A

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE HONORS DEGREE OF

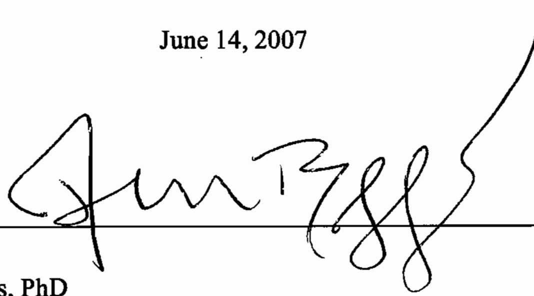
BACHELOR OF SCIENCE IN PHYSICS

# Dynamics of Complex Neural Networks: An Experimental Approach

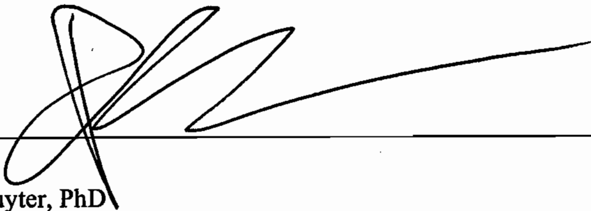
**Robert M. Koffie**

DEPARTMENT OF PHYSICS
INDIANA UNIVERSITY

June 14, 2007



Supervisor: ______________________________
*Signature*
John M. Beggs, PhD



Co-signer: ______________________________
*Signature*
Robert De Ruyter, PhD

**Abstract:** The interactions and dynamics of complex neural networks are believed to be essential in enabling the mammalian brain to undertake several key functions that underlie cognition, such as information processing, computation, and memory storage. While there have been several efforts to understand the dynamics of complex neural networks, experimental methods to explore the governing principles of neural network dynamics have been scarce. Here we report an experimental approach to studying the dynamics of complex neural networks using multi-channel microelectrode arrays. We find that neural networks operate near the critical point, with dynamical trajectories that are mostly neutral. These neutral dynamical trajectories in neocortical networks are very stable, resisting the expected effects of both inhibitory and excitatory drugs. Our findings on the dynamical trajectories of complex neural networks offer insight into the biophysical principles governing neuronal network dynamics, and provide leads to better understanding neurological diseases such as epilepsy.

## 1. Introduction

The mammalian cortex is made up of billions of neurons arranged in local circuits that form complex neural networks. The interactions and dynamics of these neural networks are believed to enable the brain to undertake such vital functions as information processing, computations, and memory storage. Understanding the dynamics of neocortical networks could offer great insight into the mechanisms of different cognitive processes, and how neurological diseases such as epilepsy and dementia impair these processes.

Towards this end, a number of research efforts have aimed at understanding the dynamical trajectories of neocortical networks. Three main hypotheses for exploring the dynamics of neural networks have emerged over the years: the first proposes that neural networks exhibit attractive dynamics, with different inputs at one point of the network resulting in similar outputs[1-3]. This hypothesis is based on the observation that computations that favor categorization cause different stimuli to be grouped into the same response, a phenomenon unique to networks with attractive dynamics. The second hypothesis propounds that neural networks display chaotic dynamics, with minute differences in input resulting in very pronounced differences in output[4-6]. Chaotic dynamics of neural networks support computations that favor discrimination insofar as subtle differences in stimuli produce different responses. The third hypothesis, proposes that neural networks exhibit neutral dynamics, with differences in inputs producing commensurate differences in responses[7-9]. Neutral dynamics of neuronal networks support computations and efficient information transmission inasmuch as one-to-one mapping between stimuli and responses is maintained. While each of these hypotheses has been significantly explored theoretically, there remains no clear experimental means of studying the dynamics of complex neural networks.

In an effort to understand the fundamental principles governing neural network dynamics, we used experimental means to test the three main hypotheses listed above. To determine dynamics in local neocortical networks, several requirements must be simultaneously met. First, the network should have identified stable states or places in state space that are frequently revisited. Second, the rate of convergence or divergence of nearby trajectories must be quantified. Third, the dynamics should be examined on the millisecond

time scale to be relevant to neural computations. Fourth, insofar as neural processing is thought to be parallel and distributed, the dynamics of widely distributed neural representations should be examined and quantified. Our studies are aimed at fulfilling all these requirements and identifying the principles governing neural network dynamics.

Using 60-channel microelectrode arrays to record spontaneous neural activity of neocortical slices from rat, we collected activity data and extracted Lyapunov exponents, standard statistical parameters for quantifying the trajectories of dynamical systems. Our experimental results allowed us to explore sequences of distributed neural activity and quantify trajectories in a way that permits thorough analysis of neural network dynamics[10-11]. We find that while neural networks exhibit attractive, chaotic and neutral dynamics at different instances, the exhibition of neutral dynamic was significantly more persistent in neuronal networks of sliced rat cortex. We also explored the effect of different network-modulating drugs, and found that neuronal networks that exhibited neutral dynamics are not significantly influenced by modulation. These findings afford new knowledge about the governing principles of neural network dynamics, which could provide insights into the biophysical nature of neurological diseases such as schizophrenia and epilepsy, where network stability is thought to be compromised[12, 13].

## 2. Materials and Methods

### 2.1 *Preparation of slices on multi-electrode arrays*

For the preparation of acute slices, coronal sections from rat brains (rats were purchased from Harlan Sprangue Dawley Inc., Indianapolis, IN) at ages 13-27 days were cut on a vibroslicer (Vibratome 3000 Sectioning system; Ted Pella Inc., Redding, CA). Cortical slices were cut to 250$\mu$m thin from the dorsolateral somatosensory cortex of the rat brain. After cutting, all tissues were immersed for 1-2 hrs in chilled artificial cerebrospinal fluid (ACSF) at pH 7.4, and then allowed to warm up to room temperature. ACSF was prepared to contain the following chemicals in aqueous solution: 124 *m*M of NaCl, 2 *m*M of KCl, 2 *m*M of $CaCl_2$, 2 *m*M of $MgSO_4$, 1.25 *m*M of $NaH_2PO_4$, 26 *m*M of $NaHCO_3$, and 10 *m*M of D-glucose. Cortical slices were then gently transferred onto a 60-channel

microelectrode array (MEA) covered with excitable aqueous solution containing 24 *m*M of $NaCl$, 5 *m*M of $KCl$, 1.25 *m*M of $NaH_2PO_4$, 26 *m*M of $NaHCO_3$, 10 mM D-glucose, and 2 mM of $CaCl_2$ (note that this excitable solution contains high $K^+$ and low $Mg^{2+}$ concentrations). Solutions with elevated levels of potassium ions have been shown to promote spontaneous activity in neocortical circuits[16].

2.2 *Preparation of MEA*

All experimental results were generated from local field potentials recordings from sliced rat cortex using 60-channel MEAs (Multichannel systems, Reutlingen, Germany). Each MEA was made up of a square (5 cm × 5 cm) plate of glass with a circular glass well attached to the center. The center of the well contained a square array of 60 microelectrodes (8 × 8 grids, corners position had no microelectrodes) made of titanium nitride. Electrodes were flat and disk-shaped with a diameter of 20 $\mu$m, and had inter-electrode spacing of 200 $\mu$m. The electrodes were attached to gold leads that ended at the edge of the glass square. These leads served as contacts for spring-loaded probes of the amplifier head-stage (Multichannel systems). A silver chloride electrode was used for ground reference. This MEA preparation approach was similar to that reported elsewhere[10]. Figure 1 shows a picture of the experimental apparatus.

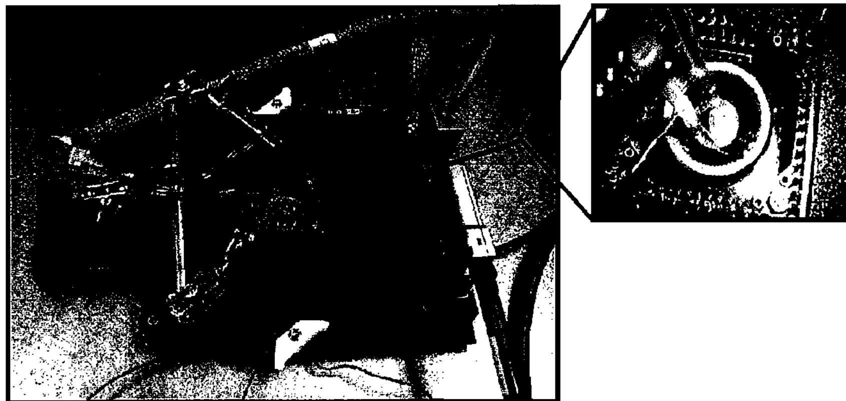


**Fig 1.** Photograph of MEA and the signal amplifier used in the experiment

2.3 *Data acquisition, filtering, and down-sampling of field potentials*

The data acquisition methods used were similar to those reported by Beggs and Plenz (2003, 2004)[10, 11]. Activities of neural networks were recorded as local field potentials (LFP) by the MEA. Raw data were sampled at 1000 Hz through the implementation of low-pass filter convolved with a Gaussian kernel. This filtering technique removed high-frequency artifacts and single-unit activity but preserved LFP waveform. To account for the down sampling of the data after filtering, the temporal bin was set to 4 msec. A threshold, determined by calculating the standard deviation (SD) for a 10 min data stream after filtering and down-sampling, was set at three times the SD. For most cases, the receiver operating characteristic curve produced by the filtered data and a Gaussian noise distribution gave the optimal threshold. LFP events were analyzed in binary form, with ones representing supra-threshold time bins and zeros representing sub-threshold time bins. Average event amplitudes divided by the SD of the analog signal were used for calculating the signal-to-noise ratio based on a 10 sec recording for each electrode.

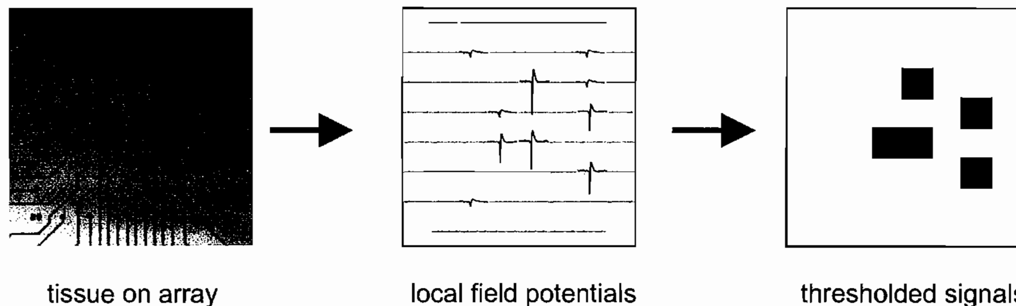



**Fig 2.** Close-up image of MEA, local field potentials of selected electrodes, and activity pattern of threshold signals (activity is shown as black)

2.4 *Extraction of neuronal avalanches*

Upon binning using methods outlined above, the data usually showed long period of inactivity punctuated by brief periods of activity, an observation similar to that reported by Beggs and Plenz on cultured neocortical networks [10, 11]. Owing to the refractory properties of neurons, this observation is expected and reaffirms that the signals recorded on the electrodes were indeed produced by neural activity. The activity periods were characterized using frames and avalanches, where a frame was defined as the pattern of supra-threshold activity on the MEA during one time bin, and avalanches of length N were defined as

consisting of N consecutively active frames preceded and followed by blank frames[14]. Based on previous findings, network dynamics were studied at the critical state, and after shuffling using methods described elsewhere[11], avalanches were extracted at 4 msec time resolutions[10]. In all cases, significant families of avalanches were defined as the clusters of activity patterns that are more similar to each other than those obtained from shuffled data (see ref. 11 for details).

2.5 *Similarity matrices and determination of Lyapunov exponents*

In other to calculate the similarity between avalanches, each $8 \times 8$ two-dimensional frame from an avalanche was converted into a $1 \times 64$ vector (note that the four corner electrodes were always blank). An avalanche of length N was formed by linking N linear $1 \times 64$ vectors together to form a single vector of $1 \times N64$. These single $1 \times N64$ vectors were then compared for similarity by calculating the Boolean similarity. In all cases, the Boolean similarity between avalanche A and avalanche B (*Sim (A, B)*) was defined as the intersection of active electrodes in avalanche A and B divided by the union of active electrodes in avalanches A and B[14].

$$Sim(A,B) = \frac{A \cap B}{A \cup B} \tag{1}$$

From the similarity, the distance ($d_i$) between two avalanches A and B can be calculated as one minus the Boolean similarity between avalanche A and avalanche B:

$$d_i = 1 - Sim(A,B) \tag{2}$$

The Lyapunov exponent ($\lambda$) was calculated by taking the logarithm of the ratio of the final distance and initial distance between avalanches within a given time interval $T$:

$$\lambda = \frac{1}{T}\log_2(\frac{d_f}{d_i}) \tag{3}$$

From the Lyapunov exponent ($\lambda$), the dynamics of the network was determined. Ideally, chaotic dynamical systems produce $\lambda > 0$, attractive dynamical systems generate $\lambda < 0$, and

neutral dynamical systems result in $\lambda = 0$. As a comparison, the slope of dynamical trajectories was also calculated using the formula below:

$$slope = \frac{\Delta d_i}{\Delta T} \quad (4)$$

To ensure that the method of analysis used is not biased towards any one type of dynamical trajectory, different models were developed to test the validity of our analytical approach (see supporting information for details). Figure 3 shows a model in which extracted families of avalanches exhibit attractive, chaotic and neutral dynamics.

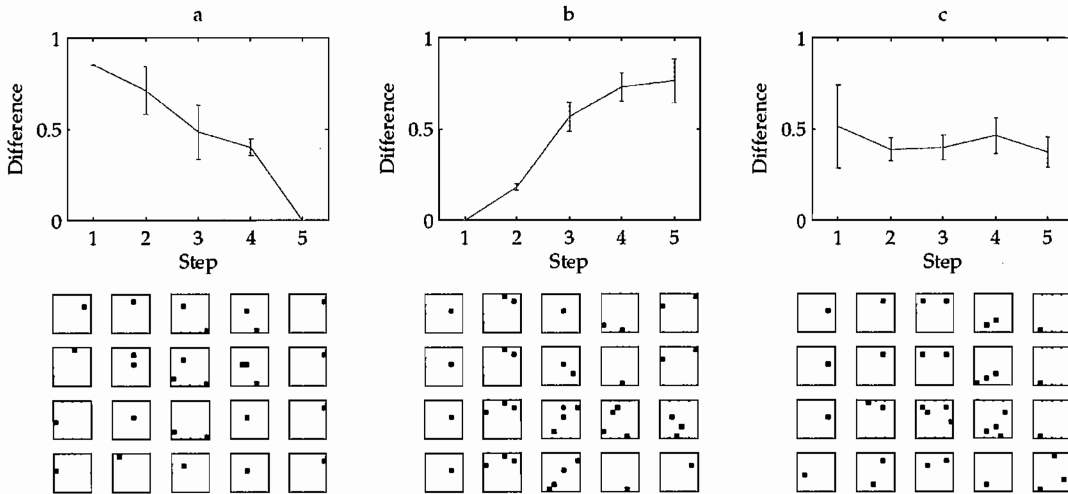



**Fig. 3.** Model of different trajectories: a) represent attractive dynamics, b) represents chaotic dynamics, c) represents neutral dynamics. Families of avalanches are shown below trajectory plots.

## 3. Results

The experimental results can be categorized into two main groups: LPF recordings without the exposure of cortical slices to modulating drugs, and LPF recordings in which neocortical slices were modulated with inhibitory and excitatory drugs (muscimol and bicuculline).

### 3.1 *Native Slice Data*

Figure 4 shows a time raster representing the activity pattern from a cortical slice (without exposure to neuroactive chemicals) recorded using a 60-channel MEA. Considering all electrodes over the entire period of recording may suggest the absence of

recurring patterns of activity. Nonetheless, upon shuffling using methods described elsewhere[10], significant families of avalanches were extracted and analyzed (fig. 5).

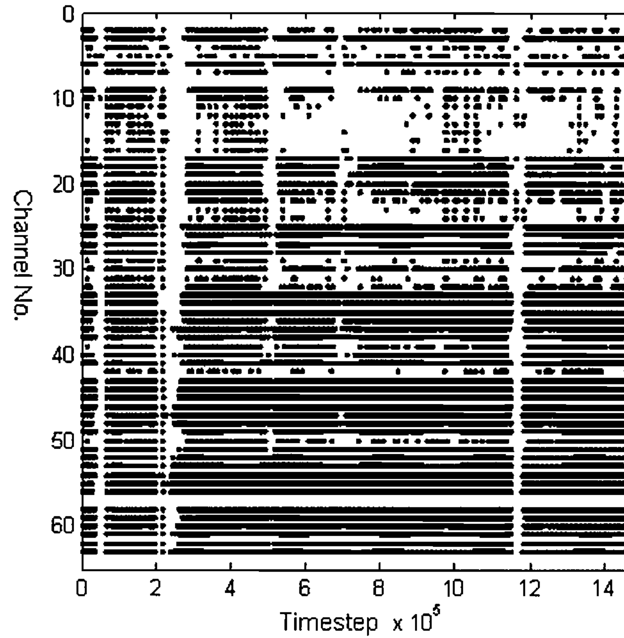



***Fig 4.*** Time raster of supra-threshold activity (active electrodes are shown in blue, inactive electrodes are shown in white)

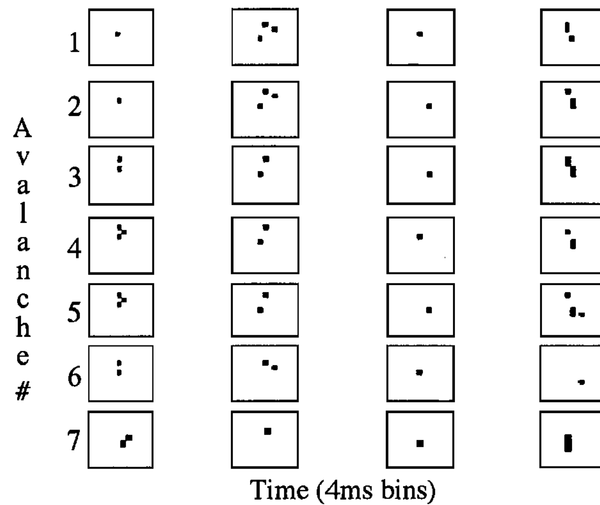



**Fig 5.** Sample extracted families of avalanches (note similarities in activity patterns in each avalanche)

For each family of avalanche extracted, trajectories were closely studied by plotting the distance ($d_i$) versus the time, and calculating the slope and Lyapunov exponent using methods outlined earlier. Figures 6-9 show the trajectory of avalanches of sizes 3, 4, 5, and

6 respectively. With the exception of avalanches of length 3, which showed chaotic dynamics ($\lambda = 0.74 \pm 0.17$), all other trajectories were found to be neutral ($\lambda \sim 0$).

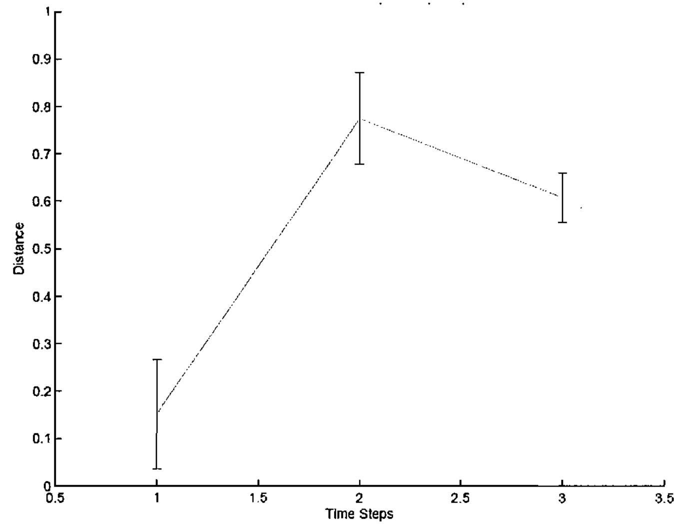



**Fig. 6**. Trajectory of avalanche of length 3 ($\lambda = 0.14 \pm 0.17$, slope = $0.27 \pm 0.5$); time step is 4 msec

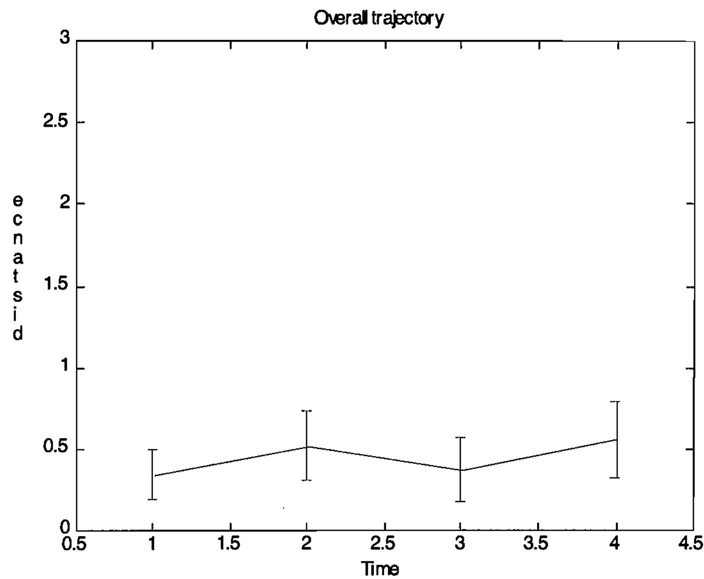



**Fig. 7**. Trajectory of avalanche of length 4 ($\lambda = 0.013 \pm 0.18$, slope = $0.017 \pm 0.05$); time step is 4 msec

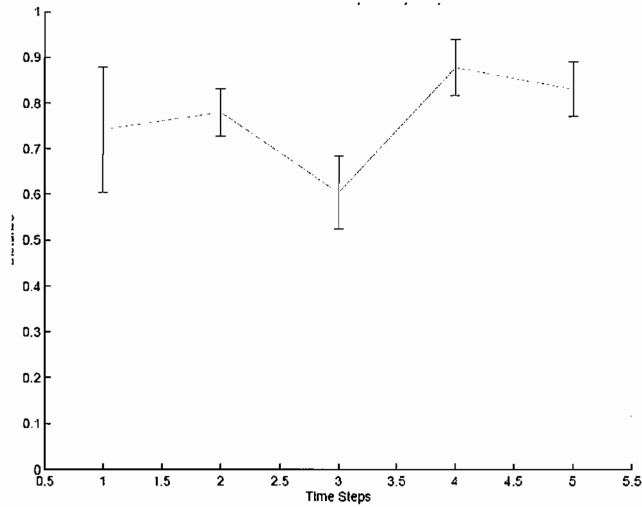



**Fig. 8**. Trajectory of avalanche of length 5 ($\lambda$ = 0.019 $\pm$ 0.13, slope = 0.011 $\pm$ 0.05); time step is 4 msec

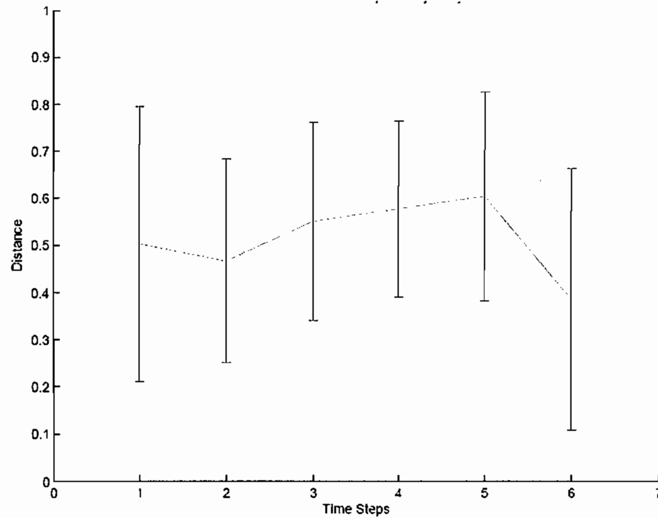



**Fig. 9**. Trajectory of avalanche of length 6 ($\lambda$ = 0.014 $\pm$ 0.27, slope = 0.017 $\pm$ 0.05); time step is 4 msec

We also found that the slopes of the dynamical trajectories were not significantly dependent on the length of avalanches in a family, suggesting that chemical modulations could be undertaken and analysis carried out using avalanches of an arbitrary length greater than three.

Figure 10 shows a scatter plot of the different slopes and $\lambda$ obtained. It is interesting to note that the $\lambda$ values obtained where more dispersed than slope values, suggesting that the Lyapunov exponent is more responsive to changes in dynamics and hence provides a more

sensitive way of quantifying the trajectories of dynamical systems. Thus, all conclusions from this study are based on average Lyapunov exponents calculated from experiments with consistent physical and chemical conditions (slopes are shown for completeness).

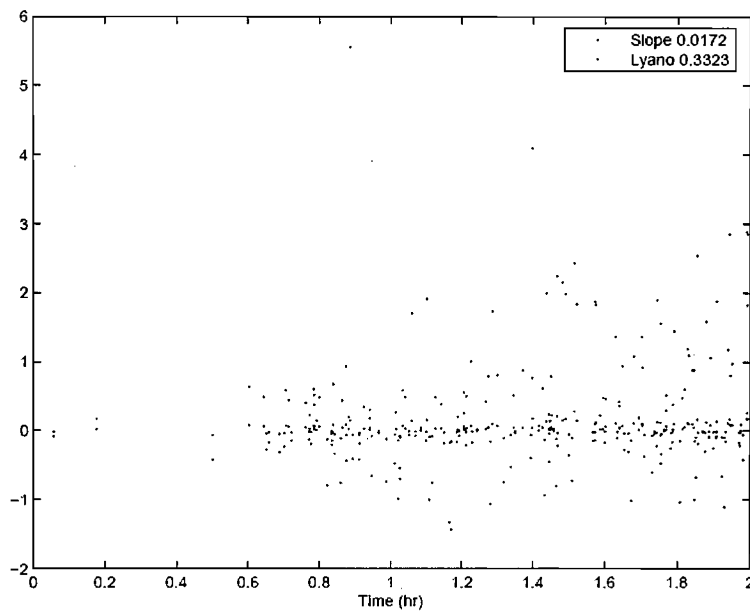



**Fig 10.** Slope and $\lambda$ plotted versus time (average $\lambda = 0.332 \pm 0.941$, average slope = $0.0172 \pm 0.0150$)

3.2 *Chemical Modulation*

Having explored the network dynamics of neuronal circuits without the influence of excitatory or inhibitory chemicals, we endeavored to understand the effect of different neuroactive chemicals on neural network dynamics. Towards this end, we exposed rat cortical slices to low concentrations (10.0 $\mu$M) of two different neuroactive drugs—muscimol ($GABA_A$ receptor agonist) and bicuculline ($GABA_A$ receptor antagonist). Figure 11 and 12 show the trajectories of an avalanche of size 6 before and after modulation with bicuculline respectively (a histogram of the calculated Lyapunov exponents of this family of avalanches is shown in figure 13 and 14). Modulation of neural networks with bicuculline appears to change the absolute mean Lyapunov exponent from $-0.316 \pm 0.519$ to $-0.528 \pm 0.942$. This change in mean compared to the overall standard

deviation indicates that there is no significant change in the Lyapunov exponents before and after modulation with bicuculline. In other words, dynamical trajectories were found to be statistically unaffected by bicucullin modulation.

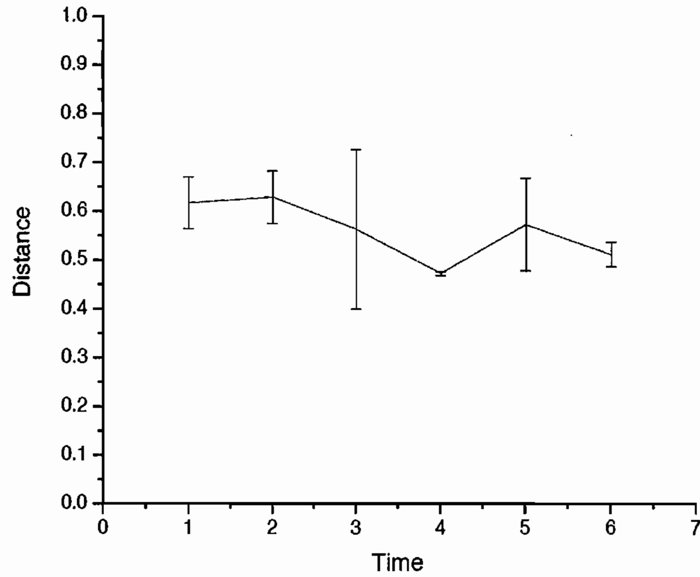



**Fig. 11**. Trajectory of avalanche of size 6 before bicuculline modulation ($\lambda$ = -0.017 $\pm$ 0.07); each time step corresponds to 4 msec.

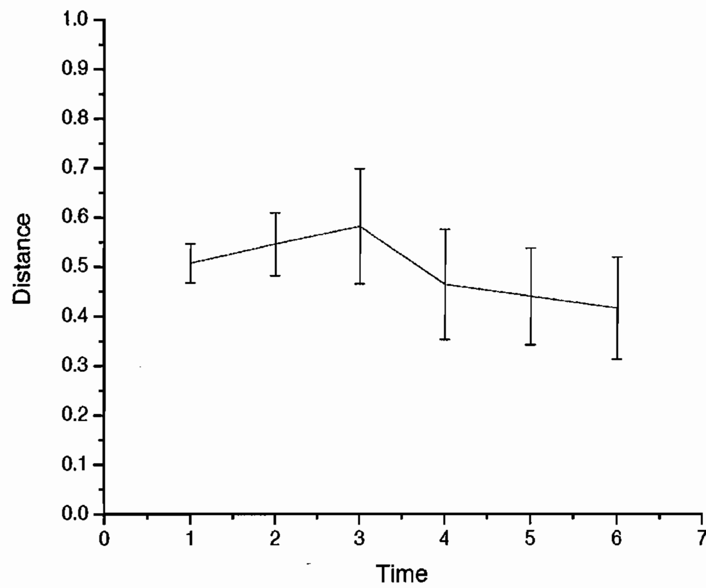



**Fig. 12**. Trajectory of avalanche of size 6 after bicuculline modulation ($\lambda$= -0.026 $\pm$ 0.13); each time step corresponds to 4 msec.

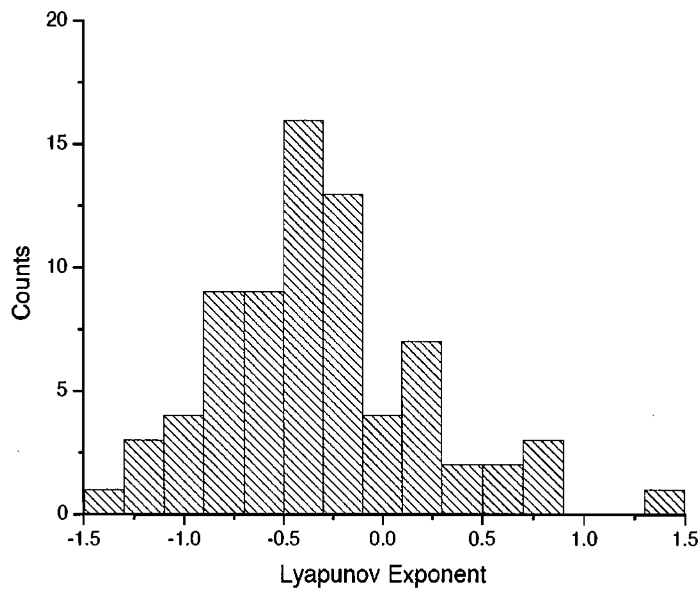



**Fig 14.** Histogram of Lyapunov exponents before bicuculline modulation (mean $\lambda \pm$ SD = -0.316 $\pm$ 0.519)

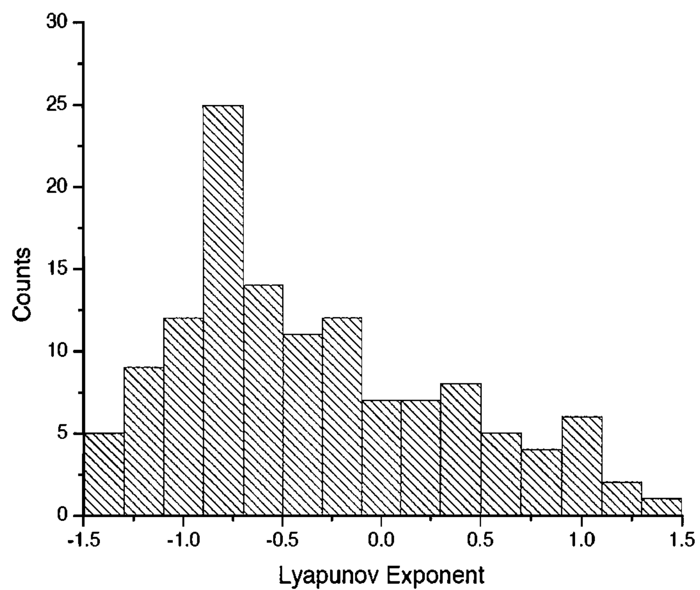



**Fig 13.** Histogram of Lyapunov exponents after bicuculline modulation (mean $\lambda \pm$ SD = -0.528 $\pm$ 0.942)

Having studied the effect of bicuculline on neuronal network dynamics in three independent experiments and observed no significant changes in dynamical trajectories, we tried to understand if another neuroactive drug, muscimol, which is known to elicit an opposite effect, could perturb the network to assume either chaotic or attractive dynamics.

Figures 14 and 15 show the dynamical trajectories of avalanches of length 5 (histogram of calculated Lyapunov exponents are shown in figures 16 and 17) obtained before and after modulation with muscimol respectively.

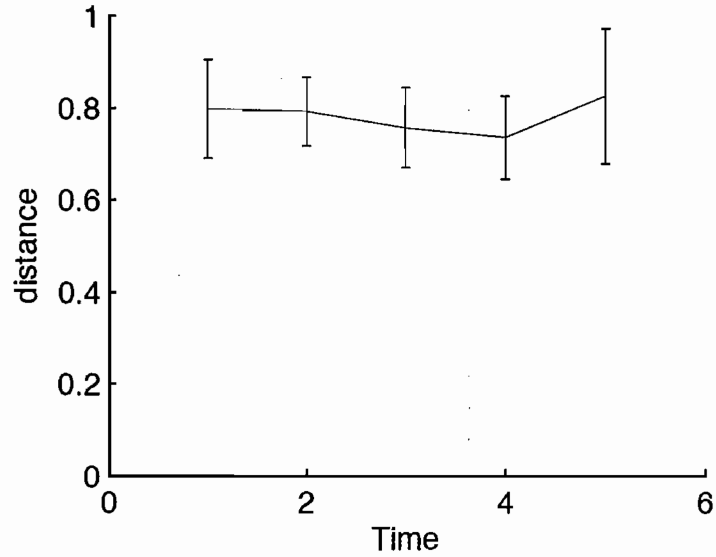



**Fig. 14**. Trajectory of avalanche of length 5 before muscimol modulation ($\lambda = -0.042 \pm 0.013$); each time step corresponds to 4 msec.

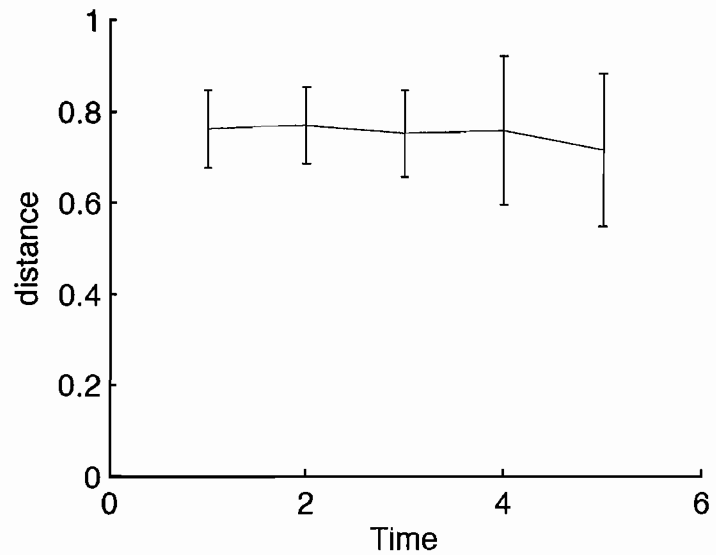



**Fig. 15**. Trajectory of avalanche of length 5 after muscimol modulation ($\lambda = -0.013 \pm 0.011$); each time step corresponds to 4 msec.

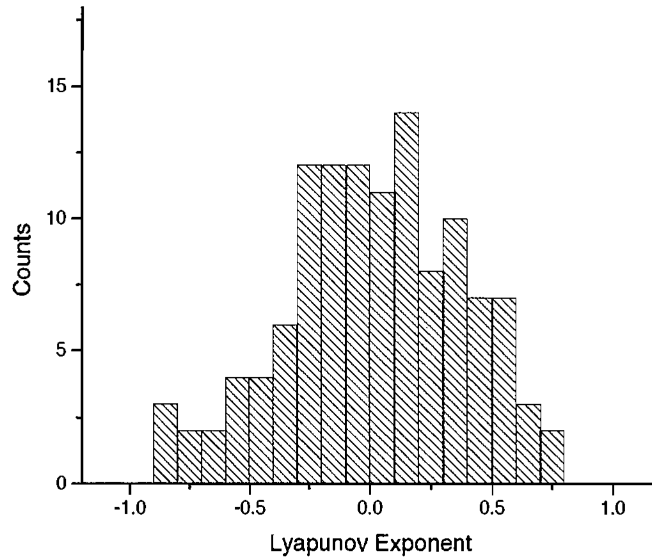



**Fig 16.** Histogram of Lyapunov exponents before muscimol modulation (mean $\lambda \pm$ SD = 0.0172 $\pm$ 0.367)

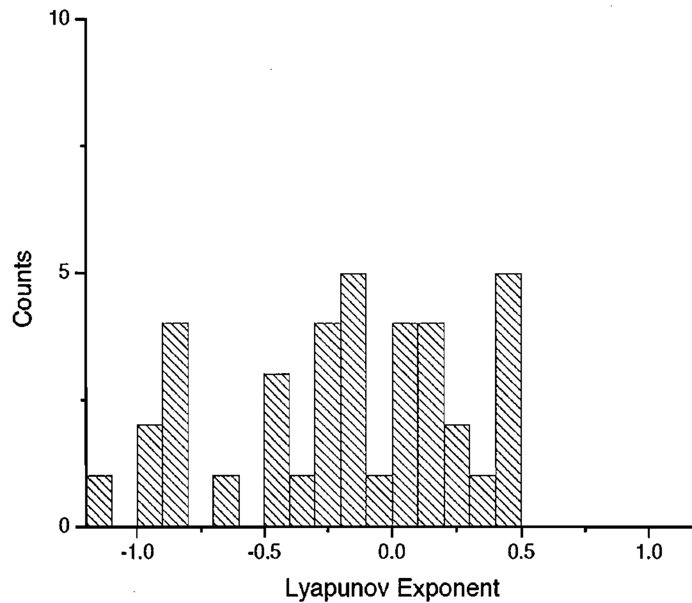



**Fig 17.** Histogram of Lyapunov exponents after muscimol modulation (mean $\lambda \pm$ SD = -0.182 $\pm$ 0.455)

Like the results obtained from modulations with bicuculline, we found that muscimol has no significant effect on the dynamical trajectories of neuronal networks (these findings were reproduced in three experiments). While a numerical difference in mean Lyapunov exponents was observed (0.0172 $\pm$ 0.367 to -0.182 $\pm$ 0.455) upon exposing tissues to muscimol, this difference was not statistically significant within the calculated experimental

errors. Interestingly, however, a significant difference in the distribution of $\lambda$ was observed upon modulating neuronal network dynamics with muscimol.

## 4. Discussions

The results obtained from our experimental approach to understanding neural network dynamics are intriguing for a number of reasons. First, we found that neural networks tend to operate near the critical point with dynamical trajectories that are consistent with a model proposed by Haldeman and Beggs[9]. We were able to observe repeating patterns of activity even hours after recording, and extract significant families of avalanches with lengths ranging from three to eight—albeit the scarcity of extracted avalanches increased with length.

Second, based on our computations of Lyapunov exponents of the trajectories of neural networks, we found that neural networks exhibited dynamics that could theoretically be classified as chaotic, attractive, and neutral. Nonetheless, our results also suggest that within statistical errors, the dynamics of the network is mostly neutral, with individual networks with chaotic or attractive dynamics averaging to produce an overall network dynamics that is essentially neutral. It is interesting to note that the calculated slope, which remained relatively unperturbed over time, was not a consistent way of quantifying the dynamics of networks; the average Lyapunov exponent, however, successfully captured the dynamical trajectories in a way that was experimentally reproduced. While networks with neutral dynamics produced Lyapunov exponents that were not exactly zero, the Lyapunov exponent values obtained were always statistically close to zero.

Third, our exploration of the effects of different modulating drugs on neuronal network dynamics revealed that neural networks with neutral dynamics are very stable. Upon exposing neocortical slices with low concentrations of bicuculline, we observed that while the firing rate of neurons increased, the Lyapunov exponent remained statistically the same. Bicuculline is a direct antagonist of the $GABA_A$ (gamma aminobutyric acid) receptor, which is an inhibitory ligand-gated ion channel[15]. Thus, bicuculline is well documented as an excitatory drug. Based on these pharmacological properties, we expected bicuculline to

not only induce an increase in firing rates of neurons, but also to produce dynamical trajectories that are chaotic. Ironically, we found that bicuculline does not significantly perturb the trajectories of neural networks. We also explored the effect of muscimol, a drug known to be an agonist of the $GABA_A$ receptor, one that induces inhibition in neural networks[15]. Logically, we expect muscimol to lower the firing rate of neurons and hence cause the networks to assume an attractive dynamics. While we found muscimol to lower the firing rate of neurons, it did not significantly affect the trajectories of neural networks. Hence, our results suggest that the dynamical trajectories of neural networks are independent of the firing rates of individual neurons of the network. Furthermore, complex neural networks appear to prefer dynamical trajectories that are mostly neutral with intermittent chaotic and attractive episodes. These findings indicate that neural networks from sliced rat cortex assume mostly neutral dynamics, supporting the hypothesis that neuronal networks can carry out computations and efficient information transmission inasmuch as one-to-one mapping between stimuli and responses is maintained.

## 5. Conclusions

Ultimately, here we demonstrate that neural networks operate near the critical point, with dynamical trajectories that are mostly neutral. While we found neural networks that exhibited dynamics that could be theoretically characterized as chaotic or attractive, our results suggest that, within numerical errors, only neutral dynamical trajectories were statistically significant. We further show that the dynamics of neural networks are very stable and not significantly affected by modulations with inhibitory or excitatory drugs. While these results are tantalizing, further experiments would have to be undertaken to bolster the results reported herein. In addition, inspired by the above findings, future works will focus on devising chemical or physical means to consistently control the dynamics of neural networks, to induce neural networks to assume chaotic or attractive dynamics and vise versa. Insight into controlling neural network dynamics will be invaluable in the pursuit of therapeutic interventions for neurological diseases such as epilepsy and schizophrenia.

## 6. References


1) Amit, D. J. (1989). *Modeling brain function : the world of attractor neural networks.* Cambridge; New York, Cambridge University Press.
2) Hopfield, J. J. (1982). "Neural networks and physical systems with emergent collective computational abilities." *Proc Natl Acad Sci U S A* **79**(8): 2554-8.
3) Seung, H.S. (1996). "How the brain keeps the eyes still." *Proc Natl Acad Sci U S A.* 93(23):13339-44. Wills, T. J., C. Lever, et al. (2005). "Attractor dynamics in the hippocampal representation of the local environment." *Science* **308**(5723): 873-6.
4) Babloyantz, A. and A. Destexhe (1986). "Low-dimensional chaos in an instance of epilepsy." *Proc Natl Acad Sci USA,* **83**(10): 3513-7.
5) Schiff, S. J., K. Jerger, et al. (1994). "Controlling chaos in the brain." *Nature* **370**(6491): 615-20.
6) van Vreeswijk, C. and H. Sompolinsky (1996). "Chaos in neuronal networks with balanced excitatory and inhibitory activity." *Science* **274**(5293): 1724-6
7) Maass, W., T. Natschlager, et al. (2002). "Real-time computing without stable states: A new framework for neural computation based on perturbations." *Neural Computation* **14**(11): 2531-2560.
8) Bertschinger, N. and T. Natschlager (2004). "Real-time computation at the edge of chaos in recurrent neural networks." *Neural Computation* **16**(7): 1413-1436.
9) Haldeman, C. and J. M. Beggs (2005). "Critical branching captures activity in living neural networks and maximizes the number of metastable States." *Phys Rev Lett* **94**(5): 058101.
10) Beggs, J.M., and Plenz D (2003). Neuronal avalanches in neocortical circuits. *J. Neuroscience,* **23**(35): 11167-77,
11) Beggs, J.M., and Plenz D (2004). Neuronal avalanches are diverse and precise activity patterns that are stable for many hours in cortical slice cultures. *J. Neuroscience,* ***24***(22): 5216-29
12) Horn, D. and E. Ruppin (1995). "Compensatory mechanisms in an attractor neural network model of schizophrenia." *Neural Computation* **7**(1): 182-205.

13) Robinson, P. A., C. J. Rennie, et al. (2002). "Dynamics of large-scale brain activity in normal arousal states and epileptic seizures." *Phys Rev E Stat Nonlin Soft Matter Phys* **65**(4 Pt 1): 041924.

14) Jimbo Y, Robinson HP (2000), Propagation of spontaneous synchronized activity in cortical slice cultures recorded by planar electrode arrays. *Bio-electrochemistry, 51*:107–115.

15) Watling, K. J. (1998), *The RBI Handbook of Receptor Classification and Signal Transduction.* RBI. 3rd Ed., 192-122.

16) Tsau Y., Guan L., Wu J. Y., (1999), Epileptiform activity can be initiated in various neocortical layers: an optical imaging study. *J. Neurophysiol.*, 82 (4): 1965 - 1973

**Acknowledgements**: This work would have been impossible without the support and guidance of Prof. John M. Beggs. I appreciate his mentoring during my time in the lab, and his efforts in training me to become a better scientist. I also thank Prof. Robert de Ruyter for reviewing this thesis and offering vital comments to improve it. I extend my appreciations to Wei Chen for assisting me with experiments, and helping with the analysis of different portions of data sets. The Barry M. Goldwater Foundation and the Roland E. McNair Scholars Program were helpful in providing funding to support me during my time in the Beggs Lab.

## 7. Supporting Information

### MATLAB Codes

*Code for extracting families of Avalanches and Analyzing Trajectories*

```
load Feb0706TIMERASTER.mat -mat
clear TIMERASTER
TIMERASTER = T1;
resultsfilename = 'Feb0706TIMERASTERpre'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else    lklllllll
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')

clear TIMERASTER TrajectoryData SlopeData LyapunovData TimesData weights

TIMERASTER = T2;
resultsfilename = 'Feb0706TIMERASTERpost'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
```

```
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')
clear all]




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%
load Feb0806TIMERASTER.mat -mat
clear TIMERASTER
TIMERASTER = T1;
resultsfilename = 'Feb0806TIMERASTERpre'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else
```

```
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')

clear TIMERASTER TrajectoryData SlopeData LyapunovData TimesData weights

TIMERASTER = T2;
resultsfilename = 'Feb0806TIMERASTERpost'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')
clear all
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
load Feb1706TIMERASTER.mat -mat
clear TIMERASTER
TIMERASTER = T1;
resultsfilename = 'Feb1706TIMERASTERpre'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')

clear TIMERASTER TrajectoryData SlopeData LyapunovData TimesData weights

TIMERASTER = T2;
resultsfilename = 'Feb1706TIMERASTERpost'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else
```

```
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')
clear all
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

load Feb1406TIMERASTER.mat -mat
clear TIMERASTER
TIMERASTER = T1;
resultsfilename = 'Feb1406TIMERASTERpre'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')

clear TIMERASTER TrajectoryData SlopeData LyapunovData TimesData weights
```

```
TIMERASTER = T2;
resultsfilename = 'Feb1406TIMERASTERpost'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')
clear all
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%


load Feb2106TIMERASTER.mat -mat
clear TIMERASTER
TIMERASTER = T1;
resultsfilename = 'Feb2106TIMERASTERpre'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
```

```
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')

clear TIMERASTER TrajectoryData SlopeData LyapunovData TimesData weights

TIMERASTER = T2;
resultsfilename = 'Feb2106TIMERASTERpost'

numshuf = 50;
[Events, Locations, IEI, Areas, Runtimes] = runstats2b(TIMERASTER);
[weights] = weightFunction(TIMERASTER);
for i=3:8
    chosenrunlength = i;
    [sFnum, sigFamilyRoster, StartTimes] = FamilyFinder(TIMERASTER, Runtimes,
numshuf, chosenrunlength);
    if sFnum ~= -1
        [trajectory, Slope, Lyapunov, Times] = TrajectoryFunction(sigFamilyRoster,
TIMERASTER, StartTimes, chosenrunlength, sFnum);
    else
        trajectory = -1;
        Slope = -1;
        Lyapunov = -1;
    end;
    TrajectoryData{i} = trajectory;
    SlopeData{i} = Slope;
    LyapunovData{i} = Lyapunov;
    TimesData{i} = Times;
end;

save(resultsfilename, 'TrajectoryData', 'SlopeData', 'LyapunovData', 'TimesData', 'weights')
clear all
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
```

***Code for Calculating Similarities and Distances***

```
function [similarity, StartTimes] = simmatrixDist(TIMERASTER, Runtimes,
chosenrunlength)
%calculates a similarity matrix based on distance, rather than Boolean
%similarity
%6/17/05
% back to the original simmatrix, which only includes nonzero outputs

%New form of simmatrix with two differences from the original: 1. This
%version calulates the matrix for all runs of a given length, including
%runs with no activity (useful for simulations, where all matrices must
%be of the same size for comparison). 2. This version returns an array of
%starting times for all the runs of the chosen run length,
%functional form of runbrowserold. To be used in batchprocessor
%JMB 04/05/01

%create a brief table of results
% number_of_runs = length(Runtimes);
% lengdist = zeros(100, 1);
% for i=1:length(Runtimes)
%       lengdist(length(Runtimes{i})) = lengdist(length(Runtimes{i})) + 1;
% end;
% occupied = find(lengdist ~= 0);

%create cell arrays of start times and start indicies for all runs of each length
%this requires that the brief table of results be run first to get the variable 'occupied'
%initialize cell arrays
% for i=1:max(occupied)
%       starttim{i} = [];
%       startind{i} = [];
% end;
% %load cell arrays
% for i=1:length(Runtimes)
%       %start times in TIMERASTER of each run sought
%       starttim{length(Runtimes{i})} = [starttim{length(Runtimes{i})} Runtimes{i}(1)];
```

```
%       %indicies in Runtimes of start of each run
%       startind{length(Runtimes{i})} = [startind{length(Runtimes{i})} i];
% end;

starttim = [];
startind = [];
for i=1:length(Runtimes)
    if length(Runtimes{i}) == chosenrunlength
        %start times in TIMERASTER of each run sought
        starttim = [starttim Runtimes{i}(1)];
        %indicies in Runtimes of start of each run
        startind = [startind i];
    end;
end;
StartTimes = starttim;

%matrix method- compress each 2-D frame into a 1-D string of binary digits. Next, string
successive
%frames from a run together to form one long row.   Finally, take all runs of the same
length and
%assemble them together in a matrix, each run forming a row of the matrix.   Multiply this
matrix
%by itself to get the number of "hits"

   holder = full(TIMERASTER(:, starttim))';
   for j=1:chosenrunlength-1
      holder = [holder full(TIMERASTER(:, starttim+j))'];
   end;
   hitsmatrix = holder*holder';

%       tic;
%       [r, c] = size(holder);
%       for i=1:r
%           for j=i:r
%               denom = max(sum(holder(i,:)), sum(holder(j,:)));
%               %similarity(i,j) = 1 - VectorDistance(holder(i,:), holder(j,:))/denom;
%               similarity(i,j) = 1 - sqrt(sum(abs(holder(i,:) - holder(j,:))))/denom;
%               similarity(j,i) = similarity(i,j);
%           end
%       end;
%       toc

   %this next section just calculates the Boolean similarity
   %hitsmatrix{i}(r(j),c(j)) is the intersection of elements between
```

```
%pattern r(j) and pattern c(j)
%the stuff in the denominator is just the total elements from both
%patterns minus their intersection, so it gives their union

for i=1:length(hitsmatrix);
      vect(i) = sum(holder(i,:));
end;
NumI = repmat(vect, length(hitsmatrix), 1);
NumJ = repmat(vect', 1, length(hitsmatrix));
DenomMat = NumI + NumJ - hitsmatrix; %this gives the union of the two vectors
HDmat = (NumI - hitsmatrix) + (NumJ - hitsmatrix); %this gives the Hamming
distance between the vectors
[rows, cols] = size(TIMERASTER);
similarity = 1 - (HDmat./DenomMat); %DenomMat/HDmat;
%similarity = 1 - ((HDmat/(rows*chosenrunlength)).^0.2); %DenomMat/HDmat;
%[Ind] = find(DenomMat == 0);
%similarity = hitsmatrix./DenomMat;
%similarity(Ind) = 0;


%      for i=1:length(hitsmatrix);
%            vect(i) = sum(holder(i,:));
%      end;
%      NumI = repmat(vect, length(hitsmatrix), 1);
%      NumJ = repmat(vect', 1, length(hitsmatrix));
%      DenomMat = NumI + NumJ - hitsmatrix;
%      [Ind] = find(DenomMat == 0);
%      similarity = hitsmatrix./DenomMat;
%      similarity(Ind) = 0;


%matrix method- compress each 2-D frame into a 1-D string of binary digits. Next, string
successive
%frames from a run together to form one long row.   Finally, take all runs of the same
length and
%assemble them together in a matrix, each run forming a row of the matrix.   Multiply this
matrix
%by itself to get the number of "hits"
%      i = chosenrunlength;
%      startlist{i} = [];
%      for j=1:length(startind{i})
%          startlist{i} = [startlist{i} Runtimes{startind{i}(j)}(1)];
%      end;
%      holder{i} = full(TIMERASTER(:, startlist{i}))';
```

```
%       for j=1:i-1
%           holder{i} = [holder{i} full(TIMERASTER(:, startlist{i}+j))'];
%       end;
%       %spy2(holder{select});
%       hitsmatrix{i} = holder{i}*holder{i}';
%
%       StartTimes = startlist{i}; %this is the list of times in TIMERASTER of when all the
runs of the chosen length start
%


%some elements of hitsmatrix will be zero, so only calculate similarity for
%nonzero elements %%%I changed this on 3/19/04 to include even zero
%elements in the matrix to make sure that all correlation densities would
%be comparable
%       [r c] = find(hitsmatrix ~= 0);
%       similarity = sparse(zeros(max(r),max(c)));
%       %[rows cols] = size(hitsmatrix{i});
%       %similarity{i} = sparse(rows, cols);
%       %this next section just calculates the Boolean similarity
%       %hitsmatrix{i}(r(j),c(j)) is the intersection of elements between
%       %pattern r(j) and pattern c(j)
%       %the stuff in the denominator is just the total elements from both
%       %patterns minus their intersection, so it gives their union
%       for j=1:length(r)
%           similarity(r(j),c(j)) =...
%               (hitsmatrix(r(j),c(j)))/(hitsmatrix(r(j),r(j)) + hitsmatrix(c(j),c(j)) -
hitsmatrix(r(j),c(j)));
%       end;
%
%
%
%
%       tic;
%       [r c] = size(hitsmatrix);
%       similarity = sparse(r,c);
%       for i=1:r-1
%           for j=(i+1):c
%               if hitsmatrix(i,j) ~= 0
%                   similarity(i,j) = hitsmatrix(i,j)/(hitsmatrix(i,i) + hitsmatrix(j,j) -
hitsmatrix(i,j));
%                   similarity(j,i) = similarity(i,j);
%               end;
%           end;
```

```
%       end;
% toc
%
%
% %some elements of hitsmatrix will be zero, so only calculate similarity for
% %nonzero elements %%%I changed this on 3/19/04 to include even zero
% %elements in the matrix to make sure that all correlation densities would
% %be comparable
% %       i = chosenrunlength;
% %       [r c] = find(hitsmatrix{i} ~= 0);
% %       similarity{i} = sparse(zeros(max(r),max(c)));
% %       %[rows cols] = size(hitsmatrix{i});
% %       %similarity{i} = sparse(rows, cols);
% %       %this next section just calculates the Boolean similarity
% %       %hitsmatrix{i}(r(j),c(j)) is the intersection of elements between
% %       %pattern r(j) and pattern c(j)
% %       %the stuff in the denominator is just the total elements from both
% %       %patterns minus their intersection, so it gives their union
% %       for j=1:length(r)
% %           similarity{i}(r(j),c(j)) =...
% %               (hitsmatrix{i}(r(j),c(j)))/(hitsmatrix{i}(r(j),r(j)) + hitsmatrix{i}(c(j),c(j)) -
hitsmatrix{i}(r(j),c(j)));
% %       end;
% %
% %       similarity = similarity{chosenrunlength};
%
```

***Code for Shuffling Matrices***

```
function rastout = mshuf(rastin)

% matched shuffle of original RASTER
% RASTER is loaded by nnp from MEADATA
% ts_ind is initiated as an n x 3 matrix, n = number of active frames
% each row is then of form [time channel voltage], with one row for each sample, time in
samples
% note the switch of columns 1 and 2 in assignment due to raster data being stored in
MEADATA as rows, whereas we want it in columns
% ts_ind ends up sorted by time, then by channel.
% there are no duplicates at this point left to sort by voltage.
% note: rastin and rastout are n x 64
% routine developed: D.P. 06/2002, mod. D.G. 07/2002
% rasters transposed: d.g. 11/2002
```

```
% Part ONE: Shuffle original RASTER
raster_data_size=size(rastin);
[ts_ind(:,1) ts_ind(:,2) ts_ind(:,3)]=find(rastin);                    % find(array)
returns    [time_j ch_i voltage_ij]

allowed_frames=unique(ts_ind(:,1));
ts_ind(:,2:3)=ts_ind(randperm(length(ts_ind(:,2))),2:3);
ts_ind=sortrows(ts_ind);
rastout = sparse(ts_ind(:,1), ts_ind(:,2), ts_ind(:,3), raster_data_size(1),
raster_data_size(2));

% Part TWO: randomization will have collide same channels into identical time bins
% the sparse-function will add these channel values together, which would change the data
set
% Solution: get all these events and randomize them again over the data set.

same_time=find(diff(ts_ind(:,1))==0);
same_channel=find(diff(ts_ind(:,2))==0);
same_both=intersect(same_time,same_channel);
if(~isempty(same_both))
    duplicates=ts_ind(same_both,2:3);
ts_ind(same_both,:)=[];
channels_with_duplicates=unique(duplicates(:,1));
for k=1:length(channels_with_duplicates)
        channel=channels_with_duplicates(k);
        available_frames=allowed_frames(find(rastout(allowed_frames,channel)==0));
        points_to_distribute=duplicates(find(duplicates(:,1)==channel),:);
frame_order=available_frames(randperm(length(available_frames)));
frame_order=frame_order(1:size(points_to_distribute,1));
ts_ind=[ts_ind;[frame_order points_to_distribute]];

end
    end

rastout = sparse(ts_ind(:,1), ts_ind(:,2), ts_ind(:,3), raster_data_size(1),
raster_data_size(2));
```

***Code for Reading Dendrogram***

```
function [level, sthresh, Zt] = dendroreader(similarity, DistanceType, DendroType)
```

```
%Function to read dendrogram into level array used for plotting rearranged similarity
matrix.
%Takes as input the matrix Z produced by transz function. To get Z, do the following:
% Y = pdist((1 - SimMat), 'Euclid');    where SimMat is the similarity matrix
% Z = linkage(Y, 'single');
% Z = transz(Z);
%The variable Z will be a list of elements given in the order of when they were merged
into binary
%groups.   For example, Z = [1 2 0.5; 1 4 .75; 1 3 1.2].   This means that element 1 and 2
were the most
%similar, and so were merged first (they had a distance between them of only 0.5). Next,
element 4
%was merged with the group that contains element 2.   Thus, element 4 was merged into a
group that
%became (1, 2, 4).   Next, element 3 was merged into a group that became (1, 2, 4, 3).   In
this manner,
%a series of groups could be constructed for every level.

%create Z from similarity matrix
%Y = RunDist((1 - SimMat), DistanceType);    %where SimMat is the similarity matrix
s = 1-similarity;
Y = s(logical(1-triu(ones(size(s)))))';

Zt = RunLinkage(Y, DendroType);

% transform Zt format (= input for dendrogram) into easy readable format
Z = transz(Zt);

%load sthresh
sthresh = Z(:, 3);

%get size of Z
[r c] = size(Z);

%load first level of cell array so that every element is in its own group
for i=1:r+1
    level{1}{i} = i;
end;

%merge elements row by row of Z
waitWin = waitbar(0,'Determining family structure ...');
for i=1:r                                           %do this for every row of Z
    waitbar(i/r, waitWin);
    %create cell array for next level
```

```
    level{i+1} = level{i};

    %identify elements to be merged
    element1 = Z(i, 1);
    element2 = Z(i, 2);

    %identify groups that contain those elements
    group1 = 0;
    group2 = 0;
    j = 1;
    for j=1:r+1         %(j < r) & ~((group1 ~= 0) & (group2 ~= 0))
        if ~isempty(level{i+1}{j})
            if ~isempty(find(level{i+1}{j} == element1))
                group1 = j;
            end;
            if ~isempty(find(level{i+1}{j} == element2))
                group2 = j;
            end;
        end;
        %j = j + 1;
    end;

    %merge the two groups into one group, placing the contents into the group with the
lower index
    level{i+1}{group1} = [level{i+1}{group1} level{i+1}{group2}];

    %purge the contents of the group with the higher index
    level{i+1}{group2} = [];
end;
close(waitWin);

% have to make level non-sparse
for k=1:1:length(level)
    count = 1;
    for m = 1:1:length(level{k})
        if (~(isempty(level{k}{m})))
            newlev{k}{count} = level{k}{m};
            count = count+1;
        end;
    end;
end;

level = newlev;
```

**Code for models used to test bias of our method of analysis**

***TrajectoryFunction***

```
function [] = TrajectoryFunction(sigFamilyRoster, TIMERASTER, StartTimes,
chosenrunlength, numSigFams)


trajectory = [];
Slope = [];
Times = [];
Lyapunov = [];
vect = [];
for numSigFams = 1:length(sigFamilyRoster)
    familyNumber = sFnum
    %get each run from this significant family in vector form
    famSize = length(sigFamilyRoster{sFnum});
    for j=1:famSize
        vect{j} = [];
        for i=1:chosenrunlength
            vect{j} = [vect{j} TIMERASTER(:,
StartTimes(sigFamilyRoster{sFnum}(j))+(i-1))'];
        end;
    end;


    %now construct the average run from all of the runs in the family
    avgVect = [];
    for j= 1:famSize
        avgVect = [avgVect; vect{j}];
    end;
    avgVect = sum(avgVect)/famSize;
```

```
        %now find the Euclidean distance, frame by frame, of each run from the
        %average run
        dists = zeros(famSize, chosenrunlength);
        for i=1:famSize
            for j=1:chosenrunlength
                dists(i,j) = 1-VectorSim(avgVect(((j-1)*64 + 1):(j*64)),
vect{i}(((j-1)*64 + 1):(j*64))); %or VectorDistance
            end;
            m = polyfit(1:chosenrunlength, dists(i,:), 1);
            Slope = [Slope m(1)];
            Times = [Times StartTimes(sigFamilyRoster{sFnum}(i))*0.004/3600];
%0.004 is the binwidth, 3600 gives times(i) in hrs.
            Lyapunov = [Lyapunov log2((chosenrunlength*m(1)+ m(2))/m(2))];
        end;

        %find and plot average of dists
    %       avgDist = sum(dists)/famSize;
    %       figure; errorbar(1:chosenrunlength, avgDist, std(dists)); ylim([0, 2]);

        trajectory = [trajectory; dists];

    end;

    %plot average trajectory with standard deviations
    [r c] = size(trajectory);
    avgTrajectory = sum(trajectory)/r;
    H=figure; errorbar(1:chosenrunlength, avgTrajectory, std(trajectory));
    ylim([0,1])
    saveas(H,'Feb0306-post-trajectory.pdf');
    %end of segment that gets overall trajectory
```

```
edges = -0.5:0.01:0.5;
N = histc(Slope,edges);
H=figure;
bar(edges,N,'histc')
averageSlope = sum(Slope)/length(Slope)
std(Slope)
saveas(H,'Feb0306-post-slope.pdf');

edges = -15:0.2:15;
N = histc(real(Lyapunov),edges);
figure;
bar(edges,N,'histc')
averageLyapunov = sum(real(Lyapunov))/length(Lyapunov)
std(real(Lyapunov))

figure;
plot(Times, Slope, 'b.');
Slope(find(isinf(Slope)))=0;
Slope(find(isnan(Slope)))=0;
SlopeAve = mean(Slope);
hold on
plot(Times, Lyapunov, 'r.');
Lyapunov(find(isinf(Lyapunov)))=0;
Lyapunov(find(isnan(Lyapunov)))=0;
LyanoAve = real(mean(Lyapunov));
hold on;
if and(SlopeAve <0, LyanoAve >0)
legend([' Slope ' num2str(SlopeAve, '%0.3f'),
    ' Lyano ' num2str(LyanoAve, '%0.4f'), ]);
```

```
elseif and(LyanoAve <0, SlopeAve >0)
legend([' Slope ' num2str(SlopeAve, '%0.4f'),
      ' Lyano ' num2str(LyanoAve, '%0.3f'), ]);
else
legend([' Slope ' num2str(SlopeAve, '%0.4f'),
      ' Lyano ' num2str(LyanoAve, '%0.4f'), ]);
end
hold off


%end of segment that gets overall trajectory and plots histogram of slopes,
%lyapunov exponents
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%save cxmodCrit TIMERASTER p sums refperiod connects WeightExponent
chosenrunlength numshuf sigFamilyRoster StartTimes -mat

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%In this segment, plot histogram of slopes and Lyapunov exponents
% for i=1:famSize
%       m = polyfit(1:chosenrunlength, trajectory(i, 1:chosenrunlength), 1);
%       slope(i) = m(1);
%       %times(i) = StartTimes(sigFamilyRoster{sFnum}(i))*0.004/3600; %0.004 is
the binwidth, 3600 gives times(i) in hrs.
%       lyapunov(i) = log2((chosenrunlength*m(1)+ m(2))/m(2));
%       %figure; plot([1 chosenrunlength], [m(2) (chosenrunlength*m(1)+ m(2))]);
%if
%       %you want to plot the fitted line
```

```
% end;
%edges = -0.5:0.01:0.5;
%N = histc(slope,edges);
%figure;
%bar(edges,N,'histc')

%edges = -4:0.2:4;
%P = histc(lyapunov,edges);
%figure;
%bar(edges,P,'histc')

%save Feb0306-pre-trajectory.mat
```

***Code for Testing Attractive Dynamics***

```
%cxmodelAttractive - to try to simulate Attractive dynamics. A test of our
%trajectory programs, to see if they can detect a trajectory that is known
%to converge.
%6/22/05

timesteps = 10000; %144000000; %200000;    %number of time 4ms steps, 250 steps
per second, 150000 in 10 minutes

%initial parameters
r = 8;   %rows
c = 8;   %columns
```

```
p = 0.005;                    %probability of spontaneous activity at a single unit
threshold = 1 - p;   %when membrane potential at a unit is over this, it becomes active
for 1 time step
alpha = 1.0;        %fraction of activity that is transmitted
sums = 1.0;     %total sum of synaptic weights
refperiod = 10;   %number of time steps that unit is refractory after firing
connects = 2;   %number of connections that each unit RECIEVES FROM other units.
Note that this is different from most programs, where
                        %it gives the number of connections sent from the given
                        %unit. This is particular to the Attractive network that we
                        %are modeling here, where all connections eventually funnel
                        %into one common destination. So there are no "fan out"
                        %connections at all. All are fan in.


%set up layers to be connected
power = round(log10(r*c)/log10(connects)); %power gives the number of layers
availables = 1:r*c;
layer = cell(power,1);
for k=1:power
    for j=1:connects^(k-1)
        layer{k} = [layer{k}, availables(floor(rand*length(availables)) + 1)];
        availables = remove(availables', layer{k}(j))';
    end;
end;


%now do attractive connections (starting from last layer back to first layer)
for k=power:-1:length(layer{2})
    for j=1:length(layer{k})
```

```
            connmat{layer{k}(j)} = layer{k-1}(floor((j-0.1)/connects) + 1);
            weightmat{layer{k}(j)} = 1.0; %weightsExp(connects, sums, 0);
        end;
    end;



    %initialize matricies
    spont = sparse(r,c);    %random "membrane potentials" between 0 and 1 for each unit
    ref = sparse(r,c);    %lists how many time steps each electrode will be refractory for
    act = sparse(r,c);    %gives driven activity at each unit, set by spont and connectivity
matrix
    current = sparse(r,c); %status of activity at current time step
    next = sparse(r,c);    %status of activity at next time step
    TIMERASTER = sparse(r*c, timesteps);    %stores times of events for plotting later
using LFPsorter routines

    %march through time steps
    for i=1:timesteps

        %display current status
        %spy3(reshape(current*10,r,c));
        %M(:, i) = getframe;
        %clear M;

        %determine spontaneous activity over threshold
    %       holder = find(rand(length(layer{power}),1) >= threshold);
    %       spont(layer{power}(holder)) = 1;
        %activate nodes at regular times, rather than randomly at any time.
        %This will avoid the problem of multiple activations during a single
```

```
    %trajectory, which could cause noise.
    if mod(i, power*2) == 0
        holder = find(rand(length(layer{power}),1) >= 0.5);
        spont(layer{power}(holder)) = 1;
    end;

%determine units to be activated next time step by current activtiy that is over
threshold
for j=1:length(weightmat)
    if current(j) ~= 0
        k = 1;
        %for k=1:connects
            if ~isempty(weightmat{j})
                if rand <= weightmat{j}(k)
                    act(connmat{j}(k)) = act(connmat{j}(k)) + (alpha * current(j));
%update act matrix
                end;
            end;
        %end;
    end;
end;
holder = find(act >= threshold);    %trim act to keep it from accumulating large
values
act(holder) = 1;



%determine activity at next time step
next = spont + act;
holder = find(ref > 0);    %turn off all units that are already refractory
next(holder) = 0;
```

```
        holder = find(current >= threshold);
        next(holder) = 0;           %turn off all units in next time step that fired in current
time step
        holder = find(next >= threshold);
        next(holder) = 1;           %make next have binary values only
        holder = find(next < threshold);
        next(holder) = 0;

        %determine refractory units for next time step
        ref = ref -1;   %decrement old values
        holder = find(ref < 0);
        ref(holder) = 0;
        holder = find(next >= threshold);   %increment new values
        ref(holder) = refperiod;

        %update current matrix and clear other matricies
        current = next;
        next = sparse(r,c);
        act = sparse(r,c);
        spont = sparse(r,c);
        TIMERASTER(:, i) = reshape(current,r*c,1);

    end;

    %see it
    figure; spy2(TIMERASTER(:,:)); grid;

    %if you want to reverse the output to create a chaotic TIMERASTER, use
    %this:
    [rows cols] = size(TIMERASTER);
```

```
TIMERASTERrev = sparse(rows, cols);
for i=1:cols-1
    TIMERASTERrev(:,i) = TIMERASTER(:, cols+1-i);
end;


%save cxmodelChaosOutput TIMERASTER p sums refperiod connects r c timesteps
–mat
```

***Code for Testing Chaotic Dynamics***

```
%cxmodelChaos - to try to simulate chaotic dynamics. A test of our
%trajectory programs, to see if they can detect a trajectory that is known
%to diverge.
%6/22/05

timesteps = 500000; %144000000; %200000;    %number of time 4ms steps, 250 steps
per second, 150000 in 10 minutes

%initial parameters
r = 3;   %rows
c = 3;   %columns

p = 0.05;               %probability of spontaneous activity at a single unit
threshold = 1 - p;   %when membrane potential at a unit is over this, it becomes active
for 1 time step
alpha = 1.0;       %fraction of activity that is transmitted
sums = 1.0;    %total sum of synaptic weights
refperiod = 10;   %number of time steps that unit is refractory after firing
```

```
connects = 2;    %number of connections that each unit makes to other units. Keep this
low to model a few strong connections.
                        %but not too low, or there will be no chaining of strong
                        %links


%set up layers to be connected
power = round(log10(r*c)/log10(connects)); %power gives the number of layers
availables = 1:r*c;
layer = cell(power,1);
for k=1:power
    for j=1:connects^(k-1)
        layer{k} = [layer{k}, availables(floor(rand*length(availables)) + 1)];
        availables = remove(availables', layer{k}(j))';
    end;
end;

%now do chaotic connections (starting from layer 1 down to last layer)
for k=1:power-1
    for j=1:length(layer{k})
        connmat{layer{k}(j)} = layer{k+1};
        weightmat{layer{k}(j)} = weightsExp(connects, sums, 0);
    end;
end;



%initialize matricies
spont = sparse(r,c);    %random "membrane potentials" between 0 and 1 for each unit
ref = sparse(r,c);    %lists how many time steps each electrode will be refractory for
```

```
act = sparse(r,c);    %gives driven activity at each unit, set by spont and connectivity
matrix
current = sparse(r,c); %status of activity at current time step
next = sparse(r,c);    %status of activity at next time step
TIMERASTER = sparse(r*c, timesteps);    %stores times of events for plotting later
using LFPsorter routines

%march through time steps
for i=1:timesteps

    %display current status
    %spy3(reshape(current*10,r,c));
    %M(:, i) = getframe;
    %clear M;

    %determine spontaneous activity over threshold
%      holder = find(rand(r,c) >= threshold);
%      spont(holder) = 1;
%      if (rand >= threshold)
%           spont(layer{1}) = 1;
%      end;

%activate mother node only at regular time intervals to avoid noisy
%propagation (ie, multiple activations of mother node during a single
%trajectory)
     if mod(i, power*2) == 0
          spont(layer{1}) = 1;
     end;
```

```
    %determine units to be activated next time step by current activtiy that is over
threshold
    for j=1:length(weightmat)
        if current(j) ~= 0
            for k=1:connects
                if ~isempty(weightmat{j})
                    if rand <= weightmat{j}(k)
                        act(connmat{j}(k)) = act(connmat{j}(k)) + (alpha * current(j));
%update act matrix
                    end;
                end;
            end;
        end;
    end;
    holder = find(act >= threshold);   %trim act to keep it from accumulating large
values
    act(holder) = 1;


    %determine activity at next time step
    next = spont + act;
    holder = find(ref > 0);   %turn off all units that are already refractory
    next(holder) = 0;
    holder = find(current >= threshold);
    next(holder) = 0;         %turn off all units in next time step that fired in current
time step
    holder = find(next >= threshold);
    next(holder) = 1;         %make next have binary values only
    holder = find(next < threshold);
    next(holder) = 0;
```

```
    %determine refractory units for next time step
    ref = ref -1;    %decrement old values
    holder = find(ref < 0);
    ref(holder) = 0;
    holder = find(next >= threshold);    %increment new values
    ref(holder) = refperiod;

    %update current matrix and clear other matricies
    current = next;
    next = sparse(r,c);
    act = sparse(r,c);
    spont = sparse(r,c);
    TIMERASTER(:, i) = reshape(current,r*c,1);

end;

%see it
figure; spy2(TIMERASTER(:,:)); grid;

%if you want to reverse the output to create an attractive TIMERASTER, use
%this:
[rows cols] = size(TIMERASTER);
TIMERASTERrev = sparse(rows, cols);
for i=1:cols-1
    TIMERASTERrev(:,i) = TIMERASTER(:, cols+1-i);
end;
%save cxmodelChaosOutput TIMERASTER p sums refperiod connects r c timesteps
-ma
```

*Code for Testing Neutral Dynamics*

```
%cxmodelNeutral - to try to simulate chaotic dynamics. A test of our
%trajectory programs, to see if they can detect a trajectory that is known
%to diverge.
%6/23/05

timesteps = 5000; %144000000; %200000;    %number of time 4ms steps, 250 steps per second, 150000 in 10 minutes

%initial parameters
r = 8;   %rows
c = 8;   %columns

p = 0.05;               %probability of spontaneous activity at a single unit
threshold = 1 - p;   %when membrane potential at a unit is over this, it becomes active for 1 time step
alpha = 1.0;        %fraction of activity that is transmitted
sums = 1.0;    %total sum of synaptic weights
refperiod = 10;   %number of time steps that unit is refractory after firing
numLayers = 4;        %number of activation layers in the network (this sets how long the run length will be)
connects = 1;   %number of connections that each unit makes to other units. Keep this low to model a few strong connections.
                    %but not too low, or there will be no chaining of strong
                    %links
```

```
%set up layers to be connected
numInLayer = floor(r*c/numLayers);
availables = 1:r*c;
layer = cell(numLayers,1);
for k=1:numLayers
    for j=1:numInLayer
        layer{k} = [layer{k}, availables(floor(rand*length(availables)) + 1)];
        availables = remove(availables', layer{k}(j))';
    end;
end;


%now do neutral connections (starting from layer 1 down to last layer)
for k=1:numLayers-1
    for j=1:length(layer{k})
        connmat{layer{k}(j)} = layer{k+1}(j);
        weightmat{layer{k}(j)} = 1; %weightsExp(connects, sums, 0);
    end;
end;


%initialize matricies
spont = sparse(r,c);   %random "membrane potentials" between 0 and 1 for each unit
ref = sparse(r,c);   %lists how many time steps each electrode will be refractory for
act = sparse(r,c);   %gives driven activity at each unit, set by spont and connectivity
matrix
current = sparse(r,c); %status of activity at current time step
next = sparse(r,c);   %status of activity at next time step
```