Exercises for Chapter 7, Quasicriticality

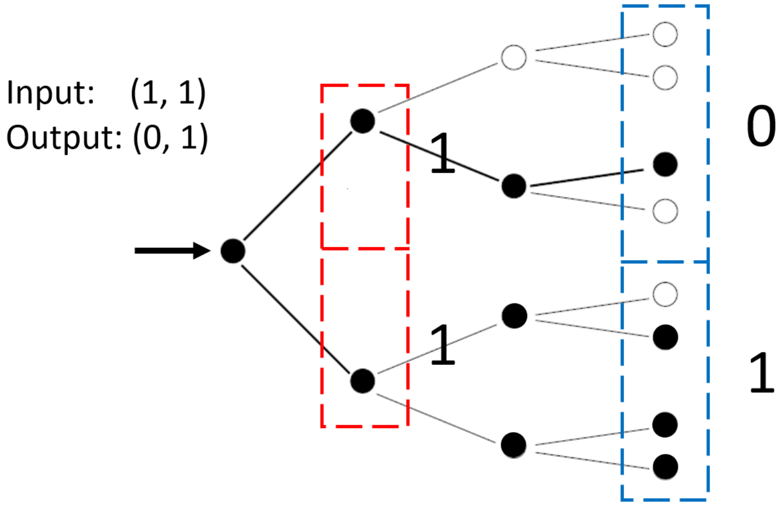
The data reviewed in Chapter 7 suggest the cortex is not operating exactly at the critical point. In addition, the exponents are found to move along a scaling line. We discussed four hypotheses to explain this situation: quasicriticality, slight subcriticality, subsampling, and Griffiths phases. In the exercises below we will illustrate and explore aspects of most of these ideas.

1. Quasicriticality

Recall that the main idea in quasicriticality is that spontaneous activity prevents a neural network from operating exactly at the critical point. There are three specific consequences of this that we will examine here. First, spontaneous activity concatenates avalanches, making their sizes and durations larger. This causes the avalanche distribution plots to become less steep; the magnitude of their exponents thus decreases (Figure 7.2). Second, spontaneous firing always causes activity to be present in the network. This removes the distinction between an absorbing phase and an active phase. The phase transition is thus abolished, and true criticality cannot be achieved (Figure 7.4). Third, the continuous background activity tends to homogenize the variability, causing the susceptibility curve to be reduced and broadened (Figure 7.5). This has a similar affect on the mutual information within the network, where peaks in these curves will also fall.

In what follows, we will examine each of these consequences of quasicriticality by subjecting the branching model (without self-organization) from the Chapter 6 exercises to different amounts of spontaneous background activity. We will approach these exercises not in the order in which they were just explained, but in order of how much computational time they take, from the least to the most.

Mutual information: Here we will plot mutual information curves predicted by quasicriticaity. The branching process model from (Zapperi et al., 1995) is run on a binary tree where each layer of the tree doubles the number of nodes it has. We can select two nodes in the second layer as “inputs,” and, say, eight nodes in the fourth layer as “outputs.” However, there is not a one-to-one correspondence of nodes here, so we will have to coarse-grain the activity in the fourth layer by taking a majority rule, shown in the figure below.



Setting up to measure mutual information in the binary tree network. We can take the configuration of activity in the second layer of the network (dashed red box) as the input. For the output, we can take any subsequent layer, as long as we coarse-grain the activity. Here, the dashed blue box shows how a majority rule can be used to assign activity values to four nodes that are descendants of the nodes from the second layer.

Once this is done, we can stimulate the network many times and calculate the mutual information between layer 2 and layer 4.

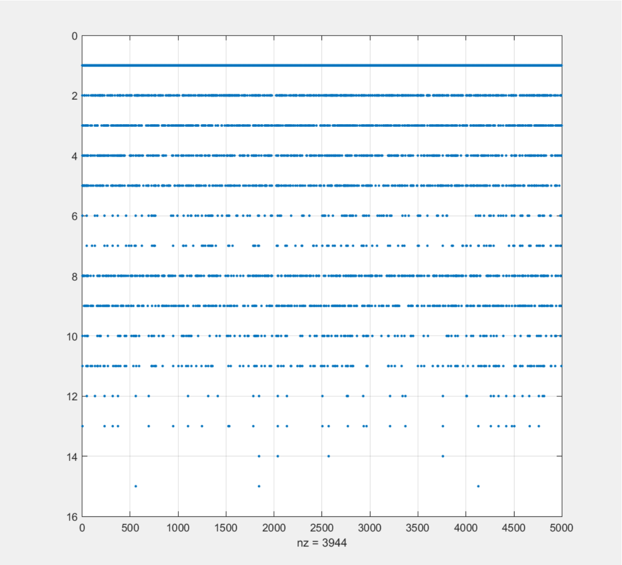
A. Exercise: Before calculating the mutual information, though, let's just see how spontaneous activity affects the raster. To do this, use BranchingProcessFunction like this:

```
[TIMERASTER] = BranchingProcessFunction(Layers, BR, iterations,
Pspont);
```

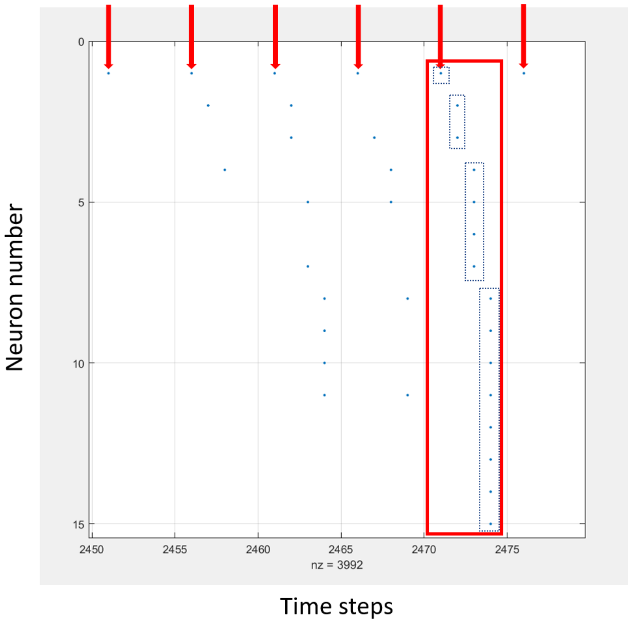
with input values of Layers = 4, BR = 1, iterations = 1000, and Pspont = 0. Take the TIMERASTER and look at it by typing:

```
figure; spy2(TIMERASTER)
```

You should see something like the figure below, where layers are plotted along the y-axis and time steps are along the x-axis:



Note that the number of blue dots fades as you go to higher network layers (from top to bottom of the plot), giving us intuition about how many layers the avalanches tend to propagate. Use the tools at the upper left of the figure to zoom in on one avalanche to see how the layers are represented in the TIMERASTER. If you cannot see the structure that easily, then rerun the function with a larger number of layers, say 8. With more layers the structure should be more obvious.



Close up of a raster plot of avalanches produced by the program BranchingProcessFunction with Layers = 4, BR = 1, iterations = 1000, and Pspont = 0. Neuron numbers from 1 to 15 are along the y-axis; time Steps are along the x-axis. Each successive layer is activated one time step later than the previous layer. Thus, the four layered network here will have avalanches run a maximum of four time steps.

The red arrows at the top point to the single neuron activated at the start of each avalanche. The blue dots show active neurons.

Note that the first avalanche only has a size of 1, the second has a size of 3, and the third has a size of 9. The Fifth avalanche, outlined by the red box, has a size of 15. In this particular avalanche, all the neurons were activated in each layer: 1 in the first layer, 2 in the second layer, 4 in the third layer and 8 in the fourth layer. The active neurons in each layer are outlined by boxes (dotted lines). Notice that each successive layer is activated one time step later than the previous layer.

Verify that you see avalanches in the manner depicted in the figures here. If you do, then you should be ready to move to the next step.

B. Exercise: Now we can explore how information is transmitted through such fading avalanches.

To explain this in slightly more detail, recall from the previous description that each time the single node in the first layer is activated, an avalanche is triggered. Because this single active node transmits to the two nodes in the second layer with a probability of 0.5, then the four configurations of nodes in the second layer (00, 01, 10, 11) will be visited with equal probability; these will be the four input/stimulus configurations for our mutual information measures. After many avalanches, we should have roughly equal numbers of each input configuration. To get the output/response, it would be convenient to observe four configurations (00, 01, 10, 00) at some downstream layer. But notice that in the higher layers, we have more than four nodes. For example, in the fourth layer, we have eight nodes. How will we map the activity in these eight nodes onto merely four configurations? To do this, we will use “coarse graining,” where we will take the average activity in a group of four nodes and round it up or down so that it can be treated as a 1 or 0, respectively. As shown in the first figure for this chapter’s exercises, this procedure creates an output/response that has the same number of nodes as the input. For every avalanche stimulated in this network, we will have an input configuration from layer 2 and a coarse-grained output configuration from layer 4. This allows us to calculate mutual information between the stimulus and the response similar to what we did before in the exercises from Chapter 4.

To calculate the mutual information, take the TIMERASTER that you just produced and feed it into BP_infoPrep:

```
[stim, resp] = BP_infoPrep(Layers, stimLayer, respLayer, iterations, TIMERASTER);
```

where Layers is the number of layers in the network (this can be the same 4 as what you used in the previous exercise, but it does not have to be this number – it can be greater), stimLayer = 2 and respLayer = 4. The more iterations you have, the better statistics you will get for the mutual information measure (10000 is usually good enough for this exercise). The output variable stim contains all the configurations (00, 01, 10, 11) observed in the simulation layer stimLayer and the output variable resp contains all the coarse-grained configurations observed in a downstream response layer, respLayer.

Next, you should take the outputs stim and resp and feed them into FindInformation, using a dim = 2 because this is the dimension, or number of bits, in the input and output streams.

```
[info, Hresp, Hcond] = FindInformation(stim, resp, dim);
```

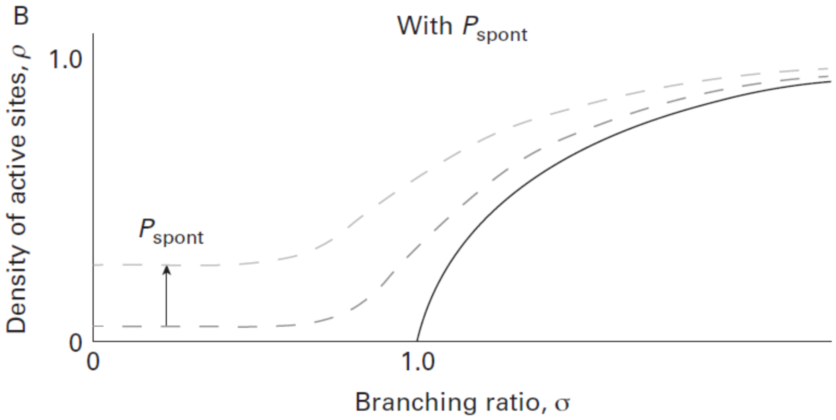
Here, the mutual information in bits will be given in the output variable info; Hresp will give the entropy of the response in bits; Hcond will give the entropy of the response, conditioned on the input, in bits. Now that you have a mutual information value, re-run the programs with the same values, except gradually increase Pspont to 0.01, 0.05.

What happens to the mutual information? Does it decline, as you might expect from quasicriticality? Why not?

To get a more complete picture of what is going on, we will now plot mutual information curves for different values of Pspont. Recall that quasicriticality does not merely predict that mutual information should drop as spontaneous activity is increased, but that the peaks of the mutual information curves should drop and that they should be shifted to lower values of the branching ratio.

C. Exercise: For plotting these curves and comparing them to each other, run PspontInfoLoop; this took about 90 seconds on my laptop to run. You should see a plot of mutual information against the branching ratio for four different values of Pspont. What do you notice about the peaks in terms of their height and their positions? Knowing how these curves look, can you now explain why you might think that increasing Pspont causes mutual information to increase for a fixed branching ratio? In other words, can you pick a branching ratio for which the value of the mutual information curve will *increase* as Pspont increases?

Changes in the phase diagram: In Figure 7.4 from the book (shown below), you can see the prediction that spontaneous activity will abolish the phase transition. As Pspont is increased, the inactive phase to the left of the critical point (where the branching ratio is < 1.0) is no longer inactive. Here we will test this idea in our branching process model.



To do this, we will sample activity in the last layer of a 6-layer network, run for 10000 iterations as we sweep the branching ratio from 0 to 2 and as we sweep Pspont through these values: 0, 0.01, 0.1, 0.25. Doing all this at once can be accomplished by running the program PspontPhaseLoop. With these settings, it took about 20 minutes running on my laptop.

D. Exercise: Run PspontPhaseLoop and then examine the phase diagram it produces. This program automatically loops through several values of Pspont and the branching ratio. While it’s output plot will not look exactly like the schematic diagram shown above, it will still have some characteristic features. You should see a black curve for the case when Pspont = 0, and then three colored curves corresponding to the other nonzero values of Pspont. Answer the following questions:

1. For what values of the branching ratio does the black curve show an inactive or absorbing phase? Recall that an absorbing phase is one where activity does not amplify but is damped (see Figure 3.3 A in the book). Where would you identify the critical point? Ideally, when Pspont = 0, the critical point should exist at the boundary between the inactive/absorbing phase and the active phase.

2. For what values of the branching ratio does the lowest colored curve show an absorbing phase? Where would you identify the critical point?

3. Does the location of the critical point (as seen with the black curve) change if the model is (a) run with more layers? Or (b) run for more iterations? Note that both changes may take considerably more computing time. To make things run faster, it might be helpful to modify the loop and only run through a limited number of branching ratio values, restricted to those that are near the location of the critical point you identified in 1 above.

Changes in exponents: Recall Figures 7.2 and 7.3 from the book that describe how spontaneous activity will affect the exponents. Can we see such changes in our simple branching model programs?

Yes, the TIMERASTERs produced for different values of Pspont can be used to assess how avalanche size and avalanche duration distribution exponents change. To do this, though, we will need to run networks with more layers for more iterations. Unfortunately, this will take longer to run on your computer.

E. Exercise: You should use Layers = 14, BR = 1.0, iterations = 100000, and run the following lines of code:

```
[TIMERASTER] = BranchingProcessFunction(Layers, BR, iterations, Pspont);
[sizeDist, durationDist, SvsT, Events] = AvalancheAnalysis(TIMERASTER);
[CCS, CCD] = GetCCDFs(sizeDist, durationDist);
[alpha, tau, gamma_est, gamma_act, error] = ExponentRelation(CCS, CCD, SvsT, 1, 8);
```

Do this for values of Pspont = 0.0, 0.00005, 0.0001, 0.0005, 0.001.

How do the avalanche size distributions change in appearance as Pspont increases? How do the avalanche size and duration exponents change?

If you want to run these all at once in a single script, just use PspontExponentLoop. This took about 2.5 hours to run on my laptop. While it is possible to use larger values of Pspont, this can take much longer to compute with a 14 layer network.

2. Slightly subcritical

Most data show living neural networks to be operating slightly below the critical point. This naturally leads to the view that the cortex would be chronically subcritical to avoid seizures while still processing information as optimally as possible (Figure 7.6). Could this idea also harmonize with the fact that the cortex is always receiving external inputs and that it is often homeostatically adjusting?

F. Exercise: To investigate the effects of external inputs, we will use BranchingProcessFunction and drive the network with increasing amounts of spontaneous activity. We will see how this affects the branching ratio. Run the function:

```
[TIMERASTER] = BranchingProcessFunction(Layers, BR, iterations, Pspont);
```

Good parameters to use here would be Layers = 10, BR = 1.0, and iterations = 1500. Give it Pspont values in this range: 0.0, 0.10, 0.15, 0.25. To measure the branching ratio in this binary tree network, use BPF_estimator like this:

```
[BRest] = BPF_estimator(TIMERASTER)
```

For each value of Pspont, plot the estimated values of the branching ratio. Mathematically, what type of relationship do you see (e.g., nonlinear, linear, increasing, decreasing)? Note that you gave BranchingProcessFunction a fixed branching ratio of 1.0 as input; this *does not* homeostatically adapt.

G. Exercise: Now we will repeat this experiment, but with a self-organizing branching process. Use the same parameters as before (with the addition of A = 1), with the program SOBP_Pspont_function, like this:

```
[TIMERASTER, BranchingRatio] = SOBP_Pspont_function(Layers, iterations, A, Pspont);
```

For each value of Pspont, plot the output variable BranchingRatio, the average value of the branching ratio after homeostatic adjustment. Mathematically, what type of relationship do you see (e.g., nonlinear, linear, increasing, decreasing)? Note that this function *does* homeostatically adapt.

Comment on how these results may or may not guide our interpretations of the experimental data.

H. Exercise: Next let's look at how the branching ratio changes over time in actual spiking data. Select one of the data sets from the data folders provided with these exercises. For example, go into the folder "OrganotypicData" and then the subfolder "Rat" and load "RatDataSet2" into the Matlab workspace. You can do this by dragging and dropping it, as we previously explained in the exercises to Chapter 3. This will produce a data object called ASDF that will serve as input to the ASDFToSparse function, as shown here:

```
[TIMERASTER, binunit] = ASDFToSparse(ASDF);
```

Next, use the TIMERASTER as input to the function BRplotter:

```
BRplotter(TIMERASTER)
```

This should produce a plot showing the branching ratio over time and a histogram of the branching ratio values measured every 10 seconds. Note that this program may occasionally produce error messages saying “Local minimum possible.” Unfortunately, there is no way to completely avoid these messages from popping up in the data sets we have. Fortunately, they do not always occur, so the estimates are still worthwhile.

Try this for several data sets, then answer the following questions.

1. Is the mean branching ratio in the data sets you sampled slightly less than 1.0?

2. Do the tails of the histogram extend equally above and below the mean of the histogram?

3. Comment on the hypothesis that the cortex consistently operates in the slightly subcritical regime.

3. Subsampling

When exponents are extracted from spiking neural networks, they often show that the exponents change in magnitude over time; they move along a scaling line. While quasicriticality has a possible explanation for why these exponents move, there is another potential explanation: subsampling. Figure 7.7 in the book shows how this could work with a fully connected model of a critical branching process. When all the nodes of the network are sampled, analysis produces exponents that agree with the mean field for a critical branching process. Yet when only some of the nodes are sampled, these exponents grow in magnitude. Thus, subsampling could potentially account for how the exponents move along a scaling line.

This hypothesis has two main components. First, it states that the number of neurons recorded will vary over time. Second, it states that as this number decreases, the exponents will increase in magnitude, meaning that the distributions will decline more steeply.

I. Exercise: We will first examine how the number of neurons recorded can vary over time. One way to see this would be by taking an hour long TIMERASTER and breaking it up into 100 second segments. In each segment, we can count the number of neurons that have fired at least once. Does this number vary widely over time, or is it relatively stable? What fraction of neurons can drop out between successive 100 second segments?

To check this, load a spike data set (organotypic cultures or dissociated cultures) and obtain a TIMERASTER (recall the instructions above, or also from the Chapter 3 exercises). Next, use the program NumNeuronsOverTime:

```
NumNeuronsOverTime(TIMERASTER, Window);
```

The Window parameter gives the number of time samples that should be contained in one sampling period. For example, if the data were binned at 1 kHz, there were 1000 samples per second. To create a sampling period of 100 seconds, Window should be set to 1000 bins/second x 100 seconds = 100000. For a recording that is about one hour (3600 seconds) you should get about 36 data points for each 100 second window. Running the program should produce two plots. The first will just be the number of neurons that fired at least once in each 100 second window. The second will be a histogram of how often each number of neurons was observed.

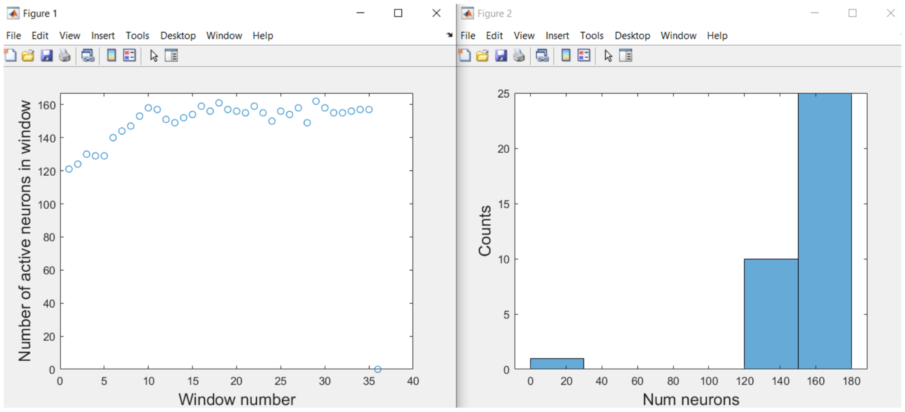


Figure 1
File Edit View Insert Tools Desktop Window Help
Number of active neurons in window
0
20
40
60
80
100
120
140
160
0
5
10
15
20
25
30
35
40
Window number
Figure 2
File Edit View Insert Tools Desktop Window Help
Counts
0
5
10
15
20
25
0
20
40
60
80
100
120
140
160
180
Num neurons

The number of neurons in a recording changes over time. The left plot shows the number of neurons in each 100 second window from a 1 hour recording. The right plot is a histogram showing how often each number of neurons was observed in a window.

Try this with several data sets and then answer the following questions:

1. Is the variability in the number of neurons recorded typically greater than 25% (assuming a window of 100 seconds)?

2. If the window length is now increased to 1000 seconds, what is this new variability?

3. If the window length is reduced to 10 seconds, what is the new variability?

4. What window length would be needed to observe a nearly power-law distribution of avalanche sizes?

J. Exercise: Now that we have some idea of how much the number of active neurons can change over the course of a recording, we can subsample a population by this amount to see if it causes changes in the exponents. To do this, use the function subSampler, like this:

```
[TIMERASTERsub] = subSampler(TIMERASTER, fraction);
```

where fraction is the proportion of neurons from the original TIMERASTER that will be included in the subsampled TIMERASTERsub. For example, if a TIMERASTER has 100 neurons and you select fraction = 0.75, then TIMERASTERsub will have 75 randomly chosen neurons. Note that here we will keep the entire one hour recording but subsample the number of neurons to see how it may change the avalanche distribution exponents.

First, obtain the exponents (alpha, tau, gamma_act) for the entire one hour recording by running this set of functions, as before (review the instructions given in the Chapter 3 exercises if you need to):

```
[sizeDist, durationDist, SvsT, Events] = AvalancheAnalysis(TIMERASTER);
[CCS, CCD] = GetCCDFs(sizeDist, durationDist);
[alpha, tau, gamma_est, gamma_act, error] = ExponentRelation(CCS, CCD, SvsT, LimL,
LimU);
```

Second, subsample the TIMERASTER by some fraction and run these programs again, obtaining a second set of exponents (alphaSub, tauSub, gamma_actSub).

Third, try this for several data sets and for several fractions. Understand that extreme subsampling may reduce the number of data points to nearly zero in a size distribution. Trial and error should give you a sense of what reasonable limits are for each data set. Plot the original exponents in the (alpha, tau) plane as before (Figures 3.12, 5.3). Do the exponents move upward along the scaling line as the number of neurons sampled decreases?

Matlab code used for exercises in this chapter, listed in order of use:

`BranchingProcessFunction`
`spy2`
`BP_infoPrep`
`bin2dec2`
`FindInformation`
`PspontInfoLoop`
`AvalancheAnalysis`
`GetCCDFs`
`ExponentRelation`
`PspontExponentLoop`
`PspontPhaseLoop`
`runStats`
`BPF_estimator`
`SOBP_Pspont_function`

`ASDFToSparse`
`BRplotter`
`NumNeuronsOverTime`
`subSampler`