GridWeaver: A Fully-Automatic System for Microarray Image Analysis using Fast Fourier Transforms

John Paul C. Vergara[1], Lenwood S. Heath[2],
Ruth Grene[3], Naren Ramakrishnan[2], and Layne T. Watson[2]

[1] Department of Information Systems and Computer Science
Ateneo de Manila University, Quezon City 1108, Philippines
[2] Department of Computer Science
Virginia Tech, Blacksburg VA 24061, USA
[3] Department of Plant Pathology, Physiology, and Weed Science
Virginia Tech, Blacksburg VA 24061, USA

Abstract

Motivation: Experiments using microarray technology generate large amounts of image data that are used in the analysis of genetic function. An important stage in the analysis is the determination of relative intensities of spots on the images generated.

Results: This paper presents *GridWeaver*, a program that reads in images from a microarray experiment, automatically locates subgrids and spots in the images, and then determines the spot intensities needed in the analysis of gene function. Automatic gridding is performed by running Fast Fourier Transforms on pixel intensity sums. Tests on several data sets show that the program responds well even on images that have significant noise, both random and systemic.

Availability: Source code (written in C, compiled using `gcc` under the Linux operating environment) is available from `http://bioinformatics.cs.vt.edu/~jpv/gridweaver` for academic use.

Contact: `jpvergara@ateneo.edu`.

Supplementary Information: For information on the *Expresso* project, visit: `http://bioinformatics.cs.vt.edu/expresso/`.

1 Introduction

The introduction of microarray technology in the mid-1990s has accelerated studies on gene expression in recent years. A single microarray experiment involves thousands of gene samples, providing large-scale gene expression data. The strategy involves hybridizing target cDNAs (complementary DNA) with an array of DNA probes deposited by a robotic printer on a series of glass slides. The cDNAs are labeled with distinct fluorescent dyes, poured over the slides, and then washed off, leaving cDNA that bind with the gene samples. A laser-scanning device then scans the slides to produce digitized images that reveal the extent of bound cDNA on each sample.

The images produced by the scanner are processed through image analysis software. Each sample corresponds to a spot in the image and the software computes the resulting intensities for each spot. Often, a test cDNA target and a reference cDNA target are used, and the relative intensities exhibited by each target for each sample are used to derive conclusions on gene expression.

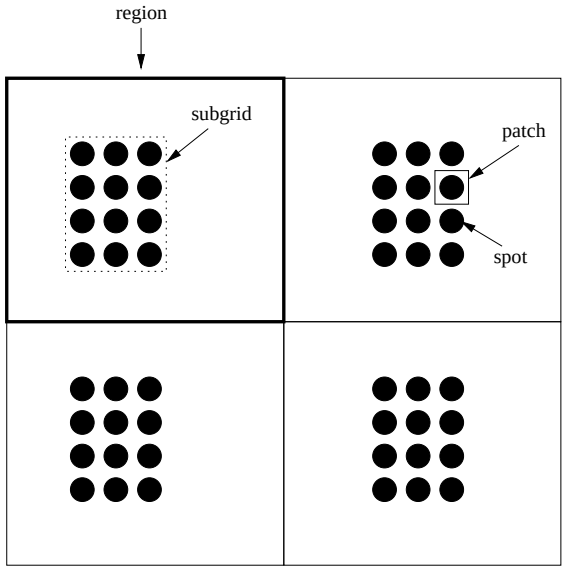

Figure 1: Subgrids and spots in a microarray image.

There has been considerable attention devoted to the automatic analysis of microarray images [1, 2, 3, 4, 5, 6, 9, 10, 11, 16, 17, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 31]. Earlier methods and software have often required much human intervention, particularly with respect to locating the spots in the images. Significant progress has been made on minimizing this human intervention and a variety of methods have been studied such as using morphological operations [1, 19, 23], graph theoretical approaches [22], Markov models [10, 24], and various statistical methods [3, 21].

This paper discusses *GridWeaver*, a microarray image analysis system that automatically locates spots in a microarray image using *Fast Fourier Transforms* (FFTs) [8]. *GridWeaver* is part of the *Expresso* project [18, 29]. The project seeks to automate all stages in microarray experiments, including microarray image analysis.

An image that a microarray scanner produces is, in its raw form, an $h \times w$ matrix of pixel intensity values (values typically range from 0 to 65535). However, it can be viewed logically as an $m \times n$ array of *subgrids*, where each subgrid contains an $s \times t$ array of *spots*. Figure 1 illustrates an example of this logical view of a microarray image. In the figure, the values for m, n, s, and t, are 2, 2, 4, and 3, respectively. A *region* is a rectangular area that bounds a subgrid, while a *patch* is a rectangular area that bounds a spot.

The image analysis system we discuss in this paper performs the necessary computations on the input images given only the values for m, n, s, and t. The system is an update of an earlier prototype [12], which had the following limitations:

- Subgrid regions were assumed to be evenly divided along the image. We have encountered images where the subgrids were relatively close to each other and the distance between the edge of the image and the subgrid closest to the edge is significant. This renders the previous method of identifying subgrid regions inadequate.

- The prototype was susceptible to arbitrary noise such as bright bands of pixels and other artifacts outside the spots and subgrids.

For the current version, the above limitations are addressed through more sophisticated uses of FFTs.

GridWeaver operates in four phases:

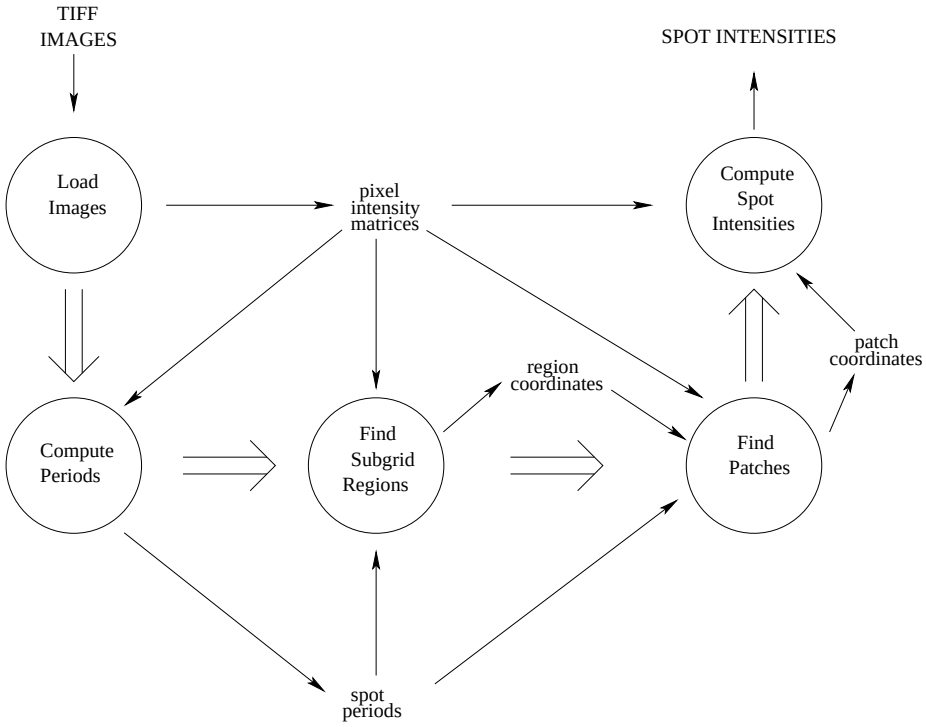

Figure 2: Image Analysis Components.

1. **Spot Period Computation.** Using FFTs on the row and column sums of pixel intensities in the image, this phase determines the vertical and horizontal distance between spots in a subgrid.

2. **Subgrid Region Identification.** This phase determines the regions that contain the individual subgrids, through further analysis of the row and column intensity sums extracted from the raw image. FFTs are obtained along the margins of the image to determine the edges of the subgrids that are closest to the edges of the image. The subgrid regions for all subgrids are then determined from these edges. The regions are described by rectangles that contain each subgrid (see Figure 1).

3. **Patch and Spot Identification.** In this phase, coordinates of the subgrids are obtained by analyzing their respective regions. Patches for each spot are then derived from these coordinates.

4. **Spot Intensity Computation.** This phase analyzes each patch to determine which pixels are part of a spot. For each spot, foreground and background intensities are computed along with other statistical information. The method used for computing spot intensities are consistent with the one introduced by Eisen and Brown [13, 14], although the software we develop provides some flexibility in this phase to allow for alternative methods (see [7], for example).

Figure 2 presents a diagram depicting the different phases and their relationships. The figure includes an initial "Load Images" phase that reads in the TIFF images and stores them as pixel intensity matrices. The four phases described above then use these matrices to perform their respective computations.

2 Systems and Methods

In this section, we discuss the key methods used in *GridWeaver*.

Fast Fourier Transforms

Row and column pixel intensity sums in an image are analyzed using Fast Fourier Transforms (FFTs). FFTs are applied to a signal that plots amplitudes over a *time* domain, to produce a corresponding signal that plots amplitudes over a *frequency* domain. An amplitude in the frequency domain depends on the amplitude and range in the time domain where the given frequency occurs. Essentially, the signal is broken down into a series of sinusoids with different frequencies.

In our context, we can consider a pixel intensity sum for a given row (or column) as the amplitude for that row (or column). Our input signal thus consists of these sums over the sequence of all rows (or columns) in the image. Observing that peaks that are almost equally spaced occur around the spot centers, applying an FFT to this signal yields a high amplitude for the frequency that corresponds to the regular occurrence of spots within a row (or column). The reciprocal of this frequency provides a period or distance between the spot centers. FFT analysis over the rows provide a vertical period, while analysis over the columns provide a horizontal period (see phase 1 described in Section 1). In turn, these periods aid in determining the coordinates of the regions and the patches for each spot in the image.

Spot Intensity Data

Eisen and Brown [13, 14] suggest a method for calculating intensities and ratios given a patch and an expected spot diameter. The method attempts to address the effects of background noise around each spot. First, spot pixels are distinguished from background pixels by assuming that the spot center is the center of the patch and by using spot diameter information to determine which pixels land within the spot circle. A local background intensity B for a patch is computed as the median intensity of all background pixels. Let I be the mean or median intensity for spot pixels in a patch. The background-corrected intensity for a spot is computed as $I' = I - B$.

In *GridWeaver*, spot centers and diameters are determined on a per-spot basis by analyzing the pixel intensity sums of a patch. Intensities for both background and foreground pixels are computed using these centers and diameters. Both mean and median intensities are provided as output to allow users the flexibility of choosing which values to use for ratio computation.

3 Algorithms

In this section, key algorithms in the gridding process are presented. Figure 3 shows the main algorithm for *GridWeaver*. The inputs to the algorithm consist of an image given as an array of pixel intensities and subgrid and spot counts along the x and y axes of the image. The algorithm returns patch information and spot intensities for each spot in the image. Four algorithms are invoked in this main algorithm, and they correspond to the different phases described in Section 1. We describe in more detail phase 2 (Algorithm FindRegions, line 2) and phase 3 (Algorithm FindPatches, lines 6–7), in this section.

Finding Subgrid Region Coordinates

Algorithm FindRegions in Figure 4 describes how region coordinates for each subgrid are obtained. The algorithm assumes that horizontal and vertical spot periods have been computed and uses these periods together with spot parameters to compute region coordinates. Margins

GridWeave(*Pix*, *height*, *width*, *numsubx*, *numsuby*, *numspotx*, *numspoty*)

INPUT: *Pix*: a two-dimensional array of pixels of an image
height: height of the image
width: width of the image
numsuby: number of subgrids along the y-axis
numsubx: number of subgrids along the x-axis
numspoty: number of spots for each subgrid column
numspotx: number of spots for each subgrid row
OUTPUT: Patch information and spot intensities for each spot in the image

```
1   rowperiod, colperiod ← GetPeriods(Pix, height, width)
2   regions ← FindRegions(Pix, height, width, numsubx, numsuby, rowperiod, colperiod)
3   for subrow ← 1 to numsuby
4       do for subcol ← 1 to numsubx
5             do allpatches[subrow, subcol] ←
6                     FindPatches(Pix, regions[subrow, subcol], rowperiod, colperiod,
7                                 numspoty, numspotx)
8                allintensities[subrow, subcol] ←
9                     ComputeIntensities(allpatches[subrow, subcol])
10
11  return allpatches, allintensities
```

Figure 3: Algorithm GridWeave

along all four sides of the image are first obtained (lines 1–5). Since the subgrids are evenly spaced, region coordinates are easily determined from these margins (lines 7–15).

Algorithm GetLeftMargin in Figure 5 demonstrates how a margin is obtained. Similar algorithms for GetRightMargin, GetTopMargin, and GetBottomMargin apply. The algorithm uses a **while** loop (lines 3–16) that tries candidate left margins spaced *colperiod* apart beginning with coordinate 1. For each candidate margin, pixel intensity sums are computed on each row along the area between the left edge of the image and the candidate margin (lines 5–7). The FFT is then computed on these row sums and the frequency with the maximum amplitude is obtained (lines 8–11). For a left margin, the area would not contain spots, so running an FFT on row sums along the area should not yield the expected spot period. Once a candidate margin yields the expected (row) spot period, this indicates that the margin has been crossed (lines 12–15). The value returned is the candidate margin minus one (column) spot period, to allow for slack (lines 17–18).

Finding Patches

Algorithm FindPatches, shown in Figure 6, determines subgrid coordinates in a region as follows. The width and height of a subgrid is first computed using period and spot count values (lines 1–2). Left and top margins within the subgrid region are then obtained (lines 4–8). Corresponding right and bottom margins need not be computed since the subgrid dimensions are already available. Patch coordinates are then determined from the margins (lines 10–16).

Margin computation on the subgrid region level is demonstrated in Algorithm GetBestColumnWindow, shown in Figure 7. Since the subgrid size is known, candidate intervals or windows within the region instead of margins are tested. The algorithm takes, as input, column sums of a given region. Intervals on the array of column sums are sampled (line 4) and a score is

FindRegions(*Pix*, *height*, *width*, *numsubx*, *numsuby*, *rowperiod*, *colperiod*)

INPUT: *Pix*: a two-dimensional array of pixels of an image
height: height of the image
width: width of the image
numsuby: number of subgrids along the y-axis
numsubx: number of subgrids along the x-axis
rowperiod: vertical distance between adjacent spots
colperiod: horizontal distance between adjacent spots
OUTPUT: Rectangle coordinates on *Pix* for each subgrid region

```
1   ▷ determine margins in the image
2   leftmargin ← GetLeftMargin( Pix, 1, width, 1, height, rowperiod, colperiod )
3   rightmargin ← GetRightMargin( Pix, 1, width, 1, height, rowperiod, colperiod )
4   topmargin ← GetTopMargin( Pix, 1, width, 1, height, rowperiod, colperiod )
5   bottommargin ← GetBottomMargin( Pix, 1, width, 1, height, rowperiod, colperiod )
6
7   ▷ compute regions, a two-dimensional array of rectangle coordinates
8   regionheight ← (bottommargin − topmargin)/numsuby
9   regionwidth ← (rightmargin − leftmargin)/numsubx
10  for subrow ← 1 to numsuby
11      do for subcol ← 1 to numsubx
12             do regions[subrow, subcol].left ← leftmargin + (subcol − 1) ∗ regionwidth
13                regions[subrow, subcol].width ← regionwidth
14                regions[subrow, subcol].top ← topmargin + (subrow − 1) ∗ regionheight
15                regions[subrow, subcol].height ← regionheight
16
17  return regions
```

Figure 4: Algorithm FindRegions

GetLeftMargin(*Pix*, *height*, *width*, *rowperiod*, *colperiod*)

INPUT: *Pix*: a two-dimensional array of pixels of an image
height: height of the image
width: width of the image
rowperiod: vertical distance between adjacent spots in the image
colperiod: horizontal distance between adjacent spots in the image
OUTPUT: The distance between the left edge of the image and
the edge of the subgrid(s) closest to the image edge.

```
1   i ← 1
2   foundedge ← false
3   while (not foundedge)
4         do margin ← i * colperiod   ▷ candidate margins are colperiod apart
5            ▷ compute row intensity sums along the area between
6               the image edge and the margin
7            rowsums ← GetRowSums( Pix, 1, height, 1, margin )
8            ▷ compute the FFT of the row intensity sums
9               and obtain the frequency with maximum coefficient
10           coeffs ← FFT(rowsums)
11           maxfreq ← frequency with maximum coefficient in coeffs
12           ▷ if the frequency matches the expected spot row frequency,
13              then the desired margin has been crossed
14           if (maxfreq = height/rowperiod)
15              foundedge ← true
16           i ← i + 1
17  margin ← margin − colperiod   ▷ move back margin by colperiod for slack
18  return margin
```

Figure 5: Algorithm GetLeftMargin

FindPatches(*Pix*, *region*, *rowperiod*, *colperiod*, *numspoty*, *numspotx*)

INPUT: *Pix*: a two-dimensional array of pixels of an image
region: rectangular coordinates of subgrid region
rowperiod: vertical distance between adjacent spots
colperiod: horizontal distance between adjacent spots
numspoty: number of spots for each subgrid column
numspotx: number of spots for each subgrid row
OUTPUT: The leftmost pixel position where the grid begins

```
1   subgridheight ← rowperiod * numspoty
2   subgridwidth ← colperiod * numspotx
3
4   ▷ compute left and top margins in subgrid region
5   colsums ← GetColumnSums( Pix, region.top, region.height, region.left, region.width)
6   leftmargin ← GetBestColumnWindow ( colsums, region.width, subgridwidth, colperiod )
7   rowsums ← GetRowSums( Pix, region.top, region.height, region.left, region.width)
8   topmargin ← GetBestRowWindow ( rowsums, region.height, subgridheight, rowperiod )
9
10  ▷ compute patches, a two-dimensional array of rectangle coordinates
11  for x ← 1 to numspotx
12      do for y ← 1 to numspoty
13             do patches[i, j].top ← region.top + topmargin + (j − 1) * rowperiod
14                patches[i, j].height ← rowperiod
15                patches[i, j].left ← region.left + leftmargin + (i − 1) * colperiod
16                patches[i, j].width ← colperiod
17
18  return patches
```

Figure 6: Algorithm FindPatches

GetBestColumnWindow(*colsums*, *width*, *subgridwidth*, *colperiod*)

INPUT: *colsums*: pixel itensity sums for each column
width: size of *colsums* array
subgridwidth: width of a subgrid
colperiod: horizontal distance between adjacent spots in the image
OUTPUT: The leftmost pixel position where the subgrid begins

```
1   bestscore ← undefined
2   bestposition ← undefined
3   for i ← 1 to (width − subgridwidth)
4       do window ← colsums[i . . . (i + subgridwidth − 1)]   ▷ extract candidate window
5          score ← getscore(window)
6          ▷ getscore returns either the sum of pixel sums or
7                 the FFT frequency coefficient in window
8          if (score > bestscore)
9            then bestscore ← score
10                bestposition ← i
11  return bestposition
```

Figure 7: Algorithm GetBestColumnWindow

obtained for each interval (line 5–6). The interval that yields the highest score is returned (lines 7–10).

The program uses two interval scoring methods. The first method involves adding all sums in the interval. Intuitively, the interval where the spots are located should contain the pixels with relatively higher intensities. The second method computes the FFT along the interval, since the interval that contains the spots ought to yield the highest coefficients for the expected spot frequency. Particularly since the first method would be susceptible to significant noise, the second method serves to validate the first method.

4 Implementation

The program is written in C, and has been compiled using `gcc version 3.2.2` running on Red Hat Linux 3.2.2-5. It uses the GNU/Linux `tiff` library package for TIFF image file processing and the `fftw3` library package [15] for FFT computation.

The program is invoked using the following format:

```
grweave tiff(s) -gx m -gy n -sx s -sy t -showgrid jpgname -output txtname
```

where `tiff(s)` specify the name(s) of the tiff image file(s) to be processed by the program. The arguments `m` and `n` describe the number of subgrids along the x-axis and y-axis, respectively, while `s` and `t` describe the number of spots in a row and column of a subgrid, respectively. An image file (named `jpgname`) may be produced that overlays the grid on the original image file(s), from which the user may visually verify whether the subgrids and the spots were correctly located. The intensity results are stored in the file named `txtname` in tab-delimited format, containing the following values, per spot, per image:

- Spot Index Information: subgrid row, subgrid column, spot row, and spot column

Image Quality	Number of Images	*Correct*	*Fair*	*Poor*
Clean	8	8 (100%)	0 (0%)	0 (0%)
Noisy	18	12 (67%)	4 (22%)	2 (11%)
Total	26	20 (77%)	4 (15%)	2 (8%)

Figure 8: Gridding Results.

- Spot Location Information: x and y coordinates of the upperleft corner of the patch, width and height of the patch, x and y coordinates of the spot center, and the diameter of the spot
- Foreground Intensity Values: mean, median, standard deviation, minimum, and maximum
- Background Intensity Values: mean, median, standard deviation, minimum, and maximum

When multiple images are indicated, grid computation is performed on an aggregate of the images (sums of pixel intensities are computed for each coordinate across the images). The program also includes options that the user may indicate at the end of the command line:

`-usefilter` When this option is included, high and low intensity values in the image(s) are filtered out before grid computation is carried out (note that the filtered image is used for gridding only, not for intensity computation). By default, gridding is carried out on the raw image(s).

`-adjustcenter` When this option is included, centers are computed on a per-spot basis based on the relative intensities present in a patch. By default, centers are assumed to be the patch centers.

`-fixedradius` By default, the program computes a radius for each spot, depending on the relative intensities present in the patch. This option assumes a fixed radius for all spots, determined by calculating the mean of all computed radii.

5 Discussion and Conclusion

The program was tested against sample microarray images. Thirteen pairs of images were selected and categorized as either clean (4 image pairs) or noisy (9 image pairs). The types of noise in the noisy images ranged from arbitrary streaks and smudges on different areas on the image to noise that appeared uniformly distributed along the image.

All of the test images had $32 = 8 \times 4$ subgrids and each subgrid contained $288 = 12 \times 24$ spots. We first executed the program on each of the 26 images, and then on the image *pairs* to determine if obtaining an aggregate of the pairs improved the gridding. The `-showgrid` option was used and the resulting image was visually inspected to verify whether the subgrids were located correctly.

Table 8 presents the results of the tests. The results fall into three categories: *correct* if all 32 subgrids were correctly located, *fair* if at least 50% of the subgrids were correctly located, and *poor* if less than 50% of the subgrids were correctly located. As the table shows, the program produced correct results for all of the clean images. For noisy images, only 2 out of the 18 images were gridded poorly. Overall, 77% of the test images, both clean and noisy, were gridded correctly. In addition, when the program was executed on *pairs* of images, some incremental improvements were observed for the images that were gridded poorly.

It was observed that whenever the program failed in gridding the images, the images contained significant noise and spot intensities were relatively low. These characteristics, coupled with the fact that the test images had subgrids that were very close to each other along the x-axis (approximately 18 pixels—roughly one spot period—between adjacent grids), reduce the chances of successful margin detection.

The program has also been tested on other datasets and the results are favorable particularly when there is enough vertical and horizontal distance between adjacent subgrids. We also note that for the cases where the subgrids are not correctly located, region coordinates are off always by a multiple of the spot period. This indicates that the spot coordinates of the actual spots detected by the program remain accurate, even in such cases.

References

[1] J. Angulo and J. Serra, *Automatic analysis of DNA microarray images using mathematical morphology*, Bioinformatics, 19 (2003), pp. 553–562.

[2] P. Bajcsy, *Gridline: Automatic grid alignment in dna microarray scans*, IEEE Trans. Image Process., 13 (2004), pp. 15–25.

[3] T. Bergemann, R. Laws, F. Quiaoit, and L. Zhao, *A statistically driven approach for image segmentation and signal extraction in cdna microarrays*, J. Comput. Biol., 11 (2004), pp. 695–713.

[4] N. Brandle, H. Bischof, and H. Lapp, *Robust dna microarray image analysis*, Mach. Vis. Appl., 15 (2003), pp. 11–28.

[5] J. Buhler, T. Ideker, and D. Haynor, *Dapple: Improved techniques for finding spots on DNA microarrays*, Tech. Rep. UWTR 2000-08-05, Department of Computer Science and Engineering, University of Washington, 2000.

[6] A. J. Carlisle, V. V. Prabhu, A. Elkahloun, J. Hudson, J. M. Trent, W. M. Linehan, E. D. Williams, M. Emmert-Buck, L. A. Liotta, P. J. Munson, and D. B. Krizman, *Development of a prostate cDNA microarray and statistical gene expression analysis package*, Molecular Carcinogenesis, 28 (2000), pp. 12–22.

[7] Y. Chen, E. R. Dougherty, and M. L. Bittner, *Ratio-based decisions and the quantitative analysis of cDNA microarray images*, Journal of Biomedical Ethics, 2 (1997), pp. 364–374.

[8] J. Cooley and J. Tukey, *An algorithm for the machine calculation of complex fourier series*, Mathematics of Computation, 19 (1965), pp. 297–301.

[9] R. Dell'Anna, F. Demichelis, M. Barbareschi, and A. Sboner, *An automated procedure to properly handle digital images in large scale tissue microarray experiments*, Comput. Meth. Programs Biomed., 79 (2005), pp. 197–208.

[10] O. Demirkaya, M. H. Asyali, and M. M. Shoukri, *Segmentation of cdna microarray spots using markov random field modeling*, Bioinformatics, 21 (2005), pp. 2994–3000.

[11] S. Dudoit, R. C. Gentleman, and J. Quackenbush, *Open source software for the analysis of microarray data*, BioTechniques, 34 (2003), pp. S45–S51.

[12] P. I. Echevarria, J. C. Punzalan, and J. P. C. Vergara, *Microarray image analysis program*, Loyola Schools Review, 2 (2003), pp. 1–14.

[13] M. Eisen, *Scanalyze users manual*. Available from `http://rana.lbl.gov/index.htm`, 1999.

[14] M. B. Eisen and P. O. Brown, *DNA arrays for analysis of gene expression*, Methods in Enzymology, 303 (1999), pp. 179–205.

[15] M. Frigo and S. G. Johnson, *FFTW: An adaptive software architecture for the FFT*, in Proceedings of the 1998 IEEE International Conference on Acoustics Speech and Signal Processing, vol. 3, IEEE, 1998, pp. 1381–1384.

[16] V. L. Galinsky, *Automatic registration of microarray images. i. rectangular grid*, Bioinformatics, 19 (2003), pp. 1824–1831.

[17] ——, *Automatic registration of microarray images. ii. hexagonal grid*, Bioinformatics, 19 (2003), pp. 1832–1836.

[18] L. S. Heath, N. Ramakrishnan, R. R. Sederoff, R. W. Whetten, B. I. Chevone, C. A. Struble, V. Y. Jouenne, D. Chen, L. M. Zyl, and R. Grene, *Studying the functional genomics of stress responses in loblolly pine using the Expresso microarray management system*, Comparative and Functional Genomics, 3 (2002), pp. 226–243.

[19] R. Hirata Jr, J. Barrera, R. F. Hashimoto, and D. O. Dantas, *Microarray gridding by mathematical morphology*, in Proceedings of SIGRAPI, International Symposium on Computer Graphics, Image Processing, and Vision, 2001, pp. 112–119.

[20] J. Ho, W. Hwang, H. Lu, and D. Lee, *Gridding spot centers of smoothly distorted microarray images*, IEEE Trans. Image Process., 15 (2006), pp. 342–353.

[21] A. N. Jain, T. A. Tokuyasu, A. M. Snijders, R. Segraves, D. G. Albertson, and D. Pinkel, *Fully automatic quantification of microarray image data*, Genome Research, 12 (2002), pp. 325–332.

[22] H.-Y. Jung and H.-G. Cho, *An automatic block and spot indexing with k-nearest neighbors graph for microarray image analysis*, Bioinformatics, 18 (2002), pp. S141–S151.

[23] M. Katzer, F. Kummert, and G. Sagerer, *Robust automatic microarray image analysis*, in Proceedings of the International Conference on Bioinformatics: North-South Networking, Bangkok, 2002, p. 12.

[24] M. Katzer, F. Kummert, and G. Sagerer, *Methods for automatic microarray image segmentation*, IEEE Trans. Nanobiosci., 2 (2003), pp. 202–214.

[25] N. Lawrence, M. Milo, M. Niranjan, P. Rashbass, and S. Soullier, *Reducing the variability in cdna microarray image processing by bayesian inference*, Bioinformatics, 20 (2004), pp. 518–526.

[26] A. Liew, H. Yan, and M. Yang, *Robust adaptive spot segmentation of dna microarray images*, Pattern Recognit., 36 (2003), pp. 1251–1254.

[27] L. Rueda and V. Vidyadharan, *A hill-climbing approach for automatic gridding of cdna microarray images*, IEEE-ACM Trans. Comput. Biol. Bioinform., 3 (2006), pp. 72–83.

[28] A. I. Saeed, V. Sharov, J. White, W. Liang, N. Bhagabati, J. Braisted, M. Klapa, T. Currier, M. Thiagarajan, A. Sturn, M. Stuffin, A. Rezantsev, D. Popov, A. Ryltsov, E. Kostukovich, I. Borisovsky, Z. Liu, A. Vinsavich, V. Trush, and J. Quackenbush, *TM4: A free, open-source system for microarray data management and analysis*, BioTechniques, 34 (2003), pp. 374–378.

[29] A. Sioson, J. I. Watkinson, C. Vasquez-Robinet, M. Ellis, M. Shukla, D. Kumar, N. Ramakrishnan, L. S. Heath, R. Grene, B. I. Chevone, K. Kafadar, and L. T. Watson, *Expresso and chips: Creating a next generation microarray experiment management system*, in Proceedings of the Next Generation Software Systems Workshop, 17th International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, 2003, p. 209.

[30] M. Steinfath, W. Wruck, H. Seidel, H. Lehrach, U. Radelof, and J. O'Brien, *Automated image analysis for array hybridization experiments*, Bioinformatics, 17 (2001), pp. 634–641.

[31] X. Wang, S. Ghosh, and S. Guo, *Quantitative quality control in microarray image processing and data acquisition*, Nucleic Acids Research, 19 (2003), pp. 553–562.