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, `runK` and `runT`. I will get into the latter in subsection 2.2.

`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 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 `XoneStepForAll` 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 `XoneStepForAll` function implements a standard Euler approximation. First, the θ s are stepped through time by $\Delta t = 1$ by handing the value of θ_t over to θ_{t-1}. 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.

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

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

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

The new θ_t of oscillator n is then

$$\theta^n_t = \theta^n_{t-1} + dt \cdot \dot{\theta}^n_t$$

This is done for all N oscillators.

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

2.1.2 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. In a loop around this one, we cycle through the different values for K to get the values of the desired $r_\infty(K)$ relationship.

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

As opposed to `runK`

Omegas – uniformly distributed

2.3 $r_\infty(K)$

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

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