Assignments UNIK/TEK4660 Spring 2018

Joseph Knutson

March 6, 2018

Runge Kutta and the Lorentz Attractor

Let (r_1, r_2, r_3) = (x,y,z) and $dx/dt = x^{(1)}$ consider the Lorentz system

$$x^{(1)} = \sigma(y - x) \tag{1}$$
$$y^{(1)} = x(\rho - z) - y \tag{2}$$
$$z^{(1)} = xy - \beta z \tag{3}$$

our job is to integrate these velocities in order to create the pathline $l(x, y, z)$. This can be done in a simple manner, using the forward euler algorithm, or in a more accurate manner user the 4th order runge kutta algorithm.

Forward Euler

Solved numerically with forward Euler, the e.o.m.(s) will look like this:

$$x_{t+h} = \sigma(y_t - x_t) * h + O(h^2) \tag{4}$$
$$y_{t+h} = [x_t(\rho - z_t) - y_t] * h + O(h^2) \tag{5}$$
$$z_{t+h} = (x_t y_t - \beta z_t) * h + O(h^2) \tag{6}$$

where $t + h$ is a timestep forward (into the future), t is the current time and $O(h^2)$ is the size of the error.

For the Lorentz attractor, $\rho = 28$, $\sigma = 10$ and $\beta = 8/3$. Implementing this solution in python over 10 time units using forward euler and $x_0 = 10$, $z_0 = 10$, $z_0 = 10$, returns the graph in figure 1

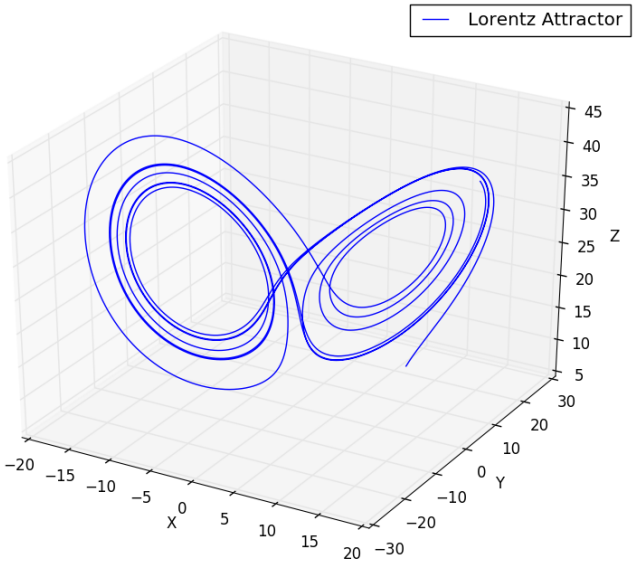

Figure 1: Lorentz attractor made in Python using the forward Euler algorithm.

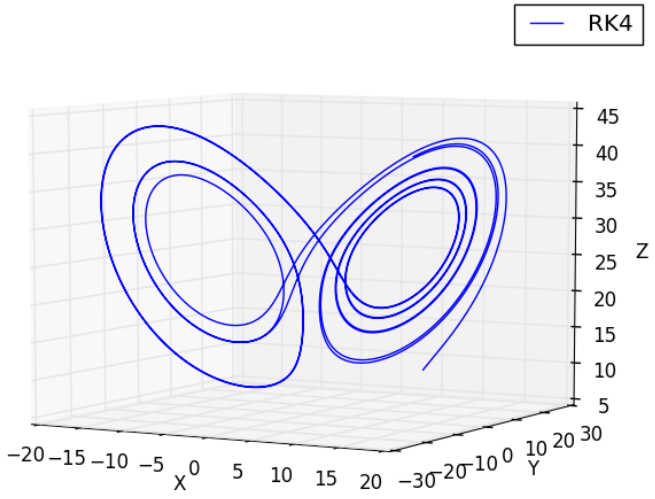

Figure 2: Lorentz attractor made in Python using the rk45 algorithm. The graph differs from the Euler solution in that the pathline stays on the same path, making the lines overlap more. The euler solution however seems to not overlap itself, making the lines more spread out.

RK4

Instead of implementing Runge-Kutta 4, we use a python module (scipy).

The scipy.integrate library has a method called ode which contains various integration methods, including rk45. The script I've implemented looks something like this:

```
u0 = [10,10,10]              #Initial positions.

def f(t,u,sigma,rho,beta):   #Right hand side of -
    x,y,z = u                #differential equations.
    return [sigma*(y-x), x*(rho-z) - y, x*y-beta*z]

#initialize integrator with function f, "dopri5" is rk45
r1 = ode(f).set_integrator('dopri5',atol=1e-6)

#integrate with initial conditions
r1.set_initial_value(u0, 0).set_f_params(sigma,rho,beta)
```

In figure 2 and 3 you can see both the rk45 solution of the Lorentz attractor and both solutions superimposed on eachother. As you might be able to pick up from these figures, the euler solution is unable to stay on track due to it's increase in numeric error.

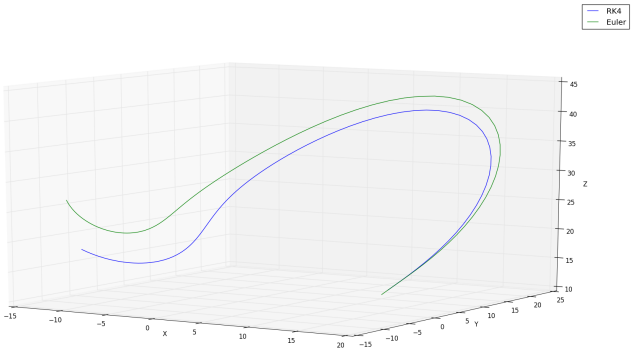

Figure 3: Line segments of both solutions. Observe how the error in the euler solution drastically increases from the more accurate rk45 solution.

Task A

We now want to calculate κ and τ, the curvature and torsion. These are expressed in equation 7 and 8:

$$\kappa = \frac{||\dot{r} \times \ddot{r}||}{||\dot{r}||^3} \tag{7}$$

$$\tau = \frac{\dot{r} \cdot (\ddot{r} \times \dddot{r})}{||\dot{r} \times \ddot{r}||^2} \tag{8}$$

Instead of deriving these expressions and the derivatives analytically, let's simply find the 2nd and 3rd differentials numerically Thanks to rk45/ode45, we now have $\vec{r}$. By using the differential equations we started with; 1, 2 and 3, we possess the velocity, $\dot{r}$ too. The expressions for the second derivatives can be found in equation 9, 10 and 11.

$$x^{(2)} = \sigma(y^{(1)} - x^{(1)}) \tag{9}$$

$$y^{(2)} = x^{(1)}(\rho - z) - xz^{(1)} - y^{(1)} \tag{10}$$

$$z^{(2)} = x^{(1)}y + xy^{(1)} - \beta z^{(1)} \tag{11}$$

You can see that the product rule has been used in both equation 10 and 11. Repeating this differentiation gives us expressions for the third derivatives.

$$x^{(3)} = \sigma(y^{(2)} - x^{(2)}) \tag{12}$$

$$y^{(3)} = x^{(2)}(\rho - z) - 2x^{(1)}z^{(1)} - xz^{(2)} - y^{(2)} \tag{13}$$

$$z^{(3)} = x^{(2)}y + 2x^{(1)}y^{(1)} + xy^{(2)} - \beta z^{(2)} \tag{14}$$

After acquiring x, y and z from the numerical integration, we can already begin implementing the code that gives us the derivatives:

```
vx = sigma*(y-x) #derivatives from 1st to 3rd order
vy = x*(rho-z)-y
vz = x*y-beta*z
x_2 = sigma*(vy - vx)
y_2 = vx*(rho - z) - x*vz - vy
z_2 = vx*y + x*vy - beta*vz
x_3 = sigma*(y_2 - x_2)
y_3 = x_2*(rho - z) - 2*vx*vz - x*z_2 - y_2
z_3 = x_2*y + 2*vx*vy + x*y_2 - beta*z_2
```

We now possess the derivatives from 1st to 3rd order.

Using these expressions, we can immediately derive the curvature and torsion's values, based on equation 7 and 8.

```
curvature = np.zeros(n) #empty arrays
torsion   = np.zeros(n)
for i in range(n):
    #first we define the elements in the equations
    d  = np.asarray([ vx[i] ,vy[i] ,vz[i]]) #r dot
    dd = np.asarray([x_2[i],y_2[i],z_2[i]]) #r dotdot
    ddd = np.asarray([x_3[i],y_3[i],z_3[i]])#r dotdotdot
    dxdd = np.cross(d,dd)                   #rdot x rdotdot
    dnorm = np.linalg.norm(d)               #||rdot||
    dxddnorm = np.linalg.norm(dxdd)         #||rdot x rdotdot||

    curvature[i] = dxddnorm/(dnorm*dnorm*dnorm)
    torsion[i] = np.dot(d, np.cross(dd,ddd)) / (dxddnorm*dxddnorm)
```

Visualizing the Lorentz attractor while also visualizing the torsion and curvature can be tricky. Using color, line thickness and vectors could be viable. Since these are scalar values, and I have no idea of how to vary line thickness, colorplots became the most optimal solution. In figure 4 and 5, you can observe how the torsion and curvature behave along the attractor independently.

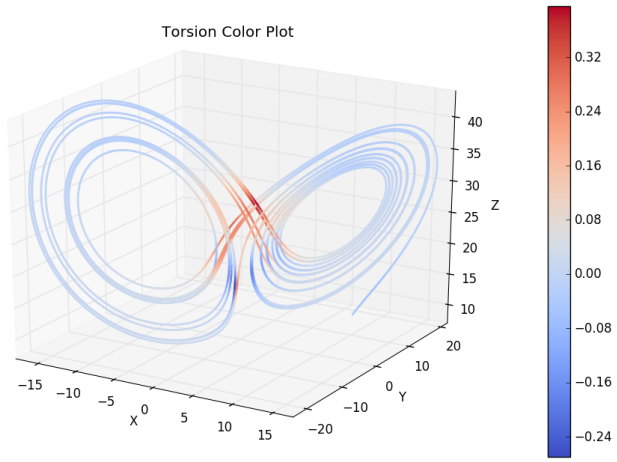

Figure 4: Colorplot of the torsion along the Lorentz attractor. We can see that the most intense values happen during the transition from one spiral to another.

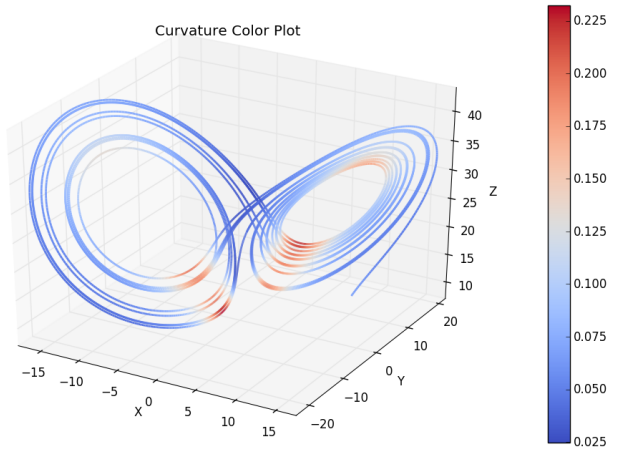

Figure 5: Colorplot of the curvature. The maxima here seem to only happen at the "poles" of the spirals.

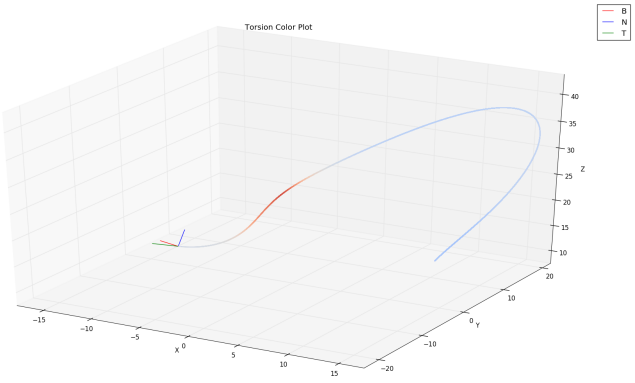

Figure 6: Torsion Animation with a live visualization of the T, N and B vectors. Sorry for the unreadable labels, R,G,B = B,T,N

Task B

Python was not suited for this task, but I made it. The tangent vector T, curve normal N and Binormal B are defined in the following equations:

$$T = \frac{\dot{r}}{||\dot{r}||} \tag{15}$$

$$N = \frac{\dot{T}}{||\dot{T}||} \tag{16}$$

$$B = T \times N \tag{17}$$

In order to find B, we need N and T. In order to find N we need T and in order to find T we need $\dot{r}$, which we already have! The code below implements these new vectors into our script:

```
1 [...]
2     T[:,i] = d[:]/dnorm          #dot r / norm(dot r)
3     dT = (T[:,i]-T[:,i-1])/dt #dot T
4     N[:,i] = dT/np.linalg.norm(dT)  #dot T/ norm(dot T)
5     B[:,i] = np.cross(T[:,i],N[:,i])#T x B
```

Line 3 in the code above is a little cheat to find the derivative of T, $\dot{T}$. I first found the initial value of T(t_0) and then subtracted it from T(t_0+dt) and divided by dt to find the derivative values repeadiately. Now having the values, using them in the animation was unjustifyingly hard. Snapshots of the animation of both the torsion and curvature plots can be seen in figure 6, 7, 8 and 9.

as for other solutions involving different values for ρ, σ and β, check out figure 10 and 11

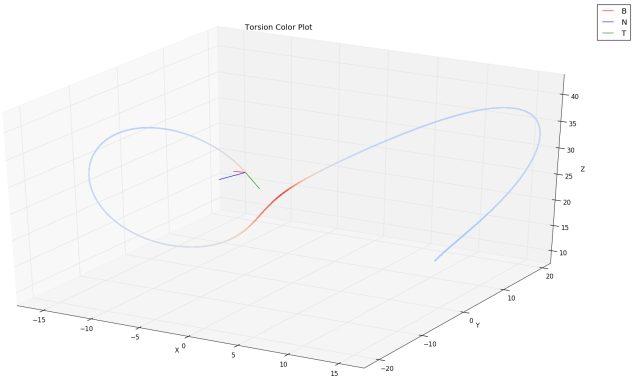

Figure 7: Torsion Animation few seconds later.

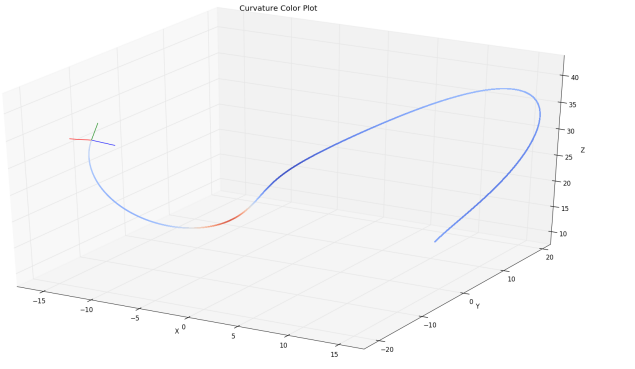

Figure 8: Curvature Animation with a live visualization of the T, N and B vectors. RGB = BTN

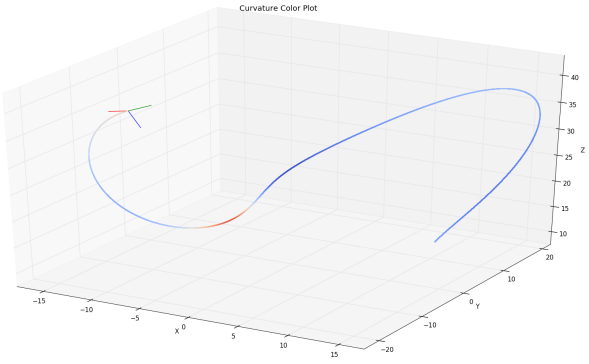

Figure 9: Curvature Animation few seconds later.

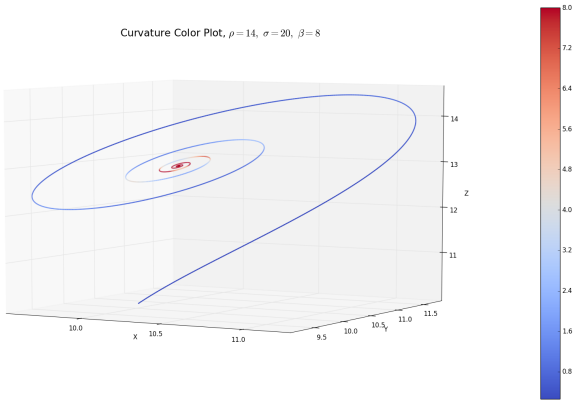

Figure 10: Curvature plot for an alternative solution which seems to converge.

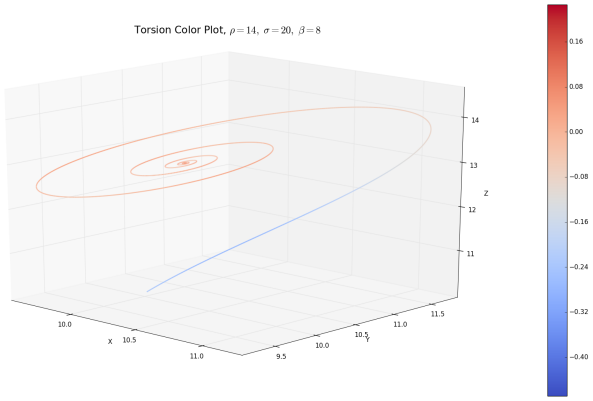

Figure 11: Torsion plot, quite homogenous.

Task C