1 Introduction

OOMPA is a suite of object-oriented tools for processing and analyzing large biological data sets, such as those arising from mRNA expression microarrays or mass spectrometry proteomics.

This vignette documents the base package, oompaBase. A critical (but invisible to the user) feature of the oompaBase package is that it defines a class union allowing you to use “numeric” or “NULL” objects in the design of an S4 class. More interesting user-visible features include alternative color schemes and vectorized matrix operations to speed the computation of row-by-row means, variances, and t-tests.

2 Getting Started

You invoke the package in the usual way:

> library(oompaBase)
3 Color Schemes

To illustrate the various color schemes, we first create a structured matrix:

```r
mat <- matrix(1:1024, ncol=1)
```

The following code is used to generate Figure 1:

```r
# windows(width=6,height=8)
opar <- par(mfrow=c(8, 1), mai=c(0.3, 0.5, 0.2, 0.2))
image(mat, col=jetColors(128), main='jetColors')
image(mat, col=wheel(64, 0.5), main='wheel, half saturation')
image(mat, col=redgreen(64), main='redgreen')
image(mat, col=blueyellow(32), main='blueyellow')
image(mat, col=cyanyellow(32), main='cyanyellow')
image(mat, col=redscale(64), main='redscale')
image(mat, col=bluescale(64), main='bluescale')
image(mat, col=greyscale(64), main='greyscale')
par(opar)
```

4 Row-by-row Matrix Operations

We now want to illustrate the “matrix operations” that allow for rapid computation of row-by-row means, variances, and t-tests. We start by creating a slightly more interesting matrix full of random data. First, we make the variance larger in the second half (by column) of the data than in the first half.

```r
ng <- 10000
ns <- 50
dat <- matrix(rnorm(ng*ns, 0, rep(c(1, 2), each=25)), ncol=ns, byrow=TRUE)
```

Next, we shift the mean for the first 500 “genes” (rows).

```r
```

In order to compute t-tests, we also assign arbitrary labels separating the “sample columns” into two groups.

```r
clas <- factor(rep(c('Good', 'Bad'), each=25))
```

Here we compute the row-by-row means.

```r
a0 <- proc.time()
myMean <- matrixMean(dat)
used0 <- proc.time() - a0
```

For comparison purposes, we perform the same computation using `apply`.
Figure 1: Eight color schemes.
> a1 <- proc.time()
> mm <- apply(dat, 1, mean)
> used1 <- proc.time() - a1

The results are the same, to within round-off error.

> summary(as.vector(myMean-mm))

<table>
<thead>
<tr>
<th></th>
<th>Min.</th>
<th>1st Qu.</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Qu.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1.332e-15</td>
<td>-2.776e-17</td>
<td>0.000e+00</td>
<td>9.300e-19</td>
<td>2.776e-17</td>
<td>8.882e-16</td>
</tr>
</tbody>
</table>

There is a measurable (although not really user-perceptible) difference in the time for the two methods.

> used0

<table>
<thead>
<tr>
<th></th>
<th>user</th>
<th>system</th>
<th>elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.01</td>
<td>0.00</td>
<td>0.02</td>
</tr>
</tbody>
</table>

> used1

<table>
<thead>
<tr>
<th></th>
<th>user</th>
<th>system</th>
<th>elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.08</td>
<td>0.00</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Here we compute the variances using two different methods.

> a0 <- proc.time()
> myVar <- matrixVar(dat, myMean)
> a1 <- proc.time()
> vv <- apply(dat, 1, var)
> a2 <- proc.time()

Again, the values are the same:

> summary(as.vector(myVar - vv))

<table>
<thead>
<tr>
<th></th>
<th>Min.</th>
<th>1st Qu.</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Qu.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-3.997e-15</td>
<td>-4.441e-16</td>
<td>0.000e+00</td>
<td>2.200e-20</td>
<td>4.441e-16</td>
<td>3.553e-15</td>
</tr>
</tbody>
</table>

However, the time savings is substantially larger.

> a1 - a0

<table>
<thead>
<tr>
<th></th>
<th>user</th>
<th>system</th>
<th>elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

> a2 - a1

<table>
<thead>
<tr>
<th></th>
<th>user</th>
<th>system</th>
<th>elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.14</td>
<td>0.00</td>
<td>0.14</td>
</tr>
</tbody>
</table>

4
Not surprisingly, there is an even bigger time savings when computing (equal variance) t-statistics.

```r
> t0 <- proc.time()
> myT <- matrixT(dat, clas)
> t1 <- proc.time()
> tt <- sapply(1:nrow(dat), function(i) {
+   t.test(dat[i,clas=="Bad"], dat[i, clas=="Good"], var.equal=T)$statistic
+ })
> t2 <- proc.time()
> summary(as.vector(tt - myT))

      Min. 1st Qu.  Median     Mean   3rd Qu.    Max.    NaNs
-4.441e-15 -1.110e-16  0.000e+00  1.181e-17  1.110e-16  5.329e-15

> t1 - t0

         user  system elapsed
0.0100000 0.0000000 0.0100000

> t2 - t1

         user  system elapsed
1.9700000 0.0000000 1.9700000
```

5 Color Coded Graphs

We frequently find ourselves producing multiple figures with a common color scheme, where each color or each symbol is used to denote samples or genes with a particular property (in the simplest case, “cancer” versus “normal”). Because we got tired of continually cutting and pasting `plot` and `points` commands and making sure the color legends stayed synchronized, we developed the `ColorCoding` and `ColorCodedPair` classes to encapsulate this notion.

We can simulate some data as an example.

```r
> x <- matrix(rnorm(100*3), nrow=100, ncol=3)
> class1 <- class2<- rep(FALSE, 100)
> class1[sample(100, 20)] <- TRUE
> class2[sample(100, 20)] <- TRUE
> class3 <- !(class1 | class2)
> codes <- list(ColorCoding(class1, "red", 16),
+               ColorCoding(class2, "blue", 15),
+               ColorCoding(class3, "black", 17))
```
Figure 2: Color coded plots of three (simulated) related variables.