Series RLC Circuit Model: Current vs. Rate of Change for Current

Name: Christ-Brian Amedjonekou

Date: 04/03/2019

MAT 4880-D692 (Math Modeling II) RLC Model Project 2

Spring 2019, Section: D692, Code: 36561

Instructor: Johann Thiel

Abstract

In this study, we set out to create an RLC Model of a Simple Series RLC Circuit. The goal was to find the steady state of the dynamical system given the $\frac{dq}{dt}$ (rate of change of q) and its rate of change $\frac{d^2q}{dt^2}$ as state variables. *RLC* circuits are used in many electronic systems, most notably as tuners in AM/FM radios. These circuits can be modeled by second-order, constant-coefficient differential equations; This is what we did here. We created a continuous dynamical system from the second-order differential equation, as mentioned above. We used sage/python to model and analyze this problem. Once modeled, finding the equilibrium points to the system was the next step as they show the value in which the model reaches its steady state. Reaching the steady state (finding the equilibrium points) indicates that the recently observed behavior of the system will continue into the future. What that means for this problem is that we'll expect the current and its of change to converge to a point and remain there, or at least this is what we believe. Finally, we classify the equilibrium points using the Eigenvalue Method, and show a graphic of the model in the form of a Phase Portrait.

Introduction

RLC circuits are simple electronic circuits most commonly used as tuners in AM/FM radios, as well as other electronic systems. As aforementioned, we can model these circuits by second-order, constant-coefficient differential equations. For this report, our goal is to determine if the current and its rate of change will reach the steady state. In order to do that we convert the second-order differential ($E(t) = L\frac{d^2q}{dt^2} + R\frac{dq}{dt} + \frac{1}{C}q$) into a continuous dynamical system, where we have charge ($x_1 = q$) and current ($x_2 = I = \frac{dq}{dt}$) as the state variables. We'll be looking to find the equilibrium points of the system as that will determine the steady state of the system; This will be verified through the Eigenvalue Method and displayed graphically in the form of a Phase Portrait.

Assumptions and Definitions

For the model of the continuous dynamical system, we'll assume the following:

$$E(t) = L\frac{d^2q}{dt^2} + R\frac{dq}{dt} + \frac{1}{C}q$$

$$Let\ x_1 = q;\ x_2 = I = \frac{dq}{dt}$$

$$\frac{\partial x_1}{\partial t} = \frac{dq}{dt} = x_2$$

$$\frac{\partial x_2}{\partial t} = \frac{d^2q}{dt^2} = \frac{E(t) - Rx_2 - \frac{1}{C}x_1}{L}$$

- where $x_1 = q = the\ charge\ of\ the\ capacitor$
- where $x_2 = \frac{dq}{dt} = rate\ of\ change\ of\ the\ charge\ (current)$
- where $E(t) = Electric\ Potential\ of\ the\ RLC\ Circuit = 300\ V$
- $R = resistance\ of\ the\ resistor = 10\ \Omega$
- $C = capacitance\ of\ the\ capacitor = \frac{1}{30}\ Farads$
- $L = inductance\ of\ the\ Inductor = \frac{5}{3}\ Henries$

The reason we convert the second order differential equation to a continuous dynamical system is because we don't have a method for evaluating a nonlinear second-order differential in its current form. As

aforementioned we convert the second-order differential ($E(t) = L\frac{d^2q}{dt^2} + R\frac{dq}{dt} + \frac{1}{C}q$) into a continuous dynamical system, where we have charge ($x_1 = q$) and current ($x_2 = I = \frac{dq}{dt}$) as the state variables. We are given the values for the resistor ($R = 10\ \Omega$), capacitor ($C = \frac{1}{30}\ Farads$), and inductor ($L = \frac{5}{3}\ Henries$). The resulting dynamical system is $\frac{\partial x_1}{\partial t} = x_2;\ \frac{\partial x_2}{\partial t} = \frac{3}{5}(300 - 10x_2 - 30x_1)$.

Analysis

To solve the system, we set both differential equations to 0; Solving for the equations set to 0 gives the equilibrium points. This is because if the rate of change is 0, it means there is no change and the dynamical system has reached the steady state. Python, specifically the Sympy (Symbolic Python) package, was used to find the result for the system. $x_1 = 10; x_2 = 0$ are the equilibrium points to the system. To classify the equilibrium points, we used the Eigenvalue Method. Simply put, we calculated the Jacobian Matrix (using a python function and SciPy), found the determinant ($\det(A - \lambda I) = 0$), and solved for λ. The Jacobian Matrix is shown in Figure 1. We found that $\lambda_1 = -3 + 3i; \lambda_2 = -3 - 3i$. Since both eigenvalues have a negative real part we classify the equilibrium point (10, 0) as stable. Other Python packages, specifically NumPy (Numerical Python) and Matplotlib (Plotting Package), were used to plot the Phase Portrait of the nonlinear system to determine which equilibrium the vectors converge to. This is shown below in Figure 2 below:

$$A = \begin{bmatrix} \frac{\partial f_1}{x_1}x_0 & \dots & \frac{\partial f_1}{x_n}x_0 \\ \frac{\partial f_n}{x_1}x_0 & \dots & \frac{\partial f_n}{x_n}x_0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ -18.0 & -6.0 \end{bmatrix}$$

Figure 1: Jacobian Matrix for RLC Circuit Continuous Dynamical System

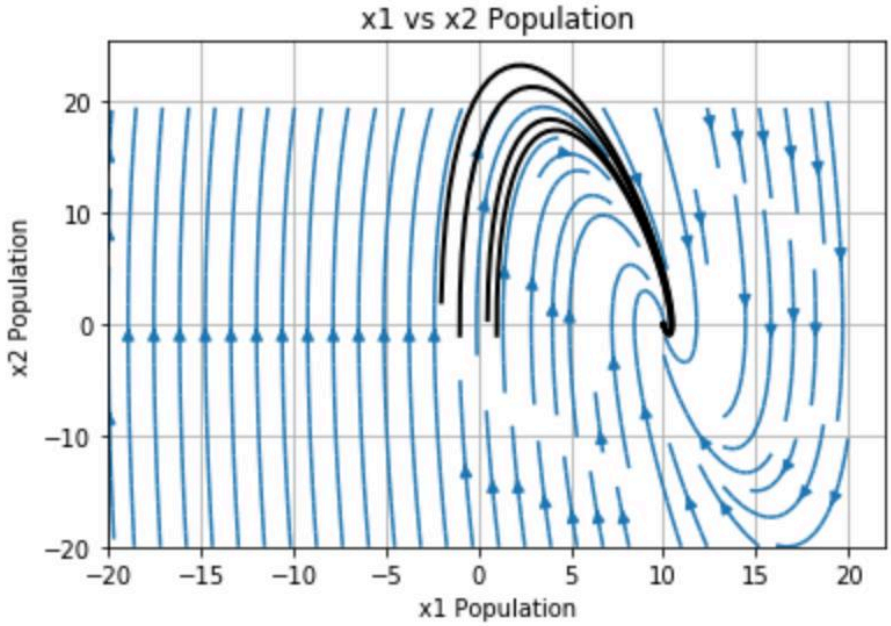


Figure 2: Phase Portrait for RLC Circuit Continuous Dynamical System

The Phase Portrait shows that the vectors of the dynamical system converge to $(10, 0)$, which seems to indicate the current vs. its rate of change will be stable at that point. The Trajectory Lines further demonstrates this, as they also converge to $x_1 = 10; x_2 = 0$ in the form of a spiral as this is determined by the eigenvalues.

Interpretations and Conclusions

Based on the results obtained from Eigenvalue Method, where both eigenvalues had negative real parts, we can assume that current and its rate of change will reach the steady state at $x_1 = 10; x_2 = 0$ and stay there for future events. This is also supported by the Phase Portrait as the trajectory lines travel in a spiral towards the equilibrium point.

Appendix to the RLC Electrical Circuit Model

April 3, 2019

- Section **??**
- Section **??**
- Phase Portrait: Discrete Astronaut Problem

0.1 Chapter 5.3 Phase Portraits

0.1.1 Theory: Short Version

- Phase Portraits combine the earlier vector plot (a graphical unit) with the eigenvalue method to create a graphical description of the dynamical system over the entire state space. The are important in the analysis of nonlinear dynamical systems because in most cases perfect analytical solutions are hard to come by.

0.1.2 Theory: Long Version

- Recall that if we're given a dynamical system $x' = f(x)$, $x = (x_1, ..., x_n)$ is an element of the state space $S \subseteq R^N$ and $F = (f_1, ..., f_n)$ is the continuous first partials in the neighborhood of an equilibrium point, x_0.
- Also recall that even if $x' = F(x)$ is not linear, we have the following:

$$F(x) \approx A(x - x_0)$$

  in the neighborhood of the equilibrium point.

  - Basically we have the linear approximation which is equivalent to the non-linear dynamical model around the equilibrium point.

- The phase portrait itself is simply the sketch of the state space showing a representative selection of the solution curves (graphing the solution curves for a few initial conditions).
- The reason why we can create a phase portrait of the state space of a non-linear dynamical system using a linear approximation, is not only because the linear approximation is equivalent to the non-linear dynamical model around the equilibrium point. ***It's also because of the principle called homeomorphism.***

Homeomorphism.

- Describes a continuous function that has a continuous inverse.
- The general idea of homeomorphism involves the shape and generic properties of these continuous functions.
  - If the function is homeomorphic, the shape of the function can change from circle to an ellipse, another circle, triangle, to square but never a figure 8 (no inverse) or a line (this would violate continuity).

Thus we have a theorem that states the following:

- If there's an equilibrium point (that is, if the eigenvalues of the system have all nonzero real parts), then there is a homeomorphism that maps the phase portrait of the linear approximation, $x' = Ax$, to the non-linear system, $x' = F(x)$, around the equilibrium point (albeit w/ some distortion).

0.1.3 Continuous Case

RLC Circuit Example

Variables

- $E_C =$ Voltage across Capacitor
- $E_R =$ Voltage across resistor
- $E_L =$ Voltage across inductor

Assumptions

- $E_C = \frac{1}{C}q$
- $E_R = RI = R\frac{dq}{dt}$
- $E_L = L\frac{d^2q}{dt^2}$
- $E(Q) = E_C + E_R + E_L = \frac{1}{C}q + R\frac{dq}{dt} + L\frac{d^2q}{dt^2}$

If we let $x_1 = q$ and $x_2 = \frac{dq}{dt}$, then the change in x_1 is $\frac{dx_1}{dt} = \frac{dq}{dt} = x_2$. The change in x_2 is $\frac{dx_2}{dt} = \frac{d^2q}{dt^2} = \frac{E - Rx_2 - \frac{1}{C}x_1}{L}$

- $f(x_1, x_2) = x_2$
- $f(x_1, x_2) = \frac{E - Rx_2 - \frac{1}{C}x_1}{L}$

- $L = 5/3$
- $R = 10$
- $C = 1/30$
- $E = 300$

Objective

1. Model the charge of the capacitor by a continuous dynamical system. Hint: transform the second-order differential equation into a dynamical system by letting $x_1 = q$ and $x_2 = \frac{dq}{dt}$.
2. Find and classify the equilibrium point(s) of the system using the eigenvalue method.
3. Sketch the phase portrait of the system near the point(s) of equilibrium.

0.1.4 Modules

```
In [1]: # Modules being called
        import math as m
        import numpy as np
        from sympy.solvers import solve
        import sympy as sp
        from matplotlib import pyplot as plt
        import scipy.linalg as sci
        import scipy.integrate as scint
```

0.1.5 Variables

```
In [2]: # Writing the Model for the deer population
        x1, x2 = sp.symbols('x1 x2')


        # init_session() displays LaTeX version of outputs; 'quiet= True' stops
        # init_session from printing messages regarding its status
        sp.init_printing()
```

0.1.6 Sympy Rendition

```
In [81]: # Output
         L = 5/3
         R = 10
         C = 1/30
         E = 300
         dx1dt = x2
```

```
dx2dt = (E-R*x2-(1/C)*x1)/L
(dx1dt, dx2dt)
```

Out[81]:

$$(x_2, \quad -18.0x_1 - 6.0x_2 + 180.0)$$

0.1.7 Equilibrium Points

(a) The equilibrium points are shown below. They occur when $\frac{dx1}{dt} = \frac{dx2}{dt} = 0$

In [86]:
```
# Solution to the model(Equilibrium Point)

equilibrium_points = solve([dx1dt, dx2dt], x1, x2)
print('Equilibrium Points: {}'.format(equilibrium_points))
```

```
Equilibrium Points: {x1: 10.0000000000000, x2: 0.0}
```

Out[86]:

$$\{x_1 : 10.0, \quad x_2 : 0.0\}$$

0.1.8 Lambda Functions

In [87]:
```
dx1_dt = lambda x1_, x2_: x2_
dx2_dt = lambda x1_, x2_: (E-R*x2_-(1/C)*x1_)/L
```

0.1.9 Functions

In [88]:
```
def plot_traj(ax1, g1, g2, x0, t, args=(), color='black', lw=2):
    """
    Plots a vector field plot.

    Parameters
    ----------
    ax : Matplotlib Axis instance
        Axis on which to make the plot
    g1 : function of the form g1(x1_)
        1/2 The right-hand-side of the dynamical system.
    g2 : function of the form g2(x1_)
        1/2 The right-hand-side of the dynamical system.
    x0 : array_like, shape (2,)
        Initial conditions of for the trajectory lines.
    t : array_like
        Time points for trajectory.
    args : tuple, default ()
        Additional arguments to be passed to f
    color : Color of the trajectory lines.
```

```
        Set to 'black'
    linewidth: abbreviated as 'lw'. Default Value = 2

    Returns
    -------
    output : Matplotlib Axis instance
        Axis with streamplot included.
    """

    # Creates the set of points initialized by the parameter 'x'
    # for the difference equations g1 = -x1**3 - 4*x1 - x2; g2 = 3*x1
    def func(x,t):
        x_1, x_2 = x
        G1 = g1(x_1,x_2)
        G2 = g2(x_1,x_2)
        return [G1,G2]

    soln = scint.odeint(func, x0, t, args=args)
    ax1.plot(*soln.transpose(), color=color, lw=lw)
    return ax1
```

0.1.10 Plotting VectorField Plot

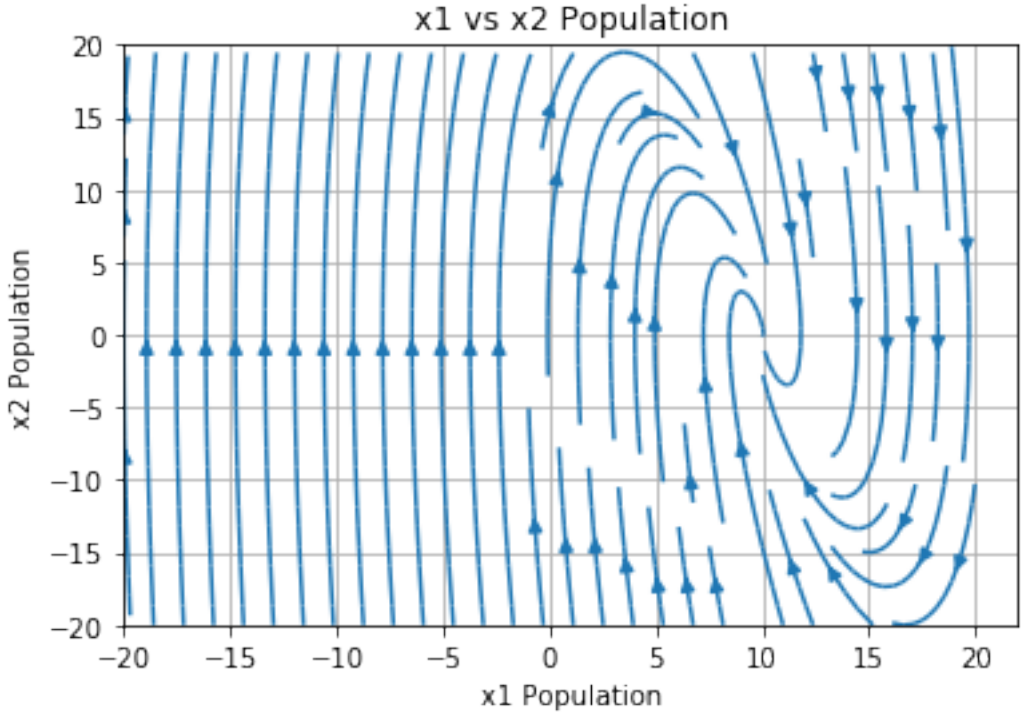
```
In [89]: def plot_vector_field(func1, func2, start= -3, stop= 3):
    """
    Plots a trajectory on a phase portrait.

    Parameters
    ----------
    func1 : function for form func1(y, t, *args)
        The right-hand-side of the dynamical system.
        Must return a 2-array.
    start : integer. Default value = 3
        The starting value.
    stop : integer. Default value = 3
        The ending value
    color : Color of the trajectory lines.
        Set to 'black'
    linewidth: abbreviated as 'lw'. Default Value = 2

    Returns
    -------
    output : Matplotlib Axis instance
        Axis with streamplot included.
    """
    #-----------------------------------------------------------------------------
    # Creates the superimposed plot for stream plot of the model, as well as dPdt = 0
    #-----------------------------------------------------------------------------
```

```
    # Part 1: Creates the length of the 'X' and 'Y' Axis
    x, y = np.linspace(start, stop), np.linspace(start, stop)
    X, Y = np.meshgrid(x, y)

    # Part 2: The approximated points of the function dx1/dt and dx2/dt which we'll use
    U, V = func1(X, Y), func2(X, Y)

    # Part 3: Creating the figure for the plot
    fig, ax1 = plt.subplots()

    # Part 4: Sets the axis, and equilibrium information for the plot
    Title, xLabel, yLabel = input('Title?: '), input('x-axis label?: '), input('y-axis
    ax1.set(title= Title, xlabel= xLabel, ylabel = yLabel)

    # Part 5: Plots the streamplot which represents the vector plot.
    ax1.streamplot(X, Y, U, V)
    ax1.grid()
    return ax1
```

In [118]: `plot_vector_field(dx1_dt, dx2_dt, -20, 20)`

```
Title?: x1 vs x2 Population
x-axis label?: x1 Population
y-axis label?: x2 Population
```

Out[118]: `<matplotlib.axes._subplots.AxesSubplot at 0x11b8da908>`



0.1.11 Jacobian Matrix

In [91]:
```
### Jacobian Matrix

        def poorManJacobian(func1, func2, var1, var2, points, Jac_matrix_On= True):
            """
            Plots the phase portrait of the actual dynamical system.

            Parameters
            ----------
            func1,func2 : sympy data type called for form sympy.core.add.Add.
                The data types of the functions we define using sympy.
                It just returns sympy.core.add.Add.
            var1, var2 : sympy data type called for form sympy.core.symbol.Symbol.
                The data types of the variables we define using sympy.
                It just returns sympy.core.symbol.Symbol.
            A_matrix_On : just a boolean check to return different things.
                If it's 'True' the it just returns the Jacobian w/o being
                evaluated at certain values of x1 and x2.
                Default value is 'True'

            Returns
            -------
            output : Matplotlib Axis instance
                Axis with streamplot included.
            """
            if Jac_matrix_On:
                Jac_matrix = sp.Array([[sp.diff(func1, var1), sp.diff(func1, var2)],
                                       [sp.diff(func2, var1), sp.diff(func2, var2)]])
                return Jac_matrix
            else:
                for point in points:
                    solMatrix = np.array([[float(sp.diff(func1, var1).subs({var1:point[0], var2
                                           float(sp.diff(func1, var2).subs({var1:point[0], var2:point[
                                          [float(sp.diff(func2, var1).subs({var1:point[0], var2:point[
                                           float(sp.diff(func2, var2).subs({var1:point[0], var2:point[
                    return solMatrix
```

In [93]:
```
points = [[10, 0]]
poorManJacobian(dx1dt, dx2dt, x1, x2, points)
```

Out[93]:

$$\begin{bmatrix} 0 & 1 \\ -18.0 & -6.0 \end{bmatrix}$$

0.1.12 Setting up the Matrix

```
In [94]: solMatrix1 = poorManJacobian(dx1dt, dx2dt, x1, x2, points, False)
         solMatrix1
```

```
Out[94]: array([[  0.,   1.],
                [-18.,  -6.]])
```

0.1.13 Eigenvalues

```
In [95]: eigenvalues1, eigenvectors1 = sci.eig(solMatrix1)
```

For $(x_1 = 10, x_2 = 0)$, both eigenvalues have negative real parts which make it stable.

```
In [96]: eigenvalues1
```

```
Out[96]: array([-3.+3.j, -3.-3.j])
```

0.1.14 Eigenvectors

For $(x_1 = 10, x_2 = 0)$.

```
In [97]: eigenvectors1
```

```
Out[97]: array([[-0.16222142-0.16222142j, -0.16222142+0.16222142j],
                [ 0.97332853+0.j        ,  0.97332853-0.j        ]])
```

0.1.15 Phase Plot Function

```
In [106]: def phasePortrait(func1, func2,  points, start= -20, stop= 20):
              """
              Plots the phase portrait of the actual dynamical system.

              Parameters
              ----------
              func1 : function for form func1(y, t, *args)
                  The right-hand-side of the dynamical system.
                  Must return a 2-array.
              start : integer. Default value = 3
                  The starting value.
              stop : integer. Default value = 3
                  The ending value
              color : Color of the trajectory lines.
                  Set to 'black'
              linewidth: abbreviated as 'lw'. Default Value = 2

              Returns
              -------
              output : Matplotlib Axis instance
                  Axis with streamplot included.
```

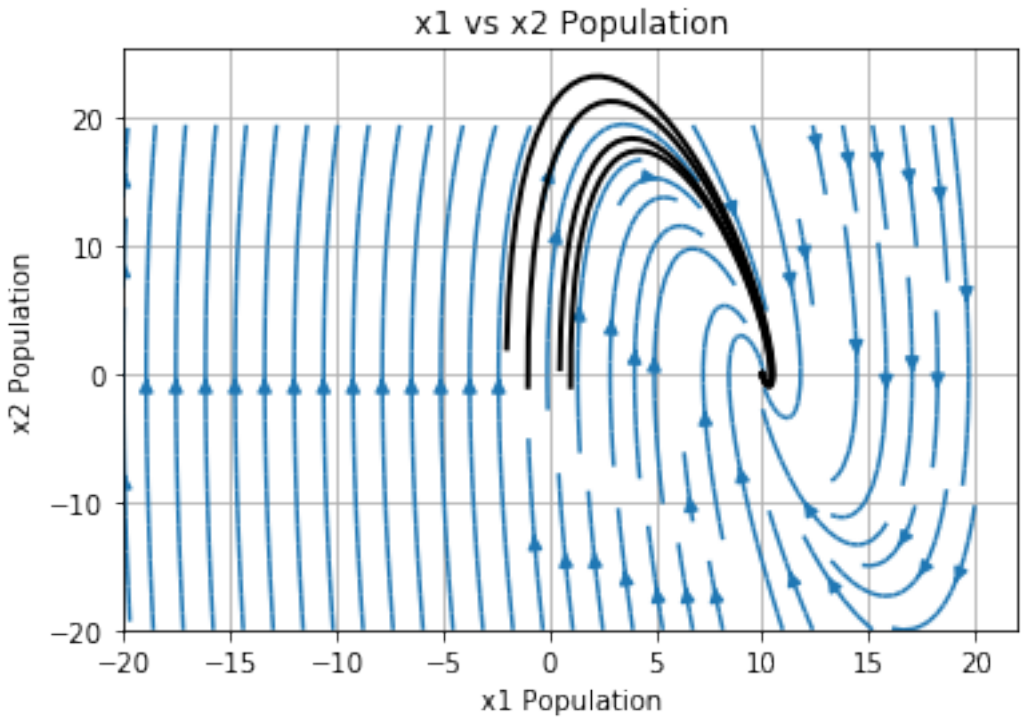
```
    """
    #-------------------------------------------------------------------------------
    # Creates the superimposed plot for stream plot of the model, as well as dPdt = 0
    #-------------------------------------------------------------------------------

    # Part 1: Creates the length of the 'X' and 'Y' Axis  and the time vector
    x, y = np.linspace(start, stop), np.linspace(start, stop)
    X, Y = np.meshgrid(x, y)
    t = np.linspace(0, 100, 5000)

    # Part 2: The approximated points of the function dx1/dt and dx2/dt which we'll us
    U, V = func1(X, Y), func2(X, Y)

    # Part 3: Creating the figure for the plot
    fig, ax1 = plt.subplots()

    # Part 4: Sets the axis, and equilibrium information for the plot
    Title, xLabel, yLabel = input('Title?: '), input('x-axis label?: '), input('y-axis
    ax1.set(title= Title, xlabel= xLabel, ylabel = yLabel)

    # Part 5: Plots the streamplot which represents the vector plot.
    ax1.streamplot(X, Y, U, V)
    ax1.grid()

    # Part 6: Plots the trajectory lines on the stream plot
    for point in points:
        plot_traj(ax1, func1, func2, point, t)
    return ax1
```

0.1.16 Plot Phase Portrait of the Non Linear Dynamical System

```
In [116]: points = np.array([[-2,2],[-1,-1],[0.5,0.4],[1, -1]])
          phasePortrait(dx1_dt, dx2_dt, points)
```

```
Title?: x1 vs x2 Population
x-axis label?: x1 Population
y-axis label?: x2 Population
```

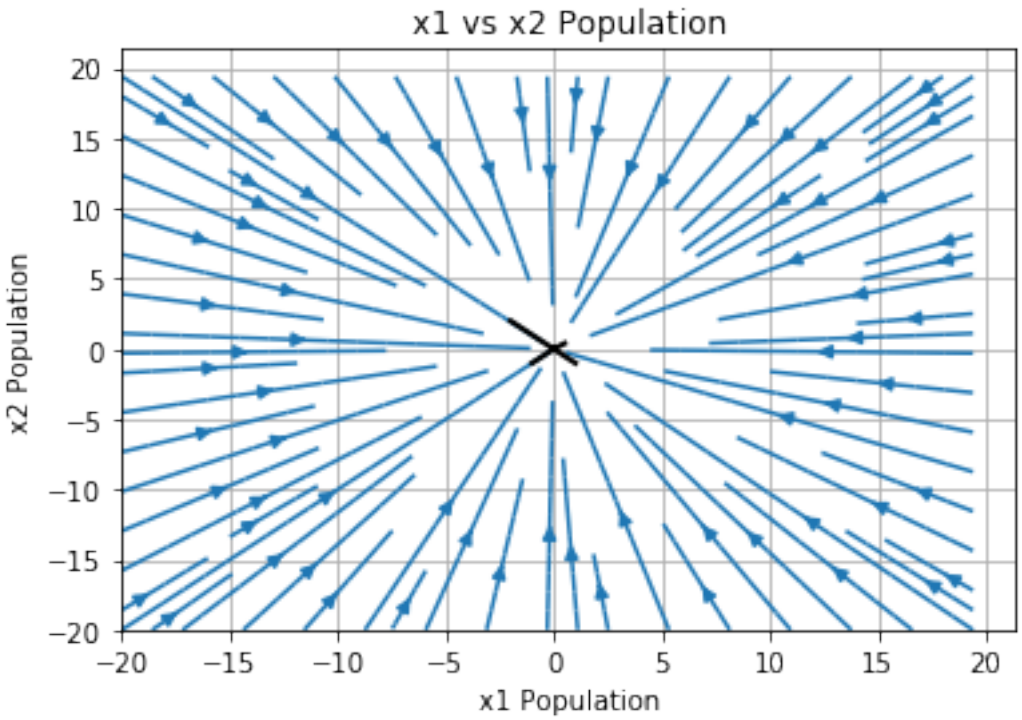
```
Out[116]: <matplotlib.axes._subplots.AxesSubplot at 0x11b73bda0>
```



0.1.17 Plot Phase Portrait of the Linearized Dynamical System

- $x' = F(x) \approx Ax = \lambda x$
- Notice how the following graph is homeomorphic to the one above

```
In [124]: eigenvalues1

Out[124]: array([-3.+3.j, -3.-3.j])

In [111]: dx1_dtL = lambda x1_, x2_: eigenvalues1[0]*x1_
          dx2_dtL = lambda x1_, x2_: eigenvalues1[1]*x2_

In [117]: phasePortrait(dx1_dtL, dx2_dtL, points)

Title?: x1 vs x2 Population
x-axis label?: x1 Population
y-axis label?: x2 Population


Out[117]: <matplotlib.axes._subplots.AxesSubplot at 0x11b8dab70>
```


x1 vs x2 Population
x2 Population
x1 Population
20
15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15
20