Analysis of data from aCGH experiments using parallel computing and ff objects

Ramon Diaz-Uriarte[1], Daniel Rico[2], and Oscar M. Rueda[3]

3-May-2016

1. Department of Biochemistry, Universidad Autonoma de Madrid Instituto de Investigaciones Biomedicas "Alberto Sols" (UAM-CSIC), Madrid (SPAIN). 2. Structural Computational Biology Group. Spanish National Cancer Center (CNIO), Madrid (SPAIN). 3. Cancer Research UK Cambridge Research Institute Cambridge, UK
rdiaz02@gmail.com, drico@cnio.es, Oscar.Rueda@cancer.org.uk

Contents

1 This vignette

This vignette presents the ADaCGH2 package using:

- Three fully commented examples that deal with the usage of the different parallelization options and types of objects (in particular, *ff* objects) available.
- Examples of using ADaCGH2 with CGHregions, Limma, and snapCGH.

All of the runnable examples in this vignette use a small toy example (they need to run in a reasonably short of time in a variety of machines). In the vignette called "ADaCGH2-long-examples" we list example calls of all segmentation methods, with different options for methods, as well as different options for type of input object and clustering. That other vignette is provided as both extended help and as a simple way of checking that all the functions can be run and yield identical results regardless of type of input and clustering.

Finally, the file "benchmarks.pdf" presents extensive benchmarks comparing the current version of ADaCGH2 (>= 2.3.6) with the former version (v. 1.10, in BioConductor 2.12), as well as some comparisons with non-parallelized executions and a discussion of recommended patterns of usage.

2 Overview:

ADaCGH2 is a package for the analysis of CGH data. The main features of ADaCGH2 are:

- Parallelization of (several of) the main segmentation/calling algorithms currently available, to allow efficient usage of computing clusters. Parallelization can use either *forking* (in Unix-like OSs) or sockets, MPI, etc, as provided by package *snow* (`http://cran.r-project.org/web/packages/snow/index.html`).

  Forking will probably be the fastest approach in multicore machines, whereas MPI or sockets will be used with clusters made of several independent machines with few CPUs/cores each.

- Optional storage of, and access to, data using the *ff* package (`http://cran.r-project.org/web/packages/ff/index.html`), making it possible to analyze data from very large projects and/or use machines with limited memory.
- Parallelization and *ff* can be used simultaneously. WaviCGH Carro et al. (2010) (`http://wavi.bioinfo.cnio.es`), a web-server application for the analysis and visualization of array-CGH data that uses ADaCGH2, consitutes a clear demonstration of the usage of *ff* on a computing cluster with shared storage over NFS.

ADaCGH2 is a major re-write of our former package ADaCGH Diaz-Uriarte and Rueda (2007) and version 2 of ADaCGH2 is, itself, a major rewrite of the version 1.x series. Over time, we have improved the parallelization and, specially, changed completely the data handling routines. The first major rewrite of ADaCGH2 included the usage of the *ff* package, which allows ADaCGH2 to analyze data sets of more than four million probes in machines with no more than 2 GB of RAM. The second major rewrite reimplemented all the reading routines, and much of the analysis, which now allow a wider range of options with increased speed and decreased memory usage, and also allows users to disable the usage of *ff*. Moreover, in the new version, a large part of the reading is parallelized and makes use of temporary *ff* objects and we allow parallelization of analysis (and data reading) using forking. Further details and comparisons between the old and new versions are provided in the document "benchmarks.pdf", included with this package.

2.1 Terminology

The following is the meaning of some terms we will use repeatedly.

***ff* object** An object that uses the *ff* package. A tiny part of that object lives in memory, in the R session, but most of the object is stored on the hard drive. The part that lives in memory is just a pointer to the object that resides in the hard drive.

RAM objects The "usual" R objects (in our case, mainly data frames and matrices); these are stored, or live, in memory.

Somewhat similar to what the documentation of the `ff` package does, we refer to these objects, that reside in memory, as RAM objects. Technically, a given data frame, for instance, need not be in RAM in a particular moment (that actual memory page might have been swapped to disk). Regardless, the object is accessed as any other object which resides in memory. Likewise, note that *ff* also have a small part that is in memory, but the data themselves are stored on disk.

forking We copy literally from the vignette of the *parallel* package R Core Team (2013): "Fork is a concept from POSIX operating systems, and should be available on all R platforms except Windows. This creates a new R process by taking a complete copy of the master process, including the workspace and state of the random-number stream. However, the copy will (in any reasonable OS) share memory pages with the master until modified so forking is very fast."

Forking is, thus, a reasonable way of parallelizing jobs in multicore computers. Note, however, that this will not work **across** machines (for instance, across workstations in clusters of workstations).

cluster We use it here to contrast it with *forking*. With *cluster*, tasks are sent to other R processes using, for instance, MPI or any of the other methods provided by package **snow** (e.g., PVM, sockets, or NWS).

For example, MPI (for "Message Passing Interface") is a standardized system for parallel computing, probably the most widely used approach for parallelization with distributed

memory machines (such as in clusters of workstations). The package **Rmpi** (and **snow** on top of **Rmpi**) use MPI. In the examples in this vignette, however, we will use clusters of type *socket*, as these are available in several OSs (including Windows), and do not require installation of MPI.

If we are running Linux, Unix, or other POSIX operating systems, in a single computer with multiple cores we can use both forking and clusters (e.g., MPI or sockets). In most cases forking will be preferable as we will avoid some communication overheads and it will also probably use less total memory. If we are running Windows, however, we will need to use a cluster even in a single multicore machine.

2.2 Suggested usage patterns summary

The following table provides a simple guide of suggested usage patterns with small to moderate data sets:

	Lots of RAM	Little RAM
Single node, many cores/node	RAM objects (?), forking *ff* objects (?), forking	*ff* objects, forking
Many nodes, few cores/node	*ff* objects, cluster	*ff* objects, cluster

The question marks denote not-so-obvious choices, where the best decision will depend on the actual details of number of nodes, size of data sets, speed of communication between nodes, etc. For large data sets, the recommended usage involves always using *ff* objects. Using *ff* objects is slightly more cumbersome, but can allow us to analyze very large data sets in moderate hardware and will often result in faster computation; see details and discussion in "benchmarks.pdf". Of course, what is "lots", "many", and "large", will depend on the arrays you analyze and the hardware.

The examples below cover all three possible usage patterns:

RAM objects, forking : section 4.

ff objects, cluster : section 5.

ff objects, forking : section 6.

2.3 Usage: main steps and choices

ADaCGH2 includes functions that use as input, or produce as output, either *ff* objects or RAM R objects. Some functions also allow you to choose between using forking and using other mechanisms for parallelization.

For both interactive and non-interactive executions we will often execute the following in sequence:

1. Check the original data and convert to appropriate objects (e.g., to *ff* objects).
2. Initialize the computing cluster if not using forking.
3. Carry out segmentation and calling
4. Plot the results

We cover each in turn in the remaining of this section and discuss alternative routes. But first, we discuss why we might want to use ADaCGH2 instead of just "doing it manually on our own".

3 The data for all the examples

We will use a small, fictitious data set for all the examples, with six arrays/subjects and five chromosomes.

The data are available as an RData file

```
> library(ADaCGH2)
> data(inputEx)
> summary(inputEx)
```

```
         ID        chromosome       position              L.1
 Hs.101850:  1   Min.   :1.000   Min.   :  1180411   Min.   :-1.07800
 Hs.1019  :  1   1st Qu.:1.000   1st Qu.: 36030889   1st Qu.:-0.22583
 Hs.105460:  1   Median :2.000   Median : 70805790   Median :-0.01600
 Hs.105656:  1   Mean   :2.284   Mean   : 92600349   Mean   :-0.03548
 Hs.105941:  1   3rd Qu.:3.000   3rd Qu.:149843856   3rd Qu.: 0.16000
 Hs.106674:  1   Max.   :5.000   Max.   :243795357   Max.   : 0.88300
 (Other)  :494                                       NA's   :5
      L.2                  m4                 m5                  L3
 Min.   :-0.795000   Min.   :-0.1867   Min.   :-4.67275   Min.   :-13.273
 1st Qu.:-0.139000   1st Qu.: 1.9790   1st Qu.:-0.02025   1st Qu.:  3.631
 Median :-0.006000   Median : 2.2807   Median : 0.43725   Median :  3.925
 Mean   : 0.007684   Mean   : 3.4504   Mean   : 1.60159   Mean   :  1.981
 3rd Qu.: 0.134000   3rd Qu.: 5.8235   3rd Qu.: 3.04475   3rd Qu.:  4.110
 Max.   : 1.076000   Max.   : 6.6043   Max.   : 9.60425   Max.   :  6.374
 NA's   :15                            NA's   :41         NA's   :9
       m6
 Min.   :-0.7655
 1st Qu.:-0.2260
 Median :-0.0440
 Mean   :-0.0351
 3rd Qu.: 0.1620
 Max.   : 0.7750
 NA's   :203
```

```
> head(inputEx)
```

```
                             ID chromosome position   L.1    L.2      m4
1*1180411*Hs.212680   Hs.212680          1  1180411    NA  0.038 6.22625
1*1188041.5*Hs.129780 Hs.129780          1  1188042    NA  0.028 6.17425
1*1194444*Hs.42806     Hs.42806          1  1194444    NA  0.042 6.17425
1*1332537*Hs.76239     Hs.76239          1  1332537    NA  0.285 5.62425
1*2362211*Hs.40500     Hs.40500          1  2362211    NA  0.058 5.85125
1*2372287*Hs.449936   Hs.449936          1  2372287 0.294 -0.006 5.68525
                            m5    L3 m6
1*1180411*Hs.212680   3.22625 6.038 NA
1*1188041.5*Hs.129780 3.17425 6.028 NA
1*1194444*Hs.42806    3.17425 6.042 NA
1*1332537*Hs.76239    2.62425    NA NA
1*2362211*Hs.40500    2.85125    NA NA
1*2372287*Hs.449936   2.68525    NA NA
```

The data are are also available (in the /data subdirectory of the package) as an ASCII text file in two formats: with columns separated by tabs and with columns separated by spaces[1].

4 Example 1: RAM objects and forking

This is the simplest procedure if you are not under Windows. It will work when data is small (relative to available RAM) and the number of cores/processors in the single computing node is large relative to the number of subjects. However, this will not provide any parallelism under Windows: we use forking, as provided by the `mclapply` function in package `parallel`, and forking is available for POSIX operating systems (and Windows is not one of those).

Using forking can be a good idea because, with fork, creating new process is very fast and lightweight, and all the child process share memory pages until they start modifying the objects, and you do not need to explicitly send those pre-existing objects to the child processes. In contrasts, if we use other types of clusters (e.g., sockets or MPI), we need to make sure packages and R objects are explicitly sent to the child or slave processes.

If you have lots of RAM (ideally all you would need is enough memory to hold one copy of your original CGH data plus the return object), you will also probably use RAM objects and not *ff* objects, as these are less cumbersome to deal with than *ff* objects. But see details in file "benchmarks.pdf".

The steps for the analysis are:

- Read the input data.
- Carry out the segmentation.

4.1 Reading data and storing as a RAM object (a "usual" R object)

We provide here details on reading data from several different sources. Of course, in any specific case, you only need to use one route.

4.1.1 Data available as a data frame in an RData file

As we said in section 3, the data are available as an R data frame (`inputEx`), which we have saved as an RData file (`inputEx.RData`).

We will use `inputToADaCGH` to produce the three objects needed later for the segmentation, and to carry out some checks for missing values, repeated identifiers and positions, etc.

```
> fnameRdata <- list.files(path = system.file("data", package = "ADaCGH2"),
+                          full.names = TRUE, pattern = "inputEx.RData")
> inputToADaCGH(ff.or.RAM = "RAM",
+                           RDatafilename = fnameRdata)

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, data.frame version

   ... done with checks; starting writing
```

[1]These two files are used in the example of the help for the `cutFile` function

```
   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells 2622213 140.1    3886542 207.6  3205452 171.2
Vcells 2789966  21.3    4701432  35.9  3844289  29.4

 Saved objects with names
 cgh.dat chrom.dat pos.dat probenames.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .
```

We need to provide the path to the RData file, which we stored in the object `fnameRData`. This RData file will contain a single data frame. In this data frame, the first three columns of the data frame are the IDs of the probes, the chromosome number, and the position, and all remaining columns contain the data for the arrays, one column per array. The names of the first three column do not matter, but the order does. Names of the remaining columns will be used if existing; otherwise, fake array names will be created.

Note the usage of `ff.or.RAM = "RAM"`, which is different from that in section 5.2. The output from the call will leave several R objects in the global environment. The name of the objects can be changed with the argument `robjnames`. These are your usual R objects (data frames and vectors); thus, they are RAM objects.

4.1.2 Data available as an R data frame

Instead of accessing the RData file, we will directly use the data frame. This way, we use `inputToADaCGH` basically for its checks. The first three columns of the data frame are the IDs of the probes, the chromosome number, and the position, and all remaining columns contain the data for the arrays, one column per array.

```
> data(inputEx) ## make inputEx available as a data frame with that name
> inputToADaCGH(ff.or.RAM = "RAM",
+                       dataframe = inputEx)

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, data.frame version

   ... done with checks; starting writing

   ... done writing/saving probeNames
```

```
   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells 2622196 140.1    3886542 207.6  3205452 171.2
Vcells 2786389  21.3    4701432  35.9  3844289  29.4

 Saved objects with names
 cgh.dat chrom.dat pos.dat probenames.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .
```

Skipping the call to `inputToADaCGH` Since our data are already available as an R data frame, and if we are not interested in the checks provided by `inputToADaCGH`, we do not need to call it. To prepare the data for later usage with `pSegment` we can just do as follows:

```
> data(inputEx)
> cgh.dat <- inputEx[, -c(1, 2, 3)]
> chrom.dat <- as.integer(inputEx[, 2])
> pos.dat <- inputEx[, 3]
```

4.1.3 Using input data from a text file

Our data can also be in a text file, with a format where the first three columns are ID, chromosome, and position, and the remaining columns are arrays[2]. `inputDataToADaCGH` allows this type of input and, inside, uses `read.table.ff`; this way, we can read a very large data set and store it as an *ff* object or a RAM object without exhausting the available RAM.

```
> fnametxt <- list.files(path = system.file("data", package = "ADaCGH2"),
+                           full.names = TRUE, pattern = "inputEx.txt")
> tmp <- inputToADaCGH(ff.or.RAM = "RAM",
+                              textfilename = fnametxt)

   ...  textfile reading: reading the ID column

   ...  textfile reading: reading the chrom column

   ...  textfile reading: (parallel) reading of remaining columns

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, ff version
```

[2]If they are not, utilities such as `awk`, `cut`, etc, might be used for this purpose.

```
   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells 2645595 141.3    3886542 207.6  3205452 171.2
Vcells 2831235  21.7    4701432  35.9  3844289  29.4

 Saved objects with names
 cgh.dat chrom.dat pos.dat probenames.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .
```

If you will be using a cluster created with `makeCluster` (see section 5.3) you will not want to use this options. You will need to create *ff* objects because, when using a cluster, and to minimize transferring data and possibly exhausting available RAM, we have written the code so that the slaves do not receive the data itself, but just pointers to the data (i.e., names of *ff* objects) that live in the disk.

Compressed text files The function `inputToADaCGH` will work with both compressed and uncompressed files. However, if you are working with a really large text file, if you start from a compressed file, you will have to add the time it takes to decompress the file; thus, you might want to decompress it, outside R, before you start all of your work if you plan on using this file repeatedly as input.

4.1.4 Using data from Limma or snapCGH

You can also use data from snapCGH and Limma. See section 7.1.

4.1.5 Reading data from a directory

Reading data from a directory is discussed in more detail in section 6.2.6, and it is the preferred approach when we have a lot of data. Since saving the results as a RAM object is not likely to be the way to go in such cases (we would exhaust available RAM), we do not discuss it here any further.

4.2 Carrying out segmentation and calling

Segmentation and calling are carried out with the `pSegment` functions. Here we show just one such example. Many more are available in the second vignette. Setting argument `typeParall` to `fork` is not needed (it is the default), but we set it here explicitly for clarity.

```
> help(pSegment)
```

```
> haar.RAM.fork <- pSegmentHaarSeg(cgh.dat, chrom.dat,
+                                  merging = "MAD",
+                                  typeParall = "fork")
```

Since the input are RAM objects, the output is also a RAM object (a regular R object, in this case a list).

```
> lapply(haar.RAM.fork, head)
```

```
$outSmoothed
         L.1       L.2       m4       m5       L3 m6
1         NA 0.0175353 5.939581 2.929741 6.055182 NA
2         NA 0.0175353 5.939581 2.929741 6.055182 NA
3         NA 0.0175353 5.939581 2.929741 6.055182 NA
4         NA 0.0175353 5.939581 2.929741       NA NA
5         NA 0.0175353 5.939581 2.929741       NA NA
6 0.05851487 0.0175353 5.939581 2.929741       NA NA

$outState
  L.1 L.2 m4 m5 L3 m6
1  NA   0  1  1  1 NA
2  NA   0  1  1  1 NA
3  NA   0  1  1  1 NA
4  NA   0  1  1 NA NA
5  NA   0  1  1 NA NA
6   0   0  1  1 NA NA
```

```
> summary(haar.RAM.fork[[1]])
```

```
      L.1                L.2                 m4               m5
 Min.   :-0.18305   Min.   :-0.080705   Min.   :0.9303   Min.   :-4.0270
 1st Qu.:-0.10712   1st Qu.:-0.004725   1st Qu.:2.0171   1st Qu.: 0.0738
 Median :-0.06615   Median : 0.017535   Median :2.1786   Median : 0.1857
 Mean   :-0.03548   Mean   : 0.007684   Mean   :3.4504   Mean   : 1.6016
 3rd Qu.: 0.05851   3rd Qu.: 0.017535   3rd Qu.:5.9396   3rd Qu.: 2.9014
 Max.   : 0.17439   Max.   : 0.056750   Max.   :5.9396   Max.   : 9.0388
 NA's   :5          NA's   :15                           NA's   :41
       L3                 m6
 Min.   :-12.960   Min.   :-0.20148
 1st Qu.:  3.919   1st Qu.:-0.09948
 Median :  3.995   Median :-0.04680
 Mean   :  1.981   Mean   :-0.03510
 3rd Qu.:  4.008   3rd Qu.: 0.06151
 Max.   :  6.055   Max.   : 0.17410
 NA's   :9         NA's   :203
```

4.3 Plotting the results

Plotting produces PNG files for easier sharing over the Internet. The plotting function takes as main arguments the names of the result from `pSegment` and the input objects to `pSegment` (we will later see, for instance in section 5.5, how to use results stored as *ff* objects). Setting argument `typeParall` to `fork` is not needed (it is the default), but we set it here explicitly for clarity.

```
> pChromPlot(haar.RAM.fork,
+            cghRDataName = cgh.dat,
+            chromRDataName = chrom.dat,
+            posRDataName = pos.dat,
+            probenamesRDataName = probenames.dat,
+            imgheight = 350,
+            typeParall = "fork")
```

5 Example 2: *ff* objects and cluster

This procedure should work even with relatively small amounts of RAM, and it will also work under Windows. However, using a cluster involves additional steps. For both interactive and non-interactive sessions we will often execute the following in sequence:

1. Check the original data and convert to appropriate objects (e.g., to *ff* objects).
2. Initialize the computing cluster.
3. Carry out segmentation and calling
4. Plot the results

Compared to section 4 we introduce here the following new major topics:

- Using *ff* objects.
- Setting up a cluster.

Note for Windows users: in this vignette, the code that uses `ff` objects has been disabled as it leads to random and difficult to reproduce problems with the automated testing procedure (from creating socket clusters to removing temporary directories). Therefore, all remaining code in this vignette is surrounded with `if(.Platform$OS.type != "windows") {some-code-here}`. This code, however, should work interactively.

5.1 Choosing a working directory

As we will use *ff* objects, we will read and write quite a few files to the hard drive. The easiest way to organize your work is to create a separate directory for each project. At the end of this example, we will remove this directory. All plot files and ff data will be stored in this new directory.

(Just in case, we check for the existence of the directory first. We also store the current working directory to return to it at the very end.)

```
> if(.Platform$OS.type != "windows") {
+
+ originalDir <- getwd()
+ ## make it explicit where we are
+ print(originalDir)
+ }

> if(.Platform$OS.type != "windows") {
+ if(!file.exists("ADaCGH2_vignette_tmp_dir"))
+   dir.create("ADaCGH2_vignette_tmp_dir")
+ setwd("ADaCGH2_vignette_tmp_dir")
+ }
```

It is **very important** to remember that the names of the *ff* objects that are exposed to the user are always the same (i.e., chromData.RData, posData.RData, cghData.RData, probeNames.RData). Therefore, successive invocations of `inputToADaCGH`, if they produce *ff* output (i.e., `ff.or.RAM = "ff"`) will overwrite this objects (and make them point to different binary *ff* files on disk). In this vignette, we keep reusing `inputToADaCGH`, but note that in all the cases we produce as output *ff* files (sections 5.2, 5.2.1, 5.2.2, 5.2.4, 6.2, 6.2.1, 6.2.2, 6.2.4), the data used as input are the same, so there is no problem here (although we will leave binary *ff* objects on disk without a corresponding *ff* RData object on the R session). In particular, note that when we show the usage of Limma and snapCGH objects as input (section 7.1), we are using RAM objects (not *ff* objects) as output, so there is no confusion.

5.2 Reading data and storing as *ff* objects

Converting the original data to *ff* objects can be done either before or after initializing the cluster (section 5.3), as it does not use the computing cluster. The purpose of this step is to write the *ff* files to disk, so they are available for the segmentation and ploting functions.

5.2.1 Data available as a data frame in an RData file

To allow the conversion to be carried out using data from previous sessions, the conversion takes as input the name of an RData that contains plain, "regular" R objects (which, if loaded, would be RAM objects).

```
> if(.Platform$OS.type != "windows") {
+ fnameRdata <- list.files(path = system.file("data", package = "ADaCGH2"),
+                          full.names = TRUE, pattern = "inputEx.RData")
+ inputToADaCGH(ff.or.RAM = "ff",
+                           RDatafilename = fnameRdata)
+ }
```

The first command is used in this example to find the complete path of the example data set. The actual call to the function is the second expression. Note that we used a path to an RData file, and do not just use a RAM object. If you are very short of RAM, you might want to do the conversion in a separate R process that exists once the conversion is done and returns all of the RAM it used to the operating system. This we cover next in section 5.2.2. An alternative approach to try to minimize RAM is available if our data are in a text file, as discussed in section 4.1.3.

5.2.2 Converting from RData to *ff* objects in a separate process

With large data sets, converting from RData to *ff* can be the single step that consumes the most RAM, since we need to load the original data into R. Even if, after the conversion to *ff*, we remove the original data and call `gc()`, R might not return all of the memory to the operating system, and this might be inconvenient in multiuser environments and/or long running processes.

We can try dealing with the above problems by executing the conversion to *ff* in a separate R process that is spawned exclusively just for the conversion. For instance, we could use the `mcparallel` function (from package **parallel**) and do:

```
> mcparallel(inputToADaCGH(ff.or.RAM = "ff",
+                                    RDatafilename = fnameRData),
+                                    silent = FALSE)
```

```
>   tableChromArray <- mccollect()
>   if(inherits(tableChromArray, "try-error")) {
+     stop("ERROR in input data conversion")
+   }
```

That way, the *ff* are produced and stored locally in the hard drive, but the R process where the original data was loaded (and the conversion to *ff* carried out) dies immediately after the conversion, freeing back the memory to the operating system.

5.2.3 Data available as an R data frame

Instead of accessing the RData file, we can directly use the data frame, as we did in section 4.1.2.

```
> if(.Platform$OS.type != "windows") {
+ data(inputEx) ## make inputEx available as a data frame with that name
+ inputToADaCGH(ff.or.RAM = "ff",
+                        dataframe = inputEx)
+ }
```

5.2.4 Using input data from a text file

As in 4.1.3, we can read from a text file. In this case, however, the output will be a set of *ff* objects.

```
> if(.Platform$OS.type != "windows") {
+ fnametxt <- list.files(path = system.file("data", package = "ADaCGH2"),
+                           full.names = TRUE, pattern = "inputEx.txt")
+
+ inputToADaCGH(ff.or.RAM = "ff",
+                        textfilename = fnametxt)
+ }
```

5.2.5 Using data from Limma or snapCGH

You can also use data from snapCGH and Limma. See section 7.1.

5.2.6 Reading data from a directory

See section 6.2.6 for further details. This option is the best option with very large data sets. The initial data reading will use forking and, once we have saved the objects as *ff* objects, we can apply all the subsequent analysis steps discussed in the rest of this section.

5.2.7 Moving a set of *ff* objects

This is not specific to ADaCGH2, but since this issue can come up frequently, we explain it here. The paths of the *ff* files are stored in the object. How can we move this R object with all the *ff* files? First, we save the R object and all the *ff* files:

```
ffsave(cghData, file = "savedcghData", rootpath = "./")
```

We then take the resulting RData object (possible a very large object), and load it in the new location, rerooting the path:

```
ffload(file = "pathtofile/savedcghData", rootpath = getwd())
```

5.3 Initializing the computing cluster

Cluster initialization uses the functions provided in `parallel`. In the example we will use a sockect cluster, since this is likely to run under a variety of operating systems and should not need any additional software. Note, however, that MPI can also be used (in fact, that is what we use in our servers). In this example we will use as many nodes as cores can be detected.

```
> if(.Platform$OS.type != "windows") {
+ number.of.nodes <- detectCores()
+ cl2 <- parallel::makeCluster(number.of.nodes, "PSOCK")
+ parallel::clusterSetRNGStream(cl2)
+ parallel::setDefaultCluster(cl2)
+ parallel::clusterEvalQ(NULL, library("ADaCGH2"))
+
+ wdir <- getwd()
+ parallel::clusterExport(NULL, "wdir")
+ parallel::clusterEvalQ(NULL, setwd(wdir))
+ }
```

The first two calls create a cluster and initialize the random number generator[3]. The third expression sets the cluster just created as the default cluster. This is important: to simplify function calls, we do not pass the cluster name around, but rather expect a default cluster to be set up. The fourth line makes the **ADaCGH2** package available in all the nodes of the cluster (notice we did not need to do this with forking, as the child processes shared memory with the parent).

The last three lines make sure the slave processes use the same directory as the master. Because we created the cluster after changing directories (section 5.1) this step is not really needed here. But we make it explicit so as to verify it works, and as a reminder that you will need to do this if you change directories AFTER creating the cluster. If you run on a multinode cluster, you must ensure that the same directory exists in all machines. (In this case, we are running on the localhost).

5.4 Carrying out segmentation and calling

Segmentation and calling are carried out with the `pSegment` functions. Here we show just one such example. Many more are available in the second vignette.

```
> help(pSegment)
```

```
> if(.Platform$OS.type != "windows") {
+ haar.ff.cluster <- pSegmentHaarSeg("cghData.RData",
+                                    "chromData.RData",
+                                    merging = "MAD",
+                                    typeParall = "cluster")
+ }
```

We can take a quick look at the output. We first open the *ff* objects (the output is a list of *ff* objects) and then call `summary` on the list that contains the results of the wavelet smoothing:

[3]We use the version from package **parallel**, instead of the one from **BiocGenerics**, as the last one is still experimental.

```
> if(.Platform$OS.type != "windows") {
+ lapply(haar.ff.cluster, open)
+ summary(haar.ff.cluster[[1]][,])
+ }
```

5.5 Plotting the results

The call here is the same as in section 4.3, except that we change the values for the arguments. As we are using *ff* objects, we also need to first write to disk the (*ff*) object with the results.

```
> if(.Platform$OS.type != "windows") {
+ save(haar.ff.cluster, file = "hs_mad.out.RData", compress = FALSE)
+
+ pChromPlot(outRDataName = "hs_mad.out.RData",
+             cghRDataName = "cghData.RData",
+             chromRDataName = "chromData.RData",
+             posRDataName = "posData.RData",
+             probenamesRDataName = "probeNames.RData",
+             imgheight = 350,
+             typeParall = "cluster")
+ }
```

Finally, we stop the workers and close the cluster

```
> if(.Platform$OS.type != "windows") {
+ parallel::stopCluster(cl2)
+ }
```

6 Example 3: *ff* objects and forking

This example uses *ff* objects, as in section 5, but it will not use a cluster but forking, as in section 4. Therefore, we will not need to create a cluster, but we will need to read data and convert it to *ff* objects.

Here we introduce no new major topics. Working with *ff* objects was covered in section 5.2 and forking was covered in section 4.2. We simply combine these work-flows.

6.1 Choosing a working directory

As we will use *ff* objects, it will be convenient, as we did in section 5.1, to create a separate directory for each project, to store all plot files and ff data. Since we already did that above (section 5.1) we do not repeat it here. However, for real work, you might want to keep different analyses associated to different working directories.

6.2 Reading data and storing as *ff* objects

We have here the same options as in section 5.2. We repeat them briefly. A key difference with respect to section 5.2 is that we are not creating a cluster, so there will be no need to export the current working directory to slave processes explictly (in contrast to 5.3).

6.2.1 Data available as a data frame in an RData file

```
> if(.Platform$OS.type != "windows") {
+ fnameRdata <- list.files(path = system.file("data", package = "ADaCGH2"),
```

```
+                       full.names = TRUE, pattern = "inputEx.RData")
+ inputToADaCGH(ff.or.RAM = "ff",
+                        RDatafilename = fnameRdata)
+ }
```

6.2.2 Converting from RData to *ff* objects in a separate process

Even if we are using forking, we might still want to carry the conversion to *ff* objects in a separate process, as we did in section 5.2.2, since the conversion to *ff* objects might be the step that consumes most RAM in the whole process and we might want to make sure we return that memory to the operating system as soon as possible.

```
> mcparallel(inputToADaCGH(ff.or.RAM = "ff",
+                                  RDatafilename = fnameRdata),
+            silent = FALSE)
> tableChromArray <- collect()
> if(inherits(tableChromArray, "try-error")) {
+   stop("ERROR in input data conversion")
+ }
```

6.2.3 Data available as an R data frame

Instead of accessing the RData file, we can directly use the data frame, as we did in section 5.2.3.

6.2.4 Using input data from a text file

```
> if(.Platform$OS.type != "windows") {
+ fnametxt <- list.files(path = system.file("data", package = "ADaCGH2"),
+                           full.names = TRUE, pattern = "inputEx.txt")
+
+ inputToADaCGH(ff.or.RAM = "ff",
+                        textfilename = fnametxt)
+ }
```

6.2.5 Using data from Limma or snapCGH

You can also use data from snapCGH and Limma. See section 7.1.

6.2.6 Reading data from a directory

This is probably the best option for very large input data. We will read **all** the files in a given directory (except for those you might explicitly specify not to). Even if your original file follows the format of the data file in 6.2.4, you might want to convert it to the format used here (where each column is a file) as the time it takes to convert the file will be more than compensated by the speed ups of reading, in R, each file on its own. With very large files, it is much faster to read the data this way (we avoid having to loop many times over the file to read each column). Reading the data is parallelized, which allows us to speed up the reading process significantly (the parallelization uses forking, and thus you will see no speed gains in Windows). Finally, to maximize speed and minimize memory consumption, we use *ff* objects for intermediate storage.

6.2.7 Cutting the original file into one-column files

We provide a simple function, `cutFile`, to do this job. Here we create a directory where we will place the one-column files (we first check that the directory does not exist[4]). Note that this will probably NOT work under Windows[5] , and thus we skip using `cutFile` under Windows, and use a directory where we have stored the files split by column.

```
> if( (.Platform$OS.type == "unix") && (Sys.info()['sysname'] != "Darwin") ) {
+   fnametxt <- list.files(path = system.file("data", package = "ADaCGH2"),
+                          full.names = TRUE, pattern = "inputEx.txt")
+   if(file.exists("cuttedFile")) {
+     stop("The cuttedFile directory already exists. ",
+          "Did you run this vignette from this directory before? ",
+          "You will not want to do that, unless you modify the arguments ",
+          "to inputToADaCGH below")
+   } else dir.create("cuttedFile")
+   setwd("cuttedFile")
+   cutFile(fnametxt, 1, 2, 3, sep = "\t")
+   cuttedFile.dir <- getwd()
+   setwd("../")
+ } else {
+   cuttedFile.dir <- system.file("example-datadir",
+                                 package = "ADaCGH2")
+ }
```

We create a new directory and carry out the file cutting there since the upper level directory is already populated with other files we have been creating. If we cut the file in the upper directory, we would later need to specify a lengthy list of files to exclude in the arguments to `inputToADaCGH`. To avoid that, we create a directory, and leave the files in the newly created directory. After cutting, we return to the former level directory, to keep that directory with only the files for input.

It is important to realize that the previous paragraph, which might seem a mess, does not reflect the way you would usually work, which would actually be much simpler, and something like the following:

1. Create a directory for your new project (lets call this directory `d1`).
2. Copy the text file with your big txt file with data to `d1`; lets call this file `afile.txt`.
3. In R, move to `d1` (for example, `setwd(" /d1")`).
4. Use `cutFile`: `cutFile("afile.txt", 1, 2, 3)`.
5. Call `inputToADaCGH`: `inputToADaCGH(ff.or.RAM = "ff", path = getwd(), excludefile = "afile.txt")`

(In this vignette the work flow was not as easy because we are running lots of different examples, with several different work flows.)

`cutFile` will run several jobs in parallel to speed up the cutting process, launching by default as many jobs as cores it can detect, and will produce files with the required naming conventions of `inputToADaCGH`. Note that `cutFile` is unlikely to work under Windows.

[4]If it exists and contains files, `inputToADaCGH` will probably fail, as it is set to read all the files in the directory.

[5]Under Macs it might or might not work; in all of the Macs we have tried it, it works, but not on the testing machine at BioC.

If you do not want to use `cutFile` you can use utilities provided by your operating system. The following is a very simple example of using `cut` under bash (which is not unlike what we do internally in `cutFile`) to produce one-column files from a file called `Data.txt`, with 77 arrays/subjects, where cutting the data part is parallelized over four processors:

```
cut -f1 Data.txt > ID.txt
cut -f2 Data.txt > Chrom.txt
cut -f3 Data.txt > Pos.txt

for i in {4..20}; do cut -f$i Data.txt > col_$i.txt; done &
for i in {21..40}; do cut -f$i Data.txt > col_$i.txt; done &
for i in {41..60}; do cut -f$i Data.txt > col_$i.txt; done &
for i in {61..80}; do cut -f$i Data.txt > col_$i.txt; done &
```

After you have cut the file, each file contains one column of data. Three of the files must be named `"ID.txt"`, `"Chrom.txt"`, and `"Pos.txt"`. The rest of the files contain the data for each one of the arrays or subjects. The name of the rest of the files is irrelevant.

When using `inputDataToADaCGH` with a directory, the output can be either *ff* objects or RAM objects. However, the latter will rarely make sense (it will be slower and we can run into memory contraints); see the discussion in file "benchmarks.pdf".

```
> if(.Platform$OS.type != "windows") {
+ inputToADaCGH(ff.or.RAM = "ff",
+                path = cuttedFile.dir,
+                verbose = TRUE)
+ }
```

We have used the previously cut files in this example. You can also check the files that live under the "example-datadir" directory and you will see six files with names starting with "col", which are the data files, and the files `"ID.txt"`, `"Chrom.txt"`, and `"Pos.txt"`. (That is the directory we would use as input had we used Windows.)

Note that, to provide additional information on what we are doing we are calling the function with the (non-default) `verbose = TRUE`, which will list all the files we will be reading.

Beware of possible different orderings of files. When reading from a directory, and since each column is a file, the order of the columns (and, thus, subjects or arrays) in the data files that will be created can vary. In particular, the command `list.files` (which we use to list of the files) can produce different output (different order of files) between operating systems and versions of R. What this means is that, say, column three does not necessarily refer to the same subject or array. **Always use the column names to identify unambiguously the data and the results**.

What about performing this step in a separate process? In sections 5.2.2 and 6.2.2 we performed the data preparation in a separate process, to free up RAM to the OS right after the conversion. You can do that too here if you want, but we have not found that necessary, since the memory consumption when reading column by column is often small. See examples with large data sets in section **??**.

6.3 Carrying out segmentation and calling

The call is similar to the one in 5.4, except for the argument `typeParall`.

```
> if(.Platform$OS.type != "windows") {
+ haar.ff.fork <- pSegmentHaarSeg("cghData.RData",
+                                 "chromData.RData",
+                                  merging = "MAD",
+                                 typeParall = "fork")
+ }
```

6.4 Plotting the results

The call here is the same as in section 5.5, except for argument `typeParall`.

```
> if(.Platform$OS.type != "windows") {
+ save(haar.ff.fork, file = "haar.ff.fork.RData", compress = FALSE)
+
+ pChromPlot(outRDataName = "haar.ff.fork.RData",
+             cghRDataName = "cghData.RData",
+             chromRDataName = "chromData.RData",
+             posRDataName = "posData.RData",
+             probenamesRDataName = "probeNames.RData",
+             imgheight = 350,
+             typeParall = "fork")
+ }
```

7 Input and output to/from other packages

7.1 Input data from Limma and snapCGH

Many aCGH studies use pre-processing steps similar to those of gene expression data. The `MAList` object, from *Limma* and `SegList` object, from *snapCGH*, are commonly used to store aCGH information. The following examples illustrate the usage of the function `inputToADaCGH` to convert `MAList` and `SegList` data into a format suitable for *ADaCGH2*.

We will start with objects produced by *snapCGH*. The following code is copied from the *snapCGH* vignette (pp. 2 and 3). Please check the original vignette for details. In summary, a set of array files are read, the data are normalized and, finally, averaged over clones. snapCGH uses limma for the initial import of data and, next, with the `read.clonesinfo` function adds additional information such as chromosome and position. The `MA` object created is of class `MAList`, but with added information (compared to a basic, original, limma MAList object). `MA2` is of type `SegList`.

```
> if(.Platform$OS.type != "windows") {
+ require("limma")
+ require("snapCGH")
+ datadir <- system.file("testdata", package = "snapCGH")
+ targets <- readTargets("targets.txt", path = datadir)
+ RG1 <- read.maimages(targets$FileName, path = datadir, source = "genepix")
+
+
+ ## This is snapCGH-specific
+ RG1 <- read.clonesinfo("cloneinfo.txt", RG1, path = datadir)
+ RG1$printer <- getLayout(RG1$genes)
+ types <- readSpotTypes("SpotTypes.txt", path = datadir)
+ RG1$genes$Status <- controlStatus(types, RG1)
+
```

```
+ RG1$design <- c(-1, -1)
+
+
+ RG2 <- backgroundCorrect(RG1, method = "minimum") ## class RGList
+ MA <- normalizeWithinArrays(RG2, method = "median") ## class MAList
+ class(MA)
+ ## now obtain an object of class SegList
+ MA2 <- processCGH(MA, method.of.averaging = mean, ID = "ID")
+ class(MA2)
+ }
```

All the information (intensity ratios and location) is available in the `MA` and `MA2` objects. We can directly convert them to *ADaCGH2* objects (we set `na.omit = TRUE` as the data contain missing values). The first call process the `MAList` and the second the `SegList`.

In this section, we use the argument `robjnames`, to produce as output a set of RAM objects with a different set of names from the default. (Note that we could also have produced *ff* files as output, using the option `ff.or.RAM = "ff"`).

```
> if(.Platform$OS.type != "windows") {
+ tmp <- inputToADaCGH(MAList = MA,
+                                robjnames = c("cgh-ma.dat", "chrom-ma.dat",
+                                              "pos-ma.dat", "probenames-ma.dat"))
+
+ tmp <- inputToADaCGH(MAList = MA2,
+                                robjnames = c("cgh-ma.dat", "chrom-ma.dat",
+                                              "pos-ma.dat", "probenames-ma.dat"),
+                                minNumPerChrom = 4)
+ }
```

We need to change the argument to `minNumPerChrom` because, after the data processing step in `processCGH`, chromosome 21 has only four observations.

The original `MAList` as produced directly from *limma* do not have chromosome and position information. That is what the `read.clonesinfo` function from *snapCGH* did. To allow using objects directly from *limma* and incorporating position information, we will use an approach to directly mimicks that in *snapCGH*. If you use and `MAList` you can also provide a `cloneinfo` argument; this can be either the full path to a file with the format required by `read.clonesinfo` or, else, the name of an object with (at least) three columns, names `ID`, `Chr`, and `Position`.

We copy from the limma vignette (section 3.2, p.8), changing the names of objets by appending ".limma".

```
> if(.Platform$OS.type != "windows") {
+ targets.limma <- readTargets("targets.txt", path = datadir)
+ RG.limma <- read.maimages(targets.limma, path = datadir,
+                           source="genepix")
+ RG.limma <- backgroundCorrect(RG.limma, method="normexp",
+                               offset=50)
+ MA.limma <- normalizeWithinArrays(RG.limma)
+ }
```

We can add the chromosomal and position information in two different ways. First, as in `read.clonesinfo` or, else, we can provide the name of a file (with the same format as required by `read.clonesinfo`). Note that `fclone` is a path (and, thus, a character vector).

```
> if(.Platform$OS.type != "windows") {
+ fclone <- list.files(path = system.file("testdata", package = "snapCGH"),
+                      full.names = TRUE, pattern = "cloneinfo.txt")
+ fclone
+ tmp <- inputToADaCGH(MAList = MA.limma,
+                              cloneinfo = fclone,
+                              robjnames = c("cgh-ma.dat", "chrom-ma.dat",
+                                            "pos-ma.dat", "probenames-ma.dat"))
+ }
```

Alternatively, we can provide the name of an object with the additional information. For illustrative purposes, we can use here the columns of the `MA` object.

```
> if(.Platform$OS.type != "windows") {
+ acloneinfo <- MA$genes
+ tmp <- inputToADaCGH(MAList = MA.limma,
+                              cloneinfo = acloneinfo,
+                              robjnames = c("cgh-ma.dat", "chrom-ma.dat",
+                                            "pos-ma.dat", "probenames-ma.dat"))
+ }
```

7.2 Using CGHregions

The CGHregions package Vosse and van de Wiel (2009) is a BioConductor package that implements a well known method van de Wiel and van Wieringen (2007) for dimension reduction for aCGH data (see a review of common regions issues and methods in Rueda and Diaz-Uriarte (2010)).

The `CGHregions` function accepts different type of input, among others a data frame. The function `outputToCGHregions` produces that data frame, ready to be used as input to CGHregions (for the next example, you will need to have the *CGHregions* package installed).

Note: **it is up to you to deal with missing values!!!** In the example below, we do a simple `na.omit`, but note that we are now working with data frames. Extending the usage of this, and other methods, to much larger data sets, using *ff*, and properly dealing with missing values, is beyond the scope of this package.

```
> if(.Platform$OS.type != "windows") {
+ forcghr <- outputToCGHregions(haar.ff.cluster)
+ if(require(CGHregions)) {
+   regions1 <- CGHregions(na.omit(forcghr))
+   regions1
+ }
+ }
```

Please note that `outputToCGHregions` does NOT check if the calls are something that can be meaningfully passed to CGHregions. In particular, you probably do NOT want to use this function when `pSegment` has been called using `merging = "none"`.

```
> if(.Platform$OS.type != "windows") {
+ ## We are done with the executable code in the vignette.
+ ## Restore the directory
+ setwd(originalDir)
+ print(getwd())
+ }
```

```
> if(.Platform$OS.type != "windows") {
+ ## Remove the tmp dir. Sys.sleep to prevent Windoze problems.
+ ## Sys.sleep(1)
+ ## What is in that dir?
+ dir("ADaCGH2_vignette_tmp_dir")
+ unlink("ADaCGH2_vignette_tmp_dir", recursive = TRUE)
+ ## Sys.sleep(1)
+ }
```

8 Why ADaCGH2 instead of a "manual" solution

It is of course possible to parallelize the analysis (and figure creation) without using ADaCGH2. To deal with very large data, the key idea is to never try to load more data than we strictly need for an analysis (which is the strategy used by ADaCGH2).

To examine the simplest scenario, let us suppose we are already provided with single-column files (as, for instance, we obtain after using the helper function `cutFile` —see section 6.2.7); if we had a single large file, we would need to think of a way of reading only specific rows of a single column.

Now, we need to think how to parallelize the analysis. We will consider two cases: parallelizing by subject (or array or column) and parallelizing by subject*chromosome.

Let's first examine the simplest case: we will parallelize by subject, as is done by ADaCGH2 with HaarSeg and CBS (these methods are very fast, and further splitting by chromosome is rarely worth it). These are the required steps:

1. Each R process needs to have access to the chromosome information; this probably requires loading a vector with chromosome positions.
2. Each process will carry these steps until all the columns/subjects have been processed:
   (a) Read the data for a specific column (or subject).
   (b) Analyze (segment) those data.
   (c) Save the results to disk.
   (d) Remove from the workspace the results and the data (and probably call the garbage collector).
3. When all analysis are completed, assemble the results somehow to allow easy access to results.

The steps above need to deal with the following possible problems:

- We need to consider how to deal with missing values, since a simple removal of missing values case by case will result in a ragged array of results, which would probably not be acceptable.
- "loading" and "saving" can be time-consuming steps: the direct way in R would be to use functions such as `scan` (for reading) and `save` (for saving), but when done repeatedly, these are likely to be slower than using *ff* objects (e.g., using `scan` will be slower than acessing data from an *ff* object).
- Much more serious can be step 3 since we need to assemble a whole object with results. If the analysis involves many arrays and/or data sets with millions of probes, then we will not be able to load all of that in memory. (The approach we use in ADaCGH2

with the use of *ff* objects is to never reload all of the results to assemble the final object, but only assemble a set of pointers to data structures on disk).

Of course, an alternative is to leave the results as a large collection of files, and never try to assemble a single object with results. This, however, is likely much more cumbersome than having a single results object with all the information available that can be accessed as need (e.g., for further plotting).

Let us now examine the second scenario, where we parallelize by subject*chromosome. This is done, for instance, with HMM or BioHMM in ADaCGH2. Why? Because the methods are sufficiently slow that a finer grained division is likely to pay off in terms of gain in speed. In this case, additional partition and reassembly of the data are required for the segmentation and merging steps. These are the main steps:

1. Each R process needs to have access to the chromosome information.
2. Each process will carry these steps until all the columns/subjects have been processed:
   (a) Read the data for a specific set of positions (those that correspond to a specific chromosome) for a given column (or subject).
   (b) Analyze (segment) those data.
   (c) Save the results to disk.
   (d) Remove from the workspace the results and the data.
3. When all segmentation steps are completed, assemble the results by column/subject for the merging step.
4. For the merging step, each process will load the data for a complete column/subject and merge them with the corresponding algorithm:
   (a) Read the data for a column/subject.
   (b) Perform merging.
   (c) Save the results to disk.
   (d) Remove from the workspace the results and the data and do garbage collection.
5. When all analysis are completed, assemble the results somehow to allow easy access to results.

This process is, of course, more convoluted than when parallelizing only by subject. As above, we need to consider how to deal with missing values, the use of repeated "scan" and "save", and the much more serious problem of putting together the complete object with all of the results.

Finally, if we were interested in analyzing the data with more than one method, we would need to modify the code above since each method uses different ways of being called (e.g., some methods require setting up specific objects before segmentation can be called).

What ADaCGH2 provides is, among other things, a way to eliminate those steps, automating them for the user, with careful consideration of fast access to data on disk, and attempts to minimize memory usage in repeated calls to the same process (which we can do successfully, as can be seen from the benchmarks for large numbers of arrays with more than 6 million probes —memory usage levels out; see the file "benchmarks.pdf"). Of course, ADaCGH2 provides other benefits (e.g., facilities for using as input the data from other packages —e.g, section 7.1— or providing output for other packages —e.g., section 7.2).

9 Session info and packages used

This is the information about the version of R and packages used:

```
> sessionInfo()

R version 3.3.0 RC (2016-04-25 r70549)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows Server 2008 R2 x64 (build 7601) Service Pack 1

locale:
[1] LC_COLLATE=C
[2] LC_CTYPE=English_United States.1252
[3] LC_MONETARY=English_United States.1252
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.1252

attached base packages:
[1] parallel  stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
[1] ADaCGH2_2.12.0 GLAD_2.36.0    ff_2.2-13      bit_1.1-12

loaded via a namespace (and not attached):
 [1] Rcpp_0.12.4.5         BiocInstaller_1.22.0  RColorBrewer_1.1-2
 [4] plyr_1.8.3            tools_3.3.0           zlibbioc_1.18.0
 [7] RSQLite_1.0.0         annotate_1.50.0       preprocessCore_1.34.0
[10] gtable_0.2.0          lattice_0.20-33       snapCGH_1.42.0
[13] Matrix_1.2-6          fastmatch_1.0-4       DBI_0.4
[16] pixmap_0.4-11         cluster_2.0.4         genefilter_1.54.0
[19] IRanges_2.6.0         S4Vectors_0.10.0      multtest_2.28.0
[22] stats4_3.3.0          grid_3.3.0            Biobase_2.32.0
[25] ffbase_0.12.3         DNAcopy_1.46.0        AnnotationDbi_1.34.0
[28] survival_2.39-2       XML_3.98-1.4          waveslim_1.7.5
[31] limma_3.28.0          ggplot2_2.1.0         MASS_7.3-45
[34] splines_3.3.0         scales_0.4.0          BiocGenerics_0.18.0
[37] strucchange_1.5-1     colorspace_1.2-6      xtable_1.8-2
[40] aCGH_1.50.0           sandwich_2.3-4        affy_1.50.0
[43] tilingArray_1.50.0    munsell_0.4.3         vsn_3.40.0
[46] zoo_1.7-12            affyio_1.42.0
```

References

Carro, A., Rico, D., Rueda, O. M., Diaz-Uriarte, R., and Pisano, D. G. (2010). waviCGH: a web application for the analysis and visualization of genomic copy number alterations. *Nucleic acids research*, 38 Suppl:W182–7.

Diaz-Uriarte, R. and Rueda, O. M. (2007). ADaCGH: A parallelized web-based application and R package for the analysis of aCGH data. *PloS one*, 2(1):e737.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rueda, O. M. and Diaz-Uriarte, R. (2010). Finding Recurrent Copy Number Alteration Regions : A Review of Methods. *Current Bioinformatics*, 5:1–17.

van de Wiel, M. A. and van Wieringen, W. N. (2007). CGHregions: Dimension Reduction for Array CGH Data with Minimal Information Loss. *Cancer informatics*, 3(0):55–63.

Vosse, S. and van de Wiel, M. (2009). *CGHregions: Dimension Reduction for Array CGH Data with Minimal Information Loss*. R package version 1.7.1.