1 Examples

This ducument will show how DyNSimF can be used to implement several models from different literature. This section should provide some inspiration on how the package can be utilized to implement different kinds of models that have different requirements. The first example will be an implementation of the SIR model (Kermack and McKendrick 1927) which shows how a basic model can be implemented. The second example is about segregation within schools, which is a version heavily inspired by the model created by Mele (2013). It will show how utility and cost can be utilized by relying on edge values and manual network updates to do manual network formation. The GitHub[1] repository of DyNSimF can be visited to explore more examples such as the craving versus self-control model (Grasman, Grasman, and van der Maas 2016) or the hierarchical Ising opinion model (van der Maas, Dalege, and Waldorp 2020).

1.1 SIR

The original SIR model was introduced by Kermack (Kermack and McKendrick 1927), it shows how different nodes in a network can change their status from susceptible (S) to infected (I), and finally to removed (R). In this basic model, a fixed population is considered, with only the three compartments S, I, and R. The S compartment is used for individuals who have not been infected with the disease and are thus susceptible to it. The I compartment denotes the part of the population that is currently infected with the disease and are also capable of spreading it to other individuals. Finally, the R compartment is used for individuals who have recovered, and been removed from the disease. In the simple case this is only by recovery, but more advanced versions may also include a reason of death which removes a person from the network. The individuals who have recovered, are not able to transmit the disease or to be infected again. In general, an individual will start in the S compartment, then when the person is infected, he will move to the I compartment and finally to the R compartment.

There are different implementations of this model; the one originally used was one using ordinary differential equations. However, another version was introduced in Milli, Rossetti, Pedreschi, and Giannotti (2018), where the transitions between compartments are a stochastic process, where the disease diffuses through the network. As this version is more intuitive to show how the disease propagates through a network, this version will be implemented. To test the implementation, another version using ordinary differential equations will be implemented as well. The version of Milli *et al.* (2018) assumes that if, during a generic iteration, a susceptible node comes into contact with a node from the infected compartment, it will become infected with probability β. Every node in the I compartment has a probability γ to transition to the R compartment. The implementation for this model using DyNSimF is fairly straightforward. The first step is to initialize the network, model, constants, and initial states. It is helpful to follow the process from section **??** whenever a new model is being created. We first create a network with a thousand nodes ($N = 1000$) and initialize the model[2]. Then we set the constants where $\beta = 0.4$ and $\gamma = 0.04$, furthermore, we will initialize the network with three infected.

[1] `https://github.com/Tensaiz/DyNSimF`

[2] Note that the model could also have been designed as a 1 node network, where the processes take place internally. In this case the node could represent a country in an international epidemic model for example.

```
# Network definition
n = 1000
g = nx.random_geometric_graph(n, 0.05)

model = Model(g)

constants = {
    'n': n,
    'beta': 0.4,
    'gamma': 0.04,
    'init_infected': 3
}
```

To model the compartments, each node will have a single state that represents its compartment. The susceptible nodes will have a state of 0, infected nodes' state will be 1, and the state of the recovered nodes is 2. We can initialize the state of the nodes by creating an array, where each index will match a node and each entry value will match the compartment the node will start in. First an array is created where every entry is 0, which would mean that the state of all nodes is susceptible. To set the state of some nodes to infected, Numpy is used to sample three random nodes from the network, and set their entries to 1, which will make them infected.

```
def initial_infected(constants):
    state = np.zeros(constants['n'])
    sampled_nodes = np.random.choice(np.arange(constants['n']),
                    constants['init_infected'], replace=False)
    state[sampled_nodes] = 1
    return state

initial_state = {
    'state': initial_infected
}

model.constants = constants
model.set_states(['state'])
model.set_initial_state(initial_state, {'constants': model.constants})
```

After finishing the configuration of the model and initializing the state of each node, we can write the actual update function that will be executed each iteration. The goal of this function will be to iterate over the susceptible neighbors of every infected node, draw a random sample, and then change the state of the neighbor to infected if the random value is lower than β. After this, for every infected node, a random sample is drawn and for each sample that is less than γ, the corresponding infected node should have its state updated to 2, so that it can become recovered. The code below shows how first the indices of the infected nodes are found using Numpy, afterwards the infected nodes are iterated over and a helper function from section **??** is used to iterate over the neighbors of each infected node. As can be seen, the state is updated to infected by changing the entry of the neighbors to 1 if the sample is less

than β. Finally, for every infected node, a random sample is drawn. Then, leveraging Numpy, a boolean array is created by comparing each sample to γ. By multiplying the boolean array by 2, all the entries where $sample < \gamma$ become 2, while the other entries become 0. Because these are all infected nodes, the entries with 0 are changed to 1, so that the non-recovered infected nodes remain infected.

```
def update_state(constants):
    state = model.get_state('state')

    infected_indices = np.where(state == 1)[0]

    # Update infected neighbors to infect neighbors
    for infected in infected_indices:
        nbs = model.get_neighbors(infected)
        for nb in nbs:
            if state[nb] == 0 and \
                np.random.random_sample() < constants['beta']:
                state[nb] = 1

    # Update infected to recovered
    recovery_chances = np.random.random_sample(len(infected_indices))
    new_states = (recovery_chances < constants['gamma']) * 2
    new_states[new_states == 0] = 1
    state[infected_indices] = new_states

    return {'state': state}
```

Finally, the update function should be added to the model, completing the process. At this point, the model can be simulated, visualized, or analyzed. A code example of a simulation of 100 iterations and a corresponding visualization is shown below:

```
model.add_update(update_state, {'constants': model.constants})
output = model.simulate(100)

visualization_config = {
    'initial_positions': nx.get_node_attributes(g, 'pos'),
    'plot_interval': 2,
    'plot_variable': 'state',
    'color_scale': 'brg',
    'variable_limits': {
        'state': [0, 2],
    },
    'show_plot': True,
    'plot_title': 'SIR probabilistic model',
}
```

```
model.configure_visualization(visualization_config, its)
model.visualize('animation')
```

By running all of the code from this example the model can be simulated and visualized. If the Numpy and Python random seeds are set to 0, a snapshot of the visualization at iteration 0 and 14 can be seen in figure 1. From the figure, it can be seen how three initial infected nodes influence the whole network and how the disease propagates. Another plot that is often shown, is the amount of nodes per compartment for every iteration. Figure 2 shows how the amount of susceptible nodes gradually decreases, while the amount of recovered start increasing. The amount of infected nodes increases at first, then gradually starts to decrease as more and more infected nodes start to recover, while there are less susceptible nodes remaining. This figure also shows the output for the other implementation of the SIR model, which uses ordinary differential equations (ODE). The remainder of this subsection will demonstrate how SIR can be implemented using these equations in DyNSimF.

The model using the ODEs was first described in Kermack and McKendrick (1927) and uses the same parameters β and γ, albeit with slightly different meanings. β describes the contact rate of the disease. An infected node will contact βN other nodes per unit of time. From the contacted nodes, the fraction that is susceptible to contacting the disease is $\frac{S}{N}$. γ is the mean recovery rate, which means that $\frac{1}{\gamma}$ is the mean period of time that infected nodes spread their disease. The differential equations describing the model are shown in equation 1, they describe the rate of change for each compartment.

$$\frac{dS}{dt} = -\frac{\beta SI}{N}, \frac{dI}{dt} = \frac{\beta SI}{N} - \gamma I, \frac{dR}{dt} = \gamma I \tag{1}$$

Because the ordinary differential equations on their own can be solved numerically or analytically to obtain the amount of individuals for each compartment, it can not be used explicitly to model the diffusion of a disease through a network, as the equations are not based on an actual network. However, because differential equations are often used in different kinds of models, an example will be shown of how the equations from equation 1 can be implemented as three different states for one node. This means that instead of modelling the dynamics of a disease spreading through a network, the internal dynamics of a node will be modelled. This is done by creating a network with 1 node and three states: S, I, and R. The values for the internal state of this single node, will show the amount of nodes for each compartment. In this example, the differential equations could represent internal processes of different nodes, where nodes could represent countries in meta-population model with a network for example. The first steps remain the same, handle imports, create a network, create the constants, and initialize the states, as shown below:

```
import networkx as nx
import numpy as np

g = nx.random_geometric_graph(1, 1)
model = Model(g)
init_infected = 3
initial_state = {
    'S': 1000 - init_infected,
```

```
    'I': init_infected,
    'R': 0,
}
constants = {
    'N': 1000,
    'beta': 0.4,
    'gamma': 0.04,
    'dt': 0.01,
}
model.set_states(['S', 'I', 'R'])
model.set_initial_state(initial_state)
```

In this case, the single node in the network has three internal states, that are each a single continuous variable. The other difference with the previous network implementation is that there is an extra constant, 'dt', which describes the amount of change per iteration, or step size. In this case every iteration should be a time step of 0.1. The next steps are like before, to create an update function that will update the network or the states. In this case, the internal state of the single node is updated each iteration. This is done by numerically integrating the ODEs using the Euler method (Euler 1845). First, the ODEs from equation 1 are written as a separate function for convenience:

```
def deriv(S, I, constants):
    N = constants['N']
    beta = constants['beta']
    gamma = constants['gamma']

    dSdt = -beta * S * I / N
    dIdt = beta * S * I / N - gamma * I
    dRdt = gamma * I
    return dSdt, dIdt, dRdt
```

This function will be used in the actual update function, where the states of the node are updated every iteration by integrating over the ODEs. The actual update function retrieves all the states, which is in this case a single continuous value, as there is only one node. Then the derivatives are calculated and multiplied by the step size, which are then added to the states of the current iteration. This function can then be added to the model in the same manner. By simulating the model 10000 times, the actual amount of iterations would be 100, due to the step size $dt = 0.01$. After simulating this, each iteration will have a node with three internal states, each state representing the amount of individuals in their respective compartment. When the internal states are plotted, they show the same result as shown in figure 2. As can be seen, the same general behavior is shown when using the ODEs as compared to the network-diffusion version. The reason they have differences is due to the stochastic processes involved with the network version, where there is a random initialization of initial infected individuals, network creation and structure, and chance to move from compartment to compartment. The ODE version of the SIR model makes the assumption that an underlying network is fully connected. The rest of the code to run this simulation is shown below:

```
def update(constants):
    dt = constants['dt']
    S = model.get_state('S')
    I = model.get_state('I')
    R = model.get_state('R')

    dSdt, dIdt, dRdt = deriv(S, I, constants)

    return {
        'S': S + (dt * dSdt),
        'I': I + (dt * dIdt),
        'R': R + (dt * dRdt)
    }
model.add_update(update, {'constants': constants})
its = model.simulate(10000)
```

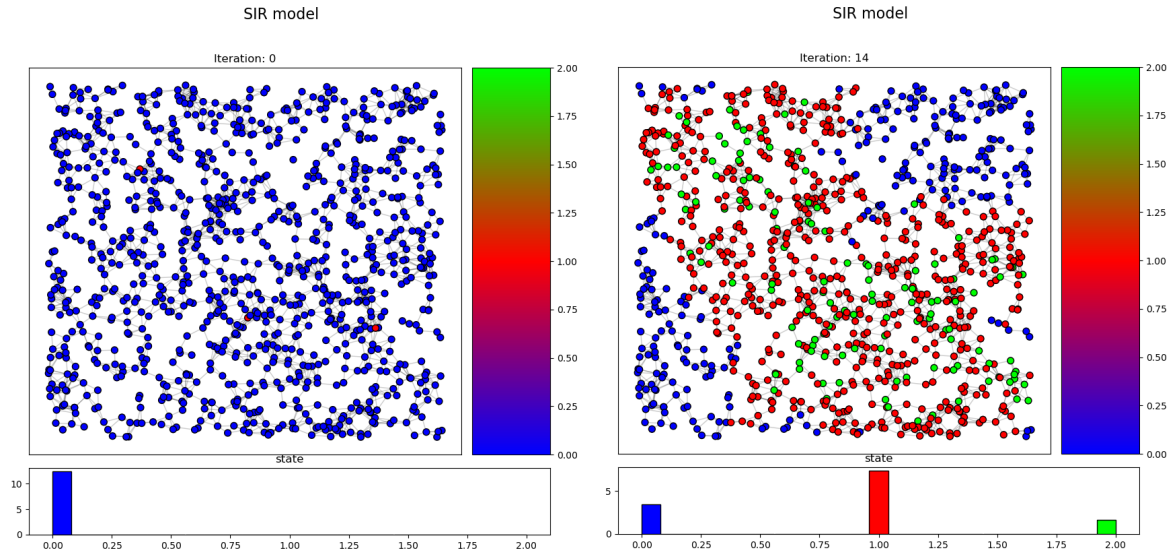

Figure 1: A snapshot of the visualization of the probabilistic SIR model. $N = 1000, \beta = 0.4$ and $\gamma = 0.04$. The random seeds are set to 0. It shows the propagation of the disease throughout the middle of the network towards the edges. The left figure shows the network at iteration 0 with 3 infected nodes. The right figure at iteration 14 shows how many nodes have become infected while some have recovered.

1.2 School Segregation

The example discussed in this subsection is a variant based on the model discussed in Mele (2013), this was chosen to demonstrate how to implement a model that relies on manual network updates using edge values, i.e. the weights between links. This agent-based model uses nodes to represent agents in a network formation game, while the network topology functions as the environment of the model. By simulating a network formation model, that is partially

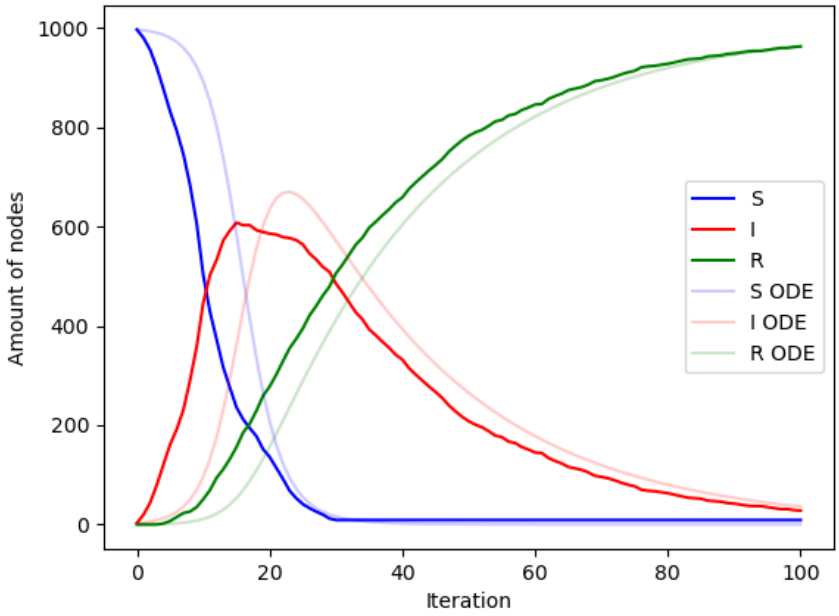

Figure 2: The amount of nodes per compartment per iteration. $N = 1000, \beta = 0.4$ and $\gamma = 0.04$. The random seeds are set to 0. It shows a comparison of the stochastic SIR network model and the implementation using ODEs. As can be seen, even though the stochasticity has impacted the model, the same behavior is shown by both models. The lines with less opacity show the behavior of the model that uses ODEs.

based on the network formation functionality described in section **??**, network structures can be generated that resemble real-life scenarios. These simulations may help with studying effects of policy interventions for example. The model that will be implemented has heterogeneous autonomous decision-making agents, where each agent has different characteristics, resembling real social networks. Each agent will sequentially revise their social connections inside the network to maximize their own utility based on their own and other agents' attributes. The network is directed, which means that if an agent i forms a connection with agent j, then agent j does not necessarily have a connection with agent i. A combination of strategic and random network formation is combined by selecting a random agent i who meets another random agent j based on a similarity probability distribution, as shown in equation 2. Where the meeting m between agents i and j is represented at time t as $m^t = i, j$ and $p_{min} > 0$ such that all meetings are possible. The probability for meetings is unbiased. Agent i now changes his connection with agent j in a meeting $m^t = i, j$ by either creating or removing a connection and thus updating g_{ij}. The choice of forming or removing a link is made based on increasing the utility based on the characteristics of the agents. Each agent has three individual attributes: sex, race, and grade. The data used as input was taken from Add Health Wave I survey (Harris and Udry 2015)[3]. For simplicity, every agent has full information on another agent that is selected for the meeting. The order of agents per iter-

[3] `https://drive.google.com/drive/folders/1CjBQYX9WXxwgdwVkgpXEOPQyrvQGapJW`

ation is determined randomly and every agent has a chance per iteration to form or remove a connection. This is different from the implementation of Mele (2013) for the sake of faster computation.

$$Pr(m^t = ij|X) = \frac{u(X_i, X_j) + p_{min}}{\sum_{i,j} u(X_i, X_j) + p_{min}} \tag{2}$$

As mentioned before, agents try to maximize their utility every iteration by forming or removing links with other agents. The utility an agent receives from forming a connection with another agent is defined in equation 3. It shows the utility of an agent i for another agent j from a network g with population attributes $X = (X_1, ..., X_n)$. The utility stems from direct friendship, reciprocated friendship, and being indirectly linked to friends of a friend. To form and maintain a link, a certain cost is required, which prevents agents from simply forming connections with every other agent in the network to get maximum utility. The total net utility U of agent i is the sum of the net utilities received from equation 3.

$$U_i(g, X) = \underbrace{\sum_{j=1}^{n} g_{ij} u_{ij}}_{\text{direct links}} + \underbrace{\gamma \sum_{j=1}^{n} g_{ij} g_{ji} u_{ij}}_{\text{mutual links}} + \underbrace{\delta \sum_{j=1}^{n} g_{ij} \sum_{k=1, k\neq i,j}^{n} g_{jk} u_{ik}}_{\text{indirect links}} - \underbrace{d_i^2 c}_{\text{cost}} \tag{3}$$

$$\epsilon \overset{i.i.d}{\sim} N(0, \sigma^2) \tag{4}$$

Where $u_{ij} = u(X_i, X_j) = exp(-b|X_i - X_j|)$ and can be seen as a similarity score. u_{ij} is normalized to 1 if agents have complete identical attributes. The cost $d_i^2 c$ consists of the outdegree d_i of the agent and a constant cost $c \in (0, 1)$. $\delta \in (0, 1)$ is the weight or importance of indirect links and $\gamma > 0$ is the weight or importance for connections that are mutual between agents. Before agent i updates his link with agent j, agent i receives an idiosyncratic shock ϵ to its preferences, which is used to model unobservable events that influence utility, which is shown in equation 4. ϵ is assumed to be a Type I extreme value i.i.d. among links and across time. This shock causes agents to sometimes create less optimal connections, adding stochasticity to the model and may prevent it from reaching a local minimum as well. A link is formed from agent i to agent j if and only if:

$$U_i(g_{ij}^t = 1, g_{\_ij}^{t-1}, X) + \epsilon_1^t > U_i(g_{ij}^t = 0, g_{\_ij}^{t-1}, X) + \epsilon_0^t \tag{5}$$

and agent i will delete or withhold from creating a link if:

$$U_i(g_{ij}^t = 1, g_{\_ij}^{t-1}, X) + \epsilon_1^t < U_i(g_{ij}^t = 0, g_{\_ij}^{t-1}, X) + \epsilon_0^t \tag{6}$$

The total utility of the network g given the population X is the sum of each agent's utility:

$$Q(g, X) = \sum_{i}^{n} U_i(g, X) \tag{7}$$

To implement this model using DyNSimF, first the model should be configured to store adjacency and edge values using the MemoryConfiguration class. Afterwards the network can be created. In this example, the network is randomly generated using Numpy to create

a network with an average degree of 5. Agents are connected to other agents with a uniform probability of $p_0 = \frac{d_0}{n}$ where $d_0 = 5$. The following constants are set: $\delta = 0.05$, $\gamma = 0.65$, $c = 0.35$, $B1 = 0.1$, $B2 = 0.1$, $B3 = 0.2$, $min\_prop = 1000$, $\sigma = 0.035$. The B parameters influence how much different traits impact utility. The constant 'X' shown in the model was taken from Harris and Udry (2015) to get the traits of students from different schools. The first step is to create a matrix containing initial utility values based on the connections in the network, as well as an initial probability matrix containing the probabilities that agents might meet based on their similar attributes.

```
def initial_utility():
    utility = np.zeros((constants['n'], constants['n']))
    race = list(constants['X']['race'])
    sex = list(constants['X']['sex'])
    grade = list(constants['X']['grade'])

    for i in range(constants['n']):
        for j in range(constants['n']):
            weighted_diffs = \
                [constants['B1']*abs(sex[i] - sex[j]),
                constants['B2'] * (0 if grade[i] == grade[j] else 1),
                constants['B3'] * (0 if race[i] == race[j] else 1)]
            utility[i, j] = math.exp(-sum(weighted_diffs))
    return utility

def initial_prop():
    prop = np.zeros((constants['n'], constants['n']))
    utility = initial_utility()
    # Loop over the person and their peers
    for i in range(constants['n']):
        for j in range(constants['n']):
            if i == j:
                prop[i, j] = 0
            else:
                prop[i, j] = utility[i, j] + constants['min_prop']
        # Normalize
        prop[i, :] = prop[i, :] / np.sum(prop[i, :])
    return prop

constants['probability'] = initial_prop()
constants['utility'] = initial_utility()
```

The functions shown above are executed and saved as constants which are later utilized by the model. Then, a function is written to directly calculate the utility of one node based on equation 3. This function is called when the utility of any node should be calculated. It looks at the direct links, indirect links, and mutual links to determine the net utility. The input of the function is an individual node, as well as the adjacency matrix of the network at the current time step.

```
def node_utility(node, adj):
    utility = constants['utility']
    # degree, connection gain and cost calculations
    d_i = adj[node].sum()
    direct_u = np.sum(adj[node] * utility[node])
    mutual_u = np.sum(adj[node] * adj.T[node] * utility[node])
    # indirect connection gain
    a = (adj.T.dot(adj[node, :]) * utility)[node]
    a[node] = 0
    indirect_u = np.sum(a)

    return direct_u + constants['gamma'] * mutual_u + \
        constants['delta'] * indirect_u - d_i ** constants['c']
```

Finally, the last function that manually updates the network based on the meetings can be written. It randomly shuffles the order of the agents and then starts the meeting process for each agent based on the similarity between agents as shown in equation 2. For each agent, the utility with and without the connection is compared. If the utility with the connection is higher, then as explained in section **??**, the connection is added and otherwise removed. When the constants have been added to the model, as well as the network update function using model.add$_{n}etwork_{u}pdate(network_{u}pdate, get_{n}odes = True), themodelcanbesimulatedandanalyzed.$

```
def network_update(nodes):
    adj = model.get_adjacency()
    order = nodes.copy()
    eps = np.random.normal(scale=constants['sigma'], size=constants['n']*2)
    np.random.shuffle(order)
    changes = {}
    P = constants['probability']
    for node in order:
        other_node = node
        while other_node == node:
            other_node = np.random.choice(nodes, p=P[node])
        existing_connection = not not adj[node, other_node]
        adj[node, other_node] = 0
        U_without = node_utility(node, adj) + eps[node]
        adj[node, other_node] = 1
        U_with = node_utility(node, adj) + eps[-node]

        if U_without > U_with and existing_connection:
            changes[node] = {'remove': [other_node]}
        elif U_without < U_with and not existing_connection:
            changes[node] = {'add': [other_node]}

    return {
        'edge_change': changes
    }
```

References

Euler L (1845). *Institutionum calculi integralis*, volume 4. impensis Academiae imperialis scientiarum.

Grasman J, Grasman RP, van der Maas HL (2016). "The dynamics of addiction: Craving versus self-control." *PLoS One*, **11**(6), e0158323.

Harris KM, Udry RJ (2015). "National Longitudinal Study of Adolescent to Adult Health (Add Health) Wave I, 1994-1995." *a href=" http://dx. doi. org/10.15139 S*, **3**.

Kermack WO, McKendrick AG (1927). "A contribution to the mathematical theory of epidemics." *Proceedings of the royal society of london. Series A, Containing papers of a mathematical and physical character*, **115**(772), 700–721.

Mele A (2013). "A structural model of segregation in social networks." *Available at SSRN 2294957*.

Milli L, Rossetti G, Pedreschi D, Giannotti F (2018). "Diffusive phenomena in dynamic networks: a data-driven study." In *International Workshop on Complex Networks*, pp. 151–159. Springer.

van der Maas HL, Dalege J, Waldorp L (2020). "The polarization within and across individuals: the hierarchical Ising opinion model." *Journal of Complex Networks*, **8**(2), cnaa010.