A Study of Kuramoto Oscillators

Aaron Schade

January 25, 2021

Contents

1 Theoretical study

2 Numerical study

Omegas normally distributed

2.1 $r_\infty(K)$ – simulated vs predicted

2.1.1 Euler method

Design and initiation
My implementation of the Euler method is object-oriented. This may not be the computationally most efficient way, but I'm still somewhat coding beginner, so I appreciate the clarity that object-oriented programming affords. The first type of object is simply an `Oscillator` with three fields: ω, θ_{s-1} and θ_s. In the Euler method it is enough to store the present and the last value of the state variable (θ). In the rest of this documentation, I will refer to *simulation* time steps with the variable s to differentiate it from the *equation* time step t.

An `Oscillator` gets initiated with its natural frequency and the current-period θ_s. The last-period θ_{s-1} is initiated as zero, since the first part of making an 'Euler step' forwards in time is handing over θ_s to θ_{s-1}.

The other object type is a population of oscillators, `OscPopulation`. It consists of a list of `Oscillators` and a construction mode, namely the probability distribution of the ω. When an `OscPopulation` is initiated, for each `Oscillator` a natural frequency and initial phase are drawn from the respective types of random distribution. An `Oscillator` object is then initiated with those values for ω and θ_s and assigned to a place in the list within the `OscPopulation` object. The `OscPopulation` object thus possess an indexable list of all oscillators.

Running the simulation
The simulation has two functions, to run in two modes, `OscPopulation.runK()` and `OscPopulation.runT()`. I will get into the latter in subsection 2.2.

To start a simulation for finding $r_\infty(K)$, a new `OscPopulation` object gets created, passing the desired distribution type of ω to it. Then `runK()` is run on this object. It loops through the different values for K, starting with resetting the oscillators in the population, then Euler-stepping all oscillators through time until T and finally calculating $r_\infty(K)$. Take each in turn.

The oscillators are reset by by finding new random values for ω and θ_s for all `Oscillators` and setting their field value equal to the newly calculated value. Same as during the original initiation, θ_{s-1} is set to zero.

The Euler-step function `OscPopulation._oneStepForAll()` is straight-forward in principle, but needs to be optimised for reasonable computation times. Originally, running task 1 with $N = 100$, $K = [0, 4]$ and $dK = 0.1$ took about 45 minutes. Since the computation time likely increases exponentially with N (more neighbouring oscillators have to be considered at *each* step) the full compute time for $N = 1000$ would have been unreasonable. Thus, I make use of the `numba` package which offers 'decorators' for functions. Decorators are functions which take functions as inputs and return modified functions. In this case, the `@jit` decorator converts my Euler-step computation function into optimised machine code using the LLVM compiler. This can yield computation speeds similar to C or FORTRAN.[1] It reduced the compute time for task 1 (with $N = 100$) from 45 minutes down to 4 minutes. To make it work, however, the function needs to take `nympy.ndarrays` as inputs, not objects, like originally the case in my object-oriented code. Therefore, I wrote the wrapper function `oneStepForAll` around `_oneStepForAll` (which actually does the computation). It retrieves the values from the `OscPopulation` object (a getter), puts them into temporary variables and then calls the optimised function `_oneStepForAll` using those temporary variables (namely, arrays for

[1] http://numba.pydata.org/

$\omega, \theta_{s-1}, \theta_s$). Finally, a setter writes the returned arrays for θ_{s-1}, θ_s onto the `Oscillator` objects in the `OscPopulation` .

The core `_oneStepForAll` function implements a standard Euler approximation. First, the θ are stepped through time by handing the value of θ_s over to θ_{s-1}. In the next step, the sum of the the *sin* of the differences between the last-period θ of oscillator n (θ^n_{s-1}) and all other oscillators j (θ^j_{s-1}) is computed:

$$sum = \sum_{j=0}^{N} = \sin(\theta^j_{s-1} - \theta^n_{s-1})$$

From this, we can calculate the discrete $\Delta\theta^n_s$

$$\Delta\theta^n_s = \omega_n + \frac{K}{N} * sum$$

The new θ_s of oscillator n is then

$$\theta^n_s = \theta^n_{s-1} + \delta t \cdot \Delta\theta^n_s$$

This is done for all N oscillators.

After all oscillators have been stepped one simulation step δs forward, this is repeated $\frac{T}{\delta t}$ times to get to the final time point T. At this point, the limiting coherence parameter $r_(T)$ is calculated for the current K and appended to a list of all $r_\infty(K)$.

A new loop with the next K begins until all K values have been exhausted. The full results are returned as a list which is then graphed. A scatter plot of $r_\infty(K)$, as opposed to a continuous line, has been chosen since a straight connection between points cannot be assumed.

2.1.2 Program profile

To make the structure of the code more clear and see which parts of the Euler-based simulation take the longest to compute, I profiled my python code. This profile shows which functions called which others as well as information about the execution of the respective function. After the name of the function, the first line shows the total time spent in this function (in itself and its children), the second how much time was spent only in itself and the third how many times the function was called.

The profile of the code for tasks 1 and 2 can be seen in figure 1. Even when optimised, the computation time for these tasks is still 1h 30min. As the profile shows, almost all of this time (87%) is spent in the main computation method `_oneStepForAll`, which was called almost half a million times. The wrapper method retrieved 1.3bn and set over 800m values.

2.1.3 Numerical integration

The strategy to integrate the consistency equation is to loop through different values of r and check which ones make the right-hand side approximately equal to one. The precision of this approximation can be adjusted by changing the `rHysteresis` variable, which sets the acceptable range of values around 1 for the right-hand side of the consistency equation. This is the `calc_r_integ(K)` function which takes K as an input.

We now cycle through the different values for K in a separate loop to get the values of the desired $r(K)$ relationship. Since this is not a computationally-intensive step, I increased the resolution of K to $\delta K = 0.01$. The hysteresis around the left-hand side value is 0.01 and the step size of candidate r is $\delta r = 0.001$. The results are then graphed in figure 2 together with the simulated points.

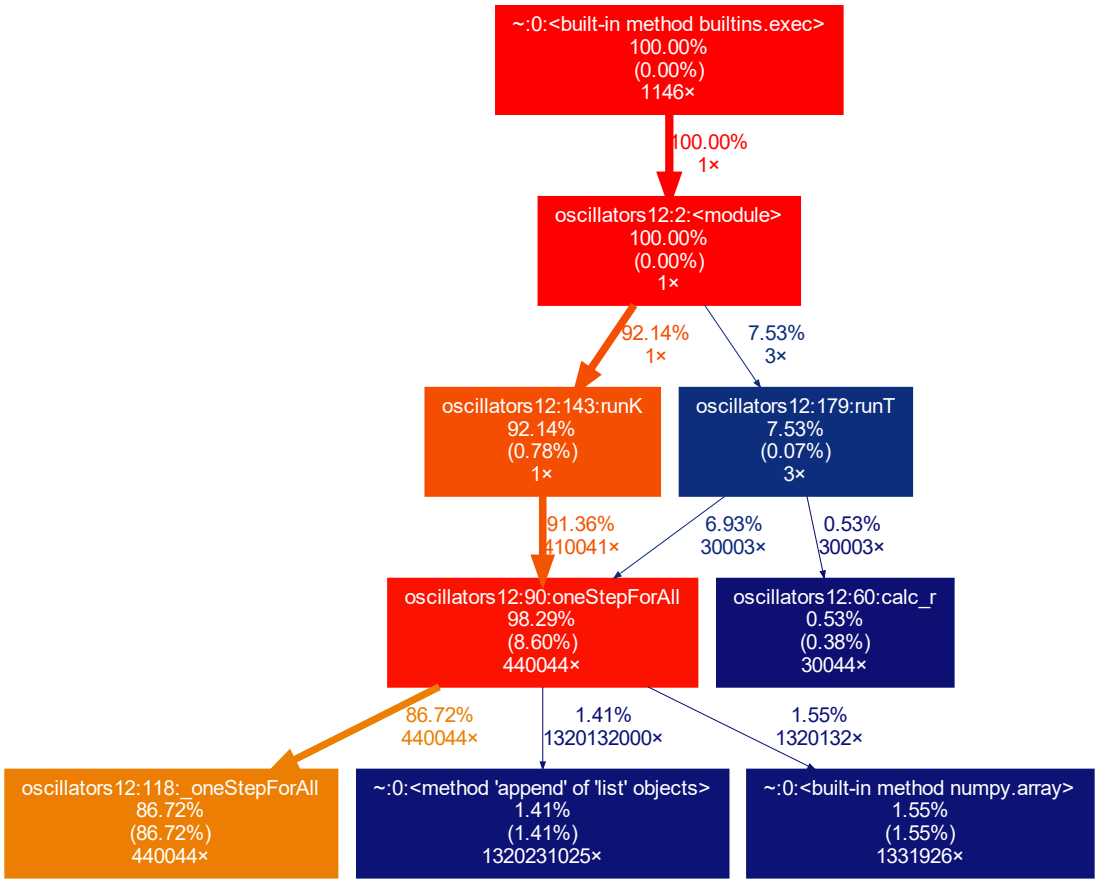


Figure 1: Profile of code for task 1 and 2

2.1.4 Results

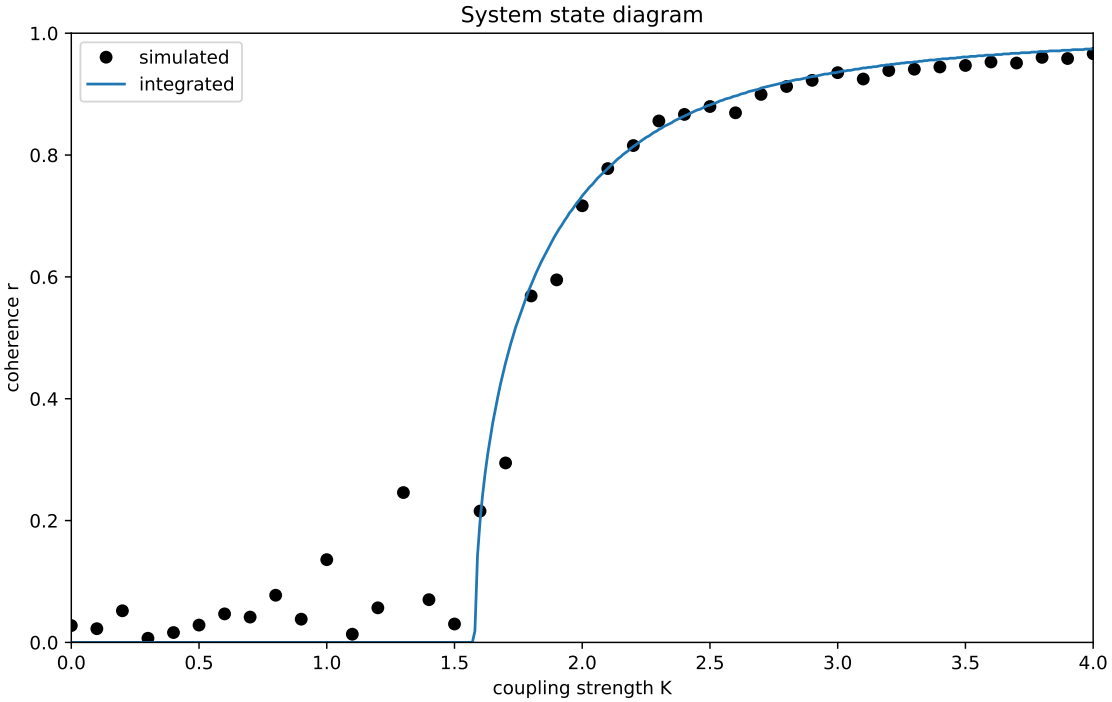


Figure 2: Diagram of limiting r_∞ for different values of K

The first observation is that, on the whole, the simulation values track the theoretical values quite closely. More precisely, the fit is *very good* for $2 < K$, *good* in $0 < K < 1$ but has large deviations in $1 < K < 2$.

My first impulse for improving the fit of the simulation lies in shrinking δt to reduce the local and global error (which are proportional to $(\delta t)^2$ and δt, respectively). Such errors happen because ODEs *on paper* are continuous and essentially have infinite precision, which is impossible to achieve *in silico*. In some non-linear dynamical models, such deviations in the discretised simulation can lead to *qualitatively* different outcomes, especially when the system exhibits sensitive depends on initial conditions (ICs). I do not believe the global error is large enough to cause the large positive deviations we observe just before the critical value $K_{crit} = 1.5958$.

Instead, I hypothesise that the high-coherence observations in the range $1 < K < 1.5$ (which theory predicts should be very close to 0), are due to special sets of ICs. Out of all possible ICs of individual oscillators, a small subset will favour convergence of all oscillators to a (partially) synchronised state. If enough oscillators have correlated ICs, this could lead to higher-than expected coherence in the system. If it is true that the outliers in the range $1 < K < 1.5$ are caused by unlikely constellations of ICs, then increasing N should diminish the occurrence of unlikely starting conditions. The probability of any IC to be coherence-favouring does not increase with N; in fact it may decrease because the union of coherence-favouring ICs between all oscillators shrinks. As N gets larger, it thus becomes progressively less likely that a sufficiently large portion of the oscillator population (a 'critical mass') is randomly drawn with collectively-coherence-favouring ICs.

2.2 $r(t)$ for $K = [1, 2, 3]$

The simulation procedure for is the same as for `runK()` , with 2 exceptions:

1. there is still a K-loop (with limited values $K = [1, 2, 3]$), but there is no resetting of the population to new random values at the beginning of each such loop.
2. $r(t)$ is now calculated at every (simulation!) time step t.

The previous method of assigning new random values to the state variables of all oscillators did not work here for some reason. Instead, a bug occurred in which the final values of the last run were used as the starting values of a new run.[2] Therefore I had to go another way: the `OscPopulation` object is created anew in every loop. After the object is initiated with the right distribution of ω, `runT()` is executed. It loops through all simulation time steps s. In one such loop, all oscillators are updated, then `calc_r()` is called to compute r_s.

`calc_r()` calculates r_s in one of two different but equivalent ways. In one way, actual complex numbers are added up and divided by N. To get a value for r, the modulus of the resulting complex number is taken.

$$r_s = \frac{1}{N} \cdot \left| \sum_{n=0}^{N} e^{i\theta_s^n} \right|$$

The other way is to compute the double sum of real and imaginary component of the complex number (using the *sin* and *cos* relationships after DeMoivre and Euler). These two sums are also divided by N, yielding the average real and imaginary component of the `OscPopulation` at simulation time s.

$$realsum = \sum_{n=0}^{N} \frac{\cos(\theta_s^n)}{N}$$

$$imsum = \sum_{n=0}^{N} \frac{\sin(\theta_s^n)}{N}$$

From there, r can be calculated through a square root:

$$r_s = \sqrt{realsum^2 \cdot imsum^2}$$

I tried both versions to double check the results and to see whether one would give a performance boost over the other, but both have approximately the same speed. In the end, `calc_r()` returns a list of $r(t)$ which will be appended to a list-of-lists with the r values for all chosen K. These different trajectories are once again graphed: figure 3.

[2] .. despite many attempts to fix this bug. My hypothesis is that this is about memory management in python. Sometimes during other assignment of variables or objects, it merely creates a pointer instead of copying the value into a separate location in memory.

2.2.1 Results

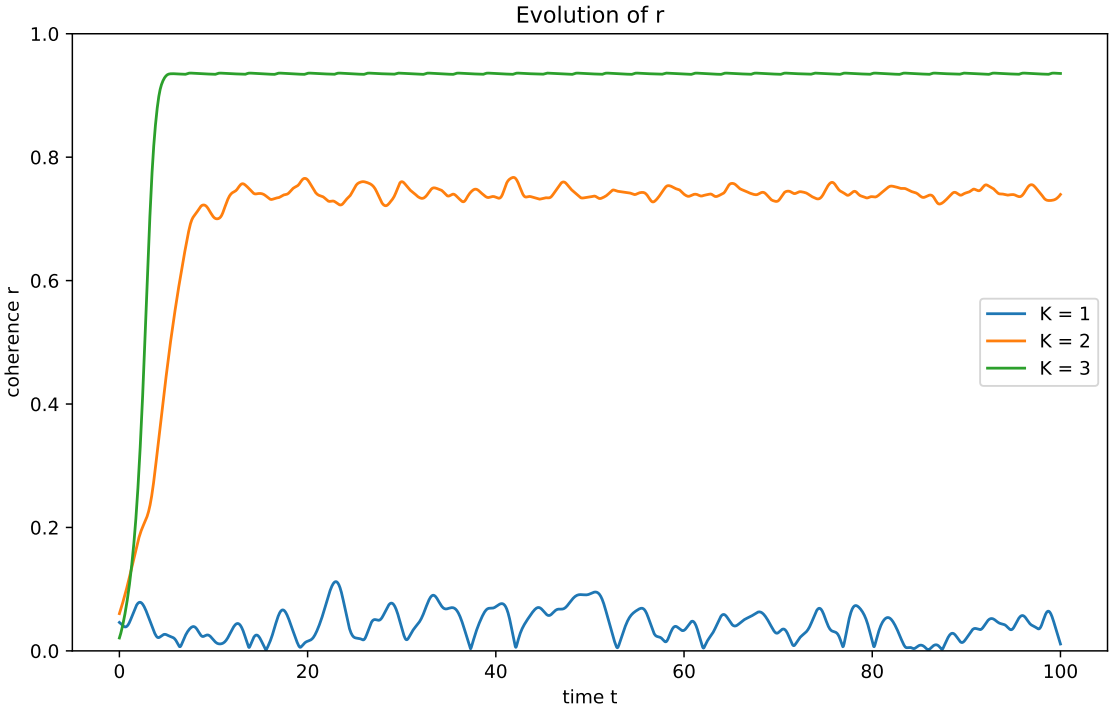


Figure 3: $r(t)$ over time for 3 different values of K

Of note is firstly that populations with $K < K_{crit} \approx 1.5985$ do not converge to coherence, while those with $K > K_{crit}$ do. In this first sense, it is in agreement with theory. The second noticeable feature is that when convergence happens, it happens quickly, before $t = 10$. Thirdly, the consistency equation predicts essentially three ranges of K with distinguishable features, as we can see in figure 2. In the first region $0 < K < 1.5$ r_∞ stays glued to zero. Afterwards, it predicts a sudden, almost discontinuous jump between $K = 1.5$ and $K = 2$. From $K = 2$ onwards, the functional form is logarithmic (or, more approximately, linear). The aforementioned jump does not carry r all the way to $r = 1$ but only to $r \approx 0.7$ – there is some way for it to still increase. In this region, r continues to increase but at a slowing pace. In this sense, the $r(t)$ graph confirms the theory as well, as r stabilises at different values depending on K. And it does not fluctuate around these values values a lot. We can therefore say that with some confidence that the values in the $r(K)$ diagram are indeed indicative of the *limiting* $r_\infty(K)$. Had there benn a lot of variation around the 'idealised' r_∞ the values we see in the $r(K)$ graph could have consisted merely of random points from within that range of variation. Instead we see that once converged, the coherence of a system does not vary greatly and the points in $r(K)$ are accurate representations of the system. This is good for the regions in which the simulated r_∞ track the theorical closely, and bad for regions where they do not. However, the argument about unlikely coincidences of ICs being the culprit resolves this problem.

Omegas uniformly distributed

2.3 $r_\infty(K)$

2.3.1 Program profile

The profile of the code for tasks 3, 4 and 5 can be seen in figure 4.

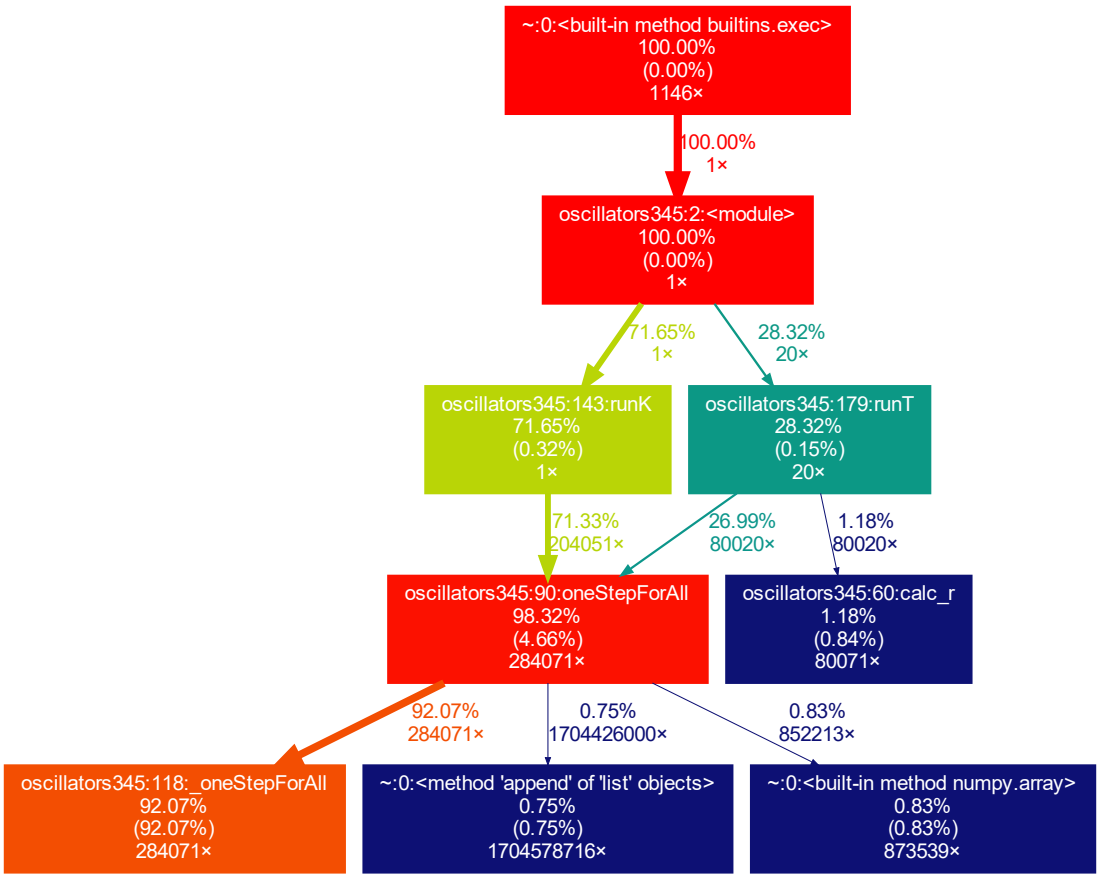
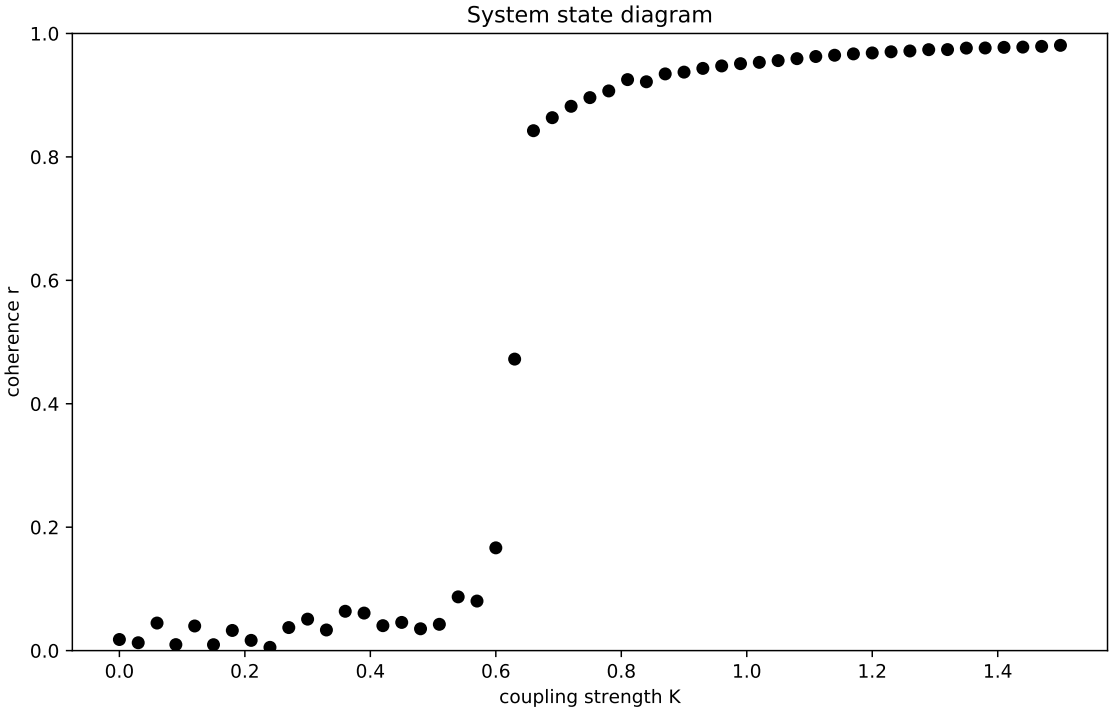
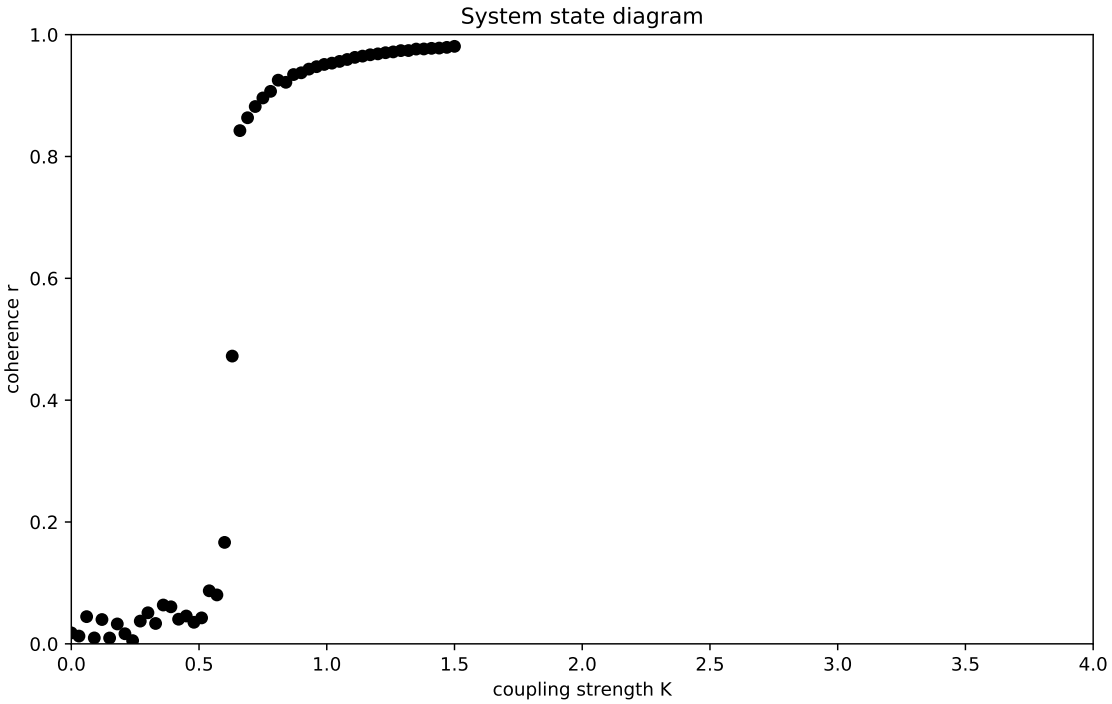


Figure 4: Profile of code for task 3, 4 and 5

2.3.2 Results



(a) zoomed in



(b) zoomed out, for better comparison with figure 2

Figure 5: $r_\infty(K)$ for different values of K, with a uniform distribution of ωs

2.4 $r(t)$ for different initial conditions θ_0

To run simulations repeatedly while keeping selected initial conditions the same, I implemented the following. First, a 'reference' `OscPopulation` object is initiated. Then a loop begins in which a 'temporary' `OscPopulation` objects (newly initialised with entirely random values) is created. The desired initial conditions from the reference object are then written into the temporary object: here, the ωs are kept, later the θs. `runT()` is executed on the temporary object and the returned trajectory saved into another list-of-lists.

2.4.1 Results

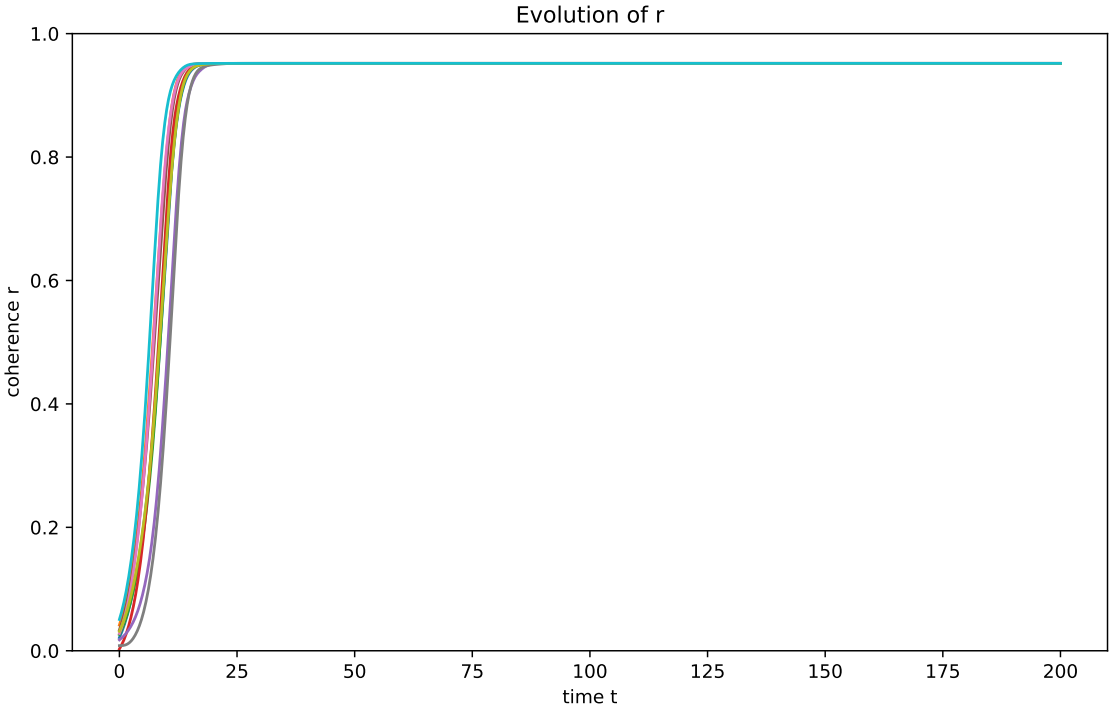


Figure 6: $r(t)$ over time for 10 different initial conditions of θ

2.5 $r(t)$ for different initial conditions ω_0

2.5.1 Results

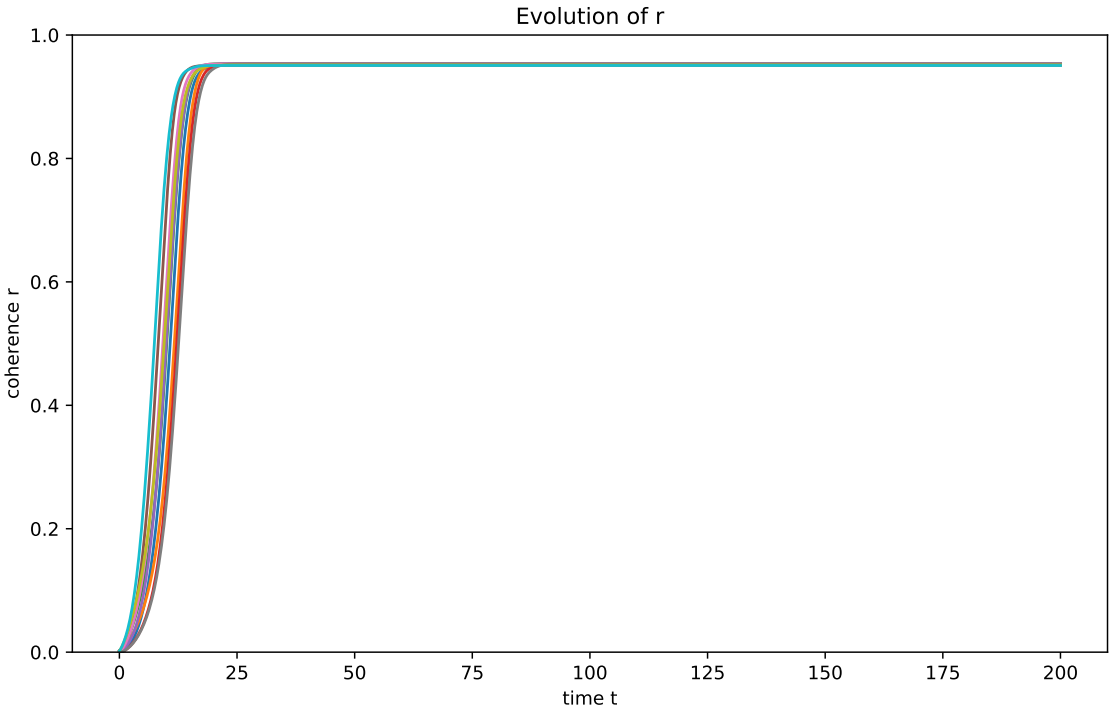


Figure 7: $r(t)$ over time for 10 different initial conditions of ω