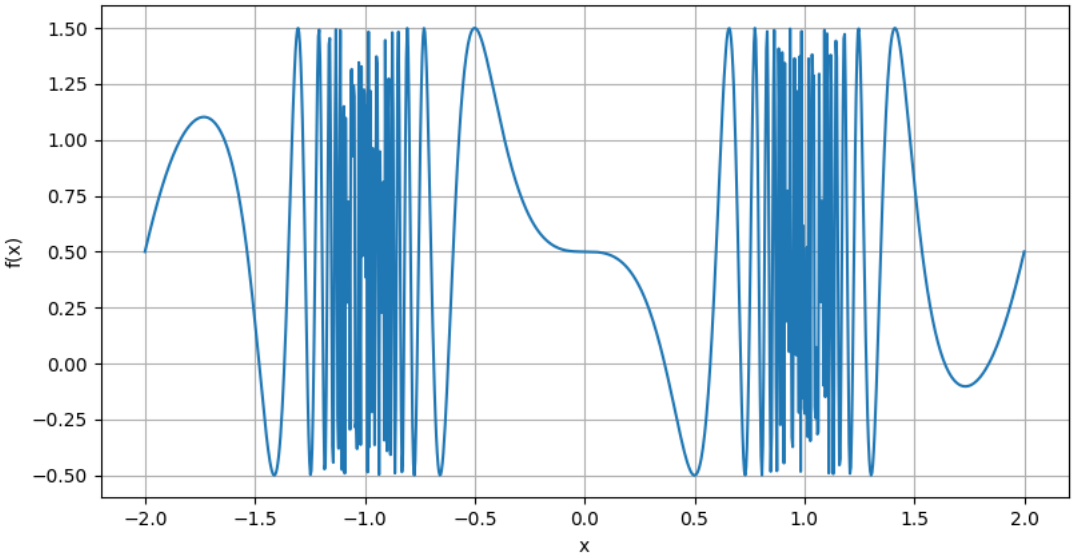
1

Both functions are located at roots.py and can run using python(3). Numpy and matplotlib are prerequisites (matplotlib used only for plotting and can be removed).

The function looks as



We can see that the function has roots in many places, then it has many roots depending on the interval.

C

For the bisection method at the interval [0,1] we get 0.363922119140625 which from the plot we can see is the minimum.
For the newton method, I used 0.1 as x_0. This value is the one which gave me a similar value to the bisection (0.36392289135578965) which is the minimal root in the [0, 1] interval.

When we run both with the `test_both()` function, we get the following output

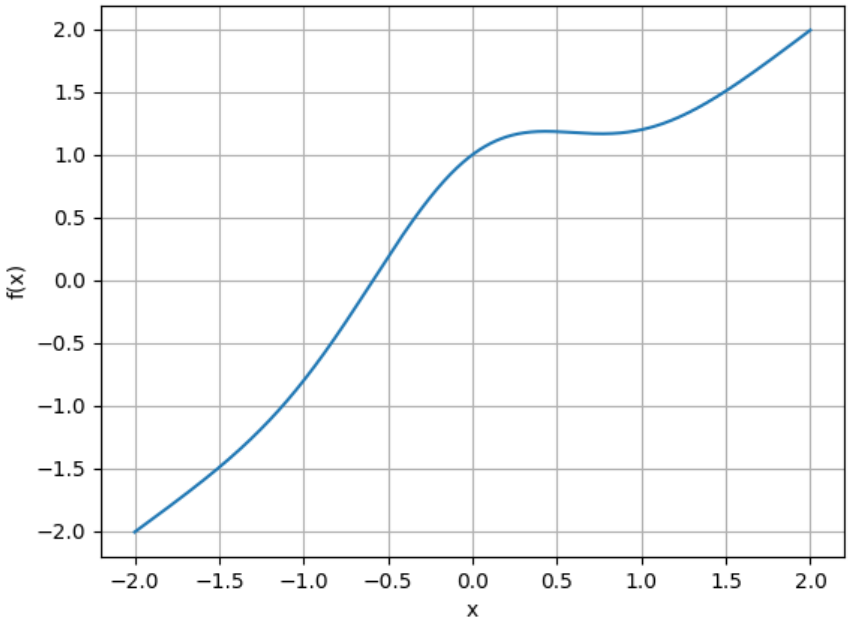
```
The bisection root is: 0.363922119140625, which evaluates
f(x)=2.091062964781898e-06
 it took it 7.987022399902344e-05s to evaluate
The newton root is: 0.36392289135578965, which evaluates
f(x)=-1.089259684672328e-06
 it took it 2.3126602172851562e-05s to evaluate
```

We notice that the bisection method took at least3 times longer than the Newton method.
The Newton method has a quadratic rate of convergence, it actually uses the direction of the function in order to get to the 0 point. While the bisection method just bisects it in the middle - regardless of how the function might be built

Of Course Newton's method has limitations, we must know the derivative and there are cases where it won't work.
While the bisection method will always find a root if it exists.

2

You can use the function test_g(B) to test the result based on B. Run with python(3)
Plotting the functions with B=1 we get



For this we get `x=-0.7164262100136756` and it fits the plot.
Then as we go higher in B, the "bumb" gets less wide.

B

Using x`0 = -0.5` for all

```
B=0.1     x = -0.7164262100136756
B=1        x = -0.7164262100136756
B=10   x = -0.3264020096329124
```

At B=100 we get a “maximum recursion depth exceeded”
Trying `x0=-0.1` we get `x = -0.13989456457805396`

I think that because the function is almost linear, but the bump is exactly at the x = 0 point, the Newton method might get stuck between two points and never progress.

Then when we start at `x0 = 0`, x_i is again jumping between 0 and 1.
As when taking the tangent from the top of the “bump” at g(0) we get a straight line to g(1), and then back to 0. Again just stuck between the two.

3

[3] $x_{i+1} = \mu x_i(1-x_i) = x_i - \frac{f(x_i)}{f'(x_i)}$ Solving it as ODE

$\mu x(1-x) = x - \frac{y}{\frac{dy}{dx}} \Rightarrow \mu x(1-x) - x = -\frac{dx \cdot y}{dy}$

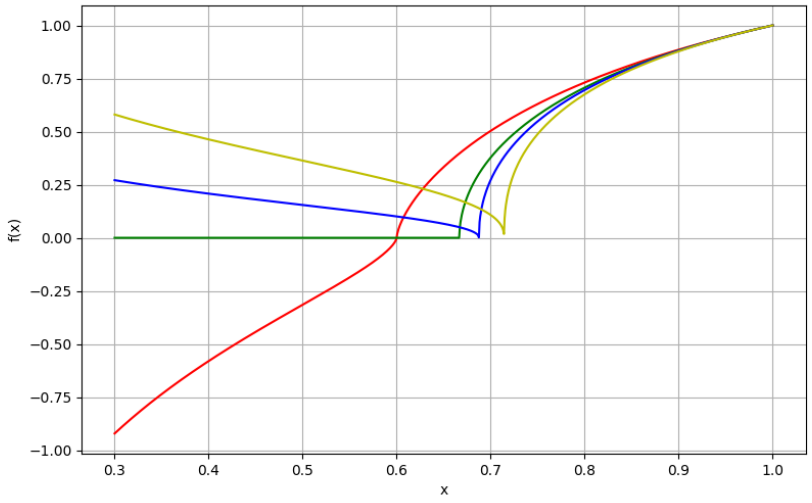
$\frac{dy}{y} = \frac{dx}{-\mu x(1-x)+x}$ integrating both sides

$\ln y = \frac{\ln(\mu(x-1)+1) - \ln x}{\mu - 1}$ exp both sides

$y = \left(e^{\ln(\mu(x-1)+1) - \ln x}\right)^{\frac{1}{\mu-1}} = \left(\frac{\mu(x-1)+1}{x}\right)^{\frac{1}{\mu-1}}$

B

At the population.py file, I am plotting the function for mu=2.5, 3, 3.2
You can run it using python(3)



Using red, green, blue, yellow we have f(mu, x) plotted with mu=2.5, 3, 3.2, 3.5 accordingly.
At mu=3 (green) we notice that the real part is flat and 0 up until ~0.67 hence there is no “population” growth just as at the original fractal-like figure.
Which then does start to “multiply” at mu>3.

4

The function myroots(N, p) resides at the complexroots.py file, runs using python(3) and uses numpy as a dependency.

I assume that p follows the np.roots convention where p[n] is the constant and our polynomial is `p[0] * x**n + p[1] * x**(n-1) + ... + p[n-1]*x + p[n]`

Running both myroots and np.roots on 3 predefined polynomials, we get the following output:

```
For p=[1, 6, 9]
 myroots(p)=[-3.+3.7252903e-08j -3.-3.7252903e-08j]
 np.roots(p)=[-3.+3.7252903e-08j -3.-3.7252903e-08j]

For p=[5, 5, 5, 7, 8]
 myroots(p)=[ 0.41742337+1.06859401j  0.41742337-1.06859401j
-0.91742337+0.61156744j
 -0.91742337-0.61156744j]
 np.roots(p)=[ 0.41742337+1.06859401j  0.41742337-1.06859401j
-0.91742337+0.61156744j
 -0.91742337-0.61156744j]

For p=[14, 15, 16, 17]
 myroots(p)=[-1.06695555+0.j -0.00223651+1.06680815j
-0.00223651-1.06680815j]
 np.roots(p)=[-1.06695555+0.j -0.00223651+1.06680815j
-0.00223651-1.06680815j]
```