Short Tutorials for Metagenomic Analysis

This manual describes metagenomic analysis with the `matR` package (Metagenomic Analysis Tools for R). The sections form a progressive set, but can also be rearranged, and many can be treated as independent 10-15 minute tutorials. If this software helps your work, please cite us: *Daniel T. Braithwaite and Kevin P. Keegan (2013). matR: Metagenomics Analysis Tools for R. R package version 0.9.9.*

Contact: `mg-rast@mcs.anl.gov`.

Contents

1 Preliminaries

1.1 Obtaining and Installing R

R is free software, easily downloaded from the R Project Homepage: `http://www.r-project.org`. Binary versions are available for Mac and Windows systems, and source code for Linux. Download and install the version appropriate for your system.

Users who already have R should *update their version*. R and its extensions are frequently updated. Keeping current is important to avoid nuisance errors.

Add-on packages for many purposes, contributed by many people, are a great strength of R. For example, see this list of packages, organized by application area: `http://cran.r-project.org/web/views/`. For a repository dedicated entirely to biological functionality, see: `http://www.bioconductor.org`.

Now install `matR`, the MG-RAST interface add-on package. For this, use:

```
> install.packages("matR", repo="http://dunkirk.mcs.anl.gov/~braithwaite/R", type="source")
```

Open an R session. Use the following command to load the `matR` package (you would use a similar command to load any other package):

```
> library(matR)
```

`matR` relies on various other packages. To install these, follow the instructions provided by running this function:

```
> dependencies()
```

At the time of this writing, the packages relied on by `matR` are: `RJSONIO`, `ecodist`, `gplots`, `scatterplot3d`. If the `dependencies` function doesn't complete successfully, these need to be installed one at a time, as follows:

```
> install.packages("RJSONIO")
> install.packages("ecodist")
> install.packages("gplots")
> install.packages("scatterplot3d")
```

Now your R environment is ready to go!

1.2 Introduction to R

Here we review some basics of working with data in R, but the treatment is necessarily brief. For detailed R language tutorials, try: `http://www.ats.ucla.edu/stat/r`.

For us, two kinds of data objects are essential in R: `matrix` and `data.frame`. First, we create a `matrix`. The function `sample` just creates a random permutation, as shown.

```
> sample(1:200)

  [1] 122 101 135  48 110 154  54  47 116  28 165   6  83 196 124 180 143   2
 [19] 139  67  52  86 100  88 114  71 145 113 130  33  21  38 197  57 119 131
 [37]   7  50  49  30 123  80  51  23  69  13 147  95 198 150 102  81  22  36
 [55] 112 188  74  93 162 187  12 155 108  46  62 138  70  65  79 199  40 128
 [73]  97 126  61  59 111  98  85  37 127  92 192   3  45 134 183 158  19 179
 [91] 105 118  25  76 107 184 115 142   8  66  14  84 164 161 159 153 172  16
[109] 166 133  68  91  90  41 186  43 185  20 104 120 141  53 175  26  34  18
[127]  73  35 148   9  94  96  24  17   4  15  82  10 156  42  39  29 168 176
[145]  31  11   5 194 125  75 146 140 163  77 160 121 109 170 132 136 177 137
[163] 151 129  60 173 171 144  64  63 169   1 193 167 200  32 174  78 190  58
[181]  99 117  89 181 191  27 189  55  56 157 106 178 103  72  87 152 195 182
[199]  44 149
```

```
> m <- matrix(sample (1:200), nrow=20, ncol=10)
> m

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]   37   84   76  187   61   14  133  106   94    38
 [2,]   68  181  169   60  186  145   48  125  166   131
 [3,]   52  146  113   86  174  101   64  171  143    47
 [4,]  119   26  162   93   54   43  137  114  129    39
 [5,]  100  189   66   90   70   92   85   30    4    72
 [6,]   62   25  163  134   59   55  188  159  149   178
 [7,]   57  156  198  160  182  110   89  193  172   184
 [8,]   40   73   83  111   45  167   56   81  109    16
 [9,]    1    3   69  104   17  185  108   91   29   132
[10,]   67   19  136  165  126  161  179   13  175    63
[11,]  176   44  168   34  147   87   58  191   15   138
[12,]  199    9  153  122   96   24  116  148    2    42
[13,]  117    6   79   99  105  121   97   98   74   107
[14,]   95   18  196  157   77   32  141  103    8   139
[15,]  200  180  190   28   11  118  183   78   23    46
[16,]   20   22  130  152  177  158   10   71  144     7
[17,]    5   75  124   33   41  151  142   53  127    31
[18,]   88  120  128  112  115  173  164   50  170    21
[19,]  195  102  140   51  192   12   65   27  135   123
[20,]   35   82   80   36  194   49  150  197  154   155
```

The `apply` function, below, applies the function specified by its last argument (in this case, `mean`) along the dimension of `m` specified by the second argument. So here we calculate the row means and then the column means of `m`.

```
> apply(m,1,mean)

 [1]  83 128 110  92  80 117 150  78  74 110 106  91  90  97 106  89  78 114 104
[20] 113
```

```
> apply(m,2,mean)

 [1]  87  78 131 101 106 100 111 105 101  85
```

Generally speaking, a `data.frame` is different from a `matrix` because it may contain non-numeric data. So, now we create a `data.frame` consisting of the *column means* and *column standard deviations* of `m`, but also containing a third, descriptive column.

```
> df <- data.frame(mu=apply(m,2,mean), sigma=apply(m,2,sd))
> df$sample <- paste("sample", LETTERS[1:10], sep = "-")
> df

    mu sigma   sample
1   87    63 sample-A
2   78    64 sample-B
3  131    44 sample-C
4  101    49 sample-D
5  106    62 sample-E
6  100    58 sample-F
7  111    51 sample-G
8  105    57 sample-H
9  101    64 sample-I
10  85    58 sample-J
```

Suppose we wanted to reorder the columns. Flexible indexing of objects is a great strength of R. Here we *replace* the first and third columns of `df` with (respectively) its own third and first columns — effectively, reordering them.

```
> df [c(1,3)] <- df [c(3,1)]
> df

         mu sigma sample
1  sample-A    63     87
2  sample-B    64     78
3  sample-C    44    131
4  sample-D    49    101
5  sample-E    62    106
6  sample-F    58    100
7  sample-G    51    111
8  sample-H    57    105
9  sample-I    64    101
10 sample-J    58     85
```

That almost worked, but notice that while the data moved, the column *labels* did not. It is possible to refer directly to the row and column labels of a `matrix` or `data.frame`, as follows.

```
> rownames(df)

 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

> colnames(df)

[1] "mu"     "sigma"  "sample"
```

Now we finish by correcting the column labels.

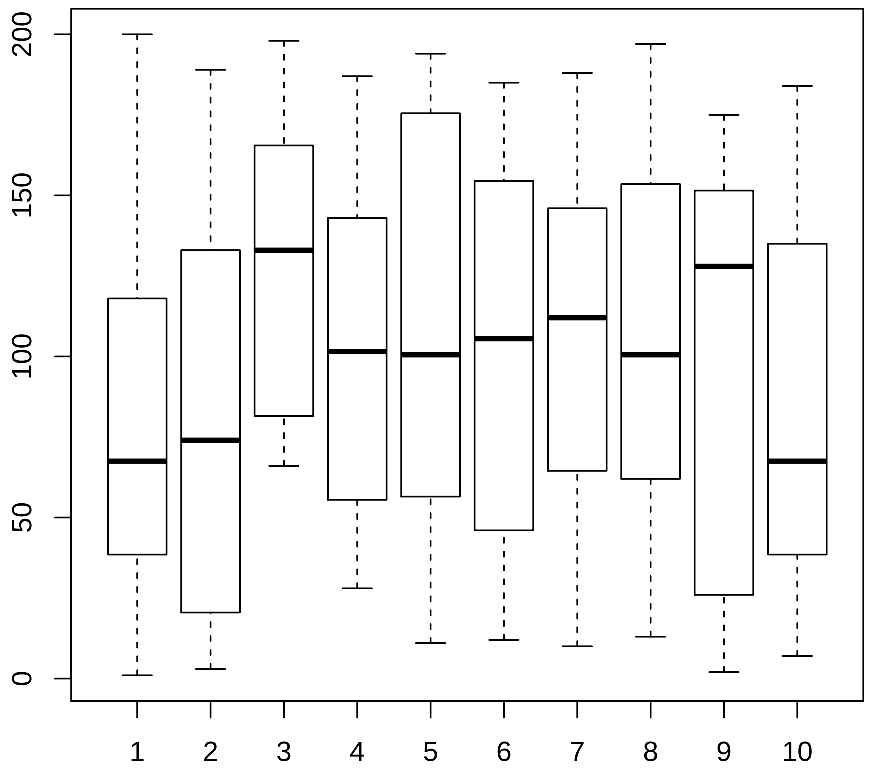
```
> colnames(df) [c(1,3)] <- colnames(df) [c(3,1)]
> df

     sample sigma  mu
1  sample-A    63  87
2  sample-B    64  78
3  sample-C    44 131
4  sample-D    49 101
5  sample-E    62 106
6  sample-F    58 100
7  sample-G    51 111
8  sample-H    57 105
9  sample-I    64 101
10 sample-J    58  85
```

Here are some commands for viewing the first elements, last elements, and overall structure of large objects.

```
> head(m)

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   37   84   76  187   61   14  133  106   94    38
[2,]   68  181  169   60  186  145   48  125  166   131
[3,]   52  146  113   86  174  101   64  171  143    47
[4,]  119   26  162   93   54   43  137  114  129    39
[5,]  100  189   66   90   70   92   85   30    4    72
[6,]   62   25  163  134   59   55  188  159  149   178

> tail(m)

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[15,]  200  180  190   28   11  118  183   78   23    46
[16,]   20   22  130  152  177  158   10   71  144     7
[17,]    5   75  124   33   41  151  142   53  127    31
[18,]   88  120  128  112  115  173  164   50  170    21
[19,]  195  102  140   51  192   12   65   27  135   123
[20,]   35   82   80   36  194   49  150  197  154   155

> str(m)

 int [1:20, 1:10] 37 68 52 119 100 62 57 40 1 67 ...

> str(df)

'data.frame':        10 obs. of  3 variables:
 $ sample: chr  "sample-A" "sample-B" "sample-C" "sample-D" ...
 $ sigma : num  63.5 64.5 43.9 49.1 62.1 ...
 $ mu    : num  86.7 78 131.2 100.7 106.5 ...
```

Finally, any introduction to R should show how it easily renders statistical graphics, as with this boxplot of the columns of `m`.

```
> boxplot(m)
```



There is a lot more to R, but the subset of commands shown here, together with the help tutorial (which is next), already enable many things!

1.3 Using R Help

In R, as with any system, it's important to know how to use the help.

First, locate the one-page quick reference for all `matR` commands:

```
> vignette("matR-quick-reference")
```

If that doesn't work, the quick reference is also available at: `http://dunkirk.mcs.anl.gov/~braithwaite/library/matR/doc/matR-quick-reference.pdf`. It may be handy to print a copy.

Help on any R command is available with:

```
> ?command
```

For example, try:

```
> ?mean
> ?sample
> ?apply
```

For keyword-based help, use the double question mark, as in these examples:

```
> ??random
> ??plot
```

Finally, to retrieve an index of all help topics *for a specific package*, use this command, replacing `matR` with the name of the relevant package:

```
> library(help="matR")
```

`matR` is updated regularly. For a summary of the latest changes, see:

```
> vignette("matR-change-log")
```

The same document is also available at: `http://dunkirk.mcs.anl.gov/~braithwaite/library/matR/doc/matR-change-log.pdf`.

1.4 Exporting and Importing Data; Saving Images

This tutorial explains how to get images out of R for publications, how to bring data into R from formats such as csv, tsv, or biom; and how to save data for use in future R sessions, in Excel, or with other programs.

`matR` provides a function, `asFile()`, that conveniently exports several kinds of object in a default format. It's not flexible but may be adequate for many purposes. Try it on any vector or matrix object:

```
> asFile(cc$raw, file="saved_matrix.txt")
```

`write.table()` and `read.table()` are the workhorse commands for exporting and importing any kind of tabular data. They have many options, as well as variants such as `read.csv()`. The following examples show the most common options. These functions are very flexible, though, so consult the help system to learn more.

```
> cc <- collection("4441679.3 4441680.3 4441682.3")
> write.table(cc$raw, file="data.txt", sep="\t")
> x <- read.table(file="data.txt")
> x
```

The functions `save()` and `load()` store R objects in a binary format for use in later R sessions. (By convention, these files end with `.Rda`.) This is helpful, for example, to store a metagenome collection or the result of an analysis that is computation-intensive. Here are some examples:

```
> cc <- collection("4441679.3 4441680.3 4441682.3")
> p <- pco(cc)
> ls()
> save(cc, p, file="saved_data.Rda")
> rm(cc, p)
> ls()
> load(file="saved_data.Rda")
> ls()
```

There is an easy method to export images from an R session. First develop the exact commands to produce the desired image interactively. For instance, suppose we want to export the following PCoA.

```
> pco(Waters, main="functional level 3", col=c(rep("red",12),rep("blue",12)))
```

To produce a pdf file, simply amend the code in this way.

```
> pdf(filename="my_pco.pdf", width=5, height=5)
> pco(Waters, main="functional level 3", col=c(rep("red",12),rep("blue",12)))
> dev.off()
```

The function `pdf()` can be replaced with others, such as `png()`. For more detail, consult the help system.

1.5 Data Type Conversions (including BIOM)

In most programming languages, it is important to know the kind (or type or class) of data objects. This can be a vexed subject in R. Our purposes require: `vector`, `matrix`, `data.frame`, `list`, `collection`, and `BIOM`.

2 Examples

2.1 Functional Comparison of Lean and Obese Mouse

2.2 HMP Samples with External Metadata

2.3 Variability of Clustering by Annotation Source

2.4 Parallel Coordinates of Brazilian Coastal Samples

2.5 Where to Find More

A gallery of additional simple examples is maintained at: `http://dunkirk.mcs.anl.gov/~braithwaite`.

3 Basics

3.1 Data in an Annotation Matrix

The columns of a `matR` matrix are labeled by sample, and rows are labeled by annotation. The annotations may be taxonomic or functional, at various hierarchy levels. Often, the matrix entries are raw counts of annotations per sample. So an "OTU table" is just one kind of `matR` matrix.

The matrix may also contain other quantities such as (for instance) normalized abundance counts, or average read length of annotated sequences, per annotation and per sample. Matrix entries may also be qualified or limited. For example, counts may be requested only from a particular annotation database.

Suppose you have selected a particular set of metagenomes. Next, in order to retrieve related data, you have to specify exactly what data you want. Such a description is called a `view` of the data, and it is spelled out with predefined options. Here are some examples of `views`:

```
> c(level="level1")
> c(annot="organism",level="phylum")
> c(entry="normed.counts",source="NOG")
```

The first line indicates counts per functional annotation at level 1 of the Subsystems hierarchy. The second indicates counts of *taxonomic* annotations at phylum level from the M5RNA database. The third indicates *normalized* counts of functional annotations from only the NOG database.

The options for data `views` are listed and fully described in the `matR` package itself. Examine these objects at the R prompt just by typing their names:

```
> view.descriptions
> view.parameters
> view.defaults
```

The last one, `view.defaults`, shows what data is retrieved if you don't choose explicitly.

3.2 Metagenome Collections

Metagenome data is always retrieved by constructing a `collection`. The samples of interest must be identified by ID. Here are some examples.

```
> IDs <- c(gut1="4441695.3", gut2="4441696.3")
> cc <- collection(IDs)
> dd <- collection("4441679.3 4441680.3 4441682.3 4441695.3 4441696.3 4440463.3 4440464.3")
> ee <- collection(file="test-IDs.txt")
```

In the first example, the samples are given names. The last example reads a list of IDs from a text file. IDs in files should be whitespace-separated. The file may also contain names in a first column and IDs in a second column. In addition to metagenome IDs, project IDs may be used. The effect is to request all metagenomes from that project. Project IDs should begin with `"mgp"`.

Choosing samples is only half the story: various data pertaining to those samples can be requested. In each of the following examples, each part of the `collection` function names and describes a distinct `view` of the data, as discussed above.

```
> collection(IDs,
+   raw=c(entry="count"),
+   nrm=c(entry="normed.counts"))
> collection(IDs,
+   L1=c(level="level1"), L2=c(level="level2"),
+   L3=c(level="level3"), L4=c(level="function"))
> collection(IDs,
+   nog=c(source="NOG"),
+   cog=c(source="COG"),
+   ko=c(source="KO"))
> collection(IDs,
+   lca=c(annot="organism", hit="lca"),
+   repr=c(annot="organism", hit="single"),
+   all=c(annot="organism", hit="all"))
```

A handy techniques is to make lists of `views`:

```
> top.levels <- list(
+   L1=c(level="level1"),
+   L2=c(level="level2"))
> all.ontologies <- list(
+   nog=c(source="NOG"),
+   cog=c(source="COG"),
+   ko=c(source="KO"),
+   sub=c(source="Subsystems"))
> all.count.methods <- list(
+   lca=c(annot="organism", hit="lca"),
+   repr=c(annot="organism", hit="single"),
+   all=c(annot="organism", hit="all"))
```

Such lists can then be used (and reused) as follows:

```
> cc <- collection (guts, top.levels)
> dd <- collection (guts, all.ontologies)
> ee <- collection (guts, all.count.methods)
```

The matrix of data corresponding to a `view` is accessed with `$` plus the appropriate name:

```
> cc$L1
> dd$nog
> ee$all
```

`views` can be specified when a `collection` is constructed, as shown above, and can also be added to an existing `collection` in this way:

```
> dd$cog <- c(source="COG")
```

Various common sense functions apply to `collections`:

```
> samples(cc)      # show metagenomes in the collection
> projects(cc)     # show projects in the collection
> names(cc)        # show names of metagenomes
> views(cc)        # show the data views in the collection
> viewnames(cc)    # show just the names of the views
> groups(cc)       # show grouping of metagenomes (if assigned)
> metadata(cc)     # access metadata
```

(For more about metadata, see below.) Values may be assigned to `names`, `viewnames`, and `groups`, as with:

```
> names(cc) <- c("new.name.1", "new.name.2")
```

Within each `view`, the names of annotations are accessed with `rownames`. Annotation names are hierarchical, and the `sep` parameter affects how the hierarchy is presented. There are four alternatives:

```
> rownames(Guts, view="raw", sep=NULL)
> rownames(Guts, view="raw", sep=FALSE)
> rownames(Guts, view="raw", sep=TRUE)
> rownames(Guts, view="raw", sep="\t")
```

The corresponding results are: annotations named by terminal hierarchy level only; a matrix of annotation names with one column per hierarchy level; annotations named by semicolon-separated concatenation of all hierarchy level names; same as previous, but with specified separator character.

Subsets may be taken of collections, as of other objects in R. Here we extract the first three samples of `dd` into a new `collection`.

```
> ff <- dd[1:3]
```

3.3 Using Metadata

Collections have metadata elements, which are named. The names of elements reflect the hierarchical nature of metadata. To see all metadata of the collection `Guts`, which is prepackaged with `matR`, simply enter:

```
> metadata(Guts)
```

Analyses usually require picking out specific metadata elements, and metadata can be indexed for that purpose. Metadata indexing is by element name(s), and an arbitrary number of indices may be specified. This is best understood by example. First, we use *one index* of *length one* to get all metadata from one sample of the collection:

```
> metadata(Guts)["4440464.3"]
```

Here is an example of metadata indexing using *two indices*, each of *length one*, to get sampling location information for all samples.

```
> metadata(Guts)["latitude", "longitude"]
```

An alternative form returns the same output in a more convenient form.

```
> metadata(Guts)["latitude", "longitude", bygroup=TRUE]
```

In this variant NA is placed when a field is missing, as in the next example.

```
> metadata(Guts)["host_common_name", "disease", ".age", bygroup=TRUE]
```

The next example obtains the entire environmental package from one metagenome using *one index* of *length two*. Only metadata fields matching *both* strings are selected:

```
> metadata(Guts)[c("4440464.3","env_package.data")]
```

Finally, this example uses *three indices* all of *length two* to select miscellaneous elements:

```
> metadata(Guts)[c("env","temp"), c("4440464.3","PI_organization"), c("0464","biome")]
```

Actually, metadata can be handled independently of annotation data. This saves time when annotation data is not needed. Metadata can be retrieved by sample, just as with the `collection` function:

```
> mm <- metadata("4441679.3 4441680.3 4441682.3 4441695.3 4441696.3")
```

Now `mm` can be used just as `metadata(Guts)` was used above.

4 Analysis

- `matR` provides new analysis methods as well as customized versions of functions included in base R and contributed packages. The latter are gratefully acknowledged: `qvalue`, `ecodist`, `gplots2`.
- `matR` functions build on existing functions by adding features and helpful defaults. Options to existing functions usually also apply to `matR` versions. The former are directly available to users who want more control, of course.
- (Some analyses have graphical representations, and others do not. A universal function, `render()`, visualizes the results of analysis computations. This functionality enables fast re-visualization (with modified parameters) of costly computations. However, the implementation is not yet complete.)
- As discussed earlier, a `matrix` within a `collection` is called a `view` and can be extracted with `$`. Conversely, a standalone `matrix` can be converted into a `collection` with class coercion via `as(my_matrix, "collection")`. Since some functions below apply to a `matrix`, and others to a `collection`, these conversions are important to understand.
- Some functions accept a grouping, which can be specified by any vector equal in length to the number of samples (columns). `collection` functions usually accept the parameters `view` and `rows`, which determine what part of the `collection` is analyzed.
- More detail on inputs, options, and outputs is given below. `matrix` functions are discussed first, then `collection` functions.

4.1 Singleton Removal and Normalization

It's a good idea to ignore abundance counts of one (singletons). The `remove.singletons()` function accomplishes that. Also, abundance values that have been normalized can be more meaningful than raw counts. For that `matR` includes the function `normalize()`.

```
> cc <- collection(...)
> ns <- remove.singletons(cc$raw)
> nrm <- normalize(r)
```

Options to both functions are detailed in the help system.

4.2 Distance between Samples and Groups

`matR` extends the base R function `dist` in several ways. Additional metrics / dissimilarities can be selected with the `method` parameter. For metagenomic analysis, the parameter `bycol` is usually appropriate, to compute distance between columns rather than rows. With groups specified, a square matrix of intra- and inter-group mean pairwise distances is returned.

```
> dist(m, method="bray-curtis", bycol=TRUE)
> dist(m, groups=c(1,1,1,2,2,2,3,3,4,4,4,4), bycol=TRUE)
```

With an additional vector specified, its distance to each row or column is computed. When groups are also specified, mean pairwise distances from the vector to each group are computed.

```
> dist(m, y, bycol=TRUE)
> dist(m, y, groups=c(1,1,1,2,2,2,3,3,4,4,4,4), bycol=TRUE)
```

See the help system for more detail.

4.3 Statistical Significance Tests

The function `sigtest` is a convenient interface to apply any of several statistical signifance tests to annotations (rows) of a matrix. The specified test is applied, given a grouping of samples (columns), to each annotation (row). The tests typically test the null hypothesis that the group means of annotation abundances (whether raw or normalized) are the same. Qvalue testing can be applied to the multiple tests, but must be explicitly requested. As with all other function below, the components of the analysis results are returned in a list.

```
> sigtest (m, groups=c(1,1,1,2,2,2,3,3,4,4,4,4), test="Kruskal-Wallis")
> sigtest (m, groups=c(1,1,1,2,2,2,3,3,4,4,4,4), test="Kruskal-Wallis", qvalue=TRUE)
> sigtest (m, groups=c(1,1,1,2,2,2,3,3,4,4,4,4), test="Kruskal-Wallis", qvalue=TRUE, fdr.level=0.01)
```

4.4 Randomization Tests

The function `randomize` facilitates randomization (or permutation) analyses. It returns the result of applying any given summary function to each of a specified number of random permutations of a matrix. Several different randomization methods are implemented.

```
> randomize (m)
> randomize (m, n=10, method="sample")
> randomize (m, n=10, method="rowwise", FUN=mean)
> randomize (m, n=10, method="dataset", FUN=colSums, na.rm=TRUE)
> randomize (m, n=10, method="complete", FUN=function (m) apply (m, MARGIN=2, hist, plot=FALSE))
```

`sample` randomization randomly permutes the entries of each column. `rowwise` randomization randomly permutes the entries of each row. `dataset` randomization randomly permutes entries across the entire matrix. `complete` randomization randomly reassigns each (unit) annotation count.

4.5 Boxplots of Diversity

Boxplots are useful to summarize the distribution of annotation counts in samples of a collection. Boxplots are produced by the `render` function applied to a collection, since they illustrate data so directly. As with other functions below that apply to collections, a `view` may be specified or omitted.

```
> render(Waters)
> render (Waters, notch = TRUE, pch = 19, cex = 0.5, names = names (waters),
+ main = "Annotation Diversity at Function Level 3", cex.axis = 1.1)
```

For applicable graphical parameters, see `?base::boxplot`. The most useful are `main`, `names`, `notch`, and `outline`.

4.6 Principal Coordinates

The `pco` function also operates on a collection object. `rows` can be used to limit the analysis to specified annotations. `comp` specifies which principal components (1, 2, or 3 may be selectd) to plot, and `method` specifies the metric / dissimilarity used (as in `dist`).

```
> pco(cc)
> col <- factor (metadata (cc) ["biome"])
> levels (col) <- c ("#1F78B4", "#E31A1C", "#B15928")
> col.vec <- as.character (col)
> pco (cc, view="norm", comp = c (2,3,4), sub = "Principal Coordinates 2 to 4", cex.sub = 1.5,
+ main = "", color = col.vec, labels = "", cex = 1.5, lty.hplot="dashed",
+ mar = c (5,5,0,3))
```

The most important graphical parameters are `col` (for 2-d plots), `color` (for 3-d plots), `labels`, and `main`. For others, see `?graphics::points`, `?graphics::text`, and `?scatterplot3d::scatterplot3d`.

4.7 Heatmap-Dendrograms

`heatmap` applies to collections and accepts optional parameters `view` and `rows`, as well.

```
> cc <- collection("....", n1 = c(entry="ns.normed.counts", level="level1"), raw=default.views$raw)
> test.result <- sigtest(cc$n1, "Kruskal")
> red.yellow <- rgb (colorRamp(c ("#FFFFCC", "#800026")) (seq(0, 1, length = 20)), max = 255)
> heatmap(cc)
> heatmap(cc, view="n1", rows=test.result$significant, main="significant annotations only", labRow=NA, l
```

Some common graphical parameters are illustrated above. See `?gplots::heatmap.2` for more possibilities.

4.8 Parallel Coordinates

5 Miscellaneous

5.1 API Calls for Extended Functionality

The full functionality of the MG-RAST API is available through `matR`. For API details, see `http://api.metagenomics.anl.gov`.

Many API resources are available with a convenient syntax using the mid-level interface function, `mGet`.

```
> example
```

For more control, use the low-level function `callRaw`. This function simply prepends the API server name and appends the session authorization key (if set) to its argument.

```
> example
```

Most API resources are returned as JSON objects and automatically parsed by `mGet` (or `callRaw`) into a list structure. JSON text can be retained with `parse=FALSE`.

5.2 Using `matR` within an iPython Notebook

`matR` is easily invoked from iPython Notebook to leverage the many advantages of that scripting environment.

5.3 Other Packages: `ggplot2`, `vegan`, `picante`

`matR` interacts easily with other R software for graphics and analysis.