import math  
import random  
import time  
import numpy as np  
import matplotlib.pyplot as plt

### Galilean Transformation

Galilean transformation is defined as:

r′ = r + vt  
t′ = t

where r is the position of the particle in an inertial frame (x, y), r′is the transformed position in a new inertial frame (x′, y′), v is a constant in which |v| ≪ c is the linear velocity vector of the new frame, and finally t and t′ are the invariant time variables. The Galilean transformation keeps some physical quantities invariant. Among them, is the distance

r = r2 −r1,

that remains invariant under the transformation in the new frame,

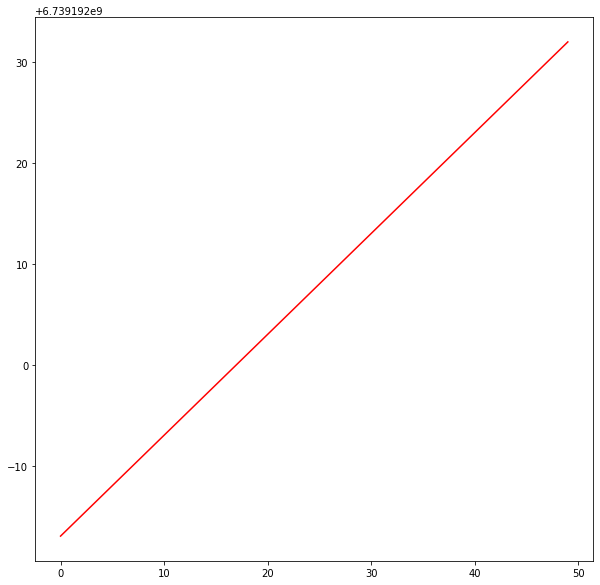
r′ = r′2 − r′1 = r = r2 − r1.

def get\_galilean\_transformation(r, v, t):  
 return r + v \* t

x = [each for each in range(50)]  
y = [get\_galilean\_transformation(each, 4, time.time()) for each in range(50)]

plt.figure(figsize = (10,10))  
plt.plot(x, y, color="red")

[<matplotlib.lines.Line2D at 0x7f8063111730>]



### Rotational Transformation

Rotational transformation is defined as

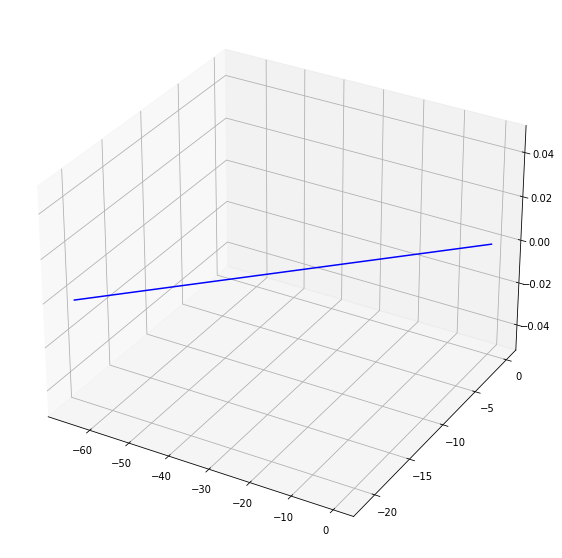
t′ = t where θ is a constant rotation angle around the z–axis, for theframe r = (x, y) to rotate into the frame r′ = (x′, y′). Rotational symmetry transformation also keeps the distance 1r invariant in the new frame. It keeps the position distribution of points unchanged as well.

def get\_rotational\_transformation(x, y, theta):  
 return x \* math.cos(theta) - y \* math.sin(theta), x \* math.sin(theta) + y \* math.cos(theta)

points = [get\_rotational\_transformation(each, each, 2.67) for each in range(50)]  
x0, y0 = zip(\*points)

plt.figure(figsize = (10,10))  
plt.axes(projection='3d')  
plt.plot(x0, y0, color="blue")

[<mpl\_toolkits.mplot3d.art3d.Line3D at 0x7f806309e820>]



### KAA Map

Therefore, the combined symmetry transformation, which is  
r′ = Rr + vt  
t′ = t  
keeps the distance r invariant in the new frame and the distribution of the positions of particles (in this case, pixels) is maintained. In the KAA map model, we introduce the transformation:  
r′ = R(θ(t))r + v(t),  
where the rotation angle θ is made time (step) dependant θ(t) and the linear velocity v is no longer constant but made time (step) dependant v(t), as well. The θ and v are polynomial functions that depend on the parameter t. This transformation destroys completely any symmetry of the group of positions r in the initial frame after being transformed into the new frame. We introduce the KAA map finally as

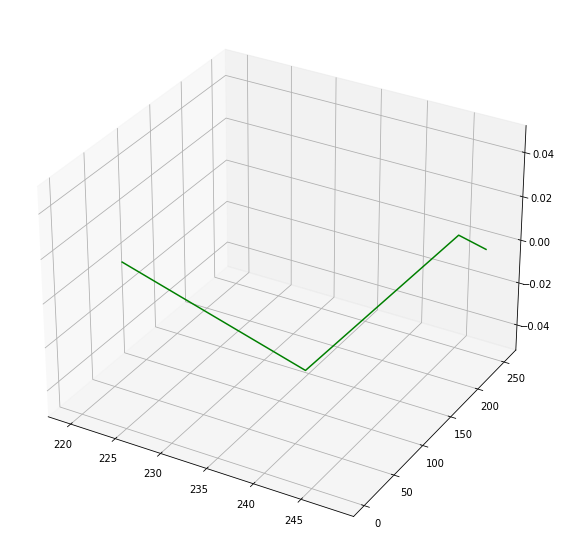
where mod N is used to confine the numbers from 0 to 255 which is the minimum and maximum value a pixel can attain.

def get\_combined\_transformation(x, y, theta, v1, v2):  
 x1 = x \* math.cos(theta) - y \* math.sin(theta) + v1 \* time.time()  
 time.sleep(0.1)  
 y1 = x \* math.sin(theta) + y \* math.cos(theta) + v2 \* time.time()  
 return x1 % 255, y1 % 255

points = [get\_combined\_transformation(each, each, 1.56, 4, 7) for each in range(50)]  
xt, yt = zip(\*points)

plt.figure(figsize = (10,10))  
plt.axes(projection='3d')  
plt.plot(xt, yt, color="green")

[<mpl\_toolkits.mplot3d.art3d.Line3D at 0x7f8063048340>]



### Logistic Map

The Logistic map is a discrete-time analog of the logistic equation for population growing. Mathematically, the Logistic map is defined as,

xi+1 = 4pxi(1 − xi)

where parameter p is within the range of [0, 1]. When p ∈ [0.89, 1], Logistic map is chaotic. Bifurcation diagram shows the output distribution of a chaotic map along its control parameter, while iteration function describes the output distributions along its inputs. It is noticed that the outputs of Logistic map distribute in a larger area when p approaches to 1.

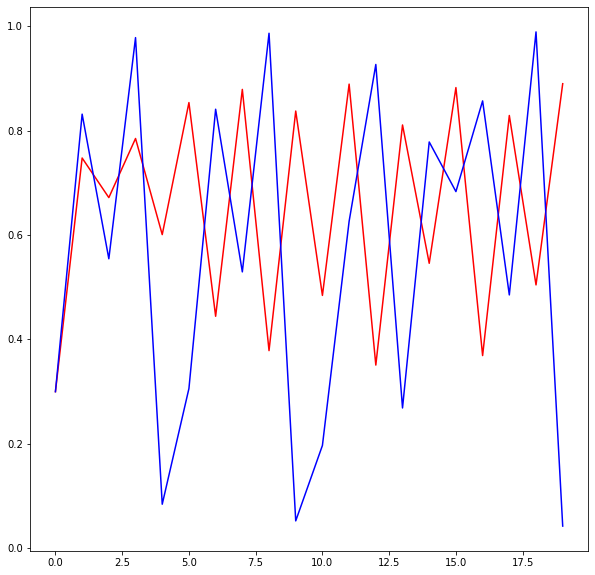
def get\_logistic\_map(current, p, limit):  
 def next\_element(x, p):  
 return 4 \* p \* x \* (1 - x)  
 result = []  
 for i in range(limit):  
 result.append(current)  
 current = next\_element(current, p)  
 return result

def get\_logistic\_map\_plot\_arrays(current, p, limit):  
 return np.array([each for each in range(limit)]), np.array(get\_logistic\_map(current, p, limit))

x1, y1 = get\_logistic\_map\_plot\_arrays(0.3, 0.89, 20)  
x2, y2 = get\_logistic\_map\_plot\_arrays(0.3, 0.99, 20)

plt.figure(figsize = (10,10))  
plt.plot(x1, y1, color="red")  
plt.plot(x2, y2, color="blue")

[<matplotlib.lines.Line2D at 0x7f8062fc62e0>]



### Sine Map

When sine function has inputs within the range of [0, π], its outputs fall into the range of [0, 1]. Sine map is derived from sine function by transforming its inputs into [0, 1]. It is defined as

xi+1 = s sin(πxi)

where parameter s ∈ [0, 1]. Sine map is chaotic when s ∈ [0.87, 1]. Although Logistic and Sine maps have totally different mathematical definitions, their chaotic behaviors are quite similar.

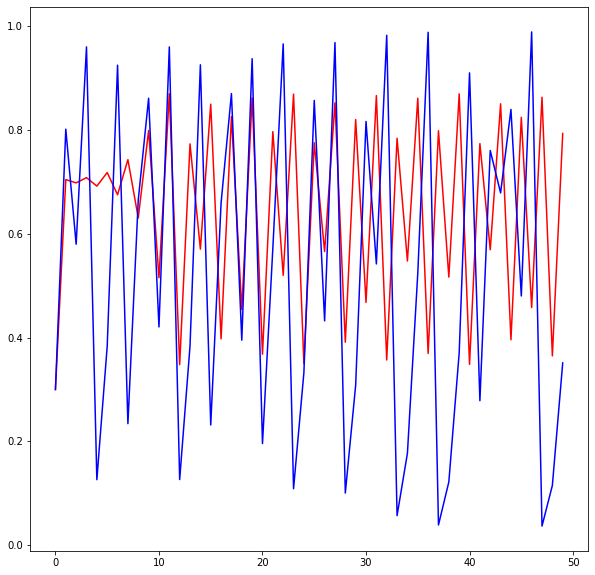
def get\_sine\_map(current, s, limit):  
 def next\_element(x, s):  
 return s \* math.sin(math.pi \* x)  
 result = []  
 for i in range(limit):  
 result.append(current)  
 current = next\_element(current, s)  
 return result

def get\_sine\_map\_plot\_arrays(current, s, limit):  
 return np.array([each for each in range(limit)]), np.array(get\_sine\_map(current, s, limit))

x3, y3 = get\_sine\_map\_plot\_arrays(0.3, 0.87, 50)  
x4, y4 = get\_sine\_map\_plot\_arrays(0.3, 0.99, 50)

plt.figure(figsize = (10,10))  
plt.plot(x3, y3, color="red")  
plt.plot(x4, y4, color="blue")

[<matplotlib.lines.Line2D at 0x7f8062f28dc0>]



### 2D Logistic-adjusted-Sine map

We give the mathematical definition of 2D-LASM,

xi+1 = sin(πμ(yi + 3)xi(1 − xi))  
yi+1 = sin(πμ(xi+1 + 3)yi(1 − yi))

where parameter μ ∈ [0, 1]. 2D-LASM is derived from Sine and Logistic maps. The logistic equation xi(1 − xi) is first scaled by a factor of μ, and fed into the input of Sine map. The phase plane is then extended from 1D to 2D. In 2D-LASM, two inputs are interactively influenced and the output pairs (xi+1, yi+1) distribute into the 2D phase plane.

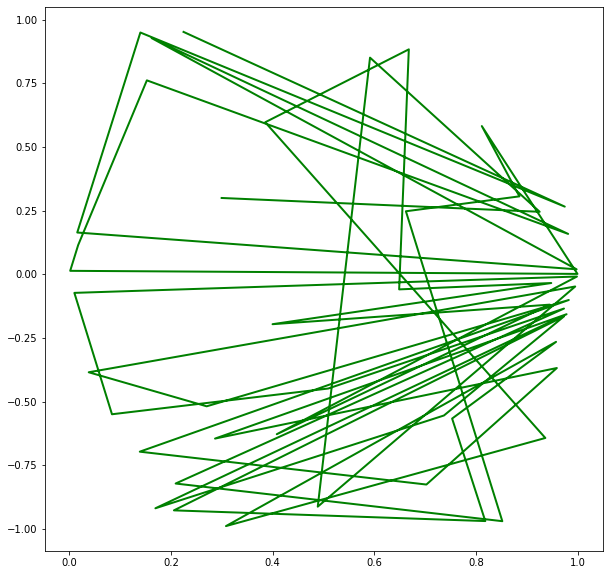
Compared with Sine and Logistic maps, it has a more complicated structure and its outputs are more difficult to be predicted.

def get\_2D\_logistic\_adjusted\_sine\_map(x, y, u, limit):  
 def get\_next\_x(x, y, u):  
 return math.sin(math.pi \* u \* (y + 3) \* x \* (1 - x))  
 def get\_next\_y(x, y, u):  
 return math.sin(math.pi \* u \* (x + 3) \* y \* (1 - x))  
 result\_x = []  
 result\_y = []  
 for i in range(limit):  
 result\_x.append(x)  
 result\_y.append(y)  
 x = get\_next\_x(x, y, u)  
 y = get\_next\_y(x, y, u)  
 return result\_x, result\_y

x5, y5 = get\_2D\_logistic\_adjusted\_sine\_map(0.3, 0.3, 0.9, 50)

plt.figure(figsize = (10,10))  
plt.plot(x5, y5, color="green", linewidth=2.0)

[<matplotlib.lines.Line2D at 0x7f8062d37b20>]



### THE LINEAR CONGRUENTIAL GENERATOR

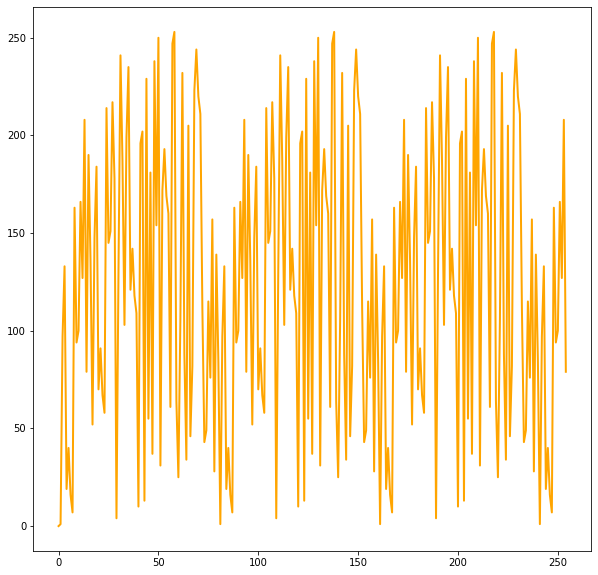
The linear congruential generator (LCG) was first published in 1960 by Thomson and Rotenberg. It is one of the best PRNGs that provide fast software and hardware implementations. This is because a LCG requires minimal memory to retain state. Evidence of the LCG’s statistical superiority as a PRNG is that a 96-bit LCG is compliant with (i.e. passes) the most rigorous BigCrush suite. Here, it is used to generate a sequence of integers from 1 to (m − 1), and is mathematically defined as,  
Xn = (aXn - 1 − 1 + c) mod m  
where a is a control parameter.

def linear\_congruential\_generator(x, a, c, limit):  
 def get\_next\_element(x, a, c, limit):  
 return (a \* x - 1 + c) % limit  
 result = []  
 for i in range(limit):  
 result.append(x)  
 x = get\_next\_element(x, a, c, limit)  
 return result

x6 = [each for each in range(255)]  
y6 = linear\_congruential\_generator(0, 96, 2, 255)

plt.figure(figsize = (10,10))  
plt.plot(x6, y6, color="orange", linewidth=2.0)

[<matplotlib.lines.Line2D at 0x7f8062ca3ee0>]



### THE BERNOULLI MAP

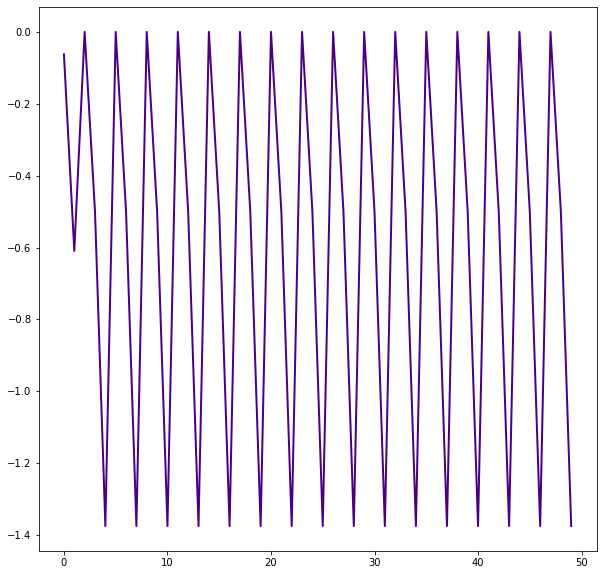
The Bernoulli chaotic map is a one dimensional chaotic map that is defined from the range −A to A. It is one example of strongly chaotic functions that display exponential decay of correlation to their equilibrium values. Not only is it easy to compute, but also provides much desired properties of chaotic behavior needed for image encryption. The Lyapunov exponent of the Bernoulli map is log2. Being an exact system (i.e. mixing and ergodic), after only a small number of iterations, the Bernoulli map when applied with 2 different sets of initial conditions, results in very different trajectories. It is mathematically defined as  
x(n + 1) =  
(Bx(n)) − A, −A <= x <= 0,  
(Bx(n)) + A, 0 <= x <= A,  
where the sequence is generated with the following parameters: A = 0.5, B = 1.75 and x(1) = (B×0.25)−A. Therefore, the range of the generated chaotic system is from −1/2 to 1/2

def get\_bernoulli\_map(x, a, b, limit):  
 def get\_next\_element(x, a, b):  
 if -a <= x <= 0:  
 return (b \* x) - a  
 elif 0 <= x <= a:  
 return (b \* x) + a  
 return 0  
 result = []  
 for i in range(limit):  
 result.append(x)  
 x = get\_next\_element(x, a, b)  
 return result

x7 = [each for each in range(50)]  
y7 = get\_bernoulli\_map((0.25 \* 1.75) - 0.5, 0.5, 1.75, 50)

plt.figure(figsize = (10,10))  
plt.plot(x7, y7, color="indigo", linewidth=2.0)

[<matplotlib.lines.Line2D at 0x7f8062c08e50>]



### THE TENT MAP

The Tent map is also a chaotic one dimensional map that displays a good chaotic sequential behaviour. Similar to the Bernoulli map, it has a Lyapunov exponent of log2, but unlike the Bernoulli map, the x−correlation function for the Tent map is δ−correlated. It is mathematically expressed as  
x(n + 1) =  
C(x(n)), 0 < x < 1/2  
C(1 − x(n)), 1/2 < x < 1  
where the sequence is generated with the parameters C = 1.5 and x(1) = 0.5.

def get\_tent\_map(x, c, limit):  
 def get\_next\_element(x, c):  
 if 0 <= x <= 0.5:  
 return c \* x  
 if 0.5 <= x <= 1:  
 return c \* (1 - x)  
 return x  
 result = []  
 for i in range(limit):  
 result.append(x)  
 x = get\_next\_element(x, c)  
 return result

x8 = [each for each in range(50)]  
y8 = get\_tent\_map(0.5, 1.5, 50)

plt.figure(figsize = (10,10))  
plt.plot(x8, y8, color="purple", linewidth=2.0)

[<matplotlib.lines.Line2D at 0x7f806139ca90>]

