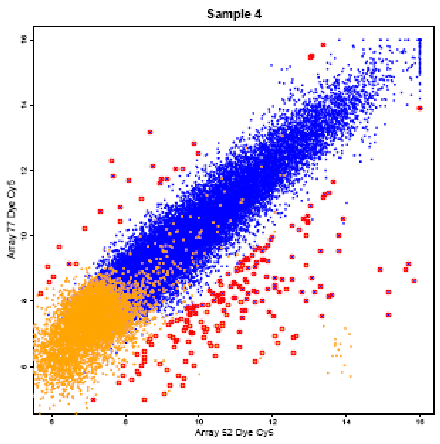
Analysis of Microarray Timecourse Data



by Stuart David James McHattie

Supervised by Katherine Denby,

Vicky Buchanan-Wollaston

and Andrew Mead of Warwick HRI

Table of Contents

Introduction

Microarray analysis can prove to be just as challenging as collecting the experimental data in the first place. Often a small number of arrays turn into a huge quantity of data at an exponential and alarming rate. Each array consists of spots for every gene being studied, and with 2-dye arrays, technical replicates, biological replicates and sometimes replicate genes on the same array, the data sets can reach several million expression data points very quickly.

The work being done in this project is the analysis of microarray data from an experiment performed to study senescence in leaves of *Arabidopsis thaliana* over a period of time. The intention is to acquire clusters of genes that are co-expressed and are therefore likely to be from the same pathways. This information could then be used to produce models of these pathways which could potentially lead to understanding the process of senescence better and producing more tolerant plants which can live longer and produce more yield for food.

The analysis is all computer based, and involves checking the data for quality, using ANOVA techniques to fit a model to the data and then using this model to select significant gene expression changes which lead to clustering of genes. The software in use during this work is partly produced by the Churchill Group in the Jackson Laboratory in Maine, USA[1], by Nicholas Heard in Imperial College London[2] and by myself at The University of Warwick Systems Biology DTC[3].

Experimental Design

The Plants

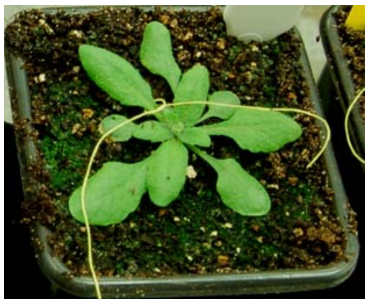


Figure 1 - Cotton marking leaf 7

The experiment from which the data was collected was performed over a 40 day period. During this time, individual plants were subjected to long days: 16 hours of sunlight. At first emergence, the seventh leaf in the development of the plant was marked by tying cotton around the base of it (Figure 1); this would act as an identifier later in the experiment. Approximately 19 days after sowing, leaf seven would be fully developed and this indicates the beginning of the time course. Over the course of the next 21 days, the leaves were harvested. The first harvest was on the morning of the 19th day. Four leaves were collected and processed to extract samples of cDNA from them. The collection of four leaves at each time point gives four biological replicates across the experiment. A further four samples were collected in the afternoon of the 19th day, giving the second timepoint. No more samples were collected until the morning of the 21st day. This trend continued until 22 samples had been collected on alternate days; 11 in the morning and 11 in the afternoon.

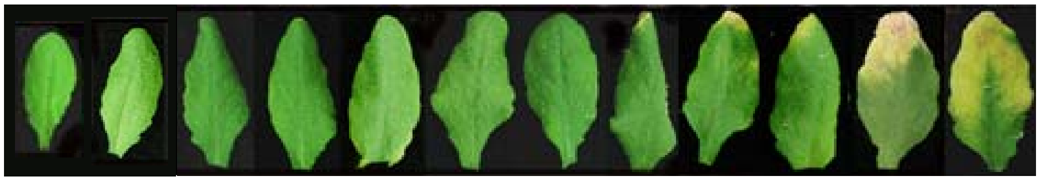


Figure 2 - Senescence of leaves over the 21 day period

During this time period, the leaves of the plants began to senesce (Figure 2) and so the cDNA samples should reflect this behaviour. Other expected changes in gene expression during the timecourse are in genes responsible for circadian clock rhythms. These are the genes which regulate response to sunlight.

The Microarrays

The microarrays used for analysis of the samples were Complete *Arabidopsis* Transcriptome Micro Arrays (CATMA). They each contain 32,488 spots, of which only 30,336 are from the genome of *Arabidopsis*; the rest being test spots with high binding affinity and blank spots. These arrays are produced by Warwick HRI using a sterile spotting machine.

The CATMA database[4] from which these arrays are based, was first produced in 2000[5] and has since developed into a more comprehensive list of genes. Each gene is allocated a gene sequence tag, and whilst the project started with 25,498 genes, this has since grown to the current figure of 30,886 genes.

The 32,488 spots on each array are arranged in meta-grids of 26 × 26 spots. In these grids, the first line are test spots and the remaining two corners are also test spots. There are 4 meta-columns and 12 meta-rows of these grids and each metagrid is produced by a different pin during the manufacture of the slides.

There are 22 time points for which 4 biological replicates exist giving 88 samples. Each sample is compared with 4 other samples on two dye CATMA arrays, giving 4 technical replicates for each sample. This gives 176 arrays, and therefore 176 arrays × 2 dyes × 30,336 spots = 10,678,272 expression values to analyse.

The technical replicates were produced in such a way that each sample is crossed with two other samples of the same time of day and two of the alternate time of day. This way, the design of the experiment was robust to removal of data where necessary.

The ℝ Project

ℝ[6] is an application written and compiled to run on all the major operating systems. The two binaries used in this project were for Windows and Macintosh OS X. It is a language and environment for statistical computing and graphics and forms the perfect basis for handling huge datasets as it is able to support many forms of data structures and the code is very good at selecting specific data from these structures to perform mathematical operations on them. There are also a large number of libraries available for ℝ to perform statistical operations, as well as some that are very specific to tasks such as handling microarray data. The interface is relatively crude, offering a mere terminal prompt to enter commands, but scripts can be written to run commands in a more automatic fashion. However, this terminal style data entry gives ℝ a lot more power than alternative statistical package, and the fact it is covered by the GNU General Public Licence means it is updated regularly to include new routines and to improve the syntax used in the language. The current version is 2.5.0, released 24/04/2007, and this is the version that has been used throughout this project.

MAANOVA

MAANOVA[7] is one of the libraries available in ℝ and was developed by the Jackson Laboratory in Maine, USA. It is an acronym for MicroArray ANalysis Of VAriance and is designed to take raw microarray data and manipulate it into a form which can be analysed for data quality and then transformed for model fitting which separates the effects of the experimental design, showing how significant the changes in gene expression are due to these different effects.

The original version of MAANOVA was developed for Matlab, but a version that runs in ℝ was developed in March 2001 with a basic but functional set of tools for analysing microarray experiment data.[8] Since then, many new functions have been added to MAANOVA, although recent updates have been for bug removal and to comply with newer ℝ routines.

Whilst MAANOVA is very comprehensive in its abilities to analyse microarray data, a lot of the routines were developed a long time ago and were designed with smaller experiments in mind. The example dataset is for 24 arrays that are 5,776 spots in size. This is considerably smaller (less than 3%) than the number of spots and arrays in the dataset being studied for this project. The key flaw is that most of the data checking functions output graphical information which are only displayed on the screen and are for assessment by eye. This could be automated to a certain extent and the graphical output could be placed in a form that can be stored; in this case a PDF file.

Other problems with MAANOVA include the fact that some of the lesser used features of the main functions are incompatible with the way that ℝ now works. One in particular is the way it handles nested mixed models, resulting in some of the automated processes having to be done by hand by understanding and interpreting the code. Finally, the code for MAANOVA, whilst open source, is very badly commented, making it hard to read, so where possible, better commenting has been used to make it easier for people to make changes in future.

Loading the Data

The data to be analysed would be presented to MAANOVA in the form of two text files. One would contain information about the gene names, their location on the array, the flag for that spot and the intensities for Cy3 and Cy5 on the array. The last three elements would be repeated for each array in the experiment, giving a table with many columns, making it impossible to view this data in Excel.

The second file contains information about each dye of each array and is called the design file. It is used to tell MAANOVA which sample is on which array and where. There is also information in this file that identifies the timepoint of each sample and which biological replicate of each timepoint the sample represents. This information becomes useful when trying to separate the data into groups for checking data quality or for identifying gene expression differences due to experimental factors.

The function which loads this data is called read.madata( ) but some small modifications were made to this function to allow it to handle complications in the design file as mentioned above regarding out of date code. Therefore, the new function name is ReadMAData( ) and an example of its use is shown here:

```
hri.raw <- ReadMAData("Full_catma.txt", designfile="Full_design.txt", metarow=1, metacol=2, row=3,
col=4, ID=5, pmt=6, spotflag=TRUE)
```

In this case, hri.raw is the raw data structure receiving the data. Full_catma.txt is the text file containing the intensity readings for each array. Full_design.txt is the text file containing the design of the experiment, identifying the details of each sample in the experiment. Metarow=1 identifies the first column of the first file as being the metarow of each spot. Similarly Metacol=2, Row=3 and Col=4 identify those columns as containing information about the positions of the spots on the arrays. ID=5 indicates that the gene IDs are in column 5 and PMT=6 indicates that the information about the intensities of spots starts from column 6 onwards. SpotFlag=TRUE tells the code to collect the array information in groups of three columns with the third being the flag for the spot. This is FALSE by default and would force the code to read the arrays in paired columns with no flag data.

The Raw Data Structure

The output object from the ReadMAData( ) function is a dataframe which has a number of sections in it. These represent the data that has been acquired from the text files used as input. The sections of this dataframe are as follows:

- n.dye – the number of dyes in the data.
- n.array – the number of arrays in the data.

- data – a matrix containing the intensities of the spots for all dyes in all arrays.
- flag – a matrix containing the flags for each spot across all arrays.
- metarow – a vector of the meta-rows for each spot.
- metacol – a vector of the meta-columns for each spot.
- row – a vector of the absolute row for each spot.
- col – a vector of the absolute column for each spot.
- ID – a vector of the gene IDs.
- testIDs – a vector of spots which should be considered test spots; identified by not being unique in the ID list.
- TransformMethod – a string showing the method of transformation applied to the data.
- design – a matrix containing the contents of the design text file.

Checking Data Quality

Checking the quality of the data before any analyses are performed is an often overlooked step in the process of analysing microarray data. However, it is arguably the most important stage, since it takes very little time to perform this stage in comparison, and gives strength to any conclusions that are drawn from the latter analyses. If, for example, a few arrays were handled badly during their manufacture or during the process of binding the samples to them, they may show significant changes in gene expression that are non-existent. By identifying these bad arrays early on, they can be excluded from the analysis. Furthermore, one biological replicate could show poor replication compared to the others, and this could be accounted for when performing the further analyses.

MAANOVA offers three functions for analysing the quality of the data; these are gridcheck( ), riplot( ) and arrayview( ). Each of these offer graphical interpretations of the data in such a form that anomalies will be demonstrated by dramatic deviation from the trend of the plots. A further two functions were required and produced since the current functions only look at arrays on an individual basis, and there was a need to look at variation between technical replicates and biological replicates as well. For this reason, techrepcheck( ) and biorepcheck( ) were also produced. All of the functions output PDF files and three of them use principal component analysis, so these functions are explained first.

Creating PDF Output

All the data checking functions output information in the form of graphics, and the original code of the functions outputs this to the screen. The screen is what is known as the default device and unless another device is specified, the screen automatically receives all the commands to draw graphics. In order to produce PDF files, a new device that writes PDF files needs to be implemented. There is a function built into ℝ called PDF( ) which takes arguments to define the file to be output and the size of the PDF file. This then generates a new device which becomes the default until it is closed again with the command dev.off( ) which closes the current device and returns to the default device (the screen). Dev.off( ) also closes the actual PDF file so that it can be opened by a reader.

In order to produce PDF output in the current functions, generally it was simply a case of opening the PDF device early on in the function’s code and then remembering to close it at the end. This is complicated slightly by the desire to have multiple PDF files (one per array for example), but as long as each instance of the device is closed before the next is opened with a different filename this didn’t prove to be a problem.

The PDF files that were output were generally dimensioned to have a width which was 66% of the height, allowing it to easily be printed on A4 without distortion. The dimensions are specified in inches and the size used in the majority of cases was 20” × 30”. In some cases, the best arrangement for the graphics were in a square shape, and in which case, the PDF was scaled to 20” × 20” which prints in the middle of an A4 sheet. The following is a simple example of how the PDF device was implemented in one of the functions:

```
pdf(file="Output/Array View Ratio Output.pdf", height=30, width=20,
    onefile=TRUE, title="Maanova ArrayViewPlus Output")

        # PLOT SOME THINGS HERE

dev.off( )
```

Principal Component Analysis

The data checking functions that output plots of correlation between the intensity of two dyes (GridCheck( ), TechRepCheck( ) and BioRepCheck( )) give clouds of data points which are hard to interpret by eye. In order to automate the process of identifying potentially bad data, Principal Component Analyses (PCA) are performed on the plots.

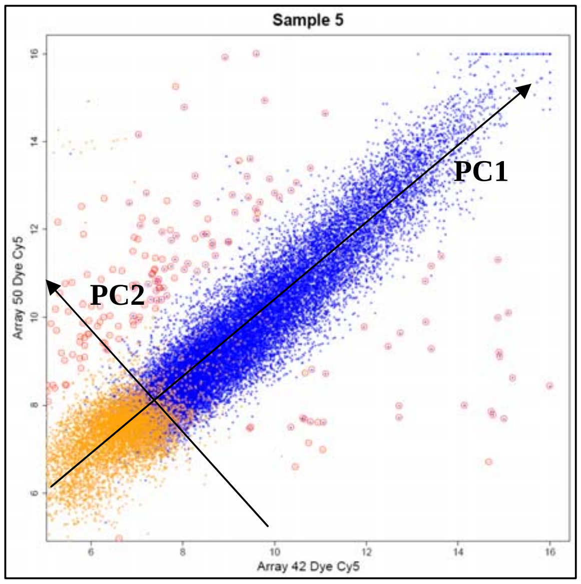


Figure 3 - Illustration of PC1 and PC2

ℝ has a built in function for performing Principal Component Analyses called prcomp( ). This takes the points being plotted in the form of a matrix with row names and returns data representing the same points on a new axis. The new x-axis represents the line of correlation through the data that gives maximum variation and the new y-axis is 90 degrees to this (Figure 3). The intersection of these two axes is centred around the mean of the original data. These axis are labelled PC1 and PC2 respectively. The data supplied would be in a straight line if the data were perfect and if this were the case, the variation due to PC1 would be 100% and the variation due to PC2 would be 0%. Therefore, the lesser the variation in the PC1 direction, the poorer the correlation and the greater the scatter. For visual effect, the points on the graph that are within the 5% tails of variation due to PC2 are circled red in the output of these functions. This gives a good idea of just how bad the scatter really is.

Due to the complexity of this procedure, the PCA feature was moved into its own function produced solely for the purpose of passing data to the prcomp( ) function built into ℝ. This is called perform2DPCA( ) and can be found in Appendix F. This function takes the data as a vector of X and a vector of Y points on the plot to be analysed. It manipulates these into a matrix suitable for the PCA in ℝ and passes them to the prcomp( ) function. The output is then analysed to find the line of best fit through the data and to generate a list of points shown in the graph against their scatter scores, calculated by looking at the value in the PC2 direction divided by the standard deviation of points in this direction. Points which are significantly scattered are placed into a file before doing a t-test on them to find out if they're significantly different from the mean of the data. The calling function is then returned a vector with many pieces of information about the PCA including the following:

- The variation in the PC1 direction.
- The standard deviation in the PC2 direction.
- The mean on the x axis.
- The mean on the y axis.
- The number of scattered points identified (outside the p-value threshold).
- The percentage this represents of the whole dataset.
- The equation of the line of best fit (the PC1 axis).
- The mean of the PC2 value of points with a positive PC2.
- The p-value of a t-test performed on these points in the PC2 direction.
- The mean of the PC2 value of points with a negative PC2.
- The p-value of a t-test performed on these points in the PC2 direction.

GridCheck( )

Syntax

```
gridcheckplus (rawdata, array1, array2, num.split, highlight.flag, flag.color, margin)

rawdata          – the object containing the raw microarray experiment data.
array1           - the array to compare dyes on.
array2           - the second array to compare with the first. (no longer needed)
num.split        - the number of pages to split each array across. (default = 2)
highlight.flag   - a Boolean expression indicating whether to plot highlighted spots in a
                   different colour. (default = TRUE)
flag.color       - the colour to use for highlighting spots. (default = “orange”)
margin           - the size of margins around plots. (default = c(4.6,5.1,3.1,1.1))
```

Information

The code for gridcheck( ) can be found in Appendix A alongside the old, relatively useless, output and the new output which is of higher clarity and is also accompanied by a lot of extra information to help automate the process of selecting potentially bad data.

The new gridcheck( ) function bares very little resemblance to the original function included with MAANOVA. The only part that remains unchanged is the unused second half which is for comparing dyes between arrays. However, the only purpose held for gridcheck( ) with the data being worked on during this project is to compare Cy3 against Cy5 for each array. Newly included features are the ability to output to PDF, improved arrangement of output plots and PCA analysis, as well as many text files showing the basis of the plots and ranking them for quality so that the identified plots can be checked by eye.

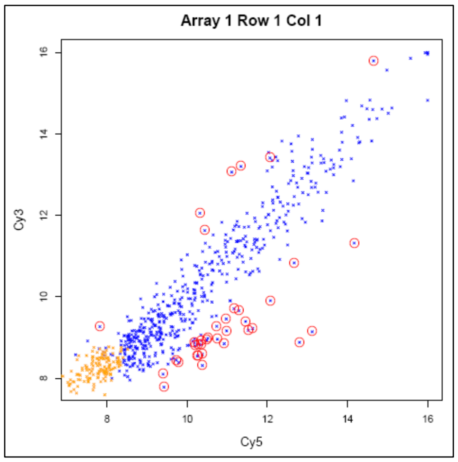


Figure 4 – Gridcheck( ) output for one meta grid

The purpose of the function is to produce plots which show how the intensities for Cy3 compare to the intensities for Cy5 on one array. For each array, 48 plots are produced; one for each meta-grid. These each show resemblance to the one shown in Figure 4. The orange points are spots which are flagged by the software which reads the scans of the arrays. The points with circles deviate from the correlation by more than 2 standard deviations in the PC2 direction.

Previously, all the plots for one array were output to the same screen, but when there are 4 meta-columns by 12 meta-rows and this is displayed on an A4 sheet, the plots become very distorted. For this reason, a decision was made to split one array across two pages. This means that each A4 page would contain 4 columns by 6 rows, which suits the aspect ratio of A4 paper. The code was also written to handle alternative numbers of splits, so if future data were to require 3 or 4 page splits, this would be easy to implement.

Further output comes in the form of tables of data:

- Arrays Summary.txt – shows the data returned by perform2DPCA( ) when the PCA is performed on the data for the whole array.
- Metagrid Summary.txt – shows the data returned by perform2DPCA( ) for each meta-grid of each array.
- Array PCAs – is a folder containing a text file for each array which has every gene in the array along with its coordinates on the plot and the level of scatter for that point.
- Meta Plot PCAs – is a folder containing a text file for every meta-grid of every array showing the genes in that metagrids along with its coordinates and the level of scatter for that point.

By taking the arrays or meta-grids with the highest levels of scatter (by sorting the tables of data in Excel) it is possible to identify data which show abnormal distributions. These can then be checked by eye in the PDF files and if necessary that data could be removed from the experiment.

RIPlot( )

Syntax

```
riplotplus (object, title, array, color, highlight.flag, flag.color, idx.highlight,
                  highlight.color, rep.connect, onScreen)

object          - the object containing the data for the experiment.
title           - the title of the plots. (auto-generated if left NULL)
array           - the number of the array to draw a plot for. (all are done if left NULL)
color           - the colour of the points in the RIPlot. (default = “blue”)
highlight.flag  - a Boolean expression indicating if flagged points should be
                  highlighted. (default = TRUE)
flag.color      - the colour of the flagged points. (default = “red”)
idx.highlight   - an index of further points to be highlighted. (if left NULL it is
                  ignored)
highlight.color - the colour to use for highlighting the indexed points.
rep.connect     - a Boolean expression indicating whether to draw lines between
                  replicates.  Only works on objects of class “madata”.  (default =
                  FALSE)
onScreen        - a Boolean expression indicating whether to output to the screen.  This
                  appears to be irrelevant now that the output goes to PDF.  (default =
                  TRUE)
```

Information

The new code for RIPlot( ) can be found in Appendix B along with examples of the old output and the new output generated by the rewritten code.

RIPlot stands for ratio-intensity plot. This is a well established method of checking the clarity of data from an individual array. Very simply, the function takes the information for both dyes of each spot on the array and gathers the ratio of these dyes and the combined intensity of them. This forms a table of paired values which are then plotted.

Originally this function output these plots to the screen which made the plots very hard to store. Furthermore, the old function only did a plot for each array, without breaking down the meta-grids. By rewriting the function, the output now goes to a PDF file and a second PDF file is also produced which contains the meta-grid RIPlots.

The shape of the plot resembles that shown in Figure 5 where the ordinate axis is the ratio and the abscissa shows the intensity. The red points are the flagged points. The blue points should show a clean funnel without any unusual variations from zero ratio. The hard lines of points towards the right of the plot are produced by the fact that if the intensity is maximum, the ratio has to be zero because the intensity is maximum on both channels. As the intensity is reduced from maximum, a maximum ratio still exists, and this is seen in the form of the hard lines of points towards the right of the plot.

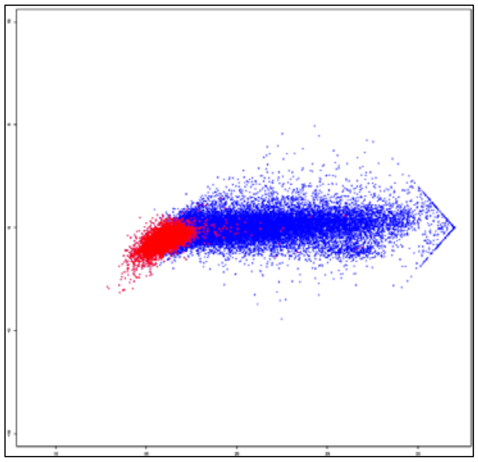


Figure 5 - The output of RIPlot( )

No automatic analysis is done on these plots as there are only as many as there are arrays, however, in case an array appears to have a strange pattern to it, the RIPlots performed on the individual meta-grids will allow a deeper look into the cause. These are stored in a separate PDF file with a similar layout to that of gridcheck (i.e. 2 pages per array with 4 columns by 6 rows on each page). This may identify if the problem on an array is due to a specific pin during its manufacture.

ArrayView( )

Syntax

```
arrayviewplus (object, ratio, array, colormap, onScreen)

object          - the object containing the data for the experiment.
ratio           - an object containing the ratios for the arrays in a matrix.  When left
                  NULL the ratios are calculated from the data.
array           - the number(s) of the array(s) to draw an array view for
colormap        - a vector containing the colours to use as a map for the ratio output.
                  If left blank, a default green to red gradient of colours are used.
onscreen        - a Boolean expression indicating whether to output to the screen.  This
                  appears to be irrelevant now that the output goes to PDF.  (default =
                  TRUE)
```

Information

The new code for ArrayView( ) can be found in Appendix C along with an example of the old output and the new output for both ratio (Figure 6) and intensity (Figure 7) of the arrays which is produced by the rewritten code.

The purpose of ArrayView( ) is to display the look of the original arrays on a larger scale with false colours so that any strange patterns in the arrays can be identified. This was previously done by displaying a graphic for each array on the screen which had a ratio of Cy3 to Cy5 represented by degrees of red to green squares for each gene. The problem with this is that the image cannot be easily stored and when the array is 312 spots tall like the CATMA arrays used in this experiment, it does not fit well onto the screen. For these reasons, the ratio output was forced into a PDF file and the graphics were split across two pages per array. This gives 104 by 156 spots per page, which matches the aspect ratio of A4 paper.

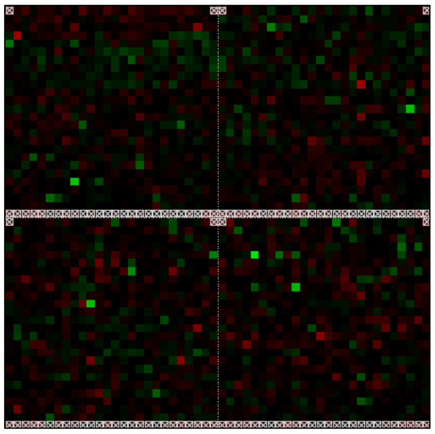


Figure 6 - Output from the Ratio function of ArrayView( )

One of the issues encountered was with the function used to draw the grid of squares; image( ). This is a built in function of ℝ and uses a colour map to define the colours in the grid. The supplied data for drawing the grid is used as a scale with the minimum and maximum values being the extent of the range for the colour map. When one value alone is extremely high or extremely low, this offsets the entire colour map, giving very dull images. More importantly, however, when the array is split into two, the two images produced are not comparable due to the fact they are both based on a different range of values. To overcome this, the minimum and maximum values had to be specified when drawing the grids. Two alternatives existed: either the minimum and maximum could be taken from the array being drawn, or from the entire dataset for the whole experiment. The former was selected, since the images were almost entirely black when the latter was selected due to the range being expanded yet further by the larger dataset having a more extreme minimum and maximum. This does mean that the arrays cannot be directly compared to one another, but the two halves of one array can be compared to one another.

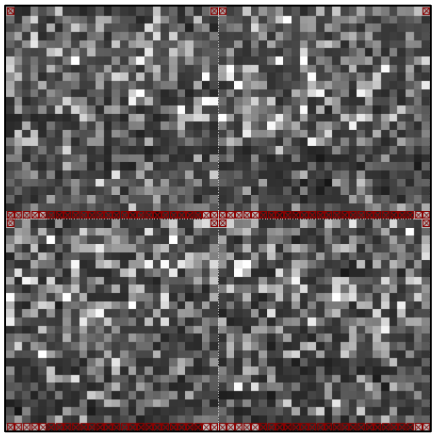


Figure 7 - Output from the Intensity function of ArrayView( )

Another new feature of the new code is the intensity plot. This is very similar to the grids drawn for the ratio of the dyes on the array, but in this instance, the greyscale image demonstrates the combined intensity of the spots. This could indicate if there was a problem with the binding of the sample to the slide since the intensity would vary across the array. The expected output is a random arrangement of intensities on the array resembling static on a television set.

Two final extra features of the new code, is identification of the test spots on the array and marking of the meta-grids with dotted lines. The test spots all have boxes with crosses placed over them so as not to distract the viewer of the output. These spots may show unusual patterns and this does not reflect a badly handled slide.

TechRepCheck( )

Syntax

```
techrepcheck (rawdata, array, sample, high.flag, flag.color, margin)

rawdata         - the object containing the raw microarray experiment data.
array           - the number of the array which you want to compare with its technical
                  replicates.  If left blank, all arrays are done.
sample          - the number of the sample which you want to compare with its technical
                  replicates.  If left blank, all samples are done.
high.flag       - a Boolean expression indicating whether flagged points should be
                  highlighted on the plots. (default = TRUE)
flag.color      - the colour to use for highlighting flagged points. (default = "orange")
margin          - the margin to leave around plots. (default = c(6.1,5.6,4.1,2.6))
```

Information

The code for TechRepCheck( ) can be found in Appendix D along with examples of the output produced.

TechRepCheck( ) is a brand new function, written from scratch, to deal with the issue of comparing technical replicates for similarity. This information can be used to identify where technical replicates are not good representations of one another. This might lead to a decision to remove certain technical replicates from the experiment to improve the overall quality of data.

The basic schema of the function's operation is that the data for the comparison between technical replicates are extracted from the rawdata and placed in a new object. The total number of possible crosses between these is calculated and a PDF output layout is created which crosses all the samples against all other samples. As shown in Figure 8, four technical replicates can be represented by six crosses and these can be arranged in a 3 × 3 grid. Technical replicate A is crossed with B, C and D in plots 1, 2 and 3 respectively. Replicate B is in plots 1, 4 and 5; replicate C is in plots 2, 4 and 6; and replicate D is in plots 3, 5 and 6. In all cases, two of the replicates are stained with Cy3 and two of the replicates are stained with Cy5. This gives four plots which have Cy3 on one axis and Cy5 on the other, whilst two of the plots will have the same dye on both axes. The code has been written so that plots 1 and 6 are the plots with the same dye whilst the remaining plots are the ones with different dyes on each axis.

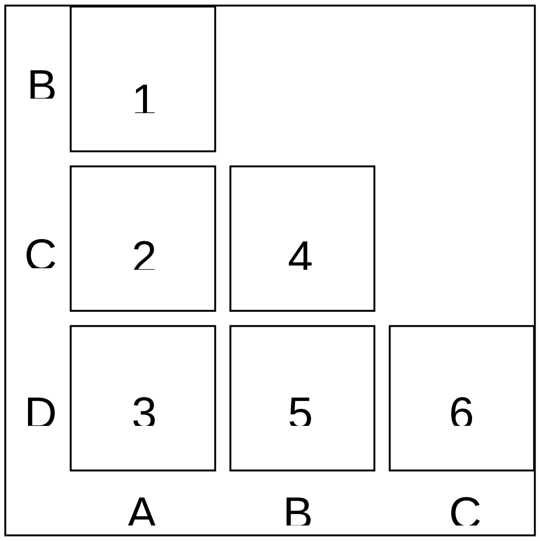


Figure 8 - Arrangement for four technical replicates

The code has been written to handle alternative numbers of replicates. If, for example, there were 5 technical replicates, 10 crosses would be performed automatically and the arrangement of plots would be adjusted into a 4 × 4 grid.

Each plot shows orange points where genes were flagged by the software that analysed the scans of the microarrays. In GridCheck( ) this was a simple process because the flagged points applied to both channels of the same array, but in TechRepCheck( ) the axes carry data from different arrays and so a decision was made to flag points if they are flagged on either of the arrays which are carrying the data in the plot. This means a larger proportion of points should be flagged in TechRepCheck( ) but this doesn't appear to present a problem as they are all clustered in the corner of each plot.

As well as producing graphics for each cross, Principal Component Analyses are performed for each cross. This leads to the graphics displaying red circles around points that are in the 5% tails of scatter in the PC2 direction, as seen in Figure 9. This also allows an assessment of technical replicate quality to be performed.

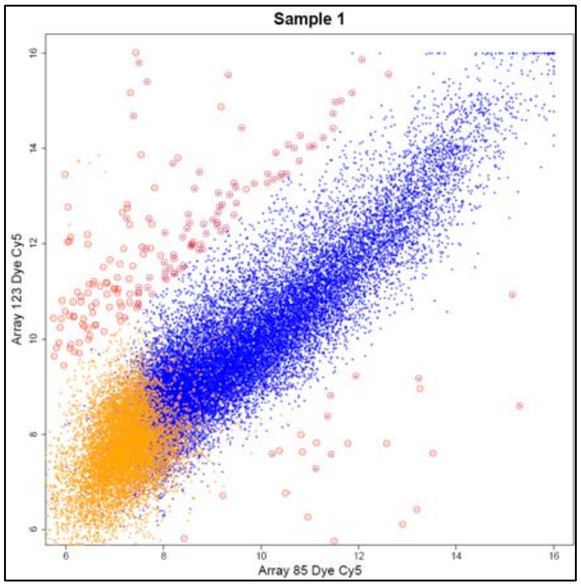


Figure 9 - Single plot from TechRepCheck( )

In order to assess the quality of each technical replicate, the variation in the data due to PC1 is considered. This is expressed as a percentage of the total variation, with the remainder being attributed to PC2. If two replicates are absolutely identical to each other, they will have a perfect correlation giving 100% of the variation in the PC1 direction and none to the PC2 direction. Therefore, if two replicates are poor comparisons to one another, the variation due to PC1 will be a low figure since there will be more scatter in the PC2 direction.

TechRepCheck( ) produces a table which shows this variation on a per replicate basis. This is done by taking the three crosses performed by each replicate and averaging the percentages of the variation due to PC1. If one replicate is significantly different to the others, it will show a very low value after this average. The other three should show homogenous levels of PC1 in comparison. An example of this output is shown in Appendix D and, as can be seen, the output has been automatically sorted by PC1 variation. In this case, Sample 72 on Array 20 is stained by Cy5 and shows the lowest PC1 variance and therefore the highest level of variation against its technical replicates in the whole experiment. By using information from other data checking routines, a decision may be made to remove this data from the experiment.

BioRepCheck( )

Syntax

```
biorepcheck (rawdata, array, high.flag, flag.color, margin)

rawdata         - the object containing the raw microarray experiment data.
array           - the number of the array which you want to compare with its biological
                  replicates.  If left blank, all arrays are done.
high.flag       - a Boolean expression indicating whether flagged points should be
                  highlighted on the plots. (default = TRUE)
flag.color      - the colour to use for highlighting flagged points. (default = “orange”)
margin          - the margin to leave around plots. (default = c(6.1,5.6,4.1,2.6))
```

Information

The code for BioRepCheck( ) can be found in Appendix E along with some of the output produced by it. The concepts behind BioRepCheck( ) are practically the same as those in TechRepCheck( ) but there is an added complication that each biological replicate is reproduced four times as technical replicates. This was overcome by averaging together the technical replicates, making the assumption that the technical replicates do not vary drastically enough to make this a problem. The outcome seems to be relatively good in this respect, with correlations of biological replicates being strong.

Other than the issue of averaging together technical replicates, the process is identical and plots are produced which are arranged in a similar fashion to TechRepCheck( ).

Once again, the question of which points should be flagged is raised. In this case, it was decided that since each axis is an average of four technical replicates, the points would be flagged if they were flagged in any of the eight arrays shown on the plot. This did not present problems as the flagged points once again clustered in the lower region of both axes.

The tabular output for BioRepCheck( ) is the same concept as the one used in TechRepCheck( ), but the details about each biological replicate are different. In this case, the timepoint which the biological replicate represents is listed along with a list of the arrays which make up the biological replicate. The name given to the biological replicate in the design file is also listed, but this was arbitrary in the first place and so does not mean much in this context.

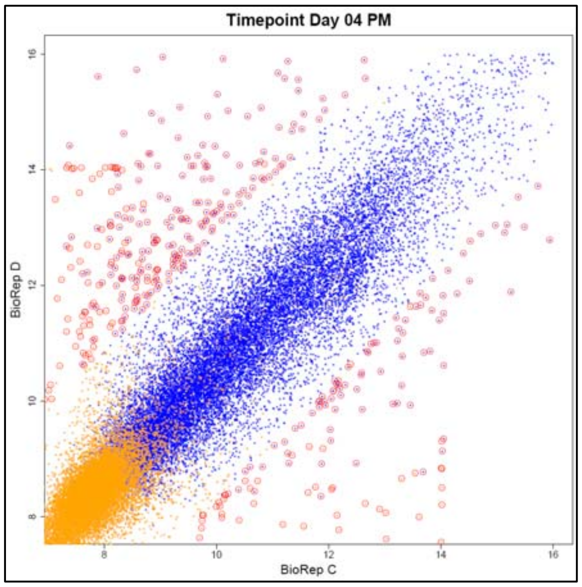


Figure 10 - Scatter due to biological replicate D on Day 4 PM

Taking the example tabular output shown in Appendix E, it can be seen that the biological replicates showing the greatest variation are the ones that are later in the time course. Day 11 is the last timepoint and by this time the plants have differentiated a lot depending on their growth conditions which will never be identical across the whole experiment. Therefore, later timepoints are expected to be present high on this list. However, the timepoint in the 4th day of observations should not be in 7th position on this list, and may represent a poor biological replicate, which is looking likely when observing the large amount of scatter in the plot for this biological replicate shown in Figure 10. Depending on the outcome of other data checking routines regarding the arrays that make up this biological replicate, this data may be excluded from the rest of the analysis.

Data Transformation

Once the data has been moderated successfully, using the data checking functions, it is necessary to transform the data. This is to eliminate variations throughout the data due to factors out of the control of the experimenter. One of the key undesirable variations is between dyes[9]. Cy3 and Cy5 have different binding affinities, with Cy3 having a greater affinity to bind to the cDNA than Cy5, meaning all samples labelled with Cy3 will appear to have a greater abundance than those with Cy5.

One other example of undesirable variation is between pin variation. However, this should be the same level of variation across all arrays. This will be discussed further shortly.

There are five transformation functions available in MAANOVA, namely Global LOESS, Regional LOESS, Shift, Linear-Log and Linear-Log Shift. The three which show most promise for transforming the data are LOESS, Shift and the Regional LOESS and so these are described in detail below.

LOESS

LOESS is a statistical method in which a line of best fit is applied to the data by means of weighted averaging of nearby points. It is the method suggested by Yang *et al.*[10] in work they were doing to find the best normalisation method for cDNA microarray data. The function built into ℝ is used for this, and takes input in the form of x and y coordinates in a matrix. The x-coordinates supplied are the intensity of both dyes for each spot on the array. The y-coordinates are the ratio between the Cy3 and Cy5 dyes for the same spots. LOESS calculates the line of best fit by returning

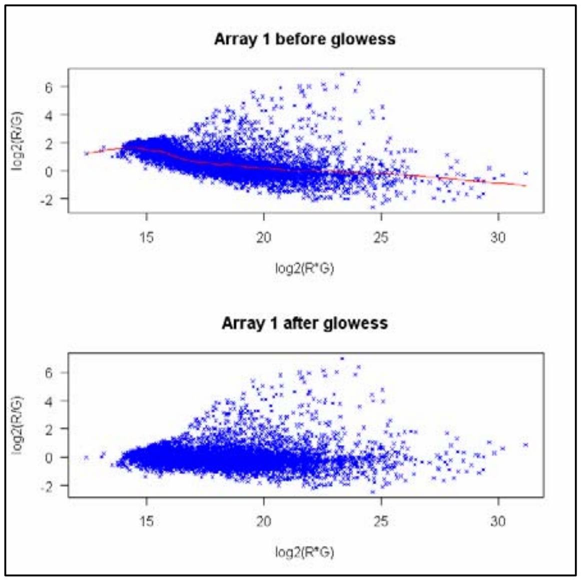


Figure 11 - Plots of the LOESS transformation

y-coordinates for each x-coordinate supplied, using a tricubic weighting algorithm so that heavy weighting is given to points with practically the same x-coordinate, but this weighting diminishes quickly with greater difference in x-value. The values returned are used to adjust the data by adding to one dye and taking it from the other, adjusting the ratio without adjusting the intensity, bringing the line of best fit to zero on the y-axis.

Figure 11 shows the LOESS transformation being applied to an array. The red line in the upper graph is the line of best fit as calculated by LOESS. The lower graph shows the data after the transformation which will be used for model fitting.

Shift

Shift( ) is a function that has been developed by the authors of MAANOVA. The reasoning behind this was that they believed the LOESS function was an overly complex routine to transform a relatively simple set of data.[11] They also believed that the span value that must be selected for LOESS could not be selected reliably, but this will be explained shortly.

Shift( ) works by modifying the ratio of all the points in the data set by the same offset. This is done by trying 100 different offsets, and for each calculating the sum of absolute deviations. Where this is at a minimum, the offset is transforming the data in the most effective way to reduce the undesirable variation in the data. This can be seen in Figure 12 where the top right plot shows the SADs at 100 different offsets. The minimum offset value (9.08 in this case) is the one used to offset the data. The plots on the left show the transformation results, and it is obvious that the data is a better fit to the intensity = 0 line than it was originally.

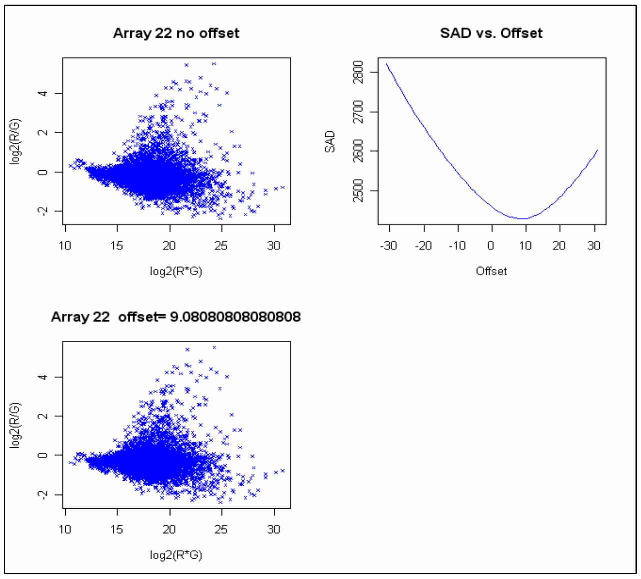


Figure 12 - Shift with a correctly identified SAD

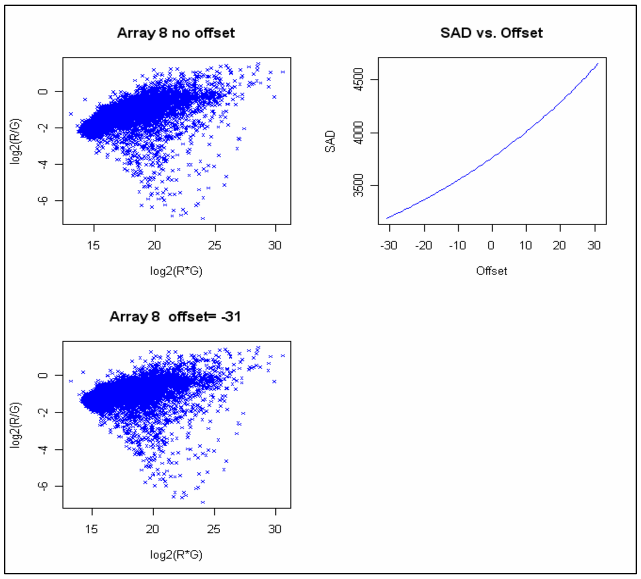


Figure 13 - Shift with a badly identified SAD

The main flaw with this function is that the minimum SAD is not always identified correctly. This is demonstrated in Figure 13 where the offset is only tested between -30 and 30 and since the minimum doesn't exist in this range, the value of -31 is used as this is at the lowest end of the curve. This value does start to offset the data correctly, but it is obvious that the data could be transformed to a greater extent.

For this reason, the function was modified so that instead of trying 100 offsets, only 5 are tried (starting with -30, -15, 0, 15 and 30). If the minimum is at -30 or 30, the frame is shifted so that the currently identified minimum is in the middle of the window. The process is then repeated until a minimum in the middle of the window is located. When this occurs, the range of the window is reduced around this minimum so that the true minimum can be effectively zoomed in upon. When a set threshold is passed, the function stops searching for the minimum SAD and transforms the data. Whilst this was a very efficient way of finding the minimum SAD, the data appeared to be over fitted and it was decided that this was not an appropriate way to transform the data.

Regional LOESS

Regional LOESS is a variation on the global LOESS function already mentioned. The only difference is that the regional LOESS takes into account that each sub-grid of each array is being produced by a different pin. For this reason, it is unreasonable to assume that all the data on an array can be used to compensate offset values by the weighting method employed by LOESS. Instead, each sub-grid has a regression put through it separately. Only data from the same sub-grid is considered for the weighting function and as such, 48 regressions are performed. These are poorly displayed in the output as they are all drawn as one line, giving a fluctuating zig zag as shown in the top plot of Figure 14. However, each regression is used to produce offsets, and one bad sub-grid will not pull the rest of the data during the transformation. The outcome is much more tidy as can be seen. This is the function of choice for the transformation stage due to its apparent accuracy.

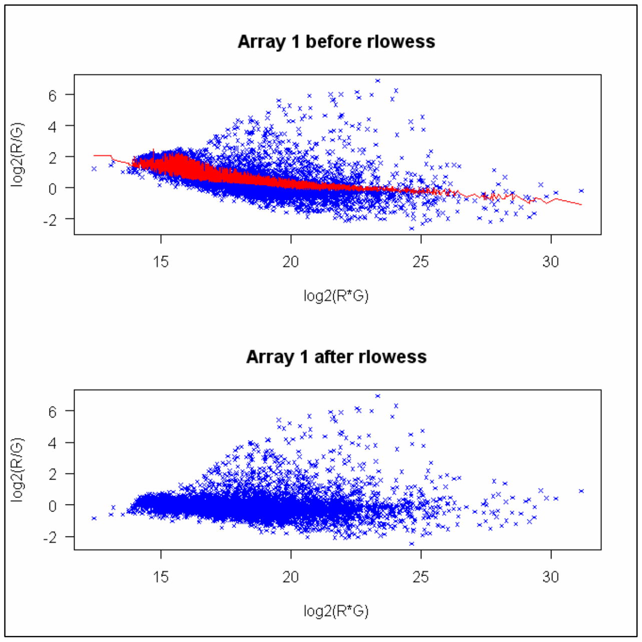


Figure 14 - Transformation from Regional LOESS

Choosing a Span Value

One drawback of the LOESS function is that is has many parameters, of which it is not obvious which values should be assigned to them. The parameter of particular interest is the span value. This is used by LOESS to adjust the range of the weighting across the data points when considering the exact offset to use for each point. The use of span is defined in the help files of ℝ and reads as follows:

The size of the neighbourhood is controlled by α (set by `span` or `enp.target`). For α < 1, the neighbourhood includes proportion α of the points, and these have tricubic weighting (proportional to $\left(1-\left(\frac{dist}{maxdist}\right)^3\right)^3$*)*

Where *dist* is the x-distance of the point causing the change to the offset and *maxdist* is the maximum distance of the points causing changes to offsets. For each point being offset, this is the furthest point within a range of points that make up $\text{span}\times 100\,\%$ of the total dataset. This is where the name *span* is derived.

Kerr *et al.*[12] explain that the use of a span value that is too small will result in bad transformation of the data and therefore not remove all the undesirable variation in the data. However, the opposite of this would be to use a value that is too large, and average the variation of the points together too much. This would mean that the transformation is too great and will remove desirable variation as well as undesirable variation. The dilemma is that every set of data probably needs a different span value for a perfect transformation, but there is no known way of calculating this value. A method was employed during the analysis of this data to work out an ideal span value, but this will be explained shortly, once the modelling of the data is complete, since the only indication of a good span value is to find out if the outcome is as desired after model fitting. In the mean time, the default value of 0.1 was used (indicating that the 10% closest points have an impact on each offset).

The syntax for transforming the data using the span value of 0.1 is as follows:

```
hri <- createData(hri.raw)
hri <- transform.madata(hri, method="rlowess", f=0.1, draw="off")
```

The output of the transformation is then stored in the variable *hri* for fitting a model.

Model Fitting

Once the data has been transformed, what remains should be data that only shows variations due to actual changes in gene expression. All variation due to experimental techniques has been removed and now it must be identified which variations are due to which parameters of the experimental design. By fitting a model to the data, this information becomes apparent. The model is effectively a way of telling ℝ which parameters were used in the experiment and then ℝ uses the design table from the raw data object to identify which arrays are associated with the values of each parameter.

To generate the model, a MAANOVA function called makeModel( ) has to be used. The syntax for this is as follows:

```
makeModel(data, design, formula, random=~1, covariate=~1)

data            - the transformed data to be used.
design          - if the experimental design is to be specified separately it can be
                  parsed here, otherwise the design table in data will be used.
formula         - a standard ℝ formula for the model in the form of ~param1 + param2 + …
random          - a list of the random terms in a mixed model.
covariate       - a list of the covariates in a mixed model.
```

This generates a matrix which indicates which arrays will contain variation for each possible value of each parameter. Once the model is generated, it can be fitted to the data which will identify the proportion of variation of each gene due to each parameter from the model. The syntax for the command for this process is:

```
fitmaanova(madata, mamodel, verbose=TRUE)

madata          - the transformated data to be used.
mamodel         - the output of the function above containing the model.
verbose         - indicates whether to output information every 100 genes about progress.
```

This command takes a very long time to complete on mixed models due to the complexity of the analysis and the sheer size of the data. On a regular 2GHz single processor desktop PC with 2 gigabytes of RAM this is in the region of 24 hours for the dataset being analysed in this project.

Fixed Model

In order to compare the fixed and mixed model effects, it is necessary to run both, and the fixed model is quicker, so this was done first. In the fixed model, all parameters are considered to be controlled by the experimenter. The syntax used in the model fitting functions were as follows:

```
model.fix <- makeModel(data=hri, formula=~Dye+Array+Sample)
anova.fix <- fitmaanova(hri, model.fix)
```

This effectively states that the model is to differentiate affects between different dyes, different arrays and different samples, assuming that all parameters were chosen by the experimenter rather than being parameters that may have random variation between the factors of the parameter. The output of this model fitting will be stored in the object *anova.fix*.

Mixed Model

The equivalent mixed model has a very similar syntax, with the addition of just one parameter in the first line:

```
model.mix <- makeModel(data=hri, formula=~Dye+Array+Sample, random=~Dye+Array)
anova.mix <- fitmaanova(hri, model.mix)
```

This time it can be seen that the *random* parameter has been specified in the makeModel function. This indicates to MAANOVA that the dye and array of each sample were not specifically selected by the experimenter, since the dye from slide to slide may be slightly different and the individual arrays may have a small amount of variation due to random chance. This is then accounted for when the model is fitted. The model is otherwise very similar to the fixed model output; which will be explained shortly.

Nested BioRep Model

The most useful model is a more complicated version of the mixed model. In this model, the individual timepoints are separated instead of just separating the samples. Doing this means that the 4 biological replicates of each biological replicate are not treated individually, and instead are incorporated into the output of the model. The syntax to do this is:

```
model.mix <- makeModel(data=hri, formula=~Dye+Array+TimePoint/BioRep, random=~Dye+Array)
anova.mix <- fitmaanova(hri, model.mix)
```

This basically says that the variation is due to dye differences, array differences and differences in the timepoints, but each timepoint is also made up of a number of biological replicates. It also says, as with the other mixed model, that the dye and array are not being selected by the experimenter and may show some random variations.

Selecting Significant Genes

The output from the model fitting is extensive (shown in Appendix G). By using the summary command it is possible to see which sections make up the fitted model. The three sections of most interest are S2, G and the matrices of the individual parameters (like TimePoint for example). The matrices with the individual parameters are of dimensions 32488 rows and in the case of TimePoint, 22 columns. The columns represent the factors of that parameter and the rows represent the spots on the array. The values in these matrices are relative expression values. Trends in these values will indicate increases or decreases in gene expression across the parameter.

S2 is another matrix and contains the remainder of the gene expression after removing all the other variations. This is an indication of variation in gene expression due to factors other than the parameters specified in the model. The first columns of this matrix are specifically related to the random terms whilst the last column is a residual for the fixed terms.

G is a vector as it is only one column. This contains a value for every gene, indicating how much has been removed from the expression of the genes across the entire dataset. This is done as a form of normalisation. By adding this value to the expression profiles identified by the parameter specific matrices and then also adding the mean of the column means across all arrays, the values of the expression will return to true $\log_2$ expression indicators. However, this process is not required for the type of analysis being performed here.

In order to pick significantly changing genes, they must be scored in some way for variation with time. In order to do this, the variance of each gene across all the timepoints is taken and this is divided by the last column of S2. Genes with high variation across the timepoints will get higher scores and by sorting the genes by these scores, a list of significant genes can be produced.

Ranking Different Spans and Numbers of Significant Genes

Several questions still exist about values that have been used as defaults throughout the processing of this data. Firstly, how is it possible to know that a span of 0.l is the right value to use? Secondly, how many genes are really significant; what score contributes the threshold for significance? Since neither of these questions can be answered directly in the time for this project, an alternative method was used that gives at least a rough estimate of the correct parameters to use.

A script was setup in ℝ that was designed to run the data through the last few stages of processing over and over with different span values for the regional LOESS transformation, and finally picking off different numbers of significant genes. The order of significance in which the genes were presented by doing this were then compared using a Spearman’s Rank of Correlation technique. This gives a percentage similarity between the compared data. The output of this type of analysis is summarised in Appendix H.

In the first two tables, all the comparisons can be seen, indicating the correlation of the two spans and the associated P-value. The trend is very obvious, with greater numbers of significant genes causing greater degrees of correlation and similar spans showing the greatest degrees of correlation.

Looking at each number of significant genes in turn, it can be seen that by 2000 genes, all the correlations are above 90% with the exception of 0.01 vs 1.00 which are expected to show higher degrees of dissimilarity anyway. With this information, 2000 significant genes were selected for this dataset.

Rearranging the data for 2000 significant genes gives the last two tables. The first makes it easier to see the correlations relevant to each span. The second shows the sums of correlations for each span value. The theory is that the span with the highest level of similarity to the others will offer the best data transformations without removing desirable variation as described above. The sums of correlations for 0.10 and 0.25 are very similar and would indicate that the true optimum is between these two spans. However, the default span is 0.10 and since the underlying reason for this choice is unknown, this is the span that was used throughout the analysis of the data to be used for clustering.

Using SplineCluster

SplineCluster is a piece of software written in C++ by Nicholas Heard as part of his thesis at Imperial College London. It takes the expression values of a set of genes (the top 2000 genes in the dataset for this study) over the time course and it looks for expressions that change in a similar pattern amongst these genes. The outcome is that the genes with similar expression patterns are placed into the same clusters and because similar expressions probably means co-regulation of those genes, it could be assumed that these genes are part of the same pathway in the cell.

SplineCluster requires two files as input: one has a vector list of time points relating to the data which will be entered. The second file is also a vector list, but has all the expression values for the first gene at each time point followed by a list of expression values for the second gene and so on until the list is $x \times y$ in length where x is the number of time points and y is the number of genes. In the case of this data set, there are 2000 significant genes and 22 time points so the file contains 44,000 values.

SplineCluster returns both a PDF file (found in appendix K) showing graphics of the clusters and a text document which is a tab delimited table indicating the cluster which a gene falls into and its expression values across the time course. SplineCluster doesn't know the names of the genes it is working with, and so two small scripts had to be written in $\mathbb{R}$. The first produces the files mentioned above for the input of SplineCluster and the second processes the tab delimited output file into a more useful file with gene names and annotations. Both scripts are shown in Appendix I.

Selecting a Prior Precision

When the SplineCluster program is executed, it can take a vast number of parameters to indicate how the clustering should be performed. Unfortunately, the documentation on these is "limited" to "non-existent" and yet one parameter appears to be very important. PriorPrecision indicates to SplineCluster how distinctly different genes should be to exist in different clusters. There appears to be no default, but the value can range from $1 > x > 0$ and higher values result in greater numbers of clusters. The value used in the past has always been 0.1 since this produces a greater number of clusters than the suggested value of 0.00001 on Nicholas Heard's website and it would be more convenient to separate similar genes than it would be to end up with two pathways in the same cluster. Therefore 0.1 was used during this process, producing 94 clusters.

Another parameter called normalisetargets can be set to 1 to have SplineCluster bring the mean of the expression profiles to zero and have the range of the amplitude of the profiles normalised so that clusters are easier to identify, but since the mean is being brought to zero as a part of preparing the input for SplineCluster and it is desired to leave the range of the expression profile unchanged, this parameter is left at zero.

Inferring Gene Purpose

Now that the genes are clustered, named and annotated, it is possible to start working out which pathways the clusters are a part of. This of course doesn't indicate their exact purpose, but gives a good idea of which pathways they may be a part of. This in turn will bring a greater understanding to which pathways are being affected by senescence in *Arabidopsis* and if enough genes in the pathways are annotated, it may be possible to identify the function of some genes without annotations.

GOStat

GOStat[13] is a bioinformatics tool used for identifying the pathways of genes based on their identifiers. In the case of this data, the AT code is known for almost every gene on the array. This code serves as a direct link to the gene and can be used to identify it in databases. GOStat uses databases of GO terms to find out the Gene Ontologies for the clusters. It then specifies how many genes in the cluster are found in a specific pathway of the cell. Where the proportion of genes in a cluster is significantly different to that of the entire genome for a specific pathway, the GO term is listed with a P-value of significance.

Results from using GOStat can be found in Appendix J. Where clusters are missing, nothing could be drawn from the GO terms present in the cluster.

Discussion and Conclusions

Discussion of GOStat Results

Many conclusions can be drawn from the GOStat results shown in Appendix J, but the most significant results are the ones with the larger numbers of genes. Smaller datasets will often show over-represented genes by chance alone, but for a significant number of genes for a particular pathway to exist in a larger dataset is unusual.

Some of the more interesting pathways to be showing up are those in clusters 8, 13, 16, 18, 36, 38, 42, 47, 76 and 85. A number of these are showing changes to the cell in terms of the cell cycle progression through S phase. This is the stage at which the DNA is replicated. Clusters 13, 18, 36, 38, 76 and 85 all show changes relating to this aspect of the cell. 13, 18, 36 and 38 show drops in the expression of proteins involving the cell cycle whilst 76 and 85 show increased expression of genes relating to increased progression through the S-phase. This may be the cell trying to avoid the S-phase of the cell cycle as they start to senesce in the later timepoints.

Other interesting responses are the ones relating to cell defense and response to parasite attack. This is seen in clusters 16, 42 and 47. In 16 the expression profile shows a decline over time and in 42 and 47 the expression profile shows an increase in expression, particularly in the later timepoints. This would indicate that the leaf is becoming more susceptible to parasite attack and the very significant representations in clusters 42 and 47 would indicate that the leaf is suddenly causing programmed cell death to the cells as senescence takes place. The increased immune responses may be a part of this process as the immune system is responsible for deciding which cells need to be destroyed. This would account for the brownish appearance to leaves that are undergoing senescence, since these patches will be the dead cells.

A number of the identified clusters, namely 8, 16, 29, 70, 74 and 76, are also showing diurnal rhythms. It would be interesting to study these in more detail by changing which data is extracted from the data after model fitting. These clusters appear to contain a lot of pathways relating to metabolism, the conversion of sugars and consumption of ATP which would be expected of diurnal rhythms since these processes are tailored particularly to the presence and absence of daylight.

Conclusions Drawn from Analysis of Data Checking

Whilst it is not the intention of this project to have a completely finished product in terms of the capabilities of the code, it is important to understand what the data checking functions offer to the user. It seems that the functions are now offering a much more comprehensive understanding of the data quality and most of the output is very obvious in its meaning, as explained in each of the sections above. However, one interesting point to note is that there appear to be constraints on most, but not all of the points in biorepcheck( ), meaning that almost all the point lie between $1 < x < 14$ and $1 < y < 14$ when both axis should show values up to a maximum of 16 (65,536 on a $\log_2$ scale). The reason behind this is the action of averaging 4 technical replicates together for each biological replicate. If one of the technical replicates is a value of maximum intensity (i.e. 65,536 or 2^{16}) and the other three are practically zero intensity (2^1) then the average will be $\frac{2^{16}}{4} = 2^{14}$. This is a reflection on the poor quality of some values between technical replicates. This is slightly worrying and should be addressed in future studies.

Possible Improvements

When the functions for data checking were produced, it was the intention that data could be removed from the final analysis if it would appear to be misleading when the data was fitted to a model. Unfortunately, during the time constraints of the project, it was not possible to analyse the data to this level and select what was to be removed. Two possible options exist:

Either it would be desirable for the software to pick out data that needs to be removed and remove it automatically or the software should present data that ought to be removed to the user and allow them to pick which data should stay and which should go. To some extent, the output produced by the functions written during this project already do give advice, but it would be nice if it were summarised and inter-function comparisons of output were done. This would require a lot more statistics and may be work for the future, but it would seem important that the user has ultimate control over the process, since computer logic doesn't always make the best decisions without knowing all the factors like expected inadequacies in the data.

Another improvement that would make this software better would be to find a method of calculating the unknown parameters with more accuracy. The one parameter that is still only understood by trial and error is the span value for the LOESS transformation. If this could be somehow deduced by the expected variability in the data, the transformation would make a much better job of only removing undesirable variations. This also applies to the prior precision parameter for SplineCluster. It would be good to get a better understanding of what this value is actually doing to the way the genes are clustered and this would allow the clusters to be more accurate, resulting in better output from GOStat.

Further Work

The function techrepcheck( ) is a very good way of comparing between technical replicates to try to identify bad readings on certain replicates. It can be seen, from the output generated by techrepcheck( ) that there are a number of points which are extreme outliers, and are circled in red on the extremities of the PC2 axis. These points represent spots on the arrays that are expressed very differently between technical replicates and would make an interesting study in how to reduce this level of technical variability in the future. This also, of course, applies to biological replicates which should show a little more variation than technical replicates, particularly in later timepoints, but the very extreme values may be of particular interest in future studies.

It was desirable, if not a little ambitious, to get to a stage during this project at which a model would have been produced for one or more of the pathways identified. This is a very large piece of work in itself, and would make a very good future study. Now that the pathways have been identified, they can be focussed upon in more detail. Of course, this study also brings light upon probable interacting genes by the clustering that was done and this may help in defining the probable functions of some of the genes lacking annotations which could then be proven in future work.

Overall this project brings a lot of new developments into the study of senescence and opens many doors to new studies which in turn may open more new doors as the picture is slowly unfolded.

References

[1] The Churchill Group in Maine, USA – http://www.jax.org/staff/churchill/labsite/

[2] Nicholas Heard, Imperial College London – http://stats.ma.ic.ac.uk/naheard/public_html/

[3] My Warwick page – http://www2.warwick.ac.uk/fac/sci/sbdtc/students/2006/stuart_mchattie/

[4] The CATMA database – http://www.catma.org/database/

[5] Crowe M.L. *et al.* (**2003**) CATMA: a complete *Arabidopsis* GST database. *Nucleic Acids Research*, **31**, pp.156-158

[6] R Development Core Team (**2006**). R: A language and environment for statistical computing. *R Foundation for Statistical Computing, Vienna, Austria.* ISBN 3-900051-07-0, URL http://www.R-project.org

[7] Hao Wu, with ideas from Gary Churchill, Katie Kerr and Xiangqin Cui (**2006**). MAANOVA: Tools for analyzing Micro Array experiments. *R package version 1.4.0*. http://www.jax.org/staff/churchill/labsite/software/Rmaanova/

[8] R/Maanova update history – http://www.jax.org/staff/churchill/labsite/software/Rmaanova/Rmaanova_Status.html

[9] Rosenzweig B.A. *et al.* (**2004**) Dye-Bias Correction in Dual Labelled cDNA Microarray Gene Expression Measurements. *Environmental Health Perspectives*, **112(4)**, pp. 480-487

[10] Yang Y.H. *et al.* (**2000**) Normalisation for cDNA Microarray Data. *Technical report 589, Department of Statistics, University of California.* http://www.stat.berkeley.edu/users/terry/zarray/Html/papersindex.html

[11] Kerr M.K. *et al.* (**2002**) Statistical Analysis of a Gene Expression Microarray Experiment with Replication *Statistica Sinica* **12**, pp. 203-217

[12] Kerr M.K. *et al.* (**2002**) Statistical Analysis of a Gene Expression Microarray Experiment with Replication *Statistica Sinica* **12**, pp. 203-217

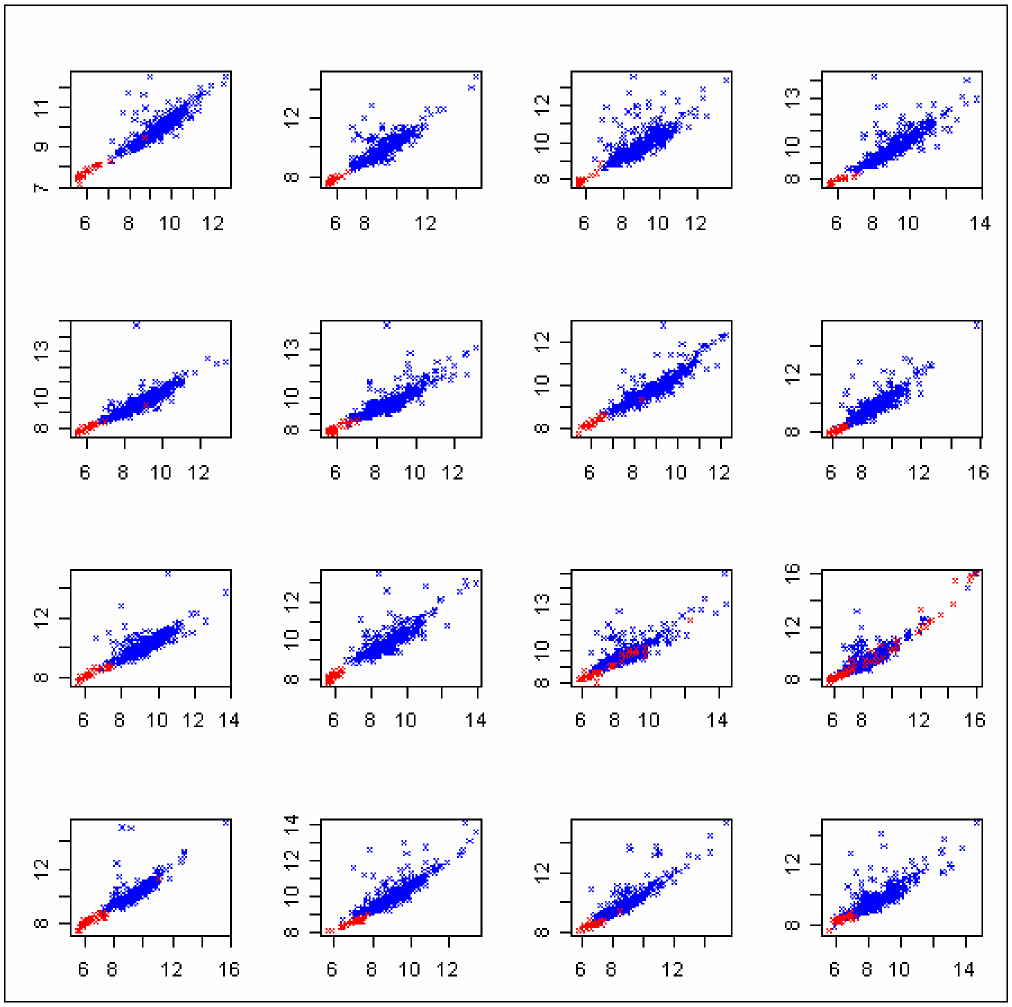
[13] The GOStat Search Form – http://gostat.wehi.edu.au/cgi-bin/goStat.pl

Appendix A – GridCheck( )

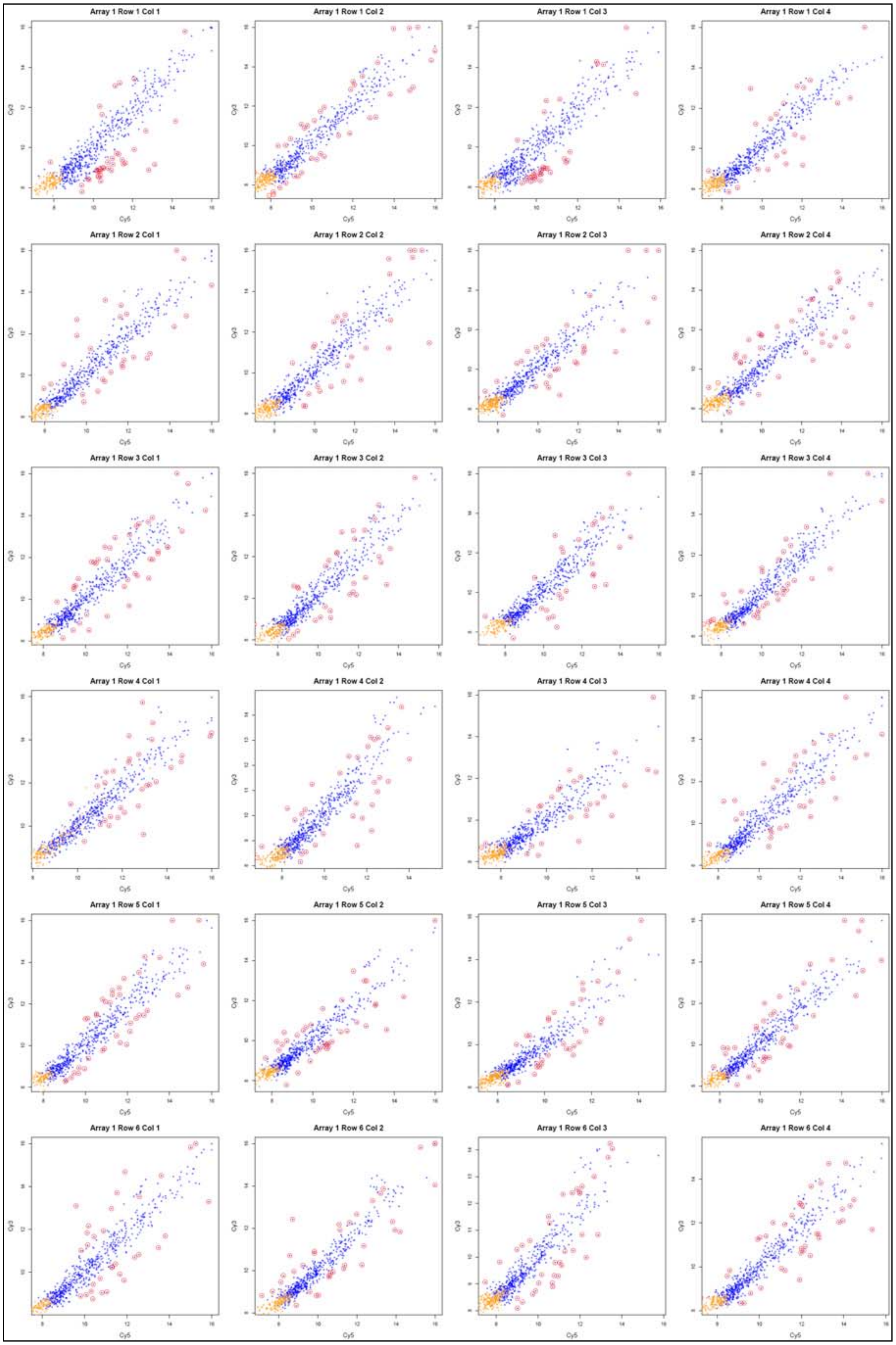
Code

```
################################################################################
#
# gridcheckplus.R
#
# copyright (c) 2002, Hao Wu and Gary A. Churchill, The Jackson Lab.
#
# written Nov, 2002
#
# Modified Dec, 2002 for mixed effect model
#
# Modified Apr, 2007 by Stuart McHattie to include Principal Components Analysis
#
# Licensed under the GNU General Public License version 2 (June, 1991)
#
# Part of the R/maanova package
#
#
# Parameters:
#
# rawdata         = The dataset in raw data format
# array1          = Integer indicating the array to analyse
# array2          = Integer indicating the array to compare with array1
# num.split       = Integer for number of pages to divide each array across
# highlight.flag  = Boolean indicating whether to highlight problematic points
# flag.color      = String expression of the colour of the problem points
# margin          = Bottom, Left, Top, Right margins for each graph
#
################################################################################

gridcheckplus <- function(rawdata, array1, array2, num.split=2,
                          highlight.flag=TRUE, flag.color="Orange",
                          margin=c(4.6,5.1,3.1,1.1))
{
  # Turn off graphics before and after the function
  graphics.off()
  on.exit(graphics.off())

  # Generate Output File Structure
  folders <- FALSE
  if (!file.exists("Output/Grid Check/Meta Plot PCAs")) {
    folders <- TRUE
    dir.create("Output/Grid Check/Meta Plot PCAs", recursive = TRUE)
  }
  if (!file.exists("Output/Grid Check/Array PCAs")) {
    folders <- TRUE
    dir.create("Output/Grid Check/Array PCAs", recursive = TRUE)
  }
    if (folders) cat("Output folders missing, autogenerating them\n")

  # check that the data format is correct
  if(class(rawdata) != "rawdata")
    stop("The first input variable is not an object of class rawdata.")

  # check number of dyes in the dataset are 2
  if(rawdata$n.dye != 2)
    stop("gridcheck only works for 2-dye arrays")

  # if there's no grid info, stop
  if( sum(c("metarow","metacol","row" ,"col") %in% names(rawdata)) != 4) {
    # rawdata contains the grid location information
    msg <- "Grid location information is incomplete in the input data object."
    msg <- paste(msg, "You cannot do grid checking!")
    stop(msg)
  }

  # get metarow, metacol and flag (pull data out of dataset)
  mrow <- rawdata$metarow
  mcol <- rawdata$metacol
  n.mrow <- max(mrow)
  n.mcol <- max(mcol)
  flag <- rawdata$flag

  # save old par parameters and setup new layout (reset graphics parameters at
  # the end)
  old.mar <- par("mar")
  old.las <- par("las")
  old.cmain <- par("cex.main")
  old.clab <- par("cex.lab")
  old.caxis <- par("cex.axis")
  on.exit(par(las=old.las, mar=old.mar, cex.main=old.cmain, cex.lab=old.clab,
              cex.axis=old.caxis), add=TRUE)

  par(las=1)

  # if the array2 parameter is missing, plot just for array1
  if(missing(array2)) {

    # if the array1 parameter is missing, plot all arrays consequtively
    if(missing(array1))
       array1 <- 1:rawdata$n.array

    # Close any graphics windows that may be left open
    graphics.off()

    # Prepare matrices to collect data about plots
    outputmat <- matrix("", ncol=16)
    colnames(outputmat) <- c("Array", "Row", "Column", "% PC1 Var", "PC2 StDev",
                             "x Mean", "y Mean", "Scattered", "% Scattered",
                             "Equation", "Upper PC1 Mean", "Upper Statistic",
                             "Upper P-Val", "Lower PC1 Mean", "Lower Statistic",
                             "Lower P-Val")
    arraysmat <- matrix("", ncol=14)
    colnames(arraysmat) <- c("Array", "% PC1 Var", "PC2 StDev", "x Mean",
                             "y Mean", "Scattered", "% Scattered", "Equation",
                             "Upper PC1 Mean", "Upper Statistic", "Upper P-Val",
                             "Lower PC1 Mean", "Lower Statistic", "Lower P-Val")

    # loop for all selected arrays
    for (i.array in array1) {

    # Open a new PDF file ready for the graphs
    pdf(file=paste("Output/Grid Check/Array ", i.array, ".pdf", sep=""),
        height=30, width=20, onefile=TRUE, title="Maanova GridCheckPlus Output")

      # loop for each split in the array plots
      for (i.split in 1:num.split) {

        # setup the layout into a grid of plots and set the margins
        layout(matrix(1:((n.mrow*n.mcol)/num.split), (n.mrow/num.split), n.mcol,
                      byrow=TRUE))
        par(mar=margin, cex.main=1.5, cex.lab=1.2, cex.axis=1)

        # loop through the metarows and metacols within the array and split num
        for(i in 1+(n.mrow/num.split*(i.split-1)):(n.mrow/num.split*i.split-1)) {
          for(j in 1:n.mcol) {

               # create an index of rows containing the required data
            idx <- which((mrow==i) & (mcol==j))

            # highlight the flagged spot (if any).  If highlighting is requested
            # (TRUE by default) and the flag data is available
            if(highlight.flag & !is.null(flag))
              # create a structure called high that contains TRUE and FALSE for
              # data points indicating which of the data points are problematic
              high <- flag[idx,i.array]!=0
            else
              # Otherwise set high to NULL
              high <- NULL

            # draw a plot of the log2 data, selecting data from rows idx and
            # columns i.array*2 drawing in colour blue, as crosses (pch=4) at
            # size 0.5
            plot(log2(rawdata$data[idx[!high],i.array*2-1]),
                 log2(rawdata$data[idx[!high],i.array*2]),
                 col="blue", pch=4, cex=0.5, xlab="Cy5", ylab="Cy3")
            title(paste("Array", i.array, "Row", i, "Col", j))


            # plot points where high==TRUE, pulling coordinates from array
            # data again as above, in colour "flag.color", overlapping the
            # previously plotted points
            points(log2(rawdata$data[idx[high],i.array*2-1]),
                   log2(rawdata$data[idx[high],i.array*2]),
                   col=flag.color, pch=4, cex=0.5)
            # Export the list of points to a file
            A <- sprintf("%03d", i.array)
            R <- sprintf("%02d", i)
            C <- sprintf("%02d", j)
            outfile <- paste("Output/Grid Check/Meta Plot PCAs/A", A, "-R", R, "-C",
                            C, ".txt", sep="")

            # Call function to perform PCA on the data
            PCAout <- perform2DPCA(xData = log2(rawdata$data[idx,i.array*2-1]),
                                yData = log2(rawdata$data[idx,i.array*2]),
                                IDs = rawdata$ID[idx], testIDs=rawdata$testIDs,
                                sig = 1.96, outfile = outfile)

            # Insert line into summary output matrix
            outputmat <- rbind(outputmat, c(i.array, i, j, PCAout))
          }
        }
      }

      # Shut the PDF file
      dev.off()

      # Do PCA on whole array:-

      A <- sprintf("%03d", i.array)
      outfile <- paste("Output/Grid Check/Array PCAs/Array ", A, ".txt", sep="")

      PCAout <- perform2DPCA(xData = log2(rawdata$data[,i.array*2-1]),
                             yData = log2(rawdata$data[,i.array*2]),
                             IDs = rawdata$ID, testIDs=rawdata$testIDs,
                             sig = 3.2906, outfile=outfile, plotpoints = FALSE)

      # Insert line into summary output matrix
      arraysmat <- rbind(arraysmat, c(i.array, PCAout))

    }
    # Remove blank rows from arrays
    tmp <- colnames(arraysmat)
    arraysmat <- arraysmat[arraysmat[,1] != "",]
    outputmat <- outputmat[outputmat[,1] != "",]

    # If only one array, convert output to matrix again
    if (length(array1) == 1) {
      arraysmat <- matrix(arraysmat, 1, length(arraysmat))
      colnames(arraysmat) <- tmp
    }

    # Output the summaries to files
    write.table(outputmat, file = "Output/Grid Check/Metagrid Summary.txt",
                sep="\t", row.names=FALSE)
    write.table(arraysmat, file = "Output/Grid Check/Arrays Summary.txt",
                sep="\t", row.names=FALSE)
  }




  # if array2 number was specified
  else {

    # have array 2, compare the same sample for array 1 and array 2
    if(missing(array1))
      stop("Miss the first array number")
    if((length(array1)!=1) | (length(array2)!=1) )
      stop("Both array1 and array2 must be an integer")

    # get the sample ids for array 1 and array 2 from design
    if(is.null(rawdata$design))
      stop("No experimental design information in rawdata. Cannot do grid check on two arrays.")
    sample1 <- rawdata$design$Sample[c(array1*2-1, array1*2)]
    sample2 <- rawdata$design$Sample[c(array2*2-1, array2*2)]
    if(length(intersect(sample1, sample2)) == 0)
      stop(paste("No common sample in array", array1, "and array", array2,
                 "Cannot do grid check"))
    # start plot
    nplot <- 0
    # get the data for two arrays
    data1 <- rawdata$data[,c(array1*2-1, array1*2)]
    data2 <- rawdata$data[,c(array2*2-1, array2*2)]
    for(i.array1 in 1:2) {
      for(i.array2 in 1:2) {
        if(sample1[i.array1] == sample2[i.array2]) {
          nplot <- nplot + 1
          if(nplot!=1) {
            # open a window on screen
            get(getOption("device"))()
#             if(.Platform$GUI == "AQUA")
#               quartz()
#             else
#               x11()
          }

          # setup the layout and margin
          layout(matrix(1:(n.mrow*n.mcol), n.mrow, n.mcol, byrow=TRUE))
          par(mar=margin)
          for(i in 1:n.mrow) {
            for(j in 1:n.mcol) {
              idx <- which((mrow==i) & (mcol==j))
              plot(log2(data1[idx, i.array1]), log2(data2[idx, i.array2]),
                   col="blue", pch=4, cex=0.5, xlab="", ylab="")
              # highlight the flagged spot (if any)
              if(highlight.flag & !is.null(flag)) {
                high <- sumrow(flag[idx, c(array1, array2)])!=0
                points(log2(data1[idx[high], i.array1]), log2(data2[idx[high], i.array2]),
                       col=flag.color, pch=4, cex=0.5)
              }
            }
          }
        }
      }
    }
  }
}
```

Old Output



New Graphical Output



New Tabular Output

PCA Scattered Points

Gene Name	X-axis	Y-axis	Scatter
CATMA1b03855	10.31628153	12.05562137	3.562043829
CATMA4a06850	9.419960178	7.781359714	-3.750619326
CATMA1b17930	13.11991379	9.142107057	-7.683690069
CATMA5A47280	10.37937837	8.312882955	-4.396096806
CATMA5A56080	12.07831762	9.906890596	-4.171278396
CATMA5A42170	11.52503135	9.177419538	-4.68527192
…	…	…	…

PCA Summary

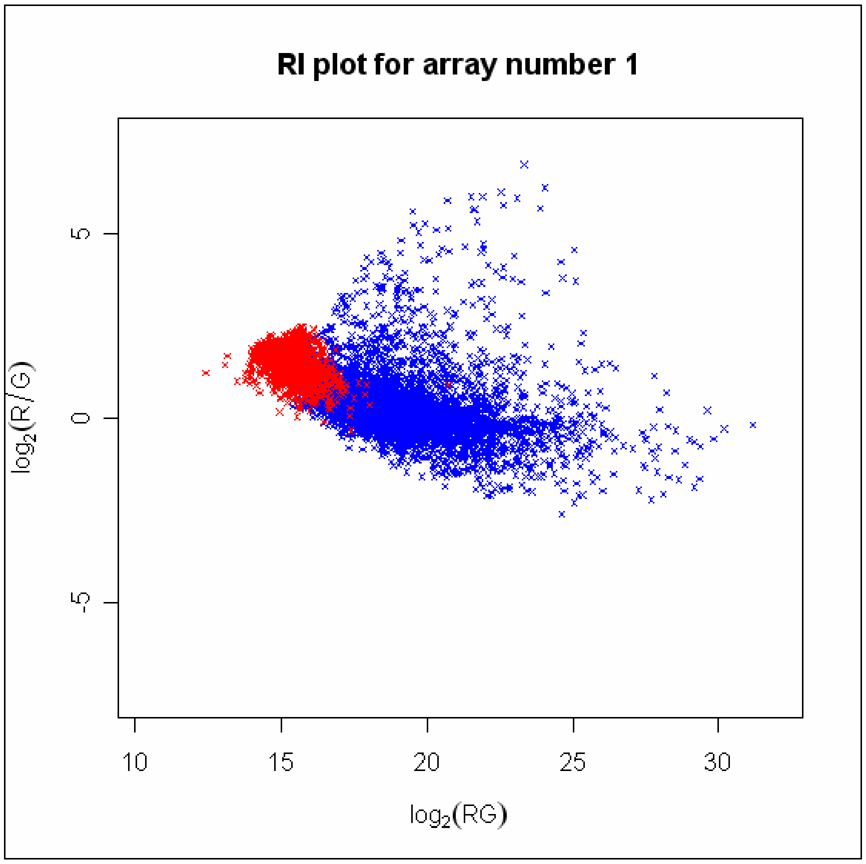
Array	% PC1 Var	PC2 StDev	x Mean	y Mean	Scattered	% Scattered	Equation	Upper PC1 Mean	Upper P-Val	Lower PC1 Mean	Lower P-Val
1	97.93	0.359146	9.831894	9.931787	331	1.06	y = -1.518 + 1.143 x	-3.62029	8.16E-24	-2.15884	1.51E-36
2	97.79	0.376477	9.850252	10.06035	363	1.17	y = -1.717 + 1.15 x	2.596093	4.98E-39	2.912319	2.33E-30
3	91.08	0.721153	10.22231	9.674846	446	1.43	y = -2.25 + 1.289 x	3.545836	1.70E-101	3.42198	2.46E-64
4	97.88	0.396246	9.854662	10.07607	212	0.68	y = -1.437 + 1.121 x	2.312182	1.22E-24	2.075496	5.31E-07
5	95.76	0.507442	9.162092	9.698013	218	0.7	y = -1.884 + 1.139 x	2.833152	6.81E-19	2.904623	4.43E-24
…	…	…	…	…	…	…	…	…	…	…	…

Appendix B – RIPlot( )

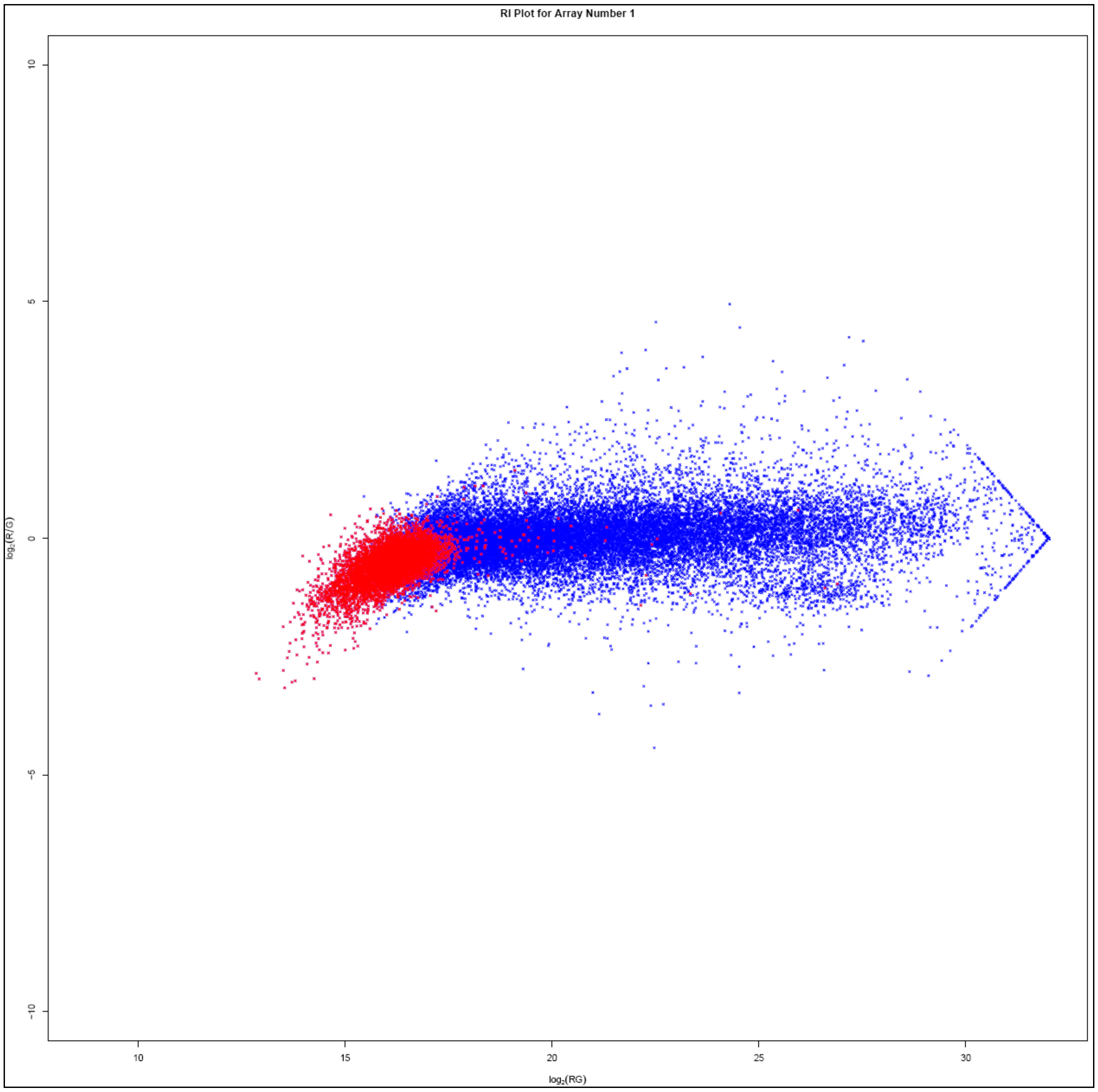
Code

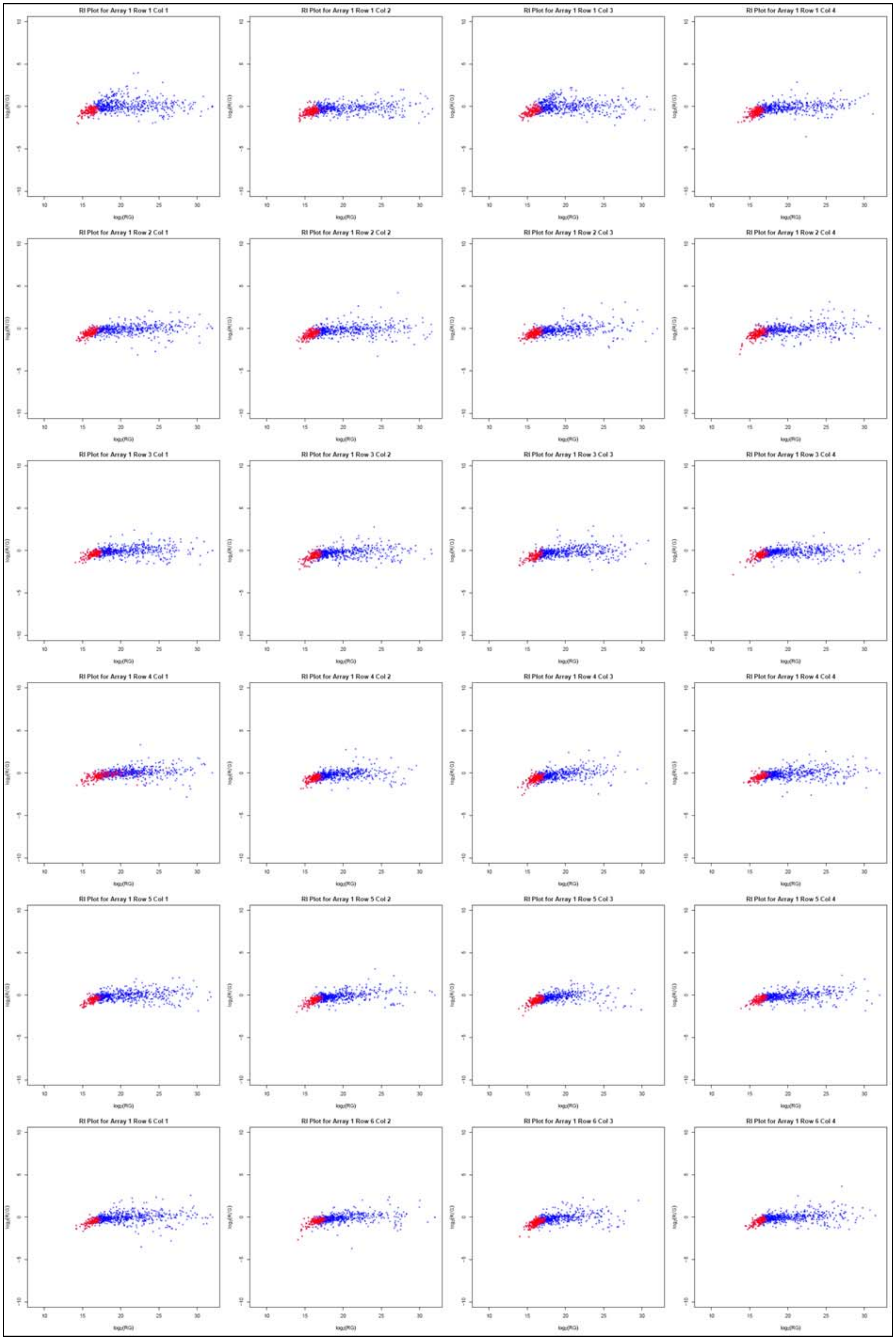
```
########################################################################
#
# riplotplus.R
#
# copyright (c) 2001, Hao Wu and Gary A. Churchill, The Jackson Lab.
#
# written Nov, 2001
# Modified Nov, 2002
# Licensed under the GNU General Public License version 2 (June, 1991)
#
# Modified Apr, 2007 by Stuart McHattie
#
# Part of the R/maanova package
#
#
########################################################################

riplotplus <-
  function(object, title, array, color="blue", highlight.flag=TRUE,
           flag.color="Red", idx.highlight, highlight.color="Green",
           rep.connect=FALSE, onScreen=TRUE)
{
  # Turn off graphics before and after the function
  graphics.off()
  on.exit(graphics.off())

  # Generate Output File Structure
  if (!file.exists("Output")) {
    cat("Output folders missing, autogenerating them\n")
    dir.create("Output", recursive = TRUE)
  }

  # Check that data only has two dyes
  if(object$n.dye != 2)
    stop("riplot works for 2-dye array only")

  if( !(class(object) %in% c("madata", "rawdata")) )
    stop("The input variable must be an object of madata or rawdata")

  # Decide whether the data needs log2 applied
  if( class(object) == "madata" ) # if this is madata
    x <- object$data
  else if( class(object) == "rawdata" ) # if this is rawdata
    x <- log2(object$data)
  else
    stop("The input variable must be an object of rawdata or madata")

  # Extract the two dimensions of the matrix (should be 2 of them)
  x.dim <- dim(x)

  # Put the spot flags from the data into a variable called flag
  flag <- object$flag

  # calculate R (ratio) and I (intensity)
  # For R subtract Cy3 from Cy5
  # For I add Cy3 to Cy5
  # Remember these are log2 already so this is equivalent to divide and multiply
  R <- x[,seq(1,x.dim[2],2)] - x[,seq(2,x.dim[2],2)]
  I <- x[,seq(1,x.dim[2],2)] + x[,seq(2,x.dim[2],2)]

  if(!is.matrix(R)) # if R is a vector, not a matrix
    R <- matrix(R, length(R),1) # convert R to a matrix
  # do same thing to I
  if(!is.matrix(I))
    I <- matrix(I, length(I),1)

  # Assign the highest absolute value of R to tmp
  tmp <- max(abs(R));

  # Set the title of the graphs or set the default one
  if(missing(title)){
    title <- NULL
    for(i in 1:object$n.array)
      title[i] <- paste("RI Plot for Array Number",i)
  }

  # Now draw the figures
  # First set all arrays to be drawn if none is specified
  if(missing(array))
    array <- 1:object$n.array



  # Part One: Draw by array (later by metagrid)
  # Open a PDF for the Output
  pdf(file="Output/RI Plot Arrays.pdf", height=20, width=20, onefile=TRUE,
      title="Maanova RIPlotPlus Output")

  # Cycle through all the arrays
  for (i in array) {

    # Draw a plot of R against I where R is log2(R*G) and I is log2(R/G)
    plot( I[,i], R[,i], xlim=c(min(I),max(I)), ylim=c(-tmp,tmp),
         xlab=expression(log[2](R*G)), ylab=expression(log[2](R/G)),
         col=color, pch=4, cex=0.5,
         main=title[i])

    # If highlight is given, redraw those points in col=highlight.color
    # Green by default
    if(!missing(idx.highlight)) {
      if(class(object) == "madata") {
        idx.gene <- matrix(1:(object$n.gene*object$n.rep),
                     object$n.rep, object$n.gene)
        high <- as.vector(idx.gene[,idx.highlight])
      }
      else
        high <- idx.highlight
      points(I[high,i], R[high,i], col=highlight.color, pch=4, cex=0.5)
    }

    # Highlight the flagged spot (if any) in flag.color (Red by default)
    if(highlight.flag & !is.null(flag)) {
      high <- flag[,i]!=0
      points(I[high,i], R[high,i], col=flag.color, pch=4, cex=0.5)
    }

    # If the data is in the madata format, join the dots between replicates
    if( rep.connect & (class(object)=="madata") ) {
      if(object$n.rep!=1) {
        # connect the dots between replicates (if any)
        idx.rep <- as.vector(repmat(t(1:object$n.gene), object$n.rep,1))
        for(j in 1:object$n.gene) {
          x <- I[idx.rep==j,i]
          y <- R[idx.rep==j,i]
          lines(x,y, type="l", col="grey")
        }
      }
    }
  }
  # Finish Loop for Arrays
  # Close PDF File
  dev.off()



  # Step Two:  Draw individual graphs for each metagrid item
  # Get the Metagrid information from the object
  mrow <- object$metarow
  mcol <- object$metacol
  n.mrow <- max(mrow)
  n.mcol <- max(mcol)

  # Set the number of pages to split arrays across
  n.split = 2

  # Now make another PDF file for the metagrids (like for gridcheck)
  pdf(file="Output/RI Plot Metagrids.pdf", height=30, width=20, onefile=TRUE,
      title="Maanova RIPlotPlus Output")
  layout(matrix(1:((n.mrow*n.mcol)/n.split), (n.mrow/n.split), n.mcol, byrow=TRUE))
  par(mar=c(5.1,4.1,2.1,1.1))

  # Loop by metacols within metarows within page splits within arrays
  for (i.array in array) {
    for (i.split in 1:n.split) {
      for(i.mrow in 1+(n.mrow/n.split*(i.split-1)):(n.mrow/n.split*i.split-1)) {
        for(i.mcol in 1:n.mcol) {

          # Generate a list of matrix rows for the given metagrid
          idx.metagrid <- which((mrow==i.mrow) & (mcol==i.mcol))

          # Plot the graph for this metagrid
          plot(I[idx.metagrid,i.array], R[idx.metagrid,i.array],
               xlim=c(min(I),max(I)), ylim=c(-tmp,tmp),
               xlab=expression(log[2](R*G)), ylab=expression(log[2](R/G)),
               col=color, pch=4, cex=0.5,
               main=paste("RI Plot for Array", i.array, "Row", i.mrow, "Col", i.mcol))

          # If highlight is given, redraw those points as before
          if(!missing(idx.highlight)) {
            if(class(object) == "madata") {
              idx.gene <- matrix(1:(object$n.gene*object$n.rep),
                                 object$n.rep, object$n.gene)
              high <- as.vector(idx.gene[,idx.highlight])
            }
            else
              high <- idx.highlight
            points(I[high,i.array], R[high,i.array], col=highlight.color, pch=4, cex=0.5)
          }

          # Highlight the flagged spots (if any)
          if(highlight.flag & !is.null(flag)) {
            high <- flag[idx.metagrid,i.array]!=0
            points(I[idx.metagrid[high],i.array], R[idx.metagrid[high],i.array],
                   col=flag.color, pch=4, cex=0.5)
          }
        }
      }
    }
  }

  # Close the PDF File
  dev.off()
}
```

Old Output



New Output


RI Plot for Array Number 1
$\log_2(R/G)$
10
5
0
-5
-10
$\log_2(RG)$
10
15
20
25
30

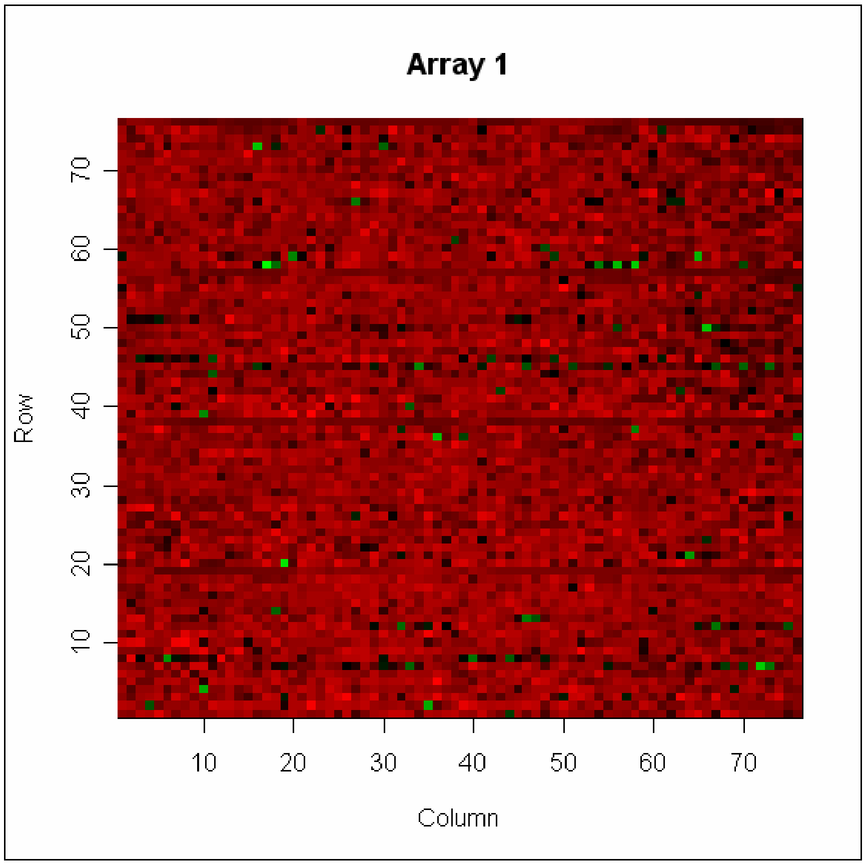

RI Plot for Array 1 Row 1 Col 1
RI Plot for Array 1 Row 1 Col 2
RI Plot for Array 1 Row 1 Col 3
RI Plot for Array 1 Row 1 Col 4
RI Plot for Array 1 Row 2 Col 1
RI Plot for Array 1 Row 2 Col 2
RI Plot for Array 1 Row 2 Col 3
RI Plot for Array 1 Row 2 Col 4
RI Plot for Array 1 Row 3 Col 1
RI Plot for Array 1 Row 3 Col 2
RI Plot for Array 1 Row 3 Col 3
RI Plot for Array 1 Row 3 Col 4
RI Plot for Array 1 Row 4 Col 1
RI Plot for Array 1 Row 4 Col 2
RI Plot for Array 1 Row 4 Col 3
RI Plot for Array 1 Row 4 Col 4
RI Plot for Array 1 Row 5 Col 1
RI Plot for Array 1 Row 5 Col 2
RI Plot for Array 1 Row 5 Col 3
RI Plot for Array 1 Row 5 Col 4
RI Plot for Array 1 Row 6 Col 1
RI Plot for Array 1 Row 6 Col 2
RI Plot for Array 1 Row 6 Col 3
RI Plot for Array 1 Row 6 Col 4
log2(R/G)
log2(RG)

Appendix C – ArrayView( )

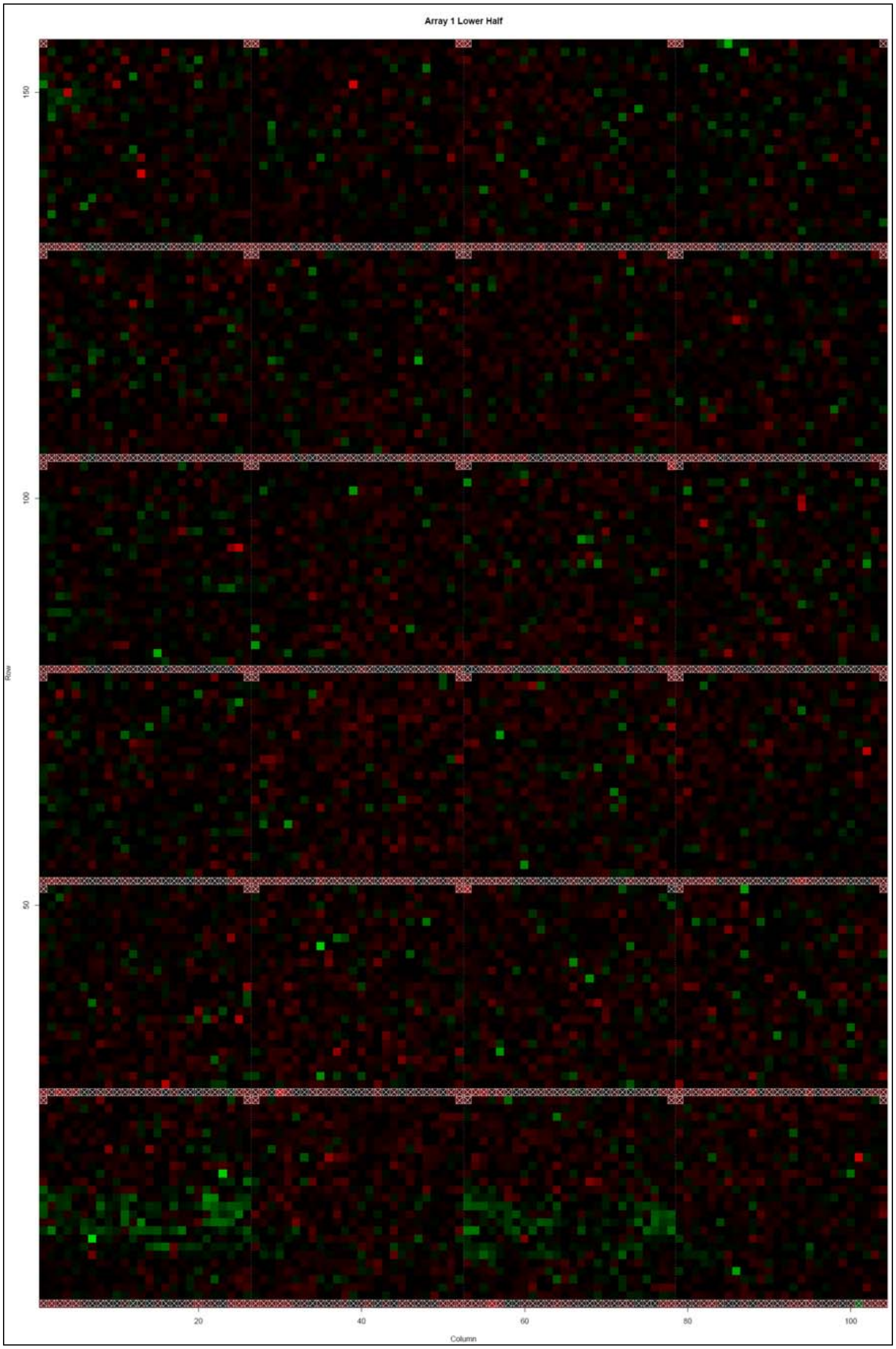
Code

```
########################################################################
#
# arrayviewplus.R
#
# copyright (c) 2001, Hao Wu and Gary A. Churchill, The Jackson Lab.
# written Nov, 2001
# Licensed under the GNU General Public License version 2 (June, 1991)
#
# Modified Apr, 2007 by Stuart McHattie
#
# Part of the R/maanova package
#
# View the spatial patter of the logratios for a 2-dye array
#
########################################################################

arrayviewplus <-
  function(object, ratio, array, colormap, onScreen=TRUE, ...)
{
  # Turn off graphics before and after the function
  graphics.off()
  on.exit(graphics.off())

  # Generate Output File Structure
  if (!file.exists("Output")) {
    cat("Output folders missing, autogenerating them\n")
    dir.create("Output", recursive = TRUE)
  }

  # local variables
  ndye <- object$n.dye

  # stop if there's no grid information
  if(sum(c("metarow","metacol","row" ,"col") %in% names(object)) != 4)
    stop("The grid location information is incomplete in the input data object")

  # stop if the input object is not madata or rawdata
  if(!(class(object) %in% c("madata","rawdata")) )
    stop("The input variable must be an object of rawdata or madata")

  # if object is madata and reps were collapsed, cannot do arrayview
  if(class(object)=="madata")
    if (object$collapse==TRUE)
      stop("The replicates were collapsed. Arrayview is unavailable.")

  # If no arrays are specified, do all arrays
  if(missing(array))
    array <- 1:object$n.array

  if(missing(ratio)) { # if ratio is not provided
    if(ndye == 1) {
      ratio <- object$data
    }
    else if(ndye == 2) {
      cat("No input ratio, make.ratio is being called.\n")

      # Generate a matrix with rows being different spots and columns being
      # different arrays containing the ratio of Cy3 to Cy5 for each spot
      ratio <- make.ratio(object)

      # Let variable total.spot be the number of rows
      total.spot <- dim(ratio)[[1]]

      # Let variable n.array be the number of columns
      n.array <- dim(ratio)[[2]]
    }
    else {
      stop("arrayview only works for 1 or 2 dye array at this time")
    }
  }
  else{ # ratio is provided, check if it's valid
    if(is.vector(ratio)) { # ratio is a vector
      total.spot <- length(ratio)
      n.array <- 1
    }
    else if(is.matrix(ratio)) { # ratio is a matrix
      total.spot <- dim(ratio)[[1]]
      n.array <- dim(ratio)[[2]]
    }
    # list of arrays
    array <- 1:n.array
    if(total.spot != length(object$row))
      stop("Number of elements in ratio doesn't match row record.")
  }

  # construct data.grid
  n.row <- max(object$row)
  n.col <- max(object$col)
  grow <- object$row + n.row*(object$metarow-1)
  gcol <- object$col + n.col*(object$metacol-1)
  n.grow <- max(grow)
  n.gcol <- max(gcol)
  data.grid <- matrix(rep(0,n.grow*n.gcol), n.grow, n.gcol)

  # assign default color map
  if(missing(colormap)) {
    r <- c(31:0,rep(0,32))/31
    g <- c(rep(0,32),0:31)/31
    b <- rep(0,64)
    colormap <- NULL
    for(i in 1:64)
      colormap[i] <- rgb(r=r[i],g=g[i],b=b[i])
  }

  # Make a list of test spot coordinates
  # Create an index of their position in the data
  idx.test <- which(object$ID %in% c("RBCS", "82B10", ""))
  # Create a new matrix to hold the coordinates
  t.spots <- matrix(rep(0,length(idx.test) * 2), ncol=2, nrow=length(idx.test))
  # Use grow and gcol to fill the matrix with the coordinates of the test spots
  for (idx in 1:length(idx.test)) {
    t.spots[idx,1] <- grow[idx.test[idx]]
    t.spots[idx,2] <- gcol[idx.test[idx]]
  }
  # Create filters for spots that are in the upper and lower half of the array
  hiset <- t.spots[,1] > n.grow / 2
  loset <- t.spots[,1] <= n.grow / 2

  # By now, ratio contains the ratio intensity of spots (from 0 red to 1 green)
  # for all arrays, Colormap contains 64 RGB values in which the first 32 are
  # bright red to blackand the last 32 are black to bright green
  # Grow and Gcol hold the row and coords for each spot in the form of a vector
  # Now the data.grid must be filled with the ratios and displayed

  # Open a PDF file for the RATIO output
  pdf(file="Output/Array View Ratio Output.pdf", height=30, width=20,
      onefile=TRUE, title="Maanova ArrayViewPlus Output")

  for(i in array) { # loop for all arrays
    for(j in 1:total.spot) { #loop over all spots
      if(is.matrix(ratio)) # if ratio is a matrix
        data.grid[grow[j], gcol[j]] <- ratio[j,i]
      else # ratio is a vector
        data.grid[grow[j], gcol[j]] <- ratio[j]
    }

    # Calculate normalisation for the images
    zmin <- min(data.grid)
    zmax <- max(data.grid)

    #Two images are generated, top and bottom half of array

    image(1:n.gcol, (n.grow / 2 + 1):n.grow,
          t(data.grid[(n.grow / 2 + 1):n.grow,]), ylab="Row", xlab="Column",
          col=colormap, main=paste("Array", i, "Upper Half"), zlim=c(zmin,zmax))
    # Plot boxes over the test spots
    points(t.spots[hiset,2], t.spots[hiset,1], col="light gray", pch=7, cex=2)
    # Plot a metagrid over the whole image
    grid(nx = max(object$metacol), ny = max(object$metarow)/2)


    image(1:n.gcol, 1:(n.grow / 2), t(data.grid[1:(n.grow / 2),]), ylab="Row",
          xlab="Column", col=colormap, main=paste("Array", i, "Lower Half"),
          zlim=c(zmin,zmax))
    # Plot boxes over the test spots
    points(t.spots[loset,2], t.spots[loset,1], col="light gray", pch=7, cex=2)
    # Plot a metagrid over the whole image
    grid(nx = max(object$metacol), ny = max(object$metarow)/2)
  }

  # Close PDF File
  dev.off()



  # For the second part, the datagrid must be filled again, but this time with
  # the overall intensity (both channels) of the spots.  These are stored in
  # the variable 'intensity'

  # Assign intensity values to the variable 'intensity'
  intensity <- matrix(rep(0, n.array * total.spot), nrow = total.spot,
                      ncol = n.array)
  for (i.array in 1:n.array) {
    intensity[,i.array] <- log2(object$data[,i.array * 2 - 1]) +
                           log2(object$data[,i.array * 2])

    # Scale so that highest intensity is 255
    intensity[,i.array] <- intensity[,i.array] / max(intensity[,i.array]) * 255
  }

  # Reconstruct the data grid to ensure the values are zero again
  data.grid <- matrix(rep(0,n.grow*n.gcol), n.grow, n.gcol)

  # Clear the colormap array and refill with 255 ascending shades of grey and
  # finally white for maximum intensity
  colormap <- NULL
  for(i in 1:256)
    colormap[i] <- rgb(i / 256, i / 256, i / 256)

  # Open a PDF file for the INTENSITY output
  pdf(file="Output/Array View Intensity Output.pdf", height=30, width=20,
      onefile=TRUE, title="Maanova ArrayViewPlus Output")

  for(i in array) { # loop for all arrays
    for(j in 1:total.spot) { #loop over all spots
      if(is.matrix(intensity)) # if ratio is a matrix
        data.grid[grow[j], gcol[j]] <- intensity[j,i]
      else # ratio is a vector
        data.grid[grow[j], gcol[j]] <- intensity[j]
    }

    # Calculate normalisation for the images
    zmin <- min(data.grid)
    zmax <- max(data.grid)

    #Two images are generated, top and bottom half of array

    image(1:n.gcol, (n.grow / 2 + 1):n.grow,
          t(data.grid[(n.grow / 2 + 1):n.grow,]), ylab="Row", xlab="Column",
          col=colormap, main=paste("Array", i, "Upper Half"),
          zlim=c(zmin, zmax))
    # Plot boxes over the test spots
    points(t.spots[hiset,2], t.spots[hiset,1], col="dark red", pch=7, cex=2)
    # Plot a metagrid over the whole image
    grid(nx = max(object$metacol), ny = max(object$metarow)/2)


    image(1:n.gcol, 1:(n.grow / 2), t(data.grid[1:(n.grow / 2),]), ylab="Row",
          xlab="Column", col=colormap, main=paste("Array", i, "Lower Half"),
          zlim=c(zmin, zmax))
    # Plot boxes over the test spots
    points(t.spots[loset,2], t.spots[loset,1], col="dark red", pch=7, cex=2)
    # Plot a metagrid over the whole image
    grid(nx = max(object$metacol), ny = max(object$metarow)/2)
  }

  # Close PDF File
  dev.off()
}
```

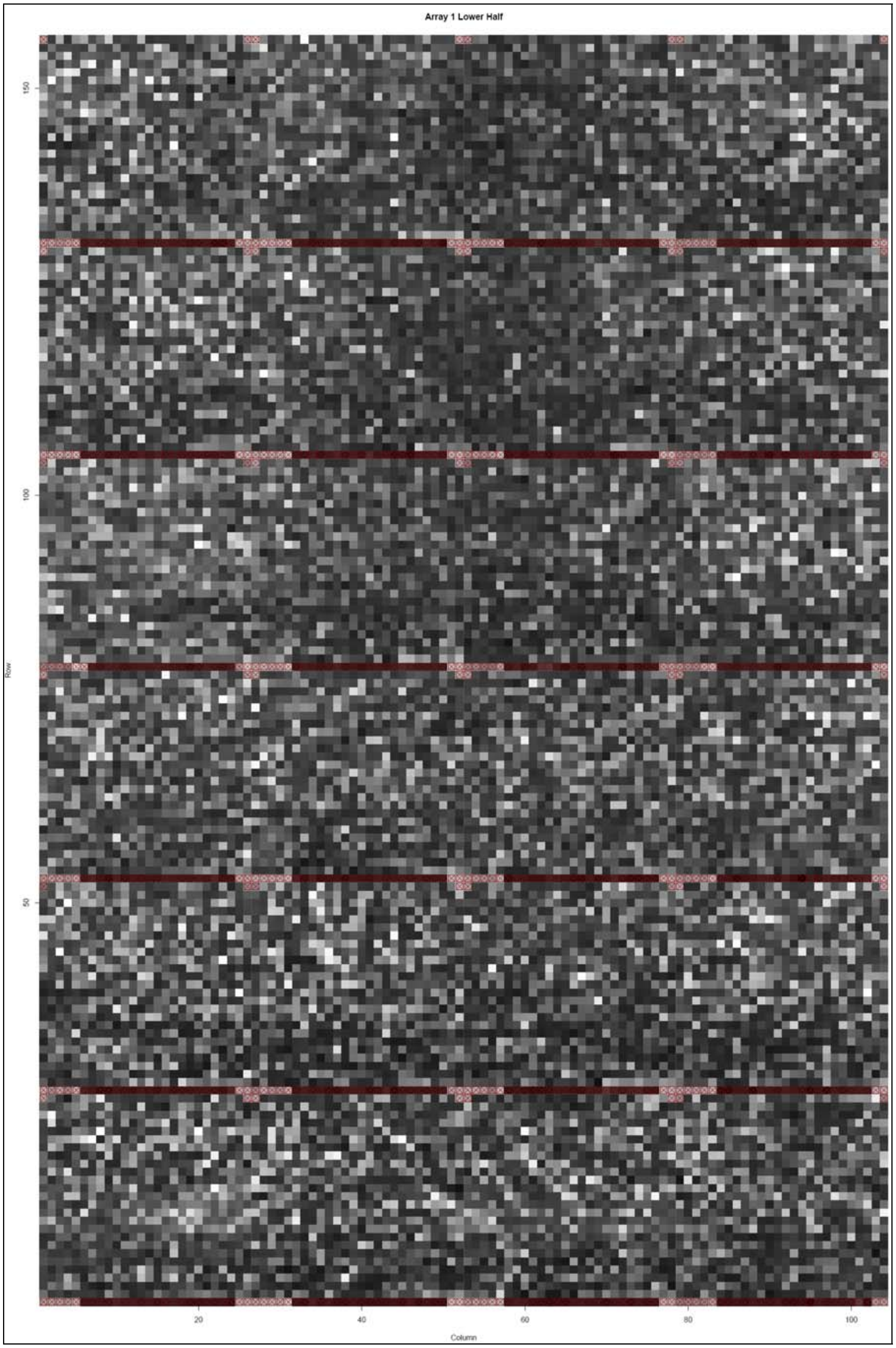
Old Output



New Ratio Output



New Intensity Output

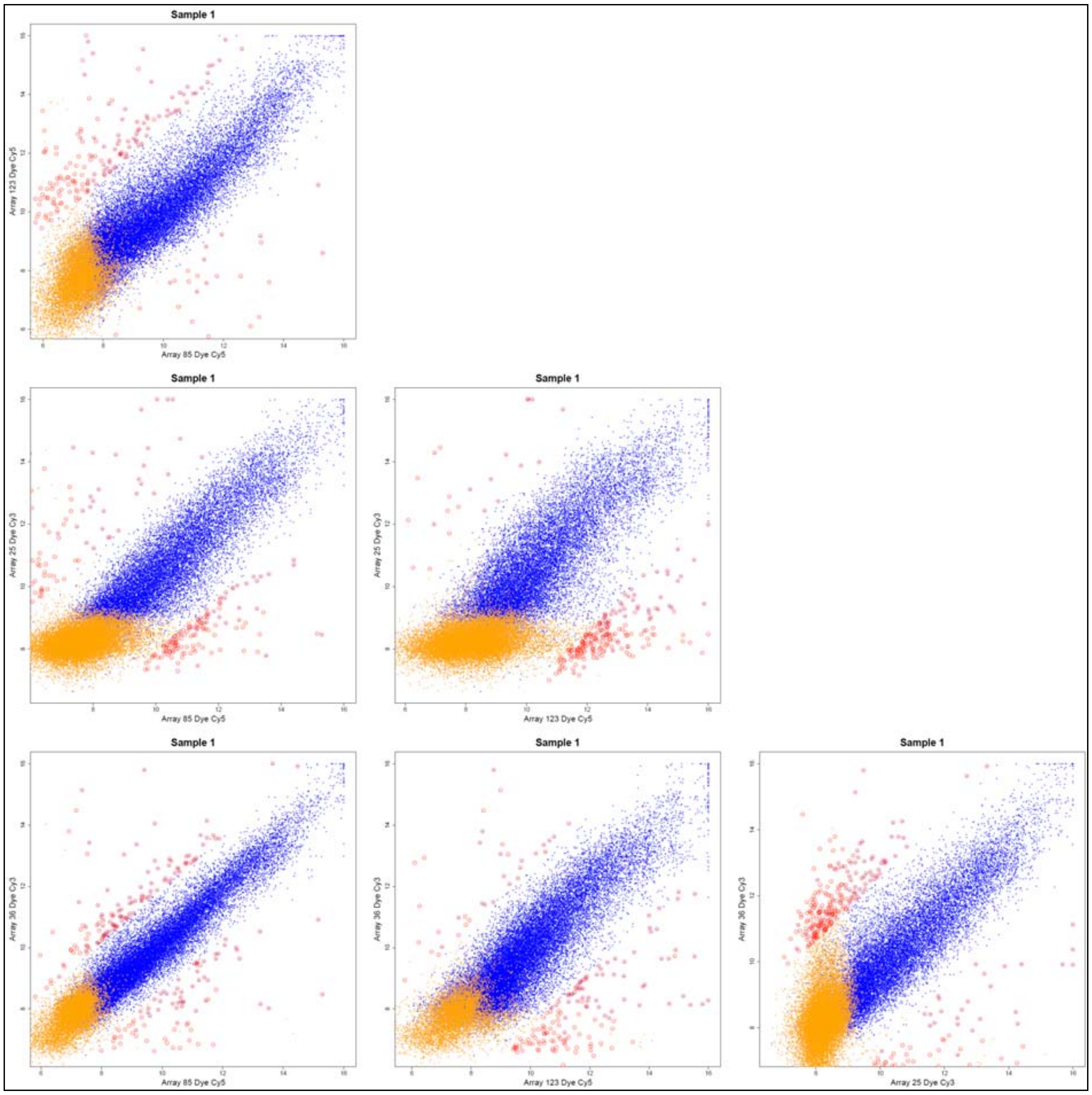


Appendix D – TechRepCheck( )

Code

```
###############################################################################
#
# techrepcheck.R
#
# adapted from gridcheck.R : Part of the R/maanova package
#
# copyright (c) 2007, Stuart McHattie
#
# written Apr, 2007
#
# Licensed under the GNU General Public License version 2 (June, 1991)
#
# Part of the R/maanova package
#
###############################################################################

techrepcheck <- function(rawdata, array, sample, high.flag=TRUE,
                         flag.color="Orange", margin=c(6.1,5.6,4.1,2.6))
{
  # Turn off graphics before and after the function
  graphics.off()
  on.exit(graphics.off())

  # Generate Output File Structure
  if (!file.exists("Output/Tech Reps/Scattered Points")) {
    cat("Output folders missing, autogenerating them\n")
    dir.create("Output/Tech Reps/Scattered Points", recursive = TRUE)
  }

  # Check that the data format is correct
  if(class(rawdata) != "rawdata")
    stop("The first input variable is not an object of class rawdata.")

  # Check number of dyes in the dataset are 2
  if(rawdata$n.dye != 2)
    stop("TechRepCheck only works for 2-dye arrays")

  # Get flag data (pull data out of dataset)
  flag <- rawdata$flag

  # Extract dye names
  dyes <- unique(rawdata$design$Dye)

  # save old par parameters and setup new layout (reset graphics parameters at
  # the end of the function)
  old.mar <- par("mar")
  old.las <- par("las")
  old.cmain <- par("cex.main")
  old.clab <- par("cex.lab")
  old.caxis <- par("cex.axis")
  on.exit(par(las=old.las, mar=old.mar, cex.main=old.cmain, cex.lab=old.clab,
              cex.axis=old.caxis), add=TRUE)
  par(las=1)

  #Create a matrix to store the summary data in
  outputmat <- matrix("", ncol=18)
  colnames(outputmat) <- c("Sample", "x Array", "x Dye", "y Array",
                           "y Dye", "% PC1 Var", "PC2 StDev",
                           "x Mean", "y Mean", "Scattered",
                           "% Scattered", "Equation", "Upper PC1 Mean",
                           "Upper Statistic", "Upper P-Val", "Lower PC1 Mean",
                           "Lower Statistic", "Lower P-Val")

  # If a sample has been specified as a parameter, use that
  if (!missing(sample)) {
    samples <- sample
  }
  # If the array is specified, use that to find which samples it's involved in
  else if (!missing(array)) {
    samples <- unique(rawdata$design$Sample[rawdata$design$Array %in% array])
  # Otherwise use all the samples
  } else {
    cat("Array variable missing, doing cross for all technical replicates\n")
    # Work out number of samples in data
    samples <- 1:max(rawdata$design$Sample)
  }

  # Loop around all the samples
  for (i.sample in samples) {
    # Output to screen which samples are being updated
    cat(paste("Updating sample", i.sample, "\n"))

    # Get the array number and dye from the design template for sample i
    array <- as.matrix(rawdata$design[rawdata$design$Sample == i.sample,
                       c("Array", "Dye")])

    # Calculate number of arrays extracted and therefore number of crosses
    n.arrays <- dim(array)[1]
    n.crosses <- n.arrays * (n.arrays - 1) / 2

    # Replace dye names with integers 1 and 2
    idx = which(array[,2] == dyes[1])
    array[idx,2] <- 1
    idx = which(array[,2] == dyes[2])
    array[idx,2] <- 2

    # Convert the matrix to integers and order by dye
    array <- matrix(as.integer(array[order(array[,2]),]), nrow = dim(array)[1],
                    ncol = dim(array)[2])

    # Create a matrix of the crosses and store as column numbers in the data
    datacols <- matrix(0, ncol = 6, nrow = n.crosses)

    # Create a variable to count the number of loops
    count <- 0

    # Loop for each array
    for (i in 1:(n.arrays - 1)) {
      # Loop for the crossed array
      for (j in (i + 1):n.arrays) {
        # Increment the counter
        count <- count + 1
        # Retain the first array number and dye information
        datacols[count, 1] <- array[i,1]
        datacols[count, 2] <- array[i,2]
        # Extract the column number for the first array
        datacols[count, 3] <- array[i,1] * 2 + array[i,2] - 2
        # Retain the second array number and dye information
        datacols[count, 4] <- array[j,1]
        datacols[count, 5] <- array[j,2]
        # Extract the column number for the second array
        datacols[count, 6] <- array[j,1] * 2 + array[j,2] - 2
      }
    }

    # Open the PDF file for the output
    pdf(file=paste("Output/Tech Reps/Sample", i.sample, ".pdf", sep = ""),
        height=30, width=30, title="Maanova TechRepCheck Output")

    # Work out layout matrix
    lm <- matrix(0, ncol = (n.arrays - 1), nrow = (n.arrays - 1))
    count <- 0
    for (i.lm in 1:(n.arrays - 1)) {
      for (j.lm in i.lm:(n.arrays - 1)) {
        count <- count + 1
        lm[j.lm, i.lm] <- count
      }
    }

    layout(lm)
    par(mar=margin, cex.main=2.5, cex.lab=2, cex.axis=1.5)

    # Loop around the crossed arrays and draw plots
    for (i.cross in 1:n.crosses) {

      # Gather data
      data1 <- log2(rawdata$data[,datacols[i.cross, 3]])
      data2 <- log2(rawdata$data[,datacols[i.cross, 6]])

      # Organise axis labels
      xlabel <- paste("Array", datacols[i.cross,1], "Dye",
                      dyes[datacols[i.cross,2]])
      ylabel <- paste("Array", datacols[i.cross,4], "Dye",
                      dyes[datacols[i.cross,5]])
      # Check that highlighting is desired and data is available
      if(high.flag & !is.null(flag)) {
        # Create a list of points to plot highlighted
        high <- vector(length = dim(flag)[1])
        for (i.high in 1:dim(flag)[1])
          high[i.high] <- (flag[i.high,datacols[i.cross, 1]]!=0) ||
                          (flag[i.high,datacols[i.cross, 4]]!=0)
      } else {
        # Otherwise set high to NULL
        high <- NULL
      }

      # Draw the data
      plot(data1[!high], data2[!high], col = "blue",
           pch = 4, cex = 0.5, xlab = xlabel, ylab = ylabel,
           main = paste("Sample", i.sample))

      # Plot points where high is TRUE, as above, in colour "flag.color",
      points(data1[high], data2[high], col=flag.color,
             pch=4, cex=0.5)

      #Now need to do Principal Component Analysis

      S <- sprintf("%03d", i.sample)
      A1 <- sprintf("%03d", datacols[i.cross,1])
      A2 <- sprintf("%03d", datacols[i.cross,4])
      outfile <- paste("Output/Tech Reps/Scattered Points/S", S, "-A", A1,
                       "-vs-A", A2, ".txt", sep="")

      PCAout <- perform2DPCA (xData = data1, yData = data2, IDs = rawdata$ID,
                              testIDs = rawdata$testIDs, sig = 3.2906,
                              outfile = outfile)

      # Insert line into summary matrix
      outputmat <- rbind(outputmat, c(i.sample, datacols[i.cross,1],
                   as.character(dyes[datacols[i.cross,2]]), datacols[i.cross,4],
                   as.character(dyes[datacols[i.cross,5]]), PCAout))
    }

    # Close PDF File
    dev.off()
  }

  # Remove blank rows from matrix
  outputmat <- outputmat[outputmat[,1] != "",]

  # Output the summary to a file
  write.table(outputmat,
              file = "Output/Tech Reps/Summary of Plots.txt", sep="\t",
              row.names=FALSE)


  # Create a new matrix to hold average PC1 percentage variances for each
  # TechRep
  techrepmat <- matrix("", ncol=4)
  colnames(techrepmat) <- c("Sample", "Array", "Dye", "% PC1 Var")

  # Loop around all the samples that have been analysed
  for (i.sample in samples) {

    # Select data for this sample from the outputmat
    suboutput <- outputmat[outputmat[,1] == i.sample,]

    # Make a list of the arrays which make up the TechRep
    arrays <- unique(as.vector(c(suboutput[,2], suboutput[,4])))

    # Loop around all the arrays in the sample
    for (i.array in 1:length(arrays)) {
      array <- arrays[i.array]
      dye <- rawdata$design$Dye[rawdata$design$Array == array &
                                rawdata$design$Sample == i.sample]
      dye <- levels(rawdata$design$Dye)[dye]

      # Find the average PC1 Variance for all the plots showing this techrep
      # and this array and write it to a new line in the matrix
      avg <- mean(as.numeric(as.vector(outputmat[outputmat[,1] == i.sample &
            (outputmat[,2] == array | outputmat[,4] == array), "% PC1 Var"])))
      avg <- signif(avg, 4)
      techrepmat <- rbind(techrepmat, c(i.sample, array, dye, avg))
    }
  }

  # Remove blank rows from matrix
  techrepmat <- techrepmat[techrepmat[,1] != "",]

  # Sort the entries by % PC1 Variance
  techrepmat <- techrepmat[order(techrepmat[,"% PC1 Var"]),]

  # Output the summary to a file
  write.table(techrepmat,
              file = "Output/Tech Reps/Summary of TechRep Quality.txt",
              sep="\t", row.names=FALSE)
}
```

Graphical Output



Tabular Output

The same as gridcheck( ) plus an assessment of technical replicate quality as shown:

Sample	Array	Dye	% PC1 Var
72	20	Cy5	90.59
24	118	Cy5	90.87
52	62	Cy5	91.3
37	44	Cy5	91.37
85	122	Cy5	91.52
24	171	Cy5	91.64
44	117	Cy5	91.69
28	44	Cy3	92.05
1	123	Cy5	92.15
72	80	Cy3	92.32
1	25	Cy3	92.34
…	…	…	…

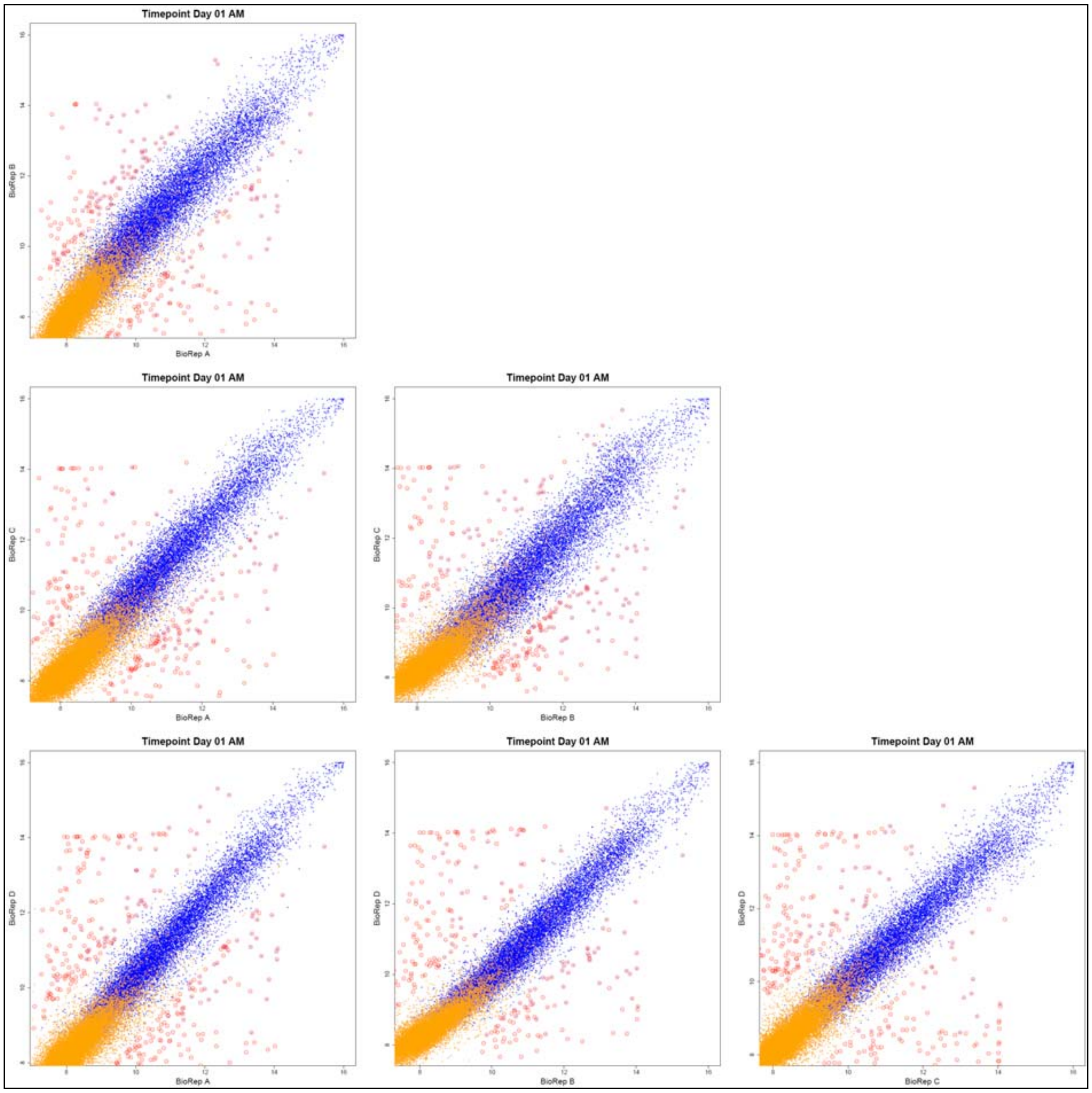
Appendix E – BioRepCheck( )

Code

```
################################################################################
#
# biorepcheck.R
#
# adapted from techrepcheck.R
#
# copyright (c) 2007, Stuart McHattie
#
# written Apr, 2007
#
# Licensed under the GNU General Public License version 2 (June, 1991)
#
# Part of the R/maanova package
#
################################################################################

biorepcheck <- function(rawdata, array, high.flag=TRUE,
                         flag.color="Orange", margin=c(6.1,5.6,4.1,2.6))
{
  # Turn off graphics before and after the function
  graphics.off()
  on.exit(graphics.off())

  # Generate Output File Structure
  if (!file.exists("Output/Bio Reps/Scattered Points")) {
    cat("Output folders missing, autogenerating them\n")
    dir.create("Output/Bio Reps/Scattered Points", recursive = TRUE)
  }

  # Check that BioRep data is loaded
  if (is.null(rawdata$design$BioRep))
    stop("There is no biological replicate data loaded; cannot perform BioRepCheck")

  # Check that the data format is correct
  if(class(rawdata) != "rawdata")
    stop("The first input variable is not an object of class rawdata.")

  # Check number of dyes in the dataset are 2
  if(rawdata$n.dye != 2)
    stop("BioRepCheck only works for 2-dye arrays")

  # Extract dye names
  dyes <- unique(rawdata$design$Dye)

  # Find names of BioReps and put them in order
  bioreps <- unique(rawdata$design$BioRep)
  bioreps <- as.vector(bioreps[order(bioreps)])

  # save old par parameters and setup new layout (reset graphics parameters at
  # the end of the function)
  old.mar <- par("mar")
  old.las <- par("las")
  old.cmain <- par("cex.main")
  old.clab <- par("cex.lab")
  old.caxis <- par("cex.axis")
  on.exit(par(las=old.las, mar=old.mar, cex.main=old.cmain, cex.lab=old.clab,
              cex.axis=old.caxis), add=TRUE)
  par(las=1)

  #Create a matrix to store the summary data in
  outputmat <- matrix("", ncol=16)
  colnames(outputmat) <- c("Time Point", "x BioRep", "y BioRep", "% PC1 Var",
                           "PC2 StDev", "x Mean", "y Mean", "Scattered",
                           "% Scattered", "Equation", "Upper PC1 Mean",
                           "Upper Statistic", "Upper P-Val", "Lower PC1 Mean",
                           "Lower Statistic", "Lower P-Val")

  # Combine Day and Time from the design file
  treattimes <- rawdata$design$TimePoint

  # If the array is specified, use that to find which time points it's
  # involved in
  if (!missing(array)) {
    timepoints <-
```

```
      unique(treattimes[rawdata$design$Array %in% array])

  # Otherwise use all the samples
  } else {
    cat("Array variable missing, doing crosses for all timepoints\n")

    # Work out names of timepoints in data and sort them
    timepoints <- unique(treattimes)
    timepoints <- timepoints[order(timepoints)]
  }

  # Loop around all the timepoints
  for (i.time in 1:length(timepoints)) {
    # Find the name of the timepoint
    timepoint <- timepoints[i.time]

    # Output to screen which samples are being updated
    cat(paste("Updating timepoint", timepoint, "\n"))

    # Gather all data for that timepoint and find mean of technical replicates
    # Extract a list of the arrays to be manipulated
    arrays <- as.matrix(rawdata$design[treattimes == timepoint,
                        c("Array", "Dye")])
    biorep <- as.matrix(rawdata$design[treattimes == timepoint,
                        "BioRep"])

    # Replace dye names with -1 and 0 (offset for column identification)
    idx = which(arrays[,2] == dyes[1])
    arrays[idx,"Dye"] <- -1
    idx = which(arrays[,2] == dyes[2])
    arrays[idx,"Dye"] <- 0

    # Convert values in arrays into integers
    arrays <- matrix(as.integer(arrays), ncol=2)

    # Create a new matrix to hold the mean data and give column headings
    meandata <- matrix(0, ncol = length(bioreps),
                       nrow = dim(rawdata$data)[1])
    colnames(meandata) <- bioreps

    # Prepare a matrix for the flags
    flag <- matrix(, ncol = length(bioreps), nrow = dim(rawdata$flag))
    colnames(flag) <- bioreps

    # Loop through bioreps, extracting data and finding the mean
    for (i.biorep in 1:length(bioreps)) {
      # Extract list of arrays in same technical replicate
      ext.arrays <- arrays[biorep == bioreps[i.biorep],]
      # Extract spot data for this technical replicate
      ext.data <- as.matrix(rawdata$data[,ext.arrays[,1] * 2 + ext.arrays[,2]])
      # Loop through all spots, calculating a mean for each and summing flags
      for (i.spot in 1:dim(ext.data)[1]) {
        meandata[i.spot,i.biorep] <- mean(ext.data[i.spot,])
        flag[i.spot,i.biorep] <- sum(rawdata$flag[i.spot,ext.arrays[,1]])
      }
    }

    # Calculate number of arrays extracted and therefore number of crosses
    n.data <- length(bioreps)
    n.crosses <- n.data * (n.data - 1) / 2

    # Open the PDF file for the output
    pdf(file=paste("Output/Bio Reps/Timepoint ", timepoint, ".pdf", sep = ""),
        height=30, width=30, title="Maanova BioRepCheck Output")

    # Work out layout matrix for crossed data
    lm <- matrix(0, ncol = (n.data - 1), nrow = (n.data - 1))
    count <- 0
    for (i.lm in 1:(n.data - 1)) {
      for (j.lm in i.lm:(n.data - 1)) {
        count <- count + 1
        lm[j.lm, i.lm] <- count
      }
    }

    # Compile cross list
    crosses <- matrix(0, ncol = 2, nrow = n.crosses)
    count <- 0
    for (i in 1:(n.data - 1)) {
      for (j in (i + 1):n.data) {
        count <- count + 1
        crosses[count,] <- c(i, j)
      }
    }

    # Apply layout matrix
    layout(lm)
    par(mar=margin, cex.main=2.5, cex.lab=2, cex.axis=1.5)

    # Loop around the crossed data and draw plots
    for (i.cross in 1:n.crosses) {

      # Gather data
      data1 <- log2(meandata[,crosses[i.cross,1]])
      data2 <- log2(meandata[,crosses[i.cross,2]])

      # Organise axis labels
      xlabel <- paste("BioRep", bioreps[crosses[i.cross,1]])
      ylabel <- paste("BioRep", bioreps[crosses[i.cross,2]])

      # Check that highlighting is desired and data is available
      if(high.flag && !is.null(flag)) {
        # Create a list of points to plot highlighted
        high <- vector(length = dim(flag)[1])
        for (i.spot in 1:dim(flag)[1])
          high[i.spot] <- (flag[i.spot,crosses[i.cross, 1]]!=0) ||
                          (flag[i.spot,crosses[i.cross, 2]]!=0)
      } else
        # Otherwise set high to NULL
        high <- NULL

      # Draw the data
      plot(data1[!high], data2[!high], col = "blue", pch = 4, cex = 0.5,
           xlab = xlabel, ylab = ylabel, main = paste("Timepoint", timepoint))

      # Plot points where high is TRUE, as above, in colour "flag.color",
      points(data1[high], data2[high], col=flag.color, pch=4, cex=0.5)

      #Now need to do Principal Component Analysis

      T <- timepoint
      B1 <- bioreps[crosses[i.cross,1]]
      B2 <- bioreps[crosses[i.cross,2]]
      outfile <- paste("Output/Bio Reps/Scattered Points/Timepoint ", T, " - ", B1,
                       "-vs-", B2, ".txt", sep="")

      PCAout <- perform2DPCA (xData = data1, yData = data2, IDs = rawdata$ID,
                              testIDs = rawdata$testIDs, sig = 3.2906,
                              outfile = outfile)

      # Insert line into summary matrix
      outputmat <- rbind(outputmat, c(as.character(T), as.character(B1),
                                      as.character(B2), PCAout))
    }

    # Close PDF File
    dev.off()
  }

  # Remove blank rows from matrix
  outputmat <- outputmat[outputmat[,1] != "",]

  # Output the summary to a file
  write.table(outputmat, file = "Output/Bio Reps/Summary of Plots.txt", sep="\t",
              row.names=FALSE)

  # Create a new matrix to hold average PC1 percentage variances for each
  # BioRep in each Timepoint
  biorepmat <- matrix("", ncol=4)
  colnames(biorepmat) <- c("Time Point", "BioRep", "% PC1 Var", "Arrays in Rep")

  # Loop around all the timepoints that have been analysed
  for (i.time in 1:length(timepoints)) {
    timepoint <- timepoints[i.time]

    # Loop around all the bioreps that were analysed
    for (i.biorep in 1:length(bioreps)) {
      biorep <- bioreps[i.biorep]

      # Find the average PC1 Variance for all the plots showing this biorep
      # at this timepoint and write it to a new line in the matrix
      avg <- mean(as.numeric(as.vector(outputmat[outputmat[,1] == timepoint &
            (outputmat[,2] == biorep | outputmat[,3] == biorep), "% PC1 Var"])))
      avg <- signif(avg, 4)
      biorepmat <- rbind(biorepmat, c(timepoint, biorep, avg,
                    paste(rawdata$design$Array[treattimes == timepoint &
                    rawdata$design$BioRep == biorep], collapse=", ")))
    }
  }

  # Remove blank rows from matrix
  biorepmat <- biorepmat[biorepmat[,1] != "",]

  # Sort the entries by % PC1 Variance
  biorepmat <- biorepmat[order(biorepmat[,"% PC1 Var"]),]

  # Output the summary to a file
  write.table(biorepmat,
              file = "Output/Bio Reps/Summary of BioRep Quality.txt",
              sep="\t", row.names=FALSE)
}
```

Graphical Output



Tabular Output

The same as gridcheck( ) and techrepcheck( ) but an alternative assessment of quality of biological replicates:

Time Point	BioRep	% PC1 Var	Arrays in Rep
Day 11 PM	A	93.57	41, 42, 78, 122
Day 09 PM	B	95.38	106, 109, 119, 124
Day 09 PM	A	95.48	59, 82, 120, 173
Day 09 PM	C	95.52	19, 90, 105, 133
Day 09 PM	D	95.54	20, 21, 74, 80
Day 10 AM	C	95.63	9, 84, 106, 169
Day 04 PM	D	96.12	35, 64, 136, 150
Day 11 PM	D	96.36	7, 40, 89, 176
Day 07 PM	A	96.42	24, 27, 73, 167
Day 07 PM	B	96.46	96, 140, 148, 165
Day 11 PM	B	96.48	77, 81, 113, 127
…	…	…	…

Appendix F – perform2DPCA( )

Code

```
################################################################################
#
# perform2DPCA.R
#
# Internal function to perform Principal Component Analysis
#
# copyright (c) 2007, Stuart McHattie
#
# written Apr, 2007
#
# Licensed under the GNU General Public License version 2 (June, 1991)
#
# Part of the R/maanova package
#
################################################################################

perform2DPCA <- function(xData, yData, IDs, testIDs, sig, outfile,
                          plotpoints = TRUE) {

  # Manipulate the data into a format for the prcomp() routine
  mandata <- matrix(ncol=2, nrow=length(xData))
  rownames(mandata) <- IDs
  mandata[,1] <- xData
  mandata[,2] <- yData

  # Remove lines where the gene name is a test spot
  mandata <- mandata[!rownames(mandata) %in% testIDs,]

  # Perform the Principal Component Analysis
  pcaoutput <- prcomp(mandata)

  # Extract information from output matrices
  varPC1 <- round(summary(pcaoutput)$importance[2,1]*100, 2)
  sdPC2 <- pcaoutput$sdev[2]
  PCb <- pcaoutput$rotation[1,2]
  PCd <- pcaoutput$rotation[2,2]
  xmean <- pcaoutput$center[1]
  ymean <- pcaoutput$center[2]
  intercept <- round(PCd/PCb*ymean+xmean, 3)
  slope <- round(PCd/PCb, 3)
  # Since the equation is -PCd/PCb to format it well for human viewing, must
  # change the sign according to the outcome of PCd/PCb (to avoid +- showing)
  if (slope < 0)
    equation <- paste("y =", intercept, "+", slope*-1, "x")
  else
    equation <- paste("y =", intercept, "-", slope, "x")

  # Identify which points show significant scatter
  scatter <- pcaoutput$x[,2]/sdPC2
  sigbool <- scatter < -sig | scatter > sig

  # Make a list of the points ready for plotting
  scatterlist <- matrix(ncol=4, nrow=length(scatter[sigbool]))
  colnames(scatterlist) <- c("Gene Name", "X-axis", "Y-axis", "Scatter")
  scatterlist[,1] <- names(scatter[sigbool])
  scatterlist[,2] <- mandata[rownames(mandata) %in% scatterlist[,1],1]
  scatterlist[,3] <- mandata[rownames(mandata) %in% scatterlist[,1],2]
  scatterlist[,4] <- scatter[sigbool]

  # Circle the points of interest if requested
  if (plotpoints) points(scatterlist[,2], scatterlist[,3], col="Red", cex=2)

  # Output results to a table
  write.table(scatterlist, file = outfile, sep = "\t", row.names=FALSE)

  # Use Student's T-Test to find out if scattered points are significantly
  # deviating from the mean on the PC1 axis
  scathi <- pcaoutput$x[sigbool,1][pcaoutput$x[sigbool,2] > 0]
  scatlo <- pcaoutput$x[sigbool,1][pcaoutput$x[sigbool,2] < 0]
  # No need to compare with huge dataset of all points
  # all.vals <- pcaoutput$x[,1]

  # Do t-test on upper scattered points
  if (length(scathi) > 1) {
    testres <- t.test(scathi)
    p.val.hi <- testres$p.value
    stat.hi <- testres$statistic
    mean.hi <- testres$estimate
  } else {
    p.val.hi <- "N/A"
    stat.hi <- "N/A"
    mean.hi <- "N/A"
  }

  # Do t-test on lower scattered points
  if (length(scatlo) > 1) {
    testres <- t.test(scatlo)
    p.val.lo <- testres$p.value
    stat.lo <- testres$statistic
    mean.lo <- testres$estimate
  } else {
    p.val.lo <- "N/A"
    stat.lo <- "N/A"
    mean.lo <- "N/A"
  }

  # Arrange a vector to be returned to the calling function
  output <- c(varPC1, sdPC2, xmean, ymean, length(scatterlist[,1]),
            round(length(scatterlist[,1])/length(mandata[,1])*100, 2), equation,
            mean.hi, stat.hi, p.val.hi, mean.lo, stat.lo, p.val.lo)
}
```

Appendix G – Model Output

Summary

```
summary(anova.mix)

                       Length   Class   Mode
yhat                   11421696 -none-  numeric
S2                        97344 -none-  numeric
loops                     32448 -none-  numeric
S2.level                      2 -none-  character
G                         32448 -none-  numeric
Dye                       64896 -none-  numeric
Dye.level                     2 -none-  character
Array                   5710848 -none-  numeric
Array.level                 176 -none-  numeric
TimePoint                713856 -none-  numeric
TimePoint.level              22 -none-  character
TimePoint:BioRep        2855424 -none-  numeric
TimePoint:BioRep.level       88 -none-  character
flag                      32448 -none-  numeric
model                        12 mamodel list
```

Individual Parameters

```
> dim(anova.mix$TimePoint)
[1] 32448    22

> anova.mix$TimePoint[1:10, 1:4]
              [,1]        [,2]        [,3]         [,4]
 [1,]  1.213715449  1.10966911  0.81981948  1.158293547
 [2,]  1.414231502  1.05306128  0.75960569  1.286807969
 [3,]  1.387664951  0.95747537  0.79515036  0.890010694
 [4,]  1.462511327  1.03066007  0.98226149  1.049809650
 [5,]  0.967061927  0.85332913  0.55617979  0.884961583
 [6,]  0.050756465 -0.23073787  0.07288790 -0.065792751
 [7,] -0.125654815 -0.10563993 -0.09511758 -0.142143538
 [8,] -0.035051019 -0.05312165 -0.05347819 -0.068638561
 [9,] -0.056957998 -0.16632648  0.06988751 -0.128748059
[10,]  0.034335940 -0.25265662  0.07434679 -0.067072408
```

Contents of S2

```
> dim(anova.mix$S2)
[1] 32448     3

> anova.mix$S2[1:10,]
              [,1]      [,2]       [,3]
 [1,] 0.0416663146 0.5146385 0.19428796
 [2,] 0.0281118749 0.6112680 0.22133446
 [3,] 0.0303857368 0.5113540 0.25987722
 [4,] 0.0195897124 0.6341246 0.22514883
 [5,] 0.0215976528 0.2646514 0.34584859
 [6,] 0.0039051894 0.1795479 0.13140303
 [7,] 0.0000000001 0.1261334 0.14007444
 [8,] 0.0028830023 0.2395086 0.09081522
 [9,] 0.0035580724 0.1558065 0.13335842
[10,] 0.0018159542 0.1573051 0.12552665
```

Contents of G

```
> length(anova.mix$G)
[1] 32448

> anova.mix$G[1:10]
 [1]  2.808159  2.819874  2.871976  2.816789  2.856149 -2.259728 -2.297752
 [8] -2.264884 -2.326288 -2.292416
```

Appendix H – Checking Spans

Spearman's Rank Correlations

Spans	10 Significant Genes		25 Significant Genes		50 Significant Genes		100 Significant Genes		500 Significant Genes		1000 Significant Genes	
	Correlation	P-value	Correlation	P-value	Correlation	P-value	Correlation	P-value	Correlation	P-value	Correlation	P-value
0.01 vs 0.05	0.854545455	0.003	0.479230769	0.016	0.816854742	0.000	0.837623762	0.000	0.924799059	0.000	0.949445281	0.000
0.01 vs 0.10	0.696969697	0.029	0.411538462	0.042	0.787659064	0.000	0.815421542	0.000	0.905008964	0.000	0.933364857	0.000
0.01 vs 0.25	0.696969697	0.029	0.38	0.062	0.758943577	0.000	0.793879388	0.000	0.885326965	0.000	0.918800251	0.000
0.01 vs 0.50	0.660606061	0.042	0.388461538	0.056	0.743481393	0.000	0.781326133	0.000	0.867901072	0.000	0.906688771	0.000
0.01 vs 1.00	0.503030303	0.138	0.343846154	0.092	0.698823529	0.000	0.687188719	0.000	0.806024792	0.000	0.865576286	0.000
0.05 vs 0.10	0.842424242	0.004	0.930769231	0.000	0.973685474	0.000	0.969816982	0.000	0.988260209	0.000	0.991648688	0.000
0.05 vs 0.25	0.733333333	0.020	0.907692308	0.000	0.94362545	0.000	0.932049205	0.000	0.967466142	0.000	0.975394923	0.000
0.05 vs 0.50	0.660606061	0.042	0.816153846	0.000	0.897238896	0.000	0.903978398	0.000	0.938064584	0.000	0.956000192	0.000
0.05 vs 1.00	0.527272727	0.118	0.766153846	0.000	0.785066026	0.000	0.764008401	0.000	0.834066792	0.000	0.895060075	0.000
0.10 vs 0.25	0.939393939	0.000	0.973846154	0.000	0.9672509	0.000	0.96780078	0.000	0.985253285	0.000	0.989979798	0.000
0.10 vs 0.50	0.854545455	0.003	0.868461538	0.000	0.923265306	0.000	0.935949595	0.000	0.957513206	0.000	0.971164415	0.000
0.10 vs 1.00	0.781818182	0.011	0.790769231	0.000	0.779495798	0.000	0.786570657	0.000	0.847372349	0.000	0.903761824	0.000
0.25 vs 0.50	0.927272727	0.000	0.914615385	0.000	0.953997599	0.000	0.975193519	0.000	0.983781503	0.000	0.98951127	0.000
0.25 vs 1.00	0.83030303	0.005	0.872307692	0.000	0.766722689	0.000	0.797563756	0.000	0.873712743	0.000	0.924306288	0.000
0.50 vs 1.00	0.903030303	0.001	0.934615385	0.000	0.881776711	0.000	0.81989799	0.000	0.915767727	0.000	0.948567217	0.000

Spearman's Rank Correlations Continued

Spans	2000 Significant Genes		5000 Significant Genes		10000 Significant Genes	
	Correlation	P-value	Correlation	P-value	Correlation	P-value
0.01 vs 0.05	0.961880628	0.000	0.977927709	0.000	0.986376711	0.000
0.01 vs 0.10	0.949138406	0.000	0.970144267	0.000	0.981655289	0.000
0.01 vs 0.25	0.933496853	0.000	0.960070842	0.000	0.975463751	0.000
0.01 vs 0.50	0.9237219	0.000	0.951657782	0.000	0.970055446	0.000
0.01 vs 1.00	0.892748341	0.000	0.930757642	0.000	0.956859547	0.000
0.05 vs 0.10	0.994756372	0.000	0.996997087	0.000	0.998411024	0.000
0.05 vs 0.25	0.982398838	0.000	0.989472702	0.000	0.994300137	0.000
0.05 vs 0.50	0.967714269	0.000	0.980192142	0.000	0.988851271	0.000
0.05 vs 1.00	0.920863795	0.000	0.954296054	0.000	0.973324788	0.000
0.10 vs 0.25	0.991892916	0.000	0.995778504	0.000	0.997752321	0.000
0.10 vs 0.50	0.977808976	0.000	0.987502682	0.000	0.993046288	0.000
0.10 vs 1.00	0.929095191	0.000	0.960512057	0.000	0.977498035	0.000
0.25 vs 0.50	0.992488637	0.000	0.995968978	0.000	0.997773825	0.000
0.25 vs 1.00	0.944815382	0.000	0.971450336	0.000	0.983978995	0.000
0.50 vs 1.00	0.965363221	0.000	0.982582002	0.000	0.990547994	0.000

2000 Significant Genes

Span 0.05	0.961881				
Span 0.10	0.949138	0.994756			
Span 0.25	0.933497	0.982399	0.991893		
Span 0.50	0.923722	0.967714	0.977809	0.992489	
Span 1.00	0.892748	0.920864	0.929095	0.944815	0.965363
	Span 0.01	**Span 0.05**	**Span 0.10**	**Span 0.25**	**Span 0.50**

Span	Sum of Correlations
0.01	4.660986
0.05	4.827614
0.10	4.842691
0.25	4.845093
0.50	4.827097
1.00	4.652885

Appendix I – Scripts to Handle Input and Output of SplineCluster

Usage

```
express <- CreateSplineClusterInput(rawdata, model, term, genes)

Run the SplineCluster program on the data files that are output, then:

ProcessSplineClusterOutput(express)

rawdata         - the raw data object, hri.raw during this project.
model           - the output from the fitmaanova( ) function, anova.mix in this project.
term            - the parameter of the model that is to be analysed, TimePoint in this
                  project.
genes           - the number of genes to be considered as significant, the default is
                  2000.
express         - an output object for the expression profiles.  This is used by
                  ProcessSplineClusterOutput( ) to identify the genes from the
                  SplineCluster output.
```

Generating the Input File

```
################################################
#
# function for extracting SplineCluster input
# Data should already be loaded
#
################################################

CreateSplineClusterInput <- function(rawdata, model, term, genes = 2000) {

       setwd("~/Documents/Work/Warwick/HRI/Modified Code")

       if (!file.exists("Output/SplineClusterData")) {
              cat("Output folders missing, autogenerating them\n")
              dir.create("Output/SplineClusterData", recursive = TRUE)
       }

       # Get the residual mean values
       residualmean2 <- model$S2[,dim(model$S2)[2]]

       # Find the ratio of the specified term against residual mean
       ratio <- apply(model[[term]], 1, var) / residualmean2

       # Make an index of the order of expression ratio
       o <- order(ratio, decreasing=TRUE)

       # Put ordered data into express and add gene names to rows
       express <- model[[term]][o,] + model$G[o] - apply(model[[term]][o,] + model$G[o], 1, mean)
       rownames(express) <- rawdata$ID[o]

       # Remove test spots and select top significance genes
       express <- express[!rownames(express) %in% rawdata$testIDs,]
       express <- express[1:genes,]

       # Put transposed data into data
       data <- NULL
       for (i in 1:genes)
              data <- c(data, express[i,])

       # Create timepoint vector
       timepoints <- 1:dim(model[[term]])[2]

       # Output four files
       write.table(express, file="Output/SplineClusterData/Expression Profiles.txt",
              quote=FALSE, sep="\t", col.names=FALSE)
       write.table(data, file="Output/SplineClusterData/SCData.dat",
              quote=FALSE, sep="\t", row.names=FALSE, col.names=FALSE)
       write.table(timepoints, file="Output/SplineClusterData/SCData_x.txt",
              quote=FALSE, sep="\t", row.names=FALSE, col.names=FALSE)
       write.table("SCData", file="Output/SplineClusterData/currentfile.txt",
              quote=FALSE, sep="\t", row.names=FALSE, col.names=FALSE)
```

```
        # Create shell file
        shell <- c(
                "#!/bin/sh",
                "../../../SplineCluster/SplineCluster targetfile=SCData.dat inputfile=SCData_x.txt
normalisetargets=0 priorprecision=0.1 membershipprobabilityfile=SCData_membership_probs_.dat",
                "cp SCData_clusters_.dat SCData.txt",
                "R CMD BATCH s_clusterplot.R")
        write.table(shell, file="Output/SplineClusterData/SCData_shell",
                quote=FALSE, sep="\t", row.names=FALSE, col.names=FALSE)

        # Set significant digits to 6
        for (i in 1:genes) {
                for (j in timepoints) {
                        express[i,j] <- signif(express[i,j], 6)
                }
        }

        cat("Files are ready for SplineCluster, confirm the folder tree\nfor SplineCluster then go
to the output folder in terminal and\ntype:\n\nbash SCData_shell\n\nthen run
ProcessSplineClusterOutput()")

        invisible(express)
}
```

Processing the Output File

```
##################################################
#
# function for extracting SplineCluster input
# Data should already be loaded
#
##################################################

ProcessSplineClusterOutput <- function(express) {

        setwd("~/Documents/Work/Warwick/HRI/Modified Code")

        if (!file.exists("Output/SplineClusterData"))
                stop("Input/Output folder missing, cannot continue")

        # Load in the output from SplineCluster and clean up columns
        SCOut <- as.matrix(read.table(file="Output/SplineClusterData/SCData.txt", sep=" ",
quote=""))
        idx <- seq(1, dim(SCOut)[2], 2)
        SCOut <- SCOut[,idx]

        # Load in gene annotations
        anno <- as.matrix(read.table(file="CATMA ID list.txt", sep="\t"))

        # Set significant digits to 6
        n.row <- dim(SCOut)[1]
        n.col <- dim(SCOut)[2]
        for (i in 1:n.row) {
                for (j in 2:n.col) {
                        SCOut[i,j] <- signif(SCOut[i,j], 6)
                }
        }

        # Create the output matrix and put the cluster numbers in
        clusters <- SCOut[,1]
        clusters <- cbind(clusters, matrix(0, nrow=n.row, ncol=3))
        colnames(clusters) <- c("Cluster", "Gene Name", "AT Code", "Annotation")

        # Identify which genes relate to the clusters
        # Take a list of the genenames, find all genes relating to
        # the first expression value, then if there are more than
        # one left, do it with the second column and continue until
        # the gene is identified, then put the name in the output
        genenames <- anno[,1]
        for (i in 1:n.row) {
                genes <- genenames
                colnum <- 1
                while (length(genes) != 1) {
                        idx <- which(express[,colnum] == SCOut[i, colnum+1])
                        idx2 <- which(genenames[idx] %in% genes)
                        genes <- genenames[idx][idx2]
                        colnum <- colnum + 1
                }
```

```
               clusters[i,2] <- genes
        }

        #Find the AT code and gene annotation
        for (i in 1:n.row) {
               idx <- which(anno[,1] == clusters[i,2])
               clusters[i,3:4] <- anno[idx,2:3]
        }

        write.table(clusters, file="Output/SplineClusterData/Clustered Genes.txt",
               quote=FALSE, sep="\t", row.names=FALSE)

        cat("Complete!!  Output is called 'Clustered Genes.txt'")
}
```

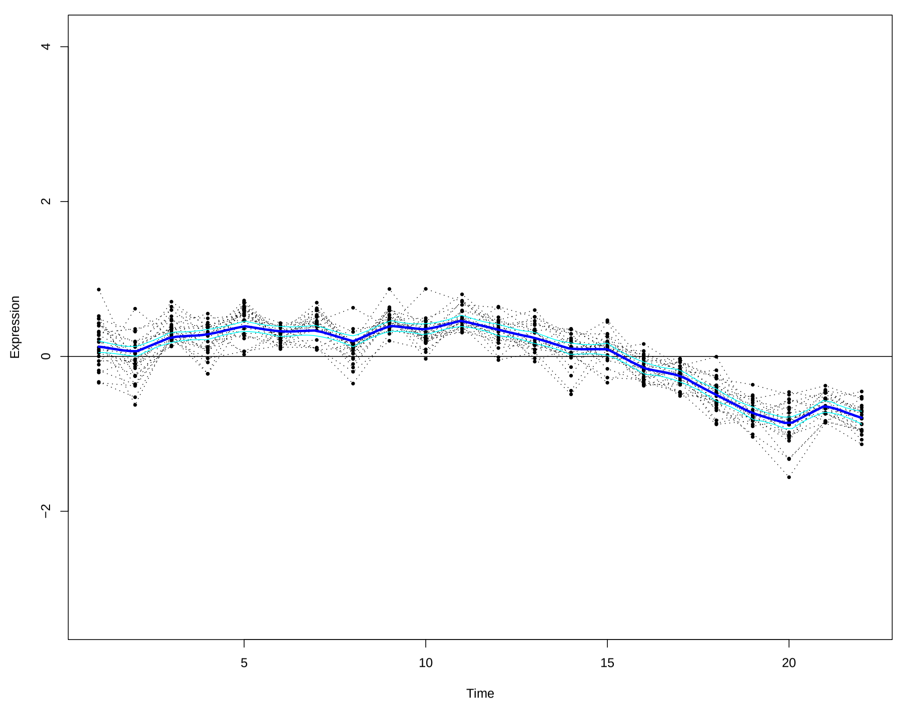
Appendix J – GOStat Results

Cluster #	# of Genes	Details of Significantly Under/Over Represented Genes
8	84	Protein formylation; gamma-aminobutyric acid transport; tRNA modification and amino acid transport.
9	13	Sulphur assimilation and utilisation; amino acid prenylation and lipidation as well as GTPase binding.
10	14	Peptidyl-proline hydroxylation to 4-hydroxy-L-proline.
13	19	DNA repair and acyl-ACP thioesterase activity.
16	144	Interaction with host species during parasite attack.
18	15	DNA-II-Helicase promoter regulation; DNA helicase activity and nucleotide excision repair.
23	34	ATPase activity and regulation of proteins involved in transport of ions and substances across membranes.
29	33	Lambda-DNA polymerase activity; GTP metabolic process; UTP biosynthetic process; lipoyltransferase activity and CTP biosynthetic process.
36	10	Meristem development; cell size regulation and transition from vegetative to reproductive phase.
37	8	CoA ligase activity and acid-thiol ligase activity.
38	10	Pollen tube development
42	12	Very significantly: Positive regulation of the immune system; programmed cell death and glycosinolate metabolic process.
47	16	Very significantly: Cellular defense response as well as adenosine, tyrosine and chorismate biosynthetic process.
52	6	Ligase activity, forming nitrogen-metal bonds and NADH dehydrogenase activity.
53	8	Sulphate adenylyltransferase activity.
67	9	Starch synthesis; GTPase binding; exocytosis and nuclear transport.
70	84	Poly-nucleotide kinase activity; fumerylacetoacetase activity; galactose biosynthesis and RNA ligase activity.
72	7	Post embryonic root development.
74	10	Galactose and hexose biosynthetic processes.
76	10	ATP-dependent DNA helicase activity.
81	19	Transport and localisation.
84	19	RNA metabolic processes and iron incorporation into metallo-sulphur cluster.
85	42	Positive regulation of S-phase in cell cycle; glycolysis; chlorophyllase activity; replisome regulation and sugar catabolism.

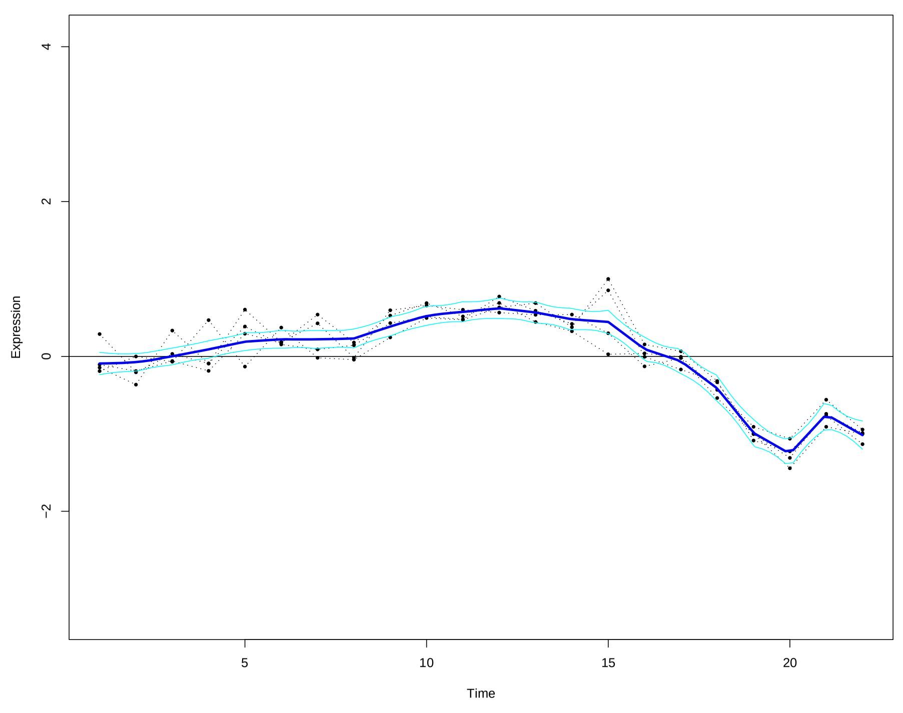
Appendix K – SplineCluster Output

The remaining pages of this report are the output from SplineCluster. These pages are un-numbered due to the complexity of incorporating them into the main document.

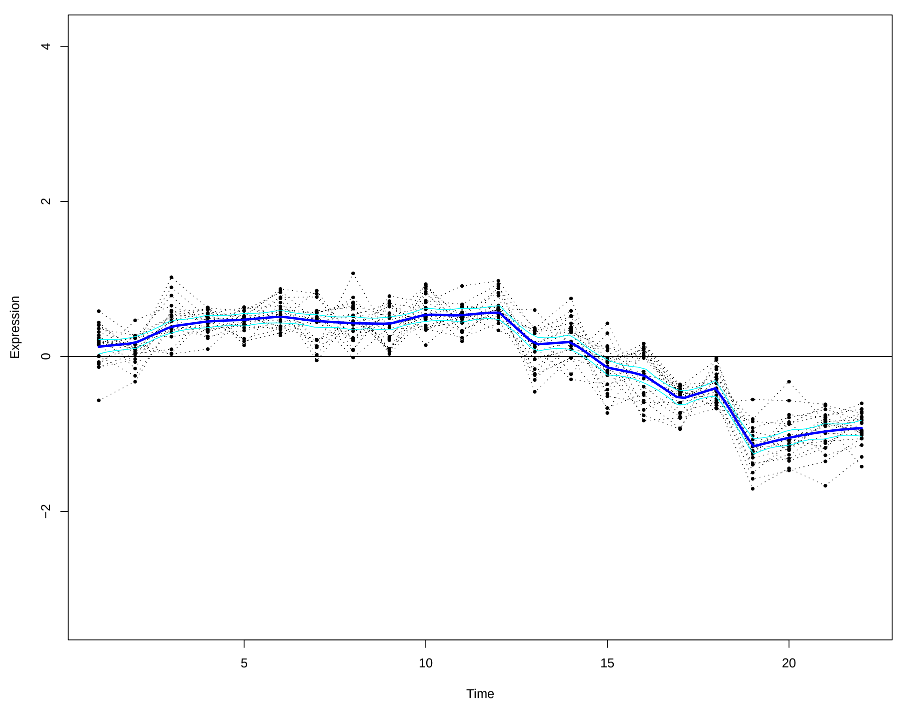
Cluster 1 (24 obsns.)



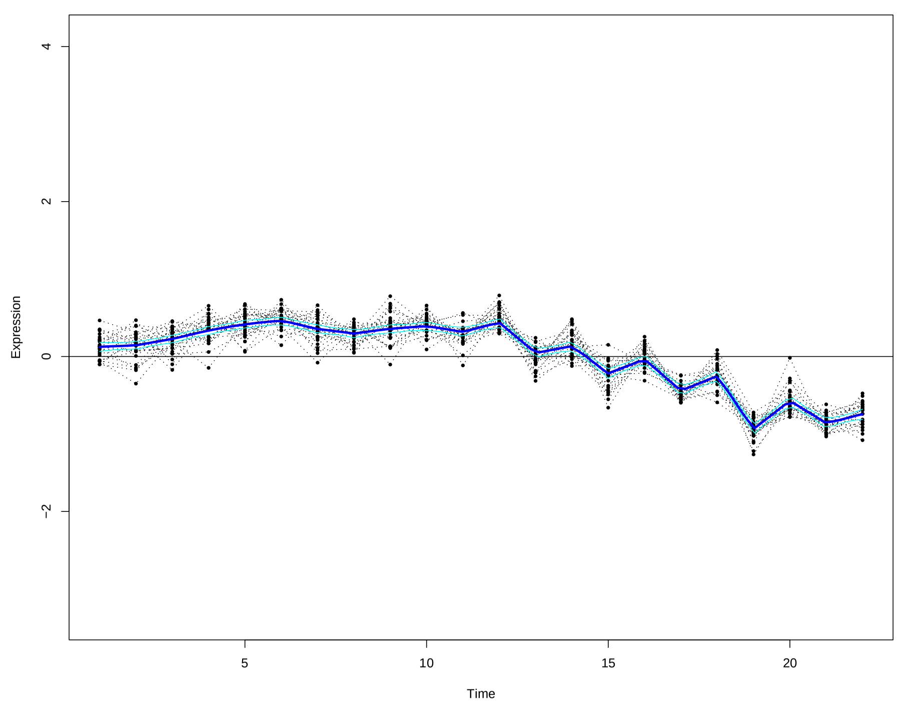
Cluster 2 (4 obsns.)



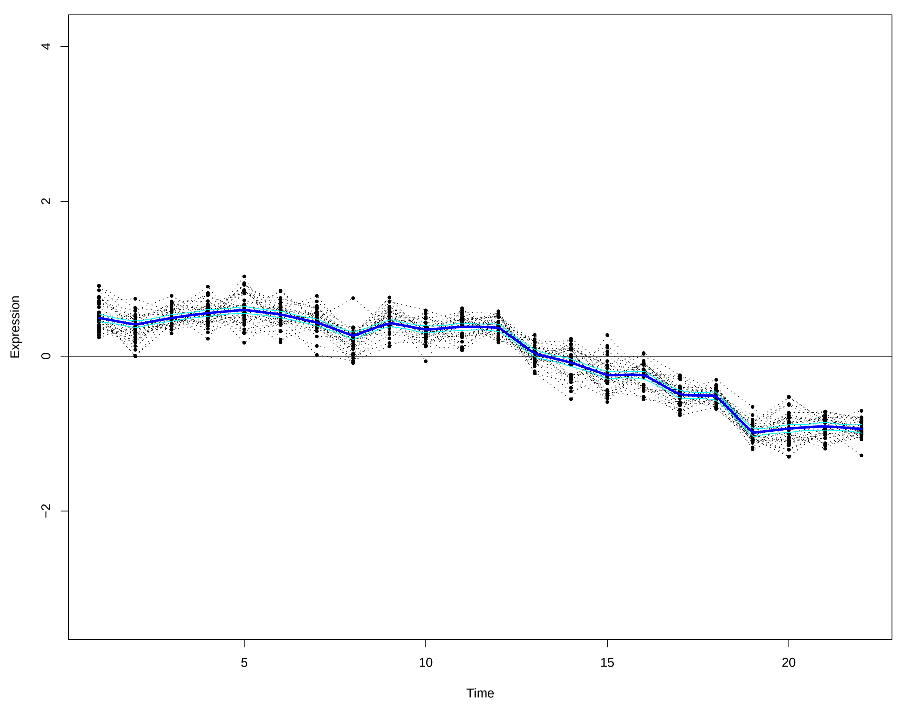
Cluster 3 (21 obsns.)



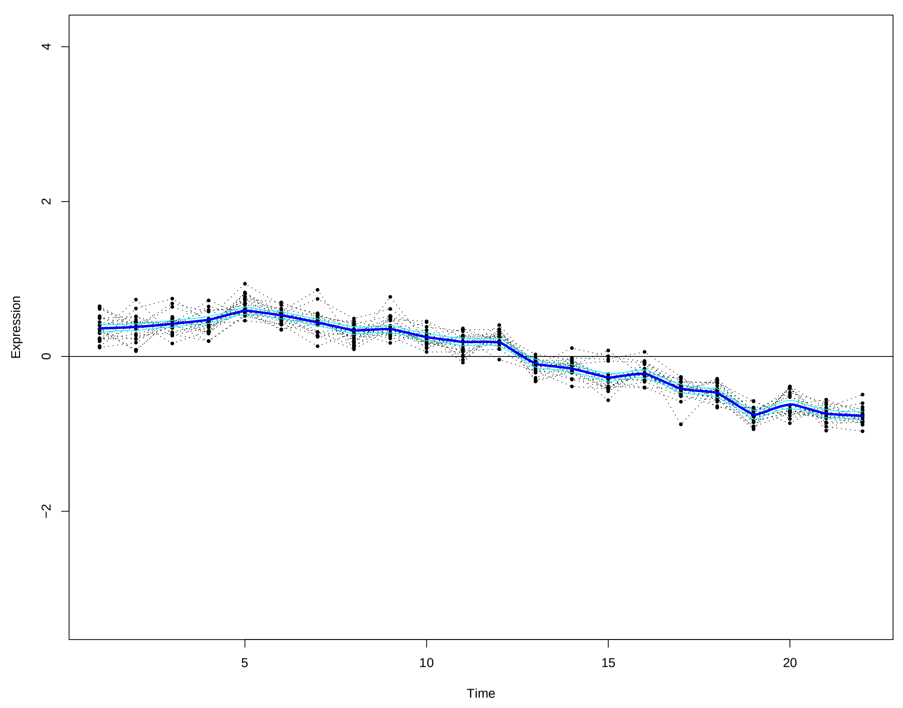
Cluster 4 (26 obsns.)



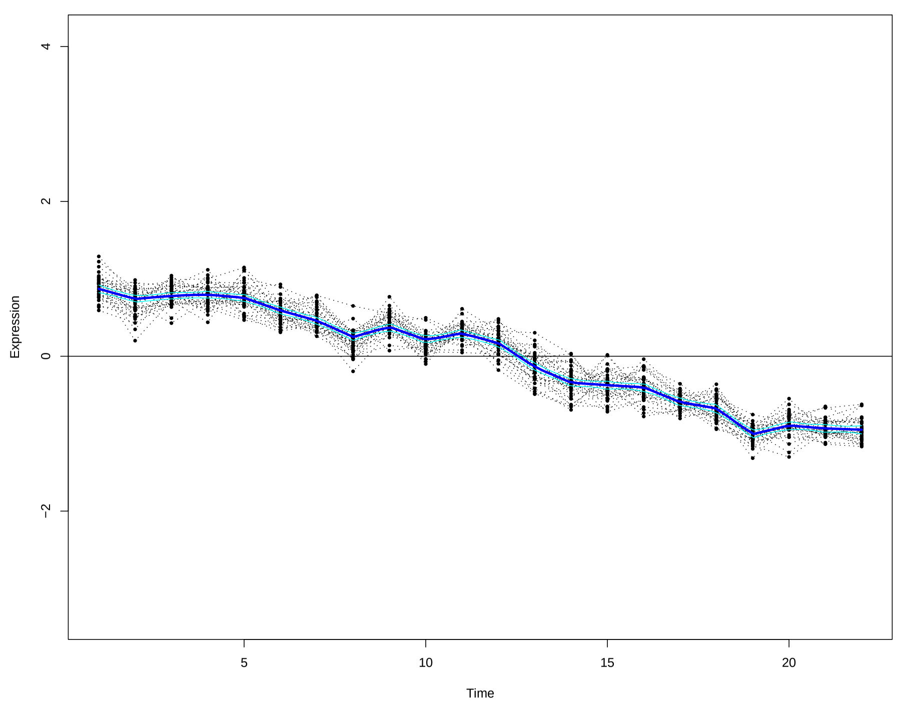
Cluster 5 (34 obsns.)



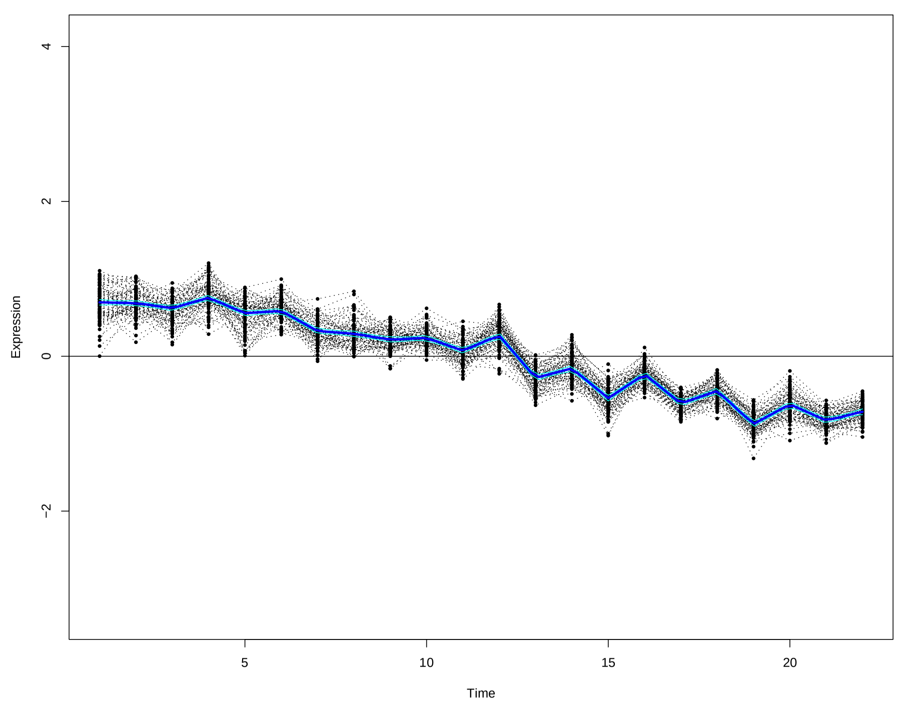
Cluster 6 (20 obsns.)



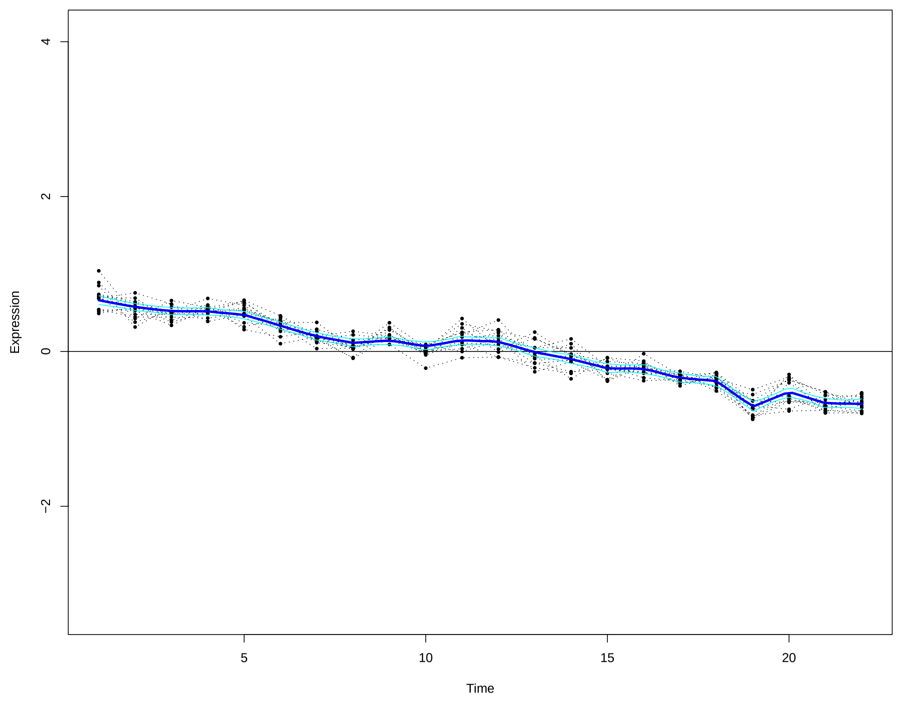
Cluster 7 (36 obsns.)



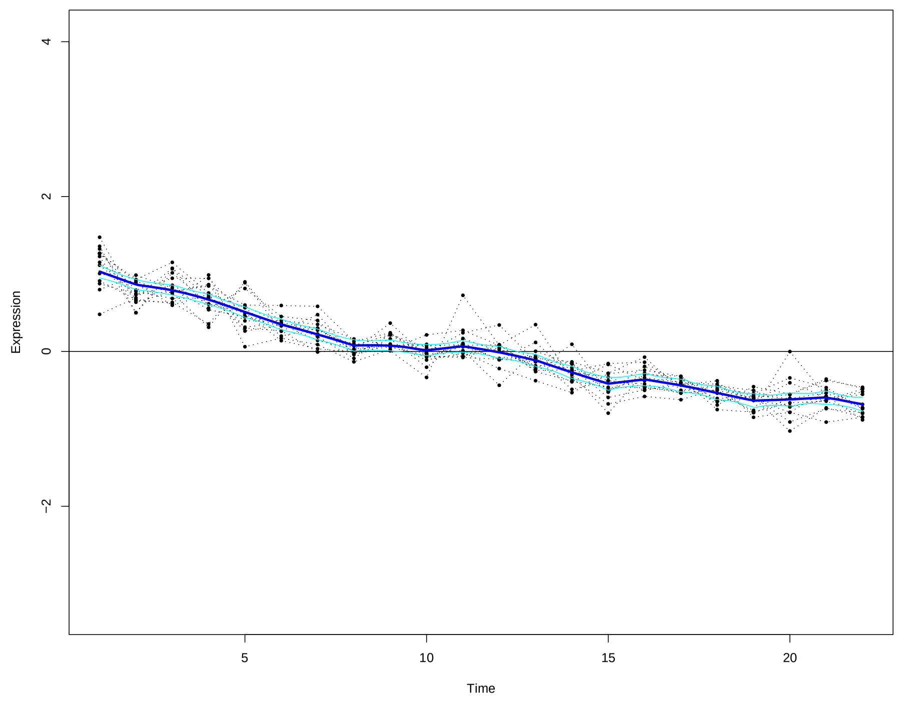
Cluster 8 (84 obsns.)



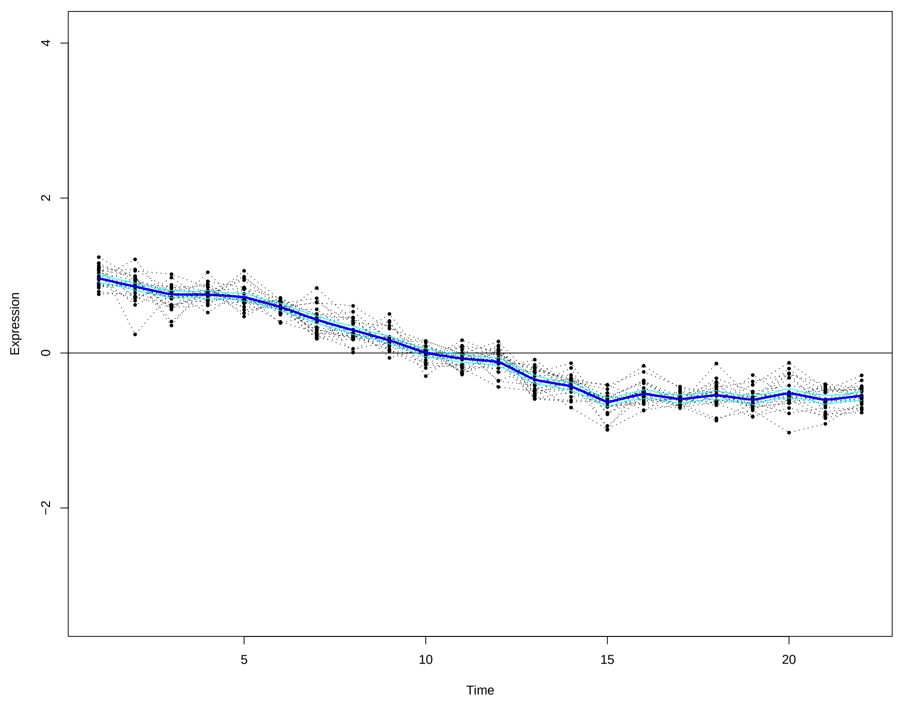
Cluster 9 (13 obsns.)



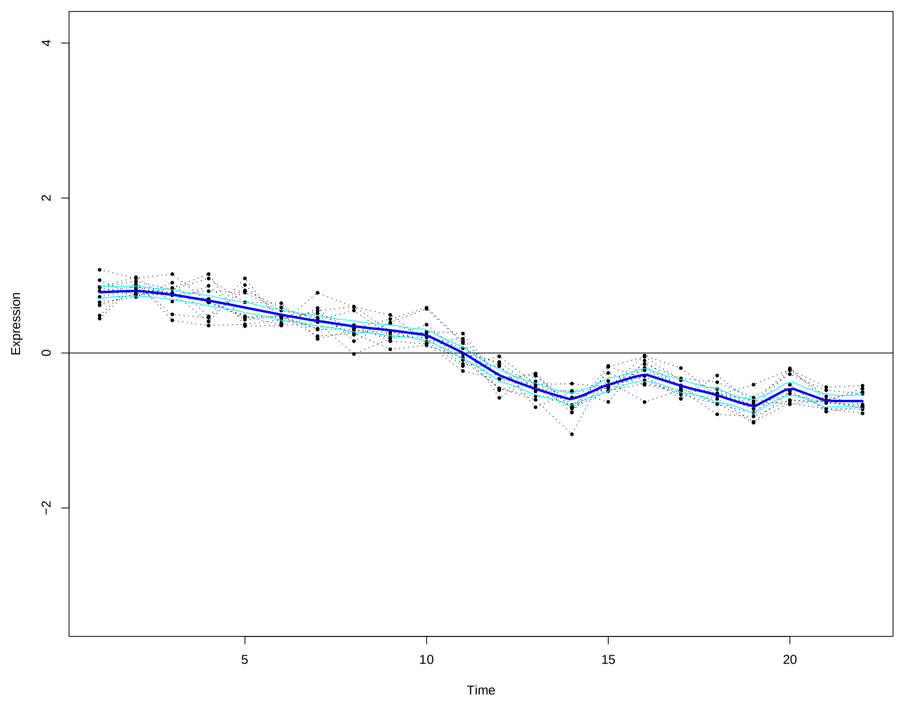
Cluster 10 (14 obsns.)

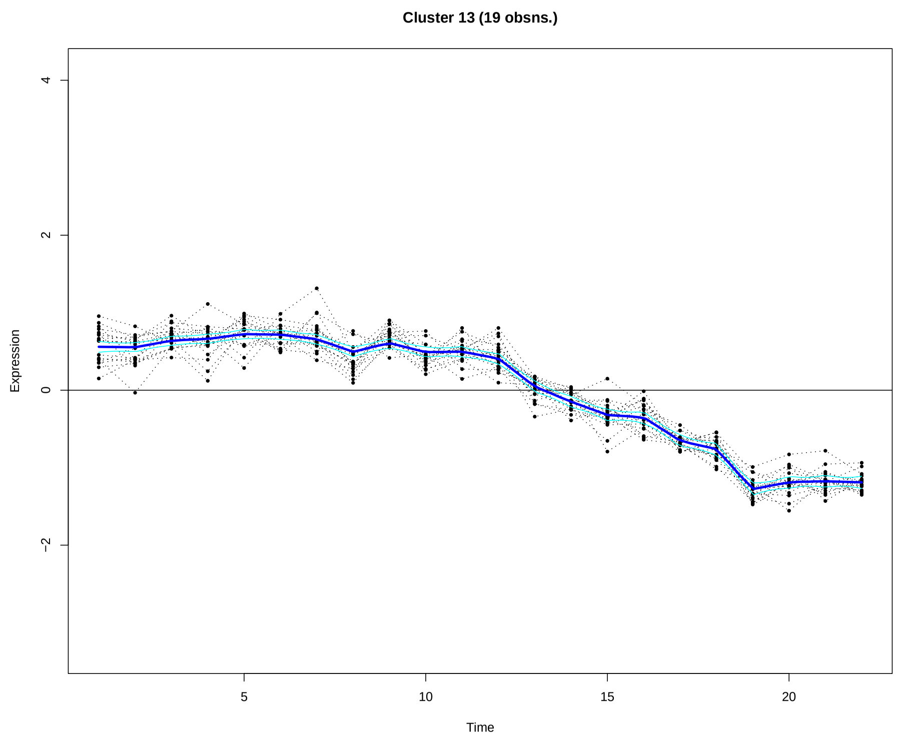


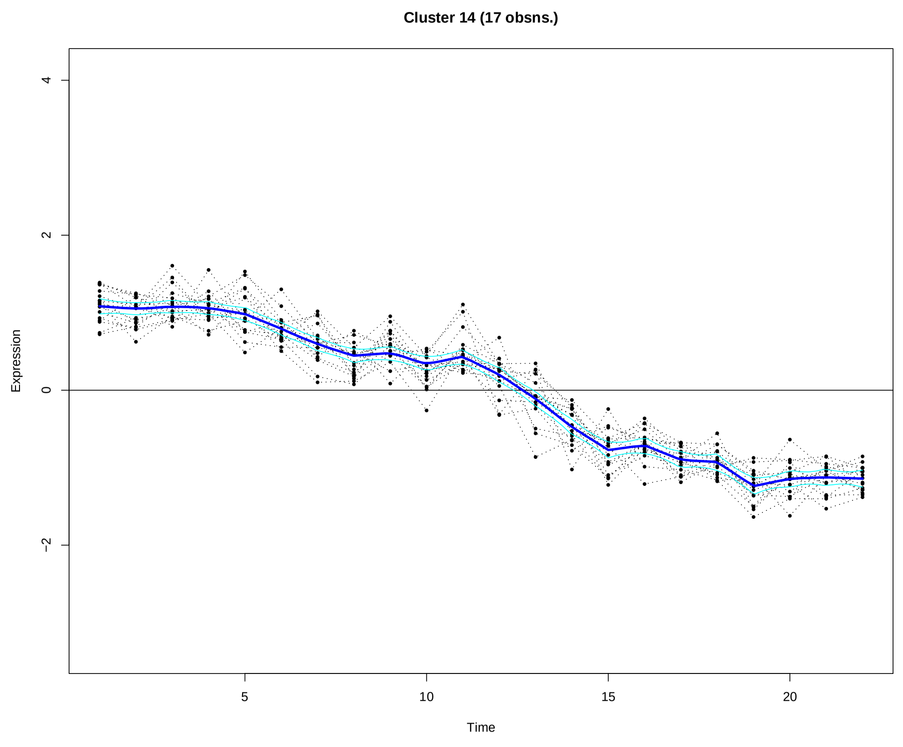
Cluster 11 (22 obsns.)

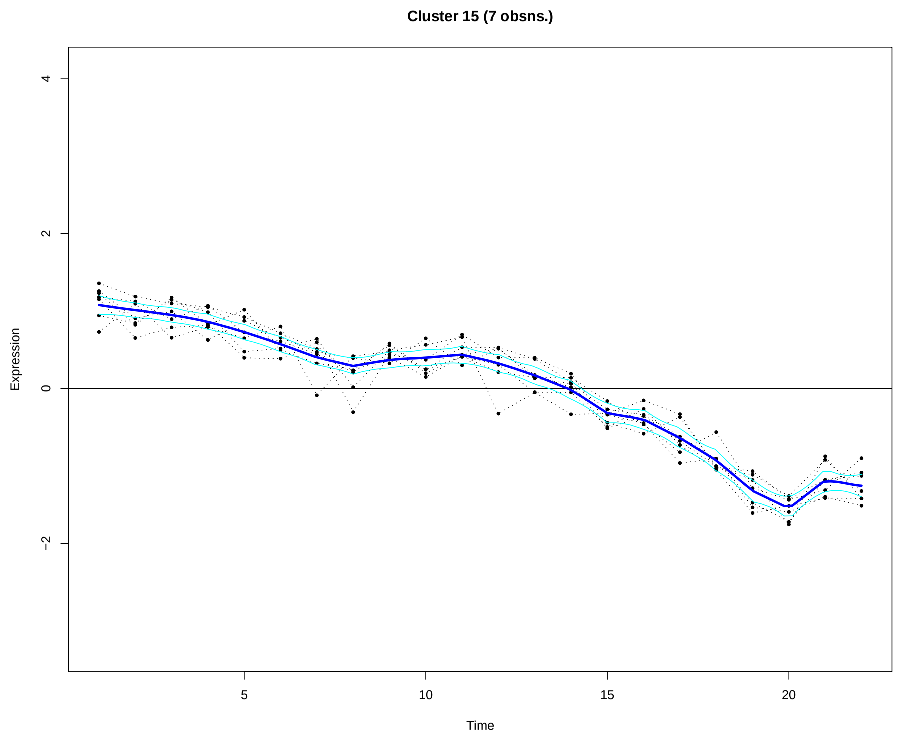


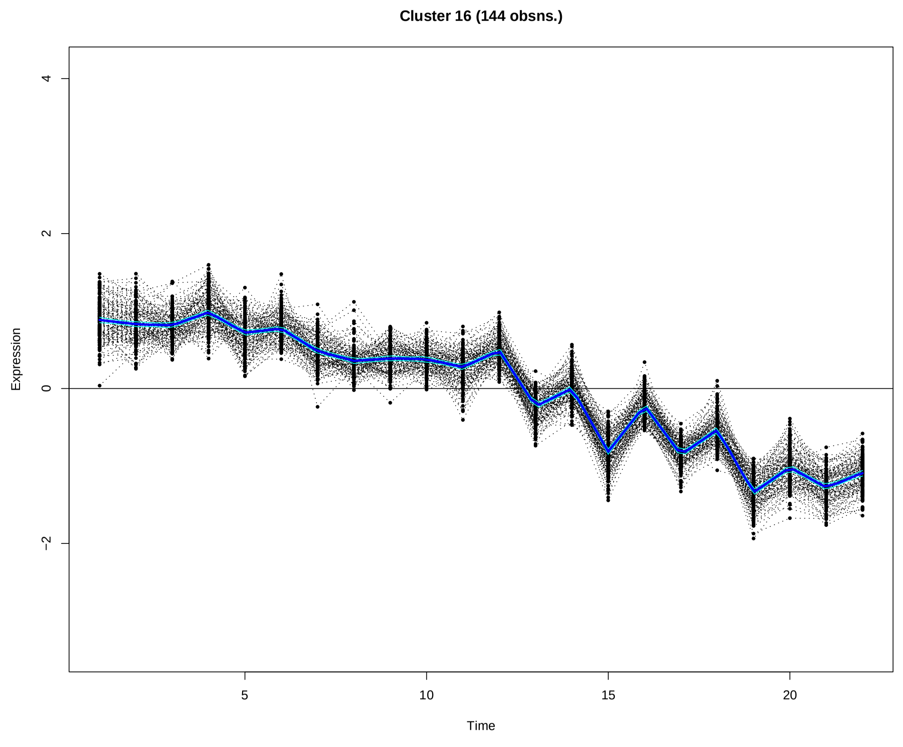
Cluster 12 (11 obsns.)



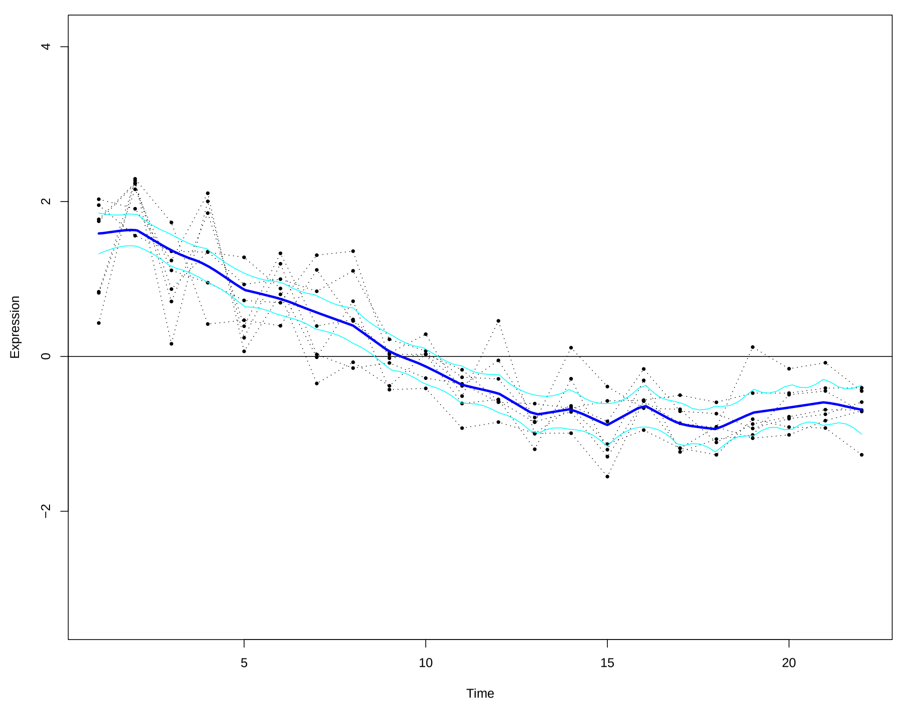

Cluster 13 (19 obsns.)
Expression
4
2
0
−2
5
10
15
20
Time


Cluster 14 (17 obsns.)
Expression
4
2
0
−2
5
10
15
20
Time

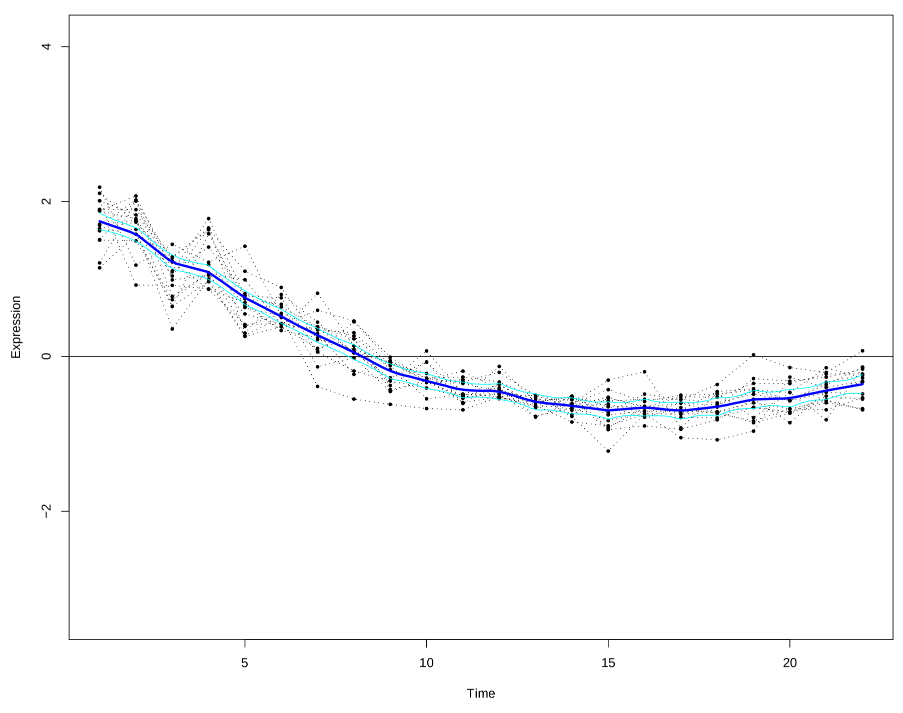

Cluster 15 (7 obsns.)
Expression
4
2
0
−2
5
10
15
20
Time


Cluster 16 (144 obsns.)
Expression
4
2
0
−2
5
10
15
20
Time

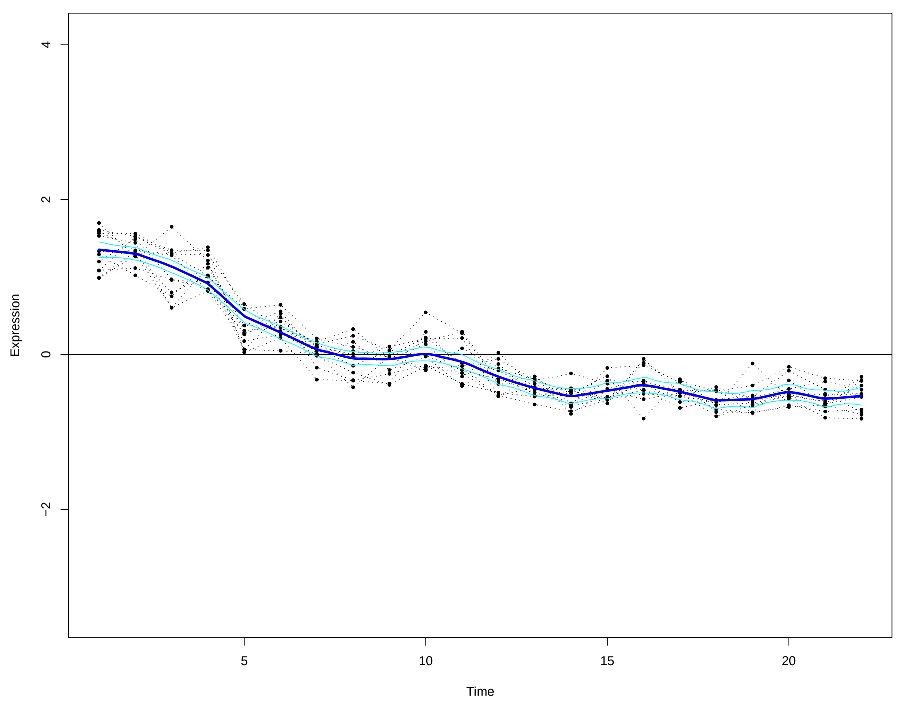
Cluster 17 (7 obsns.)



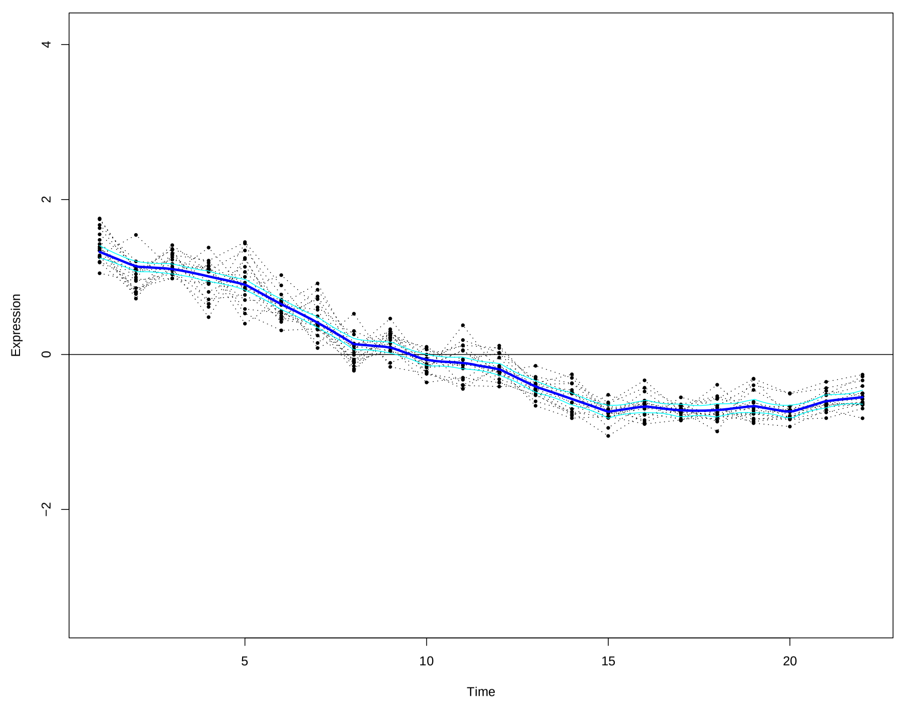
Cluster 18 (15 obsns.)



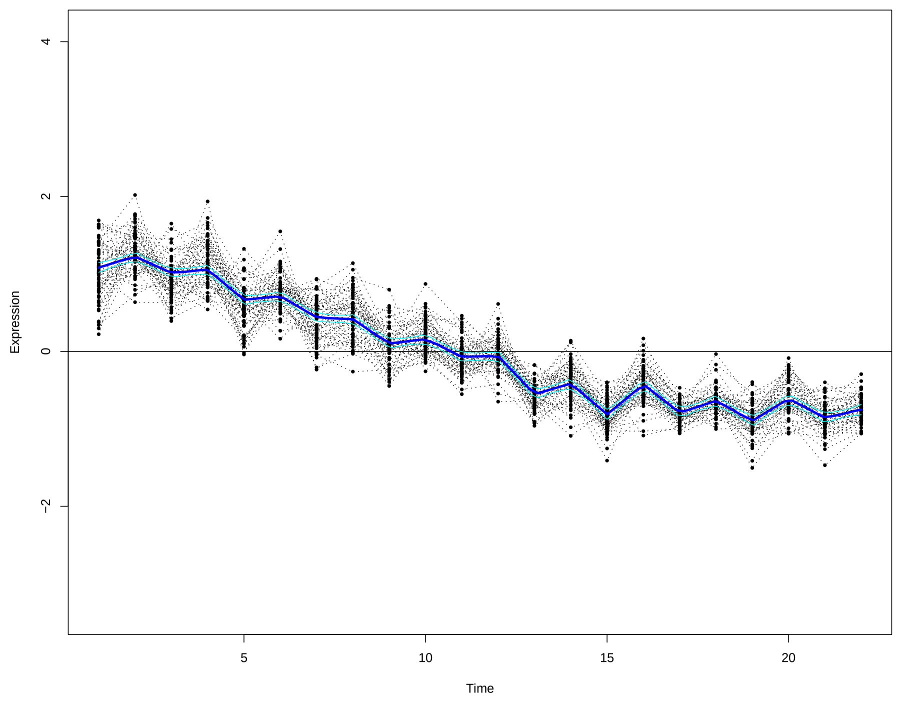
Cluster 19 (12 obsns.)



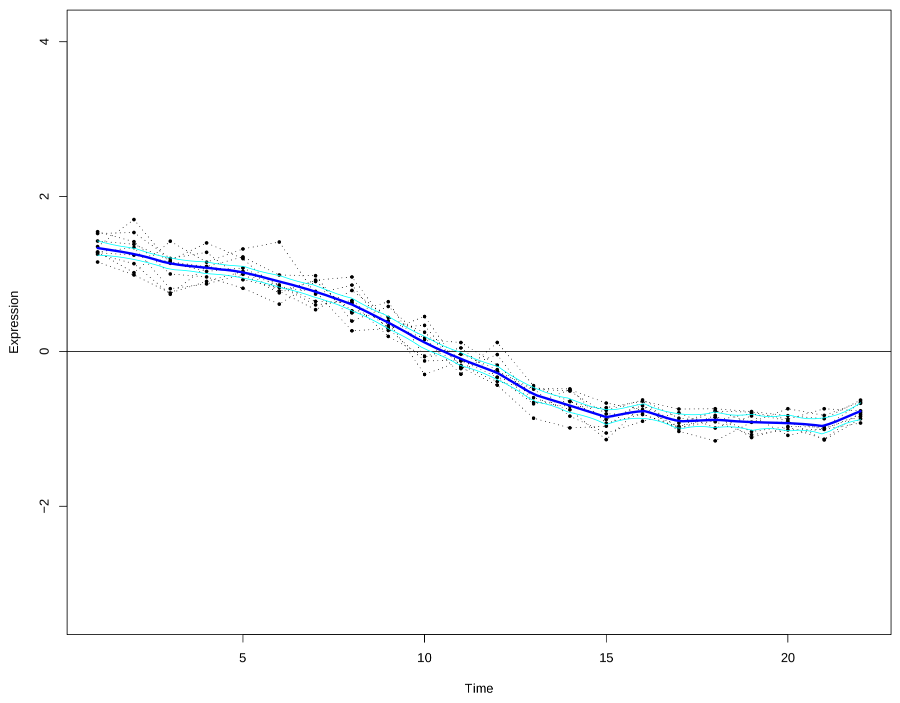
Cluster 20 (17 obsns.)



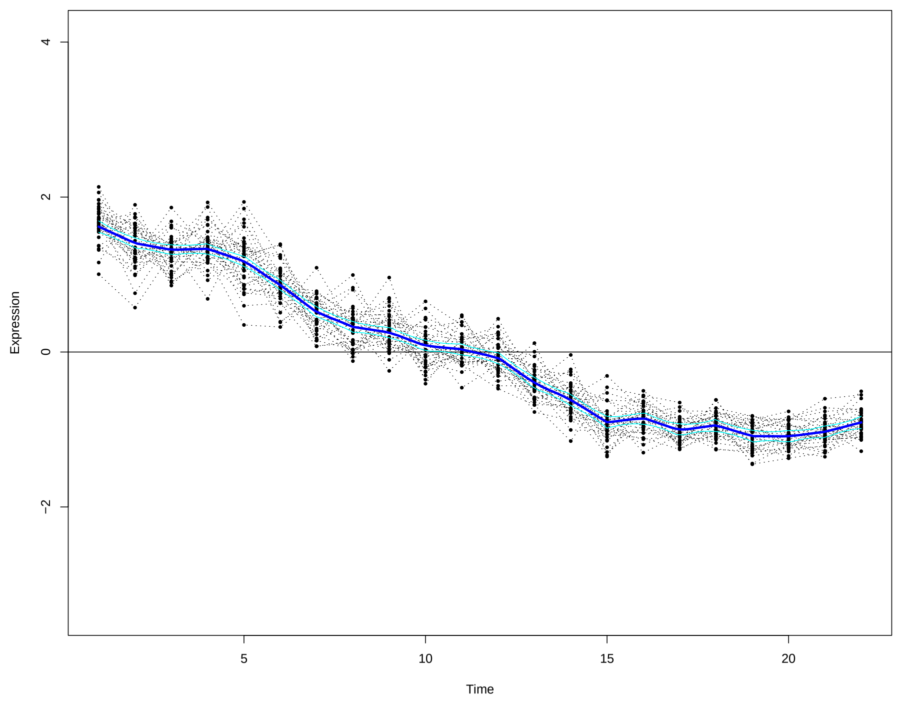
Cluster 21 (60 obsns.)



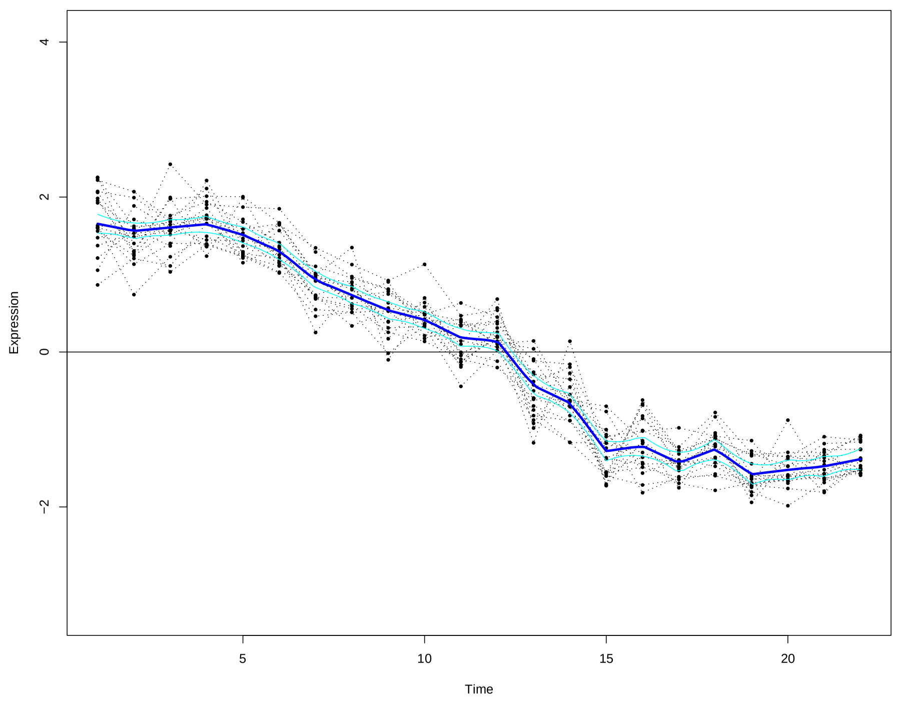
Cluster 22 (9 obsns.)

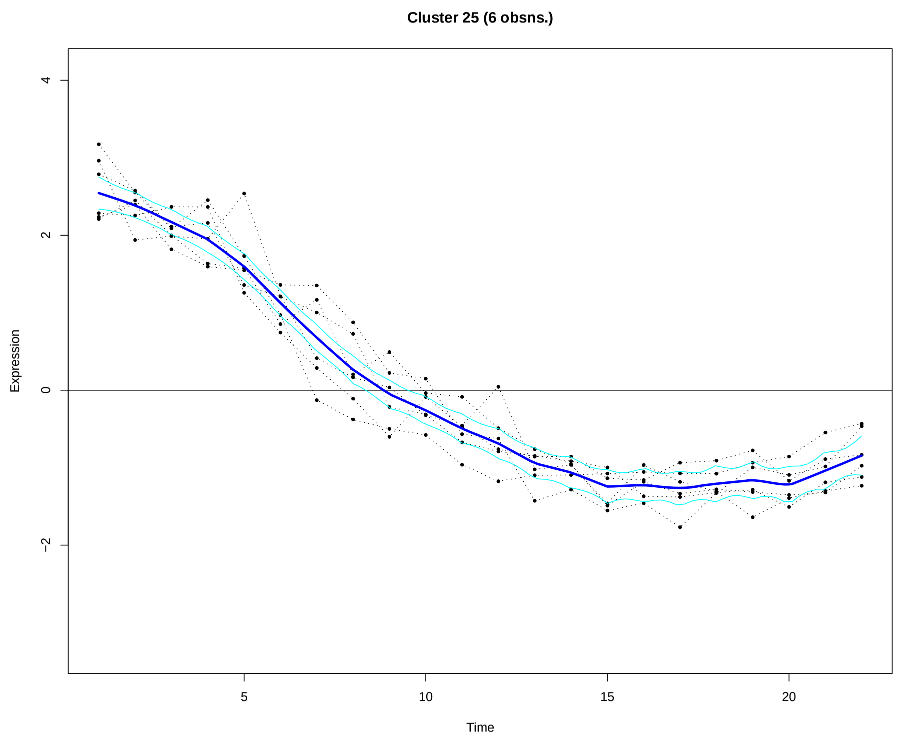


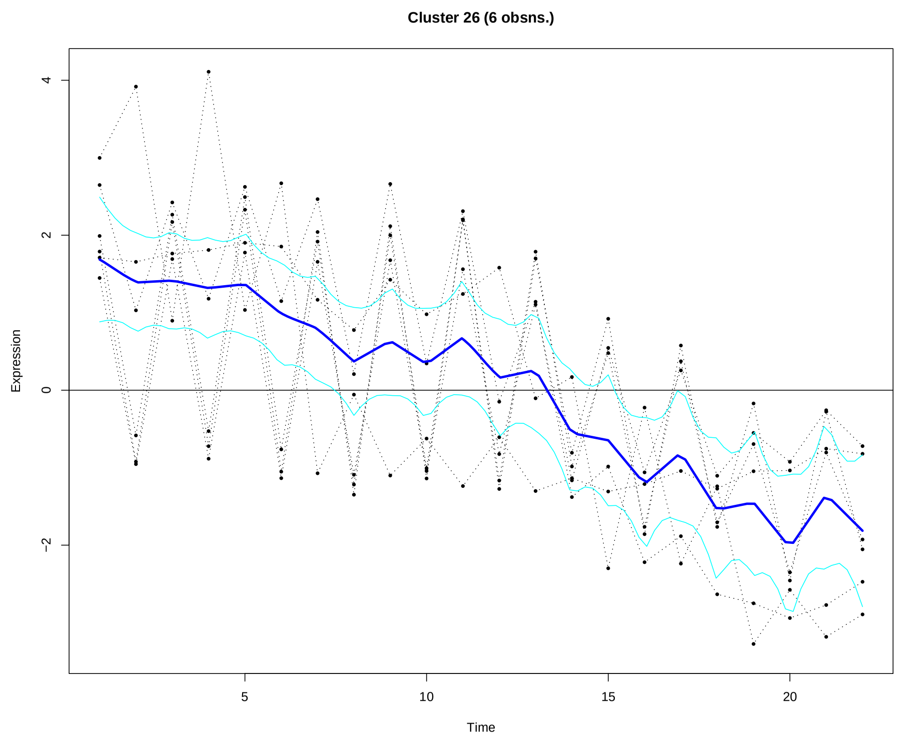
Cluster 23 (34 obsns.)

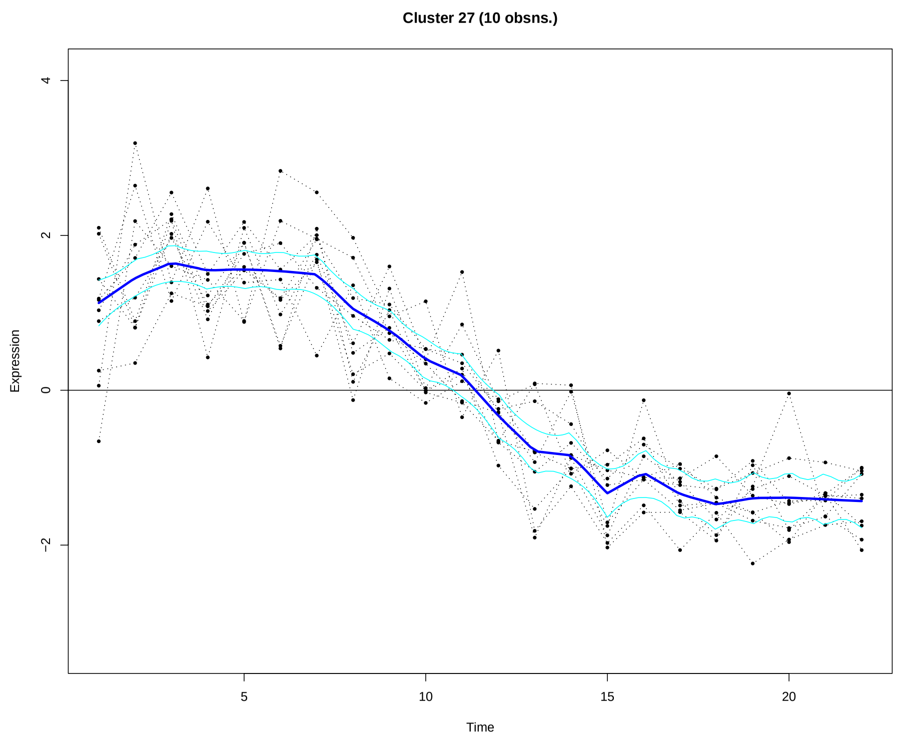


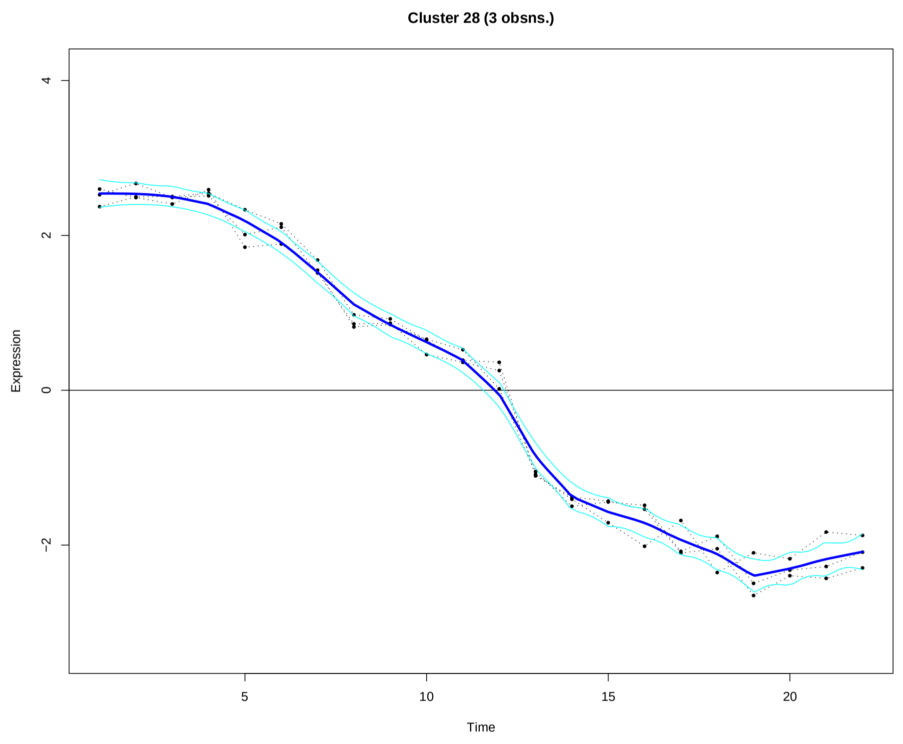
Cluster 24 (18 obsns.)



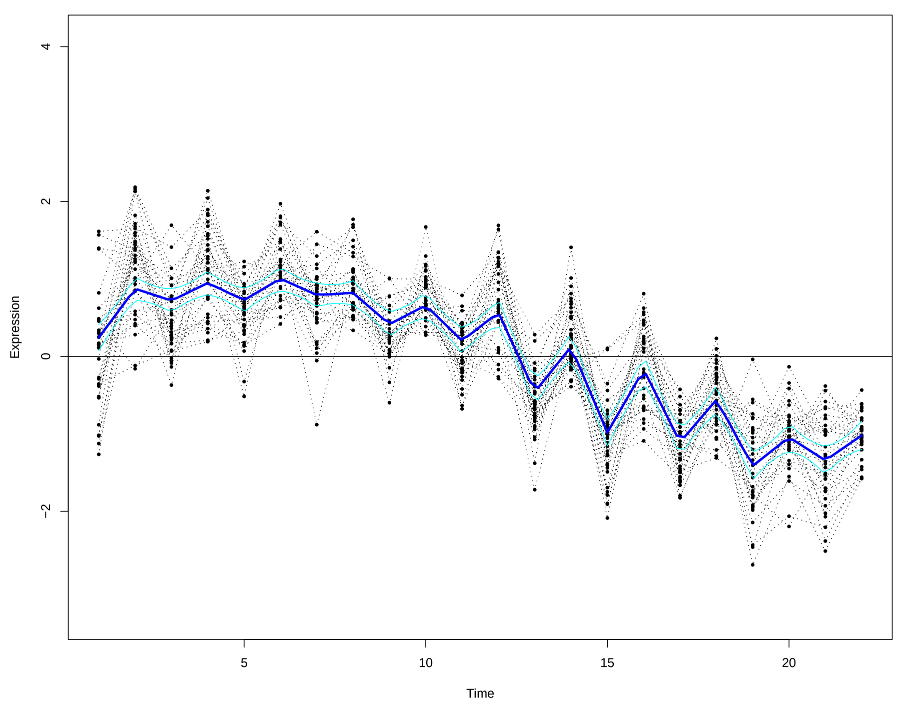

Cluster 25 (6 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20


Cluster 26 (6 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20

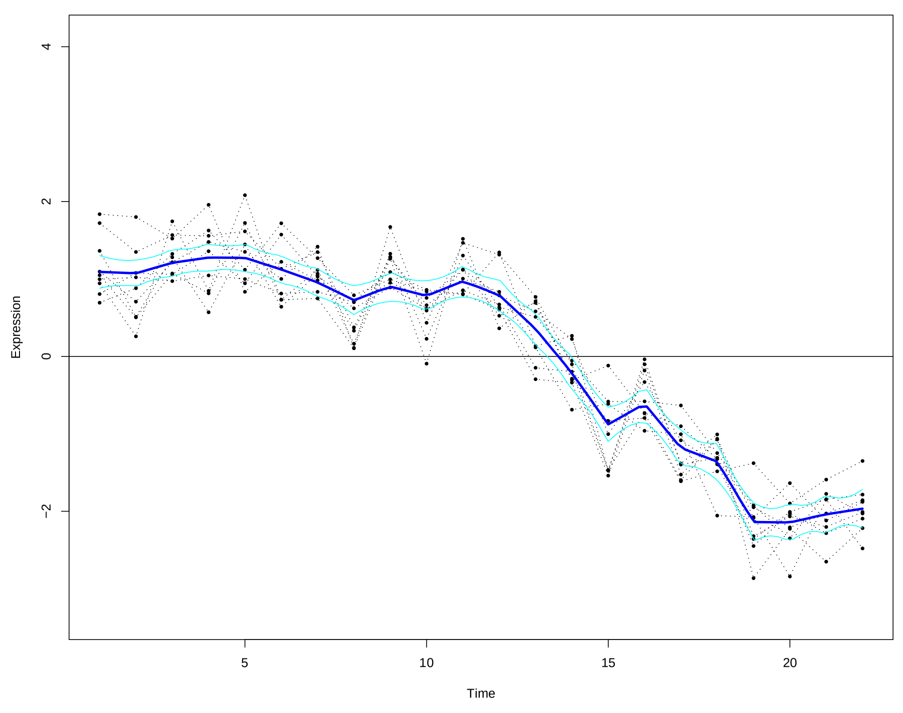

Cluster 27 (10 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20


Cluster 28 (3 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20

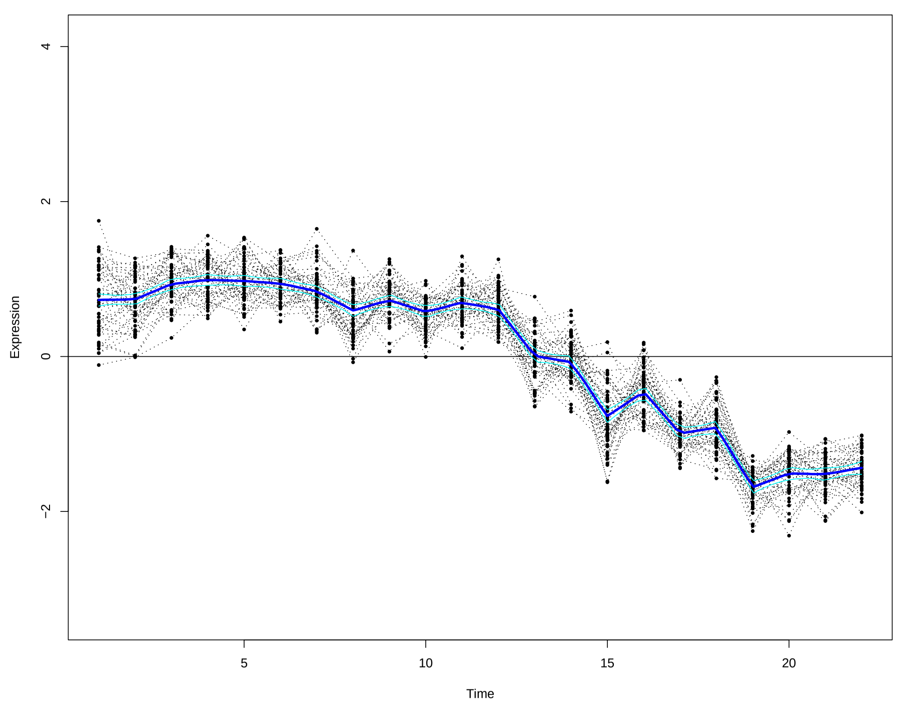
Cluster 29 (33 obsns.)



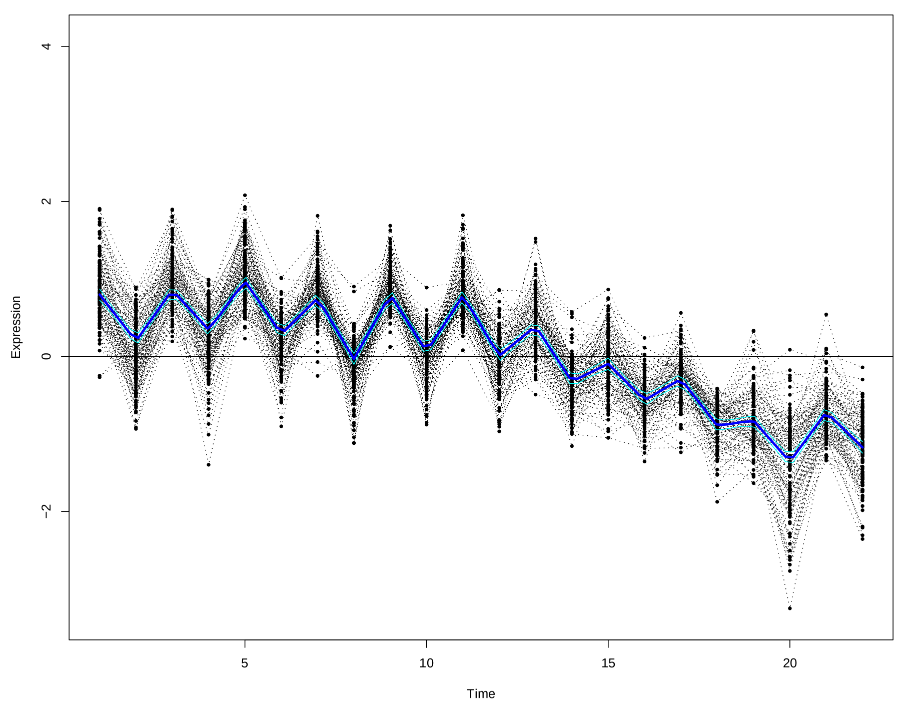
Cluster 30 (9 obsns.)



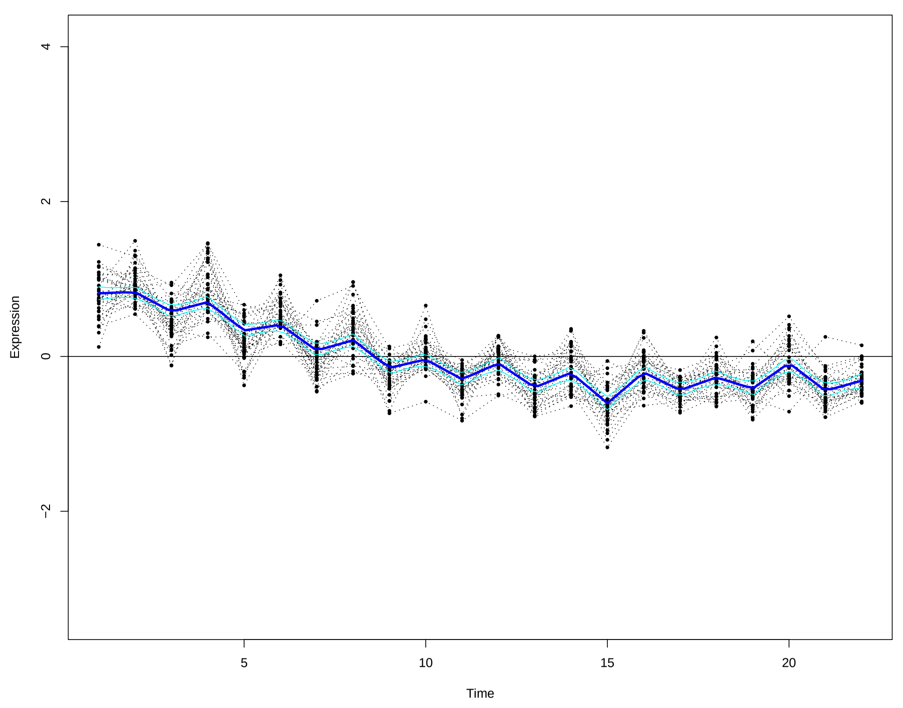
Cluster 31 (50 obsns.)



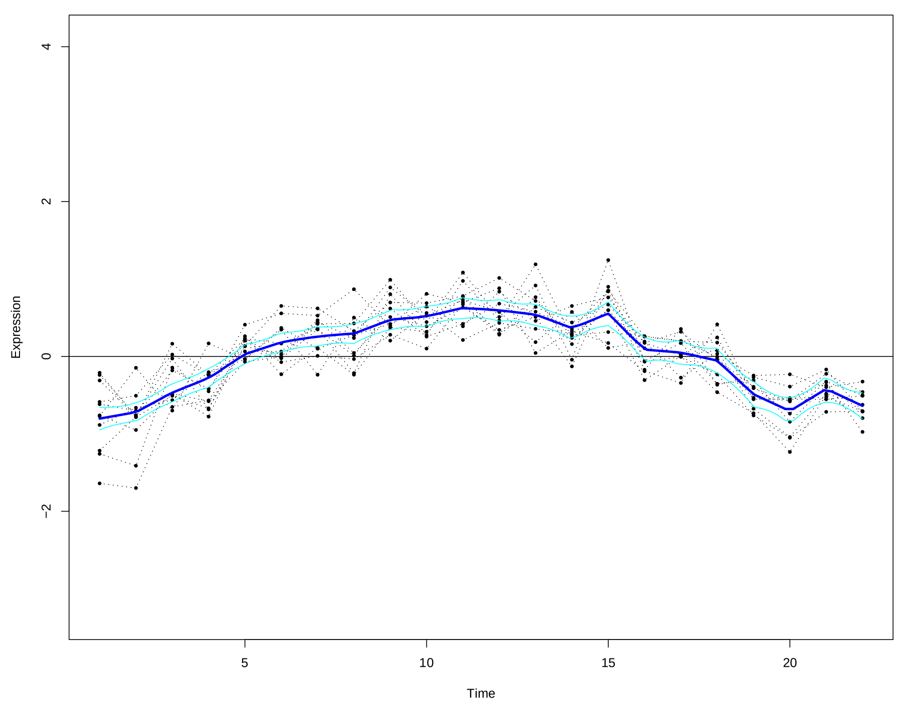
Cluster 32 (121 obsns.)



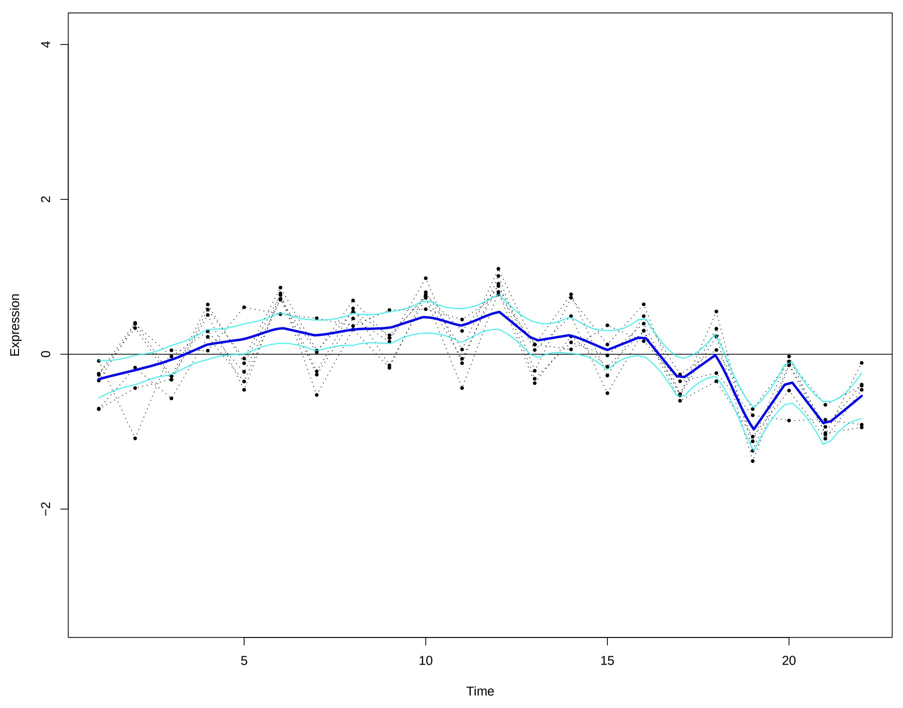
Cluster 33 (35 obsns.)



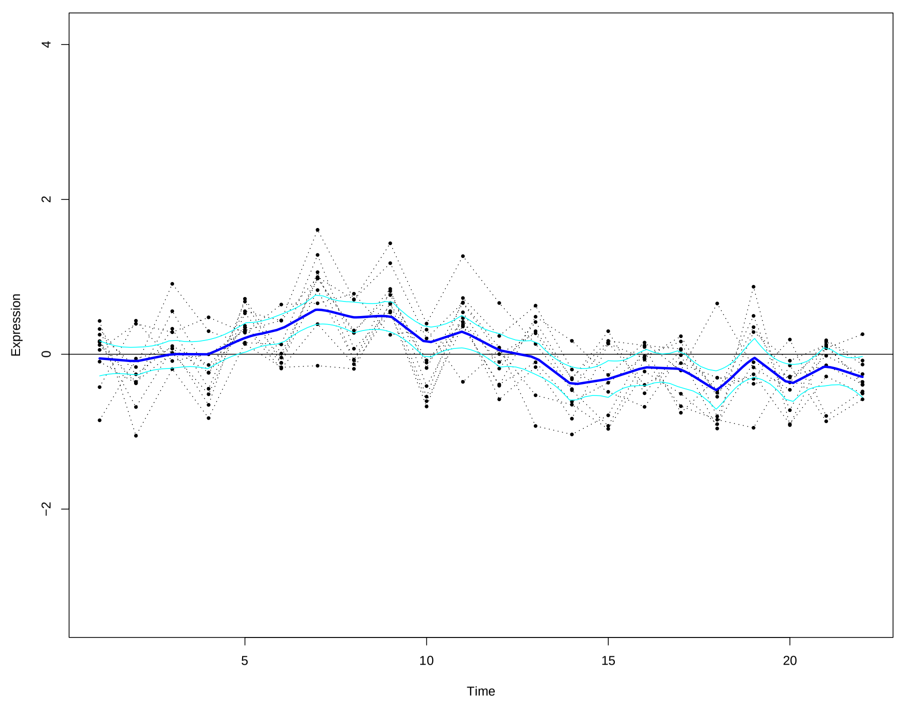
Cluster 34 (11 obsns.)



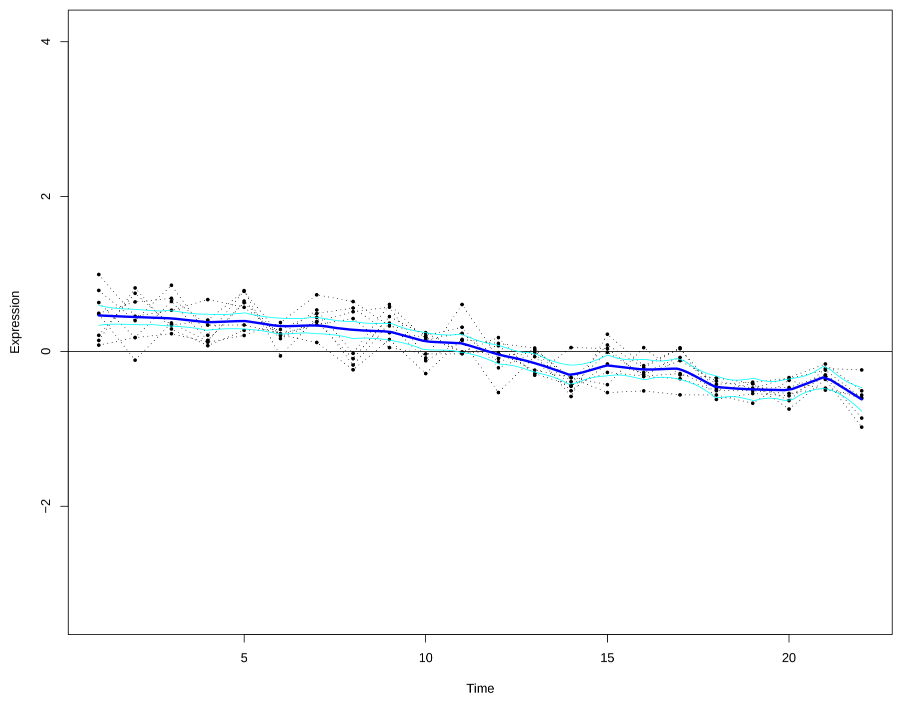
Cluster 35 (6 obsns.)



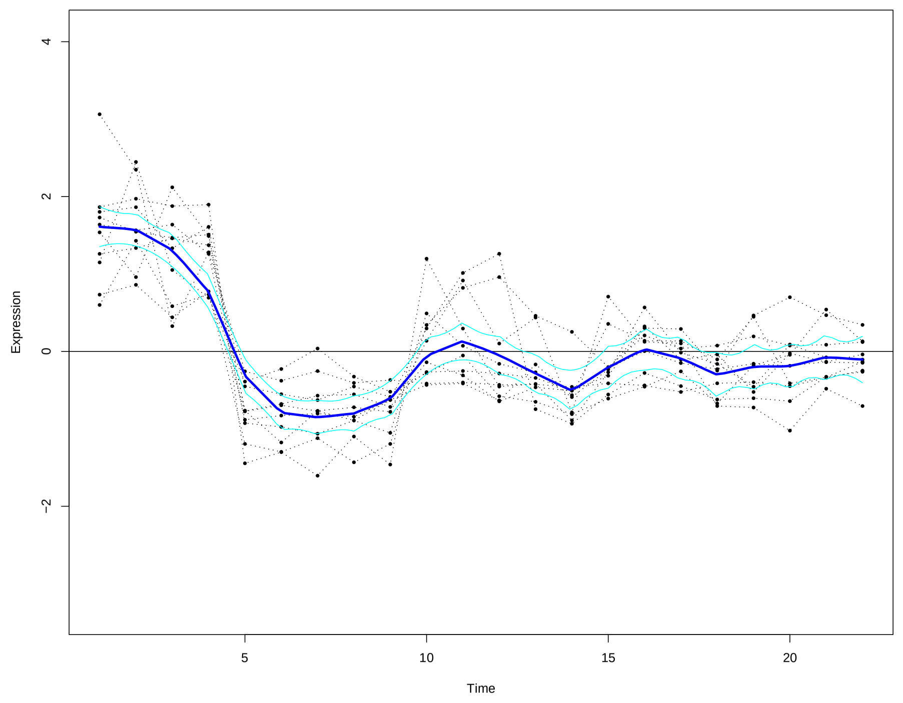
Cluster 36 (10 obsns.)



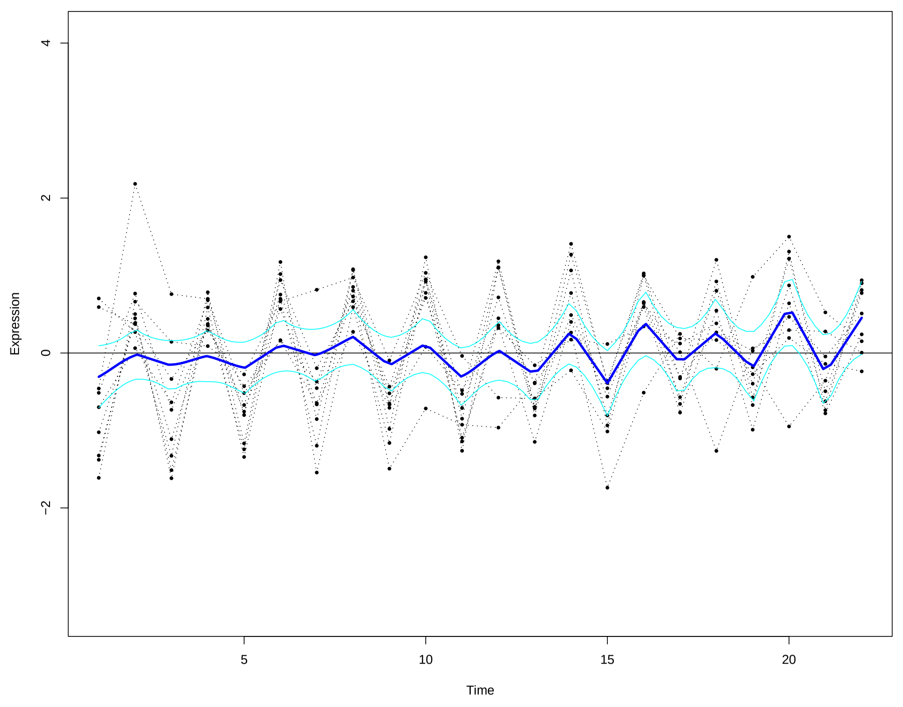
Cluster 37 (8 obsns.)



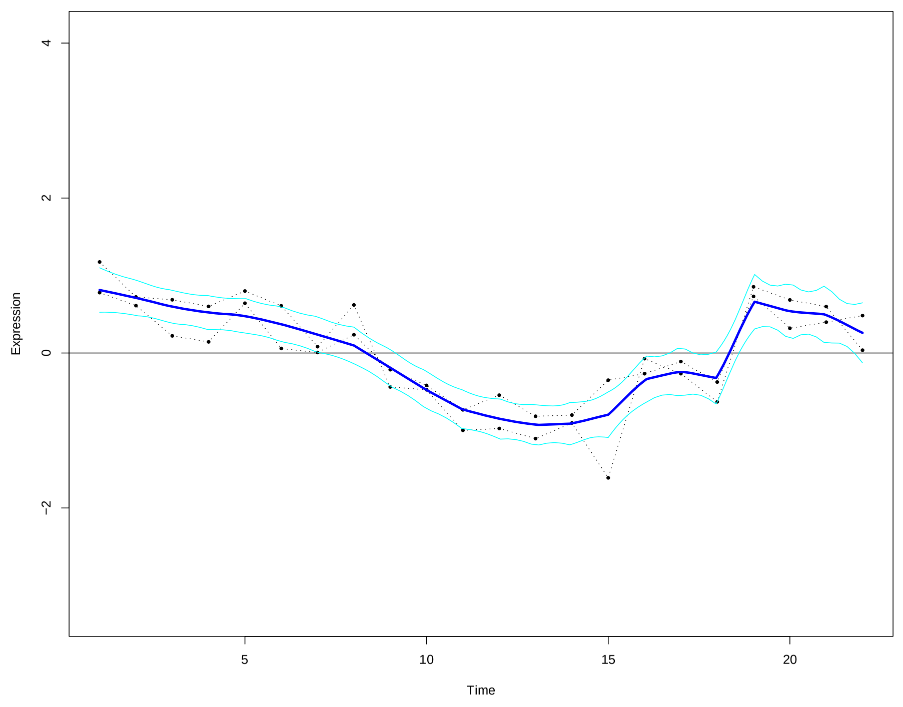
Cluster 38 (10 obsns.)

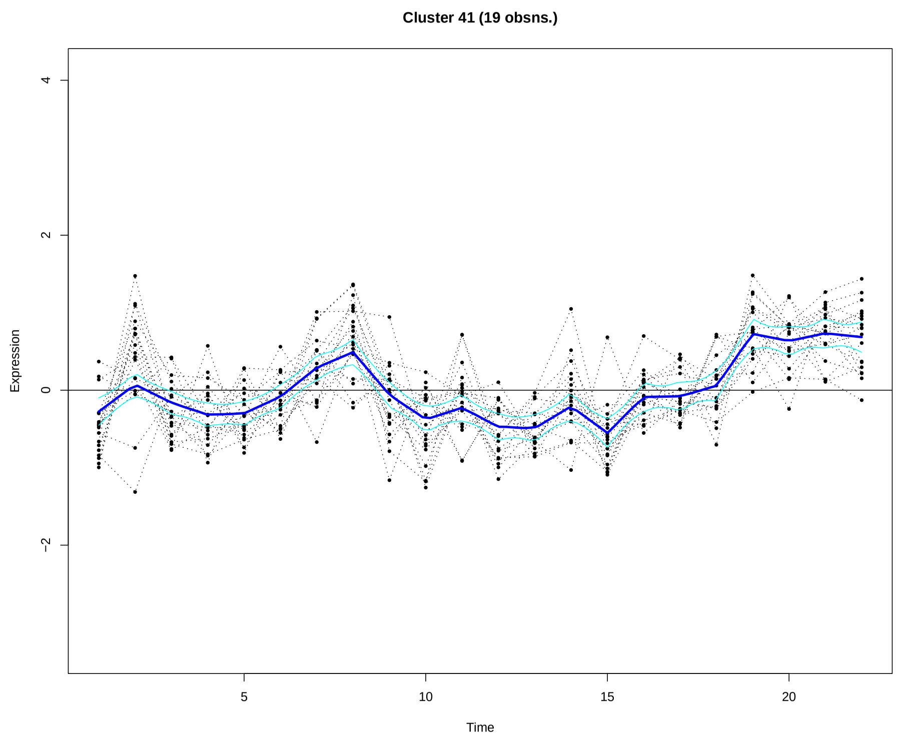
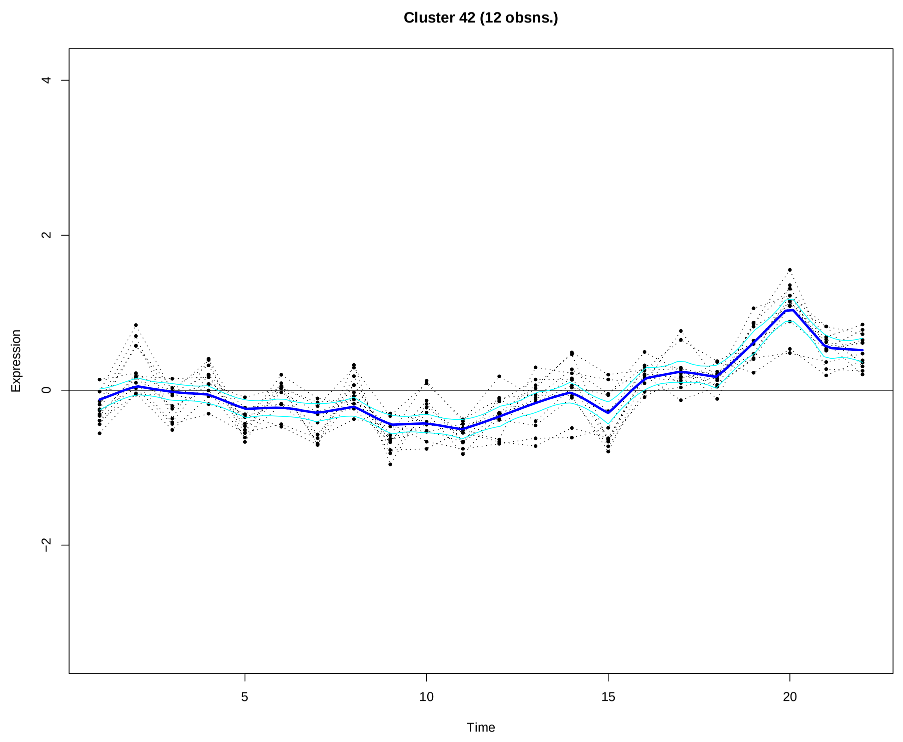
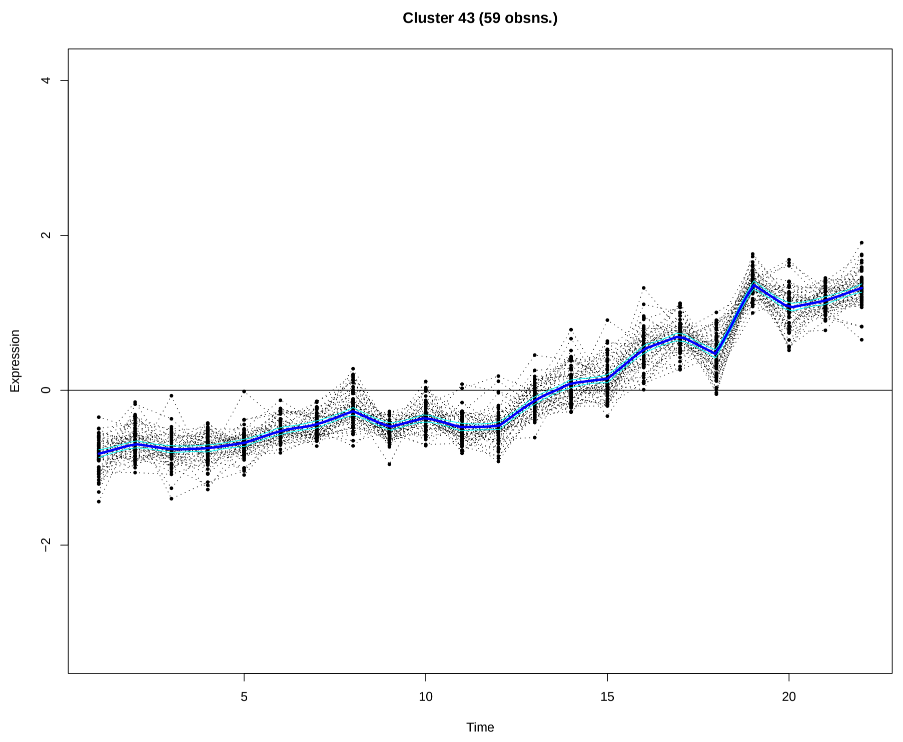
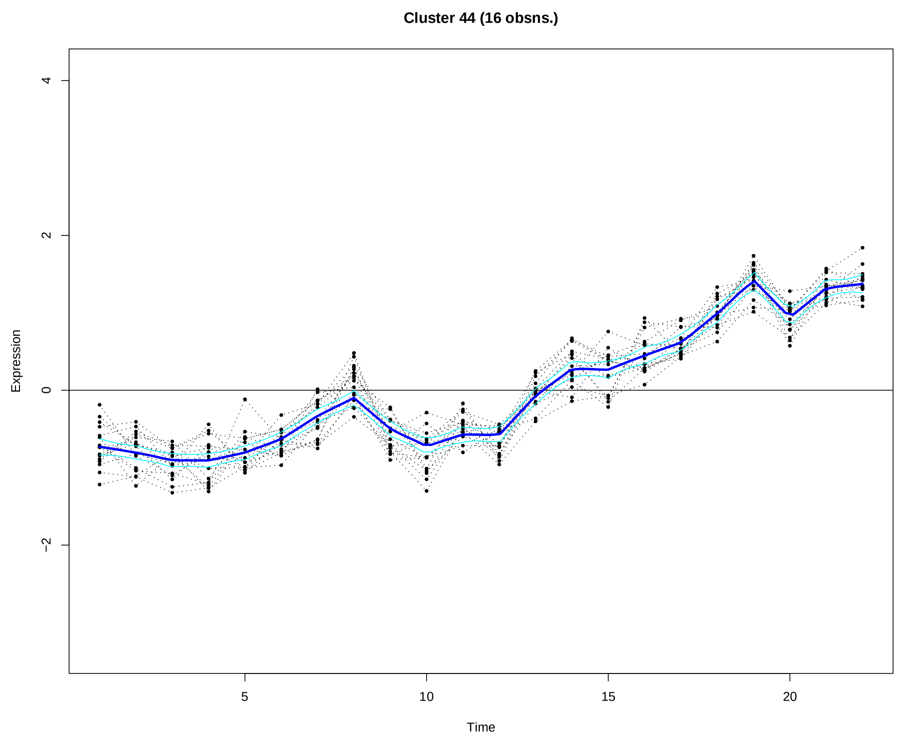


Cluster 39 (9 obsns.)

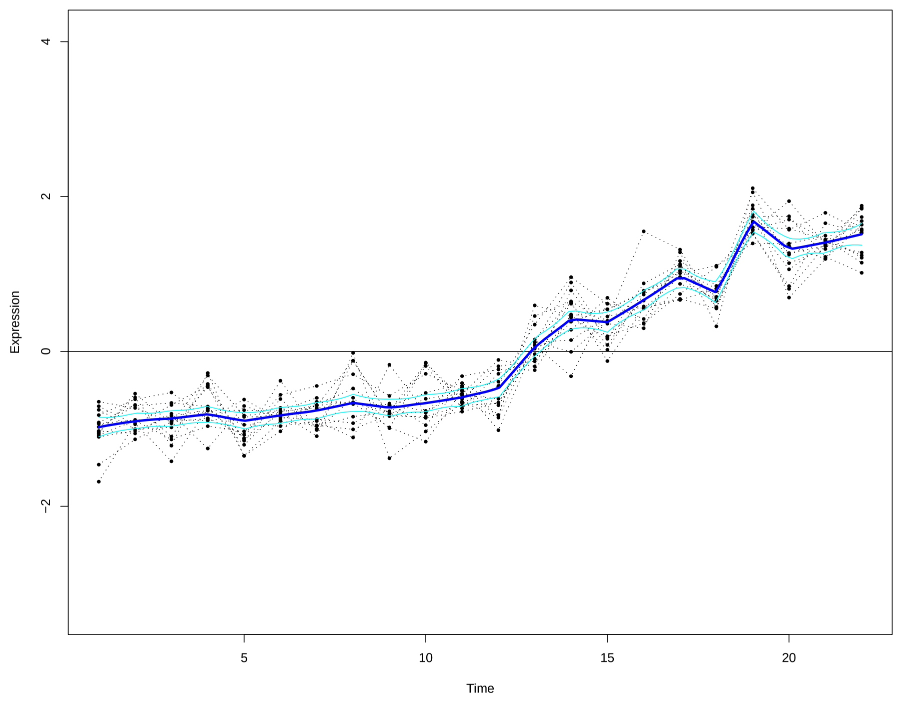


Cluster 40 (2 obsns.)

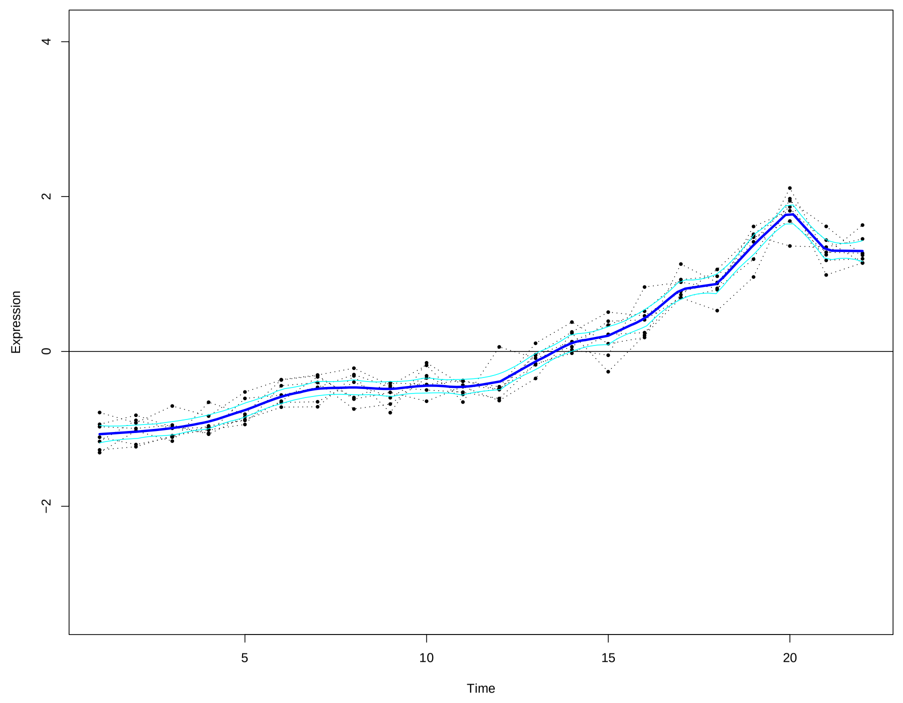



Cluster 41 (19 obsns.)
Expression
Time
0
2
4
-2
5
10
15
20

Cluster 42 (12 obsns.)
Expression
Time
0
2
4
-2
5
10
15
20

Cluster 43 (59 obsns.)
Expression
Time
0
2
4
-2
5
10
15
20

Cluster 44 (16 obsns.)
Expression
Time
0
2
4
-2
5
10
15
20

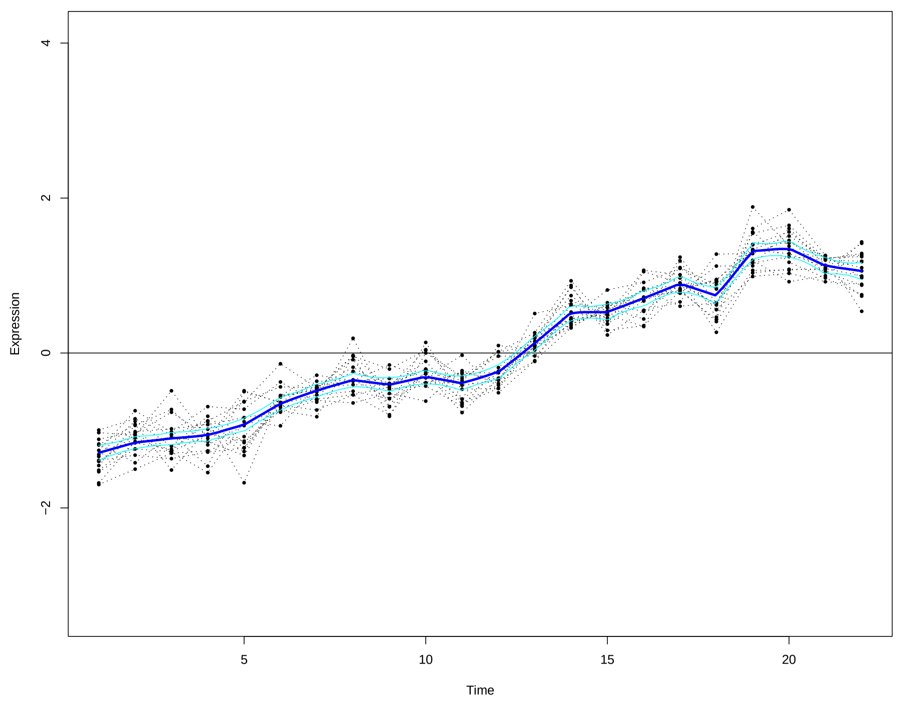
Cluster 45 (14 obsns.)



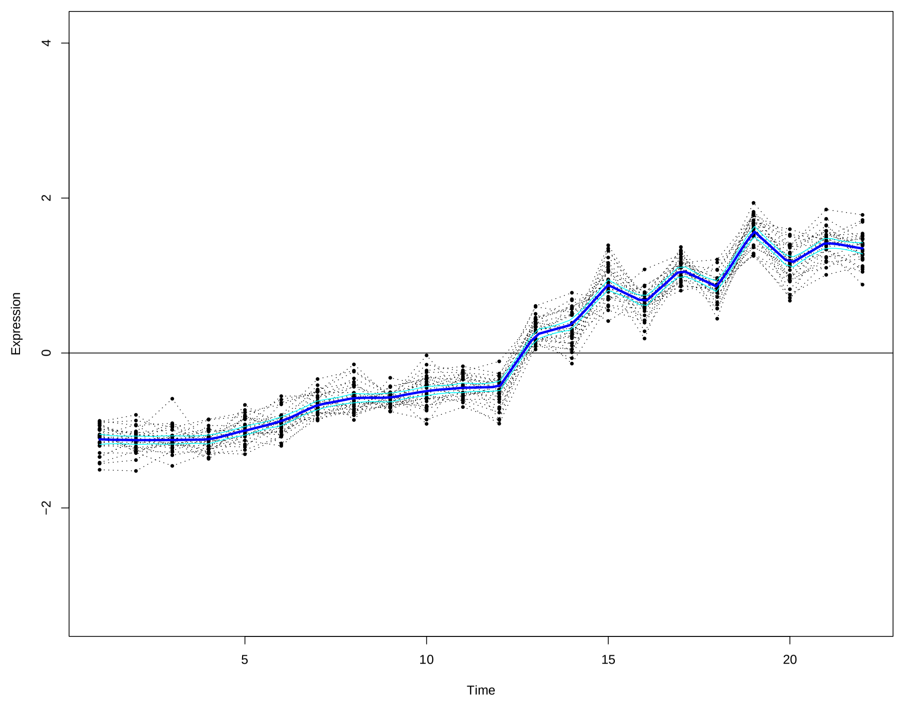
Cluster 46 (7 obsns.)



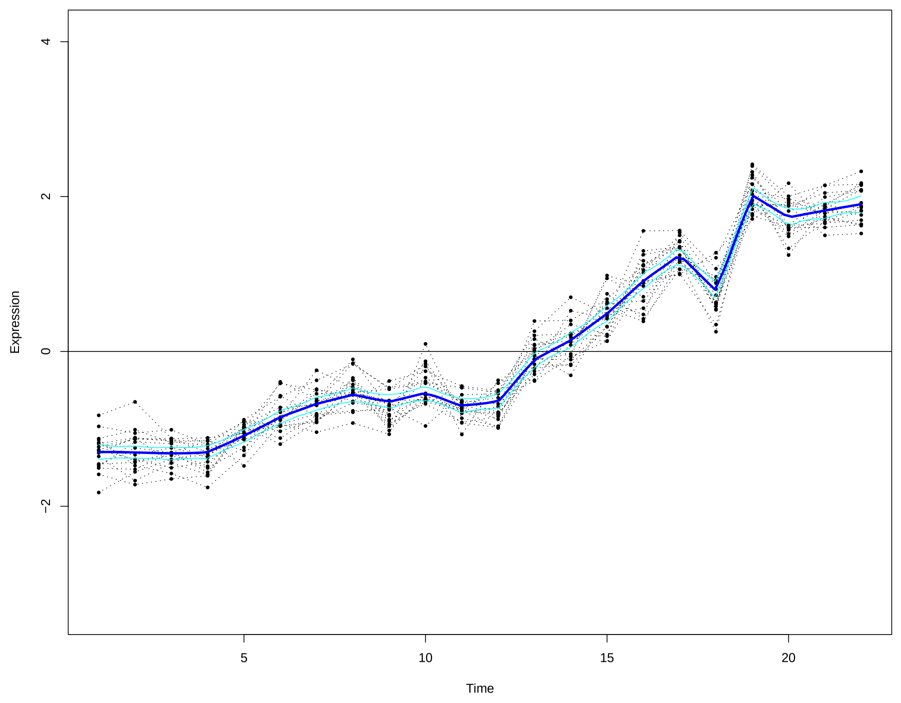
Cluster 47 (16 obsns.)



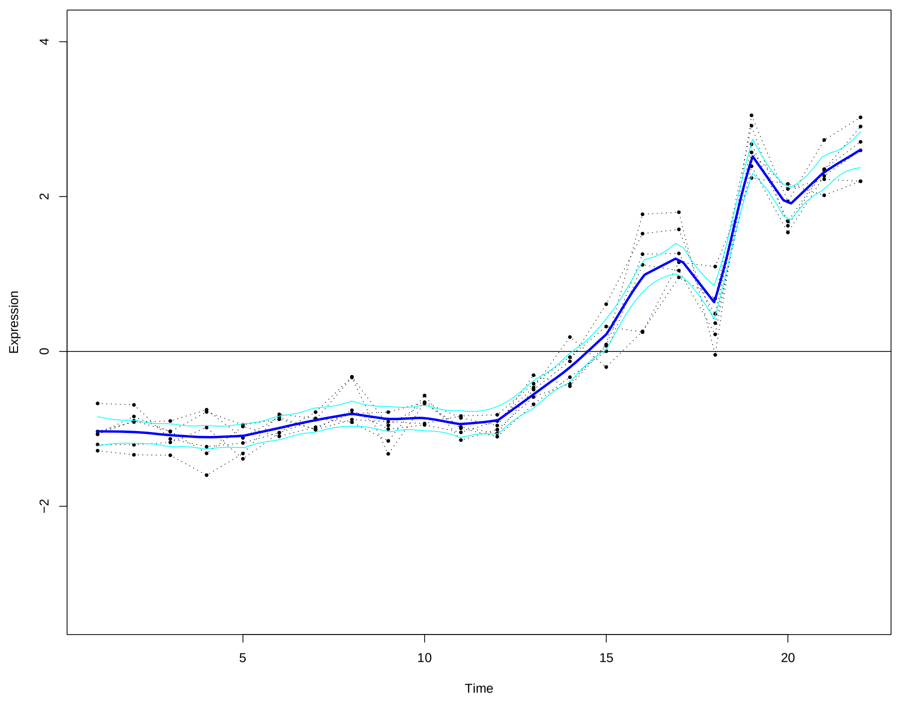
Cluster 48 (28 obsns.)



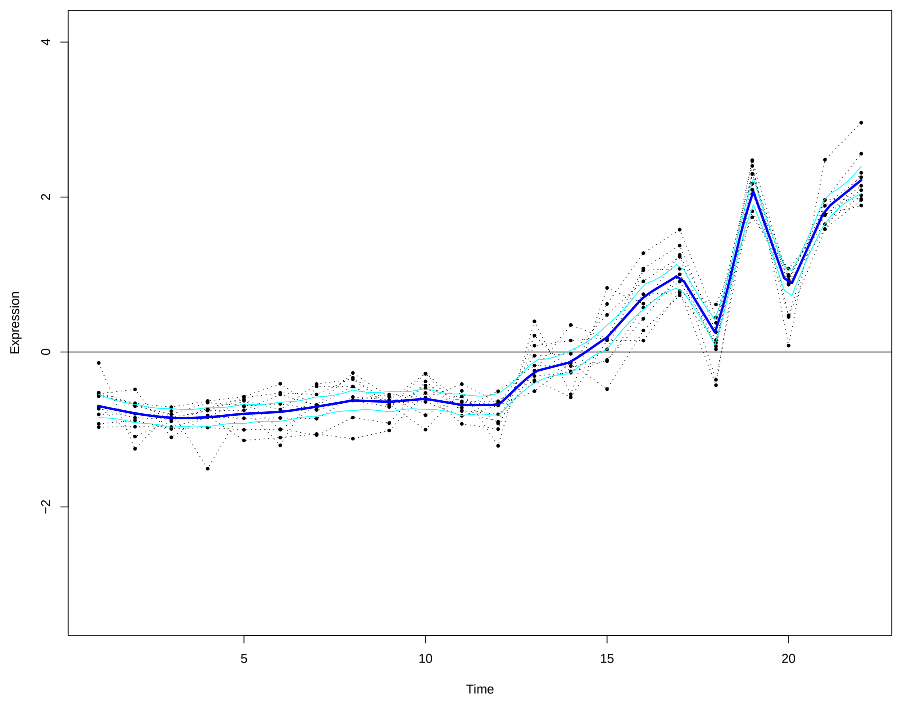
Cluster 49 (18 obsns.)



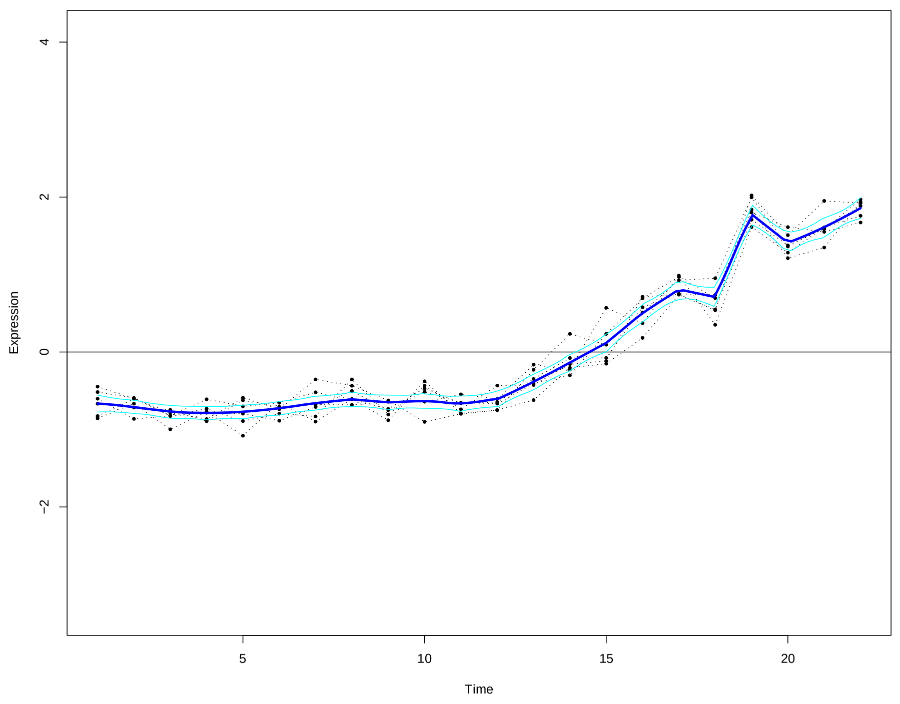
Cluster 50 (6 obsns.)



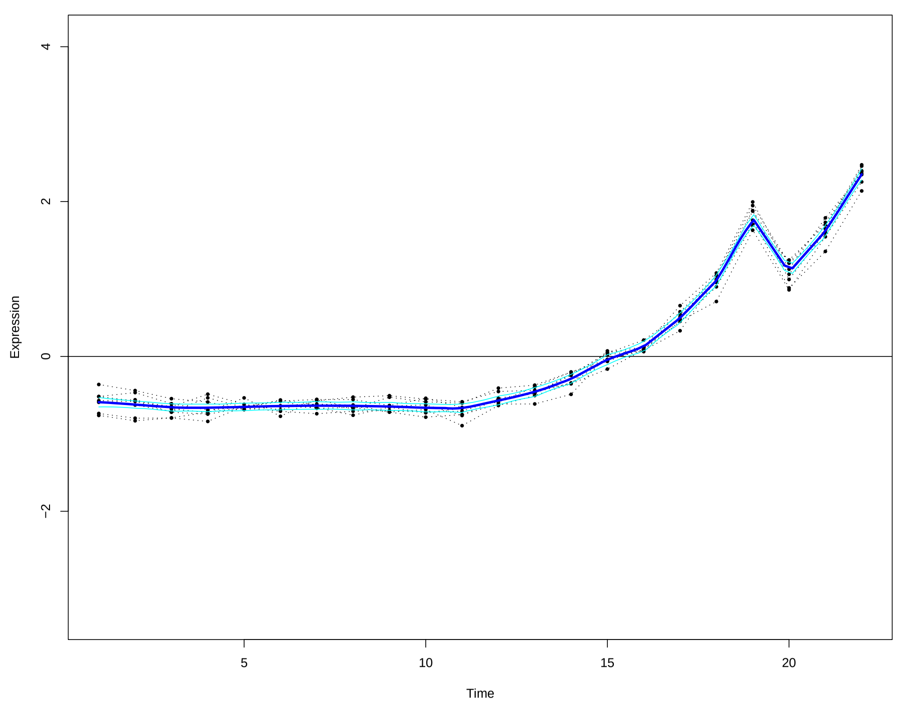
Cluster 51 (10 obsns.)



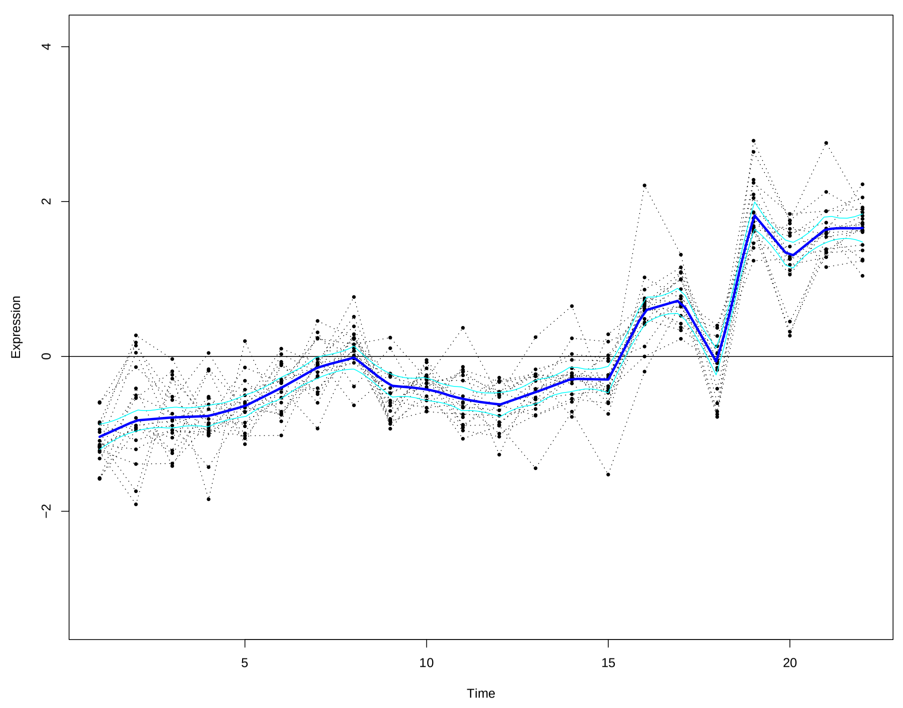
Cluster 52 (6 obsns.)



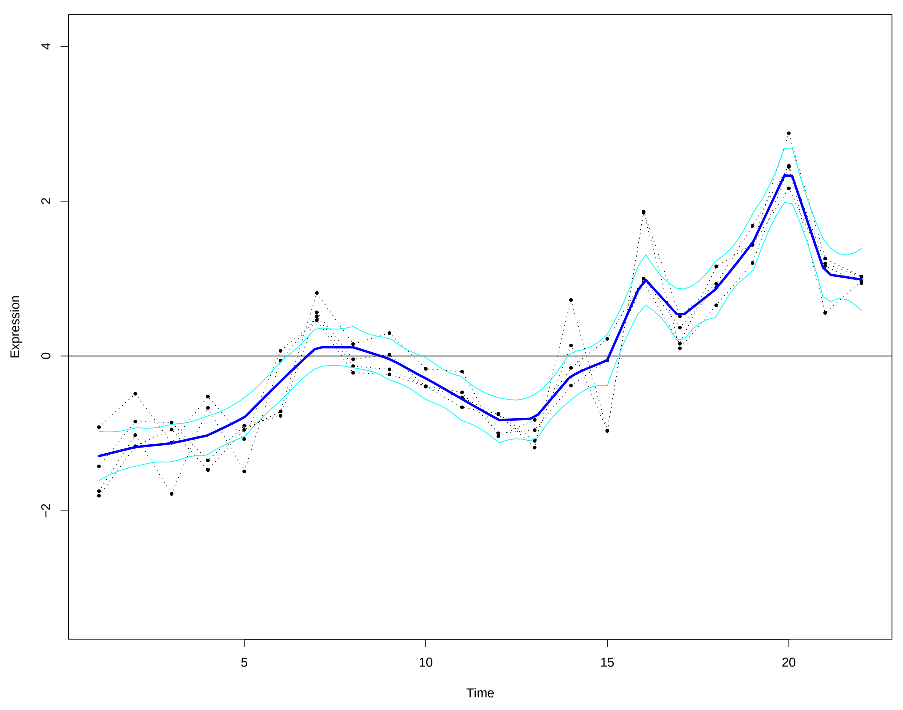
Cluster 53 (8 obsns.)



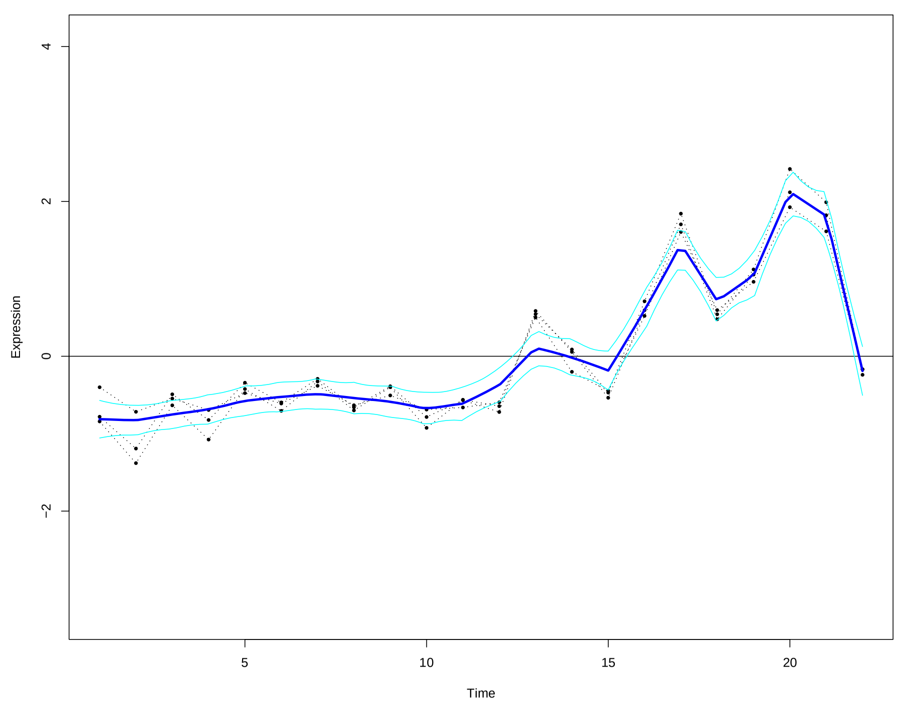
Cluster 54 (18 obsns.)



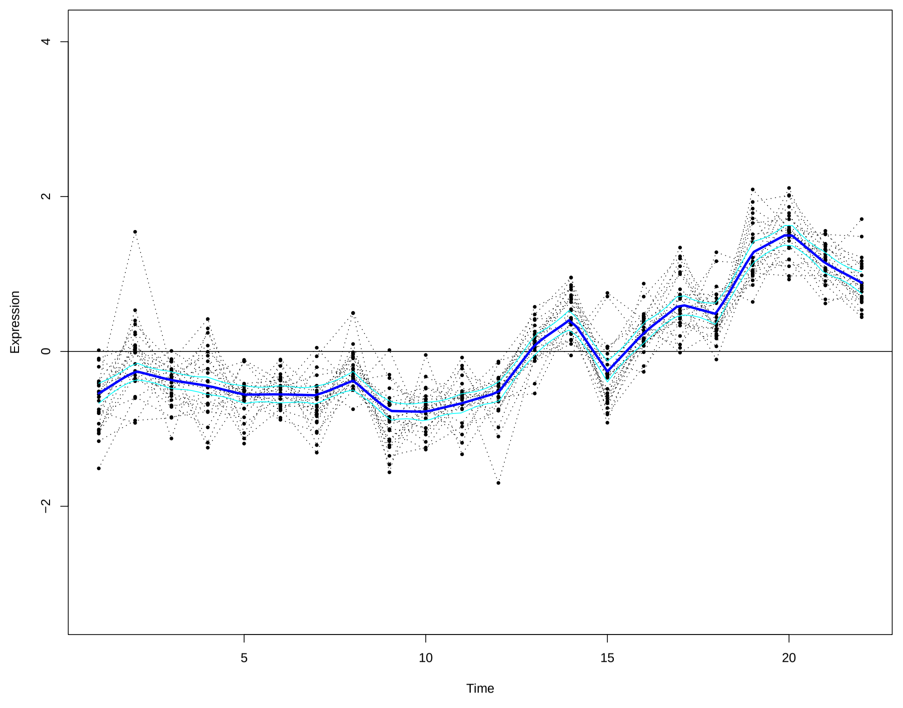
Cluster 55 (4 obsns.)



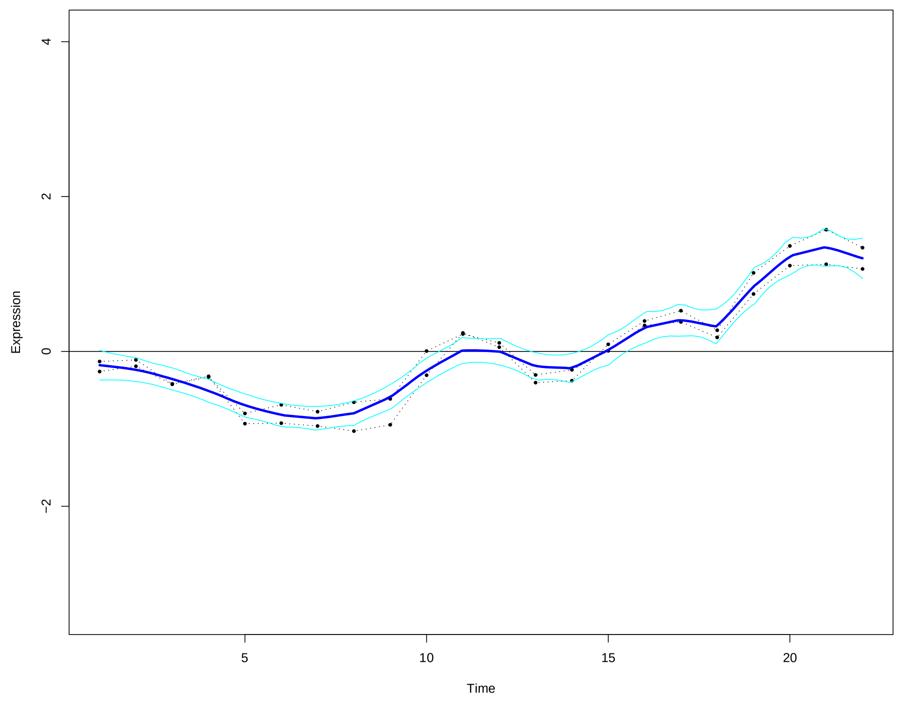
Cluster 56 (3 obsns.)



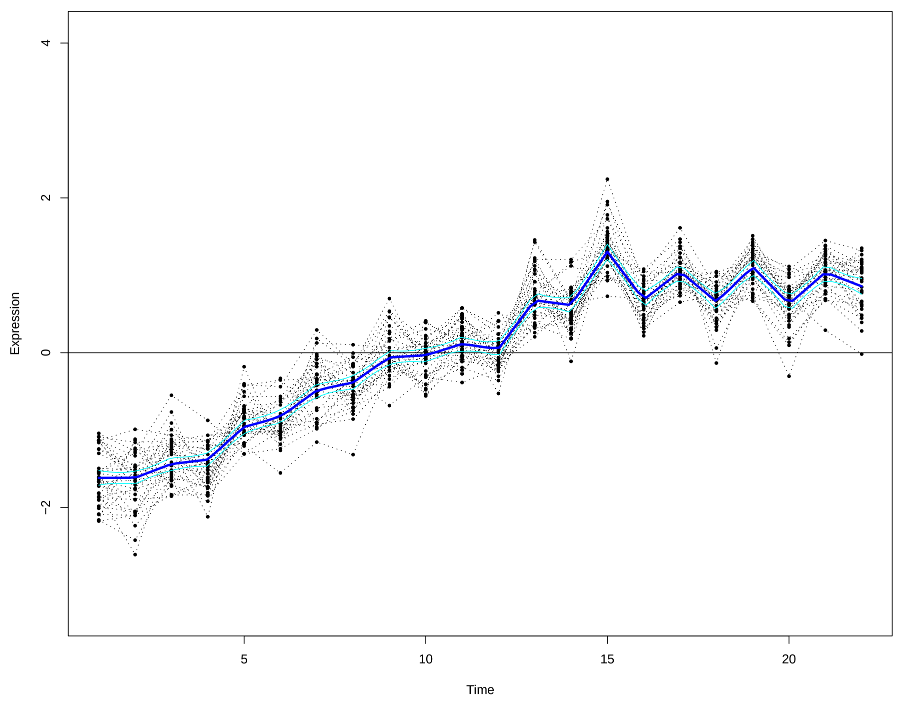
Cluster 57 (24 obsns.)



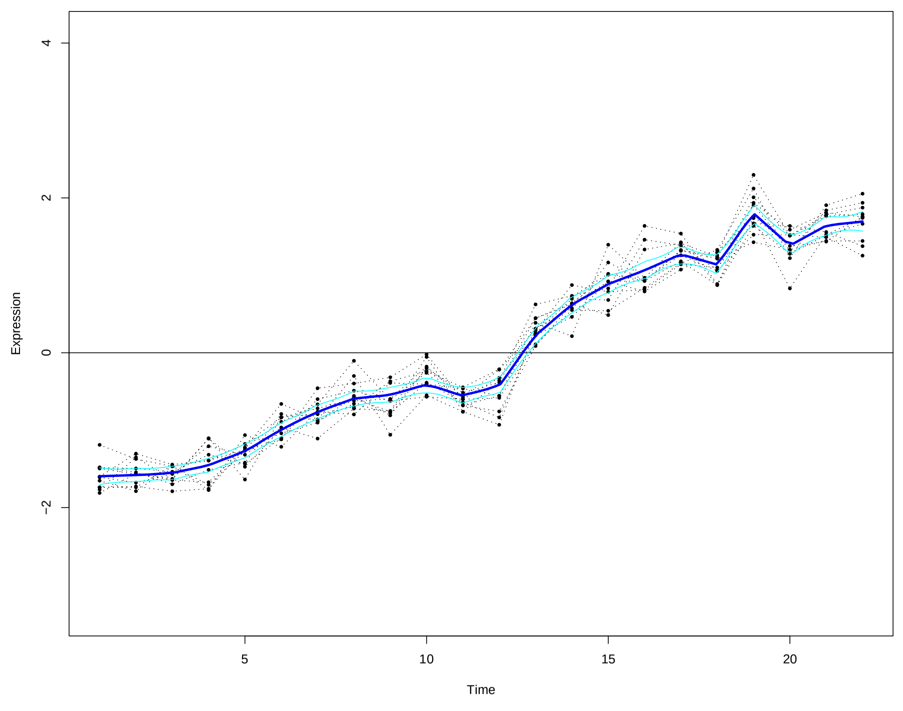
Cluster 58 (2 obsns.)

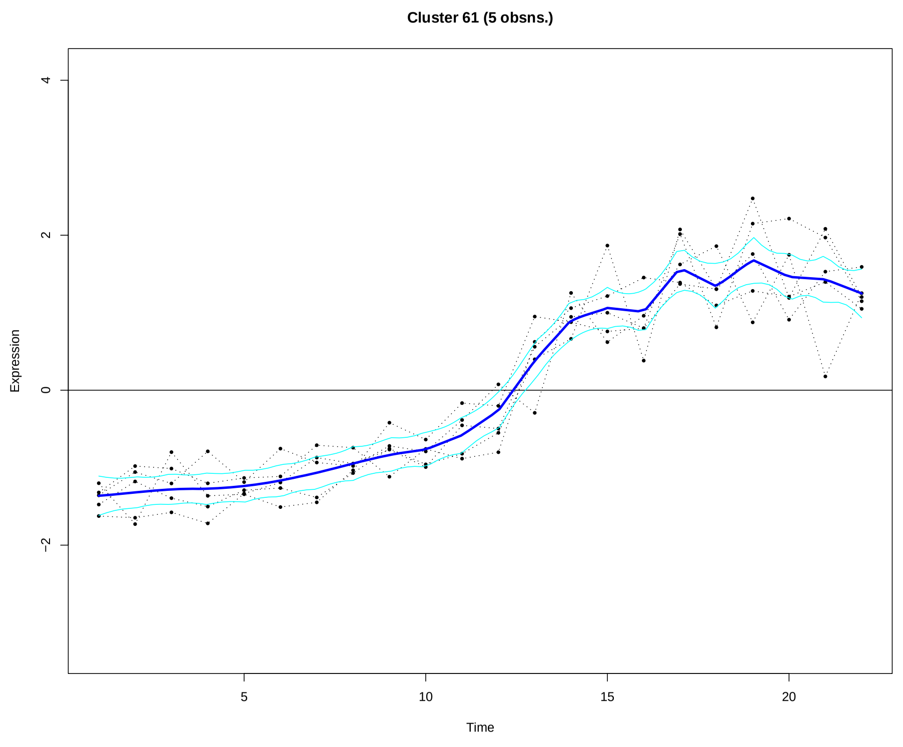


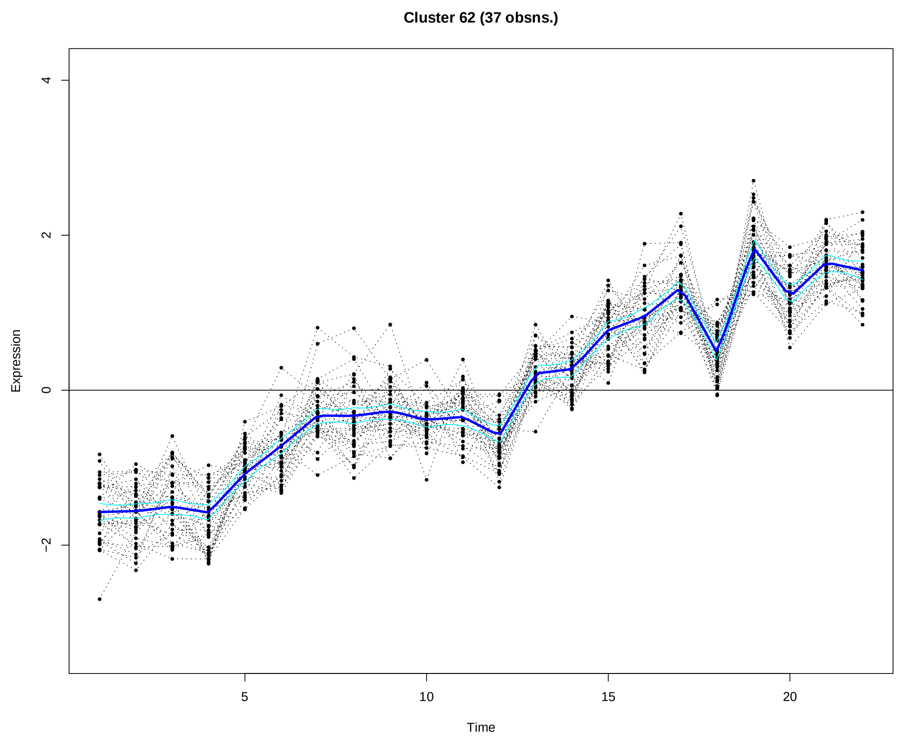
Cluster 59 (35 obsns.)

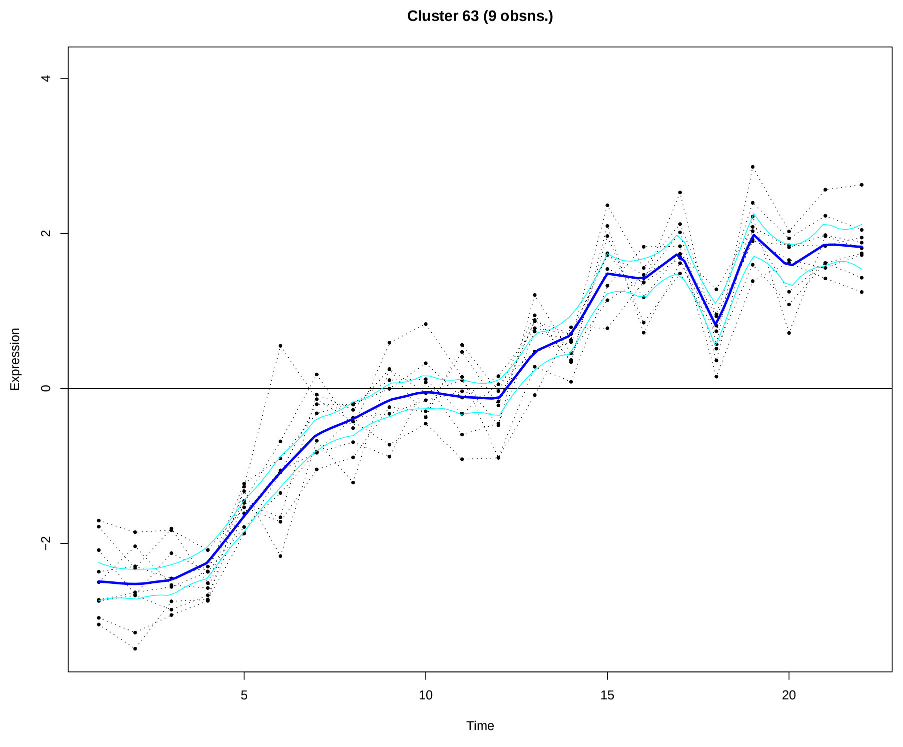


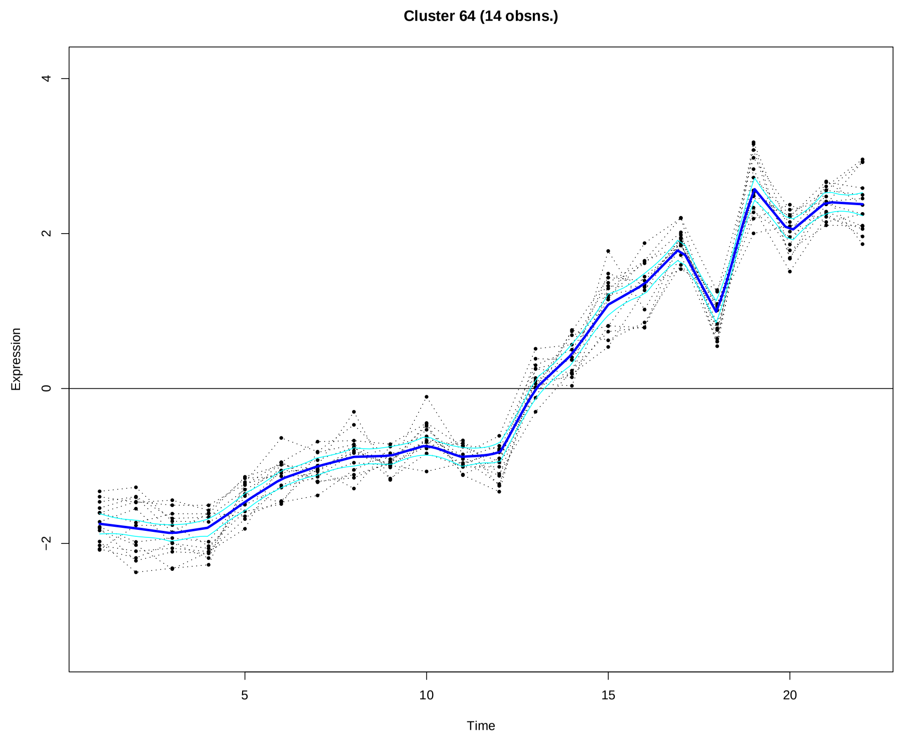
Cluster 60 (11 obsns.)



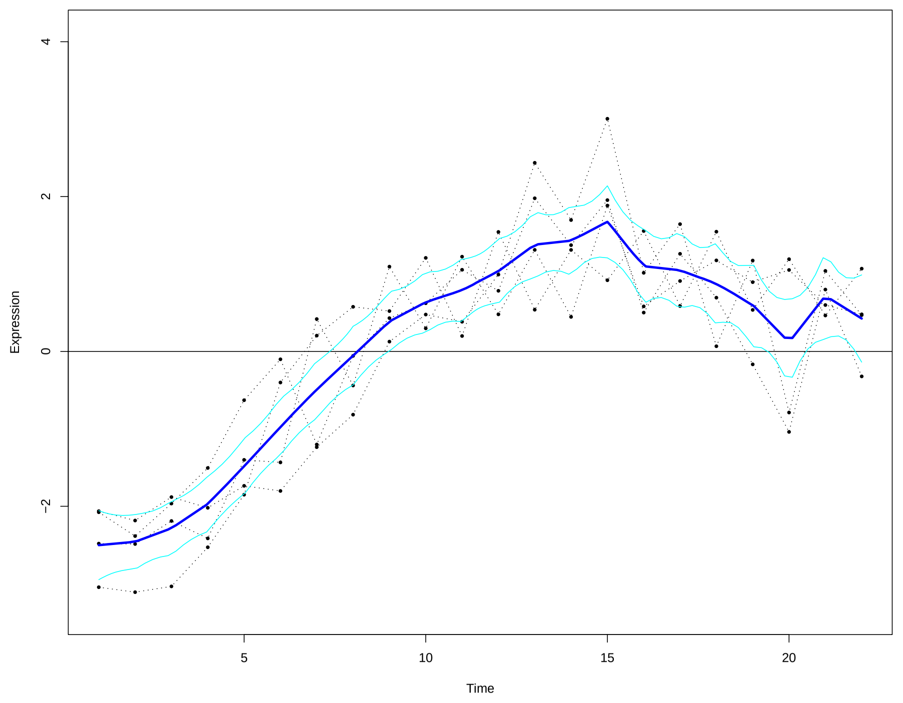

Cluster 61 (5 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20


Cluster 62 (37 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20

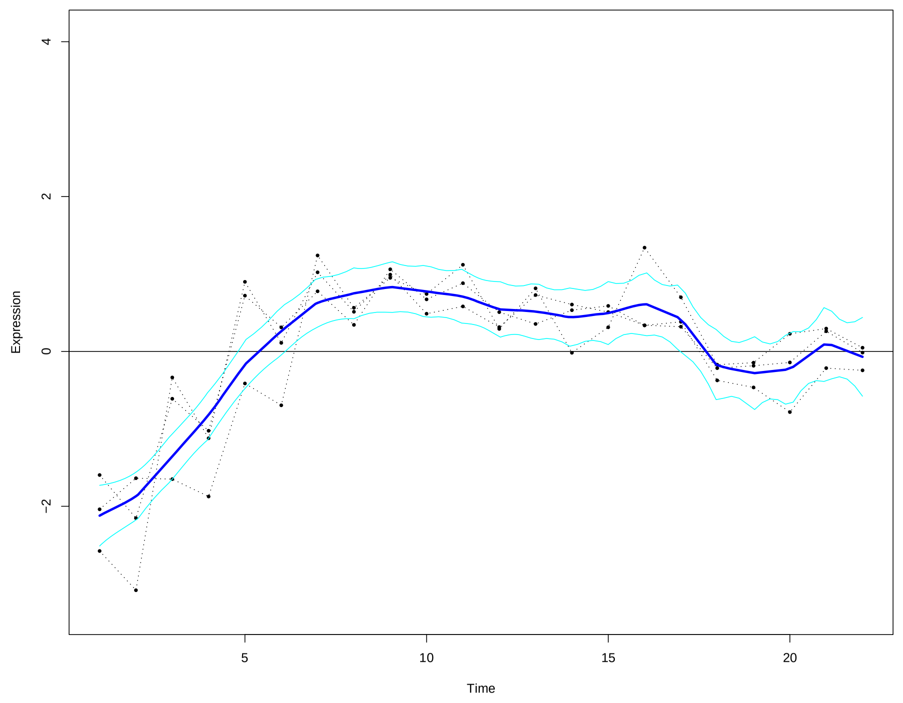

Cluster 63 (9 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20


Cluster 64 (14 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20

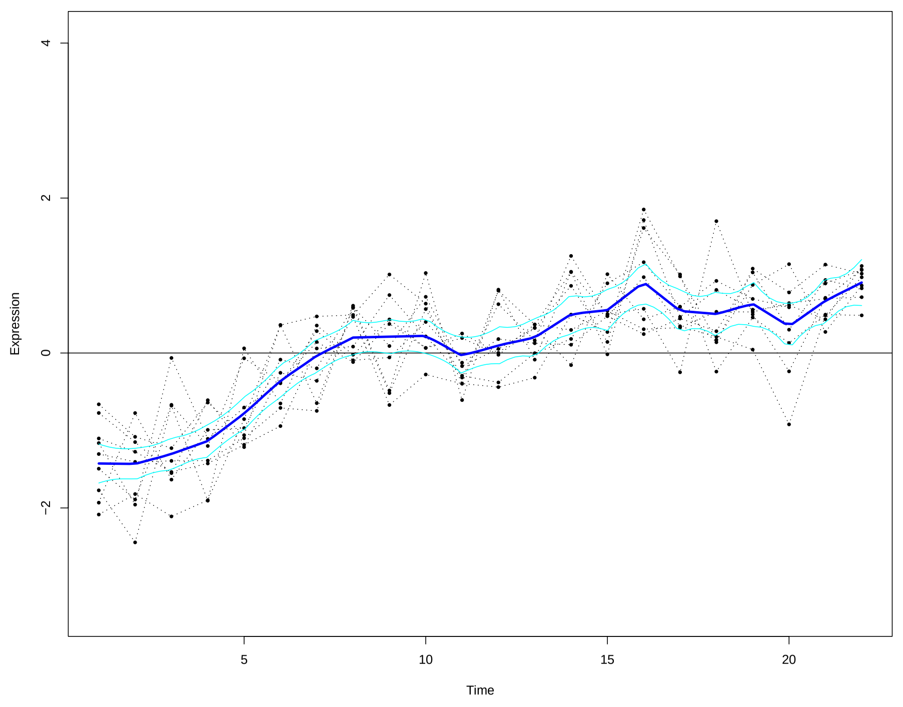
Cluster 65 (4 obsns.)



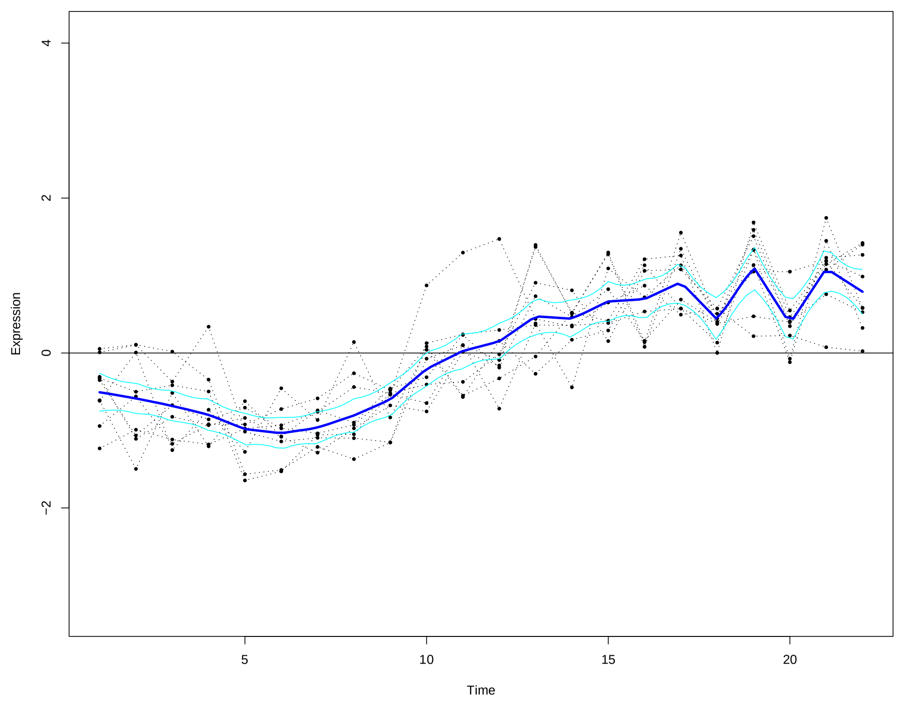
Cluster 66 (3 obsns.)

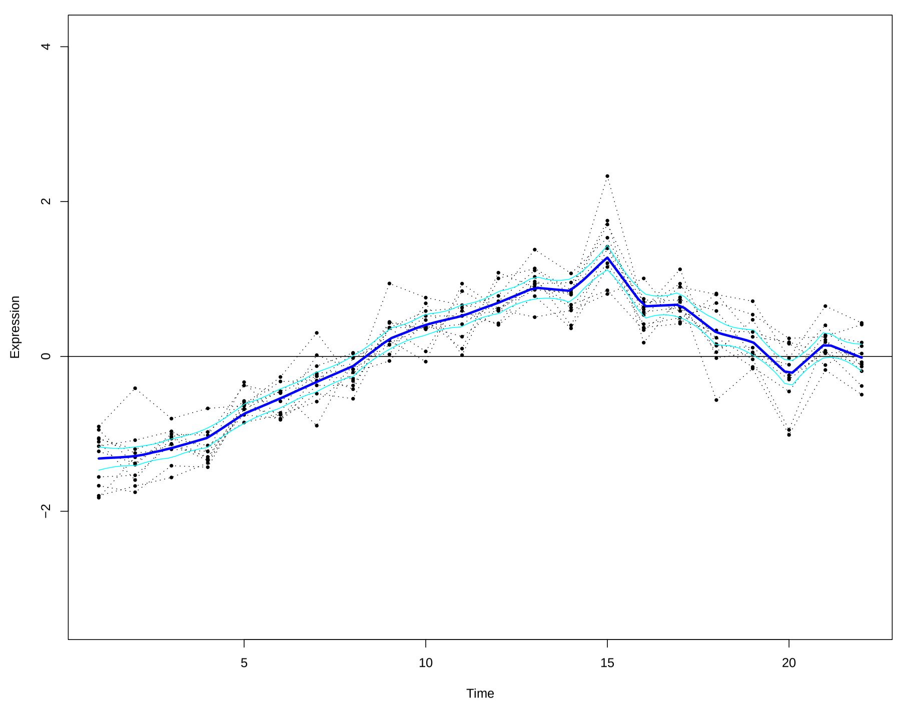


Cluster 67 (9 obsns.)

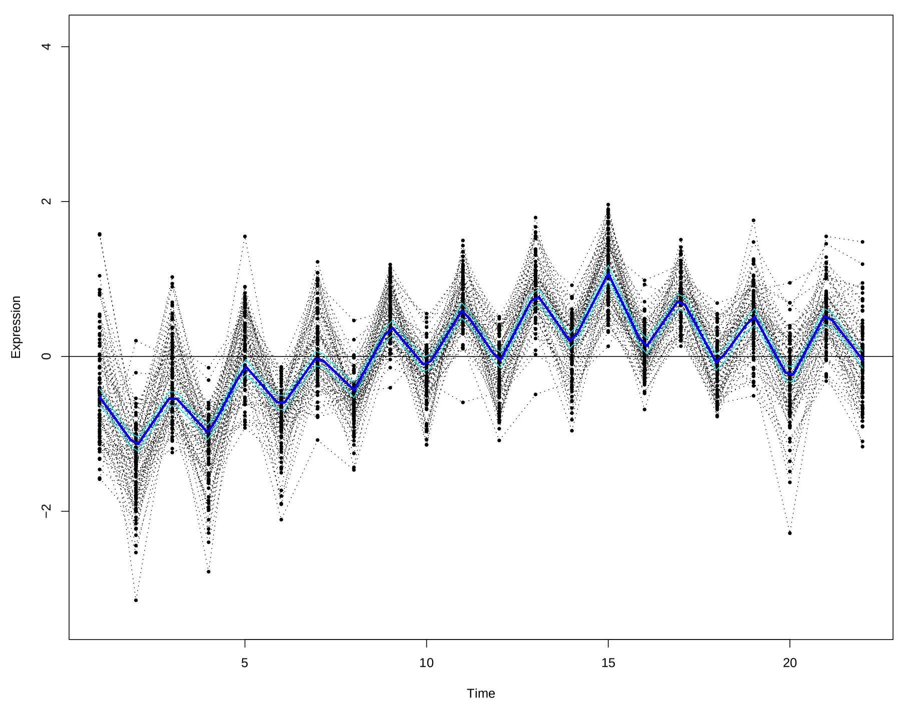


Cluster 68 (9 obsns.)

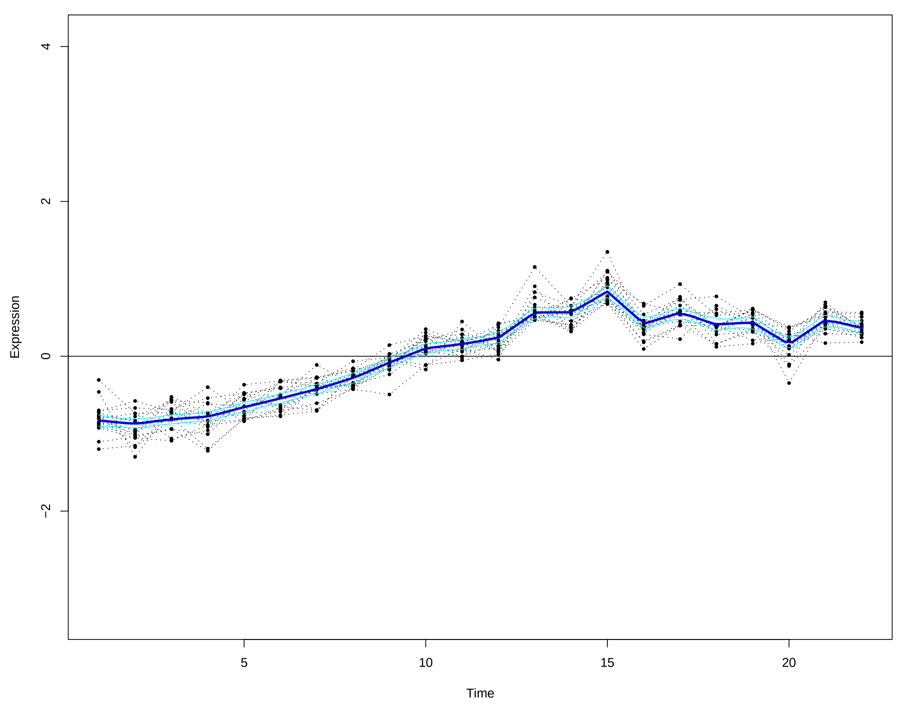




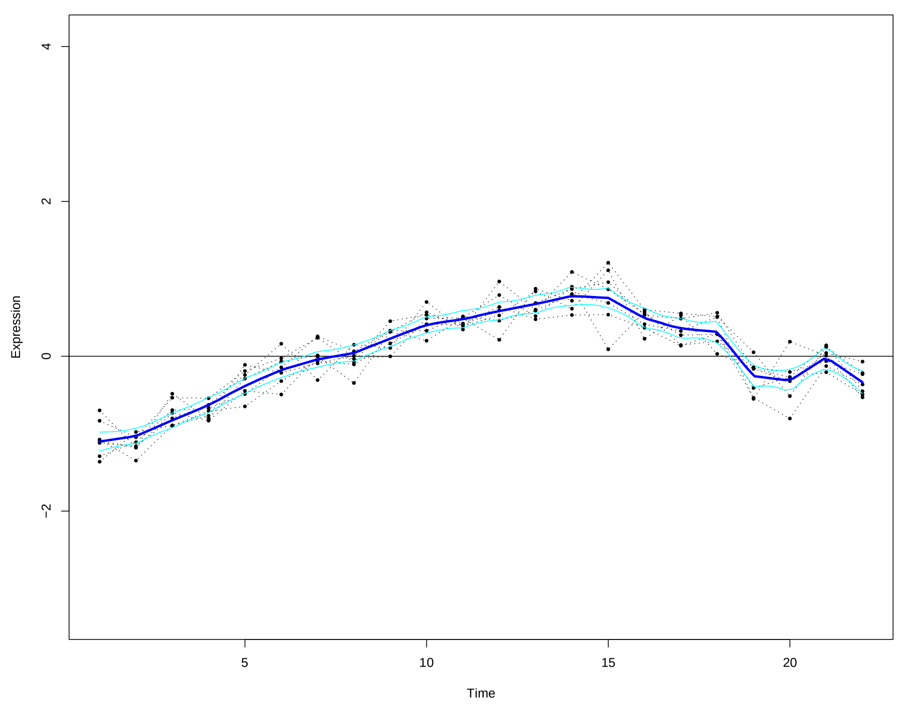
Cluster 69 (11 obsns.)



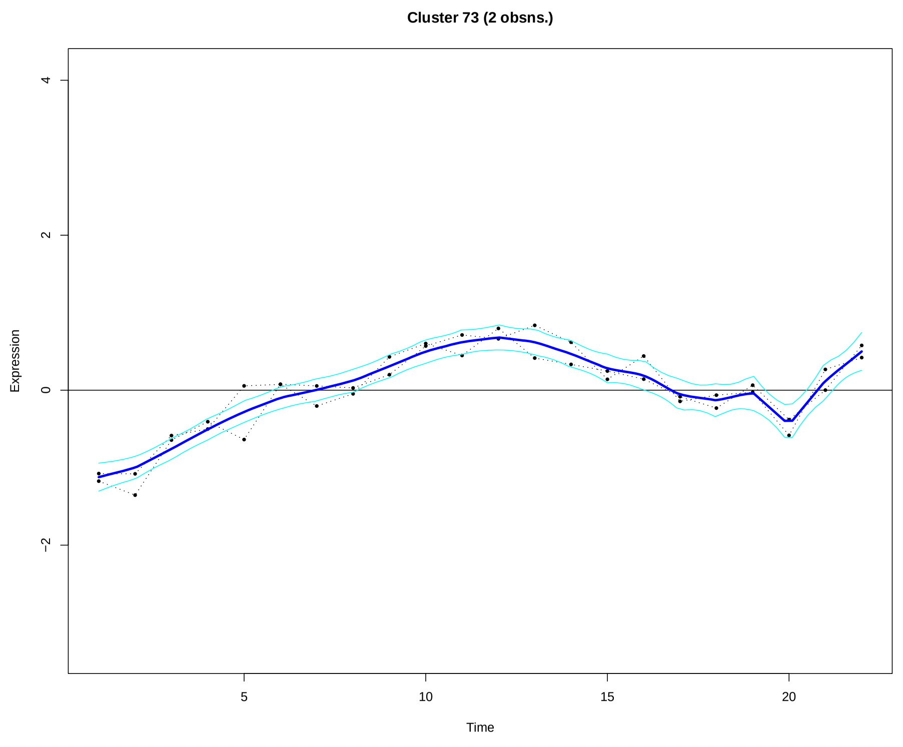
Cluster 70 (84 obsns.)

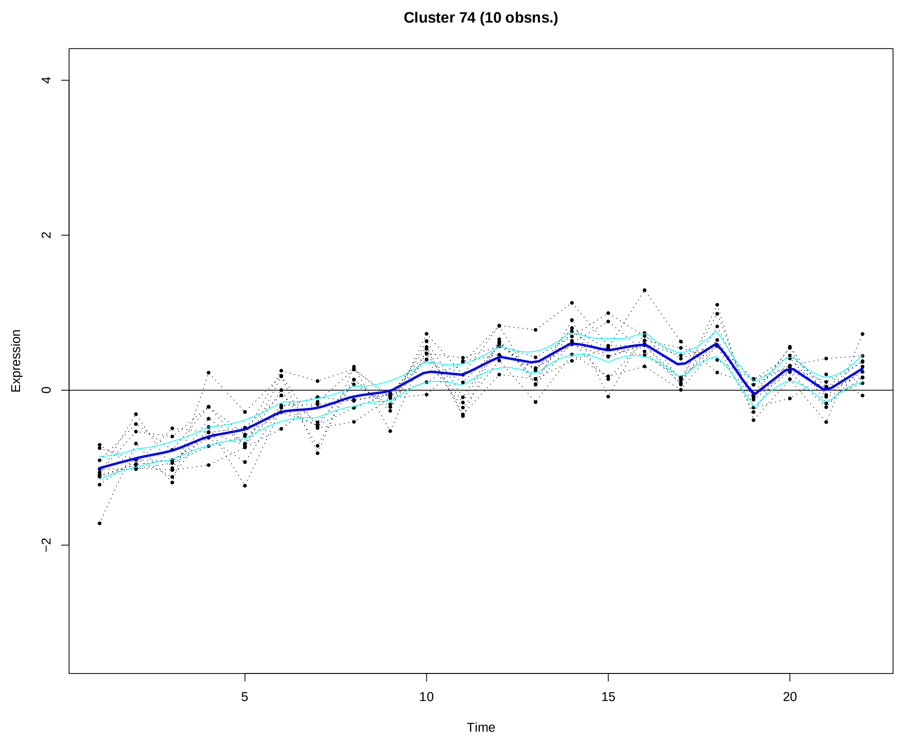


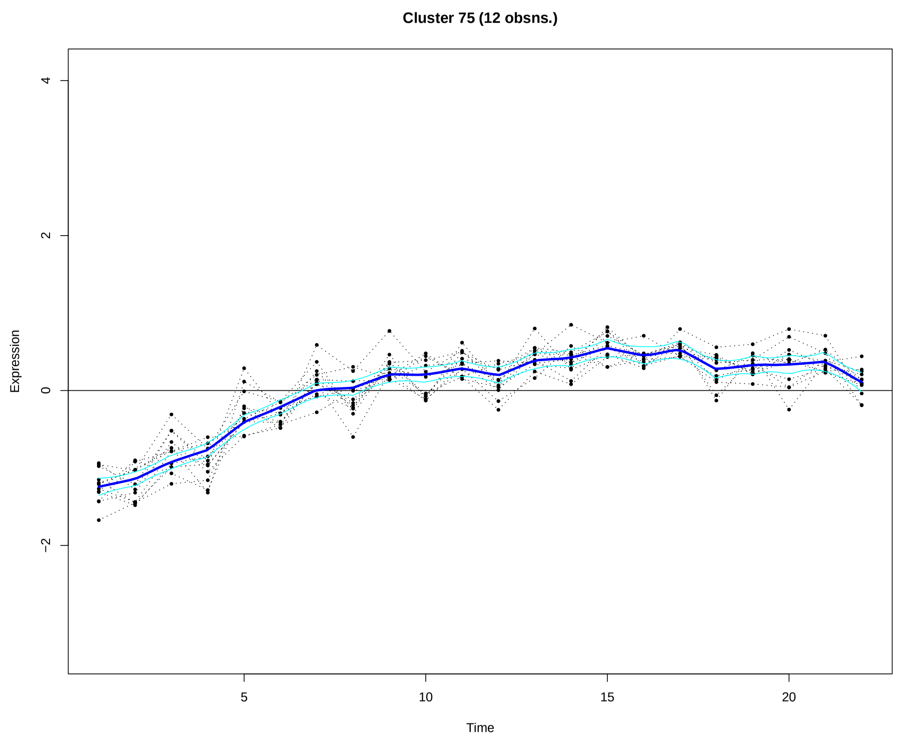
Cluster 71 (17 obsns.)

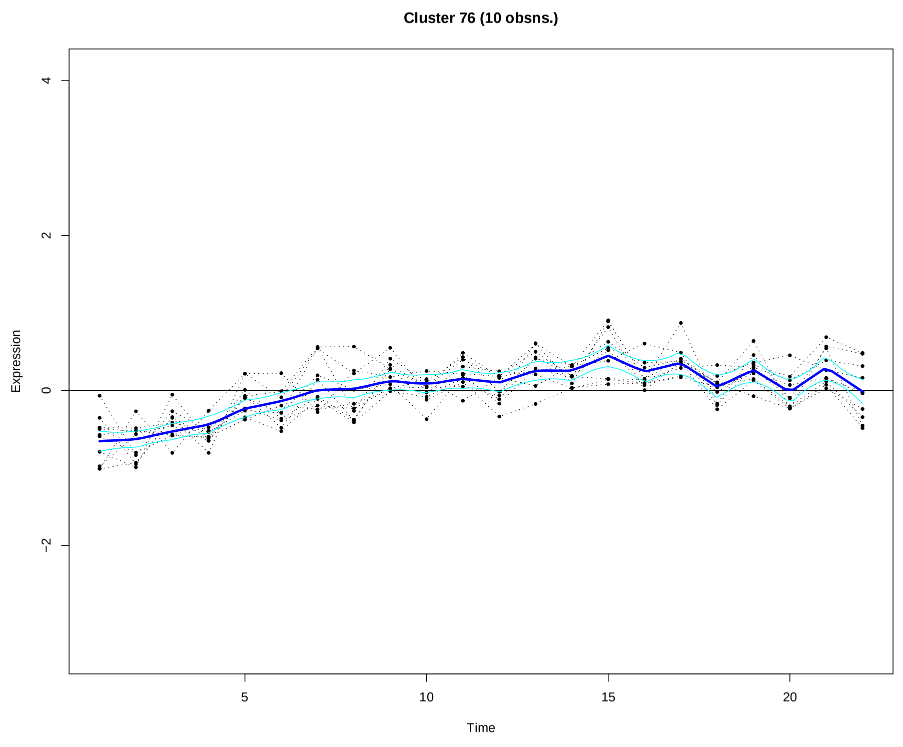


Cluster 72 (7 obsns.)

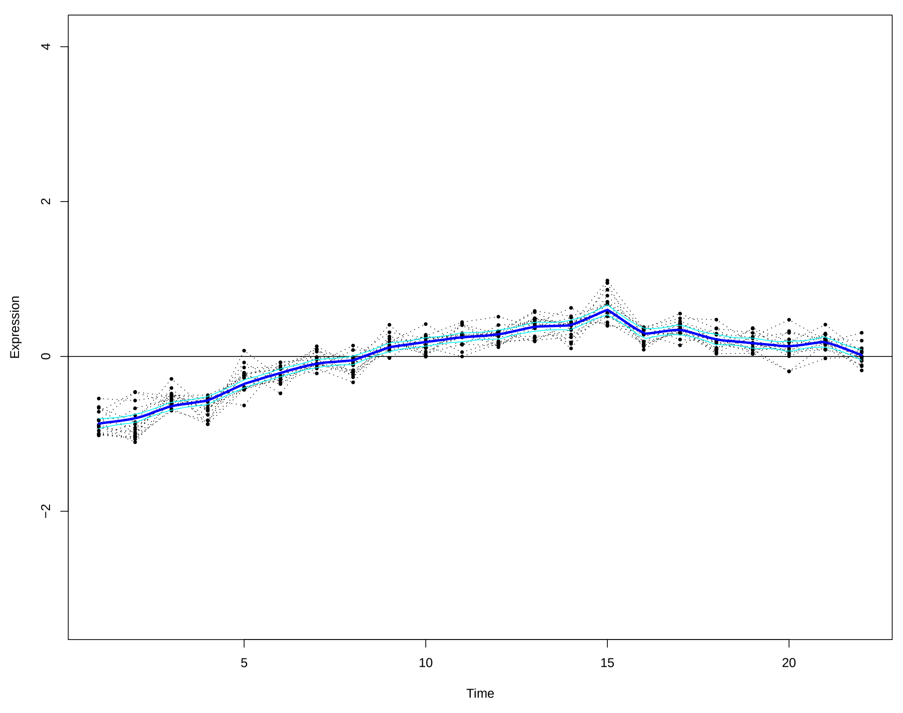

Cluster 73 (2 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20


Cluster 74 (10 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20

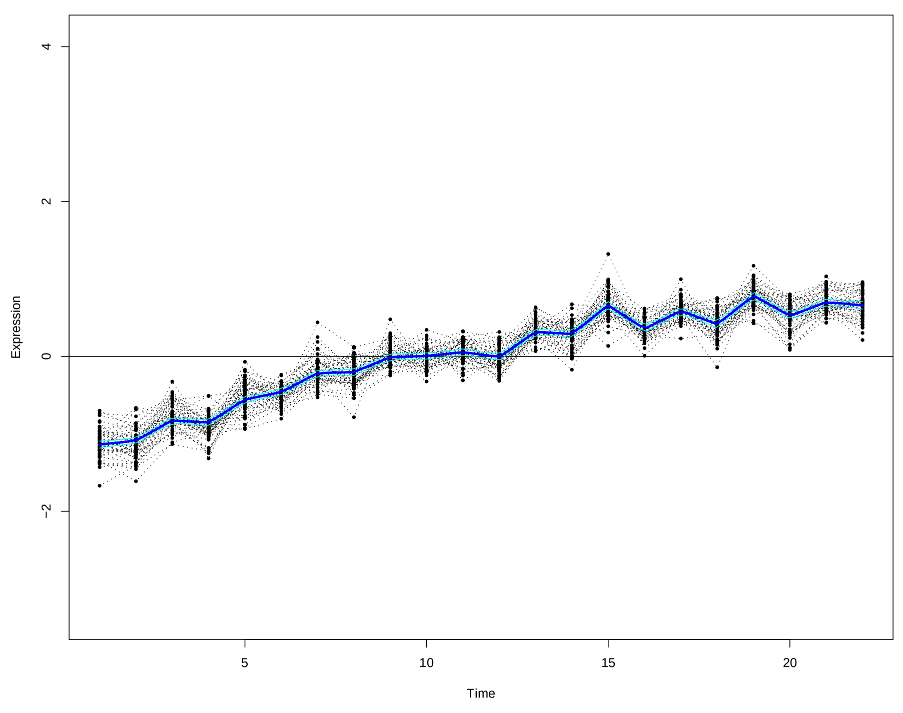

Cluster 75 (12 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20


Cluster 76 (10 obsns.)
Expression
Time
4
2
0
−2
5
10
15
20

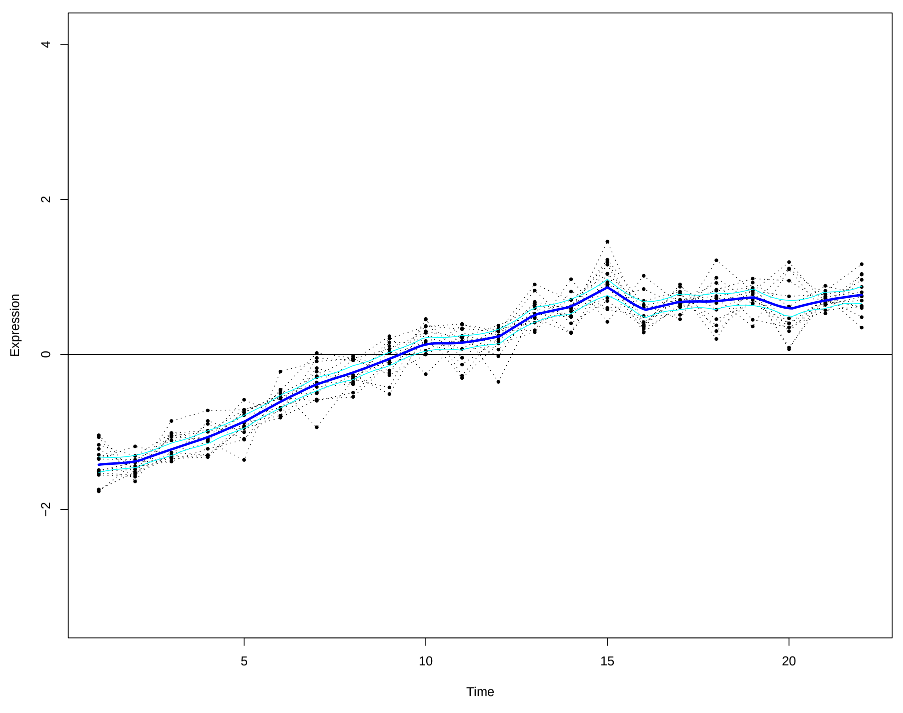
Cluster 77 (16 obsns.)



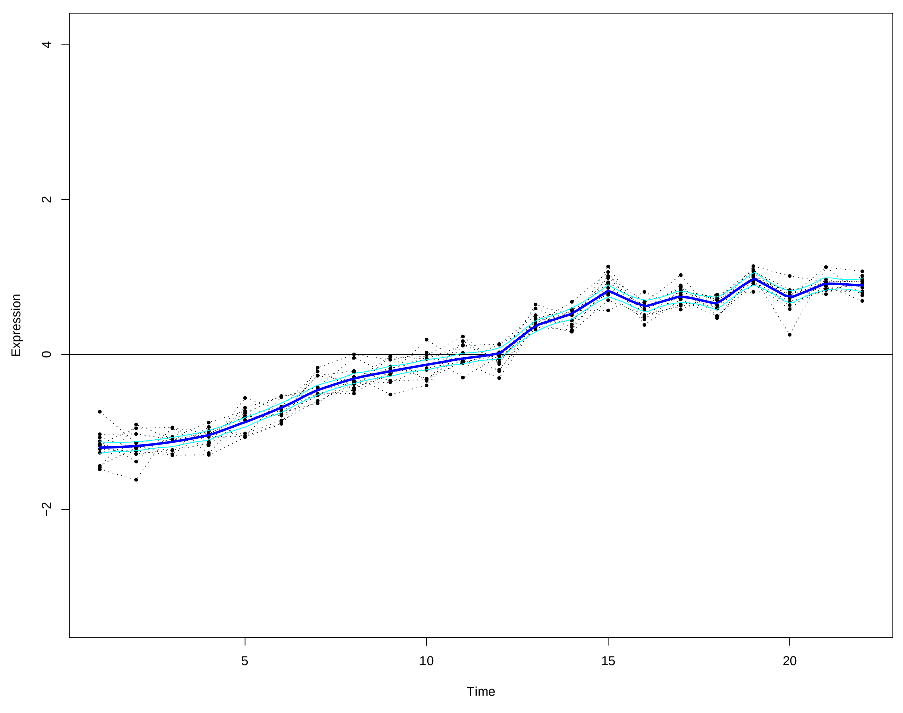
Cluster 78 (52 obsns.)



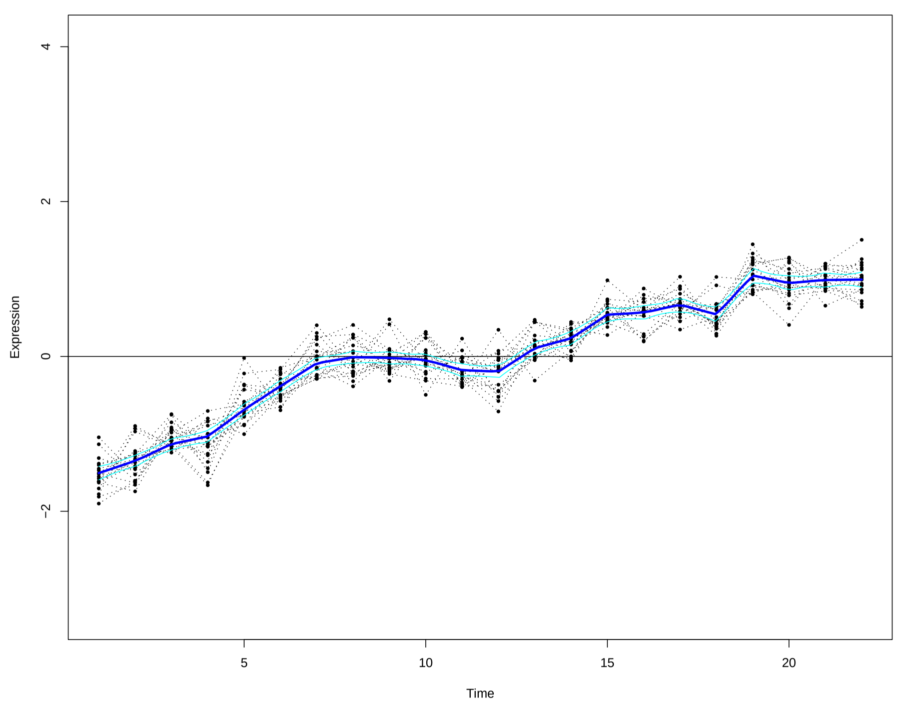
Cluster 79 (14 obsns.)



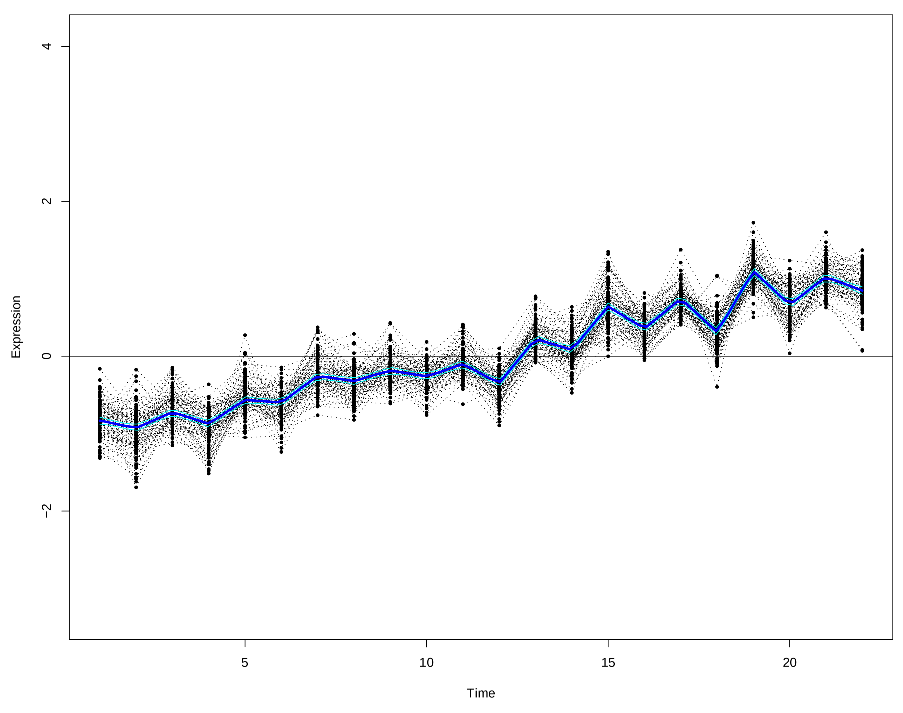
Cluster 80 (12 obsns.)



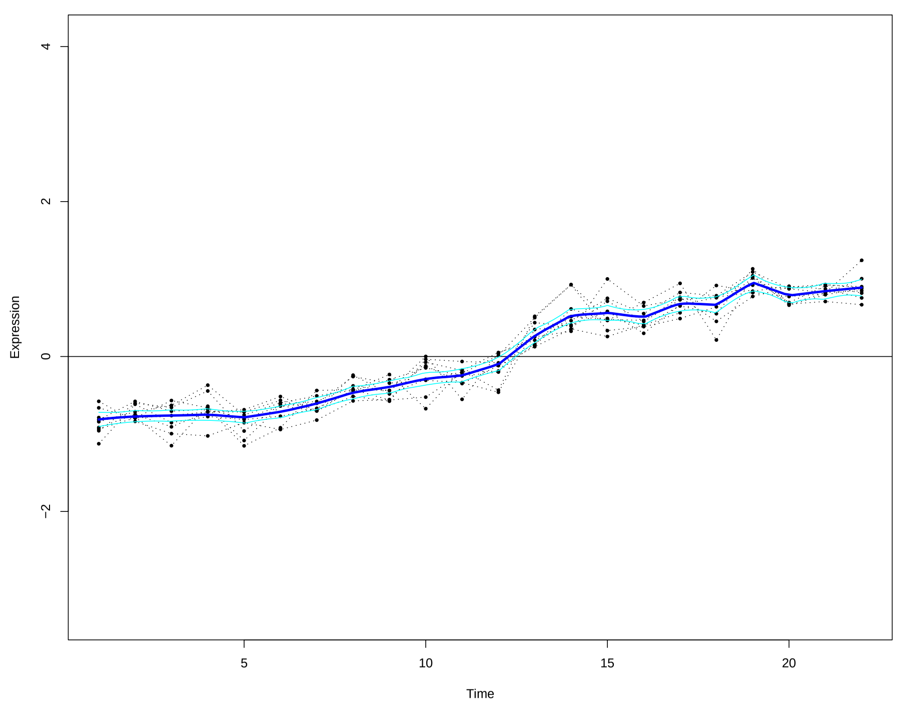
Cluster 81 (19 obsns.)



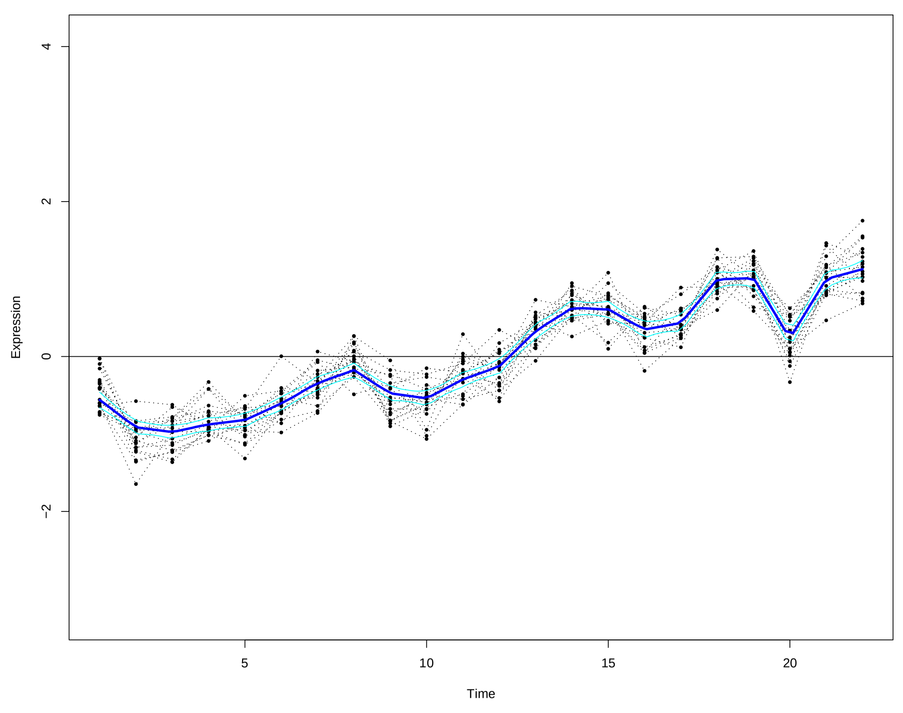
Cluster 82 (96 obsns.)



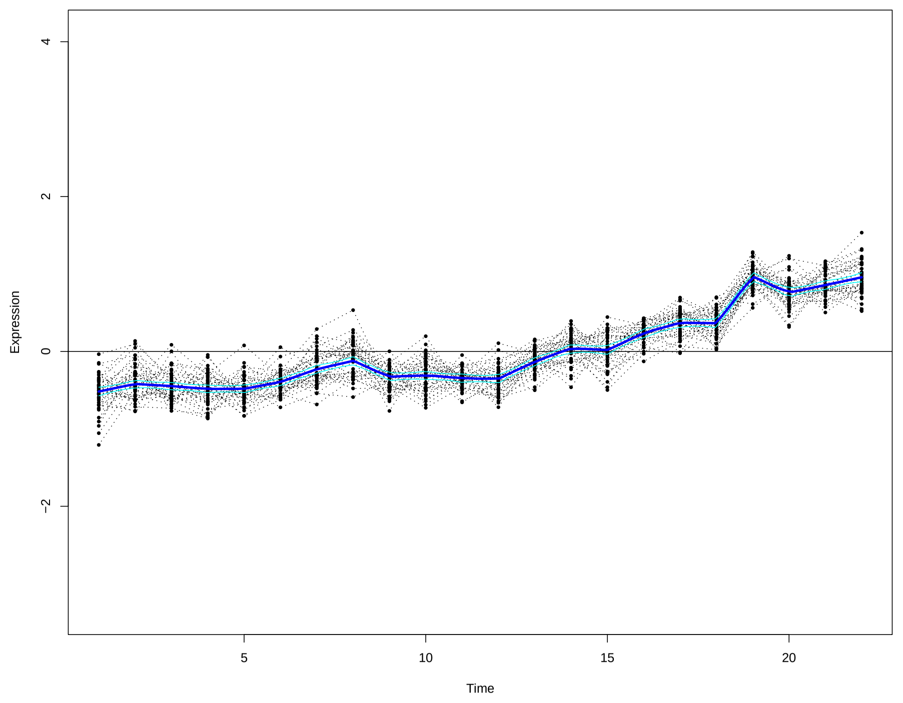
Cluster 83 (9 obsns.)



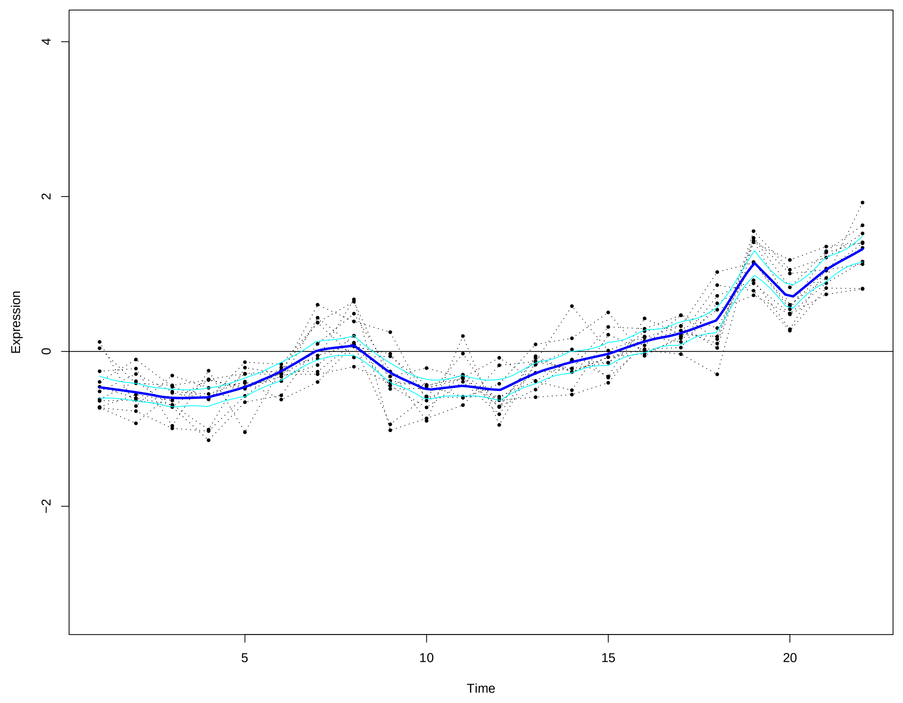
Cluster 84 (19 obsns.)



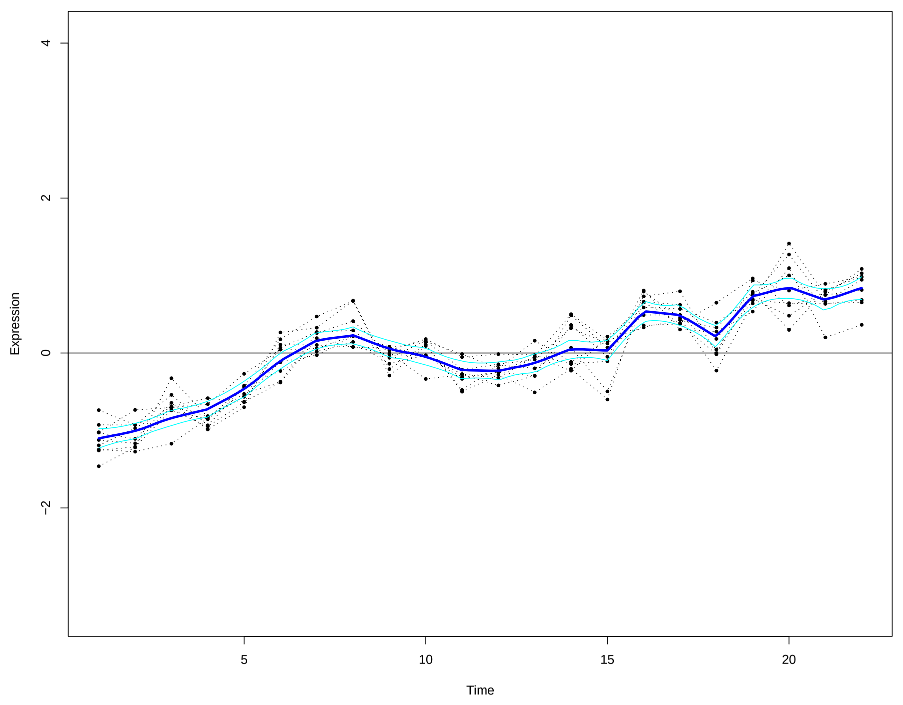
Cluster 85 (42 obsns.)



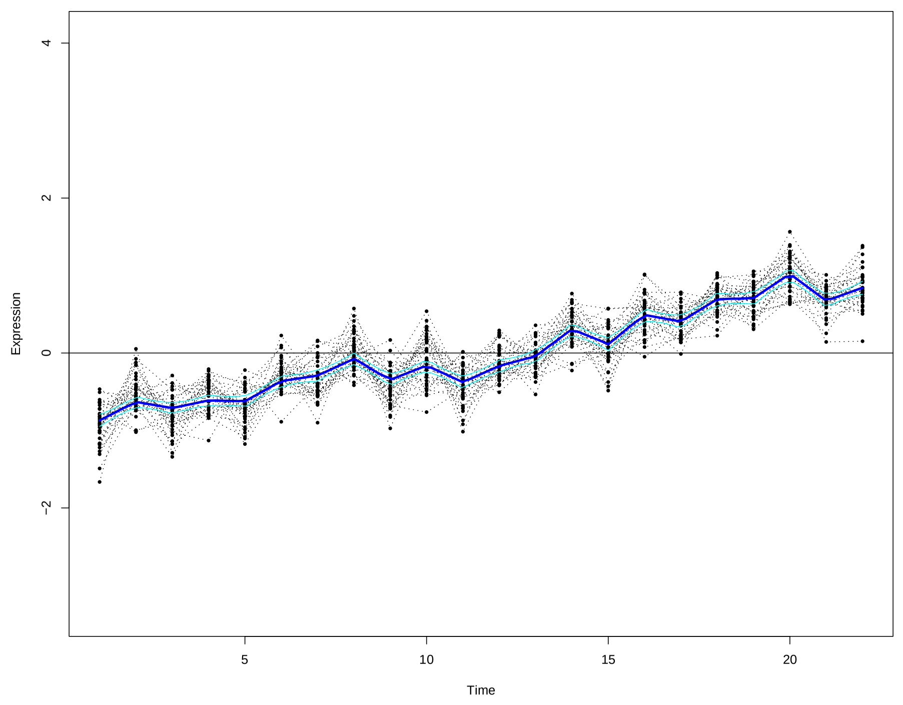
Cluster 86 (11 obsns.)



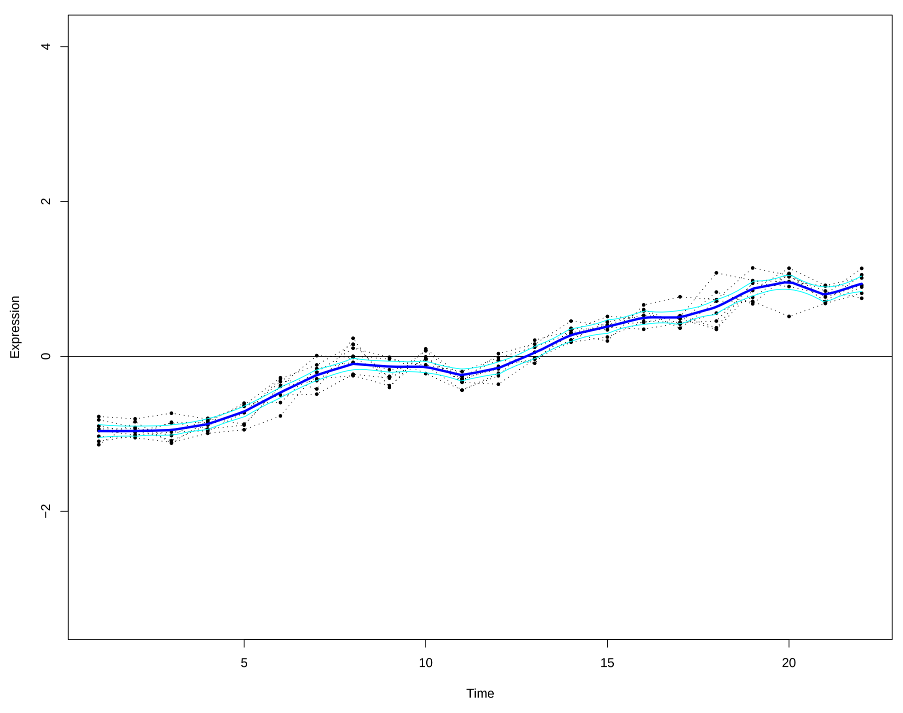
Cluster 87 (9 obsns.)



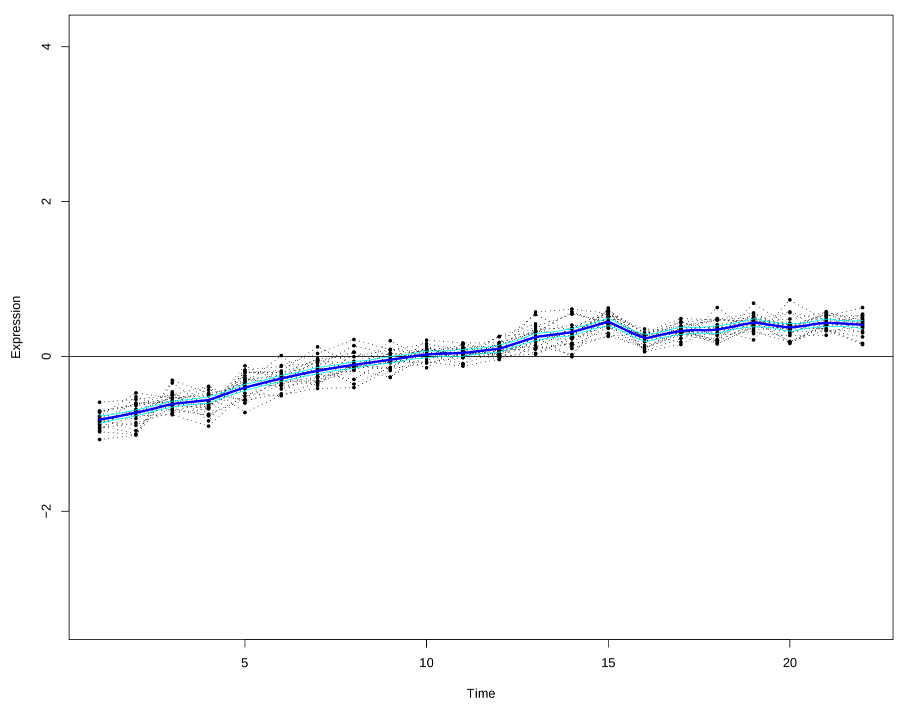
Cluster 88 (35 obsns.)



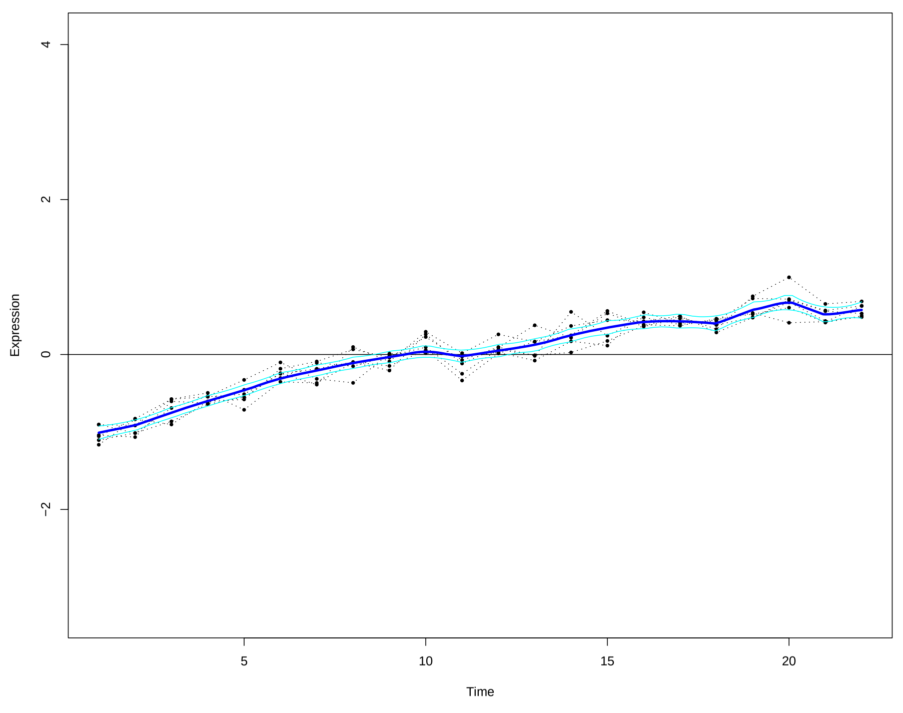
Cluster 89 (8 obsns.)



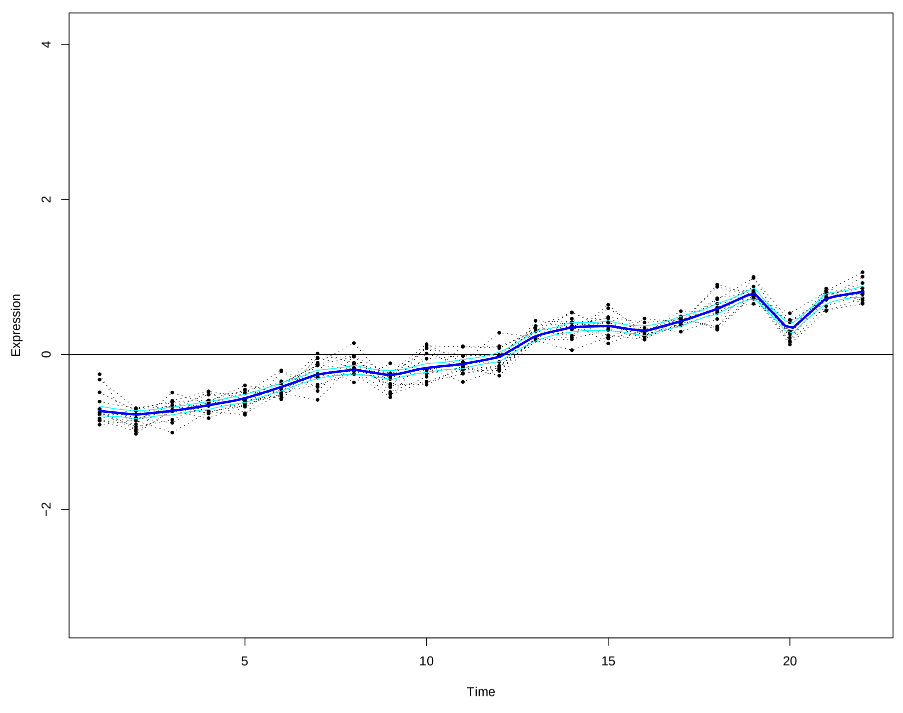
Cluster 90 (22 obsns.)



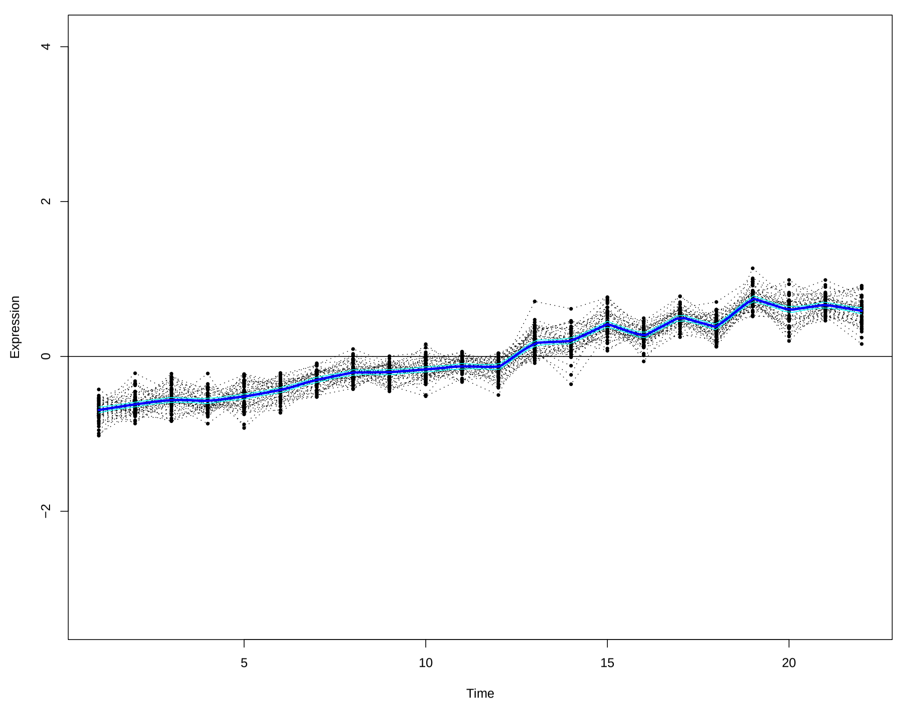
Cluster 91 (6 obsns.)



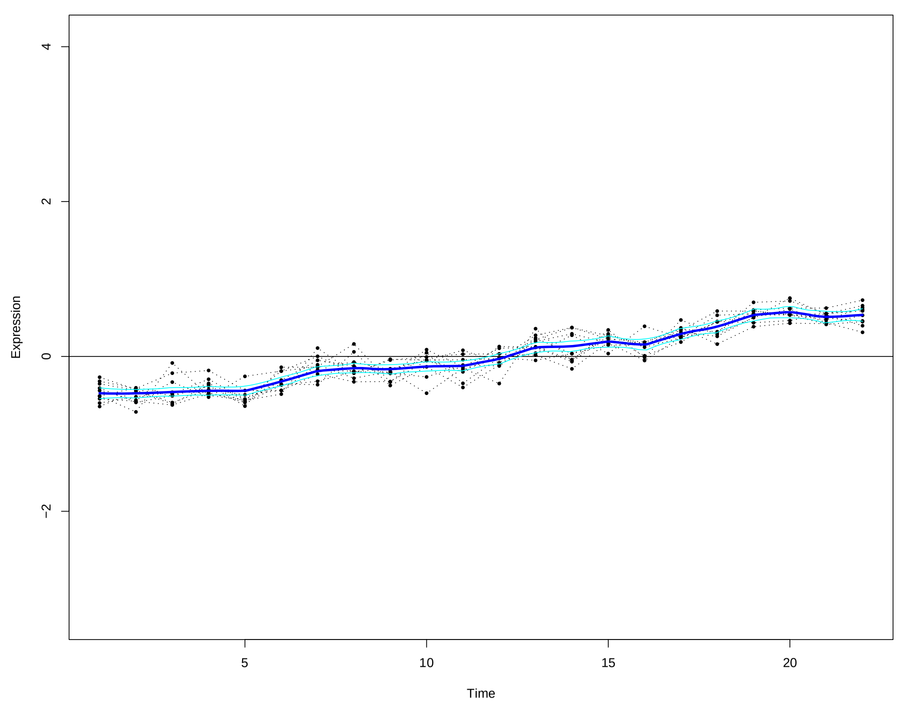
Cluster 92 (14 obsns.)

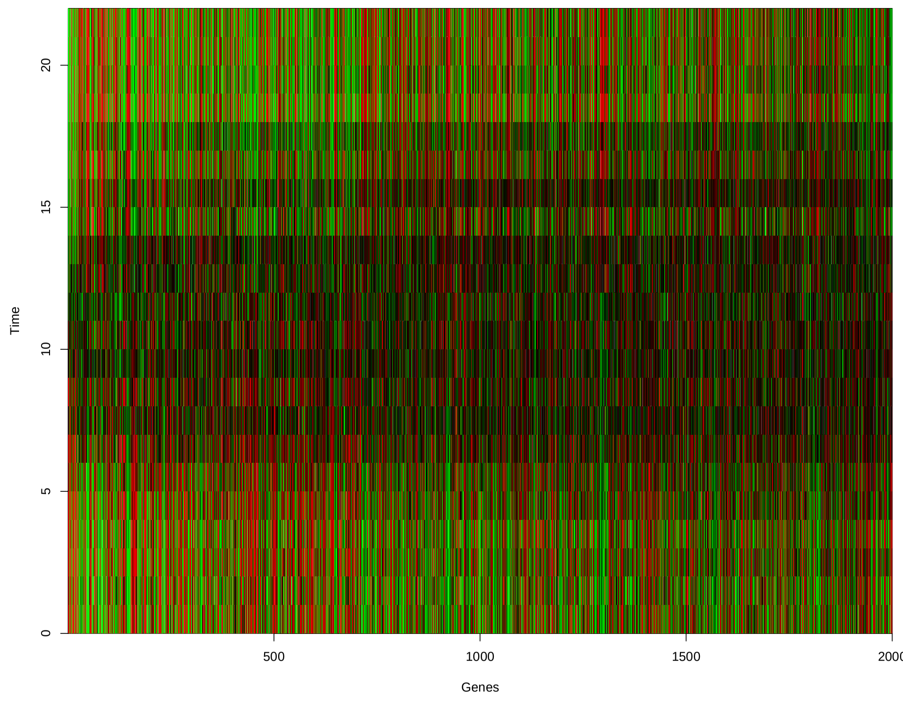


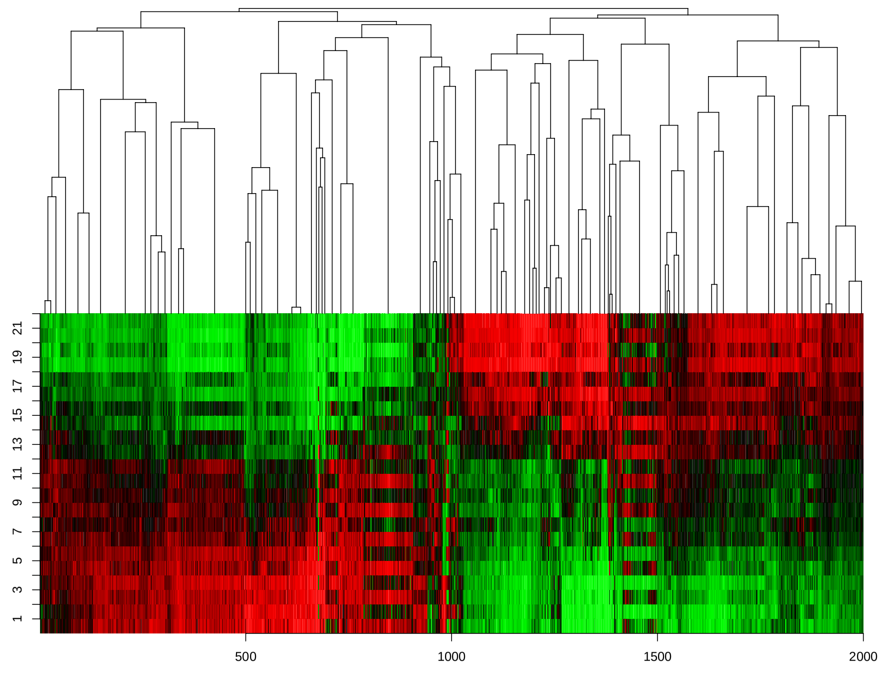
Cluster 93 (50 obsns.)

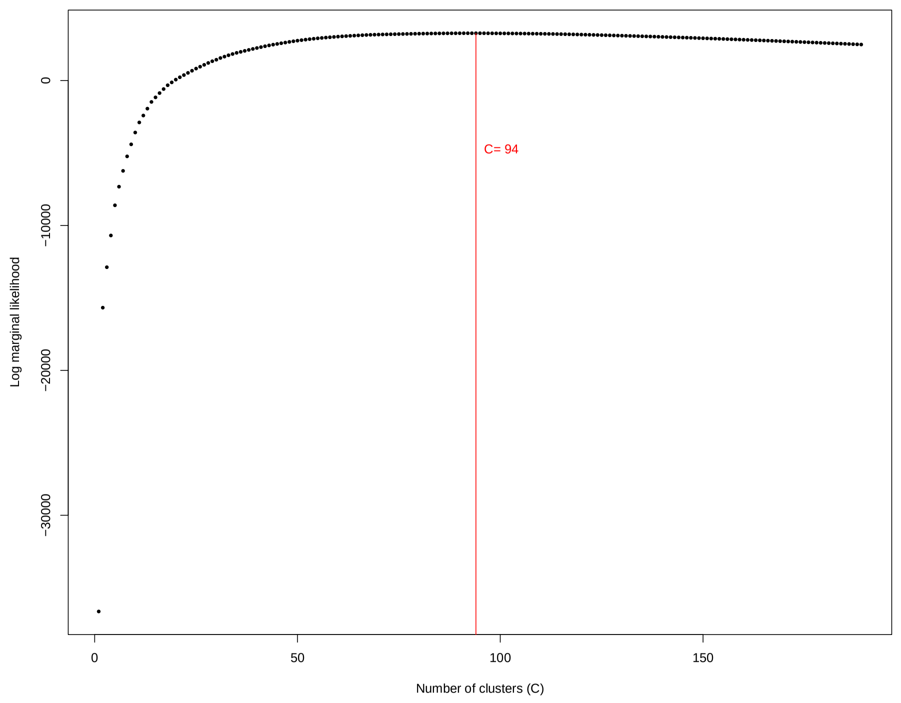


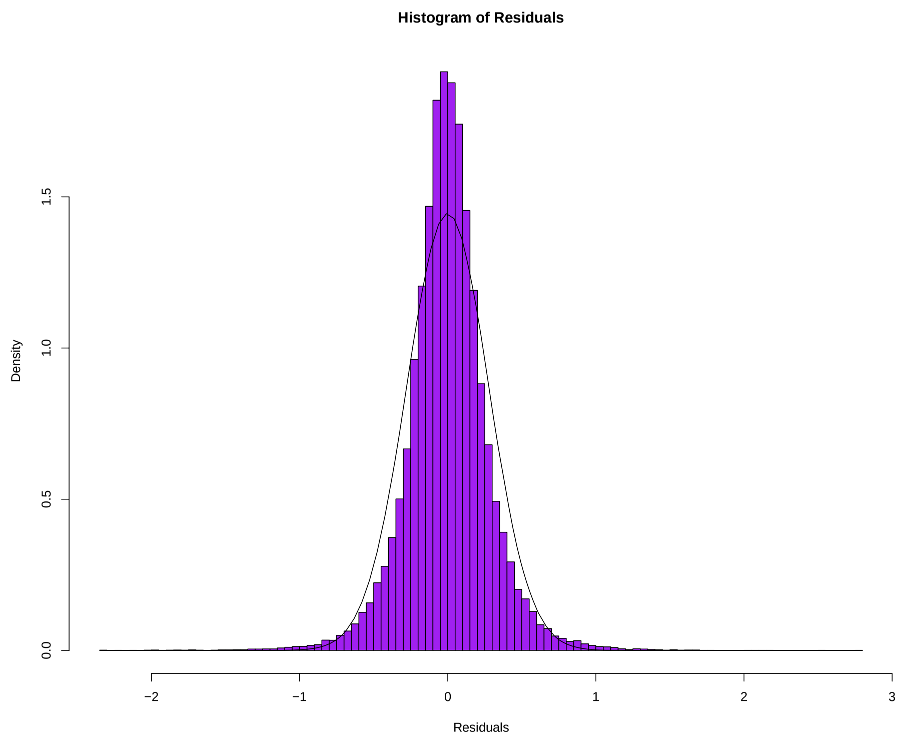
Cluster 94 (10 obsns.)








C= 94
Log marginal likelihood
Number of clusters (C)
0
−10000
−20000
−30000
0
50
100
150


Histogram of Residuals
Density
Residuals
0.0
0.5
1.0
1.5
−2
−1
0
1
2
3