Research track report 1

Lucas Perrier

30 September 2024

1 Preview

Our goal is to see the extent to which encouraging an LSTM neural network to satisfy the equations of a physical model increases its capacity to learn from observed physical data. We study this approach by considering turbulence in tokamak plasmas. The physical model we will use to understand turbulence in tokamak plasmas can be seen as a predator prey model. We first want to test this approach on data generated from simulations of a similar, but simpler, physical model. In this report, I take a simple predator prey model, the Lotka-Volterra equations, and seek to (i) understand its dynamics, (ii) simulate it. Given a set of parameters and inital conditions, I use the Runge Kutta 4 method to simulate the system of equations. I perform convergence tests for the Runge Kutta 4 method, I explore the role of parameters and initial conditions in the system and study the dynamics of the system with both time and phase plots.

2 Outline

- Lotka Volterra equations
- Runge Kutta 4 method
  - Runge Kutta 4 code
- Simple Example baseline
- Convergence test
- Role of each parameter
  - Role of α
  - Role of β
  - Role of
  - Role of δ
- Phase plot
  - Equilibrium point

3 Lotka Volterra equations

The Lotka Volterra equations are as follows,

$$\frac{dx}{dt} = \alpha x - \beta xy \tag{1}$$

$$\frac{dy}{dt} = -\gamma y + \delta xy \tag{2}$$

As a predator prey model, the elements of the equations represent the following,

- x is the prey population amount $[prey]$
- α is the prey population reproduction frequency $[\frac{1}{t}]$
- β is the prey population mortality rate $[\frac{1}{t \times prey}]$
- y is the predator population amount $[predator]$
- γ is the predator population decay frequency $[\frac{1}{t}]$
- δ is the predator population growth rate $[\frac{1}{t \times predator}]$

The Lotka Volterra equations are a set of two non linear, first order differential equations.

4 Runge Kutta 4 method

Since the Lotka Volterra equations are a set of two non linear, first order differential equations, I use the suitable Runge Kutta 4 method to simulate the system of equations. The Runge Kutta 4 is a numerical method that seeks to approximate the solution to a first order differential equation of the form,

$$\frac{da(t)}{dt} = f(a(t), t) \tag{3}$$

with initial conditions,

$$a(t_0) = a_0 \tag{4}$$

Given a value of a at time $t = t_0$, the method approximates the value of a at time $t_0 + h$, where h is a small amount of time ($h = 0.01$ for example). The method approximates the value of $a(t_0 + h)$ by using four values for the derivative of a with respect to t at time $t = t_0$. After computing $a(t_0 + h)$, the method iterates to find the value of a at the next time point, iterating until the desired length of time has been simulated. Here is the method expressed mathematically:

$$a(t_0 + h) = a(t_0) + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} h \tag{5}$$

$$k_1 = f(a(t_0), t_0) \tag{6}$$

$$k_2 = f(a(t_0) + k_1\frac{h}{2}, t_0 + \frac{h}{2}) \tag{7}$$

$$k_3 = f(a(t_0) + k_2\frac{h}{2}, t_0 + \frac{h}{2}) \tag{8}$$

$$k_4 = f(a(t_0) + k_3 h, t_0 + h) \tag{9}$$

Applied to the Lotka-Volterra system, $a(t)$ represents the vector $(x(t), y(t))$, and f is a set of functions $f_1(x) = \alpha x - \beta yx$, and $f_2(y) = -\gamma y + \delta xy$ for x and y respectively. Figure 1 illustrates the use of slope approximations to determine the value of $a(t_0 + h)$ given $a(t_0)$. I perform the simulation using python.

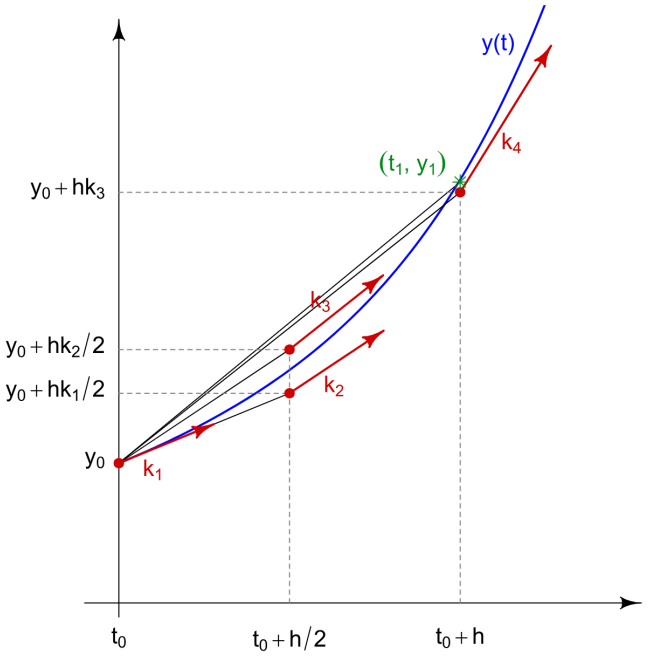


Figure 1: Slopes used by the classical Runge Kutta method

4.1 Runge Kutta 4 code

My implementation of the code is available at `https://github.com/lucasperrier/research_track/tree/main/Lotka_Volterra_RK4` . The code snippets in figures 2,3,4 illustrate a typical workflow for running a simulation.

```python
def rk4_step(model, dt, t0, y0, parameters_list):
    '''
    performs one runge kutta 4 step
    IN -
    model: model in vectorized format that you want to integrate, python function
    dt: timestep, float
    t0: value of t at current step, float, shape = (1)
    y0: value of y at current step ( x(t),y(t) ), float, shape = (1,2)
    parameters_list: list of values taken up by model [alpha, beta, gamma, delta]
    OUT -
    yout: value of y at next step ( x(t+1), y(t+1) ), float, shape = (1,2)
    '''
    f1 = model(t0,y0, parameters_list)
    f2 = model(t0+(dt/2), y0+(dt/2)*f1, parameters_list)
    f3 = model(t0+(dt/2), y0+(dt/2)*f2, parameters_list)
    f4 = model(t0+dt, y0+dt*f3, parameters_list)
    yout = y0 + (dt/6) * (f1+ 2*f2 + 2*f3 +f4)
    return yout
```

Figure 2: Python function performing one step of the Runge Kutta 4 method

```python
def lotka_volterra_system(t,y, parameters_list):
    alpha, beta, delta, gamma = parameters_list
    x,y = y
    '''
    computes dx/dt and dy/dt in lotka volterra model
    IN -
    y: x and y value, np.array, shape = (1,2)
    parameters_list: list of values taken up by model [alpha, beta, gamma, delta]
    OUT -
    value of dy/dt, dx/dt, np.array, shape = (1,2)
    '''
    dxdt = alpha * x - beta * x * y
    dydt = delta * x * y - gamma * y
    return np.array([dxdt, dydt])
```

Figure 3: Python function defining Lotka Volterra system in vectorized format, ready for use in rk4 step python function from figure 2

```python
T = boundary
# initial conditions
y0 = initial_conditions
# discrete time vector
number_of_timepoints = int(T/dt)
# Generate the time array with N+1 points to include the initial time
t = np.linspace(0, T, number_of_timepoints+1)
# initialize solution matrix (time evolution of x and y)
Y = np.zeros((2,number_of_timepoints+1))
Y[:,0] = y0
# use initial conditions as starting point in loop
yin = y0

# simulation
for i in tqdm(range(number_of_timepoints), desc="Simulation Progress"):
    yout = method(system,dt,t[i],yin, parameters_list)
    Y[:, i+1] = yout
    yin = yout
# plot solution found
#plotting_function(Y,t,parameters_list)
return Y, t
```

Figure 4: Code for running a simulation using python functions in figures 2,3

5 Simple Example baseline

I first test the coherence of the simulation visually by comparing the implementation's time series of $x(t)$ and $y(t)$ against the time series presented in this wikipedia article's 'A simple example' section [1]. The parameters take the values $\alpha = 1.1$, $\beta = 0.4$, $\gamma = 0.4$ and $\delta = 0.1$; the initial conditions are $x(0) = 10$ and $y(0) = 10$. The simulation is run until time $T = 100$ and time steps are set at $h = 0.001$.

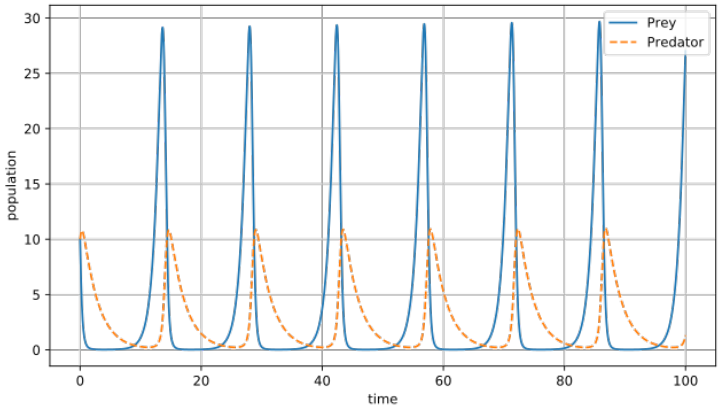


Figure 5: Simple Example time series snapshot from Wikipedia article [2]

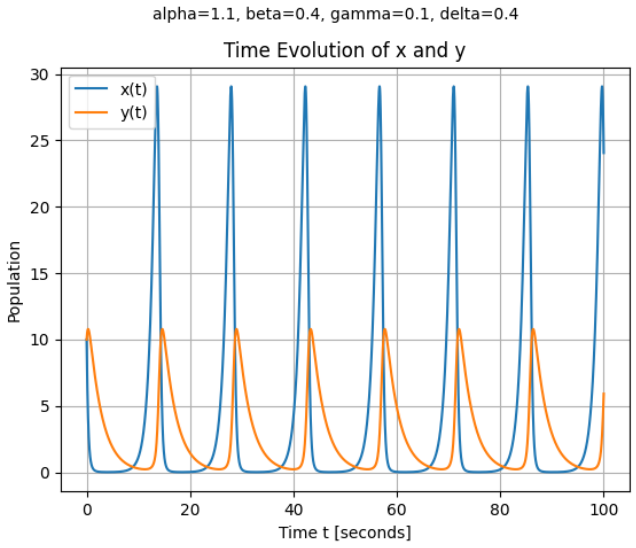


Figure 6: Simple Example time series obtained from Runge Kutta 4 implementation

The plots show the implementation works for this scenario.

6 Convergence tests

I perform convergence tests to determine whether the implementation is stable. Stability indicates that the order of convergence of the implementation matches the theoretical order of convergence. Once I establish the implementation as stable, I consider the implementation as reliable for generating data approximating the system's true solution, with the step size h chosen determining the amount of error in the simulation. The convergence test consists of seeing how reducing the step size impacts the error of a simulation. Specifically, I perform simulations with varying step sizes and compute the total error (10) of each simulation with respect to a 'true' solution u_{ref}. I use the obtained step size, error relationship to estimate the order of convergence (11) and compare the found order of convergence to a known, theoretical order of convergence. I decrease the step size as $h_1 = 0.1$, $h_2 = \frac{h_1}{2}$, $h_3 = \frac{h_2}{2}$, and so on. I calculate the order of convergence (11) by plotting $\log(E(h))$ vs $\log(h)$, fitting a line of best fit through the points of the 'log log' plot, and taking the slope of the line of best fit as the value of p. The slope is the value of the order of convergence because,

$$
\begin{aligned}
E(h) &\propto h^p \\
\Rightarrow E(h) &= C \times h^p \\
\Rightarrow \log(E(h)) &= p \times \log(h) + \log(C)
\end{aligned}
$$

The obtained value $p = 4.01$ agrees with the theoretical value of $p = 4$ for the Runge Kutta 4 method. That means the implementation is stable and I can use it to generate reliable data. I use a step size $h = 0.0001$ and time boundary $T = 50$ for the reference simulation to compute the error $E(h)$. Figure 7 shows the results of the convergence test.

$$
E(h) = \sqrt{(u_h^{(1)} - u_{\text{ref}}^{(1)})^2 + (u_h^{(2)} - u_{\text{ref}}^{(2)})^2 + \cdots + (u_h^{(n)} - u_{\text{ref}}^{(n)})^2} \tag{10}
$$

$$
p = \frac{\log\left(\frac{E(h_i)}{E(h_{i+1})}\right)}{\log\left(\frac{h_i}{h_{i+1}}\right)} \tag{11}
$$

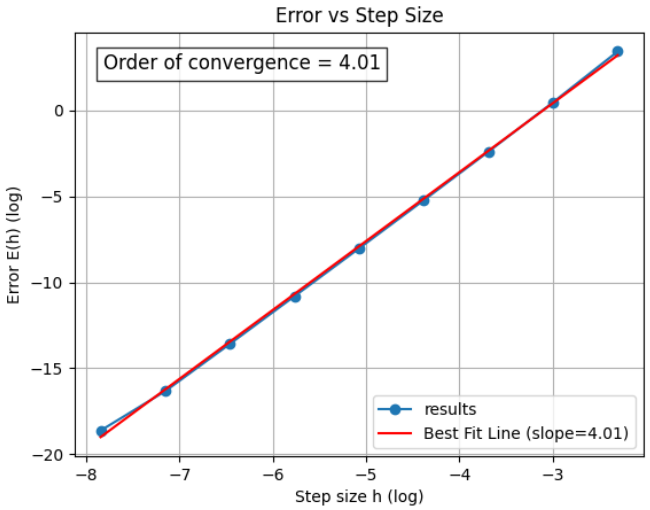


Figure 7: Obtained $\log(E(h))$ vs $\log(h)$ plot and order of convergence, showing the implementation is stable

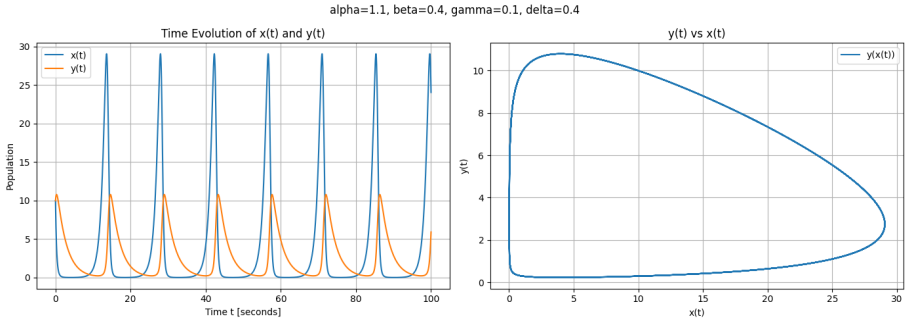
7 Role of each parameter

Varying each parameter individually and observing the impact on the time evolution of x and y clearly illustrates the interplay between x and y. The plots

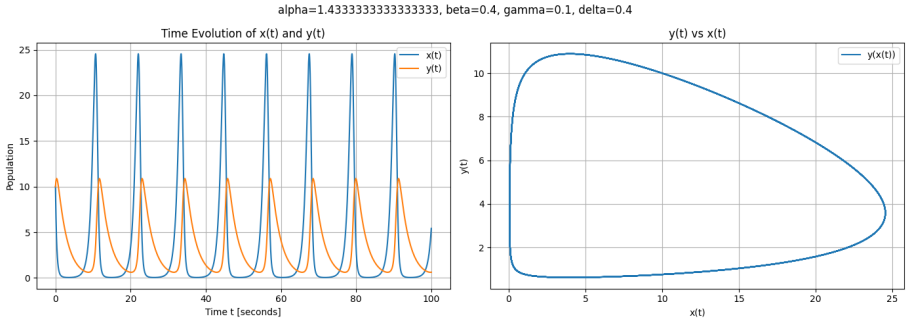
show that increasing one quantity increases the other, which in turn decreases the first quantity. The relation between the two quantities results in periodic behavior, with varying degrees of sensitivity for how one quantity impacts the other.

7.1 Role of α

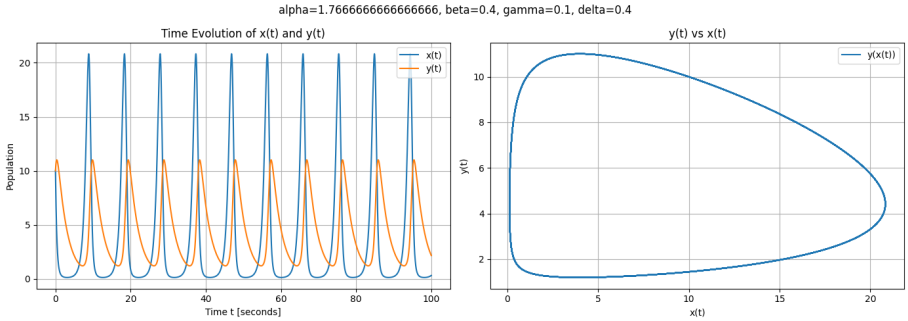
As α increases, the prey $x(t)$ grow at a faster rate, which can be seen by the higher frequency of a predator prey cycle. However, the increase in $x(t)$ leads to a larger positive rate of change of $y(t)$, which explains why increasing α reduces the maximum value of $x(t)$ reached. Although the prey grow at a faster rate, they actually end up with a smaller population for each cycle because the large increase in the prey population causes the predators population to increase as well.



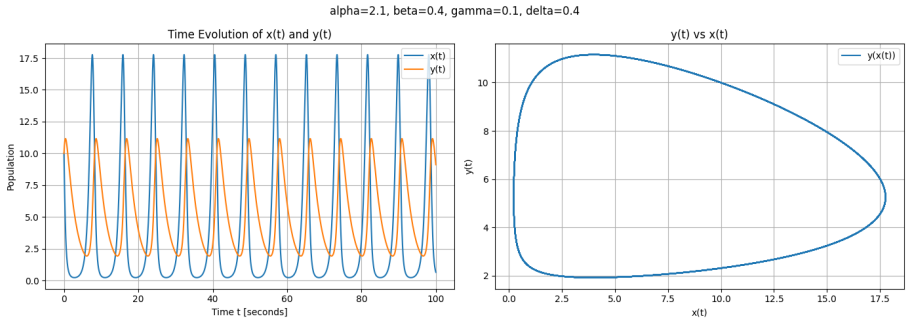
(a) $\alpha = 1.1$



(b) $\alpha = 1.4$



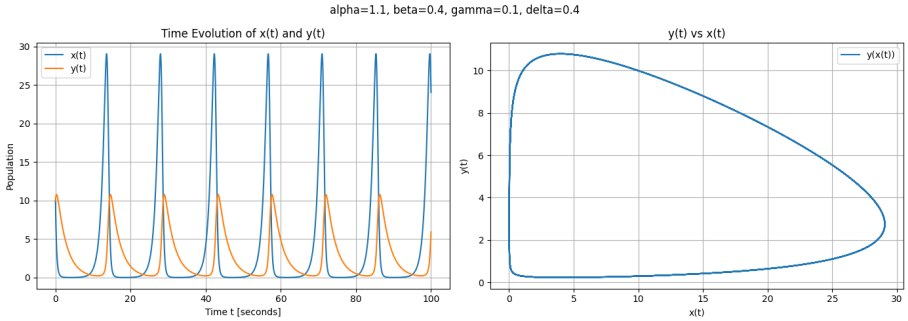
(c) $\alpha = 1.76$



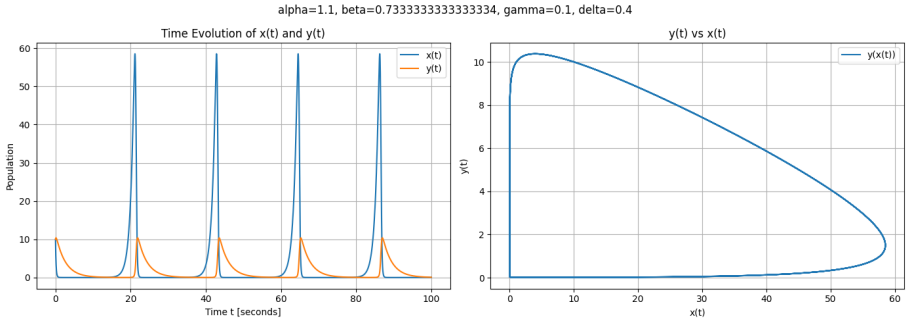
(d) $\alpha = 2.1$

7.2 Role of β

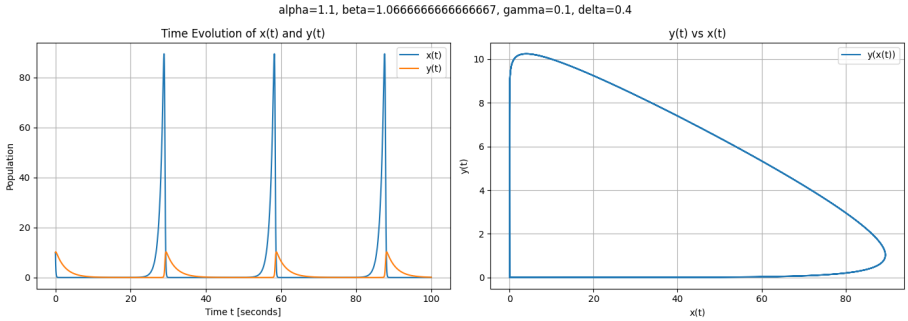
As β increases, the predators $y(t)$ have a bigger negative impact on the growth rate of the prey $x(t)$, which can be seen by the faster decrease in $x(t)$ when $y(t)$ grows. However, the faster decrease in $x(t)$ causes $y(t)$ to then decrease significantly in such a way that $x(t)$ reaches a bigger maximum value. Essentially, increasing β results in the predators y over consuming the prey x, leading to slower regeneration of each population or, in other words, a longer predator-prey cycle period.



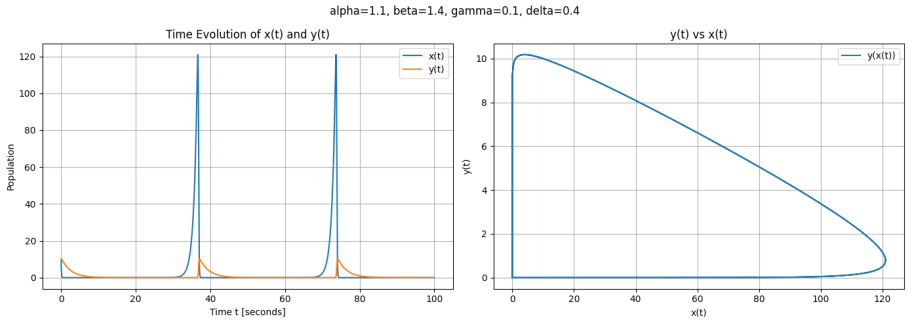
(a) $\beta = 0.4$



(b) $\beta = 0.73$



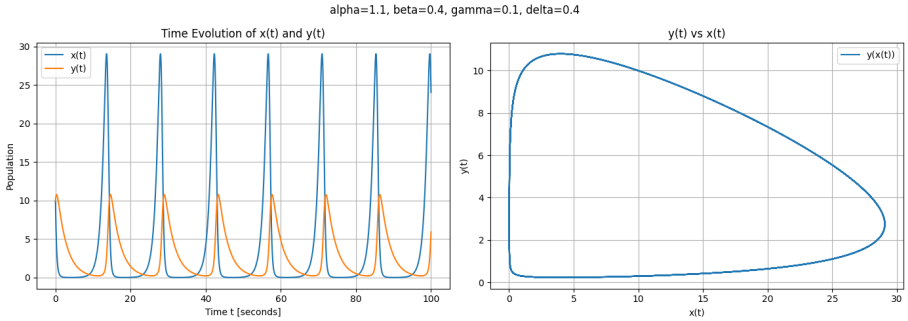
(c) $\beta = 1.06$



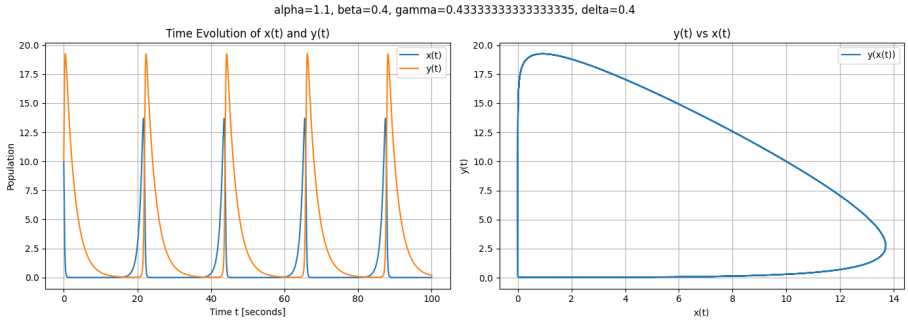
(d) $\beta = 1.4$

7.3 Role of γ

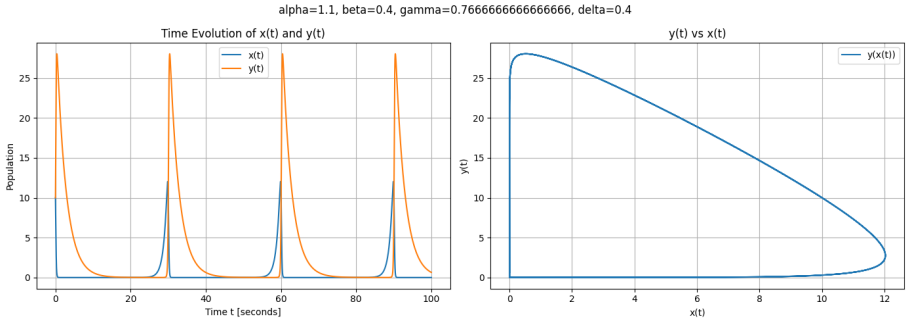
As γ increases, the predator population $y(t)$ decays faster, leading to quicker predator declines. Initially, however, the the predator $y(t)$ population grows before the sharp decline, meaning the prey $x(t)$ population doesn't necessarily recover as much as with a lower γ. This results in shorter predator-prey cycles with higher frequencies.



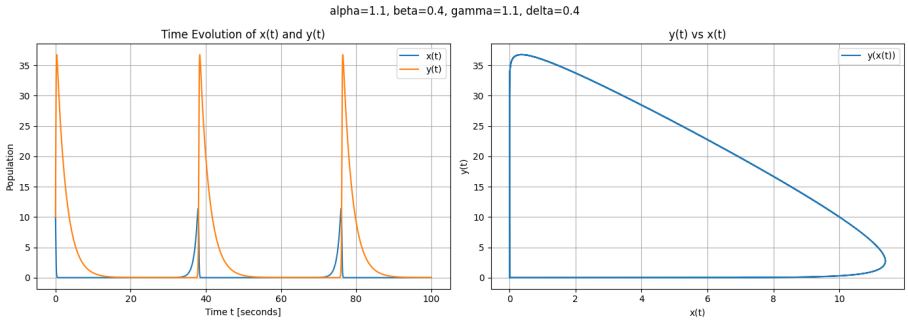
(a) $\gamma = 0.1$



(b) $\gamma = 0.43$



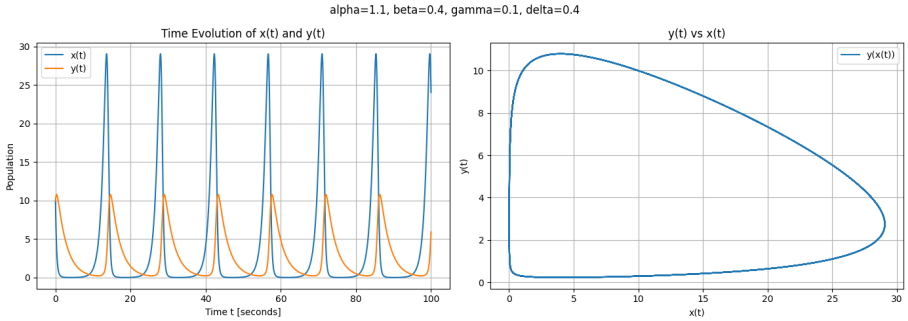
(c) $\gamma = 0.76$



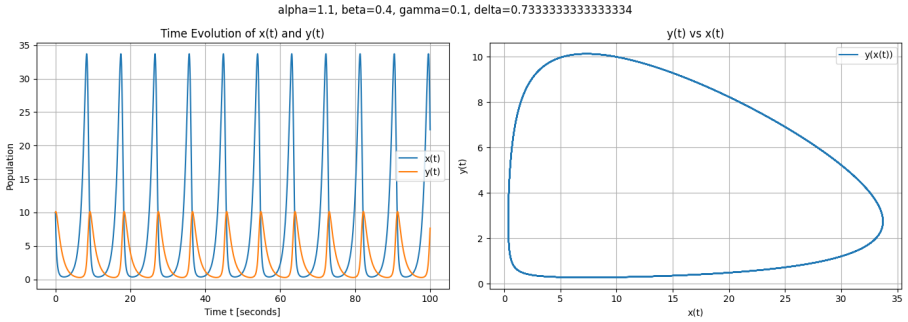
(d) $\gamma = 1.1$

7.3.1 Role of δ

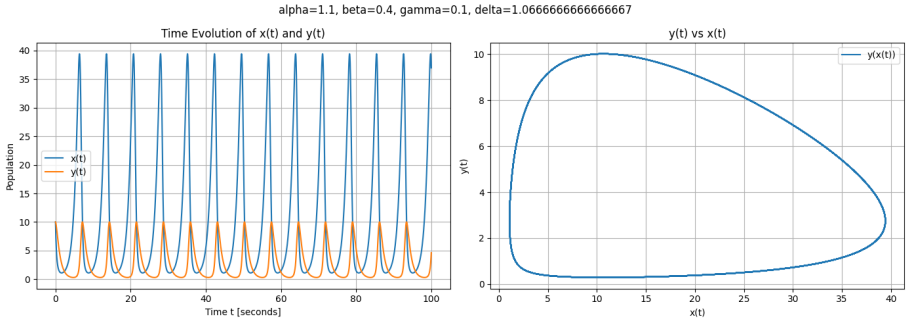
When γ increases, the predator-prey cycles become more frequent as predators $y(t)$ grow more efficiently from the available prey $x(t)$. This rapid predator growth leads to faster consumption of prey, but once predators peak and begin to decline, the prey have more time to recover, resulting in a higher maximum prey population. Additionally, the prey population doesn't drop as low because the predator population also declines quickly after its spike, allowing prey to survive and repopulate sooner. As a result, both the prey population's maximum and minimum values increase with a higher γ. Overall, the predator-prey dynamics become faster, leading to shorter and more intense cycles.



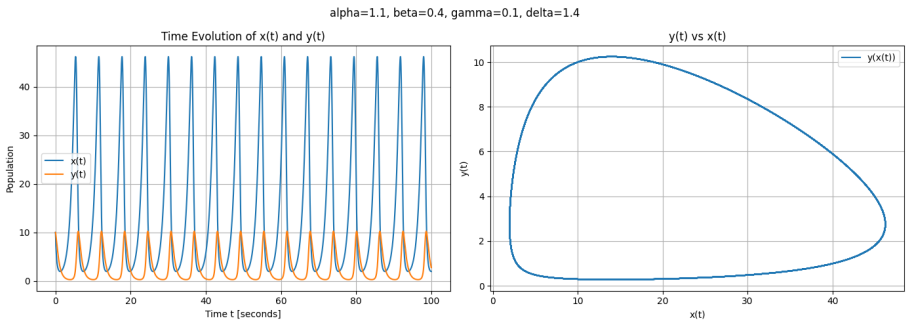
(a) $\delta = 0.5$



(b) $\delta = 0.70$



(c) $\delta = 1.06$



(d) $\delta = 1.4$

8 Phase-space plot

To visualize the dynamics of the system, I plot $y(t)$ against $x(t)$ with t as a hidden parameter. Such a plot shows many important aspects of dynamical systems, such as stability, periodicity, nature of equilibrium points. One can also simplify a complex system of equations by considering only two variables at a time. In the case of the Lotka Volterra system, I vary the initial predator population $y(t = 0)$ and observe the effect on the y and x relationship. Increasing values of $y(t = 0)$ results in more aggressive dynamics in the sense that changing x has a bigger impact on y. However, the dynamics remain periodic, shown by the 'circular' curves.

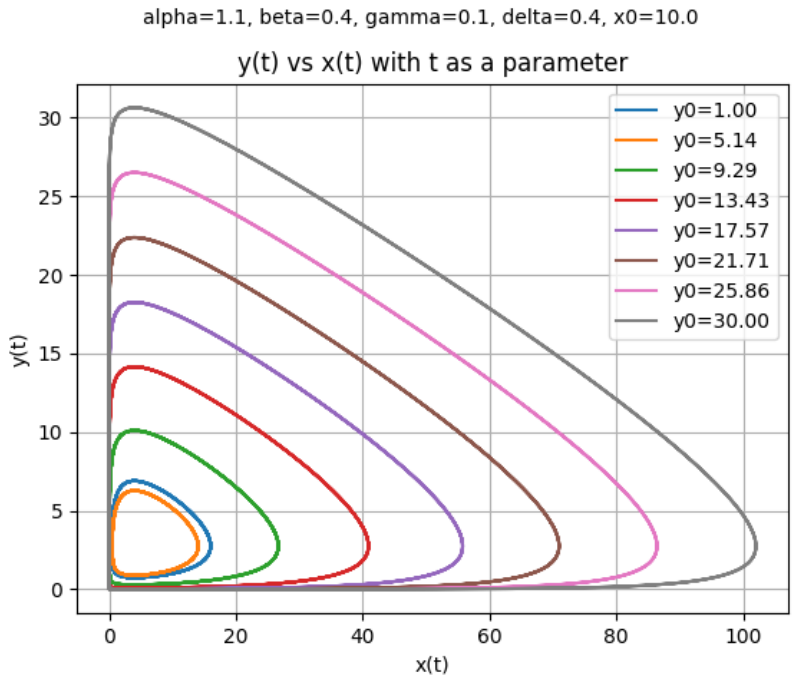


Figure 12: Phase plot of $y(t)$ vs $x(t)$ with varying initial condition $y(t = 0)$

8.1 Equilibirum point

There exists an equilibirum point for the set of parameters $\alpha = 1.1$, $\beta = 0.4$, $\gamma = 0.1$ and $\delta = 0.4$, where the curves focus around. The equilibrium point designates the combination $(x, y) = (x*, y*)$ which the system will gravitate towards. More specifically, the equilibrium point are the values of (x, y) for which $\frac{dx}{dt}$ and $\frac{dy}{dt}$ are both equal to zero. The equilibrium point is $(x*, y*) = (\frac{\alpha}{\beta}, \frac{\gamma}{\delta})$. Indeed, plotting the time evolution of the populations with the initial conditions set to the equilibrium point, ie. $x(t = 0) = \frac{\alpha}{\beta}$ and $y(t = 0) = \frac{\gamma}{\delta}$,

results in constant population levels through time. No curves appear on the phase plot since x and y are both constant.

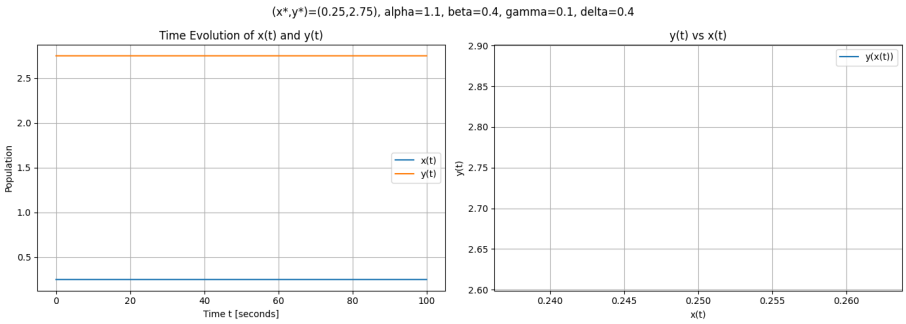


Figure 13: Time evolution (left) and phase plot (right) for equilibirum point $(x*, y*)$ set as the initial condition. The prey x and predator y populations are both constant through time.

References

[1] Wikipedia authors. Lotka–volterra equations. 2024.

[2] Krishnavedala Ian Alexander. Lotka-volterra model (1.1, 0.4, 0.4, 0.1) time series, 2024. Figure from 'Lotka-Volterra equations' Wikipedia article.