Vignette for the 'rearrvisr' package

Doro Lindtke

2019-05-02

Contents

This vignette describes the R package `rearrvisr`, which can be used to detect, classify, and visualize genome rearrangements along a focal genome. The package requires two input files: one that gives the positions of orthologous markers in an extant genome (the focal genome), and one that specifies the organization of orthologous markers in an ancestral genome reconstruction or a second extant genome (the compared genome).

This vignette provides a tutorial on how to use the package functions and interpret their output by applying them step by step to a *Drosophila* example data set. It also includes a detailed description of the implemented `rearrvisr` algorithm, and outlines the methods used to create the *Drosophila* example data set from 12 publicly available genome assemblies. See `?rearrvisr` for a brief overview of the functions of the package.

The *Drosophila* example data set used in this vignette (`MEL_markers`, `SIM_markers`, `YAK_markers`, and `MSSYE_PQTREE_HEUR`) was generated from 12 publicly available genome assemblies downloaded from Ensemble Release 91 (http://dec2017.archive.ensembl.org; *D. melanogaster*) or Ensemble Metazoa Release 37 (http://oct2017-metazoa.ensembl.org; other genomes), using steps and software as described below. An additional example data set (`TOY24_focalgenome` and `TOY24_compgenome`) was created for illustrative purposes. A short description of these example data can be accessed with the commands `?MEL_markers`, `?SIM_markers`, `?YAK_markers`, `?MSSYE_PQTREE_HEUR`, `?TOY24_focalgenome`, and `?TOY24_compgenome`.

1 Input files

Two input files are required: a representation of the focal genome in `focalgenome`, for which rearrangements will be identified, and a representation of the compared genome in `compgenome`, which serves as reference for the arrangement of markers within genome segments. Deviations in the order of focal markers from the order of reference markers indicate rearrangements. *Genome segments* are chromosomes, scaffolds, or contiguous sets of genetic markers. Genomes do not need to be assembled at chromosome-level, however the detection of rearrangements is inherently limited for assemblies that are highly fragmented. *Markers* denote orthologous genomic regions that are unique within each genome, such as genes or synteny blocks. Orthologous markers require the same ID in both input files, and need to be ordered by their absolute or relative position within their genome segments. Further details are below.

1.1 Focal genome

The focal genome representation in `focalgenome` is an extant genome in a standard linear (one-dimensional) genome map format that associates markers to genome segments and gives their map positions (e.g., in base pairs) within them. Genome segments of the focal genome are chromosomes or scaffolds and are referred to as *focal genome segments* or *focal segments*.

The focal genome map needs to be a data frame with the mandatory columns *marker* (orthologous marker ID; can be `NA` for markers that have no ortholog), *scaff* (name of the focal segment where the marker is located), *start* (absolute start position of the marker on its focal segment, for example in base pairs), *end* (absolute end position of the marker), and *strand* (the reading direction of the marker, either `"+"` or `"-"`). Additional columns are ignored and may store custom information, such as marker names. Markers need to be ordered by their map position within each focal segment, for example by running the `orderGenomeMap()` function. More information on the `focalgenome` file format is available from the `?checkInfile` command.

```
## Example for the focal genome map of Drosophila melanogaster
## (marker start and end positions are midpoints +/- 1 base pair)

head(MEL_markers)
#>   marker scaff  start    end strand            name
#> 1   1631     4   3194   3196      + MEL_FBpp0312297
#> 2    207     4  36917  36919      - MEL_FBpp0300616
```

```
#> 3    563     4  51751  51753      - MEL_FBpp0088245
#> 4  10611     4  66757  66759      - MEL_FBpp0088242
#> 5   2582     4  87156  87158      + MEL_FBpp0088228
#> 6    474     4 121621 121623      - MEL_FBpp0088240
```

Important: The column *scaff* must be a character vector, even when all focal segment names are numeric. This can be assured, for example, by using `focalgenome$scaff <- as.character(focalgenome$scaff)` after reading the input file.

1.1.1 The `orderGenomeMap()` function

The focal genome map in `focalgenome` needs to be ordered so that markers appear according to their map position within each focal segment. Focal segments can be in arbitrary order. The function `orderGenomeMap()` orders markers within each focal segment by their map position (i.e., the midpoint between positions given by the columns *start* and *end* in `focalgenome`). Focal segments can be ordered by user-defined names, in alphabetical order, or by their size. See `?orderGenomeMap` for details.

```
## Example for ordering the focal genome map of D. melanogaster

## make random order (for illustration only)
myorder <- sample(1:nrow(MEL_markers))
MEL_markers_unordered <- MEL_markers[myorder, ]
head(MEL_markers_unordered)
#>       marker scaff    start      end strand            name
#> 4573    5337    2R 18799257 18799259      - MEL_FBpp0085816
#> 7058   10177    3L 13045780 13045782      + MEL_FBpp0075620
#> 8746   15219    3R  8563017  8563019      + MEL_FBpp0081362
#> 9403    2903    3R 13421907 13421909      + MEL_FBpp0307899
#> 311    17689    2L  1992007  1992009      - MEL_FBpp0099943
#> 10413   3230    3R 22374351 22374353      + MEL_FBpp0083612

## order genome map by size of focal segments and marker position
MEL_markers_reordered <- orderGenomeMap(MEL_markers_unordered,
                                        ordnames = "all",
                                        sortby = "size")
head(MEL_markers_reordered)
#>      marker scaff   start     end strand            name
#> 8229    912    3R 1653327 1653329      + MEL_FBpp0352251
#> 8230     NA    3R 2969170 2969172      - MEL_FBpp0401450
#> 8231   9484    3R 3338383 3338385      + MEL_FBpp0289444
#> 8232   1832    3R 3535404 3535406      - MEL_FBpp0112608
#> 8233  14129    3R 3624573 3624575      + MEL_FBpp0291024
#> 8234  13818    3R 3738461 3738463      + MEL_FBpp0112618
```

1.1.2 The `checkInfile()` function

To verify that the format of the focal genome map is correct, the `checkInfile()` function can optionally be run. This function is also called internally from other functions of the package where `focalgenome` is used as input. The function will return an error message when a problem has been detected, or nothing otherwise. See `?checkInfile` for more information.

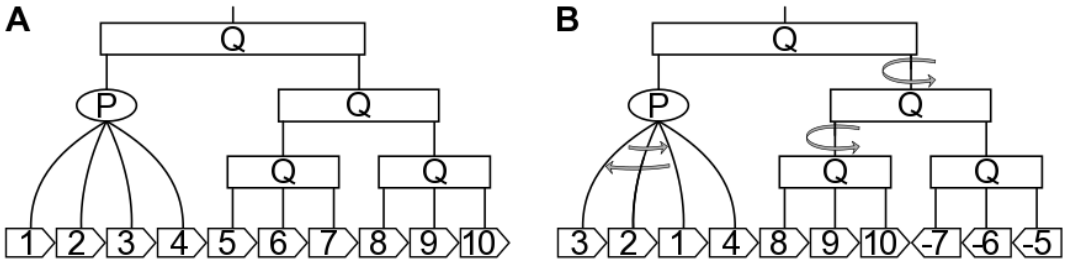

Figure 1: Example *PQ-tree*. *P-nodes* are illustrated with ovals and *Q-nodes* with rectangles. Leaves (i.e., markers) are illustrated with pentagons and include their marker ID. Children of *P-nodes* are in arbitrary order, and children of *Q-nodes* are in fixed order (including their reversal). The marker orders `1 2 3 4 5 6 7 8 9 10` in **A** and `3 2 1 4 8 9 10 -7 -6 -5` in **B** can be encoded by the same *PQ-tree*, as branches that origin from *P-nodes* can be transposed, and those that origin from *Q-nodes* can be reversed in their order (i.e., turning the *Q-node* around), as indicated by the gray arrows.

```
## Example for checking the focal genome map of D. melanogaster

checkInfile(MEL_markers, "focalgenome", checkorder = TRUE)
```

Note: If the *marker* column was assigned the class `numeric` rather than the required class `integer`, run for example the command `focalgenome <- type.convert(focalgenome, as.is = TRUE)` to fix the resulting error.

1.2 Compared genome

The compared genome representation in `compgenome` can be an ancestral genome reconstruction or an extant genome, and is organized into genome segments called *CARs* (Contiguous Ancestral Regions; following Ma *et al.*, 2006; Chauve and Tannier, 2008). For simplicity, compared genome segments are referred to as CARs throughout, but can equally be any contiguous sets of genetic markers of an extant genome. Each CAR is represented by a *PQ-tree* (described below; Booth and Lueker, 1976; Chauve and Tannier, 2008), which specifies the relative positions of markers within genome segments, incorporating ambiguity in marker order (i.e., all alternative orders).

The *PQ-tree format* joins all *PQ-trees* (i.e., CARs) of the compared genome representation in a single data frame `compgenome` (details below). For simplicity, the joined *PQ-trees* in the `compgenome` data frame are referred to as *PQ-structure* throughout, while individual *PQ-trees* are referred to as CARs or *PQ-trees*.

An ancestral genome representation in form of *PQ-trees*, as output by the software ANGES (Chauve and Tannier, 2008; Jones *et al.*, 2012), can be converted into a *PQ-structure* with the `convertPQtree()` function. A one-dimensional genome map, for example of an extant genome, can be converted into a two-dimensional *PQ-structure* with the `genome2PQtree()` function.

1.2.1 *PQ-trees*, *P-nodes*, and *Q-nodes*

A *PQ-tree* is a combinatorial structure consisting of *leaves*, *P-nodes*, and *Q-nodes* (Booth and Lueker, 1976; Chauve and Tannier, 2008; Figure 1). This structure allows the representation of ambiguity in marker order in an ancestral genome reconstruction (i.e., encoding all possible arrangements of markers). Each marker is represented by a *leaf*, which is connected to the root of the *PQ-tree* (i.e., a CAR) through a hierarchical

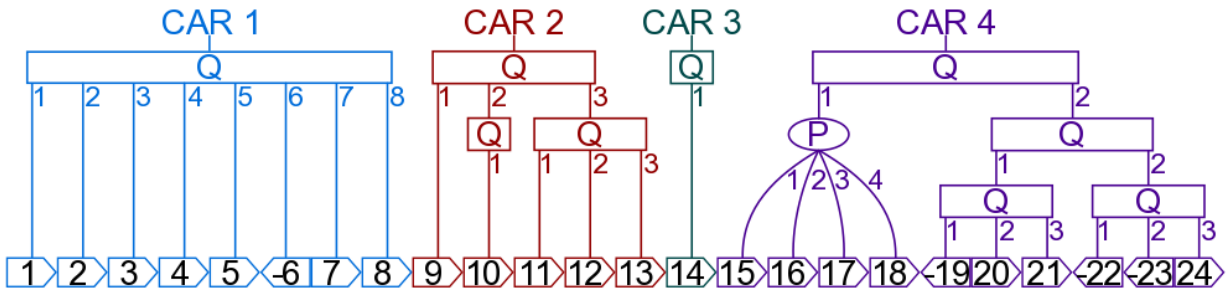

Figure 2: Graphical representation of the *PQ-structure* in the example data `TOY24_compgenome`. See text for details.

sequence of internal nodes (*P-nodes* or *Q-nodes*). The type of internal nodes and the root specify the order of their children (i.e., internal nodes or leaves at the subsequent hierarchy level): they are in arbitrary order for *P-nodes*, and in fixed order (including their reversal) for *Q-nodes*. The Figure 1 illustrates two alternative marker orders (among many) that can be encoded by a single *PQ-tree*.

1.2.2 *PQ-structure*

To facilitate the handling of *PQ-trees* in `R`, their hierarchical organization is encoded as a data frame `compgenome` with markers in rows and a description of the structure of *PQ-trees* in columns (the *PQ-tree format*, or *PQ-structure*). The *PQ-tree* in the Figure 2 illustrates graphically the tabular *PQ-structure* of the example data `TOY24_compgenome`. All branches that origin at a node are labeled with an integer ID (i.e., the *node element ID*) according to their position at that node, starting at `1`. When considered jointly through all hierarchy levels, these IDs provide the relative position of each marker within their *PQ-tree* (e.g., compare the branch labels in the Figure 2 with the entries in the rows of `TOY24_compgenome` for each marker below). The full compared genome can be represented by a single *PQ-tree* with a *P-node* at its root (i.e., CARs are positioned in arbitrary order to each other), which is encoded by integer IDs at the CAR level.

The data frame `compgenome` contains the mandatory columns *marker* (orthologous marker ID), *orientation* (the reading direction of the marker, either `"+"` or `"-"`), and *car* (ID of the CAR where the marker is located). Subsequent columns describe the internal structure of each CAR, where two columns are needed for each subordinate hierarchy level: the first column of each set gives the node type (either `"P"` or `"Q"`), and the second column the integer ID of the children of the node (i.e., the node element ID). `NA`s are permitted from the second subordinate hierarchy level onwards and are required to fill otherwise empty cells that arise when the preceding hierarchy level encodes leaves. Markers need to be ordered by their node element IDs within the *PQ-structure*. More information on the *PQ-tree format* is available from the `?checkInfile` command.

```
## Example for a compared genome representation in PQ-tree format
## ('TOY24_compgenome'; see Figure for a graphical representation)

TOY24_compgenome
#>    marker orientation car type1 elem1 type2 elem2 type3 elem3
#> 1       1           +   1     Q     1  <NA>    NA  <NA>    NA
#> 2       2           +   1     Q     2  <NA>    NA  <NA>    NA
#> 3       3           +   1     Q     3  <NA>    NA  <NA>    NA
#> 4       4           +   1     Q     4  <NA>    NA  <NA>    NA
#> 5       5           +   1     Q     5  <NA>    NA  <NA>    NA
#> 6       6           -   1     Q     6  <NA>    NA  <NA>    NA
```

```
#> 7       7          +   1     Q     7  <NA>    NA  <NA>    NA
#> 8       8          +   1     Q     8  <NA>    NA  <NA>    NA
#> 9       9          +   2     Q     1  <NA>    NA  <NA>    NA
#> 10     10          +   2     Q     2     Q     1  <NA>    NA
#> 11     11          +   2     Q     3     Q     1  <NA>    NA
#> 12     12          +   2     Q     3     Q     2  <NA>    NA
#> 13     13          +   2     Q     3     Q     3  <NA>    NA
#> 14     14          +   3     Q     1  <NA>    NA  <NA>    NA
#> 15     15          +   4     Q     1     P     1  <NA>    NA
#> 16     16          +   4     Q     1     P     2  <NA>    NA
#> 17     17          +   4     Q     1     P     3  <NA>    NA
#> 18     18          +   4     Q     1     P     4  <NA>    NA
#> 19     19          -   4     Q     2     Q     1     Q     1
#> 20     20          +   4     Q     2     Q     1     Q     2
#> 21     21          +   4     Q     2     Q     1     Q     3
#> 22     22          -   4     Q     2     Q     2     Q     1
#> 23     23          -   4     Q     2     Q     2     Q     2
#> 24     24          +   4     Q     2     Q     2     Q     3
```

1.2.3 The `convertPQtree()` function

The ancestral genome reconstruction that is generated by the software ANGES (Chauve and Tannier, 2008; Jones *et al.*, 2012) is encoded as a text file with the first row giving the name of the ancestor (preceded by `>`), followed by two rows for each CAR. The first row of each set gives the CAR ID (preceded by `#CAR`), and the second row provides the *PQ-tree* structure of that CAR. The children of each node in the *PQ-tree* are enclosed by `_P` and `P_` markups for *P-nodes*, or by `_Q` and `Q_` markups for *Q-nodes*. Markers (i.e., orthologs) that belong to a particular node are located between its corresponding markups. Markers with reversed orientation are preceded by a `-` sign. The opening (`_P` or `_Q`) and closing (`P_` or `Q_`) markups can be nested to allow the representation of the hierarchical structure of the *PQ-tree*. Such linearly encoded *PQ-trees* can be converted into a *PQ-structure* with the `convertPQtree()` function. See `?convertPQtree` for more information.

Note: To read a text file with an ancestral genome reconstruction from the software ANGES into `R`, the command `rawtree <- read.table("PATH/TO/PQTREE", sep = ",", comment.char = "", as.is = TRUE)` can be used. The `rawtree` table can then be converted into a *PQ-structure* with the command `compgenome <- convertPQtree(rawtree)`.

Important: The `convertPQtree()` function was designed to work with the ANGES `*PQTREE*` output file (e.g., the `MSSYE_PQTREE_HEUR` example file). It was not designed to work with the ancillary `*PQRTREE*` or the `*DOUBLED*` output files.

```
## Example for a linearly encoded PQ-tree for #CAR7 of the
## Drosophila ancestral genome 'MSSYE' computed with the
## software ANGES (Jones et al., 2012)
MSSYE_PQTREE_HEUR[which(MSSYE_PQTREE_HEUR[ ,1] == "#CAR7") + 1, ]
#> [1] "_Q 11698 11765 -15944 15826 _Q 15827 Q_ Q_ "

## Convert linearly encoded PQ-trees into PQ-structure
MSSYE_compgenome <- convertPQtree(MSSYE_PQTREE_HEUR)
#>
#> converting data for MSSYE
#> ... processed 20 CARs ...

## Show converted #CAR7
```

```
MSSYE_compgenome[MSSYE_compgenome$car == 7, ]
#>      marker orientation car type1 elem1 type2 elem2 type3 elem3
#> 8953  11698           +   7     Q     1  <NA>    NA  <NA>    NA
#> 8954  11765           +   7     Q     2  <NA>    NA  <NA>    NA
#> 8955  15944           -   7     Q     3  <NA>    NA  <NA>    NA
#> 8956  15826           +   7     Q     4  <NA>    NA  <NA>    NA
#> 8957  15827           +   7     Q     5     Q     1  <NA>    NA
```

Note: For the following examples it is assumed that the object `MSSYE_compgenome` has been created with the above command.

1.2.4 The `genome2PQtree()` function

Alternatively to a genome reconstruction in *PQ-tree format*, a one-dimensional genome map can be converted into a two-dimensional *PQ-structure*. This can be, for example, a genome map of an extant genome, or a genome map of an unambiguous genome reconstruction. The compared genome map has to be in the same format as the focal genome map (described above), and needs to be ordered by the map position of markers within each compared genome segment, for example by running the `orderGenomeMap()` function. An unambiguously ordered genome representation can be seen as a subclass of a *PQ-structure*, where each genome segment is encoded by a single *Q-node* that only contains leaves as children. Accordingly, the *PQ-structure* of a converted genome map has exactly five columns (i.e., *marker*, *orientation*, *car*, and two columns for node type and node element). The `genome2PQtree()` function performs such a conversion. See `?genome2PQtree` for details.

Important: Compared genome segments need to be contiguous sets of genetic markers. Genome segments that are (potentially) overlapping, such as minor scaffolds or contigs that were not assembled into chromosomes and might in fact be part of assembled chromosomes or enclosed in other scaffolds, need to be excluded.

```
## Example for converting the genome map of D. simulans into
## a PQ-structure

## Exclude potentially overlapping minor scaffolds from genome map
SIM_markers_chr <- SIM_markers[is.element(SIM_markers$scaff,
                                          c("2L", "2R", "3L", "3R", "4", "X")), ]

## Order genome map by specified chromosomes
SIM_markers_chr <- orderGenomeMap(SIM_markers_chr,
                                  c("2L", "2R", "3L", "3R", "4", "X"))

## Show original genome
head(SIM_markers_chr)
#>    marker scaff start   end strand            name
#> 68   1133    2L 19900 19902      - SIM_FBpp0221398
#> 69   6934    2L 26387 26389      - SIM_FBpp0221397
#> 70    109    2L 37177 37179      - SIM_FBpp0221396
#> 71   1470    2L 68885 68887      + SIM_FBpp0221399
#> 72   5211    2L 74739 74741      + SIM_FBpp0221400
#> 73   8187    2L 85887 85889      - SIM_FBpp0221395

## Convert genome map into PQ-structure
SIM_compgenome <- genome2PQtree(SIM_markers_chr)

## Show converted genome
head(SIM_compgenome)
```

```
#>   marker orientation car type1 elem1
#> 1   1133           -   1     Q     1
#> 2   6934           -   1     Q     2
#> 3    109           -   1     Q     3
#> 4   1470           +   1     Q     4
#> 5   5211           +   1     Q     5
#> 6   8187           -   1     Q     6

## Print a translation between chromosome names and CAR IDs
head(data.frame(chr = unique(SIM_markers_chr$scaff),
                car = 1:length(unique(SIM_markers_chr$scaff)),
                stringsAsFactors = FALSE))
#>   chr car
#> 1  2L   1
#> 2  2R   2
#> 3  3L   3
#> 4  3R   4
#> 5   4   5
#> 6   X   6
```

Note: CAR IDs will be assigned according to the order of compared genome segments in the genome map. Markers that are `NA` in the genome map will be excluded from the *PQ-structure*.

1.2.5 The `checkInfile()` function

To verify that the format of the compared genome representation is correct, the `checkInfile()` function can optionally be run. This function is also called internally from other functions of the package where `compgenome` is used as input. The function will return an error message when a problem has been detected, or nothing otherwise. See `?checkInfile` for more information.

```
## Example for checking the compared genome representation of the
## Drosophila ancestral genome 'MSSYE' that was converted into
## 'MSSYE_compgenome' above

checkInfile(MSSYE_compgenome, "compgenome", checkorder = TRUE)
```

Note: If the *marker* column was assigned the class `numeric` rather than the required class `integer`, run for example the command `compgenome <- type.convert(compgenome, as.is = TRUE)` to fix the resulting error.

2 Identifying rearrangements

The `computeRearrs()` function detects and classifies rearrangements with the `rearrvisr` algorithm. The `summarizeBlocks()` function summarizes the output of the `computeRearrs()` function for each synteny block.

2.1 The `computeRearrs()` function

The `computeRearrs()` function requires as input files an ordered map of the focal genome (`focalgenome`), for which rearrangements will be identified, and an ordered representation of the compared genome (`compgenome`), which serves as reference for the arrangement of markers within genome segments. In addition, it needs to be

specified whether markers in the ancestral genome reconstruction contain information about their orientation (`doubled`). If orientation information is not available (i.e., `doubled = FALSE`), all values in the *orientation* column of `compgenome` should be `"+"`. See `?computeRearrs` for more information on the input and output of the function. Further details are also provided in the description of the `rearrvisr` algorithm below.

```
## Example for identifying rearrangements in the D. melanogaster
## genome that occurred after divergence from the Drosophila
## ancestor 'MSSYE'

## identify rearrangements (may run a few seconds)
SYNT_MEL_MSSYE <- computeRearrs(MEL_markers, MSSYE_compgenome, doubled = TRUE)

## show names of the matrices in the output
names(SYNT_MEL_MSSYE)
#>  [1] "TLBS"     "TLWS"     "TLWC"     "IV"       "TLBSbS"   "TLBSbE"
#>  [7] "TLWSbS"   "TLWSbE"   "TLWCbS"   "TLWCbE"   "IVbS"     "IVbE"
#> [13] "nodeori"  "blockori" "blockid"  "premask"  "subnode"
```

Note: For the following examples it is assumed that the object `SYNT_MEL_MSSYE` has been created with the above command.

A basal step of the `rearrvisr` algorithm is the alignment of the compared genome representation in `compgenome` to the sorted genome map of the focal species in `focalgenome`. This can be reconstructed with the following command:

```
## Example for reconstructing the alignment between the genome of
## the Drosophila ancestor 'MSSYE' and the D. melanogaster genome

## make alignment
MEL_MSSYE<-merge(MEL_markers, MSSYE_compgenome, by = "marker",
                 sort = FALSE)

## maker IDs should be identical to the ones in SYNT_MEL_MSSYE
identical(as.character(MEL_MSSYE$marker),
          rownames(SYNT_MEL_MSSYE$TLBS))
#> [1] TRUE
```

Note: For the following examples it is assumed that the alignment `MEL_MSSYE` has been created with the above command.

Next steps: *The data returned by the `computeRearrs()` function can be visualized with the `genomeImagePlot()` function, or summarized and visualized with the `summarizeBlocks()` and `genomeRearrPlot()` functions.*

Output: Each matrix in the output stores data on detected rearrangements and additional information. Markers are in rows, and the row names of each matrix correspond to the IDs in the *marker* column of `focalgenome` and `compgenome`.

The matrices `$TLBS`, `$TLWS`, `$TLWC`, and `$IV` contain rearrangements of four different classes, and are briefly described below. Further details and information on the other matrices are provided in the function documentation and the description of the `rearrvisr` algorithm.

Rearrangements are classified as translocations between compared genome segments (i.e., CARs), analogous to interchromosomal rearrangements (`TLBS` and `TLWS`), and as translocations or inversions within compared genome segments, analogous to intrachromosomal rearrangements (`TLWC` and `IV`). `$TLBS` stores `TransLocations` between CARs `B`etween focal `S`egments; `$TLWS` stores `TransLocations` between CARs `W`ithin focal `S`egments;

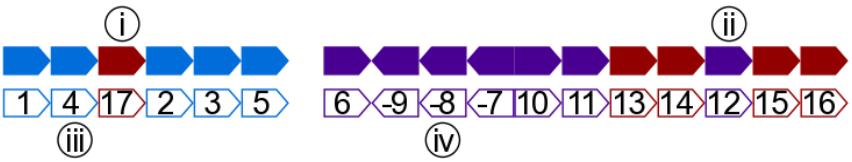

Figure 3: Four classes of rearrangements. Markers are located on two focal genome segments (left and right) and are represented by pentagons. Markers are arranged from left to right by their position in the focal genome. The top row shows markers colored according to their CAR of origin, and integers in the bottom row indicate the order of markers within the compared genome. **(i)**, a TransLocation between CARs Between focal Segments (`TLBS`; i.e., marker `17` of the red CAR is located in the middle of the left focal segment, while the remaining markers of the red CAR are on the right focal segment; note that in this example, the translocated marker `17` also belongs to class ii). **(ii)**, a TransLocation between CARs Within focal Segments (`TLWS`; i.e., marker `12` of the purple CAR is located in the middle of the red CAR within the right focal segment). **(iii)**, a TransLocation Within CARs within focal segments (`TLWC`; i.e., marker `4` has changed position within the blue CAR within the left focal segment). **(iv)**, InVersion within CARs within focal segments (`IV`; i.e., markers `7 8 9` are inverted to `-9 -8 -7` within the purple CAR).

`$TLWC` stores TransLocations Within CARs within focal segments; `$IV` stores InVersions within CARs within focal segments. See the Figure 3 for examples of these four classes of rearrangements.

Each rearrangement is represented by a separate column. Except for `TLBS`, which are identified across focal segments, columns for individual focal segments are joined across rows to save space (i.e., for `TLWS`, `TLWC`, and `IV`, which are identified within focal segments). To preserve the tabular format, these matrices are filled by zeros for focal segments with a non-maximal number of rearrangements, if necessary. If no rearrangements were detected for a certain class, the matrix has zero columns.

Markers that are part of a rearrangement have a tag value of `>0` within their respective column. Tagged markers within a column are not necessarily consecutive, for example, when a rearrangement is split into several parts through an insertion of a different CAR, or when a rearrangement has an upstream and a downstream component. Larger tag values (up to `1`) correspond to a higher confidence that the marker has been rearranged.[1] This arises from the fact that translocations can only be identified relative to each other. For example, if the marker order in the compared genome is `1 2 3 4 5`, and that in the focal genome is `1 3 4 2 5`, marker `2` might have been translocated to the right, or markers `3 4` might have been translocated to the left. In such a case, the markers that are more parsimonious to have been translocated relative to the alternative markers receive tag values of `1 - remWgt`, while alternative markers receive tag values of `remWgt` (`remWgt` can be set by the user). If such a distinction cannot be made, all involved markers are tagged with `0.5`. Markers that are part of inversions are always tagged with `1`. Full details are given in the description of the `rearrvisr` algorithm.

```
## Example for extracting inversions that have been identified on
## chromosome 2L of D. melanogaster when compared to the genome of
## the Drosophila ancestor 'MSSYE'

## extract marker IDs for chromosome 2L from alignment
MEL_2L_markers <- MEL_MSSYE$marker[MEL_MSSYE$scaff == "2L"]

## get position of markers on chromosome 2L in SYNT
tmp <- is.element(rownames(SYNT_MEL_MSSYE$IV), MEL_2L_markers)
```

[1] Note that these values are not probabilities, rather labels to tag markers that have arrangements in the focal genome that are in conflict to their arrangement in the compared genome.

```
## show inverted markers on chromosome 2L
SYNT_MEL_MSSYE$IV[rowSums(SYNT_MEL_MSSYE$IV) != 0 & tmp, ,
                  drop = FALSE]
#>       [,1] [,2] [,3] [,4] [,5]
#> 13315    1    0    0    0    0
#> 11698    0    1    0    0    0
#> 11765    0    1    0    0    0
#> 7605     0    0    1    0    0

## look at inverted marker 13315 in its genomic context
mpos<-which(MEL_MSSYE$marker == 13315)
MEL_MSSYE[(mpos - 4):(mpos + 4), c(1, 2, 5, 8, 7, 10)]
#>      marker scaff strand car orientation elem1
#> 1633   8691    2L      -   3           +   105
#> 1634   6481    2L      -   3           +   104
#> 1635    279    2L      -   3           +   103
#> 1636  14325    2L      +   3           -   102
#> 1637  13315    2L      -   3           -   110
#> 1638   3420    2L      -   3           +   101
#> 1639  14363    2L      -   3           +   100
#> 1640  13300    2L      +   3           -    99
#> 1641  12424    2L      -   3           +    98
## this shows that the genomic region where marker 13315 is located
## has been aligned in reverse direction: strand and orientation show
## inverted directionality, and the direction of elements in elem1 is
## descending; within this context, marker 13315 has identical
## directionality, which indicates an inversion (in this case, a
## single-marker inversion)
```

2.2 The `summarizeBlocks()` function

The `summarizeBlocks()` function summarizes the output of the `computeRearrs()` function for each synteny block in a more compact way. The function requires as input files the output of the `computeRearrs()` function (the list `SYNT`), and the data frames `focalgenome` and `compgenome`, which have been used to generate `SYNT`. In addition, the IDs of the focal genome segments that should be summarized need to be specified with the character vector `ordfocal`. See `?summarizeBlocks` for more information on the input and output of the function.

```
## Example for summarizing rearrangements in the D. melanogaster
## genome that occurred after divergence from the Drosophila
## ancestor 'MSSYE'

## summarize rearrangements for each synteny block
## (may run a few seconds)
BLOCKS_MEL_MSSYE <- summarizeBlocks(SYNT_MEL_MSSYE,
                                    MEL_markers, MSSYE_compgenome,
                                    c("2L", "2R", "3L", "3R", "X"))

## show names of the lists in the output
names(BLOCKS_MEL_MSSYE)
#> [1] "2L" "2R" "3L" "3R" "X"
```

```
## show names of matrices within lists
names(BLOCKS_MEL_MSSYE[[1]])
#> [1] "blocks" "TLBS"   "TLWS"   "TLWC"   "IV"     "IVsm"
```

Note: For the following examples it is assumed that the object `BLOCKS_MEL_MSSYE` has been created with the above command.

Next steps: *The data returned by the `summarizeBlocks()` function can be visualized with the `genomeRearrPlot()` function.*

Output: The function output is a list of lists for each focal genome segment in `ordfocal`. Each of these lists summarizes the alignment between `focalgenome` and `compgenome`, and the rearrangements in `SYNT`.

Each list per focal genome segment contains the data frame `$blocks`, which contains information on the alignment and structure of each *PQ-tree*, and five matrices `$TLBS`, `$TLWS`, `$TLWC`, `$IV`, and `$IVsm` that store detected rearrangements. Each synteny block is represented by a row. Note that separate blocks are also generated when the hierarchical structure of the underlying *PQ-tree* changes, therefore not all independent rows are caused by a rearrangement.

```
## print $blocks data frame for chromosome 3R of D. melanogaster
BLOCKS_MEL_MSSYE$`3R`$blocks
#>    start  end markerS markerE car type1 elemS1 elemE1 node1 nodeori1
#> 1      1  323    3157    1747   1     Q   2372   2050     0       -1
#> 2    324  775    1597   15589   1     Q    893   1344     0       -1
#> 3    776  776    1542    1542   1     Q   1346   1346     0       -1
#> 4    777  777    9155    9155   1     Q   1345   1345     0       -1
#> 5    778 1124    7278   17099   1     Q   1347   1693     0       -1
#> 6   1125 1125    9874    9874   1     Q   1695   1695     0       -1
#> 7   1126 1126   10236   10236   1     Q   1694   1694     0       -1
#> 8   1127 1250   10399   14959   1     Q   1696   1819     0       -1
#> 9   1251 1251     524     524   1     Q   1821   1821     0       -1
#> 10  1252 1252   12688   12688   1     Q   1820   1820     0       -1
#> 11  1253 1480    6868    6456   1     Q   1822   2049     0       -1
#> 12  1481 1885    1526    4203   1     Q    892    488     0       -1
#> 13  1886 1886    6780    6780   1     Q    486    486     0       -1
#> 14  1887 1887    3757    3757   1     Q    487    487     0       -1
#> 15  1888 2242   14838   13047   1     Q    485    131     0       -1
#> 16  2243 2244   18844   18865   1     Q    129    130     0       -1
#> 17  2245 2372   18849    4653   1     Q    128      1     0       -1
#>    subnode1 blockid1 blockori1 premask1
#> 1         0       13        -1        0
#> 2         0        6         1        0
#> 3         0      7.1        -1        0
#> 4         0      7.2        -1        0
#> 5         0        8         1        0
#> 6         0      9.1        -1        0
#> 7         0      9.2        -1        0
#> 8         0       10         1        0
#> 9         0     11.1        -1        0
#> 10        0     11.2        -1        0
#> 11        0       12         1        0
#> 12        0        5        -1        0
#> 13        0      4.1         1        0
#> 14        0      4.2         1        0
```

```
#> 15        0        3       -1        0
#> 16        0        2        1        0
#> 17        0        1       -1        0
## this shows that chromosome 3R is composed of 17 synteny blocks;
## details on the first four blocks are given below
```

In the `$blocks` data frame, the columns *start* and *end* give the start and end positions of the synteny block in `SYNT` (positions start at `1` separately for each focal genome segment), *markerS* and *markerE* give the marker IDs of the first and last marker per block, and *car* gives the ID of the aligned CAR.

```
## Details on the boundary between the first and second block on
## chromosome 3R of D. melanogaster between markers 1747 and 1597
## (node elements 2050 and 893; positions 323 and 324)

MEL_MSSYE[MEL_MSSYE$scaff == "3R", c(1, 2, 5, 8, 7, 10)][313:334, ]
#>      marker scaff strand car orientation elem1
#> 5887  11871    3R      +   1           -  2060
#> 5888   2352    3R      +   1           -  2059
#> 5889   9266    3R      -   1           +  2058
#> 5890  14789    3R      -   1           +  2057
#> 5891   7221    3R      +   1           -  2056
#> 5892   5094    3R      -   1           +  2055
#> 5893   3121    3R      +   1           -  2054
#> 5894   9210    3R      -   1           +  2053
#> 5895  10369    3R      +   1           -  2052
#> 5896  11781    3R      -   1           +  2051
#> 5897   1747    3R      +   1           -  2050
#> 5898   1597    3R      +   1           +   893
#> 5899   4925    3R      +   1           +   894
#> 5900   5520    3R      +   1           +   895
#> 5901   5715    3R      +   1           +   896
#> 5902   3263    3R      +   1           +   897
#> 5903  10490    3R      -   1           -   898
#> 5904  13175    3R      +   1           +   899
#> 5905  14690    3R      -   1           -   900
#> 5906   4995    3R      +   1           +   901
#> 5907   2771    3R      +   1           +   902
#> 5908   6894    3R      -   1           -   903
## this shows that the alignment changes from descending
## (i.e., inverted) to ascending (i.e., standard) direction
## at the block boundary
```

Additional columns in the `$blocks` data frame provide more information on the alignment and structure of each CAR for all hierarchy levels of the underlying *PQ-tree* (the first hierarchy level has columns with the suffix `1`, the second `2`, and so on). See the documentation of the `summarizeBlocks()` and the `computeRearrs()` function for a full description of the columns in the `$blocks` data frame.

There is only one hierarchy level for chromosome 3R of *D. melanogaster* in the example above. The column *nodeori1* stores the alignment direction of the first level node in the *PQ-tree* of the aligned CAR, with `1` indicating ascending (i.e., standard), and `-1` descending (i.e., inverted) alignment (given that the node is a *Q-node* and the alignment direction could be determined). Only one CAR (CAR `1`) has been aligned to chromosome 3R of *D. melanogaster* in descending direction. The column *blockori1* stores the orientation of each synteny block, again with `1` indicating ascending (i.e., standard), and `-1` descending (i.e., inverted) orientation (given that the bock orientation could be determined). In the above example, the first block has descending and the second block ascending direction, which is also reflected by the first and last node element

IDs of each block, which are given in the columns *elemS1* and *elemE1*.

Finally, the column *blockid1* stores the ID of each synteny block within its node. For *Q-nodes*, block IDs reflect the order of synteny blocks in `compgenome`. In the above example, the first block on chromosome 3R of *D. melanogaster* is the last block on CAR `1` of the *MSSYE* ancestor. Block IDs with `.1` or `.2` suffixes (in arbitrary order) indicate blocks that were subject to an additional subdivision step. In the above example, the third and fourth blocks were initially joined to a descending block with node elements `1346` and `1345`. Block subdivisions are used to assign a more parsimonious `TLWC` instead of an `IV` to a block.

```
## Details on the boundary between the third and fourth block on
## chromosome 3R of D. melanogaster between node elements 1346
## and 1345 (positions 776 and 777)

MEL_MSSYE[MEL_MSSYE$scaff == "3R", c(1, 2, 5, 8, 7, 10)][772:781, ]
#>      marker scaff strand car orientation elem1
#> 6346   5359    3R      +   1           +  1341
#> 6347   2932    3R      -   1           -  1342
#> 6348   5539    3R      +   1           +  1343
#> 6349  15589    3R      +   1           +  1344
#> 6350   1542    3R      +   1           +  1346
#> 6351   9155    3R      -   1           -  1345
#> 6352   7278    3R      -   1           -  1347
#> 6353   6603    3R      -   1           -  1348
#> 6354   7843    3R      +   1           +  1349
#> 6355  13365    3R      -   1           -  1350
## this shows that the larger-scale alignment around the two node
## elements is ascending (i.e., from 1341 to 1350), while elements
## 1346 and 1345 form a descending block. As their strand and
## orientation show identical directionality, it is more
## parsimonious to assign a translocation instead of an inversion,
## which would require two additional single-marker inversions to
## achieve the required inverted directionality. Accordingly, the
## initial block of two elements is split into two single-element
## blocks
```

The matrices `$TLBS`, `$TLWS`, `$TLWC`, `$IV`, and `$IVsm` summarize the four classes of rearrangements in `SYNT` for the synteny blocks characterized in `$blocks` (tag values within `$TLBS`, `$TLWS`, and `$TLWC` are the same as in `SYNT`, see above). Inversions that are part of a multi-marker inversion are stored in `$IV`, and blocks part of such inversions are tagged with `1` (with separate columns for different inversions). Single-marker inversions (i.e., markers with switched orientation) are stored in `$IVsm`. The positions of such single-marker inversions within their block are indicated by integers `>0` (with separate columns for different single-marker inversions).

```
## print $IV matrix for chromosome 3R of D. melanogaster
BLOCKS_MEL_MSSYE$`3R`$IV
#>       [,1] [,2]
#>  [1,]    0    0
#>  [2,]    1    0
#>  [3,]    1    0
#>  [4,]    1    0
#>  [5,]    1    0
#>  [6,]    1    0
#>  [7,]    1    0
#>  [8,]    1    0
#>  [9,]    1    0
#> [10,]    1    0
```

```
#> [11,]    1    0
#> [12,]    0    0
#> [13,]    0    0
#> [14,]    0    0
#> [15,]    0    0
#> [16,]    0    1
#> [17,]    0    0
## this shows that two multi-marker inversions have been detected;
## the first one spans blocks 2:11, and the second one involves
## block 16
```

```
## print $IVsm matrix for chromosome 3R of D. melanogaster
BLOCKS_MEL_MSSYE$`3R`$IVsm
#>       [,1] [,2]
#>  [1,]    0    0
#>  [2,]  419    0
#>  [3,]    0    0
#>  [4,]    0    0
#>  [5,]    0    0
#>  [6,]    0    0
#>  [7,]    0    0
#>  [8,]    0    0
#>  [9,]    0    0
#> [10,]    0    0
#> [11,]    0    0
#> [12,]    0    0
#> [13,]    0    0
#> [14,]    0    0
#> [15,]    0    0
#> [16,]    0    0
#> [17,]    0   18
## this shows that two single-marker inversions have been
## detected, the first is on position 419 within block 2, the
## second on position 18 within block 17

## look at the first single-marker inversion, which is located
## at position 743 (start position of block 2 + position of the
## inversion)
MEL_MSSYE[MEL_MSSYE$scaff == "3R", c(1, 2, 5, 8, 7, 10)][738:746, ]
#>      marker scaff strand car orientation elem1
#> 6312   5819    3R      +   1           +  1307
#> 6313   6560    3R      -   1           -  1308
#> 6314   5087    3R      +   1           +  1309
#> 6315   3137    3R      -   1           -  1310
#> 6316   3588    3R      +   1           -  1311
#> 6317   3714    3R      +   1           +  1312
#> 6318  13397    3R      +   1           +  1313
#> 6319   4303    3R      -   1           -  1314
#> 6320   5603    3R      -   1           -  1315
## this shows that marker 3588 is located in an ascending block,
## but has inverted directionality between strand and orientation
```

```
## print $TLWC matrix for chromosome 3R of D. melanogaster
BLOCKS_MEL_MSSYE$`3R`$TLWC
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
#>  [1,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
#>  [2,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
#>  [3,]  0.5  0.0  0.0  0.0  0.0  0.0  0.0  0.0
#>  [4,]  0.0  0.5  0.0  0.0  0.0  0.0  0.0  0.0
#>  [5,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
#>  [6,]  0.0  0.0  0.5  0.0  0.0  0.0  0.0  0.0
#>  [7,]  0.0  0.0  0.0  0.5  0.0  0.0  0.0  0.0
#>  [8,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
#>  [9,]  0.0  0.0  0.0  0.0  0.5  0.0  0.0  0.0
#> [10,]  0.0  0.0  0.0  0.0  0.0  0.5  0.0  0.0
#> [11,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
#> [12,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
#> [13,]  0.0  0.0  0.0  0.0  0.0  0.0  0.5  0.0
#> [14,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.5
#> [15,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
#> [16,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
#> [17,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
## this shows a total of eight translocations, each involving a
## single block. In all cases, blocks are tagged with 0.5,
## indicating that it could not be distinguished which part of
## a translocation would be more parsimonious to have changed
## position relative to the alternative part (e.g., for the
## translocation between the third and fourth block on
## chromosome 3R, described above, it cannot be distinguished
## whether node element 1346 has moved one block up, or element
## 1345 one block down)
```

3 Visualizing rearrangements

3.1 The `genomeImagePlot()` function

The `genomeImagePlot()` function visualizes rearrangements that were detected with the `computeRearrs()` function along the focal genome. Markers that are part of different classes of rearrangements are tagged with different colors at their map position, allowing the visual examination of the extent, number, and location of rearrangements. As an example, the Figure 4 visualizes rearrangements in the *D. melanogaster* genome that occurred after divergence from the *Drosophila* ancestor *MSSYE*.

The function requires as input files the output of the `computeRearrs()` function (i.e., the list `SYNT`), the data frame `focalgenome`, and the IDs of the focal genome segments that should be plotted (the character vector `ordfocal`).

The graphic can directly be output as PDF with the argument `makepdf = TRUE` (which can be much faster than plotting to screen). By default, the function automatically determines the optimal width and height of the plot (this can be turned off by setting both `makepdf = FALSE` and `newdev = FALSE`). Various function arguments allow to adjust the appearance of the graphic, including font sizes, axes labels, and plot margins. See `?genomeImagePlot` for more information and examples on the usage of the function.

Note: The automatic determination of the optimal width and height of the graphic is not possible when using the RStudio graphical device (`RStudioGD`) as default, as it does not accept width and height as arguments. In this case, the plot is send to an alternative device. (Only relevant when `makepdf = FALSE`.)

```
## Example for determining and changing the default graphics device

## determine current default graphics device
getOption("device")
## see list of available graphics devices
?Devices
## for example, set new default to X11
options(device = "X11")
```

```
## Example for visualizing rearrangements in the D. melanogaster
## genome that occurred after divergence from the Drosophila
## ancestor 'MSSYE'. The large maroon sector reveals a derived
## inversion on chromosome 3R, in line with previous work
## (Ranz et al., 2007)
genomeImagePlot(SYNT_MEL_MSSYE, MEL_markers,
                c("2L", "2R", "3L", "3R", "X"),
                main = "D. melanogaster - MSSYE")
```

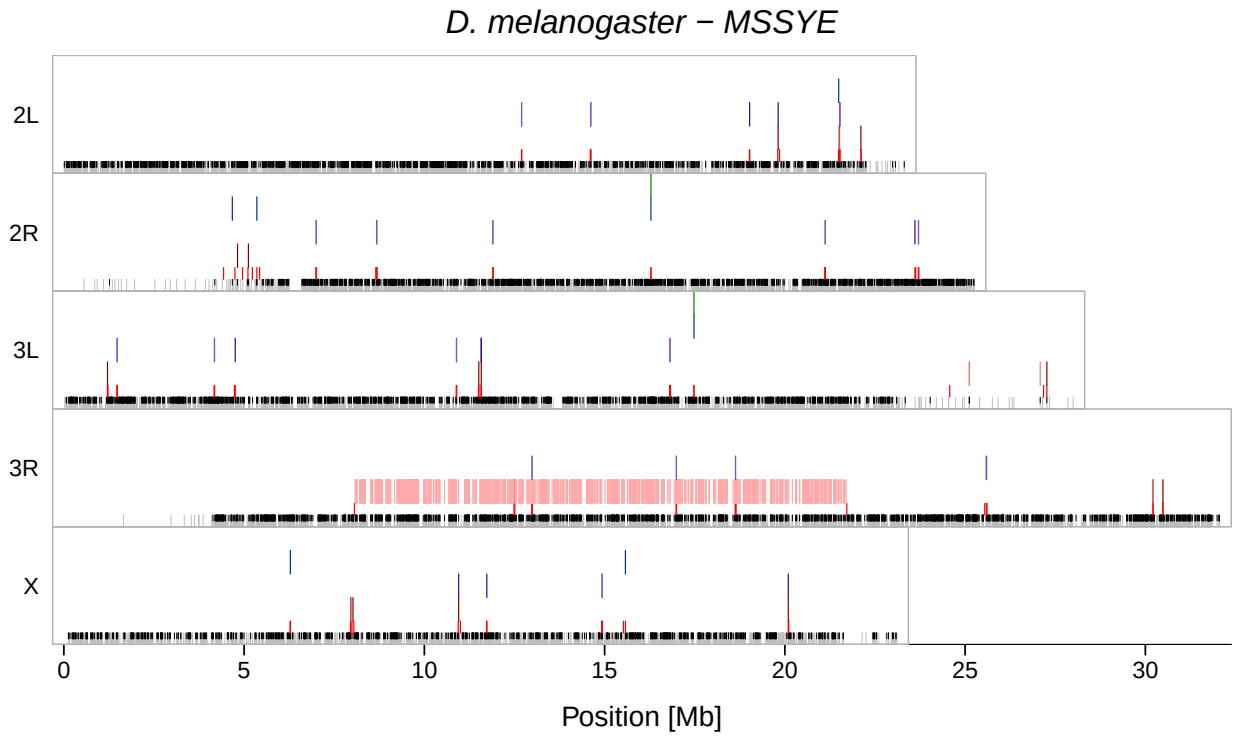

Figure 4: Graphical output of the genomeImagePlot() function

Output: On the y-axis, each focal genome segment that was included in `ordfocal` is plotted from top to bottom. Marker positions are on the x-axis. Vertical lines (ticks) indicate the positions of markers, rearrangement breakpoints, and different classes of rearrangements within six rows per focal segment.

The bottom row shows the positions of all markers contained in `focalgenome` as gray ticks, and the positions of markers present in `SYNT` as black ticks in the upper half of the bottom row. The second row from the bottom gives the positions of rearrangement breakpoints by red ticks. Ticks are drawn at the midpoints

between the positions of the two markers present in SYNT that are adjacent to the breakpoints. Only markers with black ticks can receive colored ticks in the remaining rows, which highlight markers that are part of different classes of rearrangements. Maroon indicates markers that are part of inversions within CARs within focal genome segments (IV); purple indicates markers that are part of translocations within CARs within focal genome segments (TLWC); blue indicates markers that are part of translocations between CARs within focal genome segments (TLWS); green indicates markers that are part of translocations between CARs between focal genome segments (TLBS). See the description of the rearrvisr algorithm or ?computeRearrs for more information on these classes of rearrangements.

Unless the argument remThld is set to a value smaller than that of remWgt in the computeRearrs() function, only markers that are less parsimonious to have changed position relative to alternative markers are highlighted. Lighter tints denote markers that are part of large rearrangements, while darker shades denote markers that are part of small rearrangements. To distinguish between individual large rearrangements versus multiple short adjacent rearrangements, it may be helpful to take the position of breakpoints into account.

3.2 The genomeRearrPlot() function

The genomeRearrPlot() function visualizes the output of the summarizeBlocks() function along the focal genome. The produced plot may be used to reconstruct how the compared genome segments were aligned to focal genome segments, to notice the structure of the aligned *PQ-trees*, and to comprehend why particular synteny blocks were tagged as rearranged by the computeRearrs() function. As an example, the Figure 5 visualizes rearrangements in the *D. melanogaster* genome that occurred after divergence from the *Drosophila* ancestor *MSSYE*.

The function requires as input files the output of the summarizeBlocks() function (i.e., the list BLOCKS), the data frame compgenome, and the IDs of the focal genome segments that should be plotted (the character vector ordfocal).

The graphic can directly be output as PDF with the argument makepdf = TRUE (which can be much faster than plotting to screen). By default, the function automatically determines the optimal width and height of the plot (this can be turned off by setting both makepdf = FALSE and newdev = FALSE). Various function arguments allow to adjust the appearance of the graphic, including font sizes, axes labels, and plot margins. See ?genomeRearrPlot for more information and examples on the usage of the function.

Note: The automatic determination of the optimal width and height of the graphic is not possible when using the RStudio graphical device (RStudioGD) as default, as it does not accept width and height as arguments. In this case, the plot is send to an alternative device. (Only relevant when makepdf = FALSE.) For an example on how to determine and change the default graphics device, see the section on the genomeImagePlot() function above.

```
## Example for visualizing rearrangements in the D. melanogaster
## genome that occurred after divergence from the Drosophila
## ancestor 'MSSYE' and that were summarized with the
## summarizeBlocks() function. Note that CAR 1 joins chromosomes
## 3L and 3R of D. melanogaster
genomeRearrPlot(BLOCKS_MEL_MSSYE, MSSYE_compgenome,
                c("2L", "2R", "3L", "3R", "X"),
                main = "D. melanogaster - MSSYE",
                blockwidth = 1.15, y0pad = 3)
```

Note: The function argument y0pad sets the amount of additional space between the bottom plot margin and the bottom plot area. Setting y0pad too small may result in some rearrangements for the bottom-most focal segment to be invisible because they will be outside the bottom plot area. This can be avoided by setting y0pad sufficiently large.

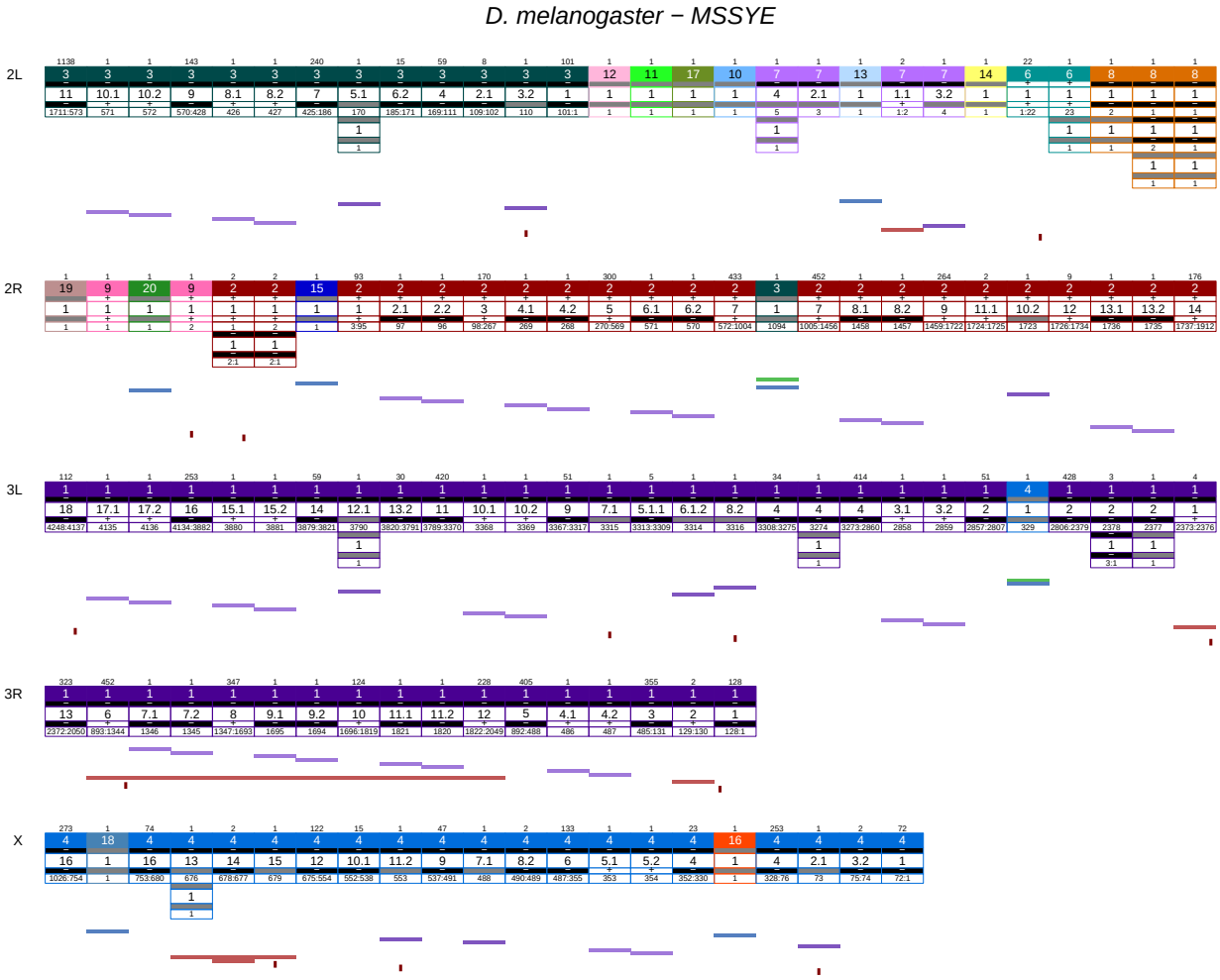

Figure 5: Graphical output of the genomeRearrPlot() function

Output: On the y-axis, each focal genome segment that was included in `ordfocal` is plotted from top to bottom. Each synteny block is represented by a column (note that separate blocks are also generated when the hierarchical structure of the underlying *PQ-tree* changes, therefore not all column boundaries are caused by a rearrangement).

The top part of each focal segment visualizes the data contained in the `$blocks` data frame in `BLOCKS` (i.e., information on the structure of each *PQ-tree* and its alignment to the focal genome). The top row gives the number of markers within each synteny block, calculated from the *start* and *end* columns in the `$blocks` data frame. The second row gives the IDs of the CARs, corresponding to entries in the *car* column in the `$blocks` data frame (different colors distinguish CARs).

For each hierarchy level, up to four additional rows (depending on the values in the argument `plotelem`) show (*i*) the alignment orientation of the *PQ-tree* node to the focal genome (white rectangles and `+` indicating ascending, and black rectangles and `-` indicating descending alignment), (*ii*) the block IDs, (*iii*) the block orientation within its node (white rectangles and `+` indicating ascending, and black rectangles and `-` indicating descending orientation), and (*iv*) the range of element IDs for each block within its node and for its level of hierarchy. These four rows correspond to entries in the (*i*) *nodeori**, (*ii*) *blockid**, (*iii*) *blockori**, and (*iv*) *elemS** – *elemE** columns in the `$blocks` data frame. See the `summarizeBlocks()` function above for a description of the `$blocks` data frame (using chromosome 3R of *D. melanogaster* as an example), and

`?genomeRearrPlot` for further details.

The bottom part of each focal segment visualizes different classes of rearrangements in the matrices `$TLBS`, `$TLWS`, `$TLWC`, `$IV`, and `$IVsm` in `BLOCKS`. See the description of the `rearrvisr` algorithm or the `?computeRearrs` documentation for more information on these classes of rearrangements. In contrast to the `genomeImagePlot()` function, the `genomeRearrPlot()` function allows to visualize rearrangements that are nested, and to distinguish between individual large rearrangements versus multiple short adjacent rearrangements.

Horizontal lines that are at identical height denote the same rearrangement (potentially disrupted by inserted CARs). Green indicates blocks that are part of translocations between CARs between focal genome segments (`TLBS`); blue indicates blocks that are part of translocations between CARs within focal genome segments (`TLWS`); purple indicates blocks that are part of translocations within CARs within focal genome segments (`TLWC`); maroon indicates blocks that are part of inversions within CARs within focal genome segments (`IV`). Single-marker inversions (`IVsm`) are indicated by a short vertical line at their position within their block, while multi-marker inversions are indicated be a horizontal line spanning the contributing block(s).

Lighter coloration denotes smaller weights for rearrangement tags in the respective matrices in `BLOCKS`. Unless the argument `remThld` is set to a value smaller than that of `remWgt` in the `computeRearrs()` function, only lines for blocks that are less parsimonious to have changed position relative to alternative blocks are plotted.

4 Description of the `rearrvisr` algorithm

4.1 Overview

Rearrangements are identified by aligning the compared genome representation in `compgenome` to the sorted genome map of the focal species in `focalgenome` (Figure 6A and B). Only the set of markers common to both genomes is considered. Markers (orthologous genes or synteny blocks) are called *doubled* if orientation information is available for the focal and the compared genome, for example when a genome reconstruction was obtained with the software ANGES (Jones *et al.*, 2012) applying the option `markers_doubled 1` (Ouangraoua *et al.*, 2011).

The *PQ-structure* (i.e., the data frame joining all *PQ-trees*, or CARs, of the compared genome) is sorted so that the order of markers corresponds to their order in the focal genome (i.e., it is aligned to the focal genome). If markers are doubled, their *directionality* in the focal genome relative to the compared genome is determined, which can be identical (both are `"+"` or both are `"-"`) or inverted (one is `"+"` and the other is `"-"`). Note that if the hierarchically lowest node of a given marker in the corresponding *PQ-tree* does not contain any other branches or markers (a *single-marker* node), the directionality is irrelevant, as nodes can be reversed. If marker directionality if available, it is taken into account for determining the alignment direction of *Q-nodes* and for detecting and classifying rearrangements within them.

Each focal genome segment (e.g., a chromosome or scaffold) is checked separately for rearrangements (except for `TLBS`, below, where all segments need to be considered jointly). Because the order of the aligned *PQ-structure* corresponds to the focal genome, any invalid sequence of *PQ-structure* elements (i.e., node element IDs) for a given subdivision of the *PQ-structure* (below; Figure 6C and D) indicates a rearrangement. The sequence of node elements will be checked hierarchically, starting at the CAR level and then descending through the nodes to the tips of the *PQ-structure* (i.e., the markers, or *leaves*), thereby subdividing the *PQ-structure*. The definition of an invalid sequence of node elements differs among CARs, *P-nodes*, and *Q-nodes*. Briefly, identical CAR IDs that are scattered across different focal genome segments indicate a translocation between CARs between focal genome segments (`TLBS`, below), unless certain conditions are met. Within focal genome segments, for CARs and *P-nodes*, the sequence of CAR IDs or node elements does not need to be sorted, but scattered elements of the same ID (e.g., element `3` within the sequence `1 4 4 3 2 2 3 3`) indicate a translocation between or within CARs (`TLWS` or `TLWC`, below; here, element `3` forms *flanks* around the *insert* consisting of element `2`). For *Q-nodes*, the sequence of elements needs to be sorted in either ascending or descending direction, any deviation from this (e.g., `4 4 3 1 2`) indicates a rearrangement (a

translocation or an inversion) within CARs (`TLWC` or `IV`, below). In many cases, it cannot be distinguished which component of a translocation has caused a distortion in the sequence of elements. This uncertainty is addressed by assigning different tag values to alternative translocation components. More details are below.

```
## PQ-structure in 'TOY24_compgenome' sorted so that the order of
## markers corresponds to their order in the focal genome in
## 'TOY24_focalgenome' (i.e., 'TOY24_compgenome' is aligned to
## 'TOY24_focalgenome'; see Figure A and B for a graphical
## representation of the alignment)

TOY24_compgenome[match(TOY24_focalgenome$marker,
                       TOY24_compgenome$marker), ]
#>    marker orientation car type1 elem1 type2 elem2 type3 elem3
#> 1       1           +   1     Q     1  <NA>    NA  <NA>    NA
#> 7       7           +   1     Q     7  <NA>    NA  <NA>    NA
#> 2       2           +   1     Q     2  <NA>    NA  <NA>    NA
#> 6       6           -   1     Q     6  <NA>    NA  <NA>    NA
#> 5       5           +   1     Q     5  <NA>    NA  <NA>    NA
#> 4       4           +   1     Q     4  <NA>    NA  <NA>    NA
#> 8       8           +   1     Q     8  <NA>    NA  <NA>    NA
#> 9       9           +   2     Q     1  <NA>    NA  <NA>    NA
#> 10     10           +   2     Q     2     Q     1  <NA>    NA
#> 3       3           +   1     Q     3  <NA>    NA  <NA>    NA
#> 13     13           +   2     Q     3     Q     3  <NA>    NA
#> 12     12           +   2     Q     3     Q     2  <NA>    NA
#> 11     11           +   2     Q     3     Q     1  <NA>    NA
#> 14     14           +   3     Q     1  <NA>    NA  <NA>    NA
#> 17     17           +   4     Q     1     P     3  <NA>    NA
#> 16     16           +   4     Q     1     P     2  <NA>    NA
#> 15     15           +   4     Q     1     P     1  <NA>    NA
#> 18     18           +   4     Q     1     P     4  <NA>    NA
#> 21     21           +   4     Q     2     Q     1     Q     3
#> 20     20           +   4     Q     2     Q     1     Q     2
#> 22     22           -   4     Q     2     Q     2     Q     1
#> 23     23           -   4     Q     2     Q     2     Q     2
#> 24     24           +   4     Q     2     Q     2     Q     3
#> 19     19           -   4     Q     2     Q     1     Q     1
```

4.2 Rearrangements between CARs

First, *best hits* between focal genome segments and CARs are identified. These are mutual best hits (i.e., the same focal segment where the maximum number of markers of a CAR are located is the same CAR where also the maximum number of markers of a focal segment are located), and secondary best hits (the focal segment where the maximum number of markers of a CAR are located, or the CAR where the maximum number of markers of a focal segment are located). Mutual and secondary *best hits* are not distinguished in the current implementation.

Second, translocations between CARs are detected. For each focal segment and each CAR, the number and positions of boundaries of a CAR (or CAR fragment) to other CARs on a given focal segment are identified. CARs with an invalid number and combination of boundaries within and across focal segments indicate

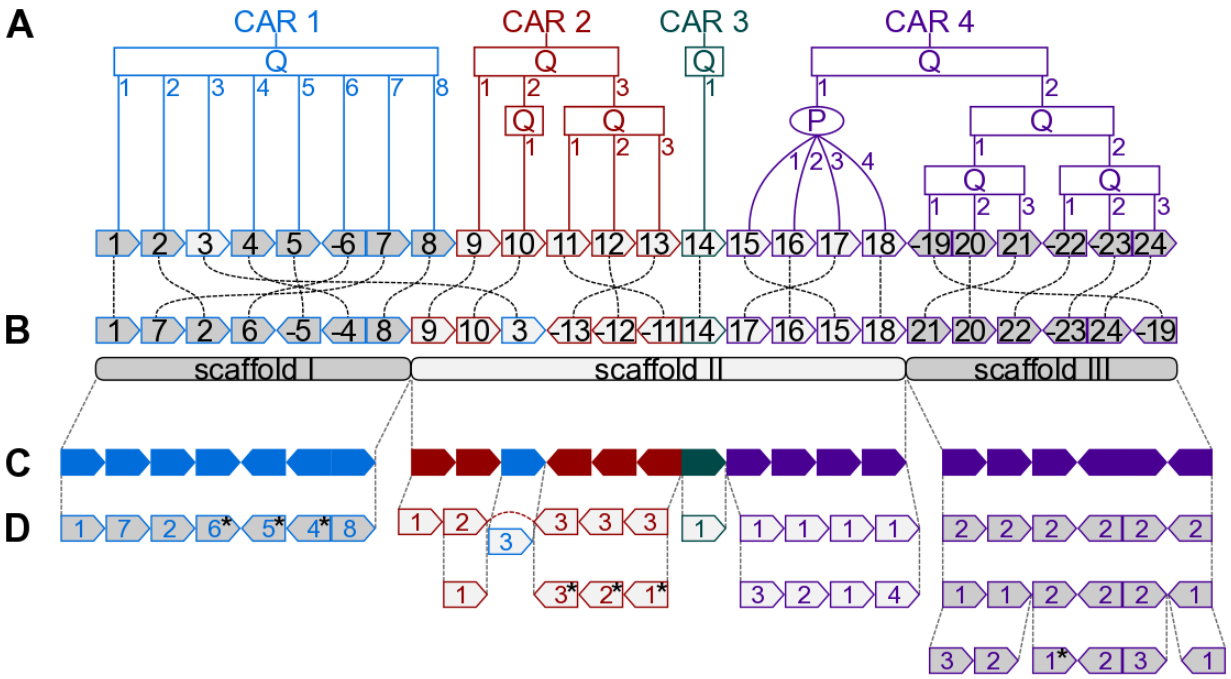

Figure 6: Alignment of the compared genome representation (i.e., *PQ-structure*) in `TOY24_compgenome` to the focal genome map in `TOY24_focalgenome`, and hierarchical subdivision of the aligned *PQ-structure* for rearrangement identification. **A**, graphical representation of the ordered *PQ-structure* in `TOY24_compgenome`. *PQ-trees* of four CARs are shown in colors. *P-nodes* are illustrated with ovals and *Q-nodes* with rectangles. Marker IDs are given by black integers within pentagons at the leaves of the *PQ-trees*. All branches are labeled with a colored integer ID (i.e., the node element ID) according to their position at their node of origin, starting at `1` for each node and hierarchy level. **B**, graphical representation of the ordered focal genome map in `TOY24_focalgenome`. Markers are shown as pentagons, filled with different grays according to their focal segment (scaffolds I – III) of origin. Black dashed lines show the alignment between the *PQ-structure* and the focal genome map. **C**, subdivision of the aligned *PQ-structure* by focal segments at CAR level (i.e., hierarchy level `1`). **D**, subdivision of the aligned *PQ-structure* by focal segments at node levels (i.e., hierarchy levels `2` – `4`). Hierarchy levels are shown in rows. The colored integers within pentagons are node element IDs of the aligned *PQ-trees*. Asterisks indicate markers that have inverted directionality in the focal genome relative to the compared genome. Subdivisions (based on identical elements at higher hierarchy levels) are indicated by small vertical gaps between markers and connected to the preceding hierarchy by gray dashed lines. For the red CAR (CAR 2), the red dashed line indicates that node elements `1 2 3 3 3` are joined across the inserted marker from the blue CAR (CAR 1).

translocations. Specifically, CARs with boundaries on different focal segments point to TranLocations between CAR fragments Between focal Segments (`TLBS`; e.g., CAR 1 in Figure 6C, which has two end boundaries on scaffold I and two internal boundaries on scaffold II), and CARs with more than two boundaries within a focal segment point to TranLocations between CAR fragments Within focal Segments (`TLWS`; e.g., CAR 2 in Figure 6C, which has one end boundary and three internal boundaries on scaffold II). The identification of translocations is refined by taking into account the size of focal segments and boundary positions. For example, for a scaffold-level assembly, CAR fragments located at the ends of two different focal segments do not necessarily indicate a translocation (e.g., CAR 4 on scaffolds II and III in Figure 6C). Likewise, a CAR with one fragment located on a large focal segment and that also fully spans a second (short) focal segment does not necessarily indicate a translocation, as the second segment might in fact be enclosed in the large segment in a perfect assembly.

Finally, the detected translocations between CARs (above) are further processed to filter out *best hits* and those flanks or inserts that are less parsimonious to have been translocated relative to their alternative components. Translocations between and within focal genome segments are distinguished (`TLBS` and `TLWS`, below).

4.2.1 Translocations between focal genome segments (`TLBS`)

If a translocation between CARs between focal segments was identified, each focal segment - CAR fragment pair is checked whether it constitutes a *best hit*. If so, markers part of the focal segment - CAR fragment *best hit* do not receive rearrangement tags, while all markers part of the non-*best hit* CAR fragments (i.e., those on other focal segments) are tagged with a value of `1` (e.g., in Figure 6C, the marker with ID `3` of CAR 1 on scaffold II would be tagged, but not the other markers of CAR 1 on scaffold I). If no *best hit* occurs, all markers part of a translocation between CARs between focal segments are tagged with a value of `0.5`.

4.2.2 Translocations within focal genome segments (`TLWS`)

If a CAR translocation within focal segments was identified, it is evaluated which component of the translocation (i.e., either flank or insert) is more parsimonious[2] to have been translocated and will be tagged (e.g., in Figure 6C, two and three markers of CAR II on scaffold II form flanks around the insert of a single marker of CAR 1). First, for each fragmented CAR within a focal segment, an insert is tagged if it does not contain more markers than either flank (if a CAR is split into more than two fragments, flank size is determined by summing markers over all CAR fragments up- or downstream of the focal insert, respectively). Otherwise, flanks are stored temporarily until all fragmented CARs have been analyzed. Second, for the temporarily stored flanks, it is evaluated whether the up- or downstream flank should be tagged. If one flank has already received an insert tag but not the other, the already tagged component is re-tagged (i.e., a flank of one CAR is also part of an insert of another CAR, e.g., for elements `1 2 2 3 2 2 2 3`, the left flank of element `3` will be tagged as it is also an insert between elements `2`). If either flank has not received insert tags before, the flank that has its node elements contained in the node elements of the other flank at a lower level of the *PQ-structure* hierarchy is tagged (only applies if the lower levels of the hierarchy are *Q-nodes*); otherwise, the flank that contains fewer markers than the other flank is tagged; or otherwise, both (equally sized) flanks are tagged. If both flanks have already received insert tags, both are re-tagged. Finally, if tags were assigned to one flank only, markers part of this flank are tagged with a value of `1 - remWgt`, while markers part of the other flank are tagged with a value of `remWgt` (where `remWgt` is a value `>= 0` and `< 0.5` that can be set by the user); if tags were assigned to both flanks, markers part of either flank are tagged with a value of `0.5`. Retained inserts are tagged with a value of `1`. As an additional step, it is evaluated whether one of two flanks, or one of two inserts, that are immediately adjacent to each other can be un-tagged, as only one of them will be sufficient to explain the distorted sequence of elements. If one of two immediatly adjacent flanks or inserts was un-tagged, its tag value is set to `remWgt`, and the other to `1 - remWgt`; otherwise, both tags are set to `0.5`. Markers part of the tagged translocation components (i.e., either flank or insert; for

[2] *More parsimonious* refers to the attempt to reduce the number of tagged markers, i.e., in the majority of cases, the fragment that involves fewer markers will be tagged.

flanks independent of their tag value) will be considered as separate *PQ-structure* subdivisions further down the *PQ-structure* hierarchy if `splitnodes = TRUE` (see below).

4.3 Rearrangements within CARs

After rearrangements between CARs have been determined, rearrangements within CARs (or CAR fragments) are identified for each focal genome segment by descending through the *PQ-structure* from the nodes to the leaves (i.e., through the hierarchy levels), thereby reflecting the hierarchical structure of rearrangements (Figure 6D). For a given focal hierarchy level, the *PQ-structure* is subdivided based on its unique elements at higher hierarchy levels. Consequently, each node is checked separately for rearrangements. If `splitnodes = TRUE`, an additional subdivision of nodes is performed based on translocations that were already identified at higher hierarchy levels (e.g., in Figure 6D, for scaffold III, marker `19` has been identified as translocated at the third hierarchy level, and thus forms a subdivision separate of markers `21` and `20` at the fourth hierarchy level).

```
## Example for a PQ-structure subdivision for scaffold 3 of
## 'TOY24_focalgenome'

## extract the sorted PQ-structure in 'TOY24_compgenome' for
## scaffold 3 of the focal genome, and determine subdivisions for
## the focal hierarchy level 4 based on node element IDs and
## translocations at the preceding hierarchy levels 1 (i.e., the
## CAR-level), 2, and 3

## --------- >>>> prepare data (for illustration only) ---------
## extract the sorted PQ-structure in 'TOY24_compgenome' for
## scaffold 3
tree <- TOY24_compgenome[match(TOY24_focalgenome$marker,
                               TOY24_compgenome$marker), ]
subtree <- tree[TOY24_focalgenome$scaff == "3", ]
## define current level of hierarchy
hierlevel <- 4
## define columns with node elements of preceding hierarchy levels
hiercol <- c(3, 5, 7)
## assuming last marker of scaffold 3 (marker 19) was identified as
## translocation at hierarchy level 3
subnodes <- cbind(c(0, 0, 0, 0, 0, 0), c(0, 0, 0, 0, 0, 0),
                  c(0, 0, 0, 0, 0, 1), c(0, 0, 0, 0, 0, 0))
## with splitnodes = TRUE (take translocations into account)
splitnodes <- TRUE
## identify required subdivisions of PQ-structure
if(splitnodes == TRUE){
  splitids <- matrix(NA, ncol = 0, nrow = nrow(subtree))
  for(d in 1:(hierlevel - 1)){
    splitids <- cbind(splitids, paste(subtree[ , hiercol[d]],
                                      subnodes[ , d], sep = "."))
  }
  allprev <- apply(splitids, 1, function(x) paste0(x, collapse = "-"))
}else{
  allprev <- apply(subtree[ , hiercol, drop = FALSE], 1,
                   function(x) paste0(x, collapse = "-"))
}
## ------------- <<<< end of data preparation -------------
```

```
## print PQ-structure for scaffold 3 of 'TOY24_focalgenome'
print(subtree)
#>    marker orientation car type1 elem1 type2 elem2 type3 elem3
#> 21     21           +   4     Q     2     Q     1     Q     3
#> 20     20           +   4     Q     2     Q     1     Q     2
#> 22     22           -   4     Q     2     Q     2     Q     1
#> 23     23           -   4     Q     2     Q     2     Q     2
#> 24     24           +   4     Q     2     Q     2     Q     3
#> 19     19           -   4     Q     2     Q     1     Q     1

## display subdivision indices
print(data.frame(marker = subtree$marker, subdivision.id = allprev,
                 stringsAsFactors = FALSE))
#>   marker subdivision.id
#> 1     21    4.0-2.0-1.0
#> 2     20    4.0-2.0-1.0
#> 3     22    4.0-2.0-2.0
#> 4     23    4.0-2.0-2.0
#> 5     24    4.0-2.0-2.0
#> 6     19    4.0-2.0-1.1
## subdivision indices are composed of elements in the 'car',
## 'elem1', and 'elem2' columns (separated by "-"), and IDs from
## nodes that were split due to identified translocations at
## preceding hierarchy levels (separated by "."); here, only one
## marker of element '1' in column 'elem2' was tagged as
## translocation and received the split ID '1' (all others are '0')

## print subdivisions of PQ-structure
uniprev <- unique(allprev[!is.na(subtree[ , 2 + (hierlevel - 1) * 2])])
for(p in 1:length(uniprev)){
  print(subtree[allprev == uniprev[p], , drop = FALSE])
  cat("\n")
}
#>    marker orientation car type1 elem1 type2 elem2 type3 elem3
#> 21     21           +   4     Q     2     Q     1     Q     3
#> 20     20           +   4     Q     2     Q     1     Q     2
#>
#>    marker orientation car type1 elem1 type2 elem2 type3 elem3
#> 22     22           -   4     Q     2     Q     2     Q     1
#> 23     23           -   4     Q     2     Q     2     Q     2
#> 24     24           +   4     Q     2     Q     2     Q     3
#>
#>    marker orientation car type1 elem1 type2 elem2 type3 elem3
#> 19     19           -   4     Q     2     Q     1     Q     1
```

For a given focal hierarchy level, the sequence of node elements is then checked separately for each *PQ-structure* subdivision. An invalid sequence of node elements points to TranLocations Within CARs (TLWC) or InVersions within CARs (IV). The definition of an invalid sequence differs between *P-node* and *Q-node* elements, and inversions can only be identified for *Q-nodes*. Note that a clear distinction between translocations and inversions can not always be made. Rearrangements are thus classified so that the number of required rearrangement events and tagged markers is reduced.

4.3.1 Translocations (`TLWC`)

4.3.1.1 *P-nodes*

As *P-nodes* contain contiguous markers with no fixed ordering, they are structurally comparable to different CARs (or CAR fragments) within focal genome segments. Likewise, scattered *P-node* elements of the same ID that form flanks around an insert indicate a translocation. The identification of translocations in *P-nodes* and the tagging scheme thus follows the same steps as described for translocations between CARs within focal genome segments (`TLWS`, above). The only difference is that for simplicity, if either flank has not received insert tags before, it is not checked whether the elements of one flank are contained in the other flank at a lower level of the *PQ-structure* hierarchy (assuming here that *P-nodes* are rare, at low hierarchical levels, and commonly contain *single-marker Q-nodes*). Thus, if initially both flanks were retained, only their size is taken into account for evaluating whether the up- or downstream flank should be tagged.

4.3.1.2 *Q-nodes*

Q-nodes contain contiguous markers with a fixed order (including their reversal), thus any deviation in the order of *Q-node* elements indicates a translocation (described here) or an inversion (described below).

Step 1: cluster node elements into blocks. As a first step, all node elements of the current subdivision are clustered into *blocks*. Node elements are ranked so that ties receive identical ranks with no gaps to higher ranks. In the following, 'elements' are referred to these element ranks, not the original node elements, which may contain gaps. Blocks are formed by elements that are consecutive, without gaps, and no change in direction. For example, elements in the sequence `1 2 2 3 6 6 5 5 5 4 7` can be clustered into three blocks, consisting of elements `1 2 2 3`, `6 6 5 5 5 4`, and `7`. Blocks are ranked according to the mean of their first and last element (again, with identical ranks for ties and without gaps), and their orientation is determined. In the example above, the three blocks have ranks `1`, `2`, and `3` and their orientations are `1`, `-1`, and `9` (where `1` denotes ascending, `-1` descending, and `9` no orientation). The sequence of block ranks `1 2 3` is subject to additional iterations of clustering until no further simplification can be achieved (in the example above, this would be the next iteration, resulting in a single block of ascending orientation). Each boundary between blocks points to an invalid sequence of node elements and thus a rearrangement.

Step 2: assign node alignment direction. The results of the last simplification step (and possibly the steps before) are used to assign the alignment direction of the underlying *Q-node* to the focal genome segment. If a final simplification to a single block exists, the orientation of the last block defines the node alignment direction, unless the last block is formed by exactly two blocks (or elements) in the penultimate iteration. In that case, the node alignment direction is switched if both penultimate blocks (or elements) have an opposed orientation (or directionality if block members are leaves) relative to the last block. This measure is parsimonious as it reduces the number of identified rearrangements (e.g., elements `2 1` with orientation `1 1` will require one translocation with ascending alignment, but two inversions with descending alignment). If a final simplification to a single block was not possible (e.g., with elements `2 4 1 3 5`), the node alignment direction is defined by the largest block in the last iteration, unless the first and last block ranks conform to an ascending or descending alignment (e.g., `1 4 2 5 3 6` defines an ascending alignment).

Step 3: identify incorrect block adjacencies. Given the assigned node alignment direction, blocks are checked whether they form correct or incorrect adjacencies according to their ranks. This is done either for the last simplification step if no single final block exists, or for the penultimate step. Incorrect adjacencies (including the node ends) indicate a translocation (e.g., block ranks `2 4 1 3 5` for an ascending alignment direction have wrong adjacencies between all blocks and at the start of the first block, but not the end). Markers that are part of blocks that are positioned between two incorrect adjacencies are tagged with a value of `1`.

Alternative steps 1 – 3 with scattered node elements. If scattered *Q-node* elements are present (e.g., in Figure 6D, node elements `1 1 2 2 2 1` for scaffold III at hierarchy level `3`), they are taken into account for both the clustering of elements into blocks and the identification of the node alignment direction. Briefly, scattered elements are only allowed at the ends of a block, and have to be separated by at least one

non-scattered element on the same block. This measure is necessary to avoid potential conflicts during the determination of block ranks. As above, additional iterations of clustering are performed until no further simplification can be achieved (scattered elements may be bound into higher-level blocks during this process). Following the clustering into blocks, for the last or penultimate step (as above), all scattered elements that were not bound into larger blocks (i.e., they form their own block) are masked. Non-masked blocks are checked whether they form correct or incorrect adjacencies (as above) according to their adjusted ranks. Masked blocks are then added and adjacencies to their up- and downstream block (or the node ends) are checked. This measure preferably identifies scattered node elements (rather than non-scattered elements) as translocated. Adjacencies are checked for both ascending and descending alignment direction, and the direction that contains fewer incorrect adjacencies is retained; otherwise additional decision steps are applied. For the retained alignment direction, markers that are part of blocks that are positioned between two incorrect adjacencies are tagged (as above).

Step 4: identify translocations from incorrect block adjacencies. Whether scattered node elements were present or not, if all blocks are positioned between incorrect adjacencies, the block(s) within those incorrect adjacencies that comprise the largest block (i.e., which contains the maximum number of markers) will be un-tagged. If more than one block contains the maximum number of markers, additional decision steps are applied to identify one block to be un-tagged. Markers that are within those adjacencies that contain the un-tagged block (if any) receive a tagging value of `remWgt`. If one block was un-tagged and only one more set of incorrect adjacencies exists, markers contained in this second set receive a value of `1 - remWgt`. If more sets exist, makers within them retain their tagging value of `1`. If a single block could not be identified to be un-tagged, all blocks remain tagged, except if only two equally-sized sets of incorrect adjacencies exist, in which case both sets receive a value of `0.5` (making tag values comparable to the ones described in step 5, below). Markers within each of the (remaining) incorrect adjacencies will be considered as separate *PQ-structure* subdivisions further down the *PQ-structure* hierarchy if `splitnodes = TRUE`.

Step 5: identify translocations from deviating block orientations. Following the identification of translocations at the last or penultimate block simplification step, for each remaining simplification level, the orientation of blocks is compared to that at the next higher level, starting from the node alignment direction and finishing at the first-level blocks that initially cluster node elements. A block that is oriented in opposite direction to its next higher-level block indicates a rearrangement (a translocation or an inversion). For example, if the penultimate clustering step resulted in block ranks `1 2 3` with orientation `1 -1 9` and ascending node alignment direction `1`, block `2` with descending orientation `-1` deviates from the ascending higher-level orientation `1` (i.e., node alignment direction) and thus indicates a rearrangement. If this descending block `2` contains lower-level blocks, an ascending lower-level block with orientation `1` within this block indicates a rearrangement as it deviates from the `-1` orientation of block `2`. This process is iterated until the first (initial) block level is reached. If no orientation (`9`) is available for a higher-level block, the orientation from the next-higher level is used.

A translocation is assigned if an oppositely orientated block contains exactly two lower-level blocks (or elements) and one of the following conditions is met: (1) one or both lower-level blocks are scattered elements that were not bound into larger blocks; or (2) one or both lower-level blocks contain at least one leave at any block level (or one or both block elements are leaves), unless either (a) the current block level is two, all elements within the two blocks are leaves and have marker directionality available, and more than half of those elements have the same relative marker directionality as the currently considered block (i.e., more than half have inverted relative directionality for a `-1` block, or identical relative directionality for a `1` block), or (b) the current block level is one, both elements are leaves, and non of the elements that have marker directionality available have the opposite relative marker directionality as the currently considered block (i.e., non has identical relative directionality for a `-1` block, or inverted relative directionality for a `1` block). In all other cases, an inversion is assigned (below).

If a translocation was assigned, the lower-level block that contains fewer markers will be tagged; otherwise, the single lower-level block (or element) that has opposite orientation (or directionality) to the current block, if any, will be tagged (if no orientation is available for a lower-level block, the orientation from the next-lower level is used); otherwise both lower-level blocks (or elements) will be tagged. If one lower-level block (or element) received a tag, markers part of this lower-level block (or element) are tagged with a value of `1 -`

`remWgt`, while markers part of the other lower-level block (or element) are tagged with a value of `remWgt`; if tags were assigned to both lower-level blocks (or elements), all involved markers are tagged with a value of `0.5`. Markers part of the translocated lower-level blocks (or elements) that received tag values of `>=0.5` will be considered as separate *PQ-structure* subdivisions further down the *PQ-structure* hierarchy if `splitnodes = TRUE`.

4.3.2 Inversions (`IV`)

Inversions can only be identified for *Q-nodes*. For a given hierarchy level and *PQ-structure* subdivision, all elements are clustered into blocks, and for each block simplification level, the orientation of blocks is compared to that at the next higher level (as for `TLWC`, steps 1, 2, and 5 above). An oppositely orientated block that was not identified as translocation (`TLWC`, step 5 above) is marked as an inversion. This is the case when an oppositely orientated block contains more than two lower-level blocks (or elements), or if an oppositely orientated block contains exactly two lower-level blocks (or elements), non of them is a scattered element that was not bound into larger blocks, and one of the following conditions is met: (1) non of the two lower-level blocks contains leaves at any block level (i.e., both blocks contain only nodes); or (2) both lower-level blocks contain only leaves at any block level (or one or both block elements are leaves), and exceptions (a) or (b) (see `TLWC`, above) are met.

In addition, inversions that involve only a single marker are identified if information on directionality is available for a leave marker at a given hierarchy level and *PQ-structure* subdivision. *Single-marker inversions* are leave markers whose directionality deviates from the expected orientation of leaves, given the (potentially nested) node alignment direction and block orientation across all block simplification levels. All markers part of an inversion (multi-marker inversions identified through an oppositely orientated block or single-marker inversions) are tagged with a value of `1`.

5 *Drosophila* data set

5.1 Data preparation

Peptide sequences and annotation information (`pep.all.fa` and `gff3` files) for genes from 12 *Drosophila* species were downloaded on Dec 23 2017 from Ensemble Release 91 (http://dec2017.archive.ensembl.org; *D. melanogaster*) or Ensemble Metazoa Release 37 (http://oct2017-metazoa.ensembl.org; *D. ananassae*, *D. erecta*, *D. grimshawi*, *D. mojavensis*, *D. persimilis*, *D. pseudoobscura*, *D. sechellia*, *D. simulans*, *D. virilis*, *D. willistoni*, and *D. yakuba*). Sequences that were shorter than 50 amino acids or contained premature stop codons were excluded. 20,803 orthologous groups were identified with OMA standalone v2.2.0 (Altenhoff *et al.*, 2015), using as guidance tree the phylogeny published in Drosophila 12 Genomes Consortium (2007), and default settings otherwise. A single representative splicing variant per gene was identified by OMA, and alternative splicing variants were excluded from subsequent analyses. The retained number of genes per species were 15,052, 14,998, 14,969, 13,818, 14,581, 16,856, 15,845, 16,409, 15,355, 14,477, 15,490, and 16,039 for *D. ananassae*, *D. erecta*, *D. grimshawi*, *D. melanogaster*, *D. mojavensis*, *D. persimilis*, *D. pseudoobscura*, *D. sechellia*, *D. simulans*, *D. virilis*, *D. willistoni*, and *D. yakuba*, respectively.

Sequences of 4,792 OMA orthologous groups that only included one-to-one orthologous genes present in all 12 species were extracted and individually aligned with MAFFT v7.407 (Katoh *et al.*, 2002; Katoh and Standley, 2013), using the iterative refinement method incorporating local pairwise alignment information (`--localpair --maxiterate 1000` settings). Alignments with not more than 20% missing data (4,308 orthologous groups) were concatenated, and a phylogenetic tree was computed with RAxML v8.2.12 (Stamatakis, 2014). The best protein substitution model (JTT with empirical base frequencies) was determined by the program (`-f a -# autoMRE -m PROTGAMMAAUTO --auto-prot=ml` settings). The resulting tree was rooted at the branch that best balanced the subtree lengths.

5.2 Ancestral genome reconstruction

Genome maps were prepared for all retained genes (i.e., excluding low quality sequences and non-representative splicing variants) based on gene position information extracted from the `gff3` files. Gene start and end positions were calculated as the average of CDS midpoints -/+ 1 base pair to avoid the occurrence of overlapping gene positions, which are not supported by the genome reconstruction software ANGES v1.01 (Jones *et al.*, 2012). A few remaining overlaps between gene positions were resolved manually. The ANGES `marker` input file was generated using the genome maps of all 12 *Drosophila* species and 20,803 OMA orthologous groups (from the OMA `OrthologousGroups.txt` file) with the R script `oma2anges.R` (available with the `rearrvisr` package). The ANGES `tree` input file was based on the best rooted RAxML tree computed above (including branch lengths). The ancestral genome of the *melanogaster* subgroup *'MSSYE'* (Drosophila 12 Genomes Consortium, 2007; i.e., separating *D. melanogaster*, *D. simulans*, *D. sechellia*, *D. yakuba*, and *D. erecta* from the remainder of the *Drosophila* species) was reconstructed using the ANGES master pipeline (`anges_CAR.py`) and options `markers_doubled 1` (infer ancestral marker orientation), `markers_unique 2` (no duplicated markers), `markers_universal 1` (no missing markers in ingroup), `c1p_telomeres 0` (no telomeres), and `c1p_heuristic 1` (using a greedy heuristic).

A total of 27,242 *Ancestral Contiguous Sets* (ACS; Jones *et al.*, 2012) were identified by ANGES, of which 26,594 ACS were organized into 20 CARs (648, or 2.4%, of ACS were discarded by the program). These CARs comprised a total of 8,973 ancestral markers, and 99.2% of them were grouped into four major CARs (i.e., the 20 CARs included 4,250, 1,914, 1,711, 1,026, 28, 23, 5, 3, 2, and 11 x 1 ancestral markers).

6 References

Altenhoff,A.M. *et al.* (2015) The OMA orthology database in 2015: function predictions, better plant support, synteny view and other improvements. *Nucleic Acids Research*, **43**, D240–D249.

Booth,K.S. and Lueker,G.S. (1976) Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, **13**, 335–379.

Chauve,C. and Tannier,E. (2008) A methodological framework for the reconstruction of contiguous regions of ancestral genomes and its application to mammalian genomes. *PLoS Computational Biology*, **4**, e1000234.

Drosophila 12 Genomes Consortium (2007) Evolution of genes and genomes on the *Drosophila* phylogeny. *Nature*, **450**, 203–218.

Jones,B.R. *et al.* (2012) ANGES: reconstructing ANcestral GEnomeS maps. *Bioinformatics*, **28**, 2388–2390.

Katoh,K. and Standley,D.M. (2013) MAFFT multiple sequence alignment software version 7: improvements in performance and usability. *Molecular Biology and Evolution*, **30**, 772–780.

Katoh,K. *et al.* (2002) MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucleic Acids Research*, **30**, 3059–3066.

Ma,J. *et al.* (2006) Reconstructing contiguous regions of an ancestral genome. *Genome Research*, **16**, 1557–1565.

Ouangraoua,A. *et al.* (2011) Reconstructing the architecture of the ancestral amniote genome. *Bioinformatics*, **27**, 2664–2671.

Ranz,J.M. *et al.* (2007) Principles of genome evolution in the *Drosophila melanogaster* species group. *PLOS Biology*, **5**, e152.

Stamatakis,A. (2014) RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, **30**, 1312–1313.