

Presence-Absence Calls on AffyMetrix HG-U133 Series Microarrays with *panp*

Peter Warren

October 22, 2008

Contents

1	Background	2
2	Preprocessing CEL file data for <i>panp</i>	3
2.1	Creating an AffyBatch object from the CEL files	3
2.2	Creating the ExpressionSet	3
3	Creating gene detection calls with <i>panp</i>	3
4	Extracting P/M/A calls and p-values	5
5	Illustrating <i>panp</i>'s decisions	5
6	References	7

Introduction

This document describes how to use *panp* to perform gene detection (make presence/absence calls) on AffyMetrix HG-U133 series microarray data. Currently, the HG-U133A and HG-U133 Plus 2.0 are supported. *panp* operates on preprocessed microarray expression data as an `ExpressionSet` object. Any preprocessing method can be used to create the `ExpressionSet`, such as `rma()`, `mas5()`, `expresso()`, or `gcrma()`. Examples show how to quickly create such an `ExpressionSet`, then how to use *panp* to generate a set of presence/absence calls and a set of p-values indicating significance of the detection for each expression value.

1 Background

Outside of the MAS-P/A method which comes as part of the MAS5.0 software for processing Affymetrix oligonucleotide array data (and is also available as `mas5calls()` in the *affy* package), there is no publicly-available method for establishing presence or absence of genes from microarray data. This means that until now the analyst was required to use MAS5.0 software to establish gene presence or absence. It seemed desirable to decouple the method used to generate gene expression values from the method used to make gene detection calls. We have therefore developed a statistical method in R, called "Presence-Absence calls with Negative Probesets" (PANP) which uses sets of Affymetrix-reported probes with no known hybridization partners. This method uses a simple empirically-derived means to generate p-values used for cutoffs, which can reduce errors that can be introduced by using fitted models. In fact, PANP allows a user to utilize any microarray data pre-processing method to generate expression values, including PM-only methods as well as PM-MM methods. Cutoffs are generated in terms of the data on each chip, so even pre-processing methods that do not normalize across chipsets can be used.

Many Affymetrix probesets are designed based on EST matches in the public databases. Normally, these can provide good target matches to predicted protein-coding genes. However, occasionally ESTs are poorly annotated as to their strand direction. As a result, some probesets have been designed in the reverse complement direction against their own transcripts. That is, these probesets cannot hybridize to the true (intended) EST target, but would hybridize instead to the reverse complement if it was transcribed. We decided to call these Negative Strand Matching Probesets (NSMPs). The initial NSMP lists were derived from Affymetrix chip annotation. We then BLATed these against the NCBI dbEST and removed outliers that showed significant EST hits. The resulting sets became our sets of NSMP negative controls, one set for each chip type.

The *panp* package consists primarily of one function, `pa.calls()`, which is used to create the set of presence/absence calls and the set of p-values. It calculates a survivor function of the probability density of the NSMP expression values after preprocessing. The user supplies p-value cutoffs (defaults 0.01/0.02) - let these be called `tightCutoff` and `looseCutoff`. Then `pa.calls()` interpolates cutoff expression values at those p-values. Finally, it makes gene presence determinations as follows, for interpolated intensities above, below and between the cutoff intensities for p-values as follows:

1. Present (P): p-values < `tightCutoff`
2. Marginal (M): p-values between `tightCutoff` and `looseCutoff`
3. Absent (A): p-values >= `looseCutoff`

2 Preprocessing CEL file data for *panp*

This vignette assumes the *affy* package has been installed, as well as the *cdf* and *probe* sequence packages for the chip type you will be using.

First, the CEL file data must be preprocessed to generate expression values. This is done in two stages: create an *AffyBatch* object, then process that to generate expression values as an *ExpressionSet*. In this example, we'll use *gcrma* for the latter, so let's load the required libraries for that and for *panp* (note that loading these packages will automatically cause to be loaded any other packages they require, such as *affy*):

```
> library(gcrma)
> library(panp)
```

2.1 Creating an *AffyBatch* object from the CEL files

Make sure the CEL files are in the current working directory, and that R is pointing to that directory. The *ReadAffy()* function reads in the CEL file data and creates an *AffyBatch* object:

```
> samples <- ReadAffy() # this reads in all CEL files it finds in the directory
```

2.2 Creating the *ExpressionSet*

The *gcrma* function is one way to do this.

```
> gcrma.ExpressionSet <- gcrma(samples)
> # You might wish to save it:
> save(gcrma.ExpressionSet, file= "gcrma.ExpressionSet.Rdata")
```

For this example, we have already done the above steps, using *gcrma* to create an *ExpressionSet* with three samples, which we will now load:

```
> data(gcrma.ExpressionSet)
```

3 Creating gene detection calls with *panp*

First, you can run the *pa.calls()* function with no arguments to obtain a summary of usage information:

```
> pa.calls()
```

USAGE:

```
pa.calls(object, looseCutoff=0.02, tightCutoff=0.01, verbose = FALSE)
```

INPUTS:

```
object - ExpressionSet, returned from expression-generating function,
        such as expresso(), rma(), mas5(), gcrma(), etc.
looseCutoff - the larger P-value cutoff
tightCutoff - the smaller, more strict P-value cutoff
verbose - TRUE or FALSE
```

OUTPUTS:

```
Returns a list of two matrices, Pcalls and Pvals:
Pvals - a matrix of P-values of same dimensions as exprs(input object). Each
        datapoint is the P-value for the probeset at the same x,y coordinates.
Pcalls - a matrix of Presence (P), Marginal (M), Absent (A) indicators
```

NULL

Now, let's run it on our `ExpressionSet`. We'll use the default p-value cutoffs of 0.01 and 0.02. So in this case, intensities above the intensity at the 0.01 cutoff will be called "P" (present); intensities between the two cutoffs will be assigned an "M" (marginal), and those below the intensity at the 0.02 p-value will get an "A"(absent).

```
> PA <- pa.calls(gcrma.ExpressionSet)
```

```
Processing 3 chips: ###
Processing complete.
```

Intensities at cutoff P-values of 0.02 and 0.01 :

Array:	value at 0.02	value at 0.01
12_13_02_U133A_Mer_Latin_Square_Expt1_R1.CEL	3.81	4.1
12_13_02_U133A_Mer_Latin_Square_Expt10_R1.CEL	3.77	4.07
12_13_02_U133A_Mer_Latin_Square_Expt11_R1.CEL	3.89	4.11

[NOTE: 'Collapsing to unique x values...' warning messages are benign.]

The screen output tells you what the intensity values are at each of the two cutoff p-values, for each of the three chips in our `ExpressionSet`. (Be aware that some pre-processing methods, such as `rma` and `gcrma`, return the expression values in $\log(2)$ form. Others return untransformed expression values. `pa.calls()` works equally well in either case.) The final output line informs the user that if any warning messages about 'Collapsing to unique values' appear, they are benign (there are none in this example). The collapsing sometimes occurs when `pa.calls()` interpolates over a large data set, and is expected.

4 Extracting P/M/A calls and p-values

The presence/absence calls and p-values are returned as two matrices, "Pcalls" and "Pvals", respectively, in the returned `list` (here, "PA"). These two matrices can now be extracted for further use in the R environment. They can also be saved as comma-separated files, in case it is desired to view them in Excel, for instance. (You can, of course, also save these as Rdata files for later use in the R environment.)

```
> PAcalls <- PA$Pcalls
> Pvalues <- PA$Pvals
> write.table(PAcalls, file = "PAcalls_gcrma.csv", sep = ",", col.names = NA)
> write.table(Pvalues, file = "Pvalues_gcrma.csv", sep = ",", col.names = NA)
```

A look at the first few P/A calls and p-values for the first chip shows some results:

```
> head(PAcalls[, 1])

1007_s_at    1053_at    117_at    121_at 1255_g_at    1294_at
      "p"        "p"        "A"        "A"        "A"        "A"

> head(Pvalues[, 1])

1007_s_at    1053_at    117_at    121_at 1255_g_at    1294_at
0.00000000 0.00000000 0.06040878 0.03161899 0.76539624 0.68257532
```

Finally, we can extract lists of probeset IDs that were called Present, Marginal and Absent. This must be done one sample at a time; here, we extract the lists for the first chip:

```
> P_list_1 <- rownames(PAcalls)[PAcalls[, 1] == "P"]
> M_list_1 <- rownames(PAcalls)[PAcalls[, 1] == "M"]
> A_list_1 <- rownames(PAcalls)[PAcalls[, 1] == "A"]
```

5 Illustrating *panp*'s decisions

Figure 1 illustrates how the `pa.calls()` function derives its P/A calls. The intensities of the NSMPs versus those of all probesets on the chip (first chip of the three) are shown. The survivor curve (1-CDF) is included, along with lines showing where the default p-value cutoffs land on that curve. This illustrates how the p-value cutoffs are interpolated into intensity cutoffs, using the survivor function. Transcripts whose intensities are above the right-most cutoff line are called present; those between the two lines are called marginal; and those below the left-most line are called absent.

Expression density: NSMPs vs. all, and NSMP survivor curve

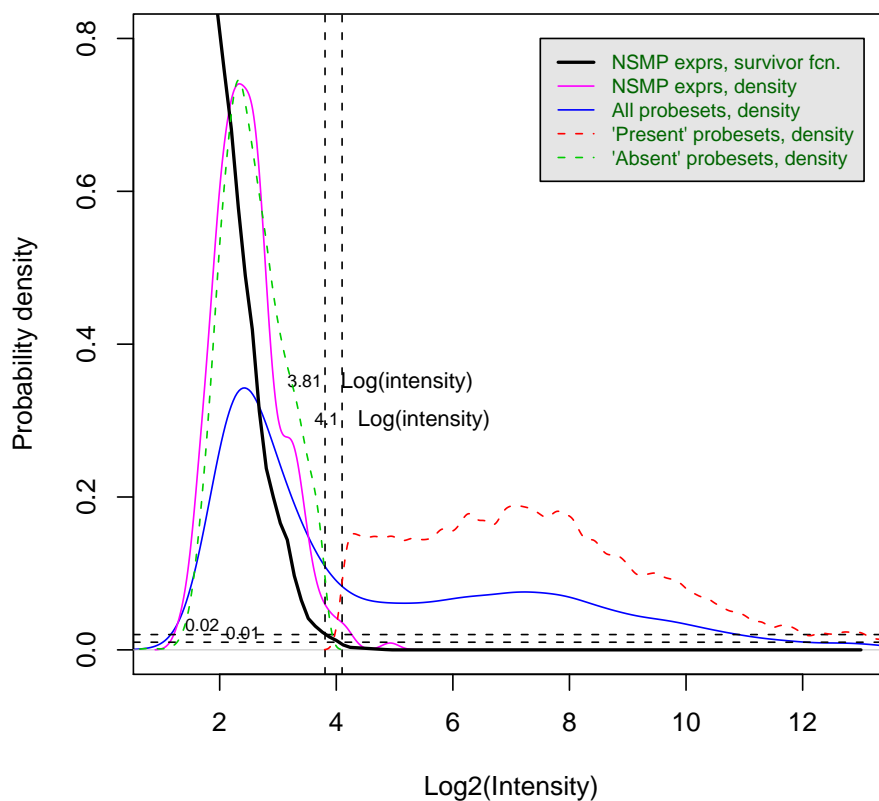


Figure 1: Probability densities of intensities for NSMPs and all probesets for sample number 1. NSMP survivor function (1-CDF) is in black. Dashed curves indicate densities for probesets called "Present" and "Absent" by *panp*. Horizontal lines indicate p-value cutoffs of 0.01 and 0.02, while vertical lines show how these are interpolated on the NSMP survivor curve to get expression cutoff values.

6 References

Warren, P., Bienkowska, J., Martini, P., Jackson, J., and Taylor, D., PANP - a New Method of Gene Detection on Oligonucleotide Expression Arrays (2007), under review