GS01 0163
Analysis of Microarray Data

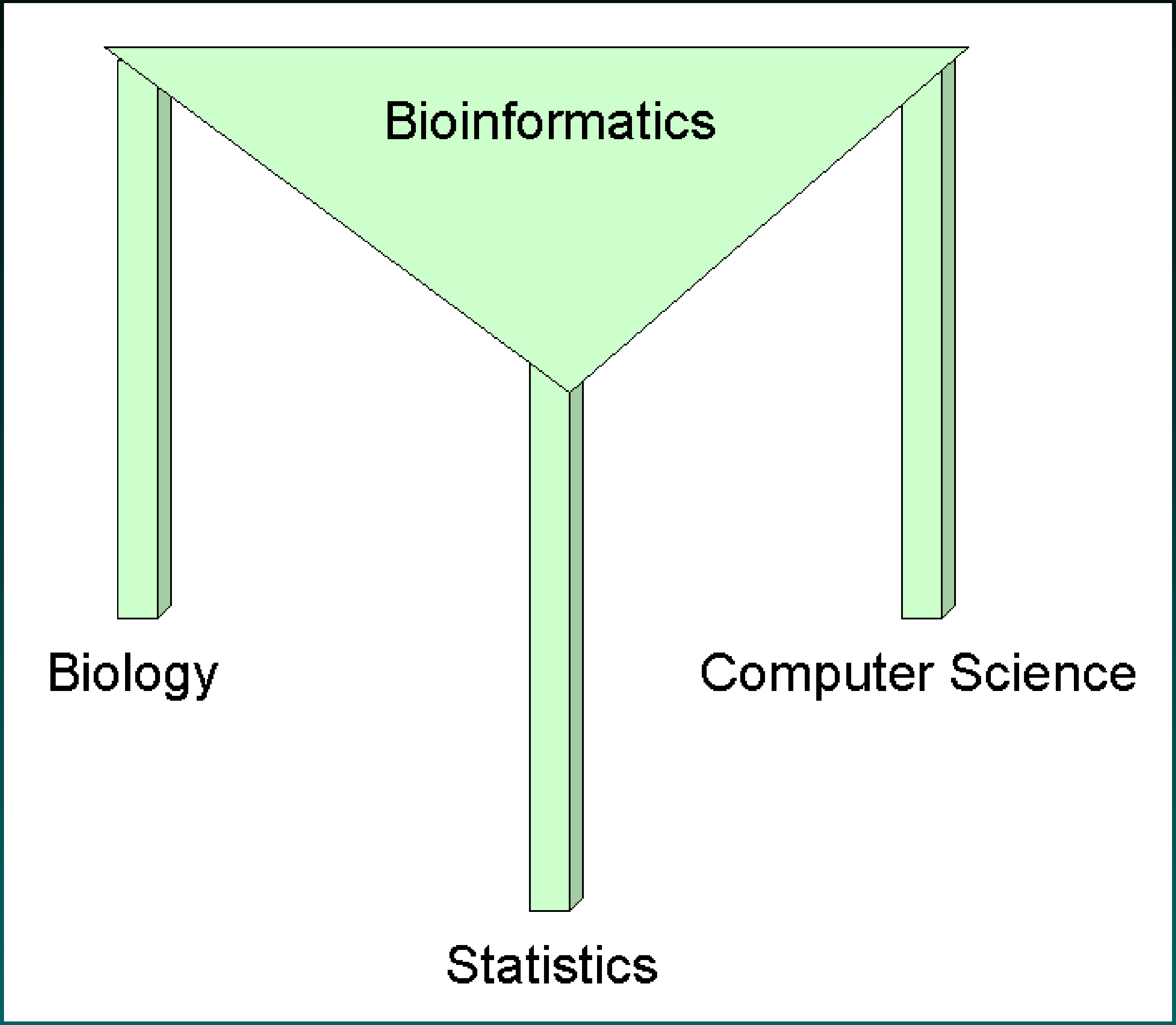
Keith Baggerly and Kevin Coombes
Section of Bioinformatics
Department of Biostatistics and Applied Mathematics
UT M. D. Anderson Cancer Center
kabagg@mdanderson.org
kcoombes@mdanderson.org

16 September 2004

Lecture 6: R and Glass Microarrays

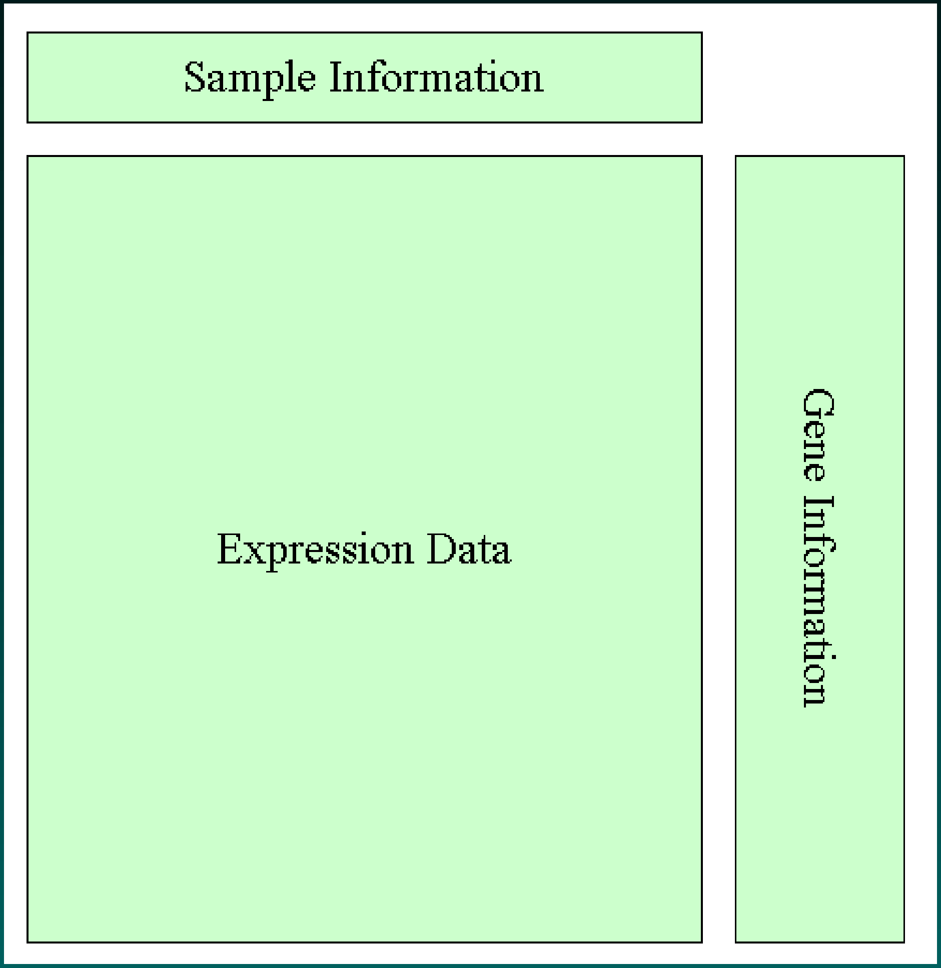
- Microarray Data Structures
- `marray` data structures
- `limma` data structures
- Toward a modular and efficient design

The threefold way



Microarray Data Structures

Recall from last time:



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: `marray` 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.

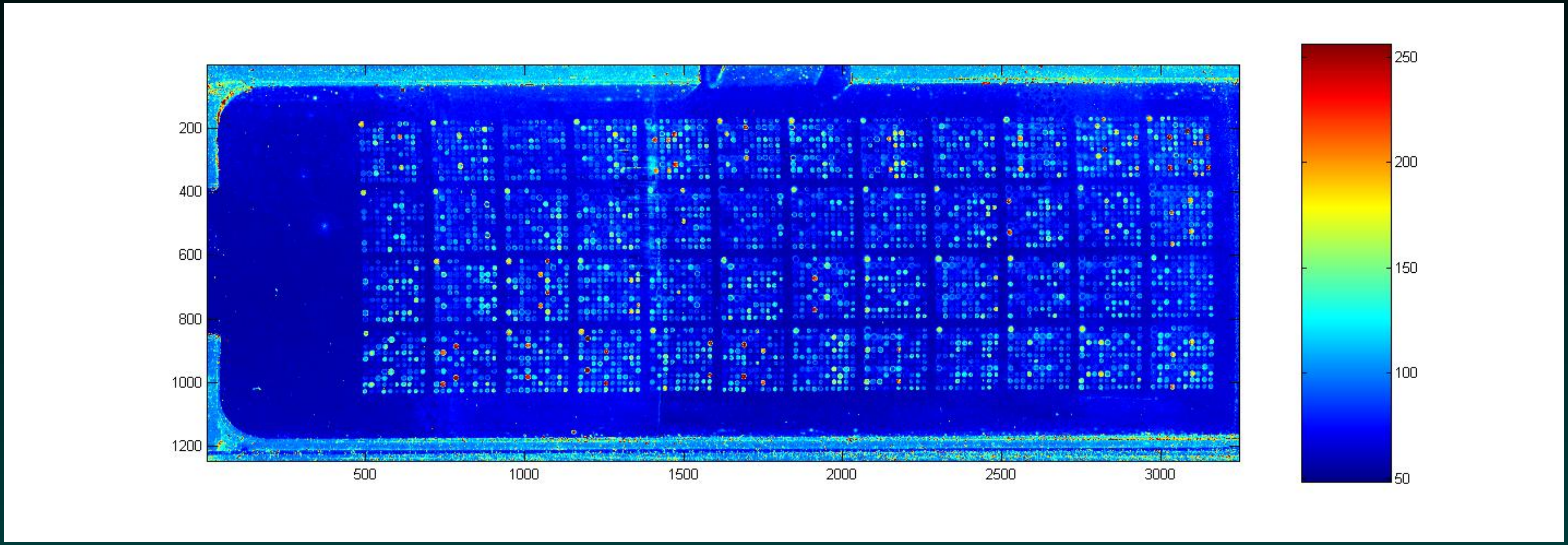
When used to describe samples, the rows correspond to microarrays and columns give information about the samples. In particular, the columns should identify the samples used in both the Cy3 and Cy5 channels.

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.

`marrayLayout` 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.

`marrayRaw` slots

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

- Four matrices of raw data (`maRf`, `maGf`, `maRb`, `maRb`) 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.

`marrayRaw` 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.

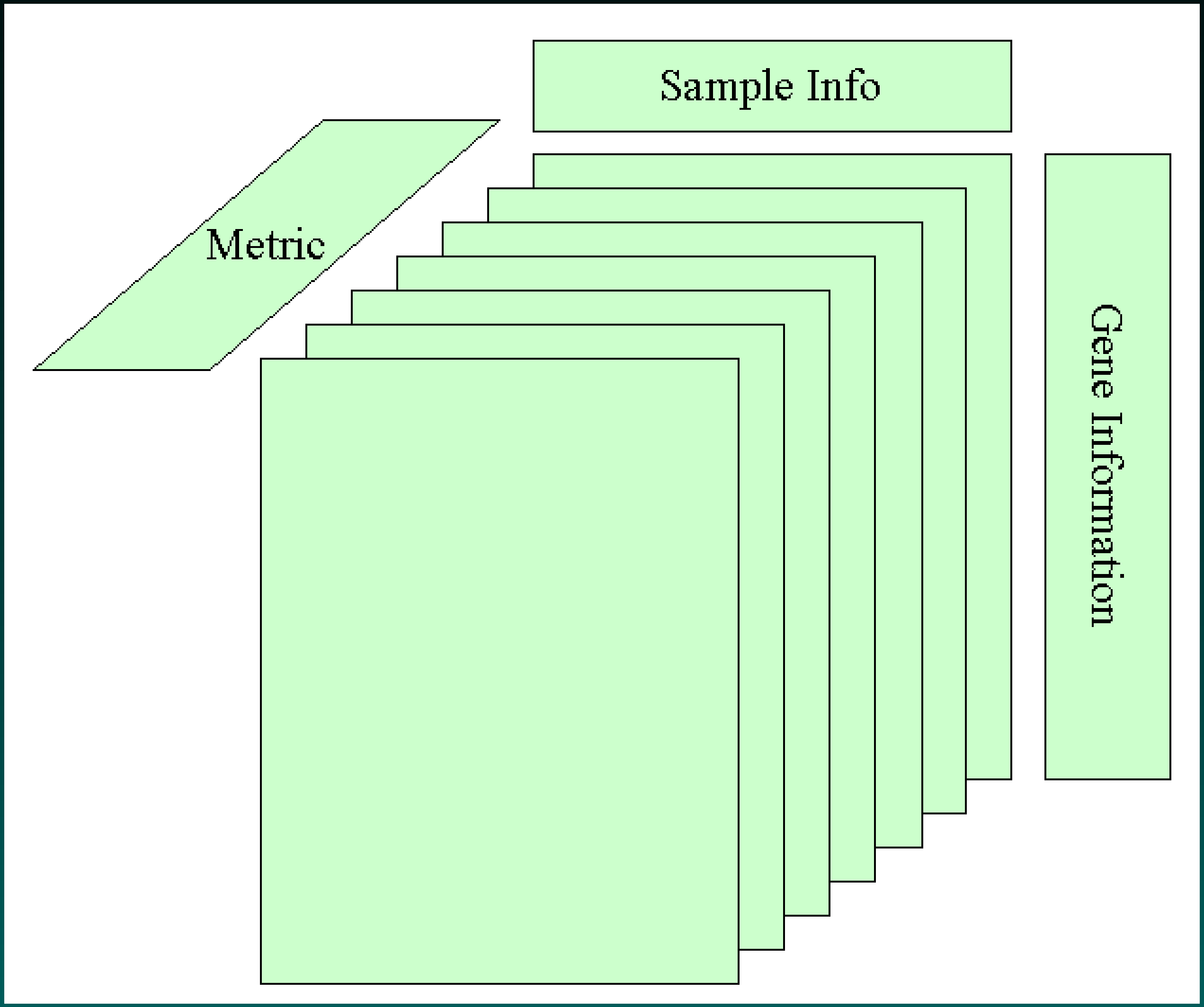
Normalization methods

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.

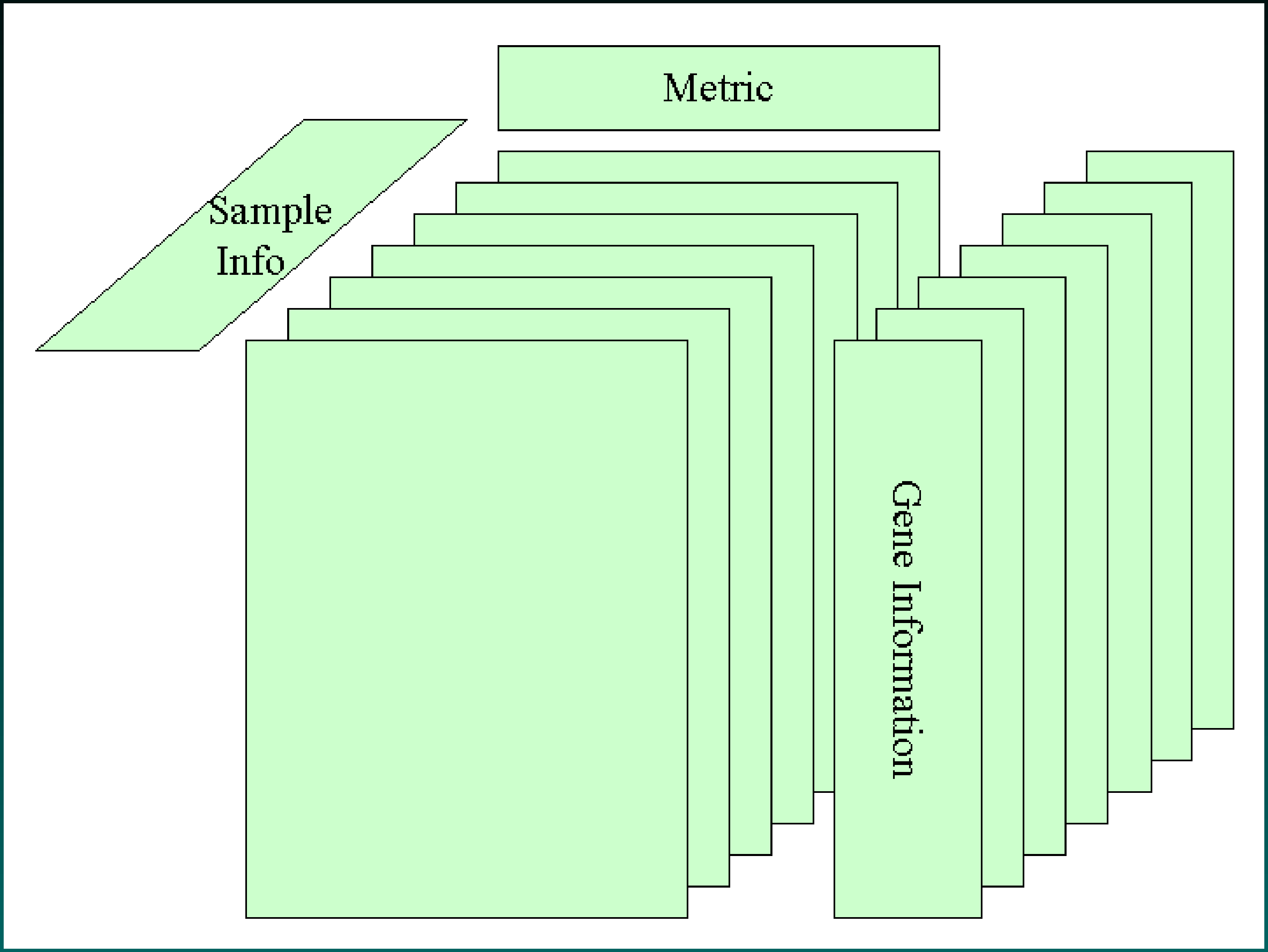
RGList : raw microarray data as a list of arrays containing

- 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 tehy use different data structures than `marray`). Each has hard-coded option lists that are too painful to enumerate (or contemplate).

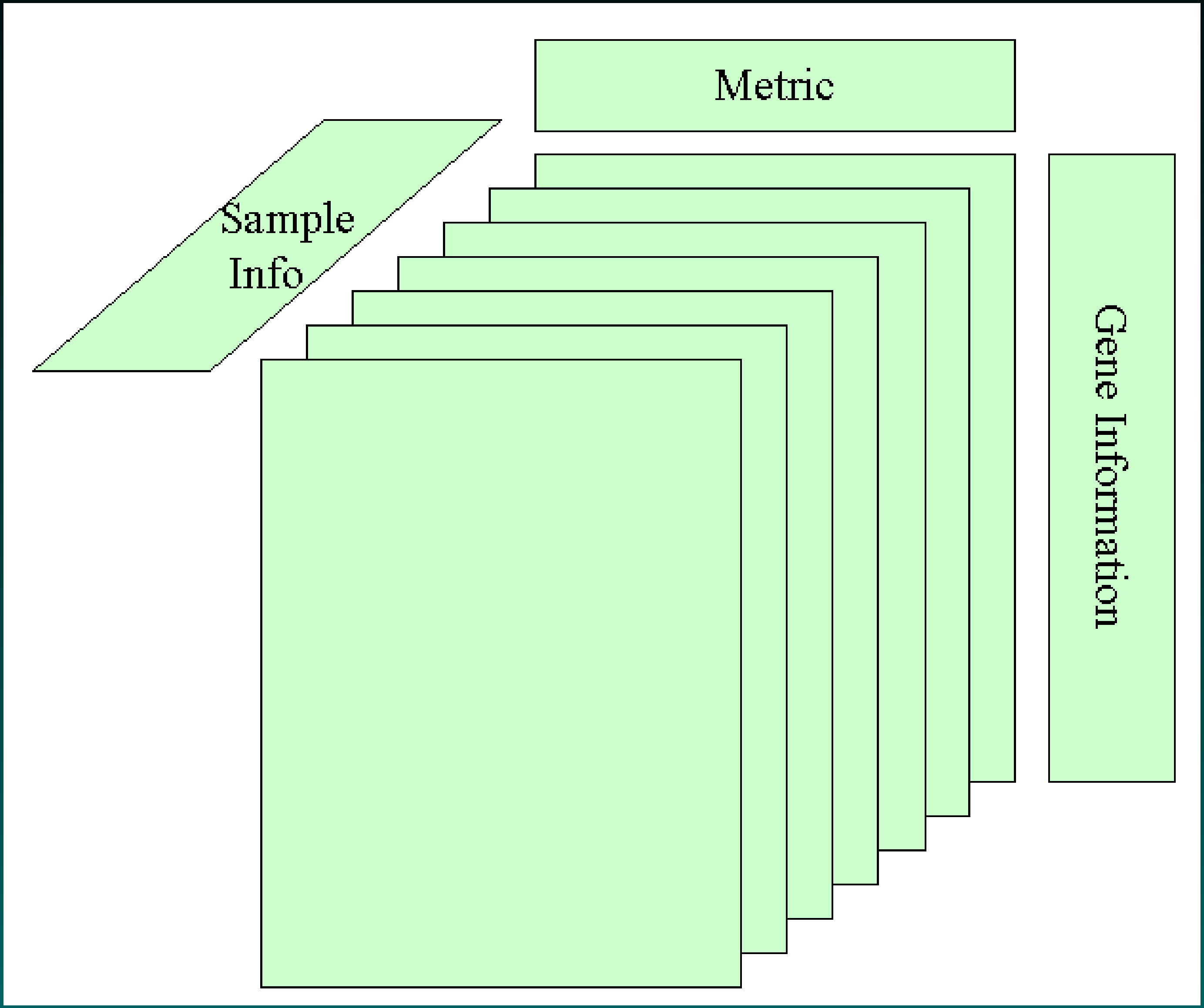
- normalizeBetweenArrays
- normalizeWithinArrays
- normalizeForPrintOrder
- normalizeRobustSpline
- normalizeMedians
- normalizeQuantiles

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

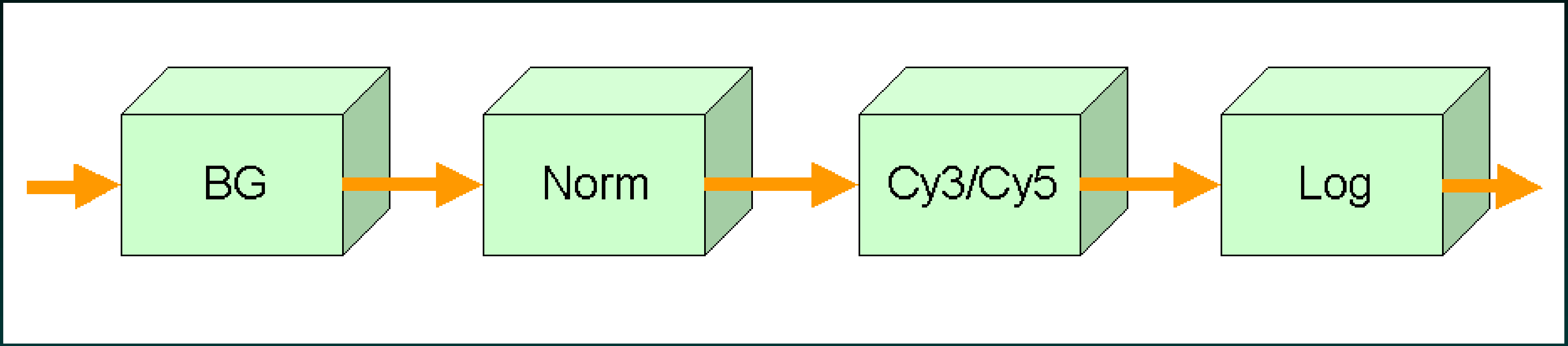
1. Each “sheet” is a slide

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!).

Getting microarray data into R

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 pacakges 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.

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

```
    agilent     arrayvision    genepix
    imagene     quantarray     smd
    spot
```

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.quants`, duplicating much of the existing code to coerce the input adta 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 typicaly arrives as text files in tab-separated values format. Different manufacturers have differnt names for teh 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)
```