Python Tools for Visualization of Periodically Driven Dynamical Systems

Noah J. Blair

May 4, 2020

Abstract

Periodic driving forces are very common in dynamical systems theory. One of the most common types is a force that is independent of the dynamical variables, and is sinusoidal in time. A Python program was written to generate a plot of the Poincare section of this type of force for a variety of sysems. These tools are used to analyze the damped driven pendulum and forced Duffing equation for various values of the driving force strength.

1 Introduction

For time independent dynamical systems, the solutions are often visualized as phase flow on the phase space. For one degree of freedom, these systems are characterized by their generalized coordinate q, and the conjugate momentum p. If X_q is the domain of q and X_p is the domain of p, then our phase space is $\mathbb{P} = X_q \times X_p$. Usually the domain of our dynamical variables is $\mathbb{R}$, so our phase space is $\mathbb{P} = \mathbb{R}^2$.
The visualization of solutions as curves in phase space is very useful for time-independent solutions, but if there is a time-dependent force, then the solution can be very difficult to interpret. This is because the phase curves may now intersect each other, and are no longer Jordan curves. For a subset of these problems, where the time dependent force is periodic in time, we may introduce the Poincare map to visualize these solutions. We will let the position in phase space be represented by $\eta(t) = (q(t), p(t))$ to write the differential equation as $\dot{\eta} = v(\eta, t)$. We will assume that the time-dependent part of the force has a period of 1 (this can be done by rescaling our units of time) to say that $v(\eta, t) = v(\eta, t + 1)$. We then define a function $\mathcal{P} : \eta(t) \mapsto \eta(t + 1)$ to solve the system over one period. We denote the point $\eta_n = \mathcal{P}^n(\eta(0))$. Here the superscript n denotes function composition, not powers. This generates a set of points in $\mathbb{P}$ to simplify the view of the dynamics of the system. This method then allows us to study the geometric aspects of time-dependent systems in a similar manner to time-independent systems. This also moves the problem from one in differential equations, to one in discrete dynamical systems which has some easier methods for solving, than a time-dependent system.

2 Sinusoidal Forces

A common type of system is one that is derivable from a Hamiltonian system, with a sinusoidal force added on. In this case the equations of motion become:

$$\begin{cases} \dot{q} = p \\ \dot{p} = f(q,p) + \gamma\cos(2\pi t) \end{cases} \tag{1}$$

where $f(q,p)$ is the time-independent force, and γ is a measure of the strength of the driving force. To implement this system in python, we first need a function to return the time independent force. I have created code to implement these forces for the linear damped oscillator, the nonlinear damped pendulum, and the undamped Duffing equation.

```
## Common unforced systems (called by one_period_sol)
# for all of these
    # q:= current position; float
    # p:= current momentum; float
# Linear oscillator
def linear_osc(q,p):
    # damping coefficient and natural frequency respectively
    b = 0.15
    w = 1.2
    # gives force
    force = -2.0*b*p - (w**2)*q
    return force
# Damped Driven Pendulum
def ddpend(q,p):
    # damping coefficient and natural frequency respectively
    b = 0.5
    w = 6.0
    # gives force
    force = -2.0*b*p - (w**2)*np.sin(q)
    return force
# Duffing Equation
def duffing(q,p):
    force = q - (q**3)
    return force
```

These take the input of the current position and momentum, and return the force $f(q,p)$ that the object would feel. These are then called by the following program as the time independent force. This uses a Runga-Kutta algorithm to solve the equation for the first period. In effect, this is the Poincare map applied once to a single point. The paramgeter g that shows up here is a measure of the strength of the driving force. This parameter will come up in later sections.

```
# Function to use runga kutta solve system over one period
# dq/dt = p
# dp/dt = f(q,p) + g*cos(2 pi t)
    # time_ind_force:= the time independent force (q,p); function
    # q:= position; float
    # p:= generalized momentum; float
```

```
    # g:= driving strength
def one_period_sol(time_ind_force,qo,po,g):
    # time step for runga-kutta
    dt = 1.0/50.0
    # starts q and p at their initial values
    q = qo
    p = po
    for j in range(0,50):
        # current value of time
        t = dt*j
        # the runga-kutta 45 correction terms
        k1q = dt*p
        k1p = dt*(time_ind_force(q,p) + g*np.cos(2*np.pi*t))
        k2q = dt*(p + (0.5*k1p))
        k2p = dt*(time_ind_force(q + (0.5*k1p),p + (0.5*k1p)) +
            g*np.cos(2*np.pi*(t+0.5*dt)))
        k3q = dt*(p + (0.5*k2p))
        k3p = dt*(time_ind_force(q + (0.5*k2p),p + (0.5*k2p)) +
            g*np.cos(2*np.pi*(t+0.5*dt)))
        k4q = dt*(p + k3p)
        k4p = dt*(time_ind_force(q + k3p,p + k3p) + g*np.cos(2*np.pi*(t+dt)))
        q += (1.0/6.0) *(k1q + 2*k2q + 2*k3q + k4q)
        p += (1.0/6.0) *(k1p + 2*k2p + 2*k3p + k4p)
    return q,p
```

The following two functions use the above function to iterate the Poincare map, and generate the first n values of the section given the initial conditions. The second function will then plot section in the phase plane in order to visualize the dynamics of the system.

```
# Generates the first n values of the poincare map
    # time_ind_force:= the time independent force (q,p); function
    # qo:= the initial position; float
    # po:= the initial momentum; float
    # g:= the strength of driving force; float
    # n:= number of periods; int
def poincare(time_ind_force,qo,po,g,n):
    # initializes the arrays
    qvals = np.zeros(n)
    pvals = np.zeros(n)
    qvals[0] = qo
    pvals[0] = po
    # computes the evolution of each point and then adds that point to the array
    for k in range(1,n):
        q,p = one_period_sol(time_ind_force,qvals[k-1],pvals[k-1],g)
        qvals[k] = q
        pvals[k] = p
    return qvals, pvals

# Plots poincare map for damped driven pendulum over n periods
    # time_ind_force:= the time independent force (q,p); function
    # qo:= the initial position; float
    # po:= the initial momentum; float
```

```
    # g:= the strength of driving force; float
    # n:= number of periods; int
def poincare_plot(time_ind_force,qo,po,g,n):
    qvals, pvals = poincare(time_ind_force,qo,po,g,n)
    plt.scatter(qvals,pvals)
    plt.show()
```

3 Linear Oscillator

Before we consider a non-linear system, it is useful to analyze a system that has an exactly solvable solution. We will consider a damped-linear oscillator that has a sinusoidal driving force applied to it. For this, we choose a damping coefficient $\beta = 0.15$ and a natural frequency of $\omega_o = 1.2$. These were chosen so that the natural frequency is different from the driving frequency, as well as so that the system is under-damped. This gives the following equations of motion:

$$\ddot{q} + 0.30\dot{q} + 1.44q = \gamma\cos(2\pi t) \tag{2}$$

Since this can be solved exactly, we are able to write an explicit form for the Poincare map.

$$\mathcal{P}(q,p) = \begin{bmatrix} 0.420114q + 0.671301p - 0.0206887\gamma \\ -0.966675q + 0.218724p - 0.0189672\gamma \end{bmatrix} \tag{3}$$

The code for this map is:

```
# Poincare map for linear oscillator with natural frequency
# w = 1.2 and damping parameter B = 0.15
    # qo:= the initial position; float
    # po:= the initial momentum; float
    # g:= the strength of driving force; float
    # n:= number of periods to solve for; int
def lin_osc_poincare(q,p,g):
    qnew = (0.420114*q) + (0.671301*p) - (0.0206887*g)
    pnew = (-0.966675*q) + (0.218724*p) - (0.0189672*g)
    return qnew, pnew

def line_osc_poincare_plot(q,p,g,n):
    qvals = np.zeros(n)
    pvals = np.zeros(n)
    qvals[0] = q
    pvals[0] = p
    for i in range(1,n):
        qnew, pnew = lin_osc_poincare(qvals[i - 1], pvals[i - 1], g)
        qvals[i] = qnew
        pvals[i] = pnew
    plt.scatter(qvals,pvals)
    plt.show()
```

6 References

Blanchard, P., Devaney, R. L., & Hall, G. R. (2012). *Differential equations.* Boston, MA: Brooks/Cole, Cengage Learning.

Percival, I., & Richards, D. (1999). *Introduction to dynamics.* Cambridge: Cambridge University Press.

Taylor, J. R. (2005). *Classical mechanics.* University Science Books.