1 W4 Lab-2

Risto Rushford, 05/16/2020
SYSC 535

1.1 Front Matter Code

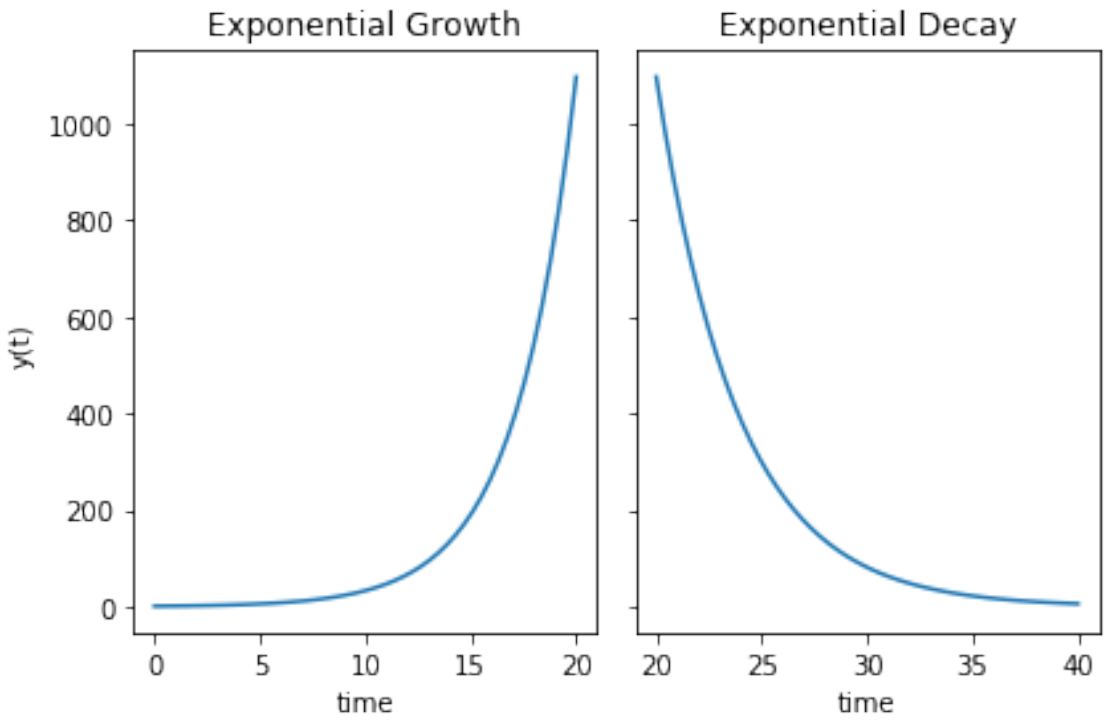
[101]:
```
import matplotlib.pyplot as plt
from scipy.integrate import odeint
import numpy as np
from numpy import *
import pylab as pl
```

1.1.1 L1.1 - Exponential Growth and Decay

[102]:
```
# First we'll define a function representing the
# ODE dy/dt = k * y(t)
def exp_growth(y, t, k):
    dydt = k * y
    return dydt

# initial condition
y0 = 1
k_plus = float(input('Enter a positive k-value with |k| < 1'))
k_neg = float(input('Enter a negative k-value with |k| < 1'))

# time points
t1 = np.linspace(0,20)

# Now to solve the ODE
y1 = odeint(exp_growth,y0,t1, args=(k_plus,))

# let's start the exponential decay
# where the growth function left off
y_n = y1[-1]
t2 = np.linspace(20,40)
y2 = odeint(exp_growth,y_n,t2, args=(k_neg,))

# Now lets plot the results
fig, axs = plt.subplots(1,2, tight_layout=True, sharey='all')
axs[0].plot(t1,y1)
axs[0].set_title('Exponential Growth')
axs[0].set_ylabel('y(t)')
axs[0].set_xlabel('time')

axs[1].plot(t2,y2)
axs[1].set_title('Exponential Decay')
```

```
axs[1].set_xlabel('time')
plt.show()
```

```
Enter a positive k-value with |k| < 1 .35
Enter a negative k-value with |k| < 1 -.26
```



I wrote multiple versions of this exercise, first using the Sayama method but I wanted to make sure that I knew how to use the odeint() function so I went back through. I also had some fun setting it up to produce two graphs side-by-side. You'll notice that the graph on the right applies the exponential decay function at precisely the value that the exponential growth function finishes on.

This was a fun exercise because I was able to quickly learn how to apply the code and demonstrate to myself that I could complete one of these exercises fairly easily and have fun doing it.

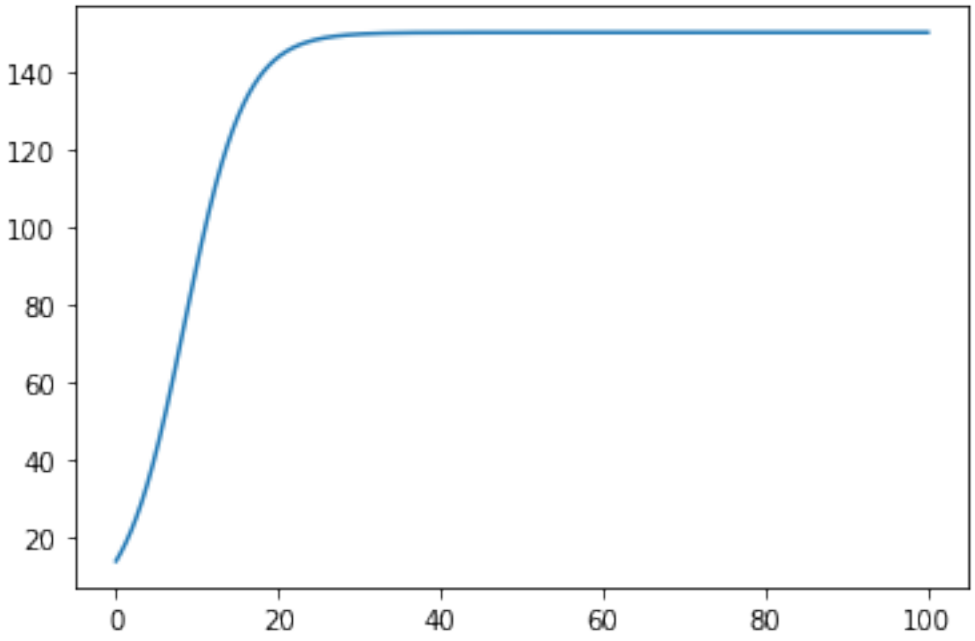
1.1.2 L1.2 Logistic Growth (ODE)

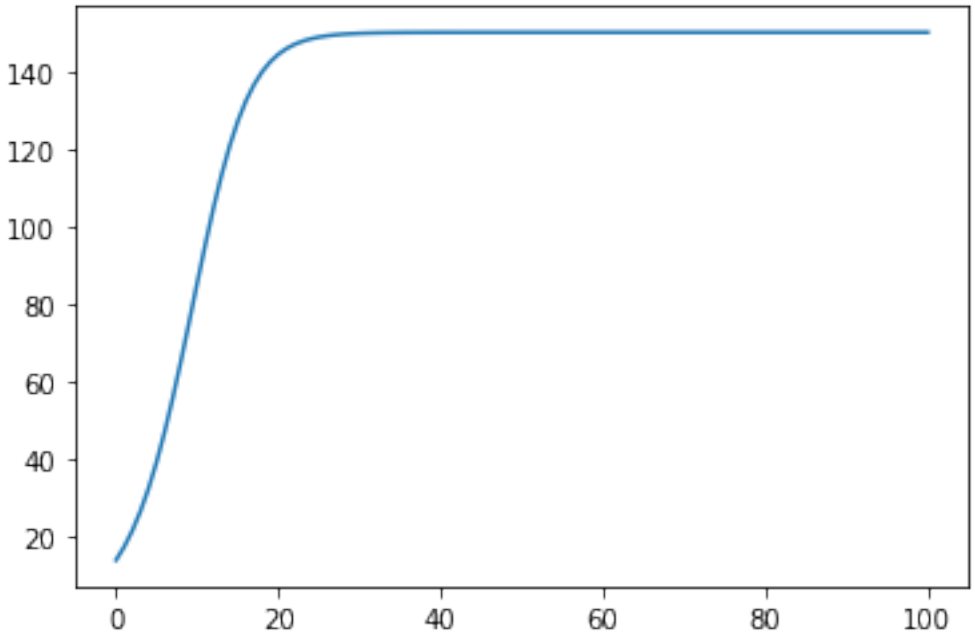
[103]:
```
# In the above example we used ODEint from the
# scipy package. Now let's try Sayama's method

P0 = float(input('What is the initial population? '))
r = float(input('Enter a rate of growth, r, with |r| < 1: '))
M = float(input('Enter a carrying capacity: '))
Dt = 0.01
```

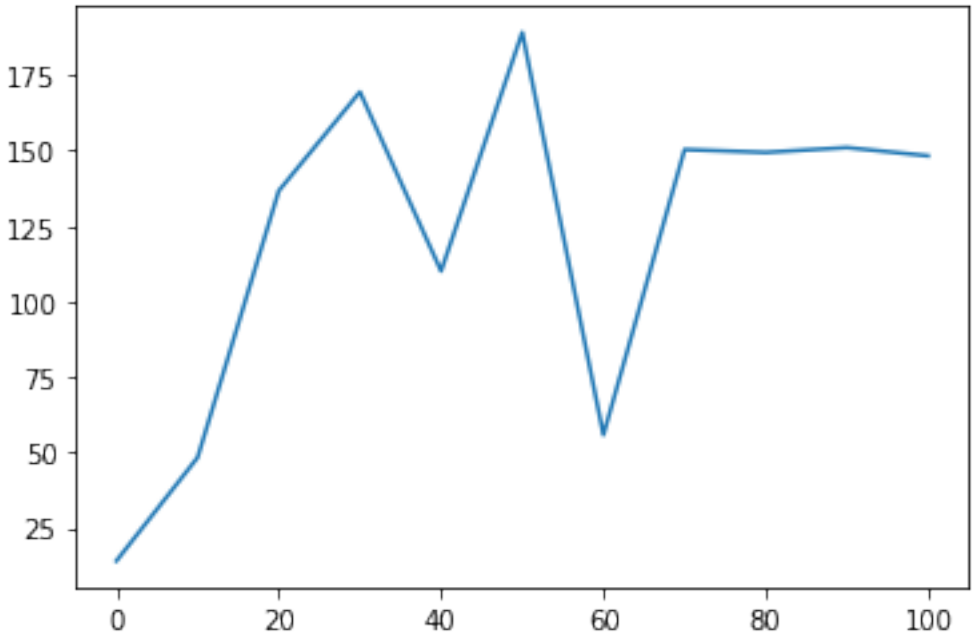
```
def initialize():
    global P, result, t, timesteps
    P = P0
    result = [P]
    t = 0.
    timesteps = [t]

def observe():
    global P, result, t, timesteps
    result.append(P)
    timesteps.append(t)

def update():
    global P, result, t, timesteps
    P = P + r * P * (1 - P / M) * Dt
    t = t + Dt

initialize()
while t < 100.:
    update()
    observe()

pl.plot(timesteps, result)
pl.show()

# Now let's explore a larger time step
Dt = 1
initialize()
while t < 100.:
    update()
    observe()

pl.plot(timesteps, result)
pl.show()

# And a yet larger time step
Dt = 10
initialize()
while t < 100.:
    update()
    observe()

pl.plot(timesteps, result)
pl.show()
```

```
What is the initial population?  14
```

```
Enter a rate of growth, r, with |r| < 1:  .27
Enter a carrying capacity:  150
```







I went ahead and used Sayama's example of the foward Euler method so that I could explore the effects of the different timesteps. I've experienced issues with timesteps when I took SySc 514 System Dynamics but I didn't have a strong appreciation for the cause of the issues. In taking this class, and Math 256 Differential Equations, I am now much more appreciative of the usefulness and limitations of Euler's method of integration.

Above we can see the effect of different timesteps used to produce the same plot. The user is prompted to input the initial parameters, but then the three timesteps DT = 0.01,1,10 are hard-coded. We don't see an obvious difference between the first and second graphs. But as the timesteps grow above 1 we quickly see that the graph becomes very jagged as the algorithm continuously oversteps the limit.

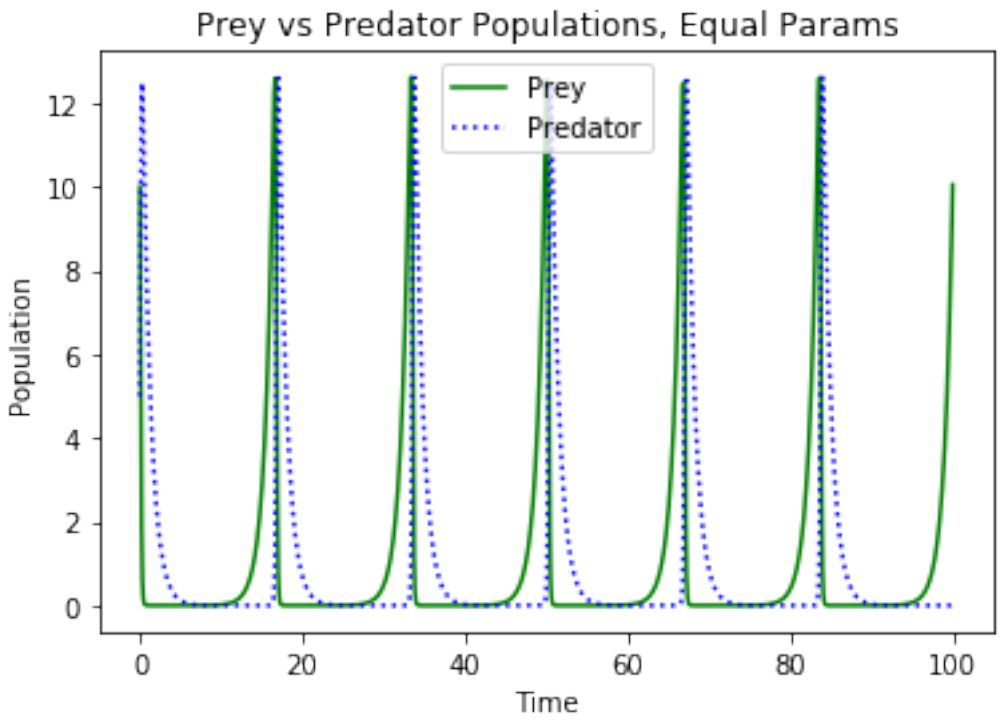
1.1.3 L1.4 Predator Prey

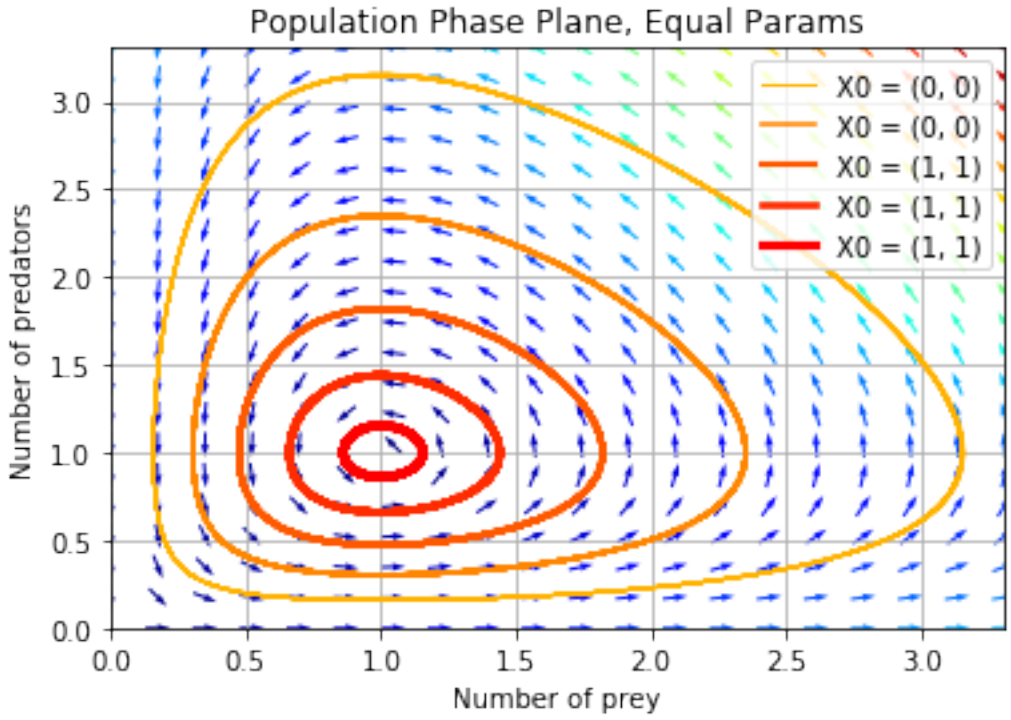
I found a tutorial at https://scipy-cookbook.readthedocs.io/items/LoktaVolterraTutorial.html which the below code is based on

[104]:
```
''' First we'll define a function representing the
ODE du/dt = au - buv and dv/dt = -cv + dbuv '''

a = 1   # Natural growth rate of prey minus predator
b = 1   # Natural dying rate of prey with predation
c = 1   # Natural dying rate of predator minus prey
d = 1   # Amount of prey consumption needed to create new predators

# Let both populations be contained in X=[u,v], then
def dX_dt(X, t=0):
```

```
    return array([ X[0]*(a - b*X[1]),
                   X[1]*(d*b*X[0] - c) ])

''' Now, as per the tutorial, we'll set the
equilibrium points, occuring when the conditional
statements are true '''
X_f0 = array([0., 0.])
X_f1 = array([c/(d*b), a/b])
all(dX_dt(X_f0) == zeros(2)) and all(dX_dt(X_f1) == zeros(2))

# Now we'll define a Jacobian matrix
def d2X_dt2(X, t=0):
    return array([[a - b*X[1], -b*X[0]],
                  [b*d*X[1], b*d*X[0] - c]])

# Near X_f0 is the extinction point of both species
A_f0 = d2X_dt2(X_f0)
A_f1 = d2X_dt2(X_f1)

''' Both population functions are periodic, so we'll
determine the eigenvalues, which are imaginary to
then determine the oscillation patterns '''
lambda1, lambda2 = linalg.eigvals(A_f1)
T_f1 = 2 * pi / abs(lambda1)   # The period of oscillation

# Finally, we can integrate:
t = linspace(0, 100, 1000)   # Time steps and duration
X0 = array([10,5])            # initial populations
X = odeint(dX_dt, X0, t)


# And to plot the results:
prey, predator = X.T
f1 = plt.figure()
plt.plot(t, prey, 'g', label='Prey')
plt.plot(t, predator, 'b:', label='Predator')
plt.legend(loc='best')
plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Prey vs Predator Populations, Equal Params')
f1.savefig('predator_prey_1a.png')

# Now we'll plot the population trajectories in a phase plane:
posit_X0 = linspace(0.3, 0.9, 5)
# And color the trajectories differently
posit_colors = plt.cm.autumn_r(linspace(0.3, 1., len(posit_X0)))
```

```
f2 = plt.figure()

for posit, col in zip(posit_X0, posit_colors):
    X0 = posit * X_f1
    X = odeint(dX_dt, X0, t)
    plt.plot(X[:,0], X[:,1], lw=3.5*posit,
             color = col, label = 'X0 = (%.f, %.f)' % (X0[0], X0[1]))

''' Now we'll define the coordinates for the
phase plane by getting the limits of each axis '''
ymax = plt.ylim(ymin=0)[1]
xmax = plt.xlim(xmin=0)[1]
num_points = 20

x = linspace(0, xmax, num_points)
y = linspace(0, ymax, num_points)

X1, Y1 = meshgrid(x, y)     # This creates a mesh grid
DX1, DY1 = dX_dt([X1, Y1]) # Growth rate of the grid
M = (hypot(DX1, DY1))
M[M == 0] = 1.              # Avoid zero division errors
DX1 /= M
DY1 /= M

# And we'll draw the direction fields as exeplified in the tutorial
plt.title('Population Phase Plane, Equal Params')
Q = plt.quiver(X1, Y1, DX1, DY1, M, pivot='mid', cmap=plt.cm.jet)
plt.xlabel('Number of prey')
plt.ylabel('Number of predators')
plt.legend()
plt.grid()
plt.xlim(0, xmax)
plt.ylim(0, ymax)
f2.savefig('predator_prey_1b.png')
```

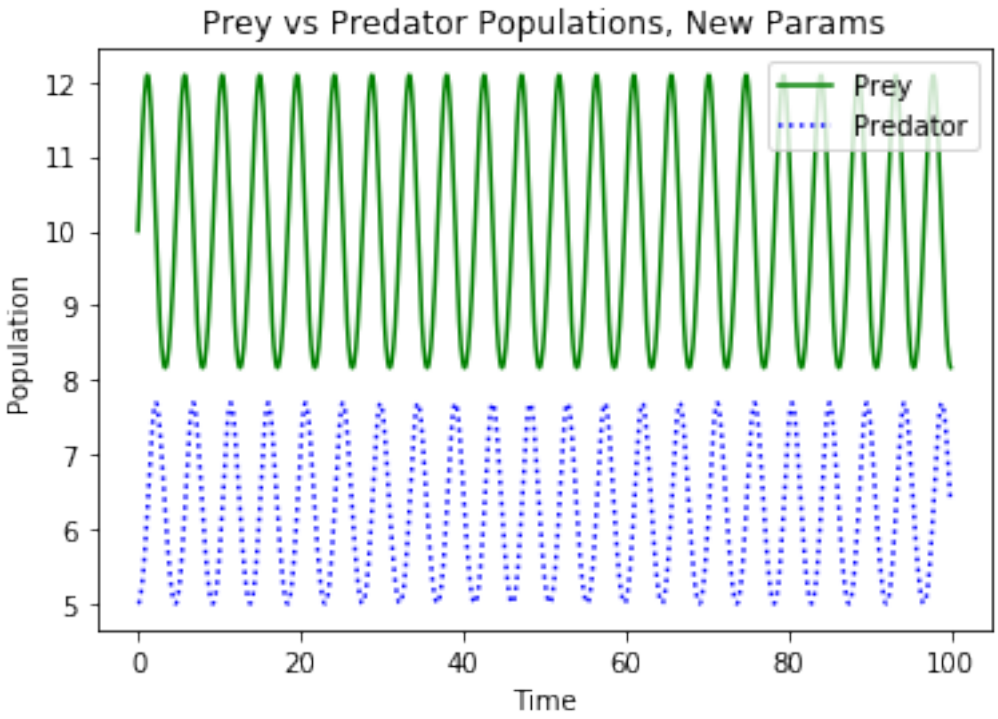

Prey vs Predator Populations, Equal Params
Prey
Predator
Population
Time
0
2
4
6
8
10
12
0
20
40
60
80
100

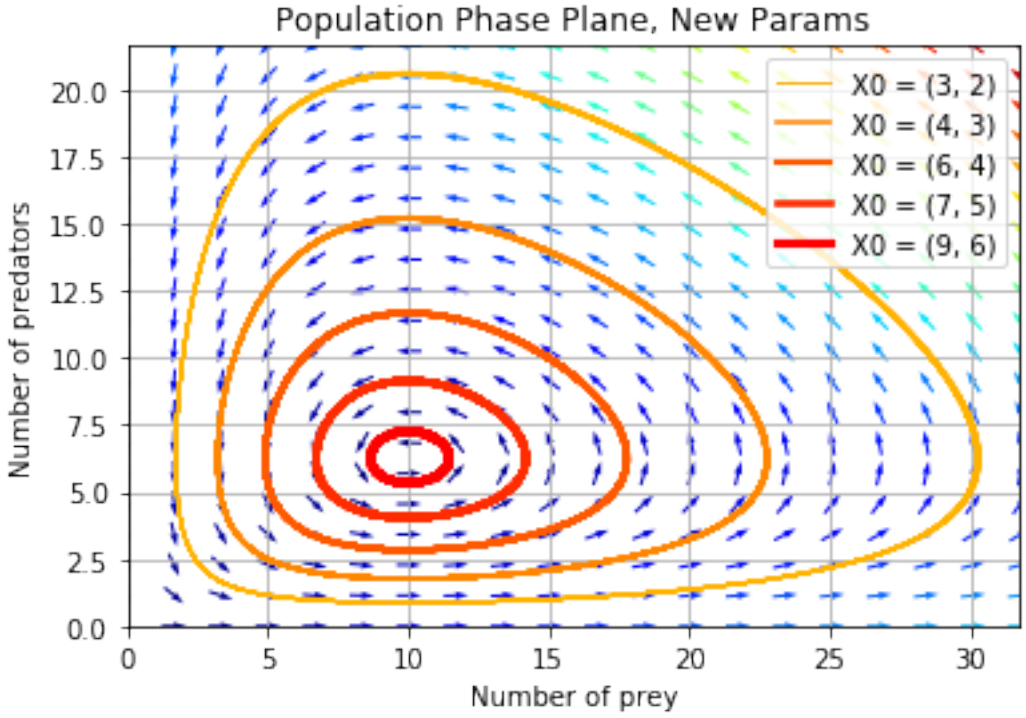

Population Phase Plane, Equal Params
X0 = (0, 0)
X0 = (0, 0)
X0 = (1, 1)
X0 = (1, 1)
X0 = (1, 1)
Number of predators
Number of prey
0.0
0.5
1.0
1.5
2.0
2.5
3.0

We notice that in each oscillation, both populations are dropping do essentially zero. This has been identified as one of the weaknesses of using ODEs to model predator/prey population dynamics. What's happening is that the populations are down to a tiny fraction very close to, but not zero. Of course, this cannot happen in reality as we need at least a breeding pair consisting of two whole individuals for each species (unless it can reproduce asexually), and oftentimes even that is not sufficient to rebount a population.

[105]:
```
# Now we can play around with parameters a-d
a = 1.25
b = .2
c = 1.5
d = .75

X_f0 = array([0., 0.])
X_f1 = array([c/(d*b), a/b])
all(dX_dt(X_f0) == zeros(2)) and all(dX_dt(X_f1) == zeros(2))

# Near X_f0 is the extinction point of both species
A_f0 = d2X_dt2(X_f0)
A_f1 = d2X_dt2(X_f1)

''' Both population functions are periodic, so we'll
determine the eigenvalues, which are imaginary to
then determine the oscillation patterns '''
lambda1, lambda2 = linalg.eigvals(A_f1)
T_f1 = 2 * pi / abs(lambda1)   # The period of oscillation

# Finally, we can integrate:
t = linspace(0, 100, 1000)   # Time steps and duration
X0 = array([10,5])            # initial populations
X = odeint(dX_dt, X0, t)


# And to plot the results:
prey, predator = X.T
f1 = plt.figure()
plt.plot(t, prey, 'g', label='Prey')
plt.plot(t, predator, 'b:', label='Predator')
plt.legend(loc='best')
plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Prey vs Predator Populations, New Params')
f1.savefig('predator_prey_2a.png')

# Now we'll plot the population trajectories in a phase plane:
posit_X0 = linspace(0.3, 0.9, 5)
# And color the trajectories differently
```

```
posit_colors = plt.cm.autumn_r(linspace(0.3, 1., len(posit_X0)))

f2 = plt.figure()

for posit, col in zip(posit_X0, posit_colors):
    X0 = posit * X_f1
    X = odeint(dX_dt, X0, t)
    plt.plot(X[:,0], X[:,1], lw=3.5*posit,
             color = col, label = 'X0 = (%.f, %.f)' % (X0[0], X0[1]))

''' Now we'll define the coordinates for the
phase plane by getting the limits of each axis '''
ymax = plt.ylim(ymin=0)[1]
xmax = plt.xlim(xmin=0)[1]
num_points = 20

x = linspace(0, xmax, num_points)
y = linspace(0, ymax, num_points)

X1, Y1 = meshgrid(x, y)     # This creates a mesh grid
DX1, DY1 = dX_dt([X1, Y1]) # Growth rate of the grid
M = (hypot(DX1, DY1))
M[M == 0] = 1.             # Avoid zero division errors
DX1 /= M
DY1 /= M

# And we'll draw the direction fields as exeplified in the tutorial
plt.title('Population Phase Plane, New Params')
Q = plt.quiver(X1, Y1, DX1, DY1, M, pivot='mid', cmap=plt.cm.jet)
plt.xlabel('Number of prey')
plt.ylabel('Number of predators')
plt.legend()
plt.grid()
plt.xlim(0, xmax)
plt.ylim(0, ymax)
f2.savefig('predator_prey_2b.png')
```


Prey vs Predator Populations, New Params
Prey
Predator
Population
Time
12
11
10
9
8
7
6
5
0
20
40
60
80
100

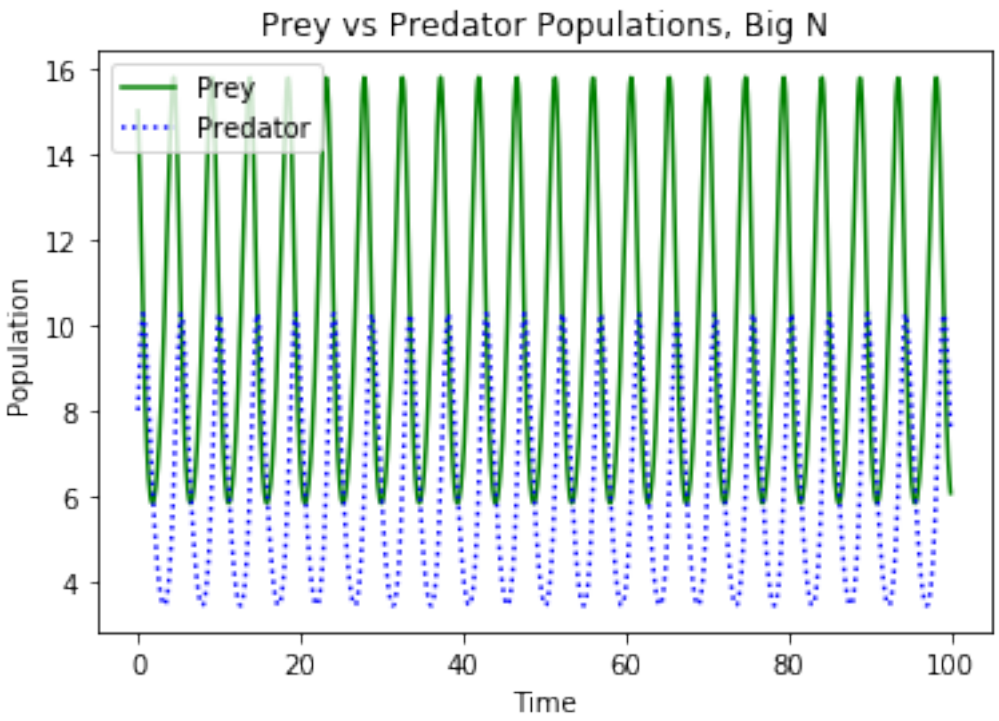

Population Phase Plane, New Params
X0 = (3, 2)
X0 = (4, 3)
X0 = (6, 4)
X0 = (7, 5)
X0 = (9, 6)
Number of predators
Number of prey
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
5
10
15
20
25
30

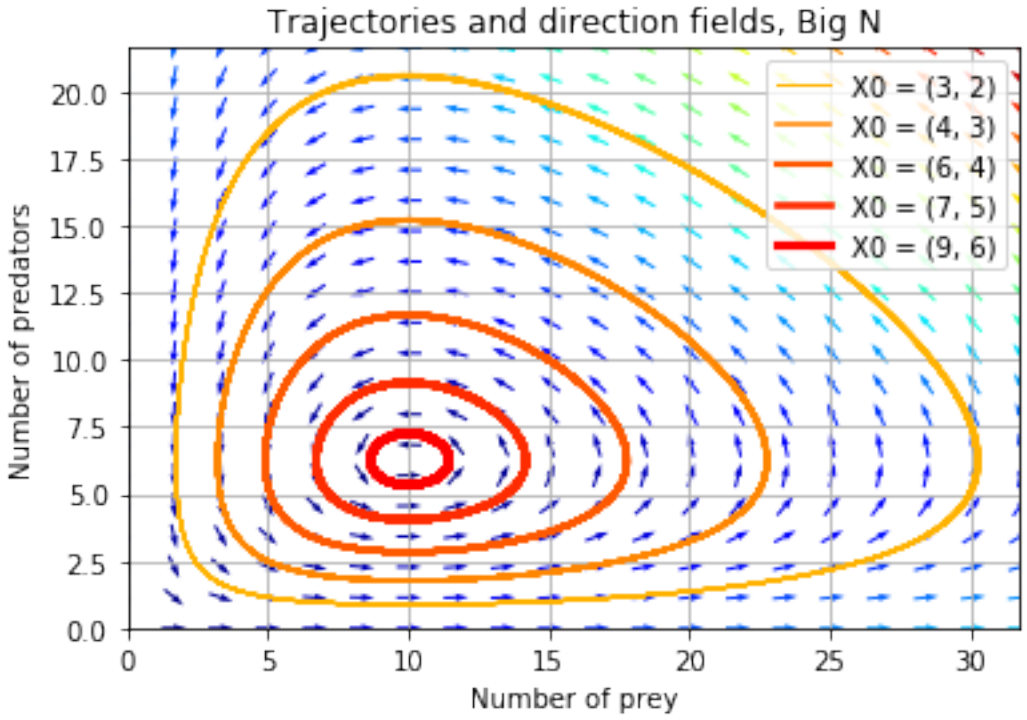
Here we see parameters which interestingly keep both populations oscillating very stably within two separate respective ranges, but given the same starting populations as before.

Now let's keep these parameters, but change the starting populations:

[106]:
```
# Now we can play around with the parameters
a = 1.25
b = .2
c = 1.5
d = .75

X_f0 = array([0., 0.])
X_f1 = array([c/(d*b), a/b])
all(dX_dt(X_f0) == zeros(2)) and all(dX_dt(X_f1) == zeros(2))

# Near X_f0 is the extinction point of both species
A_f0 = d2X_dt2(X_f0)
A_f1 = d2X_dt2(X_f1)

''' Both population functions are periodic, so we'll
determine the eigenvalues, which are imaginary to
then determine the oscillation patterns '''
lambda1, lambda2 = linalg.eigvals(A_f1)
T_f1 = 2 * pi / abs(lambda1)    # The period of oscillation

# Finally, we can integrate:
t = linspace(0, 100, 1000)    # Time steps and duration
X0 = array([15,8])             # initial populations
X = odeint(dX_dt, X0, t)


# And to plot the results:
prey, predator = X.T
f1 = plt.figure()
plt.plot(t, prey, 'g', label='Prey')
plt.plot(t, predator, 'b:', label='Predator')
plt.legend(loc='best')
plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Prey vs Predator Populations, Big N')
f1.savefig('predator_prey_3a.png')

# Now we'll plot the population trajectories in a phase plane:
posit_X0 = linspace(0.3, 0.9, 5)
# And color the trajectories differently
posit_colors = plt.cm.autumn_r(linspace(0.3, 1., len(posit_X0)))

f2 = plt.figure()
```

```
for posit, col in zip(posit_X0, posit_colors):
    X0 = posit * X_f1
    X = odeint(dX_dt, X0, t)
    plt.plot(X[:,0], X[:,1], lw=3.5*posit,
             color = col, label = 'X0 = (%.f, %.f)' % (X0[0], X0[1]))

''' Now we'll define the coordinates for the
phase plane by getting the limits of each axis '''
ymax = plt.ylim(ymin=0)[1]
xmax = plt.xlim(xmin=0)[1]
num_points = 20

x = linspace(0, xmax, num_points)
y = linspace(0, ymax, num_points)

X1, Y1 = meshgrid(x, y)     # This creates a mesh grid
DX1, DY1 = dX_dt([X1, Y1]) # Growth rate of the grid
M = (hypot(DX1, DY1))
M[M == 0] = 1.              # Avoid zero division errors
DX1 /= M
DY1 /= M

# And we'll draw the direction fields as exeplified in the tutorial
plt.title('Trajectories and direction fields, Big N')
Q = plt.quiver(X1, Y1, DX1, DY1, M, pivot='mid', cmap=plt.cm.jet)
plt.xlabel('Number of prey')
plt.ylabel('Number of predators')
plt.legend()
plt.grid()
plt.xlim(0, xmax)
plt.ylim(0, ymax)
f2.savefig('predator_prey_3b.png')
```

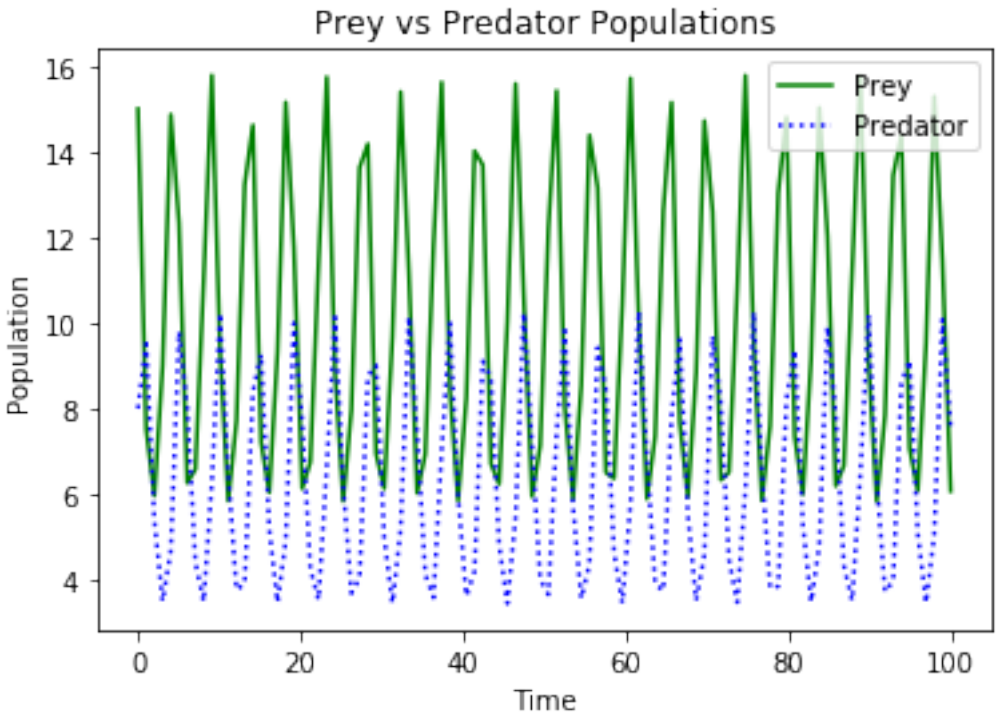

Prey vs Predator Populations, Big N
Prey
Predator
Population
Time
16
14
12
10
8
6
4
0
20
40
60
80
100

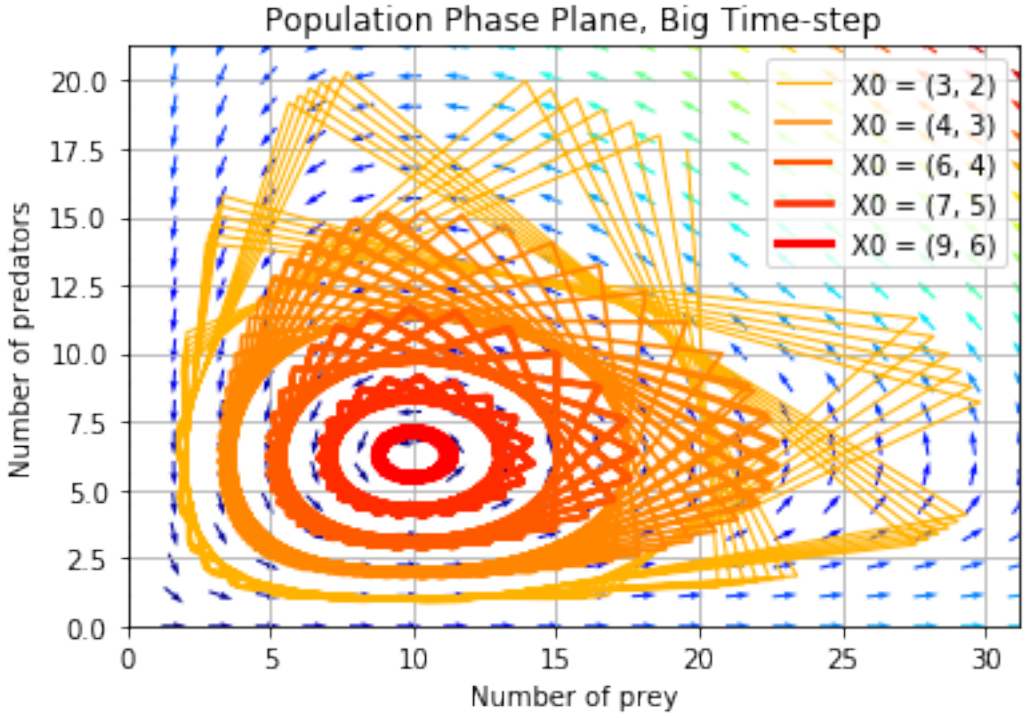

Trajectories and direction fields, Big N
X0 = (3, 2)
X0 = (4, 3)
X0 = (6, 4)
X0 = (7, 5)
X0 = (9, 6)
Number of predators
Number of prey
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
5
10
15
20
25
30

Now we see that the starting population acts like the amplitude coefficient to a sine function. Now let's try one more change, this time in the timestep:

[107]:
```
# Now we can play around with the parameters
a = 1.25
b = .2
c = 1.5
d = .75

X_f0 = array([0., 0.])
X_f1 = array([c/(d*b), a/b])
all(dX_dt(X_f0) == zeros(2)) and all(dX_dt(X_f1) == zeros(2))

# Near X_f0 is the extinction point of both species
A_f0 = d2X_dt2(X_f0)
A_f1 = d2X_dt2(X_f1)

''' Both population functions are periodic, so we'll
determine the eigenvalues, which are imaginary to
then determine the oscillation patterns '''
lambda1, lambda2 = linalg.eigvals(A_f1)
T_f1 = 2 * pi / abs(lambda1)   # The period of oscillation

# Finally, we can integrate:
t = linspace(0, 100, 100)     # Time steps and duration
X0 = array([15,8])            # initial populations
X = odeint(dX_dt, X0, t)


# And to plot the results:
prey, predator = X.T
f1 = plt.figure()
plt.plot(t, prey, 'g', label='Prey')
plt.plot(t, predator, 'b:', label='Predator')
plt.legend(loc='best')
plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Prey vs Predator Populations')
f1.savefig('predator_prey_4a.png')

# Now we'll plot the population trajectories in a phase plane:
posit_X0 = linspace(0.3, 0.9, 5)
# And color the trajectories differently
posit_colors = plt.cm.autumn_r(linspace(0.3, 1., len(posit_X0)))

f2 = plt.figure()
```

```
for posit, col in zip(posit_X0, posit_colors):
    X0 = posit * X_f1
    X = odeint(dX_dt, X0, t)
    plt.plot(X[:,0], X[:,1], lw=3.5*posit,
             color = col, label = 'X0 = (%.f, %.f)' % (X0[0], X0[1]))

''' Now we'll define the coordinates for the
phase plane by getting the limits of each axis '''
ymax = plt.ylim(ymin=0)[1]
xmax = plt.xlim(xmin=0)[1]
num_points = 20

x = linspace(0, xmax, num_points)
y = linspace(0, ymax, num_points)

X1, Y1 = meshgrid(x, y)     # This creates a mesh grid
DX1, DY1 = dX_dt([X1, Y1]) # Growth rate of the grid
M = (hypot(DX1, DY1))
M[M == 0] = 1.              # Avoid zero division errors
DX1 /= M
DY1 /= M

# And we'll draw the direction fields as exeplified in the tutorial
plt.title('Population Phase Plane, Big Time-step')
Q = plt.quiver(X1, Y1, DX1, DY1, M, pivot='mid', cmap=plt.cm.jet)
plt.xlabel('Number of prey')
plt.ylabel('Number of predators')
plt.legend()
plt.grid()
plt.xlim(0, xmax)
plt.ylim(0, ymax)
f2.savefig('predator_prey_4b.png')
```


Prey vs Predator Populations
Prey
Predator
Population
Time
16
14
12
10
8
6
4
0
20
40
60
80
100


Population Phase Plane, Big Time-step
X0 = (3, 2)
X0 = (4, 3)
X0 = (6, 4)
X0 = (7, 5)
X0 = (9, 6)
Number of predators
Number of prey
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
5
10
15
20
25
30

Now we see that when we reduce the time step to match the number of data points recorded, it is not sufficient to record a smooth graph. The output affects both the line graph and the phase plane. This is the same kind of problem that we saw above with the third graph of the **Logistic Growth** exercise.

I went a lot further with this exercise than I had with the previous two. This is partially because I was having fun going through the tutorial I had found that guided me through the initial coding process, and partially because I wanted to spend more time exploring the parameters of this more complex model.

I would say my primary benefit from this exercise was gaining more of a familiarity with the way that Python works and helping me to further develop my mental framework on how to approach problems using the language.

1.1.4 L2.1 Chaos and the Logistic Map

This exercise follows closely the tutorial provided at this page: https://ipython-books.github.io/121-plotting-the-bifurcation-diagram-of-a-chaotic-dynamical-system/

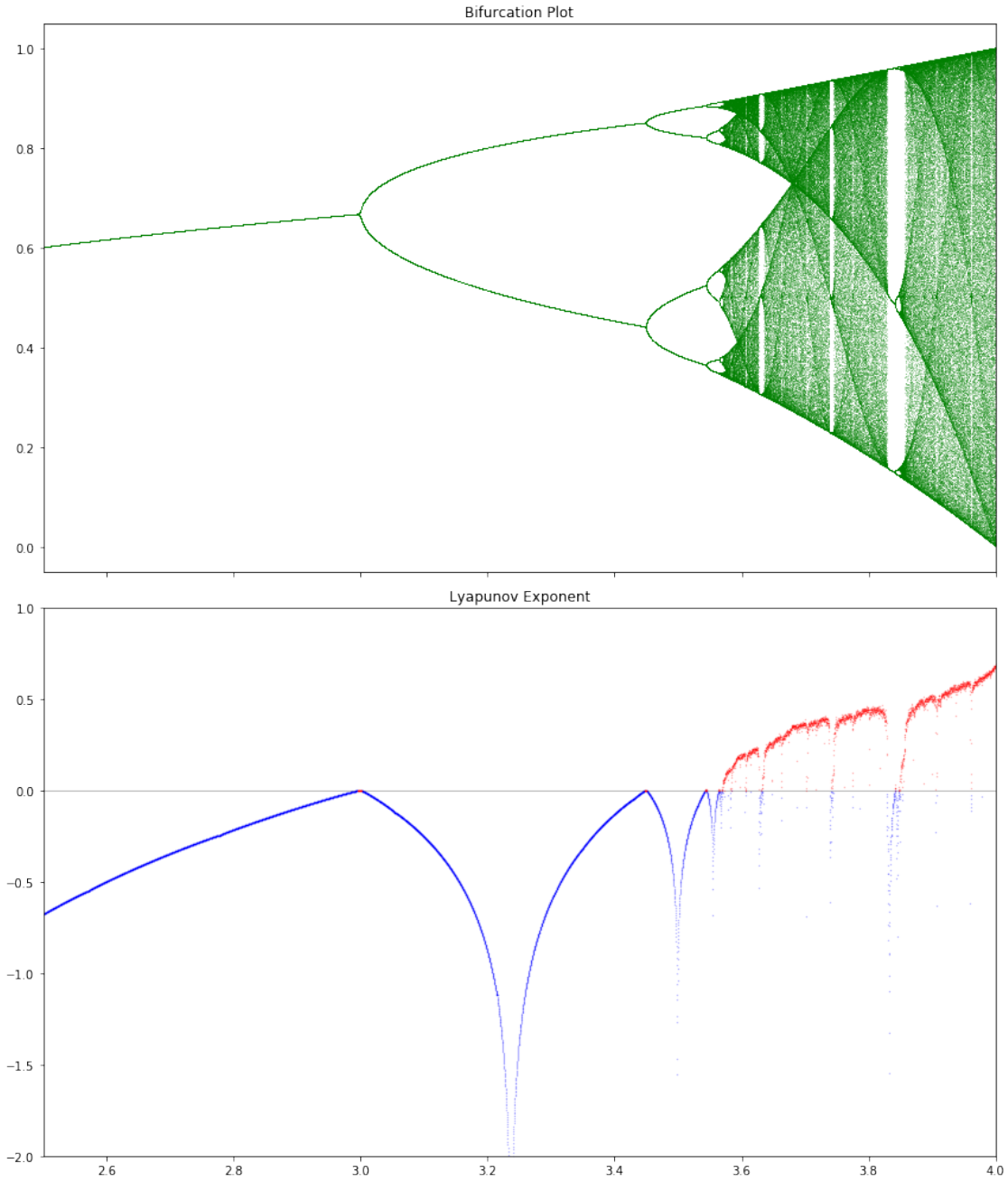
And I also watched this YouTube video on Veritasium to give myself an overview of the system being modeled: https://www.youtube.com/watch?v=ovJcsL7vyrk

These graphs demonstrate how big of a difference can be made by choosing a different value of r, with the right side graph showing chaotic behavior.

Using this system we will establish a vector of 10,000 points for r.

[108]:
```
# First we'll define the logistic function:
def logistic(r, x):
    return r * x * (1 - x)

n_points = 10000
r = np.linspace(2.5, 4.0, n_points)

# Now we'll iterate the logistic map and keep the last 100 of those to show the␣
 ↪bifurcations:
iterate = 1000
final = 100

# And to set the initial condition
x = 1e-5 * np.ones(n_points)

''' And to simulate the system. The lyapunov corresponds to the bifurcation plot.
 ↪ We'll see in the
plots below that each time the lyapunov reaches zero the system bifurcates, and␣
 ↪when it goes above
zero the system becomes chaotic '''
lyapunov = np.zeros(n_points)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 14), sharex=True)
```

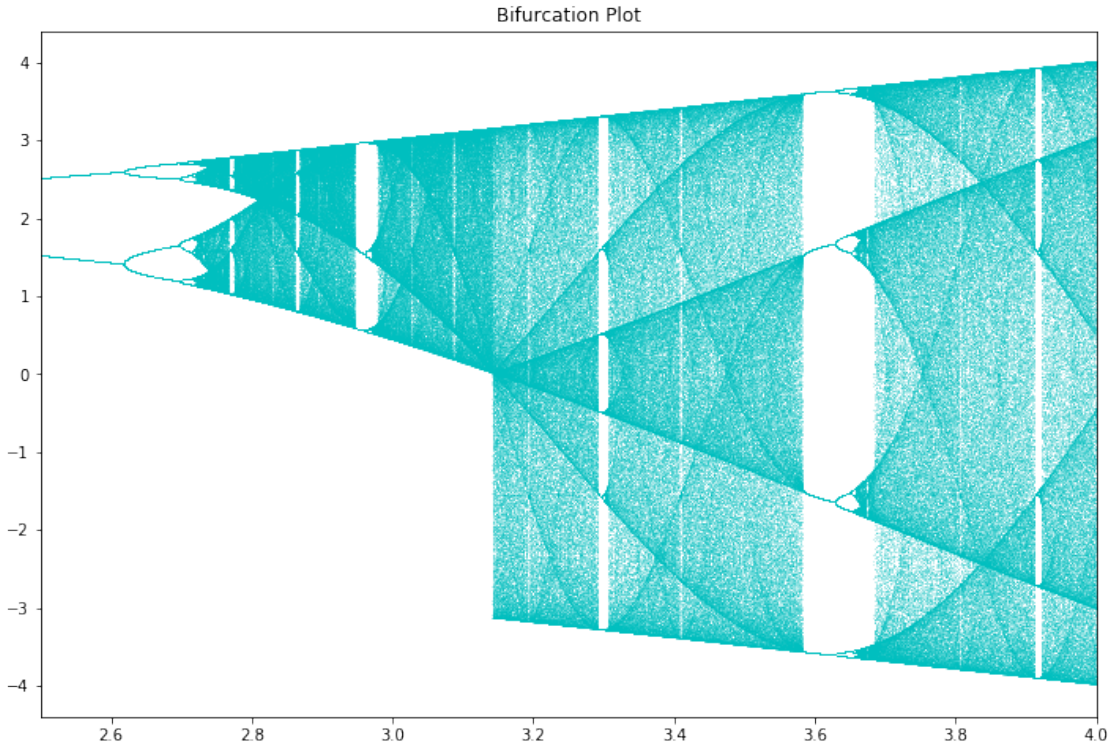
```
for i in range(iterate):
    x = logistic(r, x)

    # Computing the partial sum of the Lyapunov exponent.
    lyapunov += np.log(abs(r - 2 * r * x))

    # And to generate the bifurcation plot:
    if i >= (iterate - final):
        ax1.plot(r, x, ',g', alpha=.25)

ax1.set_xlim(2.5, 4)
ax1.set_title("Bifurcation Plot")

# Plotting the Lyapunov with a line at zero and red when positive
ax2.axhline(0, color='k', lw=.5, alpha=.5)
ax2.plot(r[lyapunov < 0],
         lyapunov[lyapunov < 0] / iterate,
         '.b', alpha=.5, ms=.5)
# Positive Lyapunov exponent.
ax2.plot(r[lyapunov >= 0],
         lyapunov[lyapunov >= 0] / iterate,
         '.r', alpha=.5, ms=.5)
ax2.set_xlim(2.5, 4)
ax2.set_ylim(-2, 1)
ax2.set_title("Lyapunov Exponent")
plt.tight_layout()
```



It turns out that chaos applies in this way to other mathematical systems. For instance, we can observe similar bifurcations if we use a sinusoidal function:

[109]:
```
def sinusoid(r, x):
    return r * sin(x)

fig, ax = plt.subplots(figsize=(12, 8))
for i in range(iterate):
```

```
    x = sinusoid(r, x)

    # And to generate the bifurcation plot:
    if i >= (iterate - final):
        plt.plot(r, x, ',c', alpha=.25)

plt.xlim(2.5, 4)
plt.title("Bifurcation Plot")
show()
```



Admittedly I'm not yet sure how to adjust the plot for the lyapunov exponent related to plotting the bifurcated sinusoid.

The challenges related to this exercise were very conceptual. I've noticed that while there are many ways to try to tackle these the methods ultimately come down to some sort of iterative process. Once the iterative process is established, it is easy to make adjustments to the code, allowing for different types of functions to be passed through the bifurcation process.