Fractal dimension estimation for the double pendulum flip-time plot

Bogye, Balázs

2019. October 25.

Contents

1 Introduction

The double pendulum is a well known chaotic system. Simulating the motion and checking the chaotic behaviour is a common task and is not the aim of this project. One of the other common tasks is plotting the flip-over time of a double pendulum started without momentum as a function of the starting angles ($\theta_{1,2}$). On the said plot fractal-like patterns emerge. This project aims to analyse this pattern and estimate its fractal dimension.

2 Motivation

As I have said the creation of the said plot is a common task, one I already had in a previous class for an extra point. Although I managed to simulate it and get a marvelous looking image. I was unable to properly analyse it or find any information about it on the internet and have been looking for a time to do this.

3 Theory

3.1 Double pendulum

The double pendulum is made up of 2 pendulums attached to each other. For parameterizing its state I use θ_1 and θ_2, their respective angle to the vertical direction (0° is down) and p_{θ_1}, p_{θ_2}, their angular velocity. Additional constants are m, the mass, l, the length of the pendulums and g, the gravitational acceleration.

The motion of the pendulum, described by equations (1-4) is chaotic, meaning a small finite difference Δ can grow exponentially, $\Delta(t) \propto e^{\lambda t}\Delta$ (where λ is the Lyapunov exponent).

$$\dot{\theta}_1 = \frac{6}{ml^2}\frac{2p_{\theta_1} - 3\cos(\theta_1 - \theta_2)p_{\theta_2}}{16 - 9\cos^2(\theta_1 - \theta_2)} \tag{1}$$

$$\dot{\theta}_2 = \frac{6}{ml^2}\frac{8p_{\theta_2} - 3\cos(\theta_1 - \theta_2)p_{\theta_1}}{16 - 9\cos^2(\theta_1 - \theta_2)} \tag{2}$$

$$\dot{p}_{\theta_1} = -\frac{1}{2}ml^2\Big(\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) + 3\frac{g}{l}\sin\theta_1\Big) \tag{3}$$

$$\dot{p}_{\theta_2} = -\frac{1}{2}ml^2\Big(-\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) + \frac{g}{l}\sin\theta_2\Big) \tag{4}$$

Motion equations of the double pendulum

3.2 Fractal dimension

Fractal dimension is a scaling property, it is the D exponent of the number n of ϵ diameter objects (usually circles or squares) needed to cover a set of points (in our case a fractal).

$$n(\epsilon) \propto \epsilon^{-D}$$

$$D = \lim_{\epsilon \to 0}\frac{\log n(\epsilon)}{\log\frac{1}{\epsilon}} \tag{5}$$

So to estimate the fractal dimension we need to cover it with decreasingly smaller boxes, count the number of boxes needed and fit a power law to $n(\epsilon)$. This is called box counting (and

the dimension is the Minkowski dimension). We can get the dimension as seen in the previous equation (5) [3].

4 Methodology

First off I need to define the shape of the fractal. I set a "cutoff" time t_{cutoff}, every point with a flip-over time $t > t_{\text{cutoff}}$ is part of the fractal (see Figure 1). For my calculations I used $t_{\text{cutoff}} = 19$

For constants I used $g = 9.81$, $l = 9.81$, $m = 1$ (one can assume these are in SI units but it doesn't really matter for the shape).

As the system has a vertical mirror symmetry I only had to simulate half the plot.

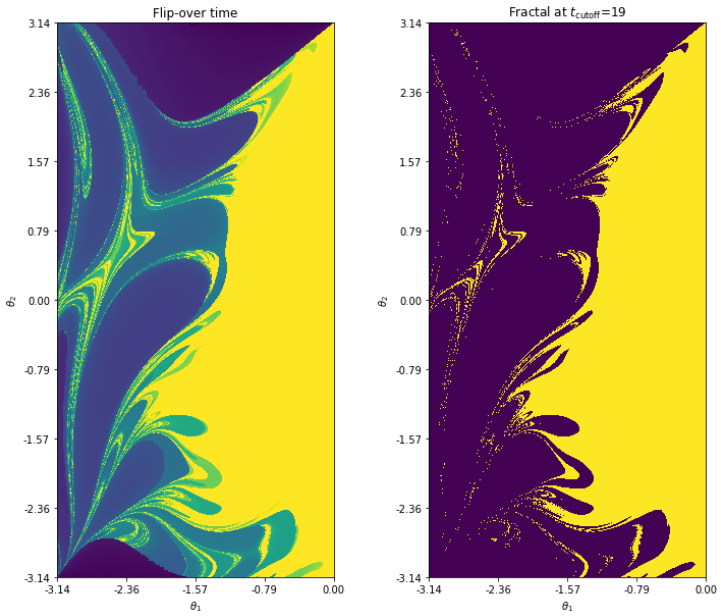

Figure 1: Getting the shape of the fractal by taking $t > t_{\text{cutoff}}$ points with $t_{\text{cutoff}} = 19$

4.1 Problems

Simulating 1 double pendulum is easy and fast, but simulating millions of double pendulums can take an extremely long time. I will need at least millions of pendulums to be able to accurately use the box counting method.

It is a chaotic system, so setting a low time-step is appropriate, but even so the errors can grow fast invalidating my data, the effect of integrational and numerical errors on the results should be analysed.

I can only calculate the flip-over time for a discrete set of points, technically I cannot tell for sure whether there is a high frequency change between two close points, so I will only find a subset of shape's points.

4.2 Multi-layered grid

To speed up the calculation process I only wanted to calculate as few flip-over times as possible. Calculating as little detail as possible, for example not using a 8192 points per radian grid

density for the central area ($\theta_{1,2} \approx 0$) of the shape, where t is definitely always more than t_{cutoff}.

To be able to properly manage the level of detail needed I used a multi-layered grid [1], where I took an initial evenly spaced square-grid over the parameter space with a side-length squares and then got the next layer/level by taking an $\frac{a}{2}$ grid. "Zooming" in on the image (calculating a part on a smaller scale) is basically just locally using a higher detailed layer. See Figure 2 for an image of the grids' layout.

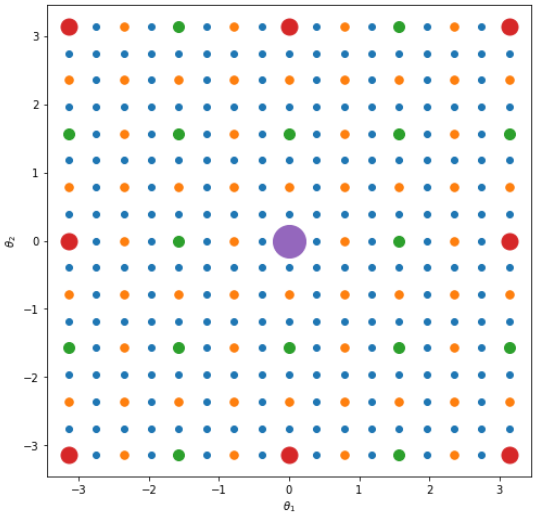

Figure 2: Layout of the grid.
Larger points mean higher levels, smaller points lower levels (or finer detail)

After getting the result as a (partially filled) multi grid I can just flatten it into the bottom layer and fill out from top to bottom the voids or just interpolate the points.

4.3 Searching algorithm

To get the correct shape of the fractal I had to simulate its edge with high precision, so I had to find it. The algorithm searches the parameter space in different "directions" to effectively accomplish this.

4.3.1 Depth

Finding parts of the edge, or at least a vague estimate of its location is possible with a small (i.e. 128x128) grid too. After this sweeping the area around edges with a lower level grid creates a more detailed image of the edge, getting closer and closer to the real contour.

[1] don't search for this, I just made this phrase up

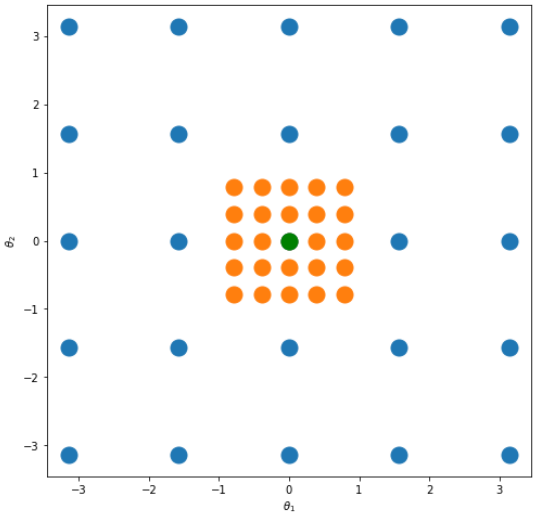

Figure 3: Searching deeper
After finding an edge (green) among the checked points (blue),
the algorithm scans the area at a higher scale (orange)

4.3.2 Breadth

Although going deeper in the grid is important, only checking locally the already found edge pieces with greater precision could lead to gaps in the contour. To get ahead of this happening, the algorithm also checks the neighouring area of the same level whenever an edge is found. As edges could possibly have been invisible in higher level grids.

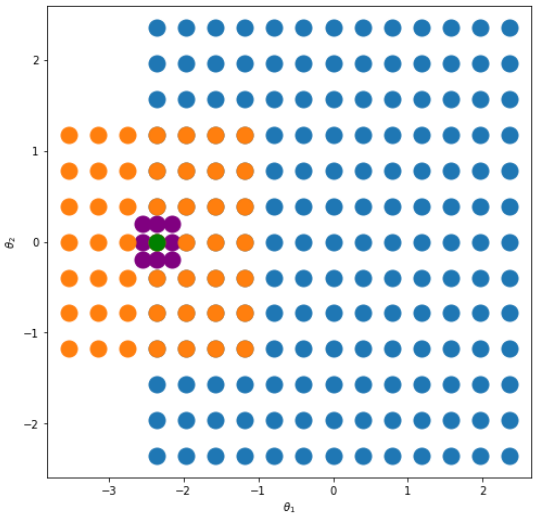

Figure 4: Searching wider
After finding an edge (green) among the checked points (blue),
the algorithm scans the neighbouring area at the same scale (orange),
(also the depth search is done too (purple))

4.3.3 Backwards

Checking the neighbouring area and spreading the search field in width can lead to the case when a lower grid level is locally scanned without its upper levels being searched, yet again

possibly missing edges or at least probably slowing down execution as breadth searching in my implementation is slower. To help this whenever a lower level grid area is searched its upper levels are searched (with decreasing detail) too.

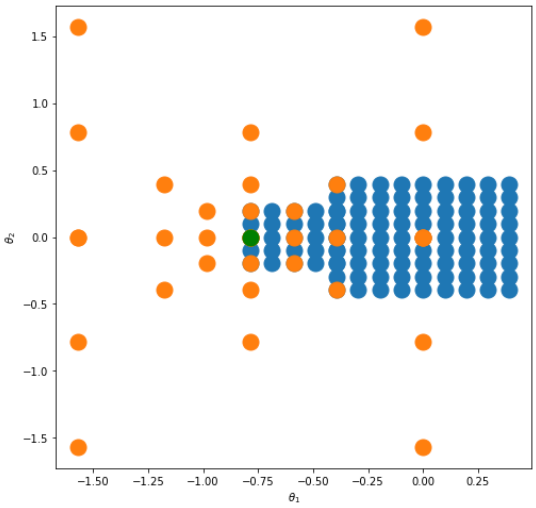

Figure 5: Searching backwards/up
After finding an edge (green) among the checked points (blue),
the algorithm scans the area at every upper scale (orange)

4.4 Implementation

The full implementation can be found at [1].

4.4.1 OpenCL

Different points of the plot require different independent simulations, so the task is highly parallelizable. This is why I am using OpenCL to run the simulation on a GPU, effectively running 10000s of simulations at the same time.

Solving the ODEs is done with a fixed time-stepped RK4 stepper, for it is easier to implement, more GPU friendly, as it has no conditional branching and results in an equal workload, all simulations running for the same time. Although it's precision is questionable,

4.4.2 C++

The C++ code layer has nothing to do but read the starting parameter list to simulate, divide the simulations into properly sized groups, write the parameters to the GPU, run the OpenCL kernel, then read back the results and pipe them to the standard output.

4.4.3 Python

The Python code on top of everything else is handling the edge finding algorithm, sending commands to the C++ code, managing the grids, and processing the results, generating the figures seen here. I chose Python for this task for its easy of use and usability for fast prototyping.

5 Results

5.1 Run

After running the simulation with $\frac{1}{1024}$s timestep for a few hours I ended up with more than 6 million flip-over time calculations, and a maximal grid depth of 13. Apparently the whole process really slowed down and the system was out of memory, slowly filling up my swap partition.

Python was having problems with the millions of location-time pairs, the giant dictionaries and arrays. Apparently it is not an optimal tool for this kind of task.

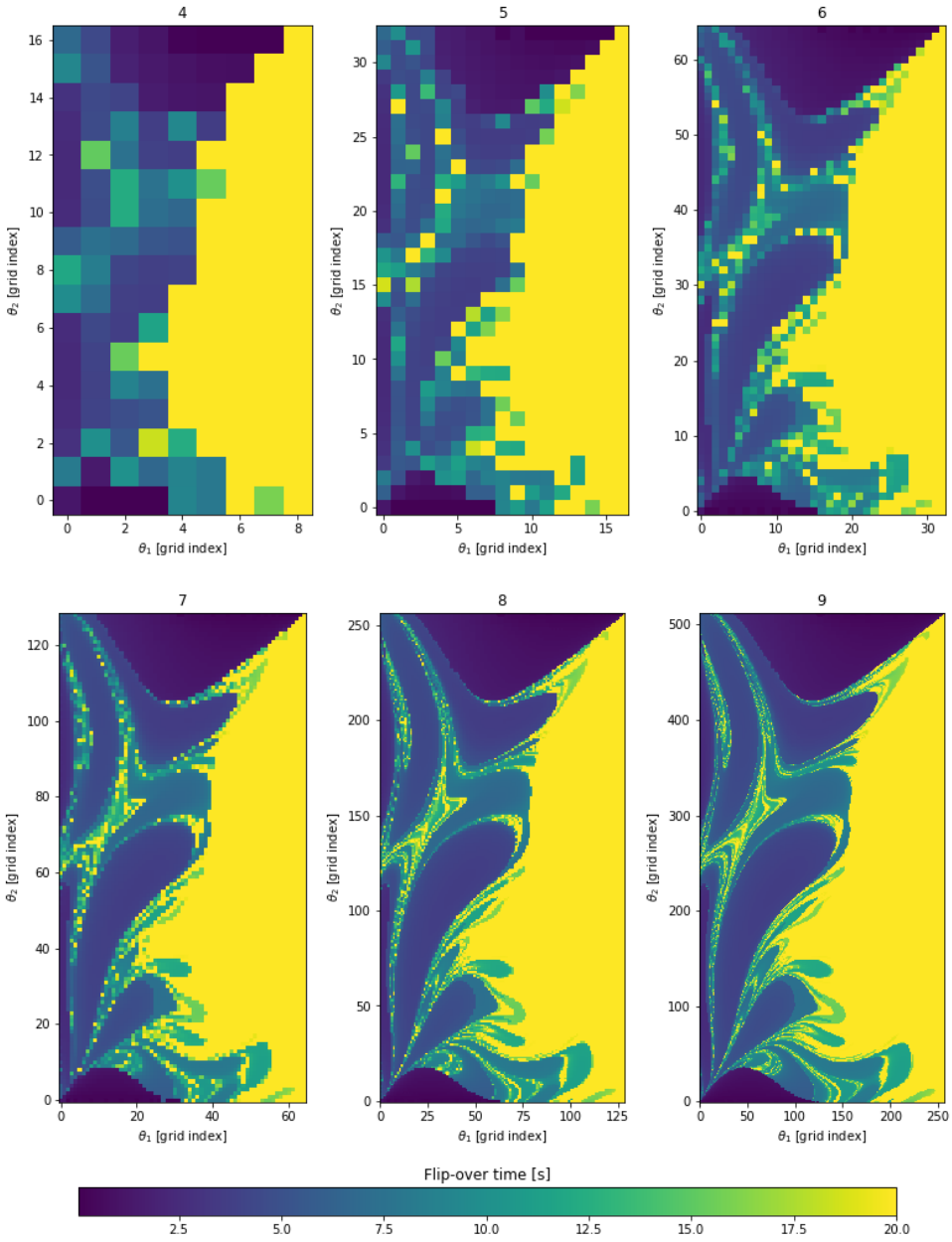

Figure 6: Interpolated flip-over times at certain grid depths

The edge finding and detailing algorithm ran rather well, taking a look at Figure 7 shows that most of the computation done was not wasted, it was always at the area where more detail was needed.

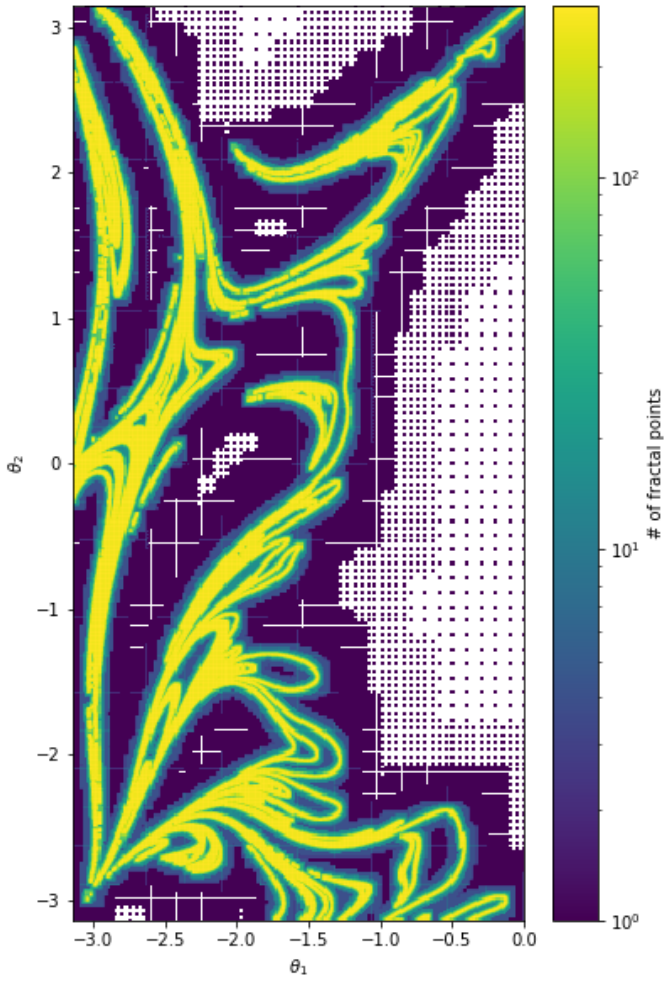

Figure 7: Density of simulations in the parameterspace
The density map beautifully draws the edges as the algorithm simulated along those more. Also the neighbouring grid search is visible, showing different levels with their growing density and ever decreasing θ radius. Zooming in on the edges, always taking a closer look on the neighbourhood looking out for possible short gaps

5.2 Zoom-in

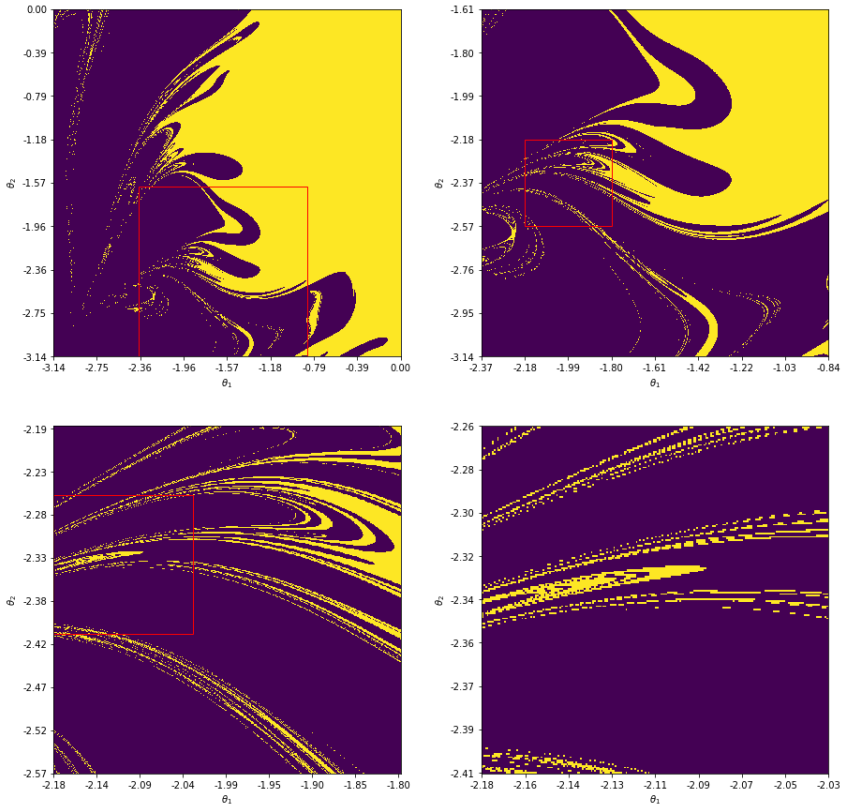

Figure 8: Zoom in on an interesting area
As can be seen in the yellow "tendrils" of the fractal there are more and more smaller ones packed densely. This results in a high frequency change that can lead to the contour getting segregated, until a small scale is used for the calculations

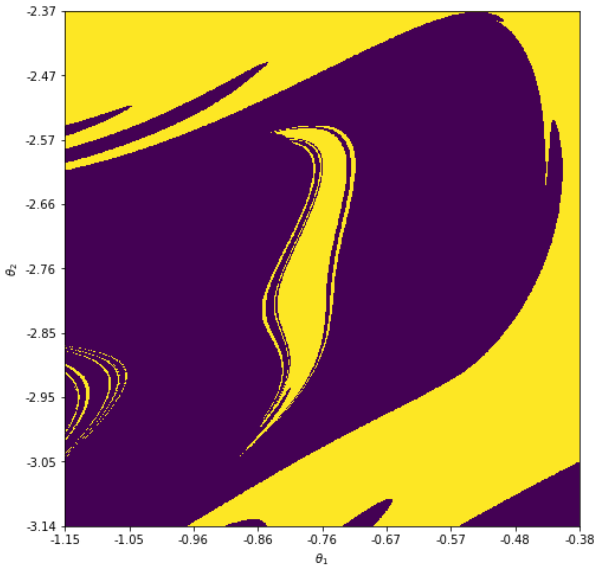

Figure 9: A seemingly island-like part of the $t > t_{\text{cutoff}}$ area. There is no connection visible to the main body of the fractal, if it does exist then it is an extremely thin strand

5.3 Box counting

5.3.1 Area

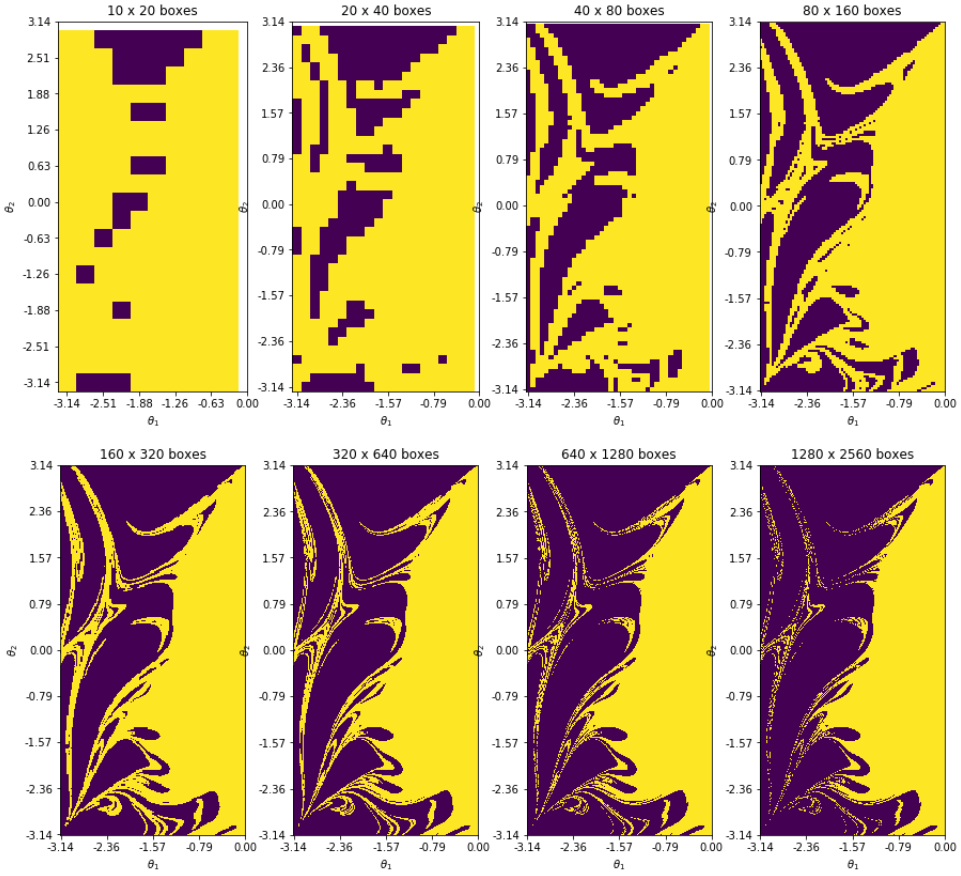

Figure 10: Box counting for the whole fractal
Note that this is not the same as the different level grid views

The results are a bit surprising. I was expecting 2, or at least a number really close to 2 for the dimension as most of its area is in the simple, convex middle area. The result (Figure 10) is close 2, although goes down to ≈1.88, also is scale dependent. Either the fractal truly acts differently on different scales (with the small tendrils being almost 1 dimensional on the right scale) and the giant middle area couldn't suppress this effect, or it is just the effect of the small scales not being properly simulated yet. The simulation always worked on the top 3 upper levels of found edges, so the bottom $\tilde{3}$ of the calculated results were not always "stable". The last scale (and point) is the pixel-by-pixel scale, and of course is not correct as it is not yet stabilized by the lower resolutions, that have not been calculated. Even though this instability exists the upwards going trend started about 4-5 points before the last scale, so it has a good chance of being a true phenomena, not just an error.

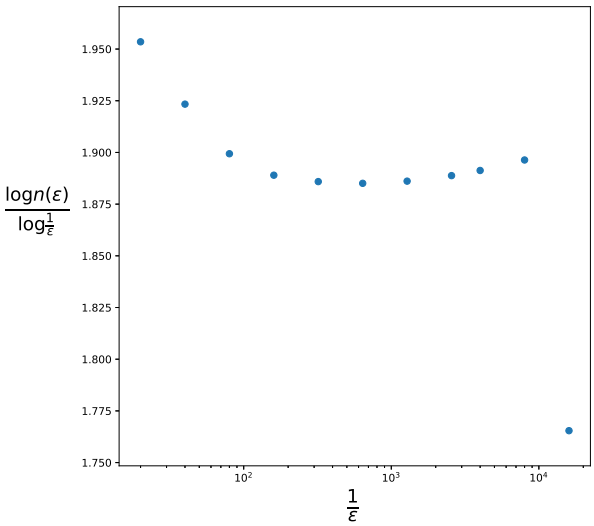

Figure 11: Number of boxes needed to cover the whole (seen) fractal as a function of the number of boxes in the box grid [2]

5.3.2 Edges

I also took the "edges" of the fractal, by edges I mean every simulation point on the finest grid (after flattening every grid into one), where there is a change (is/is not part of the fractal) in the neighbouring cells.

[2] This grid is not the many leveled grid which was used to index the data

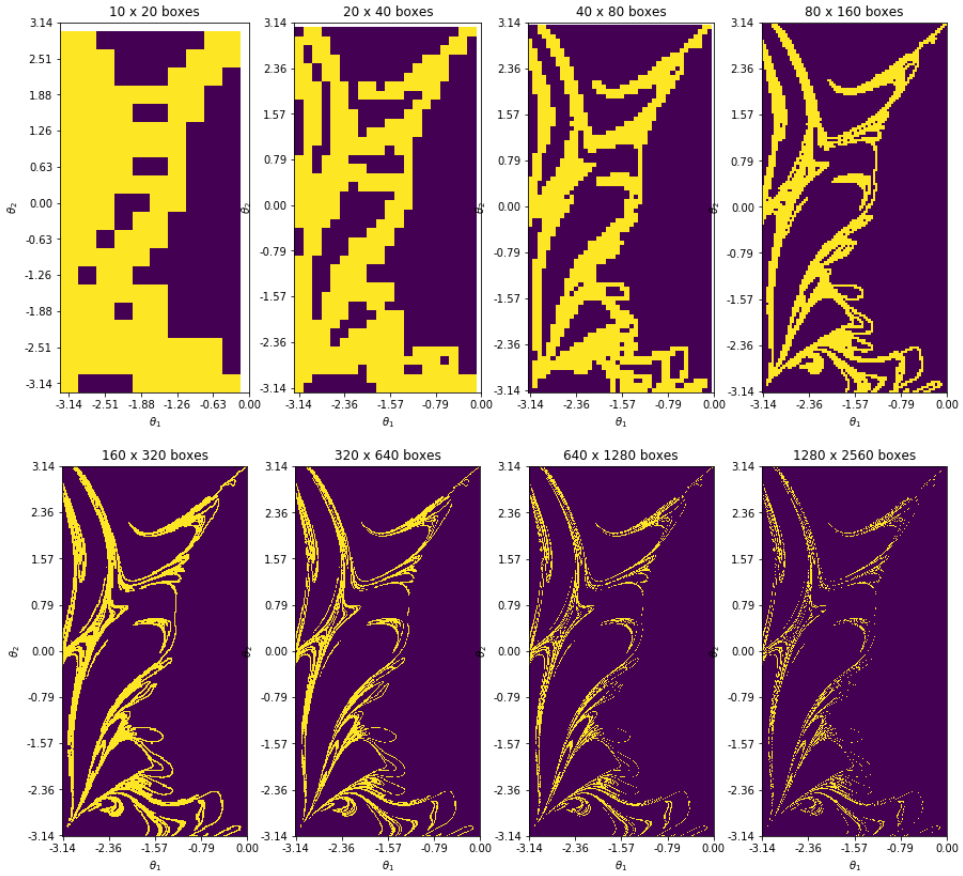

Figure 12: Box counting for the edges of the fractal

The dimension of the edges, estimated to be ≈1.6 seems to be more stable, only the last point is out of line (, and that is all right, as it was expected to be).

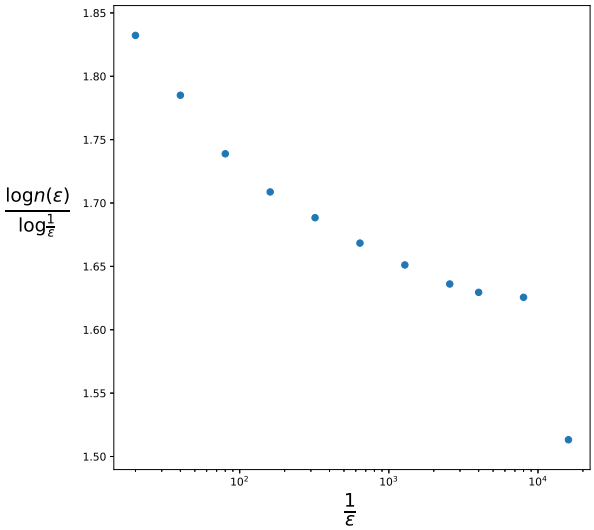

Figure 13: Number of boxes needed to cover the (found) edges of the fractal as a function of the number of boxes in the box grid

5.4 Timestep error

Unfortunately I did not have the time left to properly analyse the effect of the time-step on the shape. I was planning to compare fine details on different scale levels and then compare the difference in the estimated fractal dimension. I did check a few zoom-ins (in a not too large scale around x64) but found no significant difference by eye. But I would not like to include results that I have not had the time to properly check.

6 Discussion

I managed to create an algorithm that can iteratively zoom in on the interesting areas of a fractal, scanning them more and more precisely, just the way it is needed for box counting, while doing as few simulations as possible. I managed to run the algorithm on the fractal I wanted to analyse and got meaningful and interesting results. I could estimate the fractal dimension, and had a somewhat not perfect but still working zoom. Having more computational resources would have helped, but I think this was still manageable. The error analysis could be made better and I would've liked to examine it more thoroughly, though there is only so much time.

All in all I feel like this was an educational and interesting project and I managed to do most of what I wanted to.

References

[1] My Github repo with the code
https://github.com/Balazzs/double-pendulum-dimension

[2] Double pendulum on wikipedia
https://en.wikipedia.org/wiki/Double_pendulum

[3] Rubin H Landau, Manuel J Paez, & Cristian Bordeianu
A Survey of Computational Physics