

GS01 0163

Analysis of Microarray Data

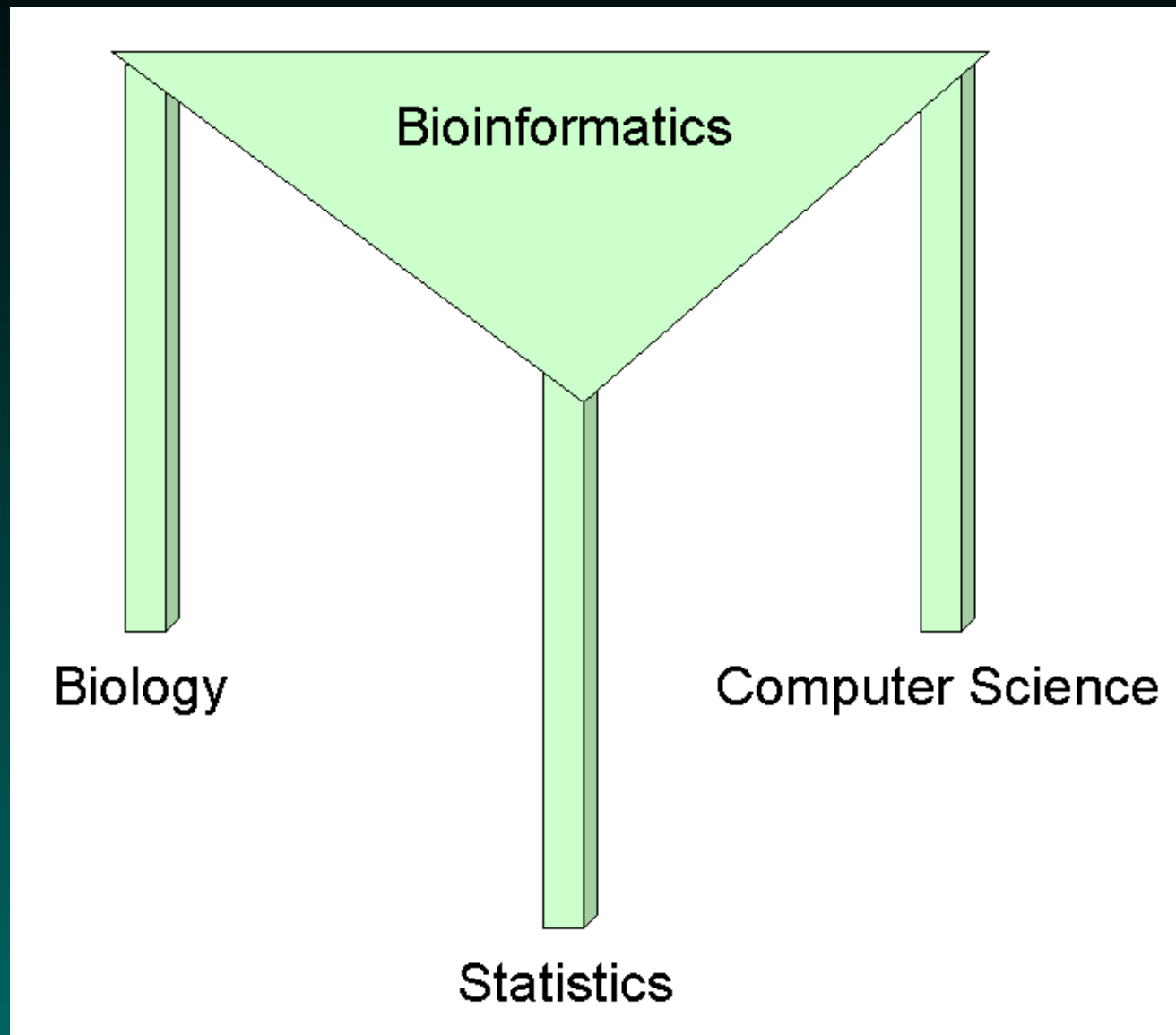
Keith Baggerly and Kevin Coombes
Section of Bioinformatics
Department of Biostatistics and Applied Mathematics
UT M. D. Anderson Cancer Center
kabagg@mdanderson.org
kcoombes@mdanderson.org

3 November 2005

Lecture 18: R and Glass Microarrays

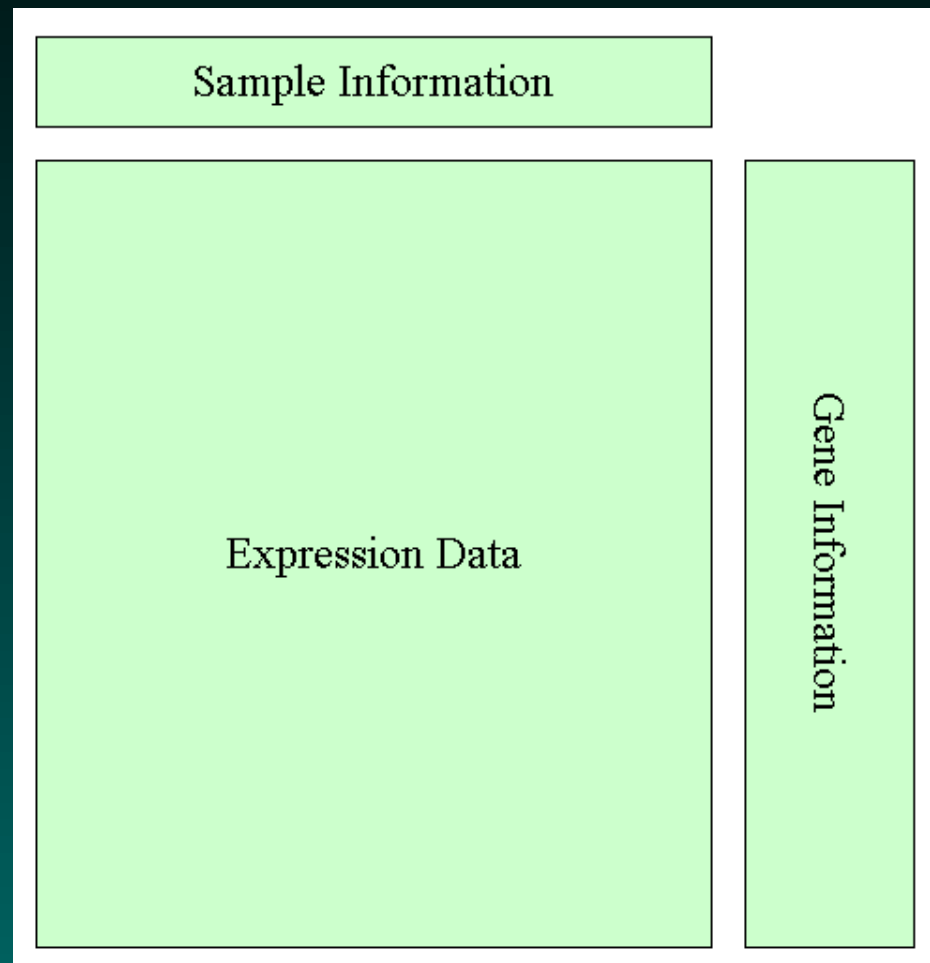
- Microarray Data Structures
 - `marray` data structures
 - `limma` data structures
- Toward a modular and efficient design
- Quantifying Glass Microarrays
- Getting down to business

The threefold way



Microarray Data Structures

Recall from Lecture 5 on R and Affymetrix arrays:



Recall: Affymetrix analysis in BioConductor

- `exprSets` combine expression data and sample information
 - Can be linked in an efficient way to gene information
- `AffyBatch` objects hold the raw data
 - Easy to construct from a directory of CEL files
 - Gene annotations updated automatically
 - Useful quality control tools
- Structured, modular preprocessing with `expresso`
 - Background correction
 - Normalization
 - PM correction
 - Summarization

Glass arrays in BioConductor

BioConductor includes two different package bundles to deal with two-color glass microarrays: `array` and `limma`.

Neither package uses the notion of an `exprSet`.

In both cases, the design seems to be less flexible and less modular than the tools for working with Affymetrix arrays.

marray data structures

The `marray` package uses four basic classes to hold the data from a collection of microarray experiments.

marrayInfo : holds sample information or gene information

marrayLayout : describes the geometry of the array

marrayRaw : holds the raw array data

marrayNorm : holds array data after normalization

The primary processing function is `maNorm`, which allows you to try a limited number of normalization methods.

Sample or Gene Information

In `marray`, the same kind of object (`marrayInfo`) is used to hold either sample information or gene information. This object is a data frame with extra information attached (like the `phenoData` objects in an `exprSet`). The extra information includes longer descriptive labels for the columns and a character string with any notes you'd like to attach to the object.

When used to describe genes, the rows correspond to spots on the array and columns to gene annotations.

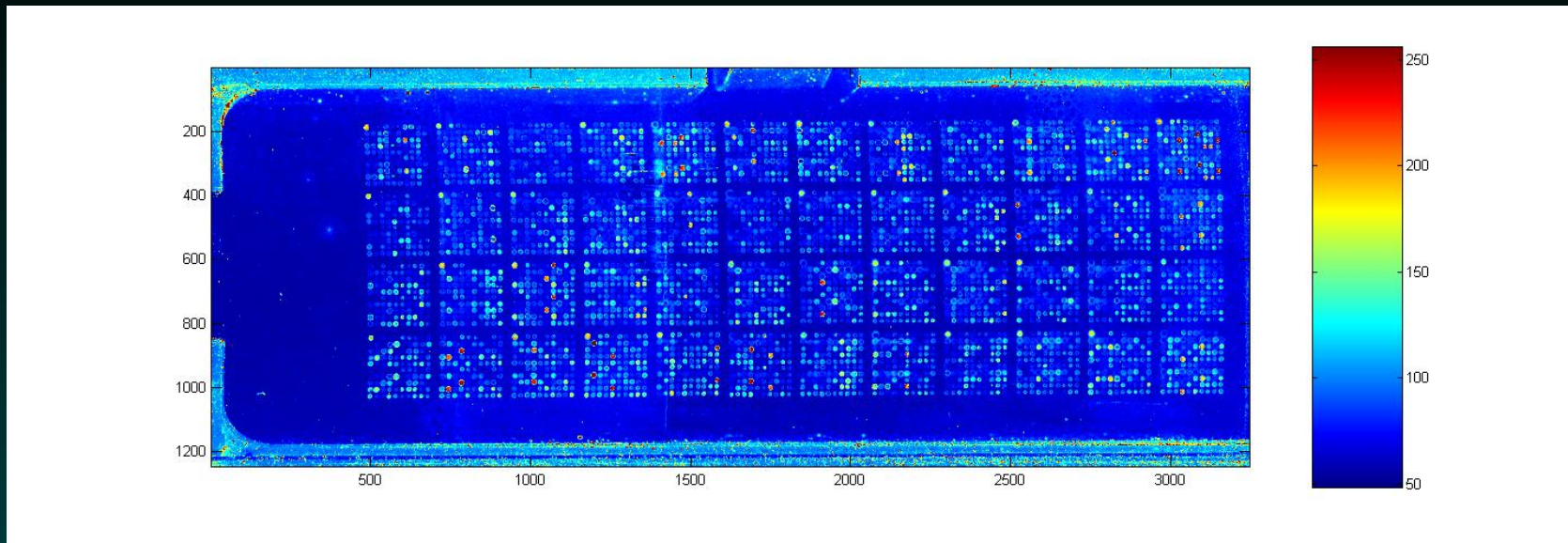
When used to describe samples, the rows correspond to microarrays and columns give information about the samples. In particular, the columns should identify the samples used in both the Cy3 and Cy5 channels.

Things I don't like about `marrayInfo`

- No gene-specific or sample-specific tools. Can only tell how to interpret the object in context.
- Forced combining of Cy3 and Cy5 sample information on the same row of the sample information

Although this is peeking ahead, it's also worth noting that every experimental data set (`marrayRaw` or `marrayNorm`) must contain its own copy of the gene-information `marrayInfo` object. This is a **terrible** design decision. It wastes space (on disk or in memory) and is impossible to maintain. If the annotations must be updated, you have to hunt down innumerable copies and update all of them.

Geometry of glass microarray designs



As we have seen previously, glass microarrays are typically laid out in a hierarchical layout, containing a rectangle of grids, each of which is a rectangle of spots. Also, each grid is spotted on the array by a different physical pin.

marrayLayout slots

The `marray` package uses an `marrayLayout` object to describe the geometry using five numbers:

maNgr : number of grid rows

maNgc : number of grid columns

maNsr : number of spot rows

maNsc : number of spot columns

maNspots : number of spots

It is perhaps odd that they store the number of spots, since it seems to me that it should always be easily computable in terms of the other four parameters.

marrayLayout slots

The `marrayLayout` object may also include three additional vectors

maSub : a logical vector: are we currently interested in this spot?

maPlate : which plate did the robot get this spot from?

maControls : what kind of material is spotted here?

Metaphors appear to be mixed here: the `maPlate` and `maControls` vectors belong to the array design, and not to the specific analysis. The `maSub` object, however, seems to be an analysis-specific filter to let you focus on specific genes.

mararrayLayout methods

They include methods to compute the following quantities, but they do not store them in the object:

maPrintTip : vector of print tips for the spots

maGridCol : vector of grid column locations

maGridRow : vector of grid row locations

maSpotCol : vector of spot column locations

maSpotRow : vector of spot row locations

More complaints

The design of `marrayLayout` is a mess.

Every `marrayRaw` and `marrayNorm` gets its own copy. This design has serious maintenance problems. Because they realize this mistake, they use methods to compute the vector locations. (Their explanation: storing them takes too much space.) A drawback of computing them, however, is that this assumes that the order of the data rows is always the same; however, different quantification packages do not produce the same row order when they quantify the spots.

mararrayRaw slots

Raw expression data from glass microarrays is stored as an `mararrayRaw` object, which contains:

- Four matrices of raw data (`maRf`, `maGf`, `maRb`, `maGb`) with red (R) and green (G) foreground (f) and background (b) estimates.
- An optional matrix (`maW`) of spot quality weights.
- `maLayout`, containing the array layout
- `maGnames`, containing the gene information
- `maTargets`, containing the sample information

As pointed out earlier, including copies of the layout and gene information is inefficient and hard to maintain.

mararrayRaw methods

maA : vector of log intensities

maM : vector of log ratios

maLR : vector of background-corrected red log intensities

maLG : vector of background-corrected red log intensities

Note that there is no option to perform any form of background correction other than simply subtracting the values supplied by the image quantification software.

marrayNorm slots

Processed expression data from glass microarrays is stored as an `marrayNorm` object. These contain copies of the `maW`, `maLayout`, `maGnames`, and `maTargets` objects from the raw source data. In place of the raw measurements, these objects contain

maA : matrix of average log intensities

maM : matrix of log ratios

maMloc : localization normalization values

maMscale : scale normalization values

Getting from `marrayRaw` to `marrayNorm`

Once we have an object in hand containing raw microarray measurements, we can simply `coerce` them into normalized values. This will do no pre-processing, simply computing the M and A values from the raw data.

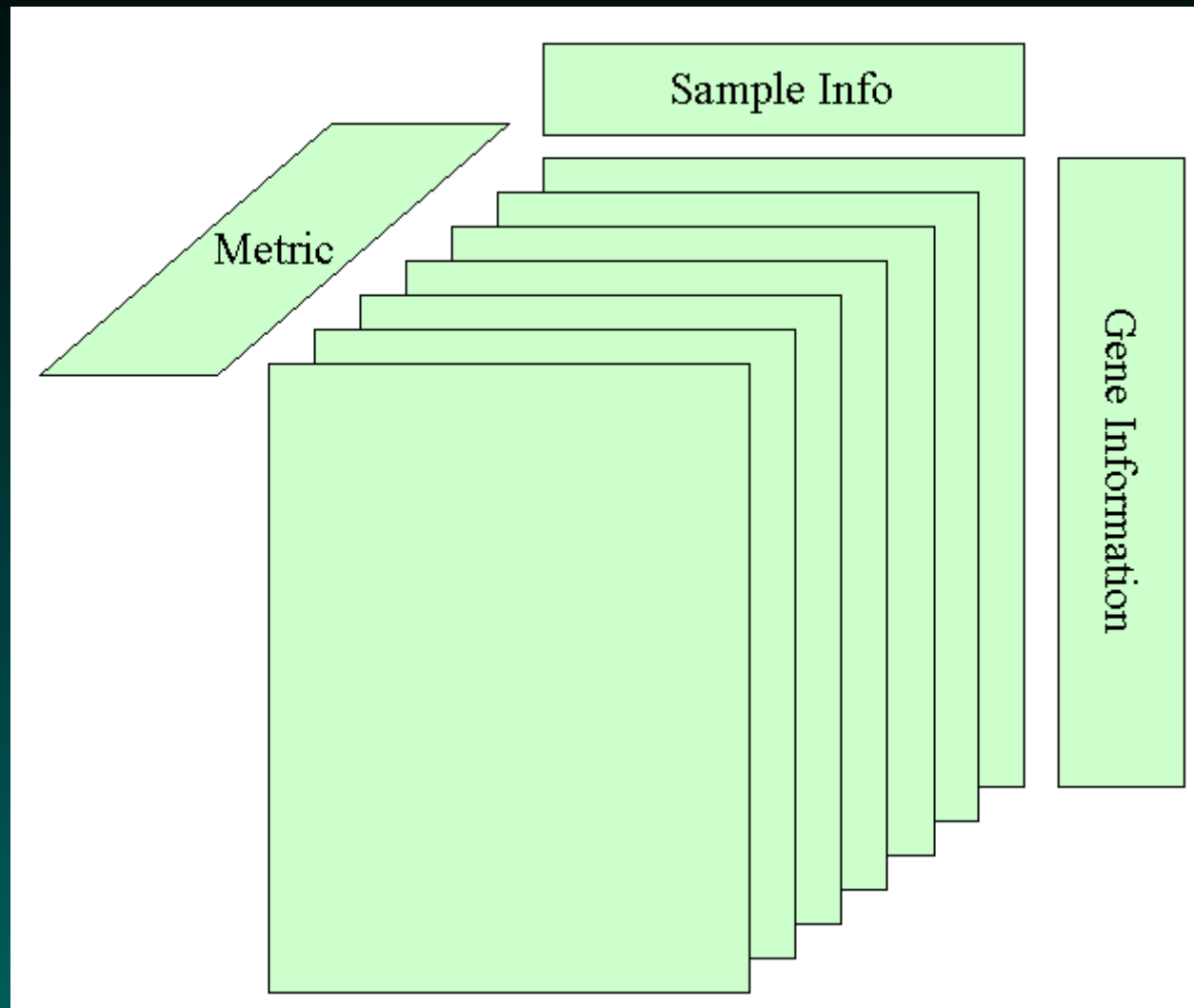
Normalization methods

In most cases, we want to normalize the data using `maNorm` (which is a wrapper around the more general function `maNormMain`). The basic function call looks like

```
> maNorm(my.raw.data, norm=method)
```

The normalization method must be specified as a character string, which must be one of the following: “none”, “median”, “loess”, “twoD”, “printTipLoess”, or “scalePrintTipMAD”. Unlike the approach taken with the Affymetrix arrays, there is no variable containing a list of normalization methods and no obvious way to add new methods. The more general method is extensible, but the way to extend it is poorly documented.

The marray data cube



Fixed, hard-coded set of metrics (Rf, Gf, Rb, Gb, W).

limma data structures

The `limma` package in BioConductor provides a different set of tools for glass microarrays.

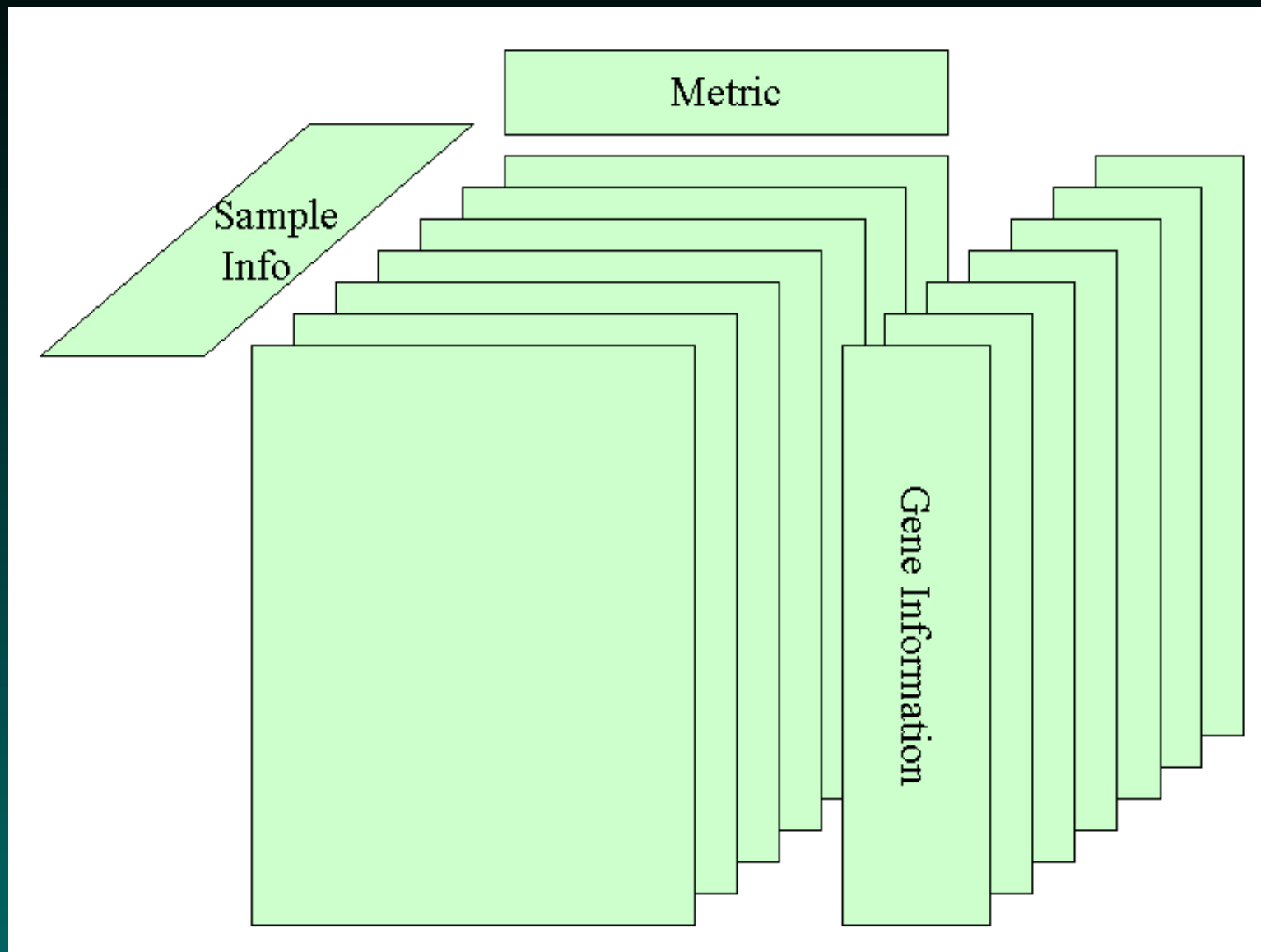
RGList : raw microarray data as a list of arrays containing

- Four matrices, `R`, `G`, `Rb`, `Gb`, containing measurements.
- Optional components `weights`, `printer`, `genes`, `targets`.

MAList : processed microarray data as a similar list with `M` and `A` components

Note that this is even more wasteful of space by making innumerable copies of the gene information....

The `limma` data cube



limma normalization methods

The `limma` package has its own normalization routines (since they use different data structures than `array`). Each has hard-coded option lists that are too painful to enumerate (or contemplate).

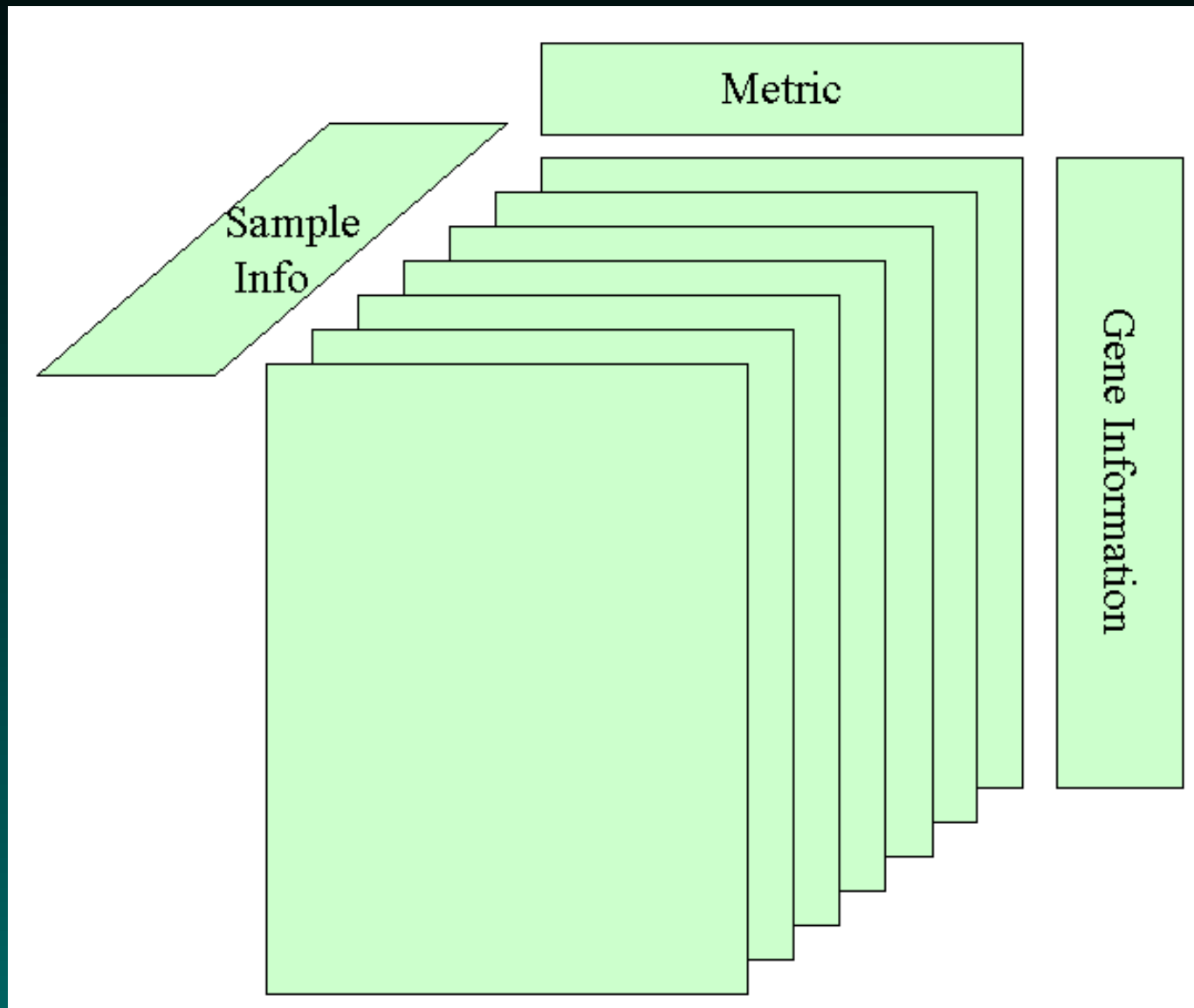
- `normalizeBetweenArrays`
- `normalizeWithinArrays`
- `normalizeForPrintOrder`
- `normalizeRobustSpline`
- `normalizeMedians`
- `normalizeQuantiles`

Toward a modular and efficient design

In case you hadn't noticed, I'm considerably less happy with the BioConductor analysis of glass microarrays than with their analysis of Affymetrix arrays. To review my main complaints:

- The data structures waste space
- The `marray` structures make it hard to combine array sets.
- It's not easy to plug in new processing algorithms (normalization or otherwise) to compare and contrast them.
- The designs do not use the `exprSet` structure, so it is hard to write high-level analysis tools that work on both kinds of arrays.

An easily extended data cube



A few design principles

- Array design should be stored in exactly one place.
 - Annotations can be updated easily.
 - No wasted space storing duplicate copies.
- Must be possible to read data from different quantification software and different array designs.
- Processing must be modular.
 - Easy to figure out what methods are available.
 - Easy to add new methods.
- After processing, should get an `exprSet`.

How should the two channels be handled?

Two possibilities

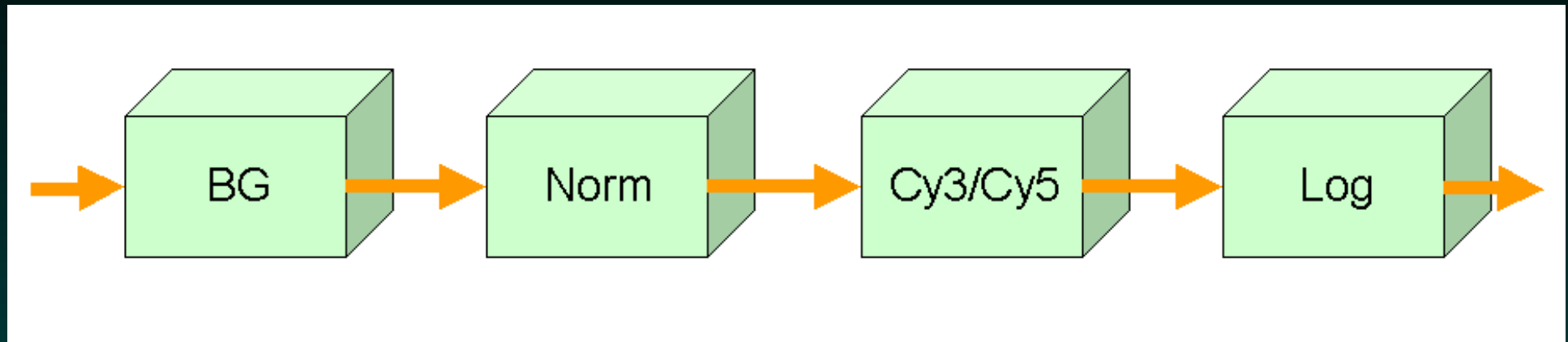
1. Each “sheet” is a slide

Slide	Cy3 Name	Cy5 Name	Cy3 Status	Cy5 Status
A1	RefMix	T1	Reference	Cancer
A2	N1	RefMix	Healthy	Reference

2. Each “sheet” is a separate channel

Slide	Channel	Sample Name	Status
A1	Cy3	RefMix	Reference
A1	Cy5	T1	Cancer
A2	Cy3	N1	Healthy
A2	Cy5	RefMix	Reference

The processing pipeline



It should be possible to plug different algorithms in for each step in the pipeline.

It should be possible to add additional steps.

Ideally, it should be possible from the final object to reconstruct the processing history (which will be needed for the methods section of an article based on the analysis!).

Quantifying Glass Microarrays

So far, I have avoided describing how glass array data gets from the image quantification files into R and/or BioConductor.

The problem: There are lots of different software packages for image quantification. Unlike the Affymetrix world (where everything starts with the DAT and CEL files), this implies that there are lots of different formats that need to be understood by a general microarray analysis package.

In particular, when you construct an object to hold microarray data, you not only need to know the array design (i.e., the geometry and the gene annotations for each spot), but you need to know what software quantified the images.

Microarray Quantification Packages

There are a variety of programs available for quantifying arrays, including

- Free:
 - UCSF Spot
 - TIGR SpotFinder
- Commercial:
 - ArrayVision (Imaging Research, Inc.)
 - ImaGene (BioDiscovery, Inc.)
 - MicroVigene (Vigene Tech, Inc.)

Microarray Quantification Packages

Most manufacturers (e.g., Agilent or the Axon GenePix) of microarray scanners also supply quantification software.

- The critical issue to note is that every quantification package uses its own:
 - methods for finding, segmenting, and quantifying spots
 - scheme for labeling the spots
 - order for reporting the spots
 - names for the measurements it reports.

The only thing they have in common is that they are all able to export the data in tab-separated-values format, with rows representing spots and columns representing measurements (like location, foreground intensity, background intensity, etc.).

Quantifying Glass Microarrays

In this lecture, we are going to assume that we have somehow managed to get our hands on a set of quantification files from a batch of glass microarrays, and that we have determined what the individual columns mean. Our next goal is to figure out how to get this data into R and BioConductor so we can start doing something useful with it.

Reading data into marray

In `marray`, they handle this problem by using a variety of “read” functions:

- `read.GenePix`
- `read.Spot`
- `read.SMD`
- `read.marrayRaw`

Reading data into `limma`

In `limma`, there is a single “read” function

```
> read.maimages(files, source=SOMETHING)
```

This function uses hard-coded text strings to support different quantification packages; source can be one of

<code>agilent</code>	<code>arrayvision</code>	<code>genepix</code>
<code>imagene</code>	<code>quantarray</code>	<code>smd</code>
<code>spot</code>		

Better data input?

Neither `marray` nor `limma` makes it easy to add new quantification packages. With `marray`, you presumably write another function of the form `read.my.quant`s, duplicating much of the existing code to coerce the input data into the desired format. In `limma`, you can't change the hard-coded strings, but you can take advantage of the many optional arguments of `read.maimages` to construct a custom data reader.

Better data input?

Conceptually, the problem has a simple form. Quantification data typically arrives as text files in tab-separated values format. Different manufacturers have different names for the columns that we care about. All we need to know is

- How to map the manufacturer's names to our standard names
- How many header lines to skip
- Whether the file contains one or two channels

If we had a description of the quantifier, we could use a single extendible function like

```
> my.stuff <- read.arrays(files, quantifier)
```

Notes on our own methods

After teaching this course for the first time last year, I implemented the “pipeline” processing idea. Code for this is contained in the **PreProcess** package that is available on our web site at

<http://bioinformatics.mdanderson.org/Software/OOMPA>

I’m still in the middle of implementing a generic microarray quantification reader...

Getting down to business

An overview of the process:

1. Create an object that knows how to map spot label identifiers to gene information.
2. Create an object that understands the geometry of the array.
3. Create an object that records the sample information.
4. Load the raw data from all the arrays.
5. Process (background correct, normalize, summarize) the raw data.
6. Get to the fun part of the analysis. . . .

A sample GenePix GAL file

The Axon GenePix scanner software creates “.gal” files that describe the geometry of a glass microarray, along with the information that describes the gene probes at each spot.

```

emacs@BSFC2-COOMBES
File Edit Options Buffers Tools Help
ATF      1
20      5
Type=GenePix ArrayList v1.0
BlockCount=16
BlockType=0
URL=http://genome-www.stanford.edu/cgi-bin/dbrun/sacchDB?find=
+Locus+%22[ID]%22
"Block1= 500, 500, 100, 24, 180, 21, 180"
"Block2= 4996, 500, 100, 24, 180, 21, 180"
"Block3= 9492, 500, 100, 24, 180, 21, 180"
"Block4= 13988, 500, 100, 24, 180, 21, 180"
"Block5= 500, 4996, 100, 24, 180, 21, 180"
"Block6= 4996, 4996, 100, 24, 180, 21, 180"
"Block7= 9492, 4996, 100, 24, 180, 21, 180"
"Block8= 13988, 4996, 100, 24, 180, 21, 180"
"Block9= 500, 9492, 100, 24, 180, 21, 180"
"Block10= 4996, 9492, 100, 24, 180, 21, 180"
"Block11= 9492, 9492, 100, 24, 180, 21, 180"
"Block12= 13988, 9492, 100, 24, 180, 21, 180"
"Block13= 500, 13988, 100, 24, 180, 21, 180"
"Block14= 4996, 13988, 100, 24, 180, 21, 180"
"Block15= 9492, 13988, 100, 24, 180, 21, 180"
"Block16= 13988, 13988, 100, 24, 180, 21, 180"
Block  Column  Row    Name    ID
1      1         1      GENOMIC 1X
1      2         1      3XSSC
1      3         1      GENOMIC 0.5X
1      4         1      3XSSC
1      5         1      GENOMIC 0.25X
1      6         1      3XSSC
1      7         1      EMPTY
--\** Demo.gal Thu Nov 3 10:29AM (Text Fill)--L23--Top-

```

The GenePix GAL file format

Axon describes the GAL file format on their web site:

http://www.moleculardevices.com/pages/software/gn_genepix_file_formats.html#gal

This is a special case of “Axon text format”. The first line of the file (ATF 1) is required, and identifies the file format. The second line (20 5) is also required. It tells us, in this case, that there are 20 additional header lines before the main data starts, and that there are 5 columns of data. The third line (Type=GenePix ArrayList V1.0) is also required and identifies the type of GAL file format. Since they have only ever defined one version of the file format, this should be the same in all GAL files.

Block-heads

The next set of header lines is optional. In this case, they have chosen to tell us (`BlockCount=16`) that there are 16 blocks (or subgrids) contained on the array. The next line (`BlockType=0`) encodes the fact that these are rectangular blocks. The `URL=...` line gives an optional web site for more information.

Note that, even though the blocks=subgrids are themselves laid out in a rectangular pattern, the format at this point does not tell us what that pattern is. Axon numbers the blocks starting with number 1 in the upper left corner, marching across one row at a time before moving down.

Block descriptions

Next, each block is described by a line of the form

```
"Block1= 500, 500, 100, 24, 180, 21, 180"
```

Each line contains 7 comma separated values describing the block. The first two entries give the **X, Y** position (in microns) of the top left corner of the block. The third value is the **diameter** of each spot in microns. The fourth value is the **number of rows**, and the fifth value is the **spacing** between spots in each row. The final two numbers are the **number of columns** and the **spacing** between spots in a column. Note that the geometry of the blocks can be inferred from the set of their X, Y positions.

Finally, the file contains a tab-separated set of information describing the spot locations and corresponding probe information.

Reading GAL files

The `mararray` package includes a function that knows how to read GAL files, called, cleverly enough, `read.Galfile`. The simplest use is:

```
> demo.gal <- read.Galfile('demo.gal',  
>   path='c://arrays/designs')
```

Warning: the following obvious attempt to read a GAL file somewhere other than the current directory will NOT work:

```
read.Galfile('c://arrays/designs/demo.gal')
```

Here the problem is that `read.Galfile` uses `path='.'` as the default value and always prepends the path to the file name.

Reading GAL files

After correctly reading the GAL file, the resulting object is a list:

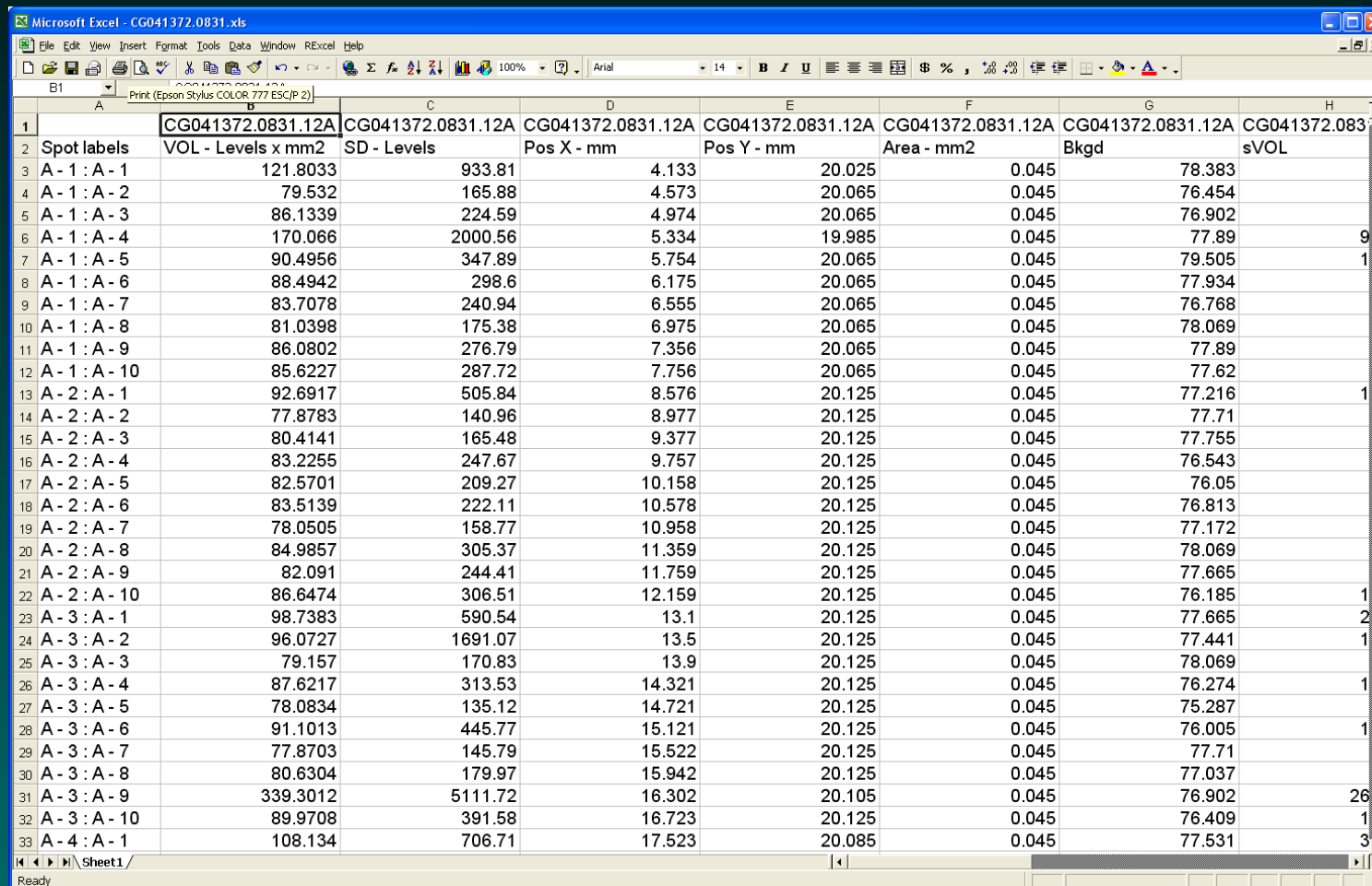
```
> class(demo.gal)
[1] "list"
> attributes(demo.gal)
$names
[1] "gnames"      "layout"      "neworder"
> class(demo.gal$gnames)
[1] "marrayInfo"
> class(demo.gal$layout)
[1] "marrayLayout"
> class(demo.gal$neworder)
[1] "integer"
```

Reading GAL files

Since the GAL file contains the gene annotations (which have now been put into an `marrayInfo` object) and the geometry (put into an `marrayLayout` object), the function is able to extract both pieces of information. Thus, when working with array quantifications from Axon, you can accomplish the first two steps in a single function.

Other formats for gene information

ArrayVision produces quantification files that include spot identifiers with absolutely no knowledge of the gene information:



	A	B	C	D	E	F	G	H
		CG041372.0831.12A	CG041372.0831.12A	CG041372.0831.12A	CG041372.0831.12A	CG041372.0831.12A	CG041372.0831.12A	CG041372.0831.12A
	Spot labels	VOL - Levels x mm2	SD - Levels	Pos X - mm	Pos Y - mm	Area - mm2	Bkgd	sVOL
3	A - 1 : A - 1	121.8033	933.81	4.133	20.025	0.045	78.383	
4	A - 1 : A - 2	79.532	165.88	4.573	20.065	0.045	76.454	
5	A - 1 : A - 3	86.1339	224.59	4.974	20.065	0.045	76.902	
6	A - 1 : A - 4	170.066	2000.56	5.334	19.985	0.045	77.89	9
7	A - 1 : A - 5	90.4956	347.89	5.754	20.065	0.045	79.505	1
8	A - 1 : A - 6	88.4942	298.6	6.175	20.065	0.045	77.934	
9	A - 1 : A - 7	83.7078	240.94	6.555	20.065	0.045	76.768	
10	A - 1 : A - 8	81.0398	175.38	6.975	20.065	0.045	78.069	
11	A - 1 : A - 9	86.0802	276.79	7.356	20.065	0.045	77.89	
12	A - 1 : A - 10	85.6227	287.72	7.756	20.065	0.045	77.62	
13	A - 2 : A - 1	92.6917	505.84	8.576	20.125	0.045	77.216	1
14	A - 2 : A - 2	77.8783	140.96	8.977	20.125	0.045	77.71	
15	A - 2 : A - 3	80.4141	165.48	9.377	20.125	0.045	77.755	
16	A - 2 : A - 4	83.2255	247.67	9.757	20.125	0.045	76.543	
17	A - 2 : A - 5	82.5701	209.27	10.158	20.125	0.045	76.05	
18	A - 2 : A - 6	83.5139	222.11	10.578	20.125	0.045	76.813	
19	A - 2 : A - 7	78.0505	158.77	10.958	20.125	0.045	77.172	
20	A - 2 : A - 8	84.9857	305.37	11.359	20.125	0.045	78.069	
21	A - 2 : A - 9	82.091	244.41	11.759	20.125	0.045	77.665	
22	A - 2 : A - 10	86.6474	306.51	12.159	20.125	0.045	76.185	1
23	A - 3 : A - 1	98.7383	590.54	13.1	20.125	0.045	77.665	2
24	A - 3 : A - 2	96.0727	1691.07	13.5	20.125	0.045	77.441	1
25	A - 3 : A - 3	79.157	170.83	13.9	20.125	0.045	78.069	
26	A - 3 : A - 4	87.6217	313.53	14.321	20.125	0.045	76.274	1
27	A - 3 : A - 5	78.0834	135.12	14.721	20.125	0.045	75.287	
28	A - 3 : A - 6	91.1013	445.77	15.121	20.125	0.045	76.005	1
29	A - 3 : A - 7	77.8703	145.79	15.522	20.125	0.045	77.71	
30	A - 3 : A - 8	80.6304	179.97	15.942	20.125	0.045	77.037	
31	A - 3 : A - 9	339.3012	5111.72	16.302	20.105	0.045	76.902	26
32	A - 3 : A - 10	89.9708	391.58	16.723	20.125	0.045	76.409	1
33	A - 4 : A - 1	108.134	706.71	17.523	20.085	0.045	77.531	3

Other formats for gene information

In this case, the core lab that produced the data also supplied a separate file with the gene annotations:

	A	B	C	D	E	F	G	H	I
	Location	IMAGE	Accession	Description	UniGene	Gene Symbol	Plate	PlateRow	PlateColumn
1	A1a1	753234	AC002404	AC002404 Human Chromosome X PAC RPC11-290C9 from the Pieter de		Data not found	1	A	1
2	A1a2	771220	BC011603	Unknown (protein for MGC:2272) [Homo sapiens], mRNA seq		RELA	1	E	1
3	A1a3	249856	NM_002895	retinoblastoma-like 1 (p107) [Homo sapiens], mRNA seq		RBL1	1	I	1
4	A1a4	149013	NM_001634	S-adenosylmethionine decarboxylase 1 precursor [Homo sapiens]		AMD1	1	M	1
5	A1a5	345430	NM_006218	phosphoinositide-3-kinase, catalytic, alpha polypeptide; r		PIK3CA	1	A	13
6	A1a6	42558	AK098055	Homo sapiens cDNA FLJ40736 fis, clone TKIDN20035		GATM	1	E	13
7	A1a7	146123	NM_002844	protein tyrosine phosphatase, receptor type, K precursor		PTPRK	1	I	13
8	A1a8	26314	NM_007269	syntaxin binding protein 3; syntaxin 4 binding protein [Homo sapiens]		STXBP3	1	M	13
9	A1a9	70002	AC135348	Homo sapiens chromosome 15, clone RP13-822L18, complete sequence		Data not found	2	A	1
10	A1a10	753420	D87466	Similar to S.cerevisiae hypothetical protein L3111 (S593 Hs.240112)		KIAA0276	2	E	1
11	A1a11	813460	AK097207	Homo sapiens cDNA FLJ39888 fis, clone SPLEN20165		PCSK7	2	I	1
12	A1a12	755975	NM_003878	Homo sapiens gamma-glutamyl hydrolase (conjugase, fc)		GGH	2	M	1
13	A1a13	812105	NM_006818	AF1Q protein; transmembrane protein [Homo sapiens], r		AF1Q	2	A	13
14	A1a14	46284	NM_023037	hypothetical protein CG003 [Homo sapiens], mRNA seq		13CDNA73	2	E	13
15	A2a1	755821	NM_003204	transcription factor 11 (basic leucine zipper type) [Homo sapiens]		NFE2L1	1	A	2
16	A2a2	823864	NM_000355	transcobalamin II precursor [Homo sapiens], mRNA seq		TCN2	1	E	2
17	A2a3	51916	AK057634	Homo sapiens cDNA FLJ33072 fis, clone TRACH20002		PLCB4	1	I	2
18	A2a4	167032	NM_002838	protein tyrosine phosphatase, receptor type, C, isoform 1		PTPRC	1	M	2
19	A2a5	188232	AK026253	Homo sapiens cDNA: FLJ22600 fis, clone HSI04447, hi		KLF4	1	A	14
20	A2a6	767638	NM_002655	pleiomorphic adenoma gene 1; Pleomorphic adenoma g		PLAG1	1	E	14
21	A2a7	42739	NM_003463	protein tyrosine phosphatase type IVA, member 1; Prote		PTP4A1	1	I	14
22	A2a8	810124	NM_002573	platelet-activating factor acetylhydrolase, isoform Ib, gar		PAFAH1B3	1	M	14
23	A2a9	811029	AK025508	Homo sapiens cDNA: FLJ21855 fis, clone HEP02277, n		SFRS14	2	A	2
24	A2a10	119914	AP005431	Homo sapiens genomic DNA, chromosome 18 clone:RP11-193E15, comp		Data not found	2	E	2
25	A2a11	292806	NM_001316	CSE1 chromosome segregation 1-like (yeast); cellular at		CSE1L	2	I	2
26	A2a12	33182	U79289	Human clone 23695 mRNA sequence		Hs.90798	2	M	2
27	A2a13	34852	AF207599	Homo sapiens pRb-interacting protein RbBP-36 mRNA, Hs.		BIRC2	2	A	14
28	A2a14	156045	NM_005698	secretory carrier membrane protein 3 isoform 1; propin		SCAMP3	2	E	14
29	A3a1	44477	NM_001078	vascular cell adhesion molecule 1, isoform a precursor; (VCAM1	1	A	3
30	A3a2	320763	NM_002970	spermidinespermine N1-acetyltransferase [Homo sapiens]		SAT	1	E	3
31	A3a3	66731	NM_000325	paired-like homeodomain transcription factor 2 isoform c		PITX2	1	I	3
32	A3a4	753184	NM_012465	tolloid-like 2 [Homo sapiens], mRNA sequence		TLL2	1	M	3

Other formats for gene information

This example illustrates a more typical situation.

1. Neither of these text files explicitly describes the geometry of the array.
2. Neither file includes separate columns to identify the 'grid and subgrid row and column positions; these are embedded in the spot labels or locations.
3. The data file uses "Spot labels" of the form $A - 1 : A - 1$, while the annotations file describes the same "Location" in the form A1a1.

Reading the gene information

When we have a simple tab-separated file (like this one) containing the gene information, we can use it to produce a `marrayInfo` object.

```
> location <- 'C://arrays/designs'  
> filename <- 'CG4.2.Version2.GeneList.txt'  
> cg42 <- read.marrayInfo(file.path(location,  
    filename), info.id=1:9, labels=6)
```

The `info.id` argument is optional; it is a list of the indices of the columns of the gene info file to include. The `labels` argument is also optional; it is the index of the column to use for labeling the gene. In our example, column 6 contains the gene symbols.

Checking the results

```
> cg42
```

```
An object of class "marrayInfo"
```

```
@maLabels
```

```
[1] " "          "RELA"      "RBL1"      "AMD1"      "PIK3CA"
10075 more elements ...
```

```
@maInfo
```

	Location	IMAGE	Accession
1	A1a1	753234	AC002404
2	A1a2	771220	BC011603
3	A1a3	249856	NM_002895
4	A1a4	149013	NM_001634
5	A1a5	345430	NM_006218

	UniGene	Gene	Symbol	Plate	PlateRow	PlateColumn
1				1	A	1
2	Hs.432975		RELA	1	E	1
3	Hs.87		RBL1	1	I	1
4	Hs.262476		AMD1	1	M	1
5	Hs.85701		PIK3CA	1	A	13
10075 more rows ...						

@maNotes

[1] "C://arrays/designs/CG4.2.Version2.GeneList.txt"

Step 2: Getting the layout

Of course, we're still not done; we have to create an `marrayLayout` object with the geometry.

```
> temp <- as.character(cg42@maInfo$Location)
> temp <- temp[length(temp)]
> temp
[1] "D12o14"
> ngr <- which(LETTERS == substring(temp, 1, 1))
> ngc <- as.numeric(substring(temp, 2, 3))
> nsr <- which(letters == substring(temp, 4, 4))
> nsc <- as.numeric(substring(temp, 5, 6))
> cg42Layout <- new('marrayLayout',
+                   maNgr=ngr, maNgc=ngc,
+                   maNsr=nsr, maNsc=nsc,
+                   maPlate=factor(cg42@maInfo$Plate
```

Checking the layout

```
> summary(cg42Layout)
```

```
Array layout:  Object of class marrayLayout.
```

```
Total number of spots: 10080
```

```
Dimensions of grid matrix: 4 rows by 12 cols
```

```
Dimensions of spot matrices: 15 rows by 14 cols
```

```
Currently working with a subset of 10080spots.
```

```
Control spots:
```

```
Notes on layout:
```

How good are the gene annotations?

It is an unfortunate fact of life that the gene annotations for glass microarrays are rarely as good as the annotations for Affymetrix microarrays. The main difficulty is that we are dealing with many different manufacturers and software producers, so there is no central repository that has a vested interest in keeping the annotations up to date.

GAL files, for example, can contain varying degrees of information, varying highly in both the level of detail and the quality and accuracy of the annotations.

How good are the gene annotations?

As a general rule, you should try to get annotations that are as close as possible to describing the actual genetic material placed on the array. In particular, **gene names, gene symbols, or UniGene cluster IDs are NOT primary identifiers of genomic material**. You want something like:

- an IMAGE clone ID,
- a GenBank sequence identifier,
- or (in the case of long oligo arrays) the actual sequence spotted on the array.

Step 3: Getting the sample information

BioConductor needs a benevolent dictator.

You might think that you already know how to read sample information into BioConductor. After all, that's what the `phenoData` class does, and that's how we handle this issue with `exprSet` or `AffyBatch` objects. But you'd be wrong.

There's ALWAYS more than one way to do it.

The `marray` way is to re-use the `marrayInfo` class, and so you can also read a sample information file in using the `read.marrayInfo` function that we described just a few slides ago for gene information. Of course, by this point, you'd probably be so tired of the whole subject that you'd be thankful that this lecture (and class-week) are over....