PROBLEM SET 3: DISSUSION AND ADVECTION
DUE: 3 NOV. 2020
GEO2300: FYSISKE PROSESSER I GEOFAG

SIGURD SANDVOLL SUNDBERG

1. Problem 1: Burgers Equation

1.1. a). An analytical solution to Burgers equation can look as follows

$$h = \frac{a}{2} - \frac{a}{2}\tanh\left(a\left(\frac{x-0.5at}{4\nu_t}\right)\right) \tag{1}$$

The following Python implementation using classes enables ut to call the calculate the analytical solution

```python

class Analytical():
    def __init__(self, a, v):
        # Constructor
        self._v = v
        self._a = a

    def __call__(self, x, t):
        # Function call
        a = self._a
        v = self._v
```

We can see from figure 1 and figure 2 that we are different wave behavior when we change the viscosity. For $\nu = 1$ we have waves which can be classified as smooth. Looking at the wave with $\nu = 0.1$, we see a steeper edge, almost right angled like. This would then resemble more of a wall rolling down the fjord, and less that of a wave. We can also see that the velocity of the two waves are roughly the same, with the middle of the wave being at the same point for both waves.

On this note, we can also see that the bottom parts the wave in figure 1 is further ahead of the wave than the rest, making it "faster" than the wave in figure 2. By this its meant that the bottom of the wave will be hitting areas of the shore quicker than that of the much steeper wave.

A wave with a large amplitude seen in figure 3, have a higher velocity down the fjord. This can be seen as for the last timestep, we are not able to spot the wave itself, only the top of the wave. Resulting in a shorter available period for which you are able to evacuate the coast.

1.2. b). Burgers equation it self goes as follows

$$\frac{\partial}{\partial t}h + h\frac{\partial}{\partial x}h = \nu_t\frac{\partial^2}{\partial x^2}h \tag{2}$$

We will discretize this equation and write it in finite difference form then matrix form using FTCS scheme. We then have the following equation

$$\frac{h_i^{n+1} - h_i^n}{dt} + h_j^n\frac{h_{j+1}^n - h_{j-1}^n}{2dx} = \nu_t\frac{h_{j+1}^n + h_{j-1}^n - 2h_j^n}{dx^2} \tag{3}$$

Date: November 23, 2020.

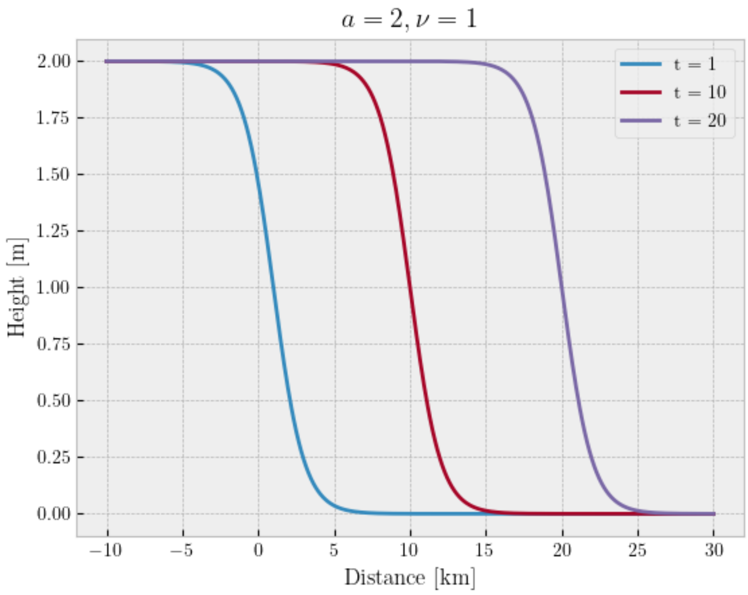

FIGURE 1. text

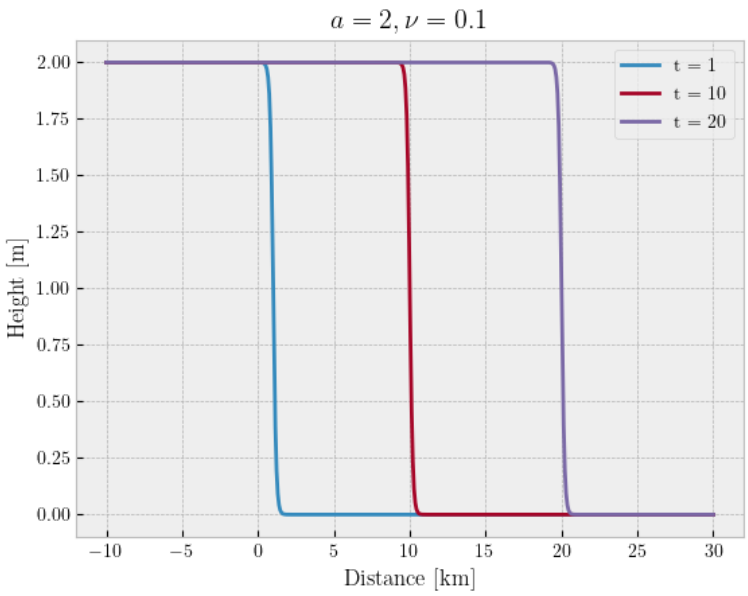

FIGURE 2. text

However calculating the term $h\frac{\partial}{\partial x}h$, the advection terms is difficult and slow. We can however approximate this by using $E = h^2/2$, using the midpoint method. This gives us a new equation on the form

$$\frac{h_j^{n+1} - h_j^n}{dt} + \frac{E_{j+1}^n - E_{j-1}^n}{2dx} = \nu_t \frac{h_{j+1}^n + h_{j-1}^n - 2h_j^n}{dx^2} \tag{4}$$

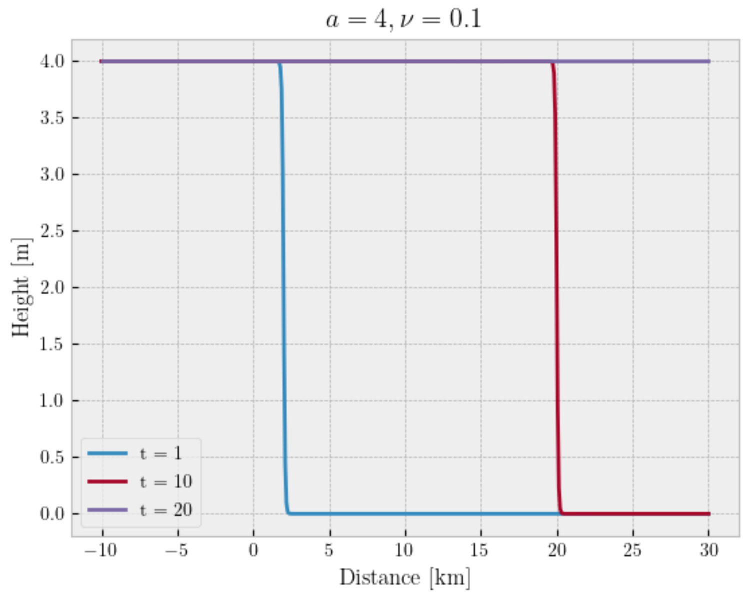

FIGURE 3. text

Re arranging this equation we have

$$h_i^{n+1} = sh_{j+1}^n + (1-2s)h_j^n + sh_{j-1}^n + c(E_{j-1}^n - E_{j+1}^n) \tag{5}$$

with $s = (dt\nu_t)/(dx^2)$ and $c = dt/(2dx)$. Our boundary conditions are given by $h(x=-10,t) = 2$ and $h(x=30,t) = 0$. Expanding our finite difference scheme for 5 grids points we have

$$\begin{aligned}
h_0 &= sh_1 + (1-2s)h_0 + sh_{-1} + cE_{-1} - cE_1 &(6)\\
&= sh_1 + (1-2s)h_0 + 2s + 2c - cE_1 &(7)\\
h_1 &= sh_2 + (1-2s)h_1 + sh_0 + cE_2 - cE_0 &(8)\\
h_2 &= sh_3 + (1-2s)h_2 + sh_1 + cE_3 - cE_1 &(9)\\
h_3 &= sh_4 + (1-2s)h_3 + sh_2 + cE_4 - cE_2 &(10)\\
h_4 &= sh_5 + (1-2s)h_4 + sh_3 + cE_5 - cE_3 &(11)\\
&= (1-2s)h_4 + sh_3 - cE_3 &(12)
\end{aligned}$$

This makes us able to create the two following matrices
(13)

$$A = \begin{bmatrix} 0 & -c & 0 & 0 & 0 \\ c & 0 & -c & 0 & 0 \\ 0 & c & 0 & -c & 0 \\ 0 & 0 & c & 0 & -c \\ 0 & 0 & 0 & c & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1-2s & s & 0 & 0 & 0 \\ s & 1-2s & s & 0 & 0 \\ 0 & s & 1-2s & s & 0 \\ 0 & 0 & s & 1-2s & s \\ 0 & 0 & 0 & s & 1-2s \end{bmatrix}$$

Our matrix vector system then looks as follows

$$\vec{h} = \mathbf{A}\vec{E} + \mathbf{B}\vec{h} + \vec{b} \tag{14}$$

where $\vec{b}$ is given by the boundary conditions. This is given by an zero vector of length n, with element $u_0 = 2s + 2c$.

1.3. **c).** The following Python implementation using Classes implements the FTCS scheme and a solver for Burgers equation.

```python
class Burgers():
    def __init__(self, s, c, n):
        self._s = s  # Problem defined variable
        self._c = c  # Problem defined variable
        self._n = n  # Dimensionality
        self._A = np.zeros([n, n])  # Matrix A
        self._B = np.zeros([n, n])  # Matrix B

    def dirichlet(self, Boundaries):
        self.bound = np.zeros(self._n)
        self.bound[0] = Boundaries[0] * s + Boundaries[0] * c
        self.bound[-1] = Boundaries[1] * s + Boundaries[1] * c

    def neumann(self):
        raise NotImplementedError

    def setup(self, Boundaries, Type):
        if Type == "D":
            self.dirichlet(Boundaries)
        elif Type == "N":
            self.neumann()
        else:
            sys.exit("No legal boundaries given")
        n = self._n
        c = self._c
        s = self._s
        for i in range(1, n - 1):
            self._A[i, i - 1] = c
            self._A[i, i + 1] = -c
            self._B[i, i - 1] = s
            self._B[i, i] = 1 - 2 * s
            self._B[i, i + 1] = s
        # Matrix boundaries
        self._A[0, 1] = -c
        self._B[0, 0] = 1 - 2 * s
        self._B[0, 1] = s
        self._A[-1, -2] = c
        self._B[-1, -1] = 1 - 2 * s
        self._B[-1, -2] = s

    def FTCS(self, dt, t_initial, t_end, U):
        self._T = t_end
        self._dt = dt
        self._u = U  # Initial function

        def Evec(): return 0.5 * (self._u * self._u)

        t = t_initial
        E = Evec()
        while t < self._T:
            t += dt
            self._u = np.dot(self._A, E) + \
                np.dot(self._B, self._u) + self.bound
            try:
                E = Evec()
            except RuntimeWarning:
                print("Overflow in E, returning to __main__")
                break

    def plot(self, x, nu=None, a=None):
```

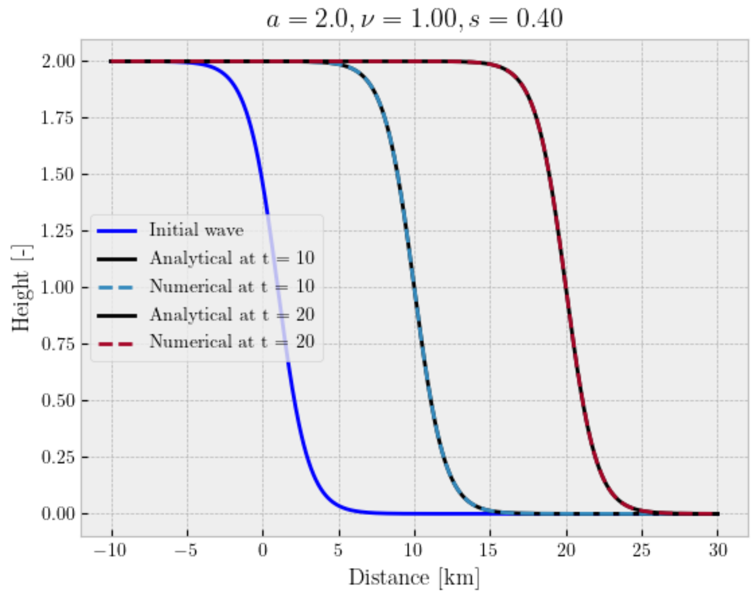

FIGURE 4. text

```
        plt.plot(x, self._u, '--', label=rf"Numerical at t = {self._T:
    d}")
        plt.title(
            rf"$a = {a:.1f}, \nu = {nu:.2f}, s = {self._s:.2f}$")
        plt.xlabel(r"Distance [km]")
        plt.ylabel(r"Height [-]")
```

Implemented in the code is overflow handling for dealing potential overflows in $\vec{E}$, where they are the most prone as this is defined as the square of $\vec{h}$

1.4. **d).** We will use the analytical solution at $h(x, t = 1)$ as the initial condition.

We see from both figure 4 and figure 5 that the numerical solution are on top of the analytical solutions, with no visible differences. Thus we can conclude that for our set of times for matrices of size $n = 750$ and $n = 1500$, for $\nu = 1$ and $\nu = 0.1$, respectively.

Looking at figure 6 we see that we have no recognizable behavior as we encountered an overflow whilst calculating $\vec{E}$. This is expected as we have chosen $s = 0.6$, which is greater than the $s < 0.5$ which is required by the FTCS scheme for it to be stable.

2. Problem 2: Chaos

2.1. **a).** Given the following equation

$$\frac{d}{dt}u + ru^2 = ru - u. \tag{15}$$

The map to this equation using $dt = 1$ looks as follows

$$u^{n+1} = ru(1 - u) \tag{16}$$

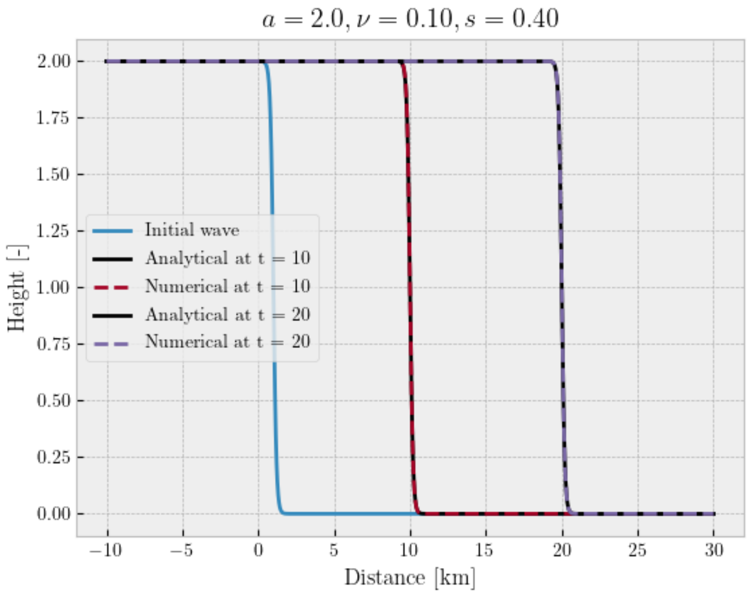

FIGURE 5. text

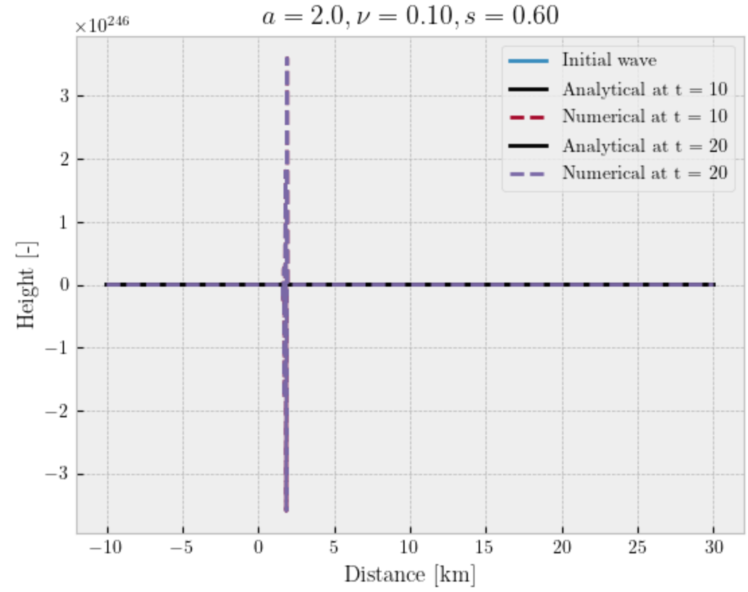

FIGURE 6. text

2.2. **b).** The steady state value(s) for this map, as a function of r can be found by setting $du/dt = 0$. which gives us $-ru^2 + ru - u = 0$. Solving this using the quadratic formula we end up with the following expression

$$u = \frac{(r-1) \pm (r-1)}{2r} \tag{17}$$

which gives two solutions.

$$u = \begin{cases} 0, \\ 1 - 1/r \end{cases} \tag{18}$$

2.3. **c).** The following Python program using Classes implements all aspects of the chaos modeling we are doing.

```python

class Chaos():
    def __init__(self, startingPoint, period, stepLength):
        """Constructor for the chaos system
        Setups our chaos systems initial variables
        Args:
            startingPoint (float): starting point for the 1d case. Can be extended to 3
            period (int): period for the simulation
            stepLength (float): step length for the simulation
        """
        # This program is currently only for a fixed time step
        stepLength = 1
        # This fixes that
        self._n = int(period / stepLength)
        self._period = period
        self._stepLength = stepLength
        self._x = np.zeros(self._n)
        self._x[0] = startingPoint
        self._u0 = startingPoint  # Potential artifact, not sure if needed
        self._timeAtWarning = self._n

    def __call__(self, r):
        """Gives the functionality of the function such that one can F(r) which returns
            wether an overflow is encountered
        Args:
            r (float): varying cases of stability
        Return:
            [boolean]: Wether an overflow is encountered
        """
        if self._x[0] != self._u0:
            self._x = np.zeros(self._n)
            self._x[0] = self._u0
        self._r = r
        for i in range(self._n - 1):
            check = self.advanceOneStep(i)
            if check:
                return True
        return False

    def advanceOneStep(self, index):
        """Advances the time step once

        Args:
            index (int): index of the array
        """
        ix = self._x[index]
        try:  # This treats cases where our equation would be numerically unstable.
            rhs = self._r * ix * (1 - ix)
        except RuntimeWarning:
            print(rf"Overflow in advanceOneStep() for r = {self._r}. Stopping calculations.")
```

```python
                self._timeAtWarning = index
                return True
            self._x[index + 1] = rhs
            return False

    def _root(self, coefficient=None):
        if coefficient is None:
            coefficient = self._r
        if coefficient == 0:
            return (coefficient, coefficient)
        else:
            root1 = (1 - coefficient) / (2 * coefficient) + ((np.sqrt
((1 - coefficient) * (1 - coefficient))) / (2 * coefficient))
            root2 = (1 - coefficient) / (2 * coefficient) - ((np.sqrt
((1 - coefficient) * (1 - coefficient))) / (2 * coefficient))
            return (root1, root2)

    def plot(self):
        roots = self._root()
        time = np.linspace(0, self._period, self._n)
        if self._timeAtWarning != self._n:
            plt.plot(time[:self._timeAtWarning], self._x[:self.
_timeAtWarning], color='k', linestyle='-', markersize=1, marker='o
', label=rf"Overflow at iteration: {self._timeAtWarning:d}, for r
= {self._r:.1f}", linewidth=.7)

        else:
            plt.plot(time, self._x, color='k', linestyle='-',
markersize=1, marker='o', label=rf"r = {self._r:.1f}", linewidth
=.7)

        # Plot the roots
        if (roots[0] == roots[1]):
            plt.plot([0, self._timeAtWarning], [roots[0], roots[0]], '
--', label=rf"$\lambda_u$ = {roots[0]:.4f}")
        else:
            plt.plot([0, self._timeAtWarning], [roots[0], roots[0]], '
--', label=rf"$\lambda^1_u$ = {roots[0]:.4f}")
            plt.plot([0, self._timeAtWarning], [roots[1], roots[1]], '
--', label=rf"$\lambda^2_u$ = {roots[1]:.4f}")

    def getArray(self):
        return self._x

    def getRoots(self, r=None):
```

This also has inbuilt error handling for dealing with overflows.

2.4. **d).** Firstly we are interested in when the system is at a steady state. A steady state is defined as $u^n = ru^n(1-u^n)$, equivalent to $du/dt = 0$. This follows a similar pattern as Markov Chains, as once we reach a steady state, we are not leaving the steady state, the system stay in the steady state.

Trying to solve this problem has been annoying as there are no visible oscillation until you reach $r < 3$, however at zoom levels of around 10^{-20}, there are visible oscillations prior to that, of amplitude equal to machine zero.

Last solution that actually converges to a root is for r = 1. Seen in figure 7 I assume that is what is asked for in this problem. However as stated, no visible oscillations where actually found until $r \geq 3$. Zero is the root in all cases. The range can be written either as $r \in [0, 1)$ or $r \in [0, 3)$.

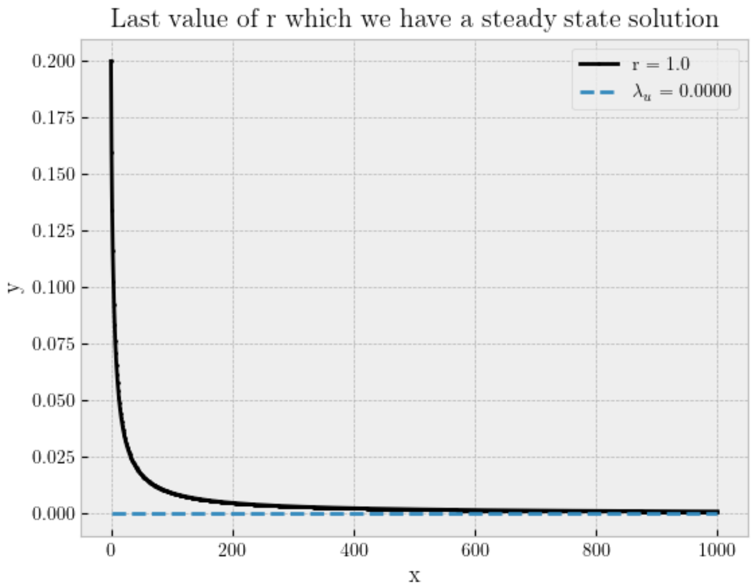

FIGURE 7. Last convergance to a root.

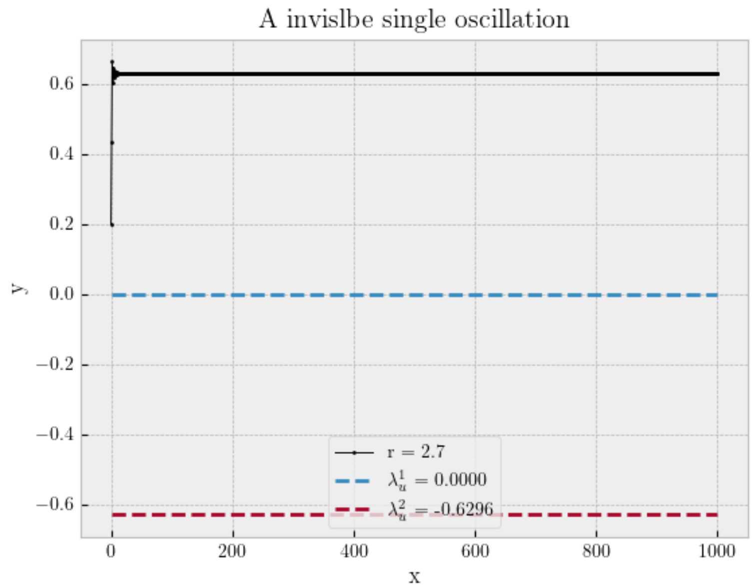

FIGURE 8. Invisible single oscillation.

2.5. **e).** Same follows for this, if we assume that we have oscillations that are invisible to the naked eye, unless heavily zoomed, the range is $r \in [1, 3.5)$ or if we only care about visible oscillations we have $r \in [3, 3.5)$.

Both figure 8 and figure 9 have single oscillations in them, however in figure 8 it is so small, that it is just above machine zero.

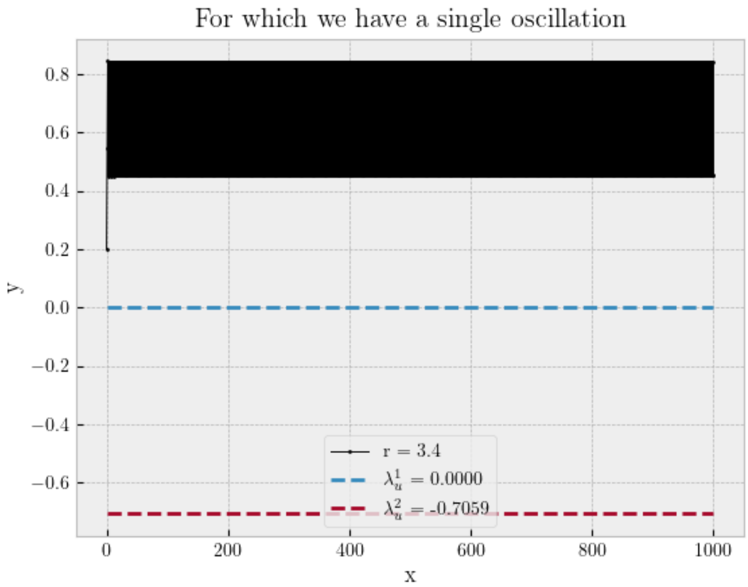

FIGURE 9. Naked eye visible single oscillations.

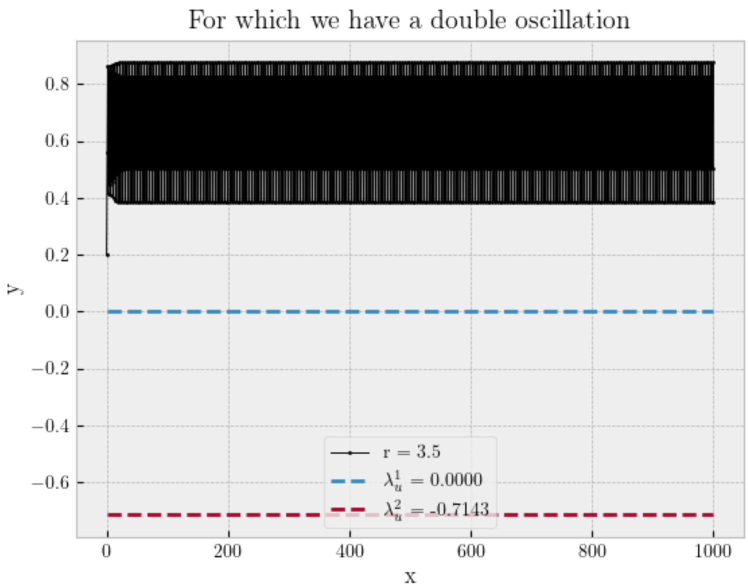

FIGURE 10. Naked eye visible double oscillations.

2.6. **f).** For a double oscillation see figure 10.

2.7. **g).** The found range for which we have chaos is $r \in [3.6, 4.1)$. The min and max values of u vary, but for $r = 4.0$, they are respectively, 0 and 1. A plot of the chaos is seen in figure 11.

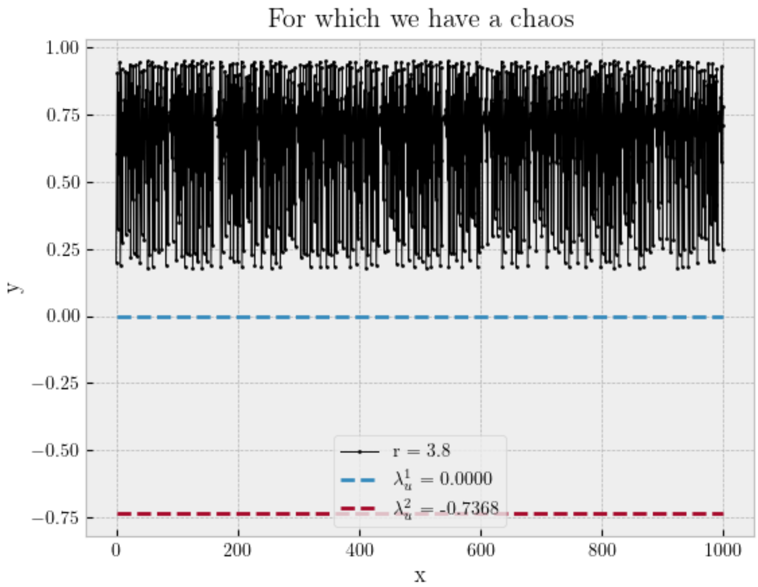

FIGURE 11. Chaos plot.

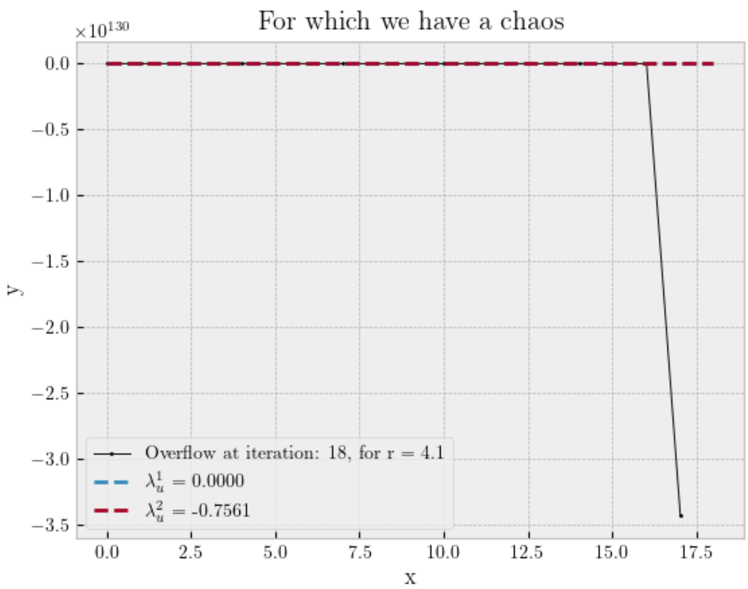

FIGURE 12. Numerically unstable.

2.8. **h).** For $r \in [4.1, \infty)$ we have an numerically unstable mapping. An example is in figure 12.