Lab 3 part 2 - Analysing Ashby's Homeostat

Return to main labs page: **Main Labs Page (https://canvas.sussex.ac.uk/courses/34985/pages/main-labs-page-3)**

The *Sandbox* framework code, and all other code for labs 1, 2 and 3 is here:

> **AS_autumn_2025_exercises.zip (https://canvas.sussex.ac.uk/courses/34985/files/6130167?wrap=1) (https://canvas.sussex.ac.uk/courses/34985/files/6130167/download?download_frd=1)**

Part 1 - evaluating performance

Introduction

In the previous lab class, we experimented with some of the parameters and features of the simulated Homeostat. We did this so that you could gain some intuition about how it works, but in this class we will go beyond intuition and look at some quantitative methods for analysing and evaluating our systems.

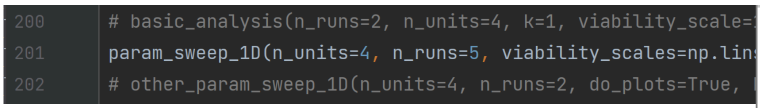
If we are going to compare the effectiveness of different adaptation functions objectively, then we need to define measures of performance, or fitness. We will begin by considering a very simple experiment where the Homeostat is started in a non-viable state, and then we watch to see if and how it finds stability. We will use the script, `lab3_part2.py`.

In the script, I have implemented a simple measure of performance, based on how much time the Homeostat units spend adapting, on average. The smaller this time is, the better, as a short time will mean that not only has the Homeostat successfully found its way to viability, but also that it has done so relatively quickly. It is easy to measure this time, as a Homeostat unit keeps track of when it is viable or searching for viability, in its `testing_hist` variable. If we add together all of the time each unit is adapting, using `sum(unit.testing_hist) * dt` for each unit, and then divide the result by the number of units, `n_units`, then we get the average adapting time for all units.

Because there is randomness involved in how a Homeostat changes its weights, then for a simple and short simulated experiment, there will be some variation in how much time the system spends adapting - e.g. we might have a system which is often very fast to adapt, but occasionally very slow. That being the case, if we want to measure the performance of a Homeostat with random search, then we need to run our simulation multiple times and look at the statistics of the collected results. Here, I have calculated average (or mean) times, by dividing by `n_runs`, but we could also measure more statistics such as the median and standard deviation of performance scores.

Introducing the code

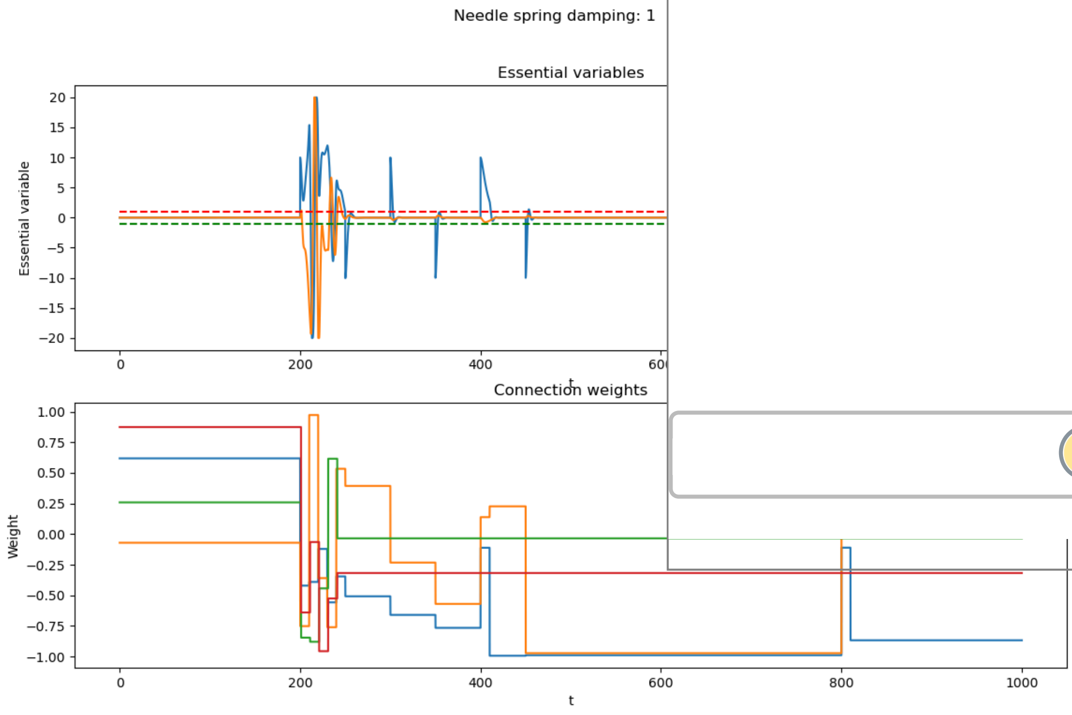
Module chat



```
# basic_analysis(n_runs=2, n_units=4, k=1, viability_scale=
param_sweep_1D(n_units=4, n_runs=5, viability_scales=np.lin
# other_param_sweep_1D(n_units=4, n_runs=2, do_plots=True,
```

Figure 2. We can select which analysis to conduct by co

In this part of the lab, we will not use a params.yaml file, b
program code files. There are 3 different kinds of analysis
which one by commenting/uncommenting the lines shown



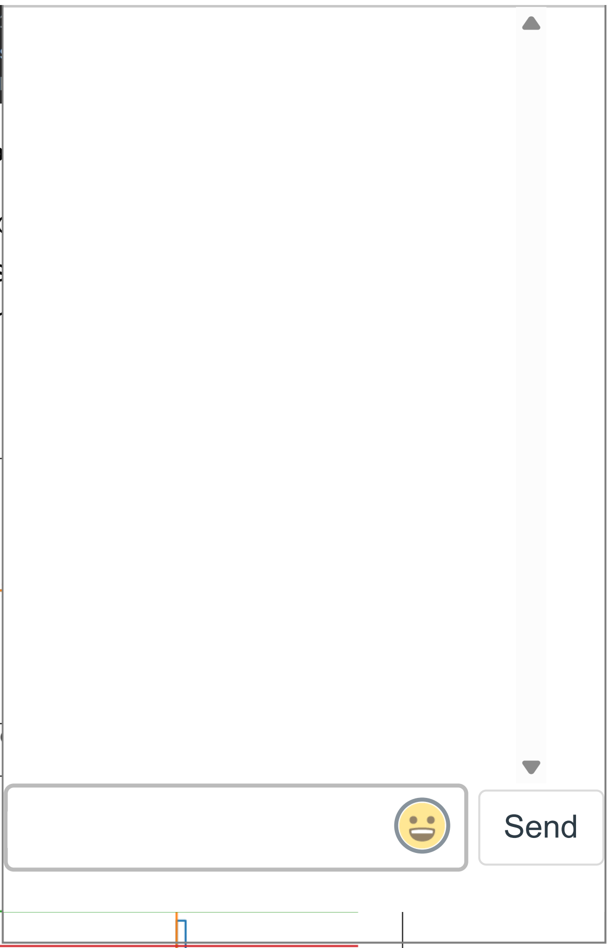


Figure 3. A kind of `DisturbanceSource` is used to experimentally manipulate the system. The `DisturbanceSource` used here is actually an `ImpulseDisturbanceSource`, which can be used to trigger a sudden change to the essential variable (θ) of a Homeostat unit. In this figure, we can see that the disturbance causes sudden spikes in the essential variable for unit 0, and that after every spike, the system starts to adjust its parameters again.

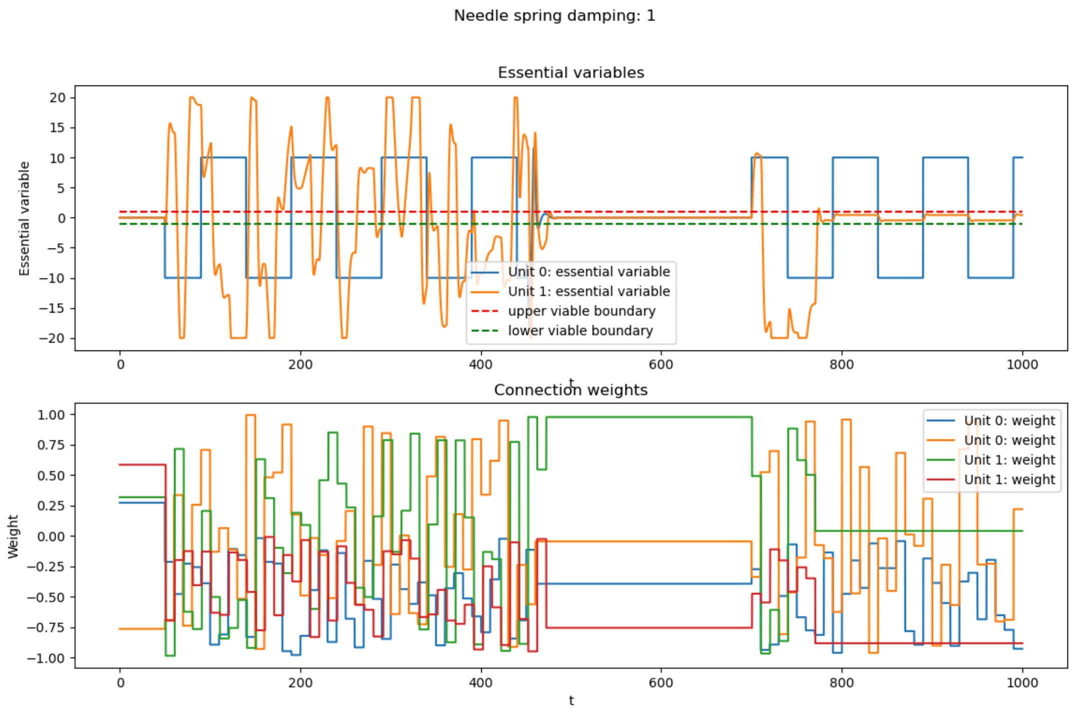


Figure 4. A SquareWaveDisturbanceSource is used to make the essential variable for one unit vary as a square wave. The disturbance is switched off for part of the experiment. Something interesting happens near to the end of the simulation - can you see what it is, and why it is significant?

The various input parameters for the functions used in **Figure 2** are as follows:

- `n_runs` ⟶ The number of times to run the simulation for each experimental condition.
- `n_units` ⟶ The number of units in the simulated Homeostat.
- `k` ⟶ The damping parameter for the Homeostat units.
- `viability_scale` ⟶ Sets the magnitudes of the boundaries of the viable region, e.g. if `viability_scale` = 1, then the upper boundary = 1 and the lower boundary = -1.
- `duration` ⟶ Sets how many simulation time steps to run the simulation for.
- `experiment` ⟶ Allows us to choose between 3 different experiments.
    - Experiment 0 is already familiar to you. In this experiment, we set the initial state of the system to be outside of the viable region, so that the system will immediately start changing it weights, in an effort to adapt.
    - Experiment 1 uses a kind of `DisturbanceSource` (you'll learn more about these soon) to disturb the Homeostat at predetermined times, by directly changing the essential variable of one of the system's units to "push" it outside of the region of viability (**Figure 3**).
    - Experiment 2 also uses a kind of `DisturbanceSource`, which makes the essential variable of one unit vary as a square wave (**Figure 4**).
- `viability_scales` ⟶ When running `param_sweep_1D`, the parameter which is swept over is `viability_scale`. For every `viability_scale` value in `viability_scales`, the simulation will be run for `n_runs` number of runs, and the average time the units spent adapting will be calculated. The lower this average, the better, as this will indicate how quickly the system adapts and finds the region of stability.
- do_plots ⟶ Determines whether or not plots are produced at the end of running the simulation.

- ks $\longrightarrow$ When running `other_param_sweep_1D`, the parameter which is swept over is `k`, the damping parameter for the Homeostat units. For every `k` value in `ks`, the simulation will be run for `n_runs` number of runs, and the average time the units spent adapting will be calculated. The lower this average, the better, as this will indicate how quickly the system adapts and finds the region of stability.

Parameter sweeps

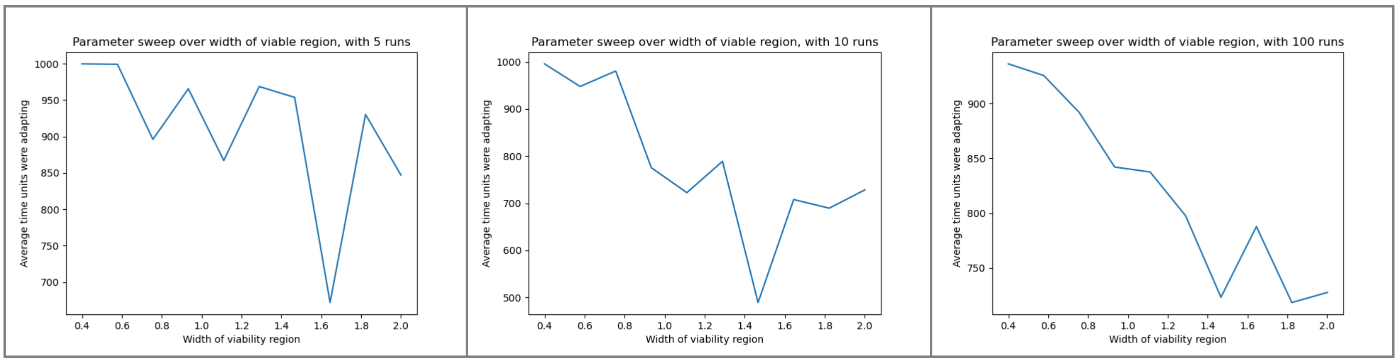


Figure 5. When running parameter sweeps on simulated systems with any degree of randomness/stochasticity, we need to run the simulation multiple times. If the number of runs is too small, e.g. 5 or 10 here, we can't trust the results, as they are too noisy. Only when we use a sufficiently large number, such as 100 in this case, can we begin to trust the trend we see in the data.

You may have covered parameter sweeps in an earlier module, or on your previous degrees, but this is the first time they have appeared on this module. The parameter sweep can be a powerful tool when analysing a system, and I hope to see many of you use it in your second coursework assignment for this module.

In principle, a 1-dimensional parameter sweep is simple: we choose a parameter of our simulated system to vary, generate a set of values for the parameter, and then we run the system with every one of the parameter values to see how they affect its behaviour.

If the simulated system, or its environmental conditions, incorporate any kind of randomness or noise, then it is possible that we will see very different results every time we run the simulation, even without changing any parameters - even a system which normally performs very badly will occasionally perform very well. Therefore, if we want to evaluate such a system's performance we cannot base our analysis on a single result - we must collect multiple results and then look at their *statistics*, e.g. average values, as in **Figure 5**. As we see in the figure, in general the higher the number of runs the better, as the results become less susceptible to randomness - with a low number of runs, the line in the graph is very noisy, and therefore any trend that we see cannot be entirely trusted. With 100 runs, the trend in the results has become more clear, but it would still be nice to use an even larger number of runs, and smooth this graph further.

Tasks

You can do these tasks in any order, and don't have to complete them all, especially if you are not going to use the Homeostat simulation in your coursework projects. However, some of the analyses & techniques used here may be applicable to your projects, so they are worth looking at for all of you.

1. How many runs?

To get a good estimate of the average performance of a Homeostat, we have to run it a sufficient number of times. For either or both of the parameter sweeps in the script, try running the function with `n_runs` = 1, 2, 5, 10, 20, and see if you can identify the point at which `n_runs` becomes large enough to provide a reasonable estimate of average performance. This will be the point at which the graph produced at the end of the parameter sweep always has roughly the same shape - when `n_runs` is too small, the curve on the graph will look different every time. Note that there is usually a trade-off involved here, between how good the estimate is and how long it takes to produce it - this is why I use the slightly vague word "reasonable" - if the time taken to produce results didn't matter, then we could simply make `n_runs` = 1000000, but that would not be practical for this simulation, so we have to compromise.

2. Other parameters

Explore the effects of changing other parameters on the performance of our simulated Homeostat. If you look at the `run_once` function, then you will see that it takes parameters including `viability_scale` and `k`, which are the parameters tested by the `param_sweep_1D` and `other_param_sweep_1D` functions. Try modifying the script to allow you to perform parameter sweeps over other parameters, e.g. `test_interval.` This is one of the parameters you already investigated earlier, and so you will already have some idea of how it affects performance, but now we are able to quantify that effect with our performance measure.

3. 2-D parameter sweeps

A 1-D parameter sweep can be very informative, in showing how the value of a single parameter affects the behaviour of a system, when all other parameters are fixed. However, a 2D parameter sweep can tell us more, as it can show how 2 parameters are related in terms of their effect on a system's behaviour (it would be nice to go to even more than 2 dimensions, but not easy to visualise).

Add a new function which conducts a parameter sweep over two parameters simultaneously.

- This can seem daunting, if you have never done it before, but the main difference between a 1-D parameter sweep and a 2-D one is just an extra (nested) for loop, which iterates over a second list of values.
- For storing and displaying the results of a 2-D parameter sweep, I normally use a numpy array and the matplotlib `imshow` function.

4. Other measures of performance

Implement other performance metrics. Sometimes we will need to use different ways of measuring performance for different experiments, and sometimes we will want to use multiple measures for a single experiment, e.g. to see how consistent they are with each other. This is not a comprehensive list, but here are a few suggestions for other ways of measuring performance:

- In experiment 0, where the system starts in a non-viable state and is not subsequently disturbed, measure the time taken to recover viability for the *first time*.
  - In this simple experiment, this measure will often, but not always produce the same number as the metric which I have implemented. It will not always be the same, as sometimes the system will return to the region of viability and then leave it again before it has time to stabilise.
  - To calculate this, use `testing_hist` as before, and find the first point at which it changes from `True` to `False.`
- Measure the average distance from viability over time.
  - This takes a little more thought, and can only be calculated using the Homeostat `thetas` list (i.e. its essential variable values over time) as well as its `upper_viabilty` and `lower_viability` parameters.
- Produce a probability distribution, or a histogram, of states for each unit in the Homeostat.
  - A high-performing Homeostat will have a narrow distribution with zero mean.

5. Other disturbances

A Homeostat can be disturbed in various ways. So far, we have seen only simple disturbances to the state of a single unit, and you have probably seen that the system adapts to these disturbances fairly quickly, as long as it only has a relatively small number of units. More generally, a Homeostat can be disturbed by changes to the states and parameters of any/all of its units. Can you think of (and code, with our help if necessary) other types of disturbances which are non-trivial and yet which the Homeostat still stands a chance of adapting to?

Part 2 - analysing dynamics

Introduction

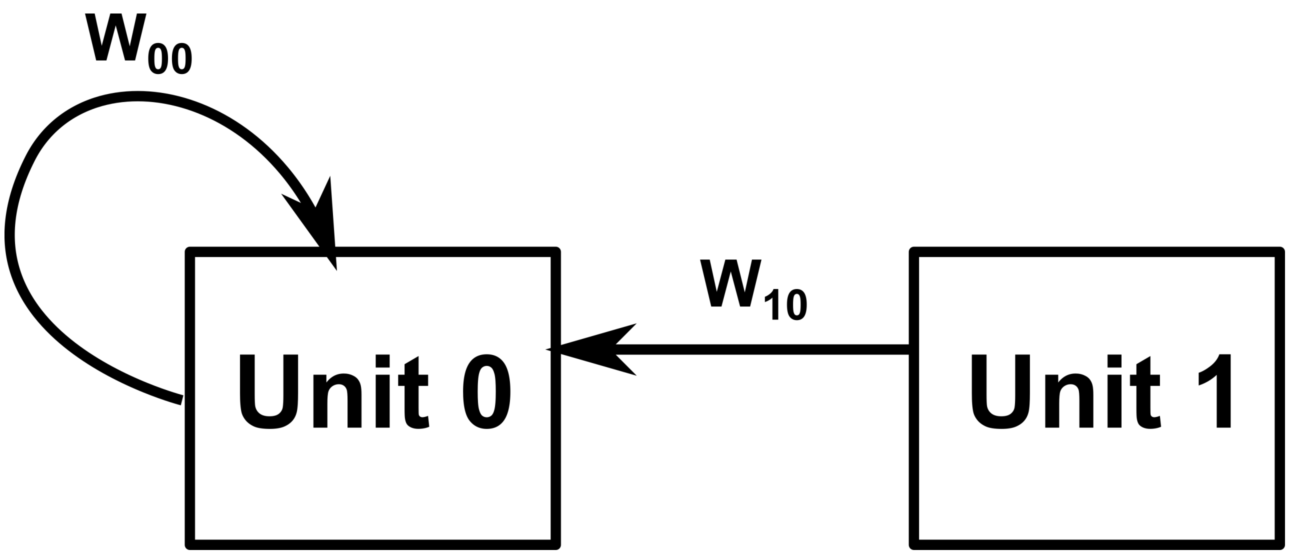


Figure 6. The system which we will analyse in this part of the lab. Unit 1 is only used to provide a constant input to unit 0. Unit 0 can change the weights on the feedback connection and the input connection, w_{00} and w_{10} respectively.

In lectures, we have seen examples of phase portraits, e.g. for pendulums and for two neurons in a CTRNN. In this part of the lab, we will use a phase portrait to analyse the behaviour of a single simulated Homeostat unit with one constant input (**Figure 6**). Analysing the behaviour of a single unit in this way won't explain the full dynamics of a network of units, but it will get us part-way there, and will make the connection between Ashby's Homeostat machine and his writing on the subject of ultrastable systems more clear.

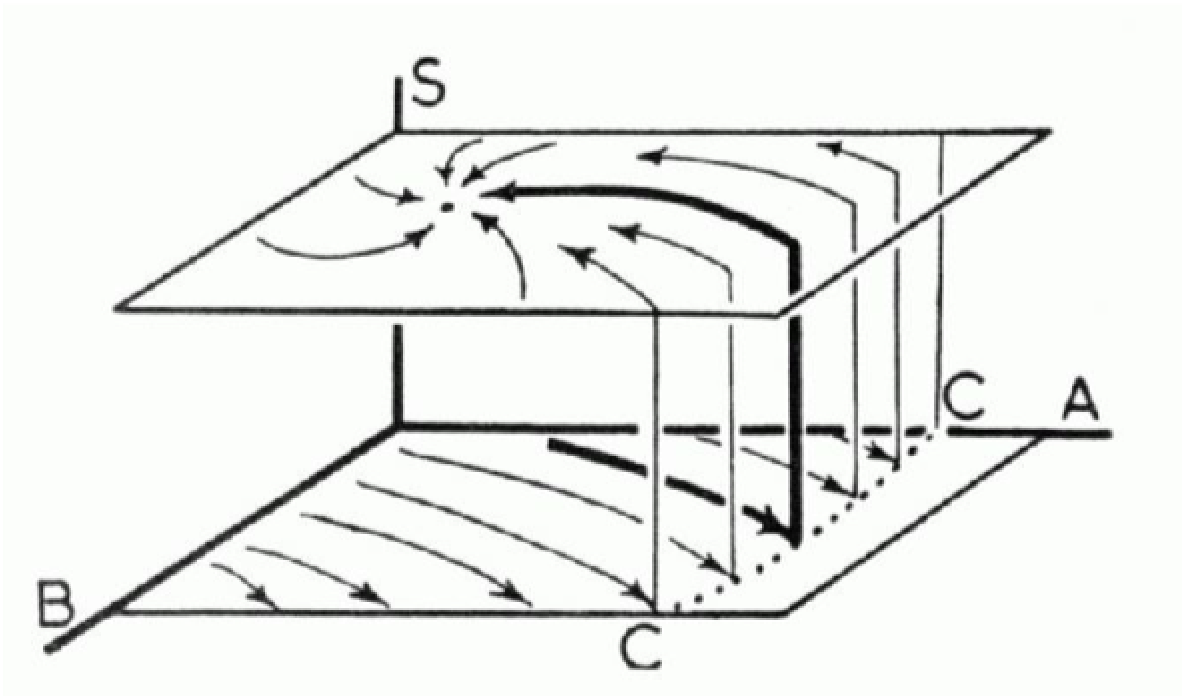


Figure 7. An example of an ultrastable switching from an unstable to a stable field in phase space.

Ashby described an ultrastable system as one which can select between different fields (collections of lines of behaviour) in phase space in order to find one which is stable under the current environmental conditions. Each unit in the Homeostat is ultrastable, and selects different fields (**Figure 7**) by changing the weights on its input connections, one of which is a feedback connection from its own output. With θ_1 being the constant value that Unit 1 outputs, the system equation for Unit 0 is:

$$\ddot{\theta}_0 = -\dot{\theta}_0 + (w_{00}\theta_0 + w_{10}\theta_1)$$

Note: this equation looks simpler than Ashby’s system equation because I have made *m = k = l = (p - q) = 1.*

In this part of the lab, you will investigate how, and consider why, changing the weights w_{00} and w_{10} selects different fields in the minimally ultrastable system of **Figure 6**.

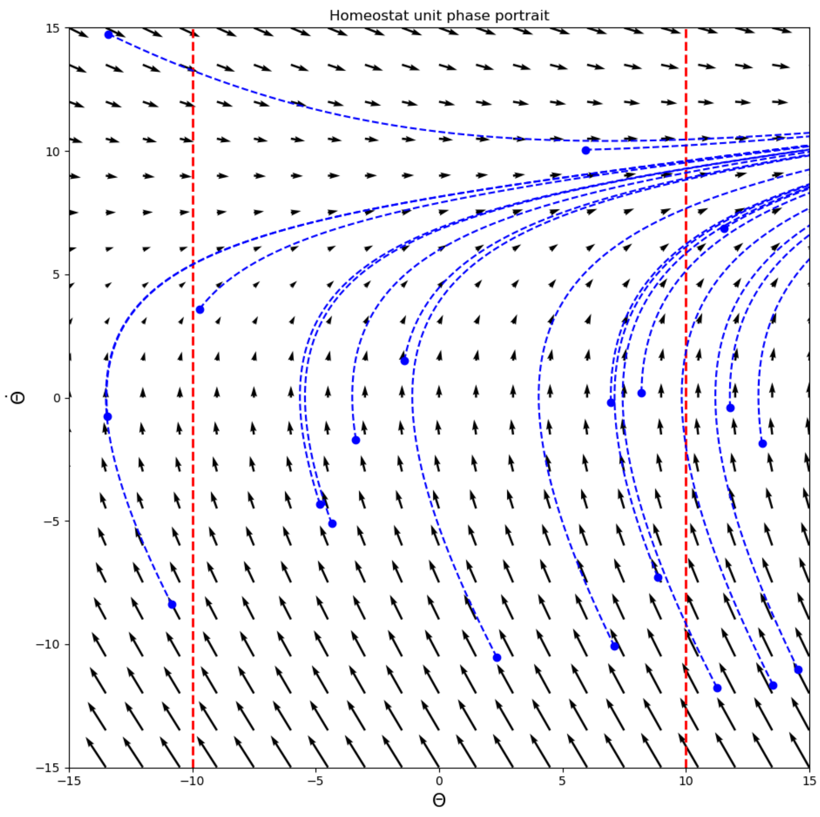
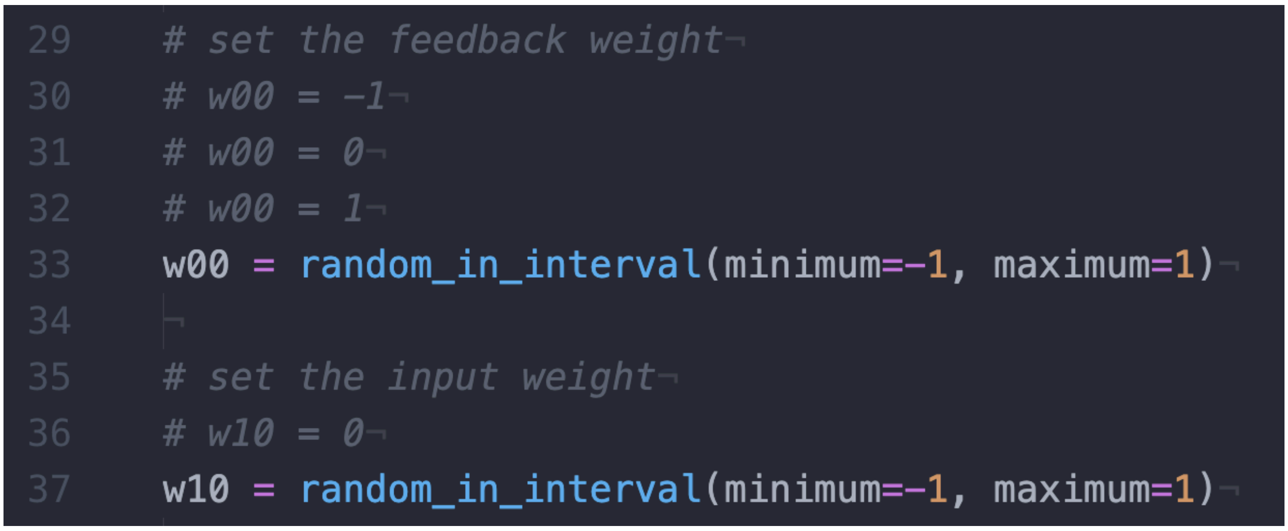


Figure 8. An example phase portrait output from `lab3part3.py`



```
# set the feedback weight
# w00 = -1
# w00 = 0
# w00 = 1
w00 = random_in_interval(minimum=-1, maximum=1)

# set the input weight
# w10 = 0
w10 = random_in_interval(minimum=-1, maximum=1)
```

Figure 9. The parameters to change to select different fields.

You will use the `lab3part3.py` script in this part of the lab. In essence, all you have to do is edit the lines of code shown in **Figure 9** (simply by commenting/uncommenting) and study the resultant fields in phase space, an example of which is show in **Figure 8**.

Tasks

For all of the conditions below, you need to look at the phase portraits and the weights that led to them *together*, to try to understand how the weights lead to the fields. When you are using randomised weights, you can read their values from the run window (or Terminal, Powershell, etc. depending on where you run the code).

- For each of these sets of conditions, run the simulation at least 5 times to see if you can recognise the pattern in your results:
  - 1. set w_{00} to 1; w_{10} to 0
  - 2. set w_{00} to 0; w_{10} to 0
  - 3. set w_{00} to -1; w_{10} to 0
- For each of these sets of conditions, run the simulation at least 10 times to see if you can recognise the pattern in your results:
  - 4. set w_{00} to 1; w_{10} to random value
  - 5. set w_{00} to 0; w_{10} to random value
  - 6. set w_{00} to -1; w_{10} to random value
- For this last set of conditions, run the simulation at least 20 times to see if you can recognise the pattern in your results:
  - 7. set w_{00} to random value; w_{10} to random value

The padlet for this lab is here:

Trouble viewing this page? Go to our diagnostics page to see what's wrong.
Skip to content

CHRIS JOHNSON 2YR

Lab 3 padlet

if you have any questions about the lab, or how to run the code, please ask them here

0This post has 0 upvotes
0This post has 0 downvotes
0This post has 0 comments
+
Add comment

Do you have any nice plots that you would like to share? For example, ones that show interesting behaviours, or just make nice patterns

0This post has 0 upvotes
0This post has 0 downvotes
0This post has 0 comments
+
Add comment

Do you have any robot behaviours which you don't know how to explain?

0This post has 0 upvotes
0This post has 0 downvotes
0This post has 0 comments
+
Add comment

+

Made with Padlet