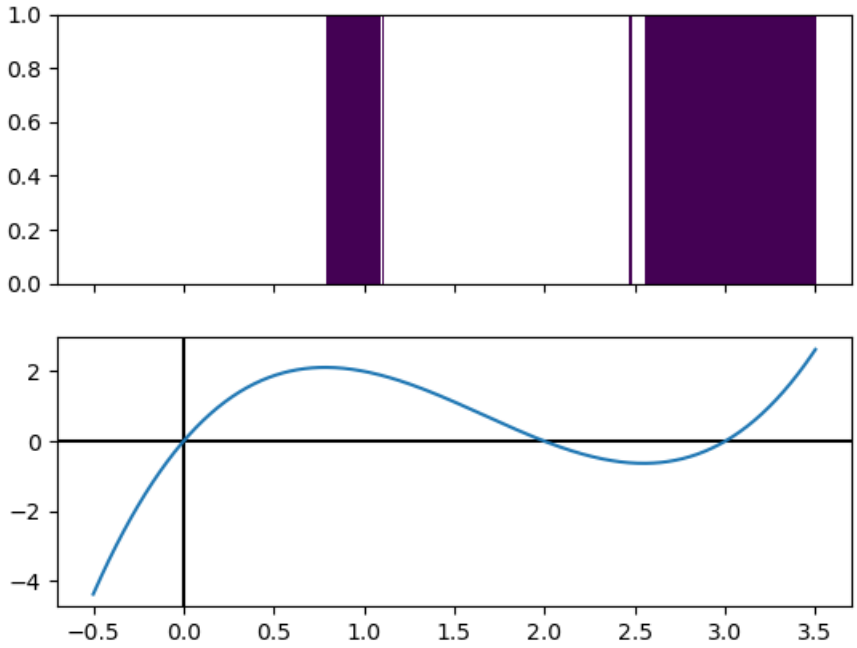
Out[93]: [<matplotlib.lines.Line2D at 0x7f8f45d0d908>]

In [94]:
```
#
# Make a graph that is "zoomed-in" to the boundary between two basins of attraction
#

N = 500
xs = np.linspace(-.5, 3.5, N)
ys = v_newt(xs)

# create a new figure
fig = plt.figure()
# create an "axis" inside the figure
ax = fig.add_subplot(1, 1, 1)

# use codes in the above cell, we can zoom-in by redefining extent to tell
# imshow that we want the units to be something eslse
# one can find the boundary between two basins of attraction of thier interest by looking at th
e graph
# I picked the interval for x: [0.6, 0.7]
new_extent = [0.6, 0.7, 0, 1]
ax.imshow(np.array([ys]), extent=new_extent, aspect="auto")
```



Out[94]: <matplotlib.image.AxesImage at 0x7f8f4253ae48>

In [95]:
```
#
# Graph the basin of attraction of (only) 3
#
N = 500
xs = np.linspace(-.5, 3.5, N)
ys = v_newt(xs)

# set a tolerence of convergence to roots
tolerence = 0.0001
# for those point that do not converge to the range 3 +- tolerence (i.e., not in basin of attra
ction of 3, A_3)
# remove thems
ys[abs(ys) >= 3-tolerence] = 3
ys[abs(ys) < 3-tolerence] = np.nan

# create two figures, one is A_3, the other is the graph of f
fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)

# plot A_3
new_extent = [xs.min(), xs.max(), 0, 1]
ax1.imshow(np.array([ys]), extent=new_extent, aspect="auto")

# Draw some axis lines on the second plot
ax2.axhline(y=0, color='k')
ax2.axvline(x=0, color='k')
# Plot the function
ax2.plot(xs, f(xs))
```



Out[95]: [<matplotlib.lines.Line2D at 0x7f8f421aceb8>]

Complex Newton's Method

In [96]:
```
#
# This function is provided for you to use later
#
def make_complex_grid(z_low, z_high, N=100):
    """Create an N x N  2d array of complex numbers whose lower-left
    corner is give by `z_low` and upper-right corner is given by `z_high`"""
    reals = np.linspace(np.real(z_low), np.real(z_high), N)
    imags = np.linspace(np.imag(z_low), np.imag(z_high), N)
    a, b = np.meshgrid(reals, imags)
    return a + b*1J
```

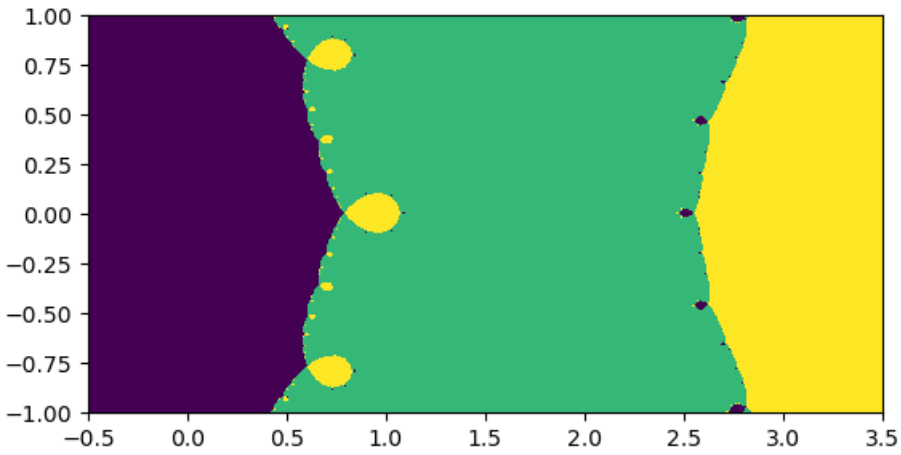
In [97]:
```
# def T_f_complex(guess):
#     # import $T_f_complex$ function, as its formualt is $T_f(x) = -f(x)/f'(x) + x$
#     guess = -f(guess)/f_prime(guess) + guess
#     return guess

def newt2(guess_array, iterations=20):
    for i in range(iterations):
        guess_array = T_f(guess_array)
    return guess_array

xs = np.array([1,2,3,4, 2000])
print("v_newt and newt2 should give similar results. v_newt:\n", v_newt(xs), "\n and newt2:\n",
newt2(xs))
```

```
v_newt and newt2 should give similar results. v_newt:
 [3.         2.         3.         3.00000019        nan]
 and newt2:
 [3.         2.         3.         3.         3.01520126]
```

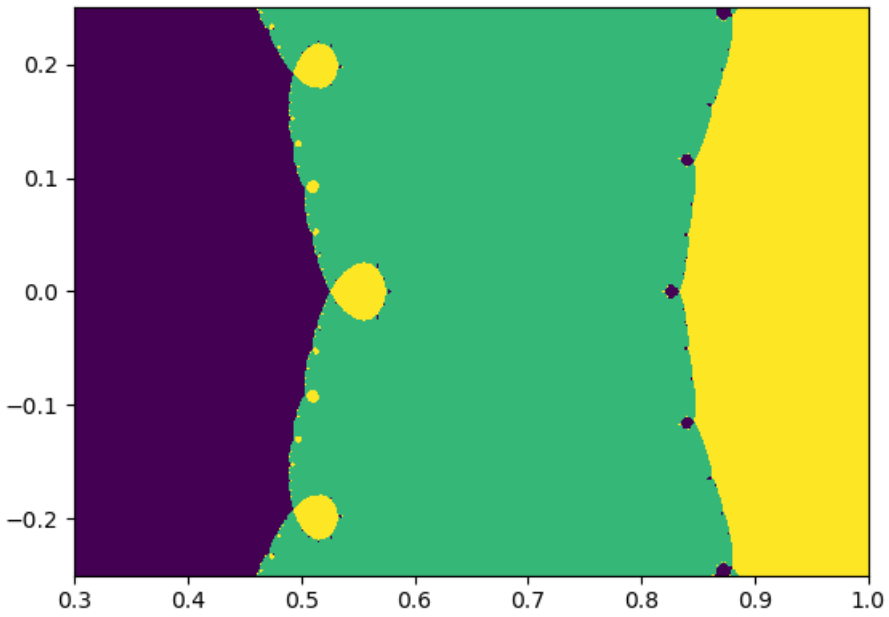
In [98]:
```
#
# There are many numpy functions that will be helpful for implementing `clamped_newt`.
# For example, `np.round`. You can google for these.
#
# Another helpful tip is fancy indexing: If `xs` is an array, to set
# all elements of `xs` which are less than four to zero, you can do
# `xs[ xs < 4 ] = 0`. To set all elements of `xs` that are less than four but greater
# than three to zero, you can do `xs[ (xs < 4) & (xs > 3) ] = 0`. Note the use of parenthesis!
#
def clamped_newt(guess_array, iterations=20):
    for i in range(iterations):
        guess_array = newt2(guess_array)
    np.round_(guess_array)

    return np.real(guess_array)

xs = np.array([1,2,3,4, 2000, -1000, 9, 0, 2.5, 2.1])
print("clamped_newt should the same outputs as newt2, but rounded to the"
      "nearest root. newt2:\n", newt2(xs), "\n and clamped_newt:\n", clamped_newt(xs))
```

```
clamped_newt should the same outputs as newt2, but rounded to thenearest root. newt2:
 [ 3.00000000e+00  2.00000000e+00  3.00000000e+00  3.00000000e+00
  3.01520126e+00 -5.23363535e-11  3.00000000e+00  0.00000000e+00
  0.00000000e+00  2.00000000e+00]
 and clamped_newt:
 [3. 2. 3. 3. 3. 0. 3. 0. 0. 2.]
```

In [99]:
```
N = 500
zs = make_complex_grid(-.5-1j, 3.5+1j, N)

fig, ax = plt.subplots()

extent = [np.real(zs).min(), np.real(zs).max(), np.imag(zs).min(), np.imag(zs).max()]

ax.imshow(clamped_newt(zs), cmap="viridis", extent=extent, origin="lower")

plt.show()
```

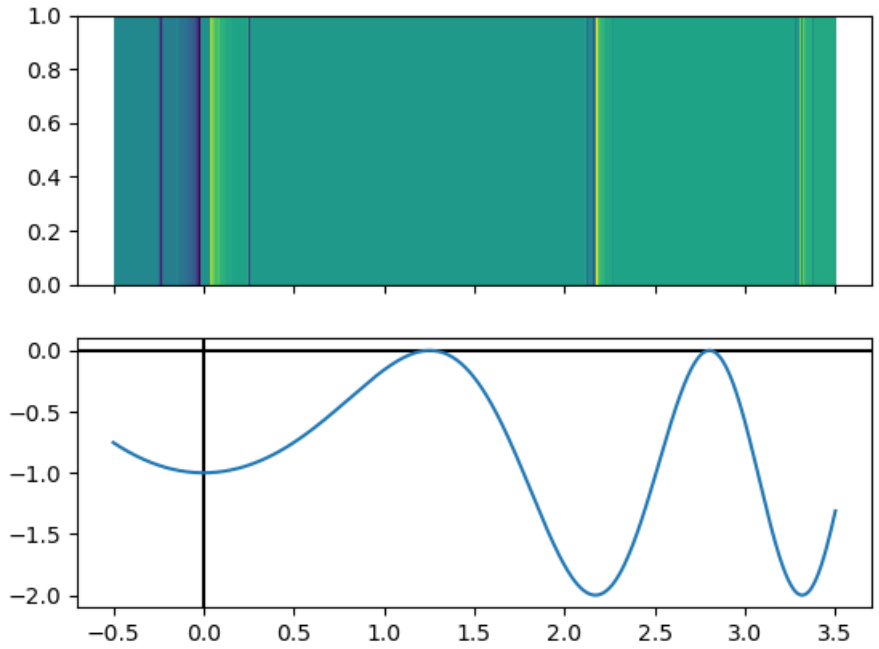


In [100]:
```
#
# Make a graph that is "zoomed-in" to a region that interests you
#
N = 500
zs = make_complex_grid(-.5-1j, 3.5+1j, N)

fig, ax = plt.subplots()

#similar to 1c, use new_extent to zoom in at 0.3 to 1.0, the weird loop circle part
new_extent = [0.3, 1, -0.25, 0.25]

ax.imshow(clamped_newt(zs), cmap="viridis", extent=new_extent, origin="lower")

plt.show()
```



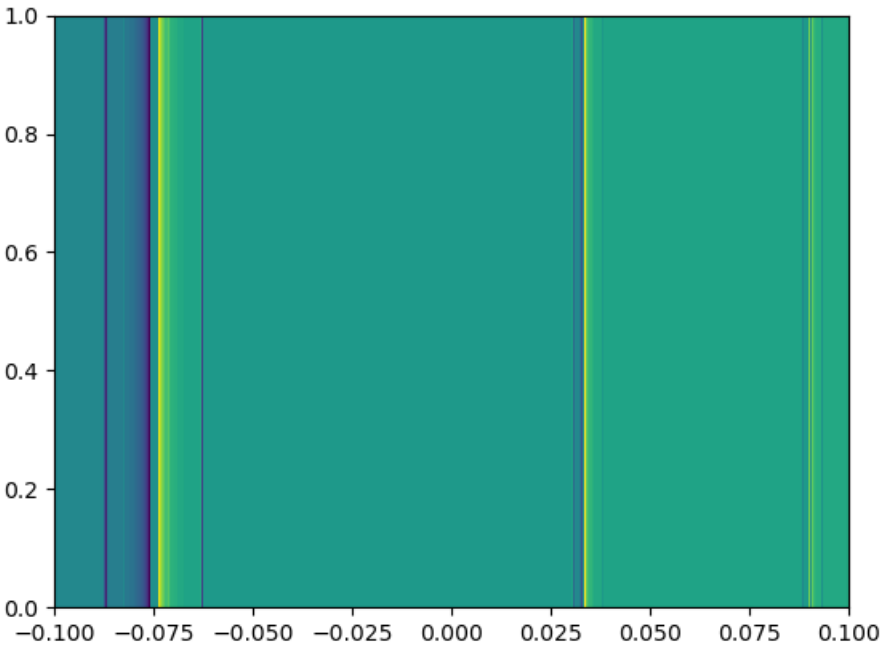
A new function

In [102]:
```
#
# Redefine `f` and `f_prime` here and plot a new Newton fractal!
#
def f(x):
    return np.sin(x**2)-1

def f_prime(x):
    return np.cos(x**2)*2*x

# N is how many points we will sample
N = 500
xs = np.linspace(-.5, 3.5, N)
ys = v_newt(xs)

# elminate wild, large values in array
ys[ys < -20] = 0
ys[ys > 20]=0

# Create a figure with two plots stacked vertically. One will be for
# the basins of attraction and one will be for graphing f.

fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)

extent = [xs.min(), xs.max(), 0, 1]

ax1.imshow(np.array([ys]), extent=extent, aspect="auto")
# Draw some axis lines on the second plot
ax2.axhline(y=0, color='k')
ax2.axvline(x=0, color='k')
# Plot the function
ax2.plot(xs, f(xs))
```



Out[102]: [<matplotlib.lines.Line2D at 0x7f8f41fe0358>]

In [103]:
```
#
# Zoom into an interesting region of your new fractal
#

N = 500
xs = np.linspace(-.5, 3.5, N)
ys = v_newt(xs)

# elminate wild values in array
ys[ys < -20] = 0
ys[ys > 20]=0

# create a new figure
fig = plt.figure()
# create an "axis" inside the figure
ax = fig.add_subplot(1, 1, 1)

# The pretty gradient colour around 0 looks very interesting. Let's zoom in!
new_extent = [-0.1, 0.1, 0, 1]
ax.imshow(np.array([ys]), extent=new_extent, aspect="auto")
```



Out[103]: <matplotlib.image.AxesImage at 0x7f8f41f5c978>

Common Fractals

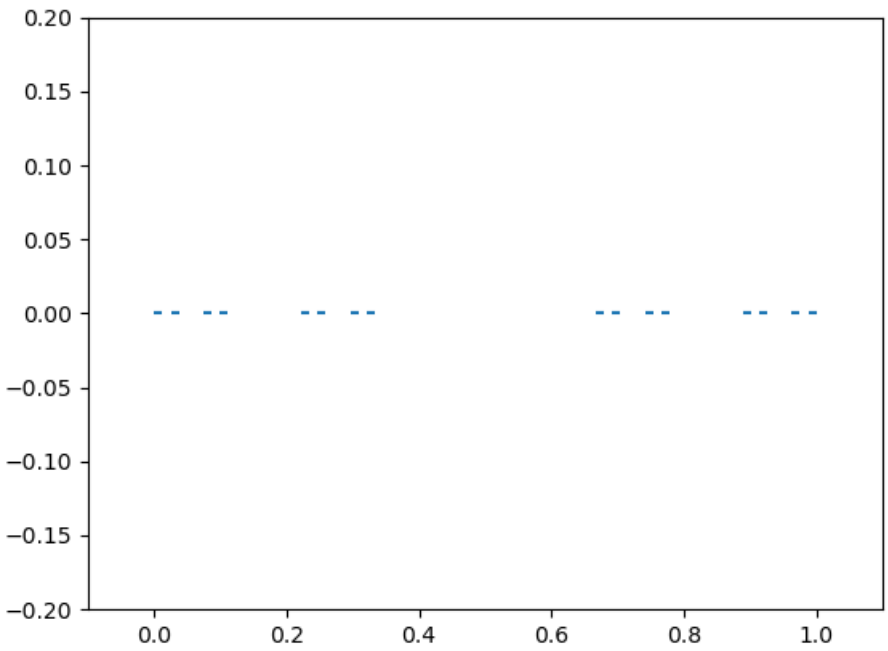
In [104]:
```
#
# Below are some functions that you might find useful
#

# needed for plotting many line segments
from matplotlib import collections

# If you want more context to understand this function, google "higher order functions"
def repeat(func, times=5):
    """Returns a function that applys `func` to its input
    `times` number of times."""
    def new_func(x):
        for _ in range(times):
            x = func(x)
        return x
    return new_func

def render_segments_to_array(segments, array, extent=[0, 1, 0, 1]):
    """Given a list of segments `segments` and a 2d numpy array `array`,
    "draw" the segments to the array. The resulting array is suitable for displaying
    with `imshow`. """
    from skimage.draw import line
    array = array.copy()
    h, w = array.shape

    for (p1, p2) in segments:
        # conver the xy-coordinates to array indices
        p1x = np.clip(int((p1[0] - extent[0]) / (extent[1] - extent[0]) * w), 0, w - 1)
        p2x = np.clip(int((p2[0] - extent[0]) / (extent[1] - extent[0]) * w), 0, w - 1)
        p1y = np.clip(int((p1[1] - extent[2]) / (extent[3] - extent[2]) * h), 0, h - 1)
        p2y = np.clip(int((p2[1] - extent[2]) / (extent[3] - extent[2]) * h), 0, h - 1)

        coords = line(p1y, p1x, p2y, p2x)
        array[coords] = 1
    return array
```

In [105]:
```
def cantorize(segments):
    """`segments` is a list of pairs of points which specify intervals. `cantorize` removes
    the middle third of each of these intervals and returns a resulting list of intervals."""
    ret = []
    for (p1, p2) in segments:
        # compute the end of the left third
        pL = (p1[0]*2/3+p2[0]/3, p1[1]*2/3+p2[1]/3)
        # add the left third
        ret.append( (p1, pL) )
        # compute the end of the right third
        pR = (p1[0]/3+p2[0]*2/3, p1[1]/3+p2[1]*2/3)
        # add the right third
        ret.append( (pR, p2) )
    return ret

# An interval is specified by a pair of points; Since we want
# a list of intervals, we make a list of lists of lists!
starting_segments = [ ((0,0), (1,0)) ]

print("`cantorize` removes the middle third of line segments in a list. "
      "It returns a list of the resulting (new) line segments. For example, "
      "When given\n",starting_segments,"\nIt produces\n", cantorize(starting_segments))
```

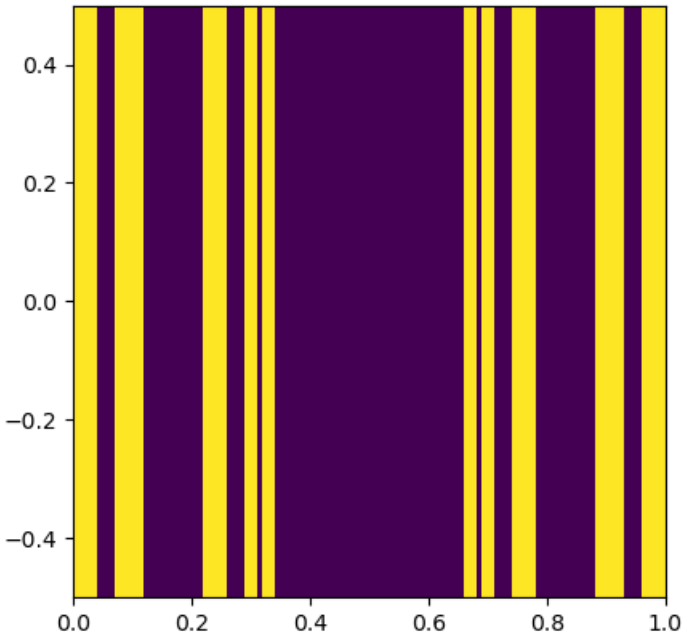
```
`cantorize` removes the middle third of line segments in a list. It returns a list of the res
ulting (new) line segments. For example, When given
 [((0, 0), (1, 0))]
It produces
 [((0, 0), (0.3333333333333333, 0.0)), ((0.6666666666666666, 0.0), (1, 0))]
```

Cantor Set

In [106]:
```
# create a function that repeats `cantroize` several times.
multi_cantorize = repeat(cantorize, 4)

fig, ax = plt.subplots()

# Turn our line segments into a matplotlib object
lines = collections.LineCollection(multi_cantorize(starting_segments))
# plot the line segments
ax.add_collection(lines)

# We can't use an extent when plotting just lines, so
# we set the bounds of the plot this way
ax.set_xlim(-.1, 1.1)
ax.set_ylim(-.2, .2)

plt.show()
```



In [107]:
```
N = 100
extent = [0, 1, -.5, .5]

cantor_intervals = multi_cantorize(starting_segments)
rendered_cantor = render_segments_to_array(cantor_intervals, np.zeros((1, N)), extent)

fig, ax = plt.subplots()

ax.imshow(rendered_cantor, cmap="viridis", extent=extent, origin="lower")

plt.show()

print("The rendered cantor set touches", rendered_cantor.sum(), "intervals of width", 1/N,
      "between", extent[0], "and", extent[1])
```




```
The rendered cantor set touches 34.0 intervals of width 0.01 between 0 and 1
```

In [109]:
```
#
# Estimate the box-counting dimension of the cantor set
#

# to use log, import math pack
import math
N = 10000
ns = np.linspace(2, 10000, N) # Test a bunch of n, taking n to infinity
extent = [0, 1, -.5, .5]

cantor_intervals = multi_cantorize(starting_segments)
ds = []
for num in ns:
    rendered_cantor = render_segments_to_array(cantor_intervals, np.zeros((1, int(num))), exten
t)
    number_of_boxes = rendered_cantor.sum()
    d = math.log(number_of_boxes)/math.log(num)
    ds.append(d)

print("The box-coundting dimension of the cantor set should be")
print(math.log(2)/math.log(3)) # class practice
print("compare to the estimated value by the computer program")
print(min(ds)) # take the minimum
```

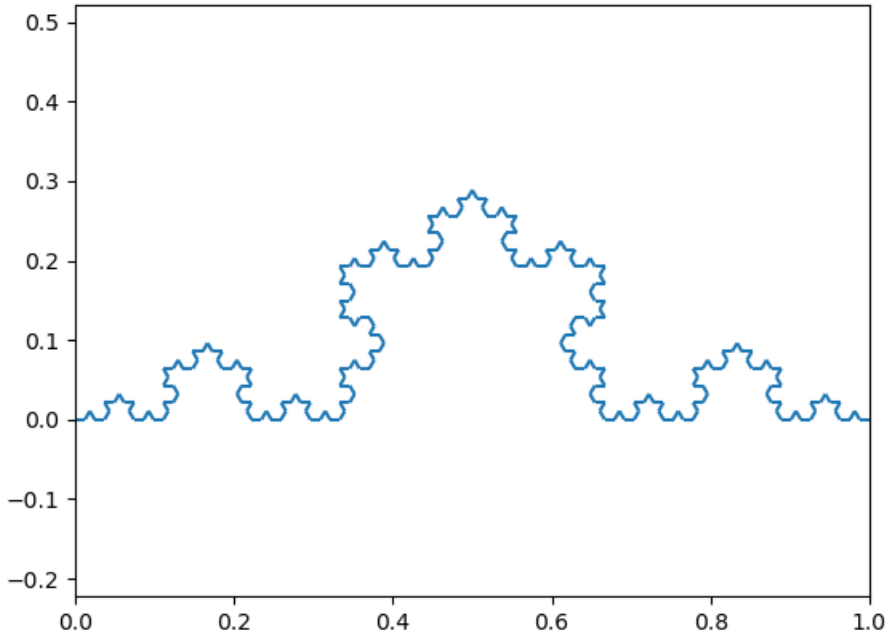
```
The box-coundting dimension of the cantor set should be
0.6309297535714574
compare to the estimated value by the computer program
0.6309488996197793
```

Koch Snowflake

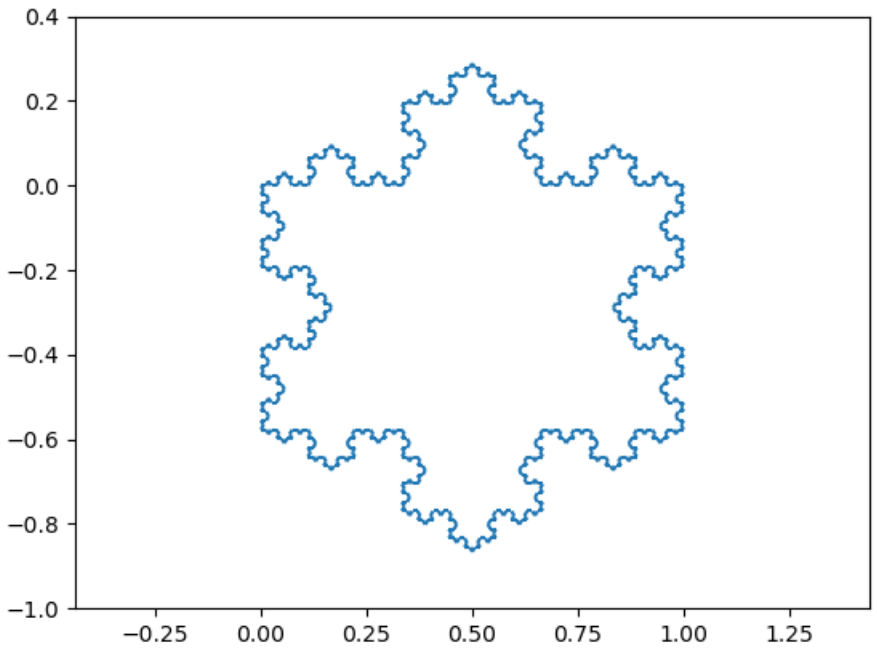
In [110]:
```
def kochize(segments):
    new_seg = []
    # label a side of the snowflake p1, p2, p3, p4, p5, where p1, p5 represents the end points
 of the initial
    # segment, and p2, p3, p4 forms the spike of the subsitution
    for (p1, p5) in segments:
        # compute the end of the left third, left first segment end = p2
        p2 = ((2*p1[0] + p5[0])/3, (2*p1[1] + p5[1])/3)
        # add the left third segment
        new_seg.append((p1, p2))
        # compute the start of the right third, right last segment start = p4
        p4 = ((p1[0] + 2*p5[0])/3, (p1[1] + 2*p5[1])/3)
        # compute the peak point of the spike, peak x-cord = p3x, peak y-cord = p3y
        # use rotation matrix of 60 degree and distance (dx, dy) between p2 and p4
        dx = p4[0] - p2[0]
        dy = p4[1] - p2[1]
        p3x = p2[0] + (dx/2 - (3)**(1/2)/2*dy)
        p3y = p2[1] + (3)**(1/2)/2*dx + dy/2
        p3 = (p3x, p3y)
        # add the left diagonal of the peak
        new_seg.append((p2, p3))
        # add the right diagonal of the peak
        new_seg.append((p3, p4))
        # add the last remaining line segment
        new_seg.append((p4, p5))
    return new_seg

starting_segments = [ ((0,0), (1,0)) ]

print("Applying one iteration of the Koch snowflake subsitution to\n", starting_segments,
      "\ngives\n", kochize(starting_segments))
```

```
Applying one iteration of the Koch snowflake subsitution to
 [((0, 0), (1, 0))]
gives
 [((0, 0), (0.3333333333333333, 0.0)), ((0.3333333333333333, 0.0), (0.5, 0.2886751345948128
7)), ((0.5, 0.28867513459481287), (0.6666666666666666, 0.0)), ((0.6666666666666666, 0.0), (1,
0))]
```

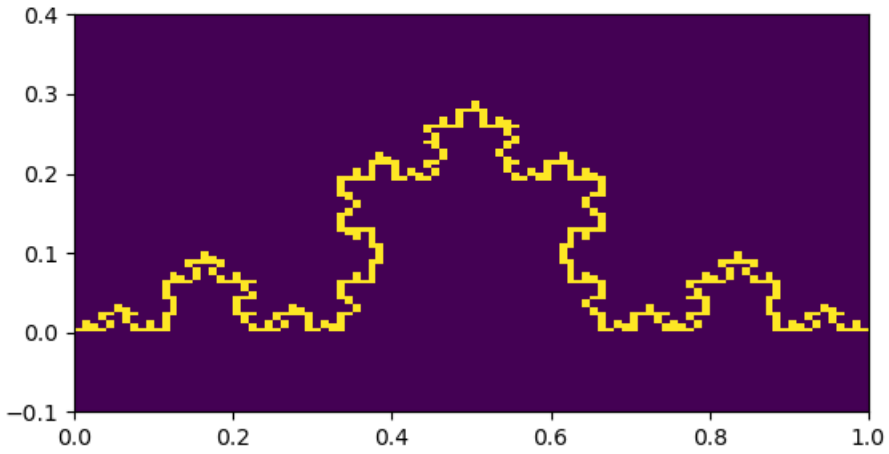
In [111]:
```
# create a function that repeats `kochize` several times.
multi_kochize = repeat(kochize, 4) #K_4, so repeates 4 times

fig, ax = plt.subplots()

# Turn our line segments into a matplotlib object
lines = collections.LineCollection(multi_kochize(starting_segments))
# plot the line segments
ax.add_collection(lines)

# Make sure the snowflake is in view
ax.set_ylim(-.1, .4)

# We want the graph to have equal scaling in all directions
# so that vertical lengths and horizontal lengths that are equal
# show up equal on the computer screen
ax.set_aspect('equal', 'datalim')

plt.show()
```



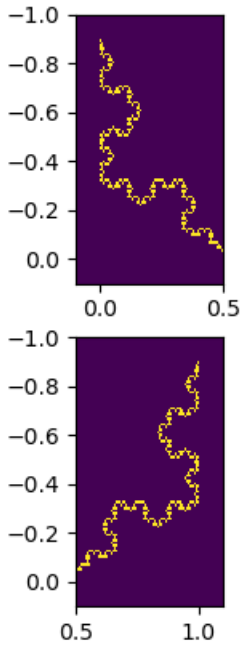
In [112]:
```
#
# Graph of the Koch snowflake (all sides, not just the top).
#

# create a function that repeats `kochize` several times.
multi_kochize = repeat(kochize, 4)

# top side of the snowflake
starting_segments = [ ((0,0), (1,0)) ]
# left bottom side of the snowflake, rotation CW 60 degree on end point (1,0) which produces
 (1/2, -(3)**(1/2)/2 )
starting_segment2 = [ ((1/2, -((3)**(1/2)/2)), (0,0)) ]
# right bottom side of the snowflake, connect the end point of segment 2 to end point of starti
ng_segments
starting_segment3 = [ ((1,0), (1/2, -((3)**(1/2)/2))) ]

fig, ax = plt.subplots()

# Turn our line segments into a matplotlib object
lines = collections.LineCollection(multi_kochize(starting_segments))
line2 = collections.LineCollection(multi_kochize(starting_segment2))
line3 = collections.LineCollection(multi_kochize(starting_segment3))

# plot the line segments
ax.add_collection(lines)
ax.add_collection(line2)
ax.add_collection(line3)

# Make sure the snowflake is in view
ax.set_ylim(-1, .4)

# We want the graph to have equal scaling in all directions
# so that vertical lengths and horizontal lengths that are equal
# show up equal on the computer screen
ax.set_aspect('equal', 'datalim')

plt.show()
```



In [113]:
```
N = 100
extent = [0, 1, -.1, .4]

koch_segments = multi_kochize(starting_segments)
rendered_koch = render_segments_to_array(koch_segments, np.zeros((N, N)), extent)

fig, ax = plt.subplots()

ax.imshow(rendered_koch, cmap="viridis", extent=extent, origin="lower")

plt.show()
```



In [114]:
```
#
# Estimate the box-counting dimension of the Koch snowflake
#

# compute each side, then take the box-counting dimension of the union of them = max bc dim of
 each side from Q6c)

# to use log, import math pack
import math
N = 100
ns = np.linspace(2, 100, N) # Test a bunch of n, taking n to infinity

# top side of the snowflake
starting_segments = [ ((0,0), (1,0)) ]
# left bottom side of the snowflake, rotation CW 60 degree on end point (1,0) which produces
 (1/2, -(3)**(1/2)/2 )
starting_segment2 = [ ((1/2, -((3)**(1/2)/2)), (0,0)) ]
# right bottom side of the snowflake, connect the end point of segment 2 to end point of starti
ng_segments
starting_segment3 = [ ((1,0), (1/2, -((3)**(1/2)/2))) ]

extent = [0, 1, -.1, .4]
extent2 = [-0.1, 0.5, 0.1, -1]
extent3 = [0.5, 1.1, 0.1, -1]

# top side line 1
koch_segments = multi_kochize(starting_segments)
rendered_koch = render_segments_to_array(koch_segments, np.zeros((N, N)), extent)
# same thing for the other two sides, line 2 and 3
koch_segments2 = multi_kochize(starting_segment2)
rendered_koch2 = render_segments_to_array(koch_segments2, np.zeros((N, N)), extent2)
koch_segments3 = multi_kochize(starting_segment3)
rendered_koch3 = render_segments_to_array(koch_segments3, np.zeros((N, N)), extent3)

# check if extent2, 3 is correct
fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=False)
ax1.imshow(rendered_koch2, cmap="viridis", extent=extent2, origin="upper")
ax2.imshow(rendered_koch3, cmap="viridis", extent=extent3, origin="upper")
plt.show()

def bc_dim(rendered_koch):
    ds = []
    for num in ns:
        number_of_boxes = rendered_koch.sum()
        d = math.log(number_of_boxes)/math.log(num)
        ds.append(d)

    return min(ds)

# minimum dimension outputs for line 1 with a bunch of N
ds1 = bc_dim(rendered_koch)
# minimum dimension outputs for line 2 with a bunch of N
ds2 = bc_dim(rendered_koch2)
# minimum dimension outputs for line 3 with a bunch of N
ds3 = bc_dim(rendered_koch3)

print("The similarity dimension of the koch snowflake should be")
print(math.log(4)/math.log(3)) # class practice
print("compare to the estimated value of box-counting dimension by the computer program for lin
e 1")
print(ds1)
print("and for line 2")
print(ds2)
print("and for line 3")
print(ds3)
print("They are very similar answers.")
```

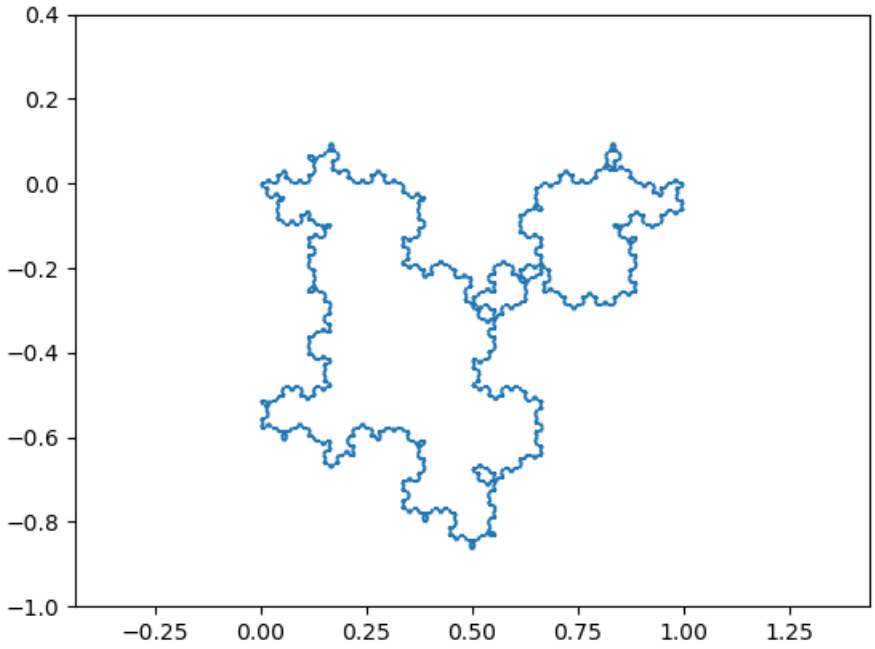



```
The similarity dimension of the koch snowflake should be
1.2618595071429148
compare to the estimated value of box-counting dimension by the computer program for line 1
1.3236914850573098
and for line 2
1.2720340221751376
and for line 3
1.272653558232912
They are very similar answers.
```

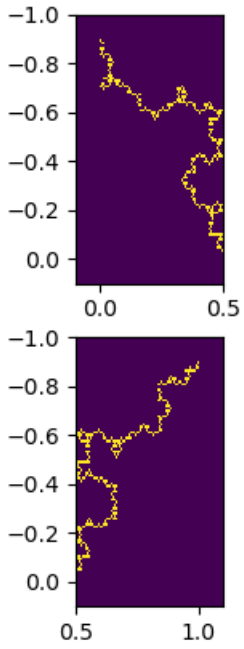
Strange Koch

In [115]:
```
# Setting the "seed" for numpy's random number generator will ensure you get the same sequence
# of random numbers each time you execute the function.
#
# After setting this, you can call np.random.uniform() multiple times to get a sequence of rand
om numbers
np.random.seed(10)
def strange_kochize(segments):
    new_seg = []
    # label a side of the snowflake p1, p2, p3, p4, p5, where p1, p5 represents the end points
 of the initial
    # segment, and p2, p3, p4 forms the spike of the subsitution
    for (p1, p5) in segments:
        # compute the end of the left third, left first segment end = p2
        p2 = ((2*p1[0] + p5[0])/3, (2*p1[1] + p5[1])/3)
        # add the left third segment
        new_seg.append((p1, p2))
        # compute the start of the right third, right last segment start = p4
        p4 = ((p1[0] + 2*p5[0])/3, (p1[1] + 2*p5[1])/3)
        # compute the peak point of the spike, peak x-cord = p3x, peak y-cord = p3y
        # use rotation matrix of 60 degree and distance (dx, dy) between p2 and p4
        dx = p4[0] - p2[0]
        dy = p4[1] - p2[1]
        p3x = p2[0] + (dx/2 - (3)**(1/2)/2*dy)
        p3y = p2[1] + (3)**(1/2)/2*dx + dy/2
        if np.random.uniform() < .5: # spike up
            p3 = (p3x, p3y)
        else: # spike down, rotate the orginal vector of p3 60 degree CCW,
            m = (p5[0]-p1[0], p5[1]-p5[1])
            p3x_down = p2[0] + (dx/2 + (3)**(1/2)/2*dy)
            p3y_down = p2[1] + -(3)**(1/2)/2*dx + dy/2
            p3 = (p3x_down, p3y_down)
        # add the left diagonal of the peak
        new_seg.append((p2, p3))
        # add the right diagonal of the peak
        new_seg.append((p3, p4))
        # add the last remaining line segment
        new_seg.append((p4, p5))
    return new_seg


#
# Make a plot of the strange Koch snowflake
#

# create a function that repeats `strange_kochize` several times.
multi_strange_kochize = repeat(strange_kochize, 4)

# top side of the snowflake
starting_segments = [ ((0,0), (1,0)) ]
# left bottom side of the snowflake, rotation CW 60 degree on end point (1,0) which produces
 (1/2, -(3)**(1/2)/2 )
starting_segment2 = [ ((1/2, -((3)**(1/2)/2)), (0,0)) ]
# right bottom side of the snowflake, connect the end point of segment 2 to end point of starti
ng_segments
starting_segment3 = [ ((1,0), (1/2, -((3)**(1/2)/2))) ]

fig, ax = plt.subplots()

# Turn our line segments into a matplotlib object
lines = collections.LineCollection(multi_strange_kochize(starting_segments))
line2 = collections.LineCollection(multi_strange_kochize(starting_segment2))
line3 = collections.LineCollection(multi_strange_kochize(starting_segment3))

# plot the line segments
ax.add_collection(lines)
ax.add_collection(line2)
ax.add_collection(line3)

# Make sure the snowflake is in view
ax.set_ylim(-1, .4)
```

```
# We want the graph to have equal scaling in all directions
# so that vertical lengths and horizontal lengths that are equal
# show up equal on the computer screen
ax.set_aspect('equal', 'datalim')

plt.show()
```



In [116]:
```
#
# Estimate the fractal dimension of the strange Koch snowflake
#
# to use log, import math pack
import math
N = 100
ns = np.linspace(2, 100, N) # Test a bunch of n, taking n to infinity

# top side of the snowflake
starting_segments = [ ((0,0), (1,0)) ]
# left bottom side of the snowflake, rotation CW 60 degree on end point (1,0) which produces
 (1/2, -(3)**(1/2)/2 )
starting_segment2 = [ ((1/2, -((3)**(1/2)/2)), (0,0)) ]
# right bottom side of the snowflake, connect the end point of segment 2 to end point of starti
ng_segments
starting_segment3 = [ ((1,0), (1/2, -((3)**(1/2)/2))) ]

extent = [0, 1, -.1, .4]
extent2 = [-0.1, 0.5, 0.1, -1]
extent3 = [0.5, 1.1, 0.1, -1]

# top side line 1
koch_segments = multi_strange_kochize(starting_segments)
rendered_koch = render_segments_to_array(koch_segments, np.zeros((N, N)), extent)
# same thing for the other two sides, line 2 and 3
koch_segments2 = multi_strange_kochize(starting_segment2)
rendered_koch2 = render_segments_to_array(koch_segments2, np.zeros((N, N)), extent2)
koch_segments3 = multi_strange_kochize(starting_segment3)
rendered_koch3 = render_segments_to_array(koch_segments3, np.zeros((N, N)), extent3)

# check if extent2, 3 is correct
fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=False)
ax1.imshow(rendered_koch2, cmap="viridis", extent=extent2, origin="upper")
ax2.imshow(rendered_koch3, cmap="viridis", extent=extent3, origin="upper")
plt.show()

def bc_dim(rendered_koch):
    ds = []
    for num in ns:
        number_of_boxes = rendered_koch.sum()
        d = math.log(number_of_boxes)/math.log(num)
        ds.append(d)

    return min(ds)

# minimum dimension outputs for line 1 with a bunch of N
ds1 = bc_dim(rendered_koch)
# minimum dimension outputs for line 2 with a bunch of N
ds2 = bc_dim(rendered_koch2)
# minimum dimension outputs for line 3 with a bunch of N
ds3 = bc_dim(rendered_koch3)

print("The similarity dimension of the strange koch snowflake should be")
print(math.log(4)/math.log(3)) # class practice
print("compare to the estimated value of box-counting dimension by the computer program for lin
e 1")
print(ds1)
print("and for line 2")
print(ds2)
print("and for line 3")
print(ds3)
print("They are again very similar answers.")
```

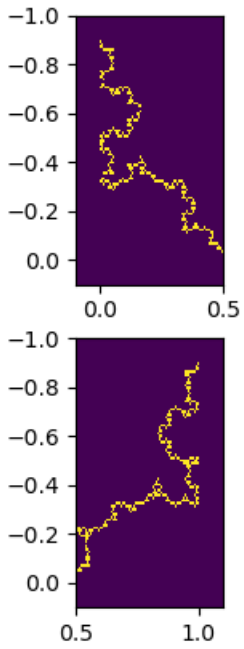



```
The similarity dimension of the strange koch snowflake should be
1.2618595071429148
compare to the estimated value of box-counting dimension by the computer program for line 1
1.3241800054904656
and for line 2
1.2638149504356693
and for line 3
1.2612221167531599
They are again very similar answers.
```

Next, we want to define a different probability for strange koch snowfalke with 75% pointing up, and 25% pointing down. We want to see if this changes the dimension of the resulting fractal. My guess is *NO*.

In [117]:
```
# define the ver2 of strange kochize, with 75% pointing up, and 25% pointing down
def strange_kochize_ver2(segments):
    new_seg = []
    # label a side of the snowflake p1, p2, p3, p4, p5, where p1, p5 represents the end points
 of the initial
    # segment, and p2, p3, p4 forms the spike of the subsitution
    for (p1, p5) in segments:
        # compute the end of the left third, left first segment end = p2
        p2 = ((2*p1[0] + p5[0])/3, (2*p1[1] + p5[1])/3)
        # add the left third segment
        new_seg.append((p1, p2))
        # compute the start of the right third, right last segment start = p4
        p4 = ((p1[0] + 2*p5[0])/3, (p1[1] + 2*p5[1])/3)
        # compute the peak point of the spike, peak x-cord = p3x, peak y-cord = p3y
        # use rotation matrix of 60 degree and distance (dx, dy) between p2 and p4
        dx = p4[0] - p2[0]
        dy = p4[1] - p2[1]
        p3x = p2[0] + (dx/2 - (3)**(1/2)/2*dy)
        p3y = p2[1] + (3)**(1/2)/2*dx + dy/2
        if np.random.uniform() < .75: # spike up
            p3 = (p3x, p3y)
        else: # spike down, rotate the orginal vector of p3 60 degree CCW,
            m = (p5[0]-p1[0], p5[1]-p5[1])
            p3x_down = p2[0] + (dx/2 + (3)**(1/2)/2*dy)
            p3y_down = p2[1] + -(3)**(1/2)/2*dx + dy/2
            p3 = (p3x_down, p3y_down)
        # add the left diagonal of the peak
        new_seg.append((p2, p3))
        # add the right diagonal of the peak
        new_seg.append((p3, p4))
        # add the last remaining line segment
        new_seg.append((p4, p5))
    return new_seg
```

In [118]:
```
#
# Estimate the fractal dimension of the ver2 strange Koch snowflake with 75/25
#
# to use log, import math pack
import math
N = 100
ns = np.linspace(2, 100, N) # Test a bunch of n, taking n to infinity

# top side of the snowflake
starting_segments = [ ((0,0), (1,0)) ]
# left bottom side of the snowflake, rotation CW 60 degree on end point (1,0) which produces
 (1/2, -(3)**(1/2)/2 )
starting_segment2 = [ ((1/2, -((3)**(1/2)/2)), (0,0)) ]
# right bottom side of the snowflake, connect the end point of segment 2 to end point of starti
ng_segments
starting_segment3 = [ ((1,0), (1/2, -((3)**(1/2)/2))) ]

extent = [0, 1, -.1, .4]
extent2 = [-0.1, 0.5, 0.1, -1]
extent3 = [0.5, 1.1, 0.1, -1]

# Define recrusive function to draw its fractal
multi_strange_kochize_ver2 = repeat(strange_kochize_ver2, 4)

# top side line 1
koch_segments = multi_strange_kochize_ver2(starting_segments)
rendered_koch = render_segments_to_array(koch_segments, np.zeros((N, N)), extent)
# same thing for the other two sides, line 2 and 3
koch_segments2 = multi_strange_kochize_ver2(starting_segment2)
rendered_koch2 = render_segments_to_array(koch_segments2, np.zeros((N, N)), extent2)
koch_segments3 = multi_strange_kochize_ver2(starting_segment3)
rendered_koch3 = render_segments_to_array(koch_segments3, np.zeros((N, N)), extent3)

# check if extent2, 3 is correct
fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=False)
ax1.imshow(rendered_koch2, cmap="viridis", extent=extent2, origin="upper")
ax2.imshow(rendered_koch3, cmap="viridis", extent=extent3, origin="upper")
plt.show()

def bc_dim(rendered_koch):
    ds = []
    for num in ns:
        number_of_boxes = rendered_koch.sum()
        d = math.log(number_of_boxes)/math.log(num)
        ds.append(d)

    return min(ds)

# minimum dimension outputs for line 1 with a bunch of N
ds1 = bc_dim(rendered_koch)
# minimum dimension outputs for line 2 with a bunch of N
ds2 = bc_dim(rendered_koch2)
# minimum dimension outputs for line 3 with a bunch of N
ds3 = bc_dim(rendered_koch3)

print("The similarity dimension of the strange koch snowflake of 75/25 probability of pointing
 up and down should be")
print(math.log(4)/math.log(3)) # class practice
print("compare to the estimated value of box-counting dimension by the computer program for lin
e 1")
print(ds1)
print("and for line 2")
print(ds2)
print("and for line 3")
print(ds3)
print("They are all very similar answers again.")
print("My guess was correct! The change of chances to point up or down did not change the dimen
sion.")
print("This makes senses since the antikoch and koch snowflake has the same box-counting dimens
ion, and we can break finite union of disjoing subsets of the snowflake to get its dimension si
nce it is a nice enough set. ")
```

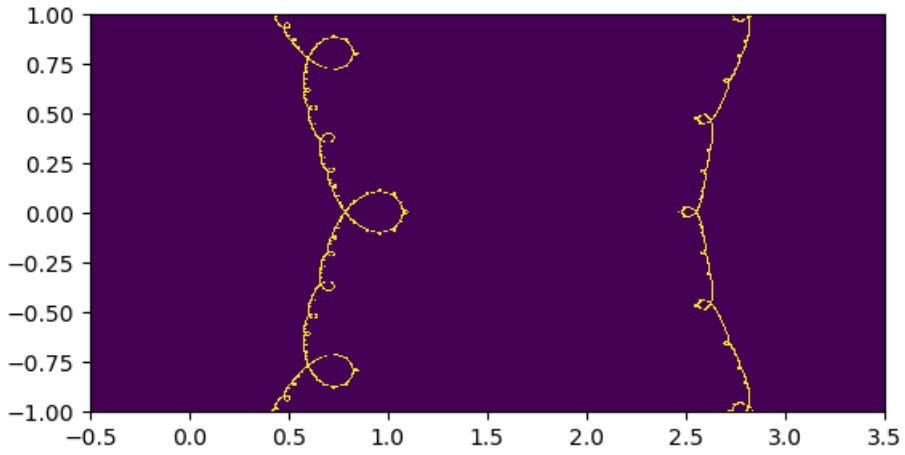



```
The similarity dimension of the strange koch snowflake of 75/25 probability of pointing up an
d down should be
1.2618595071429148
compare to the estimated value of box-counting dimension by the computer program for line 1
1.2491552768948002
and for line 2
1.2657394585211275
and for line 3
1.2625224035184224
They are all very similar answers again.
My guess was correct! The change of chances to point up or down did not change the dimension.
This makes senses since the antikoch and koch snowflake has the same box-counting dimension,
and we can break finite union of disjoing subsets of the snowflake to get its dimension since
it is a nice enough set.
```

Boundary Dimensions

In [119]:
```
# A utility function for finding the boundary between constant sets in an array.
# You can also experiment with the `canny` function from the `skimage.feature`
# module, which will do edge-detection on non-constant/noisy data
def find_edges(array):
    """Inputs a 2d array, `array`, and outputs an array containing 0s and 1s.
    If adjacent entries in `array` differ to the right, below, or diagonally, the returned
    array has a 1 in that position. Otherwise, it has a 0.

    The returned array is one smaller each dimension (rows and columns)"""
#     # make find_edges less strict
#     tolerance = 0.00001
    diff_right = (array[:-1, :-1] - array[:-1, 1:]) != 0
    diff_down = (array[:-1, :-1] - array[1:, :-1]) != 0
    diff_diag = (array[:-1, :-1] - array[1:, 1:]) != 0

    return (diff_right + diff_down + diff_diag != 0).astype(int)
```

In [120]:
```
#
# Graph the boundary of the Newton fractal determined by f(x)=x(x-2)(x-3) for complex grid
#

# set up
def f(x):
    return x*(x-2)*(x-3)

def f_prime(x):
    return 3*x*x-10*x+6

def make_complex_grid(z_low, z_high, N=100):
    """Create an N x N  2d array of complex numbers whose lower-left
    corner is give by `z_low` and upper-right corner is given by `z_high`"""
    reals = np.linspace(np.real(z_low), np.real(z_high), N)
    imags = np.linspace(np.imag(z_low), np.imag(z_high), N)
    a, b = np.meshgrid(reals, imags)
    return a + b*1J

def newt2(guess_array, iterations=20):
    for i in range(iterations):
        guess_array = T_f(guess_array)
    return guess_array

def clamped_newt(guess_array, iterations=20):
    for i in range(iterations):
        guess_array = newt2(guess_array)
    np.round_(guess_array)

    return np.real(guess_array)

# plot the boundary
# N is how many points we will sample
N = 500
zs = make_complex_grid(-.5-1j, 3.5+1j, N)

fig, ax = plt.subplots()

extent = [np.real(zs).min(), np.real(zs).max(), np.imag(zs).min(), np.imag(zs).max()]

ax.imshow(find_edges(clamped_newt(zs)), cmap="viridis", extent=extent, origin="lower")

plt.show()
```



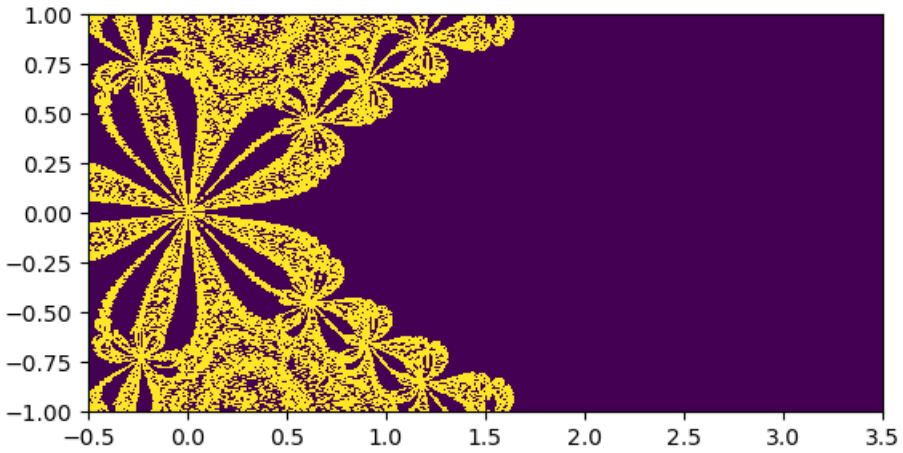
In [123]:
```
#
# Estimate the fractal dimension of the boundary of the Newton fractal given by f(x)=x(x-2)(x-3)
#

# to use log, import math pack
import math
N = 200
ns = np.linspace(2, 200, N) # Test a bunch of n, taking n to infinity
zs = make_complex_grid(-.5-1j, 3.5+1j, N)
boundary = find_edges(clamped_newt(zs)) # rendered_newton

# #plot it to see newton fractal agiain
# fig, ax = plt.subplots()

# extent = [np.real(zs).min(), np.real(zs).max(), np.imag(zs).min(), np.imag(zs).max()]

# ax.imshow(boundary, cmap="viridis", extent=extent, origin="lower")

# box-couting dimension function we have used in previous questions
def bc_dim(rendered_newton):
    ds = []
    for num in ns:
        number_of_boxes = rendered_newton.sum()
        d = math.log(number_of_boxes)/math.log(num)
        ds.append(d)

    return min(ds)

# minimum dimension outputs with a bunch of N
d_boundary = bc_dim(boundary)


print("The estimated value of box-counting dimension of the newton fractal of f by the computer program is")
print(d_boundary)
```

```
The estimated value of box-counting dimension of the newton fractal of f by the computer prog
ram is
1.7320965352745683
```

Let's graph a Newton fractal whose boundary has higher dimension than the one generated by $f(z) = z(z-2)(z-3)$!

I pick our new function $f(z) = z^5 - 1$:)

In [124]:
```
# set up
def f(x):
    return x**5 - 1

def f_prime(x):
    return 5*x**4

def make_complex_grid(z_low, z_high, N=100):
    """Create an N x N  2d array of complex numbers whose lower-left
    corner is give by `z_low` and upper-right corner is given by `z_high`"""
    reals = np.linspace(np.real(z_low), np.real(z_high), N)
    imags = np.linspace(np.imag(z_low), np.imag(z_high), N)
    a, b = np.meshgrid(reals, imags)
    return a + b*1J

def newt2(guess_array, iterations=20):
    for i in range(iterations):
        guess_array = T_f(guess_array)
    return guess_array

def clamped_newt(guess_array, iterations=20):
    for i in range(iterations):
        guess_array = newt2(guess_array)
    np.round_(guess_array)

    return np.real(guess_array)

# plot the boundary of the new f
# N is how many points we will sample
N = 500
zs = make_complex_grid(-.5-1j, 3.5+1j, N)

fig, ax = plt.subplots()

extent = [np.real(zs).min(), np.real(zs).max(), np.imag(zs).min(), np.imag(zs).max()]

ax.imshow(find_edges(clamped_newt(zs)), cmap="viridis", extent=extent, origin="lower")

plt.show() # this is actually so pretty, good pick 10 minutes ago me

# next, let's check what the dimension of this fractal is, and compare with the old f newton fr
actal

N = 200
ns = np.linspace(2, 200, N) # Test a bunch of n, taking n to infinity
boundary = find_edges(clamped_newt(zs)) # rendered_newton

# minimum dimension outputs with a bunch of N
d_new = bc_dim(boundary)


print("The estimated value of box-counting dimension of the newton fractal of f(z) = z^5 - 1 by
the computer program is")
print(d_new)
```




```
The estimated value of box-counting dimension of the newton fractal of f(z) = z^5 - 1 by the
computer program is
2.048097533691669
```

We can clearly see that the box-counting dimension for the newtown fractal of the boundary of $f(z) = z(z-2)(z-3)$ is about **1.732**, while the box-counting dimension for the newtown fractal of the boundary of $f(z) = z^5 - 1$ is about **2.048**, as needed.

Below are codes for the written component of the homework, not relevant.

In [144]:
```
# written 5 d)
def f(x):
    return 2/(x**2+1)-5/4

def preimage(n):
    return [(((n+5/4)**(-1)*2-1)**(1/2)), -(((n+5/4)**(-1)*2-1)**(1/2))]

print(preimage(0))
print([(3/5)**(1/2), -(3/5)**(1/2)])
print(preimage((3/5)**(1/2)), preimage(-(3/5)**(1/2)))

result = preimage(0)
# for i in range(20):
#     for j in range(result):
#         result[j] = preimage(result[j])
#         i+=1

# return result
```

```
[0.7745966692414834, -0.7745966692414834]
[0.7745966692414834, -0.7745966692414834]
[(6.749160367558914e-18+0.11022215339570467j), (-6.749160367558914e-18-0.11022215339570467j)]
[1.7907970621777278, -1.7907970621777278]
```