A Method for Finding Standard Error Estimates for RMA Expression Levels Using Bootstrap

Gabriel Nicholas
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/gradreports

Part of the Mathematics Commons, and the Statistics and Probability Commons

Recommended Citation
https://digitalcommons.usu.edu/gradreports/1288
A METHOD FOR FINDING STANDARD ERROR ESTIMATES FOR
RMA EXPRESSION LEVELS USING BOOTSTRAP

By

Gabriel Nicholas

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Statistics

Utah State University
Logan, Utah

2007
ABSTRACT

A Method For Finding Standard Error Estimates For
RMA Expression Levels Using Bootstrap

By

Gabriel Nicholas
Utah State University, 2007

Major Professor: Dr. John R. Stevens
Department: Statistics

Oligonucleotide arrays are used in many applications. Affymetrix GeneChip
arrays are widely used. Before researchers can use the information from these arrays, the
raw data must be transformed and summarized into a more meaningful and usable form.
One of the more popular methods for doing so is RMA (Robust Multi-array Analysis).

A problem with RMA is that the end result (estimated gene expression levels) is
based on a fairly complicated process that is unusual. Specifically, there is no closed-
form estimate of standard errors for the estimated gene expression levels. The current
recommendation is to use a na"ive estimate for the SE that is based on a simple ANOVA
model. This results in an estimated SE that is the same for all arrays even when there is
reason to believe they should be different.

This paper investigates a computationally efficient implementation of
bootstrapping as a way to get a valid estimate of the standard errors of RMA expression
levels. Oligonucleotide arrays contain a lot of data, and processing that data already
carries a significant cost in computation. Bootstrapping compounds this cost.

Consequently efforts have been made to reduce the required number of resamples while still getting a reasonable estimate of the standard error. The accompanying R function is however flexible enough to do as many resamples as are required; the tradeoff is that more resamples mean more computation time will be needed.
CONTENTS

ABSTRACT................................................................................................................. 3
CONTENTS.................................................................................................................. 5
LIST OF FIGURES....................................................................................................... 6
LIST OF TABLES.......................................................................................................... 7
INTRODUCTION........................................................................................................... 8
RMA PREPROCESSING................................................................................................. 12
BOOTSTRAPPING........................................................................................................ 14
RESULTS..................................................................................................................... 20
DISCUSSION................................................................................................................ 22
REFERENCES.............................................................................................................. 24
APPENDIX A – AN EXAMPLE OF MEDIAN POLISH................................................... 26
APPENDIX B – R SOURCE CODE................................................................................. 31
APPENDIX C – C CODE............................................................................................... 36
APPENDIX D – EXAMPLE R CODE TO USE THE FUNCTION................................. 41
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A plot of the estimated standard errors for the matrix in Table 1</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Plot of estimated standard errors vs. resample #</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Another example of bootstrapped standard errors vs. resample #</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>A different pattern for SE changes vs. resample #</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>Weighted vs. unweighted test statistics from lmFit</td>
<td>22</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>An example matrix of 16 probes for 6 arrays</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Various statistics based on the data in Table 1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>An example of a bootstrapped matrix of probe intensities</td>
<td>15</td>
</tr>
</tbody>
</table>
INTRODUCTION

Affymetrix GeneChips (www.affymetrix.com) have proven to be a useful tool for researchers interested in studying the genetic basis for biological differences. On these chips, a single gene is represented by several probes in different spots. The data consist of spot intensities after a biological sample is applied to the chip. In general, an expressed gene will have higher intensity values in its representative probe spots. The data from the chip must be preprocessed before useful analyses can take place. The three steps in preprocessing are background correction – removing array-specific ‘noise’, normalization – adjusting the probe values so comparison across multiple arrays is possible, and summarization – combining individual probes values into a single summary statistic for each gene for each array (Gentleman et. al 2005). One of the more popular preprocessing methods is RMA (Robust Multi-Array Analysis), which uses a convolution background correction, quantile normalization, and median polish for summarization. This combination has been found to produce estimates for gene expression levels that have desirable properties with regards to precision and accuracy, as compared to other competing methods e.g. MAS5 (Irizarry et. al 2003).

The problem is that getting a good estimate for the standard error of these expression levels mathematically is very difficult due to the exploratory nature of the median polish. The suggested course of action has been to use a ‘naïve’ nominal estimate that fits an ANOVA model then uses the standard errors of that ANOVA model as the estimate for the standard errors of the RMA expression levels (Irizarry 2003). Already this seems somewhat unsatisfactory, as the estimates for standard error values are for
statistics that are never actually used or computed by RMA. Additionally, this naïve approach always yields estimates that are the same for each array. Even if the observed probe intensities on one array have greater variability than the probe intensities on another array (for the same gene,) they are both given the same estimated standard error. Consider the following example (using exaggerated artificial data.)

The cell values in Table 1 represent background corrected, normalized, log2 transformed perfect-match probe intensities for 16 probes (the columns) on 6 arrays (the rows). Approximate RMA expression levels, naïve SE estimates, and within-array probe means and standard deviations can all be computed. These are shown in Table 2.

| Table 1 – An example matrix of 16 probes for 6 arrays |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A1 | 10.2 | 14.5 | 5.1 | 13 | 12.4 | 6.2 | 8.8 | 9.2 | 14.7 | 18.4 | 5.3 | 6.5 | 6.5 | 16.4 | 14.3 | 12.9 |
| A2 | 11 | 15.3 | 6.4 | 12.7 | 14.4 | 7.1 | 8.4 | 9.6 | 15 | 17.9 | 8.3 | 8 | 6.4 | 18.8 | 15.9 | 11.9 |
| A3 | 9.7 | 15 | 4.9 | 13.8 | 14.7 | 6.2 | 7.9 | 9.6 | 15.1 | 18.6 | 6.1 | 6.2 | 6.9 | 15.7 | 13 | 12.3 |
| A4 | 9.5 | 13.8 | 6.6 | 13.5 | 16.3 | 6.6 | 8.4 | 9.5 | 14.4 | 18.7 | 7 | 6.7 | 7.3 | 16.7 | 13.8 | 13.9 |
| A5 | 8.8 | 9.9 | 7.9 | 10.3 | 10.9 | 9.9 | 8.5 | 9.8 | 10.7 | 12.2 | 8.9 | 9 | 9.4 | 11.5 | 11.2 | 11.1 |
| A6 | 12.3 | 11.8 | 9 | 11.7 | 12.4 | 9.7 | 9.1 | 11 | 11.4 | 13 | 9.4 | 10.2 | 9.9 | 11.3 | 11.5 | 10.9 |

| Table 2 – Various statistics based on the data in Table 1 |
|---|---|---|---|---|
| Array | RMA expr. | Naïve SE | Probe mean | Probe s.d. |
| A1 | 11.475 | 0.431 | 10.900 | 4.240 |
| A2 | 11.838 | 0.431 | 11.569 | 3.936 |
| A3 | 11.025 | 0.431 | 10.981 | 4.299 |
| A4 | 11.425 | 0.431 | 11.419 | 4.140 |
| A5 | 9.950 | 0.431 | 10.000 | 1.207 |
| A6 | 11.163 | 0.431 | 10.913 | 1.235 |
The naïve estimates are meant to be used as an approximate SE for the RMA expression level. But the probe standard deviations indicate that there is more variability in the probe intensities for the first 4 arrays (A1–A4) than there is in the probe intensities for the last 2 arrays (A5, A6). It’s reasonable to expect that at least some of this variability would survive the median polish and affect the relative sizes of the standard errors of the RMA expression levels. This is not reflected in the naïve SE estimates. Also consider the following: Let $Y_{ijn} = \log_2$ of background corrected, quantile normalized PM intensity of probe j of gene n on array i. Then the RMA model is:

$$Y_{ijn} = \mu_{i,n} + \alpha_{j,n} + \epsilon_{ijn},$$

where $\mu_{i,n}$ is the quantity of interest (the RMA expression level for array i on gene n), $\alpha_{j,n}$ is the effect for probe j on gene n, and $\epsilon_{ijn}$ is the error term. The naïve estimate is the standard error for $\mu_{i,n}$ from the ANOVA fit to equation [1]. The assumptions for the ANOVA model require that the error terms be iid $N(0, \sigma^2)$, but due to the median polish the independence assumption will not be met.

Better estimates for these standard errors may be found by bootstrapping. The resampling scheme is based on the view that the measurements of each probe in a gene on an array can be thought of as an individual measurement for the gene expression level. By bootstrapping the probe values within each gene for each array and then reapplying the median polish, approximate sampling distributions for the RMA expression levels can be generated. However, there is a strong probe effect in the data. The variance for specific probes across multiple arrays tend to be much smaller than the variance for related probes on the same array. This is important because it affects the results of the median polish procedure.
AFFYBATCH OBJECTS IN R

The raw data from the Affymetrix GeneChip is imported into R in the form of an Affybatch object. Inside the Affybatch object there will be one array for each observation in the sample; i.e., if an experimenter takes tissue samples from 30 people and gets a GeneChip for each, then after importing the data the Affybatch object will contain 30 arrays. Each array contains information for every gene included in the study. The number of genes ranges from several thousand up to the tens of thousand. There are nearly always more genes than there are observations. The data for each gene is stored as probe-level intensities. There are two categories of probe intensity data: mismatch intensities and perfect-match intensities. The perfect-match intensities are a measure of how much RNA was found for a particular gene. There are anywhere from 10-25 perfect-match probe intensities for each gene; each perfect-match intensity can be thought of as a measurement of how 'active' that gene was in the test subject. The number of probes can be different for each gene, although many of them will tend to have the same number of probes. The other type of probe intensity is the mismatch intensity. The Affybatch object contains a mismatch intensity for every perfect-match intensity on the array. The mismatch intensities are constructed such that they should not pick up any real 'signal.' The idea is that by looking at the mismatch intensities an estimate can be made of how much background noise there is on the array. This can then be removed, leaving only the interesting 'signal' part of the data. The RMA preprocessing method ignores mismatch intensities entirely.
RMA PREPROCESSING

RMA is a preprocessing method (one of several competing methods) that is used to estimate gene expression levels from raw probe intensities. The expression level is an estimate of how ‘active’ a particular gene is inside a particular organism. RMA preprocessing consists of three steps.

1. Background correction
   The method used by RMA for background correction is simply called RMA convolution. RMA assumes that probe intensities are the sum of 2 random variables – background noise (with a normal distribution,) and ‘signal’; (with an exponential distribution.) In this case, the ‘signal’ is the true amount of binding due to the presence of the proper type of RNA. Specifically, it uses the model \( PM = M + S \), where \( PM \) is the perfect match intensity, \( M \) is the normally distributed background noise, and \( S \) is the exponentially distributed signal. The estimated background corrected probe intensities are then \( E[S|M] \).

2. Normalization
   After background correction the probe intensities must be normalized across arrays. The intensities on different arrays may be on slightly different scales or have different distributions and so direct comparison of background corrected intensities might not make sense. RMA uses quantile normalization, which attempts to make the distribution of probe intensities the same for each array. From Bolstad et al (2003):

   The method is motivated by the idea that a quantile–quantile plot shows that the distribution of two data vectors is the same if the plot is a straight diagonal
line and not the same if it is other than a diagonal line. This concept is extended to \( n \) dimensions so that if all \( n \) data vectors have the same distribution, then plotting the quantiles in \( n \) dimensions gives a straight line along the line given by the unit vector \((1/\sqrt{n}, \ldots, 1/\sqrt{n})\). This suggests we could make a set of data have the same distribution if we project the points of our \( n \) dimensional quantile plot onto the diagonal.

The quantile normalization method was selected for use in RMA over alternative methods (Cyclic Loess, Contrasts, or various scaling methods) based on performance (variance and bias criteria) and computational efficiency.

3. Summarization

In the last step, the background corrected and normalized probe intensities for each gene need to be summarized into a single number, which represents the estimated expression level for that gene, for each array. By looking at the results of a spike-in study (where ‘real’ expression levels were known), Irizarry et al (2003) found that a log-scale linear additive model (as in equation [1]) works well when estimating expression levels. For this reason the RMA method uses the log_2 transformed probe intensities in a version of Tukey’s median polish procedure (Tukey 1977) to generate expression levels. A result of using the median polish is that it causes the estimated expression levels to have a very complicated distribution, confounding any attempt to look at it analytically. An example of how the median polish works can be found in Appendix A. Also, note that using the median polish means that the estimated expression level for a gene on an array depends not only on the probe intensities for that array, but on the probe intensities on all of the other arrays. In other words, while the raw GeneChip intensities may be statistically independent between arrays, the RMA expression levels are not.
BOOTSTRAPPING

Bootstrapping provides a straightforward method for finding statistics for complicated distributions. The question arises here of how to do the resampling. Typically one would resample from the observations and then recompute the statistic of interest. It is worth carefully considering how to do this in the case of Affybatch objects.

The goal is to get an estimate of the standard error of the expression levels for each gene on each array. The RMA expression level for a particular gene is a function of the perfect-match probe intensities for that gene. The probes can each be thought of as a separate measurement of the expression level for that gene. In the case of the MAS5 method the expression level for a particular gene is simply a robust average of the probes for that gene on the array (Affymetrix 2002). If bootstrapping MAS5 expression levels were to be done, then one could resample probe intensities within an array for the gene of interest. But due to its use of the median polish procedure, the RMA expression level is also affected by the probes from other arrays. Additionally, there is a strong probe effect. Probe intensities for a specific probe across arrays vary far less than probe intensities for different probes within a single array (Li and Wong 2001). Taken together, these point to a reasonable resampling method. Consider again the matrix of probe intensities from Table 1, which shows the intensities of 16 probes for a specific gene on 6 arrays. What this paper proposes is that the resampling should be done on the probes (which for this example is the columns) across all arrays at once. This way ensures that each array resamples from its own probes, but the probe effect is preserved. One such realization of a matrix resampled using this scheme is shown in Table 3.
Table 3 – An example of a bootstrapped matrix of probe intensities

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>6.5</td>
<td>18.4</td>
<td>10.2</td>
<td>5.3</td>
<td>5.1</td>
<td>12.4</td>
<td>8.8</td>
<td>14.3</td>
<td>18.4</td>
<td>18.4</td>
<td>12.4</td>
<td>14.5</td>
<td>5.1</td>
<td>6.5</td>
</tr>
<tr>
<td>A2</td>
<td>6.4</td>
<td>17.9</td>
<td>11</td>
<td>8.3</td>
<td>6.4</td>
<td>14.4</td>
<td>8.4</td>
<td>15.9</td>
<td>17.9</td>
<td>17.9</td>
<td>14.4</td>
<td>15.3</td>
<td>6.4</td>
<td>8</td>
</tr>
<tr>
<td>A3</td>
<td>6.9</td>
<td>18.6</td>
<td>9.7</td>
<td>6.1</td>
<td>4.9</td>
<td>14.7</td>
<td>7.9</td>
<td>13</td>
<td>18.6</td>
<td>18.6</td>
<td>14.7</td>
<td>15</td>
<td>4.9</td>
<td>8</td>
</tr>
<tr>
<td>A4</td>
<td>7.3</td>
<td>18.7</td>
<td>9.5</td>
<td>7</td>
<td>6.6</td>
<td>16.3</td>
<td>8.4</td>
<td>13.8</td>
<td>18.7</td>
<td>18.7</td>
<td>16.3</td>
<td>13.8</td>
<td>6.6</td>
<td>6.7</td>
</tr>
<tr>
<td>A5</td>
<td>9.4</td>
<td>12.2</td>
<td>8.8</td>
<td>8.9</td>
<td>7.9</td>
<td>10.9</td>
<td>8.5</td>
<td>11.2</td>
<td>12.2</td>
<td>12.2</td>
<td>10.9</td>
<td>9.9</td>
<td>7.9</td>
<td>9</td>
</tr>
<tr>
<td>A6</td>
<td>9.9</td>
<td>13</td>
<td>12.3</td>
<td>9.4</td>
<td>9</td>
<td>12.4</td>
<td>9.1</td>
<td>11.5</td>
<td>13</td>
<td>13</td>
<td>12.4</td>
<td>11.8</td>
<td>9</td>
<td>10.2</td>
</tr>
</tbody>
</table>

Note that each full column in this new matrix is a copy of one of the columns in the original matrix. But within each row, the probe intensities have been randomly resampled. This is to make sure that the column medians still make sense when performing the median polish.

Upon this resampled matrix the median polish procedure is performed, generating an estimated RMA expression level for each array. These expression levels are saved, and resampling continues until enough expression levels are available to compute reasonable standard errors. This method of resampling should preserve at least some of the covariance that is brought on by the median polish. However, since it is not resampling raw probe intensities and is instead resampling from the background corrected and normalized intensities, any covariance imparted by the background correction and normalization processes is destroyed (or at least not accounted for). This has unknown impact on the standard error estimates.
Figure 1 - A plot of the estimated standard errors for the matrix in Table 1

Figure 1 is a plot of the standard errors for the hypothetical PM intensity matrix shown in Table 1, at each step in 1000 resamples. As expected, the standard errors for arrays A1-A4 are significantly higher than the standard errors for arrays A5 and A6. Specifically, the estimates for arrays 1-6 are 2.11, 1.99, 2.09, 2.02, 0.85 and 0.75 respectively. The horizontal dashed line at the bottom of the graph shows the naïve ANOVA SE estimate of 0.43, which as usual is lower than the bootstrap estimates. Once again this plot reinforces the argument against using the naïve estimate, as it assigns the same SE to all 6 arrays when they should be different.

Other Examples

The plots for this section use data from a mouse dataset containing 6 arrays. These data are publicly available from the Gene Expression Omnibus at www.ncbi.nlm.nih.gov/projects/geo/ as GEO accession GDS1300, and are used here only as a convenient example. Several genes were selected at random and bootstrapping was
performed according to the method described above. After each resample a new standard error was computed and saved. Figure 2 shows the bootstrap estimated standard errors for the expression levels of all 6 arrays after each of 1000 resamples for one of the selected genes in the mouse dataset. The vertical axis is the estimated standard error, and the horizontal axis is the number of completed resamples.

![Estimated standard errors at each resample step](image)

Figure 2 – Plot of estimated standard errors vs. resample #

In this example, the standard errors would be around 0.40 for two of the arrays, and just under 0.37 or so for the other arrays. It’s worth pointing out that near the beginning of the process the standard error estimate jumps around, but after a certain number of resamples, the estimate doesn’t change substantially. Also note the horizontal
dashed line near the bottom of the graph. This dashed line represents the naïve nominal estimate for the standard error, which for this gene in the mouse dataset is 0.027.

Figure 3 – Another example of bootstrapped standard errors vs. resample #

The standard errors for the majority of genes for these mice follow the pattern shown in Figures 2 and 3. They vary a great deal in the beginning, reaching a high point early on before dying down as more and more resamples are generated until they finally stabilize around a particular value. This appears to be the most common behavior for the genes considered in this dataset, but there are other genes that don’t follow this pattern.
Figure 4 — A different pattern for SE changes vs. resample #

For the realization of the bootstrap shown in Figure 4, the estimated standard errors start (more or less) low then gradually rise as more resampling takes place. Still, beyond 400 resamples the estimated standard errors don’t change very much.

The converging behavior of the standard errors indicates a possible area for trimming computation time. If one can determine whether or not the variances for a gene have started to ‘converge’ to some value, then the procedure ought to be able to stop sooner into the process without doing the full amount of bootstraps for every gene and still get a reasonable estimate. The rationale is that since the bootstrap estimate may not change substantially beyond a certain number of bootstraps, then there is no need to continue resampling the current gene. This can potentially save a huge amount of computation time.
A function has been made to do this in R. The R code and C code are included in Appendices B and C. For the included R function, there are 4 parameters that will determine when and if the bootstrapping should stop early. There is a parameter for a minimum number of resamples (min.resamples), maximum number of resamples (max.resamples), another for a minimum percent change (min.diffpct), and the last one is the step size (step.size). The function will always perform at least min.resamples number of resamples. It will perform at most max.resamples number of resamples. The min.diffpct and step.size variables are a little more complicated. After performing at least min. resamples number of resamples, the function computes the maximum variance over all of those completed iterations for all arrays. This maximum variance is stored in a variable named max.var. Then, after every subsequent step.size number of resamples the variances for all arrays are once again computed (over all resamples) and compared to the variances at the previous step. If the variances have all changed less than (max.var * min.diffpct) then the variances are assumed to have converged and the function will stop resampling on the current gene and move on to the next.

The end result is a MxN matrix of variances, where M is the number of arrays and N is the number of genes. The exprSet object that is returned by the RMA function (which contains the estimated RMA expression levels) contains a slot for standard errors, into which the square root of the matrix may be inserted for use in subsequent analyses.

RESULTS

Originally, this method was implemented fully in R and performed the full number of bootstraps on every gene. This turned out to be very time consuming.
Computing standard errors for the GDS mouse dataset (6 arrays, ~45000 genes, with a default of 1000 resamples) would have taken an estimated 640 hours of straight computing time on a modest machine (1.8 GHz PC with 1.5 gigs RAM running WinXP). Clearly this would not do.

One way to reduce the computation load is to reduce the number of genes upon which to perform the bootstrapping. When this is possible it works well and has a direct impact on the amount of computation needed. The amount of time required decreases in direct proportion to the reduction in genes. However this will not always be an option.

An important optimization is to use a C implementation of the median polish instead of the R version. By using a version that is based on the C code for the actual RMA function, this also has the benefit that the generated expression levels will be exactly the same as those generated by RMA rather than just an approximation (the bootstrap SE estimates won’t change much, but it’s better to be exact whenever possible.) Incorporating this C version of the median polish into the function results in a significant savings in computation time.

The other major optimization for the function is the use of the stopping criteria to cut the number of bootstraps. By periodically checking for convergence in the SE estimates, a huge amount of wasted effort can be avoided. The exact amount of time and computation saved will depend on the values of the passed stopping parameters.

To put it in perspective, the optimizations mentioned above reduced the run time of the function on the mouse dataset (6 arrays, 45000 genes) down to 8 hours. While 8 hours is still a significant amount of time, it is far more reasonable than the 640 for the original implementation.
DISCUSSION

One application for this is to use the inverse variances as weights in the lmFit function from the LIMMA package (Smyth 2004) to improve on the tests for differential expression. Some example R code for doing this is included at the end of Appendix B. After generating variance estimates for the first 1000 genes in the mouse dataset, the inverses of the variances were used as weights in the lmFit function. The effect of the weights on the test statistics is shown in Figure 5.

![Figure 5 - Weighted vs. unweighted test statistics from lmFit](image)

For this example, the test statistics are testing for a difference in expression level between 2 types of mice (wild type mice vs. ApoE deficient) i.e. testing for differential expression. It appears that for the majority of genes tested the weights didn’t have that big an effect, but there are a handful where it did.
Another possible application is to account for the variance when computing similarity measures to be used in clustering or for graphical methods like heatmaps where similarity between genes is important. Using the weights as a measure of the reliability of the expression level may improve the usefulness of these methods.

This paper has looked at a method for finding a reasonable estimate for the variance and standard errors of RMA expression levels. However, one subject that has not been explored but is still important is the covariance of the RMA expression levels for a gene between arrays. The exact nature of the covariance imparted by the median polish function is unknown. The background correction and normalization procedures also have an effect on this covariance. What is troubling to some statisticians is that this covariance is a side effect of the RMA preprocessing method, as opposed to being the result of a biological process in the observed organisms. Further study into the nature of this introduced dependence would be beneficial.
REFERENCES


Gentleman, R., Carey, V., Huber, W., Irizarry, R., Dudoit, S., (2005), Bioinformatics and Computational Biology Solutions Using R and Bioconductor. Springer


APPENDICES
APPENDIX A – An example of median polish

Here is an example of the median polish procedure performed on a small array. In this example, the rows can be thought of as the different arrays while the columns represent the probes for a specific gene. So the value in a particular cell \((i,j)\) is the background corrected, normalized log2 transformed perfect match probe intensity for array \(i\) and probe \(j\), for the gene of interest. The data values are made-up for this example. Starting with the original data, find the row medians.

<table>
<thead>
<tr>
<th></th>
<th>Probe 1</th>
<th>Probe 2</th>
<th>Probe 3</th>
<th>Probe 4</th>
<th>Probe 5</th>
<th>Probe 6</th>
<th>Probe 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.1</td>
<td>7.4</td>
<td>4.1</td>
<td>6.6</td>
<td>5.9</td>
<td>11.4</td>
<td>8.3</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>7.1</td>
<td>5.8</td>
<td>8.1</td>
<td>9.3</td>
<td>9.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8.7</td>
<td>11.2</td>
<td>5.6</td>
<td>9.1</td>
<td>3.8</td>
<td>7.4</td>
<td>10.3</td>
</tr>
<tr>
<td></td>
<td>4.8</td>
<td>4.6</td>
<td>3.8</td>
<td>5.5</td>
<td>7.9</td>
<td>8.8</td>
<td>7.3</td>
</tr>
<tr>
<td></td>
<td>7.9</td>
<td>8.8</td>
<td>7.4</td>
<td>8.8</td>
<td>13.2</td>
<td>7.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11.4</td>
<td>13.2</td>
<td>10.3</td>
<td>12.9</td>
<td>8.6</td>
<td>11.7</td>
<td>9.4</td>
</tr>
<tr>
<td></td>
<td>8.3</td>
<td>7.7</td>
<td>5.8</td>
<td>7</td>
<td>8.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row Medians</td>
<td>8.3</td>
<td>7.7</td>
<td>5.8</td>
<td>7</td>
<td>8.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subtract the corresponding row median from every element of the array, and then find the column medians.

Row medians removed:

<table>
<thead>
<tr>
<th></th>
<th>Probe 1</th>
<th>Probe 2</th>
<th>Probe 3</th>
<th>Probe 4</th>
<th>Probe 5</th>
<th>Probe 6</th>
<th>Probe 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-2.2</td>
<td>0.7</td>
<td>0.4</td>
<td>-3.5</td>
<td>-0.4</td>
<td>3.1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-0.3</td>
<td>-0.6</td>
<td>3.5</td>
<td>-3.1</td>
<td>1.1</td>
<td>5.5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-1.7</td>
<td>0</td>
<td>-0.2</td>
<td>-2</td>
<td>1.6</td>
<td>4.5</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>-0.4</td>
<td>2.1</td>
<td>1</td>
<td>-3.3</td>
<td>-1.5</td>
<td>5.9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-2.7</td>
<td>0.7</td>
<td>-0.6</td>
<td>-4.4</td>
<td>0</td>
<td>3.1</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column Medians:</td>
<td>-1.7</td>
<td>0.7</td>
<td>0.4</td>
<td>-3.3</td>
<td>0</td>
<td>4.5</td>
<td>0</td>
</tr>
</tbody>
</table>

Now subtract the column medians from each element of the array, and once again find the row medians.
Repeat these steps until convergence (although the process is not guaranteed to converge.) If at any time all of the medians are zero, then this is ‘convergence’ and the procedure stops. It can also stop if all of the medians are simply approaching zero but not reaching it. Alternately it can stop after a certain number of iterations. RMA will stop after 11 iterations, or if the sum of row medians changes by less than 0.01 from one iteration to the next. Continuing the example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>-0.3</td>
<td>0.2</td>
<td>0.2</td>
<td>0</td>
<td>-0.2</td>
<td>-1.2</td>
<td>0.2</td>
</tr>
<tr>
<td>A2</td>
<td>0.4</td>
<td>-2.3</td>
<td>2.1</td>
<td>-0.8</td>
<td>0.1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>A3</td>
<td>0</td>
<td>-0.7</td>
<td>0.6</td>
<td>1.3</td>
<td>1.6</td>
<td>0</td>
<td>1.5</td>
</tr>
<tr>
<td>A4</td>
<td>0.7</td>
<td>0.8</td>
<td>0</td>
<td>-0.6</td>
<td>-2.1</td>
<td>0.8</td>
<td>-0.6</td>
</tr>
<tr>
<td>A5</td>
<td>-0</td>
<td>1</td>
<td>0</td>
<td>-0.1</td>
<td>1</td>
<td>-0.4</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Subtracting the column medians one more time causes the subsequent row and column medians to all be zero. This means that the process has converged and can now stop.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>-0.3</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>-0.3</td>
<td>-1.2</td>
<td>0</td>
</tr>
<tr>
<td>A2</td>
<td>0.4</td>
<td>-2.5</td>
<td>2.1</td>
<td>0.7</td>
<td>0</td>
<td>0</td>
<td>-1.2</td>
</tr>
<tr>
<td>A3</td>
<td>0</td>
<td>-0.9</td>
<td>-0.6</td>
<td>1.4</td>
<td>1.5</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td>A4</td>
<td>0.7</td>
<td>0.6</td>
<td>0</td>
<td>-0.5</td>
<td>-2.2</td>
<td>0.8</td>
<td>-0.8</td>
</tr>
<tr>
<td>A5</td>
<td>-0</td>
<td>0.8</td>
<td>0</td>
<td>0</td>
<td>0.9</td>
<td>-0.4</td>
<td>1.6</td>
</tr>
</tbody>
</table>

|     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |

|     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |

|     | -0.2  | 1     | 0     | 0.6   | -1    | -0.4  | 1.8   |

|     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
What is left is what Tukey calls the residuals. Subtract these residuals from the original data to get the "Fit." After computing the fit, once again find the row and column medians.

"Fit" (original data - residuals)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>Row Medians</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>6.4</td>
<td>9</td>
<td>8.5</td>
<td>4.7</td>
<td>8.2</td>
<td>12.6</td>
<td>8.3</td>
<td>8.3</td>
</tr>
<tr>
<td>A2</td>
<td>7</td>
<td>9.6</td>
<td>9.1</td>
<td>5.3</td>
<td>8.8</td>
<td>13.2</td>
<td>8.9</td>
<td>8.9</td>
</tr>
<tr>
<td>A3</td>
<td>4.1</td>
<td>6.7</td>
<td>6.2</td>
<td>2.4</td>
<td>5.9</td>
<td>10.3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>A4</td>
<td>5.9</td>
<td>8.5</td>
<td>8</td>
<td>4.2</td>
<td>7.7</td>
<td>12.1</td>
<td>7.8</td>
<td>7.8</td>
</tr>
<tr>
<td>A5</td>
<td>5.9</td>
<td>8.5</td>
<td>8</td>
<td>4.2</td>
<td>7.7</td>
<td>12.1</td>
<td>7.8</td>
<td>7.8</td>
</tr>
</tbody>
</table>

Column Medians: 5.9 8.5 8 4.2 7.7 12.1 7.8

Take any row from the Fit and subtract out its median to get the column effects.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>-1.9</td>
<td>0.7</td>
<td>0.2</td>
<td>-3.6</td>
<td>-0.1</td>
<td>4.3</td>
<td>0</td>
</tr>
</tbody>
</table>

This will be the same no matter which row is selected. Take any column from the Fit and subtract out the row medians to get the row effects.

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>-1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now select any column from the fit, and this time subtract out the row effect to get a new column of adjusted values, then find the median of this new column.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>8.3</td>
</tr>
<tr>
<td>A2</td>
<td>8.9</td>
</tr>
<tr>
<td>A3</td>
<td>6</td>
</tr>
<tr>
<td>A4</td>
<td>7.8</td>
</tr>
<tr>
<td>A5</td>
<td>7.8</td>
</tr>
</tbody>
</table>
This median of this column is the overall mean, which in this case it 7.8. So at the end of
the procedure there is an overall mean, a vector of column effects, a vector of row effects,
and a matrix of residuals.

<table>
<thead>
<tr>
<th>Overall mean</th>
<th>7.8</th>
<th>-1.9</th>
<th>0.7</th>
<th>0.2</th>
<th>-3.6</th>
<th>-0.1</th>
<th>4.3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>-0.3</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>-0.3</td>
<td>-1.2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1.1</td>
<td>0.4</td>
<td>-2.5</td>
<td>2.1</td>
<td>-0.7</td>
<td>0</td>
<td>0</td>
<td>-1.2</td>
<td>0</td>
</tr>
<tr>
<td>Row Effects:</td>
<td>-1.8</td>
<td>0</td>
<td>-0.9</td>
<td>-0.6</td>
<td>1.4</td>
<td>1.5</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td>0</td>
<td>0.7</td>
<td>0.6</td>
<td>0</td>
<td>-0.5</td>
<td>-2.2</td>
<td>0.8</td>
<td>-0.8</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-0</td>
<td>0.8</td>
<td>0</td>
<td>0</td>
<td>0.9</td>
<td>-0.4</td>
<td>1.6</td>
<td>0</td>
</tr>
</tbody>
</table>

The original data can be represented as $Y_{ij} = M + R_i + C_j + E_{ij}$, where $M$ is the overall
mean, $R_i$ is the row effect for row $i$, $C_j$ is the column effect for column $j$, and $E_{ij}$ is the
residual for cell $i,j$.

The RMA expression level is taken to be the overall mean plus the array effect (which in
this case is the row effect.) The figure below shows the estimated expression levels next
to the original data, along with the row means and medians for comparison.

![](image)

In this case the RMA expression level for each row (array) is fairly close to either the
mean or median. It does seem to be a reasonable and robust estimate.
Note that the \textit{RMA} function in the Bioconductor suite in R does not use this algorithm exactly. It uses an “optimized” version of this function that builds the row and column effects over each iteration, rather than computing them at the end. The version in the Affy package gives slightly different results than you get if you do a median polish using the \textit{medpolish} function in the base R package. Also, the RMA function has gone through at least one modification that changed the estimated expression levels (see the RMA help in R.) If you need to get the exact RMA expression levels then you’ll need to use the actual RMA functions.
APPENDIX B – R Source code

```
# rma.bootstrapvar(affy.object, num.resamples, subset, min.vardiff, save.file, filename,
#                 bg.correct, normalize)
# affy.object must be an affybatch object that has already been BG corrected and
# normalized
# gene.subset should be a vector of gene names upon which to do the bootstrapping
# min.resamples is the minimum number of resamples to perform. In actuality the
# minimum number
# is min.resamples+step.size. The way it works is that after doing min.resamples
# number of resamples the function finds the variances up to that point. It then
# proceeds to do step.size more resamples before recomputing the variances (using
# all of the bootstrap generated data up to that point.) If the change in variances
# are all less than min.diffpct then the function moves on to the next gene
# step.size is the number of resamples to do between checking the variances to see if
# it should stop
# min.diffpct is a number specifying a minimum difference in variances from one step to
# the next. If the proportional change in variances between steps is less than
# min.diffpct then the variances are assumed to have 'converged' and the function
# moves on to the next gene
# max.resamples is the maximum number of resamples to perform. The function will only
# ever do up to max.resamples if max.resamples number of bootstraps, even if the
# variances have not settled down. If max.resamples is not greater than
# min.resamples then it is made to be min.resamples+1
#
# do.bg.correct and do.normalize are both included as options in case you want to apply
# your own background correction or normalization before passing the object to this
# function. This might be useful if you're interested in using GCRMA background
# correction instead of RMA convolution for example
# By default, the function will save the resulting data frame to a file before exiting.
# The resulting .Rdata file contains an object named 'var.dataframe' with a row for
# each gene and a column for each array. The value in each cell is the estimated
# variance.
# The function returns a data frame containing the bootstrap variances for each gene
# for each array.

rma.bootstrapvar <- function(affy.object, gene.subset=NULL, min.diffpct=0.05,
                             min.resamples=100, step.size=100, max.resamples=1000, save.file=TRUE, filename=NULL,
                             bg.correct=TRUE, do.normalize=TRUE)
{
  ## warning message
  cat('
')
  cat('WARNING -- This function can take an EXTREMELY long time to run!', '
')
  cat('Make sure that you have read the documentation thoroughly.', '
')

  ## Perform background correction unless directed otherwise
  if (do.bg.correct == TRUE)
  {
    cat('Performing RMA convolution background correction', '
')
    affy.object <- bg.correct.rma(affy.object)
  }

  ## Perform normalization unless directed otherwise
  if (do.normalize == TRUE)
  {
    cat('Performing quantile normalization', '
')
    affy.object <- normalize(affy.object, method="quantiles")
  }

  narrays <- length(affy.object)

  ## if no subset gets passed then all genes will be processed
  if (is.null(gene.subset))
  { gene.subset <- geneNames(affy.object) }

  num.genes <- length(gene.subset)

  ## extract the probe intensities and put them in a structure for easy retrieval
```
```r
#cat('Retrieving PM intensities', '
')
#pm.intensity.mat <- probeset(affy.object, geneNames(affy.object)[gene.subset])

## Set up the structures to hold the bootstrapped intensities, resulting variances and initial variances
out.expr <- rep(0, narrays)
gene.variances <- array(data=NA, dim=c(num.genes, narrays))
initial.variances <- array(data=NA, dim=c(min.resamples, narrays))
old.variances <- rep(0, narrays)
new.variances <- rep(0, narrays)

## check to see that parameters make sense
if (min.resamples < 1)
{ min.resamples <- 1 }
if (max.resamples <= min.resamples)
{ max.resamples <- min.resamples + 1 }

#########################################################################
## THE BIG LOOP ###################################################################
#########################################################################

cat('BEGINNING MAIN LOOP', '
')
begintime <- date()
for (current.gene in 1:num.genes)
{
  ##current.probes <- pm.intensity.mat[[current.gene]]@pm
  current.probes <- probeset(affy.object, gene.subset[current.gene])[[1]]@pm
  slower, but gives the correct values :P
  nprobes <- length(current.probes[l,])
  resample.vector <- seq(1:nprobes)
  out.expressions <- matrix(data=0, nrow=narrays, ncol=max.resamples)

  ## do the initial min.resamples number of bootstraps and check the variance at each step, saving the max
  for (current.resample in 1:min.resamples)
  { 
    resampled.probes <- rbind(current.probes[sample(resample.vector, nprobes, replace=TRUE)],)
    gene.mat <- t(resampled.probes)
    results <- .c("rbv_median_polish", gene.mat, as.integer(nprobes),
                  as.integer(narrays), out.expr)
    out.expressions[,current.resample] <- results[[4]]
  }
  if (current.resample > 1)
  {
    initial.variances[current.resample,] <-
    apply(out.expressions[,1:current.resample], 1, var)
  }

  old.variances <- initial.variances[current.resample,]
  new.variances <- old.variances
  vardiff <- rep(0, narrays)
  ## set the minimum difference stopping criteria to be min.diffpct * the maximum of the observed variances
  ## for the initial rnu of bootstraps. this is taken over 2:current.resample because the 1st row will be NAS
  ## due to taking the variance of a single observation
  min.vardiff <- max(initial.variances[2:current.resample],) * min.diffpct
  STOP <- FALSE
  ## continue generating bootstraps until the variances 'converge' or until we reach the maximum number
  ## of bootstraps specified by the user. The bootstraps are done in chunks of size step.size
  while (STOP == FALSE)
  {
    loop.start <- current.resample + 1
    loop.end <- loop.start + step.size - 1
    if (loop.end > max.resamples)
    { loop.end <- max.resamples }
    for (current.resample in loop.start:loop.end)
    { resampled.probes <- rbind(current.probes[sample(resample.vector, nprobes, replace=TRUE)],)
      gene.mat <- t(resampled.probes)
      results <- .c("rbv_median_polish", gene.mat, as.integer(nprobes),
                   as.integer(narrays), out.expr)
      out.expressions[,current.resample] <- results[[4]]
    }
  }

STOP <- FALSE
```
if (loop.end == max.resamples)
{
    STOP <- TRUE
}
new.variances <- apply(out.expressions[,1:loop.end], 1, var)
vardiff <- abs(old.variances - new.variances)
old.variances <- new.variances

## if the biggest change in variance (out of all the arrays) is smaller than
## 'convergence' and we stop doing bootstraps for this gene and move on to the
## next
if (max(vardiff) < min.vardiff)
{
    STOP <- TRUE
}
gene.variances[current.gene,] <- new.variances
}
endtime <- date()
cat('Starting time: ', begin.time, '\n')
cat('Ending time: ', endtime, '\n')
var.dataframe <- data.frame(gene.variances)
names(var.dataframe) <- rownames(pData(affy.object))
rownames(var.dataframe) <- gene.subset

## save the results to a file unless the user has elected not to do so
if (save.file == TRUE)
{
    ## if no filename has been specified (the default) then create an appropriate one to
    use
    if (is.null(filename))
    {
        trimmed.date <- gsub("\:", "", date(),)
        filename <- paste("rmabootstrapvar", trimmed.date, ".Rdata", sep="")
    }
    save(var.dataframe, file=filename)
cat ('Your data has been saved into ', filename, '\n')
}

## return the resulting dataframe and exit
return(var.dataframe)

medpolish.rma <- function(pm.mat)
{
    nprobes <- as.integer(length(pm.mat[,1]))
narrays <- as.integer(length(pm.mat[1,]))
dummy.vec <- rep(0, narrays)
exprs <- C("rbv_median_polish", pm.mat, nprobes, narrays, dummy.vec)[[4]]
return(exprs)
}

## code to find bootstrap SEs for specific genes using RMA expression levels
library(affy)
setwd("C:\Documents and Settings\Gabe\Desktop\School\Project")
dyn.load("rbvmedpolish.dll")
source("rmabootstrapvar v3.R")
gds.data <- ReadAffy(filenames = c("GSM44658.CEL", "GSM44663.CEL", "GSM44659.CEL", "GSM44660.CEL", "GSM44661.CEL", "GSM44662.CEL"))
gds.bgcorrected <- bg.correct.rma(gds.data)
gds.normalized <- normalize(gds.bgcorrected, method="quantiles")
affy.object <- gds.normalized

num.resamples <- 1000
my.ids <- sample(1:length(geneNames(affy.object)), 10)

pm.indices <- pmindex(affy.object)
gene.names <- geneNames(affy.object)
um.arrays <- length(affy.object)
gene.subset <- gene.names[my.ids]
um.genes <- length(affy.object[,1])
if (length(gene.subset) != 0)
num.genes <- length(gene.subset)

num.probes <- array(data=NA, dim=num.genes)
for (i in 1:num.genes) { num.probes[i] <- length(probe.intensities[[i]]@pm) / num.arrays }

out.expressions <- array(data=0, dim=c(num.arrays, num.genes, num.resamples))
out.se <- array(data=NA, dim=c(num.arrays, num.genes, num.resamples))
naive.se <- matrix(data=NA, nrow=num.arrays, ncol=num.genes)

gene.naive.RMA <- function(y)
{
  ybar.i.n <- apply(y,2,mean)
  ybar.jn <- apply(y,1,mean)
  ybar..jn <- t(matrix(rep(ybar..jn,ncol(y)),ncol=ncol(y)))
  ybar...n <- matrix(ybar...n,nrow=nrow(y),ncol=ncol(y))
  yhat <- Ybar.i.n-Ybar..jn-Ybar...n
  RSS <- sum((y-yhat)^2)
  df <- (nrow(y)-1)*(ncol(y)-1)
  mu.hat <- ybar.i.n
  var.mu.hat <- rep((RSS/df/nrow(y)),ncol(y))
  se.mu.hat <- sqrt(var.mu.hat)
  return(cbind(mu.hat,se.mu.hat))
}

for (current.gene in 1:num.genes)
{
  current.probes<- probeset(affy.object, gene.subset[current.gene])[[1]]@pm
  naive.se[,current.gene] <- gene.naive.RMA(log2(current.probes))[,2]
  num.probes <- length(current.probes[,1])
  resample.vector <- seq(1:num.probes)
  for (current.resample in 1:num.resamples)
  {
    resampled.probes <- rbind(current.probes[sample(resample.vector, replace=TRUE),J])
    processed<- medpolish.rma(resampled.probes)
    out.expressions[,current.gene,current.resample] <- processed
    if (current.resample > 1) {
      out.se[,current.gene,current.resample] <- apply(out.expressions[,current.gene,1:current.resample], 1, sd) }
  }

  ## change GENE and copy/paste the following section of code to make a plot of SEs vs. resample for whichever GENE you select
  GENE=10
  plot(c(1:1000), out.se[1,GENE,], type='l', main="Estimated standard errors at each resample step", xlab="Resample #", ylab="Standard Error", ylim=c(0, max(out.se[,GENE,2:num.resamples])))
  abline(h=naive.se[1,GENE], lty=2)
  for (current.array in 2:num.arrays)
  { lines(c(1:1000), out.se[current.array,GENE,]) }
  naive.se[1,GENE]
}

library(limma)
type <- c(0,0,0,1,1,1)
design <- cbind(Intercept=1,type=type)
# make 'original' fit
fit <- lmFit(eset,design)
e.fit <- eBayes(fit)
top.all <- topTable(e.fit,coef=2,adjust="BH")
gn.eB <- top.all$ID[top.all$adj.P.Val<0.05]
t <- order(top.all$ID)
top.all <- top.all[t,]
# make 'weighted' fit
gene.subset <- geneNames(gds.data)[1:1000]
## the bg corrected normalized intensities should still be in memory from earlier
## if they're not then set do.bg.correct and do.normalize to TRUE (or just leave
## them out of the call completely, and they'll default to TRUE)
data.variances <- rma.bootstrapvar(gds.normalized, gene.subset=gene.subset,
do.bg.correct=FALSE, do.normalize=FALSE)

# use the inverse of the variances for weights
wt.mat <- 1/(as.matrix(data.variances))
w.fit <- lmFit(eset, design, weights=wt.mat)
w.e.fit <- eBayes(w.fit)
w.top.all <- topTable(w.e.fit, n=nrow(eset), coef=2, adjust="BH")
w.gn.eB <- w.top.all$ID[w.top.all$adj.P.Val<0.05]
t <- order(w.top.all$ID)
w.top.all <- w.top.all[,]

plot(top.all$t, w.top.all$t, xlab='Test statistic from unweighted linear model', ylab='Test statistic from weighted linear model')
abline(0,1)
APPENDIX C – C code

/src/rma/rma.c

```c
#include <rma_structures.h>
#include <rma_common.h>
#include "rma_background2.h"
#include <qnorm.h>
#include <R.h>
#include <Rdefines.h>
#include <Rmath.h>
#include <Rinternals.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#defines
#defines

#include <rma_structures.h>
#include <rma_common.h>
#include "rma_background2.h"
#include <qnorm.h>
#include <R.h>
#include <Rdefines.h>
#include <Rmath.h>
#include <Rinternals.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double rbv_median(double *x, int length);
double rbv_sum_abs(double *z, int rows, int cols);
void rbv_get_row_median(double *x, double *rdelta, int rows, int cols);
void rbv_get_col_median(double *z, double *cdelta, int rows, int cols);
void rbv_subtract_by_row(double *x, double *rdelta, int rows, int cols);
void rbv_subtract_by_col(double *z, double *cdelta, int rows, int cols);
void rbv_rmod(double *r, double *rdelta, int rows);
void rbv_cmod(double *c, double *cdelta, int cols);
int rbv_compare_double(const void *a, const void *b);

/measures
/measures
```

---

THE MAJORITY OF THIS CODE HAS BEEN COPIED DIRECTLY FROM
THE RMA2.C FILE INCLUDED IN THE RMA SOURCE CODE

The original author for that code is B.M. Bolstad

The compare_double function was copied from the example code
given for comparison functions at the GNU.org website. No
author was listed.

The only function which has changed significantly is the
median_polish function. It has been modified to accept a
MxN matrix of background corrected normalized perfect-match
intensities, where M is the number of arrays, and N is the
number of probes. You can call this from within R using
the medpolish.rma() wrapper function.

The only other significant change from the RMA2.c code is
that the function names have all been changed ("rbv." prefix)
to avoid any potential problems of name collisions with the
otherwise identical RMA versions. This may not have been
necessary, but the compiled .dll file is only 7.5k, so meh.

---

/* #include "rma_structures.h" */
#include <rma_common.h>
#include "rma_background2.h"
#include "qnorm.h"
#include <R.h>
#include <Rdefines.h>
#include <Rmath.h>
#include <Rinternals.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double rbv_median(double *x, int length);
double rbv_sum_abs(double *z, int rows, int cols);
void rbv_get_row_median(double *x, double *rdelta, int rows, int cols);
void rbv_get_col_median(double *z, double *cdelta, int rows, int cols);
void rbv_subtract_by_row(double *x, double *rdelta, int rows, int cols);
void rbv_subtract_by_col(double *z, double *cdelta, int rows, int cols);
void rbv_rmod(double *r, double *rdelta, int rows);
void rbv_cmod(double *c, double *cdelta, int cols);
int rbv_compare_double(const void *a, const void *b);

** measures
** measures
```
void rbv_median_polish(double *data, int *num_probes, int *num_arrays, double *results)
{
    int i,j,iter;
    int maxiter = 10; // counters
    // the maximum number
    of iterations before giving up
    int nprobes = *num_probes, narrays = *num_arrays;
    double eps=0.01; // a minimum
difference, determines when we stop
    double oldsum = 0.0, newsum = 0.0; // variables used to compute a
difference to compare against eps
    double t = 0.0; // the overall "mean"
    // (not the same as a regular arithmetic mean)
    double delta; // a general
    // use variable?
    double *rdelta = calloc(nprobes, double); // the row effects (built up over iterations)
    double *cdelta = calloc(narrays, double); // the column effects (built up over
    // iterations)
    double *r = calloc(nprobes, double); // stores the row effect
    double *c = calloc(narrays, double); // stores the column effect
    double *z = calloc(nprobes*narrays, double); // a copy of the data matrix, upon
    // which to work
    // z is an array or double that holds the log2 intensities
    // the columns of z correspond to the arrays
    // the rows of z correspond to the probe intensities across arrays
    for (i = 0; i < nprobes * narrays; i++)
    z[i] = log(data[i])/log(2.0);
    for (iter = 1; iter <= maxiter; iter++) // perform up to 10
        iterations
        {
            rbv_get_row_median(z, rdelta, nprobes, narrays); // get the row medians
            rbv_subtract_by_row(z, rdelta, nprobes, narrays); // subtract the row medians
            from each element
            rbv_rmod(r, rdelta, nprobes); // update the row effects
            delta = rbv_median(c, narrays);
            for (j = 0; j < narrays; j++)
            { c[j] = c[j] - delta; }
            t = t + delta; // update the overall 'mean' effect
            rbv_get_col_median(z, cdelta, nprobes, narrays); // get the column medians
            rbv_subtract_by_col(z, cdelta, nprobes, narrays); // subtract the column
            medians from each element
            rbv_cmmd(c, cdelta, narrays); // update the column effects
            delta = rbv_median(r, nprobes);
            for (i = 0; i < nprobes; i++)
            { r[i] = r[i] - delta; }
            t = t + delta; // update the overall 'mean' effect
            newsum = rbv_sum_abs(z, nprobes, narrays);
            if (newsum == 0.0 || fabs(1.0 - oldsum/newsum) < eps)
                break;
            oldsum = newsum;
    }
    for (j=0; j < narrays; j++)
    { results[j] = t + c[j]; }
    Free(rdelta);
    Free(cdelta);
    Free(r);
    Free(c);
    Free(z);
}

int rbv_compare_double(const void *elem1, const void *elem2)
{
    // This code was lifted from the Gnu.org webpage
    // It was the example code for a compare function
    const double *da = (const double *)elem1;
    ...
const double *db = (const double *)elem2;
return (*da > *db) - (*da < *db);
}

double rbv_median(double *x, int length){
int i;
int half;
double *buffer = Calloc(length, double);
for (i = 0; i < length; i++)
buffer[i] = x[i];
gsort(buffer, length, sizeof(double), (int(*)(void*, const void*))rbv_compare_double);
half = (length + 1)/2;
if (length % 2 == 1){
med = buffer[half - 1];
}
else{
med = (buffer[half] + buffer[half-1])/2.0;
}
Free(buffer);
return med;
}

double rbv_sum_abs(double *z, int rows, int cols){
int i, j;
double sum = 0.0;
for (i=0; i < rows; i++)
for (j=0; j < cols; j++)
sum+=fabs(z[j*rows+i]);
return sum;
}

void rbv_get_row_median(double *z, double *rdelta, int rows, int cols){
int i,j;
double *buffer = Calloc(cols, double);
for (i = 0; i < rows; i++){
for (j = 0; j < cols; j++)
buffer[j] = z[j*rows+i];
}
double *buffer = calloc(rows, double);
for (j = 0; j < cols; j++){
    for (i = 0; i < rows; i++){
        buffer[i] = z[j * rows + i];
    }
    cdelta[j] = rbv_median(buffer, rows);
}
Free(buffer);

void rbv_get_col_median(double *z, double *cdelta, int rows, int cols){
    int i, j;
    double *buffer = calloc(rows, double);
    for (j = 0; j < cols; j++){
        for (i = 0; i < rows; i++){
            buffer[i] = z[j * rows + i];
        }
        cdelta[j] = rbv_median(buffer, rows);
    }
    Free(buffer);
}

void rbv_subtract_by_row(double *z, double *rdelta, int rows, int cols){
    int i, j;
    for (i = 0; i < rows; i++){
        for (j = 0; j < cols; j++){
            z[j * rows + i] -= rdelta[i];
        }
    }
}

void rbv_subtract_by_col(double *z, double *cdelta, int rows, int cols){
    int i, j;
    for (j = 0; j < cols; j++){
        for (i = 0; i < rows; i++){
            z[j * rows + i] -= cdelta[j];
        }
    }
}
/***********************************/
**
** void rbv_rmod(double *r, double *rdelta, int rows)
**
** double *r - vector of length rows
** double *rdelta - vector of length rows
** int rows, cols dimensions of matrix
**
** add elementwise *rdelta to *r
**
***********************************************************************************/

void rbv_rmod(double *r, double *rdelta, int rows){
    int i;
    for (i = 0; i < rows; i++){
        r[i] = r[i] + rdelta[i];
    }
}

***********************************************************************************/
**
** void rbv_cmod(double *c, double *cdelta, int cols)
**
** double *c - vector of length rows
** double *cdelta - vector of length rows
** int cols length of vector
**
** add elementwise *cdelta to *c
**
***********************************************************************************/

void rbv_cmod(double *c, double *cdelta, int cols){
    int j;
    for (j = 0; j < cols; j++){
        c[j] = c[j] + cdelta[j];
    }
}
APPENDIX D – Example R code to use the function

```r
library(affy)
setwd("C:\\data")
dyn.load("rbvmedpolish.dll")
source("rmabootstrapvar.R")
data(affybatch.example)

## compute SEs for all genes, don't save the results to a file
results1 <- rma.bootstrapvar(affybatch.example, save.file=FALSE)

## only do it for a subset of genes
gene.subset <- geneNames(affybatch.example)[20:50]
results2 <- rma.bootstrapvar(affybatch.example, gene.subset=gene.subset, save.file=FALSE)

## get the RMA expression level for a single gene using medpolish.rma()
example.bg <- bg.correct.rma(affybatch.example)
example.no <- normalize(example.bg, method="quantiles")
gene.pm <- probeset(example.no, geneNames(example.no)[1])[1]@pm
example.rma.single <- medpolish.rma(gene.pm)
```
A METHOD FOR FINDING STANDARD ERROR ESTIMATES FOR RMA EXPRESSION LEVELS USING BOOTSTRAP

By

Gabriel Nicholas

A report submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Statistics

Utah State University
Logan, Utah

2007