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 of a beginner with code, so I appreciate the clarity that object-oriented programming affords.

The first type of object is simply an `Oscillator` with three variables: ω, θ_{t-1} and θ_t. A $\Delta t = 1$ is enough in the Euler method. An `Oscillator` gets initiated with its natural frequency and the current-period θ. The last-period θ is initiated as zero, since the first part of making an 'Euler step' forwards is to hand over θ_t to θ_{t-1}.

The other object type is the `OscPopulation` . It consists of a list of `Oscillators` and a construction mode, namely the distribution according to which the ωs are distributed. When an `OscPopulation` is initiated, for each `Oscillator` a natural frequency and initial phase are drawn from the respective type of random distribution. An `Oscillator` object is then initiated with those values and assigned to a place in the list within the `OscPopulation` object.

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

To start a simulation, a new `OscPopulation` gets created, passing the desired distribution type of ωs to it. Then `OscPopulation.runK()`is run on this object. `OscPopulation.runK()`loops through the different values for K, starting with resetting the oscillators in the population, then Euler-stepping all oscillators through until T, and finally calculating r_∞. The oscillators are reset by by finding new random values for ω and current-period θ; as in the original initiation, last-period θ is set to zero.

The Euler step 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, since 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. These decorators are functions which take other functions as inputs and return optimised functions. In this case, the `@jit` decorator converts my core Euler step computation function into optimised machine code using the LLVM compiler. This can yield computation speeds similar to C or FORTRAN.[1] For me, it took 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 was originally the case in my object-oriented code. Therefore, I wrote the wrapper function `oneStepForAll`. It gets the values from the `OscPopulation` object, puts them into temporary values and then calls the optimised function `_oneStepForAll` using those temporary values (namely, arrays for $\omega, \theta_{t-1}, \theta_t$). The returned arrays for θ_{t-1}, θ_t are then written onto the `Oscillator` objects in the `OscPopulation` .

The core `_oneStepForAll` function implements a standard Euler approximation. First, the θ s are stepped through time by handing the value of θ_t over to θ_{t-1}.[2] In the next step, the sum of the difference between the current oscillator n 's θ_{t-1} and all other oscillators j 's θ_{t-1} is computed.

[1] http://numba.pydata.org/
[2] This is not a whole equation-time step, but a simulation-time step.

$$sum = \sum_{j=0}^{N} = \sin(\theta_{t-1}^{j} - \theta_{t-1}^{n})$$

From this, we can calculate the discrete $\dot{\theta}_t^n$

$$\dot{\theta}_t^n = \omega_n + \frac{K}{N} * sum$$

The new θ_t of oscillator n is then

$$\theta_t^n = \theta_{t-1}^n + dt \cdot \dot{\theta}_t^n$$

This is done for all N oscillators.

After all oscillators have been stepped one simulation step δt 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 and appended to a list of all $r_K(T)$.

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

2.1.2 Program profile

To make the structure 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 (through the arrows) as well as information about the execution of the respective function. After the name, a rectangle shows first the total time spent in this function (in itself and its children), in the line below how much time was spent only in itself and finally how many times the function was called.[3]

The profile of the code for tasks 1 and 2 can be seen in figure 1.

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. In a loop around this one, we cycle through the different values for K to get the values of the desired $r(K)$ relationship. The results are then graphed in figure 2 together with the simulated points.

[3] The latter is not up to scale: to construct the profile, I ran the simulations with only 20% of the N stated in the task.

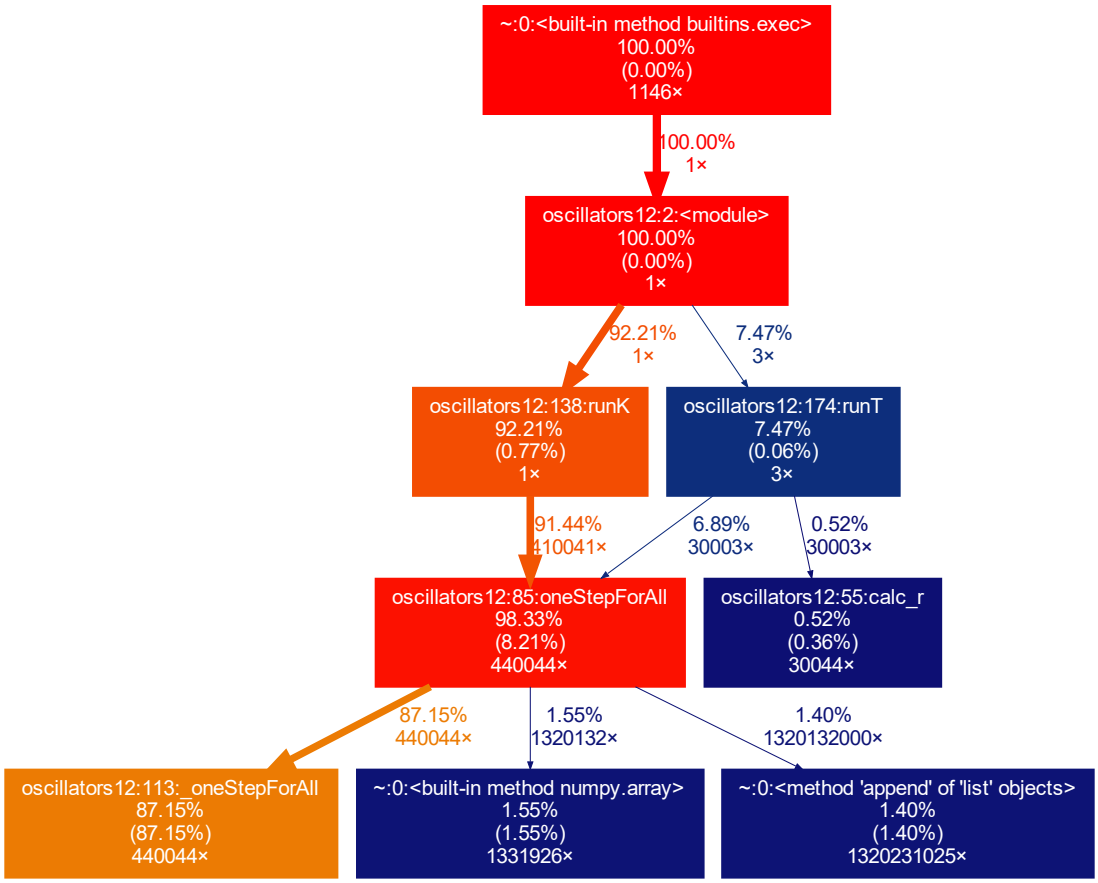


Figure 1: Profile of code for task 1 and 2

2.1.4 Results

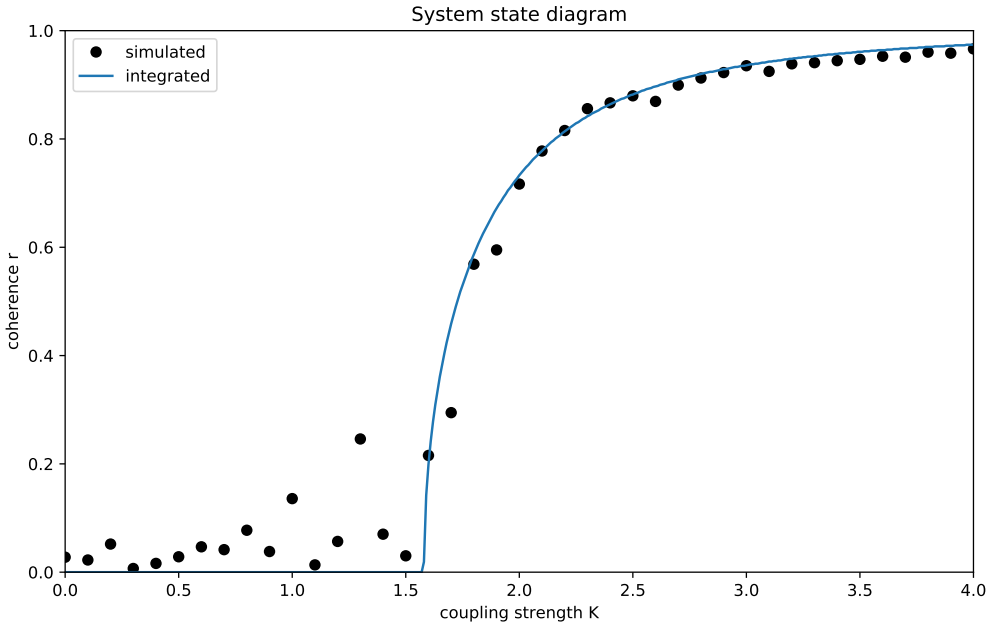


Figure 2: Diagram of limiting r_∞ for different values of K

The first observation is that

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

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

1. there is still 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 for some reason here. Instead, a bug occurred in which the final values of the last run were used as the starting values of a new run.[4] Therefore I had to go another way: now I recreate the `OscPopulation` object anew in every loop.

After that, `OscPopulation.runT()`is executed, which calls `OscPopulation.calc_r()` at every time step. `calc_r()` calculates $r(t)$ 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.

$$r_t = |\sum_{n=0}^{N} e^{\theta_t^n \cdot i}|$$

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's and Euler's formula). These two sums are also divided by N, yielding the average real and imaginary component of the `OscPopulation` at this time. From there, r can be calculated through a square root:

$$realsum = \sum_{n=0}^{N} \frac{\cos(\theta_t^n)}{N}$$
$$imsum = \sum_{n=0}^{N} \frac{\sin(\theta_t^n)}{N}$$
$$r = \sqrt{realsum^2 \cdot imsum^2}$$

I tried both versions to double check the results and hoping 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.

[4] 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

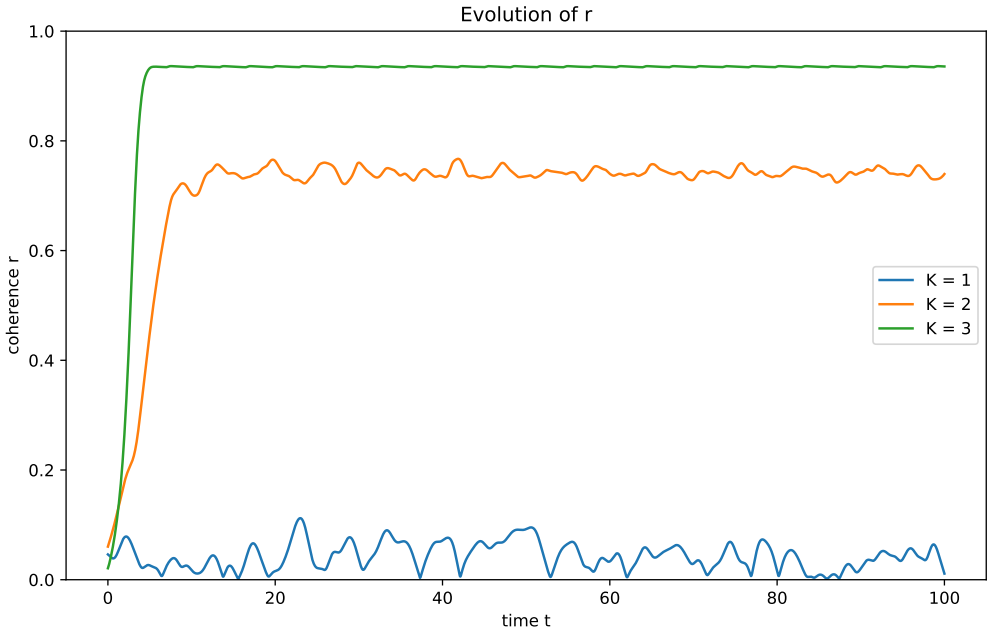


Figure 3: $r(t)$ over time for 3 different values of K

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.

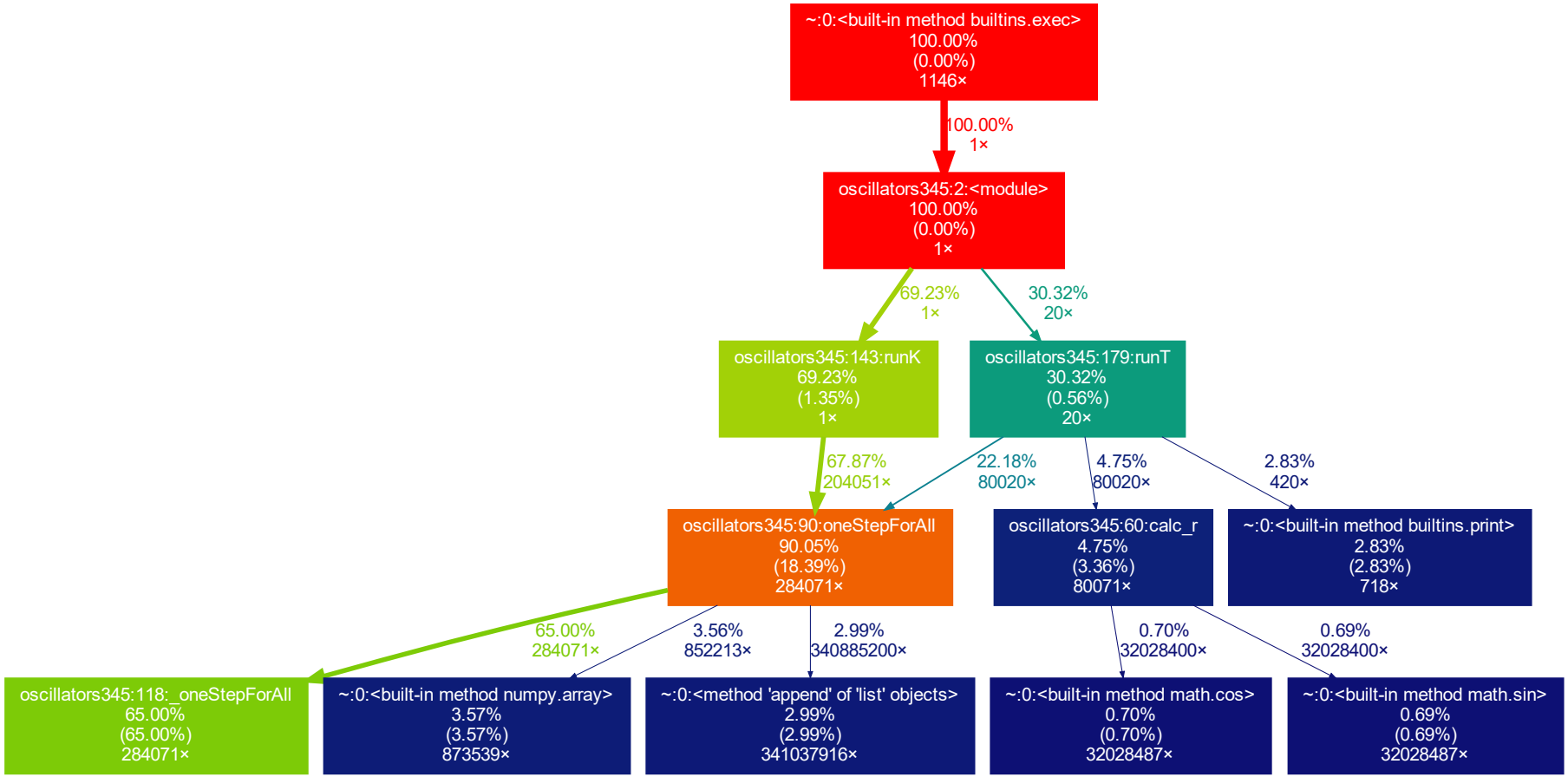
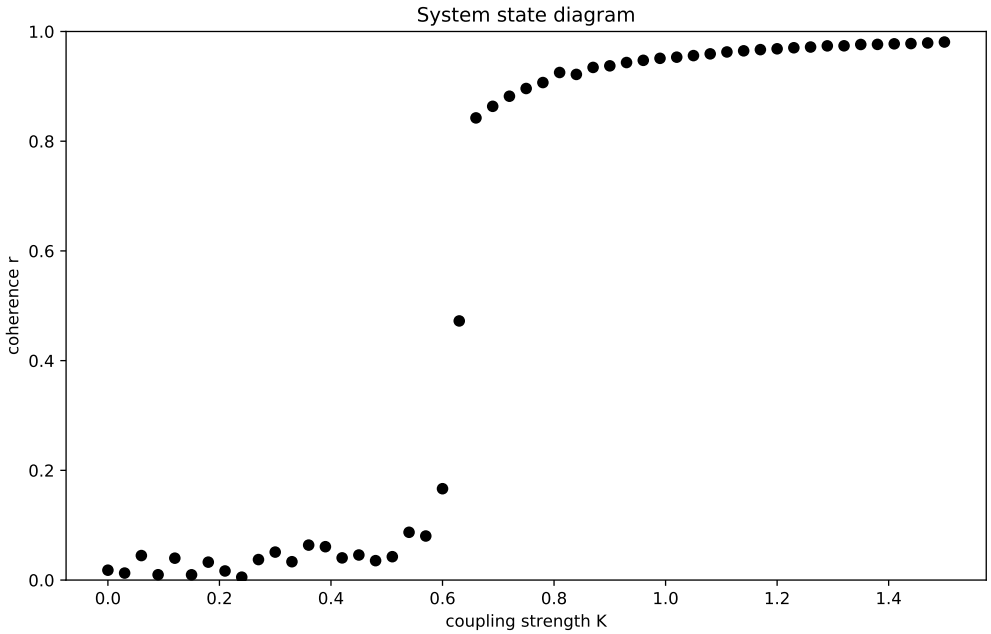
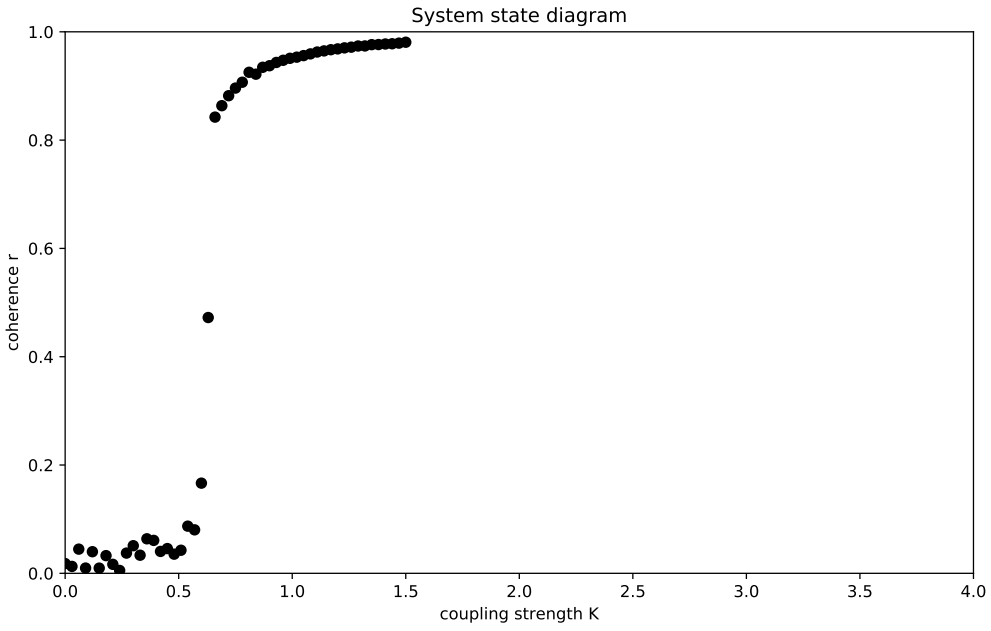


Figure 4: Profile of code for task 3, 4 and 5

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

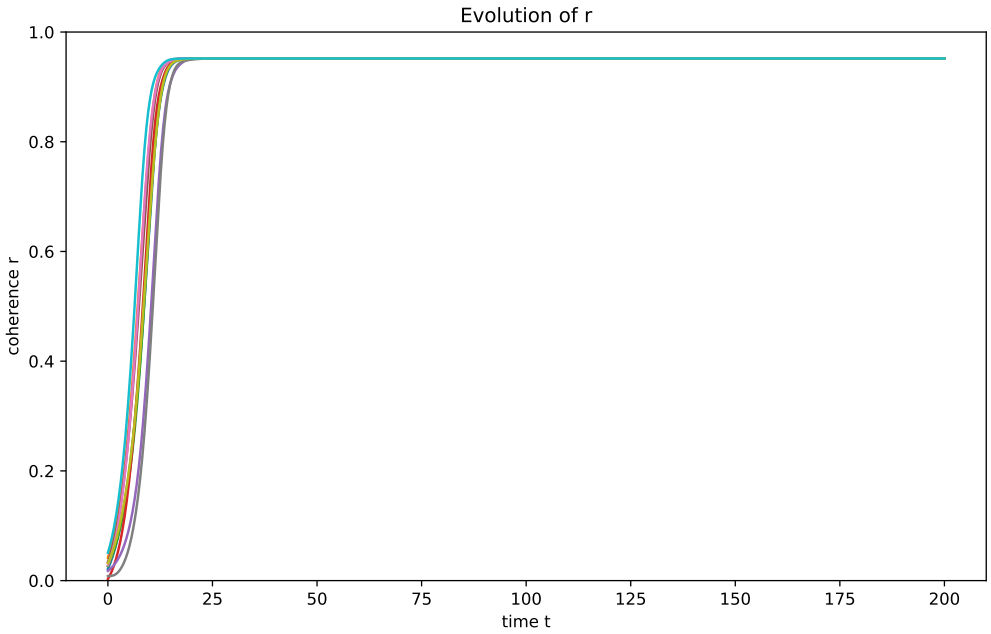


Figure 6: $r(t)$ over time for 10 different initial conditions of θ

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

2.5.1 Results

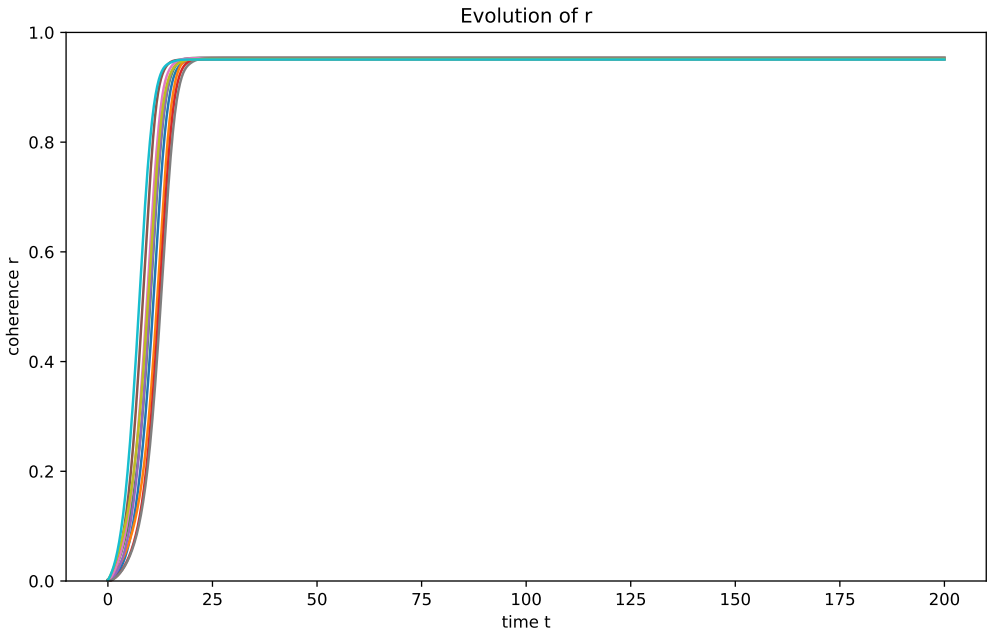


Figure 7: $r(t)$ over time for 10 different initial conditions of ω