Computational Physics Final Project - Dripping Tap

Simulation coded in C, analysis and plotting done in Python

The goal of this project was to recreate the model and key results of a dripping tap from the paper "Dripping Faucet" by "A. D'Innocenzo and L. Renna, 1996", in particular figures 2 & 4 from said paper.

The Model

They modelled a dripping tap by treating the drop as a mass on a spring, increasing in mass as a rate R, which would drip once the spring stretched to a critical value Xc. This can be seen in the following equations of motion:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v$$
$$M\frac{\mathrm{d}v}{\mathrm{d}t} = Mg - kx - (R+b)v$$
$$\frac{\mathrm{d}M}{\mathrm{d}t} = R$$

Using cgs units, x is position in cm, M is mass in grams, v is velocity in cm/s, R is the flow rate ml/s (assuming the density of water is 1g/cm^3), g is the gravitational acceleration of earth, k is the surface tension of the drop (or, alternatively, the spring constant) in dynes/cm, and b is a parameter related to friction, with units of grams/s.

3 additional important equations are: (1) $\Delta M = \alpha M_c v_c$, (2) $\Delta M = \alpha v_c$, and:

$$x_0 = x_c - r\frac{\Delta M}{M_c}$$

(1) and (2) are the 2 different methods for reducing the mass upon a drip occurring, both containing a constant of proportionality, alpha, which is different depending on the reduction method chosen. Mc and xc are the critical values at which a drip occurs.

The third equation calculates what position the remaining mass should return to after a drip, based on a single sphere of water dripping. $r = (3\Delta M/4\pi\rho)^{1/3}$ where rho, the density of water, is taken to be 1.

The values of all of the parameters are contained in my run_params1 and run_params2 files (2 files as there are 2 different values of alpha used) and those parameters are summarised in the table on the next page:

α (g/(s cm))	Initial mass m (g)	Initial velocity v (cm/s)	Surface tension k (dynes/cm)	g (cm/cm^2)	b (g/s)	Critical length xc (cm)
0.25 (1)	0.01	0.1	475	980.665	1	0.19
0.025 (2)						

The code

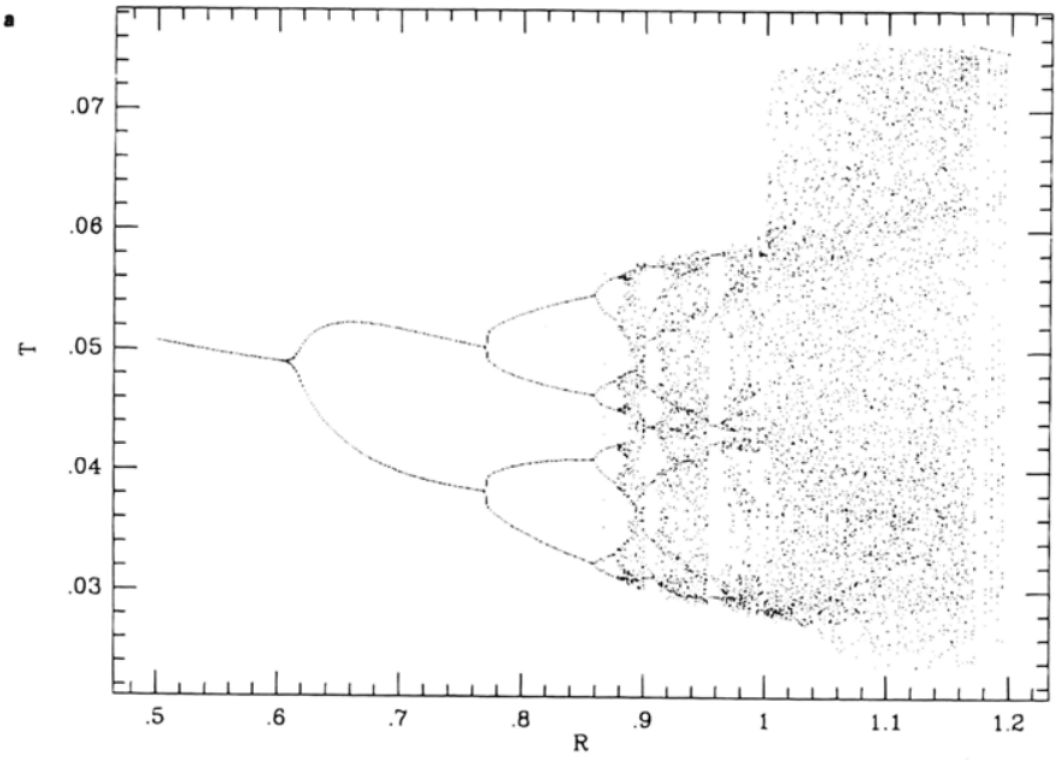
The code works as follows: There is a for loop which loops over a regime of flow rates, this is set by default to start at 0.5, and to increase to around 1.3. This was originally set to loop until 1.5, however, flow rates above 1.3 start encountering issues (this will be elaborated on later). This for loop is parallelised using OpenMP. Within this loop, the solver function is called, which contains a while loop to loop over time for the actual simulation, which is performed by GSL's ODE solver. By default GSL's solver is set to use a Runge-Kutta 4 but I have tested it with other stepping mechanisms. This was written using the examples on GSL's ODE solver page:
https://www.gnu.org/software/gsl/doc/html/ode-initval.html
The solver runs until a drip occurs, ie, when x exceeds xc. At this point, a check stops that solver and frees it, and then a function is called to again use GSL's solver to perform an integration backwards, to find the precise point at which the tap dripped, as suggested in M. Henon (1982), and as mentioned in the paper being followed. At this point, using the values obtained for the drip point, the mass is reduced, the position is reset, and the data is written to a file. The original solver is then set running again, continuing from the point of the drip. This process will repeat until the while loop reaches a specified stop time, or a failsafe maximum number of iterations. Once that is reached, the flow rate will increase, and the solver will run again, until the maximum flowrate is reached.

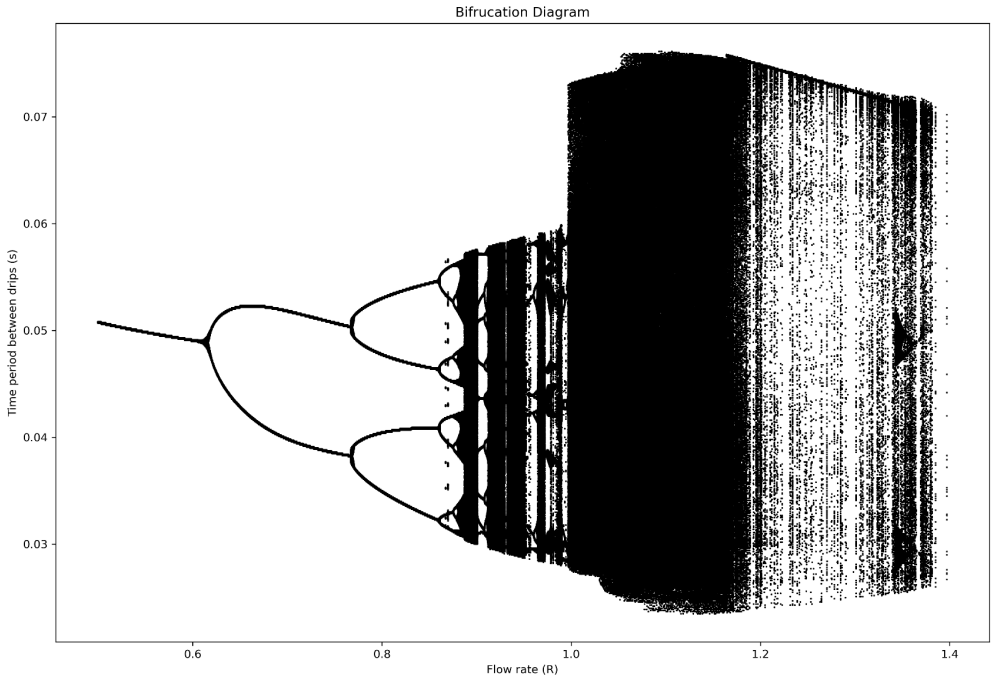
In python, simple code is written to read the data from the output files, and to loop through the data to extract the time periods between drips, as is necessary for producing the bifurcation map.

The Results - Recreation of Figures 2 & 4

All plots were produced using a Runge-Kutta 4
The next couple of pages will contain comparisons between the paper's figure 2, and my resulting bifurcation maps trying to recreate these figures. Key differences to note include that my plots go up to a flow rate of 1.5, as opposed to the paper's flow rate stopping around 1.2. Data above a flow rate of 1.35 or so may be invalid, due to the solver producing errors at such high flow rates, however, the resulting points on the bifurcation maps seem interesting to me, so those flow rates have been included. Other differences include my maps looking a lot darker in the regions where there are points, as I have plotted many more than just the 50 points per value of R that the paper did. Of note as well is that the paper ran each simulation for 200 drips, and discarded the first 150 points, as the initial drips can produce spurious points as the simulation gets started. In contrast, to remove these dodgy data points, I only plotted drips that occur after the first 10 seconds of simulation time. Drips

before t = 10 were ignored when plotting. In all cases that follow, the plot from the paper will be above, and my own recreation will be below.





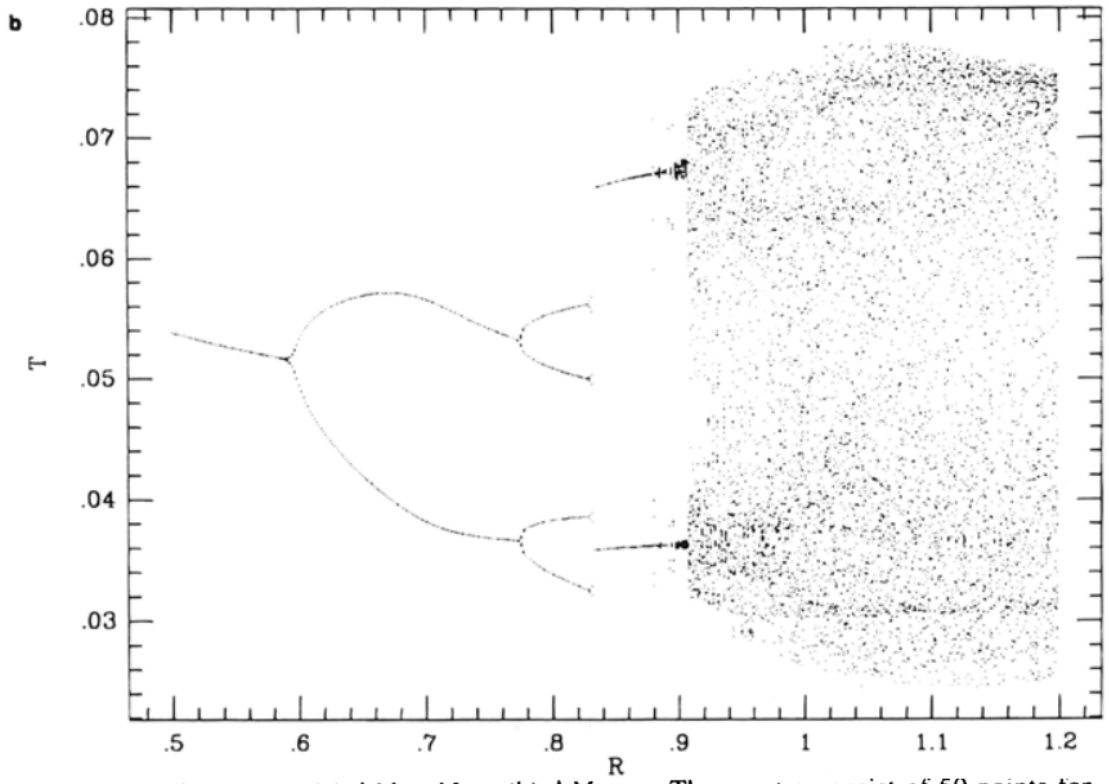
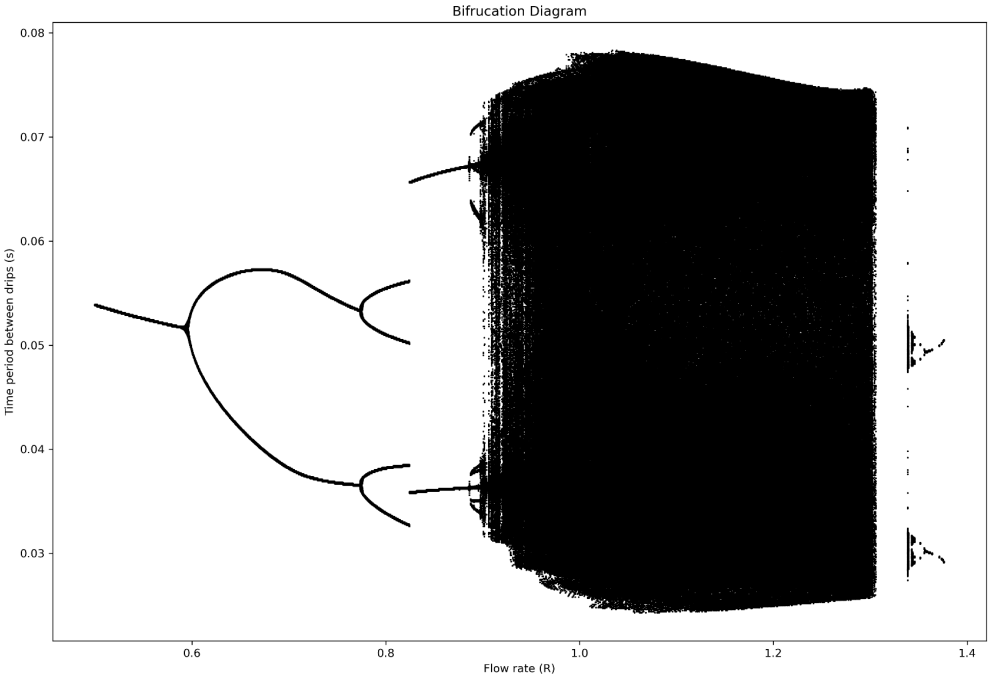
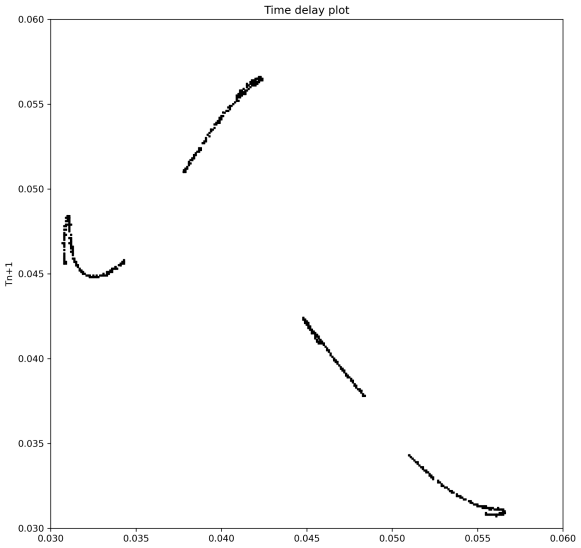
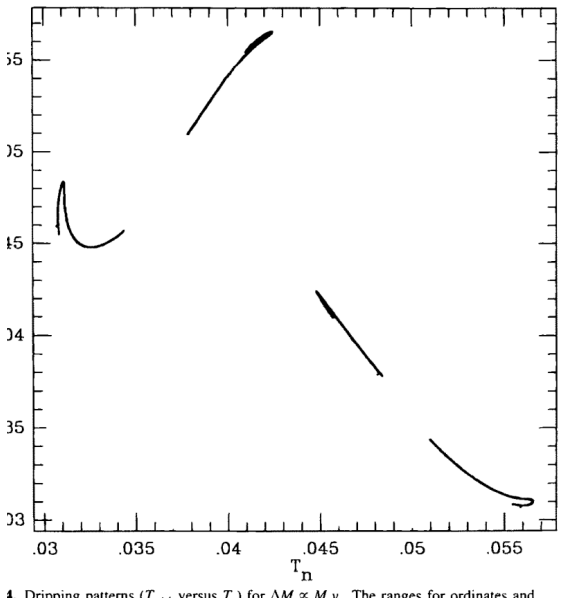


Fig. 2. Drip spectra: (a) $\Delta M \propto M_c v_c$; (b) $\Delta M \propto v_c$. The spectra consist of 50 points for each R value. The values of the other parameters are given in the text.

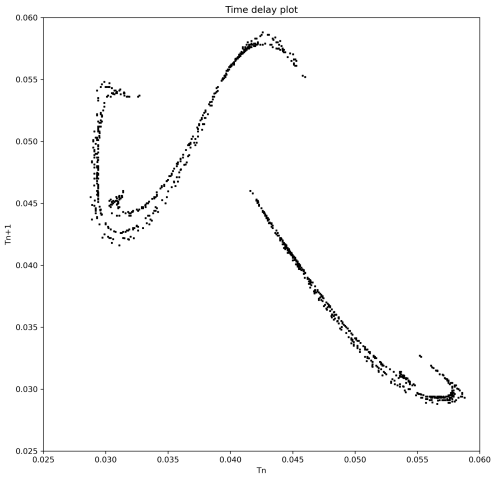


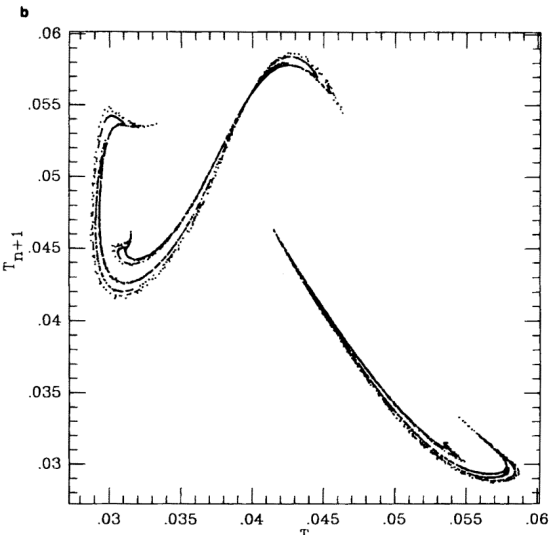
As can be seen, I have managed to recreate both plots from figure 2 to a high degree of accuracy, and in more detail than the original paper. In the second plot, a small region around 1.35 can be seen to perhaps stop being chaotic, however it is hard to draw conclusions, as at these flow rates, the solver quickly starts producing errors. This is likely because at large flow rates, the model of a mass on a spring breaks down, and an approach closer to the Navier-Stokes equations may be necessary to characterise this behaviour. It is possible then that the few points seen around 1.35 characterise the few initial drops usually seen to come from a tap before smooth flow begins, however, this is just speculation on my part. Further verification of my plots adherence to theirs will be discussed later, when talking about my recreation of figure 3 of the paper.

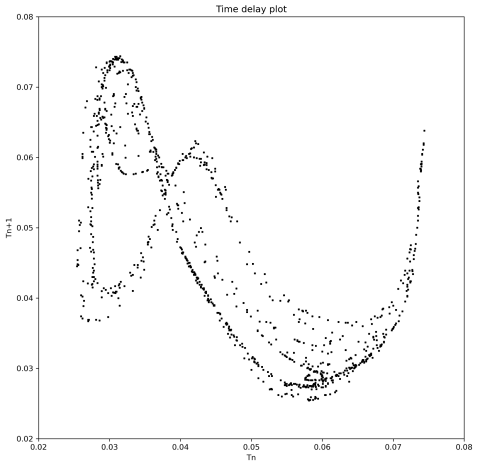


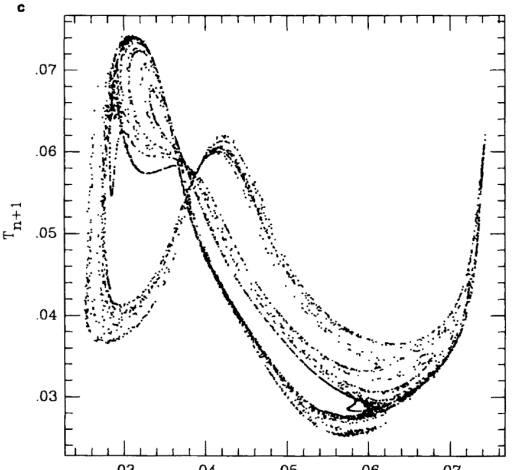


1 Dripping patterns (T_{n+1} versus T_n) for $\Delta M \propto M_d$. The ranges for ordinates and





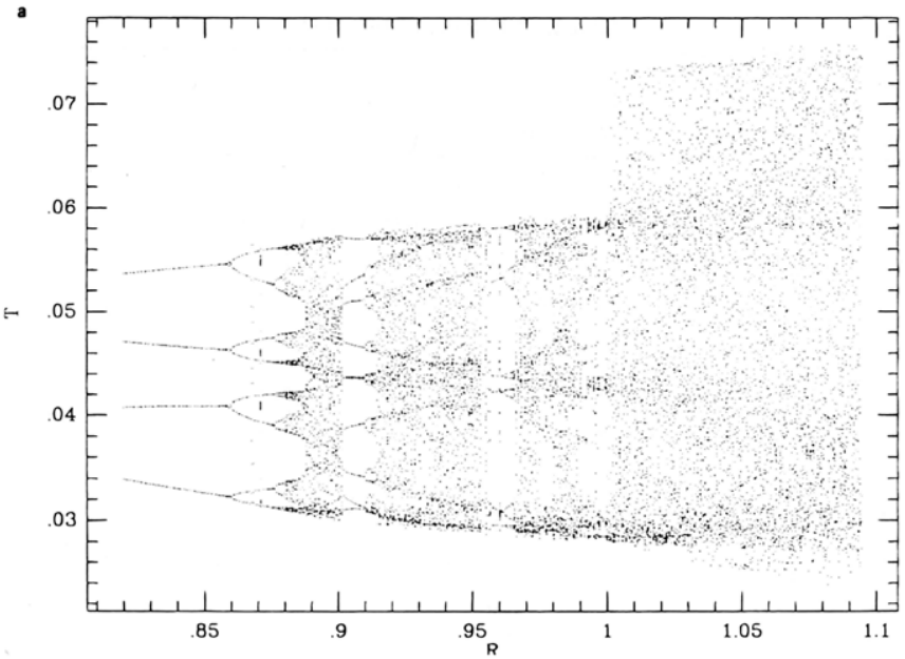


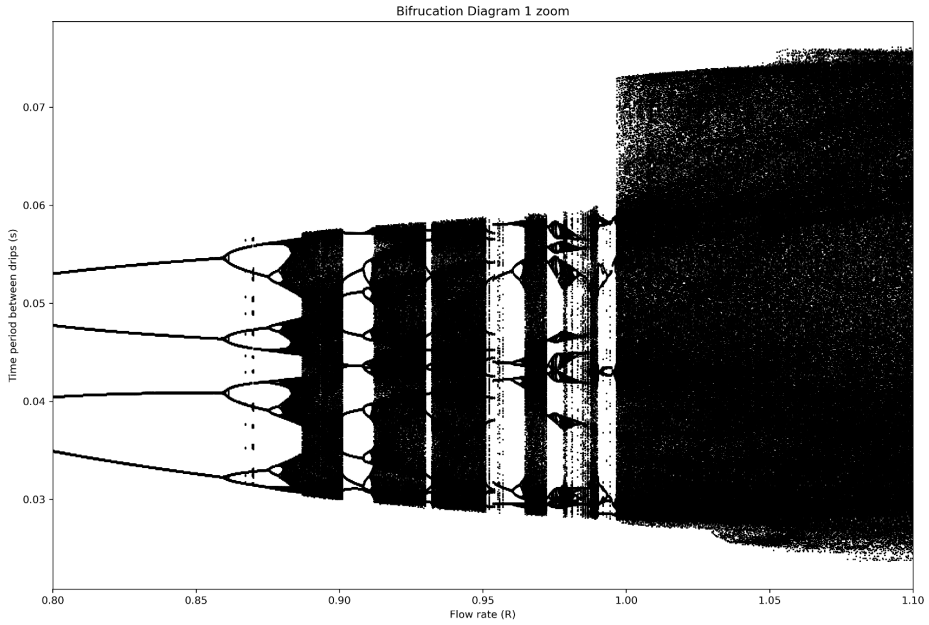


The 3 comparisons above are comparisons to figure 4 of the paper, and are time delay plots, with mine on the left and the paper's on the right. As can be seen, the shapes of mine match the ones from the paper pretty much exactly, albeit in less detail as I appear to have plotted less points. I am unsure as to why this is, it is possible they ran the simulation for longer at these given values of R.

Other comparisons, efficiency analysis, and mass conservation

I have recreated figure 3 of the paper, as it is a zoom in on the most interesting regions of the bifurcation map. Additionally, it helps to illustrate how similar my plots are to those of the paper. That comparison will be shown below:





These are zooms of the first set of plots from figure 2. As my plot contains many more points than theirs, it brings out more detail in the brief non-chaotic regions between large regions of chaos.

The comparison between the zooms of plot 2 of figure two is:

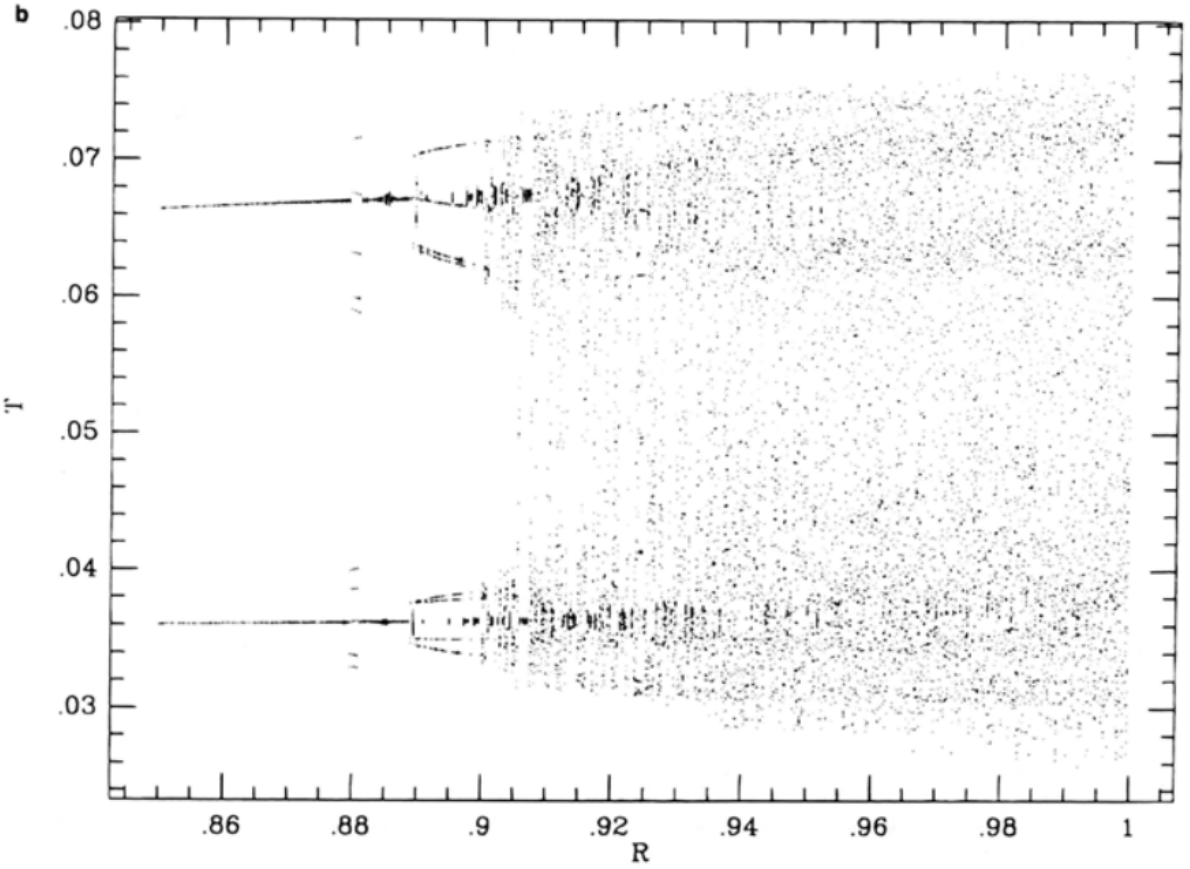
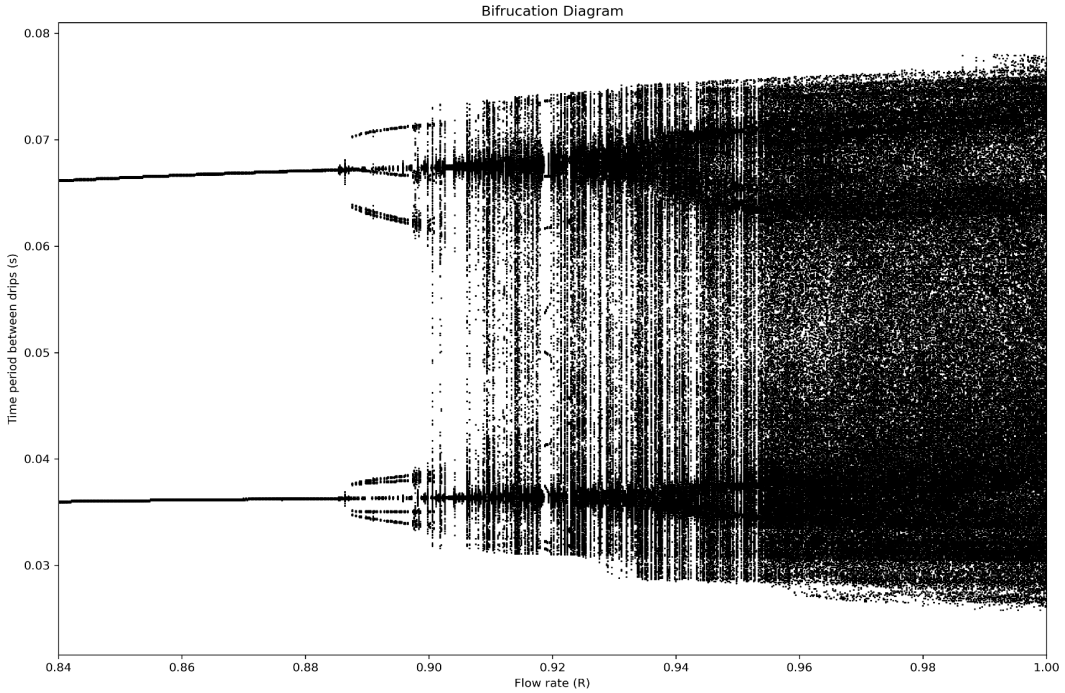


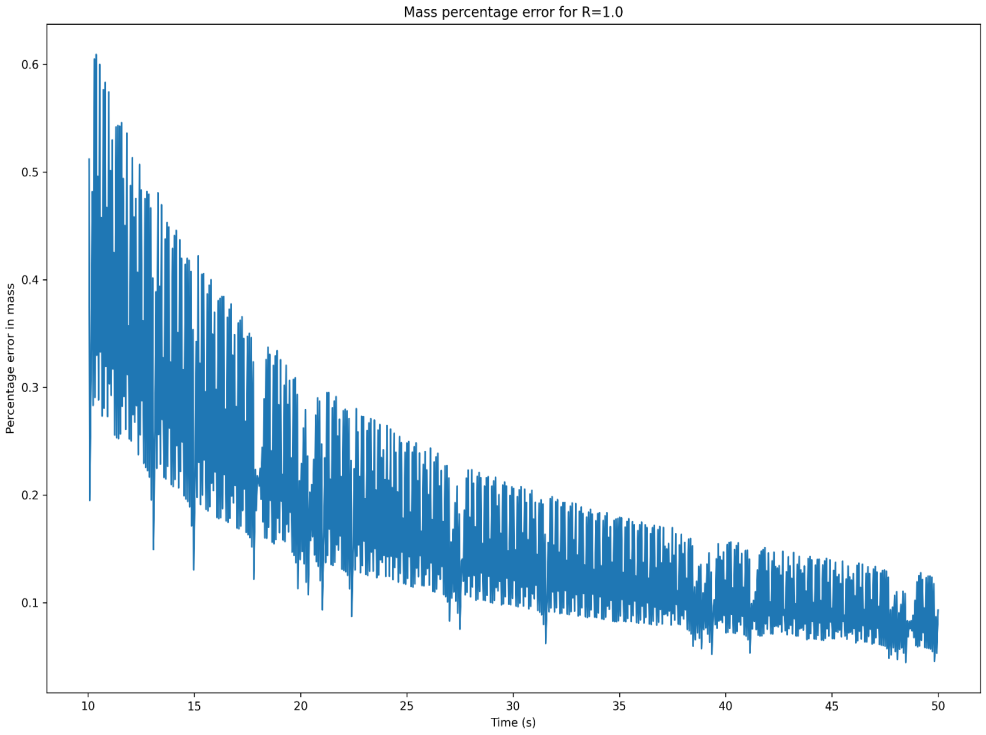
Fig. 3. Enlargement of spectra of Fig. 2.



As this plot is mostly just a simple 2 periods, or chaos, it is hard to note any meaningful comparison other than my chaotic regions being darker, due to a larger density of points. However, it can be seen that both plots contain the characteristic cone-like shape right before the chaotic region. Again, these help to verify that I have produced the same bifurcation maps, just with higher point density.

Lastly, 2 other things to note are mass conservation, and an efficiency comparison between different solver types in GSL.

Firstly, mass conservation:

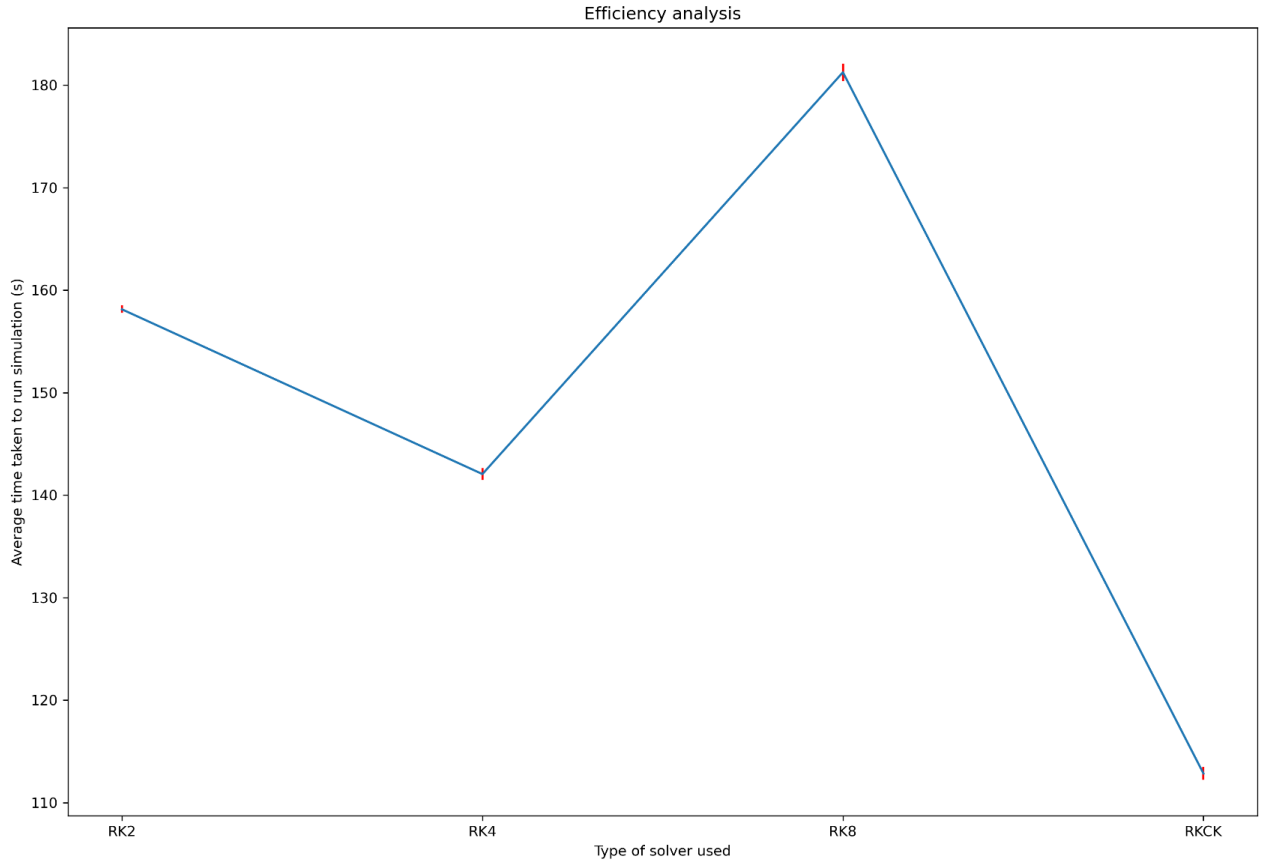


This is the percentage error on the mass of the water in the simulation. This was calculated by, at each drip time, calculating how much mass went into the simulation since the last drip by multiplying R by the time since the last drip, subtracting how much mass should be lost by each drip, and dividing that by the mass in (then multiplying by 100 to get a percentage). The mass in and mass out are never reset, and so it is the total mass in minus the total mass out, since the simulation began. It is interesting that the error decreases with time, as if there was a constant small amount of difference between the mass in and mass out, the error would be constant too, however, it seems that the difference between the mass in and the mass out decreases with time.

The first ten seconds have been omitted from the plot, because as has been noted previously, the data produced by the initial few seconds is spurious. However, it is worth noting that said spurious data was included in the calculation of the mass error (due to the method calculation) which may explain the decrease in error over time, as originally the error starts high (due to the first few drops),

and then stabilises to almost zero once the simulation stabilises, leading to a decreasing percentage error over time.
Regardless, I am happy to conclude that the simulation conserves mass, and the paper notes that v is left unchanged by the drop in order to conserve momentum.

Lastly, the efficiency analysis:



4 types of stepper were used. A Runge-Kutta 2, a Runge-Kutta 4, a Runge-Kutta Prince-Dormand (8, 9), and a Runge-Kutta Cash-Karp (4, 5). These were each timed using the bash time command, recording the user time. I timed each type 6 times, and then took the mean and standard error and plotted those, with the standard error displayed as the red error bars. The raw data for this can be seen below, and in the supplied excel sheet.

	A	B	C	D
1	RK4 User time	RK8	RK2	RKCK
2	142.744	180.963	157.966	112.353
3	141.103	183.019	157.681	112.58
4	142.407	180.999	158.478	111.912
5	141.837	181.435	158.453	113.029
6	142.631	180.507	158.504	113.725
7	141.579	180.586	157.664	113.473
8				

Overall, it can be seen that the fastest method was by far the Runge-Kutta Cash-Karp 4th order method, seemingly owing to its suitability to adaptive stepsize algorithms, which is what was being

used by GSL's solver. Second fastest was the simple Runge-Kutta 4th order method. The runge kutta 2 being slower than the runge kutta 4 may seem odd at first, however I believe it is due to the gsl solver using an adaptive timestep, as although the RK2 requires less evaluations, in order to converge, it will also require more timesteps, and as such, it takes more time overall due to the larger number of timesteps being done. Lastly, the slowest was the RK8 method, which is probably due to the large number of function evaluations slowing the simulations down.

Conclusion

Overall, I am happy to conclude that I have faithfully recreated the figures from the paper as requested, and that my code simulates a dripping tap as correctly as the original paper does.