University of Groningen

Internship Report

A Framework for Multi-Agent Modeling of Social Influence

Author:
Anton Laukemper
(s379572)

Supervisors:
prof. dr. Michael Wilkinson
prof. dr. Gerard Renardel

August 16, 2019

rijksuniversiteit
groningen

1 Introduction

Multi-agent modeling, or also called agent-based modeling, is a way of examining how complex large-scale phenomena emerge from the interactions of interdependent individuals. While simple models can also be mathematically analyzed, most agent-based models rely on computational simulations. In social science, the first uses of this methodology date back to the sixties [6] but the creation of agent-based social simulations (ABSS) first gained real prominence with the introduction of the "Sugarscape" model by Joshua Epstein and Robert Axtell in 1996 [4].

ABSS are at the intersection of social science, agent-based computing and computer simulations and have the advantage that they can be used to test the logical validity of hypotheses, which helps to build and improve theoretical foundations of further experiments and informs the design of empirical research. One specific niche in the research on social phenomena is the field of opinion dynamics, where the polarization of opinions of a population and the emergence of consensus through social influence are of special interest. Social influence describes the phenomenon that individuals "modify their opinions, attitudes, beliefs or behaviour towards resembling more those of others they interact with" [7]. One of the most influential investigations into the macro-level effects of social influence on the opinion distribution of a population was presented by Axelrod in 1997 [2], when he asked the question how individuals who can only grow more similar, still exhibit cultural differences on a global scale. Since then, the research community is still tackling this question with new approaches and different kinds of models.

Much of this work builds on previous models and theories, claiming to alter only parts of it to check new assumptions and hypotheses. However, for implementing a computational simulation, the theory and all assumptions must be described explicitly and formally, with every parameter having concrete values. Often researchers make implicit assumptions in their experiments that are not necessarily reported. A survey by Marco Janssen in 2017 showed that only 10 percent of all publications on agent-based modeling actually make their source code publicly available [12]. This leads to the problem that other researchers have to start from scratch when they want to replicate, modify or extend existing simulation models. When these models are recreated, often only from a verbal description, small changes in the implementation can change the outcome of the experiment. Replication and extension of other scholars' simulation models is therefore a difficult and error-prone process. A framework for the creation of these experiments helps to alleviate some of these problems as the researchers do not have to re-implement whole simulations from scratch.

Another issue of computational simulations is their scalability. Network size and connectivity of the network are themselves variables that can influence the dynamics of processes in the network. Because of the stochastic elements of the simulations, some outcomes have a higher probability to happen in smaller networks simply by chance. Therefore we cannot necessarily infer from simulations with 100 agents what happens in a population with 100,000 agents. Results of smaller experiments thus have to be explicitly confirmed for experiments with a larger network of agents. For simulations with high computational complexity, up-scaling by such proportions can lead to enormous run-times and the need for powerful computational resources. Scalability in this context then means, that it is possible to execute an experiment with more computational resource, proportional to the up-scaling in network size or connectivity, so that the run time of the experiment does not increase extensively.

In this report, I will present a python package with the working title "MAMF - A multi-agent modeling framework". This package is a modular framework for the creation of agent-based simulations on social influence, that offers researchers to exchange sub-parts of a simulation specification, while keeping everything else as is, and at the same time a scalable way of executing the simulation.

While there already exist many frameworks for the creation of multi-agents system models like *Ascape* [16], *Swarm* [1], and the *Multi-Agent Simulation Suite MASS* [10], none of them specialize on the topic of opinion dynamics. The advantage of this framework over the existing ones is that it already offers pre-implemented building blocks and especially the infrastructure around setting up a simulation that other researchers can use, which is an attempt to unify methodology and theory of a growing field in the area of computational social science.

The following section will give background information that is useful for understanding the architecture of the software package on the one hand, and an overview of the sociology literature on which the implementation of many components of the *MAMF* package are built on the other hand. Section 3 presents the framework's general structure, each component in detail and also a validation of the results that it produces. In section 4 and 5 I evaluate the two main requirements for this package, namely its modularity and scalability respectively. The report finished with a discussion and a conclusion.

2 Literature Review

The aim of the package presented here is to be on the one hand modular and extensible and on the other hand scalable in execution time. Therefore I will first address the literature on programming methods that tackle these issues. Afterwards I refer to the sociology literature that the already implemented components of are based on.

2.1 Software Architecture

Since the framework is devised in a way that combines multiple exchangeable components, interfaces that prescribe input, output and behaviour of each component need to be defined. Each component will then have multiple realizations that all implement the same interface. A classic design pattern used for the creation of different objects that share the same interface is the factory method pattern already described by the "Gang of Four" [8]. Situations where this pattern is applicable have the following characteristics. A client (an application or component of an application) depends on a concrete implementation of an interface, but which implementation is used is variable, and further implementations could be added. In this situation a factory method is implemented that is responsible for instantiating the implementations of the interface. The client can then request a specific instantiation from the factory method with the help of an identifier, that determines which implementation is chosen. The factory method returns the concrete implementation according to the value of the identifier to the client, who can then use that object just like any other object. This pattern offers encapsulation and separation of concerns because the object and its creation are decoupled from another. The client does not care how the object is created, and they do not need to change if the object changes, as long as the object still satisfies the interface. If another implementation is added, that implementation has to be added only in one place, namely the factory method, which makes maintenance of the code base much easier.

Another fundamental principle of programming is the SOLID principle [14], which is more of a set of programming guidelines. One of these guidelines, the interface-segregation principle (ISP) [14] is very important for a modular and extensible system. Since interfaces typically force anyone that is using that interface to implement any function that it specifies, large interfaces potentially run into the problem of forcing individual clients, i.e. the class that implements the interface, to implement methods it does actually not need. The ISP therefore recommends to keep interfaces as small as possible, creating so called role interfaces that just define one method. This avoids situations in which the change of one part of an interface forces every of its clients to change as well, even though they might not even be addressed by that change.

Besides being modular and extensible, the other requirement for the simulation framework presented here is a scalable execution, meaning that it is possible to increase the network size, network connectivity or number of simulations without an explosion in computation time as long as the computational resources are increased accordingly. This can be achieved by parallelizing the execution of the experiment and distributing the necessary computations to multiple cores or different machines. The research fields of parallel computing and distributed computing study how this can be done optimally, so a short introduction into this topic follows.

Since the yearly increase of processing power of computer chips has been curbed by physical limitations, manufacturers are relying on putting multiple cores into one central processing unit to increase performance [15]. Each core is able to run its own tasks independently from the other cores and if they need to communicate they can do it easily because they are all located on the same chip. Despite the fact that this hardware architecture has potential for running computationally intensive programs, most programs cannot by nature exploit this potential. They often have to be modified or completely rewritten in order to be executed parallely.

Although there is no clear distinction between parallel computing and distributed computing, there seems to be agreement that parallel computing entails that multiple tasks are run on cores that are physically close to each other, usually on the same machine, while distributed programming describes the execution of independently created programs running on different loosely coupled machines that work together on solving the same tasks [15].

Another distinction can be drawn along the lines of memory. The processing units can either all share the same working memory, so that each unit can read and write to each location in the shared memory, or alternatively each unit has their own working memory. In a shared-memory system the processing units can be coordinated through updates of the same memory locations while in a distributed-memory system the units must be coordinated explicitly with messages through a network [15].

In distributed computing programs, each machine in the system, also called node, will definitely have their own memory, while a parallel program can run on a shared-memory system or a distributed-memory system.

Broadly generalized this leaves us with two ways of parallelizing a task.

Either by running it on multiple cores or processors of one computer (multi-processing) or by distributing it to a cluster of multiple computers. Furthermore, there is a third way applicable to programs where coordination and communication does not matter at all. These are so called “embarrassingly parallel” programs. This is often the case if we want to run the same program with different parameter inputs, or trivially repetitions with the same parameters, because then we can just run one instance of the program on each processing unit available.

The framework presented in this report will make use of a mixture of these three approaches by calling a number of nodes in a cluster independently from each other with different parameters. Every node then computes its part of the problem on all of its cores in a multi-processing approach.

2.2 Types of Social Influence Models

The social influence process as it is implemented in this package is based mainly on the paper “The Dissemination of Culture” by Robert Axelrod, published in 1997 [2], and subsequent variations and extensions of it. Axelrod tried to find an answer to the question why we observe high diversity in culture on the macro level even though individuals tend to be influenced by others, getting more similar culturally on the micro level. He created a agent-based model and wanted to find out whether it is possible to observe multiple distinct cultures if the only assumptions are that *culture* is represented as a set of multiple categorical traits, more similar people are more likely to interact (*homophily*), which also includes the assumption that there is no interaction between maximally dissimilar individuals, and people that interact a lot grow more similar (*positive influence*). Each agent has a vector with multiple features that could represent for example favourite band, favourite dish and favourite football club, that can take values from a finite set of traits e.g. Iron Maiden, Jethro Tull or Beatles for favourite band, spaghetti carbonara, hamburger or sauerkraut and sausages for favourite dish, etc. His simulation has discrete time steps and in each step an agent is picked randomly from the network, another agent from their neighborhood is picked and with a probability proportional to the share of cultural traits they have in common, the agent selected first adopts a trait, they previously disagreed on, from the other agent. If their cultural similarity is 0, i.e. they do not share any traits, they cannot influence each other. As soon as the cultural similarity between each connected agent in the network is either 1 or 0, the simulation converged because no influence can happen since all agents are either totally similar, or totally dissimilar.

Axelrod's results showed that higher numbers of features decrease the overall cultural diversity in equilibrium because a larger feature vector increases the probability that there is at least some cultural overlap, enabling agents to become more similar. If the number of traits per features is higher, the opposite effect is true and there are multiple clusters of agents with the same feature vector at the point of convergence.

Multiple follow-up simulation experiments were carried out after this study. One important addition was the introduction of different, so-called, "communication regimes" that determine how the agents interact. Flache and Macy [5] tested a model where an agent was not influenced by one other agent, but rather by multiple of its neighbors at the same time. Keijzer, Mäs and Flache [13] implemented a communication regime representing the broadcasting communication on social media, where one agent influences multiple other neighbors per time step.

Other modifications include the use of continuous instead of categorical features, often combined with a threshold value at which it is assumed that agents are too dissimilar to still interact [9, 3]. At this threshold the 'bounds of confidence' into the expertise or integrity of the other person are reached, so they are also called "bounded confidence" models [7].

The literature offers many more extensions such as a public and private opinion with different influence functions [17] or models of negative or also called repulsive influence [11] and there are still many possible assumptions and theories unexplored. This framework therefore aims to facilitate the combination of different model assumption, and the extension of the theory that has been tested so far.

3 The Framework

This package's purpose is to provide a framework for the creation of multi-agent simulation experiments on social influence and similar social processes. The design of the framework rests on the idea that the core process of any social influence model is on an abstract level the same. On the structural level, there is always a number of agents with certain attributes in a network. In each time step of the simulation an agent from the network will be picked, a subset of the other agents in the network are chosen, and then they exert either one-directional or bi-directional influence on each other.

The framework builds upon the python package *networkx*, which is a tool for the creation of graphs with nodes, edges and possibly attributes for both of these. It furthermore offers functions to analyze and visualize these graphs and thus provides essential elements for a program that is supposed to simulate and analyze the processes that emerge in a social network of agents. Additionally, the *pandas* package is used to create so called “DataFrames”, which are tabular data structure objects used in this case to store the output of the simulations.

The main goal of this package is the possibility to exchange specific parts of an existing model, or to extend it with new parts, while keeping everything else the same. To achieve this modularity, the process of the simulation is split into different components, so that each component is only responsible for one maximally reduced part of the process. For these components a generalized interface is defined that prescribes the input, behaviour and output of the components’ functions. This way, each component can be implemented in various ways, and the concrete realization can be replaced as long as each implementation adheres to the defined interface.

To achieve easy handling of multiple concrete implementations of the same interface, the factory method pattern was used. Since python does not support interfaces, abstract base classes serve as the interfaces. An abstract base class is a class with fields and methods like any other class except that it can also contain abstract methods that have no body and are marked with a ”@abstract” decorator tag. Any class that inherits from the abstract base class must then provide an implementation for each abstract method or it cannot be instantiated. For each component a factory method exists that receives a string as an identifier and based on this identifier chooses the concrete implementation of the component.

Since all methods of the components are “static” methods, i.e. class methods, and are thus not called on an instance of the class but the class itself, the factory method pattern is here not applied in its pure form, because the factory methods do not return instances of the requested classes but rather call the relevant method on that class and return the output, if it exists. This is also the reason why in most components the factory function is named after the main method of the respective class it produces. It can be distinguished by its additional parameter “realization” which serves as the identifier for the factory.

As every implementation of the functions could be parameterized in their own specific way, which could not be captured by a generalized abstract function, there needs to be an option to pass implementation-specific parameters to these functions. In *python* this can be elegantly done with the ***kwargs* argument shown in listing 1. It stands for “key-worded arguments” and is a dictionary object containing as the dictionary key the name of a parameter, and as the dictionary value the parameter’s value. This way, the concrete implementation of an abstract method can receive multiple parameters as arguments additional to the arguments specified by the interface for that function.

```
# a function defined with one necessary argument and the kwargs
    argument
def kwargs_example(parameter1, **kwargs):
    print("parameter 'parameter1' has value %s" % parameter1)
    for key, value in kwargs.items():
        print("parameter '%s' has value %s" % (key,value))
# can be called like this
kwargs_example("first_parameter",
                parameter2="second_parameter",
                parameter3="third_parameter")
# or with an unpacked dictionary (unpacking is achieved through
    the two asteriks):
parameter_dict = {"parameter2": "second_parameter",
                "parameter3": "third_parameter"}
kwargs_example("first_parameter", **parameter_dict)

# and produces the following output:
parameter 'parameter1' has value first_parameter
parameter 'parameter2' has value second_parameter
parameter 'parameter3' has value third_parameter
```

Listing 1: Example code showing how the ***kwargs* argument can be used

Figure 1 shows the overall structure of the package with all of its relevant subpackages. Not shown are folders related to documentation purposes and test, as well as the *setup.py* file that is needed for installing the package.The "__init__.py" files are empty files that designate which folders function as python (sub)packages

The framework specifies two components for the initialization of a simulation (*network_init* and *agents_init*) four components for running a simulation (*focal_agent_sim*, *neighbor_selector_sim*, *influence_sim*, *network_evolution_sim*) and a component that is responsible for determining how "dissimilarity" or "distance" is defined (*dissimilarity_component*). There is one subpackage for each component plus an additional subpackage called *tools* that contains various functionalities that are not crucial for the execution of a simulation but support it in some way. While these components could be imported and used directly by the user, the package also offers two modules containing classes that automatically plug the components together and allow for an easier setup of either a single simulation or an experiment consisting of multiple simulations. These modules are therefore called *Simulation* and *Experiment* and can be found in the root package.

Except for the *network_init* component and the *tools* subpackage, the subpackages are all structured in the same way. Each has one module with the same name as the component which contains the abstract base class and a factory function. Furthermore, it has one module for each concrete implementation of the abstract base class, distinguished by their names in camel case instead of snake case. The reason for separating the implementations from the base class into different modules was to have independent imports and to avoid that it turns into one very long, unreadable, module as more implementations get added to the package.

```
mamf
├── network_init
│   ├── __init__.py
│   └── network_init.py
├── agents_init
│   ├── __init__.py
│   ├── agents_init.py
│   ├── RandomCategoricalInitializer.py
│   └── RandomContinuousInitializer.py
├── focal_agent_sim
│   ├── __init__.py
│   ├── focal_agent_sim.py
│   └── RandomSelector.py
├── neighbor_selector_sim
│   ├── __init__.py
│   ├── neighbor_selector_sim.py
│   └── RandomNeighborSelector.py
├── influence_sim
│   ├── __init__.py
│   ├── influence_sim.py
│   ├── Axelrod.py
│   └── BoundedConfidence.py
├── network_evolution_sim
│   ├── __init__.py
│   ├── network_evolution_sim.py
│   └── NetworkHomophily.py
├── dissimilarity_component
│   ├── __init__.py
│   ├── dissimilarity_calculator.py
│   ├── HammingDistance.py
│   └── EuclideanDistance.py
├── tools
│   ├── __init__.py
│   ├── NetworkDistanceUpdater.py
│   ├── OutputMeasures.py
│   └── ClusterExecutionScript.py
├── __init__.py
├── Experiment.py
└── Simulation.py
```

Figure 1: The relevant package structure.

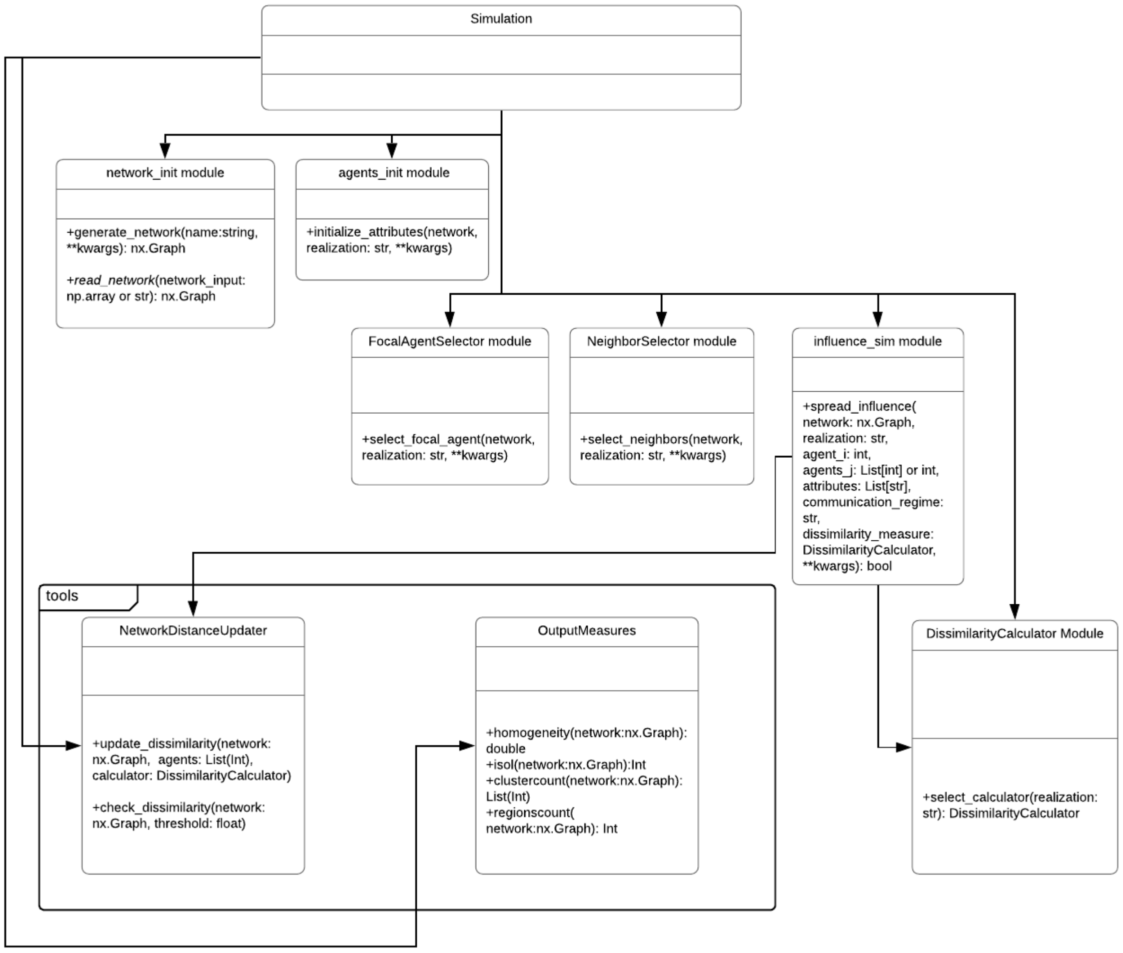

Simulation
network_init module
+generate_network(name:string, **kwargs): nx.Graph
+read_network(network_input: np.array or str): nx.Graph
agents_init module
+initialize_attributes(network, realization: str, **kwargs)
FocalAgentSelector module
+select_focal_agent(network, realization: str, **kwargs)
NeighborSelector module
+select_neighbors(network, realization: str, **kwargs)
influence_sim module
+spread_influence(network: nx.Graph, realization: str, agent_i: int, agents_j: List[int] or int, attributes: List[str], communication_regime: str, dissimilarity_measure: DissimilarityCalculator, **kwargs): bool
tools
NetworkDistanceUpdater
+update_dissimilarity(network: nx.Graph, agents: List(Int), calculator: DissimilarityCalculator)
+check_dissimilarity(network: nx.Graph, threshold: float)
OutputMeasures
+homogeneity(network:nx.Graph): double
+isol(network:nx.Graph):Int
+clustercount(network:nx.Graph): List(Int)
+regionscount(network:nx.Graph): Int
DissimilarityCalculator Module
+select_calculator(realization: str): DissimilarityCalculator

Figure 2: A UML diagram showing which class or module calls which modules. The *Experiment* class is left out because it does not call any other classes' methods but only creates *Simulation* instances.

3.1 Components

Figure 2 shows a diagram that depicts which class calls methods of other classes or modules. Most of the function calls go out from the *Simulation* class, but the *influence_sim* module also has access to the *DissimilarityCalculator* class and the *NetworkDissimilarityUpdater* because it might need to calculate and update the dissimilarity between agents in the network. Each subpackage and module of the main package will be described in more detail in the following sections.

3.1.1 network_init

The task of the *network_init* component is to create the *networkx* graph object around which the rest of the simulation process revolves. While this package is still designed along the lines of the factory method pattern, it is the only package without an abstract base class because the *product* is the already defined networkx graph. Nevertheless, there are still different kinds of networks that can be created. This happens in the functions "_produce_grid_network", "_produce_ring_network", and "_produce_spatial_random_graph" shown in figure 3. The factory method in this module is called "generate_network" that takes the desired network topology and potential parameters for its creation as an argument and then calls one of those three function to return the respective graph object. Furthermore it includes the function "read_network" that either takes a path to a file containing an edge list or a numpy array with an adjacency matrix as arguments and from this creates a networkx graph.

This subpackage is the only component where the *products* and the factory function are together in the same module. The reason for this is that the range of possible network structures that the package would offer in a finished version is rather limited. If a user wants a very specific network topology they should create it manually and pass it to the *Experiment* or *Simulation* class, that will be later described, as an argument. It is not the task of this package to offer a large range of possible network topologies.

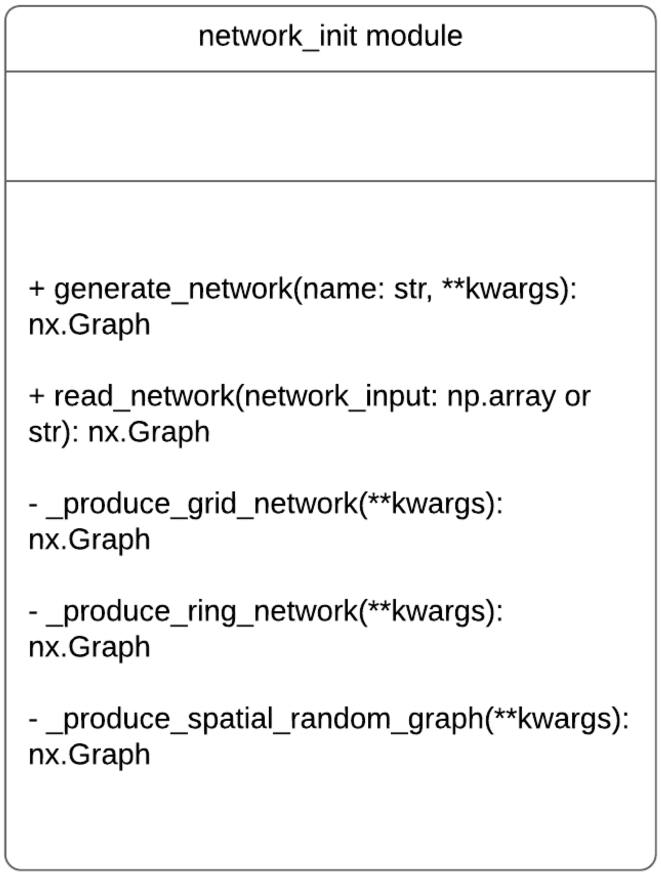

network_init module
+ generate_network(name: str, **kwargs): nx.Graph
+ read_network(network_input: np.array or str): nx.Graph
- _produce_grid_network(**kwargs): nx.Graph
- _produce_ring_network(**kwargs): nx.Graph
- _produce_spatial_random_graph(**kwargs): nx.Graph

Figure 3: The functions of the *network_init* module.

3.1.2 agents_init

In the *agents_init* component the attributes of the agents in the network are initialized. The module contains the abstract base class *AttributesInitializer* which has two already implemented method "set_categorical_attribute" and "set_continuous_attribute" that give to each agent in the network a new attribute and draw its value from a given distribution. These methods can be used by the concrete implementations of this class. The base class further defines the "initialize_attributes" method. Its arguments are the graph object on which the agents shall be initialized and a key-worded arguments dictionary, that contains potential arguments to the concrete implementations of this method. The implementations then determine how exactly the attributes of an agent get initialized, e.g. randomly, dependent on the position in the network, or correlated to the attributes of its neighbors. Which implementation is used to initialize the agents can be decided with the factory function that is also located in the *agents_init* module. The whole structure of the subpackage is visualized in fig. 4

Implementations of the *AttributesInitializer* can potentially be used to set all kinds of attributes, also those that are not able to change through the process of influence as for example sex and age. Combinations of discrete and continuous attributes are therefore theoretically possible, however in the current version there is no *DissimilarityCalculator* that supports a mixture of both types.

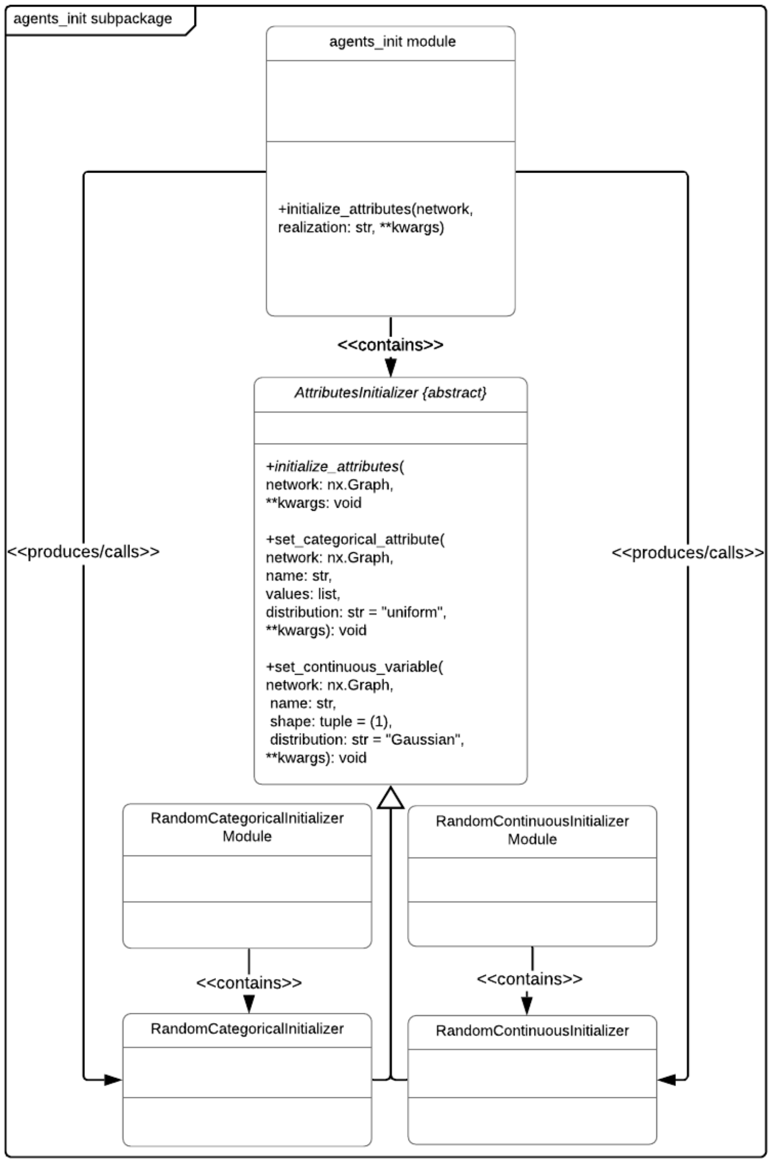

agents_init subpackage
agents_init module
+initialize_attributes(network, realization: str, **kwargs)
<<contains>>
AttributesInitializer {abstract}
+initialize_attributes(network: nx.Graph, **kwargs: void
+set_categorical_attribute(network: nx.Graph, name: str, values: list, distribution: str = "uniform", **kwargs): void
+set_continuous_variable(network: nx.Graph, name: str, shape: tuple = (1), distribution: str = "Gaussian", **kwargs): void
<<produces/calls>>
<<produces/calls>>
RandomCategoricalInitializer Module
RandomContinuousInitializer Module
<<contains>>
<<contains>>
RandomCategoricalInitializer
RandomContinuousInitializer

Figure 4: A more detailed diagram of the *agents_init* subpackage. The following components' subpackages are designed in the same way.

3.1.3 focal_agent_sim

The only responsibility of this component is to pick the focal agent for the next influence step from the network. Specified by the interface *FocalAgentSelector* is thus only the "select_agent" method which returns the index of the chosen agent. This focal agent will then either be the source or the target of influence, depending on the communication regime. The agent could be chosen randomly, or from a distribution that makes e.g. agents with more ties more likely to be picked, reflecting the potential mechanism that more popular agents get more chances to influence, or be influenced. The decision of how the agent is chosen will then be made in the concrete implementations of the *FocalAgentSelector*

3.1.4 neighbor_selector_sim

Similar to the *focal_agent_sim* component, this component is responsible for choosing the partner(s) of the focal agent for the influence process. The abstract base class *NeighborSelector* defines the "select_neighbors" method that receives the index of the focal agent and the communication regime under which the agents operate, and returns the indices of the selected agents. In the one-to-one case, one single neighbor from the focal agent is chosen, while in the one-to-many and in the many-to-one case multiple neighbors are chosen.

The NeighborSelector's responsibility is explicitly not to select the neighbors based on their dissimilarity to the focal agent, because this is considered to be part of the influence component, described in the next subsection. Rather this component focuses only on the attributes of the actors which could e.g. select agents from the opposite sex with a lower probability to model segregational mechanisms.

3.1.5 influence_sim

The influence component certainly has the most important task in the process and has also the most extensive interface. The module contains the *InfluenceOperator* as an abstract base class which defines the "spread_influence" method. It receives the indices of the focal agent as well as those of the selected neighbors, a list of attributes that are allowed to be changed in the influence step, the communication regime, a dissimilarity calculator that will be used to update changed distances, and a key-worded argument dictionary for potential parameters of the concrete influence function. From that it determines whether the actors are able to influence each other, by checking their dissimilarity, decides how they influence each other, e.g. determine the attribute that will get modified, and then changes the influenced attributes on the respective agents. The list of attributes is needed because the agents could potentially have attributes like sex or age that are not able to be influenced.

In an earlier version, the abstract class defined the "one_to_many" method and the "many_to_one" method, as any other communication regime is just a special instance of one of these two. However, since some influence functions are not meant to work in both regimes, the interface was too large, violating the interface segregation principle, so it was reduced to just one method that takes the communication regime as a parameter.

3.1.6 network_evolution_sim.

The idea behind this component is that the network structure might change over time. The abstract base class "NetworkModifier" defines the method "rewire_network" that removes existing ties or adds new ones depending on the attributes of the agents, their position in the network, or simply randomly. Its arguments are simply the network that will be modified and a key-worded arguments dictionary with possible parameters. This way mechanisms like homophily in tie selection can be incorporated in the simulation.

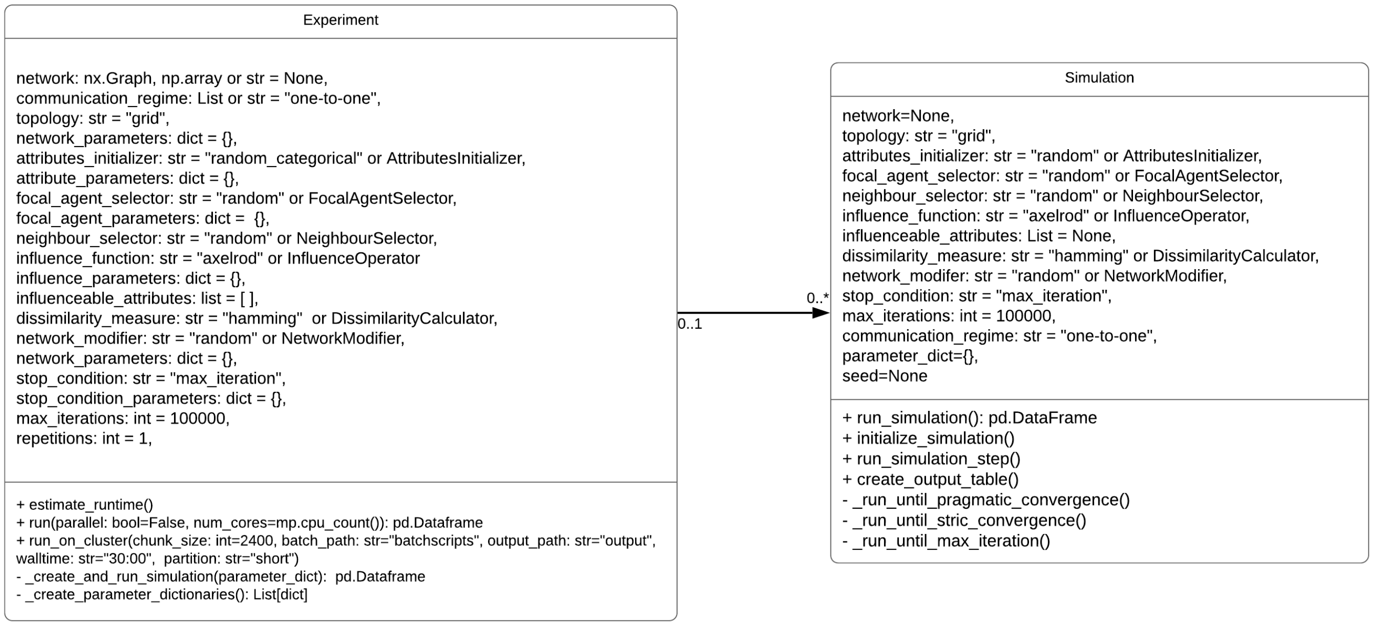
3.1.7 dissimilarity_component

The dissimilarity component contains the "DissimilarityCalculator" class which is the abstract base class that defines the interface for the classes that are used to determine how *dissimilar* two agents are to each other. This dissimilarity could either be based on their attributes or other conceptions of similarity, e.g. how many neighbors they have in common.

The interface defines two methods, "calculate_dissimilarity", that calculates the dissimilarity between two agents and returns that value, and "calculate_dissimilarity_networkwide" which is called once after the initialization of the network. It calculates the distance between each neighbor in the network and sets that value as an attribute on the edge between them. This way it can be easily accessed by the influence function and does not have to be computed each time, which saves computation time.

3.1.8 Simulation and Experiment

All of the previously mentioned components can be seen as building blocks to create one's own simulation of social influence processes, so they can all be used directly. However, to increase user experience quality and to have a simple and accessible way to set up the experiments, the package includes the *Simulation* class and, as a wrapper around it, the *Experiment* class. All parameters and methods of these classes are shown in fig. 5.



```
Experiment

network: nx.Graph, np.array or str = None,
communication_regime: List or str = "one-to-one",
topology: str = "grid",
network_parameters: dict = {},
attributes_initializer: str = "random_categorical" or AttributesInitializer,
attribute_parameters: dict = {},
focal_agent_selector: str = "random" or FocalAgentSelector,
focal_agent_parameters: dict =  {},
neighbour_selector: str = "random" or NeighbourSelector,
influence_function: str = "axelrod" or InfluenceOperator
influence_parameters: dict = {},
influenceable_attributes: list = [ ],
dissimilarity_measure: str = "hamming"  or DissimilarityCalculator,
network_modifier: str = "random" or NetworkModifier,
network_parameters: dict = {},
stop_condition: str = "max_iteration",
stop_condition_parameters: dict = {},
max_iterations: int = 100000,
repetitions: int = 1,

+ estimate_runtime()
+ run(parallel: bool=False, num_cores=mp.cpu_count()): pd.Dataframe
+ run_on_cluster(chunk_size: int=2400, batch_path: str="batchscripts", output_path: str="output",
walltime: str="30:00",  partition: str="short")
- _create_and_run_simulation(parameter_dict):  pd.Dataframe
- _create_parameter_dictionaries(): List[dict]
```

```
Simulation

network=None,
topology: str = "grid",
attributes_initializer: str = "random" or AttributesInitializer,
focal_agent_selector: str = "random" or FocalAgentSelector,
neighbour_selector: str = "random" or NeighbourSelector,
influence_function: str = "axelrod" or InfluenceOperator,
influenceable_attributes: List = None,
dissimilarity_measure: str = "hamming" or DissimilarityCalculator,
network_modifer: str = "random" or NetworkModifier,
stop_condition: str = "max_iteration",
max_iterations: int = 100000,
communication_regime: str = "one-to-one",
parameter_dict={},
seed=None

+ run_simulation(): pd.DataFrame
+ initialize_simulation()
+ run_simulation_step()
+ create_output_table()
- _run_until_pragmatic_convergence()
- _run_until_stric_convergence()
- _run_until_max_iteration()
```

Figure 5: A UML diagram with parameters, their types and default values, and the methods of *Simulation* and *Experiment* class

The constructor of a *Simulation* takes among other arguments the specifications of which realization of each component should be used. This can be either done by passing a string that will then be used as an identifier for the respective factory methods, or by passing an instance of a class the implements the respective interface. This way the user can implement their own realization of a component and still use the *Simulation* class to automatically run the simulation.

Other parameters of this class include the communication regime, the maximum number of iterations it shall run, and more importantly, it gets a dictionary that contains all additional parameters that need to be passed to the specific simulation components. This dictionary contains the name of the parameter as a key and the value for it as a value. Each component receives this dictionary and accesses via the key the parameter it needs. After a Simulation object is created, the user has two options for running the simulation. Either the simulation can be initialized and run manually with the two functions "initialize_simulation" and "run_simulation_step", or one can use the fully automatic "run_simulation" method that itself calls the previous two methods. For the use of this method a stop-condition has to be determined which defaults to running the maximum number of iterations. The other currently available stop conditions either check whether influence is theoretically still possible and stop if it is not, or check whether the network has not changed for a certain amount of time steps.

Because a proper experiment does not consist of only one simulation, the package also offers with the *Experiment* class a way to easily create and run multiple simulations, if needed, at the same time. The Experiment takes almost the same arguments as the Simulation, but the difference is now that the parameter values can be a list of multiple values. Inside the "_create_parameter_dictionaries" method of the class, a dictionary is created for each possible combination of values from these lists. All of these dictionaries are then combined in a list. Since the results of one simulation do not have much meaning because of the stochastic nature of the process, an additional argument determines the number how often each simulation shall be repeated.

The Experiment class has also two methods for the user to run all simulations. The "run" method either creates and runs one simulation after the other for each parameter combination from the input parameters, or it runs multiple simulations in parallel on the different cores of the machine if the "parallel" parameter is set to "true". The number of cores that shall be used for that can be set with the "num_cores" parameter. This method makes use of the "multiprocessing" python package, which allows for the utilization of all processors of the machine on which the experiment is run. Each processor creates and runs a simulation with one parameter combination from the list of parameter dictionaries, returns the results and then receives the next parameter combination to run. All results are combined to one *DataFrame* object.

Alternatively one can also use the "run_on_cluster" method if the experiment shall run on a SLURM-based high performance computing cluster. SLURM is a workload manager for supercomputers and computer clusters. It is a software used in the majority of the TOP500 supercomputers to distribute jobs, i.e. computational tasks, to the right machines in the cluster.

Other workload managers that use a similar interface for submitting jobs can also be supported in future versions.

By distributing the experiment on a cluster its execution can be parallelized on multiple machines instead of only one. If this method is called, the list of parameter combinations is shuffled, split into chunks and then for each chunk, a job is submitted to the cluster manager. Each job calls the *ClusterExecutionScript* script contained in the *tools* subpackage, which runs all parameter combinations from one chunk in parallel on one machine. Each job creates their own csv file as output, which have to be merged to one file to get the results for the whole experiment. The list of parameter dictionaries is shuffled before it is split into chunks, so that each job takes approximately the same time.

The "run_on_cluster" takes parameters that modify the settings for the job on the cluster such as the desired partition from which the node shall be selected and the time that is allocated to the job. Other parameters determine where the files that are produced by the method will be stored. The most important parameter is the "chunk_size" because it will specify to how many different machines on the cluster the execution of the simulations will be distributed.

3.1.9 tools

The *tools* package includes all modules that are not necessarily part of a simulation but still support it in some way. The *NetworkDistanceUpdater* contains the function *update_dissimilarity()* which receives a list of nodes and a *DissimilarityCalculator* and then uses that calculator to recalculate the dissimilarity between the agents in the list and all of their neighbors. This function is usually called by the *InfluenceOperator* after the feature vectors of one or multiple agents changed. Furthermore, the module contains the *check_dissimilarity()* function which checks for every edge in the network whether its dissimilarity attribute is lower than a given threshold. If all of them are lower, *False* is returned because no change is possible anymore. This function is called by the *Simulation* class to check for convergence.

Another modules contains a function for each output measure that is taken from the network either after convergence or in-between time steps and is called by the *Simulation* class or by the user manually.

The last module, "ClusterExecutionScript", is used as a script and called by the *Experiment* class when it is run on a cluster. The script receives three arguments when it is executed. First, the index of the chunk of parameter combinations it is supposed to run, second, the path to a folder of serialized python objects that contain all parameters necessary for running the simulations, and third the path to the folder where the output shall be written to.

3.2 Validation

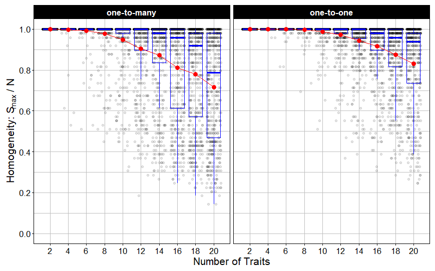
To validate whether the package works as intended the experiment on cultural complexity from [13] was replicated. This experiment extends a variation of the classical social influence experiment by Axelrod with the 'one-to-many' communication regime. That means that every time step the focal agent interacts with all of its neighbors, as long as they are not perfectly similar or dissimilar. The focal agent then influences them regarding a feature on which he disagrees with at least one neighbor.

The experiment in question investigates what effect *cultural complexity*, i.e. the dimensionality of the feature vector and the number of possible traits, has on the cultural diversity of a population at the point of convergence.

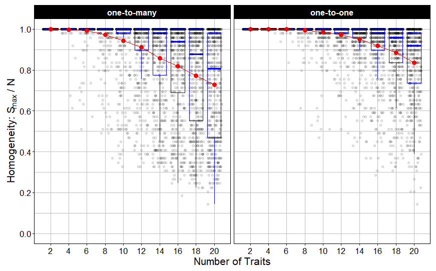
Figure 6, 7 and 8 show comparisons between the two communication regimes on the level of the experiment but also comparisons between the code that was used for the experiments published in [13] and the same experiment implemented with the MAMF package. Both experiments show qualitatively the same results. A higher number of features gives more opportunities to be similar, increasing the probability that an agent can be influenced. Over time, it is then more likely that all agents eventually adopt the same traits for each feature.

Increasing the number of traits per feature has the opposite effect. It becomes increasingly unlikely that two agents have any features in common, creating boundaries where no agents can influence each other anymore. This leads to a higher number of isolates and a lower chance of consensus in the network.

While this does not prove the correctness of the framework package, it indicates that the implemented Axelrod influence function behaves as it is supposed to and the surrounding infrastructure does not add negative side effects.

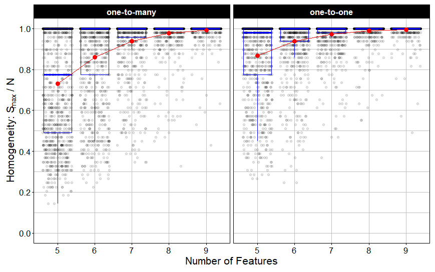


(a) Original Code

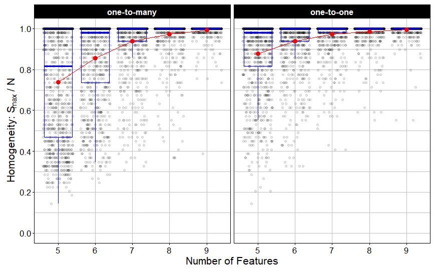


(b) With Framework

Figure 6: Comparing the relationship of the number of traits to the homogeneity

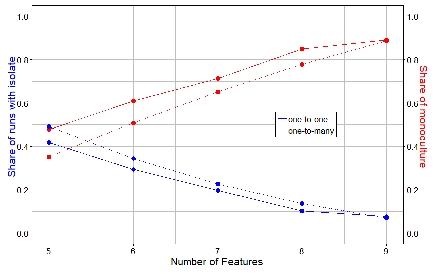


(a) Original Code

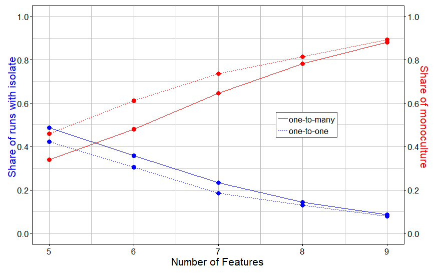


(b) With Framework

Figure 7: Comparing the relationship of the number of features to the homogeneity



(a) Original Code



(b) With Framework

Figure 8: Comparing the number of isolates

4 Modularity and Extensibility

4.1 Modularity

The architecture of this framework was designed in such a way that many different implementations of each component can be added independently from another, so that it eventually offers a suite of building blocks that can all be combined with each other to cover the majority of relevant social influence simulations. The package can naturally grow with new theoretical discoveries and is open to custom implementations of the components, so that each researcher can adapt the package to their needs.

Even though the components are supposed to be decoupled from each other through their interfaces there are still some dependencies that need to be considered when creating a simulation. The decision with the most impact on the other modules is the choice of the attribute type of the agents. On the one hand you can run into problems because the combination of attribute type and DissimilarityCalculator might work but not give meaningful results, like calculating the Hamming distance between two continuous feature vectors, which will always be close to one even though they might be quite similar. On the other hand some combinations are on a practical level not possible as for example calculating the euclidean distance between two nominal feature vectors. The choice of attribute type is equally relevant for the *influence_sim* component as certain influence functions calculate for example the average of the feature values or subtract or add values to them and thus presuppose continuous values.

To create a package that only accepts working input would certainly be an ambitious project, because every possible combination would need to be tested for validity. It was for this framework rather decided that the responsibility of creating valid input lies within the user and it is the package's responsibility to only throw informative errors and warnings if the user chooses incompatible building blocks.

4.2 Extensibility Example Case: Bounded Confidence

The first version of the framework included only an implementation of the classic Axelrod experiment. That means that there are only categorical features, the distance between the agents is determined with the hamming distance, and the Axelrod influence function in which the probability to influence each other is dependent on the dissimilarity between each other. The basic version of the package was supposed to include at least one other known influence model, so I decided to implement the “bounded confidence” model because it also presupposes in most cases continuous attributes [9].

In this section I describe how to add a new model to the package at the example of the bounded confidence model. This model is based on theories from cognitive psychology as well as social psychology and assumes that agents’ opinions move closer to each other when they interact, unless they disagree to much, in which case no influence is possible.

Since in this model opinions gradually approach other opinions, it assumes a continuous feature space and thus a new *AttributesInitializer* was needed. The class *RandomContinuousInitializer* was created in a new module with the same name in the *agents_init* subpackage and it inherits the abstract base class *AttributesInitializer.* To be able to call it easily from the *Simulation* class, the option to choose this realization of the *AttributesInitializer* had to be added to the factory method in the *agents_init* module. To make this addition also visible to the user, the documentation of the *Experiment* and *Simulation* class needed to be changed.

Since the selection of the agents for the influence process stays the same, no new FocalAgentSelector or NeighborSelector had to be implemented.

Because the Hamming Distance produces no sensible results for continuous features, another distance measure had to be added. The class *EuclideanDistance* was created in the a module of the same name in the *dissimilarity_component* subpackage. It inherits from the *DissimilarityCalculator* class and thus implements the „calculate_dissimilarity“ method and „calculate_dissimilarity_networkwide” method. It calculates, as the name suggests, the euclidean distance of the feature vectors of two agents. Again, the respective factory method, the “select_calculator” method, which follows the original factory pattern and returns an instance of that calculator, had to be extended to include the new measure and again the documentation of the *Simulation* and *Experiment* class needed to be adapted.

Last but not least, the new influence function itself had to be implemented. For this the new class “BoundedConfidence” in the module of the same name in the *influence_sim* subpackage inherits from the *InfluenceOperator* class and is called by the “spread_influence” factory method in the *influence_sim* module. As with every other component, the documentation of the two interface classes *Experiment* and *Simulation* needed to be changed, so that the user can see what options for the influence_function parameter are available.

It can be seen that the addition of a new realization of one of the components calls for a change in the existing code in only one place, the respective factory method, and a change in documentation in the interface classes.

The framework is also extensible in other ways. It could be decided to add another output measure to the "OutputMeasures" module in the tools package. This module is used in the "create_output_table" method in the "Simulation" class, where the new output measure would need to be included. Furthermore, new stop conditions could be added that also take components with noise into account. Certain kinds of noise are a problem for the existing stop conditions because noise makes change always possible and the simulation never "converges" completely. If such a stop condition has to be added to the package, it would be done by creating a new private method in the *Simulation* class. This method initializes the simulation and calls the "run_simulation_step" methods iteratively until it converges. The option to call that method has to be added to the "run_simulation" method and the documentation has to reflect that change.

5 Scalability

5.1 Minimal Execution Time and Scaling in Network Size

The time it takes a simulation run to converge is always dependent on the type of influence function, the complexity of the feature vector of the agents, the number of agents, and the communication regime that is simulated. Measuring how much time a full simulation takes is therefore not a meaningful criterion to evaluate the execution time of the algorithm.

To assess how much time the overhead of the package infrastructure adds to the simulation process, I implemented an "empty" influence function that represents the most minimal influence function possible, accessing the feature vectors of the agents that are influenced, and rewriting one value. It is then measured how much time it takes to run 100,000 simulation steps with a random *FocalAgentSelector* and a random *NeighborSelector*.

In order to also check how the execution time scales with network size, this experiment was tested with a small network of 49 agents and a large network of 100,000 agents, each agent connected to 4 neighbors. The initialization of the network, such as the setting of the attributes, was done beforehand and is not included in the time measurement. The 100,000 time steps were run 100 times and the shortest run time was recorded.

Table 1 shows that the network size has, as expected, almost no effect on the execution time of the simulation steps. The reason is that each

Test condition	Minimum time (seconds)
one-to-one, 49 Agents	1.21
one-to-many, 49 Agents	1.55
one-to-one, 100,000 Agents	1.21
one-to-many, 100,000 Agents	1.59

Table 1: 100*100,000 steps with an 'empty' influence function

time step, only a subset of the whole network is selected for the influence process. The execution time will however scale with the number of network neighbors per agent if the communication regime is either "one-to-many" or "many-to-one", as the influence function will iterate through the list of neighbors.

If 100,000 simulation steps are run with an actual empty influence function, the minimal execution time is 0.917 seconds, which represents the time it takes to select the agents randomly.

To give a way for researchers to estimate the time it will take their simulations to run, I also measured 100,000 time steps with the "Axelrod" influence function and the "Bounded Confidence" influence function for a small network, testing all communication regimes. It should be noted that the list of attributes was passed to the function as an argument, so that it does not have to be called from the graph object in each time step, which would increase the computation time significantly. The results can be seen in tables 2 and 3.

Test condition	Minimum time (seconds)
one-to-one, 49 Agents	2.09
one-to-many, 49 Agents	4.70
many-to-one, 49 Agents	19.16

Table 2: 100*100,000 steps with the "Axelrod" influence function

Test condition	Minimum time (seconds)
one-to-one, 49 Agents	1.5
one-to-many, 49 Agents	1.78
many-to-one, 49 Agents	1.83

Table 3: 100*100,000 steps with the "Bounded Confidence" influence function

5.1.1 Memory

Next to the limits in computation time, scaling experiments up is also heavily limited by memory constraints. With the help of the "pympler" Python package it is possible to monitor memory allocation of complex python objects and detect memory leaks in a running program. The package was used to determine how memory requirements scale with network size, network density, i.e. the number of edges in the graph, and number of attributes including node attributes and edge attributes.

The first goal was to determine the amount of memory that is needed to store one single node. With the *networkx* package it is possible to create empty graphs that contain only node without any edges between them, so I created between 2000 graphs with 1 to 2000 nodes and measured the size of the graph object. Figure 9 shows how this size increases. At one single node, the graph object starts with a size of 2048 bytes and each further node that is added increases this size by 512 bytes, except for a pattern of exponentially increasing intervals, where adding another agent adds also an exponentially increasing amount of memory size to the object. Figure 10 shows this relationship more clearly. The mechanism behind this are internal memory re-allocations of the array that contains the pointers to the node objects inside the graph object. It seems like the nodes are iteratively added when such a graph is constructed and thus the array has to be increased in size every time it gets filled.

This does not pose a serious problem to the up-scaling of the simulations, as these re-allocations only happen very rarely at larger network sizes so that overall the memory requirements of the graph object scale linearly with the number of nodes.

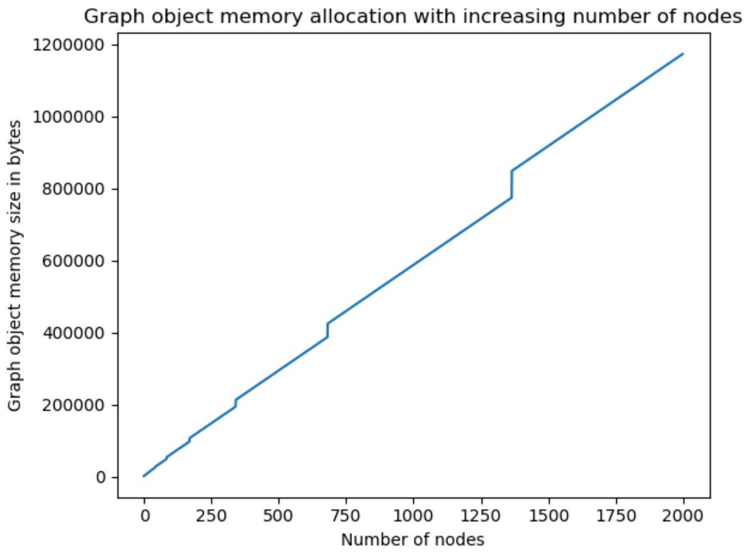


Figure 9: Size of the graph object in bytes in relation to number of nodes

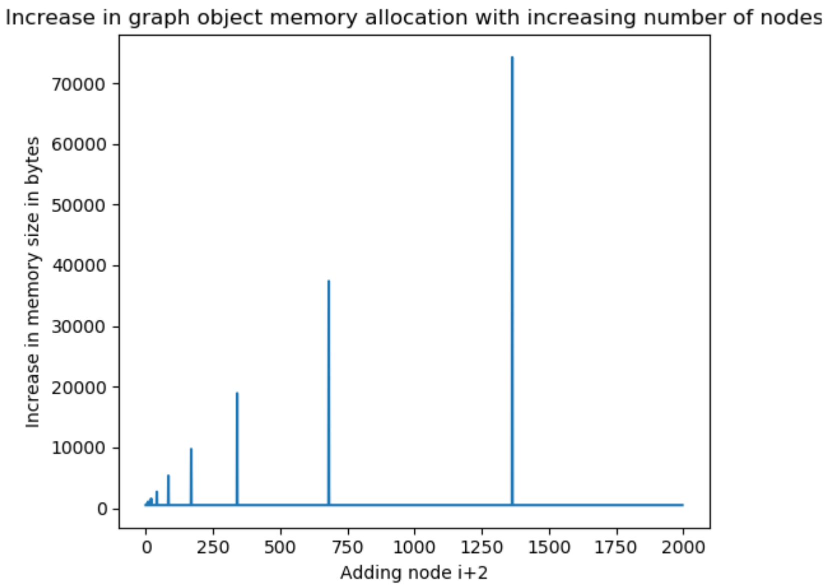


Figure 10: Increase in memory size (in bytes) when adding node number i+2

Similar experiments were executed to gather information about the object size in memory of one edge, one node attribute and one edge attribute. I created a graphs with 10,000 nodes and added 100 times 10,000 edges to the graph between two random nodes. Fig. 11 shows that the memory size of the graph object also increases fairly linear with more edges, but the detailed look at the incremental difference of each edge shows a more complicated pattern, which can be seen in fig 12.

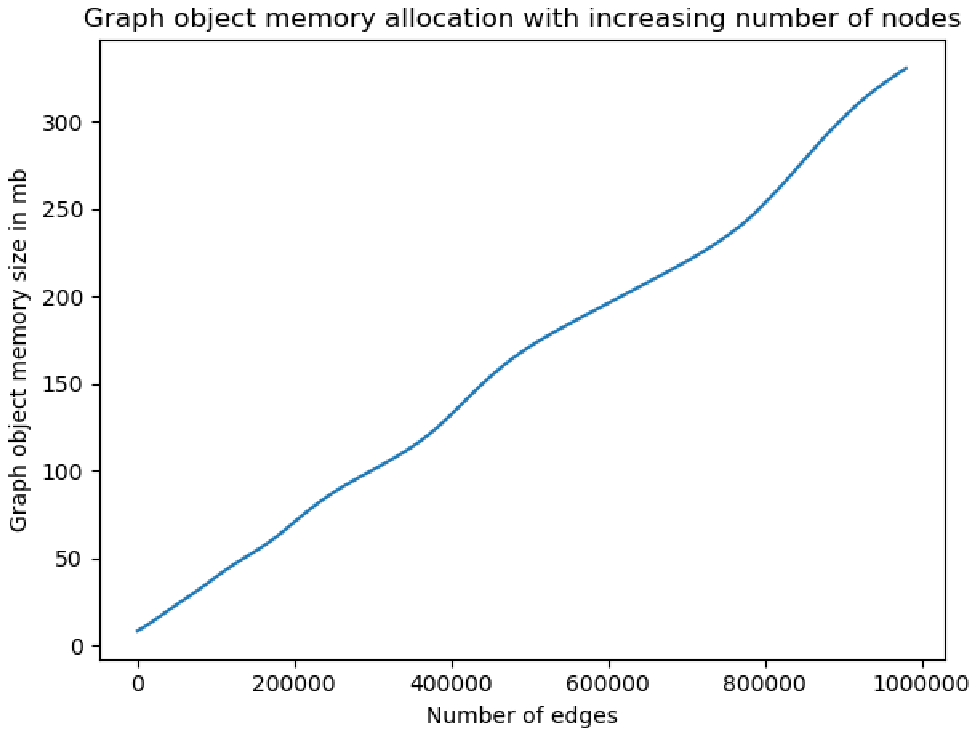


Figure 11: Size of the graph object in megabytes in relation to number of edges.

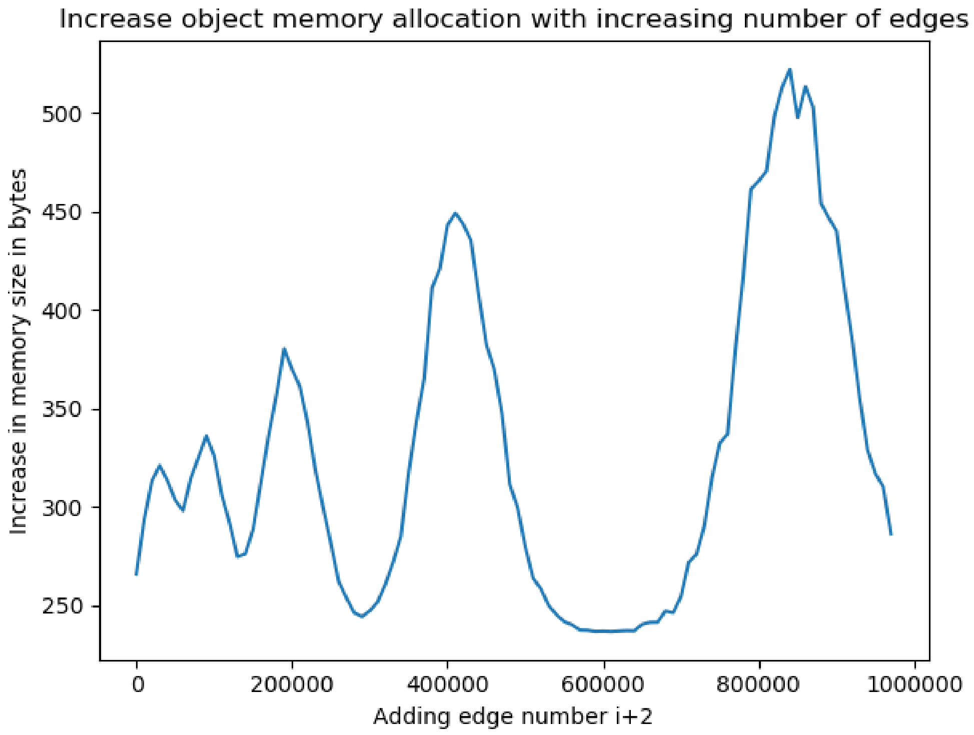


Figure 12: Increase in memory size (in bytes) when adding edge number i+2

Another experiment has indicated seem to be similar memory re-allocations for node attributes, they most definitely do not matter much because the agents in the simulations will not have that many attributes that this will weight in any way.

Edge attribute size scaled completely linearly so that the memory requirements of a network can be estimated with the following formula:

$$\begin{aligned} object\ size\ (bytes) &= 512 \times number\ of\ nodes \\ &+24 \times number\ of\ nodes \times number\ of\ attributes \\ &+300 \times number\ of\ edges \\ &+80 \times number\ of\ edges \times number\ of\ edge\ attributes \end{aligned} \tag{1}$$

Where 24 is the number of bytes that get allocated for each node attribute if it is a continuous attribute and thus represented by a float variable. If the attributes are nominal, this number would need to be replaced by the size of the string objects, if it is chosen to represent the attributes as strings. The number of bytes per edge, in the formula given as 300, would be dependent on the number of edges.

To see how much memory is allocated for the graph objects in practice I created 4 different conditions, a small sparse graph, a small dense graph, and the same for a large graph. Their sizes where then measured right after creation, after initializing 5 node attributes and then after calculating the distances, thus setting one edge attribute.

Test condition	Size in MB	5 attributes	edge attribute
100 nodes, 200 edges	0.11	0.122	0.127
100 nodes, 4950 edges	1.67	1.708	1.827
100,000 nodes, ca. 200,000 edges	109.5	121.5	126.32
100,000 nodes, ca 5,000,000 edges	1853.9	1865.9	1986.0

Table 4: Object sizes in memory for different conditions.

Most of the operations during one simulations modify one and the same graph object. Only in few functions that object is copied. Consequently, one needs approximately two times the estimated graph object size as availabe working memory on the machines that are supposed to run the simulations, to not run out of memory.

5.2 Speed-up in Comparison to Existing Solution

Before demonstrating the advantage of the cluster mode of the experiment I will first point out the importance of multiprocessing on a single machine. To show the speed-up of calling the Experiment's "run" method with the "parallel" parameter set to True, I created a simplified version of the classical Axelrod experiment that only tests 54 different parameter combinations and distributed its execution over a varying number of cores. Each core number setting was tested 10 times and the shortest time was recorded.

In figure 13 one can see the time it took to run the experiment on different number of cores, forming the curve shape that is typical for parallel computing experiments. On a single core the experiment takes 158.25 seconds but using a second processor already reduces the time to 92.18 seconds. The marginal improvement in time decreases and reaches its limit at the full 24 cores that the nodes on the Peregrine cluster offer, when the full experiment only takes 20.15 seconds, showing a speed-up of factor 7.86.

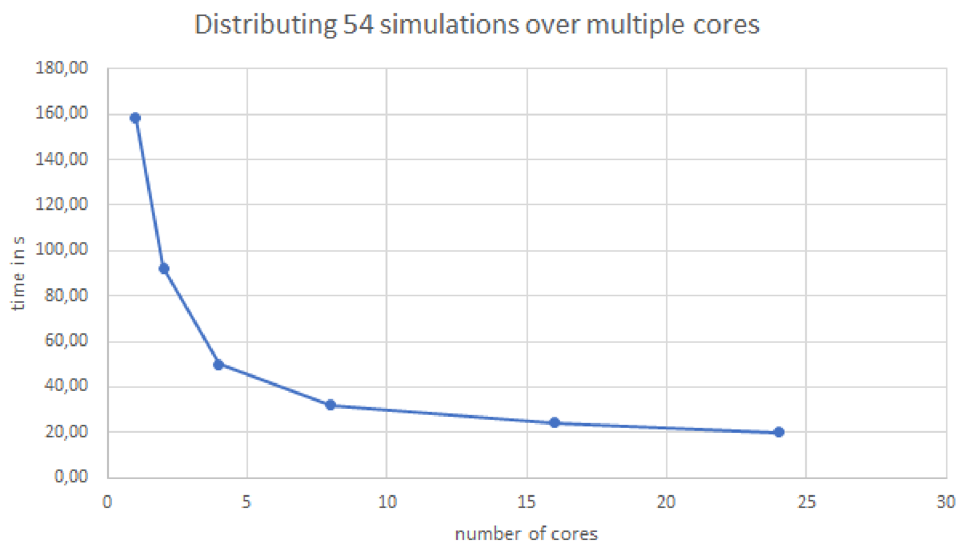


Figure 13: Minimal execution time of 54 simulations with multiple cores

Even though measuring the execution time of full simulation might not be a very meaningful measure in absolute terms, we can still inspect whether execution time improves to already existing implementations of the same simulation. For this purpose, the time it took to run the experiment described in section 3.2 was compared to the time it took to run the same experiment with the original code that was used to produce the results published in [13].

Both experiments are distributed among 24 cores of one node, so that the conditions are the same. The experiments run 100 different parameter settings and repeat each of these 100 times, thus running 10,000 simulations.

In figure 14 it can be seen that the same experiment takes with 27:03 min over 25% longer if it is implemented with the MAMF package. This is to be expected to some degree, because the framework adds a certain amount of overhead through the infrastructure it provides, as for example the factory methods that need to select the desired realization of each component each time step.

Nevertheless, the *Experiment* class offers with the "run_on_cluster" method another way to decrease computation time by distributing the simulations not only to multiple cores on one machine but to multiple nodes in a cluster. To see what speed-up comes from this execution mode, the same experiment was distributed once to 5 and once to 10 nodes on the Peregrine cluster, a powerful high-performance cluster located in Groningen. Each node was running its chunk of the simulations on all of its 24 cores. Ignoring the waiting time in the queue of the cluster's job manager, the computation time of the node that ran the longest was taken as the total time of the experiment.

Figure 14 shows that using multiple nodes cuts down the execution time down by a significant margin. Running the 10,000 simulations on 5 machines took between 05:13 min and 05:38 min and cutting the chunk size in half, thus each machine running a chunk of 1000 simulations, took between 02:44 min and 03:02 min. The limit of this speed-up would be set by the time it takes one simulation to converge, since the Experiment could in a maximally optimal way be split up into chunks of 24 simulations, such that each processor computes one simulation. This would obviously only make sense for very long simulations where few parameter combinations are explored.

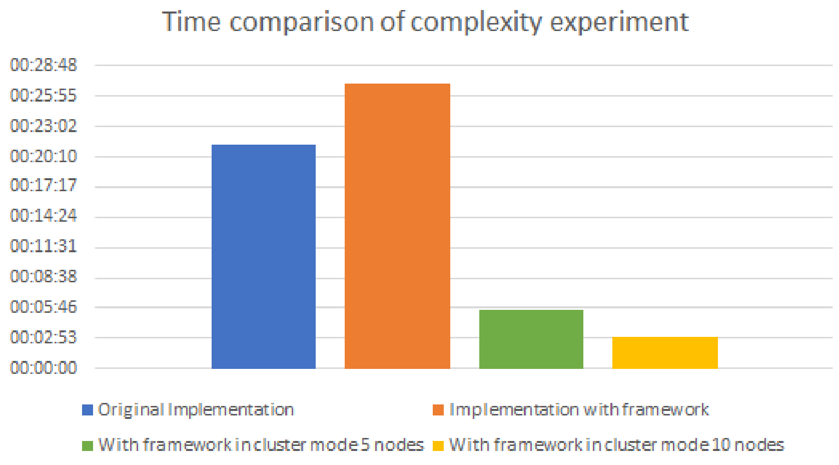


Figure 14: The execution time of 10,000 simulations on multiple cores and multiple nodes

6 Discussion

The goal of this framework was to provide the infrastructure for researchers in the field of opinion dynamics to save time on setting up their experiments as well as running them. The modular structure of the framework, even though it has its limits, allows users to combine many different building blocks to create experiments that explore new theories or validate the behaviour of existing models by testing them under new conditions. The option to pass the *Simulation* or the *Experiment* your own implementations of the components gives researchers the opportunity to tailor the package to their needs while reusing those parts of a simulation that are almost always the same. Thus, a lot of time spend coding can be saved.

Nevertheless, the package comes with some weaknesses in its current form. One major disadvantage is that it currently supports only undirected graphs as many of the pre-implemented components break when used on directed graphs. This is, however, not something inherent to the nature of those components and will be changed in future versions.

Another drawback that is more inherent to the architecture of the framework, is the way how the distance between agents is stored as edge attributes on the edge that connects them. This means that any kind of model that assumes influence between agents that are not directly connected, have to recalculate the distances between the relevant agents in each time step, which takes longer time than accessing the attribute of the edge that connects them.

A third problem comes from the fact that every parameter name that is defined for a specific implementations of a component has to be unique, since they are all stored in the same dictionary in the *Simulation*. This bears potential for conflict in the process of maintaining and extending the framework, as every contributor has to check for that and there might be disagreement on which names to use for what.

Even though the cluster mode of the Experiment class is not implemented in a true distributed programming approach, because the nodes do not communicate with each other, one advantage comes from this "embarrassing" parallelization. The user can run their experiment with a simple python script and does not have to deal with setting up the batch files that are necessary to submit jobs to the cluster. This makes the use of super computer clusters that use SLURM more approachable for researchers in this field an might allow some to run larger experiments they would otherwise not have tried.

7 Conclusion

In this report I presented the new python package "MAMF" that offers researchers in the field of opinion dynamics a framework with a library of classes and functions that help to create and run computational simulations of social influence. The framework consists of multiple building blocks or components, that can either be used directly, or plugged together with accessible interface classes. The users are not limited to the component implementations that are included in the package, but can implement their own version of each building block and combine it with the other components to run their simulations. The interface classes make use of multiprocessing and distributed computing to minimize the time it takes to execute multiple simulations.

References

[1] Swarm. `http://www.swarm.org/wiki/Swarm_main_page`. Accessed: 2019-04-09.

[2] R. Axelrod. The Dissemination of Culture: A Model with Local Convergence and Global Polarization. *J. Conflict Resolut.*, 41(2):203–226, 1997.

[3] G. Deffuant, D. Neau, F. Amblard, and G. Weisbuch. Mixing beliefs among interacting agents. *Advances in Complex Systems*, 3:87–98, 01 2000.

[4] J. M. Epstein and R. Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. The Brookings Institution, Washington, DC, USA, 1996.

[5] A. Flache and M. W. Macy. Local convergence and global diversity: From interpersonal to social influence. *Journal of Conflict Resolution*, 55(6):970–995, 2011.

[6] A. Flache and M. Mäs. *Multi-Agenten-Modelle*, pages "491–514". Springer Fachmedien Wiesbaden, Wiesbaden, 2015.

[7] A. Flache, M. Mäs, T. Feliciani, E. Chattoe, G. Deffuant, S. Huet, and J. Lorenz. Models of social influence: Towards the next frontiers. *Journal of Artificial Societies and Social Simulation*, 20, 10 2017.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] R. Hegselmann and U. Krause. Opinion dynamics and bounded confidence: Models, analysis and simulation. *Journal of Artificial Societies and Social Simulation*, 5:1–24, 2002.

[10] M. Ivanyi, L. Gulyás, R. Legendi, S. Bartha, V. Kozma, and R. Meszaros. The multi-agent simulation suite. 09 2007.

[11] W. Jager and F. Amblard. "uniformity, bipolarization and pluriformity captured as generic stylized behavior with an agent-based simulation model of attitude change". *"Computational & Mathematical Organization Theory"*, pages "295–303", "2005".

[12] M. A. Janssen. The practice of archiving model code of agent-based models. *Journal of Artificial Societies and Social Simulation*, 20(1):2, 2017.

[13] M. A. Keijzer, M. Mäs, and A. Flache. Communication in online social networks fosters cultural isolation. *Complexity*, 2018, 2018.

[14] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[15] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[16] M. Parker. Ascape. `http://ascape.sourceforge.net/index.html#Introduction`. Accessed: 2019-04-09.

[17] M. Ye, Y. Qin, A. Govaert, B. D. O. Anderson, and M. Cao. An influence network model to study discrepancies in expressed and private opinions. *CoRR*, abs/1806.11236, 2018.

8 Appendix

The appendix contains the documentation of the package in pdf form. However, I recommend to read the documentation online on `https://antonlaukemper.github.io/mamf/` instead for better readability.

MAMF

Release 12.07.2019

Anton Laukemper

Jul 10, 2019

CONTENTS

Here you can find the API documentation for the world's first package on social influence simulations. I recommend to start with the `Simulation` class and the `Experiment` class to understand how to create your own multi-agent system experiment.

Here are some examples of how to create experiments:

```
# creating an Experiment with the default values, which recreates the classic axelrod␣
↪experiment conditions
# in this example we want to try all kinds of communication regimes 1000 times

# Alternative way to import the experiment:
# from mamf import Experiment
# experiment = Experiment()

import mamf

import networkx as nx
from typing import List
from mamf import dissimilarity_calculator

experiment = mamf.Experiment(communication_regime=["one-to-one", "one-to-many", "many-
↪to-one"], repetitions=1000)
results = experiment.run()
print(results)

#creating a Simulation
simulation = mamf.Simulation()
results = simulation.run_simulation()
print(results)

# creating a Simulation with your own influence function and running it step by step
class my_influence_function(mamf.InfluenceOperator):

    @staticmethod
    def spread_influence(network: nx.Graph, agent_i: int, agents_j: List[int] or int,␣
↪regime: str,
                         dissimilarity_measure: dissimilarity_calculator, attributes:␣
↪List[str]=None, **kwargs) -> bool:
        # print("enter your implementation here")
        pass

simulation = mamf.Simulation(influence_function=my_influence_function())
simulation.initialize_simulation()
for i in range(100):
    simulation.run_simulation_step()
results = simulation.create_output_table()

# using the building blocks manually with the respective factory methods
# only the influence function is here used directly (just as an example to show how␣
↪to import it, you could also use:
# mamf.InfluenceOperator.spread_influence(network, "axelrod", focal_agent,neighbors,
↪"one-to-one", mamf.HammingDistance())
# instead of using the factory method you can alway import the classes directly from␣
↪the respective module

from mamf.influence_sim.Axelrod import Axelrod # import the axelrod infuence function

network = mamf.generate_network("ring")
```

41

(continues on next page)

(continued from previous page)

```
mamf.agents_init.initialize_attributes(network, realization="random_categorical")
calculator = mamf.dissimilarity_calculator.select_calculator("euclidean")
calculator.calculate_dissimilarity_networkwide(network)

for i in range(100):
    focal_agent = mamf.focal_agent_sim.select_focal_agent(network, "random")
    neighbors = mamf.neighbor_selector_sim.select_neighbors(network, "random", focal_
↪agent, "one-to-one")
    Axelrod.spread_influence(network, focal_agent, neighbors, regime="one-to-one",␣
↪dissimilarity_measure=mamf.HammingDistance())

results = mamf.OutputMeasures.clustercount(network)
print(results)

# using the building blocks manually with your own influence function
network = mamf.generate_network("ring")
mamf.agents_init.initialize_attributes(network, realization="random_categorical")
calculator = mamf.dissimilarity_calculator.select_calculator("euclidean")
calculator.calculate_dissimilarity_networkwide(network)

for i in range(100):
    focal_agent = mamf.focal_agent_sim.select_focal_agent(network, "random")
    neighbors = mamf.neighbor_selector_sim.select_neighbors(network, "random", focal_
↪agent, "one-to-one")
    my_influence_function.spread_influence(network, focal_agent, neighbors, regime=
↪"one-to-one",
                                           dissimilarity_measure=mamf.
↪HammingDistance())

results = mamf.OutputMeasures.clustercount(network)
print(results)
```

42

CHAPTER

ONE

HERE IS A LIST OF ALL COMPONENTS OF THIS FRAMEWORK:

1.1 mamf

1.1.1 Experiment module

class `mamf.Experiment.`**`Experiment`**(*network: networkx.classes.graph.Graph = None*, *communication_regime: List[T] = 'one-to-one'*, *topology: str = 'grid'*, *network_parameters: dict = {}*, *attributes_initializer: str = 'random_categorical'*, *attribute_parameters: dict = {}*, *focal_agent_selector: str = 'random'*, *focal_agent_parameters: dict = {}*, *neighbor_selector: str = 'random'*, *neighbor_parameters: dict = {}*, *influence_function: str = 'axelrod'*, *influence_parameters: dict = {}*, *influenceable_attributes: list = None*, *network_modifier: str = 'random'*, *network_modifier_parameters: dict = {}*, *dissimilarity_measure: str = 'hamming'*, *stop_condition: str = 'max_iteration'*, *stop_condition_parameters: dict = {}*, *max_iterations: int = 100000*, *repetitions: int = 1*)

Bases: `object`

The main class for creating and running experiments. Each simulation consists of 7 modular components, where each component is independent from the others and can thus be replaced by a different implementation of that component. These components are:

The network structure - Can either be loaded from empirical data or initialized with the NetworkGenerator.

The AttributesInitializer - This component initializes the attributes of each agent in the network and can also be used to add attributes during the simulation.

The FocalAgentSelector - Each simulation step, this component picks an agent from the network, either randomly or based on their characteristics.

The neighborhoodSelector - Selects a subset of the agents in the network based on the focal agent given by the FocalAgentSelector

The InfluenceFunction - Determines how the selected agent and the selected neighborhood influence each other.

The NetworkModifier - Changes the structure of the network during the simulation.

The DissimilarityCalculator - Determines how the dissimilarity/distance between two agents is calculated

These components have different concrete implementations that might take specific parameters that are passed as a dictionary. In these dictionaries, the keys are the names of the parameters and the values their respective value. It is also possible to pass a list of values as the dictionary value, which then creates a simulation for each value, making it possible to easily compare simulation runs with e.g. different number of agents.

Parameters

- **network** (*nx.Graph, np.array or String*) – This is either a preloaded networkx graph, an adjacency matrix as a numpy array, or the full path to a file with and edge list.
- **communication_regime** (*List or String = "one-to-one"*) – Options are "one-to-one", "one-to-many" and "many-to-one". For this parameter, it is possible to pass a list of multiple of these options.
- **topology** (*String = "grid"*) – Options are "grid", "ring" and "spatial_random_graph".
- **network_parameters** (*dict = {}*) – This dictionary should contain all optional parameters for creating the network structure. Refer to the specific documentation of the network types to see what can be modified.
- **attributes_initializer** (String = "random_categorical" or `AttributesInitializer`) – Either be a custom AttributesInitializer or a string that selects from the predefined choices: ["random_categorical", "random_continuous"...]
- **attribute_parameters** (*dict = {}*) – Optional dictionary that includes the name of attributes you want to set and a list of possible values for each.
- **focal_agent_selector** (str = "random" or `FocalAgentSelector`) – Either a custom FocalAgentSelector or a string that selects from the predefined options ["random", ...]
- **focal_agent_parameters** (*dict = {}*) – Optional dictionary that includes the parameters for the FocalAgentSelector.
- **neighbor_selector** (str = "random" or `NeighborSelector`) – Either a custom NeighborSelector or a string that selects from the predefined options ["random", ...}
- **neighbor_parameters** (*dict = {}*) – Optional dictionary that includes the parameters for the NeighborSelector.
- **influence_function** (str = "axelrod" or `InfluenceOperator`) – Either a custom influence function or a string that selects from the predefined options ["axelrod", "bounded_confidence", ...}
- **influence_parameters** (*dict = {}*) – Optional dictionary that includes the parameters for the InfluenceFunction.
- **influenceable_attributes** (*list = []*) – With this list you select all attributes that are allowed to be changed by the influence function. If the list is empty, all attributes are affected by influence.
- **network_modifier** – (String = "random" or `NetworkModifier`) Either a custom NetworkModifier or a string selecting from the predefined options ["random", ...]
- **dissimilarity_measure** (String = "hamming" or `DissimilarityCalculator`) – Either a custom DissimilarityCalculator or a string that selects from the predefined options ["hamming", "euclidean", ...}
- **stop_condition** (*String = "pragmatic_convergence"*) – Determines at what point a simulation is supposed to stop. Options include "strict_convergence", which means that it is theoretically not possible anymore for any agent to influence another, "pragmatic_convergence", which means that it is assumed that little change is possible anymore, and "max_iteration" which just stops the simulation after a certain amount of time steps.

- **`stop_condition_parameters`** (*`dict = {}`*) – This dictionary should contain all optional parameters that influence how convergence is determined.
- **`max_iterations`** (*`int = 100000`*) – The maximum number of iterations a Simulation should run.
- **`repetitions`** (*`int = 1`*) – How often each simulation should be repeated.

`estimate_runtime`()

This function creates the parameterDictList if that hasn't happened already and then infers from its length the runtime of the whole experiment.

Returns estimated time of simulation in seconds

`run`(*parallel: bool = False*, *num_cores=8*) → pandas.core.frame.DataFrame

Starts the experiment by first creating the parameter_dict_list if that hasn't happened already and then creates a simulation for each parameter combination in the parameter_dict_list. If parallel is true, the simulations are run on multiple cores on the machine, their number determined by num_cores.

Parameters

- **`parallel`** – Boolean that determines in which mode the simulations will run.
- **`num_cores`** – Determines the number of cores in the machine that will be utilized for the execution.

Returns A dataframe that contains one row per Simulation.

`run_on_cluster`(*chunk_size: int = 2400*, *batch_path: str = 'batchscripts'*, *output_path: str = 'output'*, *walltime: str = '30:00'*, *partition: str = 'short'*)

This method can be used to execute large Experiments on a SLURM cluster. It creates a number of sbatch scripts that are immediatly executed to send jobs to the SLURM job manager. To be able to use this method the script with the Experiment must be executed on a SLURM server.

Parameters

- **`chunk_size`** – Determines how many Simulations should run per node in the cluster.
- **`batch_path`** – The path to the folder where the batchscripts will be created. If the folder not exists yet, it will be created.
- **`output_path`** – The path to the folder where the output will be saved. If the folder not exists yet, it will be created.
- **`walltime`** – The expected time one node maximally needs for computing its chunk of simulations.
- **`partition`** – If the SLURM cluster has multiple partitions, it can be decided where to run the jobs with this parameter.

1.1.2 Simulation module

class `mamf.Simulation.`**`Simulation`**(*network=None*, *topology: str = 'grid'*, *attributes_initializer: str = 'random_categorical'*, *focal_agent_selector: str = 'random'*, *neighbor_selector: str = 'random'*, *influence_function: str = 'axelrod'*, *influenceable_attributes: List[T] = None*, *dissimilarity_measure: str = 'hamming'*, *network_modifier: str = 'random'*, *stop_condition: str = 'max_iteration'*, *max_iterations: int = 100000*, *communication_regime: str = 'one-to-one'*, *parameter_dict={}*, *seed=None*)

45

Bases: `object`

This class is responsible for initializing and running a single experiment until the desired stop criterion is reached. The Simulation class contains three different stop criterion implementations as methods, but more can be added. The class is initialized in a similar way as the Experiment class but it does not accept multiple parameter values per parameter and also all optional parameters are passed in one combined dictionary.

Parameters

- **`network`** (*`nx.Graph=None`*) – A Graph object that was created from empirical data.
- **`topology`** (*`String = "grid"`*) – Options are "grid", "ring" and "spatial_random_graph".
- **`attributes_initializer`** (String = "random_categorical" or `AttributesInitializer`) – Either be a custom AttributesInitializer or a string that selects from the predefined choices: ["random_categorical", "random_continuous"...]
- **`focal_agent_selector`** (str = "random" or `FocalAgentSelector`) – Either a custom FocalAgentSelector or a string that selects from the predefined options ["random", ...]
- **`neighbor_selector`** (str = "random" or `NeighborSelector`) – Either a custom NeighborSelector or a string that selects from the predefined options ["random", ...}
- **`influence_function`** (str = "axelrod" or `InfluenceOperator`) – Either a custom influence function or a string that selects from the predefined options ["axelrod", "bounded_confidence", ...}
- **`influenceable_attributes`** (*`List = None`*) – This is a list of the attribute names, that may be changed in the influence step
- **`dissimilarity_measure`** (String = "hamming" or `DissimilarityCalculator`) – Either a custom DissimilarityCalculator or a string that selects from the predefined options ["hamming", "euclidean", ...}
- **`network_modifier`** – (String = "random" or `NetworkModifier`) Either a custom NetworkModifier or a string selecting from the predefined options ["random", ...]
- **`stop_condition`** (*`str = "max_iteration"`*) – Determines at what point a simulation is supposed to stop. Options include "strict_convergence", which means that it is theoretically not possible anymore for any agent to influence another, "pragmatic_convergence", which means that it is assumed that little change is possible anymore, and "max_iteration" which just stops the simulation after a certain amount of time steps.
- **`communication_regime`** (*`str = "one-to-one"`*) – Options are "one-to-one", "one-to-many" and "many-to-one".
- **`parameter_dict`** – A dictionary with all parameters that will be passed to the specific component implementations.

`run_simulation``()` → pandas.core.frame.DataFrame

This method initializes the network if none is given, initializes the attributes of the agents, and also computes and sets the distances between each neighbor. It then calls different functions that execute the simulation based on which stop criterion was selected.

Returns A pandas Dataframe that contains one row of data. To see what output the output contains see *`create_output_table()`*

`initialize_simulation``()`

This method initializes the network if none is given, initializes the attributes of the agents, and also computes and sets the distances between each neighbor.

46

run_simulation_step()

Executes one iteration of the simulation step which includes the selection of a focal agent, the selection of the neighbors and the influence step. If the user passed their own implementations of those components, they will be called to execute these steps, otherwise the respective factory functions will be called.

create_output_table() → pandas.core.frame.DataFrame

This method measures multiple characteristics of the network in its current state and writes them to a dataframe. It currently contains the following columns:

Seed: The random seed that was used.

Network Topology: Which network topology was used.

Simulation Steps: For how many iterations the simulation ran (so far).

Successful Influences: How often an agent was successfully influenced by another agent.

Number of Clusters: The total number of clusters in the network #todo reference to what a cluster is

Cluster Sizes: A list containing the sizes of each cluster in descending order.

Number of Isolates: How many isolates the network contains. #todo here also want a reference to what an isolate is

Homogeneity: A number between 0 and 1 representing the ratio of the size of the biggest cluster to the number of agents in the network. #todo and again a reference would be nice

Returns A pandas Dataframe with one row.

_run_until_pragmatic_convergence()

Pragmatic convergence means that each "step_size" time steps it is checked whether the structure of the network and all attributes are still the same. If thats the case, it is assumed that the simulation converged and it stops.

Parameters **step_size** (*int=100*) – determines how often it should be checked for a change in the network.

_run_until_strict_convergence()

Here the convergence of the simulation is periodically checked by assessing the distance between each neighbor in the network. Unless there is no single pair left that can theoretically influence each other, the simulation continues.

Parameters

- **threshold** (*float=0.0*) – A value between 0 and 1 that determines at what distance two agents can't influence each other anymore.
- **check_each_step** (*boolean=True*) – A boolean that determines whether convergence should be checked each step or only every hundreth step to save time.

_run_until_max_iteration()

1.2 network_init

1.2.1 network_init module

mamf.network_init.network_init.**generate_network**(*name: str*, ***kwargs*) → networkx.classes.graph.Graph

This is the factory method that returns a specific network.

Parameters

- **`name`** – A string with the name of the network type. Possible options: “spatial_random_graph”, “ring”, “grid”
- **`kwargs`** – A dictionary containing the parameter names as keys and their respective values as values to be passed to the function that produces the network.

Raises ValueError if not one of the possible network topologies is selected.

Returns A networkx Graph object.

`mamf.network_init.network_init.`**`read_network`**(*network_input: numpy.ndarray*)

This function takes the structure of an empirically measured network, provided as an adjacency matrix, and creates a network graph object out of it.

Parameters **`network_input`** – Either an adjacency matrix where the entry (i,j) represents whether an edge between i and j exists or if applicable, its strength, or the path to a file containing an edge list

`mamf.network_init.network_init.`**`_produce_grid_network`**(***kwargs*) → networkx.classes.graph.Graph

This method produces a grid graph, connected to itself as a torus. Each agent on the grid is connected to its neighborhood depending on the parameters “neighborhood” and “radius”.

Parameters

- **`num_agents`** (*`int=49`*) – How many agents the network contains.
- **`neighborhood`** (*`String=moore`*) – Either “von_neumann” or “moore”. Von Neumann connects each agent to its 4 neighbors in each cardinal direction. In a Moore neighborhood, each agent is connected to their 8 immediate neighbors.
- **`radius`** (*`int=1`*) – Increases the number of connections further than the immediate neigbourhood. An agent in a Moore neighborhood with radius 2 has 24 connections.

Returns A networkx Graph object.

`mamf.network_init.network_init.`**`_produce_ring_network`**(***kwargs*) → networkx.classes.graph.Graph

This method produces a ring network with varying number of neighbors. Furthermore it is possible to rewire the network following the Maslov-Sneppen algorithm todo: put in reference . to vary the transitivity of the network.

Parameters

- **`num_agents`** (*`int=49`*) – How many agents the network contains.
- **`num_neighbors`** (*`int=2`*) – The number of neighbors of each agent must be an even number.
- **`ms_rewiring`** (*`float=0`*) – 1 means that num_agents agents are drawn from the network to be rewired. If one wants to be sure that most agents are rewired, this number should be higher than 1.

Returns A networkx Graph object. #todo: is this a correct description?

`mamf.network_init.network_init.`**`_produce_spatial_random_graph`**(***kwargs*) → networkx.classes.graph.Graph

This method produces a spatial random graph constructed by rewiring a grid network. Spatial random graphs resemble real social networks to some degree as the have low tie density, short average geodesic distance, a high level of transitivity, a positively skewed actor-degree distribution, and a community structure. #todo: fill in reference here

Parameters

48

- **`num_agents`** (*`int=49`*) – How many agents the network contains.

- **`min_neighbors`** (*int=8*) – How many neighborhood each agent should have at least.
- **`proximity_weight`** (*float=1*) – Determines how much spatial distance in the grid matters in the rewiring process.

Returns A networkx Graph object.

1.3 agents_init

1.3.1 agents_init module

class `mamf.agents_init.agents_init.`**`AttributesInitializer`**

Bases: `abc.ABC`

Initializes and changes attributes of nodes in the network.

static **`initialize_attributes`**(*network: networkx.classes.graph.Graph*, ***kwargs*)

Gives initial values to the nodes in the network. Values could e.g. be based on their position in the network.

Parameters

- **`network`** – The network that will be modified.
- **`kwargs`** – This dictionary contains all the implementation-specific parameters.

`mamf.agents_init.agents_init.`**`set_categorical_attribute`**(*network: networkx.classes.graph.Graph*, *name: str*, *values: list*, *distribution: str = 'uniform'*, ***kwargs*)

Adds a categorical attribute to all nodes in a network. The values for that attribute are drawn from a list of possible values provided by the user.

Parameters

- **`network`** – The graph object whose nodes' attributes are modified.
- **`name`** – the name of the attribute. This is used as a key to call the attribute value in other functions.
- **`values`** – A list that contains all possible values for that attribute.
- **`distribution`** – 'gaussian', 'uniform', or 'custom' are possible values.
- **`kwargs`** – a dictionary containing the parameter name and value for each distribution, these are:

  for gaussian: loc and scale. loc would be the index of the most common value in the values list

  for custom distribution: c. an array-like containing the probabilities for each entry in the values list.

`mamf.agents_init.agents_init.`**`set_continuous_attribute`**(*network: networkx.classes.graph.Graph*, *name: str*, *shape: tuple = 1*, *distribution: str = 'uniform'*, ***kwargs*)

adds a possibly multidimensional attribute to all nodes in a network. The values of the attribute are drawn from a distribution that is set by the user.

Parameters

- **network** – The graph object whose nodes' attributes are modified.
- **shape** – sets the output shape of the attribute value. Allows e.g. for multidimensional opinion vectors
- **name** – the name of the attribute. This is used as a key to call the attribute value in other functions
- **distribution** – "gaussian", "exponential", "beta" are possible distributions to choose from
- **kwargs** – a dictionary containing the parameter name and value for each distribution, these are:

  loc, scale for gaussian

  scale for exponential

  a, b for the beta distribution

`mamf.agents_init.agents_init.`**`initialize_attributes`**(*network: networkx.classes.graph.Graph*, *realization: str*, ***kwargs*)

This function works as a factory method for the AttributesInitializer component. It calls the initialize_attributes function of a specific implementation of the AttributesInitializer and passes to it the kwargs dictionary.

Parameters

- **network** – The network that will be modified.
- **realization** – The specific AttributesInitializer that shall be used to initialize the attributes. Options are "random", ..
- **kwargs** – The parameter dictionary with all optional parameters.

1.3.2 RandomCategoricalInitializer module

class `mamf.agents_init.RandomCategoricalInitializer.`**`RandomCategoricalInitializer`**

Bases: *mamf.agents_init.agents_init.AttributesInitializer*

Implements the AttributesInitializer as a random initialization of arbitrary discrete features.

static initialize_attributes(*network: networkx.classes.graph.Graph*, ***kwargs*)

Randomly initializes a number of discrete features for each node.

Parameters

- **network** – The graph object whose nodes' attributes are modified.
- **num_features** (*int=5*) – How many different attributes each node has.
- **num_traits** (*int=3*) – The range of values each attribute can take. 3 means that an attribute can be either 0, 1 or 2

1.3.3 RandomContinuousInitializer module

class `mamf.agents_init.RandomContinuousInitializer.`**`RandomContinuousInitializer`**

Bases: *mamf.agents_init.agents_init.AttributesInitializer*

Implements the AttributesInitializer as a random initialization of arbitrary continuous features.

static initialize_attributes(*network: networkx.classes.graph.Graph*, ***kwargs*)
Randomly initializes a number of continuous features between 0 and 1 for each node.

Parameters

- **network** – The graph object whose nodes' attributes are modified.
- **num_features** (*int=1*) – How many different attributes each node has.

1.4 focal_agent_sim

1.4.1 focal_agent_sim module

class `mamf.focal_agent_sim.focal_agent_sim.`**FocalAgentSelector**
Bases: `abc.ABC`

This class is responsible for sampling the focal agent for the influence process.

static select_agent(*network: networkx.classes.graph.Graph*, *agents: List[int] = []*, ***kwargs*) → int
This method selects an agent from a network for the influence process. Based on the communication regime, the selected agent is either the source or the target of influence.

Parameters

- **network** – the network from which the agent shall be selected.
- **agents** – A list of the indices of all agents in the network

:returns The index of the selected agent.

`mamf.focal_agent_sim.focal_agent_sim.`**select_focal_agent**(*network: networkx.classes.graph.Graph*, *realization: str*, *agents: List[int] = []*, ***kwargs*) → int
This function works as a factory method for the FocalAgentSelector component. It calls the select_agent function of a specific implementation of the FocalAgentSelector and passes to it the kwargs dictionary.

Parameters

- **network** – The network from which the focal agent will be selected
- **realization** – The specific FocalAgentSelector that shall be used to sample the focal agent. Options are "random", . . .
- **agents** – A list of the indices of all agents in the network.

:returns The index of the focal agent in the network.

1.4.2 RandomSelector module

class `mamf.focal_agent_sim.RandomSelector.`**RandomSelector**
Bases: *mamf.focal_agent_sim.focal_agent_sim.FocalAgentSelector*

Implements the FocalAgentSelector as a uniformly random process

static select_agent(*network: networkx.classes.graph.Graph*, *agents: List[int] = []*, ***kwargs*) → int
51
This method selects a random agent from a network for the influence process.

Parameters

- **network** – The network from which the agent shall be selected.
- **agents** – A list of the indices of all agents in the network.

:returns The index of the selected agent.

1.5 neighbor_selector_sim

1.5.1 neighbor_selector_sim module

class mamf.neighbor_selector_sim.neighbor_selector_sim.**NeighborSelector**

Bases: abc.ABC

The NeighborSelector is responsible for picking certain agents from the environment of the focal agent. These agents are then either the source or the target in the following influence step. Possible ways to select neighbors is either by picking all other agents, i.e. global influence (simulation influence from news and polls), all neighbors in the immediate neighborhood, or only single neighbors.

static select_neighbors(*network: networkx.classes.graph.Graph*, *agentID: int*, *regime: str*, ***kwargs*) → Iterable[int]

This method selects a subset of agents from the network that are in some way relevant for the influence process regarding the focal agent. This subset could be e.g. a single agent from the direct neighborhood, the whole neighborhood, or all agents, excluding the focal agent.

Parameters

- **network** – the network from which the agent shall be selected.
- **agentID** – the index of the focal agent, who is either the source or target of influence.
- **regime** – Whether the focal agent interacts with only one or many agents from his or her neighborhood. If "one-to-one": One neighbor to which the focal agent has an outgoing tie is selected. If "one-to-many": All neighbors to which the focal agent has an outgoing tie are selected. If "many-to-one": All neighbors from which the focal agent has an incoming tie are selected.
- **kwargs** – Additional parameters specific to the implementation of the neighborSelector.

Returns a list of the indices of the relevant other agents.

mamf.neighbor_selector_sim.neighbor_selector_sim.**select_neighbors**(*network: networkx.classes.graph.Graph*, *realization: str*, *agentID: int*, *regime: str*, ***kwargs*) → Iterable[int]

This function works as a factory method for the neighborSelector component. It calls the select_neighbors function of the specific neighborSelector and passes to it the index of the focal agent and the communication regime.

Parameters

- **network** – The network from which the agents are selected.

- **`realization`** – The specific implementation of the neighborSelector. Options are "random", . . .
- **`agentID`** – An integer that represents the index of the focal agent in the network.
- **`regime`** – Either "many_to_one", "one_to_many" or "one_to_one".
- **`kwargs`** – Additional parameters specific to the implementation of the neighborSelector.

Returns A list with the indices of the selected other agents.

1.5.2 RandomneighborSelector module

class `mamf.neighbor_selector_sim.RandomNeighborSelector.`**`RandomNeighborSelector`**

Bases: *`mamf.neighbor_selector_sim.neighbor_selector_sim.NeighborSelector`*

Implements the neighborSelector in such a way that either all neighbors are selected in the case of one-to-many and many-to-one communication, or a random neighbor in the case of one-to-one communication.

static select_neighbors(*network: networkx.classes.graph.Graph*, *agentID: int*, *regime: str*, ***kwargs*) → Iterable[int]

Selects a random agent from the direct neighborhood of the focal agent in the case of one-to-one communication, and all direct neighbors otherwise.

Parameters

- **`network`** – The network from which the agent shall be selected.
- **`agentID`** – The index of the focal agent, who is either the source or target of influence.
- **`regime`** – Whether the focal agent interacts with only one or many agents from his or her neighborhood. If "one-to-one": One neighbor to which the focal agent has an outgoing tie is selected. If "one-to-many": All neighbors to which the focal agent has an outgoing tie are selected. If "many-to-one": All neighbors from which the focal agent has an incoming tie are selected.
- **`kwargs`** – Additional parameters specific to the implementation of the InfluenceOperator.

Raises ValueError if not one of the possible options for the communication_regime is chosen.

Returns A list of the indices of the relevant other agents.

1.6 influence_sim

1.6.1 influence_sim module

class `mamf.influence_sim.influence_sim.`**`InfluenceOperator`**

Bases: `abc.ABC`

The InfluenceOperator is responsible for executing the influence function of the simulation. The influence function can be something like bounded confidence, negative influence or only positive influence.

static spread_influence(*network: networkx.classes.graph.Graph, agent_i: int, agents_j: List[int], regime: str, dissimilarity_measure: mamf.dissimilarity_component.dissimilarity_calculator.DissimilarityCalculator, attributes: List[str] = None, **kwargs*) → bool

This function is responsible for executing the influence process. The call of this function can be seen as an interaction between agents that either results in successful influence or not. Unsuccessful influence

53

attempts can also be interpreted as no interaction at all. The function returns true if influence was successfully exerted.

Parameters

- **`network`** – The network in which the agents exist.
- **`agent_i`** – the index of the focal agent that is either the source or the target of the influence
- **`agents_j`** – A list of indices of the agents who can be either the source or the targets of the influence. The list can have a single entry, implementing one-to-one communication.
- **`attributes`** – A list of the names of all the attributes that are subject to influence. If an agent has e.g. the attributes "Sex" and "Music taste", only supply ["Music taste"] as a parameter for this function. The influence function itself can still be a function of the "Sex" attribute.
- **`regime`** – This string determines the mode in which the agents influence each other. In 'one-to-one' the focal agent influences one other agent, in 'one-to-many' multiple other agents and in 'many-to-one' the focal agent is influenced by multiple other agents in the network.
- **`dissimilarity_measure`** – An instance of a `DissimilarityCalculator`.

Returns true if agent(s) were successfully influenced

`mamf.influence_sim.influence_sim.`**`spread_influence`**(*network: networkx.classes.graph.Graph, realization: str, agent_i: int, agents_j: List[int], regime: str, dissimilarity_measure: mamf.dissimilarity_component.dissimilarity_calculator.Diss attributes: List[str] = None, **kwargs*) → bool

This function works as a factory method for the influence component. It calls either the many_to_one or the one_to_many function of a specific implementation of the InfluenceOperator and passes to it the kwargs dictionary. Which function is called is based on the selected communication regime.

Parameters

- **`network`** – The network that will be modified.
- **`realization`** – The specific implementation of the InfluenceOperator. Options are "stochasticOverlap", "axelrod".
- **`agent_i`** – the index of the focal agent that is either the source or the target of the influence
- **`agents_j`** – A list of indices of the agents who can be either the source or the targets of the influence. The list can have a
- **`attributes`** – A list of the names of all the attributes that are subject to influence. If an agent has e.g. the attributes "Sex" and "Music taste", only supply ["Music taste"] as a parameter for this function. The influence function itself can still be a function of the "Sex" attribute.
- **`regime`** – Either "many_to_one", "one_to_many" or "one_to_one".
- **`dissimilarity_measure`** – An instance of a `DissimilarityCalculator`.

Returns true if agent(s) were successfully influenced

54

1.6.2 Axelrod module

class mamf.influence_sim.Axelrod.**Axelrod**

Bases: *mamf.influence_sim.influence_sim.InfluenceOperator*

Implements the InfluenceOperator in a way that recreates the original Axelrod experiment.

static spread_influence(*network: networkx.classes.graph.Graph, agent_i: int, agents_j: List[int], regime: str, dissimilarity_measure: mamf.dissimilarity_component.dissimilarity_calculator.DissimilarityCalculator, attributes: List[str] = None, **kwargs*) → bool

In the influence function as Axelrod modeled it #todo insert reference agents are more likely to influence each other if they are more similar. If an agent successfully influences one or more agents, the influenced agents adopt one feature on which they disagreed from the influencing agent. In the case of many-to-one communication, the influenced agent adopts the mode value of a feature on which there is no consensus among the influencing agents.

Parameters

- **network** – The network in which the agents exist.
- **agent_i** – the index of the focal agent that is either the source or the target of the influence
- **agents_j** – A list of indices of the agents who can be either the source or the targets of the influence. The list can have a single entry, implementing one-to-one communication.
- **attributes** – A list of the names of all the attributes that are subject to influence. If an agent has e.g. the attributes “Sex” and “Music taste”, only supply [“Music taste”] as a parameter for this function. The influence function itself can still be a function of the “Sex” attribute.
- **regime** – Either “one-to-one”, “one-to-many” or “many-to-one”
- **dissimilarity_measure** – An instance of a `DissimilarityCalculator`.
- **kwargs** – Additional parameters specific to the implementation of the InfluenceOperator.

Returns true if agent(s) were successfully influenced

1.6.3 BoundedConfidence module

class mamf.influence_sim.BoundedConfidence.**BoundedConfidence**

Bases: *mamf.influence_sim.influence_sim.InfluenceOperator*

static spread_influence(*network: networkx.classes.graph.Graph, agent_i: int, agents_j: List[int], regime: str, dissimilarity_measure: mamf.dissimilarity_component.dissimilarity_calculator.DissimilarityCalculator, attributes: List[str] = None, **kwargs*) → bool

The bounded confidence model is from the family of similarity bias models. These models assume that how strongly agents influence each other is dependent on how similar they are. In the bounded confidence case the influence ‘strength’ is either the ‘convergence-rate’ or 0, if the agents are more similar than the threshold ‘confidence_level’ or below it, respectively. In the one-to-one communication regime, the agents can also influence each other if the ‘bi-directional’ parameter is set to true.

Parameters

- **network** – The network in which the agents exist.
- **agent_i** – the index of the focal agent that is either the source or the target of the influence

- **agents_j** – A list of indices of the agents who can be either the source or the targets of the influence. The list can have a single entry, implementing one-to-one communication.
- **attributes** – A list of the names of all the attributes that are subject to influence. If an agent has e.g. the attributes "Sex" and "Music taste", only supply ["Music taste"] as a parameter for this function. The influence function itself can still be a function of the "Sex" attribute.
- **regime** – Either "one-to-one", "one-to-many" or "many-to-one"
- **dissimilarity_measure** – An instance of a `DissimilarityCalculator`.
- **kwargs** – Additional parameters specific to the implementation of the InfluenceOperator. Possible parameters are the following:
- **confidence_level** (*float=0.8*) – A number between 0 and 1 determining the cut-off value for the dissimilarity at which agents do not interact anymore. 1 means that even the most dissimilar agents still interact, 0 means no interaction.Passed as a kwargs argument.
- **convergence_rate** (*float=0.5*) – A number between 0 and 1 determining how much an agent adopts other agents features. If it is one, the influenced agent takes the value of the influencing agent. Passed as a kwargs argument.

Returns true if agent(s) were successfully influenced

1.7 network_evolution_sim

1.7.1 network_evolution_sim module

class `mamf.network_evolution_sim.network_evolution_sim.`**NetworkModifier**

Bases: `abc.ABC`

The NetworkModifier changes the structure of the network. It can build or remove edges based on how agents are connected and what attributes they have.

static rewire_network(*network: networkx.classes.graph.Graph*, ***kwargs*)

Creates new connections or deletes existing ones. Can be used to implement coevolution of networks and model selection processes.

Parameters **network** – The network that will be modified.

`mamf.network_evolution_sim.network_evolution_sim.`**rewire_network**(*network: networkx.classes.graph.Graph*, *realization: str*, ***kwargs*)

This function works as a factory method for the NetworkModifier component. It calls the rewire_network method of a specific implementation of the AttributesInitializer and passes to it the kwargs dictionary.

Parameters

- **network** – The network that will be modified.
- **realization** – The specific NetworkModifier that shall be used to initialize the attributes. Options are "maslov_sneppen", ..
- **kwargs** – The parameter dictionary with all optional parameters.

56

1.7.2 NetworkHomophily module

class mamf.network_evolution_sim.NetworkHomophily.**MaslovSneppen**

Bases: *mamf.network_evolution_sim.network_evolution_sim.NetworkModifier*

static rewire_network(*network: networkx.classes.graph.Graph*, ***kwargs*)

This function picks random agents from the network and connects them to each other.

Parameters network – The network that is modified.

1.8 dissimilarity_component

1.8.1 DissimilarityCalculator module

class mamf.dissimilarity_component.dissimilarity_calculator.**DissimilarityCalculator**

Bases: abc.ABC

This class is responsible for determining the distance between nodes, either from one node to another, or for every agent in the network to another. The distance could be based on their attributes or actual geodesic distance.

static calculate_dissimilarity(*network: networkx.classes.graph.Graph*, *agent1_id: int*, *agent2_id: int*) → float

This function calculates how dissimilar two agents are based on their attributes and/or their distance in the network. Can for example be used to determine whether a neighbor is selected for the influence process.

Parameters

- **network** – The network in which the agents exist.
- **agent1_id** – The index of the first agent.
- **agent2_id** – The index of the agent to compare with.

:returns a float value, representing the distance between the two agents

static calculate_dissimilarity_networkwide(*network: networkx.classes.graph.Graph*)

Calculates the distance from each agent to each other and sets that distance as an attribute on the edge between them.

Parameters network – The network that is modified.

mamf.dissimilarity_component.dissimilarity_calculator.**select_calculator**(*realization: str*) → mamf.dissimilarity_compo

This function works as a factory method for the dissimilarity_component. It returns an instance of the Calculator that is asked for.

Parameters realization – The type of DissimilarityCalculator. Possible options are ["hamming", "euclidean"]

Returns An instance of a DissimilarityCalculator

57

1.8.2 EuclideanDistance module

class `mamf.dissimilarity_component.EuclideanDistance.`**`EuclideanDistance`**

Bases: *`mamf.dissimilarity_component.dissimilarity_calculator.DissimilarityCalculator`*

Implements the DissimilarityCalculator as a calculator of the euclidean distance

static calculate_dissimilarity(*network: networkx.classes.graph.Graph*, *agent1_id: int*, *agent2_id: int*) → float

Calculates euclidean distance of the agents' feature vectors.

Parameters

- **`network`** – The network in which the agents exist.
- **`agent1_id`** – The index of the first agent.
- **`agent2_id`** – The index of the agent to compare with.

Returns A float value, representing the distance between the two agents.

static calculate_dissimilarity_networkwide(*network: networkx.classes.graph.Graph*)

Calculates the distance from each agent to each other and sets that distance as an attribute on the edge between them.

Parameters **`network`** – The network that is modified.

1.8.3 HammingDistance module

class `mamf.dissimilarity_component.HammingDistance.`**`HammingDistance`**

Bases: *`mamf.dissimilarity_component.dissimilarity_calculator.DissimilarityCalculator`*

Implements the DissimilarityCalculator as a calculator of the Hamming distance

static calculate_dissimilarity(*network: networkx.classes.graph.Graph*, *agent1_id: int*, *agent2_id: int*) → float

Computes the Hamming Distance between two Agents, i.e. returns the proportion of features that the two agents have not in common. 1 means therefore total dissimilarity, and 0 is total overlap. Only works with categorical attributes.

Parameters

- **`network`** – The network in which the agents exist.
- **`agent1_id`** – The index of the first agent.
- **`agent2_id`** – The index of the agent to compare with.

:returns a float value, representing the distance between the two agents

static calculate_dissimilarity_networkwide(*network: networkx.classes.graph.Graph*)

Calculates the distance from each agent to each other and sets that distance as an attribute on the edge between them.

Parameters **`network`** – The network that is modified.

58

1.9 tools

1.9.1 NetworkDistanceUpdater module

`mamf.tools.NetworkDistanceUpdater.`**`update_dissimilarity`**(*network: networkx.classes.graph.Graph, agents: List[int], calculator: mamf.dissimilarity_component.dissimilarity_calculat*

This method recomputes the edges between a certain set of agents and all their neighbors and then modifies the edges between them respectively.

Parameters

- **`calculator`** – An implementation of the DissimilarityCalculator class
- **`network`** – The network that is updated.
- **`agents`** – A list containing the indices of all agents whose edges should be updated.

`mamf.tools.NetworkDistanceUpdater.`**`check_dissimilarity`**(*network: networkx.classes.graph.Graph, threshold: float*)

This function is used to check whether influence is theoretically still possible. For that is checked whether any edge has a dissimilarity value inbetween a certain range of possible values that allow for further influence steps. This range has 1 as the upper limit and a given threshold as lower limit.

Parameters

- **`network`** – The network that is checked.
- **`threshold`** – The lower limit of dissimilarity that makes it possible for agents to influence each other.

Returns True if there is still change possible, False otherwise

1.9.2 OutputMeasures module

`mamf.tools.OutputMeasures.`**`homogeneity`**(*network*)

A measure of how much consensus exists in the network.

Parameters **`network`** – The network to be measured.

Returns The homogeneity measure 'S_max / N'

`mamf.tools.OutputMeasures.`**`isol`**(*network*)

Counts how many agents belong to no cluster.

Parameters **`network`** – The network to be measured.

Returns The count of isolates in the graph.

`mamf.tools.OutputMeasures.`**`clustercount`**(*network*)

Counts how many clusters exist in the network. #todo: reference on what a cluster is

Parameters **`network`** – The network to be measured.

Returns The number of clusters in the network.

`mamf.tools.OutputMeasures.`**`regionscount`**(*network*)

Counts how many agents belong to each cluster in the network and returns a list of these numbers.

Parameters **`network`** – The network to be measured.

Returns A list with the size of all the clusters in the graph.

1.9.3 ClusterExecutionScript module

class mamf.tools.ClusterExecutionScript.**parallelSimulation**

Bases: object

This class exists solely for the purpose of running a chunk of different parameter combinations in parallel on one machine. This script is called from the command line with 3 arguments.

Parameters

- **index of the chunk that this instance shall run.** (*The*) –
- **path to the folder with the pickle files from which the Simulation parameters are read.** (*The*) –
- **path to where the output shall be written.** (*The*) –

main()

create_and_run_simulation(*parameter_dict*)