Serialization and Stability Analysis in Swarm

Paul E. Johnson
Dept. of Political Science
University of Kansas
pauljohn@ku.edu

Overview

- Definitions
- Desirability
- How To

Serialization

Definition:

- Save state of simulation
- Restore state of simulation
- Familiar problem for computer game authors

Stability

Stability:

Study Impact of:

- changes in parameters/inputs
- changes in agent behavior rules/information

desirability 1

Save Time

- Take a slow model
- Save its progress up to a point
- Reload and explore

desirability 2

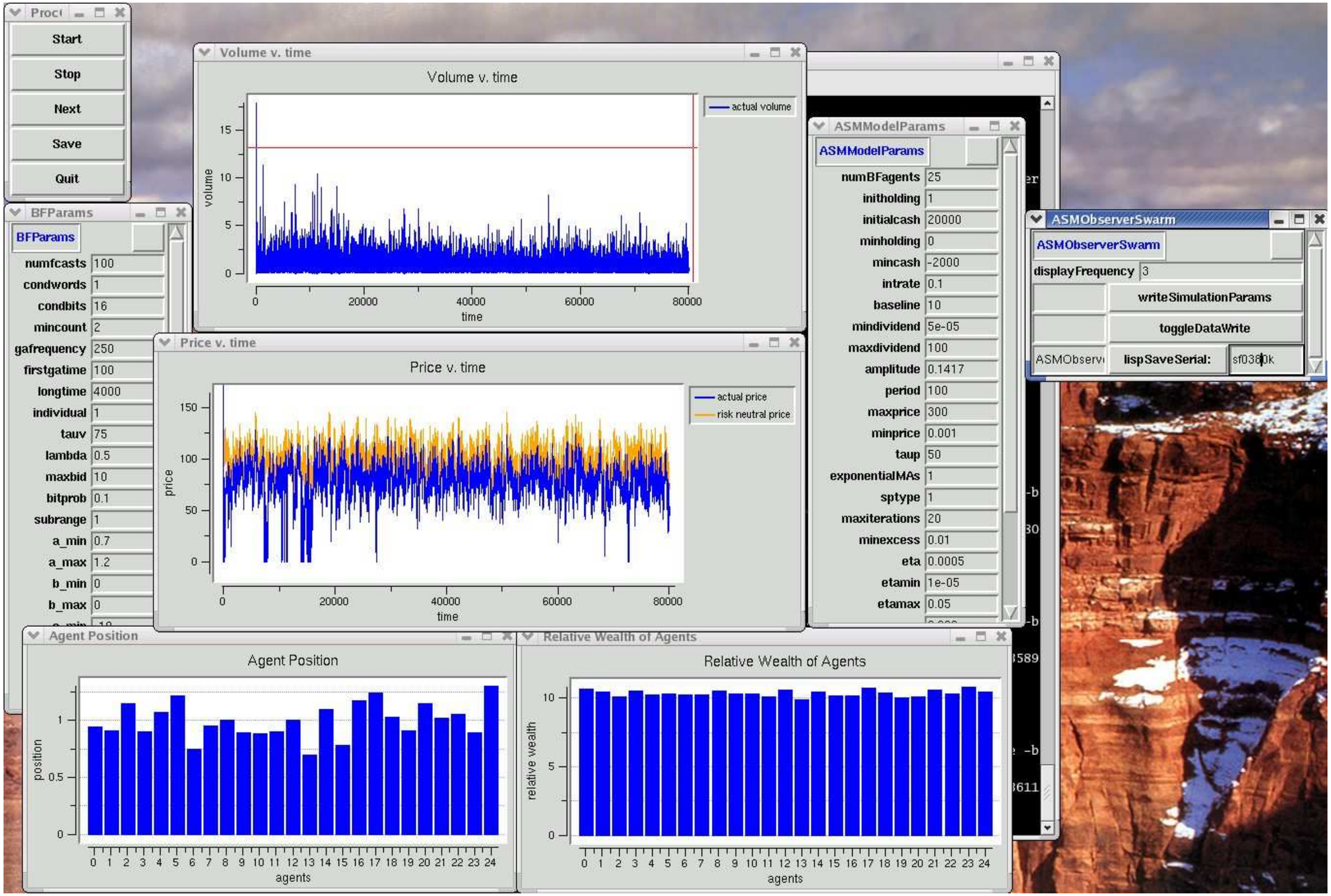
- Test stability of “emergent” property
- GUI can explore alternative “what if” scenarios
- Do rigorous re-analysis in batch mode

Example I: ASM

Example: Artificial Stock Market

- http://ArtStkMkt.sourceforge.net
- ASM-2.4 will introduce serialization
- Each run takes several hours: 300,000+ iterations

After 3 hours...

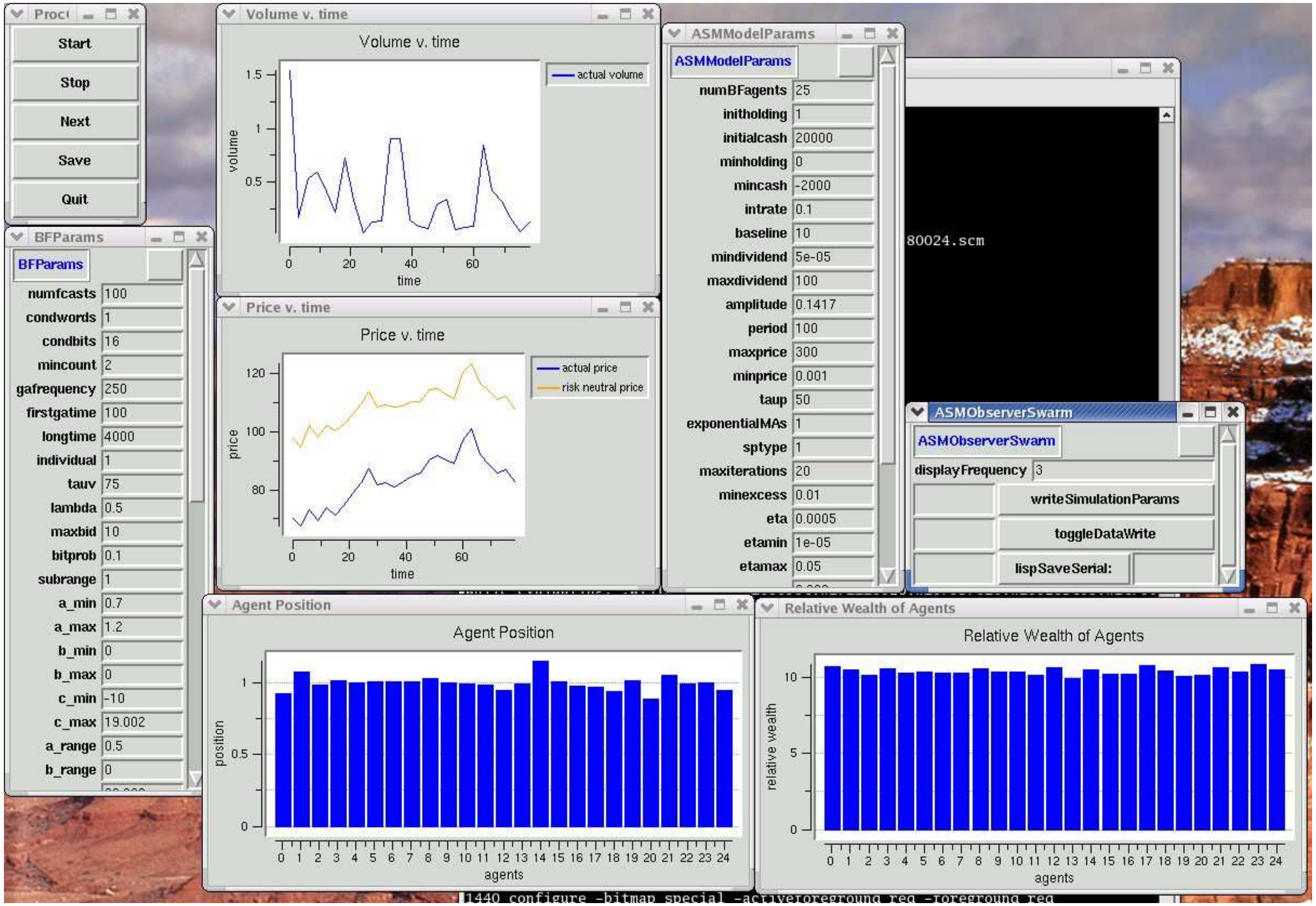


ASM2

ASM (Continued)

- It takes so long because agents are learning how to understand the market.
- “Rational Expectations” equilibrium emerges.
- Is equilibrium “upset” by changes in agent behavior?

ASM Restored from file

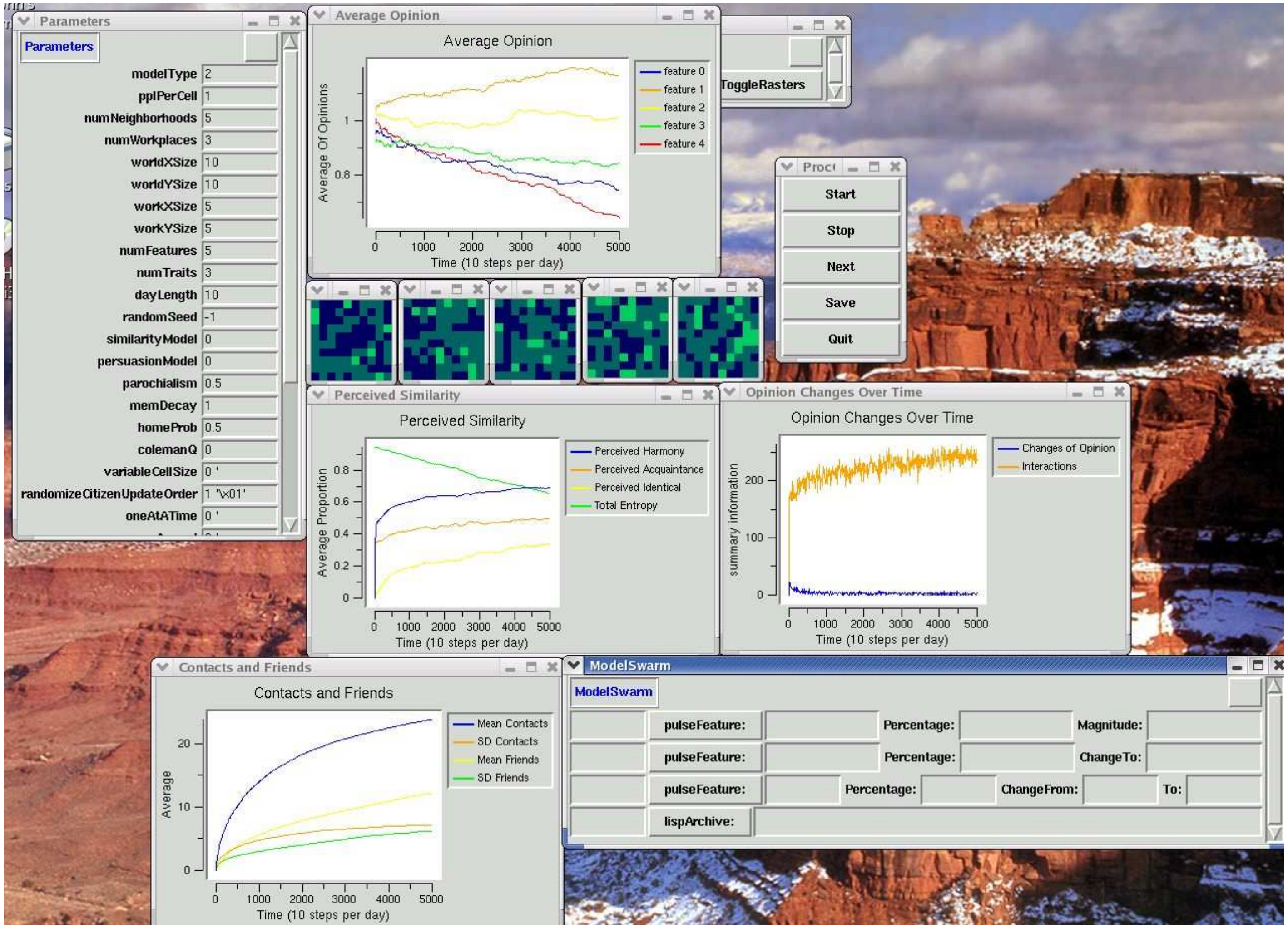


Example II: Opinion

Public Opinion Networks

- Huckfeldt, Johnson, & Sprague (J. of Politics, 2003; Autoregressive Impact in Social Networks)
- People interact, exchange opinions, adjust their views.

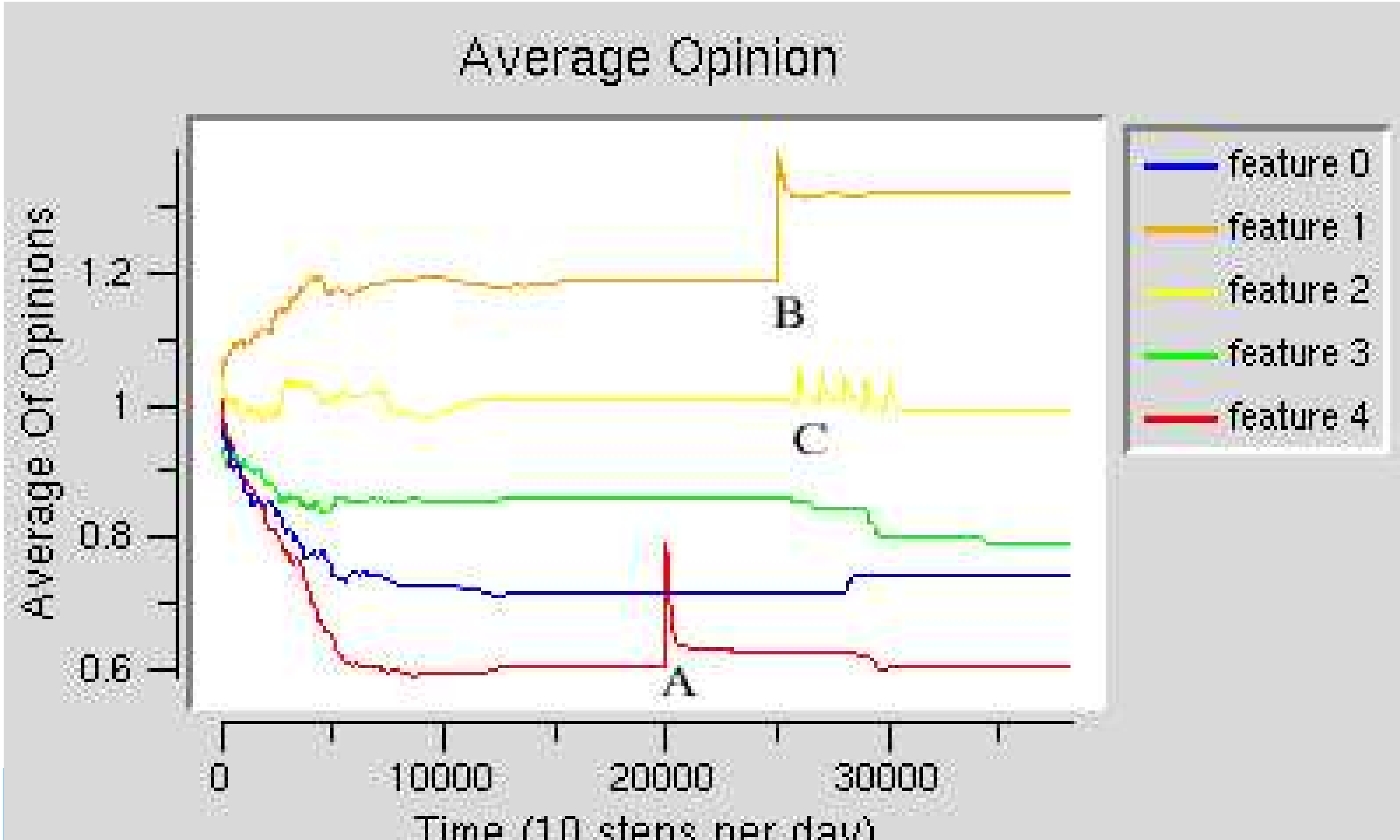
Opinion Model



Opinion 2

GUI stability analysis:

A,B,C represent “interactive tweaks”



How To

How To?

- Swarm supports serialization in hdf5 and lisp data formats
- lisp yields a *.scm file familiar to Swarm users.
- Swarm-2.2 upgrade necessary to
  - save Swarm Arrays (and objects in them)
  - save dynamically allocated arrays of integers and doubles

Recall “getWithZone:key”

- Tutorial introduces “getWithZone:key:” as a way to create instances of objects according to values designated in a *.scm file.
- End result: object created, the “createBegin:” and “createEnd” methods of your class are never called.

Shallow Save

Put Shallow

- save ints, doubles, characters, static arrays of same
- does not save “objects”, like Grid2d, List, Schedules, pointers & dynamic memory

Deep Save

Put Deep:

- save int, double, characters, etc
- Attempts to save objects and Swarm things like Collections (Lists, SwarmArrays, Maps) and Spaces (Grid2d).
- Recursive: Any object that has a “lispOutDeep:” method will be saved.

Deep Save

- Does not seamlessly understand (ignores):
  - dynamically allocated memory
  - pointers to objects that don't answer to "lispOutDeep:"

Challenges 1

Naive Approach: Put Deep a whole model

Try this in the “Model Swarm” level of Heatbugs (or just about any Swarm Model):

```
id dataArchiver = [LispArchiver create: [self getZone] setPath:
“myFile.scm”];
[dataArchiver putDeep: "model" object: self];
[dataArchiver sync];
[dataArcviver drop];
```

Challenges 2

Very unsatisfactory result:

- It tries to Save Everything in Model:
  - agent list
  - Grid2d (and agents in there)
- Redundant Copies of agents are saved in both the agent list and Grid2d.
- agents have a reference to the Grid2d world inside them, and it will attempt to save that.

Challenges 3

But Put Shallow does not save enough information.

Proposed Solution

We need a way to fully save the:

- state of the agents one-time-only
- any parameters needed to fully recreate the model {Grid, Lists, Data Structures, etc}.

Recommended Strategy

1. Save parameter objects shallow
2. Save agent list deep.

Example from Model Swarm

In ModelSwarm.m:

```
id dataArchiver = [LispArchiver create: [self getZone] setPath:
dataArchiveName];
[dataArchiver putShallow: "model" object: self];
[dataArchiver putShallow: "parameters" object: parameters];
[dataArchiver putDeep: "agentList" object: agentList];
[dataArchiver sync];
[dataArchiver drop];
```

It works its way down

putDeep is RECURSIVE.

“lispOutDeep:” on agentList will trigger

- save of agentList object
- triggers “lispOutDeep” for each object in agentList
- and recursively for variables each agent,
- and for each object in each object, etc.

putDeep: Method in subclasses

’Barefoot’ approach for a class called “Friend”

```
- (void)lispOutDeep: stream
{
  [stream catStartMakeInstance: "Friend"];
...insert commands to save variables here
  [stream catEndMakeInstance];
}
```

Example:

Suppose a class has a dynamically allocated array "culture", no other complications

```
- (void)lispOutDeep: stream {
  [stream catStartMakeInstance: "Attribute"];
    [super lispOutVars: stream deep: NO];    //save IVARs!!
      [super lispStoreIntegerArray: culture Keyword: "culture" Rank: 1
                   Dims: &numCultureFeatures Stream: stream];
  [stream catEndMakeInstance];
}
```

Complications

Some classes get more complicated because of inheritance and different kinds of variables.

Complications 1

Complications: Data Structures inside Agents.

- Map of “Attributes” using other agents as keys.
- Redundancy: Standard “lispOutDeep” for Map will deep-save keys and data.
- Necessary to redesign simulation so that keys are not duplicate objects.
- Can’t have anything “interesting” in createEnd!

Complications 2

Complications: Inheritance chain of Agents.

I create 2 kinds of methods

1. lispOutDeep: to start/end the instance by name
2. bareLispOutDeep: just does work of storing data, does not start/end instance

Example

```
- (void)lispOutDeep: stream
{
  [stream catStartMakeInstance: "HJCitizen"];
  [super bareLispOutDeep: stream];
  [self bareLispOutDeep: stream];
  [stream catEndMakeInstance];
}
```

Reset 1

Resetting the model to its last state.

In buildObjects:

1. Recreate space and other structures from stored parameters.

main.m: if input file given, recreate Parameters object from the saved file.

ModelSwarm.m uses those Parameters to create new Grids and other structures that match needs.

2. Recreate agents from saved file.

ModelSwarm.m: if input file is given, restore agent list from saved file.

Reset 2

In the ModelSwarm's "lispLoadAgents:"

- read in the collection of agents
- iterate over agents to restore references
  - setWorld:
  - tell each agent to do whatever is needed to restore its information structures about the world. (Survey neighborhood, etc)
- Don't run any "init" methods that are only needed on the first-run of the model.

Stability in the opinion model

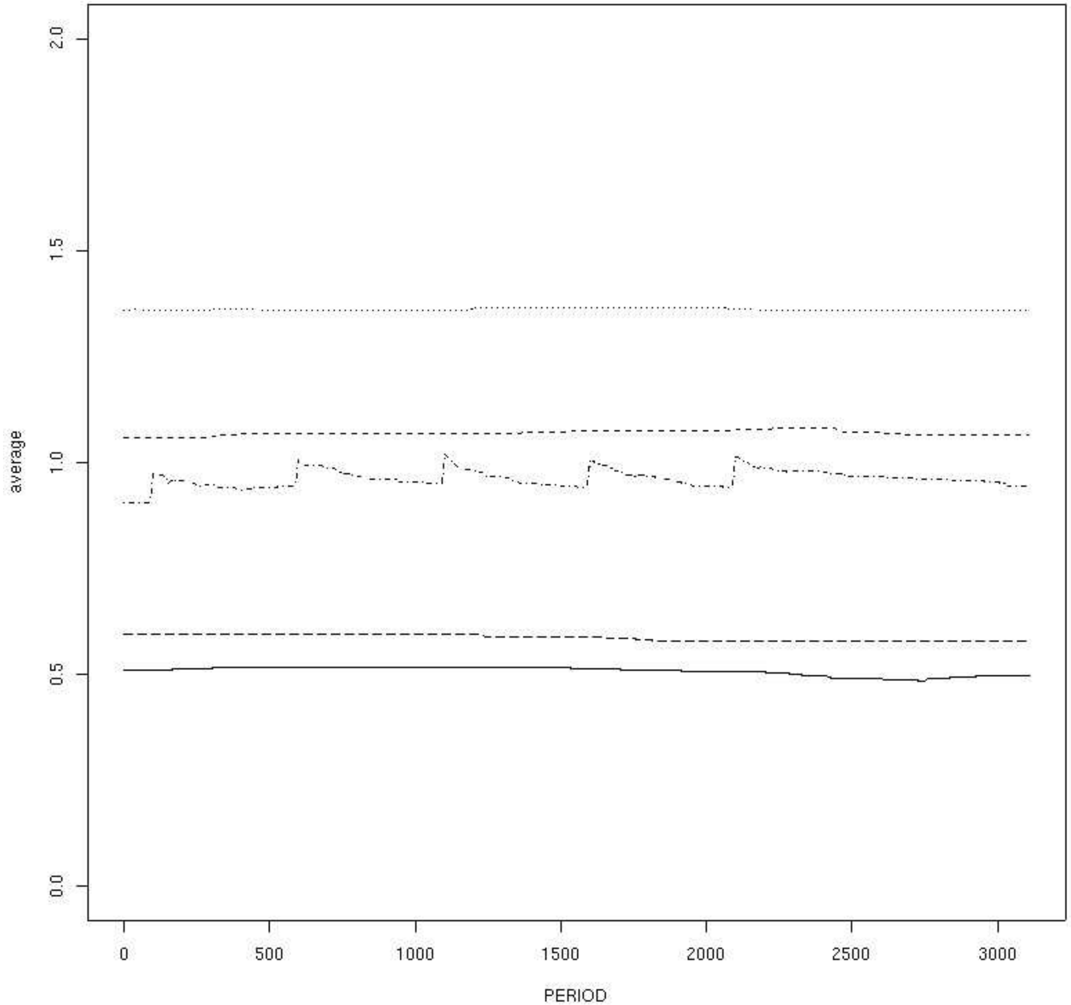
A model is run to its equilibrium state.

- Equilibrium means that no single agent has changed any opinion for 10 full cycles through the society.
- Each agent who is exposed to a contrary point of view does not change because a majority of that agent's "friends" do not support the newly suggested opinion.
- Save the state of the simulation

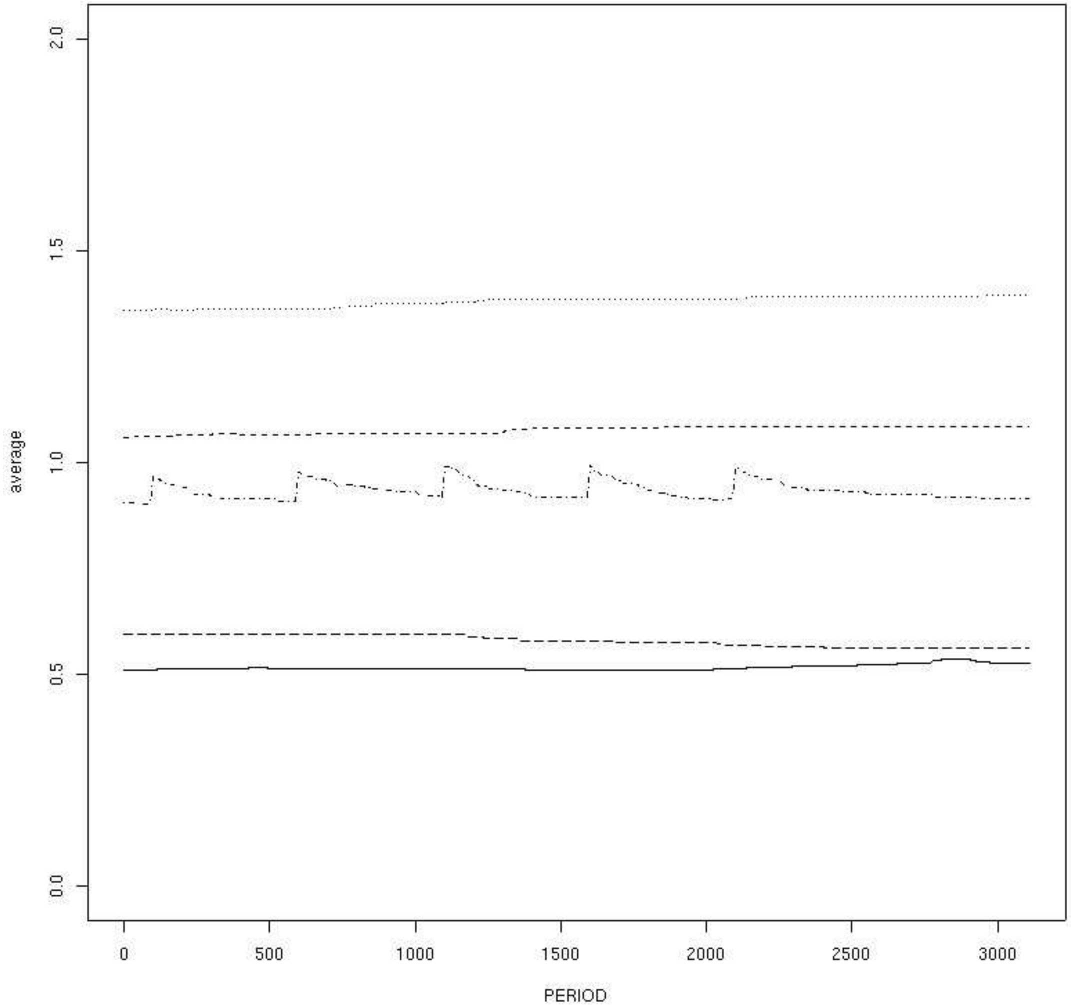
Shock the same outcome 20 times

- Then it is restarted repeatedly and the networks are subjected to random shocks.
- At times 10, 60, 110, 160, 210, and 260, 5 percent of the agent opinions are changed.
- If the opinion is 0 or 2, it is changed to one.
- Note how the networks absorb the shock in a very similar way across replications

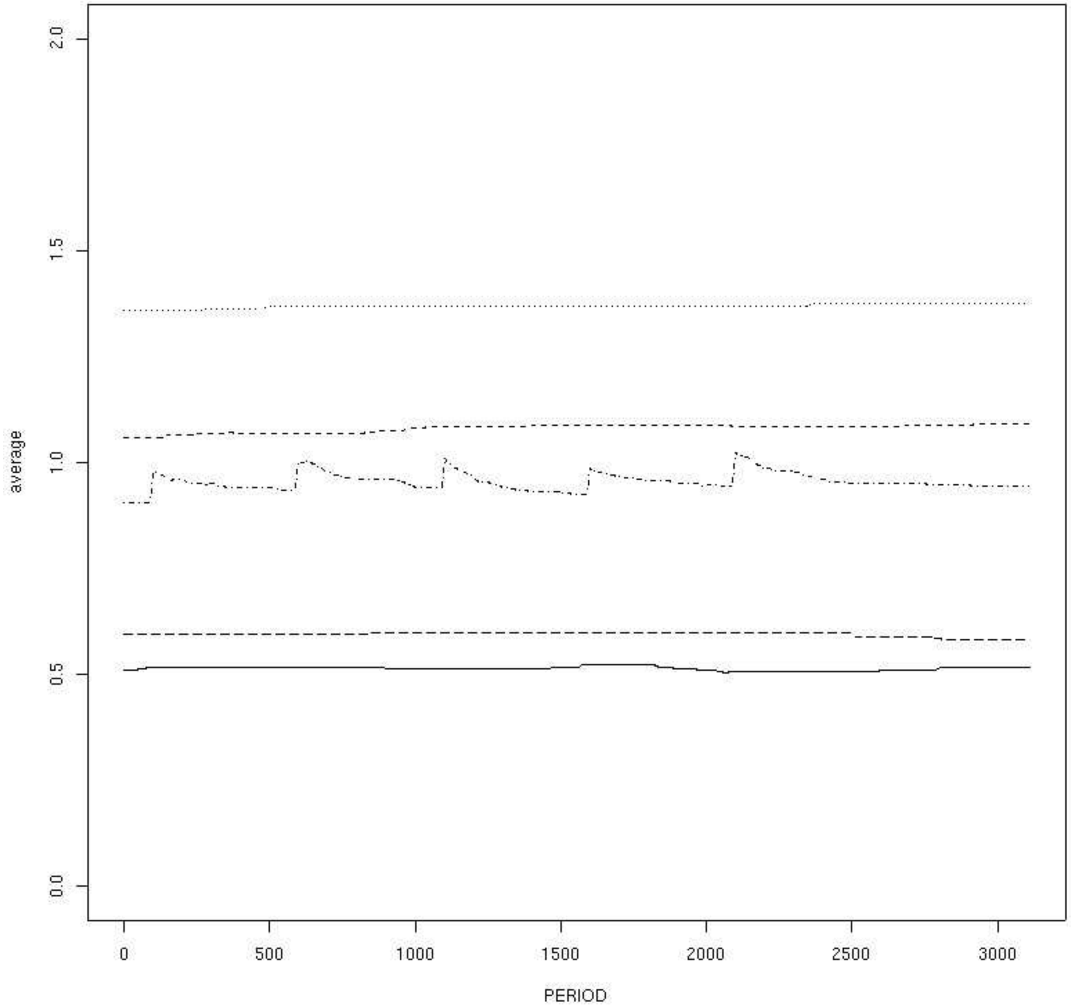
1



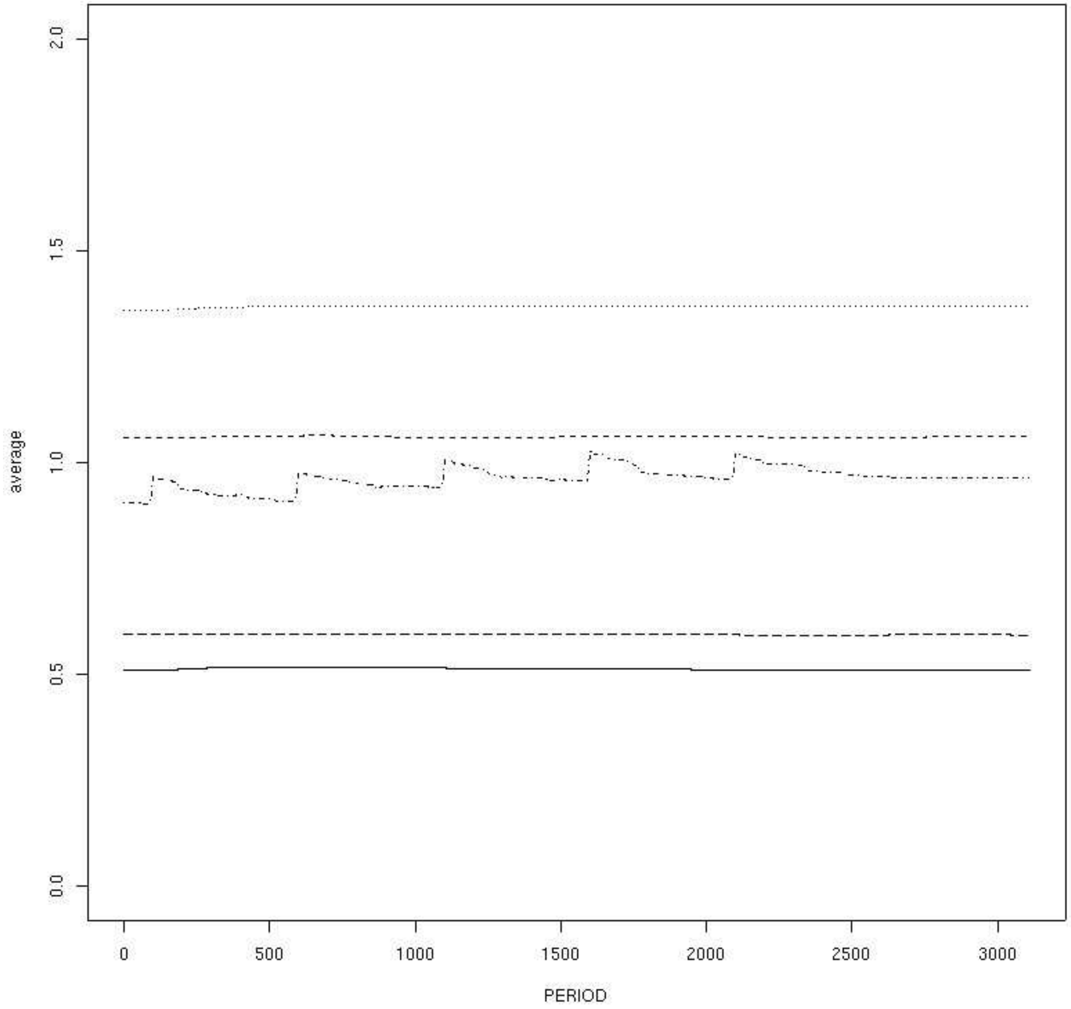
2



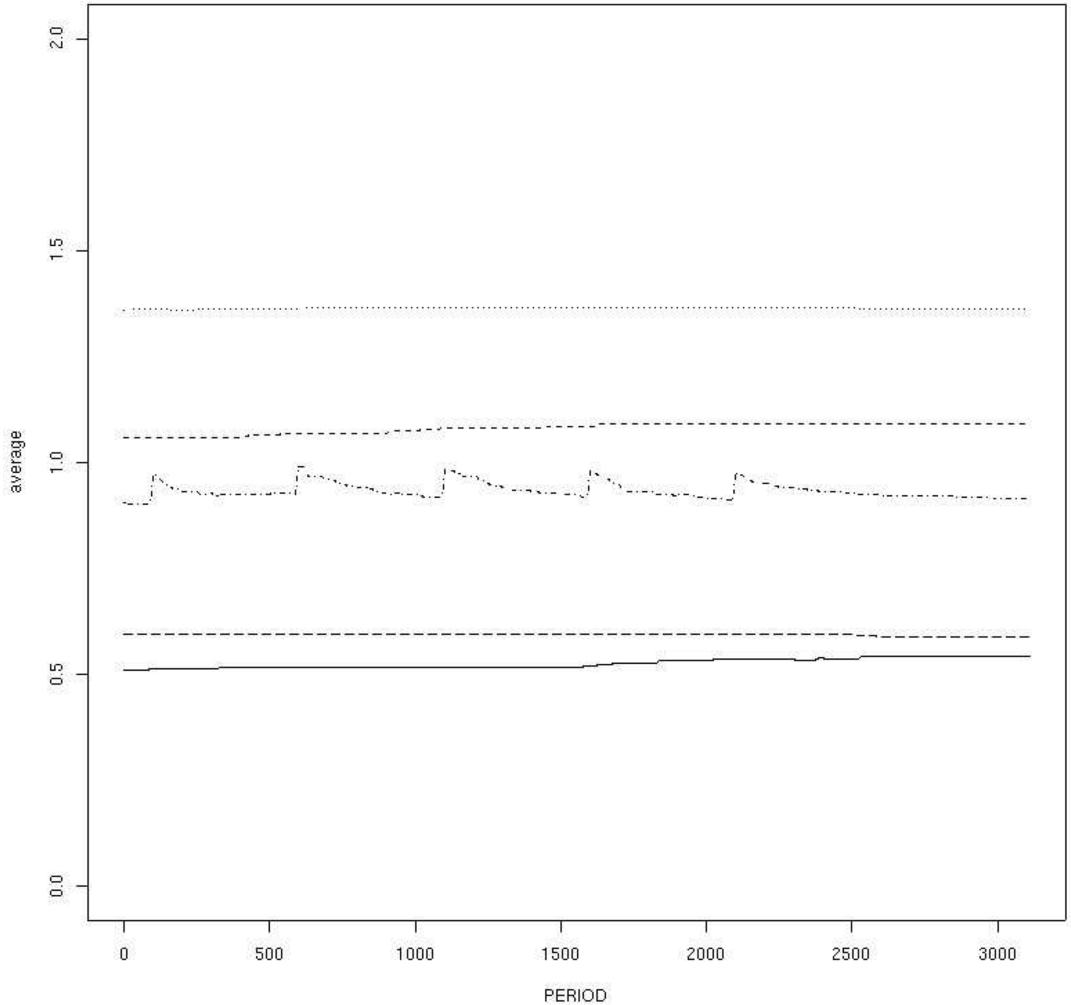
3



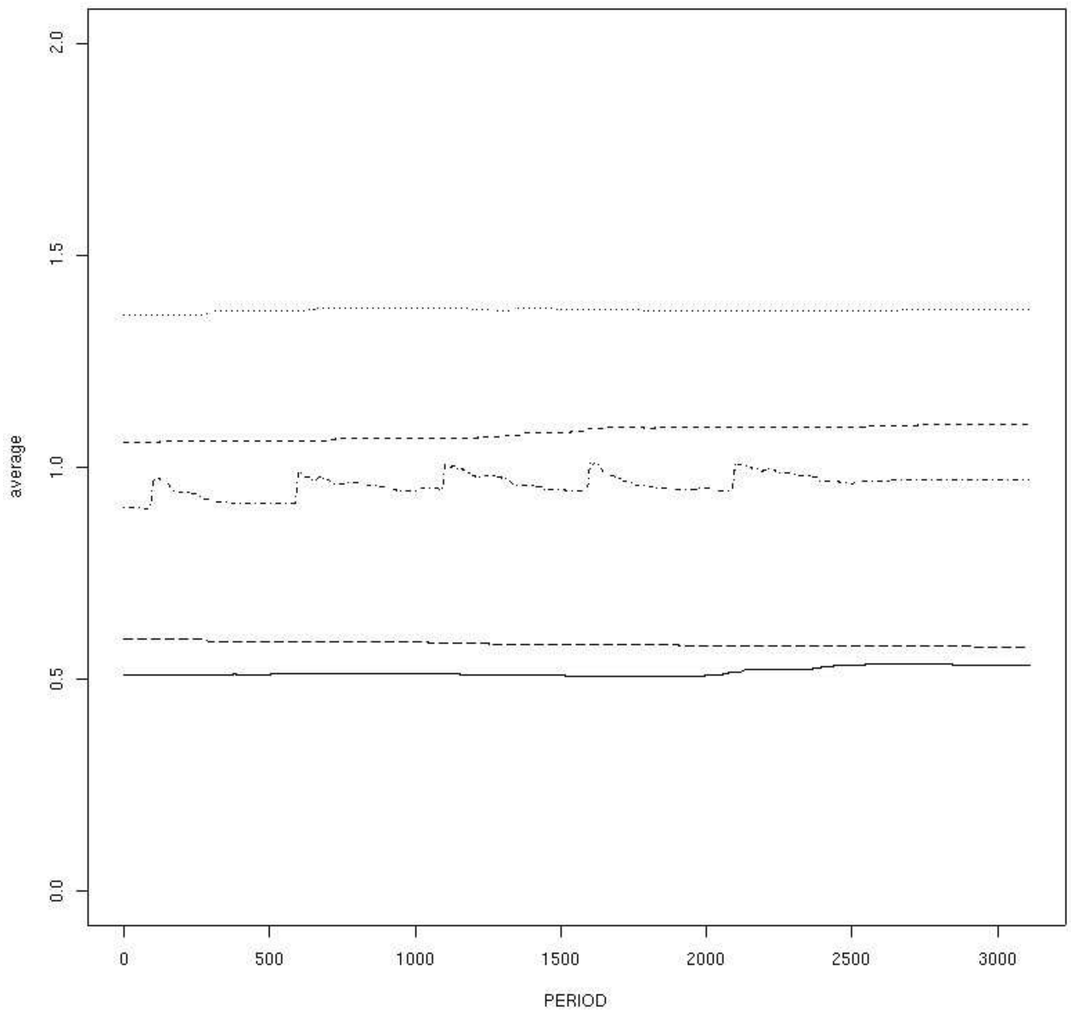
4



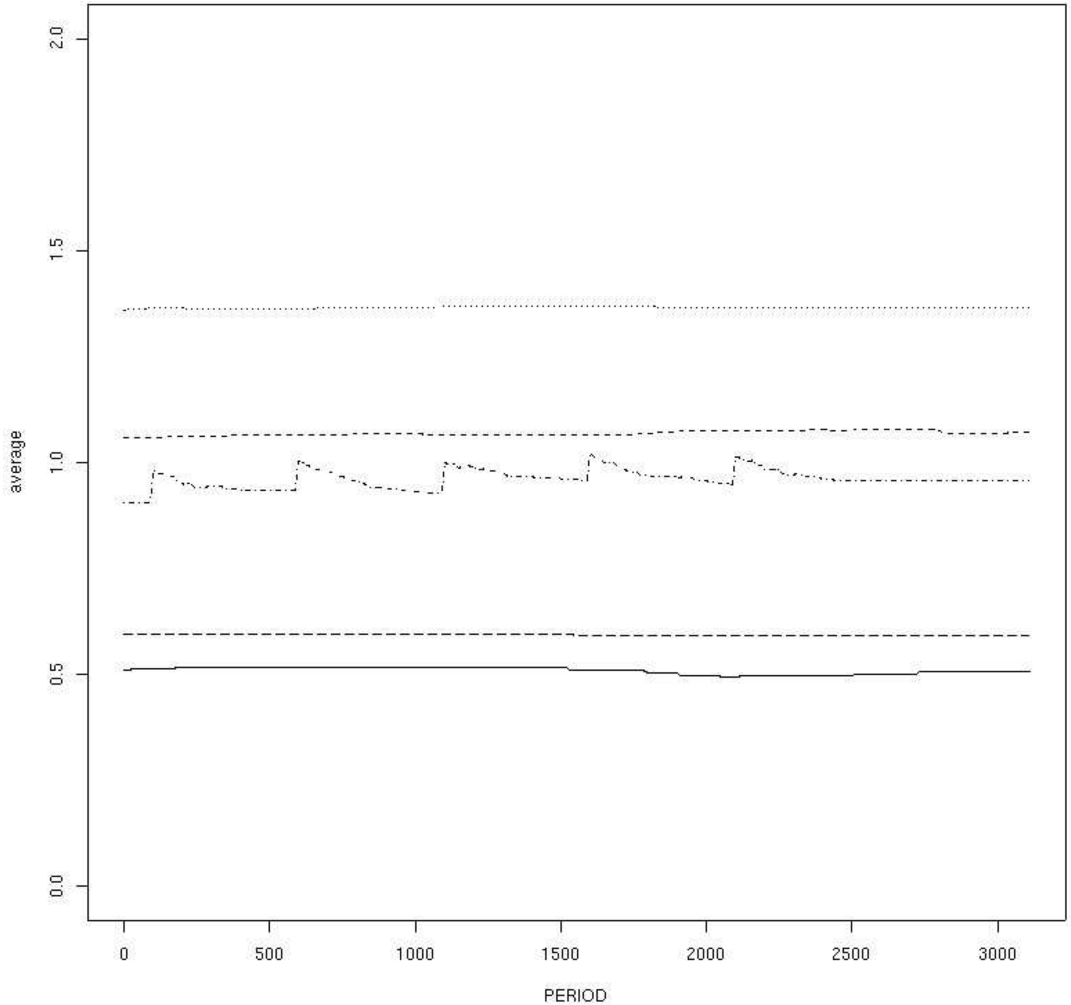
5



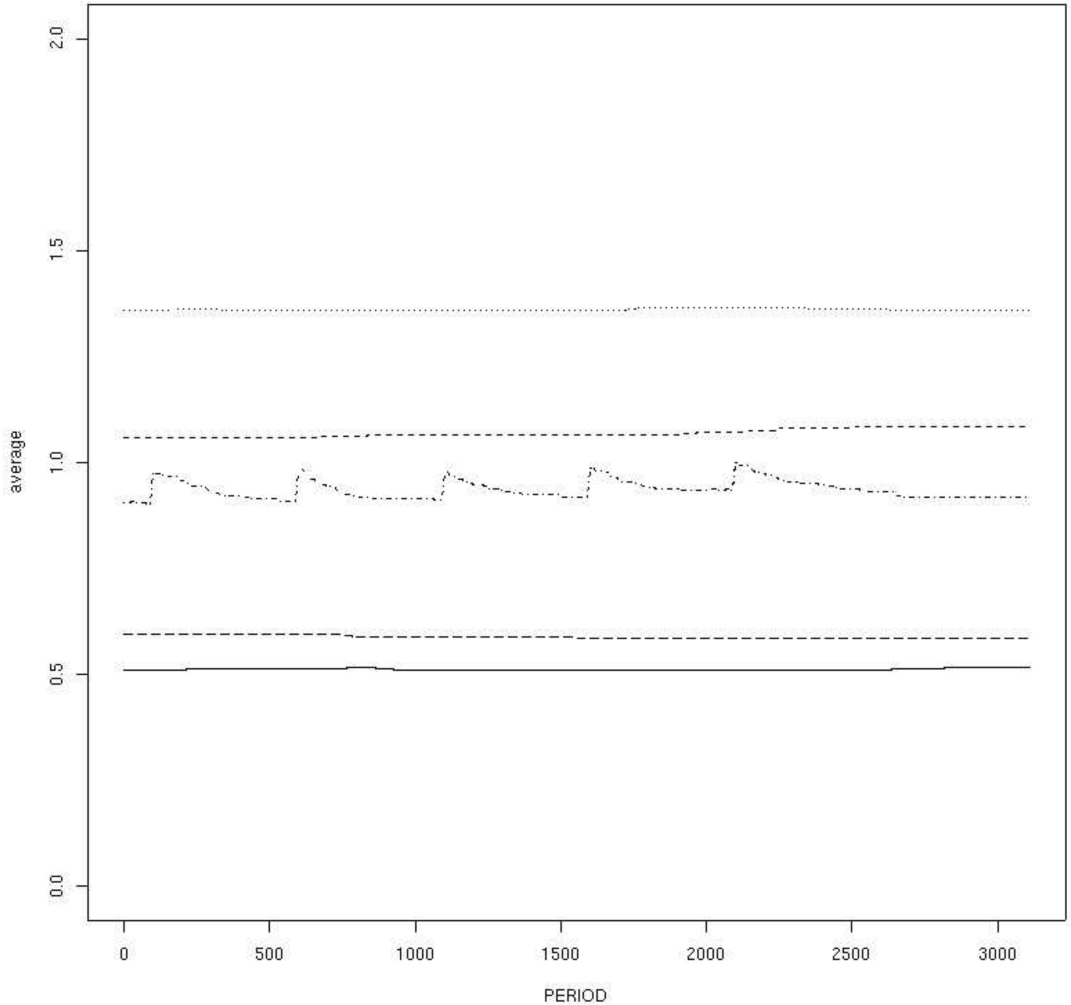
6



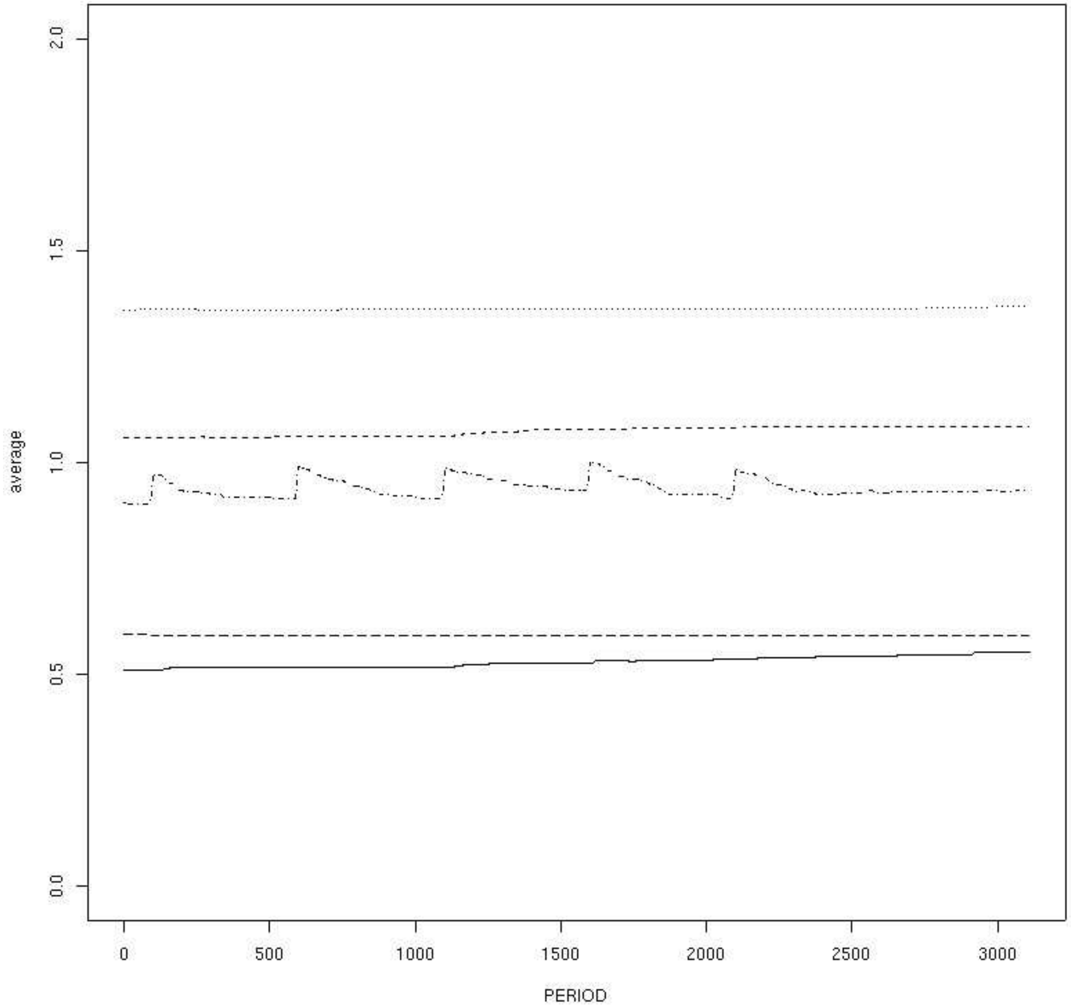
7



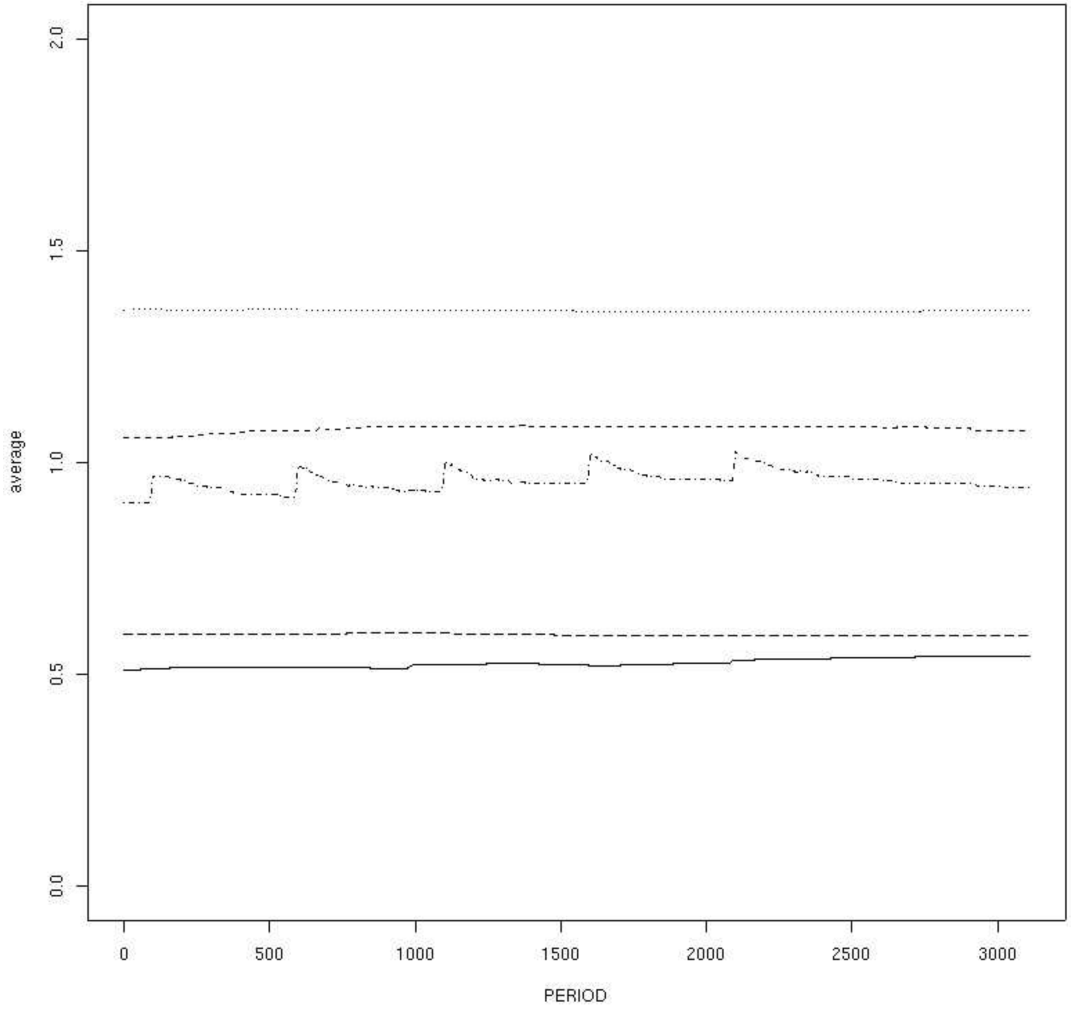
8

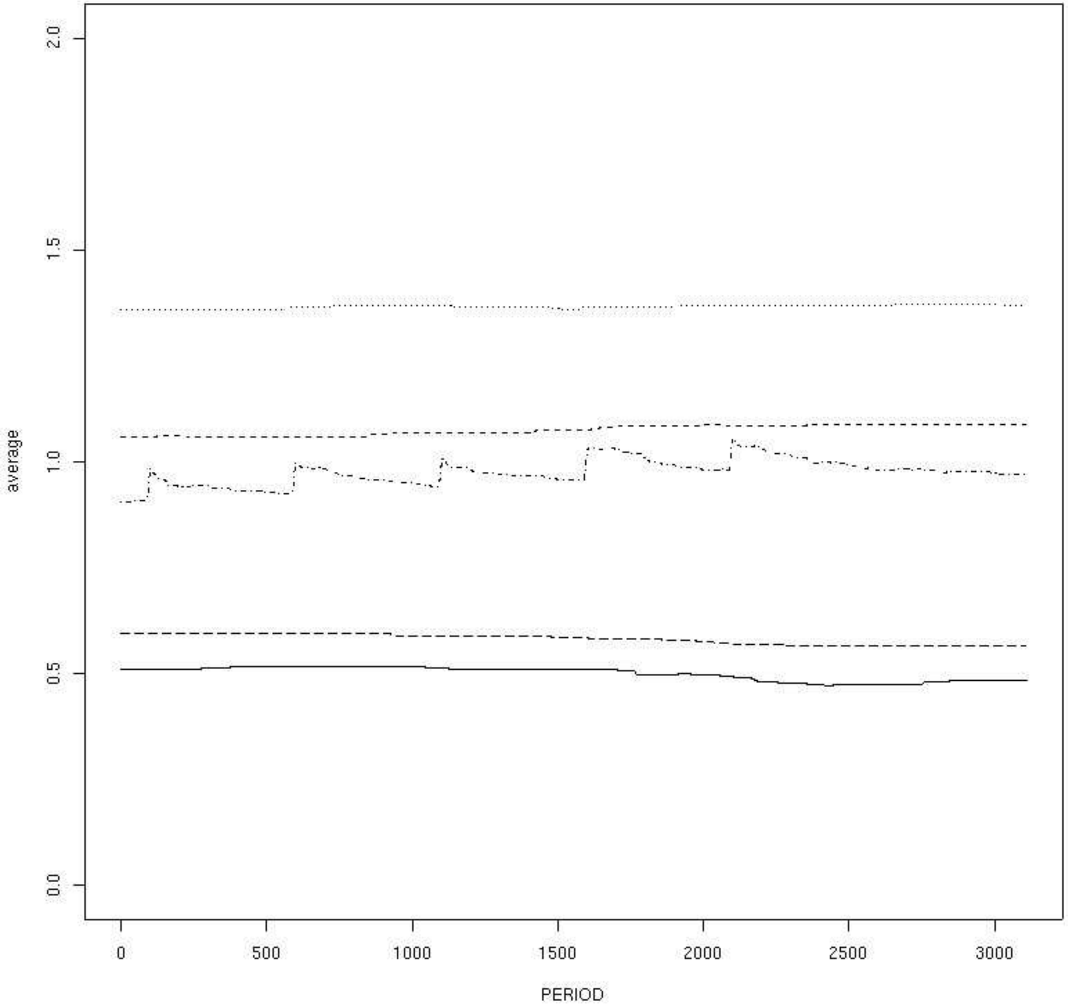


9



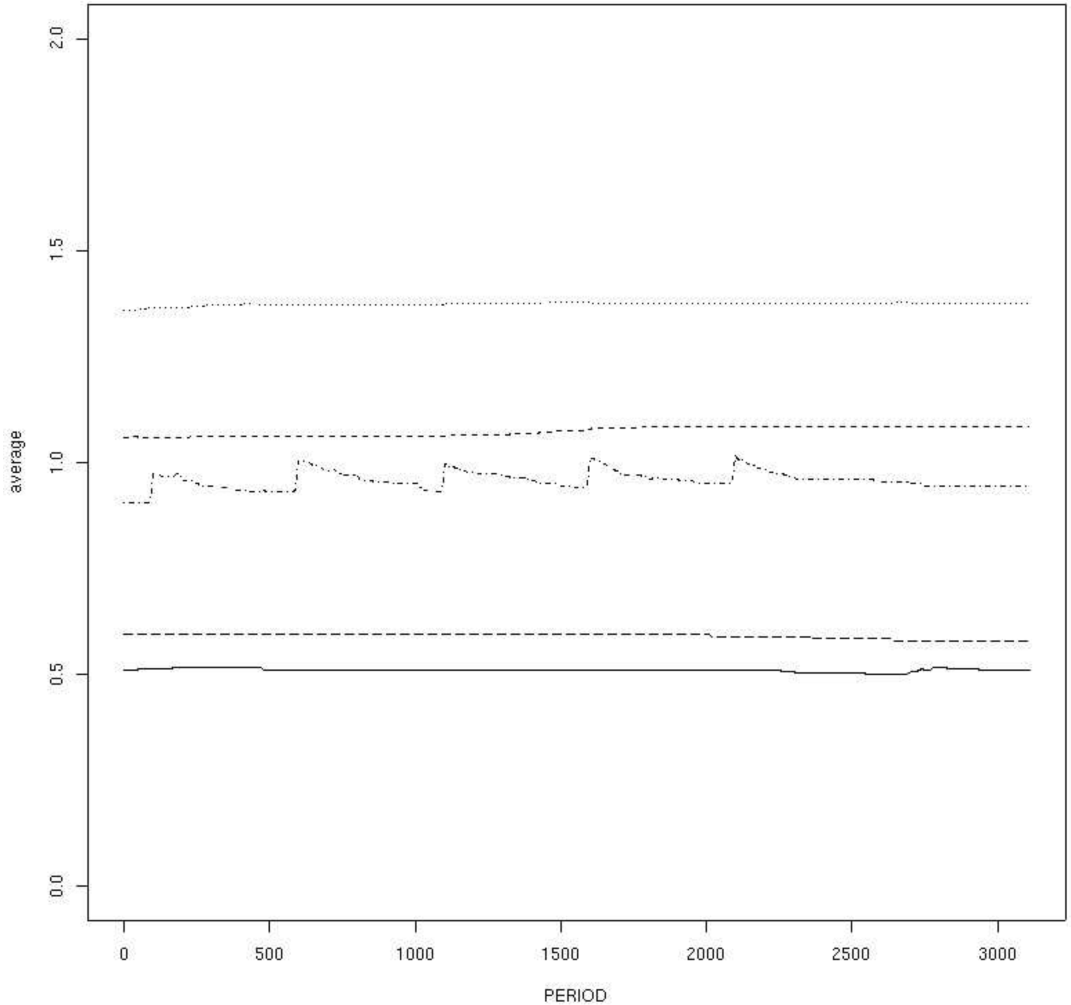
10



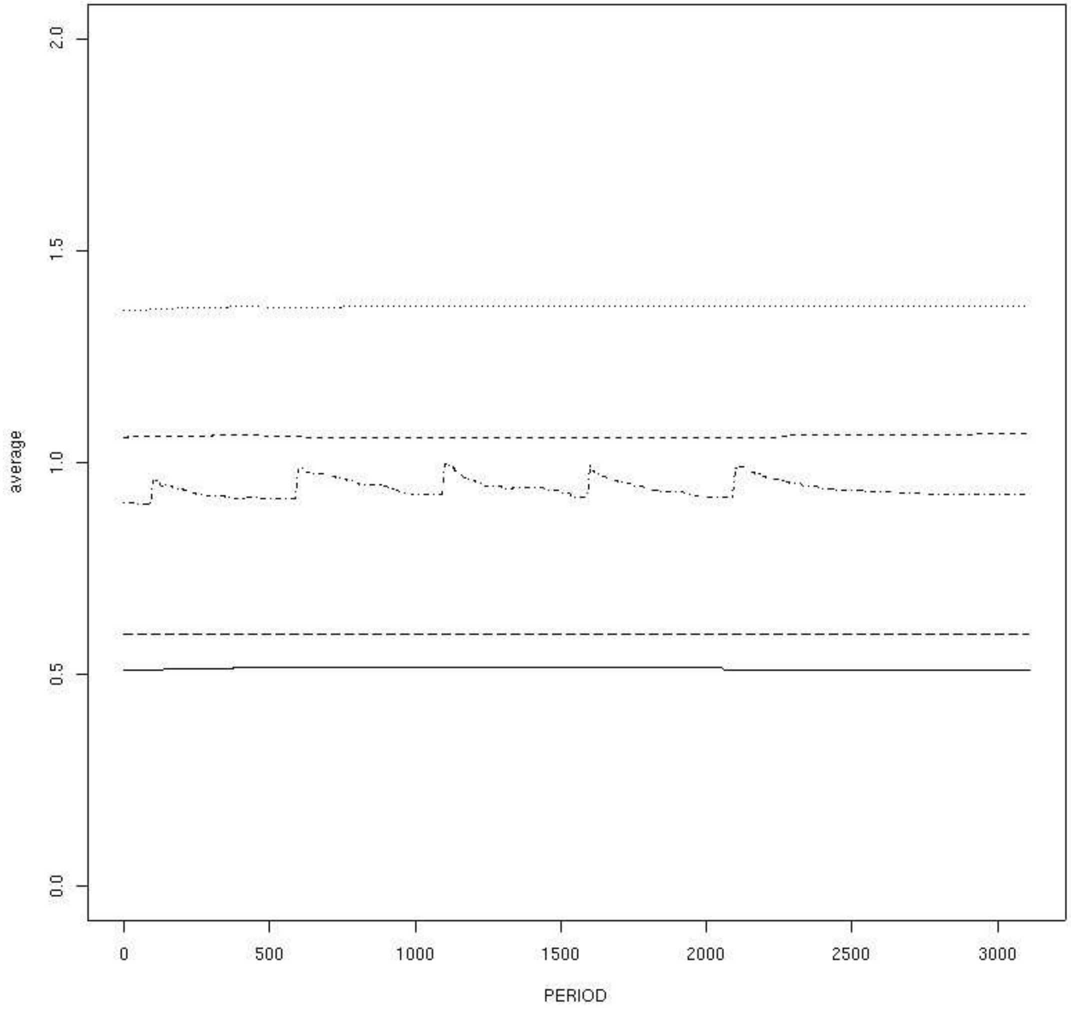


2.0
1.5
1.0
0.5
0.0
average
0
500
1000
1500
2000
2500
3000
PERIOD

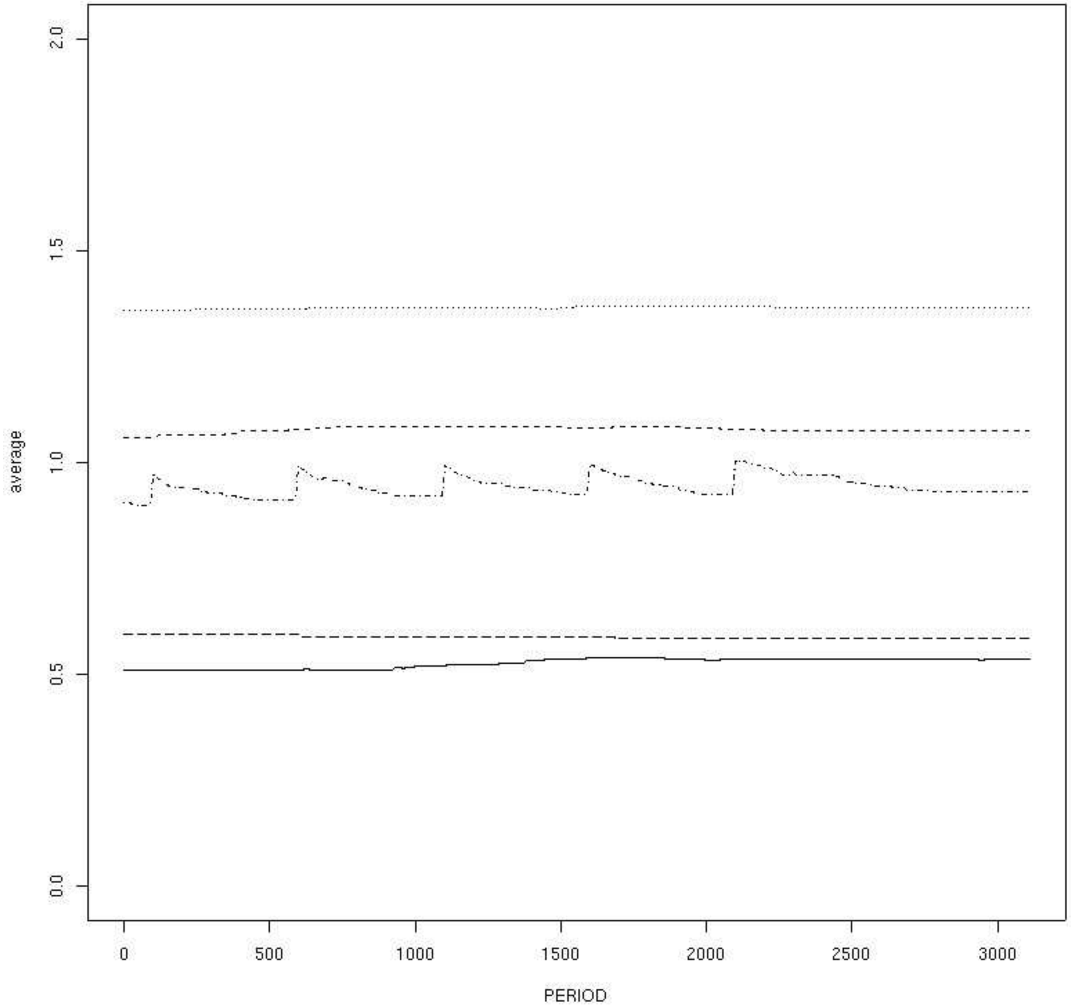
12



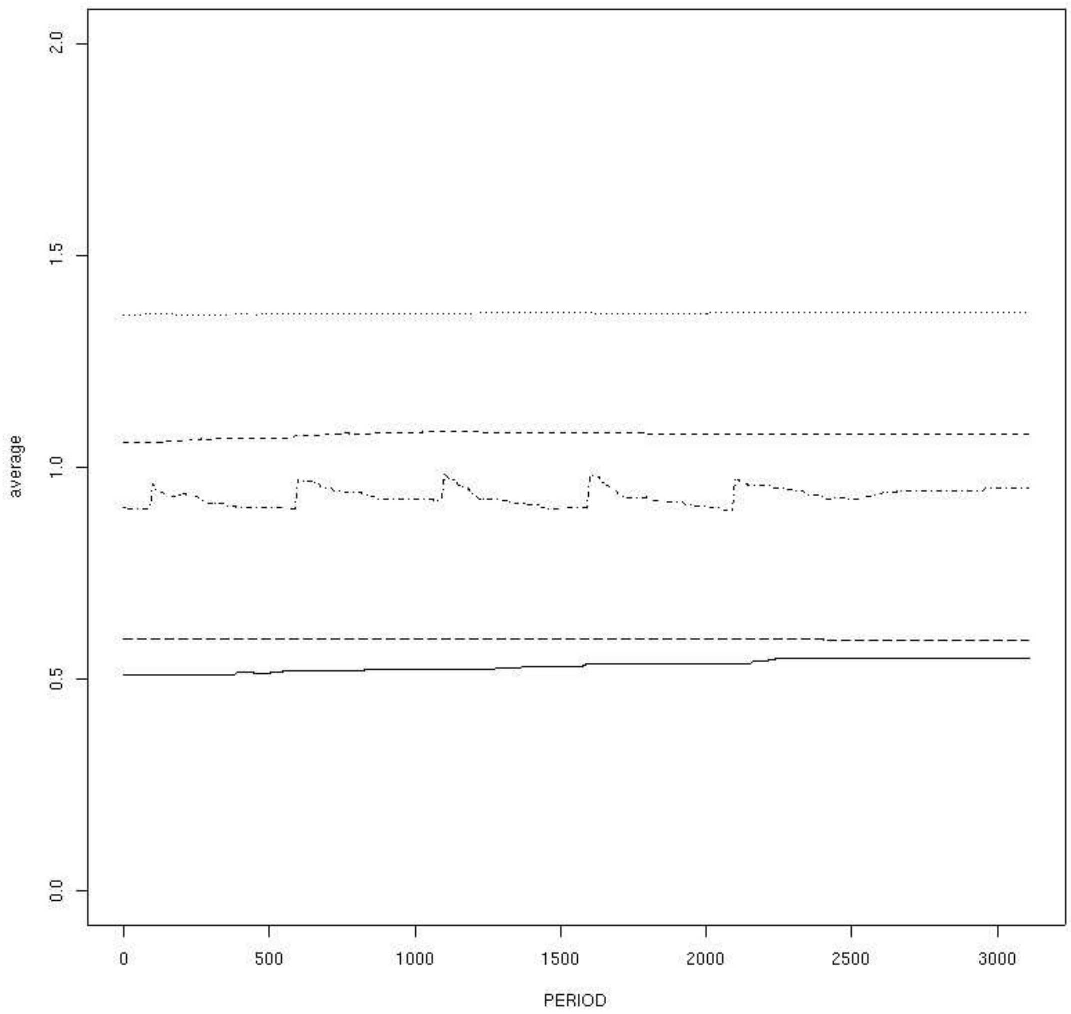
13



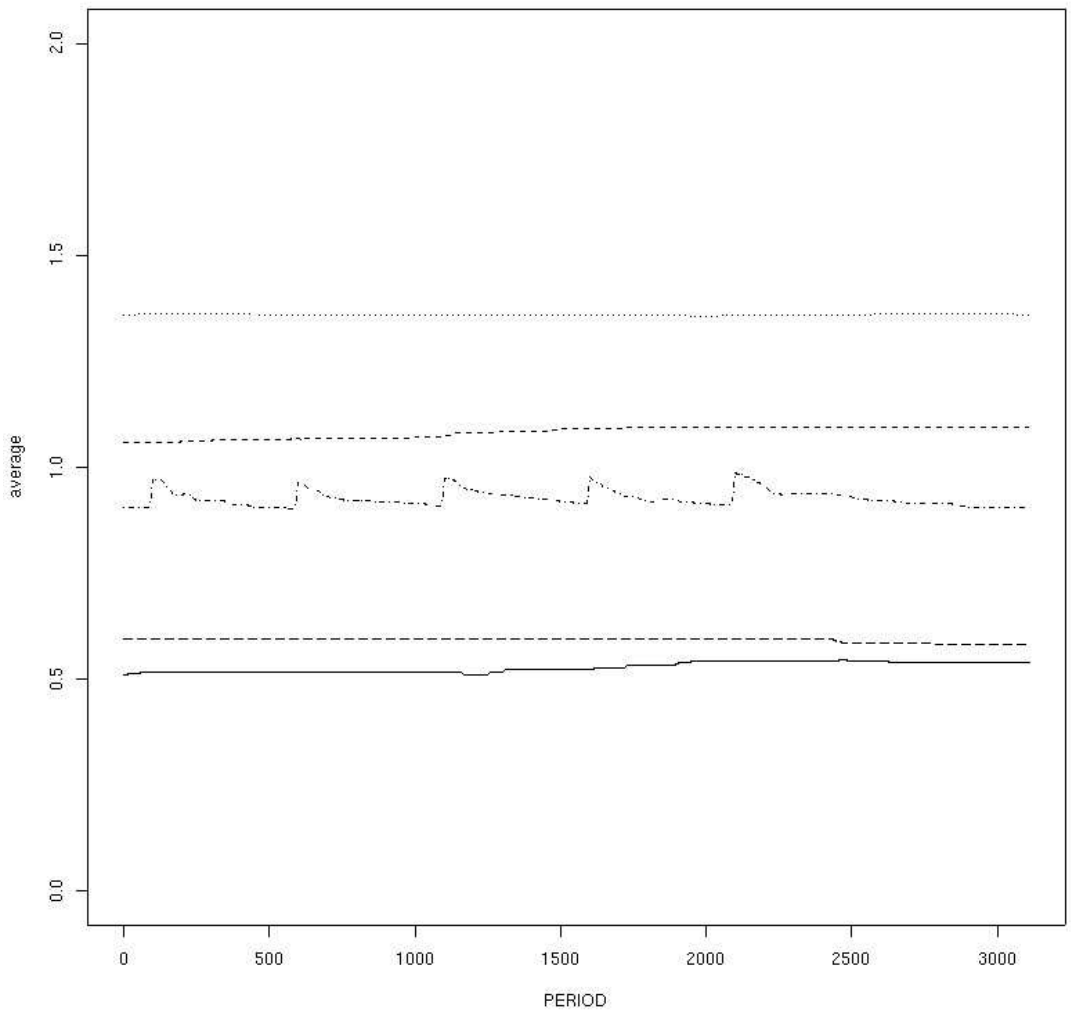
14



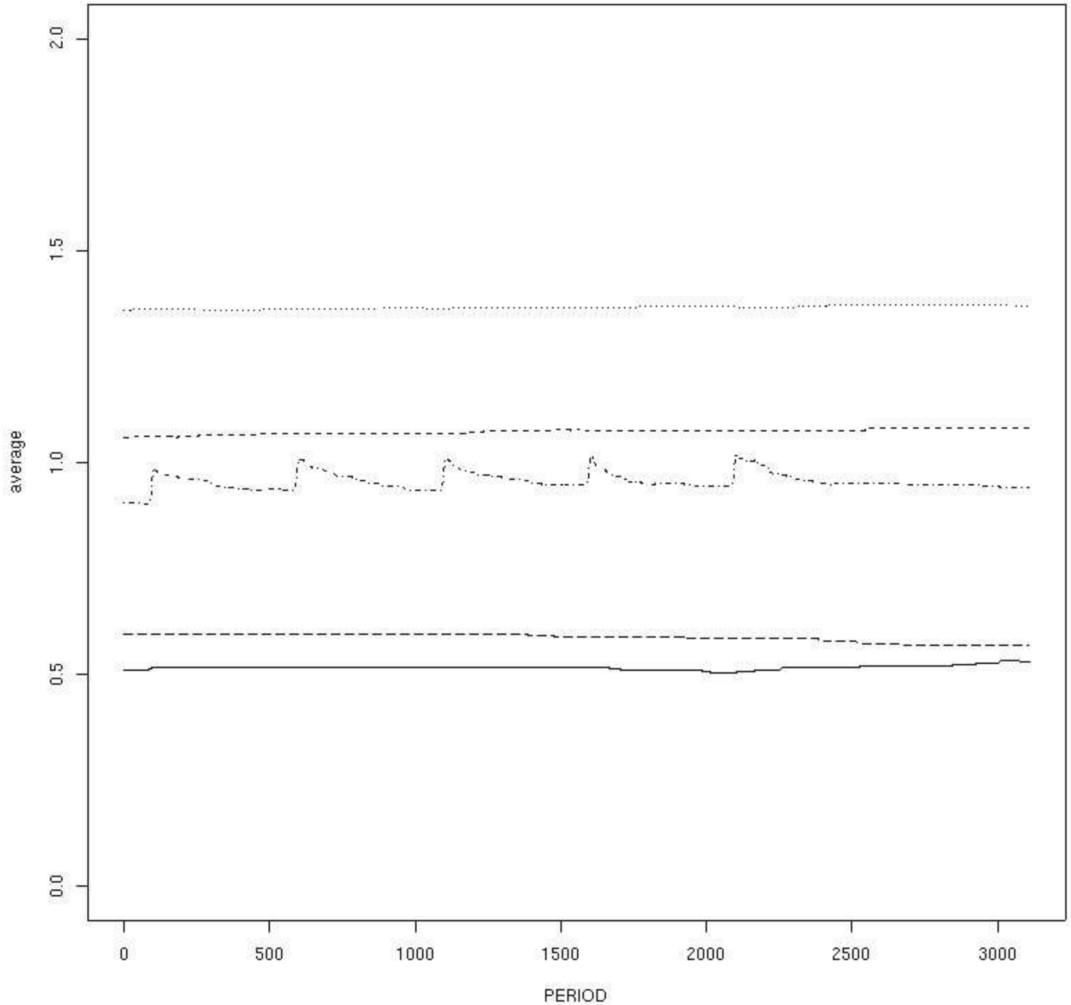
15



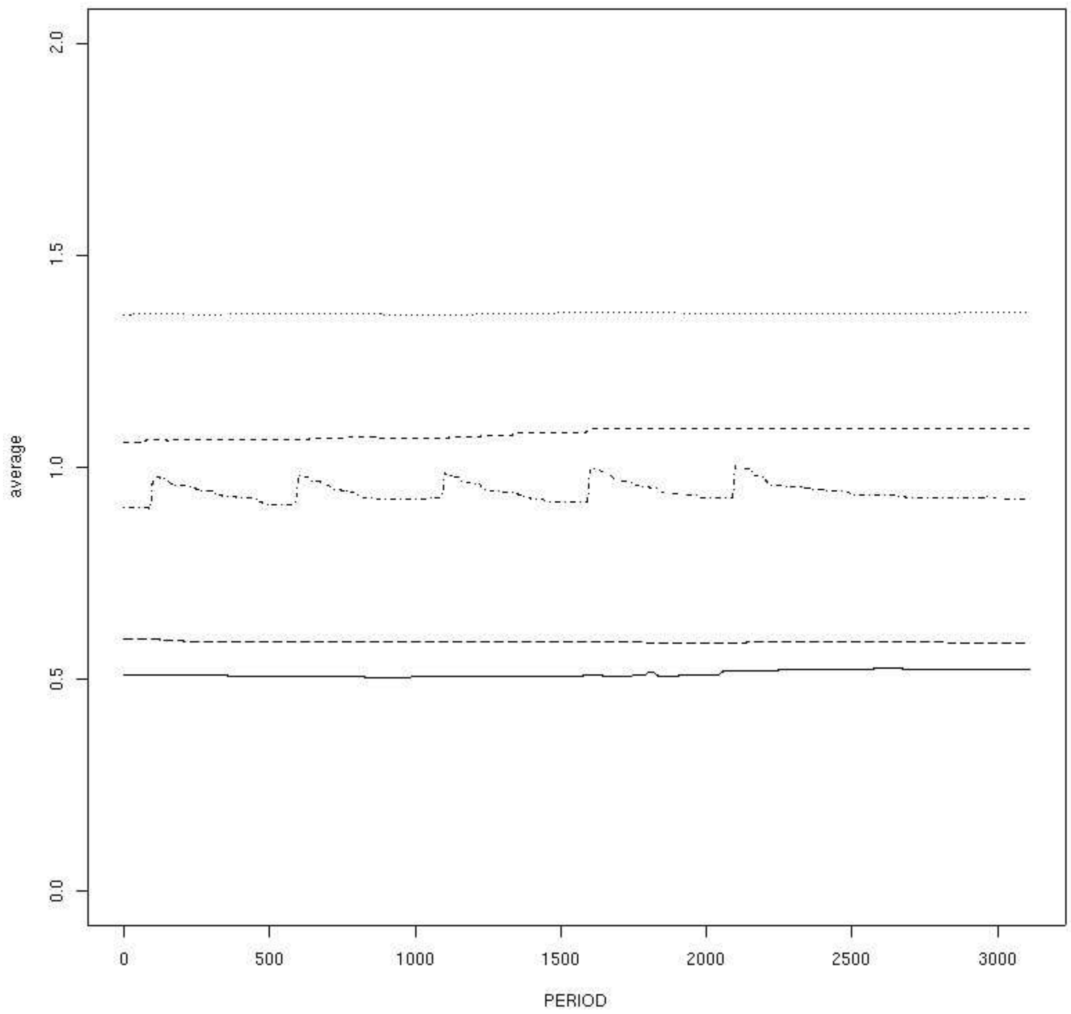
16



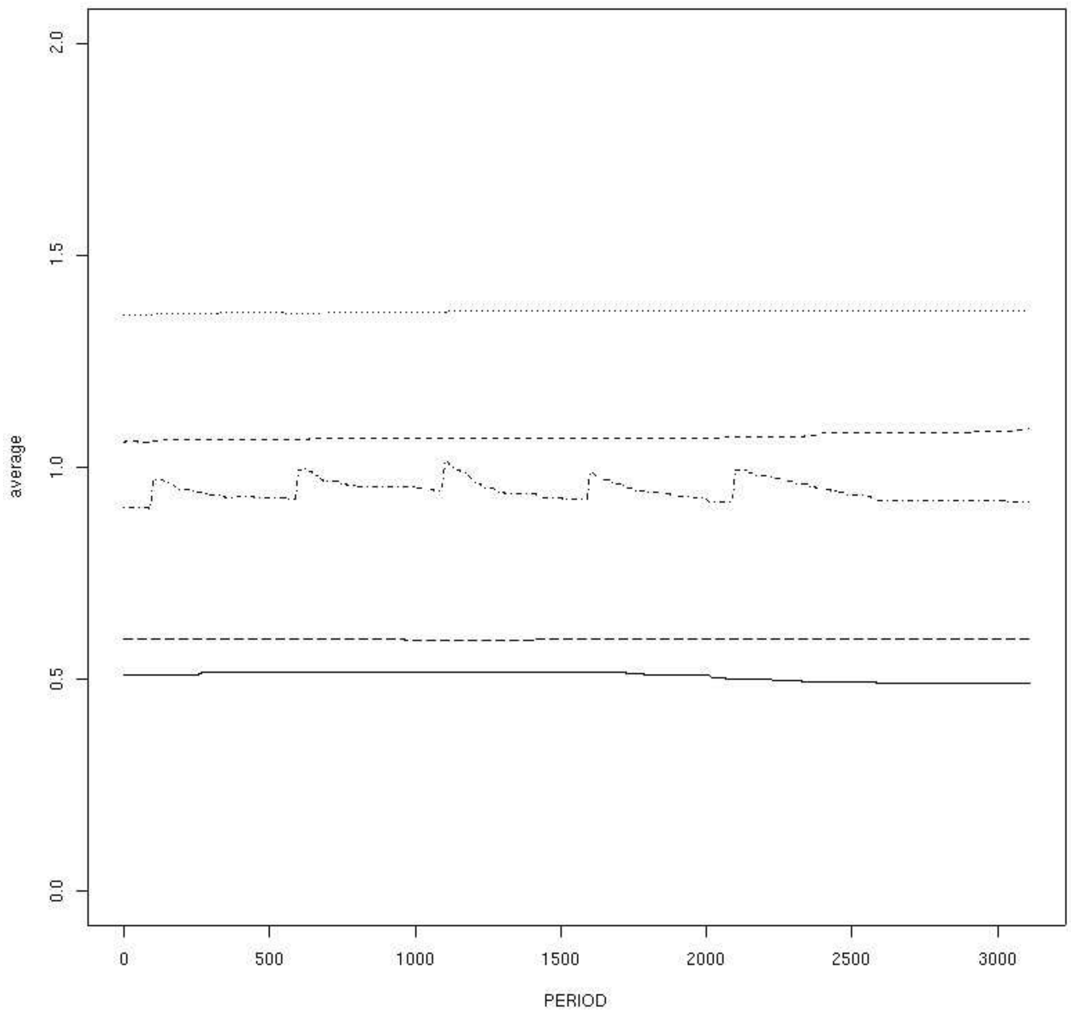
17



18



19

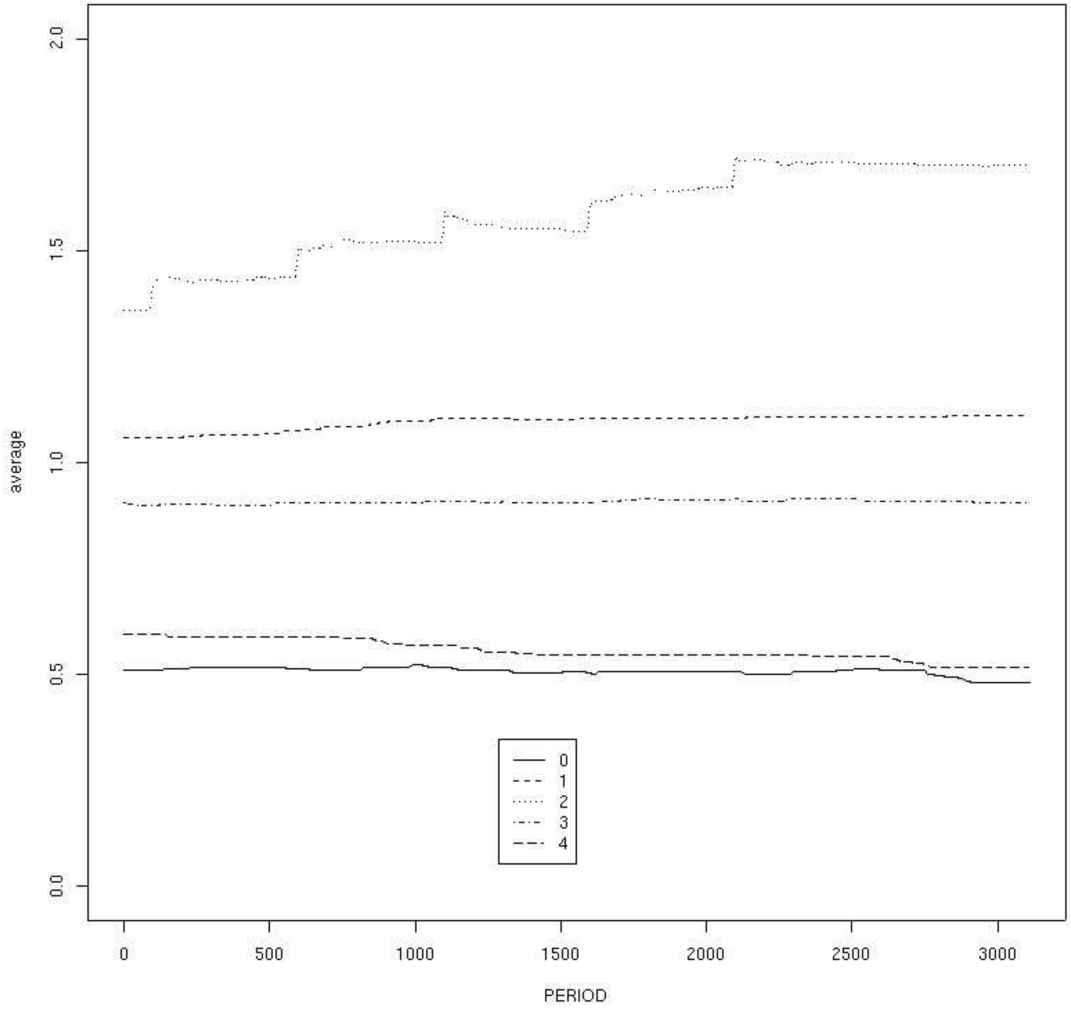


Compare!

Now we shock a different opinion in a different way.

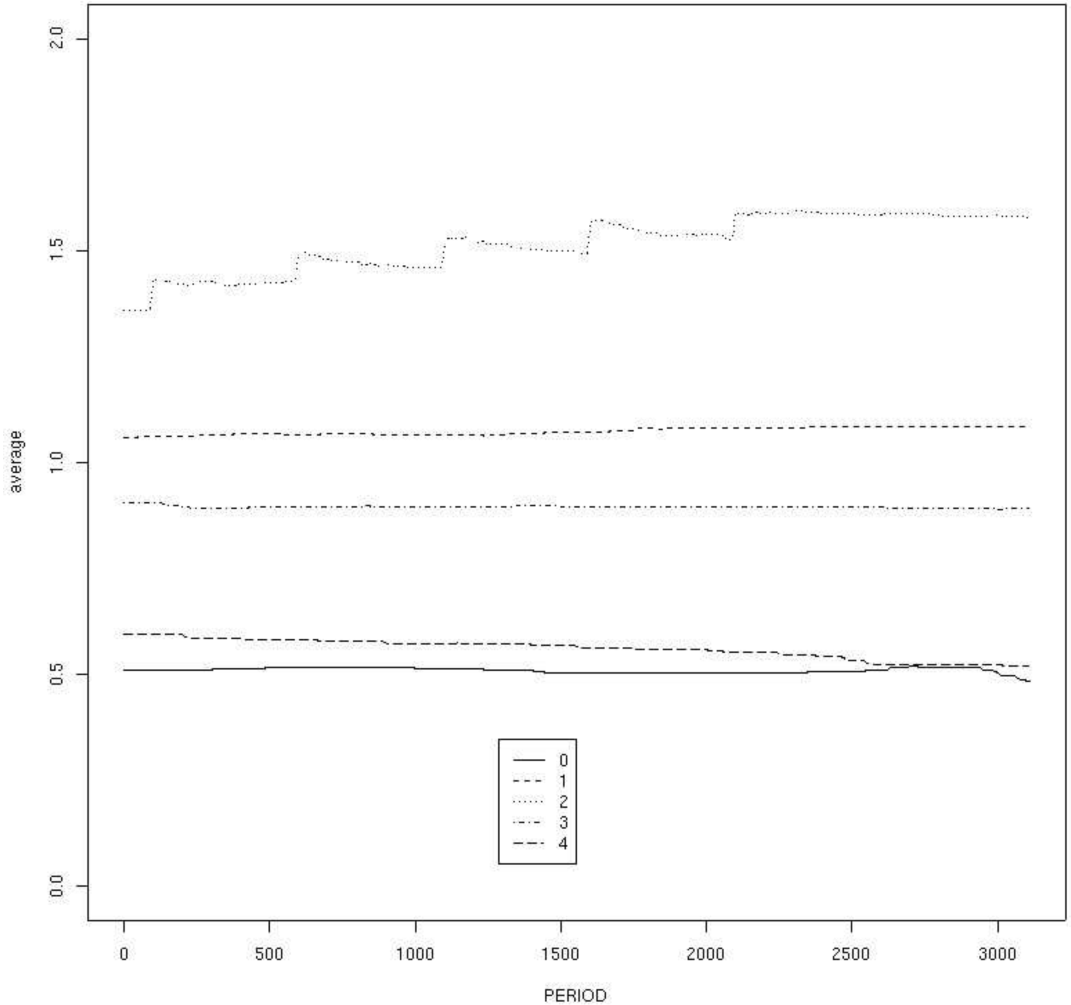
- we find 5% of the agents, and we change their opinion to 2 from either 0 or 1.
- observe that there is more variation across runs that begin with the same conditions and apply a stochastically equivalent shock.

1



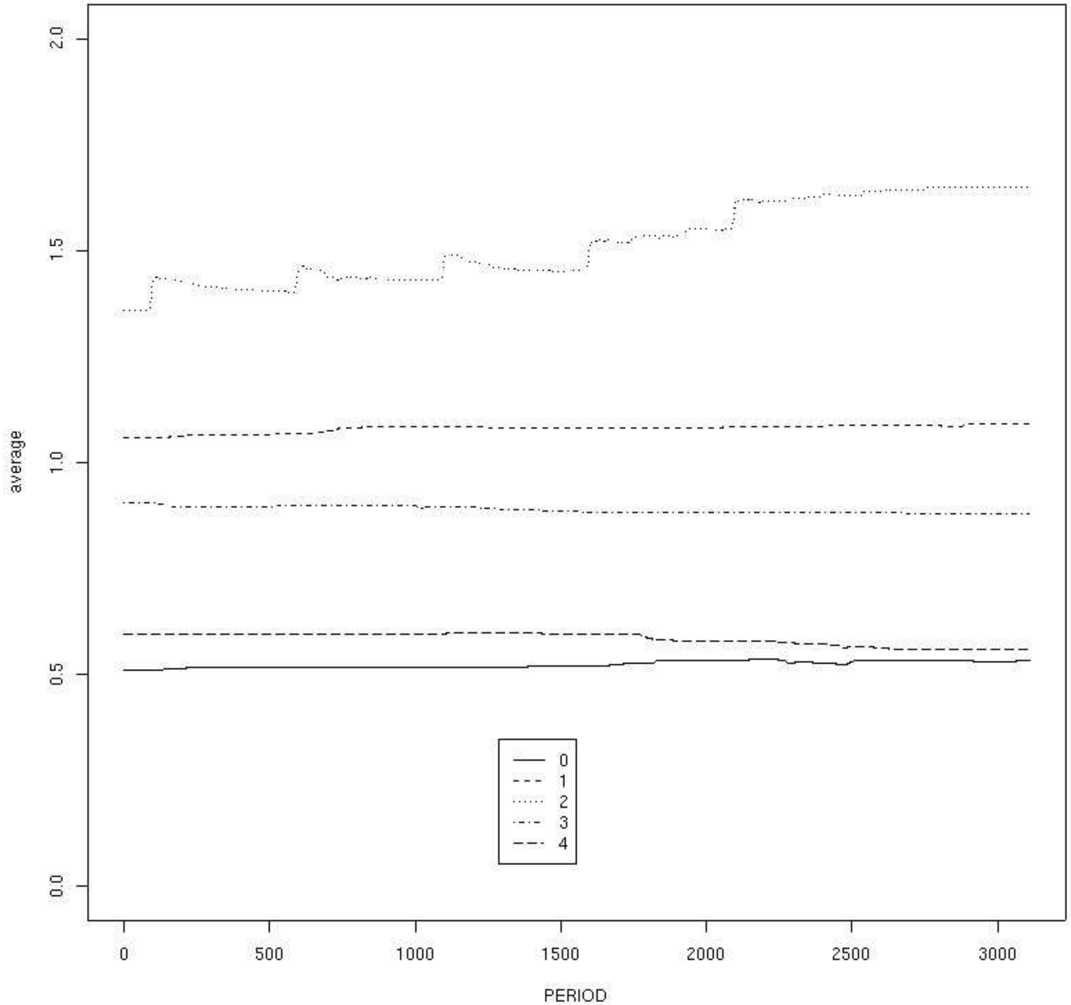
average
PERIOD
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
0
1
2
3
4

2



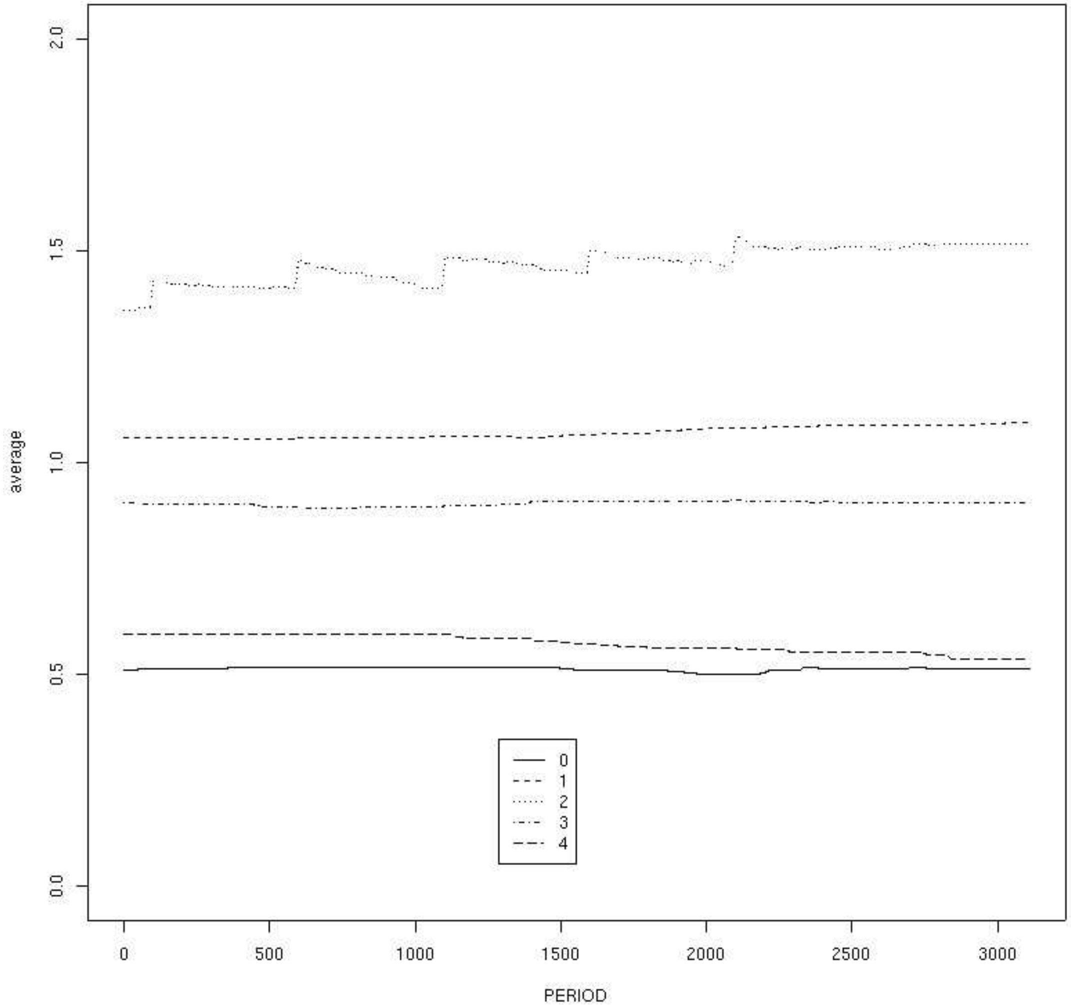
average
2.0
1.5
1.0
0.5
0.0
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

3



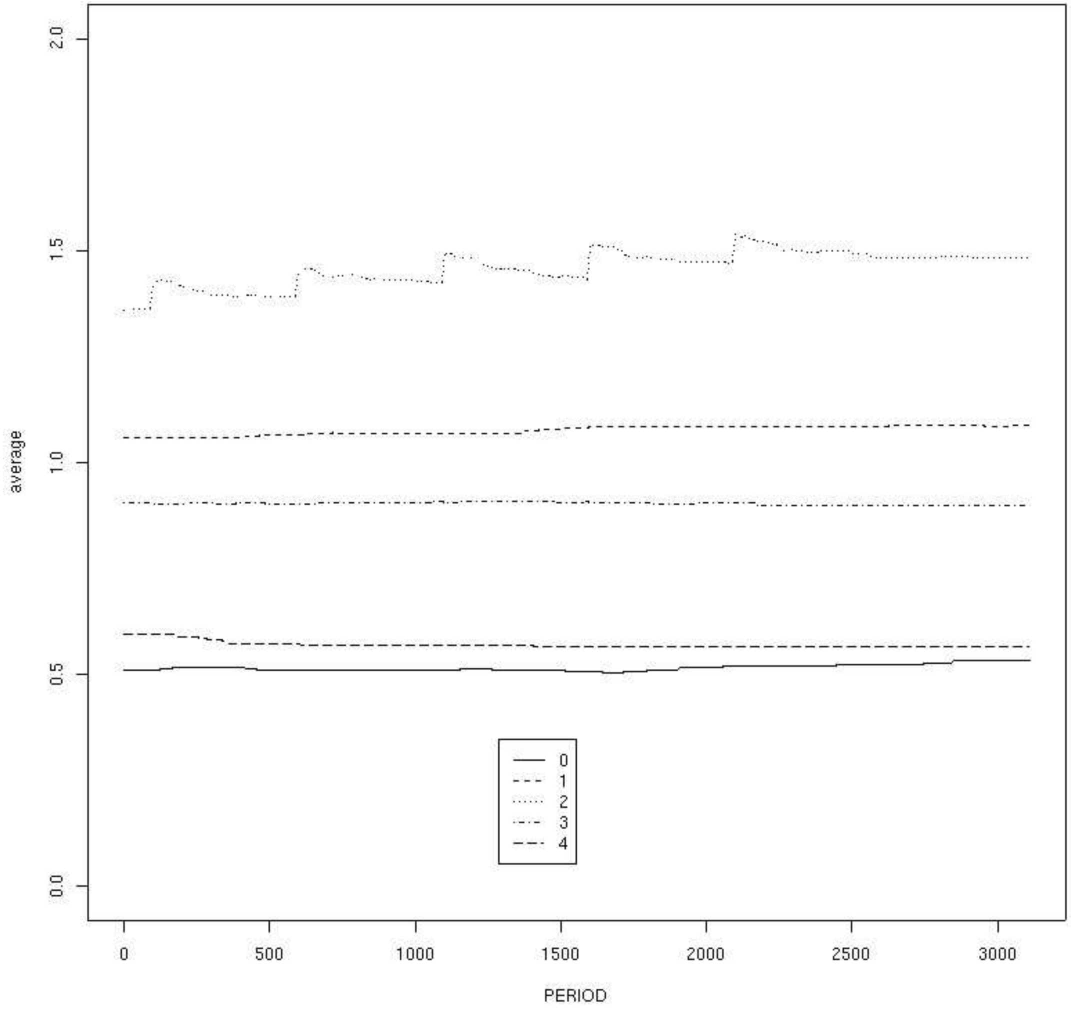
2.0
1.5
1.0
0.5
0.0
average
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

4



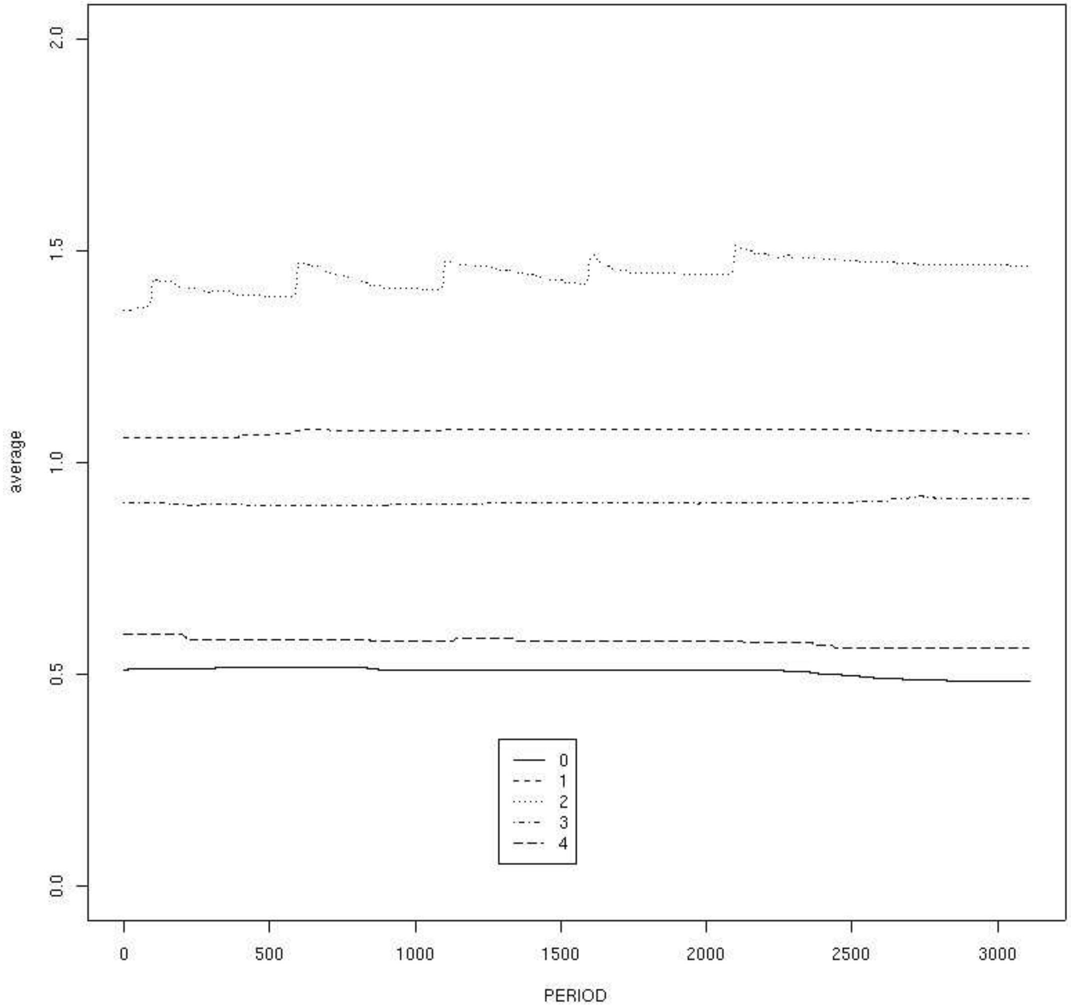
average
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

5



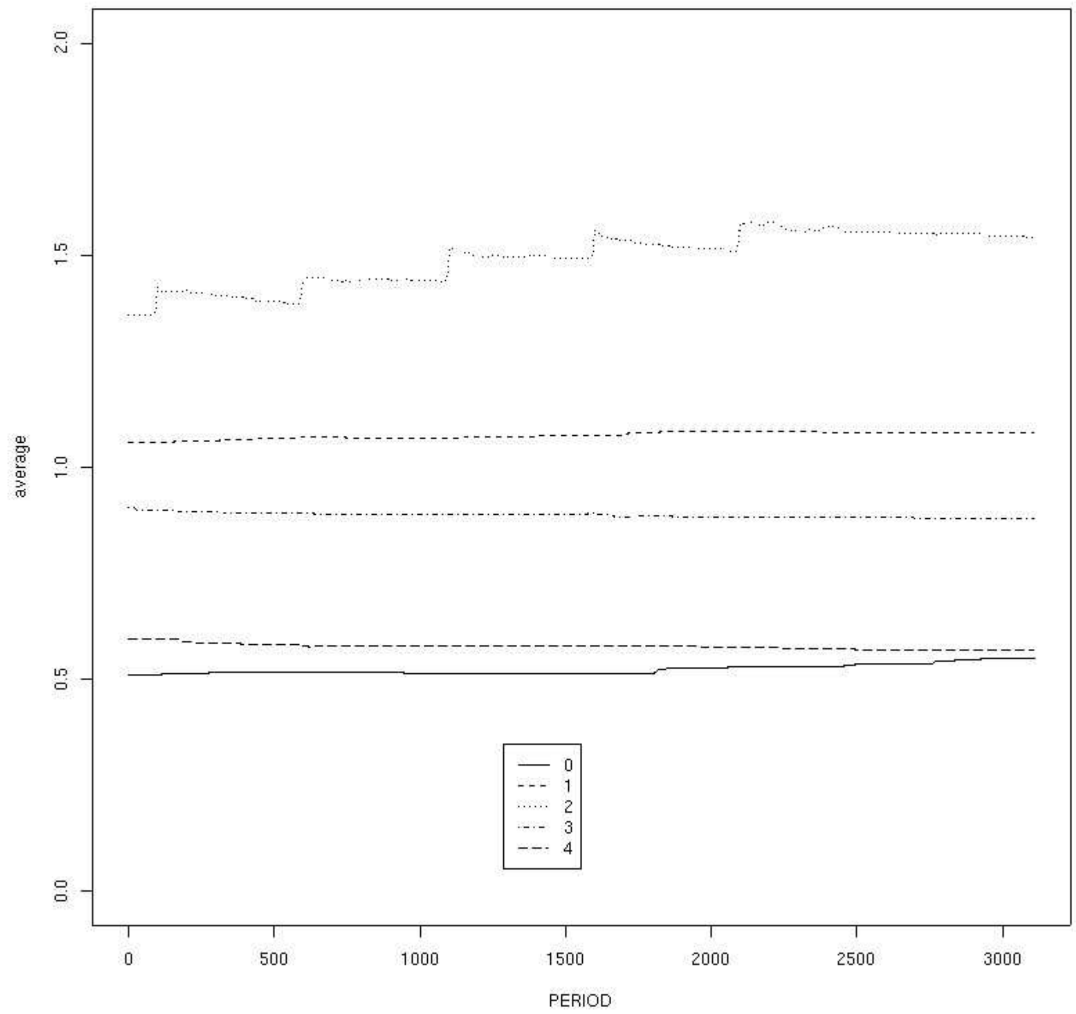
2.0
1.5
1.0
0.5
0.0
average
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

6



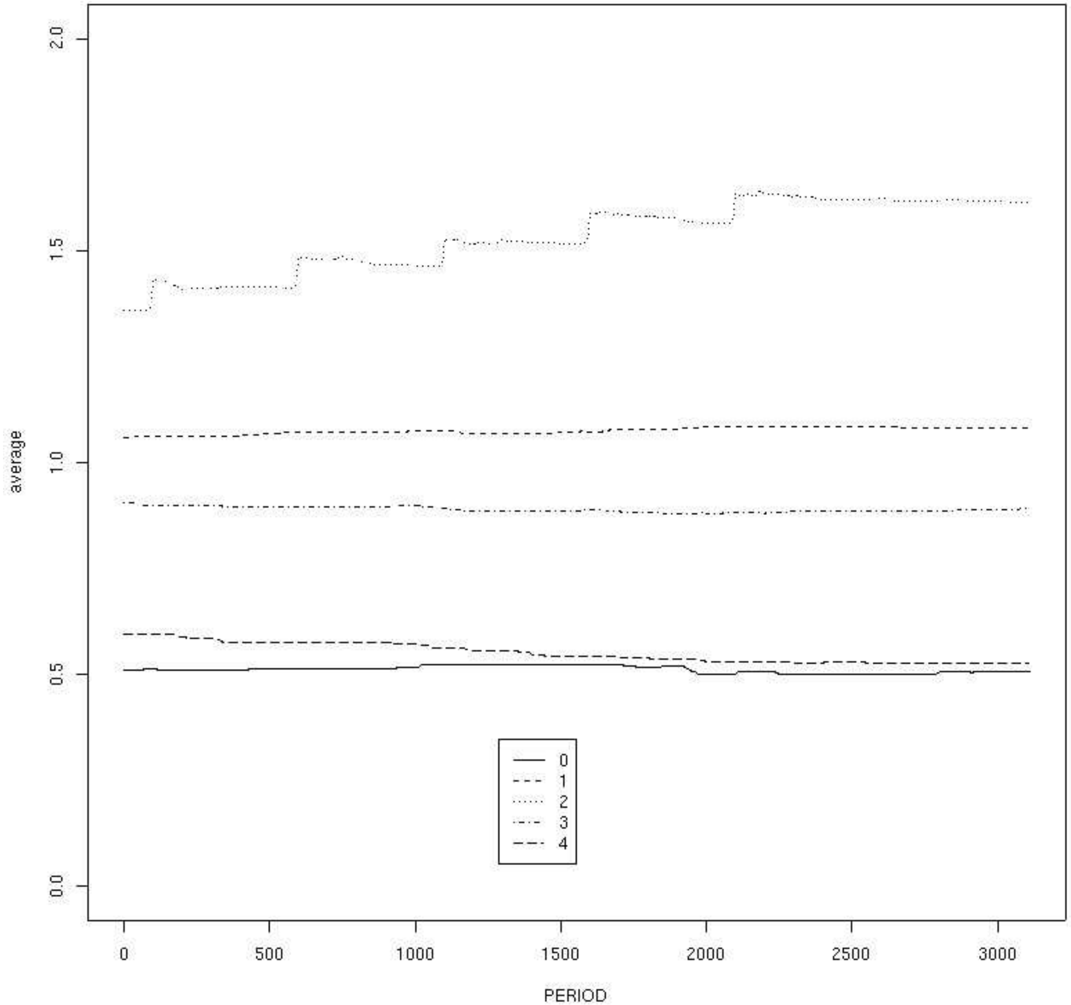
average
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

7

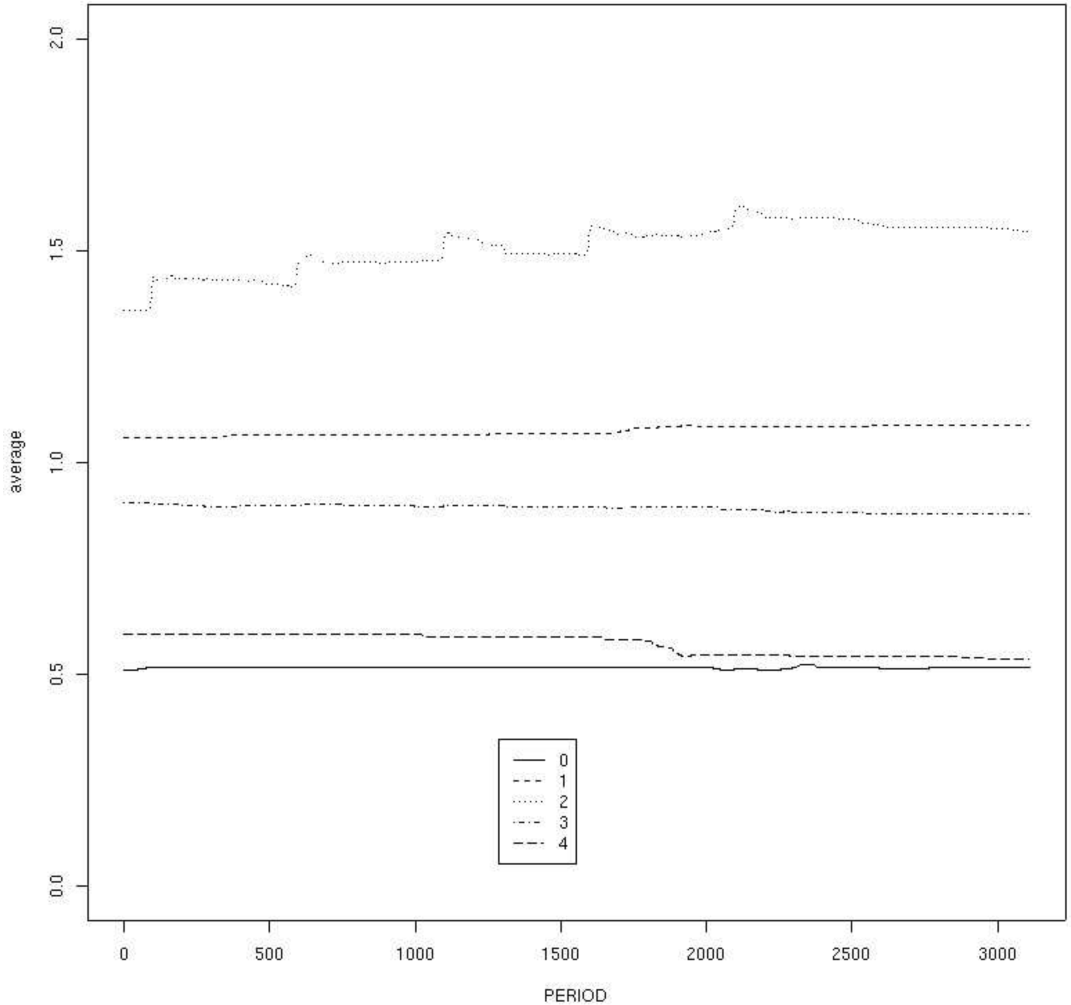


average
PERIOD
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
0
1
2
3
4

8

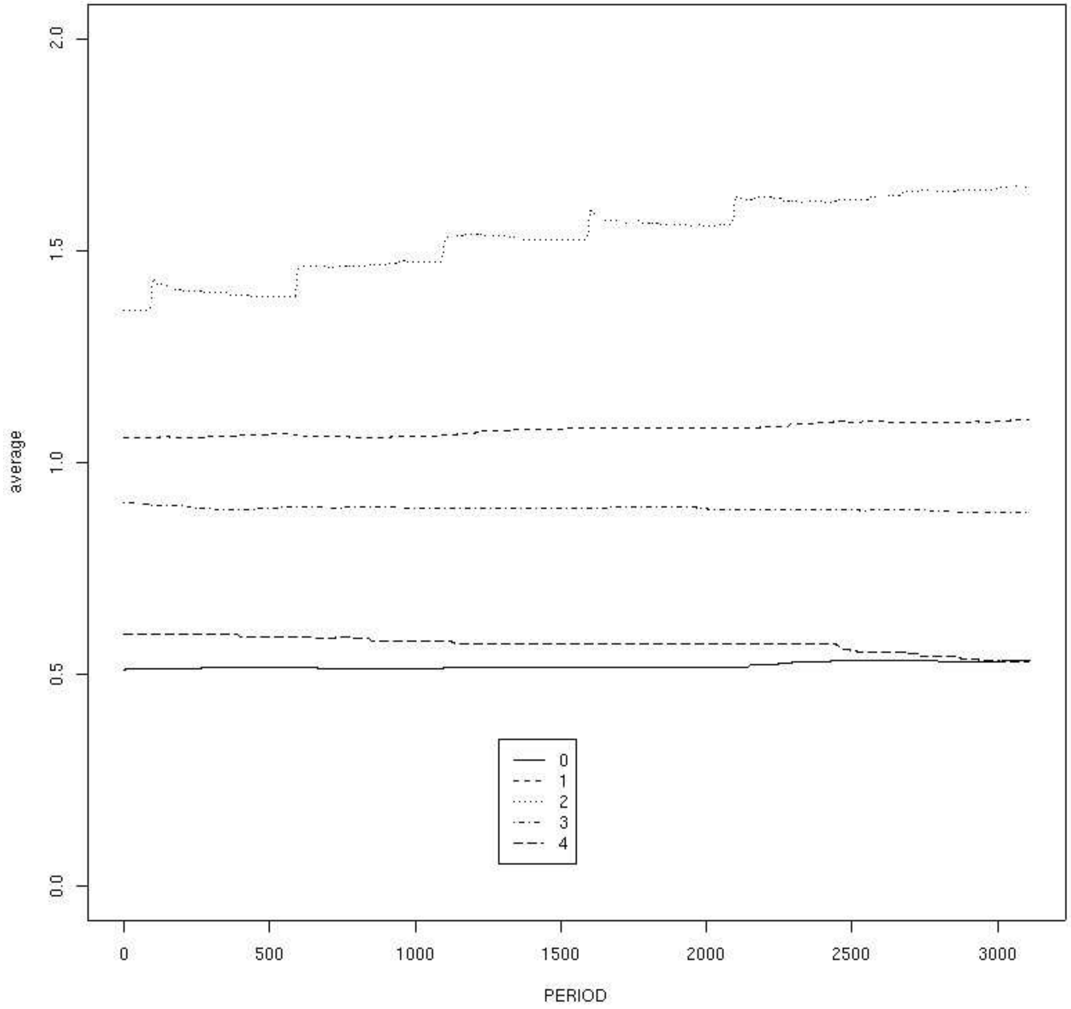


average
2.0
1.5
1.0
0.5
0.0
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

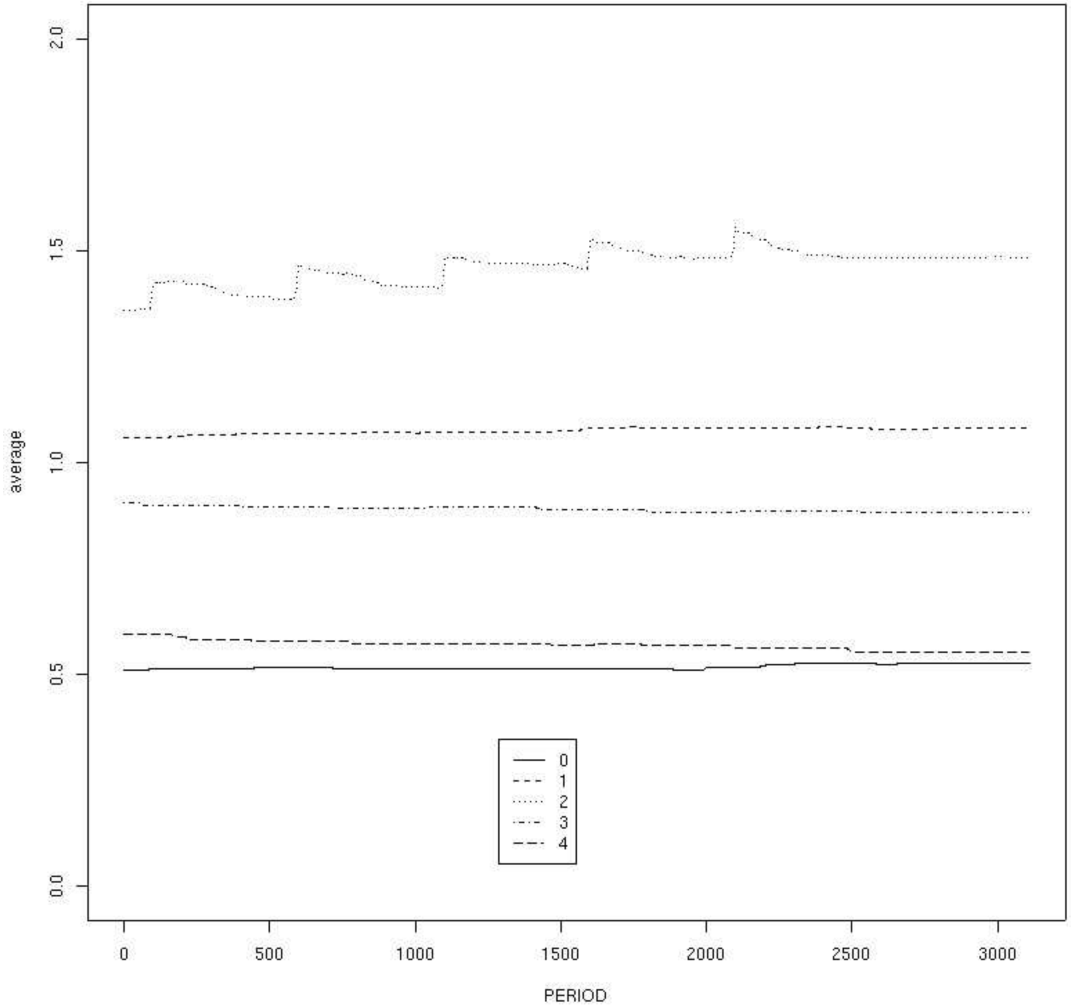


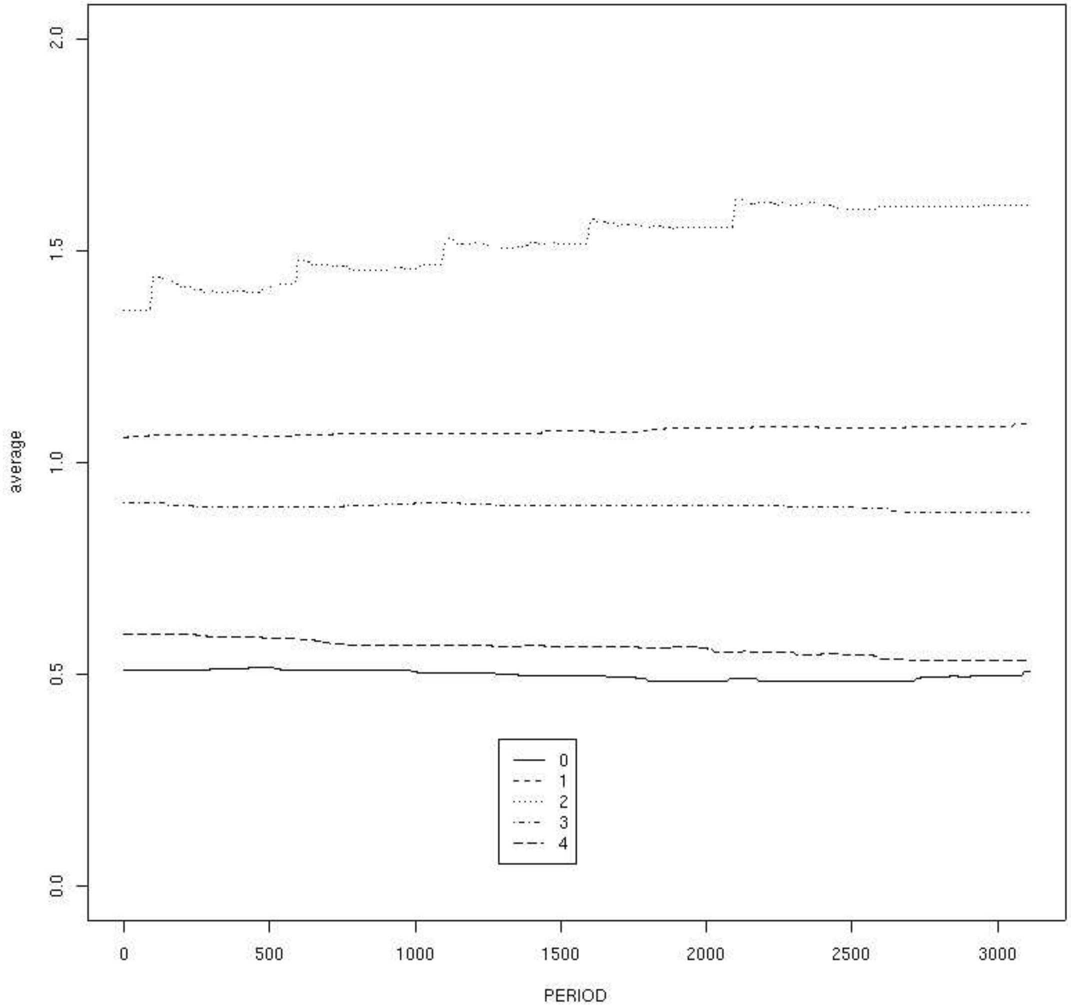
average
PERIOD
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
0
1
2
3
4

10

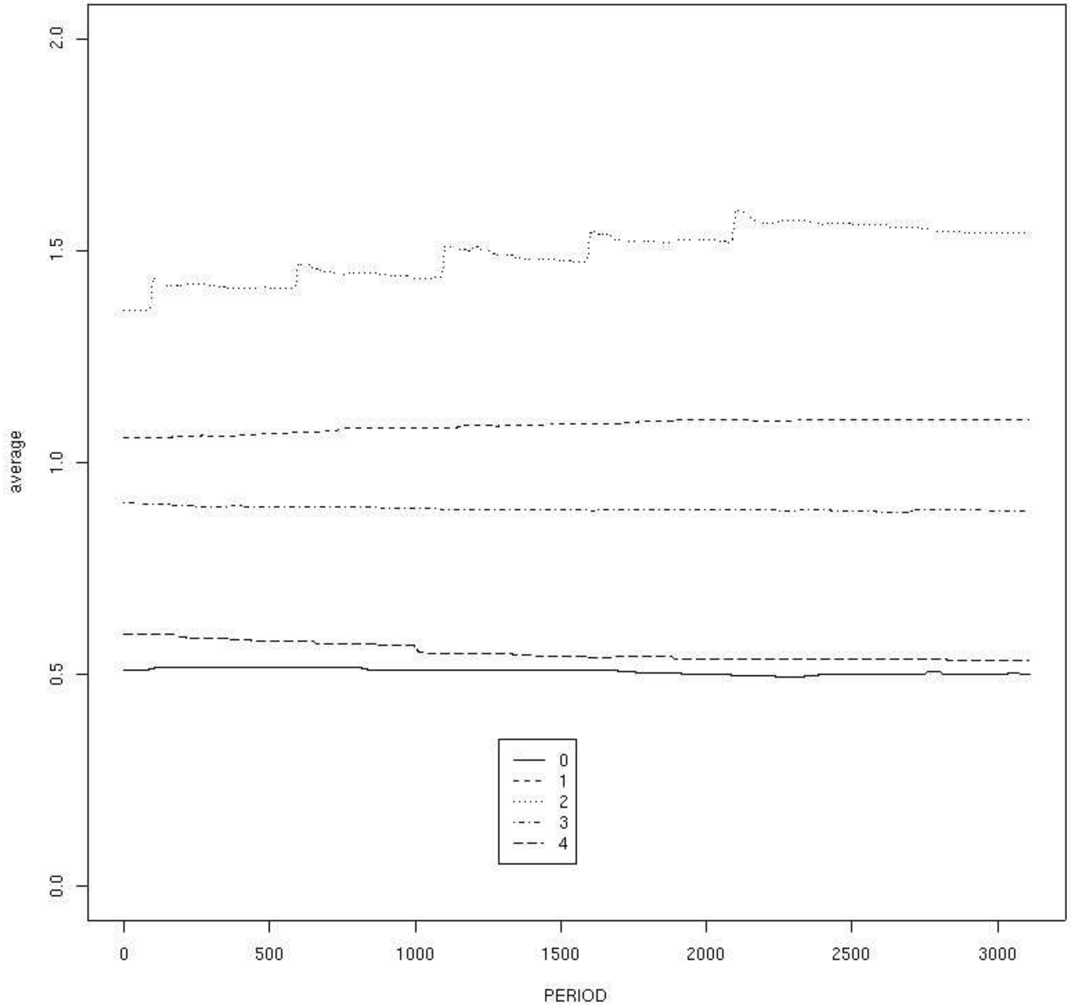


2.0
1.5
1.0
0.5
0.0
average
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4


average
2.0
1.5
1.0
0.5
0.0
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

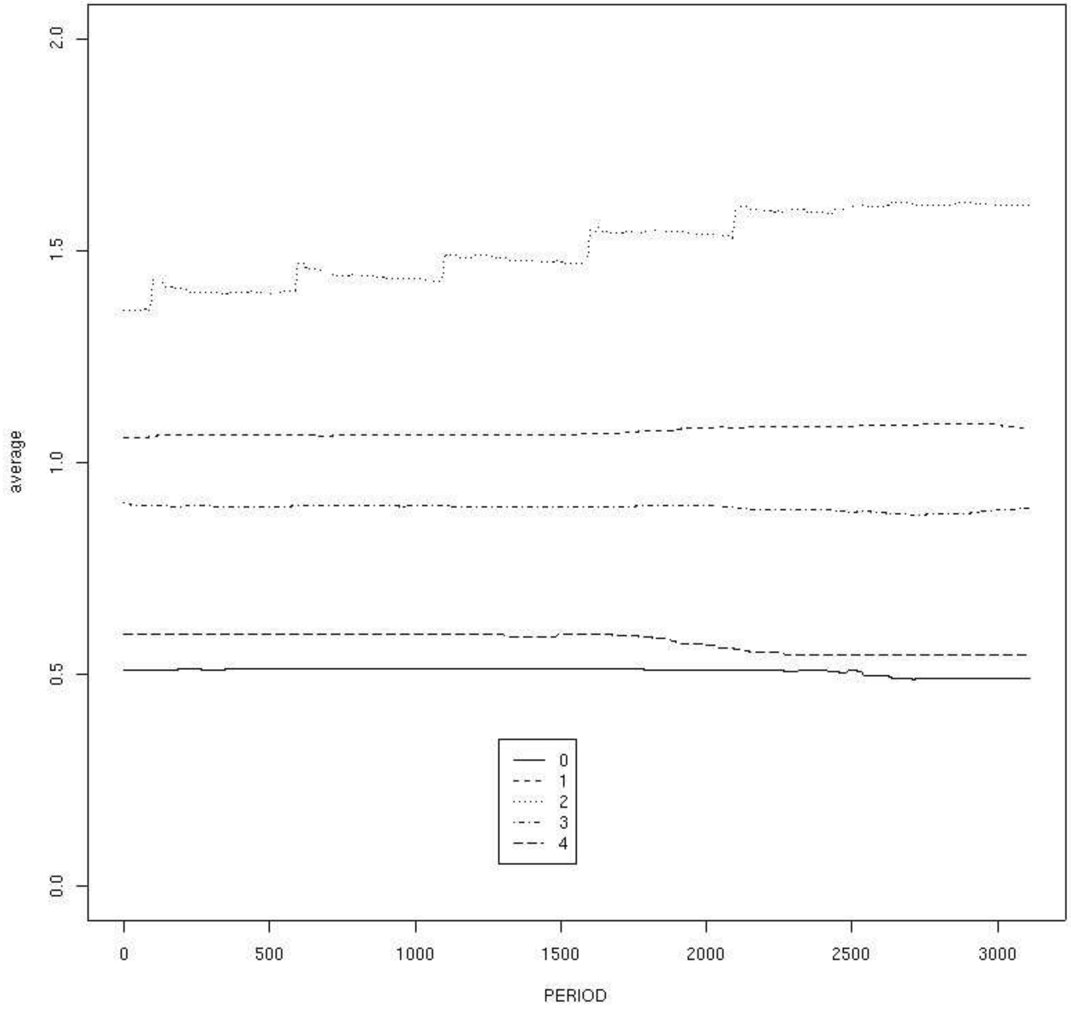

average
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

13



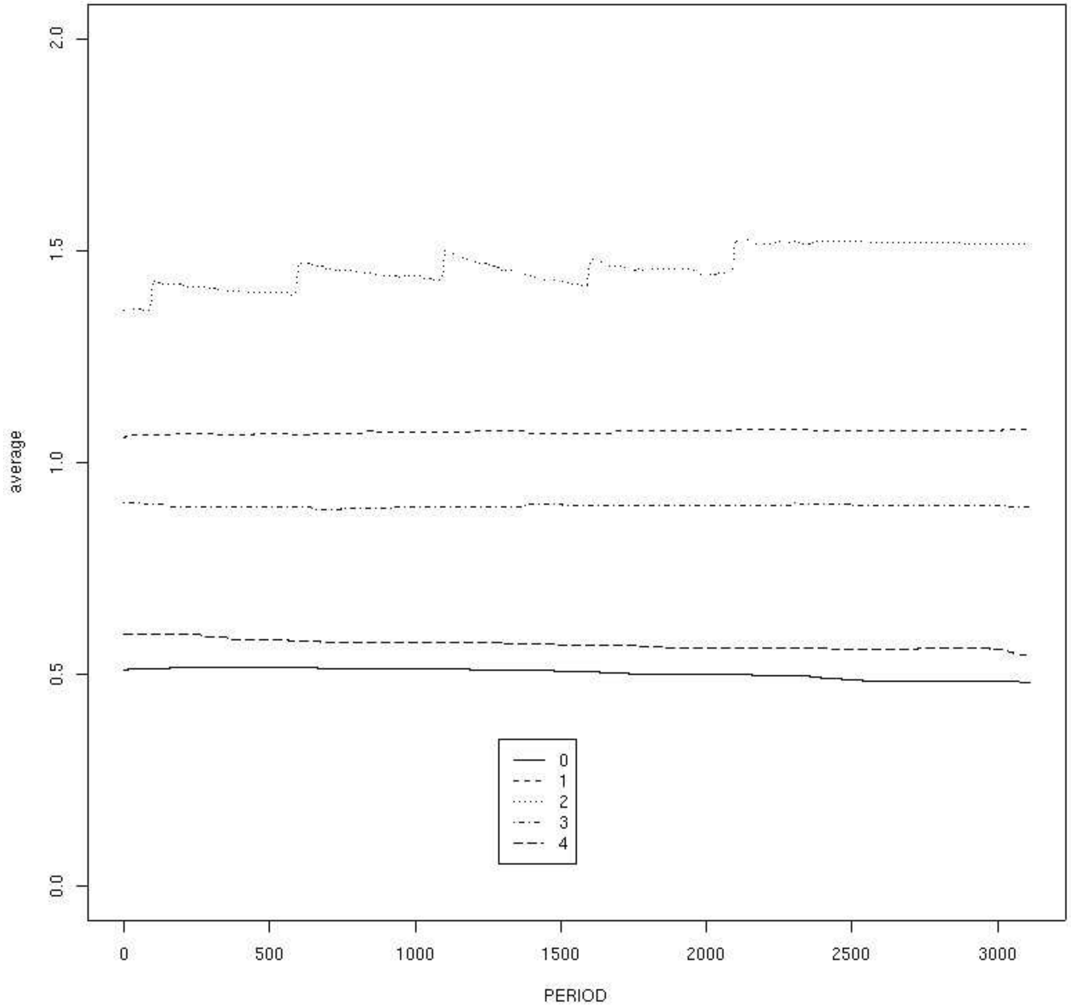
average
PERIOD
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
0
1
2
3
4

14



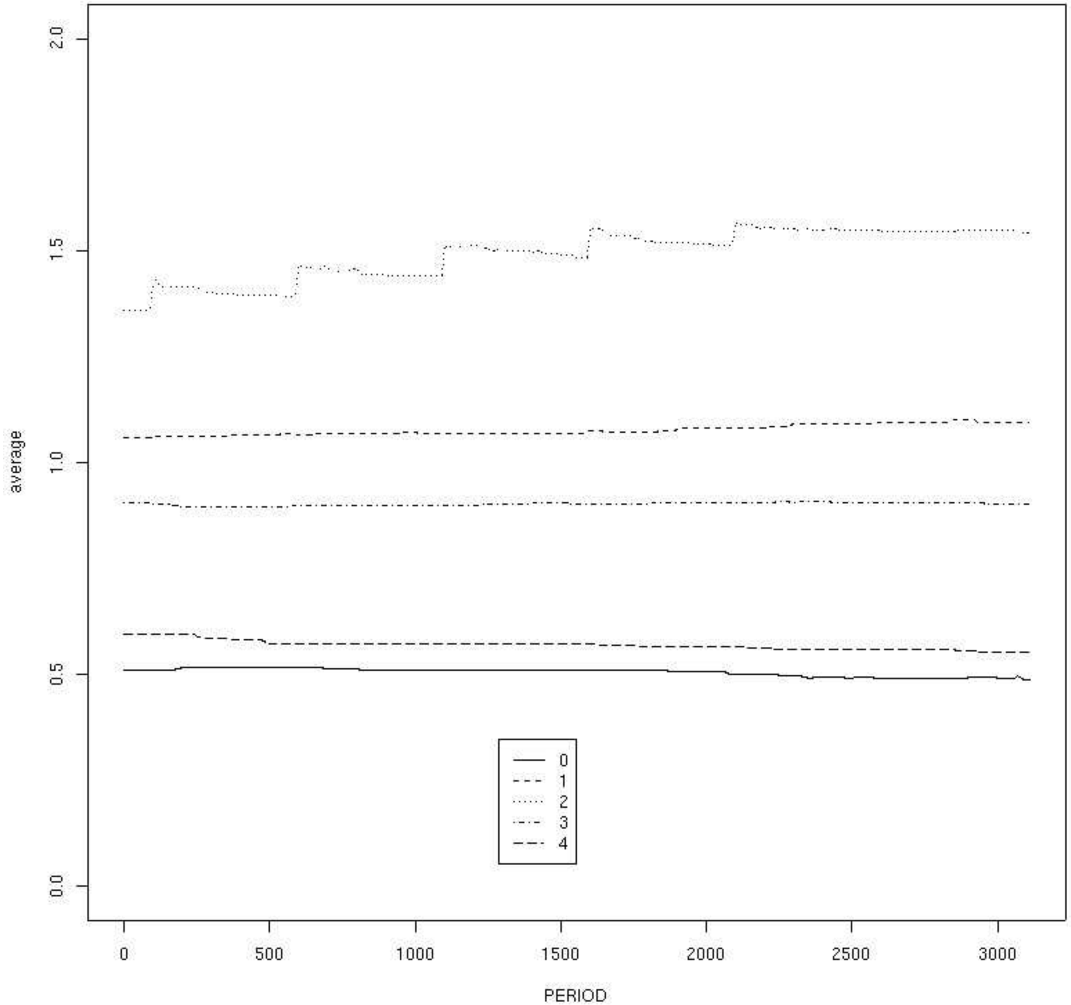
average
2.0
1.5
1.0
0.5
0.0
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

15



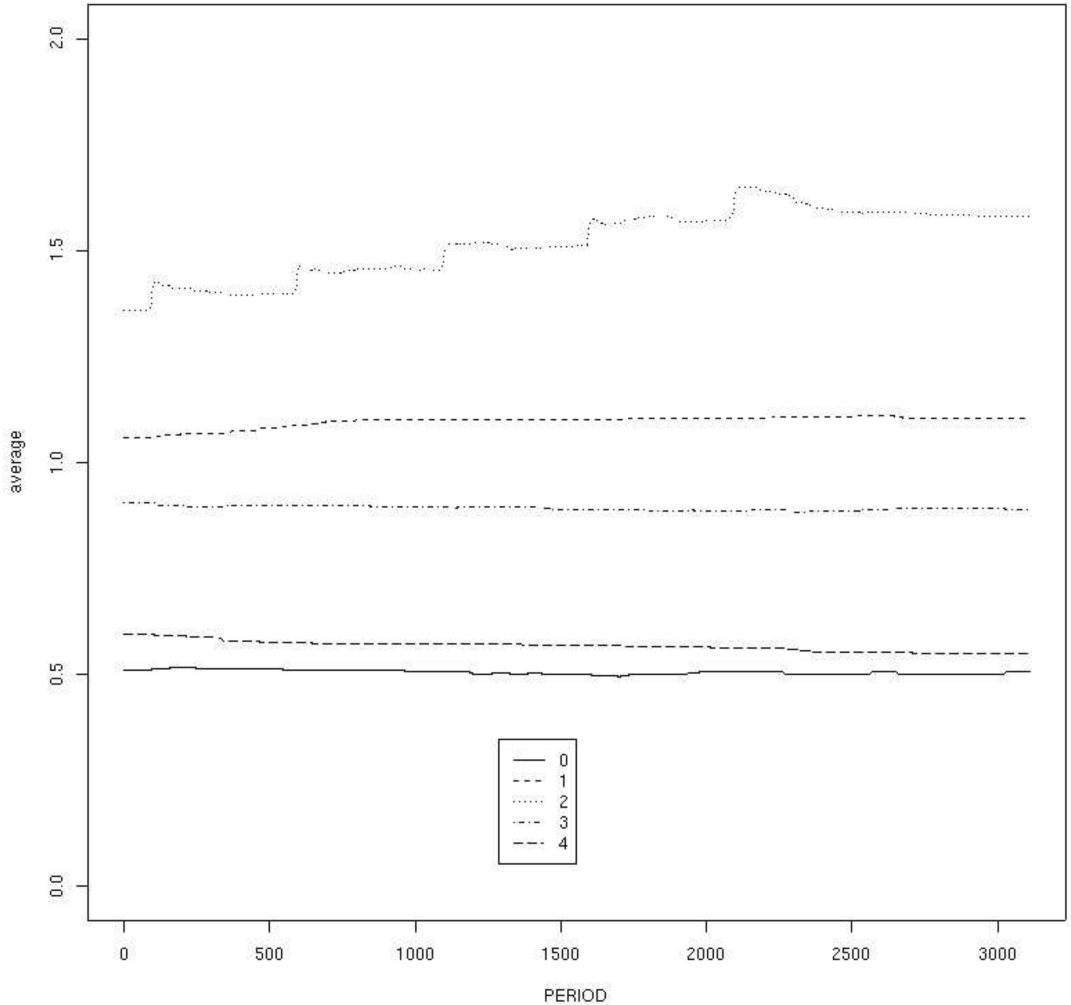
average
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

16

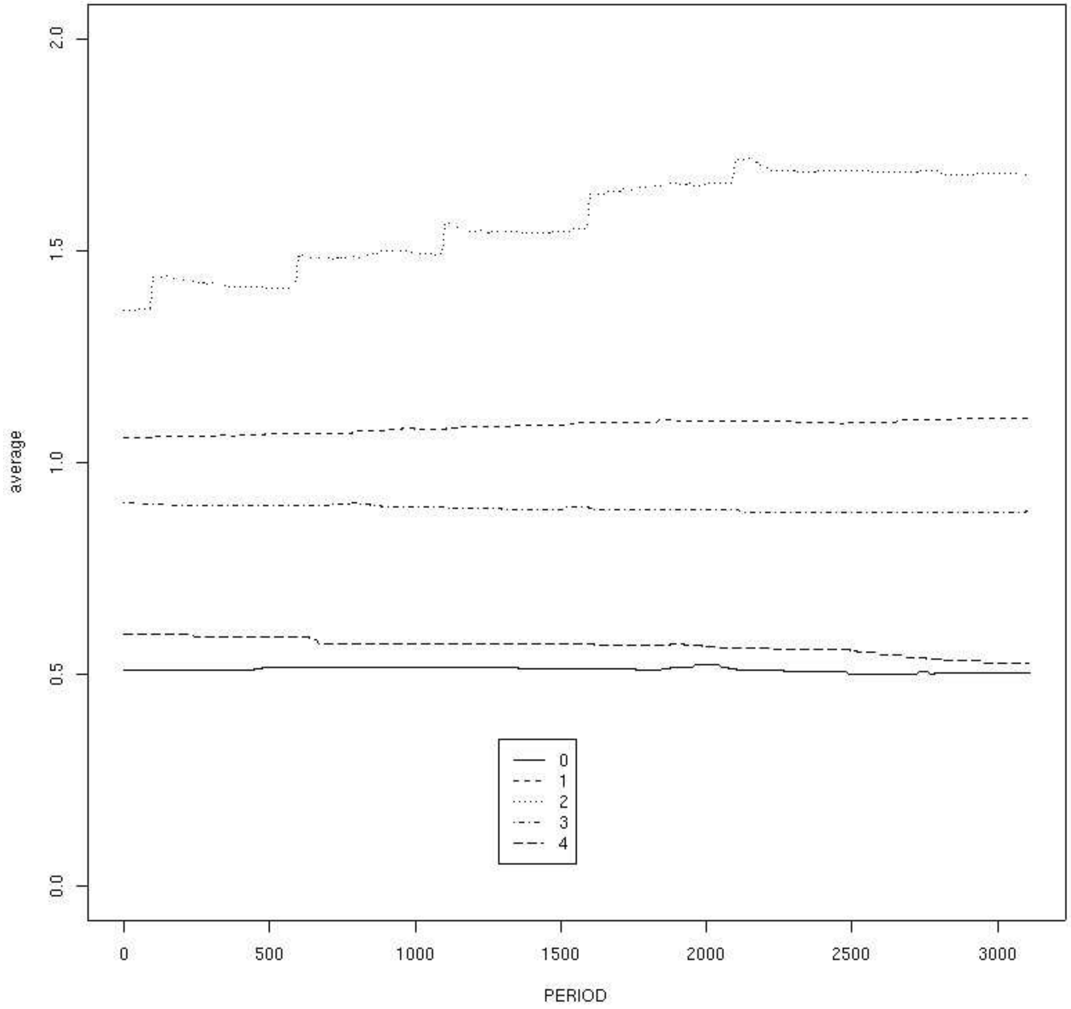


2.0
1.5
1.0
0.5
0.0
average
0
1
2
3
4
0
500
1000
1500
2000
2500
3000
PERIOD

17

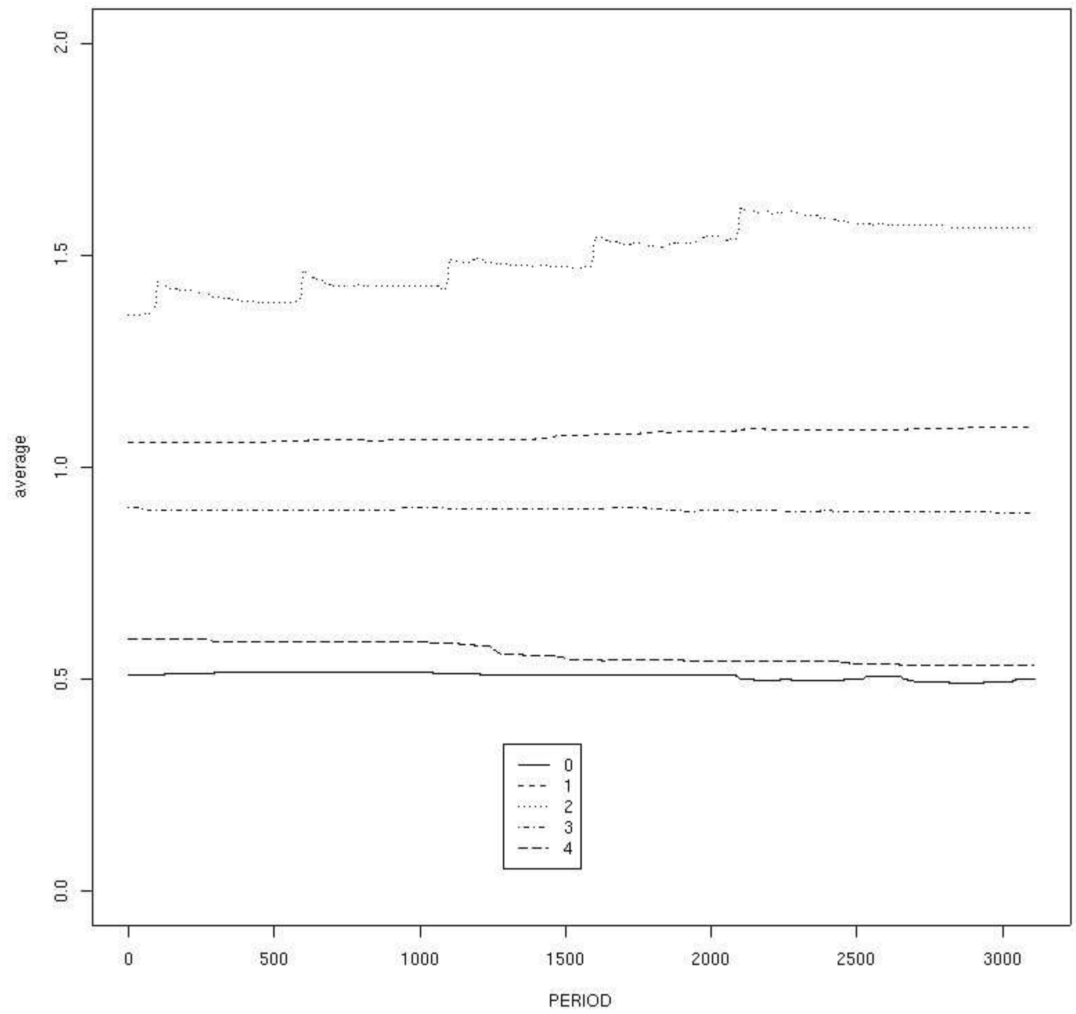

average
PERIOD
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
0
1
2
3
4

18



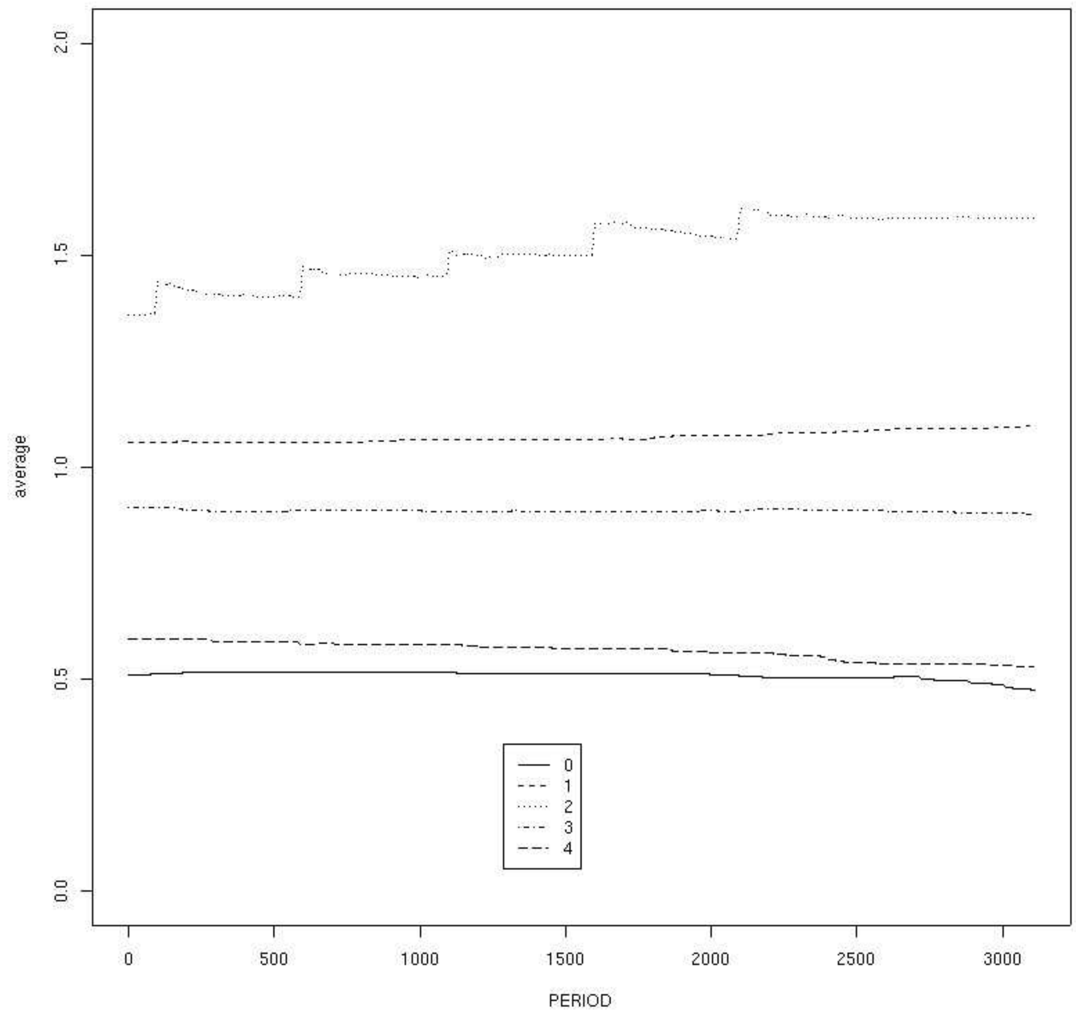
average
PERIOD
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
0
1
2
3
4

19



average
PERIOD
0.0
0.5
1.0
1.5
2.0
0
500
1000
1500
2000
2500
3000
0
1
2
3
4

20



average
2.0
1.5
1.0
0.5
0.0
0
500
1000
1500
2000
2500
3000
PERIOD
0
1
2
3
4

Conclusion

- Serialization is Workable
- Serialization is potentially useful