1 Software Input

Upon cloning the repository, you will notice there are files called "INFILE.yaml" and "DEFAULT.yaml"; these files allow one to adjust certain parameters to obtain desired results from the simulation, and all of the different parameters is described below.

Quick link variable descriptions:

analysis: (type: n/a) This parameter should *never* be adjusted, when the software looks for the parameters listed under the analysis heading, it first looks to this variable. This defines to so called analysis section of the input file.

main menu

initial_entities: (type: integer) This parameter is the number of initial entities in the simulation. The entity type and characterizations are determined before any time will pass in the simulation.

main menu

max_entities: (type: integer) This parameter is the maximum number of entities (users) that will be allowed in the simulation. Once the maximum number of users has been reached in the simulation, the program will no longer add users but will not terminate.

main menu

max_time: (type: integer) This parameter is the maximum time allowed in the simulation. The units of time in the simulation is minutes. The simulation will terminate once the maximum amount of time has been reached. Conveniently, you can put 'minute', 'hour', 'day', or 'year', and the simulation will know the value in minutes.

main menu

max_real_time: (type: integer) This parameter is the maximum time allowed in real life. Similarly to the max_time parameter, the simulation will terminate once the maximum amount of time has been reached. Also, you can put 'minute', 'hour', 'day', or 'year', and the simulation will know the value in minutes.

main menu

enable_interactive_mode: (type: boolean) While the simulation is running, you can press 'ctrl-c'. If this variable is set to true, an interpreter will be activated and you can call functions in the interpreter. If you press the 'tab' key, you can go through the built in functions. The interpreter was designed to allow one to manipulate the network. If you are in the interpreter, you can press 'ctrl-c' and the simulation will resume. If you type 'exit()' while in the interpreter the simulation will terminate.

main menu

use_barabasi: (type: boolean) If true, a certain simulation method will be implemented that results in a network that models a scale-free network with a scale-free exponent of 3. This method is very similar to the 'barabasi.game' in the igraph package found in R or Python. The follow model (follow_model) in the simulation should be set to 'preferential' to achieve desired result. The simulation method is as follows; an entity is added into the network, and then follows another entity based on the degree of the entities in the current network. A higher degree results in a higher probability of being followed. If set to false, the simulation will run normally.

main menu

use_random_time_increment: (type: boolean) If true, the time in the simulation will be incremented at a non-constant rate; the increment of time is determined by

$$\Delta t = \frac{-\ln(u)}{R_{\text{tot}}}$$

where u is a random number in the interval $0 < u \leq 1$, and R_{tot} is the sum of the rates for the simulation. On average, the value of $-\ln(u)$ is unity, and therefore you can increment time by $1/R_{\text{tot}}$; this is how time is incremented if set to false.

main menu

use_followback: (type: boolean) If set to true, there may be a follow back prior to a follow. When an entity A follows entity B, if entity B's followback_probability (which can be found in the entities section of the input file) is non-zero, then entity B can follow entity A without moving forward in time. This is why it is called a 'flawed followback'; in reality there would be some time prior to the initial follow where the followback would occur.

main menu

follow_model: (type: string) Currently, there are 6 follow methods implemented in the software; 'random', 'entity', 'preferential', 'preferential-entity', 'hashtag', and 'twitter'; to use a follow model simply do

follow_model: random

The 'random' follow method causes entities to follow other entities randomly; if the number of entities is set to constant and this method is on, you will achieve an Erdös-Rényi degree distribution (Poisson distribution).

The 'entity' follow method causes entities to follow other entities based on title alone; the titles are set in the entities section of the input file. The probabilities to to follow each entity can also be set in the weights portion of the entities section. The probabilities are normalized, so the probabilities should be set with respect to one another.

The 'preferential' follow method follows the preferential attachment model outlined by Barabási and Albert. The basics behind the method is that the degree of the entity determines the probability of following the entity; the greater the degree the greater the probability of the entity being followed. The thresholds and weights variables outlined in the follow_ranks section determines how the degree of the entities are binned and the corresponding weights for each bin. If you want to achieve the preferential attachment method similar to the use_barabasi method you can set the thresholds to increment by 1, with linear spacing from 0 to the max number of entities and set the weights to increment by 1 with linear spacing from 1 to the max number of entities.

The 'preferential-entity' follow method is just the 'preferential' follow method nested in the 'entity' follow method. Firstly, a certain entity type will be selected, then

based on the degrees of the entities within the certain entity type, an entity will be selected to follow.

The 'hashtag' follow method is a mechanism introduced to follow other entities based on hashtags. If the use_hashtag_probability parameter is non-zero, then entities will attach hashtags to their tweets. Depending on the ideology and location of the entity, they are then placed into a dynamic array. If another user wants to follow via hashtag, they look to these dynamic arrays to find a specific entity that relates to their hashtag preferences. These preferences can be set in the hashtag_follow_options section.

The 'twitter' follow model is a model that incorporates all of the above follow mechanisms. The weights for each mechanism can be set for each follow method. See model_weights for more information.

main menu

model_weights: (type: n/a) In the model_weights, you can set the probability of calling each follow method. You can use arbitrary units to set the weighting for each follow method because the weights are summed and then normalized to find the probability of calling each method. As an example, if you would like to call the follow methods equally you would set it as so:

model_weights: {random: 0.20, preferential: 0.20, entity: 0.20, preferential_entity: 0.20, hashtag: 0.20}

main menu

stage1_unfollow: (type: boolean) If set to true, entities can be flagged when followed based on their tweet rates. If the tweet rate of the newly followed entity is greater than twice the average tweet rate of the pre-existent entities you follow, than the entity is placed into an array. This array is looked to when the unfollow function is called. This algorithm is supposed to encapsulate an unfollow method on Twitter. In Twitter if your feed is being dominated by a user you may become annoyed and want to unfollow that user. This is the process that occurs in this unfollow method.

main menu

unfollow_tweet_rate: (type: double) This tweet rate is not associated with the stage1_unfollow method described above. This is a more simplified unfollow algorithm and can be considered the stage 0 unfollow method. When an entity tweets, we look to see how many tweets they have put out since the entity was created. If the tweet rate of the entity exceeds the value set here, then the entity will be randomly unfollowed by one of its followers.

main menu

use_hashtag_probability: (type: double) This is a probability (0.0-1.0) that determines how often entities attach hashtags into their tweets. If set to 0.0, then none of the tweets will have hashtags. If set to 0.5, then half of the tweets will have hashtags. If 1.0, then all of the

tweets will have hashtags.

main menu

rates: (type: n/a) This parameter does not need to be changed nor removed. Used to point to the global add rate for the simulation. This defines the global rates part of the input file.

add: (type: n/a) This parameter allows one to adjust the rate at which users are being added into the network. To have different numbers of entities, you can adjust the add weights in the entities section of the input file.

function: (type: string) The function for the add rate can either be constant or linear; other options can be coded into the software if necessary. If function is set to 'constant', then you must also declare the constant value by setting the 'value' parameter to a real constant > 0. An example of of constant add rate is as follows;

add: {function: constant, value: 1.0}

This will set the add rate to 1 entity per minute. If function is set to 'linear', then you must set 2 other variables, the 'y_intercept' and 'slope' variables. An example of a linearly increasing add rate is as follows;

add: {function: linear, y_intercept: 1.0, slope: 0.1}

This will create an add user rate of the form

$$R_{\text{add}} = 1.0 + \text{n\_months} \cdot 0.1$$

where n_months is the number of months that have passed in the simulation. The number of months is determined by a constant approximate month of $30 \cdot 60 \cdot 24$ minutes. This constant value can be changed in the software to achieve desired result.

main menu

output: (type: n/a) This parameter should not be removed, when the simulation has completed, functions that do further analysis on the network can be controlled here, a few built in functions are shown below. This defines the output portion of the input file.

main menu

save_network_on_timeout: (type: boolean) If set to true, all of the information of the network is sent into the file declared for the save_file when the simulation is terminated.

main menu

load_network_on_startup: (type: boolean) If set to true, all of the information of the network that was sent into the file declared for the save_file will be loaded from when the simulation begins.

main menu

ignore_load_config_check: (type: boolean) If you try to load a network from a

save_file that does not have the same parameters as the input file and this parameter is set to false, then you will receive a message on the screen and the simulation will not start. This message will tell you that you must set this parameter to true. Once you do this you can load the saved network and continue simulating that network with different parameters. An example is to generate a random graph and save it. Then you can load this network with different parameters generating a different network.

main menu

save_file: (type: string) This is the file where all of the information of the network will be stored if save_network_on_timeout is set to true. The file name can be anything you like. Also if load_network_on_startup is set to true, the simulation will look for this file to load the existing network information.

main menu

stdout_basic: (type: boolean) If set to true, the number of months in the simulation will be outputted to the screen. Once all of the analysis in complete in the simulation, a message will also be printed to the screen.

main menu

stdout_summary: (type: boolean) If set to true, the amount of time, number of entities, number of follows, number of tweets, number of retweets, total rate, and real time spent between each successive output is outputted to the screen.

main menu

visualize: (type: boolean) If set to true, the information from the network is outputted to 2 files, "network.dat" and "network.gexf". The "network.dat" file consists of two columns; the first column is a list the entity ID's in order, and the second column is the entity ID's for who the first column is following. For example if entity ID 1 is following entity ID 13, 14, and 19, then it would be outputted like:

1	13
1	14
1	19

The structure of this file can be referred to as an edge list. This file can easily be read into Python or R to perform any further analysis. The "network.gexf" file is an xml file that can be used to visualize the network in a program called Gephi.

main menu

entity_stats: (type: boolean) If set to true, additional calculations will be done to find the relationships between different entity groups and degree distributions for each entity. For every entity set in the entities section of the input file, there will be a file created with the information mentioned previously. The output files created will take the form "name_info.dat" where the name is the name created for the entity.

main menu

degree_distributions: (type: boolean) If set to true, the cumulative, in-degree, and out-degree distributions will be outputted to "cumulative_distribution.dat", "in-degree_distribution.dat", and "out-degree_distribution.dat". The data in the files can be easily plotted with Gnuplot.

main menu

tweet_analysis: (type: boolean) If set to true, further analysis will be done to create a distribution similar to a degree distribution. The tweet and retweet distributions come from how often the entities tweet in the network; they tell you the probability that a given entity would have tweeted or retweeted n times, and n is first column in the data files produced. The files created are titled "tweet_distro.dat" and "retweets_distro.dat."

main menu

retweet_visualization: (type: boolean) If set to true, there will be a Gephi file called "retweet_viz.gexf" that is produced after the simulation terminates. This file can be found in the 'output' directory. The graph produced is a visualization of the most retweeted tweet in the network.

main menu

main_statistics: (type: boolean) If set to true, there will be a text file entitled "main_stats.dat" that is generated after the simulation has finished. This file can be found in the 'output' directory. It provides some general statistics for the simulation performed.

main menu

tweet_ranks: (type: n/a) This variable must stay in the input file, it allows the code to determine the thresholds for the tweets in the simulation. The entities in the simulation will be categorized in the thresholds provided. For example if a threshold is 10, then the entities with 10 or less tweets will be stored an a list. Once a entity in this list tweets more than 10 times, the entity will be moved into a bin with a higher threshold.

main menu

retweet_ranks: (type: n/a) This has the same functionality as the tweet ranks, except it is for retweets. See above description about tweet ranks for more information.

main menu

follow_ranks: (type: n/a) This has the same functionality as the tweet and retweet ranks, except it is for the number of followers an entity has. The major difference between the follow ranks and the tweet/retweet ranks is that the thresholds have weights associated with them for follow ranks. If you adjust these weights correctly, you can achieve the preferential attachment

model. The descriptions for how you can set your thresholds for tweets, retweets, and follows, as well as how to set the weights for follows is below.

thresholds: (type: n/a) The thresholds parameter always take 4 variables; bin_spacing, min, max, and increment.

bin_spacing: (type: string) The strings that can be used for the bin_spacing variable are linear, quadratic, and cubic; other bin spaces can be implemented if necessary. If linear, the bins for grouping entities will be spaced based on the increment, if quadratic, the bins for grouping entities will be spaced based on the increment squared, and if cubic, the bins for grouping entities will be spaced based on the increment cubed.

min: (type: integer) This variable sets the minimum threshold for grouping entities. If the entities have a value less than the minimum threshold, then no binning will occur.

max: (type: integer) This variable sets the maximum threshold for grouping entities. If the entities exceed the maximum value, they will be grouped in a bin where the threshold can not be exceeded.

increment: (type: integer) This variable sets the separation between adjacent thresholds for grouping users. If bin_spacing is linear, then the separation between bins is the increment, if quadratic, the separation between bins is the increment squared, if cubic, the separation between bins is the increment cubed.

An example of linear thresholds is

thresholds: {bin_spacing: linear, min: 10, max: 100, increment: 10}

This will create thresholds 10, 20, 30,..., 100; the entities will be grouped according to these thresholds.

main menu

weights: (type: n/a) This follows the same procedure as thresholds. Instead of defining the thresholds for grouping entities, it defines the probability of selecting one of the groups of entities.

main menu

tweet_observation: (type: n/a) This denotes the section of the input file that provides the decay function for the retweet rates. Experimentally, the retweet rate for a tweet decays over time by approximately $1/t$ where t is the time in minutes.

main menu

density_function: (type: n/a) This density function is the function that describes how the retweet rates decay over time. Since the input file is intertwined with Python, you can actually declare mathematical functions with Python syntax as the density function. An example is '2.45 / (x)**1.1'. This function is integrated with scipy and each discrete element of this function is divided by the total integral to normalize the function. This ensures that it is truly a probability density function. The discrete elements of the function

can be adjusted by the parameters mentioned below. The entities that have the chance to be retweeted are binned according to the values of the function, and as time progresses they switch bins to have smaller value.

main menu

x_start: (type: double) This is the initial value of the density function. Make sure that the value of the density function evaluated at this starting value does not cause a discontinuity in the function.

main menu

x_end: (type: double) This is the final value of the density function. Make sure that the value of the density function evaluated at this end value does not cause a discontinuity in the function.

main menu

initial_resolution: (type: double) This is the initial value for determining where the integral of the density function is evaluated at. If the x_start value is 5 and this parameter is set to 1, then the first integral will be from 5 to 6.

main menu

resolution_growth_factor: (type: double) This value changes how the integrals are evaluated for the density function. Since the resolution of the function found experimentally needs to be more accurate initially then after some amount of time, we have introduced this parameter. For example if x_start is 5, the initial_resolution is 1, and this parameter is 1.5, then the integrals will be evaluated at [5, 6], [6, 8.5], [8.5, 13.5], etc.

main menu

time_span: (type: double) This value determines when the density function will disappear allowing no more retweets or follow by retweets to occur. Since the function found experimentally approaches 0 quite slowly, it is convenient to define a time span for the function to improve efficiency of the retweet algorithm. Like the max_time parameter you can use convenient strings like 'minute', 'hour', 'day', 'month', or 'year'. You can also use Python syntax and multiply numbers to these strings.

main menu

ideologies: (type: n/a) This section of the input file determines abstract characterizations of entities. The motivation behind it is political views on Twitter and how different or similar people based on political views would act on retweets or hashtags. Make sure that the number of ideologies declared here is the same as 'N_BIN_IDEOLOGIES' variable in the "config_static.h" header file. Once you declare ideologies you can then set the weights for each ideology in the regions section of the input file. You can also set the weights of a certain entity type tweeting about their ideology in the entities section of the input file.

main menu

naming_ideologies: (type: string) The names of your ideologies is completely up to you and the syntax for declaring ideologies is as such:

ideologies:
- name: Red
- name: Blue

You can see that I have set 2 ideologies, Red and Blue. The weights for each ideology must be set in the regions section of the input file.

main menu

regions: (type: n/a) This section of the input file is for declaring different regions where the entities in the network are 'from'. When an entity is created, where the entity is 'from' is selected randomly with weights that can be set. Here you can set the weights for many different parameters, all of which is explained below.

main menu

naming_regions: (type: string) The name of your region can be anything that consists of a string. The name is a necessary part of the region so you will get a warning if it is not set.

main menu

region_add_weight: (type: double) As mentioned above, when an entity is added into the network, we randomly generate where they are from. The weights set for each region declares the probability of an entity being from a specific region. The weights for all of the regions are summed and then divided by the sum to create a probability. The units for the weights of each region should be the same to normalize them correctly. The syntax for declaring the add weight is as such:

regions:
- name: Canada
add_weight: 5
- name: USA
add_weight: 10

From these weights, two thirds of the population will be from USA and one third from Canada. main menu

preference_class_weights: (type: n/a) After declaring preference classes which describe how retweets pass from entity to entity, you must set the weights for each preference class in the region section. For more information on preference classes, click here: preference_classes. Let's say you have defined preference classes 'Pref1' and 'Pref2'. The correct syntax for setting the weighting of these preference classes is as follows:

regions:

- name: Canada
add_weight: 5
preference_class_weights: {Pref1: 10, Pref2: 10}

Here the weights for each preference classes are also summed and divided by the sum to generate a probability. These probabilities are used when an entity is created to determine which preference class they use.

main menu

ideology_weights: (type: n/a) After declaring ideologies which define a characteristic for an entity, you must set the weights for each ideology in the region section. For more information on ideologies, click here: ideologies. Let's say you have defined ideologies 'Red' and 'Blue'. The correct syntax for setting the weighting of these ideologies is as follows:

regions:
- name: Canada
add_weight: 5
ideology_weights: {Red: 10, Blue: 10}

Here the weights for each ideology are also summed and divided by the sum to generate a probability. These probabilities are used when an entity is created to determine which ideology they are.

main menu

language_weights: (type: n/a) After declaring languages in 'config_static.h' which defines the language for the entity, you must set the weights for each language in the region section. Let's say you have defined languages 'English', 'French', and 'English+French'. The correct syntax for setting the weighting of these languages is as follows:

regions:
- name: Canada
add_weight: 5
language_weights: {English: 10, French: 10, English+French: 10}

Here the weights for each language are also summed and divided by the sum to generate a probability. These probabilities are used when an entity is created to determine which language they speak.

main menu

config_static: (type: n/a) There are constant values that can be found in 'config_static.h'. Some of the static values can be changed in the input file for simplicity. An example of this is the humour_bins described below.

main menu

humour_bins: (type: int) To determine how a retweet is passed based on humour, there are different values dedicated to each humour bin which describes how a tweet will be passed

from entity to entity based on humour. The value for each bin is what ever value is set in the preference_classes section for 'humourous' multiplied by the probability density function.

main menu

preference_classes: (type: n/a) This section of the input file is for the declaration of preference classes which describe how tweets are passed in the system. Here you must set the tweet_transmission for different scenarios based on different entity characteristics; all of which is explained below.

main menu

naming_preference_classes: (type: string) You can name your preference classes however you want. The names of your preference classes are needed in the regions section of the input file to determine the weights for an entity having a certain preference class. An example is as follows:

```
preference_classes:
  - name: Pref1
```

main menu

tweet_transmission: (type: n/a) Here you must set the transmission probabilities for different situations which can be found below. The transmission probabilities are then multiplied by the density_function which has been found experimentally to decrease over time.

main menu

plain_tweets: (type: n/a) In the entities section, there are different types of tweets that entities can tweet. One of the tweet types is 'plain' which is a generic tweet. When an entity tweets a plain tweet, the entities that follows the entity who tweeted may retweet the tweet depending on the tweet transmission value and density function. An example of the syntax for declaring a plain tweet preference class is:

```
preference_classes:
  - name: Pref1
tweet_transmission:
    plain:
  EntityType1: 0.1
  EntityType2: 0.1
    else: 0.1
```

As you can see from above, the entity types declared in the entities section are used in the preference classes. If you also declared an entity type 'EntityType3' then that entity type would fall under the 'else' set above. You can see that the transmission probabilities for all of the entity types are the same.

main menu

different_ideology: (type: n/a) In the entities section, there are different types of tweets that entities can tweet. One of the tweet types is 'ideological' which is a tweet related to an entities ideology. When an entity tweets a ideological tweet, the entities that follows the entity who tweeted may retweet the tweet depending on the tweet transmission value and density function. An example of the syntax for declaring a ideological tweet preference class is:

```
preference_classes:
- name: Pref1
tweet_transmission:
different_ideology:
EntityType1: 0.1
EntityType2: 0.1
else: 0.1
```

As you can see from above, the entity types declared in the entities section are used in the preference classes. If you also declared an entity type 'EntityType3' then that entity type would fall under the 'else' set above. You can see that the transmission probabilities for all of the entity types are the same.

main menu

same_ideology: (type: n/a) In the entities section, there are different types of tweets that entities can tweet. One of the tweet types is 'ideological' which is a tweet related to an entities ideology. When an entity tweets a ideological tweet, the entities that follows the entity who tweeted may retweet the tweet depending on the tweet transmission value and density function. An example of the syntax for declaring a ideological tweet preference class is:

```
preference_classes:
- name: Pref1
tweet_transmission:
same_ideology:
EntityType1: 0.1
EntityType2: 0.1
else: 0.1
```

As you can see from above, the entity types declared in the entities section are used in the preference classes. If you also declared an entity type 'EntityType3' then that entity type would fall under the 'else' set above. You can see that the transmission probabilities for all of the entity types are the same.

main menu

humourous tweets: (type: n/a) In the entities section, there are different types of tweets that entities can tweet. One of the tweet types is 'humourous' which can be thought of as how often they tweet humourous tweets. When an entity tweets a humourous tweet, the entities that follows the entity who tweeted may retweet the tweet depending on the tweet transmission value and density function. An example of the syntax for declaring a

ideological tweet preference class is:

```
preference_classes:
- name: Pref1
tweet_transmission:
humourous:
EntityType1: 0.1
EntityType2: 0.1
else: 0.1
```

As you can see from above, the entity types declared in the entities section are used in the preference classes. If you also declared an entity type 'EntityType3' then that entity type would fall under the 'else' set above. You can see that the transmission probabilities for all of the entity types are the same.

main menu

follow_reaction_prob: (type: double) If an entity is selected to act on a retweet and they are not following the original tweeter, then they can either retweet the retweet or follow the original tweeter. This probability set here (from 0 to 1) describes how often an entity would follow rather than retweeting the retweet. They need to be set for every preference class. An example is below:

```
preference_classes:
- name: Pref1
follow_reaction_prob: 0.5
```

This probability set above would allow for half follows and half retweeted retweets.

main menu

entities: (type: n/a) This variable should *never* be removed from the input file. This allows the code to read in all of the different entities declared. One can declare as many different entities as one wants, just be sure to properly declare all of the variables covered below.

- name: (type: string) The names of entities is preference; they can be anything one wants. The entities included in the software are Standard, Celebrity, Bot, and Organization.

main menu

weights: (type: n/a) This variable allows the code to read in the add and follow variables for each entity. This should always be declared in the input file for every entity.

main menu

add: (type: double/float) This variable declares the percentage of the certain entity type in the network. Before the simulation is done, a loop is placed over all the different entities declared, and the add variables are summed. The value of add declared in each entity type will then be divided by the add sum. For example if

entity type A has an add value of 75, and entity type B has an add value of 25, then the network will consist of 75% of entity type A, and 25% of entity type B.

main menu

follow: (type: double/float) This variable sets the weights for following each entity type. Like the add variable, these weights are summed, and then each follow variable is divided by the total. This variable is not used if the follow_model is random or preferential, but is used for the other models. For example if entity type A has a follow variable of 85, and entity type B has a follow variable of 15, then entity type A will be followed 85% of the time, and entity type B will be followed 15% of the time.

main menu

tweet_type: (type: double) When an entity tweets they can be different types of tweets they have. These tweet types are incorporated with the preference_classes for retweeting. There are four different tweet types currently implemented and they are 'ideological', 'plain', 'musical', and 'humourous'. The weights associated with each tweet type can also be set along with the tweet type. These weights can be different for each entity type and the weights are summed and each weight is divided by the sum to produce a probability for generating each tweet type. An example of how to set the tweet types is below:

```
entities:
- name: EntityType1
  weights:
    tweet_type:
      ideological: 1.0
      plain: 1.0
      musical: 1.0
      humourous: 1.0
```

From above you can see that all of the weights are the same and therefore there would be the same amount of each tweet type for EntityType1.

main menu

followback_probability: (type: double) This determines the probability that the entity type will follow another entity back using the use_flawed_followback method.

main menu

hashtag_follow_options: (type: n/a) When entities tweet in the network, they can also attach hashtags to their tweets. When they do attach hashtags, they are binned according to their ideology and region. If another entity searches for a hashtag based on their characteristics, they can follow entities with a specific ideology or location. They can either follow entities that have the same ideology and region as them or just ideology and different region. An example of these options is below:

```
entities:
```

- name: EntityType1
hashtag_follow_options:
care_about_region: true
care_about_ideology: true

From this example, the EntityType1 will only follow other entities that have the same ideology and region because they care about the region and ideology for the hashtag follow method. If both are set to false, then the EntityType1 will follow entities with any ideology from any location.

main menu

rates: (type: n/a) Every entity *must* have rates set for tweeting and following. These rates will be updated after a month passes by in the simulation; as mentioned previously the number of months in the simulation are determined by a constant value defined for a month $= 30 \cdot 60 \cdot 24$.

main menu

follow: (type: n/a) Like the function parameter to add entities into the network, the functions that can be set for the follow rate are 'constant' and 'linear'. The usage for defining the follow rate is the *exact* same as the entity add rate. See link for further implementation details.

main menu

tweet: (type: n/a) Again, like the function parameter to add entities into the network, the functions that can be set for the tweet rate are 'constant' and 'linear'. The usage for defining the tweet rate is the *exact* same as the entity add rate. See link for further implementation details.

main menu