# 计算物理第5次作业

**吴远清 2018300001031**

## 单摆系统的计算与表征

导入相关库和设置常数:

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.animation import FuncAnimation
from math import pi,sin

g = 9.8
```

我们构建`pendulum`类,来实现对混沌摆的计算,分析与结果可视化.

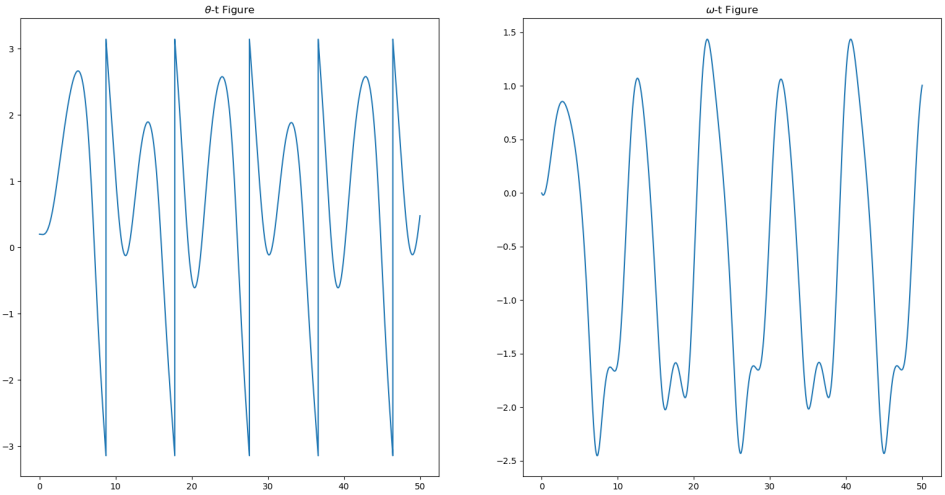
```
class pendulum():
    def __init__(self,l = 9.8,Fd =1.44,omegad = 2/3.0,dt = 0.001, q = 0.5,
theta0 = 0.2):
        self.l = l
        self.Fd = Fd
        self.omegad = omegad
        self.dt = dt
        self.q = q
        self.theta0 = theta0
    def calculate(self, t0 = 50):
        self.theta = []
        self.omega = []
        self.t = []
        k = g/self.l
        w = 0
        a = self.theta0
        t_ = 0
        while t_ <= t0:
            self.omega.append(w)
            self.theta.append(a)
            self.t.append(t_)
            w += (-k*sin(a)-self.q*w+self.Fd*sin(self.omegad*t_)) * self.dt
            a += w * self.dt
            if a > pi:
                a -= 2*pi
            elif a < -pi:
                a += 2*pi
            t_ += self.dt
    def showTrajectory(self):
        plt.subplot(1,2,1)
        plt.plot(self.t, self.theta)
        plt.title('$\\theta$-t Figure')
        plt.subplot(1, 2, 2)
        plt.plot(self.t, self.omega)
        plt.title('$\\omega$-t Figure')
        plt.show()
```

```
    def __animeInit(self):
        self.ax.set_xlim(min(self.theta), max(self.theta))
        self.ax.set_ylim(min(self.omega), max(self.omega))
        self.text_pt.set_position((min(self.theta), min(self.omega)))
        return self.ln,self.text_pt,
    def __animeUpdate(self, frame):
        self.xdata.append(self.theta[frame])
        self.ydata.append(self.omega[frame])
        self.ln.set_data(self.xdata, self.ydata)
        self.point.set_data(self.theta[frame], self.omega[frame])
        self.text_pt.set_text("t=%.3f"%(self.t[frame]))
        return self.ln,self.point,self.text_pt,
    def showPhaseGraph(self):
        plt.plot(self.theta, self.omega)
        plt.show()
    def animePahseGraph(self):
        self.fig, self.ax = plt.subplots()
        self.xdata, self.ydata = [], []
        self.ln, = plt.plot([], [])
        self.point, = plt.plot(self.theta[0], self.omega[0],"ro")
        self.text_pt = plt.text(0, 0, '', fontsize=16)
        ani = FuncAnimation(self.fig, self.__animeUpdate,
frames=range(len(self.t)),
                    init_func=self.__animeInit, blit=True, interval = 10)
        plt.show()
    def PoincareSection(self, isShow = True, waiting = 0):
        x = []
        y = []
        for i in range(len(self.t)):
            if self.t[i] > 2*pi/self.omegad*waiting:
                if abs(round(self.t[i]/(2*pi/self.omegad)) -
(self.t[i]/(2*pi/self.omegad))) < 0.001:
                    x.append(self.theta[i])
                    y.append(self.omega[i])

        self.PoincareSectionX = x
        self.PoincareSectionY = y
        if isShow:
            plt.scatter(x,y,s=5)
            plt.show()
    def bifurcationDiagram(self):
        x,y,cx,cy = [],[],[],[]
        for fd in np.linspace(1.35,1.48,20):
            self.Fd = fd
            self.calculate(800*pi/self.omegad + 2)
            index = int((600*pi/self.omegad)/self.dt)
            flag = 600*pi
            for i in range(index, len(self.t)):
                if (self.omegad * self.t[i] - flag - 2 * pi) > 0:
                    if np.abs(self.omegad * self.t[i - 1] - flag) <
np.abs(self.omegad * self.t[i] - flag):
                        x.append(fd)
                        y.append(self.theta[i - 1])
                    else:
                        x.append(fd)
                        y.append(self.theta[i])
                    flag += 2*pi
            y_ = np.array(y)
```

```
            x_ = np.array(x)
            index_ = x_ == fd
            ylist = y_[index_]
            li = [{'c':ylist[0],'list':[ylist[0]]}]
            for i in ylist[1:]:
                cflag = False
                for j in range(len(li)):
                    if np.abs(i - li[j]['c']) < 0.1:
                        cflag = True
                        s_ = li[j]['list']
                        s_.append(i)
                        li[j] = {'c':np.mean(s_),'list':s_}
                if not cflag:
                    li.append({'c':i,'list':[i]})
            print(fd, len(li))
            for i in li:
                cx.append(fd)
                cy.append(i['c'])
        plt.scatter(x,y,s = 5)
        plt.scatter(cx,cy,color='red', s = 8)
        plt.show()
```

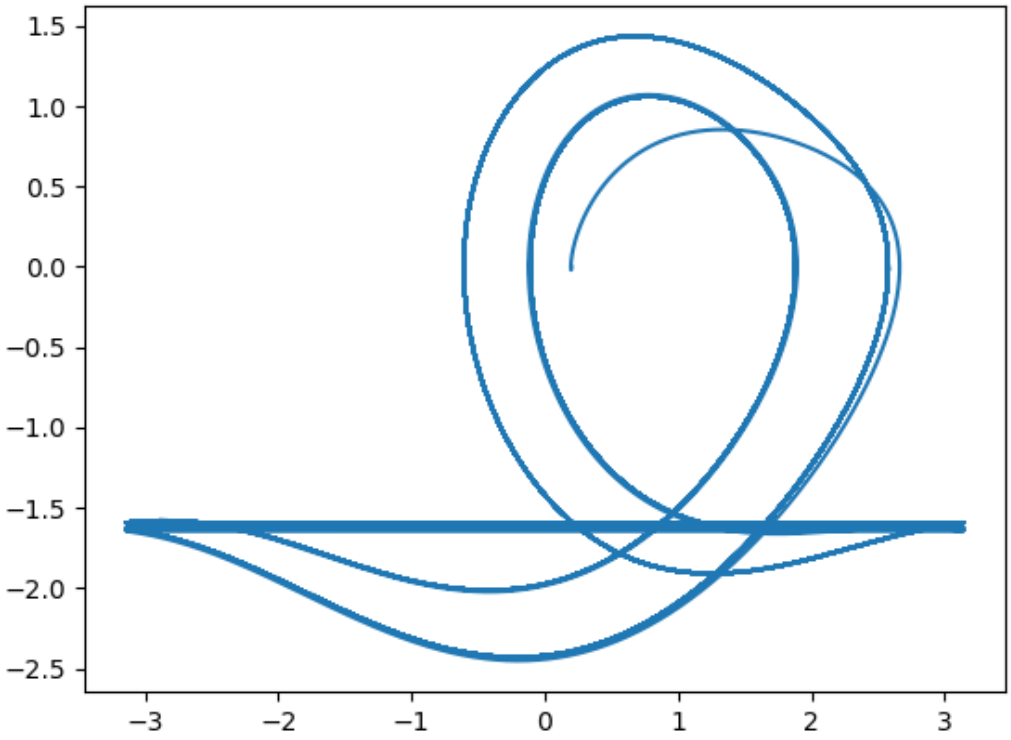
`calculate`方法实现了Euler-Cromer算法, 并将计算结果储存在内部.

`showTrajectory`方法可以直接显示$\theta - t$曲线.展示单摆运动轨迹.




`__animeInit`和`__animeUpdte`用于实现动画功能.

`showPhaseGraph`实现了绘制$\omega - \theta$图的功能,可以展示单摆运动在相空间中的轨迹.

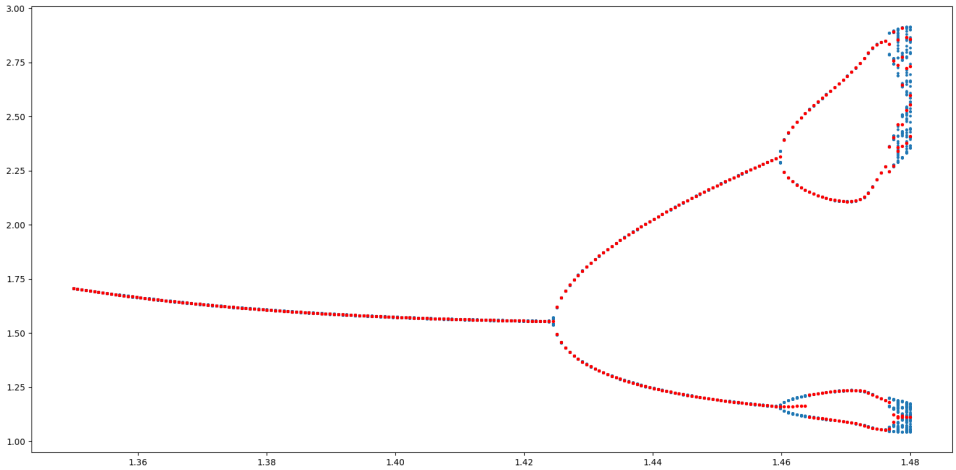



`animePahseGraph` 通过 `matplotlib` 中的 `FuncAnimation` 函数,实现了动画展示$\omega - \theta$图的功能,可以观察到单摆的混动情况,周期运动和相轨道的演化过程.

动图无法展示,可自行运行尝试

`PoincareSection` 实现了绘制指定系统的庞加莱截面图的功能,并提供 `waiting` 参数,来设置在观测前等待的周期数. 运行结果见第一题.

`bifurcationDiagram` 实现了绘制指定系统的分叉图的功能. 其中, 我们仿照聚类算法的思想, 对每个$F_d$所得数据点进行分类, 可以表征当前的分叉数.



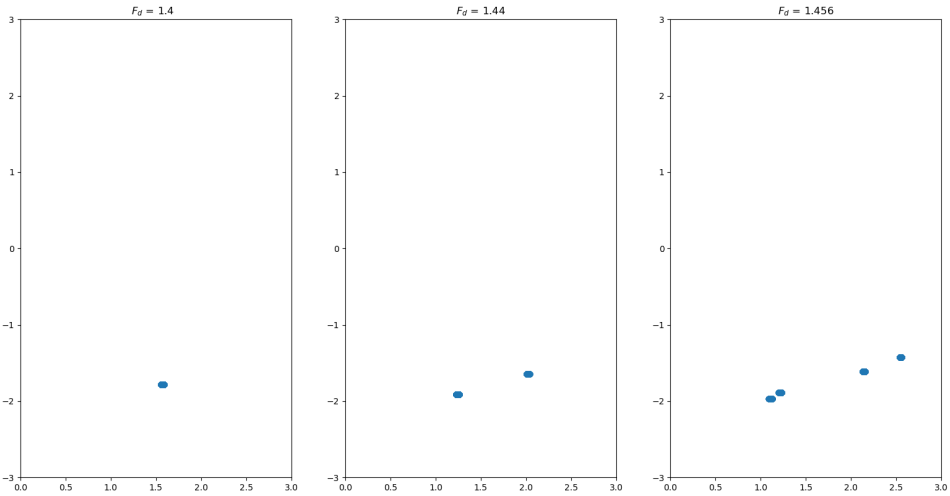

# 第一题

实现了 `pendulum` 类后,对第一题的计算可以很简单的实现:

```
plt.subplot(1,3,1)
```

```
plt.xlim(-0,3)
plt.ylim(-3,3)
p1 = pendulum(Fd = 1.4)
p1.calculate(5000)
p1.PoincareSection(False, waiting=300)
plt.scatter(p1.PoincareSectionX, p1.PoincareSectionY)
plt.title('$F_d$ = 1.4')

plt.subplot(1,3,2)
plt.xlim(-0,3)
plt.ylim(-3,3)
p1 = pendulum(Fd = 1.44)
p1.calculate(5000)
p1.PoincareSection(False, waiting=300)
plt.scatter(p1.PoincareSectionX, p1.PoincareSectionY)
plt.title('$F_d$ = 1.44')

plt.subplot(1,3,3)
plt.xlim(-0,3)
plt.ylim(-3,3)
p1 = pendulum(Fd = 1.465)
p1.calculate(5000)
p1.PoincareSection(False, waiting=300)
plt.scatter(p1.PoincareSectionX, p1.PoincareSectionY)
plt.title('$F_d$ = 1.456')

plt.show()
```

计算结果如图:




可以看出,确实符合规律.

# 第二题

此处，为了提升计算速度,我们使用`Cython`重构代码,并使用`multiprocessing`进行多线程加速.模块代码重构如下:

```
# FileName:main_Cython.pyx
from libc.math cimport sin
import matplotlib.pyplot as plt
```

```
cdef calculate(double l=9.8, double Fd = 1.44,double omegad = 2/3.0, double dt =
0.001, double q = 0.5, double theta0 = 0.2, double t0 = 50):
    cdef theta = []
    cdef omega = []
    cdef t = []
    cdef double k,w,a,t_,pi,g
    g = 9.8
    pi = 3.141592653
    t = []
    k = g/l
    w = 0
    a = theta0
    t_ = 0
    while t_ <= t0:
        omega.append(w)
        theta.append(a)
        t.append(t_)
        w += (-k*sin(a)-q*w+Fd*sin(omegad*t_)) * dt
        a += w * dt
        if a > pi:
            a -= 2*pi
        elif a < -pi:
            a += 2*pi
        t_ += dt
    return theta, omega, t

def bifurcationDiagram(double omegad, double dt):
    cdef double pi, flag, fd
    cdef int i, index
    pi = 3.141592653
    cdef x = []
    cdef y = []
    for fd in np.linspace(1.35,1.48,200):
        a,w,t = calculate(9.8, fd, omegad, dt, 0.5, 0.2, 800*pi/omegad + 2)
        index = int((600*pi/omegad)/dt)
        flag = 600*pi
        for i in range(index, len(t)):
            if (omegad * t[i] - flag - 2 * pi) > 0:
                x.append(fd)
                y.append(a[i])
                flag += 2*pi
    plt.scatter(x,y,s = 5)
    plt.title("$\Omega_d$={}, dt={}".format(omegad,dt))
    plt.savefig("./{}-{}.png".format(omegad, dt))
```

编译后调用,对比计算速度:

- Python,`bifurcationDiagram(dt = 0.001,omegad = 2/3.0)`:19.76 Secs
- Cython,`bifurcationDiagram(dt = 0.001,omegad = 2/3.0)`:1.62 Secs

可见提速约12倍.

计算代码如下:

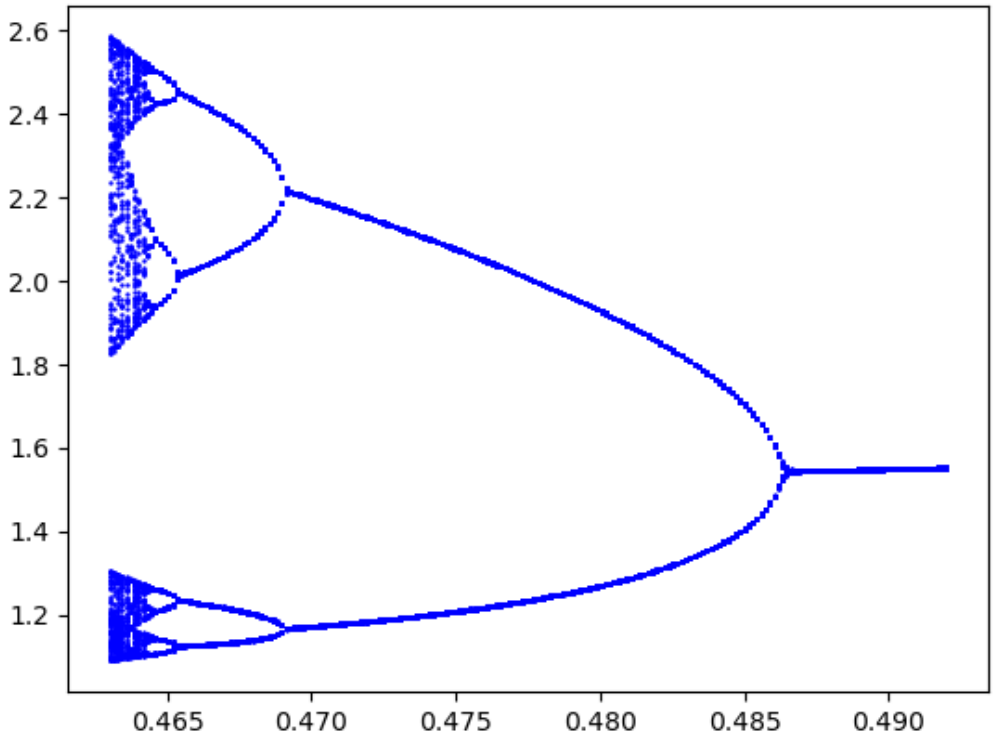
```
from main_Cython import bifurcationDiagram
from multiprocessing import Pool
import numpy as np
```

```
import matplotlib.pyplot as plt

def main():
    wdList = np.linspace(1/5, 2, 1000)
    pool = Pool(processes=16)
    for i in range(len(wdList)):
        pool.apply_async(bifurcationDiagram, args = (wdList[i], 0.001))
    print('Submit success!')
    pool.close()
    pool.join()

if __name__ == '__main__':
    main()
```

实际计算中,设置为16进程计算.

对于$F_d$,结果已展示在上面.

对于$q$,结果如下:




可以发现,$\delta_n$均大于1.