Lab assignment #1: Python basics

Nicolas Grisouard (Instructor), Mohamed Shaaban (TA & Marker) and
Mikhail Schee (TA)

Due Friday, September 18th 2020, 5 pm

---

1. **Modelling a planetary orbit**

   (a) Nothing to submit.

   (b) Pseudocode:

      i. Define constants G and M_s, using AUs, solar masses and years as units,

      ii. choose time step $\Delta t \in \mathbb{R}$, and a number $N \in \mathbb{N}$ of time steps to execute,

      iii. initialize an array of length N containing all values of t.

      iv. initialize empty arrays of lengths N for v_x, v_y, x, y, and set the first value of each to their initial conditions (eqns. 10 in text).

      v. compute $r = \sqrt{x^2 + y^2}$,

      vi. Update velocity first: `vx[i+1] = vx[i] - Dt*G*M_s*x/r**3`, same for `vy`,

      vii. Update positions with the new velocities, according to the *Euler-Cromer* method: `x[i+1] = x[i] + vx[i+1]*dt`, and same for `y`.

      viii. Once loop is finished, compute angular momentum per unit mass of planet $\sigma = xv_x - yv_y$, and

      ix. plot.

   (c) For code, see `L01-407-2020-Q1-sol.py`, with the setting `alpha = 0.` at the beginning of the file.

   Fig. 1 shows the velocity components of Mercury as a function of time. Note how the velocities are asymmetrical. This is because the trajectory is elliptical, i.e., the planet spends less time in high-velocity/Sun-close regions.

   Fig. 2 shows the trajectory of Mercury. It does looks elliptical, though this fact is not apparent without adding a few visual cues. To that effect, I added the $x = 0$ and $y = 0$ axes to highlight the fact that the curve is not centred. You can also verify the slightly off-unitary aspect ratio by eyeing the $x, y = 0.4$ AU marks.

   Now, onto the angular momentum $(\vec{r} \times \vec{v}) \cdot \hat{z} = xv_y - yv_x$, where $\vec{r}$ is the vector from the Sun to Mercury, $\vec{v}$ is Mercury's velocity, and $\hat{z}$ the directing vector of Mercury's orbit

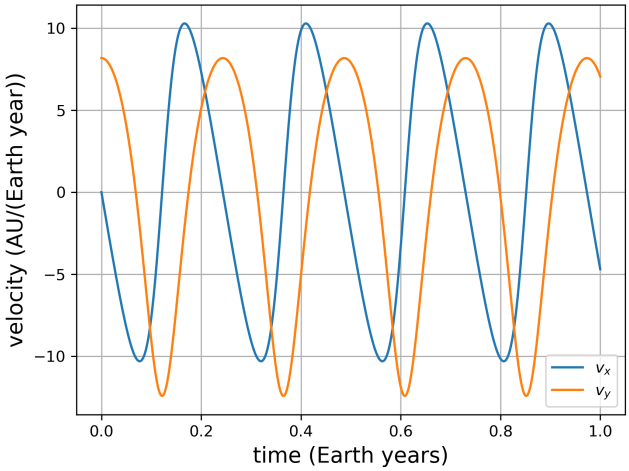

Figure 1: Cartesian and Newtonian velocity components of Mercury.

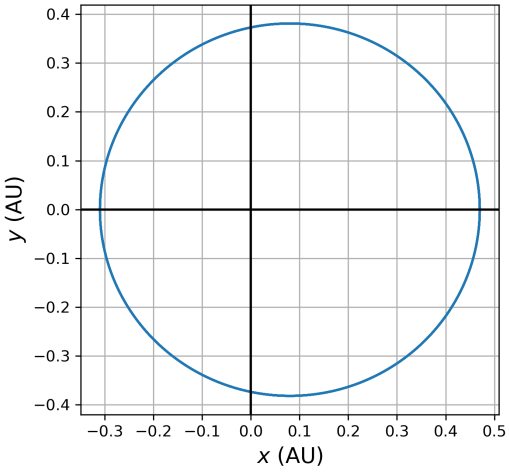

Figure 2: Newtonian trajectory of Mercury.

plane (note that I dropped Mercury's mass: we only want to check that the vector is constant, and we are assuming Mercury's mass constant). I show this plot in fig. 3.

As far as the eye can tell, it is constant *(Note: you can also show it by computing the spread numerically, rather than relying on your eyes).* Note the offset of 3.8399 AU^2/yr indicated on top of the y axis, meaning is it constant within 10^{-13} at least.

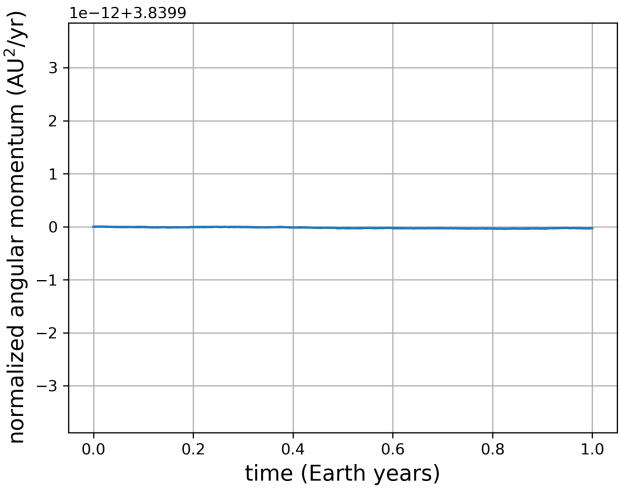

Figure 3: Newtonian angular momentum of Mercury.

(d) For code, see `L01-407-2020-Q1-sol.py` (same as before), with the setting `alpha = 0.01` at the beginning of the file.

Figs. 4 show the velocity components and trajectory of Mercury as a function of time, when $\alpha = 0.01$. Notice the shift in time: this is the precession of Mercury's orbit. Try and increase N (the number of time steps), and you will see a regular pattern over time.

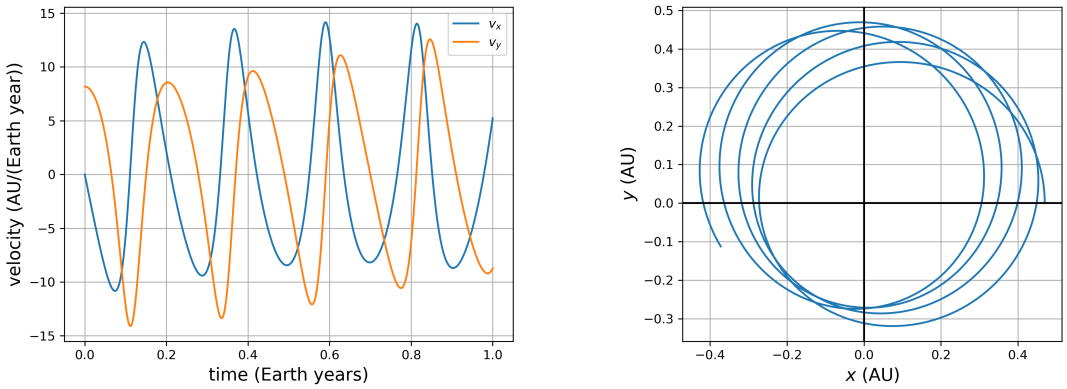

Figure 4: Cartesian, relativistic velocity components of Mercury (left), and corresponding trajectory of the planet (right).

Fig. 5 shows the new angular momentum, which is as constant as before.

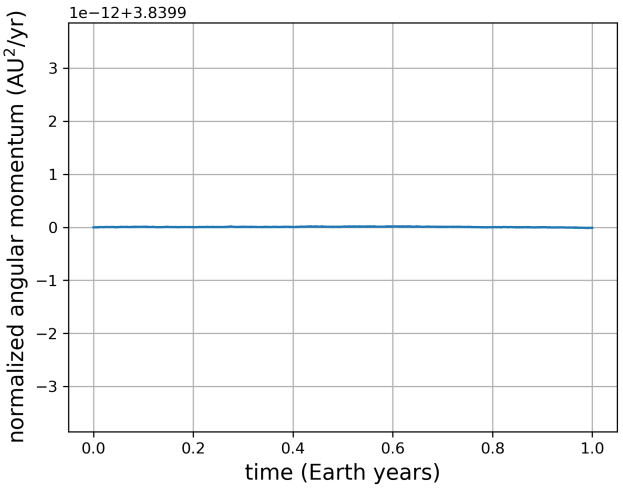

Figure 5: Relativistic angular momentum of Mercury.

2. **Bifurcation and chaos** See `L01-407-2020-Q2-sol.py` for the code.

   (a) Here is an example, storing everything into lists. `NumPy` arrays would have been a more natural choice, but for a simple application such as this, it does the trick.

      i. Define the numerical values of r, x_0 and p_{max}.

      ii. Initialize the list of populations with a one-element list, said element being the initial population.

      iii. Create a for loop.

      iv. In loop, store last population as temporary variable, for clarity.

      v. To the list of populations, append the new population computed from the logistic map.

      vi. Plot x_p, adding a grid and labels.

   (b) Nothing to submit.

   (c) The left panel of fig. 6 shows that for $r = 2$, the normalized population quickly stabilizes to a constant value of about 0.5. As the maximum reproduction rate increases however, the behaviour becomes more complicated. For $r = 2.5$, the population oscillates a bit, but then stabilizes again, this time at about $x_\infty \approx 0.6$. For $r = 3.1$, the system eventually oscillates around a value that is slightly higher than 0.6. For the higher values of r that I plotted, the system enters a series of large fluctuations that are seemingly unpredictable.

   (d) See the right panel of fig. 6 for the bifurcation diagram.

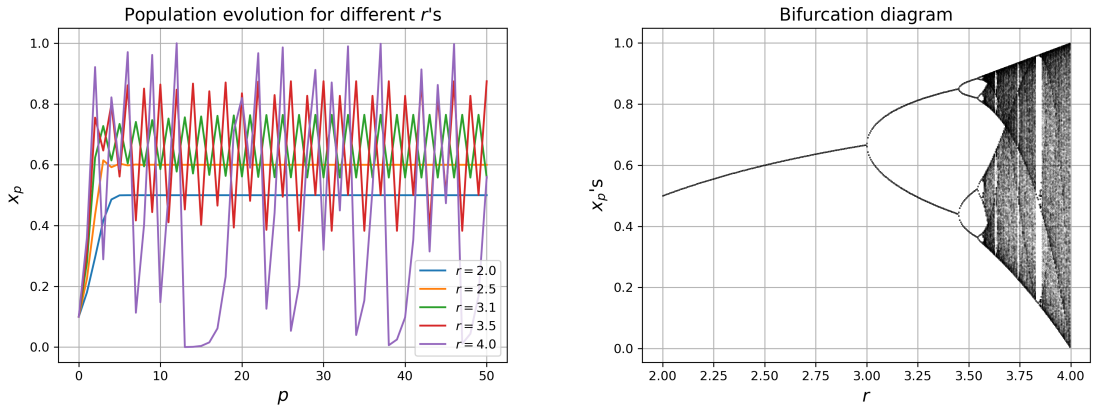

Figure 6: Left: Evolution of the insect populations for a few different maximum reproduction rates r. Right: Bifurcation diagram, i.e., values visited by x_p lists as a function of r.

- For $r < 3$, we see that the last values of x_p are all the same, indicating the stabilization to a point attractor we mentioned in the previous question.
- For $3 < r \lesssim 3.45$, there are two point attractors and in connecting with the result for $r = .13$ above, we can infer that the system oscillates between these two values.
- The $r \approx 3.45$ value exhibits a transition past this point, the system visits not 2, but 4 different values over the same number of years. Assuming that similarly to the previous case, the system visits these four points in succession and periodically (you can check that these points are indeed periodic points on the x_p vs. p plot for $r = 3.5$), it takes twice as long for the system to go back to a given value, hence "period doubling".
- The system then experiences a few more period doubling transitions, then states where the system visits regimes around a few countable values, but by $r = 3.7$, it appears that the system is chaotic (note that chaos and strange attractors can be tricky concepts and should be handled with care). This would correspond to the $r = 4$ curve of the previous question.

(e) Case in point, about how chaos is a tricky concept: in this region, the system transitions from what seems to be fully unpredictable (though bounded; so, still an attractor), to a series point attractors, period doublings, and finally back to fully unpredictable-but-bounded. This fractal structure is typical of strange attractors.

(f) Based on fig. 6 right and our answer above, we chose $r = 3.75$ to answer this question. We also limited the maximum value of $\max(|\epsilon|)$ to be 10^{-7} (i.e., 10^{-6} times the initial populations x_0), defined as

```
epsilon = 2e-7*(random()-0.5)
```

We show an example of two evolution histories on the left panel of fig. 7. Firstly, we can see that the two trajectories are visually indistinguishable until about $p = 40$, wihch gives an indication of how many times we need to iterate before the divergence between

the two curves statistically saturates. We actually ran the code until $p_{max} = 70$ to see the saturation, though it is clear that 40 was enough.

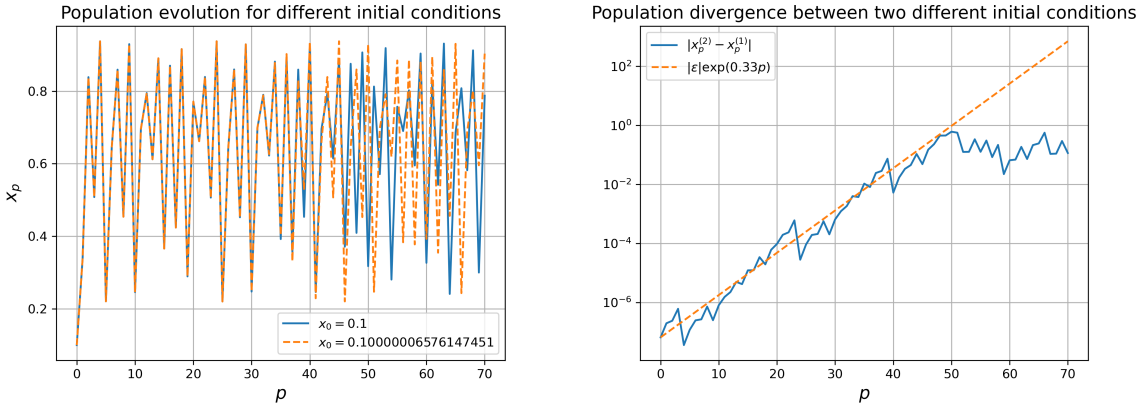

Figure 7: Evolution of two populations when $r = 3.75$ and $-10^{-6} \leq \epsilon/x_0 < 10^{-6}$, where ϵ is the difference in initial populations. Left: $x_p^{(1)}$ and $x_p^{(2))}$; right: $|x_p^{(2)} - x_p^{(1)}|$.

(g) See fig. 7, right panel: a Lyapunov exponent $\lambda = 0.33$ did the trick. Oh and by the way: $a = |\epsilon|$ was sufficient.

3. **Timing Matrix multiplication** *Note: it is very difficult to get reliable performance on one's own computer. I would get a spike in the duration whenever I opened a big application, compiled this LaTeX file, or watched a cat video. I resorted to using a workstation that was powerful enough, not to be affected by Instabookchat.*

   See fig. 8 for the method, described in the text. As you can see, the second panel (time vs. N^3) shows a linear progression, as predicted in the textbook.

   See fig. 9 for a method that gives the same result, but with the `cdot` method of numpy. First, compare the range of the vertical axes with that of 8. The range for the "silly" method is 40 times higher that when using the `cdot` method. Moreover, the dot method is affected by big spikes that increase the range of values. It was likely related to the activity of the machine I was using, in which case using a dedicated high-performance computing machine would reduce the computation time even more.

   Also, the N^3 dependence of the execution time seems gone, but even if there was one, these spikes would probably mask it. We can't conclude on this aspect without using a more reliable machine.

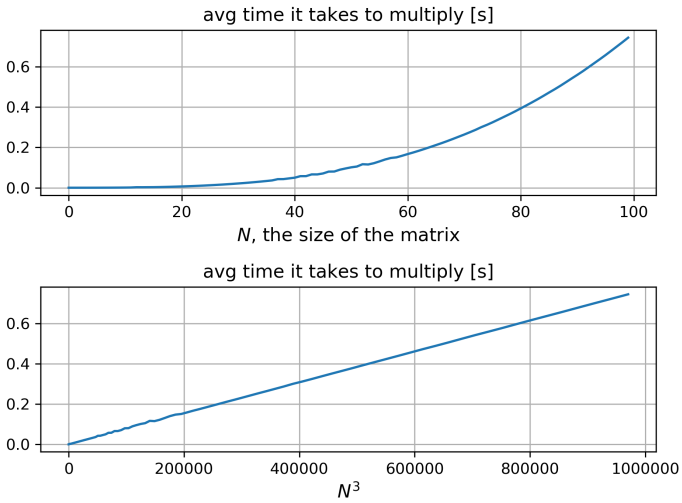

Figure 8: Average time it takes to multiply two matrices of sizes $N \times N$, with the method described in Example 4.3 of the textbook.

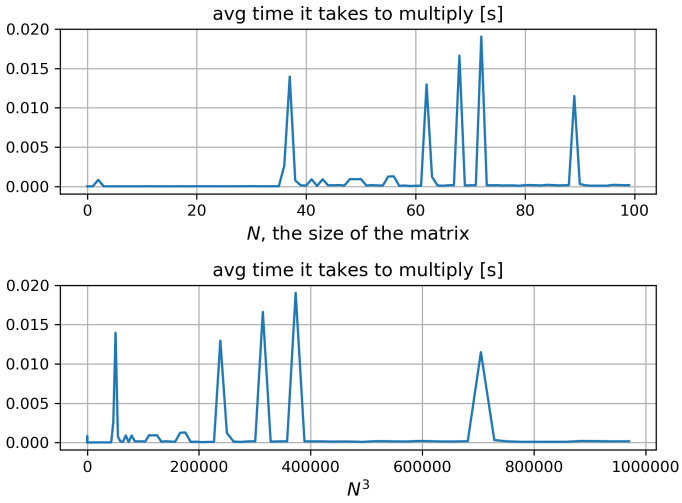

Figure 9: Average time it takes to multiply two matrices of sizes $N \times N$, with the `cdot` method of numpy.