Quantitative Analysis and Visualization of Signaling Networks

Hands-on Activities

The following exercises will use sample datasets we have collected by live cell microscopy.

data1.mat This dataset was obtained from cells expressing a FRET-based ERK reporter (EKAR-EV) and a cell cycle reporter, Geminin-mCherry (a modification of the Fucci cell cycle reporter), which begins to increase in signal at the onset of S-phase, and accumulates until the cell completes mitosis. The dataset contains 2 channels, 'fret' and 'rfp'. The FRET channel was calculated as CFP/YFP, so a decrease in the signal represents an increase in ERK activity. The cells in this dataset were edited so that each track begins immediately following the previous mitosis.

data2.mat This dataset was obtained from cells expressing an AMPK FRET reporter and an mCherry translocation-based Akt reporter. The dataset contains 6 channels – CFP, YFP, and RFP for both nucleus and cytosol – and a ratio of the CFP and YFP cytosolic channels (cfpCyt/yfpCyt), which indicates AMPK activity. The Akt reporter translocates from the nucleus to the cytosol when phosphorylated, so rfpCyt/rfpNuc can be used as a readout of Akt activity.

data3.mat This dataset was obtained from cells expressing an ERK FRET reporter and the mCherry Akt reporter mentioned above. The channels are the same as in data2.mat.

1. Visualization

In this section we will cover a few of the basics for how to generate data visualizations in MATLAB. We will also use some custom scripts we have written to make plotting this particular type of data faster and easier.
First, we will practice with some basic MATLAB plotting functions that are useful.

Exercise 1: Making a heatmap
Step 1.1: Load data2.mat by clicking on it in the Matlab file browser, or by using the load command at the prompt. Confirm that you have loaded the variables by looking at the workspace pane.

Step 1.2: Create a heatmap visualization of one of the variables by typing
`imagesc(variable)`.

Step 1.3: Experiment with different colormaps for your heatmap. Go to Edit>Colormap. In the pop-up Colormap Editor window, under Tools>Standard Colormaps, you will find various choices. 'Jet' is one of the most popular, and was the MATLAB default for many years. 'Parula' is the new MATLAB default, and has some nice advantages (for example, it still reads correctly if printed in black and white; Jet does not). Experiment with moving the color sliders, changing the colors on sliders (try double clicking on them), or choosing different Color Data min/max values. Can you improve the clarity of the data display?

Step 1.4: Use the zoom button (magnifying glass with a '+' sign) to look at portions of your heatmap in detail. Find a region where you can clearly see the "bands" of reporter activity in a few cells. Note that double-clicking will zoom you back out to the original scale. If you want to manually define the specific region you want to focus on, you can go to Tools>Edit Plot. After selecting this option, double click on the numbers on the x or y axis of the plot. This will open the plot editor window, where you can manually enter the limits of the x and y axes. Note that you can also specify ranges when you make the original plot command. For example:
`imagesc(fret(40:60, 160:320))` will plot rows 40 through 60, and columns 160 through 320.

Exercise 2: Making line plots
Step 2.1: Create a simple line plot by typing
```
figure,plot(fret(46,:))
```

This will plot the 46th row of your data matrix at all time points. (You could choose any row, but this one has a good signal for this example.) Note that ':' is MATLAB shorthand for "the entire range". Also, note that typing 'figure' first creates a new plot window. If you don't type this, MATLAB will just use the last figure window you have open, and overwrite your last plot.

Step 2.2: Note that the data for the cell you have plotted drop to zero for the first 20 time points or so. This is because there were no data points for the cell at these times, because it hadn't been born (mitosed?) yet. Because this throws off the scale of the plot, try
`figure,plot(fret(46,21:end))`to just plot the positive values.

Step 2.3: Experiment with changing the line style. You can do this by selecting Tools>Edit Plot and then double clicking on the plotted data. In the editor window you will be able to change the line color, width, and style, and add markers for each data point if you wish. You can also make these choices at the command line when you make the plot. For example:
```
figure,plot(fret(46,21:end),'r.-')
```
The 'r' specifies red (other choices: b,g,y,k,m,c) and the '.-' specifies a line with small dot markers.
```
figure,plot(fret(46,21:end),'go')
```
This should give you green circle markers with no line. Not the best choice in this case, but useful sometimes.

Step 2.4: Plot both of the channels from the same cell so we can see them together:
```
figure,subplot(2,1,1),plot(fret(46,21:end))
subplot(2,1,2),plot(rfp(46,21:end),'r')
```

Exercise 3: Using the custom plotting scripts
Step 3.1: The custom plotting scripts are all run using the "ct_trackvis" command.
```
ct_trackvis('heatmap',fret)
```
This will make a heatmap with a grayscale colormap. You can use other colormaps like this:
```
ct_trackvis('heatmap',fret,'cmap','parula')
```

Step 3.2: Use the same command to plot the mean:
```
ct_trackvis('mean',fret)
```
Note that the mean looks like it has some problems at the start of the movie. This is because there are a lot of missing values early on, and they are being included in the mean. Fortunately this script has a convenient way to deal with this. We just have to indicate that the zeros represent missing values:
```
ct_trackvis('mean',fret,'missing','zero')
```

Step 3.3: Several other plot formats are available. Try using the following
```
ct_trackvis('ensemble',fret,'missing','zero')
ct_trackvis('histogram',fret,'missing','zero')
ct_trackvis('triple',fret,'missing','zero')
```

2. Noise

In this section we will use several techniques for reducing noise and filtering out invalid points from your data set.

Exercise 4:
Step 4.1: Load data1.mat and view the rfp data.
```
ct_trackvis('triple',rfp,'missing','zero');
```
Note that there are some outliers showing up as spurious peaks downward, as well as points where all tracks shift. We provide the function `ct_outlier` to identify and remove these based on a threshold values, a maximum derivative or from manually specified time points. For continuity, the points removed are replaced by values interpolated from nearby points.

Step 4.2: Remove outliers by setting a lower threshold value, a maximum derivative, and manually specifying a time point.

```
rfp_o = ct_outlier(rfp, [200, Inf], 10, 247);
```

View the results. Note the choices in selecting these thresholds; all are chosen by observing this particular data. The derivative value is chosen to be larger than any reasonable change across one time point (note the y-axis scale).

```
ct_trackvis('triple', rfp_o, 'missing', 'zero');
```

Step 4.3: Smooth high frequency noise by applying a low-pass filter. We provide the function `ct_filter`, which employs MATLAB's functions `butter` and `filtfilt` to create a filter with minimal effects on the frequency content you want to keep and filter your data. Providing `ct_filter` with the flag `'low'` tells it to use a low-pass filter. The next input (set to 5 here) indicates the period of peaks/oscillations you want to eliminate, in number of time points. The final input is optional and indicates that missing data values are set to `0` in these data.

```
rfp_f = ct_filter(rfp_o, 'low', 5, 0);
```

View the results:

```
ct_trackvis('triple', rfp_f, 'missing', 'zero');
```

Note in the `rfp` data that there are also several points where the value of all tracks appears to shift (examine time range 100 to 200). These may be correlated disturbances during the imaging process, with many possible sources, including: changes in the light source/path, small positional shift of the sample on the stage or filters in their housings, or the presence of additional light for a time. Such disturbances and particularly difficult to correct, e.g. by manually identifying the affected region and adding a fixed value. It is only recommended to correct these disturbances when they are severe and the data is otherwise quite tractable.

Step 4.4: Remove drift in the baseline (low frequency noise). The `fret` data, in principle, should be bounded and have no long term trends, but this example drifts, likely due to photobleaching of the YFP acceptor. The `ct_filter` function can be used to remove this, making further analysis easier. First view the fret data (after removing an inconvenient outlier).

```
fret_o = ct_outlier(fret, [0.3, Inf]);
ct_trackvis('triple', fret_o, 'missing', 'zero');
```

Next, remove the drift with period of 200 time points or larger, by specifying `'high'`, meaning a high-pass filter. Note that this will remove the mean value and center the data on zero; `ct_filter` automatically adds back the mean value near the start of each track.

```
fret_h = ct_filter(fret_o, 'high', 200, 0);
ct_trackvis('triple', fret_h, 'missing', 'zero');
```

These data look better overall, however, because the mean value was removed all parts of the track are now centered. This means that near peaks the values will be depressed, putting the mean of the peaks near the mean of the baseline. Depending on your downstream usage, this may be undesirable. This can better be observed on a single track.

```
figure;  plot(fret_o(1,:));  hold on;  plot(fret_h(1,:), 'r:');
```

Instead, correct drift by removing the trending baseline, by finding the local minimum value within a window of each point. This method is also implemented in `ct_filter`, under the flag `'base'`, here indicating the size of the window to consider local, `50` time points. This window should be significantly larger than any features you want unaffected.

```
fret_b = ct_filter(fret_o, 'base', 50, 0);
```

View the results alongside the previous example track, and on the whole dataset.

```
plot(fret_b(1,:), 'k-');
ct_trackvis('triple', fret_b, 'missing', 'zero', 'cmap', 'orange');
```

3. Features

In this section we will apply methods to identify pulses of activity within the data and to quantitatively characterize the shape of activity features.

Exercise 5: Evaluating derivatives for rate analysis

Step 5.1: Using the rfp data, filter heavily (low-pass) to leave only the trend. It is important to consider carefully the scale of the changes you want to evaluate. When evaluating derivatives of shorter, faster events, choose a shorter cutoff period, ideally well shorter than the target feature.

```
rfp_f = ct_filter(rfp_o, 'low', 50, 0);
ct_trackvis('triple', rfp_f, 'missing', 'zero');
```

Step 5.2: Evaluate the derivative of the smoothed data.

```
drfp_f = diff(rfp_f,1,2);
ct_trackvis('triple', drfp_f, 'missing', 'zero');
```

Exercise 6: Evaluating oscillatory behavior

Step 6.1: Load data2.mat, which contains some apparently oscillatory signals. Take the nuclear to cytoplasm ratio of the rfp channel to get the FOXO signal. Select an example with clear oscillations and one appearing flat. Plot each.

```
osc = rfpNuc(24,:)./rfpCyt(24,:);
flat = rfpNuc(1,:)./rfpCyt(1,:);
figure; plot(osc); hold on; plot(flat, 'r-');
```

Step 6.2: Using MATLAB's `pwelch` function, view the power spectrum of each. Because `pwelch` does not handle `NaN`s, they are excluded when passing data to the function. The `set` and `axis` commands below are to modify the plot axes to use a log scale and for better visibility (`gca` refers to the currently selected figure axes, those just created).

```
figure; subplot(2,1,1); pwelch( osc(~isnan(osc)) );
set(gca, 'XScale', 'log'); axis tight;
subplot(2,1,2); pwelch( flat(~isnan(flat)) );
set(gca, 'XScale', 'log');axis tight;
```

Note the difference between the power spectra (the hump at $2*10^{-1}$), indicating the presence of oscillations in the first, but not second example. The peak of the hump indicates the mean frequency of the oscillation, and that the hump is wide (not a sharp peak) indicates that the oscillation is not a pure sine wave, but contains multiple frequencies.

Step 6.3: Consider an example from data1.mat with peaks, but not a truly oscillatory signal. Repeat the same procedure.

```
nosc = ct_filter(fret(12,:), 'base', 50, 0);
flat = ct_filter(fret(2,:), 'base', 50, 0);
figure; plot(nosc); hold on; plot(flat, 'r-');
figure; subplot(2,1,1); pwelch(nosc(~isnan(nosc)));
set(gca, 'XScale', 'log'); axis tight;
subplot(2,1,2); pwelch(flat(~isnan(flat)));
set(gca, 'XScale', 'log');axis tight;
```

Note here that there is no notable difference between the two power spectra, no hump to indicate oscillations. Inconsistent peaks in the signal do not create a power spectrum peak, but rather have a distributed effect over the spectrum. Such signals are better analyzed by explicitly evaluating peaks.

Exercise 7: Identifying peaks

Step 7.1: Find peaks in clean data using MATLAB's `findpeaks` function. This function takes the derivative of the data and finds all the points where the derivative changes from positive to negative.

```
[pv, loc] = findpeaks(osc);
figure; plot(osc); hold on; plot(loc, osc(loc), 'rv');
```

Note that there are a few 'peaks' found that are very small and originate from noise. If the signal is mostly clean and the noise is much lower amplitude than the peaks of interest, these peaks can be filtered out based on their height compared to local minima, or in recent versions of MATLAB, via a metric called prominence. Prominence describes the height over the landscape between the peak and any taller peaks.

```
[pv, loc] = findpeaks(osc, 'MinPeakProminence', 0.03, 'Annotate', 'extents');
```

Exercise 8: Finding peaks in noisy or complex data

For finding peaks in noisy data, we provide a function, `ct_getpeaks`, which identifies peaks and valleys via the derivative and compares the rises/falls between them against a pair of thresholds to determine if they are significant enough to consider. The thresholds are set based on the local minimum and maximum of the signal over a window, which may be provided as an input. The window should be well longer than features you wish to detect, but short enough to allow adaptation to changing baselines. We recommend starting with approximately 1.5 times the longest peak you expect.

Step 8.1: Find peaks and plot locations. Note that the output of `ct_getpeaks` includes 2 columns, the start of a peak region and end, similarly for valleys.

```
[pk, vy] = ct_getpeaks(nosc, 25);
figure; plot(nosc); hold on; plot(pk(:,1), nosc(pk(:,1)), 'rv');
```

Note that a variety of peaks are reported that are small and appear to arise largely from noise. It can be useful to smooth the data prior to identifying peaks. This can be done beforehand, or within `ct_getpeaks` via an optional parameter (`smooth`, which ultimately uses `ct_filter`).

Step 8.2: Find peaks after first filtering the data.

```
nosc = ct_filter(nosc, 'low', 5);
[pk, vy] = ct_getpeaks(nosc, 25);
figure; plot(nosc); hold on; plot(pk(:,1), nosc(pk(:,1)), 'rv');
```

Step 8.3: Filter peaks by setting a minimum rise requirement.

```
[pk, vy] = ct_getpeaks(nosc, 25, 'min_rise', 0.04);
figure; plot(nosc); hold on; plot(pk(:,1), nosc(pk(:,1)), 'rv');
```

Exercise 9: Characterizing peaks

To more carefully evaluate pulsatile data, characteristics of peaks can be examined and compared, including amplitude, duration, and rise and fall times. We provide the function `ct_pulseanalysis` to evaluate these parameters for each peak in a dataset (peaks are found via `ct_getpeaks`). The procedure looks within a window to either side of an identified peak region for apparently baseline regions, based on the derivatives and local values. Then the rises and falls are defined as the regions between 10% and 90% of the baseline to peak height, on either side of the peak. The metrics are then calculated from these regions and their spacing.

Step 9.1: Perform analysis on the noisy data, filtering peaks by a minimum rise (the parameter is passed along to `ct_getpeaks`), and defining a local window within which to seek baseline regions near peaks, via the `maxw` parameter. `ct_pulseanalysis` returns a structure containing the metrics within named fields. Use a provided function, `ct_viewpeaks`, for convenience in plotting the peak locations, rise and fall regions, and mean amplitudes.

```
z = ct_pulseanalysis(nosc, 'min_rise', 0.04, 'maxw', 30);
ct_viewpeaks(nosc,z);
```

Note on fitting signal and features to functions

Where applicable, particular feature or whole signals may be used to fit functions (i.e. curve fitting). For example, the decay from a well sampled peak could be fit to exponential functions to evaluate if the process follow simple first order kinetics or appears more complex. Curve fitting extends directly to model-based analysis, by fitting a custom function (the model) to the signal or specific features. Basic curve fitting can be performed using MATLAB's `fit` function. Fitting custom models may be approached by defining (simpler) models with `fittype` for use with the `fit` function, or employing MATLAB's general optimization functions, e.g. `fmincon`.

4. Trends and Statistics

Exercise 10: Evaluating features over time

Having extracted and characterized features across dynamic data tracks, it can be useful to evaluate how these feature change over time. One means of doing this for sparse features, such as peaks, is to evaluate them within windows of time and compare.

Step 10.1: Load, filter, analyze peaks and view a sample signal

```
load('data1.mat');  sig = fret(48,:);  sig = ct_filter(sig, 'base', 50, 0);
z = ct_pulseanalysis(sig, 'narm', 20, 'smooth', 3, 'maxw', 30);
ct_viewpeaks(sig,z);
```

Step 10.2: Create a simple windowing of the signal, and determine the number of peaks per window. Plot the number of peaks as a function of window number.

```
nt = length(sig);  wsz = 100;  nt = floor(nt/wsz)*wsz;
twin = [(1:wsz:nt)',(wsz:wsz:nt)'];  nw = size(twin,1); np_win = zeros(1,nw);
for s = 1:nw;
    np_win(s) = nnz(z.mpos > twin(s,1) & z.mpos < twin(s,2));
end
figure; plot(np_win);
```

Step 10.3: For better resolution, use a moving window evaluating the number of peaks in the window around each point (truncated at the edges). To do this simply, we will use the MATLAB function `smooth`, which calculates moving averages. By multiplying by the window size again, the result is the total number rather than the average (with truncated window sizes at the edges).

```
fvec = zeros(1,length(sig));  fvec(floor(z.mpos)) = 1;
wsz_vec = [1:wsz/2,wsz*ones(1,length(sig)-wsz),wsz/2:-1:1];
np_win = smooth(fvec,wsz)'.*wsz_vec;
figure; plot(np_win);
```

Step 10.4: To observe how the mean value changes, filter all traces to remove baseline and get the windowed mean over all traces, and plot. This uses the MATLAB function `nanmean` to calculate means, which ignores NaNs.

```
fret_b = ct_filter(fret,'base',50,0);
figure; plot(smooth(nanmean(fret_b,1), wsz, 'moving'));
```

Step 10.5: To examine pulse features, run the analysis over the whole dataset.

```
z = ct_pulseanalysis(fret_b, 'narm', 20, 'smooth', 3, 'maxw', 30);
```

Step 10.6: Show mean number of pulses per track. This is done in a loop over all tracks, by making a matrix of zeros and setting only the time locations of peaks to 1. The mean over tracks is then evaluated, neglecting missing data (NaNs) and effectively counting the peaks, followed by the sum over the windows.

```
[nC, nT] = size(fret); d = zeros(nC, nT);  d(isnan(fret_b)) = nan;
for s = 1:nC;  d(s, floor(z(s).mpos)) = 1; end
dwin = nanmean(d,1); dwin = smooth(dwin, wsz, 'moving').*wsz_vec';
figure; plot(dwin);
```

Step 10.7: Additionally, evaluate the standard error of the mean (SEM) number of pulses and plot the +/- 1 SEM interval. The procedure is as with means, but calculating variance with `nanvar`.

```
nvt = mean(~isnan(fret_b),1);
swin = nanvar(d,[],1)./nvt;  swin = sqrt( smooth(swin, wsz, 'moving') );
hold on; plot(dwin-swin, 'r--'); plot(dwin+swin, 'r--');
```

Step 10.8: Show the mean pulse amplitude over time. This time, the initial matrix is filled with NaNs to avoid dividing by the number of tracks (i.e. diluted with all the zeros from tracks not peaking at that time point). Instead, we report the mean of the peaks that did occur at a time point. Note that this is the mean over the time window, rather than the sum, as when counting peaks.

```
d = nan(nC, nT);
for s = 1:nC; d(s, floor(z(s).mpos)) = z(s).amp_mean; end
dwin = nanmean(d,1); dwin = smooth(dwin, wsz, 'moving'); figure; plot(dwin);
```

__Exercise 11: Making comparisons with scalar data__

In this exercise, we would like to ask the question of whether there is a statistically significant difference in the level of ERK activity between cells in G0/G1 and cells in S/G2/M. Because we have data for the ERK reporter and the Geminin reporter (which is low during G0/G1, and high during S/G2/M), we have all the information necessary to test this question. As an initial approach to this question, we can simply split all of the data points into either G0/G1 or S/G2/M classes.

Step 11.1: First, we need to define an appropriate threshold. To get a sense of this, plot all of the cell cycle reporter data:

```
ct_trackvis('ensemble', rfp, 'missing', 'zero')
```

From this plot, it is fairly clear that cells at the start of the cell cycle (where each track starts) have a fluorescence value of approximately 210, with a spread between 209 and 211. Upon entering S-phase, we can see that they rise to values of 220 or greater. For the purpose of discriminating pre-S vs. post-S cells

Step 11.2: To mark all of the data points where the rfp value exceeds 212, use the following command:

```
gpos=rfp>212;
```

This creates a new variable, gpos (you can call it whatever you like) that is an array of 1s and 0s the same size as the rfp array. 1s indicate positions where rfp>212, 0s indicate where rfp is less than or equal to 212.

Step 11.3: Now we can divide the FRET data into groups based on the cell cycle status of the cell.

```
fp=fret(gpos);
fn=fret(~gpos);
```

This creates two sets of data. fn has all the FRET measurements from the Geminin-positive cells, fn has all the FRET measurements from the Geminin-negative cells. We still have the problem that any missing data values, which have a FRET measurement of 0, are included in these sets. We can deal with this by simply redefining fn and fp to include only values that are above 0:

```
fp=fp(fp>0;
fn=fn(fn>0);
```

Step 11.4: Now we are ready to compare the medians:

```
median(fp)
median(fn)
```

They should come out to be different, but is this a statistically significant difference? Before performing any test it is usually a good idea to look at the distributions of the samples:

```
figure, hist(fp)
figure, hist(fn)
```

Both samples appear to be approximately normally distributed, with approximately equal variances, so an unpaired T-test is appropriate:

```
[h,p]=ttest2(fp,fn)
```

This should come out statistically significant. In fact the p-value shows up as zero, because we are using so many measurements. But there are some problems in the way we’ve done this. We’ve considered all the time points as though they are independent measurements, which is not correct. We also haven't accounted for drift. Try running this same analysis, but using the drift-corrected data calculated above, or using only a single time point from the data where there are both Geminin-positive and Geminin-negative samples present. If you are advanced at MATLAB or up for a challenge, try writing a script to identify the point in each cell track where S-phase begins (i.e., Geminin first rises over a threshold), and compare FRET measurements before and after this point within the same cell.

Exercise 12: Making comparisons with dynamic data

Step 12.1: Load data2 and take the nuclear to cytoplasm ratio to get the FOXO signal. We will be using MATLAB's `corr` function to evaluate Pearson's linear correlation coefficient. Because `corr` does not handle NaNs, we will record which entries are NaN, and exclude them from the analysis.

```
load('data2.mat');  foxo = rfpCyt./rfpNuc;
nnd = ~isnan(CYratioCyt) & ~isnan(foxo);
```

Step 12.2: Calculate the correlation coefficient between each pair of AMPKAR and FOXO signals, in a loop, because each signal may have a different distributions of NaNs that need to be neglected.

```
[nC, nT] = size(CYratioCyt);  c = zeros(1,nC); p = c;
for s = 1:nC; if ~any(nnd(s,:)); continue; end
    [c(s), p(s)] = corr(CYratioCyt(s, nnd(s,:))', foxo(s, nnd(s,:))');
end
```

Step 12.3: Plot a histogram of the correlation coefficients, to evaluate the distribution over individual cells.

```
figure; hist(c);
```

Step 12.4: Repeat the correlation procedure, using data3.mat to evaluate EKAR against FOXO.

Step 12.5: Evaluate the alignment of EKAR and FOXO signals, by calculating the cross-correlation with MATLAB's `xcorr` function. Similar to `corr`, it requires elimination of NaNs and must be run in a loop. Cross-correlation is calculated by shifting one signal in time and taking the product of the two signals at each time-shifted position with some normalizing. The resulting vector of cross-correlation values will peak at the shift, or lag, that optimally lines up the two signals. Delay between two processes may be evaluated in this way. This loop also calculates the peak lag value for each signal pair.

```
c = cell(nC,1); lag = c; mx_lag = zeros(nC,1);
for s = 1:nC; if ~any(nnd(s,:)); continue; end
    [c{s}, lag{s}] = xcorr(CYratioCyt(s, nnd(s,:))', ...
        foxo(s, nnd(s,:))', 'coeff');
    [~,mxi] =  max(c{s});    mx_lag(s) = lag{s}(mxi);
end
```

Step 12.6: Plot the cross-correlation vector against the corresponding lag values, for one signal pair. Then plot a histogram of the peak lag values.

```
figure; plot(lag{1}, c{1});
figure; hist(mx_lag);
```

Note that these signals are very well correlated, with no optimal lag values other than zero.

5. Exporting publication quality images

Option 1: If you want to make figures in Microsoft PowerPoint

Step 1: Choose one of the figures created above, or make a new one. In the figure window, go to File>Save As. In the 'Save as type' drop down menu, choose Enhanced metafile (*.efm). Choose a file name and hit Save.

Step 2: If you have Power Point, open it and go to Insert>Pictures. It is possible to right-click on the image, and choose 'Edit Picture', and say 'Yes' when it asks you if you want to convert it to a Microsoft Office drawing object. This will allow you to edit lines and text within the figure. However, it makes heatmaps blurry. A more advanced option for moving figures to PowerPoint is this MATLAB custom package:

http://www.mathworks.com/matlabcentral/fileexchange/30124-smart-powerpoint-exporter

Option 2: If you want to make figures in Adobe Illustrator

Step 1: Choose one of the figures created above, or make a new one. In the figure window, go to File>Save As. In the 'Save as type' drop down menu, choose PDF file (*.pdf). Choose a file name and hit Save.

Step 2: Open the .pdf file in Illustrator. The text and lines will be editable; experiment with changing the font size and line widths. There will be a number of white background squares that it is usually easiest to delete.