ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with MATLAB

Project Report

Simulating the Deployment of an Arabian Spring Movement

Sebastian Heinekamp & Tileman Conring
& Fabian Wermelinger

Zurich
December 15, 2011

Agreement for Free-Download

We hereby agree to make our source code for this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Sebastian Heinekamp Tileman Conring Fabian Wermelinger

Abstract

Contents

1 Individual Contribution

The arising problems during the project were mainly processed and solved by team work of all the members. However, due to the large amount of work, the implementation of the model in MATLAB® was split into three parts. Tileman took care of the network generation, Sebastian wrote the function that converted the information of the network into an array of agents and Fabian implemented the solver function.

2 Introduction and Motivation

A few months ago in some Arabian countries there were some kind of revolutionary movements. In some countries they were successful while in other countries they struggled. Our aim of this research project is to understand the mechanisms of the success of the Arabian Spring movements. We want to build a society based on a small world graph with, e.g. three clusters, where each cluster represents a country adjacent to another. These clusters shall be formed with individual probabilities, generating different homogeneity of each country as such. With the focus on one country, we want to analyze the spreading of opinion changes between nodes (where each node has an agent assigned to) in that particular cluster. Further, the propagation of the opinion formation across the clusters shall be investigated.

Previous research showed that behavioral diffusion in social networks is different from, e.g., the spread of an infectious desease [2]. In social networks, an individual usually requires multiple contact with another individual(s) in order to change its behavior. Also, the depth and speed of the behavioral diffusion is larger in small world networks than randomly generated networks [2]. The main content of this work is the development of a model to simulate the behavioral diffusion for events such as the Arabian Spring [6]. The network is composed of three sub-networks, each representing a country. One question to follow is to find out the influence on the whole system if these countries were constructed as small world networks or random networks. In one of these countries a "seed" is placed to initiate an imbalance in the society's behavior. The information then spreads across the network depending on the behavior of its neighbors (and may be also its higher order neighbors, i.e., neighbors of neighbors and so on). The diffusion across countries is realized through the link between two individuals from different countries who know each other. However, the threshold of willingness to change behavior may be different for an individual of the country which is the root of the behavioral change than that of an individual of a neighbor country, since it is only affected indirectly.

Once a model has been developed, the development of the behavioral diffusion of the model may be compared to the results of the experiment conducted in [2], which is of similar type as the implemented model.

3 Description of the Model

3.1 Kuran's Unanticipated Revolutions

Most of the well known revolutions like the French revolution of 1789, the Russian Revolution of February 1917 and the Eastern bloc revolution of 1989, evolved surprisingly fast. The most recent example is the Arabian Spring. The economist Timur Kuran developed a model to explain why the suppression of the government works fine for a long period and then a revolution can grow entirely unexpected in a very short time span [4]. To explain that phenomena, Kuran described a expressive equilibrium in which the opposition is forced to remain hidden and is strongly discouraged out of fear from the government. But as soon as a few individuals speak up, they can eventually spark a revolution, because now other individuals dare to express their opinion as well. Like most economists Kuran assumes, that the individuals act rational. Furthermore he assumes, that every individual has two opinions. One is the actual private opinion, the private preference and the other is the publicly expressed opinion, the public preference, that is shown to the other members of the society. The case in which an individual's public and private preference don't match is referred to as preference falsification. While the private actual opinion changes slowly over a longer time period, the publicly expressed opinion can change rather fast, since it is only the decision of the individual to lie or not to lie about his actual opinion. To decide whether or not to change the publicly expressed opinion, the individual has to way of his the (expected) costs of punishment in case of a failure of the revolution and the cost of preference falsification. The point at which the costs match each other is the individual's threshold. Because Kuran assumes that the expected costs of opposition depend on the size of the opposition movement, the threshold depends on the actual opinion and the size of the opposition. Dependent on the threshold contribution, a revolution can cascade reasonably fast [3]. To Illustrate how the revolution can develop in theory lets take a stair like distribution of the thresholds as shown in figure 1 with 10% of the individuals with a threshold of 0, 10% with a threshold of 0.1, 10% with a threshold of 0.2 and so on. The first 10% will start the opposition immediately, but then the threshold of the second 10% is reached and the change of the public preference cascades through the whole population.

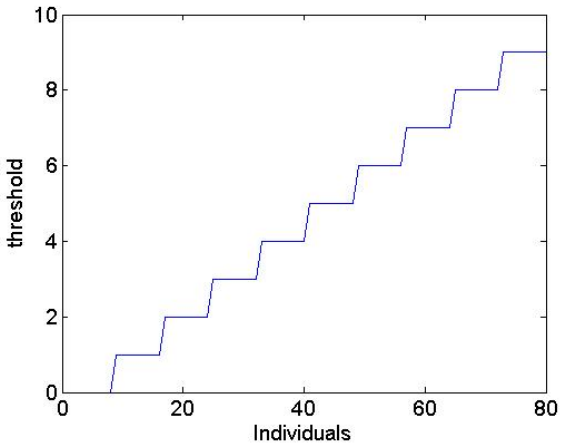

Figure 1 stair like distribution of the thresholds

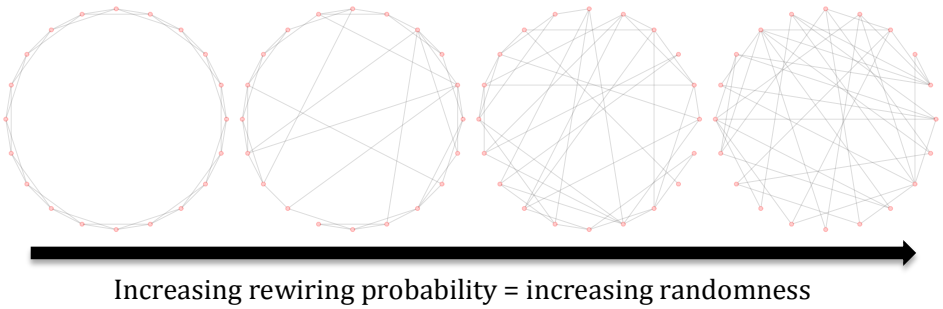

Figure 2 The influence of the rewiring probability varying between 0 and 1 on a network with 20 nodes and k = 4 (graphs are generated with the use of the smallworld.m function, found in Appendix B)

3.2 Description of the Networks

As mentioned in section 2 we want to simulate the spread of opinions on a small world network. To have a comparison we also simulated the change of opinions on a random graph. Small world networks and random graphs also have a close connection as figure 2 shows. In this figure we visualized the Watts and Strogatz model for different rewiring probabilities. If the rewiring probability varied from 0 to 1 a regular graph passes over into a small world network and then into an random graph. In the following we are going to describe the properties of the two network types used in our model. For more details see [5].

3.2.1 Small World Network

The most important property of an small world network is that most nodes could be reached from every other by quite a small number of steps. As described in the paper [7] the typical distance L of two random picked nodes is proportional to the log of the total number of nodes.

$$L \propto \log N \tag{1}$$

In our model we use the model proposed by Watts and Strogatz. This implements the two main properties of small world networks, which are

1. small average path length
2. high clustering coefficient

The Watts Strogatz algorithm follows a simple rule. As in figure 2 the plot on the left shows first there is a regular ring lattice constructed, where every node has exactly for a chosen integer k, k neighbours. In a second step every node is taken an rewired with a chosen probability. It is done in a way avoiding loops and duplication i.e. connecting a node to itself.
A lot of real world examples follow small world type networks. The most famous experiment showing this was done by S. Milgram showing that the average distance between two random persons in the world is five. Other examples that follow the two most important properties of small world networks are the world wide web, social networks and for example also gene networks.
One of the main criticism about small world networks is that they produce unrealistic

degree distributions. Many real networks follow the preferential attachement models for example proposed by Barabasi, which give us scale free networks. This means that the produced networks have hubs with a lot of links but also many nodes with just a few links to other nodes. But scale free networks fail to have a high clustering coefficient which also a lot of real world networks have. So neither of the models is perfect.

3.2.2 Random Graph

As the name already says random graphs are generated with some random processes. One of the most important differences of a random graph compared to a small world network is the lack of local clustering. Also the degree distribution is binomial. We used an implementation after a model proposed by *Erdós* and *Réyi*. It could be understood as the limiting case of the Watts Strogatz Algortihm described in section 3.2.1 for a rewiring probability of 1. An example is the graph on the right in figure 2.

3.3 Solving Process

Once a network has been defined, the relation between the nodes is fully determined. If $\mathcal{N}$ is the set of all nodes contained in the network, then each node i has a set of neighbors $\mathcal{S}_i \subset \mathcal{N}$. The neighbors $j \in \mathcal{S}_i$ of node i may be *direct* or *indirect* neighbors. A direct neighbor j is one that is linked adjacent to node i, where an indirect neighbor node is one that is a neighbor of a neighbor in a successive fashion. Each node is characterized by two main features, i.e., a state and a threshold.

Definition 1. The state of a node is a binary variable. The value 0 means that the state for a situation is *pro*, whereas the value 1 is *contra* the situation. The state is subject to change.

Definition 2. The threshold of a node is a numeric value on the interval $[0, 1]$. A threshold of 0 means that the node will remain in its state no matter what the external influences are. A threshold of 1 acts opposite. That is, a node with a threshold of 1 will *always* change its state from pro to contra, no matter what the external influences are. If it already was in contra state, it will remain in that state. Hence, a node with a threshold of 1 acts as a "seed" node to initiate some event. The threshold is a constant.

With the above definitions, the solving process is as follows:

1. Loop over time
2. Generate a sequential update list
3. Loop over the sequential update list
4. Calculate the neighbor state residual and compare it with the threshold of the actual node
5. If the condition applies, update the state of the node

3.3.1 The Neighbor State Residual

The neighbor state residual is a scalar which is a measure of an overall state for all neighbor nodes $j \in \mathcal{S}_i$. It is used to decide whether a nodal state is changed by comparing it with the nodal threshold. If there are n elements in $\mathcal{S}_i$, then let $\boldsymbol{r}_i \in A^n$ be a vector of dimension n, where its elements r_j hold the state of each node $j \in \mathcal{S}_i$. The set A holds the two elements 0 and 1, which represent the state. The neighbor state residual, ρ_i, is then defined to be the Euclidean norm of $\boldsymbol{r}_i$, i.e.,

$$\rho_i = \|\boldsymbol{r}_i\| \tag{2}$$

So if ρ_i is greater or equal to the threshold of node i, then the state will be forced to contra, i.e., to the value 1. The calculation then proceeds for all nodes in the sequential update list before the next time step is processed.

4 Implementation in MATLAB®

The implementation in MATLAB® is mainly based on functions, which are sequentially called in a main script file. All the model parameter are defined in a single structure, which is passed to all the called functions. The parameter and their meaning are also declared in the main script file. The main script can be found in Appendix A.

4.1 Implementation of the networks

The full code of the network generating functions can be found in Appendix B and Appendix C. This section describes shortly the implementation of Section 3.2 in MATLAB® code. For more details on the implementation of the networks see [1].

We used two different implementations for the small world network and the random graph although the random graph is just a limiting case in the Watts and Strogatz Model. The Small world model has three input parameters: the total number of nodes, the mean degree and the rewiring probability. The output of the implementation is a sparse symmetric adjacency matrix representing the network graph. We used the format sparse to save memory. The implementation first constructs an regular lattice, where each node has in total the given mean degree. In a second step in two nested for loops we iterate over all nodes exploiting the symmetry and rewire the loops with the specified rewiring propability. As mentioned the output needs to be symmetric which is after the rewiring not the case but now we can make use of the symmetry in order to get the full symmetric adjacency matrix.

The random graph is an implementation of the *Erdós* and *Réyi* model. The input are the total number of nodes and the probability that two nodes are connected and the output is again a sparse symmetric adjacency matrix representing the generated network graph. In the implementation is first the number of non-zero values in every row for a adjacency matrix just containing 0 and 1 generated. In a second step this number is for every row distributed with a binomially distribution with two specified parameters - the total number of nodes and the probability that two nodes are connected. At the end is again made use of the symmetry of the network adjacency matrix.

In both implementations we have a big for loop over the network in order to construct the three sub-networks that have some links inbetween.

The plots of the networks were done using the igraph library in R. The layout used the in this package implemented Fruchterman and Reingold algorithm, which is a force-based algorithm.

4.2 Implementation of the Agents function

The complete MATLAB® code can be found in Appendix D. The agents function fulfils two purposes. The first is to set and determine the properties every single node (agent) in the network has. As second it acts as interface between the generation of the network and the solver.

The function consists of a for loop that iterates through all nodes of the network and generates a list containing all the agents, with there properties. These properties are the state, the threshold, a list of all the neighbours and which country or cluster the agent is part of. The state tells whether the agent already joined the revolution (1) or not (0),

therefore all agents initially get the value 0. The neighbour property is a list of all the agents the particular agent has a connection to in the network.

The threshold is the percentage of surrounding agents that sill have to have state 0, so the agent does not join the opposition. It gets determined by a uniformly distributed random number between 0 and 1. Dependent on the given parameters a certain percentage can get a fixed threshold. Furthermore a few agents get a threshold of 1. Those are the agents, that start the revolution. How many of these rebels are placed in the network and how far they are apart depends on the input parameters.

4.3 Implementation of the Solver Function

The full code of the solver function can be found in Appendix E. This section describes the implementation of Section 3.3 in MATLAB® code. Additionally, the actual solver implemented in MATLAB® also supports a variant of a SIR (Susceptible Infected Recovered) model for a cellular automaton.

The solver consists mainly of two nested *for*-loops, where the outer loop is over a time vector and the inner loop is over a sequential list of agents, which are to be updated in the active time-step. In each of these loops a call to a subroutine may be executed. Since the subroutines are solver specific functions, they are appended at the bottom of the solver function. Therefore, they are only visible to the solver. The subroutines are sufficiently commented in the code and will not be discussed here in more detail. However, in order to aid the understanding of how the solver works in general, the statements in the nested loops need further explanation.

The main algorithm starts at line 33 and ends on line 75 in the source code found in Appendix E. The very first thing that is done in every time-step is the calculation of the individual cluster residuals, as well as the global residual for the whole network. If the whole network consists of only one cluster, the two residuals are the same. The calculated residuals provide a measure of the development of the social behavior over time. If it is close to zero, the majority of people are content with the current situation. If it is close to one, the opposite is true. On line 39 the sequential agent update list is calculated. It is a uniformly distributed list of random agent indices that get update in the current time-step. The list is of random length (also uniformly distributed) but at most `maxUpdatedAgents` long. The constant `maxUpdatedAgents` is calculated on line 26 and is some percentage (defined in the parameter structure) of $\mathcal{N}$, thus, if the set $\mathcal{U}$ contains the sequential update list of the current time-step, then $\mathcal{U} \subseteq \mathcal{N}$.

On line 40, a number of noisy agents is calculated in the same way as it is descried above for the `maxUpdatedAgents` constant. The only difference is that `nNoize` is a percentage of the set $\mathcal{U}$. The *if*-block from line 42–51 then adds the noize and removes the noisy agents from the sequential update list. The update algorithm starts with the *for*-loop on line 53. The algorithm first checks whether the agent is subject to update. That is, if its state is zero and if ρ_i is strong engough to pass the agents threshold. Then the infection phase of the SIR implementation takes effect. If $\beta = 1$, the infection is not applied at all. If all of these tests are true, then the agent's state is updated to contra (i.e., "1") on line 59. Note that ρ_i is computed with a recursive function in order to allow an arbitrary depth of indirect neighbors. If `nbrDepth` is set to unity in the parameter list, then only direct neighbors are taken into account.

The *if*-block from line 66 to 69 applies the removal phase of the SIR implementation. Note that if $\gamma = 0$, the expression is never true and the removal phase is ignored. Therfore,

if one sets the parameter $\beta = 1$ and $\gamma = 0$, then the SIR model is turned off. The last *if*-block is a write operation, which generates a `.csv` file. The file is simply the list of agents with contra state at the current time-step. We have used this file to generate visualizations of time dependent developments in the network. We have used R[1] to generate the network plots.

4.3.1 Limitations

The current implementation of the solver function is limited by means that it only allows state updates in one direction. That is, an agent's state can only be changed from pro to contra. For further development, one may consider to allow the reverse as well. However, this implies that if the initial agents all have a pro state and at least one agent acts as a seed, then noize must be added in order to allow an agent to change its state in the reverse direction.

The current implementation also assumes that each agent knows all of its direct and indirect neighbors. Such an assumption is conservative, as it might be very well true that an agent only knows some of its neighbors, this is especially true for the indirect neighbors. Hence, an additional parameter could be introduced that defines such behavior. This would have immediate effect on ρ_i and eventually influences the decision making of the agent. Also, note that if the number of indirect neighbors is increased (by increasing the `nbrDepth` parameter), the computation time of the solver is also increased (depending on the network, it may increase drastically).

[1] R is a collection of software packages for statistical computing and graphic generation. See `http://www.r-project.org/` for more information.

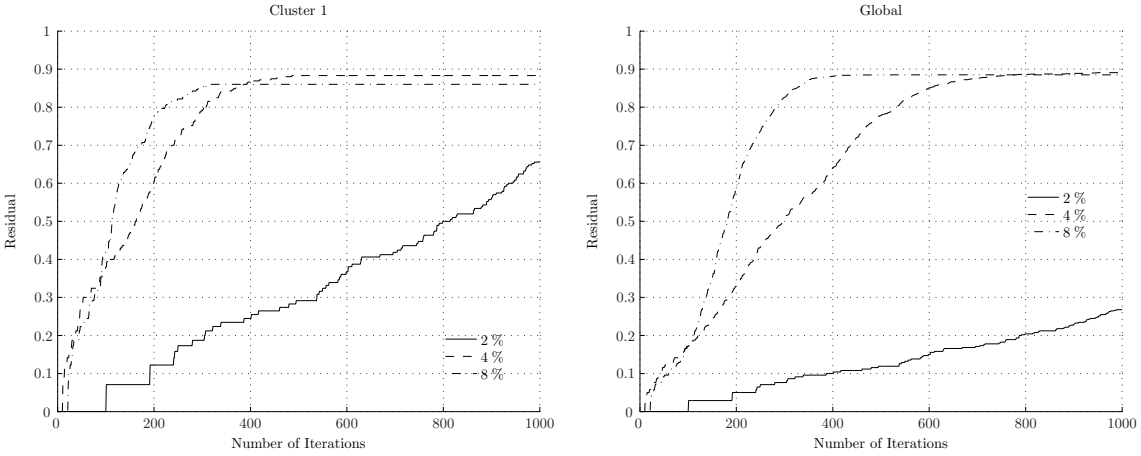

Figure 3 On the left the residual for the cluster in which the riots started is plotted while on the right the global residual, i.e., all three cluster combined, is shown. In both it is shown for values of 0.02, 0.04 and 0.08 of `maxAgentupdate`

5 Discussion of Results

5.1 Influence of some parameters in our model

5.1.1 Influence of `maxAgentupdate`

As figure 3 shows the value for `maxAgentupdate` just gives us a different scaling of the time axis. If it is bigger the total residual rises faster while if it is smaller the total residual rises slower. As a result of this a change in the value of `maxAgentupdate` could help to speed up the simulation in order to get quicker results.

5.1.2 Influence of `noize`

As figure 4 shows, the noize could have quite an influence on the results of the simulation. In our network we have in total 1200 nodes. All the simulations in figure 4 have the same `maxUpdate`=0.02. This means that in every time step 24 agents get updated. If the noize is now for example 0.01 only 0.24 agents choose randomly their state. In the case of a noize of 0.1 get 2.4 agents every time step a randomly after a uniform distribution chosen mind state. After this "noize update" the concerned agents are removed from the sequential update list so that they could not be updated twice.
As figure 4 now shows and explained above a noize of 0.01 does not really change the results of the simulation as it has nearly no influence. But as the series of five plots on the right in figure 4 shows a noize of 10 % does have quite a big influence on the results of the simulation. Before this big noize the riots where not really starting in the networks now they spread quickly.
Compared to reality we think that the noize tends to be close to 0. Who is just completly random changing his mind? It could be that some people change their mind independently from their neighbors and these should be covered by the noize. We think that a noize in reality should be something between 0 and 0.06. Refer to Section 3.1 for deeper thoughts on this topic, as well as private and public mind states.

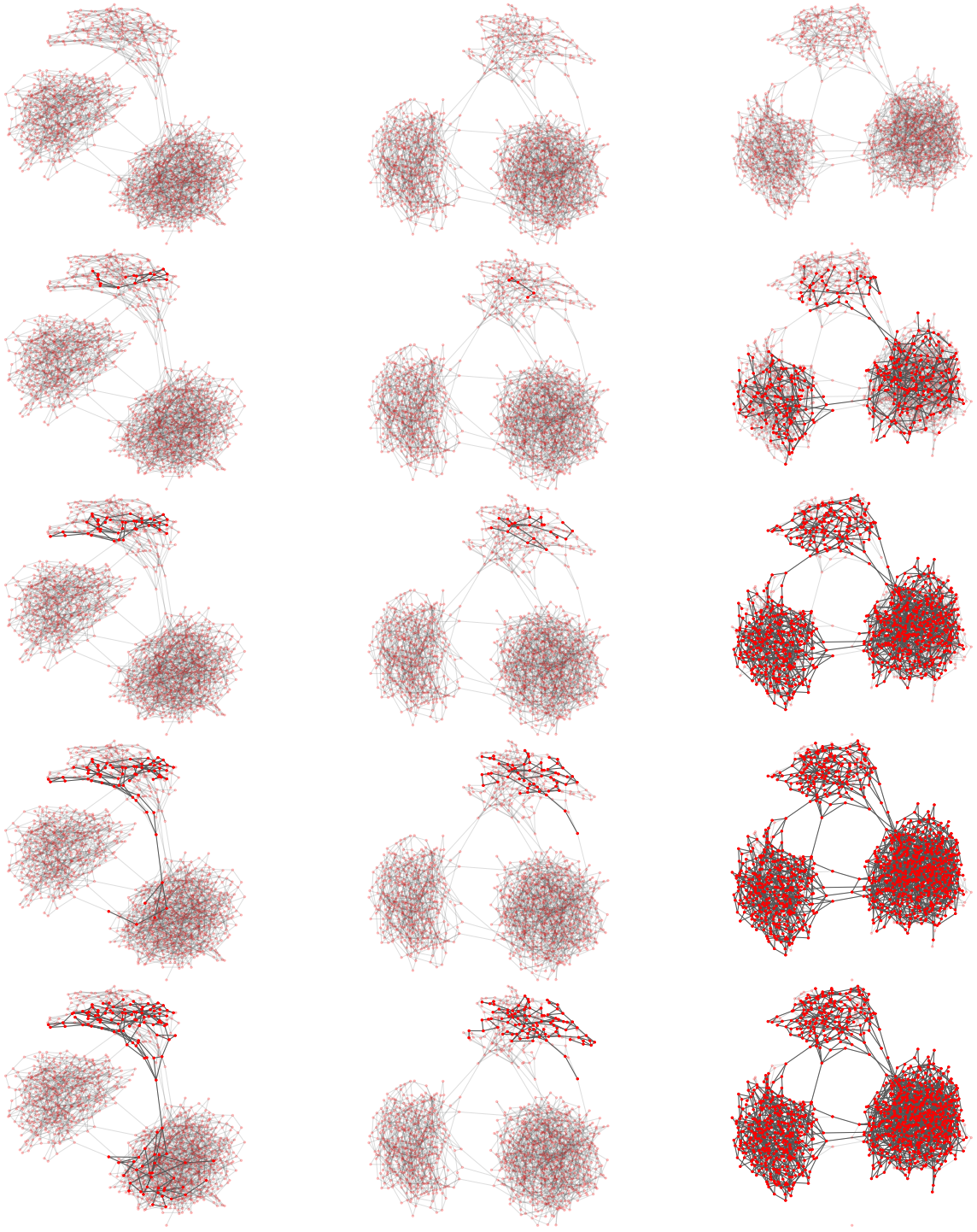

Figure 4 The influence of the noize for the three cases with on the left noize = 0, in the middle 0.01 and on the right 0.1. From top to bottom are the different time steps with the beginning and then increasing in 250 time steps till a 1000 time steps.

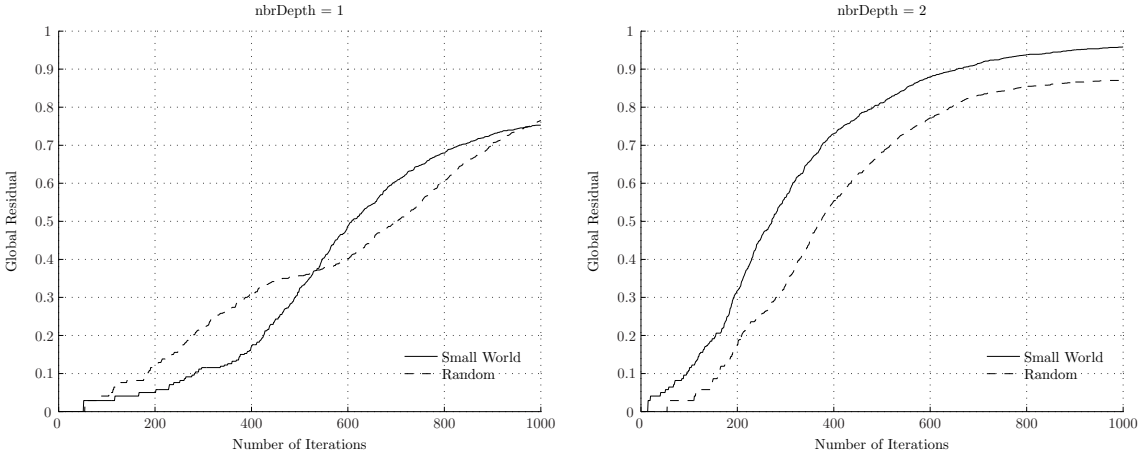

Figure 5 The residuals for neighbor depth 1 (on the left) and 2 (on the right) compared for a small world network and a randrom graph. The other parameters for the small world network are as in 6 and for the random graph as in 9.

5.1.3 Influence of `nbrDepth` = Neighbor Depth

As figure 6 shows has an iteration over neighbor depth 2 a faster dynamic compared to a neighbor depth of 1. Through a neighbor depth of 2 there is also a quicker jump over from one cluster to the next cluster. This could be explained simply because there is a high probability that with one node in between there is a link between two clusters. The residuals belonging to figure 6 are shown in figure 5.

5.1.4 Influence of the threshold distribution

The figures 7 and 8 show, that the revolution can only cascades without hindrance if the thresholds are distributed stair like as described in 3.1. The uniformly distributed threshold we use, equals a stair like distribution with infinitely small steps. In the extreme case in with we set a threshold of 0.2, for 50% of the agents, the growth of the opposition stops very soon at a low level. The more moderate case of a threshold of 0.4 for half of the agents, the growth of the oppsition is not prevented, but clearly hindered and slowed down.

On the other hand a threshold distribution that is a favourable for the opposition, like in the example with a threshold of 0.6, the growth of the opposition gets accelerated and the opposition reaches greater percentage of the network.

5.1.5 Influence of the network type

As the figure 9 and the two plots in figure 5 show has the network type an influence on the outcome of the simulation. As we use quite well connected networks the influence is not big but still significant. The plots show that the spread on a random network could sometimes be faster but in general is slower and reaches a lower final residual. The cause for this could be that there exist not so well connected parts of the network into which it is harder for the riots to spread.

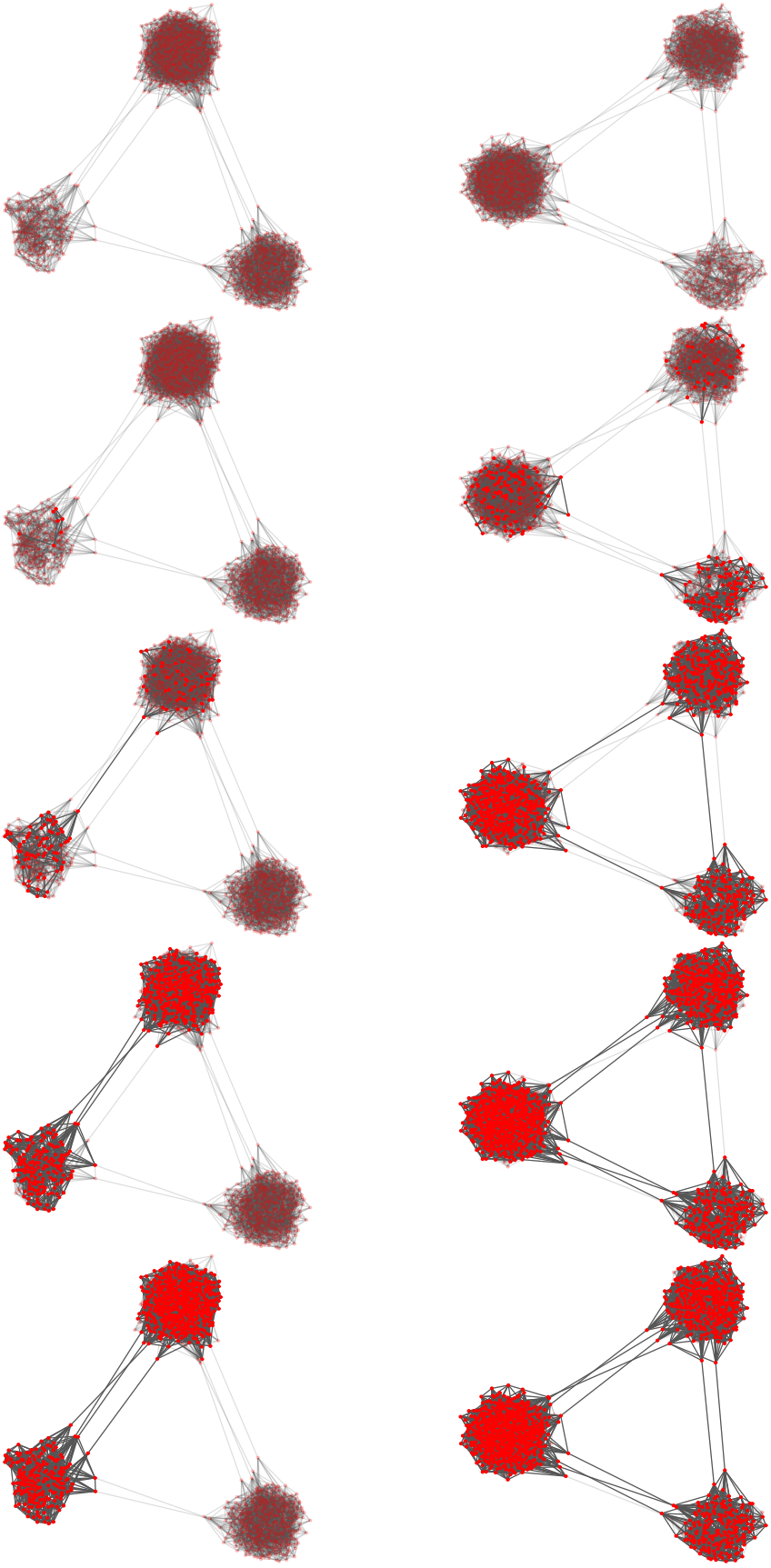

Figure 6 For this graphs we used a small world network with mean degree 12, a maximal update of 0.02 and 0 noize. In the left series of plots we had a neighbor depth of 1 and in the right series a neighbor depth of 2.

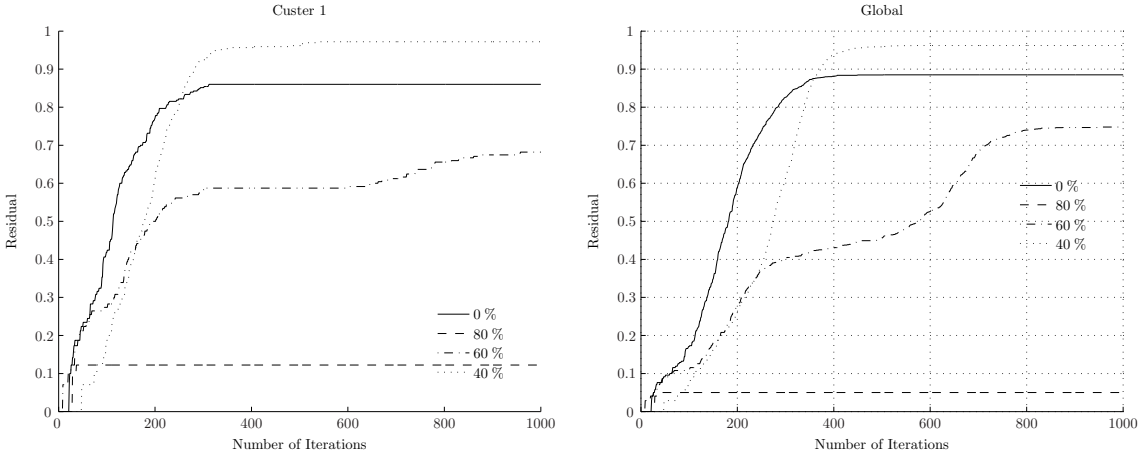

Figure 7 The left shows the residual for the cluster in which the riots started, while the right shows the residual over all cluster. Both show the different results for a threshold of 0.0, 0.2, 0.4 and 0.6, for half of the agents. The other half of the agents still have a uniformily distributed threshold.

5.2 Comparison with a Real Experiment

So far we were not able to verify the simulated data with real data related to the Arabian Spring development in northern Africa. There is not enough time to research data regarding the Arabian Spring. However, there is a possibility for comparison to a similar experiment presented in [2]. The author tested the effects of network structures on behavioral diffusion. The study involved randomly assigned participants to join a health community in the internet. The behavior of the agents was based on the decision of whether to join an online health forum or not, based on the action of their "health-buddies", which are their neighbors. The similarity to the implemented model in Section 4 and the experiment is that they share the same mechanism of decision making and both offer only two choices to make. The experiment was conducted on two different network structures, i.e., a small world network according to [7] and a randomly rewired network. The difference, however, is that the implemented model does not have a meaningful time scale, since no parameter were defined based on real data. Although, the characteristics of the decision making over time remains similar.

By comparing Figure 5 and Figure 2 in [2], one can see that with a lower number of neighbors (Figure 2A in [2]) and a smaller population size, intersection of the two residuals may be possible. The simulated data for the plot in Figure 5 was conducted with the parameter `kHalf` set to six. This means that each node has 12 neighbors on the average (for a neighbor depth of one). There were only six neighbors for the experiment, however. The left plot in Figure 5 shows the residuals for a neighbor depth of one. These residuals also intersect each other for a fewer number of neighbors. Where eventually, the behavior diffuses faster in the small world networks for the experiment. The simulation indicates almost equal residuals for the small world and random types of networks at the end of iteration (left plot). This might be due to too few iterations or other unknown uncertainties. On the other hand, if each agent has 12 direct neighbor agents, then there will be a total of around 144 direct and indirect neighbors for a neighbor depth two. The right plot in Figure 5 is for a neighbor depth two. Compared to Figure 2E and F in [2], both show steeper gradients during the first few time steps. The small world network then diffuses information faster than the random network for all later time steps.

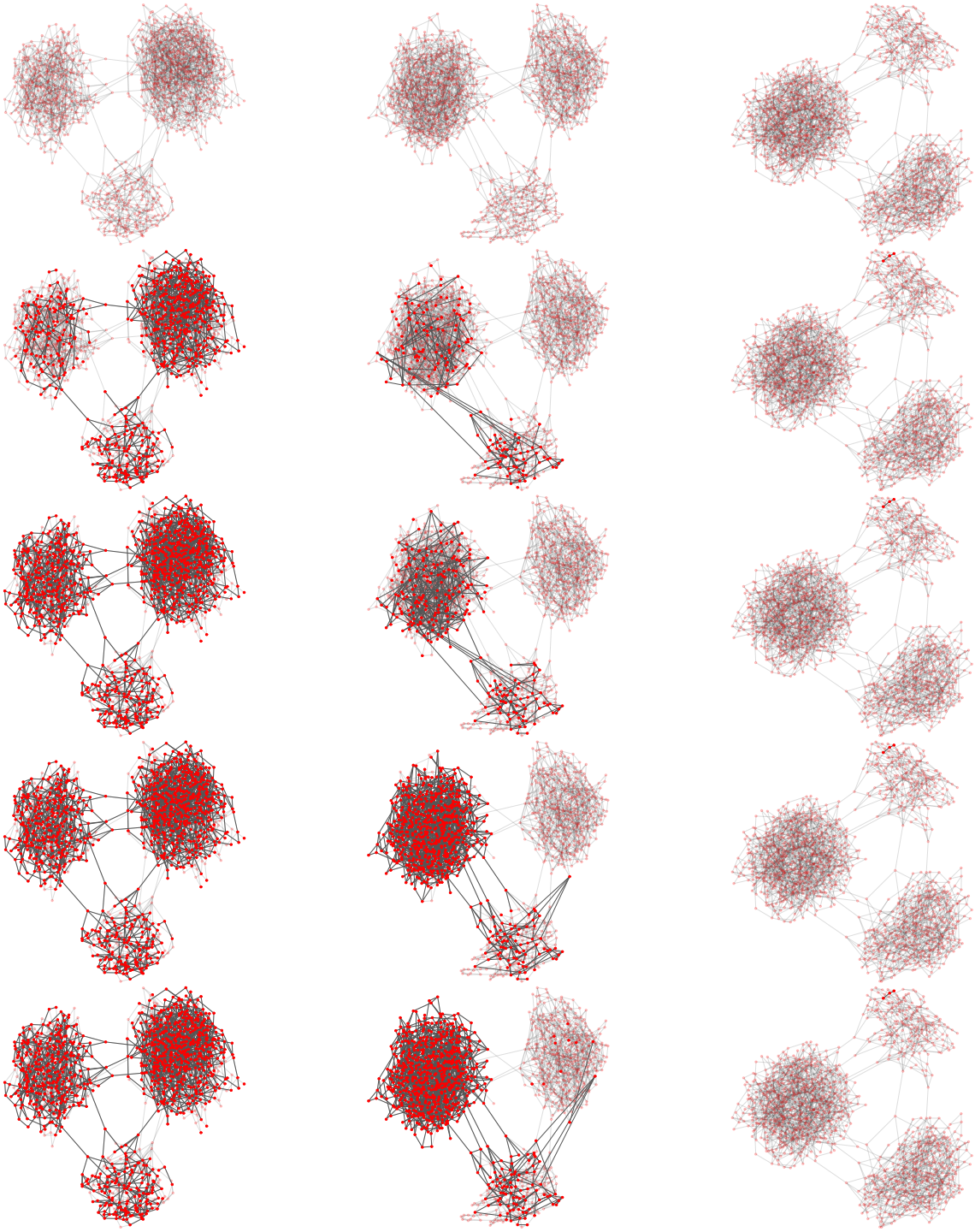

Figure 8 For this graphs we used a small world network with mean degree 12, a maximum update of 0.08, 0 noize and a neighbour depth of 2. The left graphs are for comparison, while the middle and right graphs are generated with a not uniformly distributed threshold. In the the middle graphs half of the agents have a threshold of 0.4 and in the right graphs one of 0.2. The other half of the agents stays uniformly distributed.

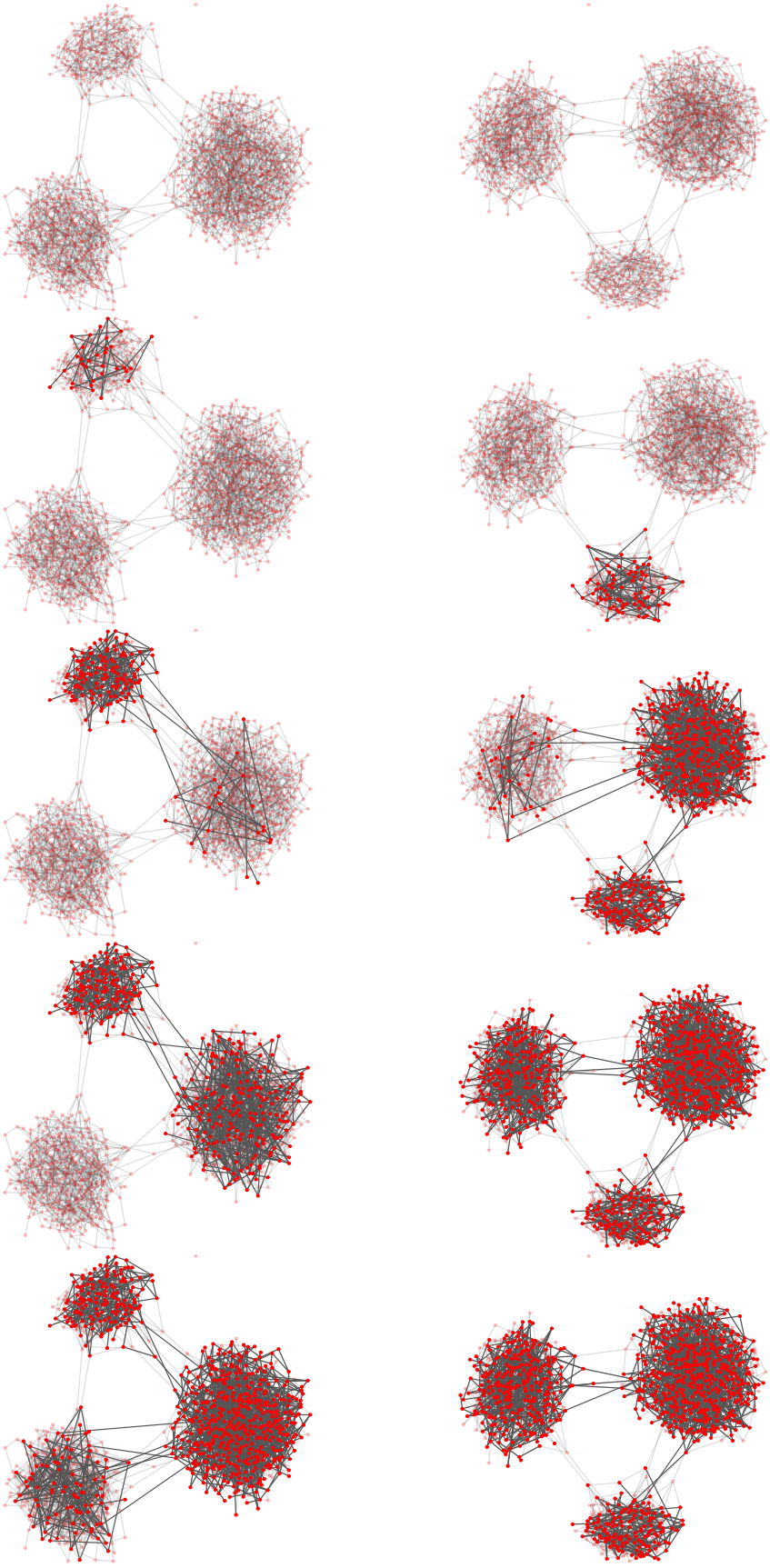

Figure 9 The figure above shows the influence of the neighbor depth for the spread of riots in a random network. The left series has network depth one and the right network depth two. The total network has a size of 1200.

There are indeed similarities between the simulated data and the experiment conducted on a similar model in [2]. However, the presented comparison above is in no way a validation of the model. It only shows that both have certain characteristics in common which does relate the implemented model of Section 4 to a real situation in some way. This may be one as in [2] or an Arabian Spring movement. However, a valid parameter set must be determined for both situations.

6 Conclusion

All in all our simulations showed some interesting phenomena and sensitivities to different parameters which we are going to summerize in the following. Through the project some interesting new questions arised while others could not be sufficiently be answered.

As described in the previous sections the type of the network did not had a to big influence on the results of our simulation. Perhaps were these quite similar results for the different network types a result of the well connected networks we used for most of our simulations. The biggest difference about the random graph and small world network that our simulations showed was that for a random graph the spread of opinions across different clusters was not as predictable as for small world networks. This could be a result that not all the nodes of a random graph have such a short connection as in a small world network. Overall our simulations showed for different parameters that the diffusion of opinions is in small world networks in principal better.

Section 5.2 showed that there is a relation of the model implemented in MATLAB® to a real world event. In order to perform more event specific simulations, parameter sets determined from data of the particular event must be provided. In general we conclude that the experiment and the model behave closer to each other if the elements in $\mathcal{N}$ and $\mathcal{S}_i$ *increase*. However, the number of elements in $\mathcal{S}_i$ must still be significantly less than the number of elements in $\mathcal{N}$.

The next step in developing the model would be to implement a probability which takes into account that an agent might not know all of its direct and indirect neighbors. The weight of that probability should be on its *indirect* neighbors. The agents threshold field was always uniformly distributed for all simulated data. It would be interesting to see the effect of other distribution models and their impact on the simulation results. For example, a normal distribution might be an alternative. A normal distribution might yield similar results as discussed in Section 5.1.4 if there are too many agents with thresholds that are not too distinct.

References

[1] S. Brugger and C. Schwirzer, *Opinion formation by "employed agents" in social networks*. Project work done in the course Lecture with Computer Exercises: Modelling and Simulating Social Systems with MATLAB, May 2011, 2011.

[2] D. Centola, *The Spread of Behavior in an Online Social Network Experiment*, science, 329 (2010), p. 1194.

[3] K. Donnay, *Hard, harder, ultra hard: The role of state hardness in the arab spring revolutions*. Seminar paper, 08 2011.

[4] T. Kuran, *Sparks and prairie fires: A therory of unanticipated political revolution*, Public Choice, 61 (1989), pp. 41–74.

[5] M. E. J. Newman, *Networks: An Introduction*, Oxford University Press, 2010.

[6] UCDP, *Timeline arab spring*, November 2011. conflictdatabase@pcr.uu.se.

[7] D. J. Watts and S. H. Strogatz, *Collective dynamics of 'small-world' networks*, Nature, 393 (1998), pp. 440–442.

A main.m

```
% --------------------------------*- Matlab -*---------------------------------
% Filename: main.m
% Description: Main script to execute the simulation
% Author: Fabian Wermelinger
% Email: fabianw@student.ethz.ch
% Created: Thu Nov 10 21:20:16 2011 (+0100)
% Version:
% Last-Updated: Wed Dec 14 16:16:32 2011 (+0100)
%           By: Fabian Wermelinger
%     Update #: 174
% -------------------------------------------------------------------------------
% main.m starts here
% -------------------------------------------------------------------------------
clear( 'all' );

% -------------------------------------------------------------------------------
% set paths
% -------------------------------------------------------------------------------
whereIs.main = pwd;
cd( '..' );
whereIs.src = pwd;
whereIs.func = [whereIs.src filesep 'func'];
addpath( whereIs.func );
cd( whereIs.main );

% -------------------------------------------------------------------------------
% parameter
% -------------------------------------------------------------------------------
% the nodes variable define the number of nodes in the network.  each
% element of the vector defines the number of nodes in that cluster, hence
% the total number of nodes in the global network is sum( par.nodes ) and
% the number of clusters in the global network is length( par.nodes ).
par.nodes = [200 400 600];

% this parameter defines the maximum percentage of updated agents per time
% step.  it is an upper bound, the actual updatet agents may also be less.
par.maxAgentUpdate = 0.04;

% this parameter defines the percentage of agents from the sequential update
% list to introduce noize by a randomly set mind state with the randi()
% function.
par.noisyAgent = 0.0;

% this parameter defines the considered depth of neighbor agents of a root
% agent.  E.g., 1 means consider only the immediate neighbors of root, 2
% means consider also the neighbors of neighbors and so on.
par.nbrDepth = 1;

% used for network generation.  khalf is the mean degree half and alpha is
% the rewiring probability.
par.kHalf = [2 2 2];
par.alpha = [1 1 1];
```

```
53
54 % number nodes between the different networks
55 par.between = [2 4 2];
56
57 % Gives the possibility to build in a favour for the rebls or the system.
58 % for possitive thresholdoffsets the thresholds of all the agents gets a
59 % bit higher, but by less than the given offset.
60 % likewise for negative thresholdoffsets.
61 par.thresholdoffset = 0.01;
62
63 % fix threshold for part of the network
64 % the first parameter is the partion in procentige that gets a fixed
65 % threshold. The second is the threshold these agents get.
66 par.fixedthreshold = [0 , 0];
67
68 % this parameter determines in which country and how many rebles are placed
69 % in the network.
70 % riot(1) is the country, riot(2) is the number of rebls.
71 par.riot = [1 1];
72 % stretch rang: [0.5 2]
73 % determines how far the rebls are placed apart. (inverse proportional)
74 par.stretch = 1;
75
76 % the time variable defines the start and end time of the simulation with a
77 % two element vector [tStart tEnd].  the nTime variable defines the number
78 % of nodes in the time domain.
79 par.time = [0 100]; % [day]
80 par.nTime = 1000;
81
82 % the beta and gamma variables define the infection rate and the immunity
83 % rate, respectively, of the SIR model.  Each cluster has its own value.  If
84 % beta = 1 & gamma = 0 -> the SIR model is not used at all.
85 par.beta = ones( size(par.nodes) ); % [day^-1]
86 par.gamma = zeros( size(par.nodes) ); % [day^-1]
87
88 % defines the write interval, measured by the time step, for the .csv disk
89 % write for post-processing purposes.  used later for network plotting.  a
90 % value of 10 means that every tenth time step a file is written.
91 par.csvInterval = 250;
92
93 % ------------------------------------------------------------------------
94 % start simulation
95 % ------------------------------------------------------------------------
96 [res, initStat, endStat, agent, S] = runSim( par );
97 plot( res );
98
99 % ------------------------------------------------------------------------
100 % main.m ends here
101 % ------------------------------------------------------------------------
102
```

B smallworld.m

The main part of this code was taken from [1].

```
%% ------------------------------*- Matlab -*--------------------------------
% Filename: smallworld.m
% Created: Thu Nov 10 21:29:23 2011 (+0100)
% Version:
% Last-Updated: Wed Nov 23 16:58:06 2011 (+0100)
%           By: Tileman Conring - tconring@gmail.com
%     Update #: 10
% -----------------------------------------------------------------------------
% smallworld.m starts here
% -----------------------------------------------------------------------------
%
% Modeling and Simulating Social Systems with MATLAB
% http://www.soms.ethz.ch/matlab
% Authors: Stefan Brugger and Cristoph Schwirzer, 2011

function S = smallworld(par)
% Generate a small world graph using the "Watts and Strogatz model" as
% described in Watts, D.J.; Strogatz, S.H.: "Collective dynamics of
% 'small-world' networks."
% A graph with n*k/2 edges is constructed, i.e. the nodal degree is n*k
% for every node.
%
% INPUT
% n: [1]: number of nodes of the graph to be generated
% kHalf: [1]: mean degree/2
% beta: [1]: rewiring probability
%
% OUPUT
% A: [n n] sparse symmetric adjacency matrix representing the
% generated graph

%generate empty sparce matrix
S = sparse( sum(par.nodes) );

%generate the vector with the positions of the sub networks that cluster
s = ones( 1, length(par.nodes)+1 );
for i = 2:length(s)
    s(i) = s(i-1) + par.nodes(i-1);
end

%generate small world network
for ni = 1:length(par.nodes)
n = par.nodes(ni);
% Construct a regular lattice: a graph with n nodes, each connected to k
% neighbors, k/2 on each side.
k = par.kHalf(ni)*2;
rows = reshape(repmat([1:par.nodes(ni)]', 1, k), par.nodes(ni)*k, 1);
columns = rows+reshape(repmat([[1:par.kHalf(ni)] ...
    [par.nodes(ni)-par.kHalf(ni):n-1]], par.nodes(ni), 1), ...
    par.nodes(ni)*k, 1);
columns = mod(columns-1, par.nodes(ni)) + 1;
```

```
B = sparse(rows, columns, ones(par.nodes(ni)*k, 1));
A = sparse([], [], [], par.nodes(ni), par.nodes(ni));

% With probability beta rewire an edge avoiding loops and link
% duplication. Until step i, only the columns 1:i are generated making
% implicit use of A's symmetry.
for i = [1:par.nodes(ni)]

% The i-th column is stored full for fast access inside the following
% loop.
    col= [full(A(i, 1:i-1))'; full(B(i:end, i))];
    for j = i+find(col(i+1:end))'
        if (rand()<par.alpha(ni))
            col(j)=0;
            k = randi(par.nodes(ni));
            while k==i || col(k)==1
                k = randi(par.nodes(ni));
            end
            col(k) = 1;
        end
    end
    A(:,i) = col;
end

% A is not yet symmetric: to speed things up, an edge connecting i
% and j, i < j implies A(i,j)==1, A(i,j) might be zero.
T = triu(A);
A = T+T';

%write the generated subnetwork at its position in the empty sparse matrix
S( s(ni):par.nodes(ni)+s(ni)-1, s(ni):par.nodes(ni)+s(ni)-1 ) = A;
end

%connect subnetworks with a specified number of connections
count=1;
for b=1:(length(par.nodes)-1)
for c=b:(length(par.nodes)-1)
    add=round([(rand(par.between(c),1) * ((s(b+1)-1) - s(b)) + s(b)), ...
        (rand(par.between(c),1) * ((s(c+2)-1) - s(c+1)) + s(c+1))]);
    for d =1:length(add)
        S(add(d,1),add(d,2))=1;
        S(add(d,2),add(d,1))=1;
    end
    count=count+1;
end
end


%export the network as a .csv file
csvwrite('SW.csv', full(S));

end % small_world(...)
```

C randomgraph.m

The main part of this code was taken from [1].

```
%% --------------------------------*- Matlab -*---------------------------------
% Filename: smallworld.m
% Created: Thu Nov 10 21:29:23 2011 (+0100)
% Version:
% Last-Updated: Wed Nov 23 16:58:06 2011 (+0100)
%           By: Tileman Conring - tconring@gmail.com
%     Update #: 10
% -----------------------------------------------------------------------------
% randomgraph.m starts here
% -----------------------------------------------------------------------------
%

function S = randomgraph(par)
%uses the function written by Stefan Brugger and Christoph Schwirzer to
%create a random network
%
%in our example the function is called three times to create a cluster of
%three random graphs
%
% INPUT
% n: [1]: number of nodes of the graph to be generated
% beta: [1]: rewiring probability
%
% OUPUT
% A: [n n] sparse symmetric adjacency matrix representing the generated
% graph

%generate empty sparce matrix
S = sparse( sum(par.nodes) );

%generate the vector with the positions of the sub networks that cluster
s = ones( 1, length(par.nodes)+1 );
for i = 2:length(s)
    s(i) = s(i-1) + par.nodes(i-1);
end

for ni = 1:length(par.nodes)

% call random graph function
A=random_graph(par.nodes(ni), par.alpha(ni));

%write the generated subnetwork at its position
S( s(ni):par.nodes(ni)+s(ni)-1, s(ni):par.nodes(ni)+s(ni)-1 ) = A;
end

%connect subnetworks with a specified number of connections
count=1;
for b=1:(length(par.nodes)-1)
for c=b:(length(par.nodes)-1)
    add=round([(rand(par.between(c),1) * ((s(b+1)-1) - s(b)) + s(b)), ...
        (rand(par.between(c),1) * ((s(c+2)-1) - s(c+1)) + s(c+1))]);
```

```
52      for d =1:length(add)
53          S(add(d,1),add(d,2))=1;
54          S(add(d,2),add(d,1))=1;
55      end
56      count=count+1;
57  end
58  end
59
60  %export the network as a .csv file
61  csvwrite('random.csv', full(S));
62
63  end % randomgraph
64
65  % Modeling and Simulating Social Systems with MATLAB
66  % http://www.soms.ethz.ch/matlab
67  % Authors: Stefan Brugger and Cristoph Schwirzer, 2011
68
69  function A = random_graph(n, p)
70  % Generates an undirected random graph (without self-loops) of size n (as
71  % described in the Erdoes-Renyi model)
72  %
73  % INPUT
74  % n: [1]: number of nodes
75  % p: [1]: probability that node i and node j, i != j, are connected by an
76  % edge
77  %
78  % OUTPUT
79  % A: [n n] sparse symmetric adjacency matrix representing the generated
80  % graph
81
82  % Note: A generation based on sprandsym(n, p) failed (for some values of
83  % p sprandsym was far off from the expected number of n*n*p non-zeros),
84  % therefore this longish implementation instead of just doing the
85  % following:
86  %
87  %    B = sprandsym(n, p);
88  %    A = (B-diag(diag(B))~=0);
89  %
90
91  % Idea: first generate the number of non-zero values in every row for a
92  % general 0-1-adjacency matrix. For every row this number is distributed
93  % binomially with parameters n and p.
94  %
95  % The following lines calculate "rowsize = binoinv(rand(1, n), n, p)",
96  % just in a faster way for large values of n.
97
98  % generate a vector of n values chosen u.a.r. from (0,1)
99  v = rand(1, n);
100 % Sort them and calculate the binomial cumulative distribution function
101 % with parameters n and p at values 0 to n. Afterwards match the
102 % sorted random 0-1-values to those cdf-values, i.e. associate a binomial
103 % distributed value with each value in r. Each value in v also
104 % corresponds to a value in r: permute the values in rowSize s.t.
105 % they correspond to the order given in v.
106 [r index] = sort(v); % i.e. v(index) == r holds
```

```
107 rowSize = zeros(1, n);
108 j = 0;
109 binoCDF = cumsum(binopdf(0:n, n, p));
110 for i = 1:n
111     while j<n && binoCDF(j+1)<r(i)
112         j = j + 1;
113     end
114     rowSize(i) = j;
115 end
116 rowSize(index) = rowSize;
117
118 % for every row choose the non-zero entries in it
119 nNZ = sum(rowSize);
120 I = zeros(1, nNZ);
121 J = zeros(1, nNZ);
122 j = 1;
123 for i = 1:n
124     I(j:j+rowSize(i)-1) = i;
125     J(j:j+rowSize(i)-1) = randsample(n, rowSize(i));
126     j = j + rowSize(i);
127 end
128
129 % restrict I and J to indices that correspond to entries above the main
130 % diagonal and finally construct a symmetric sparse matrix using I and J
131 upperTriu = find(I<J);
132 I = I(upperTriu);
133 J = J(upperTriu);
134 A = sparse([I;J], [J;I], ones(1, 2*size(I, 2)), n, n);
135
136 end % random_graph(...)
```

D Agents.m

```
% --------------------------------*- Matlab -*---------------------------------
% Filename: agents.m
% Description: It takes a matrix discribing a network and generates the
% agents.
% Author: Sebasitan Heinekamp
% Email: heinekas@student.ethz.ch
% Created: Sun Nov 13 20:05:26 2011 (+0100)
% Version:
% Last-Updated: Wed Nov 23 17:13:06 2011 (+0100)
% By:
% Update #: 0
% ------------------------------------------------------------------------------
% agents.m starts here
% ------------------------------------------------------------------------------
function [ agent ] = agents( networkmatrix, par  )
%AGENTS generates the agents out of a network.

[r c] = size(networkmatrix);
numberofagents = r;
agent = struct('citizen', [], 'state', [], 'nbr', [], 'threshold', []);
agent(numberofagents).citizen = 0;

% counts the number of already initialised rebls
totrebls =0;

for i=1:numberofagents

    % Determine in which country the citizen lives
    agent(i).citizen = 0;
    k = 1;
    maxNode = 0;
    while(agent(i).citizen == 0)
        maxNode = maxNode + par.nodes(k);
        if i <= maxNode
            agent(i).citizen = k;
        end
        k = k + 1;
    end

    % Set state
    % Says whether the agent is rioting (1) or not (2).
    agent(i).state = 0;

    % Get threshold
    % The threshold is randomly generated and can be set off in either
    % direction to favour the system or the rebls
    agent(i).threshold = (rand(1)+par.thresholdoffset)/(1+par.thresholdoffset);

    % handle fixed thresholds
    if rand(1) <= par.fixedthreshold(1)
        agent(i).threshold = par.fixedthreshold(2);
    end
```

```
    % Find connected agents (neighbours)
    % goes through the network and finds all the neighbours of the current
    % agent so the solver saves time.
    agent(i).nbr = [];
    for j=1:c,

        if networkmatrix(i,j) ~= 0

            agent(i).nbr = [agent(i).nbr j];

        end

    end

    % initialise roit origins
    % the number of rebles are placed in the network. How far the rioting
    % agents are apart depends on par.strech. for par.stretch = 2 or larger
    % the rebles are in the closest neighbourhood of each other, while for
    % small the rebles are placed further apart. for to small par.stretch
    % values the distance between the rebles might be larger than the
    % network so not all rebls are placed in the network.
    % for par.stretch smaller(or equal) to 0.5 no agents will beplaced at
    % all.
    % the rebles are caracterised by a threshold of 1. That means that they
    % will turn against the system as soon as they are updated.
    if agent(i).citizen == par.riot(1)
        if totrebls <= par.riot(2)
            if 0.5 <= (rand(1)*par.stretch)
                agent(i).threshold = 1;
                totrebls = totrebls +1;
            end
        end
    end




end


end
```

E solverSIRv3.m

```
% --------------------------------*- Matlab -*---------------------------------
% Filename: solverSIRv3.m
% Description: New, modified version of solverSIR.m, version 3
% Author: Fabian Wermelinger
% Email: fabianw@student.ethz.ch
% Created: Thu Dec  1 21:59:51 2011 (+0100)
% Version:
% Last-Updated: Wed Dec 14 09:12:18 2011 (+0100)
%           By: Fabian Wermelinger
%     Update #: 53
% -------------------------------------------------------------------------------
% solverSIRv3.m starts here
% -------------------------------------------------------------------------------
function [res, initStat, finalStat] = solverSIRv3( agent, par )
% Solve the opinion formation problem using a SIR model.
    if ( length(agent) ~= sum(par.nodes) )
        [stack, I] = dbstack( '-completenames' );
        error( errorMsg('WrongNumberOfNodes', stack) );
    end

    % create time vector
    t = linspace( par.time(1), par.time(2), par.nTime );

    % init residual vector(s)
    res = zeros( length(t), length(par.nodes) + 1 ); % +1 is for global res
    maxUpdatedAgents = int32( par.maxAgentUpdate*length(agent) );

    % get initial agent statistics
    initStat = fieldStat( agent );
    csvBasename = 'riotAgents';
    writeCSV( [csvBasename num2str(0) '.csv'], agent );

    % start iteration
    for i = 1:length( t )
        res(i,:) = calcResidual( agent, par );
        % generate a random list of agents to be updated in this time step.
        % The list has a random length, but at most par.maxAgentUpdate
        % (percent of total agents)
        aList = randi( length(agent), [randi(maxUpdatedAgents) 1] );
        nNoize = floor( par.noisyAgent*length(aList) );
        runSolver = true;
        if ( logical(nNoize) )
            for j = 1:nNoize
                agent(aList(j)).state = randi( [0 1], [1] ); % add noize
            end
            runSolver = false;
            if ( nNoize < length(aList) )
                aList = aList( (nNoize+1):length(aList) );
                runSolver = true;
            end
        end
        if ( runSolver )
```

```
53              for j = 1:length( aList )
54                  if ( ~logical(agent(aList(j)).state) && ...
55                       (1 - agent(aList(j)).threshold) <= ...
56                       nbrStateRes(agent(aList(j)), agent, par.nbrDepth) )
57                      if ( rand() <= par.beta(agent(aList(j)).citizen) )
58                          % SIR infection
59                          agent(aList(j)).state = 1; % if the neighbor
60                                                     % residual is
61                                                     % larger than the agent
62                                                     % threshold, set its mind
63                                                     % state to 1.
64                      end
65                  end
66                  if ( rand() < par.gamma(agent(aList(j)).citizen) )
67                      % SIR removal
68                      agent(aList(j)).state = 0;
69                  end
70              end
71              if ( ~mod(i, par.csvInterval) )
72                  writeCSV( [csvBasename num2str(i) '.csv'], agent );
73              end
74          end
75      end
76      % get final agent statistics
77      finalStat = fieldStat( agent );
78      return;
79
80  % ------------------------------------------------------------------------
81  % subroutines
82  % ------------------------------------------------------------------------
83  function stat = fieldStat( agent )
84  % Gather information of the agent field.
85      stat.nOnes = 0; % number of agents with a state 1
86      stat.nInitiator = 0; % number of initiator agents, i.e., agents with a
87                           % threshold of 1 and state 0
88      for i = 1:length( agent )
89          if ( logical(agent(i).state) )
90              stat.nOnes = stat.nOnes + 1;
91          else
92              if ( agent(i).threshold == 1 )
93                  stat.nInitiator = stat.nInitiator + 1;
94              end
95          end
96      end
97      return;
98
99  function stateRes = nbrStateRes( root, agents, depth )
100 % Calculate the residual of all neighbor states of the root agent.  The
101 % variable depth defines how many neighbors of neighbors shall be considered
102     global state k nbrCount; % used for the functions getTotalNbr and
103                              % getAllStates, which are both recursive
104     nbrCount = 0; % init neighbor counter
105     getTotalNbr( root, agents, depth ); % get total number of neighbors
106     state = zeros( nbrCount + 1, 1 ); % +1 is for the root agent itself and
107                                       % will not be needed later on
```

```
108     k = 0;
109     getAllStates( root, agents, depth ); % get the states of all neighbours
110                                          % and write it into the vector
111                                          % state
112     state = state(1:length(state) - 1); % cut off the last element since it
113                                         % is the state of root itself.
114                                         % However, only neighbor states are
115                                         % of interest.
116     maxRes = norm( ones(size(state)) ); % maximum residual, i.e., all
117                                         % neighbors have state 1
118     stateRes = norm( state )/maxRes; % calculate normalized residual of all
119                                      % neighbors of the root agent
120     return;
121
122 function getTotalNbr( root, agents, depth )
123 % Count the total number of agents, which are neighbors (and higher order
124 % neighbors, according to depth) of the root agent.  Write result into
125 % global variable nbrCount.  This function is recursive.
126     global nbrCount
127     if ( depth == 0 )
128         return;
129     elseif ( depth == 1 )
130         nbrCount = nbrCount + length( root.nbr );
131         return;
132     else
133         for i = 1:length( root.nbr )
134             getTotalNbr( agents(root.nbr(i)), agents, depth - 1 );
135         end
136         nbrCount = nbrCount + length( root.nbr );
137         return;
138     end
139
140 function getAllStates( root, agents, depth )
141 % Get the states of all neighbors (and higher order neighbors, according to
142 % depth) of the root agent. Write the states into the global variable
143 % state.  This is a recursive function.  The very last state is the one from
144 % the root agent itself.
145     global state k
146     if ( depth == 0 )
147         k = k + 1;
148         state(k) = root.state;
149         return;
150     else
151         for i = 1:length( root.nbr )
152             getAllStates( agents(root.nbr(i)), agents, depth - 1 );
153         end
154         k = k + 1;
155         state(k) = root.state;
156         return;
157     end
158
159 function res = calcResidual( agent, par )
160 % calculate residual for each cluster as well as global (all clusters
161 % together).  The residual is a scalar between 0 and 1 which describes the
162 % overall behavior of a social network (cluster).  A global network may
```

```
% contain multiple connected (or non-connected) cluster.
    res = cell( 1, length(par.nodes) + 1 );
    for i = 1:length( par.nodes )
        res{i} = zeros( par.nodes(i), 1 );
    end
    res{length(res)} = zeros( length(agent), 1 );

    % init array addressing
    counter = ones( size(res) );
    for i = 1:length( agent )
        % local residual vector entry for agent(i)
        res{agent(i).citizen}(counter(agent(i).citizen)) = agent(i).state;
        counter(agent(i).citizen) = counter(agent(i).citizen) + 1;
        % global residual entry for agent(i)
        res{length(res)}(counter(length(counter))) = agent(i).state;
        counter(length(counter)) = counter(length(counter)) + 1;
    end
    % calculate maxRes for normalization
    maxRes = zeros( size(res) );
    for i = 1:length( par.nodes )
        maxRes(i) = norm( ones(par.nodes(i), 1) ); % local
    end
    maxRes(length(maxRes)) = norm( ones(sum(par.nodes), 1) ); % global

    % normalize
    tmpRes = zeros( size(res) );
    for i = 1:length( res )
        tmpRes(i) = norm( res{i} )/maxRes(i);
    end
    res = tmpRes;
    return;

function writeCSV( filename, agent )
% write every agent with state 1 to .csv file
    fid = fopen( filename, 'w' );
    for i = 1:length( agent )
        if ( logical(agent(i).state) )
            fprintf( fid, '%i\n', i );
        end
    end
    fclose( fid );
    return;

% --------------------------------------------------------------------------
% solverSIRv3.m ends here
% --------------------------------------------------------------------------
```