Introduction to Computational Physics – Exercise 9

Simon Groß-Bölting, Lorenz Vogel, Sebastian Willenberg

June 26, 2020

The Lorenz Attractor

The Lorenz attractor problem is given by the following coupled set of differential equations:

$$\dot{x} = -\sigma(x - y) \tag{1}$$

$$\dot{y} = rx - y - xz \tag{2}$$

$$\dot{z} = xy - bz \tag{3}$$

As discussed in the lecture, the fixed points are $\begin{pmatrix}0 & 0 & 0\end{pmatrix}$ for all r, and (for $r > 1$) the points $C_\pm = \begin{pmatrix}\pm a_0 & \pm a_0 & r-1\end{pmatrix}$ with $a_0 = \sqrt{b(r-1)}$. For the entire exercise, please use $\sigma = 10$ and $b = 8/3$. The value of r can be experimented with. When you create numerical solutions you can make plots in 2-D projection (e.g. in the (x, y)- or (x, z)-plane). You can also try a full 3-D plot.

Task: Solve numerically, using `rk4`, the above coupled set of equations for the values $r = 0.5$, 1.17, 1.3456, 25.0 and 29.0. Choose the initial conditions near one of the fixed points: $C_\pm$ for $r > 1$ and $(0, 0, 0)$ for $r < 1$. Explain the behavior, as much as possible, with the stability properties of the fixed points.

To solve the problem numerically we can use the Runge-Kutta Algorithm to approximate the functions x, y and z. It is important to note that $x(t)$, $y(t)$ and $z(t)$ are time dependant. We will start by importing the packages that we will need for this exercise:

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
```

The `numpy` package allows us to work with arrays while the `matplotpib.pyplot` and `mplot3d` packages will allow us to plot our data in a 3 dimentional graph.
The next step is to define our parameters.

```python
#Initial Values
sig = 10
b = 8/3
r = np.array([0.5,1.17, 1.3456, 25.0, 29.0])
a0 = np.sqrt(b*(r.astype('complex')-1))
```

When defining the parameter $a_0 = \sqrt{b(r-1)}$ we have to set the type of r to `complex` to avoid gettng a `nan` value in the case that $r < 1$.
The differential equations are defined in the following manner:

```python
#Lorenz System
def dx(x,y,z,sig):
    return -sig*(x-y)
def dy(x,y,z,r):
    return r*x-y-x*z
def dz(x,y,z,b):
    return x*y-b*z
```

All the differential equations in our lorenz system are dependant of the functions `x`, `y` and `z` and a variable σ, r or b. That means that we have to rewrite our `rk4` algorithm in such a way that we will iterate over `x`, `y` and `z` instead of iterating over `y` and `t` (or rather `x`). The `rky_step` function looks as follows:

```
def rk4_step(x0, y0, z0, f1, f2, f3, h, f1_args = {}, f2_args = {}, f3_args = {}):
    ''' Simple python implementation for one RK4 step.
        Inputs:
            x_0     - M x 1 numpy array specifying all variables of the first ODE at
            the current time step
            y_0     - M x 1 numpy array specifying all variables of the second ODE at
            the current time step
            z_0     - M x 1 numpy array specifying all variables of the third ODE at
            the current time step
            f       - function that calculates the derivates of all variables of the ODE
            h       - step size
            f1_args - Dictionary of additional arguments to be passed to the function f1
            f2_args - Dictionary of additional arguments to be passed to the function f2
            f3_args - Dictionary of additional arguments to be passed to the function f3
        Output:
            xp1 - M x 1 numpy array of variables of the first ODE at time step x0 + h
            yp1 - M x 1 numpy array of variables of the second ODE at time step x0 + h
            zp1 - M x 1 numpy array of variables of the third ODE at time step x0 + h
            tp1 - time step t0+h
    '''
    k1 = h * f1(x0, y0, z0, **f1_args)
    l1 = h * f2(x0, y0, z0, **f2_args)
    m1 = h * f3(x0, y0, z0, **f3_args)

    k2 = h * f1(x0 + k1/2., y0 + k1/2., z0 + k1/2., **f1_args)
    l2 = h * f2(x0 + l1/2., y0 + l1/2., z0 + l1/2., **f2_args)
    m2 = h * f3(x0 + m1/2., y0 + m1/2., z0 + m1/2., **f3_args)

    k3 = h * f1(x0 + k2/2., y0 + k2/2., z0+ k2/2, **f1_args)
    l3 = h * f2(x0 + l2/2., y0 + l2/2., z0+ l2/2, **f2_args)
    m3 = h * f3(x0 + m2/2., y0 + m2/2., z0+ m2/2, **f3_args)

    k4 = h * f1(x0 + k3/2., y0 + k3/2., z0+ k3/2, **f1_args)
    l4 = h * f2(x0 + l3/2., y0 + l3/2., z0+ l3/2, **f2_args)
    m4 = h * f3(x0 + m3/2., y0 + m3/2., z0+ m3/2, **f3_args)

    xp1 = x0 + 1./6.*(k1 + 2.*k2 + 2.*k3 + k4)
    yp1 = y0 + 1./6.*(l1 + 2.*l2 + 2.*l3 + l4)
    zp1 = z0 + 1./6.*(m1 + 2.*m2 + 2.*m3 + m4)

    return(xp1, yp1, zp1)
```

```
def rk4(x0, y0, z0, f1, f2, f3, h, n, f1_args = {}, f2_args = {}, f3_args = {}):
    ''' Simple implementation of RK4
        Inputs:
            x_0     - M x 1 numpy array specifying all variables of the ODE at
            the current time step
            y_0     - M x 1 numpy array specifying all variables of the ODE at
            the current time step
            z_0     - M x 1 numpy array specifying all variables of the ODE at
            the current time step
            f1      - function that calculates the derivates of all variables of
            the first ODE
            f1      - function that calculates the derivates of all variables of
            the second ODE
            f1      - function that calculates the derivates of all variables of
            the  third ODE
            h       - step size
            n       - number of steps
            f1_args - Dictionary of additional arguments to be passed to the function f1
            f2_args - Dictionary of additional arguments to be passed to the function f2
            f3_args - Dictionary of additional arguments to be passed to the function f3
        Output:
            yn - N+1 x M numpy array with the results of the integration for
            every time step (includes y0)
            xn - N+1 x 1 numpy array with the time step value (includes start x0)
    '''
    xn = np.zeros(n+1); xn[0] = x0
    yn = np.zeros(n+1); yn[0] = y0
    zn = np.zeros(n+1); zn[0] = z0

    for n in np.arange(1,n+1,1):
```

```
        xn[n], yn[n], zn[n], = rk4_step(x0 = xn[n-1], y0 = yn[n-1], z0 = zn[n-1],
        f1 = f1, f2 = f2, f3 = f3, h = h,
        f1_args = f1_args, f2_args = f2_args, f3_args = f3_args)
    return(xn, yn, zn)
```

The only thing that is left to get the results is to plot our values:

```
plt.figure()
ax = plt.axes(projection="3d")
ax.scatter3D(0,0,0,s=10,color='red')
ax.scatter3D(3,3,3,s=10,color='green')
for i in range(0,1):
    if r[i] <= 1:
        ax.scatter3D(*rk4(x0=3, y0=3, z0=3,
        f1=dx, f2=dy, f3=dz, h=0.01, n=10000,
        f1_args={'sig':sig}, f2_args={'r':r[i]}, f3_args={'b':b}), s=1,
        label='r={}'.format(r[i]))
    else:
        ax.scatter3D(*rk4(x0=a0[i]+1, y0=a0[i]+1, z0=r[i],
        f1=dx, f2=dy, f3=dz, h=0.01, n=1000,
        f1_args={'sig':sig}, f2_args={'r':r[i]}, f3_args={'b':b}), s=1,
        label='r={}'.format(r[i]))
plt.legend()
plt.show()
```

It is important to note here that our initial conditions were set to `x0, y0, z0 = 0, 0, 0` in the case that $r < 1$. That means that our initial conditons are always real.

Task: Determine the sequence z_k for $r = 26.5$, where z_k is a local maximum in z on the solution curve after k periods. Plot z_{k+1} as a function of z_k. When sufficient points are there, connect the points. The resulting function $z_{k+1} = f(z_k)$ has an intersection with the diagonal $z_{k+1} = z_k$. It is a fixed point of the function $f(z_k)$. Is the slope m of this function > 1, < -1 or between -1 and $+1$? Notice: The theory of discrete maps says that there is no periodic solution if $|m| > 1$. So, in such a case we can deduce that this solution of the Lorenz system is not periodic.