Date Created: Feb 7, 2022 @ 20.33 mst

Overview:

This began with the RK4 approximation of the Lorenz Attractor, moved into thinking about different matrix opterations on the RK4 output data, then applying PySINDy to the RK4 data and a subset of the RK4 data. Notes are included throughout the notebook to describe the objective. Changes can be made and new contributions are encouraged.

The notebook is outlined as follows:

Ideas: The next part of this is try computing this on a computer with more power, specifically the one from CU Boulder research computing.

Part 1: Approximating the System of ODEs

Summary: 3D model of the Lorenz ODE system approximated using the 4th-order Runge-Kutta method.

In [1]:
```
import matplotlib.animation as animation
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from numpy import linalg as LA # LA is a standard convention for linalg

%matplotlib notebook
```

Defining the variables needed for the fourth order Runge Kutta method.

In [2]:
```
x = input('choose a starting value for x: ')  # choose initial condition, x
y = input('choose a starting value for y: ')  # choose initial condition, y
```

```
z = input('choose a starting value for z: ')  # choose initial condition, z

t0 = 0.
tf = 100.
dt = 0.01  # time step sufficiently small
tmax = 100.
vector_t = np.arange(t0, tmax, dt) # time scale
n = len(vector_t)

rho = 28.  # check wiki page for info of rho depencancy/interesting values
sigma = 10.
beta = 8./3.


k1 = np.array([0., 0., 0.]) # initialize
k2 = np.array([0., 0., 0.])
k3 = np.array([0., 0., 0.])
k4 = np.array([0., 0., 0.])
```

```
choose a starting value for x: 1
choose a starting value for y: 1
choose a starting value for z: 1
```

Defining the three equations of the Lorenz Attractor.

In [3]:
```
def equation_1(t, x, y, z):
    return sigma*(y - x)


def equation_2(t, x, y, z):
    return rho*x - y - x*z


def equation_3(t, x, y, z):
    return x*y - beta*z
```

A function is created with the 4th-order Runge Kutta method to approximate the ODE.

The following are the variables that need to be passed through the function:

- t : Time
- x, y, z : 3-dim coordinates
  - the initial conditions are defined as x0, y0, z0 and stored in these
  - each iteration stores the new value in these variables
  - all three are stored in an array
  - the function returns this array at each iteration
    - i.e. RK4(t, ... , dt) returns a 3 by 1 array
- equation_1, ... , equation_3 : Equations in the system of ODE's to be solved
- dt : Time step
  - The spatial grid is defined above

The function returns a 3 by 1 array with x, y, z components at each step in time, t.

In [4]:
```
def RK4(t, x, y, z, equation_1, equation_2, equation_3, dt):
```

```
    k1[0] = dt*equation_1(t, x, y, z)
    k1[1] = dt*equation_2(t, x, y, z)
    k1[2] = dt*equation_3(t, x, y, z)

    k2[0] = dt*equation_1(t + dt/2., x + k1[0]/2., y + k1[1]/2., z + k1[2]/2.)
    k2[1] = dt*equation_2(t + dt/2., x + k1[0]/2., y + k1[1]/2., z + k1[2]/2.)
    k2[2] = dt*equation_3(t + dt/2., x + k1[0]/2., y + k1[1]/2., z + k1[2]/2.)

    k3[0] = dt*equation_1(t + dt/2., x + k2[0]/2., y + k2[1]/2., z + k2[2]/2.)
    k3[1] = dt*equation_2(t + dt/2., x + k2[0]/2., y + k2[1]/2., z + k2[2]/2.)
    k3[2] = dt*equation_3(t + dt/2., x + k2[0]/2., y + k2[1]/2., z + k2[2]/2.)

    k4[0] = dt*equation_1(t + dt, x + k3[0], y + k3[1], z + k3[2])
    k4[1] = dt*equation_2(t + dt, x + k3[0], y + k3[1], z + k3[2])
    k4[2] = dt*equation_3(t + dt, x + k3[0], y + k3[1], z + k3[2])

    x = x + (1./6.)*(k1[0] + 2.*k2[0] + 2.*k3[0] + k4[0])
    y = y + (1./6.)*(k1[1] + 2.*k2[1] + 2.*k3[1] + k4[1])
    z = z + (1./6.)*(k1[2] + 2.*k2[2] + 2.*k3[2] + k4[2])

    return np.array([x, y, z])
```

Collecting Data

Three lists are created to store the x, y, and z values at each iteration, and a loop is used to call the function at each time step, dt, on the defined time domain. So 'rk4_data' can be called later and it will have a long list of entries, where earch entry is an array with an x, y, z stored from each iteration.

In [5]:
```
rk4_data = np.zeros((n, 3)) # setting initial conditions
rk4_data[0, 0] = x
rk4_data[0, 1] = y
rk4_data[0, 2] = z


for i in range(n - 1): # loop the stages over i
    rk4_data[i+1, :] = RK4(vector_t[i], rk4_data[i, 0], rk4_data[i, 1],
                           rk4_data[i, 2], equation_1, equation_2, equation_3, dt)
```

Equilibrium Points

Summary: Implementation of Newton's method to find the root of Chen's Equations.

- The nonlinear equations used are Chen's Equation
- The partial derivatives were calculated by hand and stored in the Jacobian matrix
- Both equations are stored in the vector function as an array.
- x0, x1, x2 are used to store the users input of the initial condition to be called on at the end in the printed result.
- x is the initial condition used by the algorithm which stores x0, x1, and x2 in an array.
- The two variables in the nonlinear equations are stored in x after each iteration.
- Count stores the number of iterations.
- The max counter is put in for the case that the initial guess does not result in a solution.
- Epsilon is defined as machine epsilon.
- fval is the value of y or f(x).
- Fnorm is the 2 norm calculated by NumPy

Note: since x,y values are stored in an array called x, x[0]=x, x[1]=y, x[2]=z values

In [6]:
```
rk4_data
```

Out[6]:
```
array([[ 1.        ,  1.        ,  1.        ],
       [ 1.01256719,  1.2599178 ,  0.98489097],
       [ 1.04882371,  1.52399713,  0.97311422],
       ...,
       [-1.41955619, -2.53405237, 11.48240346],
       [-1.53606646, -2.75357769, 11.2187938 ],
       [-1.66343252, -2.99519337, 10.96893939]])
```

In [7]:
```
print(rk4_data[0,:])
```

```
[1. 1. 1.]
```

In [8]:
```
print(rk4_data[:10, :])
```

```
[[1.         1.         1.        ]
 [1.01256719 1.2599178  0.98489097]
 [1.04882371 1.52399713 0.97311422]
 [1.10720885 1.79830989 0.96515895]
 [1.18686802 2.08854014 0.96173722]
 [1.28755706 2.40015446 0.96380606]
 [1.40957066 2.73854561 0.97260817]
 [1.55369006 3.10915397 0.98973112]
 [1.72114638 3.51756946 1.01718652]
 [1.9135962  3.96961501 1.05751186]]
```

In [9]:
```
st = np.array([int(rk4_data[0, 0]), int(rk4_data[0, 1]), int(rk4_data[0, 2])])


print('These are the initial conditions [x y z] = ', st)
```

```
These are the initial conditions [x y z] =  [1 1 1]
```

In [10]:
```
def Vect_Function(st):  # the vector function stores f and g
    return np.array([sigma*(st[1] - st[0]),
                     rho*st[0] - st[1] - st[0]*st[2],
                     st[0]*st[1] - beta*st[2]])


def Jacobian(st):  # partial derivatives calculated/added by hand
    return np.array([[-sigma, sigma, 0],
                     [rho - st[2], -1., st[0]],
                     [st[1], st[0], -beta ] ])
```

In [11]:
```
def Newton_SystEq(Vect_Function, Jacobian, st, epsilon):

    fval = Vect_Function(st)
    Fnorm = LA.norm(fval, ord=2)  # L2 norm by numpy
    iteration_count = 0

    # loop until epsilon or max iterations is reached
    while abs(Fnorm) > epsilon and iteration_count < 100:
        term = LA.solve(Jacobian(st), -fval)
        st = st + term
        fval = Vect_Function(st)
        Fnorm = LA.norm(fval, ord=2)
        iteration_count += 1  # add one iteration to the counter

        # not necessary to print but fun to look at for small approx.
        print('We get ', fval, 'after ', iteration_count, 'iterations')
```

```
    if abs(Fnorm) > epsilon:
        iteration_count = -1  # go back one iteration once epsilon is passed

    return st, iteration_count  # x output is an 1x2 array, count is an int


st, iteration_count = Newton_SystEq(Vect_Function, Jacobian, st, epsilon=1.0e-6)
```

```
We get  [1.11022302e-15 1.40514455e+00 8.99838414e-01] after  1 iterations
We get  [-6.93889390e-17 -1.70303398e-02  2.70739133e-03] after  2 iterations
We get  [-1.08420217e-18  6.40149699e-07  3.97849752e-07] after  3 iterations
```

In [12]:
```
# results
print('\nThe root value is ', st)
print('\nThe number of iterations to reach a solution is ', iteration_count,
      ' when initial conditions x and y are ', rk4_data[0, 0], ',', rk4_data[0, 1], ' and
```

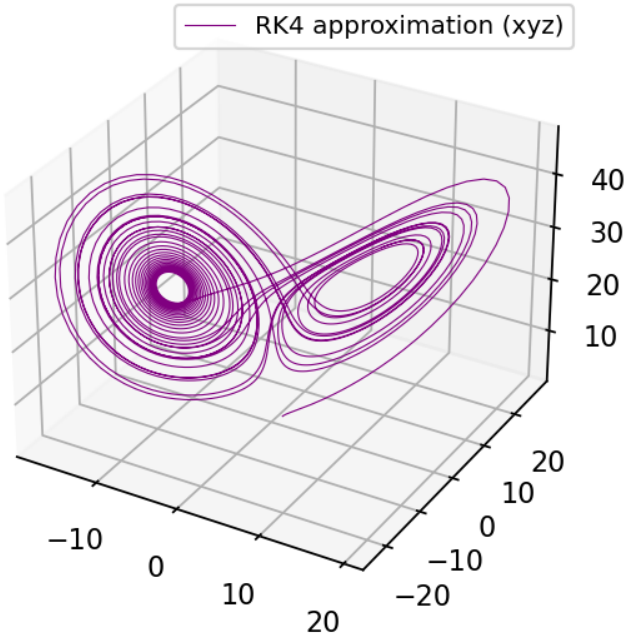
```
The root value is  [ 2.37092480e-08  2.37092480e-08 -1.49193657e-07]

The number of iterations to reach a solution is  3  when initial conditions x and y are
1.0 , 1.0  and  1.0
```

Animation

This will run a short animation to give a visual of the Lorenz Attractor. The time this simulation runs has been restricted to 3,000 frames with 5ms interval with no repeat. The reason why this has been restricted is to lower run time and the attractor is mostly formed within that time.

The purple plot is the Runge-Kutta approximation in three variables and the green plot is in two variables.

In [41]:
```
fig = plt.figure('Lorenz System of ODEs', figsize=(5, 4), dpi=100)
ax = fig.gca(projection='3d')


def animate(i): # A function to call the animation
    ax.clear()
    ax.set(facecolor='white')
    ax.plot(rk4_data[:i, 0], rk4_data[:i, 1], rk4_data[:i, 2],
            color='purple', lw=0.5, label='RK4 approximation (xyz)') # plots x,y,z
    # ax.plot(rk4_data[:i, 0], rk4_data[:i, 1],
            # color='green', lw=0.5, label='RK4 approximation (xy)') # plots x,y
    ax.legend(fontsize=9)


ani = animation.FuncAnimation(
    fig, animate, np.arange(3000), interval=5, repeat=False) # anim object; 3,000 frames,
plt.show()
```




```
/tmp/ipykernel_34897/2172200437.py:2: MatplotlibDeprecationWarning: Calling gca() with key
word arguments was deprecated in Matplotlib 3.4. Starting two minor releases later, gca()
will take no keyword arguments. The gca() function should only be used to get the current
axes, or if no axes exist, create new axes with default keyword arguments. To create a new
axes with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```

Part 3.1: Hankel Matrix

SciPy is used to create the Hankel matrix from the RK4 data that is stored in the variable 'rk4_data'. Refer to the link in the references section for documentation on the Hankel function used in the SciPy library.

NOTE: The Hankel matrix might be wrong here but it is a matrix to do computations on until it can be fixed. It does not seem right but I moved on to the next part and included it anyways.

In [14]:
```
# import hankel function from SciPy
from scipy.linalg import hankel
```

In [15]:
```
rk4_data # check that the data is stored in 'rk4_data'
```

Out[15]:
```
array([[ 1.        ,  1.        ,  1.        ],
       [ 1.01256719,  1.2599178 ,  0.98489097],
       [ 1.04882371,  1.52399713,  0.97311422],
       ...,
       [-1.41955619, -2.53405237, 11.48240346],
       [-1.53606646, -2.75357769, 11.2187938 ],
       [-1.66343252, -2.99519337, 10.96893939]])
```

NOTE: The interval is sliced very small to the 50th iteration since my computer took a long time to compute the matrix up to 500 iterations. Notice that the length of both matrices for only 50 iterations is printed in the following cells and is only 150 elements long. At 500 iterations, the length of both matrices is 3,000 and the

length of both matrices when considering all iterations is 30,000 elements. This would make a 30,000 by 30,000 matrix to compute on.

Ideas:

- Fix the Hankel matrix
- Compute this on a computer with more processor/GPU cores
- Use a different method
- ...

In [16]:
```
hank = hankel(rk4_data[:50,:]) # python makes slicing the interval easy


print('This is the Hankel matrix computed by SciPy:\n', hank)
print('\nA check of the length of the Hankel matrix: ', len(hank), ' elements')
```

```
This is the Hankel matrix computed by SciPy:
 [[ 1.          1.          1.         ...  2.25793267 -8.86967641
  33.48646646]
 [ 1.          1.          1.01256719 ... -8.86967641 33.48646646
   0.        ]
 [ 1.          1.01256719  1.2599178  ... 33.48646646  0.
   0.        ]
 ...
 [ 2.25793267 -8.86967641 33.48646646 ...  0.          0.
   0.        ]
 [-8.86967641 33.48646646  0.         ...  0.          0.
   0.        ]
 [33.48646646  0.          0.         ...  0.          0.
   0.        ]]

A check of the length of the Hankel matrix:  150  elements
```

The dot product of the matrix stored in the last variable is made square and stored in the variable hankel_matrix by taking the dot product of it and the transpose, then the matrix is printed and the length is checked again.

In [17]:
```
hankel_matrix = hank @ hank.T # dot product between Hankel and Hankel.transpose


print('This is the square Hankel matrix:\n', hankel_matrix)
print('\nA check of the length of the Hankel matrix: ', len(hankel_matrix), ' elements')
```

```
This is the square Hankel matrix:
 [[ 4.99183753e+04  2.50591002e+04  2.49167447e+04 ...  2.68747227e+01
   2.46167900e+01  3.34864665e+01]
 [ 2.50591002e+04  4.99173753e+04  2.50581002e+04 ...  2.72955535e+01
   2.46167900e+01  3.34864665e+01]
 [ 2.49167447e+04  2.50581002e+04  4.99163753e+04 ...  3.54669845e+01
   2.50376209e+01  3.34864665e+01]
 ...
 [ 2.68747227e+01  2.72955535e+01  3.54669845e+01 ...  1.20511286e+03
  -3.17041254e+02  7.56101868e+01]
 [ 2.46167900e+01  2.46167900e+01  2.50376209e+01 ... -3.17041254e+02
   1.20001460e+03 -2.97014122e+02]
 [ 3.34864665e+01  3.34864665e+01  3.34864665e+01 ...  7.56101868e+01
  -2.97014122e+02  1.12134344e+03]]

A check of the length of the Hankel matrix:  150  elements
```

Part IIIb: LU Decomposition

SciPy is used to compute the Cholesky factor, then two loops created to solve the forward and backward substitutions in the LU decomposition. Refer to the references for documentation on the Cholesky function in the SciPy library.

In [18]:
```
# import chelesky function from SciPy
from scipy.linalg import cholesky
```

SciPy makes this easy. The Cholesky function is used and the square Hankel matrix data is passed, along with a second term that specifies the output should be lower triangular.

In [19]:
```
L = cholesky(hankel_matrix, lower=True)
len_L = len(L)


print(L)
```

```
[[ 2.23424205e+02  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 1.12159290e+02  1.93229576e+02  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 1.11522137e+02  6.49479068e+01  1.82375869e+02 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 ...
 [ 1.20285637e-01  7.14404194e-02  9.54762689e-02 ...  2.32200335e+01
   0.00000000e+00  0.00000000e+00]
 [ 1.10179602e-01  6.34433110e-02  4.73179152e-02 ... -7.30141249e+00
   2.28872981e+01  0.00000000e+00]
 [ 1.49878419e-01  8.63025724e-02  6.12281305e-02 ...  3.89956962e+00
  -6.16698901e+00  2.20368522e+01]]
```

Create a function for the forward and backward substitution.

In [20]:
```
def solveLU(L, U, b):
    L = np.array(L, float)
    U = np.array(U, float)
    b = np.array(b, float)
    n = len(L)
    y = np.zeros(n)
    x = np.zeros(n)

    for i in range(n-1):
        sumj = 0
        for j in range(i):
            sumj += L[i, j]*y[j]
        y[i] = (b[i]-sumj)/L[i, i]

    for i in range(n-1, -1, -1):
        sumj = 0
        for j in range(i+1, n):
            sumj += U[i, j] * x[j]
        x[i] = (y[i]-sumj)/U[i, i]

    return x
```

Create a function to loop and solve.

In [21]:
```
def run_solver(n):
```

```
    for k in range(n+1):
        L = cholesky(hankel_matrix, lower=True)
        LT = np.transpose(L)
        b = np.dot(hankel_matrix, np.ones(len_L))

        x = solveLU(L, LT, b)

    return x
```

Call and print the solver, which will output the solution of the LU decomp for the sliced interval of x, y, and z values from interation 0 to 50.

In [22]:
```
print('This is the solution:\n', run_solver(i))
```

```
This is the solution:
 [0.98801284 0.98376328 0.98609366 1.00168428 1.00616456 1.0061438
 1.00680205 1.00738955 1.00656719 1.00672281 1.00729545 1.00632403
 1.00593889 1.00641594 1.00542839 1.00452432 1.00484095 1.00397868
 1.00264012 1.00273606 1.00212305 1.00048964 1.00031888 1.00004334
 0.99829702 0.99783768 0.99793839 0.99628566 0.99554892 0.99600834
 0.99465904 0.99369571 0.99444121 0.99358546 0.99248871 0.99340224
 0.9931867  0.99208901 0.99302424 0.99352954 0.99259193 0.9933966
 0.99461835 0.99401071 0.99455076 0.99638812 0.99626166 0.99644324
 0.99869968 0.99915566 0.99894114 1.00134119 1.00240297 1.0018183
 1.0040405  1.00563665 1.00476889 1.00649071 1.00845468 1.00744
 1.0083867  1.01047433 1.00947814 1.00946606 1.01138792 1.01058002
 1.00954635 1.01100857 1.0105369  1.0085516  1.00929822 1.00926485
 1.00652445 1.00637517 1.00681759 1.0036243  1.00250347 1.00338326
 1.00011408 0.99806917 0.99926901 0.99633928 0.99354831 0.99487826
 0.99270123 0.98946871 0.99068357 0.98962449 0.98636307 0.98719394
 0.98751555 0.98470802 0.98491053 0.98671088 0.98484632 0.98426305
 0.98741711 0.98690113 0.98552513 0.98965549 0.99070878 0.98872344
 0.99323108 0.99580798 0.99357482 0.99774612 1.00151137 0.99949223
 1.00266198 1.00704812 1.00567926 1.00739218 1.01172436 1.01128945
 1.01139499 1.01503751 1.01559124 1.01423888 1.01671131 1.01808008
 1.01563294 1.01666683 1.01851518 1.01543261 1.01497242 1.01689926
 1.01363634 1.01180763 1.01343632 1.01038262 1.00745048 1.00849226
 1.0059465  1.00227578 1.00256324 1.0007289  0.99674685 0.99624343
 0.99523295 0.99138915 0.99015616 0.98983817 0.98608422 0.98498483
 1.00314946 1.11324972 1.46870177 1.08321271 0.73054971 0.        ]
```

Part IV: Poincaré Map

Four lists are created. Three lists are used to store each x, y, and z values, and a fourth is created to store the iteration.

The goal is to restrict one of the three dimensions of the RK4 output data to get a 2D system. This is done of two variables, first y then the x values. A interval is defined on each of the two restricted variables and the any point that falls within that tolernce is stored in the the list above.

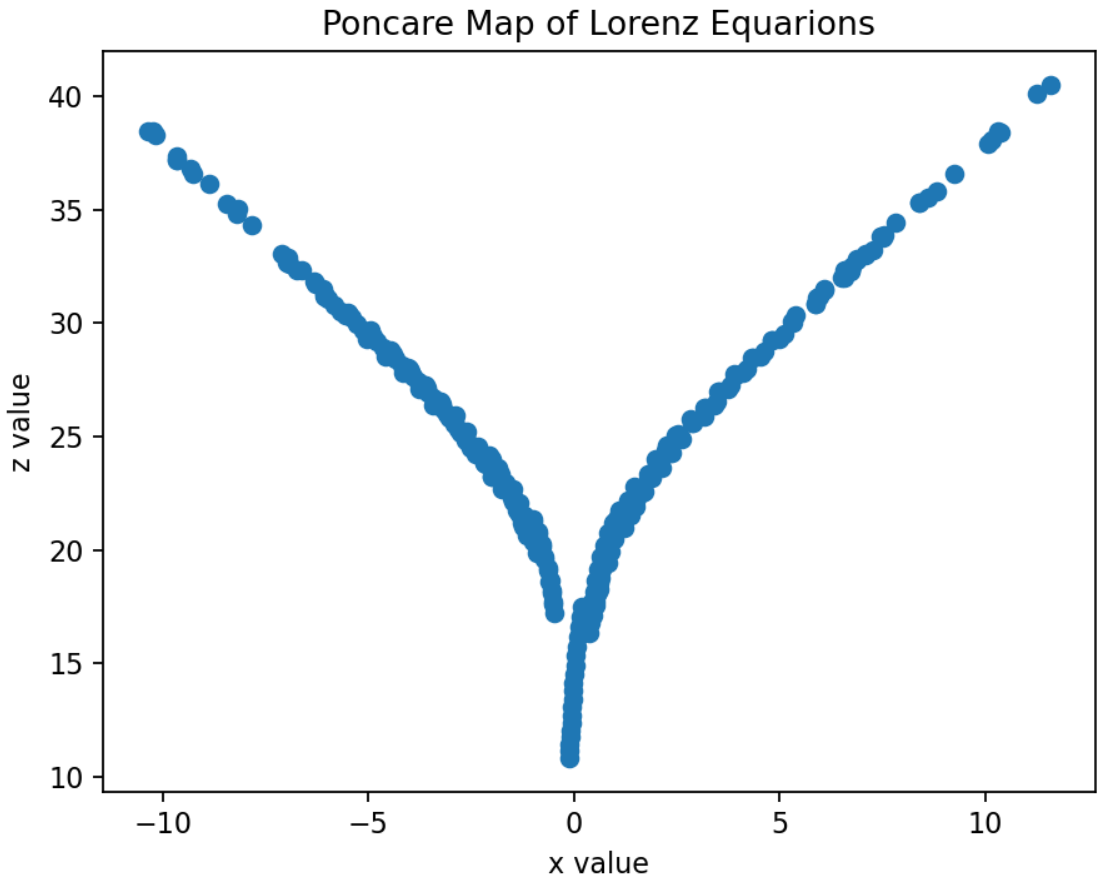
Two functions are used to call these loops through the data stored in 'rk4_data'.

In [23]:
```
# four lists initialized
poinc_x = []
poinc_y = []
```

```
poinc_z = []
iteration_count = []
```

In [24]:

```
def center_poinc(i):

    for i in range(0, n-1, 1):
        if rk4_data[i, 1] < 0.25 and rk4_data[i, 1] > -0.25:
            poinc_x.append(float(rk4_data[i, 0])) # adds elements to lists
            poinc_y.append(float(rk4_data[i, 1]))
            poinc_z.append(float(rk4_data[i, 2]))
            iteration_count.append(float(i))

    return np.array([poinc_x, poinc_y, poinc_z, iteration_count])


p = center_poinc(i) # call function, define as p


plt.figure() # 2D plot
plt.title("Poncare Map of Lorenz Equarions")
plt.ylabel('z value')
plt.xlabel('x value')
plt.scatter(p[0], p[2])
plt.show()
```



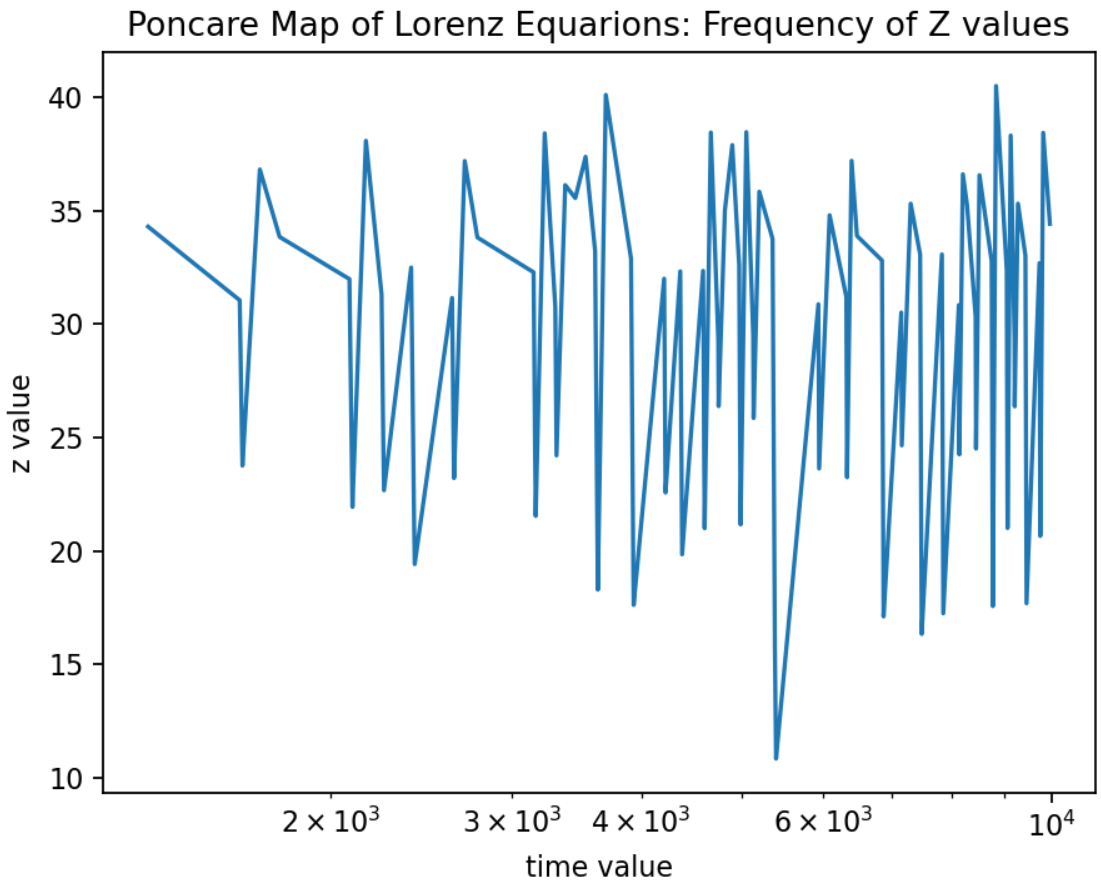
Another 2 dimensional semi-log plot is created to compare the z value to the time it occured. Notice that the frequency the attractor passes through the plane increases as time goes on.

NOTE: I am not sure about this. I have very little experience with these graphs.

Ideas:

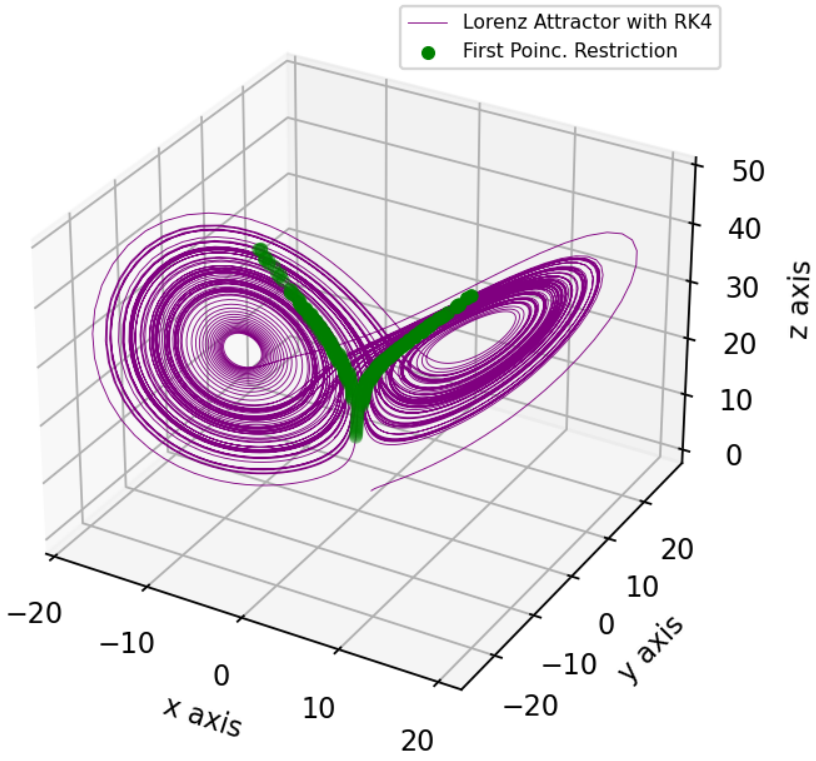
- Check this graph
- Add more graphs
- ...

In [25]:
```
# semilog plot
plt.figure()
plt.title("Poncare Map of Lorenz Equarions: Frequency of Z values")
plt.xlabel('time value')
plt.ylabel('z value')
plt.semilogx(p[3], p[2])
plt.show()
```



A three-dimensional graph is created to show the slices position along the attractor.

In [26]:
```
plt.figure()
ax2 = plt.axes(projection='3d')
ax2.clear()
ax2.set(facecolor='white')
ax2.set_xlabel('x axis')
ax2.set_ylabel('y axis')
ax2.set_zlabel('z axis')
ax2.plot3D(rk4_data[:i,0], rk4_data[:i,1], rk4_data[:i,2], 'purple', lw=0.4, label='Loren
ax2.scatter3D(p[0], p[1], p[2], color='green', lw=0.5, label='First Poinc. Restriction')
ax2.legend(fontsize=7)
plt.show()
```

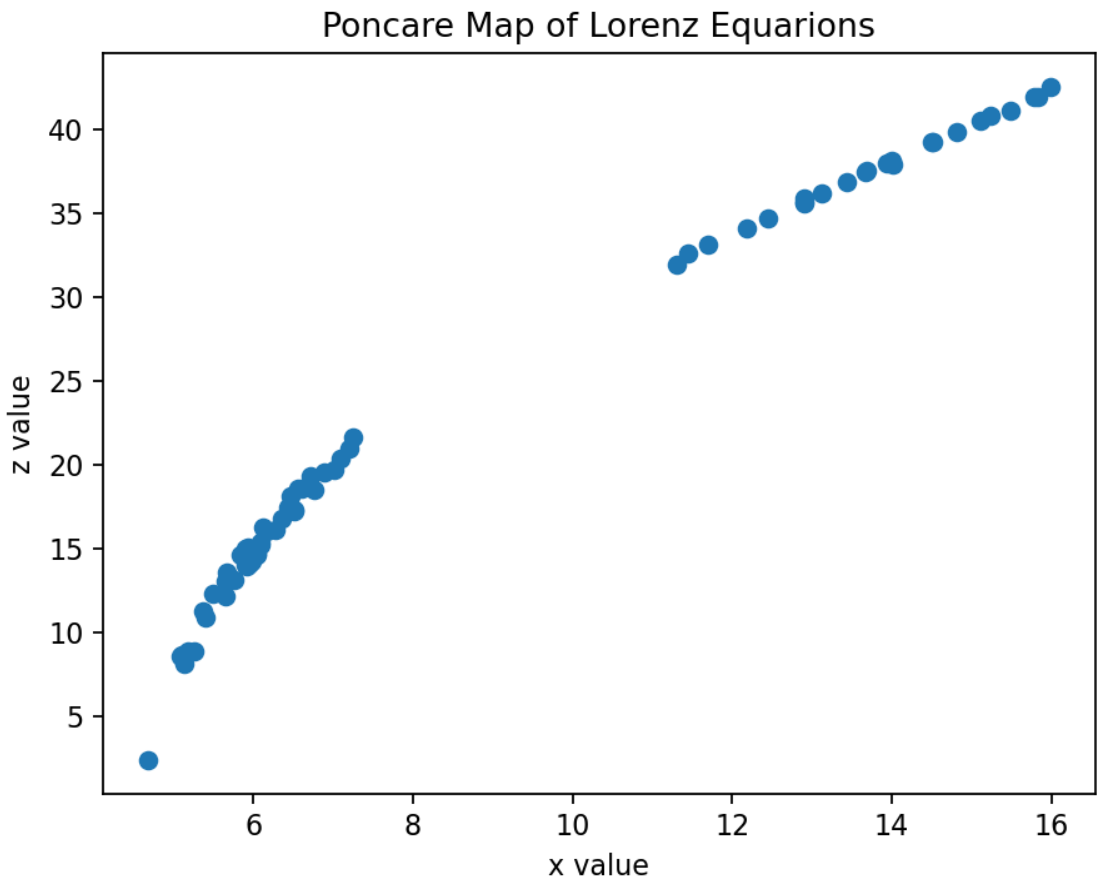


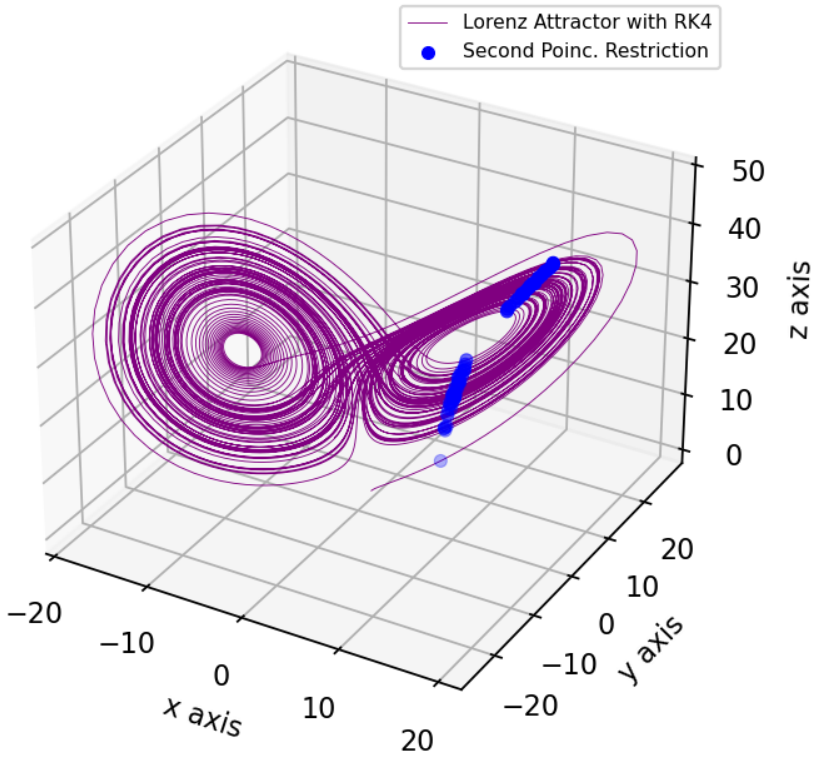
Clear out the lists and then do the loop again but for another interval.

In [27]:
```
poinc_x = []
poinc_y = []
poinc_z = []
iteration_count = []
```

In [28]:
```
def side_poinc(i):
    for i in range(0, n-1, 1):
        if rk4_data[i, 1] < 10.25 and rk4_data[i, 1] > 9.75:
            poinc_x.append(float(rk4_data[i, 0]))
            poinc_y.append(float(rk4_data[i, 1]))
            poinc_z.append(float(rk4_data[i, 2]))
            iteration_count.append(float(i))

    return np.array([poinc_x, poinc_y, poinc_z])


p = side_poinc(i)


plt.figure() # 2D plot
plt.title("Poncare Map of Lorenz Equarions")
plt.ylabel('z value')
plt.xlabel('x value')
plt.scatter(p[0], p[2])
plt.show()

plt.figure() # 3D plot
ax2 = plt.axes(projection='3d')
ax2.clear()
```

```
ax2.set(facecolor='white')
ax2.set_xlabel('x axis')
ax2.set_ylabel('y axis')
ax2.set_zlabel('z axis')
ax2.plot3D(rk4_data[:i,0], rk4_data[:i,1], rk4_data[:i,2], 'purple', lw=0.4, label='Lorenz
ax2.scatter3D(p[0], p[1], p[2], color='blue', lw=0.5, label='Second Poinc. Restriction')
ax2.legend(fontsize=7)
plt.show()
```





Each time the slice was crossed by the attractor, the iteration is recorded in the list 'iteration_count'. The distance between each crossing can be computed with the following loop. The output of this loop is a list of the time difference between each point in the blue scatter plot on the last attractor.

In [29]:
```
# Time diffference between each step
# could be used to find the avg. between entire stream
n = len(iteration_count[:i])
time_dif_list = []

for i in range(0, n, 1):
    time_dif = abs(iteration_count[i-1] - iteration_count[i])
    time_dif_list.append(time_dif)


time_dif_list
```

Out[29]:
```
[9908.0,
 1377.0,
 360.0,
 299.0,
 22.0,
 83.0,
 314.0,
 272.0,
 161.0,
 67.0,
 69.0,
 72.0,
 464.0,
 22.0,
 92.0,
 345.0,
```

```
22.0,
47.0,
22.0,
50.0,
22.0,
82.0,
278.0,
107.0,
207.0,
226.0,
22.0,
210.0,
172.0,
21.0,
43.0,
22.0,
43.0,
22.0,
44.0,
66.0,
69.0,
22.0,
130.0,
185.0,
272.0,
160.0,
22.0,
45.0,
70.0,
22.0,
52.0,
94.0,
360.0,
22.0,
132.0,
22.0,
75.0,
501.0,
22.0,
129.0,
22.0,
138.0,
226.0,
154.0,
93.0,
361.0,
77.0,
22.0,
132.0,
22.0,
71.0,
361.0,
76.0]
```

The list of time differences can be summed with another simple loop then divided by the number of differences to find the average time difference. The output is a float and should be rounded to some integer. It is left as a float for now because of round-off errors for whatever it might be used in.

In [30]:
```
n_time_dif_list = len(time_dif_list)
sum_time_dif = 0

for i in range(n_time_dif_list):
    sum_time_dif += time_dif_list[i]
    avg_time_dif = sum_time_dif/n_time_dif_list
```

In [31]:
```
print('The average amount of iterations between each point is: ', avg_time_dif, ' iteratic
```

```
The average amount of iterations between each point is:  287.18840579710144  iterations
```

Part V: PySINDy

The idea for this next part is to use the output data from the 4th-order Runge Kutta approximation in part 1 with the PySINDY library to get the system of ODE's that created the chaotic data.

From there, the restricted data created by the Poincaré map will be used to try to do the same. Remember that a interval was used to collect the points on the restricted element. If the tolerence is chosen too small, i.e. just one value and not an interval, then too few points are collected. Too large and it accounts for all of the data.

This section will show that PySINDy works great on all of the data but does not show anything helpful when one of the elements are restricted to most intervals.

Ideas:

- If the RK4 simulation of the Lorenz Atractor is run for a longer time, can enough data be recorded so that PySINDy to obtain the Lorenz equations, even as the interval of the restriction on one of the three variables is decreased.

In [32]:
```
# import PySINDy library
# Look in the references for documentation and installation
import pysindy as ps
```

In [33]:
```
rk4_data # check that the RK4 approximation data is stored properly
```

Out[33]:
```
array([[ 1.        ,  1.        ,  1.        ],
       [ 1.01256719,  1.2599178 ,  0.98489097],
       [ 1.04882371,  1.52399713,  0.97311422],
       ...,
       [-1.41955619, -2.53405237, 11.48240346],
       [-1.53606646, -2.75357769, 11.2187938 ],
       [-1.66343252, -2.99519337, 10.96893939]])
```

PySINDy used on all of the RK4 data.

This is the standard process given in the PySINDy tutorials on YouTube. Check the link in the references section for this video and read the comments next to the code for code description. We see that PySINDy has no problem finding the equations when the list of all RK4 data is imported.

In [34]:
```
feature_names = ['x', 'y', 'z']
opt = ps.STLSQ(threshold=0.1) # this is lambda
model = ps.SINDy(feature_names=feature_names, optimizer=opt)
model.fit(rk4_data, t=dt) # input all data from 'rk4_data'
model.print() # print the PySINDy model
```

```
x' = -9.978 x + 9.978 y
y' = 27.802 x + -0.961 y + -0.994 x z
z' = -2.659 z + 0.997 x y
```

PySINDy used on the restricted data.

PySINDy is used again with the restricted data from above. The function used above is changed so that 0's are added to the lists for each value that is not within the interval on the plane.

In [35]:
```
# NOTE: Here is the restriction on one element
plane_value = 10 # chosen 'plane'
threshold = 1 # size of the interval/tolerance
upper_bound = plane_value + threshold/2 # center the interval on the plane value
lower_bound = plane_value - threshold/2


n = len(vector_t) # check val


poinc_x_sparse = [] # create new lists
poinc_y_sparse = []
poinc_z_sparse = []
```

In [36]:
```
def collect_data(i):

    for i in range(0, n-1, 1):
        if rk4_data[i, 0] < upper_bound and rk4_data[i, 0] > lower_bound:
            poinc_x_sparse.append(float(rk4_data[i, 0]))
            poinc_y_sparse.append(float(rk4_data[i, 1]))
            poinc_z_sparse.append(float(rk4_data[i, 2]))
        else:
            poinc_x_sparse.append(0) # adding the zeros
            poinc_y_sparse.append(0)
            poinc_z_sparse.append(0)

    return np.array([poinc_x_sparse, poinc_y_sparse, poinc_z_sparse])


p = collect_data(i)
```

In [37]:
```
poinc_sparse = p.T # take the transpose of the data to flip it


print('The sparse data:\n', poinc_sparse, '\nNOTE: some of the entries should be non-zero.
```

```
The sparse data:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 ...
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
NOTE: some of the entries should be non-zero.
```

Use the same commands as above and PySINDy gives a lot of extra terms. There is not a lot of data collected over the chosen time interval when x is restricted to 9.5<x<10.5.

In [38]:
```
feature_names = ['x', 'y', 'z']
opt = ps.STLSQ(threshold=0.1) # this is lambda
model = ps.SINDy(feature_names=feature_names, optimizer=opt)
model.fit(poinc_sparse, t=dt)
model.print()
```

```
x' = -7386.089 x + 4698.710 y + -332.523 z + -1424.907 x^2 + 709.307 x y + 807.160 x z + -
105.261 y^2 + -242.731 y z + -53.604 z^2
```

```
y' = -5064.820 x + 4044.919 y + -370.077 z + -1183.361 x^2 + 516.059 x y + 645.784 x z + -
82.725 y^2 + -190.305 y z + -42.992 z^2
z' = -0.178 1 + -26628.152 x + 14836.263 y + -110.825 z + -4039.702 x^2 + 2175.943 x y + 2
405.407 x z + -327.878 y^2 + -752.824 y z + -165.233 z^2
```

Instead of increasing the time the RK4 is run, the data is restricted to a larger interval, then the interval is decreased to see what happens. It turns out that the interval needs to be very large, possibly all of the data points, for PySINDy to give anything without a lot of extra terms.

In [39]:
```
# NOTE: Here is the restriction on one element
plane_value = 0
threshold = 38 # this might include all terms since the interval is -19<x<19
upper_bound = plane_value + threshold/2
lower_bound = plane_value - threshold/2


n = len(vector_t)


poinc_x_sparse = [] # clear lists
poinc_y_sparse = []
poinc_z_sparse = []
```

In [40]:
```
def recollect_data(i):

    for i in range(0, n-1, 1):
        if rk4_data[i, 0] < upper_bound and rk4_data[i, 0] > lower_bound:
            poinc_x_sparse.append(float(rk4_data[i, 0]))
            poinc_y_sparse.append(float(rk4_data[i, 1]))
            poinc_z_sparse.append(float(rk4_data[i, 2]))
        else:
            poinc_x_sparse.append(0)
            poinc_y_sparse.append(0)
            poinc_z_sparse.append(0)

    return np.array([poinc_x_sparse, poinc_y_sparse, poinc_z_sparse])


p = recollect_data(i)


poinc_sparse = p.T # flip


feature_names = ['x', 'y', 'z']
opt = ps.STLSQ(threshold=0.1) # this is lambda
model = ps.SINDy(feature_names=feature_names, optimizer=opt)
model.fit(poinc_sparse, t=dt) # loading the data
model.print() # print
```

```
x' = -9.897 x + 9.898 y
y' = -0.134 1 + 27.879 x + -1.045 y + -0.995 x z
z' = -0.529 1 + 0.190 x + -0.175 y + -2.608 z + 0.988 x y
```

Result: If the interval is chosen smaller, then PySINDy gives a lot of extra terms.

References:

Controling the Lorenz Equations (manuscript) - http://www-personal.umich.edu/~riboch/publications/riboch-ControlofLorenz.pdf

Control of the Lorenz Chaos by the Exact linearization - https://link.springer.com/content/pdf/10.1007/BF02458982.pdf

R. Hermann and A Krener. Nonlinear controllability and obserability. IEEE Transactions on In Automatic Control, 22(5):728-740, October 1977. https://ieeexplore-ieee-org.colorado.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=1101601

Poincare Maps (overview video) - https://www.youtube.com/watch?v=HbnFQJPwZoI

Poincare Maps (properties) - https://people.math.wisc.edu/~angenent/519.2016s/notes/poincare-map.html

pySINDy - https://github.com/dynamicslab/pysindy

Hankel Matrix - https://en.wikipedia.org/wiki/Hankel_matrix

SciPy Hankel Matrix - https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.hankel.html

---

In [ ]: