PHY254 Computational Assignment 2020

Due Tuesday, December 22nd by 11:59 p.m.
You can use multiple file submissions on Quercus,
spread over whatever time you like before the deadline.

The idea of this assignment is to give you more experience exploring nonlinear systems numerically, and to introduce you to some of the more sophisticated tools, like python's built-in ODE solver `odeint`. Numerical studies are a bit like experiments; you have to try things and report the most interesting things you observe. You don't need to hand in every plot you make, just a sample of the best results. Write a main document (which can be handwritten, or whatever you usually do) that answers the basic questions and clearly refers to the other files you upload. Name all your files in an obvious way, indicating which question they are supposed to answer, like "`Q1a.py`" etc. Label your plots and comment your codes, but don't go too crazy about it. Hand in the successive versions of the code that solve each part of a question, rather than just the final version. You can re-use any bits of the codes I have posted during the course. You can use online information, as long as you do not blindly copy someone else's code. Properly cite your sources in comments.

Use `.pdf` files for plots, but if these become too large (as may happen when plots have a very large number of points), you can tell python to use `.png` files instead.

1. **Solving oscillations numerically:** We will first explore the accuracy of the various ways that oscillatory equations can be solved numerically. We previously discussed the difference between the Forward Euler and Euler Cromer methods. Here, we compare Forward Euler and python's built-in ODE solver `odeint`.

   Consider a mass m attached to a spring of constant k sliding back and forth on the x axis which is frictionless. We know that this is an undamped simple harmonic oscillator which ought to conserve its total energy.

   (a) Write a program in python to solve this system by replacing the time derivatives with a Forward Euler finite difference approximation. For example, write

   $$\dot{x} = \frac{dx}{dt} = \frac{x(t + \Delta t) - x(t)}{\Delta t}$$

   and then rearrange the above x equation for $x(t + \Delta t) = ...$ To do this, you will have to choose a time step, as well as values of k and m. Choose your time step so that one period contains several (5 or more) time steps.

   (b) Solve this system analytically and plot the numerical and analytic solution for the position x, the velocity $v = \dot{x}$ and the total energy E against time, for the same time axis. Vary Δt and see what happens. What are the main differences between the analytic and numerical solutions?

Instead of manually writing a time stepping scheme every time we need to solve a differential equation, we can use `odeint` which is more accurate and faster than our simple time stepping. It also handles details like the size of the time step.

The manual pages for `odeint`, which is in the `scipy` library, are here:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html .
Note that `odeint` is based, ultimately, on `FORTRAN` routines!

To use `odeint`, we need to cast our ODE into a system of 1st order ODEs, as we often did in class. For the simple harmonic oscillator example that would mean defining the following variables:

We will set $v = \dot{x}$ so

$$\dot{x} = v \tag{1}$$
$$\dot{v} = -(k/m)x \tag{2}$$

In this way, our second order ODE is expressed as two first order ODEs in x and v; the right hand side terms represent the time rate of change of the variables. The ODE routine `odeint` solves ODEs that are set up in this way. The syntax for the function `odeint` is a little different from what you might be used to. It takes the form:

```
odeint(rhs,x0,t)
```

where `rhs` is another function which outputs the "right hand side" terms. You can use any name you want instead of `rhs`, but it is a good descriptive name.

To use `odeint`, you set the initial conditions in the vector `x0` (this is the starting value for all your newly defined variables put into one vector: $(x_0, v_0, ...)$ and set the array `t` to correspond to those times when you wish to see output. The routine will then figure out the best numerical integration technique to produce the solution at the desired times. Thus, the spacing of the points in array `t` will *not* influence the accuracy of the solution. The solver's set of techniques is beyond the scope of the course, but at heart it works similarly to our simple methods. In a more advanced course, or in your own work in the future, you should examine the workings of `odeint` and similar ODE solvers.

For more information on odeint you can upload the scipy package that has it and then ask for 'help' on it. For example, in a python shell type:

```
from scipy.integrate import odeint
```

then hit enter and type:

```
help(odeint)
```

and hit enter. This should bring up a help file with information on `odeint`.

(c) Now repeat part (b), except plot the output of the `odeint` function in addition to the analytic solution and the Forward Euler time stepping scheme. How do the results differ? Which do you think is a better numerical approximation?

2. **The van der Pol oscillator.**

In class, we discussed a prototypical smooth auto-oscillator, the nonlinear van der Pol equation. This oscillator has many applications, including to modelling neuron firing dynamics and the motion of human vocal cords. We previously solved this oscillator numerically using crude time stepping. Now we return to this problem to study it more carefully, including the effect of forcing.

The forced version of the van der Pol equation is

$$\ddot{x} - \mu(1 - x^2)\dot{x} + x = A\cos(\omega t) \tag{3}$$

The oscillator itself has only one parameter, μ, while the forcing adds two more, A and ω. As usual, we write $\phi = \omega t$, so that $\dot{\phi} = \omega$.

(a) **The unforced van der Pol oscillator.**
To get started, write a python code that uses `odeint` to solve the equation with no forcing (A=0), for any μ. Recall that we learned in class that the solution approaches a *limit cycle* for later times. Use your code to generate and plot time series of x and $\dot{x}$, and phase space plots in the 2D space of $(x, \dot{x})$. Do this for a selection of μ values with $0.1 \leq \mu \leq 15$ and various initial conditions. Avoid using zero initial conditions, as $(0, 0)$ is an unstable fixed point. Can this system ever be chaotic, for any value of μ? If so, what value? If not, why not?

(b) Devise a numerical means of estimating the period of the unforced oscillator for each μ. Make a plot of the numerical period T *vs.* μ for $0.1 \leq \mu \leq 15$ by adding a loop over μ to your code from part (a). (You may wish to look at the code that was posted as part of the solutions to Homework problems #2, in which some python tricks were used to estimate the period of the oscillation near a minimum of the Lennard-Jones potential). Only measure T for motion that is on the limit cycle.

It can be shown analytically that $T = 2\pi$ for small μ, when the motion is almost sinusoidal. For large μ, it turns out that T is almost proportional to μ as the motion approaches a square-wave like relaxation oscillation. A huge effort has been made over the years to calculate $T(\mu)$ asymptotically for large μ. The known leading terms are (take a deep breath)

$$T(\mu) \approx [3 - 2\log 2]\,\mu + \frac{3a}{\mu^{1/3}} - \frac{2}{3\mu}\log(\mu) - \frac{b}{\mu^2} + ... \tag{4}$$

with $a = 2.33810741$ and $b = 1.3246$. Compare your numerical result to these predictions by including the theoretical lines for large and small μ on your plot, keeping in mind that Eqn. 4 is only supposed to work for large μ. Do they agree at all? You may extend the range up to $\mu \sim 100$ to really test the theory.

[The first few terms of Eqn 4 were discovered by Mary Cartwright in 1952. Some of the theory of $T(\mu)$ is discussed on page 214 of this textbook. See also this 2018 paper.]

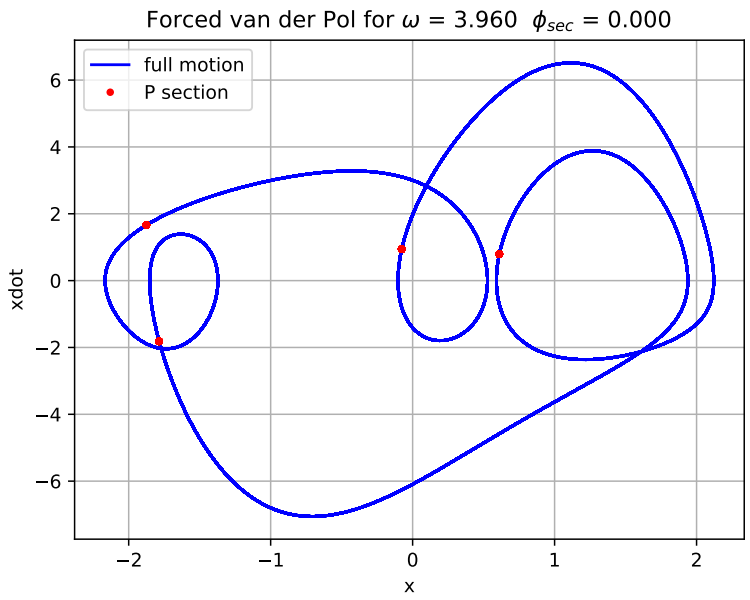

Figure 1: The phase space for the forced van der Pol oscillator with $\mu = 3$, $A = 15$ and $\omega = 3.96$, showing just the later stage of the motion near the attractor. This is a periodic state with period 4, as indicated by the red dots which show the Poincaré section for $\phi_{\rm sec} = 0$.

The forced van der Pol oscillator.

Now generalize your code to include the forcing term $A\cos(\phi) = A\cos(\omega t)$. The phase space is now 3D, spanned by $(x, \dot{x}, \phi)$.

To keep things as simple as possible, fix $\mu = 3$ and $A = 15$ and just vary the frequency of driving ω in the range $3.90 \leq \omega \leq 4.10$. As you will see, this is already very complex! It turns out the forced van der Pol oscillator has multiple attractors with multiple basins of attraction, which are selected by different initial conditions. To make sure we all fall in the same basin, let's also fix our initial conditions to be $x_0 = 2$, $\dot{x}_0 = 0$ and $\phi_0 = 0$.

Guessing good values of ω for the following questions are easier *after* you have seen the results of part (e), but it is more straightforward from a code development point of view to try hunting around visually first. You can come back and look again when you have done part (e).

(c) Use your generalized code to visually hunt for periodic or chaotic motions by plotting x and $\dot{x}$ *vs.* t and the $(x, \dot{x})$ projection of the 3D phase space, varying only ω in the above narrow range. Note that you will need to only plot the later part of the motion, and integrate for a long time, in order to be near the attractor. You will also need to use enough points to keep the plots smooth. The projection of a periodic limit cycle will look like a reasonably simple closed curve, which might cross itself, while chaos will look like a dense scribble. Counting the crossings tells you something about the period of the limit cycle relative to ω. Make labelled plots of some of the interesting states you find.

As an example, you should be able to reproduce the blue curve in Fig. 1

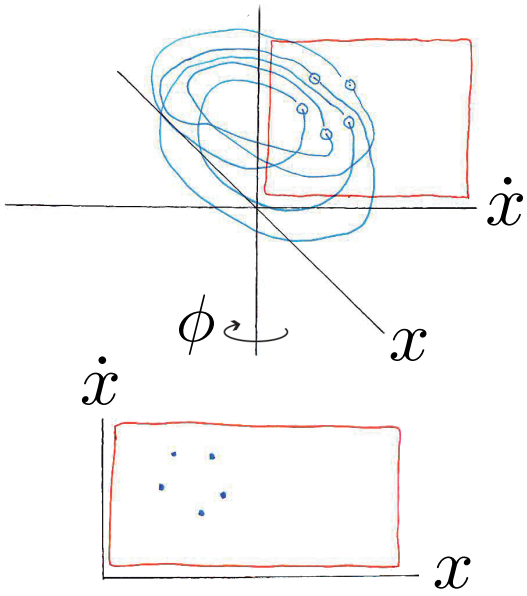

Figure 2: A schematic diagram of a Poincaré section for a periodically forced system. This shows the periodic variable ϕ as an angle wrapped around an axis, rather than in a periodic box, as we drew in class. But the idea is the same. We count the crossings of a plane of constant ϕ (the red box) as points in the Poincaré section.

Poincaré sections.

The 3D phase space $(x, \dot{x}, \phi)$ is difficult to visualize, even in projection. We can gain some insight into the dynamics using a Poincaré section which is a two dimensional slice through a higher dimensional phase space. Each time the trajectory intersects with our surface we make a point. This is not the same as a 2D projection of a 3D phase space, but something new and potentially simpler. This technique was introduced by Poincaré during his study of the three body problem.

In a periodic system, the motion repeats itself after a certain amount of time so the Poincaré section is made up of a finite number of discrete points (Figure 2). If there is more than one frequency in the system and the ratio of these two frequencies is not rational, then the system will never repeat exactly. In this case, the trajectory is constrained to lie on a closed surface in the shape of a torus in phase space and a Poincaré section will be a slice of that torus. This means that the set of points in figure 2 will be a closed curve (if you compute a long enough time series to fill it in).

Finally, the system may not be constrained by its dynamics to lie on any simple phase space surface and a complex, possibly fractal, Poincaré section will emerge. In this case, the system is in a fully chaotic state. Not all chaotic Poincaré sections are obviously fractal; the ones for our driven van der Pol oscillator are not very fractal looking, unfortunately. But they are, at small enough scales.

We will consider the Poincaré section formed by x, $\dot{x}$ at a constant $\phi \mod 2\pi$. We have provided you with a python function that return a list of points which comprise a Poincaré section. To use this function, put `poincare.py` in the same directory as your python script and add `import poincare` at the top of your script.

The usage of this function is as follows:

`poincaresection(x,xdot,phi,t,omega,phimod2pi)`

- `x`, `xdot`, `phi`, `t`: Arrays containing the time series output of `odeint`.
- `omega`: The frequency that the system is being forced at.
- `phimod2pi`: This is a number between 0 and 2π which specifies the coordinate where the Poincaré section should be plotted. This corresponds to the angular coordinate of the red box in Figure 2.

Note that you need to let `odeint` run for long enough and output enough time steps in order to get enough points on the Poincaré section. This may take a while to run.

(d) Now modify your code for plotting the phase space to call the `poincaresection` function. Choose a value of `phimod2pi` $= \phi_{\text{sec}}$ at which to make the section. Plot some Poincaré sections at interesting values of ω and ϕ_{sec}. You can also plot the Poincaré section as red dots on top of the blue phase space trajectory, as in Fig. 1. Try varying ϕ_{sec} to see how the section simplifies the messy trajectory.

Hint: The Poincaré section for $\omega = 4.07, \phi_{\text{sec}} = 0$ looks like a duck.

(e) **The bifurcation diagram.**
Finally, we can try to visualize the global dependence of the Poincaré sections on the parameter ω, the frequency of the driving force, using a tool called a **bifurcation diagram**. We saw some of these in class. They can also be fractal.

Make a large number (at least a few hundred) Poincaré sections for $3.90 \leq \omega \leq 4.10$, with fixed $\phi_{\text{sec}} = 0$, $\mu = 3$, $A = 15$, by looping over ω. Plot all the x values you find in each section as a function of ω. This means you should have x on the vertical axis and ω on the horizontal axis. (Plotting $\dot{x}$ instead of x works equally well). You should see clearly the periodic and chaotic regions that emerge as a function of ω.

This may need to run for quite a while, so debug with fewer points and work up. Plot enough points to see the bifurcations clearly, using a small enough marker size. You may want to zoom in on shorter ranges of ω.

What does this plot suggest about how the chaos arises in this system?

Once you have seen this plot, you may wish to go back and try some different values of ω in parts (c) and (d).

I hope you have enjoyed doing this assignment!