Social dynamics of substance use through minds and models

van den Ende, M.

Publication date
2025

Dynamics *in* and Dynamics *of* Networks using DyNSimF

This chapter has been adapted from: Maarten W. J. van den Ende, Mathijs Maijer, Mike H. Lees, Han L. J. van der Maas. *Dynamics in and dynamics in networks using DyNSimF*. Under review in *Journal of Computational Science*.

Abstract

Advances in formal theories, network science, and data collection technologies make complex-agent networks and adaptive networks increasingly powerful tools in the fields' of complexity science and computational social science. We present DyNSimF; an open source package that facilitates the modelling of adaptive networks, capturing complex interacting dynamics *on* a network as well as dynamics *of* (the structure of) a network.

Capable of complex agent-based simulations on a dynamic network, it is able to capture individual-level dynamics as well as dynamics of the network structure, and how these interact and evolve. By capturing the emergent behaviour resulting from the interactions of node states and network topology, we argue that DyNSimF will help modellers to gain a fundamentally better understanding of complex network systems.

The package can handle both weighted and directional links, is computationally scalable and efficient, and includes a generic utility-based edge selection framework. DyNSimF provides a generic modelling framework for dynamics networks and includes visualisation methods and tools to aid in the analysis of models. It is designed to be extensible and aims to be easy to learn and work with, allowing non-experts to focus on model development, while being highly customisable and extensible to allow for complex custom models.

3.1 Introduction

The field of complexity science offers new ways of studying many of the world's current problems and natural phenomena. Over the past decade, this field has experienced significant growth, helping to unravel the complex interactions seen in social behaviour (ABSS; Davidsson, 2002), nature, economics (Tesfatsion, 2003; Ferrary and Granovetter, 2009) and in many other areas (CAN; Mei et al., 2015; Bettencourt and West, 2010; Vespignani, 2009), as they provide a way to use surrogate reasoning (Swoyer, 1991) and enable collaboration across disciplines to solve these complex multidisciplinary problems (Cohen and Havlin, 2010; Forsman et al., 2014; Fortunato, 2011; Fan et al., 2020; Arthur, 2014).

The behaviour of complex systems is characterised as consisting of many interacting components, each of which adapts and interacts in a non-linear way. They are driven by feedback loops in which macroscopic patterns of the system affect microscopic interactions, which in turn lead to new macroscopic dynamics. For example, in modelling the dynamics of social opinion, a small change in an individual's information can lead to a catastrophic change in opinion, which can then have a cascading effect on the opinions of many surrounding agents (van der Maas et al., 2020). To represent and reason about such systems, it is necessary to view the system holistically, capturing the interactions between elements and how these interactions evolve over time (Phelan, 2001). Agent-based models (Gilbert, 2019) and complex networks (Barbati et al., 2012; Sloot et al., 2013) are two common approaches and have proven successful in describing the behaviour and interactions in many application domains, such as protein-protein interactions, financial networks, ecosystems and climate change (Mei et al., 2015), as well as social simulation domains (Gilbert, 2004; Conte et al., 2012). Examples are social divide and opinion dynamics (Giardini and Vilone, 2021; van der Maas et al., 2020), crime and crisis management (Lorig et al., 2021; Donges et al., 2009) and altruism (Neumann, 2020).

These techniques have contributed to important scientific breakthroughs (Mei et al., 2015; Davidsson, 2002). However, in the past, they have predominantly dealt with either the dynamics *on* the network or the dynamics *of* the network. In real-world complex networks, however, the states of the nodes and the structure of the network interact and evolve with each other. This state-topology co-evolution is recognised as the next challenge in complex network research, leading to the emerging field of *Adaptive Networks*. (Gross and Blasius, 2007; Mata, 2020; Sayama et al., 2013; Wang et al., 2019). Increasing computational power and modern data collection methods to obtain large-scale longitudinal data have made the modelling of adaptive networks increasingly feasible. However, the implementation of a general framework for adaptive network modeling would be the next step for this emerging field.

In this paper we present DyNSimF, a Python Foundation (2021) software package focused on facilitating adaptive network modelling: it allows the use of agent-based models to simulate complex phenomena in a dynamic network; it allows modelling dynamic networks with complex, continuous internal dynamics within each node. In this paper, we show how node dynamics and network dynamics can be intuitively coupled using DyNSimF. We do this by elaborating a basic implementation of the Susceptible-Infected-Recovered (SIR) model of spreading phenomena with self-quarantining nodes, a canonical example in adaptive networks (Sayama et al., 2013; Cherifi et al., 2019; El-Sayed

et al., 2012). More detailed examples, such as an opinion dynamics model (van der Maas et al., 2020) and a comprehensive psychological model of the spread of smoking (Grasman et al., 2016), can be found in the package documentation[1].

While other simulation software such as Nepidemix (Ahrenberg et al., 2016), hybridModels (Marques et al., 2020), and most notably NDlib (Rossetti et al., 2018) can work to some extent with networks that are dynamic, DyNSimF is novel in that it allows for complex internal dynamics with multiple non-discrete states, and allows the internal dynamics of these nodes to directly affect their local network. Since the network also affects the internal dynamics of each node, DyNSimF is novel in its ability to simulate adaptive networks: interactions between the internal dynamics of nodes and the dynamics of the network structure. DyNSimF provides this while retaining a high level of computational performance, in the paper we compare the performance of an SIR simulation implemented in three similar frameworks. Our results show that for differing network sizes, DyNSimF is faster in all cases and offers up to 21× speed up in the best case. We provide DyNSimF as a well suited framework for modelling many kinds of complex phenomena, ranging from social and psychological models in artificial societies to ecosystems, economic models and many other kinds of (complex) agent-based simulations on a (dynamic) network.

3.2 Methodology

DyNSimF facilitates the interaction of the changing nodes with their changing local network by modeling the dynamics within the nodes and the dynamics of the network structure separately. As shown in Figure 3.1, these two user-defined models are then coupled to facilitate interactions between them. The *nodal model* defines how the nodes' internal variables—here called *states*—of each node change, depending on its own characteristics and the characteristics of its surrounding nodes. The *connectivity model* determines how nodes change their connections and thus how the network structure changes. It takes information about its own states, the states of its connections, and the states of the nodes with which it can form new connections. It uses this information to decide which nodes to disconnect or connect to. In subsequent iterations, it is these new connections that are used as input to the node model, thus affecting subsequent internal states. In short, the interplay of the *nodal model* with the *connectivity model* describes how the network structure reacts (or adapts) to changes in the states of the nodes, and how the states of the nodes are affected by the network structure.

Figure 3.1 shows the flowchart of the steps DyNSimF uses to simulate models including the data flow of each subsystem. This process is performed for each node before going into the next iteration step. The following sections describe each of these parts and their interactions in-depth. Once at least the nodal model is defined, an initial network configuration, and the initial node states are given, simulations can be run. With only the nodal model defined, no network changes are made, and DyNSimF operates as a more traditional, non-dynamic network model capable of dealing with floats, or real numbers, as states. After initializing, a custom sampling method can be used indicating for which nodes and in what sequence the model should run. Also, the user can select or specify certain metrics to be stored during the simulations, useful for later analysis.

[1] https://dynsimf.readthedocs.io/en/latest/

The implementation and detailed functionality of the features above are discussed in the following sections.

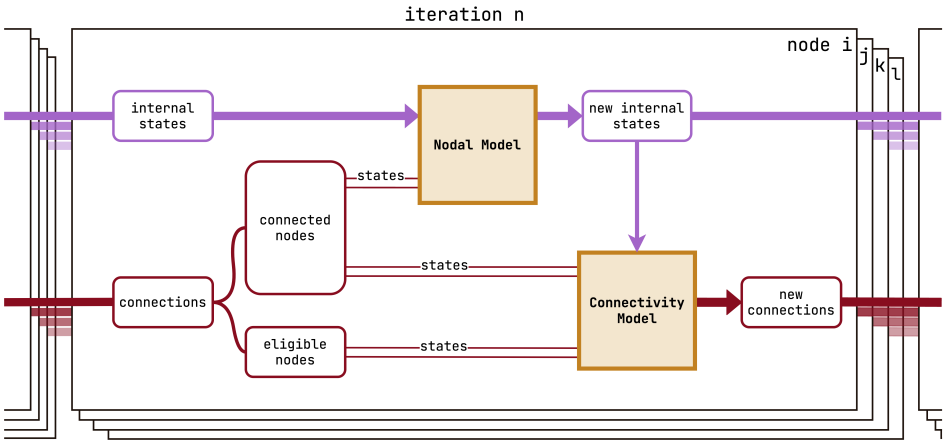

Figure 3.1: Workflow of DyNSimF for each iteration. Each slice in the 'stack' represents a node, and the sequence of iterations follows left to right. For each node, the *nodal modal* takes the internal states from previous iterations as well as the states of its connections as input. Its output is the node's new internal states. These states are used by the *connectivity model*. The connectivity model also takes the states of the nodes connected to node i, and of the nodes that are eligible for new connections. It then calculates if and how the node changes its local network, i.e., if connections are broken or new connections are made. This process is repeated for each node, either in a sequence defined by a sample function, see Section 3.5, or all at once when using matrix calculations, see Section 3.3. The new network is stored and passed to the following iteration, along with the new internal states of the nodes. After this, the following iteration begins.

Nodal model

In DyNSimF, each node can have its own internal dynamics. Thus, each node can be seen as a model in itself, similar to the agents in an agent-based model. The *nodal model* is defined by the user and represents the behaviour of each node. It describes how the states of the nodes change over time, depending on their past states and their local network. The internal states of the nodes are changed during the simulation as defined by one or more *update* functions. By default, these functions are implemented as difference equations; DyNSimF calculates the value of all node states of the current time step using information from the previous time step. This means that by default DyNSimF uses Euler's method to solve differential equation-based models, although it is possible to implement a custom solver. DyNSimF is built on Numpy (Harris et al., 2020) arrays and routines, so if the custom model implements these routines as well, DyNSimF can be highly optimised.

Connectivity model

DyNSimF provides methods for modelling dynamic networks: nodes can be removed or added, and their connections — also called 'ties' or 'edges' — can be modified. This can be done in two ways: by manually changing the NetworkX (Hagberg et al., 2008) graph, or by defining a *utility-cost* function and letting DyNSimF handle changes in the network structure via a clear classification of the ways agents can communicate and exchange information (Angus and Hassani-Mahmooei, 2015). Changing the network manually consists of providing the model with the added or removed nodes and the changed links, as explained in Section 3.5. Here we explain the *utility-cost* method, which is another generic way of defining how nodes change their edges.

Utility-cost

The *utility-cost* methodology is based on the idea that nodes are agents that behave strategically: they make rational decisions based on the estimated costs of an action against its potential benefits, or utility (Myerson, 2013). In network formation, this means that agents assess the utility and cost of their current and possible future connections, and adjust their edges accordingly (Mele, 2010). While the 'utility' nomenclature is based on behavioural economics, social science approaches are similar, such as the work of Snijders et al. Snijders et al. (2010). They describe in detail the different 'effects' that can influence the attractiveness of individuals to connect with each other. These effects depend on the internal states of the involved agents as well as on their position in the network and its local topology.

In DyNSimF, each node in the network can be seen as an agent that aims to maximise its total utility: the sum of the utility from its connections. Each node can gain a certain amount of utility by forming new connections, but maintaining or creating these connections can come with a certain 'cost'. In social networks, this cost may represent the amount of effort or time an individual spends making and maintaining friendships, while in economic models it could represent some sort of economic benefit. Each node has a finite amount of this resource; it is limited in its ability to make and maintain connections. Therefore, the sum of the costs associated with all of a node's connections cannot exceed a certain level. When this threshold is exceeded, nodes are forced to remove connections, presumably those with high cost and low benefit. Finally, the information available to agents when choosing their new connections has a strong impact on the network dynamics. This can be reflected in a restriction of the information received about the internals of other nodes, or in a restriction of the nodes eligible for new connections.

3.3 Software implementation

This section demonstrates the practical workflow of DyNSimF by creating a disease spread model on a network with self-quarantine dynamics, based on the work of Lagorio et al. (2011) (Lagorio et al., 2011) and similar approaches described by Cherifi et al. (2019) (Cherifi et al., 2019), Wang et al. (2019) (Wang et al., 2019), and Gross et al. (2006) (Gross et al., 2006). This spread of disease can also be interpreted as the spread of controversial opinions or behaviours, such as heroin or problematic alcohol use. Many SISa, SIR

and SI-type models have been used to model spreading phenomena in social systems (Hill et al., 2010a,b; van den Ende et al., 2022). We also provide possibilities for more complex models and detailed technical instructions for their implementation. Additional models, such as an opinion dynamics model and a comprehensive psychological model of addiction, can be found in the documentation or in the example document available on GitHub (dyn, 2023b).

Here, we reproduce a simplified version of 'strategy A' of Lagorio et al. (2011) Lagorio et al. (2011): every time step, every infected node attempts to transmit the 'disease' to a susceptible neighbour. Simultaneously, each node with an infected neighbour has a probability w to self-quarantine and disconnect from the infected node. After a fixed time, infected nodes recover. Depending on the infection rate, the recovery time and especially the self-quarantine probability, this can lead to an endemic or epidemic state.

Installation

There are several ways to install DyNSimF. The easiest way is to use PyPI[2] (pyp, 2022), which can be obtained by installing the python setuptools or, if using anaconda, (ana, 2020) by running `conda install pip`. The package can then be installed by running `pip install DyNSimF`. The library can also be installed by downloading it from GitHub (dyn, 2023b). After unpacking it to the desired directory, the package can be installed by running `python setup.py install`.

Setup

To set up a model, we need to import the necessary dependencies. Each model created with DyNSimF uses a NetworkX graph. The states and relationships of each node are tracked using this graph object. With Numpy (Harris et al., 2020) imported, DyNSimF can implement its array operation routines, increasing the speed when working with large networks. Finally, to set up a DyNSimF model, it is necessary to import the Model class. When modelling a static network, `dynsimf.models.Model` can be used. For dynamic networks you should use `UtilityCostModel`, which is explained in Section 3.3.

```
import numpy as np
import networkx as nx
from dynsimf.models.Model import Model
```

Next, we define the initial graph. This should be a NetworkX `Graph` object. DyNSimF can work with undirected and directed, weighted and unweighted edges, and with both graphs as well as multigraphs. The graph can be defined manually or be created with NetworkX graph generators. An example would be: `g = nx.erdos_renyi_graph(n = 1000, p = 0.1)`, which would store an Erdös-Rényi graph (Erdős and Rényi, 1960) (i.e., a binomial graph with 1000 nodes and a 0.1 probability for edge creation). The example below creates a graph with small-world properties by using `watts_strogatz_graph`.

Once the network has been defined, the `Model` class object can be defined. This model requires the previously defined NetworkX graph and takes this as an argument to its constructor. When constructing the model, it is also possible to specify a random

[2] https://pypi.org/project/dynsimf/

seed, as well as a configuration object which defines the data to keep track of for use in analysis, discussed in Section 3.4.

```
n = 25
g = nx.watts_strogatz_graph(n = n, k = 5, p = 0.2)
model = Model(g)
```

Nodal model

Initialization

To create the node model, we first need to define the constants and the states it uses and set their initial values. DyNSimF implements the constants as a dictionary. This dictionary is the input to the `model.constants` object. The constants can be of any type, and they can either be a single value, where each node has the same constant, or, if each node has a different constant, they can be a list of the same size as the number of nodes.

To tell the package which state variable names are used in the model, the `model.set_states()` function should be used. It takes a list of strings with the names of the states as input. Their initial values can then be set using `model.set_initial_state()`. The first argument of this function is a dictionary with as keys the names of the state variables and as values lists of their initial values for each node. As with constants, the initial value for states can be a single-shared value or a list with an individual value for each node. The argument can also be a function that returns this list. If it is a function, there is another argument: a dictionary that specifies the arguments passed to this custom function. This can be seen in the example code below.

```
constants = {
    'n': n,
    'initial_infected': 2,
    'beta' : 0.2,
    'recovery_time': 51
}

def initial_phase(constants):
    # Initialize everyone as Susceptible, value 0
    phase = np.zeros(constants['n'])
    # Give the initial Infected I value 1
    phase[np.random.choice(
        len(phase), constants['initial_infected'])] = 1
    return phase

model.constants = constants
model.set_states(['phase', 'toi'])
model.set_initial_state(
    {'phase': initial_phase},
    {'constants': model.constants}
)
# time of infection
model.set_initial_state(
```

```
    {'toi': np.zeros(n)})
```

In this example, we use `phase` to indicate in which phase of each node is (Susceptible: 0, Infected: 1 and Recovered: 2). Further, there is a state called `toi` (time of infection), keeping track of how many time steps a node is infected. This is 0 for each node, but will increase every time step for an infected node.

Update functions

The way in which the internal states of the nodes are changed during the course of the simulation is defined by *update* functions. These functions calculate the value of all node states of the current time step using information from the previous time step. This means that by default DyNSimF uses Euler's method to solve differential equation based models. This can have an impact on accuracy, depending on the chosen time step and the time sensitivity of the system. If you need to use a different solver, you can use the `get_previous_nodes_states()` discussed in Section 3.5 to implement your own. This quarantine model has a simple forward-time scheme, but more complicated examples based on differential equations with multiple interacting states can be found in the supplementary materials.

The user-defined update functions should return a dictionary where the keys refer to the state and the values are lists, ordered by node id, giving the new values for that state per node. This means that it is possible for one function to update all states of all nodes at once, or to have multiple functions updating different states. The update functions can be added to the model using the `model.add_update()`. Again, the first argument to this function is the update function created by the user. The second argument is a dictionary containing the key/value pairs of the argument to be passed to the function. If the update function is to apply only to a selection of nodes, a third argument, a boolean called `get_nodes`, can be passed. Then only the list of nodes passed as an argument will be updated by this function.

When the update function is created and added to the model, a simulation with a static network can already be performed. This can be done with `model.simulate(N)`, where N is the number of iterations the model should be running for.

```
def update_states(constants):

    SIR = model.get_state('phase')
    infected_time = model.get_state('toi')
    I = np.where(SIR == 1)[0]

    # Infected nodes have beta chance to infect neighbors
    for infected_node in I:
        nbs = model.get_neighbors(infected_node)
        for nb in nbs:
            if np.random.rand(1) < constants['beta']:
                SIR[nb] = 1

    # Heal after certain time
```

```
            infected_time[infected_node] += 1
            if infected_time[infected_node] >
                constants['recovery_time']:
                SIR[infected_node] = 2

        return {'phase': SIR, 'toi': infected_time}

model.add_update(update_states, {'constants': constants})
model.simulate(50)
```

Connectivity model

One of the main novelties of DyNSimF is that it provides methods for modelling dynamic networks: nodes can be removed or added, and their connections can be modified. As can be seen in Figure 3.1, by default in DyNSimF, updates are calculated for each node and state at each time step. Following these internal node updates, the node reconfigures its connections. This reconfiguration can be implemented in two ways: by manually changing the NetworkX graph, as explained in Section 3.5, or by defining a *utility-cost* function and letting DyNSimF handle changes in the network structure. The latter is demonstrated for our example below.

Utility-Cost

To implement the Utility-Cost methodology in DyNSimF, a utility and cost function should be defined by the user. In addition, the information available to the agents when evaluating possible future links should be specified.

There are two ways to define the utility and cost functions. These functions can either return a value for two given nodes, or they can return a complete matrix, allowing Numpy routines. To tell DyNSimF which function is used, the enumeration class `FunctionType` is used, which is either `FunctionType.MATRIX` or `FunctionType.PAIRWISE`. In the case of a pairwise function, the utility or cost function should take 2 arguments, one for node i and the other for agent j. If `MATRIX` is chosen, the utility or cost function should return a matrix with dimension $n \times n$, where n is the number of nodes in the network. Each entry M_{ij} then represents either the utility that agent i receives from connecting to agent j, or the cost associated with each connection.

The information available to each agent can be defined by setting the argument of the `set_sampling_function()`. Passing `SampleMethod.ALL` assumes *complete information*: any node can be connected to any other node. Alternatively, `SampleMethod NEIGHBORS_OF_NEIGHBORS` restricts a node so that it can only connect to nodes within a distance of two— 'neighbours of neighbours'. Finally, a custom sampling function can be selected using `SampleMethod.CUSTOM`. This custom function should be passed as the second argument to `set_sampling_function()` and will return a $n \times n$ binary matrix g, with entry $g_{ij} = 1$ if agent j is a possible connection for agent i. To add dynamic network formation to a model using utility and cost functions, instead of using the `Model` class shown earlier, the extended class `UtilityCostModel` should be used, with an additional

argument– `cost_threshold` indicating the threshold for the maximum sum of the costs associated with each node's connections.

In the example below we have added a `MemoryConfiguration` object. This indicates that the adjacencies (a $n \times n$ matrix indicating which nodes are connected) should be stored in the memory needed to visualise the dynamic network. More information about this configuration is given in the Section 3.4.

The cost threshold in this example is one minus the rewiring probability w used by Lagorio et al. (2011). The `cost_calculation` makes it so that a susceptible node will remove an edge connecting to an infected neighbouring node if a drawn random number is higher than this cost threshold. For simplicity, in this example we only allow the model to break connections; we give the nodes a custom sampling function that returns a matrix of zeros such that no node is eligible to establish a connection with. With no eligible connections, there is no need to calculate which node would have the highest utility, so the specific function we give for utility does not matter in this case. Finally, we add these functions to the model using `FunctionType.MATRIX` as we return full matrices for cost and utility. After saving the result of `simulate` we can start to analyse our results.

```
from dynsimf.models.UtilityCostModel import UtilityCostModel
from dynsimf.models.UtilityCostModel import FunctionType
from dynsimf.models.UtilityCostModel import SampleMethod
from dynsimf.models.Model import ModelConfiguration
from dynsimf.models.components.Memory import
    MemoryConfiguration
from dynsimf.models.components.Memory import
    MemoryConfigurationType


cfg = {
'adjacency_memory_config': \
    MemoryConfiguration(MemoryConfigurationType.ADJACENCY, {
        'memory_size': 0
    }),
}

cost_threshold = 0.45
n = 25
g = nx.watts_strogatz_graph(n = n, k = 5, p = 0.2)
model = UtilityCostModel(g, cost_threshold, \
    config = ModelConfiguration(cfg))

def cost_calculation():
    # Define variables
    SIR = model.get_state('phase')
    I = np.where(SIR == 1, 1, 0)
    S = np.where(SIR == 0, 1, 0)

    # Probabilities to disconnect
    p_disconnect = np.random.rand(constants['n'],
        constants['n'])
```

```
        # Only disconnect with infected
        cost = p_disconnect × I

        # Only susceptibles disconnect
        cost = S × cost.T

        return cost

    def utility_calculation():
        # No new connections -> placeholder utility
        utility = np.ones((constants['n'], constants['n']))
        return utility

    def sampling_method():
        # No new connections, no nodes eligible
        return np.zeros((constants['n'], constants['n']))

    model.add_utility_function(utility_calculation, \
        FunctionType.MATRIX)
    model.add_cost_function(cost_calculation, \
        FunctionType.MATRIX)
    model.set_sampling_function(SampleMethod.CUSTOM, \
        sampling_method)

    output = model.simulate(50)
```

3.4 Analysis

Memory configuration

By default, the output of the `simulate` function returns data that can be used for analysis. It is a dictionary that can contain four metrics: the states, the adjacencies, the edge values and the utilities[3]. Its structure is shown in Figure 3.2. By default DyNSimF stores only the states of each node for each simulation step. However, using the `ModelConfiguration` object, the user can specify which variables DyNSimF should keep track of and how.

The `ModelConfiguration` object is passed when the `Model` object is invoked. There are four types of properties that the user can configure to be stored in memory during simulation. They are specified using keys: `state_memory_config` for states, `adjacency_memory_config` for adjacencies, `edge_values_memory_config` for edge values and `utility_memory_config` for utilities.

The second argument is another dictionary containing the actual configuration for this type. These settings are the keys: `save_disk`, `memory_size`, `save_interval` and `memory_interval`. The `save_disk` option is a boolean value that allows the user to save their output metrics to a file on disk during the simulation. This is a `.txt` file with the

[3]Note that the 'edge values' are only relevant when using weighted links and the 'utility' is only relevant when using the utility-cost method.

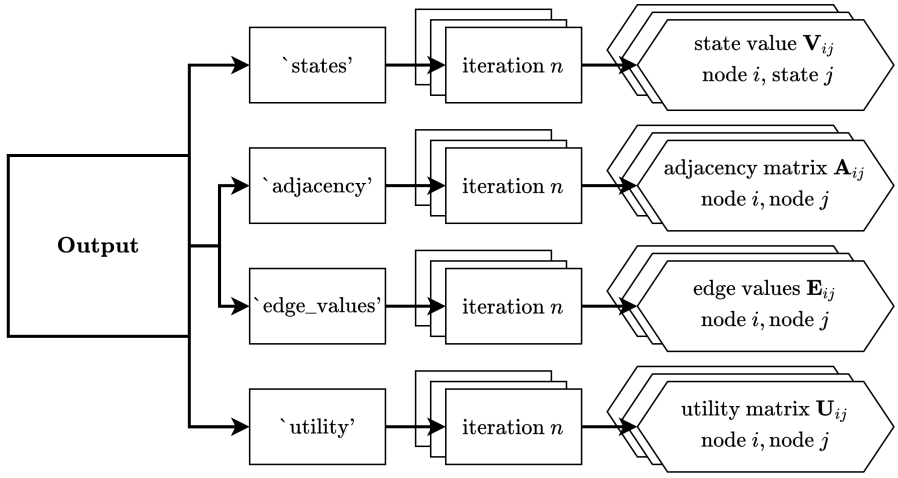

Figure 3.2: A representation of the data structure of the output of the `simulate()` function. Each square is a dictionary, starting with output, which has four keys as shown above. Each of those keys (e.g., `states`) points to a value which is another dictionary. The keys of this dictionary are the iteration number, and they refer to the matrix of that metric of that iteration. The order of the columns in the 'states' array matches the order of the states set using the `set_states()` function.

name of the type, e.g. `STATE.txt`, and is saved in the `./output/` folder. The `memory_size` can be used to set the maximum number of iterations the values should be stored in the output of the `simulation` function, where -1 indicates that they should not be stored at all. The values `save` and `memory_interval` determine how many iterations should be between saving.

The sample code below creates a model where the states of the nodes are saved at each iteration of the simulation: every iteration they are saved to disk, while every 10 iterations they are saved to memory.

```
from dynsimf.models.Model import Model
from dynsimf.models.Model import ModelConfiguration
from dynsimf.models.components.Memory import MemoryConfiguration
from dynsimf.models.components.Memory import
    MemoryConfigurationType

cfg = {
    'state_memory_config': \
        MemoryConfiguration(MemoryConfigurationType.STATE, {
            'memory_size': 1,
            'save_disk': True,
            'save_interval': 1,
            'memory_interval': 10
        }),
```

```
}
model = UtilityCostModel(g, ModelConfiguration(cfg))
```

Property functions

DyNSimF can keep track of custom, user-defined properties of the model during simulations by using 'property functions'. This allows different algorithms to be implemented to keep track of network coefficients, network scale metrics or any other measure. The initialisation of a property function requires four inputs: its name, the user-defined function to be performed, the arguments of that function, and an iteration interval. During the simulation, these property functions are executed according to their iteration intervals, and their output is stored as values in a dictionary with each 'property name' as a key. After running the simulation, this dictionary can be retrieved using `model.get_properties()`. After importing the class, an object can be created by creating an instance and adding this instance to the model as follows:

```
from dynsimf.models.components.PropertyFunction \
    import PropertyFunction

prop_1 = PropertyFunction(
    'property name',
    custom_function,
    iteration_interval,
    {'param_1': argument}
    )

model.add_property_function(prop_1)
```

Visualization

DyNSimF provides methods to dynamically visualise the network and the states of the nodes during a simulation. An example of a frame of the quarantine model is shown in Figure 3.3. The network is shown with nodes coloured according to a user-selected state. Plots of (a selection of) the states are displayed below the network bar. This animated visualisation is rendered as a Matplotlib (Hunter, 2007) window and can be saved as an animated `.gif` file.

Visualisation methods offer many configuration options. Here we highlight some of the most important ones. These are the `plot_interval`, which describes the interval of iterations to be plotted, the `plot_variable`, which specifies which state the colour of the nodes represents, the `layout`, which specifies the node positioning algorithm to be used, and the `plot_output`, which specifies where the `.gif` should be saved. If not all states are to be displayed in the histograms, specific states can be selected using the `histogram_states` key.

A dictionary containing these visualisation options should be passed as the first argument to `configure_visualisation` of a `Model` object, as shown in the example code below. The second argument should be the output of the simulation function. The `visualize('animation')` function on the `Model` object launches a Matplotlib window to

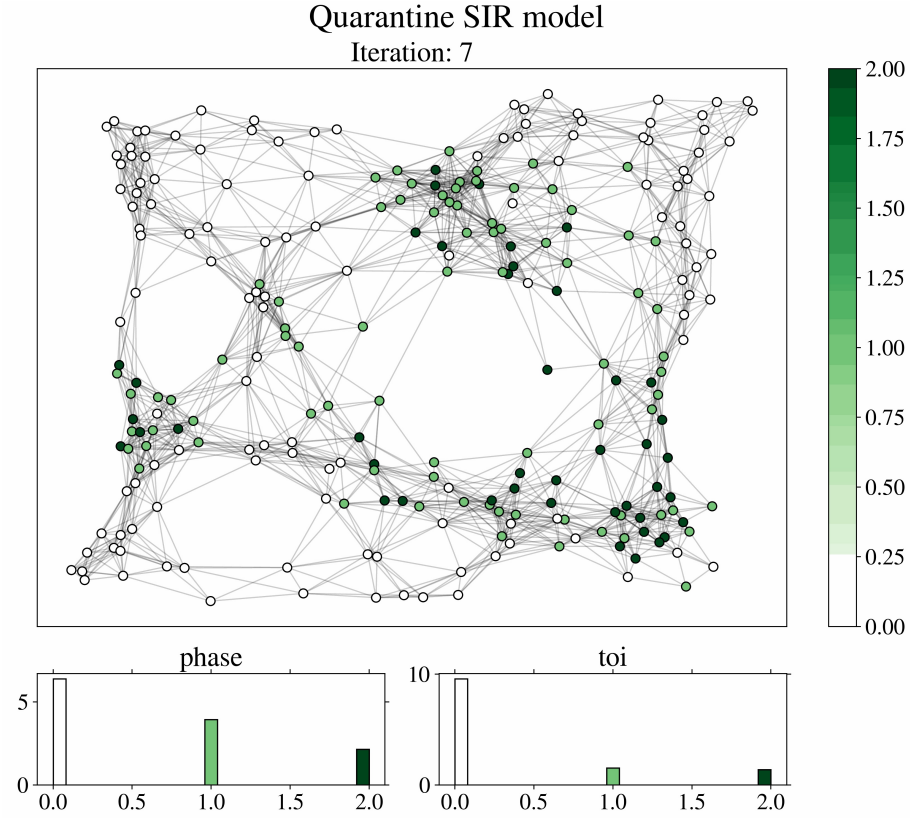

Figure 3.3: One frame produced by the visualization tool. It shows the title, current iteration, the network, and bar plots of the model states. To the right of the network, a color bar showing the color scale of the nodes is displayed. The colors of the nodes represent a specific state of the model, which in this case is the 'phase'.

display the animation and, if required, save the animation. The code below results in Figure 3.3.

```
visualization_config = {
    'plot_variable': 'phase',
    'fixed_positions': \
        nx.drawing.layout.spring_layout(g, seed = seed),
    'variable_limits': {'phase' : [0,2]},
    'color_scale': 'Greens',
    'show_plot': True,
    'plot_output': './animations/quarantine.gif',
    'plot_title': 'Quarantine SIR model',
    'directed': False,
    'edge_values': 'no'
}
```

```
model.configure_visualization(visualization_config, output)
vis = model.visualize('animation')
```

3.5 Additional functionalities

Interacting with DyNSimF

Models that combine the dynamics *on* and *of* a network require access to various characteristics of the network, such as the neighbours of a node or the values for different states of a node during previous iterations. Several functions have been implemented to provide access to this data. The list below shows the most common functions that modellers can use when creating a new model.

- `get_state(state_name)` Returns the values of a state for all nodes. This function is generally used to update the state for all nodes by retrieving the current values and modifying them.
- `get_utility()` This function returns a square $n \times n$ matrix A such that its element A_{ij} is the currently computed utility for the edge from vertex u_i to u_j.
- `get_previous_nodes_states(n)` Get all node states from the n'th previous saved iteration. If t is the current iteration, then the states from iteration $t-n$ are returned. With this function it is possible to use differential equations or to get the average state values of the nodes for the last few iterations. Note that the memory must be configured to store at least as many iterations as $t - n$.
- `get_previous_nodes_utility(n)` This function returns the matrix from the `get_utility()` function from a previous iteration, given as argument n, similar to `get_previous_nodes_states(n)`.
- `get_neighbours(node)` Returns all neighbours of a node. This function returns a list of NetworkX node indicators that are adjacent to the node given as the argument to the function.
- `get_neighbors_neighbors(node)` This function can be used to get only the neighbours of the neighbours of a node. It returns the row of the neighbours of neighbours adjacency matrix that matches the node given as the argument, but returns only a list of the NetworkX numbers of the neighbours of neighbours.
- `get_adjacency()` Retrieves the adjacency matrix. This is a square $n \times n$ matrix A such that its element A_{ij} is 1 if there is an edge from vertex u_i to vertex u_j and 0 if there is no edge (Biggs et al., 1993, definition 2.1, p. 7).
- `get_neighbors_neighbors_adjacency_matrix()` This function returns an adjacency matrix for neighbours of neighbours. This is also a square *n timesn* matrix A such that its element A_{ij} is 1 if there is a vertex u_k that has an edge from u_i to u_k and there is an edge from vertex u_k to vertex u_j. It is 0 if there is no such vertex.

Conditions

Sometimes updates should only occur when certain conditions are met for certain nodes. To achieve this, nodes can be filtered using *conditions*. Only the nodes that meet all the conditions will be updated by the appropriate update function. There are three types of conditions that can be used. The *Stochastic condition* triggers an update for each node based on a given probability P. The *Threshold condition* filters nodes based on a chosen variable, threshold and operators $\leq, <, >$ *and* $\geq$. The threshold is a single given value and the variable can be any variable in the model. *Custom Conditions* filter nodes by applying a user defined function. This function should return a list of the identifiers of the nodes that satisfy the condition. Custom conditions are executed on all nodes at each time step, just like the other conditions, which means that they can be implemented efficiently. Conditions can be chained; nodes that satisfy one condition can be passed on to the next. This can be done by adding the argument `chained_condition` and passing another condition object.

A condition is implemented by adding it to the `model.add_update` object. In the example below, if a node has two or less connections, it adds a neighbour; the function `add_neighbour` is called. The argument `get_nodes` returns the nodes that satisfy the condition. This is an example of a manual network update, which is explained in Section 3.5.

```
from dynsimf.models.components.conditions.Condition \
    import ConditionType
import
    dynsimf.models.components.conditions.ThresholdCondition
    as tc

def add_neighbor(nodes):
    changes = {'edge_change': {i: i+1 for i in nodes}}
    return changes

# condition indicating 2 or less than 2
cfg_t = tc.ThresholdConfiguration(tc.ThresholdOperator.LE, 2)
# passes the adjacency of a node for the input of condition
nb_condition =
    tc.ThresholdCondition(ConditionType.ADJACENCY, cfg_t)

# add_neighbor gets called, nodes passed satisfy the
    condition
model.add_network_update(add_neighbor, \
    condition = nb_condition, get_nodes = True)
```

Schemes

Schemes are another way DyNSimF can change the way updates are performed. They allow the user to specify a node sampling function: a function that returns a selection and order of nodes for which certain updates should take place. In addition, the schemes allow for 'external' perturbations, such as a 'labour supply shock' used in economic

models. To do this, schemes can restrict the execution of certain updates to a particular selection of iterations during the simulation. A lower and upper bound can be set within the definition of the scheme, and the passed updates will only occur if the number of iterations is within these bounds.

Below is shown how a scheme with updates is added to a model, which could, for example, represent the introduction of super-spreaders or a super-spreading event. Once the scheme and update classes have been imported, a `Update` object can be created, in this case linked to a previously defined function called `update_fun`. Then a `Scheme` object is created, taking a user defined sampling function as the first argument. The second argument is a dictionary which takes four keys: the arguments to the sampling function, the lower and upper iteration bounds, and a list of updates. Once the scheme has been created, it can be added to the model using the `add_scheme` function.

```
from dynsimf.models.components.Scheme import Scheme
from dynsimf.models.components.Update import Update
u = Update(update_fun)
s = Scheme(sample_function, {
        'args': {'graph': model.graph},
        'lower_bound': 0,
        'upper_bound': 10,
        'updates': [u]
    })
model.add_scheme(s)
```

In the example, the update function `update_fun` would be executed during the first 10 iterations. In this case the function `sample_function` takes an argument called `graph`, which is the graph of the model. When creating a graph, any of the keys can be omitted except `updates`. If the lower bound is omitted, it is assumed that the scheme should be active from iteration zero, while if the upper bound is omitted it is assumed to be active until the end of the simulation.

Manual network updates

It is possible to manipulate the network directly by adding a network-update function to the model via `model.add_network_update()`. This function should return a dictionary with specific keys, indicating the type of change to the dictionary values. Figure 3.4 shows the changes made to a network by the sample code below, demonstrating all the possible options.

There are two keys for manipulating nodes: `add` and `remove`. The corresponding value to the `remove` key should be a list of the names of all nodes to be removed from the network. When using the `add` key, the corresponding value should be a list of dictionaries, where each dictionary in the list is used as an initialisation for a new node. As you can see in the example code below, this dictionary should contain two keys: `neighbours` and `states`. The `neighbors` key should have a list for the corresponding value, where each entry in the list should be the name of a neighbour. This way, new nodes can be immediately connected to other nodes when they are added to the network. The `states` key contains a dictionary similar to `set_initial_state`: keys for the name of each state

and the corresponding value for its initial value. In this way, nodes can be added to the network with their connections and states defined.

Finally, it is possible to change the edges of individual nodes by using the `edge_change` key in the `add_network_update()` function. Its value should be a dictionary where each key corresponds to the name of the node whose connections in the network are to be updated. Then, for each node in the dictionary, another dictionary with 3 possible keys can be created: `add`, `remove` and `overwrite`. This allows the user to add or remove connections, or to overwrite the entire set of neighbours with new ones. All three keys should have lists as values, where each list contains identifiers of the nodes to add, remove or overwrite edges with. If a weight is to be added to a particular edge, its value should be a list containing the node name, the incoming weight value and the outgoing weight value. An example of a network update function is shown below. Note that when we make manual changes to the network we do not use `UtilityCostModel` but `Model`.

```
model = Model(g, seed = seed, config = ModelConfiguration(cfg))

def change_network():
    changes = {
        'add': [{
            'neighbors': [1, 2],
            'states': {
                'phase': 1,
                'toi': 0
            }
        }],
        'remove': [5],
        'edge_change': {
            0: {
                'add': [1, 2],
                'remove': [3]
            },
            4: {
                'overwrite': [3, 6]
            }
        }
    }

    return changes
model.add_network_update(change_network)
```

3.6 Related software

DyNSimF has the novel ability to handle dynamic networks as well as complex internal dynamics with multiple continuous states. However, there are several other resources available to facilitate the modelling and simulation of complex network-based models in Python. This section discusses the capabilities of these other available solutions and how DyNSimF differs from the others, using the table 3.1.

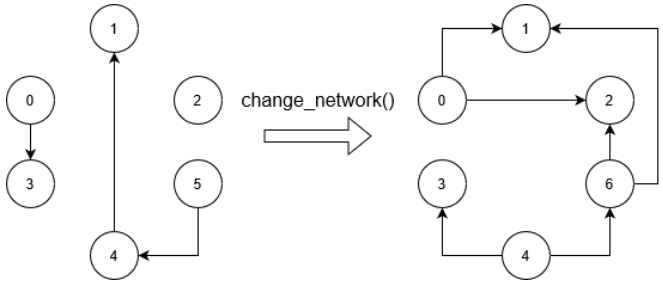

Figure 3.4: An example update of a network using the manual update function shown in 3.5.

NDlib (Rossetti et al., 2018) is the package most similar to DyNSimF. NDlib was created to solve major problems in other packages, such as Nepidemix (Ahrenberg et al., 2016) (Python) and EpiModel (Jenness et al., 2018) (R). While these packages are capable of incorporating custom network spreading models, they suffer from steep learning curves and have limited implementation of newer models, as they focus only on SIR-type models. NDlib also includes a visual interface, making the package even more accessible to a wider audience. DyNSimF offers internal dynamics with continuous states that directly affect the local network, directional edges as well as parellisation capabilities over NDlib. In addition, DyNSimF provides the *utility-cost* methodology for managing the dynamics of the network structure.

Epidemics on Networks (EoN) (Miller and Ting, 2020) is based on the book 'Mathematics of Epidemics on Networks: from Exact to Approximate Models' (Kiss et al., 2017). The package provides tools to reproduce models from this book, with less focus on custom complex models. What it does offer over other network diffusion packages are many out-of-the-box models, various ODE solvers, stochastic dynamics, and advanced visualisations including animations. It also includes analysis tools, although these are often specific to the models included.

Epydemic (Dobson, 2020) aims to be the 'common platform' for epidemic simulation. While it allows for custom compartmentalised models, it focuses mainly on scalability and efficiency for conventional SIR-type models.

HybridModels (Marques et al., 2020), an R Team (2021) package, uses a hybrid method of dynamic network and population simulation. While DyNSimF models each agent as a node, which HybridModels Marques et al. (2020) would describe as an individual-based model, this level of precision is not always necessary. For example, with the input of a dynamic motion network, hybridModels can handle subpopulations that migrate along edges in a network. It then performs the coupled differential equations on the populations within each node.

Performance comparison

DyNSimF can facilitate the use of Numpy routines, significantly increasing the speed of simulations, as can be seen in Table 3.2. DyNSimF uses Numpy arrays under the hood,

	DyNSimF	NDlib	Nepidemix	EoN	epydemic	EpiModel	hybridModels
Int. Dyn.	√						
Dyn. Network	√	√	√		√	√	√
Stoch. Dyn.				√	√	√	√
Edge types	++	+		+			
Visualization	√	√	√	√		√	
Visual Itf.		√					

Table 3.1: Similar software compared. The first set of rows is about the internal capabilities while the second set of rows is about the tools provided to use and analyze models. 'Int. Dyn', or 'Internal Dynamics', is the capability of having continuous states of a node influence each other, making each agent more complex. 'Dyn. Network', or 'Dynamic Network', is the ability to add and remove nodes or edges. 'Stoch. Dyn.', or 'Stochastic Dynamics', is the ability to implement discrete event algorithms, such as Gillespie simulations. 'Visualization' indicates whether the package has tools to show the dynamic behavior. 'Visual Itf', or 'Visual Interface' represents a user-friendly, non-programming interface to interact with the package. 'Edge Types' shows how complex the edges of the networks can be. Every package has edges between nodes. However, + indicates that these edges can be weighted, and ++ that they can be weighted and directional.

and if the modeller also uses data in the form of arrays, Numpy routines can be used for them. Models are only as fast as their slowest component, so it is useful to ensure that every operation is performed on matrices. This means, for example, that instead of requesting the neighbours for each node, the adjacency matrix containing each neighbour for each node should be used. Below is an example of how the infection and recovery updates of the quarantine model can be done using only Numpy array computations.

```
def update_states_numpy(constants):
    # define variables
    SIR = model.get_state('phase')
    I = np.where(SIR == 1, 1, 0)
    adjacency = model.get_adjacency()
    random = np.random.rand(constants['n'], constants['n'])

    # infection calculation
    # list of nodes eligible to be infected with probability
        p
    eligible = I * adjacency * random
    # check whether infection event happened
    bool_to_infect = np.any(eligible, axis = 1, \
        where = eligible > (1 - constants['beta']))
    nodes_to_infect = np.where(bool_to_infect)[0]
    SIR[nodes_to_infect] = 1
```

```
        # recovery calculation
        infected_time = model.get_state('toi')
        # add time to infected toi
        infected_time[np.where(SIR == 1)[0]] += 1
        # if infected long enough, recover
        SIR[np.where(infected_time >
            constants['recovery_time'])] = 2

        return {'phase': SIR, 'toi': infected_time}
```

Table 3.2 shows execution times for simulating a SIR model on a network. The different packages all simulate a SIR model with a random geometric graph using a 0.75 distance threshold value for connecting nearby nodes. The simulations were run 10 times per library and the average execution times, as well as the standard deviations, are shown in the table for different numbers of nodes. The parameters used for the SIR model are $\beta = 0.001$ and $\gamma = 0.01$, while the initial fraction of infected nodes is 5%. All simulations were run on a Macbook Pro with an 8-core i9 processor and 16GB of RAM. The execution times only indicate the time spent on the 25 iterations; it does not take setup time into account. It can be seen how DyNSimF with Numpy outperforms other packages by an order of magnitude. It is noteworthy that these execution speeds were acquired from simulating a highly connected network, amplifying efficiency gains in dense network calculations. Further, the simulations are run with a basic implementation of the packages and are not manually optimized, e.g., epydemic is run without the use of epyc (Dobson, 2022).

Package	**Graph size (# nodes)**					
	10^2		10^3		10^4	
	mean	*std*	*mean*	*std*	*mean*	*std*
DyNSimF matrix	0.005	0.0028	0.18	0.005	37.5	11.13
DyNSimF for-loop	0.014	0.0034	3.59	0.060	482.4	51.26
NDlib	0.012	0.0019	1.32	0.029	139.7	9.90
EoN	0.007	0.0013	0.78	0.025	113.3	4.92
epydemic	0.051	0.0045	3.72	0.206	789.3	199.49

Table 3.2: Mean and standard deviations of the computation time for 10 runs of 25 iterations of a network SIR model for different packages and different network sizes. Every simulation was run using $\beta = 0.001$ and $\gamma = 0.01$. The initial fraction of infected nodes is 5%. The network used for the model is a random geometric graph with a 0.75 distance threshold value for connecting nearby nodes.

3.7 Conclusion

We have discussed the methodology of using DyNSimF to study complex agent networks and adaptive networks. DyNSimF, is an open source, flexible framework for studying custom complex network dynamics using Python. It is the first package to facilitate the

modelling of adaptive networks: studying both the dynamics *on* the network and *from* the network, as well as their complex interactions.

At a technical level, DyNSimF is based on NetworkX, whose core is familiar to many people in network science, making DyNSimF highly integrative and suitable for many existing analyses. It has been designed with a focus on consistency, readability and extensibility, and aims to improve usability over alternatives. Numpy's array-based routines can be used, resulting in very competitive modelling speeds that outperform most alternatives. DyNSimF includes customisable dynamic visualisations, particularly useful for early-stage modelling, and supports custom metrics for analysis.

DyNSimF is designed to take the back-end and programming work off the shoulders of modellers. With the only requirements being the network type and equations for the internal model of the nodes, we hope to make network modelling accessible to more researchers and encourage the application of existing models to a network. We hope that DyNSimF will allow novice modellers to focus less on programming and more on modelling, while still allowing experienced modellers to build complex, fast, custom models for a wide range of use cases. Our goal for DyNSimF is to become the new standard in complex agent modelling and adaptive network modelling.

DyNSimF is a package for Python, available on PyPI (pyp, 2022). Documentation is available on Readthedocs (dyn, 2023a) and source code is available from the GitHub repository (dyn, 2023b).