Homework 1

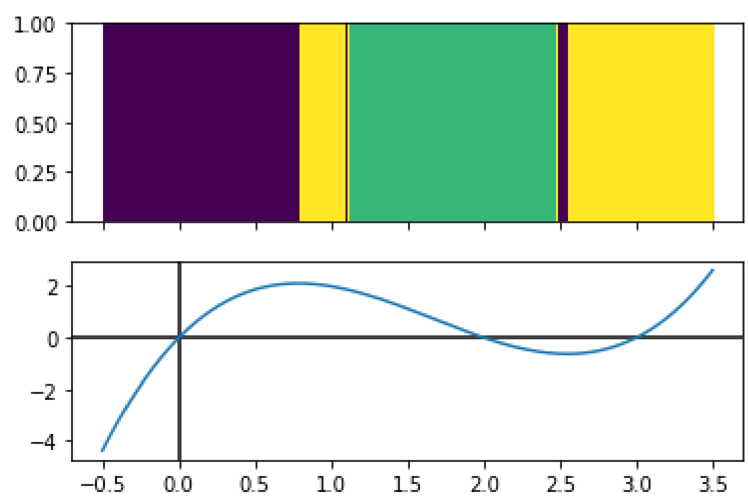
Do the programing part of Homework 1 in this notebook. Predefined are function *stubs*. That is, the name of the function and a basic body is predefined. You need to modify the code to fulfil the requirements of the homework.

In [1]:
```
# import numpy and matplotlib
import numpy as np
import matplotlib.pylab as plt
# We give the matplotlib instruction twice, because firefox sometimes gets
# note these `%`-commands are not actually Python commands. They are Jupyte
%matplotlib notebook
%matplotlib notebook
```

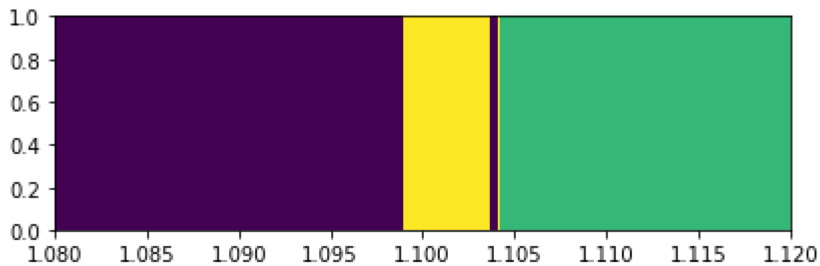
In [47]:
```
def f(x):
    return x*(x-2)*(x-3)

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

def T_f(guess):
    #do one iteration of the newton method. f_prime(guess) != 0
    return guess - (f(guess)/f_prime(guess))

def newt(guess, max_iterates = 20, tolerance=0.0001):

    # Repeat a maximum of 20 times or until our guess is close enough to a
    while(max_iterates > 0 or not abs(f(guess)) <= tolerance):
        guess = T_f(guess)
        max_iterates -= 1      #We do not want the max iterations to exceed

    #reutn root or np.nan if no root is found
    return guess if abs(f(guess)) <= tolerance else  np.nan

v_newt = np.vectorize(newt)
```

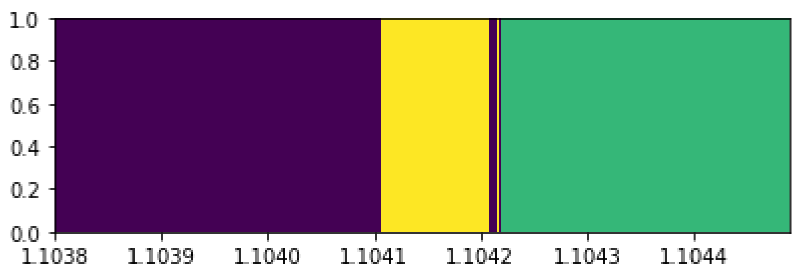
In [3]:
```
# N is how many points we will sample
N = 500
xs = np.linspace(-.5, 3.5, N)
ys = v_newt(xs)

# 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[3]:
```
[<matplotlib.lines.Line2D at 0x111da6f10>]
```



In [4]:
```
# Make a graph that is "zoomed-in" to the boundary between two basins of at
# Sample 500 points from the interval [1.08, 1.12]
xs1 = np.linspace(1.08, 1.12, 500)
ys1 = v_newt(xs1)

# Sample 1000 points from the interval [1.1038, 1.10449]
# Resolution was increased to show the repetitive structure would still hol
xs2 = np.linspace(1.1038, 1.10449, 1000)
ys2 = v_newt(xs2)

# Create a figure with two plots stacked vertically. They will not share x
# of each other
fig, (ax1, ax2) = plt.subplots(nrows=2)

ax1.imshow(np.array([ys1]), extent=[xs1.min(), xs1.max(), 0, 1], aspect="au
ax2.imshow(np.array([ys2]), extent=[xs2.min(), xs2.max(), 0, 1], aspect="au
plt.tight_layout()
```





In [ ]:

Complex Newton's Method

In [46]:
```
#
# 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 b*1J + a
```

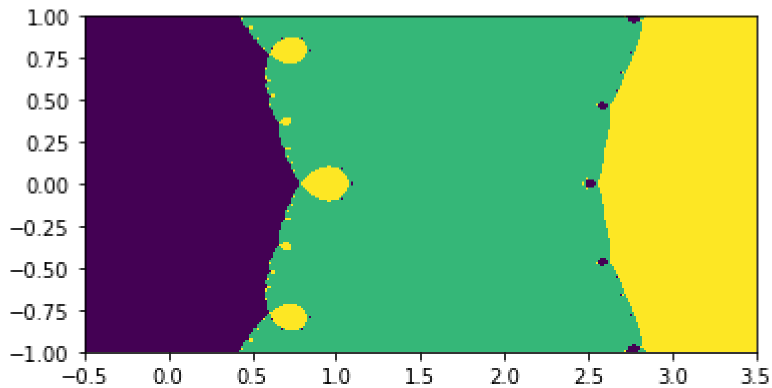
In [7]:
```
def newt_second(guess_array, tries=20):
    # do the iteration 20 times or until a root is found
    while(tries > 0):
        guess_array = T_f(guess_array)
        tries -= 1

    #reutn root or np.nan if no root is found
    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)
```

```
v_newt and newt2 should give similar results. v_newt:
 [3.         2.         3.         3.         3.00000012]
 and newt2:
 [3.         2.         3.         3.         3.01520126]
```

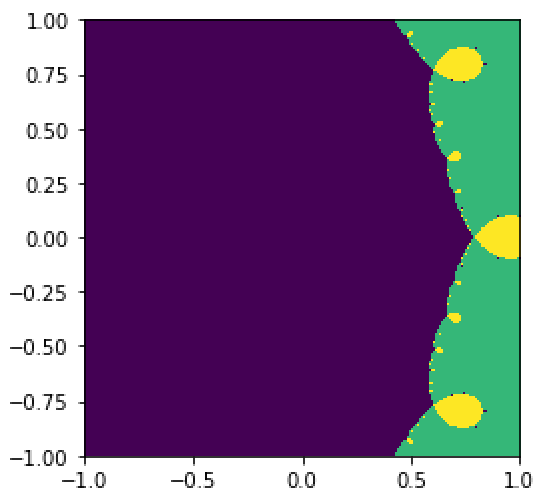
In [8]:
```
# There are many numpy functions that will be helpful for implementing `cla
# For example, `np.round`. You can google for these.
#
# Another helpful tip is fancy indexing: If `xs` is an array, to set
# all elements of `xs` which are less than four to zero, you can do
# `xs[ xs < 4 ] = 0`. To set all elements of `xs` that are less than four b
# than three to zero, you can do `xs[ (xs < 4) & (xs > 3) ] = 0`. Note the
#
def clamped_newt(ga, iterations=20):
    ga = newt2(ga, iterations)
    ga[ ga < 1 ] = 0
    ga[ (1 <= ga) & (ga < 2.5) ] = 2
    ga[ 2.5 <= ga] = 3
    return np.real(np.round(ga))

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

```
clamped_newt should the same outputs as newt2, but rounded to the nearest
root. newt2:
 [3.         2.         3.         3.         3.01520126]
 and clamped_newt:
 [3. 2. 3. 3. 3.]
```

In [9]:
```
N = 1000
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(

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

plt.show()
```



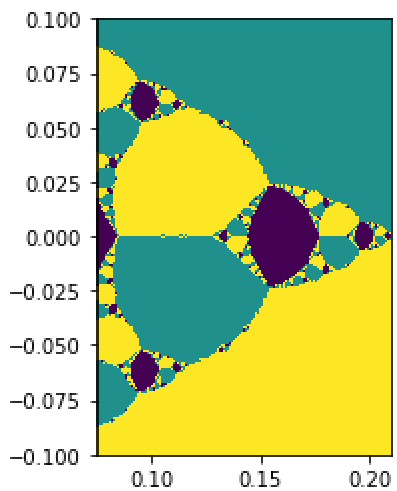
In [10]:
```
N = 1000
zs = make_complex_grid(-1-1j, 1+1j, N)

fig, ax = plt.subplots()

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

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

plt.show()
```



A new function

In [41]:
```
def f_new(x):
    return 3*x**3 - 2*x + 1

def f_new_prime(x):
    return 9*(x**2) - 2

def my_fractal(x, iterations = 30):
    #similar to clamped newt but for function 3*x^3 - 2*x + 1
    x = newt2(x, iterations)
    x[np.isreal(x)] = 1
    x[np.imag(x) > 0] = 2
    x[np.imag(x) < 0] = 3
    return np.real(np.round(x))
```

In [30]:
```
N = 500
zs = make_complex_grid(.075-0.1j, 0.21+0.1j, N)

fig, ax = plt.subplots()

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

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

plt.show()
```



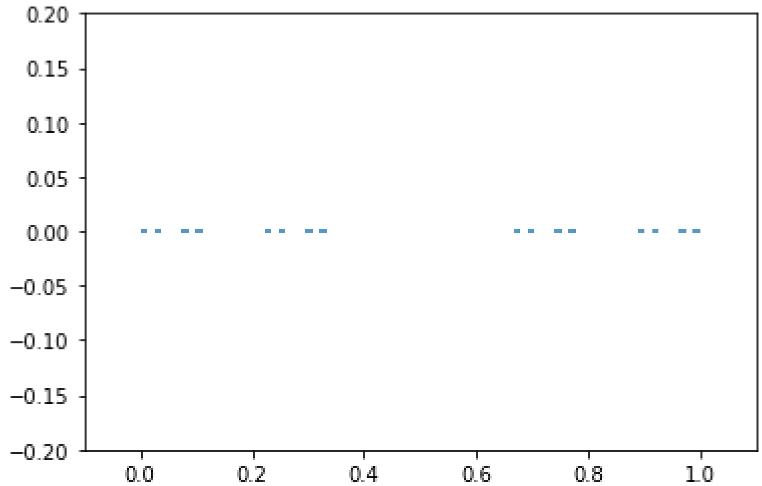
Common Fractals

In [31]:
```python
#
# 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 orde
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 d
    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
        p2x = np.clip(int((p2[0] - extent[0]) / (extent[1] - extent[0]) * w
        p1y = np.clip(int((p1[1] - extent[2]) / (extent[3] - extent[2]) * h
        p2y = np.clip(int((p2[1] - extent[2]) / (extent[3] - extent[2]) * h

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

In [ ]:

Cantor Set

In [33]:
```
# create a function that repeats `cantroize` several times.
multi_cantorize = repeat(cantorize, 4)

fig, ax = plt.subplots()

lines = collections.LineCollection(multi_cantorize(starting_segments))
# plot line
ax.add_collection(lines)


ax.set_xlim(-.1, 1.1)
ax.set_ylim(-.2, .2)

plt.show()
```



In [ ]:

In [ ]:

Koch Snowflake

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [45]:
```
#
# Estimate the box-counting dimension of the Koch snowflake
#
extent = [0, 1, -.5, .5]
#different box lengths
lengths = np.linspace(10, 100, 50)

first_part = [((0,0), (1,0))]
koch_parts = kochize(starting_segments)

x = np.log(lengths)
y = np.log([render_segments_to_array(koch_segments, np.zeros((1, int(N))),
# The best fit. The slope approximates dimension.
slope, intercept = np.polyfit(x, y, 1)
print("The dimension is approximately", slope)
```

```
The dimension is approximately 1.0130760162782304
```

Strange Koch

In [ ]:

In [ ]:

Boundary Dimensions

In [ ]:

In [26]:
```
#
# Graph the boundary of the Newton fractal determined by f(x)=x(x-2)(x-3)
#
```

In [ ]:
```
#
# Estimate the fractal dimension of the boundary of the Newton fractal give
#
```

In [ ]: