Universidad de Sonora

División de Ciencias Exactas y Naturales

Física computacional

11 de mayo del 2017

Act 8: Efecto mariposa

Alumna: Chávez Gutiérrez Yanneth Tzitzin
Profesor: Carlos Lizárraga Celaya.

1. Breve resumen

En esta practica se analizará el efecto mariposa, de dónde surgió, qué es lo que representa y cómo podemos por medio de comandos en Python y las ecuaciónes diferenciales de Edward Lorenz hacer una interpretación de este efecto.

2. Introducción

En 1963 Edward Norton Lorenz tratando de comprender y simplificar el comportamiento de la atmósfera se dio a la tarea de interpretarlo por medio de ecuaciones, cómo esta a pesar de la complejidad que la representa, podria ser reducido su compartamiento a algunas ecuaciones diferenciales.

Pero al simplificar y simplificar el problema se encontro en un punto en el que el modelo de la atmósfera se redujo a tres simples ecuaciones. La evolucion de una simple particula en ella. La temperatura, la velocidad del viento y la humedad eran variables tomadas en cuenta.

$$\frac{dx}{dt} = a(y - x)$$
$$\frac{dy}{dt} = x(b - z) - y$$
$$\frac{dz}{dt} = xy - cz$$

Si seguimos las trayectorias de dos atmósferas que se encuentran muy cerca la una de la otra, las evoluciones parecieran seguir una misma ruta, pero en una parte del trayecto se separan completamente, convirtiendose totalmente diferentes. A eso es a lo que se le conoce como el "caos"la dependencia significativa en las condiciones iniciales. Es lo que se representa ahora como el efecto mariposa. Es una manera facil de comprender lo que pasa en este fenómeno.

Supongamos que el "efecto mariposa" es un concepto que hace referencia a la noción del tiempo y a las condiciones iniciales dentro del marco de la teoría del caos. La idea es que, dadas unas condiciones iniciales de un determinado sistema caótico, la más mínima variación en ellas puede provocar que el sistema evolucione en formas completamente diferentes. Sucediendo así que, una pequeña perturbación inicial, mediante un proceso de amplificación, podrá generar un efecto considerablemente grande a mediano o corto plazo de tiempo.

Para la siguiente sección se utilizaran los códigos proporcionados por Matplotlib y Pythonic Perambulations.

3. Resultados

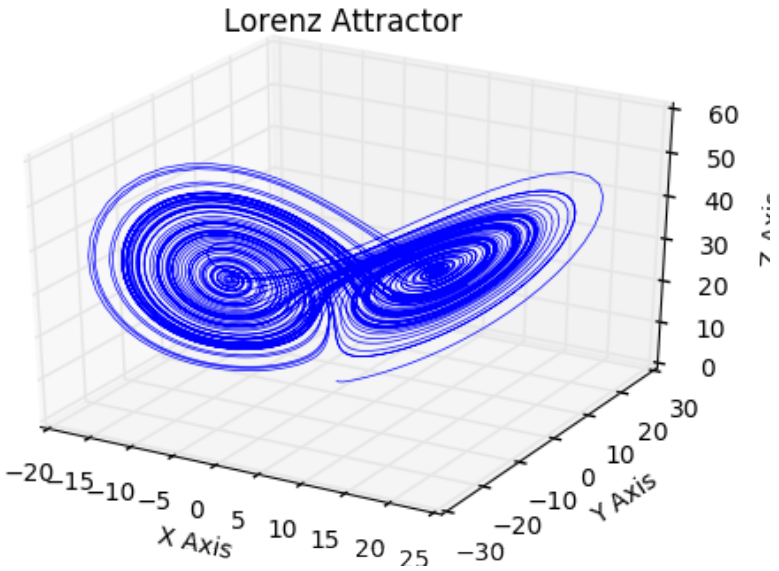
Utilizando el código de *Matplotlib* de Python se graficó el atractor de Lorenz. De igual forma se jugará con distintos ángulos de vista para visualizar los cambios en el atractor.

Código utilizado:

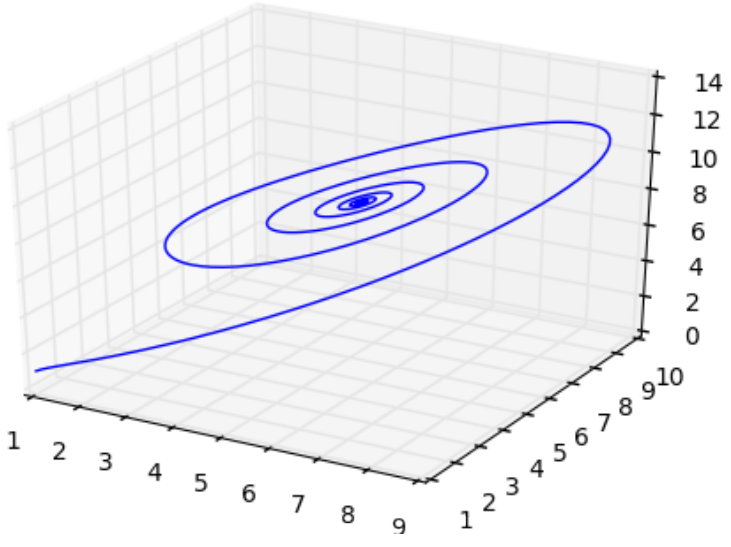
```
# Plot of the Lorenz Attractor based on Edward Lorenz's 1963 "Deterministic
# Nonperiodic Flow" publication.
# Note: Because this is a simple non-linear ODE, it would be more easily
#       done using SciPy's ode solver, but this approach depends only
#       upon NumPy.

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def lorenz(x, y, z, s=10, r=28, b=2.667):
    x_dot = s*(y - x)
    y_dot = r*x - y - x*z
    z_dot = x*y - b*z
    return x_dot, y_dot, z_dot


dt = 0.01
stepCnt = 10000
```

```
# Need one more for the initial values
xs = np.empty((stepCnt + 1,))
ys = np.empty((stepCnt + 1,))
zs = np.empty((stepCnt + 1,))

# Setting initial values
xs[0], ys[0], zs[0] = (0., 1., 1.05)

# Stepping through "time".
for i in range(stepCnt):
    # Derivatives of the X, Y, Z state
    x_dot, y_dot, z_dot = lorenz(xs[i], ys[i], zs[i])
    xs[i + 1] = xs[i] + (x_dot * dt)
    ys[i + 1] = ys[i] + (y_dot * dt)
    zs[i + 1] = zs[i] + (z_dot * dt)

fig = plt.figure()
ax = fig.gca(projection='3d')

ax.plot(xs, ys, zs, lw=0.5)
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("Lorenz Attractor")

plt.show()
```

Obteniendo la siguiente figura:



Al jugar con diferentes valores para los ángulos podemos notar como con cambios en las condiciones iniciales obtenemos muy diferentes trayectorias:



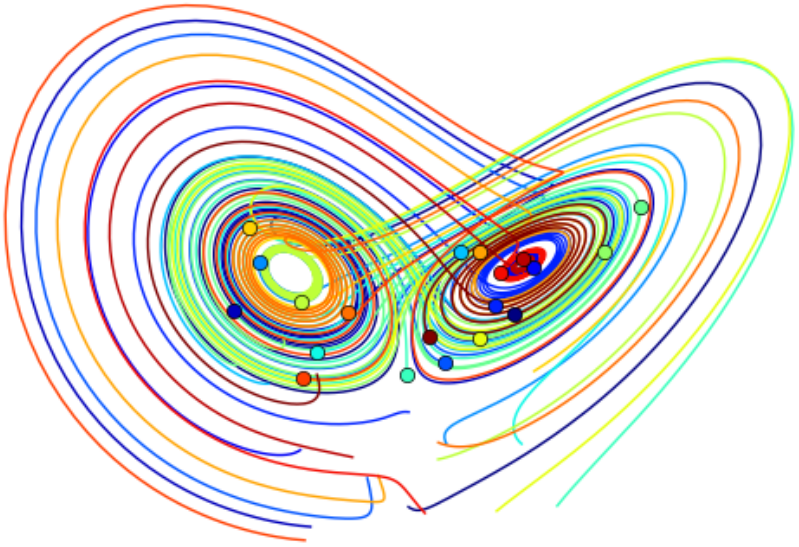
Ahora utilizando el código de *Pythonic Perambulations* se graficó el atractor de Lorenz. De igual forma se jugará con distintos ángulos de vista para visualizar los cambios en el atractor.

Código utilizado:

```
import numpy as np
from scipy import integrate

# Note: t0 is required for the odeint function, though it's not used here.
def lorentz_deriv(params, t0, sigma=10., beta=8./3, rho=28.0):
    """Compute the time-derivative of a Lorentz system."""
    x, y, z = params
    return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

x0 = [1, 1, 1]  # starting vector
t = np.linspace(0, 3, 1000)  # one thousand time steps
x_t = integrate.odeint(lorentz_deriv, x0, t)

import numpy as np
from scipy import integrate

from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import cnames
from matplotlib import animation

N_trajectories = 20


def lorentz_deriv(params, t0, sigma=10., beta=8./3, rho=28.0):
    """Compute the time-derivative of a Lorentz system."""
    x, y, z = params
    return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]


# Choose random starting points, uniformly distributed from -15 to 15
np.random.seed(1)
x0 = -15 + 30 * np.random.random((N_trajectories, 3))

# Solve for the trajectories
```

```
t = np.linspace(0, 4, 1000)
x_t = np.asarray([integrate.odeint(lorentz_deriv, x0i, t)
                  for x0i in x0])

# Set up figure & 3D axis for animation
fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1], projection='3d')
ax.axis('off')

# choose a different color for each trajectory
colors = plt.cm.jet(np.linspace(0, 1, N_trajectories))

# set up lines and points
lines = sum([ax.plot([], [], [], '-', c=c)
             for c in colors], [])
pts = sum([ax.plot([], [], [], 'o', c=c)
           for c in colors], [])

# prepare the axes limits
ax.set_xlim((-25, 25))
ax.set_ylim((-35, 35))
ax.set_zlim((5, 55))

# set point-of-view: specified by (altitude degrees, azimuth degrees)
ax.view_init(30, 0)

# initialization function: plot the background of each frame
def init():
    for line, pt in zip(lines, pts):
        line.set_data([], [])
        line.set_3d_properties([])

        pt.set_data([], [])
        pt.set_3d_properties([])
    return lines + pts

# animation function.  This will be called sequentially with the frame number
def animate(i):
    # we'll step two time-steps per frame.  This leads to nice results.
    i = (2 * i) % x_t.shape[1]
```

```
    for line, pt, xi in zip(lines, pts, x_t):
        x, y, z = xi[:i].T
        line.set_data(x, y)
        line.set_3d_properties(z)

        pt.set_data(x[-1:], y[-1:])
        pt.set_3d_properties(z[-1:])

    ax.view_init(30, 0.3 * i)
    fig.canvas.draw()
    return lines + pts

# instantiate the animator.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=500, interval=30, blit=True)

# Save as mp4. This requires mplayer or ffmpeg to be installed
anim.save('lorenz_attractor.mp4', fps=15, extra_args=['-vcodec', 'libx264'])

plt.show()
```



4. Conclusión

En esta práctica se pudieron crear animaciones por medio de códigos en Python. En la carpeta Github se adjuntan las animaciones obtenidas con el código anterior. La original y la obtenida despues de jugar con los parámetros del código.

5. Referencias

(1) https://ociointeligente.wordpress.com/2011/12/21/el-efecto-mariposa- la-teoria-del-caos-para-dummies/

(2) $https://matplotlib.org/2,0,0/examples/mplot3d/lorenz_attractor.html$

(3) $https://github.com/gboeing/lorenz-system$

(4) $https://jakevdp.github.io/blog/2013/02/16/animating-the-lorentz-system-in-3d/$