A Tutorial on PCA Interpretation using CompClust

Joe Roden, Diane Trout & Brandon King

Version 1.2
December 13, 2005

Contents

1 Introduction

1.1 Purpose

This tutorial is designed to introduce software developers to the PCAGinzu modules contained in the CompClust Python package. The CompClust tools for comparative cluster analysis are described in (Hart et al., 2005). See Section 9 for additional information on CompClust as well as the publication describing our PCA interpretation approach. After following this tutorial a software developer aught to be able to load his own gene expression microarray datasets and labelings, perform PCA on the data, and generate interpretations of the PCA results as described in (Roden et al., 2005).

The approach taken in this tutorial is to guide a software developer to become familiar with the CompClust package's application programming interface (API) so that he or she may proceed to use the API to perform PCA interpretation with their own datasets.

1.2 Prerequisites

Software developers interested in using the PCAGinzu components to interpret PCA results should be familiar with Python. For a tutorial on Python, see http://docs.python.org/tut/tut.html.

Users may also want to be familiar with key CompClust concepts, in particular, Datasets, Labelings, and Views. This tutorial will attempt to describe most of these concepts as they are introduced. If you'd like to learn more there are additional tutorials you might consider following. For additional tutorial references, including CompClust, see Section 9.

If you intend to use the CompClust software to perform the commands while you read the tutorial, we expect that you will be using either of the following versions of CompClust:

CompClustShell for Windows This is a convienent Windows package which contains all of the software packages needed to begin to explore CompClust, including the PCA interpretation software. It is a Windows executable that provides a Python shell with CompClust and dependent packages pre-loaded.

CompClust Python Package Software developers can download and install the complete CompClust source code distribution which contains the CompClust programming interface in Python. Note that the CompClust source installation requires additional Python packages, and so the installation can be involved.

If you're installing from source and have your own Python, we recommend the http://ipython.scipy.org/ enhanced interactive Python shell. Some of our examples take advantage of IPython's features.

1.3 CompClust Background

CompClust provides software tools to explore and quantify relationships between clustering results. Its development has been largely built around requirements for microarray data analysis, but can be easily used for other types of biological array data and that of other scientific domains.

1.4 Principal Components Analysis (PCA) Background

Principal Components Analysis (PCA) is a numerical procedure for analyzing the sources of variation present in a multi-dimensional dataset. We employ it to analyze gene expression microarray datasets, but the concept is general to any multidimensional dataset worth analyzing.

We carry out PCA by applying singular value decomposition (SVD) to the covariance matrix of D, $cov(D)$, to produce the decomposition that contains the eigenvectors of $cov(D)$ in the columns of U and eigenvalues in the diagonal of S such that the eigenvalues are sorted by descending size.

Each covariance eigenvector, or principal component, explains a fraction of the total variance contained in the dataset, and each principal component P_{n+1} is orthogonal to the previous principal component P_n. such that they define the basis of a new vector space P.

1.5 Interpreting Principal Components

It is our belief that PCA is one of a set of tools that can help investigators better understand the sources of variation present in a microarray dataset. Each principal component measures, and to some degree models, some source of variance observed in the dataset. At the simplest level one observes the variance explained by each principal component, and perhaps takes note of that principal component's eigen vector. By analyzing each principal component a bit further we can better appreciate what is driving that component's variation. Our strategy is to identify the data points (genes) at the extremes of each principal component axis, and then determine which conditions are driving those outlier genes to be significantly differentially expressed. Beyond that, the covariate factors that correlate well with the significant conditions can be identified, so we may hypothesize that they are substantial sources of gene expression variation.

2 Data Preparation

2.1 Import the necessary software

The first step is to start a Python shell and within it, import the necessary Python packages. If one is using the windows compclust-shell program, this step has already been done for you; however it is perfectly acceptable to re-import the same modules.

```
## compClust modules
from compClust.mlx import datasets
from compClust.mlx import labelings
from compClust.mlx import views
from compClust.mlx import wrapper
from compClust.mlx import pcaGinzu
from compClust.mlx import pcaGinzuScoring
from compClust.iplot import IPlotTk as IPlot
from compClust.util import LoadExample
from compClust.config import config

## Useful standard python modules
import os
```

If you encounter an error executing these commands, your Python environment may not have access to all of the necessary Python packages. In CompClustShell this should not happen. Using a normal Python shell, it's possible that you have not properly set your Python path shell varible PYTHON-PATH properly, or that you have not installed all of the Python extensions on which CompClust depends.

2.2 Load a dataset

For this tutorial will be starting with the Cho Yeast cell cycling data set ((Cho et al., 1998)) because it is fairly small and easy to work with.

We have added new utility functions to make it very easy to load a number of example datasets into CompClust. These example loading functions download the original data and annotation files from remote web locations, load them into memory, attach row and column labels (e.g. gene names, condition covariates, etc.), and return a Dataset object that can be used for subsequent analyses. The example loading functions save a local copy of the resulting dataset in your home directory to speed subsequent dataset loading.

We can use these one of these simple dataset loading commands to get the Cho example data into memory as a Dataset object:

```
cho = LoadExample.LoadCho()
```

To keep dataset loading easy for now, we recommend simply executing the above commands and then skipping to Section 3.

If you are interested in understanding how to load your own dataset, the following sections describe in more detail how to manually load the Cho example dataset and associated labelings from tab-delimited text files. Alternatively, you can study the LoadCho function within the compClust.util.LoadExample.py module, or refer to the other CompClust tutorials described in Section 9 for additional examples and details.

2.2.1 Loading a dataset from text files

The first step of loading a dataset from a text file requires that we create a Dataset class object, using the Dataset constructor. The constructor can create a Dataset given a wide range of arguments, e.g. as strings (which are interpreted as file names), list of lists, and numeric arrays. In this tutorial we'll only demonstrate how to construct a Dataset from a text file as that is the most common usage pattern.

Dataset objects can be constructed from text files that are formatted as follows. A data file should have one line (row) per observation (e.g. a gene), with its column values (e.g. the conditions) separated by a delimiter such as the tab or comma character. The tab delimiter is default, but you can specify an alternative delimiter by passing an extra argument e.g. *delimiter=','* to the constructor. If the first column's data values are not numeric (e.g. if they are gene names), that situation will be detected and the entire first column will be ignored. All lines in the file are read as data unless a line begins with the Python comment character '#'. We are working to create a "smart" data file loader that will handle a wider range of input file formats.

When loading a dataset and its annotations from text files, there are two approaches for specifying the location of data files to load. You can change your current working directory to the directory that contains your dataset files, and then simply reference each file by its short file name (Option 1). Or you can construct the complete path name for each of the example files that you will be loading (Option 2). Either way, you have to know where your data files are prior to loading the data.

Where is your data?

If you are using CompClustShell for Windows (and you used the default installation options), the Cho example dataset files should be located in the directory:

```
  C:\Program Files\CompClustShell\Examples\ChoCellCycling
```

If you are using the CompClust source distribution instead, the example dataset can be found within the CompClust source code in the directory (shown here using UNIX path separators):

```
  compClust/gui/Examples/ChoCellCycling
```

For convenience, especially if you wish to specify full path names when you load your files, you can define a variable that holds the data root directory explicitly, e.g. in Windows:

```
dataroot = 'C:\Program Files\CompClustShell\Examples\ChoCellCycling'
```

Or in a Unix-based operating system the path might be, for example:

```
dataroot = '/User/sam/compClust/CompClust/gui/examples/ChoCellCycling'
```

In CompClust, we store the Cho example data directory location in a configuration variable so we have a sure-fire way to know where our example data is, no matter which operating system you have:

```
dataroot = config.cho_data_dir
```

Once you know where your data is, you can either change to that directory and load with short file paths, or include data directory in the full file path.

Option 1: Change directories, then load files

Changing directories is very easy in CompClustShell and IPython (on which CompClustShell is based). You can use a simple "cd" command, e.g.:

```
cd Examples/ChoCellCycling
```

If you're using a basic Python shell, giving explicit examples for changing directories can be complicated because the exact form depends on the operating system. Here we'll keep it simple and assume you've defined the *dataroot* variable as above properly for your circumstances, so that you can simply type:

```
os.chdir(dataroot)
```

Once you have changed to the directory containing the datafiles, you can then construct a Dataset from a text file (and name it 'cho') as follows:

```
cho = datasets.Dataset('ChoCycling.dat','cho')
```

Option 2: Loading files specified with full file paths

If instead you wish to specify full path to the data files (rather than changing directories), assuming you've defined the data path as above, you would use this command:

```
cho = datasets.Dataset(os.path.join(dataroot,'ChoCycling.dat'),'cho')
```

As a final check that creating the Dataset was successful, the following "cho.numRows" Python statement should return 384. E.g.:

```
cho.numRows()
384
```

2.2.2 Attach labelings to a dataset

(The following examples assume we used Option 1 above, namely that we changed directories prior to loading files. If you used Option 2 you can adapt the statements as needed to use your complete data path).

Once we have a dataset loaded we can attach one or more "Labelings" to it to provide additional information about the various rows and columns (or even individual data values) of the dataset. For instance one typically wants a unique name (e.g. a gene ID) for each row attached to the dataset so the summary plots can tell you the name of an interesting looking vector. With the Cho data set, this would be ORF (gene) identifiers. The file "ORFs.rlab" contains one identifier per line and can be associated with the dataset as follows:

```
orfs = labelings.Labeling(cho, 'orfs')
orfs.labelRows('ORFs.rlab')
cho.setPrimaryRowLabeling(orfs)
```

A "primary" row labeling needs to have a unique value for each row. For the Cho dataset the ORF identifiers serve that purpose.

Once we have our primary labeling we might as well attach some other useful pieces of information. Included with the Cho dataset are additional row annotations containing such things as gene common names, and cluster membership (the cluster IDs to which a gene belongs). We have cluster labelings for both manually-assigned clusters (assigned by Cho in (Cho et al., 1998)) and computer-generated clusters (derived by the CompClust DiagEM algorithm). We can load these additional labelings as follows:

```
names = labelings.Labeling(cho, 'common name')
names.labelRows('CommonNames.rlab')

em = labelings.Labeling(cho, 'diagem clusters')
em.labelRows('EM.rlab')

cho_clustering = labelings.Labeling(cho, 'cho clusters')
cho_clustering.labelRows('ChoClassification.rlab')
```

We can also attach information to the columns of the dataset. The primaryColumnLabeling is used for many of the condition (x-axis) labels along the plots. For the Cho data set, this would be the time (in hours) at which each sample was taken for the time course experiment.

```
times = labelings.Labeling(cho, 'time points')
times.labelCols('times.clab')
cho.setPrimaryColumnLabeling(times)
```

3 Construct a PCAGinzu object

We begin principal component analysis (PCA) and interpretation of a given dataset by simply constructing one of the "PCAGinzu" class of objects, e.g.:

```
p = pcaGinzu.pcaGinzu(cho,verbose=True)
```

In the above example, we constructed a basic "pcaGinzu" class object given the Cho dataset. We also turned on an optional verbose flag so we could observe what the is happening during construction of the PCAGinzu object. Behind the scenes three things happened:

1. a RowPCAView is constructed- this is a "view" of the data after the PCA eigenvectors are calculated and the data has been rotated into the PCA space;
2. the genes (rows) at the extremes of each principal component are identified and labeled; and
3. the conditions (columns) in which those extreme genes (rows) are identified and labeled.

There are two different sub-types of PCAGinzu objects that one can instantiate, each providing a slightly different graphical view of the analyzed dataset. The "PCAGinzu" class is defined in compClust.iplot package (See iplot import command in section 2.1). It will create interactive plots using the IPlot API (which can be rendered in either Tk or static plots (for web pages)).

Alternatively there is also "pcaGinzuVisualizeMatplotlib" class within compClust.mlx.pcaGinzu module, which will render static, publication-quality plots using matplotlib. pcaGinzuVisualizeMatplotlib also has an additional batch processing function which can iterate over all of the principal components and create all the available PCAGinzu analysis output for each component.

No matter which version of PCAGinzu object we create, we need to pass in the dataset we want to analyze. The dataset is the only required argument to these constructors. The following subsections describe some of the pcaGinzu object's optional parameters that further control the PCA interpretation analysis.

3.1 Determining the Extreme Rows (aka Extreme Genes)

When constructing the PCAGinzu object, the dataset to be analyzed is the only required argument. One of the most important optional arguments allows users to specify how the extreme data points, aka the "extreme genes", (formerly referred to as "outliers" or "outlier rows") are selected.

The preferred method that we implemented permits the user to specify a likelihood threshold below which a data point (a row, e.g. a gene) is considered extreme. All of the data points form a 1-dimensional distribution of values when projected along a particular principal component's axis. We assume the values are distributed roughly in a Gaussian shape (we've observed this to be true for most gene expression data).

For each data point we can compute a likelihood of membership to that Gaussian distribution. Points having a likelihood less than or equal to this cutoff (at either end of the axis) are itentified as "outliers" aka "extreme points" for that principal component. This threshold can be controlled by specifying the optional *outlierCutoff* argument, e.g.:

```
p = pcaGinzu.pcaGinzu(cho,outlierCutoff=0.05)
```

The lower the likelihood threshold, the fewer points will be included in the extreme point set. Likewise, the higher the threshold, the larger the set of extreme points. In some cases there will be NO extreme points at one or both ends of a principal component's axis. In this case you might consider increasing the threshold.

An older ad-hoc method we employed permits the user to specify an explicit number (e.g. 10 or 50) of the most extreme points to select at each end of a principal component's axis. This can be specified using the optional *nOutliers* argument, e.g.:

```
p = pcaGinzu.pcaGinzu(cho,nOutliers=10)
```

Only one of the two above optional arguments, *outlierCutoff* or *nOutliers*, can be given to the constructor. If neither argument is given the analysis defaults to *outlierCutoff*=0.001.

In order to better understand the two outlier selection approaches, we can graphically compare the extreme point sets resulting from each approach. Figure 1 shows a scatter plot of the data points in the PC1 vs. PC2 space with the extreme points selected using *outlierCutoff*=0.05 highlighted. Figure 2 shows the same scatter plot but with the extreme points selected *noutliers*=10 highlighted. In cases where the distribution of a principal component is slightly skewed, the *outlierCutoff* approach may identify more extreme points at one end of the distribution. The *nOutliers* approach will always generate an even number of high and low extreme points.

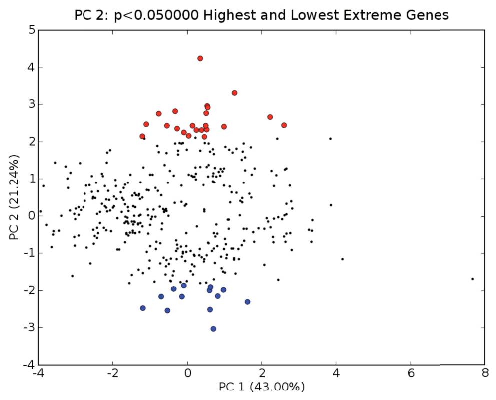

Figure 1: Cho PC2 Extreme Genes at outlierCutoff=0.05

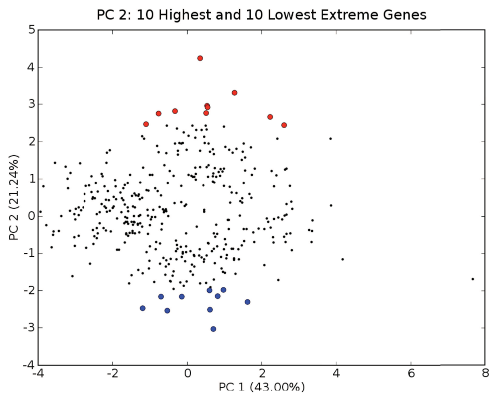

Figure 2: Cho PC2 Extreme Genes at nOutliers=10

Determining how many extreme points to use, and thus the appropriate settings of these parameters for a particular data set, is somewhat heuristic. In gene expression analysis, identifying fewer extreme genes allows you to obtain small but well-refined gene sets. Recall that these extreme genes are used to identify groups of conditions (e.g. tissues or samples) in which the genes are differentially expressed.

If you have a small number of genes, e.g. approximately 20 high and 20 low extreme genes, you will more precisely focus your search for conditions in which only those genes are differentially expressed. If you have a large number of genes, e.g. approximately 200 high and 200 low extreme genes, you get a broader, more inclusive set of genes, and subsequently will identify only those conditions in which the larger sets of 200 high vs. 200 low extreme genes are differentially expressed. It is plausible that two analyses, one producing fewer extreme points and one producing larger extreme points, can each provide informative results that emphasize different phenomena.

3.2 Setting the significant condition group threshold

Another important optional parameter is the *sigCutoff* threshold. For each principal component we partition the original dataset columns (aka conditions, e.g. samples, tissues, etc.) into three condition groups *Up*, *Flat*, and *Down*, depending on the value of this principal component's high and low extreme points in the original dataset column space.

sigCutoff is the significance level below which you reject the hypothesis that the high extreme points and low extreme points are drawn from the same distribution. A Wilcoxon rank sum test is used to determine the likelihood of this hypothesis for each original dimension (condition), and the conditions that meet this threshold are labeled as "Up" or "Down" conditions, depending on the direction of the difference. The default value of this threshold is *sigCutoff* = 0.05.

For an original column dimension (condition) of the dataset, if the mean of one principal component's high extreme points is much higher than the mean of that principal component's low extreme points, this condition is likely to be placed in the "Up" condition set for that principal component. Likewise, if the mean of the low extreme points is much higher than the mean of the high extreme points, this condition is likely to be placed in the "Down" condition set. The exact determination is based on whether the p-value of the Wilcoxon rank sum test is equal to or lower than *sigCutoff*.

It is also possible to estimate the Student's t-Test p-value as an alternative. We chose Wilcoxon rank-sum test because it has an adjustment for low sample sizes, and in part because it could compare two different sample set sizes.

4 Review PCA results using Tk graphics

The advantage of the Tk plots (available within the CompClust IPlot package) is that some are designed to be interactive. Users can click on data points or line plots to find out the name of an interesting vector (e.g. to get the identity of an extreme gene).

To create the interactive plots, first we need to create the IPlot-based PCAGinzu object. In the case of the yeast cell cycling dataset this is fairly quick, but can take tens of minutes to nearly an hour for something like a 33,000 x 158 dataset such as the GNF dataset.

```
ipcaginzu = IPlot.PCAGinzu(cho,outlierCutoff=0.05)
```

4.1 View PCA projection scatter plots

Given this new IPlot-based PCAGinzu object, we will next create and display a PCAGinzu data point PCA projection scatter plot. This will allow us to see where all of the points lie in a subspace of the principal component dimensions (e.g. in the principal component 1 vs. principal component 2 sub-space), and visualize the points (e.g. genes) that are extreme for one principal component.

The following call to plotPCvcPCWithOutliersInY will create a scatter plot displaying all of the data points (rows, e.g. genes) in a 2D subspace of (a projection within) the principal component dimensions. For example, we will plot principal component 1 along the X axis, and component 2 along the Y axis. The high and low extreme points for the second principal component that have likelihoods less than or equal to 0.05 are highlighed in red and blue, respectively. That value extreme point likelihood cutoff was set by the above PCAGinzu call.

```
ipcaginzu.plotPCvsPCWithOutliersInY(0,1)
```

The above command produces the plot that was previously shown in Figure 1.

Users can plot any pair of principal components against each other. Our standard is to plot PC_{n-1} vs. PC_n, or in some cases PC_n vs. PC_1 because PC_1 often corresponds to the magnitude of expression. Whichever principal copmonent is specified by the second argument to the function will appear on the Y-axis and it's extreme points will be highlighted.

These plots are interactive, so clicking on a single data point allows you to see the identity of that specific gene.

Note: One crucial detail is that the IPlot-based PCAGinzu object uses zero-based array indexing, e.g. it counts each principal component as 0, 1, 2, etc. The web interface is built using this version, and because Python arrays and lists are 0-based, it was simpler to make these functions compatible. The MLX-based pcaGinzu and pcaGinzuVisualizeMatlab objects are 1-based, i.e. arguments are naturally numbered as the principal components are 1, 2, 3, etc.

4.2 View extreme point trajectory plots

Now that we have identified the points (e.g. genes) that occupy the extremes of a particular principal

There are two extreme point trajectory plots: the first shows the extreme point trajectories in the same order as the dataset's original columns are ordered; the second shows the trajectories with the columns (the x-axis) reordered to emphasize the dimensions (conditions) in which the high and low extreme points are most separated.

The first trajectory plot is sorted in the original column order (again remember that this IPlot-based API uses 0-based indexing). The following call will plot the extreme point trajectories for the second principal component in original order:

```
ipcaginzu.plotPCNOutlierRowsInOriginalColumnOrder(1)
```

In the resulting figure, it's clear that the high genes at one end of principal component 2 are co-expressed in a specific phase of the cell cycle, and the low genes at the other end of the PC2 axis are co-expressed in the opposite phase of the cell cycle.

These plots are interactive, so clicking on a single trajectory plot's vertex allows you to see the identity of that specific gene.

The second trajectory plot is sorted by the difference of the mean "high" vectors (red) and the mean "low" vectors (blue), in order to emphasize the conditions that most affect that principal component. The following call creates a plot of the extreme point trajectories in that order (mean(high) - mean(low)) for the second principal component.

```
ipcaginzu.plotPCNOutlierRowsInSigGroupOrder(1)
```

This plot can be made much more meaningful by looking at the identities of the reordered conditions. This particular plot's X-axis does not (yet) have the ability to show those identities, but we can see the ordering in the PCA interpretation text output, as well as with the equivalent matplotlib plotting function shown later in the tutorial.

5 Review PCA results using matplotlib graphics

Matplotlib was designed to provide a Matlab like environment within Python, when using the matplotlib plots it is convenient to have the rest of matplotlib available. Matplotlib is one package that CompClust depends on, and it can be obtained from http://matplotlib.sourceforge.net. First we will import the matplotlib package into our shell:

```
from matplotlib.pylab import *
```

Both classes, pcaGinzu and pcaGinzuVisualizeMatplotlib, share the same constructor, so they take the same list of parameters, e.g. the dataset to operate on, the *outlierCutoff* or *nOutliers* number of high and low outliers, the significance threshold, and what is the highest numbered principal component to analyze.

The following function call creates a PCAGinzu object using the Cho dataset, this time using nOutliers=10 and sigCutoff=0.01:

```
pcaginzu = pcaGinzu.pcaGinzuVisualizeMatplotlib(cho, nOutliers=10, sigCutoff=0.01)
```

Note: Again, an incredibly important difference between the previous IPlot-based functions and the following matploglib-based functions is that in the matplotlib functions the principal component numbers use 1-based indexing. That is, the first principal component is 1, the second is 2, etc.

5.1 View outlier scatter plot

Here we create and display a scatter plot of the first and second principal components,

```
pcaginzu.plotPCvsPCWithOutliersInY(1,2)
```

Note: You might need to execute the command "show()" following the above statement in order to make the plot window appear. If you are using ipython in a UNIX-based operating system, you can optionally use "ipython -pylab" to make figures appear without the show() command. The Matplotlib http://matplotlib.sourceforge.net documentation has more information for interacting with plot windows, including zooming and panning within plots and saving the figure to files.

5.2 View outlier trajectory plots

We will now display the extreme point trajectories, first in original order, i.e. the unsorted native ordering of the dataset. This ordering would be the same ordering as any other unsorted plot that one might chose to create.

```
pcaginzu.plotPCNOutlierRowsInOriginalColumnOrder(2)
```

In the next plot we will see the same data as above but sorted to emphasize thohse conditions with the greatest the difference of mean "high" and mean "low" expression. "High" vectors are the red ones, defined as those that have the maximum value on the given principal component axis. Likewise, the low vectors in blue are (as one might guess) are the ones that started off as the most negative of all the values in the selected principal component.

This graphic frequently has an "X" shape to it. The left side of the plot can emphasize any conditions in which the high extreme points (in red) have significantly higher values than the low extreme points. Likewise, the right side of the plot emphasizes any conditions in which the low extreme points (in blue) have significantly higher values than the high extreme points.

```
pcaginzu.plotPCNOutlierRowsInSigGroupOrder(2)
```

As with the IPLot-based equivalent graphic, this reordering of the conditions is not very meaningful without knowing the ordering. An optional second argument allows us to attach names to the X-axis, which is useful if the number of dimensions to plot is not very large. (In the GNF dataset, for example, we can't attach labels to the 158 dimensions and still expect to see the distinct labels). Here, we retrieve and attach the time points in hours for the cell cycle data, and provide those labels to the routine.

```
l = cho.getLabeling('time points')
times = [x[0] for x in l.getLabelsByCols(range(17))]
pcaginzu.plotPCNOutlierRowsInSigGroupOrder(2,times)
```

Now it is clearer that the high extreme genes are much higher than the low genes at around 9-10 hours and 16 hours. Likewise the low genes are much higher than the high genes at approx 3-6 hours. There is essentially no significant difference in the high and low gene's expression in hours 12-15 and hours 1-2.

6 List a principal component's extreme points and significant conditions

Though plots are a nicely visual way to look for interesting vectors, to actually attempt to find which PCA condition might be associated with a particular labeling is more easily done by looking at lists.

6.1 Get a list of extreme points

To get a report of the high and low extreme point sets for a given principal component, you can call the pcaGinzu.getOutputForPCNOutliers() function. This function takes as input the principal component number (in 1-origin indexing) that you want to view, plus a list of row labeling names that you want to see included in the report. These labelings need to be row labelings, such as those created by the labeling.labelRows function (described in the data loading section of this tutorial).

```
pc1egs = pcaginzu.getOutputForPCNOutliers(1, ['cho clusters', 'diagem clusters', 'common name'])
```

The resulting list of lists returned by that command should contain the same information as the following table:

PC-1 10 High/Low	PC-1 Value	diagem clusters	cho clusters	common name
high	7.66496608144	4	M	WSC4
high	4.17705992859	4	M	YOL019W
high	3.87001547533	4	M	HOF1
high	3.84837129339	5	Early G1	SUR1
high	3.35847421047	4	M	BUB3
high	3.34419451262	4	M	CDC5
high	3.34037443779	4	M	YML034W
high	3.27043501501	4	M	COT1
high	2.82125356759	4	M	HDR1
high	2.80269836671	4	G2	YIL158W
low	-3.26930983481	2	Late G1	HO
low	-3.39173779979	2	Late G1	RNR1
low	-3.44848612811	2	Late G1	YLR183C
low	-3.56081559301	2	Late G1	CDC54
low	-3.59901879069	2	Late G1	YOR144C
low	-3.60057680134	2	Late G1	HST3
low	-3.64414373489	2	Late G1	TOF1
low	-3.79017028645	2	Late G1	SPH1
low	-3.86578252915	2	Late G1	YPL264C
low	-3.93550058084	2	Late G1	HHO1

Interestingly it appears that principal component one helps to differentiate between M phase and the Late G1 phase of the cell cycle. A similar report for principal component 2 suggests that PC2 discriminates between Early G1 and S/G2 phase (which corresponds well with the PC2 extreme gene trajectory plots generated earlier).

The getOutputForPCNOutliers() returns the above result as a Python list of lists. We provide a convenience function called pcaGinzu.write2DStringArrayToFile() if you want this output this result to a file. This function takes the result of the getOutput command, a file name, and an optional delimiter (which defaults to tab). Here is an example howing how to call this output function:

```
pcaGinzu.write2DStringArrayToFile(outliers, 'pc1-extreme-genes.txt')
```

Later we will show how to generate the extreme point lists for all principal components in a batch operation mode.

6.2 View significant condition list

While getOutputForPCNOutliers() lists interesting extreme points and can show interesting patterns along the rows of a dataset, getOutputForSigGroups() displays the conditions in the extreme gene data-driven ordering, specifically ordered by mean(high) - mean(low). Conditions are

Unfortunately the yeast cell cycling dataset only has one covariate, time (the "time points" column labeling), so it is not really a good example of this feature.

```
pcaginzu.getOutputForSigGroups(1, ['time points'])
```

PC-1 10-outlier Up/Flat/Down Columns	times
up	7 h
up	8 h
up	6 h
up	15 h
up	16 h
up	14 h
up	5 h
flat	13 h
flat	4 h
flat	0 h
down	9 h
down	12 h
down	10 h
down	11 h
down	3 h
down	1 h
down	2 h

If one looks carefully one can start to see hints of the cell cycle that was captured by this output. It is most instructive to look at the top and bottom of the list where the differences are greatest. At the top, at 6,7,8 hours we observe the strongest high over low differences. At the other end of the condition list, at a completely different phase of the cell cycle, hours 1,2,3 show strongest low over high (or "down") conditions. This is also evident in the corresponding trajectory plots:

```
pcaginzu.plotPCNOutlierRowsInOriginalColumnOrder(1)
pcaginzu.plotPCNOutlierRowsInSigGroupOrder(1,times)
```

7 Analyzing condition covariates

As an additional means of interpreting principal components, we would like to determine if any individual principal components offer a means to predict values of condition covariates attached to the dataset columns. There are a number of approaches for determining if a covariate correlates well with a specific principal component. The approach we took is to identify those covariates correlate with a principal component's derived condition partitioning. As described in the previous section, a principal component's extreme points can be used to partition the conditions into three groups "up", "flat", and "down".

For a given principal component PCn, we can compute a score for each condition (e.g. tissue or sample) covariate indicating the degree to which that covariate is correlated with that PCn's significant condition grouping into the Up, Flat, or Down conditions. A score will be generated for each user-supplied column labeling attached to a dataset (i.e. all of the dataset's covariate annotations, e.g. age, tissue, treatment). For discrete covariates a normalized mutual information (NMI) score is computed, indicating the degree agreement between the covariate's discrete values and the condition Up/Flat/Down grouping (a higher score means better agreement). For continuous covariates, Wilcoxon rank sum tests are generated, giving the likelihood of 2 sets of covariate values belonging to the same distribution. Three pairs of condition groups are scored: (Up vs. Flat), (Up vs. Down) and (Flat vs. Down).

Beyond identifying extreme points and the condition partitionings they generate, the above covariate analysis allows us an additional data-driven, unsupervised approach to help interpret each principal component. A report of condition covariates that correlate with condition partitions essentially generates a number of hypotheses at a specified significance threshold. It is then up to the researcher to follow up on those hypotheses and verify whether an observed correspondence between e.g. a particular set of of genes, tissues and covariates is relevant and in fact meaningful. Of course this covariate analysis approach is inherently limited by the number of column covariates available. In the case of the Cho dataset, and likewise with the GNF human dataset we have only one column covariate (in Cho we only have Time, and in GNF we only have tissue).

7.1 Covariate analysis of the PGC diabetes dataset

To provide a more meaningful demonstration of this feature, we will switch to an analysis of a human diabetes expression dataset (Mootha et al., 2003) from Broad Institute Cancer Program dataset repository http://www.broad.mit.edu/cgi-bin/cancer/datasets.cgi. We will refer to this as the PGC diabetes dataset.

This dataset contains 10,983 gene probes measured across 43 skeletal muscle samplesfrom 3 diagnosis groups: normal glucose tolerance (NGT, n=17); impaired glucose tolerance (IGT, n=8); and Type 2 diabetes mellitus (DM2, n=18). We used our PCA interpretation software to perform an unsupervised analysis of the DM2 vs. NGT subset (comparable to the previous published result). As described in our Methodology section, PCEG sets were determined using an extremeThresh likelihood threshold of 0.001, which yielded about 50 high and 50 low extreme genes per principal component. For more information on this dataset, see (Roden et al., 2005).

Because this dataset contains more than 50 covariates, it provides us a nice opportunity to interpret each principal component by searching for covariates that correlate well with expression patterns in the PCEG sets.

We can perform the principal component analysis of this dataset as follows:

```
pgc = LoadExample.LoadPGC()
p = pcaGinzu.pcaGinzuVisualizeMatplotlib(pgc,verbose=True,outlierCutoff=0.001)
```

The PCAGinzu software has a variety of functions to support the covariate analysis. The scoreColumnLabelingsForPCN() function attempts to identify any column labelings that correlate well with a particular principal component's condition partitioning. Due to the slightly complex nature of the scores we calculate (one score for discrete covariates and three scores for continuous), this function returns a list of ColumnScore objects that contain the results, one ColumnScore per covariate. To analyze all covariates against a single principal componenent e.g. PC12, you can call:

```
pc12scores = p.scoreColumnLabelingsForPCN(12)
```

However, because the details of the list of ColumnScore representation are a bit involved and there are so many scores to calculate and examine, we will move ahead and instead show how to generate all covariate scores across all principal components. This complete covariate analysis, which may take a minute or so depending on your hardware and the size of the dataset, can be performed as follows:

```
covariateScores = pcaGinzuScoring.getAllResults(p,1,35)
```

This result is a dictionary containing a set of covariate scores, indexed by principal component number. One way we recommend viewing the results is to generate a summary of only those covariates whos scores meet or exceed a given threshold, indicdating they may be interesting and suggestive of further study. Because low scores are better for continuous, and high scores are better for discrete, there are two separate functions for showing summaries of covariate scores. For continous covariates their score must be less than or equal to the given threshold; likewise a discrete covariate's score must be greater than or equal to the given threshold. E.g.:

```
continuousSummary = pcaGinzuScoring.summarizeContinuousResults(covariateScores,0.01)
discreteSummary = pcaGinzuScoring.summarizeDiscreteResults(covariateScores,0.8)
```

Each summary returned is a list of the principal components and, for each, a list of any covariates that were observed to correlate well with that principal component. You can observe in the continuous covariate summary for example that at a Wilcoxon significance threshold of 0.01, principal component 12 is well correlated with four different Type2c fiber covariates. One may then "drill down" to the value distributions underlying these scores to investigate the relationship, e.g.:

```
p.plotScores('Type2c_Area_(Percent)',12)
```

Again it's unfortunate that the PGC dataset has only two covariates that are discrete: "status" and "Samplename_at_WICGR". Neither column correlated with a principal component at a level exceeding the default average NMI threshold of 0.8. If either did, we would want to drill down to see the underlying confusion matrix that illustrates high mutual information between that covariate and a principal component's Up/Flat/Down column labeling.

The pcaGinzuScoring module has a variety of functions to dump you output the results of covariate analysis to a file. Here we send the complete table of results to a file (after generating a results table dictionary):

```
resultsTable = pcaGinzuScoring.generateResultsTable(covariateScores)
pcaGinzuScoring.writeResultsTableDictToFile('all-covariate-scores.txt', resultsTable)
```

The resulting file will contain one column per covariate and one row per principal component. The cells contain the minimum of the three Wilcoxon significance scores for continuous covariates, and the average NMI score for discrete covariates. For continuous covariates, if there are too few data points to compare in any of the Up/Flat/Down partitions (controlled by an optional minSetSize argument to scoreColumnLabelingsForPCN, defaults to minSetSize=2), a value of 2 is reported so that situation can be recognized. For a large number of covariates this is a dense result table and is only useful if you are going to filter the results further by e.g. by loading into a spreadsheet and applying a threshold.

The following example shows how to use writeSignificantResultsToFile to send a covariate analysis significant results summary to a file:

```
pcaGinzuScoring.writeSignificantResultsToFile('significant-continuous-covariate-scores.txt', continuo
pcaGinzuScoring.writeSignificantResultsToFile('significant-discrete-covariate-scores.txt', discreteSu
```

The resulting files offers a nice summary of the most relevant covariates across all of the principal components.

It might be useful to analyze a particular labeling of interest to determine which principal component correlates best. For example,

```
pcaGinzuScoring.displayScoresForColLabeling('status', covariateScores)
```

Among others, principal component 14 seems to have the highest correlation to the status covariate, and so perhaps warrants a bit of investigation.

7.2 Covariate analysis of Cho dataset

The scoreColumnLabelingsForPCN() function attempts to identify any column labelings that correlate well with a particular principal component's condition partitioning.

Unfortunately the Cho yeast cell cycling dataset only has one covariate, time (the "time points" column labeling), so it is not really a good example of this feature. Furthermore, it was loaded as a discrete covariate, so we're not able to do a proper analysis of what is naturally a continuous covariate (unless we explicitly define a new column labeling containing time as a numeric covariate). Nonetheless, we can give it a try as a discrete variable to see what happens.

The function that computes covariate scores for a specific principal component is called as follows:

```
pcaginzu.scoreColumnLabelingsForPCN(1)
```

This returns a ColumnScore object to hold 1 discrete covariate correlation score, and 3 continuous covariate correlation scores.

We can determine which principal component best correlates with time, by looking at all of the principal components. E.g.:

```
for i in range(1,pcaginzu.rowPCAView.numCols+1):
  print i, pcaginzu.scoreColumnLabelingsForPCN(i)[0].scores
```

The above commands result in the following output:

```
1 0.682976972108
2 0.686202158131
3 0.625067714467
4 0.657751569535
5 0.582238520955
6 0.539481092126
7 0.602061519391
8 0.563922061919
9 0.0
10 0.539481092126
11 0.539481092126
12 0.539481092126
13 0.0
14 0.0
15 0.539481092126
16 0.0
17 0.539481092126
```

Apparently principal component 2 generates the condition partitioning that best explains time in this dataset.

8 Performing batch PCA interpretation

One advantage of the matplotlib based pcaGinzu object is that it contains a function called generateResults() which implements a batch analysis mode. This function can be used to generate all of the graphical and textual results across all principal components.

Because calls to generateResults() will end up creating a large number of plots (one PCA projection plot and two trajectory plots per principal component), we recommend you execute generateResults() in a fresh python shell session in which matplotlib figures are not automatically appearing. If you are using CompClustShell or a plain Python shell, start a new session and do not use the show() command prior to calling generateResults(). If you are using an ipython shell, start a new session without using the optional "-pylab" argument. It is not a huge problem if all of the plots appear, as you can always use close('all') to get rid of them. But they really slow down the plot generation.

So in a new shell, import the necessary packages (see 2.1 for all of our import statements, or just execute the a minumum set below), then load the dataset, and create the pcaGinzu object:

```
from compClust.mlx import pcaGinzu
from compClust.util import LoadExample

cho = LoadExample.LoadCho()
pcaginzu = pcaGinzu.pcaGinzuVisualizeMatplotlib(cho, nOutliers=10, sigCutoff=0.01)
```

The following call will create a complete set of of plots (PCA projections with extreme points, trajectory plots, etc) and tab-delimited text tables (extreme point and significant condition lists) in the current directory for all of the principal components analyzed. The two arguments passed to the function specify a list of row labeling names and a list of column labeling names that should be included in the resulting extreme point and significant condition text output files.

```
pcaginzu.generateResults(['cho clusters', 'diagem clusters', 'common name', 'orfs'], ['time points'])
```

A set of extreme point scatter and trajectory plots, as well as extreme point lists and ordered condition lists with signficant condition groups will be generated in the current directory.

While a directory full of files can be helpful in certain circumstances, we also find the CompClustWeb interface a useful means to browse through the results of PCA interpretation.

9 Further Information

9.1 Publication: Mining Gene Expression Data by Interpreting Principal Components

The web site http://woldlab.caltech.edu/publications/pca-bmc-2005 contains additional information including the supplemental materials and links to the publication itself.

9.2 The CompClust Python Package

CompClust is a Python package written using the pyMLX and IPlot APIs. It provides software tools to explore and quantify relationships between clustering results. Its development has been largely built around requirements for microarray data analysis, but can be easily used for other types of biological array data and that of other scientific domains.

If your interested in learning more about what you can do with the CompClust Python Package, check out the tutorials in the next section. When using the CompClust Python package the analysis possibilities are only limited by your imagination ... Well, that and CPU power and RAM, your understanding of Python and the CompClust Python package. Well, at least you don't have to be limited by what the GUI can do.

The latest version of the CompClust package can be found at http://woldlab.caltech.edu/compClust/.

9.3 Other CompClust Tutorials

The following tutorials can be found at http://woldlab.caltech.edu/compClust/.

9.3.1 CompClustTk Manual and Tutorial

"The CompClustTk Manual and Tutorial", written by Brandon King, contains general introductory information for CompClust as well as specific information on how to use CompClustTk.

9.3.2 A First Tutorial on the MLX schema

"A First Tutorial on the MLX schema", written by Lucas Scharenbroich, covers the MLX schema. One should read this if one wanted to explore the full power of compClust using python.

9.3.3 A Quick Start Guide to Microarray Analysis using CompClust

"A Quick Start Guide to Microarray Analysis using CompClust", written by Christopher Hart, covers how to use the Python CompClust environment to do microarray analysis. It may give one a better understanding of the IPlot tools (Trajectory Summary, Confusion Matrices, etc.). It will also teach one how to use some of the more advanced features of CompClust which haven't been exposed to CompClustTk and CompClustWeb.

10 Acknowledgements

References

Cho, R. J., Campbell, M. J., Winzeler, E. A., Steinmetz, L., Conway, A., Wodicka, L., Wolfsberg, T. G., Gabrielian, A. E., Landsman, D., Lockhart, D. J., and Davis, R. W. (1998). A genome-wide transcriptional analysis of the mitotic cell cycle. *Mol Cell*, 2(1):65–73. (eng).

Hart, C. E., Sharenbroich, L., Bornstein, B. J., Trout, D., King, B., Mjolsness, E., and Wold, B. J. (2005). A mathematical and computational framework for quantitative comparison and integration of large-scale gene expression data. *Nucleic Acids Research*, 33(8):2580–2594.

Mootha, V. K., Lindgren, C. M., Eriksson, K. F., Subramanian, A., Sihag, S., Lehar, J., Puigserver, P., Carlsson, E., Ridderstrale, M., Laurila, E., Houstis, N., Daly, M. J., Patterson, N., Mesirov, J. P., Golub, T. R., Tamayo, P., Spiegelman, B., Lander, E. S., Hirschhorn, J. N., Altshuler, D., and Groop, L. C. (2003). Pgc-1alpha-responsive genes involved in oxidative phosphorylation are coordinately downregulated in human diabetes. *Nat Genet*, 34(3):267–273.

Roden, J., King, B., Trout, D., Wold, B., and Hart, C. E. (2005). Mining gene expression data by interpreting principal components. *BMC Bionformatics (in press)*.