Synchrony Data Project - Lybrand Lab

October 31, 2024

Contents

1 Logic

1.1 Clustering Coefficient

Let c_{ij} represent an existing connection (i.e. Synchronization Index above average) between node i and node j. The following set, N_i, then, expressed all nodes, n, that are connected to node i, where C represents a set of all existing connections within the network:
$N_i = \{n_j : c_{ij} \in C\}$

For convenience, $|N_i| = l_i$. If node i is connected to 3 nodes, $l_i = 3$.
For node i, a connection c_{ij} is referred to as a direct connection, while connection c_{jk}, where both n_j and n_k are elements of N_i, is an indirect connection.

The clustering coefficient of node i is calculated by observing the indirect connections, defined by the following expression:
$CC_i = \frac{|\{c_{jk} : n_j \in N_i, n_k \in N_i\}|}{l_i(l_i-1)}$

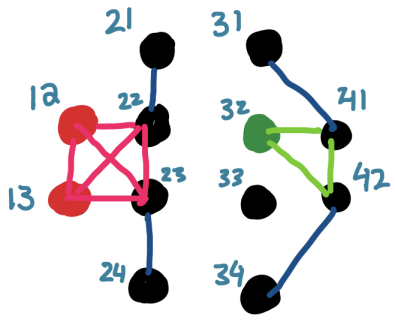
Thus, a high Clustering Coefficient suggests that the nodes connected to node i are all strongly connected to each other. For a node with a clustering coefficient of 1, every possible indirect exists. We call a node "significant" if its Clustering Coefficient is above average for its phase. A node is "perfect" if its Clustering coefficient is equal to 1. By the range of a Clustering Coefficient, all perfect nodes are significant, but not all significant nodes are necessarily perfect.

1.2 Network

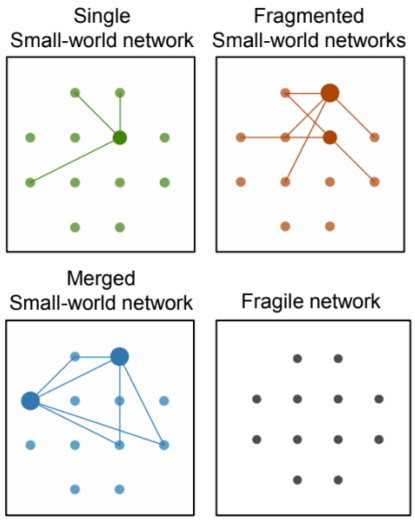
A perfect node's network is defined as the complete set of channels that connect to the perfect node. For example, if channel 12 is a perfect node and is connected to channels 13, 22, and 42, the set of nodes 12, 13, 22, 42 would be considered channel 12's network. By definition, all elements of a perfect node's network must be connected to all other elements of that network. Now, suppose channel 13 is also a perfect network. All of channel 13's connections must then be connected and all of channel 12's connections (that is all elements of channel 12's network) must be connected to channel 13. Thus, by definition of the Clustering Coefficient, we conclude that, if 2 or more perfect nodes are connected, their network must be identical. We record each perfect node's network as a distinct network. In the image below, we notice 2 distinct networks, one (red) centered around channels 12 and 13, and one centered around channel 32:
For the sake of consistency with contemporary Graph Theory conventions, we refer to these networks as Small-World Networks (SWNs).

1.3 Classification of Networks

We classify maps into 4 categories, dependent on the presence of networks and the connection between them. Maps with no significant nodes, and therefore no networks, are considered fragile. Maps with a singular significant node are considered single SWNs. In maps with multiple significant nodes, if the significant nodes are connected, and therefore form a singular network (refer to 1.2), such



maps are considered Merged SWNs. Finally, if not all of the significant nodes are connected, we refer to the map as having Fragmented SWNs.



2 Code

2.1 Data Organization

The first step of running the code is to ensure that the files are organized properly. The code, as it is written, presumes 3 distinct directories: a Data directory, an Origin Directory, and a Destination directory. The Data directory must be the path to a folder containing all the h5 files. The origin directory should be a blank folder which, after the first component of the code is finished running, will house the raw unsegmented data derived from the h5 files in the form of CSV files. Finally, the Destination directory should be a blank folder that will house the output of the entire code. After the code is done running, the Destination folder will have a folder for each individual well in the dataset. Inside each well's folder will be 4 more folders: CSV Files, Synchrony Maps, Heatmaps, and Other Data.

- CSV Files - A folder containing CSV files produced by splitting the CSV file derived from the h5 file.
- Synchrony Maps - A folder containing a PNG Image of the synchrony map for each phase
- Heatmaps - A folder containing a PNG Image of the synchrony indices between every pair of channels visualized as a heatmap
- Other Data - A folder containing a series of spreadsheets. Data for the clustering coefficients of every channel, Synchrony Data for each pair of channels, Independent Network Data, the number of perfect nodes, and a scatterplot of the size of each network for each phase can be found in the folder.

NOTE: All of the items in the CSV files folder MUST be of type .csv and must be proper MEA-derived CSV files, otherwise the code will crash and fail to perform the operation successfully.

2.2 .h5 Files

This section dissects the portion of the code that reads through the h5 files in data_dir (the aforementioned data directory) and saves them in or_dir (Origin Directory).

```
for filename in os.listdir(data_dir):
    print(filename)
    f = os.path.join(data_dir, filename)
    file_name_list = filename.split('_')
```

The code above creates a list of the filenames in data_dir and iterates through the h5 files. The last line separates the filename by underscores to interpret all the attributes of the file.

```
    if '.h5' in file_name_list[-1]:
        data = McsPy.McsData.RawData(f)
        rec = data.recordings[0]
        stream = extract_streams(rec)
        ana_str = stream['Analog'][0]
        channels = list(ana_str.channel_infos.keys())
        print(len(channels))
        ch_data = {}
        for i in range(4):
            temp_data = {}
            for j in range(12):
                data = McsPy.McsData.RawData(f)
                rec = data.recordings[0]
                stream = extract_streams(rec)
                ana_str = stream['Analog'][0]
                temp_data[j] =
                    list(ana_str.get_channel(channels[i*12+j])[0])
```

The code above uses the McsPy package to extract the data from the h5 file and format it in a dictionary to save later as a CSV.

```
            fname = 'Well_'+str(i)
            if 'Gabazine' in filename:
                fname+= r"_GABA"
            elif 'lido' in filename:
                fname+= r"_Lidocaine"
            elif "CNQX_AP5" in filename:
                fname+= r"_Glutamate"
            else:
                fname+= r"_Baseline"
            if 'Control_8w' in filename:
                fname+= r"_Control"
```

```
        else:
            fname+= r"_Blast"
        fname += r".csv"
```

The portion of the code above is dependent on the experiment. It's a series of if statements that analyze the name of the h5 file and create a name for the CSV file.

```
        destination = os.path.join(or_dir, fname)
         pd.DataFrame.from_dict(temp_data).to_csv(destination)
```

The above piece of code saves the data frame as a CSV.

2.3 Synchrony and Clustering Coefficient Analysis

The following code analyzes the raw data in the h5 files and generates synchrony maps, heatmaps, and other data.

```
for file in os.listdir(or_dir):
    well = str(file[:-4])
    completed_wells = []
    if well not in completed_wells:
        print(file)
        control = pd.read_csv(os.path.join(or_dir,file))
```

The above code iterates through and reads the CSV files extracted from the h5 files.

```
        dest_well = os.path.join(dest_dir, well)
        try:
            os.makedirs(dest_well)
        except:
            continue
        csv_path = os.path.join(dest_well, 'CSV_Files')
        try:
            os.makedirs(csv_path)
        except:
            continue

        synch_map_path = os.path.join(dest_well, 'Synchrony Maps')
        try:
            os.makedirs(synch_map_path)
        except:
            continue

        synch_cc_data = os.path.join(dest_well, 'Other Data')
        try:
            os.makedirs(synch_cc_data)
        except:
            continue
        heat_dir = os.path.join(dest_well, 'Heatmaps')
        try:
            os.makedirs(heat_dir)
        except:
            continue
```

The code shown above creates 4 folders within each well's folder (refer to Section 2.1 Data Organization).

```
        for i in range(n):
            #split into n files
            df = control.iloc[int(i * (len(control)/n)): int((i+1) *
                len(control)/n), :]
```

```
        df.to_csv(os.path.join(csv_path, well + '_' + str(i)
            +'.csv'), index=False)
        print(well + '_' + str(i) +'.csv')
```

This bit of code splits the large CSV file derived from the .h5 file into n files, where n is a preset number; n is set right above the code shown in Section 2.2 and can be set to any number less than the number of data points in the h5 file.

```
phases = []
clus_coeff = []
n_nodes = []
total_conn = []
sig_node_well = []
syn_data = {'threshold': []}
for channel in channels:
    syn_data[channel] = []
```

This code initiates a series of lists that will be used to hold the data until they are saved.

```
for p, file in enumerate(os.listdir(csv_path)):

    data = pd.read_csv(os.path.join(csv_path, file))
    print(file[:-4])

    if file[:-4].split(sep='_')[-1] == 'Control':
        phases.append(0)
    else:
        phases.append(int(file[:-4].split(sep='_')[-1]))

    r_dict = {'Unnamed: 0': 'time'}
    for i in range(12):
        r_dict[str(i)] = 'ch' + str(channels[i])
    data = data.rename(r_dict, axis='columns')
    print(data.keys())
    t = data['time']
    for i in range(12):
        r_dict[str(i)] = 'ch' + str(channels[i])
    data = data.rename(r_dict, axis='columns')
```

This block of code reads each of the (now split up) files and renames the indices to channel names.

```
        synchron_data = synch_data(data, channels, t) # extracts
            synchrony index data
        print(synchron_data)
```

Here we call the synch_data function that calculates the synchronization index for every pair of channels. The function is shown below:

```
def synch(ch1, ch2,t):
# Select Channels to compare
# Change to channels of interest (e.g. ch12, ch21, etc.)
    s1 = ch1
    s2 = ch2

# Set up FFT and convolution
# Create complex Morlet Wavelet
    srate = len(ch1)
    center_freq = 5                                    # filter frequency
    time = np.linspace(-1, 1, srate)

  # Create complex sine wave
    sine_wave = np.exp((1j*2) * np.pi * center_freq * time)

  # Create Gaussian window
    s = 7 / (2 * np.pi * center_freq)
    gaus_win = np.exp((-np.power(time,2) / (2 * np.power(s,2))))

  # Create Morlet Wavelet
    cmw = np.multiply(sine_wave, gaus_win)
    half_wavN = (len(time)-1)/2

  #FFT of wavelet
    n_wavelet = len(cmw)
    n_data   = len(t)
    n_conv   = n_wavelet + (n_data-1)

  #FFT of wavelet
    waveletX = np.fft.fft(cmw)
    max_waveletX = np.amax(waveletX)
    waveletXf = np.divide(waveletX, max_waveletX)

  # compute Hz for plotting
    hz_min = 0.0
    hz_max = np.floor((n_conv/2)+1)
    hz_half = np.multiply(srate, 0.5)
    hz = np.linspace(hz_min, hz_max, len(t))


  ## Calculate phase angle for each channel
  # analytic signal of channel 1
    fft_data = np.fft.fft(s1)                          #fft of
        channel 1
    ch1_conv = np.multiply(waveletX, fft_data)         #convolution
        of channel 1
    an_s1 = np.fft.ifft(ch1_conv)                      #inverse fft
        to return analog signal
```

```
  # collect real and phase data
    phase_data1 = np.angle(an_s1)
    real_data1 = np.real(an_s1)

      # analytic signal of channel 2
    fft_data = np.fft.fft(s2)                                #fft of
        channel 2
    ch2_conv = np.multiply(waveletX, fft_data)               #convolution
        of channel 2
    an_s2 = np.fft.ifft(ch2_conv)                            #inverse fft
        to return analog signal

      # # collect real and phase data
    phase_data2 = np.angle(an_s2)
    real_data2 = np.real(an_s2)

      ### CALCULATE PHASE ANGLE DIFFERENCE
    phase_angle_differences = phase_data2-phase_data1        #phase
        angle difference
    euler_phase_differences = np.exp(1j*phase_angle_differences) #euler
        representation of angles
    mean_complex_vector = np.mean(euler_phase_differences)   #mean
        vector (in complex space)
    phase_synchronization = np.absolute(mean_complex_vector) #length of
        mean vector (M from Me^ik)
    return phase_synchronization

#Given Data array, Channels in Data, and the number of measurements for
    each timepoint (t), this function creates a dictionary that
    contains all combinations of synchrony indices
def synch_data(data, channels,t):
  synchron_data = {}
  for i in channels:
    temp = {} #new dictionary for each channel
    sum = 0
    for j in channels:
      if i != j: #iterates through all non-equal channel pairs
        key = "ch"+str(j)
        ch1 = data["ch"+str(i)]
        ch2 = data["ch"+str(j)]
        temp[key] = synch(ch1, ch2,t) #calculates and stores synchrony
            index
        sum += synch(ch1, ch2,t)
    temp["avg"] = sum / (len(channels) - 1) # uses Sum to calculate
        Average
    synchron_data[i] = temp # Adds channel's synchrony data to the
        master dictionary
  return synchron_data #returns a dictionary of dictionaries
```

The code above is, in the actual python file, in the first few lines, with all of the other functions. The code below is a continuation of the code that calls the functions and thus continues to iterate through every phase of every well.

```
#generate heatmaps
l = list(synchron_data.values())
for ch in range(len(l)):
    l[ch]=list(l[ch].values())
hm = np.random.rand(12,12)
for i in range((12)):
    hm[i][i] = 1
    for j in range(12):
        if j < i:
            hm[i][j] = l[i][j]
        elif j>i:
            hm[i][j] = l[i][j-1]
plt.figure(figsize=(12, 12))
heat_map = sns.heatmap(hm, linewidth=1, annot=True)
plt.title(well + ' Phase ' + str(p))
plt.savefig(os.path.join(heat_dir, str(well) + ' Phase ' +
     str(p) + ' Heatmap'))
plt.close('all')
```

The code above iterates forms a matrix with the synchrony data and saves this matrix as a heatmap using the **Seaborn** pathway.

```
sum_val = 0
for channel in synchron_data:
    sum_val += (synchron_data[channel]["avg"])
threshold = sum_val / len(synchron_data.keys())

#synchrony data
for channel in channels:
     syn_data[channel].append(synchron_data[channel].copy())
syn_data['threshold'].append(threshold)
```

This block of code reformats the synchrony data into a dictionary and calculates the connection threshold, which is equivalent to the average synchrony index across all pairs of channels across the phase.

```
#calculates connections
for channel in synchron_data:
    for chan in synchron_data[channel]:
        synchron_data[channel][chan] =
            [synchron_data[channel][chan],
            synchron_data[channel][chan] >= threshold]

# creates a dictionary containing all connections
connections = {}
```

```
for chan in synchron_data:
    temp = []
    for channel in synchron_data[chan]:
        if channel != 'avg':
            if synchron_data[chan][channel][1]:
                temp.append(int(channel[2:]))
    connections[chan] = temp
```

Using the threshold, we generate a dictionary with all of the connections between channels.

```
# Generates Clustering Coefficient data
cluster_coeff = {}
print(connections)
for channel in connections:
    sum_connections = 0
    for chan in connections[channel]:
        for chan2 in connections[channel]:
            if chan2 != chan:
                if chan in connections[chan2]:
                    sum_connections += 1
    try:
        cluster_coeff[channel] = sum_connections / 2 /
            nCr(len(connections[channel]), 2)
    except:
        cluster_coeff[channel] = 0


avg_cc = np.mean(list(cluster_coeff.values()))
clus_coeff.append(avg_cc) # calculates and records average
    clustering coefficient
cc_threshold = avg_cc

# calculates and records numbers of significant nodes (nodes
    with CC above average)
n_node = 0
sig_nodes = cluster_coeff
sig_node_well.append(sig_nodes)
for channel in channels:
    if cluster_coeff[channel] > avg_cc:
        n_node += 1
n_nodes.append(n_node)
```

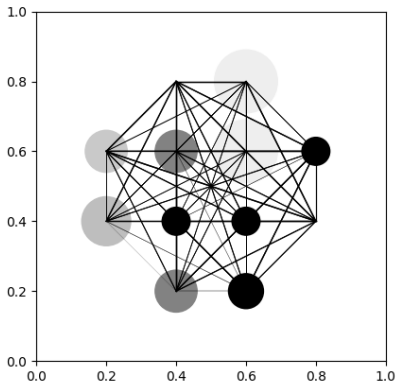
The code then calculates the Clustering Coefficient of each channel, making use of a nCr function. The average clustering coefficient and the number of significant nodes are calculated and saved.

```
#Complete Synchrony Maps
    # Creates coordinates
```

```
coords = []
for i in range(4):
    for j in range(4):
        coords.append([.2 * (i + 1), .2 * (5 - (j + 1))])
coords.remove([.2, .2])
coords.remove([.2, .8])
coords.remove([.8, .2])
coords.remove([.8, .8])

k = max_val(cluster_coeff)

channel_coord = {}
for i in range(len(channels)):
    channel_coord[channels[i]] = coords[i]

# Generates template for map
figure, axes = plt.subplots()
axes.set_aspect(1)
plt.xlim(0, 1)
plt.ylim(0, 1)
for channel in channel_coord:
    draw_circle = plt.Circle(tuple(channel_coord[channel]),
        0.01*(len(connections[channel])),
                          color=str(1 -
                              (cluster_coeff[channel] ** 5 +
                              1 - k)))
    axes.add_artist(draw_circle)
axes.add_artist(draw_circle)
for i in channels:
    for j in channels:
        if i != j:
            ch1 = data["ch" + str(i)]
            ch2 = data["ch" + str(j)]
            connectpoints(channel_coord[i], channel_coord[j],
                synch(ch1, ch2, t))
phat = os.path.join(comp_synch, file[:-4] +
    '_Complete_Synchrony_Map')
plt.savefig(phat)
plt.close('all')
```

This code creates a template for the synchrony maps and uses the Clustering Coefficient and Synchrony Index data to generate the maps. A **complete** synchrony map plots has a line between every pair of channels, regardless of whether a connection exists (i.e. the synchrony index is over the threshold). The thickness of the lines is proportional to the synchrony index between the channels. As for the nodes, the color of the node is dependent on the clustering coefficient, with a darker node having a higher clustering coefficient. The size of the nodes is proportional to the number of nodes that it's connected to.

Example:

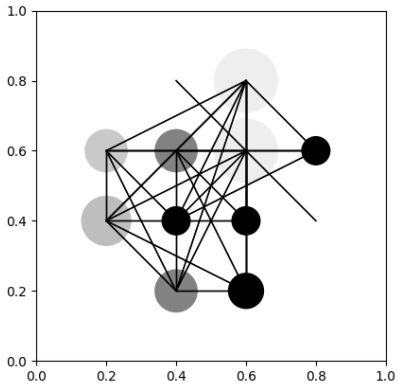


```
#Normal Synchrony Maps:
    # Creates coordinates
    coords = []
    for i in range(4):
        for j in range(4):
            coords.append([.2 * (i + 1), .2 * (5 - (j + 1))])
    coords.remove([.2, .2])
    coords.remove([.2, .8])
    coords.remove([.8, .2])
    coords.remove([.8, .8])

    k = max_val(cluster_coeff)

    # Assigns channels to their respective coordinates
    channel_coord = {}
    for i in range(len(channels)):
        channel_coord[channels[i]] = coords[i]

    # Generates template for map
    figure, axes = plt.subplots()
    axes.set_aspect(1)
    plt.xlim(0, 1)
    plt.ylim(0, 1)
    for channel in channel_coord:
        draw_circle = plt.Circle(tuple(channel_coord[channel]),
            0.01 * (len(connections[channel])),
                                color=str(1 -
                                    (cluster_coeff[channel] ** 5 +
                                    1 - k)))
        axes.add_artist(draw_circle)
```

```
axes.add_artist(draw_circle)
for i in channels:
    for j in channels:
        if i != j:
            ch1 = data["ch" + str(i)]
            ch2 = data["ch" + str(j)]
            if synch(ch1, ch2, t) >= threshold:
                connectpoints(channel_coord[i],
                    channel_coord[j], 0)

phat = os.path.join(norm_synch, file[:-4] +
    '_Normal_Synchrony_Map')
plt.savefig(phat)
plt.close('all')
```

This code creates a template for the synchrony maps and uses the Clustering Coefficient and Synchrony Index data to generate the maps. A **normal** synchrony map plot only plots lines between existing connections (i.e. pairs of channels where the synchrony index is over the threshold). The lines are of uniform thickness. As for the nodes, the color of the node is dependent on the clustering coefficient, with a darker node having a higher clustering coefficient. The size of the nodes is proportional to the number of nodes that it's connected to.

Example:



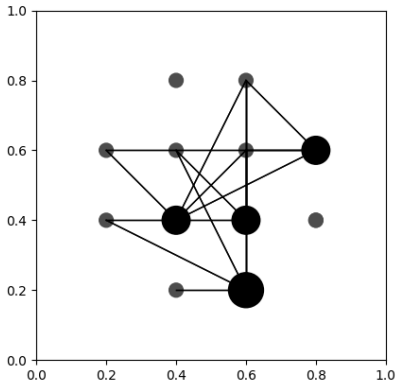

```
#Simplified Synchrony Maps
    # Creates coordinates
    coords = []
    for i in range(4):
        for j in range(4):
```

```
            coords.append([.2 * (i + 1), .2 * (5 - (j + 1))])
    coords.remove([.2, .2])
    coords.remove([.2, .8])
    coords.remove([.8, .2])
    coords.remove([.8, .8])

    k = max_val(cluster_coeff)

    # Assigns channels to their respective coordinates
    channel_coord = {}
    for i in range(len(channels)):
        channel_coord[channels[i]] = coords[i]

    # Generates template for map
    figure, axes = plt.subplots()
    axes.set_aspect(1)
    plt.xlim(0,1)
    plt.ylim(0,1)
    for channel in channel_coord:
        if cluster_coeff[channel] == 1:
            draw_circle =
                plt.Circle(tuple(channel_coord[channel]), 0.01 *
                (len(connections[channel])),
                            color=str(0))
        else:
            draw_circle =
                plt.Circle(tuple(channel_coord[channel]), 0.02,
                            color=str(0.3))

        axes.add_artist(draw_circle)
    axes.add_artist(draw_circle)
    for i in channels:
        for j in channels:
            if i != j:
                ch1 = data["ch" + str(i)]
                ch2 = data["ch" + str(j)]
                if (cluster_coeff[i] == 1 or cluster_coeff[j] ==
                    1) and synch(ch1, ch2, t) > threshold:
                    connectpoints(channel_coord[i],
                        channel_coord[j], 0)
    phat = os.path.join(simp_synch, file[:-4] +
        '_Simplified_Synchrony_Map')
    plt.savefig(phat)
    plt.close('all')


CC_channels = {'Phase': phases, 'Average Clustering
    Coefficient': clus_coeff, 'Number of Nodes': n_nodes}
for channel in channels:
    CC_channels[channel] = []
```

```
    for ph in sig_node_well:
        CC_channels[channel].append(ph[channel])
print(CC_channels)
```

This code creates a template for the synchrony maps and uses the Clustering Coefficient and Synchrony Index data to generate the maps. A **simplified** synchrony map plot only plots lines between connections from perfect nodes (nodes with a clustering coefficient of 1). The lines are of uniform thickness. All perfect nodes are black with a size proportional to the size of their network (refer to section 1.2). The non-perfect nodes are all grey and of uniform size.

Example:




```
CC_channels = {'Phase': phases, 'Average Clustering
    Coefficient': clus_coeff, 'Number of Nodes': n_nodes}
for channel in channels:
    CC_channels[channel] = []
    for ph in sig_node_well:
        CC_channels[channel].append(ph[channel])
print(CC_channels)

CC_data = pd.DataFrame.from_dict(CC_channels)
CC_data = CC_data.sort_values(by='Phase')
CC_data.to_csv(os.path.join(synch_cc_data, str(well) +
    '_Clustering Coefficient Data.csv'), index = False)
plt.subplot(1, 2, 1)
plt.plot(CC_data['Phase'].tolist(), CC_data['Average Clustering
    Coefficient'].tolist(),      label='Average Clustering
    Coefficient')
plt.title('Average Clustering Coefficient')
plt.subplot(1, 2, 2)
```

```
        plt.plot(CC_data['Phase'].tolist(), CC_data['Number of
            Nodes'].tolist(), label='Number of Significant Nodes')
        plt.title('Nodes')
        plt.savefig(os.path.join(synch_cc_data, str(well) + '_Clustering
            Coefficient Plots'))

        syn_data['Phases']=phases
        pd.DataFrame.from_dict(syn_data).to_csv(os.path.join(synch_cc_data,
            str(well) + '_Synchrony Data.csv'), index = False)
        plt.close('all')
```

Finally, all of the data is saved in the folders.

2.4 Independent Networks

```
    for well in os.listdir(dest_dir):
    if "Stim CSV" not in str(well): #ignores the directory with the Raw
         Data CSV Files
        print(well)
        dest_well = os.path.join(dest_dir, well)
        synch_cc_data = os.path.join(dest_well, 'Other Data')
        CDataWell = os.path.join(synch_cc_data, str(well) + '_Clustering
            Coefficient Data.csv')
        SDataWell = os.path.join(synch_cc_data, str(well) + '_Synchrony
            Data.csv')
        coefficient_data = pd.read_csv(CDataWell)
        synch_data = pd.read_csv(SDataWell)
```

The code above iterates through every well and reads the synchrony index data and clustering coefficient data.

```
        coeffic_data = coefficient_data.values.tolist()
        coefficient_data = []
        sig_nodes = []
        for i in coeffic_data:
            coefficient_data.append(i[4:])
            sig_nodes.append(i[3])
        s_data = synch_data.values.tolist()
        synch_data = []
        conn_data = []
        average = []
        total = []
        phases = []
        per_nodes = []
        ind_connections = []
        channels = ['ch12', 'ch13', 'ch21', 'ch22', 'ch23', 'ch24',
            'ch31', 'ch32', 'ch33', 'ch34', 'ch42', 'ch43']
        #iterates by phase (i is the phase number)
```

```
for i in range(len(s_data)):
    print("Phase:" + str(i))
    hub_nodes = {}
    d=s_data[i]
    #l is a list of dictionaries that shows the sychrony indexes
        between all channels. i.e. [{ch13: 0.1, ch21: .2, ...},
        {ch12:0.1, ch21:.3, ...}, ...]
    l = []
    for j in range(11):
        l.append(str_to_dict(d[j+2]))
    c = coefficient_data[i]
```

The code above takes the clustering coefficient and synchrony data and formats it.

```
    cluster_coeff = {}
    for j in range(len(c)):
        if c[j] == 1:
            hub_nodes[channels[j]] = l[j]
```

A list of all perfect nodes is generated.

```
    for x in range(len(c)):
        cluster_coeff[channels[x]] = c[x]
    threshold = 0
    for dic in l:
        threshold+= dic['avg']
    threshold = threshold/12
    for hub in hub_nodes:
        conn = []
        for chan in hub_nodes[hub]:
            if hub_nodes[hub][chan] >= threshold:
                conn.append(chan)
        if 'avg' in conn:
            conn.remove('avg')
        hub_nodes[hub] = conn
```

The connections of all the perfect nodes are calculated and stored.

```
    hubs = list(hub_nodes.keys())
    print("Per Nodes: " + str(len(hubs)))
    print(hubs)
    if len(hubs) > 1:
        networks = [hubs[0]]
        for hub in hubs[1:]:
            new = True
            for h in networks:
                if hub not in hub_nodes[h]:
                    new = False
```

```
                if new:
                    networks.append(hub)
            ind_con = len(networks)
        elif len(hubs) == 1:
            networks = [hubs[0]]
            ind_con = 1
        else:
            networks = []
            ind_con = 0
```

The number of independent networks are calculated. If a phase has no perfect nodes, we say that it has no networks.

```
        print(networks)
        print("Ind Con: " + str(ind_con))

        ind_connections.append(ind_con)
        per_nodes.append(len(hubs))
        phases.append(i)

    pd.DataFrame({'Phases':phases, 'Independent
        Networks':ind_connections, 'Perfect
        Nodes':per_nodes}).to_csv(os.path.join(synch_cc_data,
        str(well) + '_Independent_Networks.csv'))
```

The number of perfect nodes in the phase and the number of Independent Networks is saved in the "Other Data" folder.

```
for well in os.listdir(r'G:\Shared drives\Lybrand MEA Drive\NEW Blast
    Analysis'):
    if not str(well).startswith("Well_9"):
        data_path = os.path.join(os.path.join(r'G:\Shared drives\Lybrand
            MEA Drive\NEW Blast Analysis', well), 'Other Data')
        CDataWell = os.path.join(data_path,str(well) + '_Clustering
            Coefficient Data.csv')
        SDataWell = os.path.join(data_path,str(well) + '_Synchrony
            Data.csv')
        coefficient_data = pd.read_csv(CDataWell)
        synch_data = pd.read_csv(SDataWell)
        coeffic_data = coefficient_data.values.tolist()
        coefficient_data = []
        sig_nodes = []
        X = []
        Y = []
        X2 = []
        A = []
        channels = ['ch12', 'ch13', 'ch21', 'ch22', 'ch23', 'ch24',
            'ch31', 'ch32', 'ch33', 'ch34', 'ch42', 'ch43']
        for i in coeffic_data:
```

```
    coefficient_data.append(i[4:])
    sig_nodes.append(i[3])
s_data = synch_data.values.tolist()
for i in range(len(s_data)):
    hub_nodes = {}
    d = s_data[i]
    # l is a list of dictionaries that shows the sychrony indexes
        between all channels. i.e. [{ch13: 0.1, ch21: .2, ...},
        {ch12:0.1, ch21:.3, ...}, ...]
    l = [str_to_dict(d[1]), str_to_dict(d[2]), str_to_dict(d[3]),
        str_to_dict(d[4]), str_to_dict(d[5]), str_to_dict(d[6]),
        str_to_dict(d[7]), str_to_dict(d[8]), str_to_dict(d[9]),
            str_to_dict(d[10]), str_to_dict(d[11]),
        str_to_dict(d[12])]
    c = coefficient_data[i]
    cluster_coeff = {}
    for j in range(len(c)):
        if c[j] == 1:
            hub_nodes[channels[j]] = l[j]
    for x in range(len(c)):
        cluster_coeff[channels[x]] = c[x]
    threshold = 0
    for dic in l:
        threshold += dic['avg']
    threshold = threshold / 12
    for hub in hub_nodes:
        conn = []
        for chan in hub_nodes[hub]:
            if hub_nodes[hub][chan] >= threshold:
                conn.append(chan)
        if 'avg' in conn:
            conn.remove('avg')
        hub_nodes[hub] = conn
    total_conn = 0
    num_nod = 0
    for hub in hub_nodes:
        total_conn += len(hub_nodes[hub])
        num_nod += 1
    for hub in hub_nodes:
        X.append(i)
        Y.append(len(hub_nodes[hub]))
    if num_nod != 0:
        X2.append(i)
        A.append(total_conn/num_nod)
plt.title(well + " Perfect Nodes Scatter Plot")
plt.ylabel('# Of Connections')
plt.xlabel("Phase")
plt.ylim(0,13)
plt.xticks(np.arange(0,49,step=2))
plt.scatter(X,Y)
```

```
plt.scatter(X2,A, marker = 's', alpha = 0.3, c = 'red')
save_path = os.path.join(data_path,str(well) + '_Clustering
    Scatter.png')
plt.savefig(save_path)
plt.close()
```

Based on the data extracted from the previous analysis, this code generates scatterplots of each perfect node's size.